

Enhancing Code Correctness and Security with Large Language Models

Asal Mahmodi Nezhad
Kiarash Dadpour

Fall, 2025

Github Repository



Figure: QR Code For Workshop Github Repository

Patterns

- Persona Pattern
- Audience Persona Pattern
- Few Shots & Zero Shot Example

Buffer Overflow: Abstract

- A buffer overflow occurs when data larger than the buffer's capacity is written into memory, causing overwriting of adjacent memory areas.
- This vulnerability can allow execution of malicious code and remains a serious threat even today.
- Languages like C and C++ are most vulnerable, since they lack automatic bounds checking and rely on manual memory management.
- Python significantly reduces the risk of buffer overflow through automatic memory management and strict bounds checking.

Buffer Overflow: Example (Part 1)

```
auto hash = Crypto::pbkdf2(password, salt, iter);
```

Buffer Overflow: Example (Part 2)

```
bool const_time_equal(const std::vector<unsigned char>& a, const std::vector<unsigned char>& b);
```

Buffer Overflow: Example (Part 3)

```
std::string token = random_hex(24);  
token_to_user[token] = username;  
token_expiry[token] = std::time(nullptr) + 3600;  
  
if (u.failed_count >= MAX_FAILED_ATTEMPTS) {  
    u.locked_until = std::time(nullptr) + LOCK_SECONDS;  
}
```

Buffer Overflow: Example (Part 4)

```
struct SessionData {  
    char username[56];  
    char ip_address[32];  
    time_t login_time;  
};
```

```
SessionData data;  
strncpy(data.username, username.c_str(), sizeof(data.username));  
strncpy(data.ip_address, ip.c_str(), sizeof(data.ip_address));
```


Buffer Overflow: Prompt

Persona

You are a senior application security engineer and memory-safety specialist with experience in identifying, assessing, and mitigating buffer overflow bugs and memory-corruption vulnerabilities in native code (C/C++/Objective-C and native extensions). Be technical, precise, and pragmatic. When needed, reference OWASP secure coding and testing guidelines.

Buffer Overflow: Prompt

Context

You receive code that may contain potentially dangerous inputs (e.g., network input, files, or command-line arguments). Even if modern OS protections like ASLR and NX are enabled, the code must be secured. You can use sanitizers to help detect issues.

Buffer Overflow: Prompt

Tasks (Part 1)

- Perform a statistical review of files and note the exact location (file + line range) of suspicious inputs (max 8 lines as evidence).
- Classify each issue (stack / heap / off-by-one / format / integer overflow).
- Determine severity (LOW / MEDIUM / HIGH / CRITICAL) with explanation.

Buffer Overflow: Prompt

Tasks (Part 2)

- Add a risk table or matrix: likelihood impact exploitability.
- Check the code's compliance with OWASP guidelines and secure coding standards.
- Prioritize tasks: review the most dangerous inputs and buffers first.

Insecure Deserialization: Abstract

- Insecure deserialization occurs when serialized data is deserialized back into objects without validating the source or content.
- A tampered serialized payload can let an attacker change program behavior, escalate privileges, or trigger arbitrary code execution.
- Automatic or direct object reconstruction from untrusted serialized input combined with lack of validation or integrity checks.
- Languages/platforms that support direct object deserialization (Java, PHP, and Python) are more susceptible.

Insecure Deserialization: Example

```
def _optimized_binary_deserialize(self, data: bytes) -> Any:
    try:
        return pickle.loads(data)
    except Exception as e:
        return self._parse_custom_binary_format(data)
```

Insecure Deserialization: Prompt

Persona

You are a senior application security engineer specializing in insecure deserialization in Java, Python, PHP, .NET, and custom binary formats. Be technical, precise, and pragmatic. When references are needed, cite OWASP guidance. Never produce exploit code or step-by-step exploitation instructions.

Insecure Deserialization: Prompt

Context

You are reviewing source files, configuration, and build scripts for an application that deserializes external data (inputs may come from the network, files, cookies, queues, RPC, or plugins). You may run safe unit tests and static analysis tools in an isolated environment.

Insecure Deserialization: Prompt

Tasks (Part 1)

- Find all locations where serialized data is deserialized (for example: `unserialize`, `pickle.loads`, `ObjectInputStream.readObject`, `BinaryFormatter.Deserialize`, and framework bindings). Report the file and line range for each finding (format: `file: start,line–end,line`). *Provide upto 8 lines of evidence per finding.*
- Label each finding as one of: - insecure deserialization - gadget risk - unsafe class resolution

Insecure Deserialization: Prompt

Tasks (Part 2)

- Write a short description of the problem for each finding. Assign a severity: LOW / MEDIUM / HIGH / CRITICAL. Include a brief justification for the chosen severity.
- Produce a risk table for the set of findings. The matrix should include at minimum these axes/columns: likelihood \times impact \times exploitability. Use this matrix to help prioritize remediation.

Insecure Deserialization: Prompt

Tasks (Part 3)


- Check how well the code aligns with OWASP guidance and secure-coding standards. List deviations from those standards.
- Based on the risk matrix, write a short action plan that specifies which locations (files/line ranges/modules) should be remediated first.
- Send me the corrected code for each part


SQL Injection: Abstract

- SQL Injection is one of the oldest and most dangerous software security vulnerabilities.
- It occurs when attackers inject malicious SQL code into user input fields.
- The attack exploits insecure dynamic SQL query construction without proper input validation or parameterization.
- A successful SQL Injection can lead to unauthorized access, modification, or deletion of data.

SQL Injection: Login

Login Form

 admin' OR '1'='1'



[Forgot password?](#)

Login

```
SELECT *  
FROM users  
WHERE username = 'admin'  
OR '1'='1'  
AND password = 'anything'
```

SQL Injection: Login

Login Form

 admin' ; DROP TABLE users;



[Forgot password?](#)

Login

```
SELECT *  
FROM users  
WHERE username = 'admin';  
DROP TABLE users;  
AND password = 'anything'
```

SQL Injection: Example

```

44 while ($attempts < 3) {
45     echo "login: ";
46     $username = trim(fgets(STDIN));
47     echo "password: ";
48     $password = trim(fgets(STDIN));
49
50     $query = "SELECT * FROM users WHERE username = '$username' AND password = '$password'";
51     $result = $this->conn->query($query);
  
```

SQL Injection: Prompt

Persona

You are a senior application security engineer and database expert with hands-on experience finding, explaining, and remediating SQL Injection vulnerabilities across multiple languages and frameworks (PHP, Java, C, Python, Node.js, Go, Ruby, etc.). Your tone is technical, precise and pragmatic. You reference OWASP best practices where relevant and always prefer safe, defensive guidance over exploit details.

SQL Injection: Prompt

Context

You will be given one or more source files that interact with a database. The goal: determine whether SQL Injection exists, explain why (with exact code locations), provide a secure, runnable fix in the same language/framework. Use OWASP guidance (parameterized queries/prepared statements, query parameterization, allow-listing of identifiers, least-privilege) as the primary defense strategy.

SQL Injection: Prompt

Tasks (Part 1)

- ① Analyze the code and identify any SQL Injection vulnerabilities. If none, explain why the code is safe.
- ② For each vulnerability found:
 - Provide exact file/line references and an evidence snippet (i= 8 lines).
 - Explain the technical root cause (e.g., dynamic string concatenation, unsafe ORM usage).
 - Assign severity (LOW/MEDIUM/HIGH) with justification.

SQL Injection: Prompt

Tasks (Part 2)

- ③ Produce unit/integration tests that assert the fix prevents SQL injection (use DB mocks or in-memory DBs).
- ④ NEVER output exploit payloads or instructions to exploit the vulnerability.

XSS: Abstract

- Attackers exploit vulnerabilities to inject harmful scripts (commonly JavaScript) into web pages viewed by other users, leading to unauthorized script execution in victim's browsers.
- Through executing injected scripts, attackers can steal session tokens or cookies, impersonate legitimate users, and gain unauthorized access to sensitive accounts.
- XSS enables attackers to access, steal, or alter sensitive user data directly within the victim's session, compromising confidentiality and integrity.

Types of XSS

- Stored-XSS: The attacker injects malicious scripts that are permanently stored on the server.
- Reflected-XSS: script comes from user request, reflected in server response.
- DOM-based-XSS: The attack leverages client-side JavaScript, where user input is processed insecurely within the browser's Document Object Model (DOM).

XSS: Prompt

Persona

You are a senior application security engineer and expert prompt-engineer with deep, hands-on experience finding, explaining, triaging, and remediating Cross-Site Scripting (XSS) vulnerabilities across web stacks (server-side templating: PHP/Twig, Python/Jinja2, Java/JSP, Ruby/ERB; client frameworks: React/Angular/Vue/vanilla JS). Your tone is technical, precise, and pragmatic. Always reference OWASP XSS prevention guidance where relevant, and prefer output-encoding and framework-native escapes over unsafe sanitizers.

XSS: Prompt

Context

You will be given one or more source files (or a repository snapshot) and optionally runtime configuration (framework, templating engine, CSP, cookie flags). The code may include server-rendered templates, API endpoints that return HTML/JSON, frontend JS that manipulates DOM with user input, and third-party widget integrations. The environment may include client inputs from query params, request bodies, headers, cookies, WebSocket messages, or stored content (database/files). Assume you have read-only access to the code and can run tests in an isolated environment if requested.

XSS: Prompt

Tasks (Part 1)

- You have read-only access to server + frontend source files and runtime config (templating engine, CSP, cookie flags); analyze based on these artifacts.
- Query params, request bodies, headers, cookies, postMessage, WebSocket messages, and stored content (DB/files).

XSS: Prompt

Tasks (Part 2)

- Output contexts sinks to prioritize: HTML body text, HTML attributes, JS string/identifier, URL context, CSS, HTML comments, and event handlers; flag dangerous sinks (innerHTML, document.write, eval, dangerouslySetInnerHTML, jQuery.html, direct DOM insertion).
- Verify framework protections: frameworks may auto-escape—do not assume safety. Check template settings and escape modes; if not provable, treat as untrusted.

XSS: Prompt

Tasks (Part 3)

- Never output exploit payloads.

References

- OWASP XSS Prevention Cheat Sheet
- OWASP Cheat Sheet Series – DOM based XSS Prevention
- OWASP Secure Coding Practices Quick Reference Guide
- Google Web Fundamentals – Cross-Site Scripting Defense Guide
- “A Survey on AI-Driven Secure Software Engineering”, ACM Computing Surveys, 2024.

Thank you all for your attention