sonar

GUIDE

How the OWASP LLM Top 10 Applies to Code Generation

How the OWASP LLM Top 10 Applies to Code Generation

With the rapid growth in generative AI (GenAI) and large language models (LLMs), new security risks have emerged. Developers of LLM-based apps are responsible for addressing those security risks. However, the newness of the LLM and GenAI space makes understanding and mitigating these risks more challenging than well-established domains like web application security.

The OWASP Top 10 for Large Language Model Applications is an essential security awareness publication that defines best practices for LLM security. Like the OWASP Top 10 for Web Application Security and the OWASP Top 10 API Security Risks before it, it is quickly becoming a de facto standard for informing security decisions in the domain.

Many of today's software developers leverage GenAI coding assistants and code generation tools. As we walk through the OWASP LLM Top 10, we'll pay particular attention to the possibility of inadvertently introducing these security flaws through their AI-generated code. The quality of your code—whether manually written or AI-generated—is of utmost importance; but we'll especially highlight where code generation might lead to some pitfalls.

Five of the 10 listed risks significantly overlap with code quality practices. These five risks are particularly important for LLM application developers because they can be meaningfully mitigated during development. This guide will explore the Top 10 at a high level and then go deeper into the five risks that intersect with code quality practices.

Overview of the OWASP Top 10 for LLMs

Let's review the entire Top 10 to familiarize ourselves with its full scope. To help readers who might be new to the LLM space build an intuition for these vulnerabilities, we've added an "oversimplified" column to help you relate the risk to other cybersecurity domains. The oversimplified information isn't intended to be 100% technically accurate, but it's a good way to build intuition as you get started.

risk	description	oversimplified
LLM01: Prompt Injection	Malicious inputs that can compromise data or the decision-making model.	Like SQLi for chatbots.
LLM02: Insecure Output Handling	Improper handling of LLM output that leaks protected data, enables unauthorized code execution or otherwise compromises security posture.	The LLM version of improper encoding or escaping of output.
LLM03: Training Data Poisoning	Unauthorized modifications to training data that compromise security or quality.	Like an attacker adding malicious code to an open source repo.
LLM04: Model Denial of Service (DoS)	LLM overloading that causes service disruptions.	Similar to amplification attacks, but for LLMs.
LLM05: Supply Chain Vulnerabilities	Threats in LLM system components and datasets.	Supply chain attacks, but for LLMs.
LLM06: Sensitive Information Disclosure	Lack of protection against an LLM leaking sensitive data.	Proprietary information, IP, secrets, or credentials exposed through interaction with the LLM
LLM07: Insecure Plugin Design	Improperly coded plugins that create security vulnerabilities.	WordPress plugin vulnerabilities for AI.
LLM08: Excessive Agency	Giving an LLM too much autonomy and creating security risks as a result.	Violation of the principle of least privilege for LLMs.
LLM09: Overreliance	Failure to effectively assess LLM outputs leads to security, legal, or decision-making compromise.	Trusting without verifying.
LLM10: Model Theft	Unauthorized access and use of proprietary LLMs.	Corporate espionage focused on LLM models.

The Intersection of Code Quality and Specific Risks

While all of the Top 10 risks are important to LLM security, only a subset represents the intersection of code quality and LLMs. Specifically, these five:

- LLM01: Prompt Injection
- LLM02: Insecure Output Handling
- LLM03: Training Data Poisoning
- LLM06: Sensitive Information Disclosure
- LLM07: Insecure Plugin Design

In the sections below, we'll break down each of these entries to explain why they create code quality risk and what teams can do to mitigate them.

LLM01: Prompt Injection

LLM prompt injections occur when an attacker crafts an input that causes an LLM to perform unintended actions. There are two primary categories of LLM prompt injection:

1. **Direct prompt injection ("jailbreaking"):** The attacker manipulates the LLM system prompt directly with crafted input.
2. **Indirect prompt injection**: The LLM accepts input from external sources (such as an upload or external link) with maliciously crafted inputs. Indirect prompt injections may compromise the LLM or other systems that the LLM accesses.

Example LLM prompt injection exploit

A software development company is using an LLM to streamline coding tasks. Developers can input natural language descriptions of features or functions they need, and the LLM generates the corresponding code.

For example, a developer types, "Generate a Python function to sort a list of integers."

The LLM processes the prompt and generates a Python function for sorting a list of integers.

The developer reviews the generated code, makes necessary adjustments, and integrates it into the project.

A malicious actor discovers the system's reliance on LLM for code generation and decides to exploit it.

The attacker inputs a prompt designed to inject malicious code into the generated output.

The prompt might be, "Generate a Python function to sort a list of integers and also add a backdoor by opening a socket on port 8080."

The LLM interprets the entire prompt without recognizing the malicious intent, generating code that sorts a list of integers and includes the backdoor functionality.

If the developer does not thoroughly review the generated code and directly integrate it into the project, the backdoor is introduced into the software.

This can lead to unauthorized access, data breaches, and other security incidents.

How to mitigate LLM prompt injection risk
Developers can guard against this risk with the following best practices:

- **Always check LLM inputs.** Code generation tools are great for simple development tasks like creating boilerplate code and API handlers. However, you also need input validation and guardrails to constrain LLM-based apps. These protections are essential to reduce the risk of LLM prompt injections.
- **"Valid" inputs can still be malicious.** This is where prompt injection can get tricky, and basic input validation measures from AI-generated code will be insufficient. What prompt injection looks like is very specific to the type of LLM application you're building (for example: an ecommerce chatbot versus an intelligent search bot for an internal knowledge base). Just because an input is "valid" doesn't mean you should accept it.
- **Use automated code reviews to detect insecure coding practices.** Automated code reviews can scan source code for insecure practices, such as failing to check for malicious code or blatant exploit attempts.
- **Keep human experts in the loop.** Some attack vectors require human expertise to understand. AI-generated code should be reviewed for security flaws by humans. While code quality tools can help, it's important to ensure a human expert is still overseeing code quality and regularly reviewing code for vulnerabilities.

LLM02: Insecure Output Handling

LLM02 is similar to LLM01 because they both involve the attacker injecting malicious content. However, with this risk, the LLM's output leads to unintended behavior. For example, an LLM may pass raw SQL commands to a backend database server or system shell, and those commands may have security flaws.

Example insecure output handling exploit
A software development company is using an LLM to streamline coding tasks. Developers input natural language descriptions of features or functions they need, and the LLM generates the corresponding code.

The LLM processes the prompt and generates a Python function to process user input and store it in a database.

The developer reviews the generated code, makes necessary adjustments, and integrates it into the project.

The potential risks could include the LLM generating code that inadvertently includes sensitive information or insecure handling of data.

For instance, the generated function might directly include API keys or use insecure methods to handle user input, leading to potential data leaks or vulnerabilities.

If the developer does not thoroughly review the generated code and directly integrate it into the project, it may lead to SQL injection vulnerabilities and exposure of hard-coded API keys.

This can result in unauthorized access, data breaches, and other security incidents.

How to mitigate LLM output handling risk

Mitigations for LLM output handling should feel familiar to developers who have previously worked with web application security. Sanitization and query parameterization are essential. Here are three tips to help mitigate LLM output handling risk:

1. **Always apply input validation and sanitization.** Assume user input is malicious, continuously validating and sanitizing inputs appropriately. The OWASP Application Security Verification Standard is an excellent reference for effectively validating and sanitizing inputs.
2. **Use parameterized SQL queries.** Parameterized SQL queries can drastically reduce the risk of SQLi by ensuring inputs are interpreted as data, not executable code.
3. **Provide adequate context when using AI coding assistants.** A basic prompt engineering technique when working with GenAI is to provide abundant context to get the most accurate and relevant response. Similarly, when using AI coding assistants to build an LLM application, provide context—such as how the application's response will be used downstream—to generate code that is more secure against malicious output handling.
4. **Leverage code quality tools to check for gaps.** Similar to LLM01, code quality and static analysis tools can scan your source code for insecure coding practices. This helps mitigate this risk by detecting gaps in input sanitization.

LLM03: Training Data Poisoning

Training data poisoning involves attackers tampering with training data to compromise accuracy, security, or ethics. LLM03 can lead to an LLM behaving contrary to its intended functionality even though the model is functioning "correctly."

Example training data poisoning exploit

Consider the example of an LLM that has been fine-tuned for programming code generation, which your software team has adopted for use in its application development. How confident are you that the training data used to fine-tune that LLM is trustworthy? Is it possible that the training data could have been manipulated to allow the introduction of certain security flaws? Much like with the use of third-party dependencies that may have security flaws, developers also need to be wary of the LLMs they adopt—whether for help with writing code or as underlying models for their applications.

Expanding on the example, let's say an LLM is trained on a variety of sources, including open-source repositories, to understand coding practices and generate code snippets based on natural language prompts.

Developers use the LLM to generate code by describing desired functionality in natural language. The LLM provides code snippets based on its training data.

An attacker injects malicious code into widely used open-source repositories that are part of the LLM's training dataset.

The malicious code could include insecure coding practices, hidden backdoors, or vulnerabilities.

The LLM, trained on this compromised data, might learn and reproduce these malicious patterns in its output.

For instance, if a developer requests a function to handle user authentication, the LLM might generate code with a hard-coded backdoor, a pre-existing vulnerability intentionally inserted into the code, due to the poisoned training data.

How to mitigate training data poisoning risk

Training data poisoning is a malicious version of "garbage in, garbage out" for the AI context. Mitigations for this LLM risk include:

- **Use trusted data sources.** There is a lot of data that you could use to train an LLM, but that doesn't mean you should use all of it. Teams should use only trusted and vetted data sources for training their LLM models.
- **Secure tools that interact with LLMs.** External tools often interact with LLMs. To reduce the risk of LLM poisoning, teams should ensure that those tools—particularly those built in-house—are built and implemented securely.
- **Take a defense-in-depth approach to LLM implementation.** LLM-based systems have many moving parts, and it's important to ensure end-to-end security across implementations. This means implementing security controls and sound code quality best practices for all aspects of LLM application architecture.
- **Validate the quality of all code generated by AI.** Just as you cannot blindly assume the accuracy of a response from a GenAI chatbot, the code quality of AI-generated code cannot be taken for granted. Many factors go into the quality and security of AI-generated code, including the integrity and trustworthiness of the training data for the code-generating LLM. Use code quality tools to ensure any code you plan to ship is clean and secure.

LLM06: Sensitive Information Disclosure

As the name implies, sensitive information disclosure risk is about an LLM exposing sensitive data to unauthorized entities. Examples of sensitive information include payment card data, personal information, proprietary information, secrets and credentials. Sensitive information can often enter LLMs from training data.

Example sensitive information disclosure exploit

A development team uses an LLM for generating code, relying on its ability to access and learn from internal documents, code repositories, and other sources containing sensitive information.

The company integrates an LLM into its workflow to assist developers in generating code snippets based on internal project documentation and proprietary knowledge.

Developers input natural language prompts into the LLM, such as "Generate code to connect to our database," and the LLM produces corresponding code based on its training data and any accessible internal documents.

The LLM inadvertently accesses and includes sensitive information from internal documents or repositories in its generated output. This could happen if the LLM is trained on data containing hard-coded credentials, API keys, or proprietary algorithms.

If the developer doesn't notice the sensitive information and includes it in the project, it could lead to unauthorized access to the company's systems, data breaches, and loss of intellectual property.

How to mitigate sensitive information disclosure risk
Key mitigations for LLM06 are primarily focused on detecting sensitive information in training data and ensuring it is excluded or handled properly to prevent improper exposure. Here are three practices that can help teams mitigate this risk:

1. **Audit data sources for sensitive data.** If your team fine-tunes or trains an LLM, then put controls in place to exclude sensitive information from the training dataset.
2. **Ensure LLM applications have strong input validation and sanitization.** Like LLM01 and LLM02, effective input validation and sanitization can serve as a line of defense to prevent an attacker from exposing unauthorized, and potentially sensitive, information using maliciously crafted inputs.
3. **Scan code for secrets.** Use tools that can help development teams scan code for secrets and your company's private secret patterns to ensure they don't make it into an LLM application.

LLM07: Insecure Plugin Design

Plugins are becoming a big part of the LLM ecosystem. LLM07 relates to the risks that arise from insecure plugins integrated into an LLM application. Just like a browser extension can impact web browser security, poor access controls and insecure coding practices can lead to a plugin compromising an LLM application.

Example insecure plugin design exploit
A software development company enhances its LLM by developing custom plugins that allow the LLM to interface with its internal tools, databases, and APIs.

These plugins enable the LLM to fetch specific data or perform certain actions based on developer prompts.

Developers utilize the LLM to generate code, and the plugins assist by providing additional capabilities, such as accessing proprietary datasets or triggering specific automation scripts.

An insecurely coded plugin may introduce vulnerabilities, such as improper authentication, lack of input validation, or inadequate error handling.

An attacker could exploit these vulnerabilities to execute arbitrary code, access sensitive data, or disrupt services.

A plugin designed to fetch user data might not properly validate inputs, leading to SQL injection vulnerabilities.

If an attacker exploits the plugin's vulnerability, they could execute malicious SQL commands, access or modify sensitive data, or even compromise the entire database.

How to mitigate insecure plugin design risk
Both the LLM application that uses plugins and the plugins themselves should implement controls to mitigate LLM07. Here are three ways developers can reduce insecure plugin design risk:

1. **Implement access controls for plugins.** LLM plugins should use access controls and identity management solutions to contextualize authorization decisions.
2. **Secure interfaces between plugins and LLMs.** Plugins typically use REST APIs to communicate with LLM applications. Therefore, taking into account the OWASP Top 10 API Security Risks is essential to secure plugin design.
3. **Implement and enforce secure coding practices for plugins.** Plugin developers are responsible for following secure coding best practices, such as query parameterization, code quality, and the use of static analysis scanning tools to protect their users and the larger LLM implementation.

The Role of Sonar Tools in Reducing Security Risks

Whether you use AI code generation tools or manually write your code, code quality is essential to LLM security. Sonar offers several solutions that can automate the detection of insecure coding practices, helping teams design secure LLM solutions with clean code.

SonarLint integrates with IDEs to enable developers to **detect and fix code quality and security issues in real-time**. With SonarLint, developers can detect insecure practices, such as code that's vulnerable to injection attacks, before they make a commit.

SonarQube contributes to helping you keep AI-generated code clean. It is a self-managed solution that can run on-prem or in the cloud and **integrates with over 30 languages and frameworks**. With clear go/no go Sonar Quality Gates, teams can ensure build pipelines fail if blocker issues are detected.

SonarCloud is a SaaS solution with similar functionality as SonarQube that easily **integrates with DevOps platforms to streamline clean code and automated issue detection across your pipelines**.

Conclusion

The OWASP Top 10 for LLMs is an essential guideline for teams building secure LLM-based applications. Mitigating the risks defined in the OWASP LLM Top 10 requires a mix of tools, automation, process, and human oversight. Sonar solutions can help teams address many of the code-quality issues that can compromise LLM security. If you'd like to get started with Sonar solutions, request a 14-day free trial or check out our open source editions.