OWASP®

OWASP Top 10 for LLM

VERSION 1.0

***Published:** August 1, 2023*

OWASP.ORG/WWW-PROJECT-TOP-10-FOR-LARGE-LANGUAGE-MODEL-APPLICATIONS

OWASP Top 10 for LLM

LLM01

Prompt Injection

This manipulates a large language model (LLM) through crafty inputs, causing unintended actions by the LLM. Direct injections overwrite system prompts, while indirect ones manipulate inputs from external sources.

LLM02

Insecure Output Handling

This vulnerability occurs when an LLM output is accepted without scrutiny, exposing backend systems. Misuse may lead to severe consequences like XSS, CSRF, SSRF, privilege escalation, or remote code execution.

LLM03

Training Data Poisoning

Training data poisoning refers to manipulating the data or fine-tuning process to introduce vulnerabilities, backdoors or biases that could compromise the model's security, effectiveness or ethical behavior.

LLM04

Model Denial of Service

Attackers cause resource-heavy operations on LLMs, leading to service degradation or high costs. The vulnerability is magnified due to the resource-intensive nature of LLMs and unpredictability of user inputs.

LLM05

Supply Chain Vulnerabilities

LLM application lifecycle can be compromised by vulnerable components or services, leading to security attacks. Using third-party datasets, pre- trained models, and plugins add vulnerabilities.

LLM06

Sensitive Information Disclosure

LLM's may inadvertently reveal confidential data in its responses, leading to unauthorized data access, privacy violations, and security breaches. Implement data sanitization and strict user policies to mitigate this.

LLM07

Insecure Plugin Design

LLM plugins can have insecure inputs and insufficient access control due to lack of application control. Attackers can exploit these vulnerabilities, resulting in severe consequences like remote code execution.

LLM08

Excessive Agency

LLM-based systems may undertake actions leading to unintended consequences. The issue arises from excessive functionality, permissions, or autonomy granted to the LLM-based systems.

LLM09

Overreliance

Systems or people overly depending on LLMs without oversight may face misinformation, miscommunication, legal issues, and security vulnerabilities due to incorrect or inappropriate content generated by LLMs.

LLM10

Model Theft

This involves unauthorized access, copying, or exfiltration of proprietary LLM models. The impact includes economic losses, compromised competitive advantage, and potential access to sensitive information.

LLM01

Prompt Injection

Attackers can manipulate LLM’s through crafted inputs, causing it to execute the attacker's intentions. This can be done directly by adversarially prompting the system prompt or indirectly through manipulated external inputs, potentially leading to data exfiltration, social engineering, and other issues.

EXAMPLES

- Direct prompt injections overwrite system prompts.
- Indirect prompt injections hijack the conversation context.
- A user employs an LLM to summarize a webpage containing an indirect prompt injection.

PREVENTION

- Enforce privilege control on LLM access to backend systems.
- Implement human in the loop for extensible functionality.
- Segregate external content from user prompts.
- Establish trust boundaries between the LLM, external sources, and extensible functionality.

ATTACK SCENARIOS

- An attacker provides a direct prompt injection to an LLM-based support chatbot.
- An attacker embeds an indirect prompt injection in a webpage.
- A user employs an LLM to summarize a webpage containing an indirect prompt injection.

LLM02

Insecure Output Handling

Insecure Output Handling is a vulnerability that arises when a downstream component blindly accepts large language model (LLM) output without proper scrutiny. This can lead to XSS and CSRF in web browsers as well as SSRF, privilege escalation, or remote code execution on backend systems.

EXAMPLES

- LLM output is entered directly into a system shell or similar function, resulting in remote code execution.
- JavaScript or Markdown is generated by the LLM and returned to a user, resulting in XSS.

PREVENTION

- Apply proper input validation on responses coming from the model to backend functions.
- Encode output coming from the model back to users to mitigate undesired code interpretations.

ATTACK SCENARIOS

- An application directly passes the LLM-generated response into an internal function responsible for executing system commands without proper validation.
- A user utilizes a website summarizer tool powered by a LLM to generate a concise summary of an article, which includes a prompt injection.
- An LLM allows users to craft SQL queries for a backend database through a chat-like feature.

LLM03

Training Data Poisoning

Training Data Poisoning refers to manipulating the data or fine-tuning process to introduce vulnerabilities, backdoors or biases that could compromise the model's security, effectiveness or ethical behavior. This risks performance degradation, downstream software exploitation and reputational damage.

EXAMPLES

- A malicious actor creates inaccurate or malicious documents targeted at a model's training data.
- The model trains using falsified information or unverified data which is reflected in output.

PREVENTION

- Verify the legitimacy of targeted data sources during both the training and fine-tuning stages.
- Craft different models via separate training data different use-cases.
- Use strict vetting or input filters for specific training data or categories of data sources.

ATTACK SCENARIOS

- Output can mislead users of the application leading to biased opinions.
- A malicious user of the application may try to influence and inject toxic data into the model.
- A malicious actor or competitor creates inaccurate or falsified information targeted at a model's training data.
- The vulnerability Prompt Injection could be an attack vector to this vulnerability if insufficient sanitization and filtering is performed.

LLM04

Model Denial of Service

Model Denial of Service occurs when an attacker interacts with a Large Language Model (LLM) in a way that consumes an exceptionally high amount of resources. This can result in a decline in the quality of service for them and other users, as well as potentially incurring high resource costs.

EXAMPLES

- Posing queries that lead to recurring resource usage through high-volume generation of tasks in a queue.
- Sending queries that are unusually resource-consuming.
- Continuous input overflow: An attacker sends a stream of input to the LLM that exceeds its context window.

PREVENTION

- Implement input validation and sanitization to ensure input adheres to defined limits, and cap resource use per request or step.
- Enforce API rate limits to restrict the number of requests an individual user or IP address can make.
- Limit the number of queued actions and the number of total actions in a system reacting to LLM responses.

ATTACK SCENARIOS

- Attackers send multiple requests to a hosted model that are difficult and costly for it to process.
- A piece of text on a webpage is encountered while an LLM-driven tool is collecting information to respond to a benign query.
- Attackers overwhelm the LLM with input that exceeds its context window.

LLM05

Supply Chain Vulnerabilities

Supply chain vulnerabilities in LLMs can compromise training data, ML models, and deployment platforms, causing biased results, security breaches, or total system failures. Such vulnerabilities can stem from outdated software, susceptible pre-trained models, poisoned training data, and insecure plugin designs.

EXAMPLES

- Using outdated third-party packages.
- Fine-tuning with a vulnerable pre-trained model.
- Training using poisoned crowd-sourced data.
- Utilizing deprecated, unmaintained models.
- Lack of visibility into the supply chain is.

PREVENTION

- Vet data sources and use independently-audited security systems.
- Use trusted plugins tested for your requirements.
- Apply MLOps best practices for own models.
- Use model and code signing for external models.
- Implement monitoring for vulnerabilities and maintain a patching policy.
- Regularly review supplier security and access.

ATTACK SCENARIOS

- Attackers exploit a vulnerable Python library.
- Attacker tricks developers via a compromised PyPi package.
- Publicly available models are poisoned to spread misinformation.
- A compromised supplier employee steals IP.
- An LLM operator changes T&Cs to misuse application data.

LLM06

Sensitive Information Disclosure

LLM applications can inadvertently disclose sensitive information, proprietary algorithms, or confidential data, leading to unauthorized access, intellectual property theft, and privacy breaches. To mitigate these risks, LLM applications should employ data sanitization, implement appropriate usage policies, and restrict the types of data returned by the LLM.

EXAMPLES

- Incomplete filtering of sensitive data in responses.
- Overfitting or memorizing sensitive data during training.
- Unintended disclosure of confidential information due to errors.

PREVENTION

- Use data sanitization and scrubbing techniques.
- Implement robust input validation and sanitization.
- Limit access to external data sources.
- Apply the rule of least privilege when training models.
- Maintain a secure supply chain and strict access control.

ATTACK SCENARIOS

- Legitimate user exposed to other user data via LLM.
- Crafted prompts used to bypass input filters and reveal sensitive data.
- Personal data leaked into the model via training data increases risk.

LLM07

Insecure Plugin Design

Plugins can be prone to malicious requests leading to harmful consequences like data exfiltration, remote code execution, and privilege escalation due to insufficient access controls and improper input validation. Developers must follow robust security measures to prevent exploitation, like strict parameterized inputs and secure access control guidelines.

EXAMPLES

- Plugins accepting all parameters in a single text field or raw SQL or programming statements.
- Authentication without explicit authorization to a particular plugin.
- Plugins treating all LLM content as user-created and performing actions without additional authorization.

PREVENTION

- Enforce strict parameterized input and perform type and range checks.
- Conduct thorough inspections and tests including SAST, DAST, and IAST.
- Use appropriate authentication identities and API Keys for authorization and access control.
- Require manual user authorization for actions taken by sensitive plugins.

ATTACK SCENARIOS

- Attackers craft requests to inject their own content with controlled domains.
- Attacker exploits a plugin accepting free-form input to perform data exfiltration or privilege escalation.
- Attacker stages a SQL attack via a plugin accepting SQL WHERE clauses as advanced filters.

LLM08

Excessive Agency

Excessive Agency in LLM-based systems is a vulnerability caused by over-functionality, excessive permissions, or too much autonomy. To prevent this, developers need to limit plugin functionality, permissions, and autonomy to what's absolutely necessary, track user authorization, require human approval for all actions, and implement authorization in downstream systems.

EXAMPLES

- An LLM agent accesses unnecessary functions from a plugin.
- An LLM plugin fails to filter unnecessary input instructions.
- A plugin possesses unneeded permissions on other systems.
- An LLM plugin accesses downstream systems with high-privileged identity.

PREVENTION

- Limit plugins/tools that LLM agents can call, and limit functions implemented in LLM plugins/tools to the minimum necessary.
- Avoid open-ended functions and use plugins with granular functionality.
- Require human approval for all actions and track user authorization.
- Log and monitor the activity of LLM plugins/tools and downstream systems, and implement rate-limiting to reduce the number of undesirable actions.

ATTACK SCENARIOS

An LLM-based personal assistant app with excessive permissions and autonomy is tricked by a malicious email into sending spam. This could be prevented by limiting functionality, permissions, requiring user approval, or implementing rate limiting.

LLM09

Overreliance

Overreliance on LLMs can lead to serious consequences such as misinformation, legal issues, and security vulnerabilities. It occurs when an LLM is trusted to make critical decisions or generate content without adequate oversight or validation.

EXAMPLES

- LLM provides incorrect information.
- LLM generates nonsensical text.
- LLM suggests insecure code.
- Inadequate risk communication from LLM providers.

PREVENTION

- Regular monitoring and review of LLM outputs.
- Cross-check LLM output with trusted sources.
- Enhance model with fine-tuning or embeddings.
- Implement automatic validation mechanisms.
- Break tasks into manageable subtasks.
- Clearly communicate LLM risks and limitations.
- Establish secure coding practices in development environments.

ATTACK SCENARIOS

- AI fed misleading info leading to disinformation.
- AI's code suggestions introduce security vulnerabilities.
- Developer unknowingly integrates malicious package suggested by AI.

LLM10

Model Theft

LLM model theft involves unauthorized access to and exfiltration of LLM models, risking economic loss, reputation damage, and unauthorized access to sensitive data. Robust security measures are essential to protect these models.

EXAMPLES

- Attacker gains unauthorized access to LLM model.
- Disgruntled employee leaks model artifacts.
- Attacker crafts inputs to collect model outputs.
- Side-channel attack to extract model info.
- Use of stolen model for adversarial attacks.

PREVENTION

- Implement strong access controls, authentication, and monitor/audit access logs regularly.
- Implement rate limiting of API calls.
- Watermarking framework in LLM's lifecycle.
- Automate MLOps deployment with governance.

ATTACK SCENARIOS

- Unauthorized access to LLM repository for data theft.
- Leaked model artifacts by disgruntled employee.
- Creation of a shadow model through API queries.
- Data leaks due to supply-chain control failure.
- Side-channel attack to retrieve model information.

Key Reference Links

- Arxiv: Prompt Injection attack against LLM-integrated Applications
- Defending ChatGPT against Jailbreak Attack via Self-Reminder
- GitHub: OpenAI Chat Markup Language
- Arxiv: Not what you've signed up for: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection
- AI Village: Threat Modeling LLM Applications
- OpenAI: Safety Best Practices
- Snyk: Arbitrary Code Execution
- Stanford: Training Data
- CSO: How data poisoning attacks corrupt machine learning models
- MITRE: ML Supply Chain Compromise
- MITRE: Tay Poisoning
- Backdoor Attacks on Language Models: Can We Trust Our Model's Weights?
- Arxiv: Poisoning Language Models During Instruction Tuning
- ChatGPT Data Breach Confirmed as Security Firm Warns of Vulnerable Component Exploitation
- What Happens When an AI Company Falls Victim to a Software Supply Chain Vulnerability
- OpenAI: Plugin Review Process
- Compromised PyTorch-nightly dependency chain