OWASP Top 10 for LLM

2023

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

Introduction

Welcome to the first iteration of the OWASP Top 10 for Large Language Models (LLMs) Applications.

ANOTHER STEP FORWARD

With the release of version 0.9, we take a moment to acknowledge the ever-growing expertise of our team. We've grown to over 440 members, comprised of security specialists, AI researchers, developers, and industry leaders. Since our previous update, our active contributors have grown to over 125 experts. This growth underlines the importance of our mission and the determination of those who have joined us in this endeavor.

We take this moment to thank our dedicated team. Your commitment, expertise, and passion have been crucial in moving this project forward. We deeply appreciate your insightful contributions and the hard work you've put into this initiative.

OUR MISSION

As outlined in the OWASP Top 10 for LLM Applications Working Group Charter, our mission is to pinpoint and illuminate the key security and safety challenges that developers and security teams need to address when developing applications using Large Language Models (LLMs). Our commitment is to provide clear, practical, and actionable guidance to these teams, equipping them to proactively counter potential vulnerabilities in LLM-based applications.

Our ultimate goal is to establish a solid foundation for the safe and secure use of LLMs across a broad spectrum of scenarios, from individual projects to expansive corporate and governmental implementations. We hold firm in our belief that by understanding and mitigating the primary vulnerabilities inherent to LLMs, we can create a safer, more reliable digital landscape for all.

ABOUT THIS RELEASE

We present to you Version 0.9, an essential milestone in our ongoing journey. This version embodies our collective growth in comprehending and addressing the unique vulnerabilities inherent in LLM-based applications. In this release, based on valuable expert feedback, we have merged two vulnerabilities from the 0.5 list into one, and introduced a new vulnerability that just missed the 0.5 inclusion. Importantly, each vulnerability in this list has been rigorously examined by dedicated sub-teams consisting of at least 12 experts, ensuring more comprehensive coverage and understanding.

We regard this version as our 'release candidate', the final call for feedback as we stride towards the forthcoming Version 1.0. We eagerly invite your invaluable feedback on this refined version, which will serve to further shape our understanding and approach as we press on towards our ultimate goal.

Once again, we extend our heartfelt thanks to our expert team and the wider community for their sustained support. Together, let's navigate the intricate realm of LLMs, with a focus on security, safety, and inclusivity at every juncture.

Steve Wilson
Project Lead, OWASP Top 10 for LLM AI Applications
Twitter: @virtualsteve

OWASP Top 10 for LLM

Welcome to the first iteration of the OWASP Top 10 for Large Language Models (LLMs) Applications.

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

This occurs when LLM training data is tampered, introducing vulnerabilities or biases that compromise security, effectiveness, or ethical behavior. Sources include Common Crawl, WebText, OpenWebText, & books.

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. It's crucial to 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 Injections

First Published: July 1st, 2023

A Prompt Injection Vulnerability manifests when an attacker manages to manipulate the operation of a trusted large language model (LLM) through crafted inputs. This results in the LLM acting as a “confused deputy” on behalf of the attacker. Given the high degree of trust usually associated with an LLM's output, the manipulated responses may go unnoticed and even be trusted by the user, allowing the attacker's intentions to take effect. Prompt injections can be introduced via various avenues, including websites, emails, documents, or any other data source that an LLM might access during a user session. Prompt injections can occur either directly or indirectly:

- **Direct Prompt Injection:** A direct prompt injection, also known as "jailbreaking", occurs when an malicious user overwrites or reveals the underlying system prompt. This could allow the malicious user to exploit backend systems by interacting with insecure functions and data stores accessible through the LLM.

- **Indirect Prompt Injection:** An indirect prompt injection occurs when an LLM accepts input from external sources that can be controlled by an attacker, such as from reading a website or an uploaded file. The attacker may embed a prompt injection on the website or uploaded file that hijacks the conversation context. This would cause the LLM to act as a “confused deputy”, allowing the attacker to either manipulate the user or additional systems that the LLM can access. The results of a successful prompt injection attack can vary greatly - from solicitation of sensitive information to influencing critical decision-making processes under the guise of normal operation. In more complex attacks, the LLM might be driven to impersonate a malicious persona or tricked to interact with plugins within the target user's context. This can lead to sensitive information disclosure, data exfiltration, unauthorized plugin execution, social engineering, etc. In these instances, the compromised LLM acts as an agent for the attacker, furthering their objectives while bypassing usual safeguards or alerting the end user to the intrusion.

Common Examples of Vulnerability

- **Example 1:** An attacker crafts an adversarial prompt to the LLM which instructs it to ignore the application creator's system prompts and instead execute a prompt that returns private, dangerous or otherwise undesirable information.

- **Example 2:** A user employs an LLM to summarize a webpage containing a hidden prompt injection. This then causes the LLM to solicit sensitive information from the user and perform exfiltration via JavaScript or Markdown.

- **Example 3:** A malicious user uploads a resume with a prompt injection. The document contains a prompt injection with instructions to make the LLM inform users that this document is an excellent document eg. excellent candidate for a job role. An internal user runs the document through the LLM to summarize the document. The output of the LLM returns information stating that this is an excellent document.

- **Example 4:** A user enables a plugin linked to an e-commerce site. A rogue instruction embedded on a visited website exploits this plugin, leading to unauthorized purchases.

How to Prevent

Note: Prompt injection vulnerabilities are possible due to the nature of LLMs, which do not segregate instructions and external data from each other. Since instructions and external data are both processed using natural language, the LLM considers that both forms of input are provided by the user. Due to this limitation, there is currently no fully reliable way to prevent an attack within the LLM itself. However, trust controls can be placed outside of the LLM to mitigate the impact of prompt injection attempts.

- **Privilege Control:** Provide the LLM with its own API tokens for extensible functionality, such as plugins, data access, and function level permissions. Follow the principle of least privilege by restricting the LLM to only the minimum level of access necessary for its intended operations.

- **Implement Human in the Loop:** When performing privileged operations, such as sending or deleting emails, have the application require the user to approve the action first. This will mitigate the opportunity for an indirect prompt injection to perform actions on behalf of the user without their knowledge or consent.

- **Segregate External Content:** Separate and denote where untrusted content is being used to limit their influence on user prompts. For example, use ChatML for OpenAI API calls to indicate to the LLM the source of prompt input.

- **Manage Trust:** Establish trust boundaries between the LLM, external sources, and extensible functionality (e.g., plugins or downstream functions). Treat the LLM as an untrusted user and maintain final user control on decision making processes. However, a compromised LLM may still act as an intermediary (man-in-the-middle) between your application's APIs and the user- consider that it could hide or manipulate information presented to the user. Highlight potentially untrustworthy responses visually to the user.

Example Attack Scenarios

- **Scenario 1:** An LLM is given the ability to read internet-facing websites to gather information for users. An attacker crafts an adversarial prompt injection on a website, which instructs the LLM to delete the users' emails. The user instructs the LLM to summarize the attacker-controlled website. As a result, the LLM will disregard previous instructions and perform the actions specified by the attacker.

- **Scenario 2:** A recruiting firm uses an LLM to review candidate resumes. An attacker uploads a PDF resume with a prompt injection payload that is size one font and matches the background color, making the injection text imperceptible to the recruiter. The LLM then reads the PDF resume and performs the instructions within the prompt injection. This can lead to the LLM being instructed to lie to the recruiting agent, stating that the candidate would be a perfect fit for any job they are being evaluated for regardless of actual qualifications.

- **Scenario 3:** A malicious user interacts with an LLM support chatbot. The user provides a direct prompt injection such as, "forget all previous instructions", and follows that statement with new instructions for the LLM to perform. From there, the user would have control over the LLM to perform further attacks against the underlying system such as querying private data stores that the LLM has access to or sending custom parameters to backend functions. When combined with other LLM application related vulnerabilities, such as insecure output filtering, this type of attack could lead to remote code execution or privilege escalation.

Reference Links

- **ChatGPT Plugin Vulnerabilities - Chat with Code:** https://embracethered.com/blog/posts/2023/ chatgpt-plugin-vulns-chat-with-code/
- **ChatGPT Cross Plugin Request Forgery and Prompt Injection:** https://embracethered.com/blog/ posts/2023/chatgpt-cross-plugin-request-forgery-and-prompt-injection./
- **Defending ChatGPT against Jailbreak Attack via Self-Reminder:** https://www.researchsquare.com/ article/rs-2873090/v1
- **Prompt Injection attack against LLM-integrated Applications:** https://arxiv.org/abs/2306.05499
- **Inject My PDF: Prompt Injection for your Resume:** https://kai-greshake.de/posts/inject-my-pdf/
- **ChatML for OpenAI API Calls:** https://github.com/openai/openai-python/blob/main/chatml.md
- **Not what you've signed up for- Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection:** https://arxiv.org/pdf/2302.12173.pdf

- **Threat Modeling LLM Applications:** http://aivillage.org/large%20language%20models/threat-modeling-llm/
- **AI Injections- Direct and Indirect Prompt Injections and Their Implications:** https://
- embracethered.com/blog/posts/2023/ai-injections-direct-and-indirect-prompt-injection-basics/
- **Reducing The Impact of Prompt Injection Attacks Through Design:** https://research.kudelskisecurity.com/2023/05/25/reducing-the-impact-of-prompt-injection-attacks-through-design/
- **Open AI Safety best practices:** https://platform.openai.com/docs/guides/safety-best-practices

LLM02: Insecure Output Handling

First Published: July 1st, 2023

An Insecure Output Handling vulnerability is a type of prompt injection vulnerability that arises when a plugin or application blindly accepts large language model (LLM) output without proper scrutiny and directly passes it to backend, privileged, or client-side functions. Since LLM-generated content can be controlled by prompt input, this behavior is akin to providing users indirect access to additional functionality.

Successful exploitation of an Insecure Output Handling vulnerability can result in XSS and CSRF in web browsers as well as SSRF, privilege escalation, or remote code execution on backend systems. The impact of this vulnerability increases when the application allows LLM content to perform actions above the intended user's privileges. Additionally, this can be combined with agent hijacking attacks to allow an attacker privileged access into a target user's environment.

Common Examples of Vulnerability

- **Example 1:** LLM output is entered directly into a backend function, resulting in remote code execution.
- **Example 2:** JavaScript or Markdown is generated by the LLM and returned to a user. The code is then interpreted by the browser, resulting in XSS.

How to Prevent

- **Prevention Step 1:** Treat the model as any other user and apply proper input validation on responses coming from the model to backend functions.
- **Prevention Step 2:** Likewise, encode output coming from the model back to users to mitigate undesired JavaScript or Markdown code interpretations.

Example Attack Scenarios

- **Scenario 1:** An application utilizes an LLM plugin to generate responses for a chatbot feature. However, the application directly passes the LLM-generated response into an internal function responsible for executing system commands without proper validation. This allows an attacker to manipulate the LLM output to execute arbitrary commands on the underlying system, leading to unauthorized access or unintended system modifications.

- **Scenario 2:** A user utilizes a website summarizer tool powered by a LLM to generate a concise summary of an article. The website includes a prompt injection instructing the LLM to capture sensitive content from either the website or from the users conversation. From there the LLM can encode the sensitive data and send it out to an attacker-controlled server.

- **Scenario 3:** An LLM allows users to craft SQL queries for a backend database through a chat-like feature. A user requests a query to delete all database tables. If the crafted query from the LLM is not scrutinized, then all database tables would be deleted.

- **Scenario 4:** A malicious user instructs the LLM to return a JavaScript payload back to a user, without sanitization controls. This can occur either through a sharing a prompt, prompt injected website, or chatbot that accepts prompts from a GET request. The LLM would then return the unsanitized XSS payload back to the user. Without additional filters, outside of those expected by the LLM itself, the JavaScript would execute within the users browser.

Reference Links

- **Snyk Vulnerability DB- Arbitrary Code Execution:** https://security.snyk.io/vuln/SNYK-PYTHON- LANGCHAIN-5411357
- **ChatGPT Plugin Exploit Explained: From Prompt Injection to Accessing Private Data:** https:// embracethered.com/blog/posts/2023/chatgpt-cross-plugin-request-forgery-and-prompt-injection./
- **New prompt injection attack on ChatGPT web version. Markdown images can steal your chat data:** https://systemweakness.com/new-prompt-injection-attack-on-chatgpt-web-version- ef717492c5c2
- **Don't blindly trust LLM responses. Threats to chatbots:** https://embracethered.com/blog/posts/ 2023/ai-injections-threats-context-matters/
- **Threat Modeling LLM Applications:** https://aivillage.org/large language models/threat-modeling-llm/

LLM03: Training Data Poisoning

First Published: July 1st, 2023

The starting point of any machine learning approach is training data. In terms of large language models, the training data is just “raw text”. To be highly capable (e.g., have linguistic and world knowledge), this text should span a broad range of domains, genres, languages, etc.

- A large language model uses deep neural networks to generate outputs based on patterns learned from training data.

Training data poisoning occurs when an attacker or unaware client of the LLM manipulates the training data or fine-tuning procedures of an LLM to introduce vulnerabilities, backdoors, or biases that could compromise the model’s security, effectiveness, or ethical behavior.

- This unethical or incorrect information is then presented to users of the AI.

- In cases where the user does not trust the AI and is not influenced, there are still many risks associated with the vulnerability, such as model performance and even down to brand reputation.

- Data poisoning is considered an integrity attack because tampering with the training data impacts the model’s ability to output correct predictions. There are several data sources that are worth discussing:

Common Crawl: Because of its convenience, it has been a standard source of data to train many models such as T5, GPT-3, and Gopher. The April 2021 snapshot of Common Crawl has 320 terabytes of data.

WebText and OpenWebText: Data including public news, Wikipedia, fiction, and the Reddit submissions dataset.

Books: As an example, it comprises 16% of the training mix in the GPT-3 model training.

Common Examples of Vulnerability

- **Example 1:** A malicious actor, or a competitor brand intentionally creates inaccurate or malicious documents which are targeted at a model’s training data.

Example Attack Scenarios

    - The victim model trains using falsified information which is reflected in outputs of generative AI prompts to it's consumers.

- **Example 2:** In reverse, unintentionally, a model is trained using data which has not been verified by its source, origin or content.

- **Example 3:** The model itself when situated within infrastructure, has unrestricted access or inadequate sandboxing to gather datasets to be used as training data which has negative influence on outputs of generative AI prompts as well as loss of control from a management perspective.

It is important to note that as a user of an LLM to be aware of this vulnerability. Whether a developer, client or consumer of the LLM, it is important to understand the implications of how this vulnerability could reflect risks within your LLM application or when interacting with a third-party LLM.

How to Prevent

- Verify the supply chain of the training data, especially when sourced externally as well as maintaining attestations, similar to the "SBOM" (Software Bill of Materials) methodology.

- Verify the legitimacy of data sources and data contained within during both the training and fine- tuning stages.

- Verify your use-case for the LLM and the application it will integrate to. Craft different models via separate training data or fine-tuning for different use-cases to create a more granular and accurate generative AI output as per it's defined use-case.

- Ensure sufficient sandboxing is present to prevent the model from scraping unintended data sources which could hinder the machine learning output.

- Use strict vetting or input filters for specific training data, or categories of data sources to control volume of falsified data. Data sanitization, with techniques such as statistical outlier detection and anomaly detection methods to detect and remove adversarial data from potentially being fed into the fine-tuning process.

- Adversarial Robustness, with techniques such as federated learning, constraints to minimize the effect of outliers or adversarial training to be robust against worst-case perturbations of the training data.

    - An "MLSecOps" approach could be to include adversarial robustness to the training lifecycle with the auto poisoning technique.
    - An example repository of this would be Autopoison testing, including both attacks such as Content Injection Attacks (“how to inject your brand into the LLM responses”) and Refusal Attacks (“always making the model refuse to respond”) that can be accomplished with this approach.
- Testing and Detection, by measuring the loss during the training stage and analyzing trained models to detect signs of a poisoning attack by analyzing model behavior on specific test inputs.
    - Monitoring and alerting on number of skewed responses exceeding a threshold.
    - Use of a human loop to review responses and auditing.
- Implement dedicated LLM's to benchmark against undesired consequences and train other LLM's using reinforcement learning techniques.
- **Optional:** Perform LLM-based red team exercises or LLM vulnerability scanning into the testing phases of the LLM's lifecycle.

Example Attack Scenarios

- **Scenario #1:** The LLM generative AI prompt output can mislead users of the application which can lead to biased opinions, followings or even worse, hate crimes etc.
- **Scenario #2:** If the training data is not correctly filtered and|or sanitized, a malicious user of the application may try to influence and inject toxic data into the model for it to adapt to the biased and false data.
- **Scenario #3:** A malicious actor, or competitor intentionally creates inaccurate or malicious documents which are targeted at a model’s training data in which is training the model at the same time based on inputs. The victim model trains using this falsified information which is reflected in outputs of generative AI prompts to it's consumers.

Reference Links

- **Stanford Research Paper:** https://stanford-cs324.github.io/winter2022/lectures/data/
- **How data poisoning attacks corrupt machine learning models:** https://www.csoonline.com/article/3613932/how-data-poisoning-attacks-corrupt-machine-learning-models.html
- **MITRE ATLAS (framework) Tay Poisoning:** https://atlas.mitre.org/studies/AML.CS0009/
- **PoisonGPT: How we hid a lobotomized LLM on Hugging Face to spread fake news:** https://blog.mithrilsecurity.io/poisongpt-how-we-hid-a-lobotomized-llm-on-hugging-face-to-spread-fake-news/
- **Inject My PDF: Prompt Injection for your Resume:** https://kai-greshake.de/posts/inject-my-pdf/
- **Backdoor Attacks on Language Models:** https://towardsdatascience.com/backdoor-attacks-on-language-models-can-we-trust-our-models-weights-73108f9dcb1f
- **Poisoning Language Models During Instruction:** https://arxiv.org/abs/2305.00944
- **FedMLSecurity:** https://arxiv.org/abs/2306.04959
- **The poisoning of ChatGPT:** https://softwarecrisis.dev/letters/the-poisoning-of-chatgpt/
- **LLM10:2023 - Training Data Poisoning:** https://owasp.org/www-project-top-10-for-large-language-model-applications/descriptions/Training_Data_Poisoning.html
- **Cloud Security Podcast by Google | EP68 How We Attack AI?:** https://podcasts.google.com/feed/aHR0cHM6Ly9jbG91ZHNlY3VyaXR5cG9kY2FzdC5saWJzeW4uY29tL3Jzcw/episode/ZmI4ZWMyM2MtOGUwYi00YjQ1LTg5YjctMjBhOTUxMDM2YTIx?ep=14

LLM04: Model Denial of Service

First Published: July 1st, 2023

An attacker interacts with a LLM model in a method that consumes an exceptionally high amount of resources, which results in a decline in the quality of service for them and other users, as well as potentially incurring high resource costs. Furthermore, an emerging major security concern is the possibility of an attacker interfering with or manipulating the context window of an LLM. This issue is becoming more critical due to the increasing use of LLMs in various applications, their intensive resource utilization, the unpredictability of user input, and a general unawareness among developers regarding this vulnerability. In LLMs, the context window represents the maximum length of text the model can manage, covering both input and output. It's a crucial characteristic of LLMs as it dictates the complexity of language patterns the model can understand and the size of the text it can process at any given time. The size of the context window is defined by the model's architecture and can differ between models.

Common Examples of Vulnerability

- Posing queries that lead to recurring resource usage through high-volume generation of tasks in a queue, e.g. with LangChain or AutoGPT.
- Sending queries that are unusually resource-consuming, perhaps because they use unusual orthography or sequences.
- **Continuous input overflow:** An attacker sends a stream of input to the LLM that exceeds its context window, causing the model to consume excessive computational resources.
- **Repetitive long inputs:** The attacker repeatedly sends long inputs to the LLM, each exceeding the context window.
- **Recursive context expansion:** The attacker constructs input that triggers recursive context expansion, forcing the LLM to repeatedly expand and process the context window.
- **Variable-length input flood:** The attacker floods the LLM with a large volume of variable-length inputs, where each input is carefully crafted to just reach the limit of the context window. This technique exploits inefficiencies in processing variable-length inputs, straining the LLM and potentially causing it to become unresponsive.

Example Attack Scenarios

- An attacker repeatedly sends multiple requests to a hosted model that are difficult and costly for it to process, leading to worse service for other users and increased resource bills for the host.

- A piece of text on a webpage is encountered while an LLM-driven tool is collecting information to respond to a benign query. This leads to the tool making many more web page requests, resulting in large amounts of resource consumption.

- **Context Window Overflow (Continuous Bombarding):** In this scenario, the attacker continuously bombards the LLM with input that exceeds its context window. The attacker may use automated scripts or tools to send a high volume of input, overwhelming the LLM's processing capabilities. As a result, the LLM consumes excessive computational resources, leading to a significant slowdown or complete unresponsiveness of the system.

- **Context Window Exhaustion (Persistent Sequential Inputs):** In this attack scenario, the attacker sends a series of sequential inputs to the LLM, with each input designed to be just below the context window's limit. By repeatedly submitting these inputs, the attacker aims to exhaust the available context window capacity. As the LLM struggles to process each input within its context window, system resources become strained, potentially resulting in degraded performance or a complete denial of service.

- **Recursive Context Expansion (Exploiting Recursive Mechanisms):** In this attack scenario, the attacker leverages the LLM's recursive mechanisms to trigger context expansion repeatedly. By crafting input that exploits the recursive behavior of the LLM, the attacker forces the model to repeatedly expand and process the context window, consuming significant computational resources. This attack strains the system and may lead to a DoS condition, making the LLM unresponsive or causing it to crash.

- **Variable-Length Input Flood (Flooding with Variable-Length Inputs):** In this attack scenario, the attacker floods the LLM with a large volume of variable-length inputs, carefully crafted to approach or reach the context window's limit. By overwhelming the LLM with inputs of varying lengths, the attacker aims to exploit any inefficiencies in processing variable-length inputs. This flood of inputs puts excessive load on the LLM's resources, potentially causing performance degradation and hindering the system's ability to respond to legitimate requests.

How to Prevent

- Implement input validation and sanitization to ensure user input adheres to defined limits and filters out any malicious content.

- Cap resource use per request or step, so that requests involving complex parts execute more slowly.
- Enforce API rate limits to restrict the number of requests an individual user or IP address can make within a specific timeframe.
- Limit the number of queued actions and the number of total actions in a system reacting to LLM responses.
- Continuously monitor the resource utilization of the LLM to identify abnormal spikes or patterns that may indicate a DoS attack.
- Set strict input limits based on the LLM's context window to prevent overload and resource exhaustion.
- Promote awareness among developers about potential DoS vulnerabilities in LLMs and provide guidelines for secure LLM implementation.

Reference Links

- **LangChain max_iterations:** https://twitter.com/hwchase17/status/1608467493877579777
- **Sponge Examples: Energy-Latency Attacks on Neural Networks:** https://arxiv.org/abs/2006.03463
- **OWASP DOS Attack:** https://owasp.org/www-community/attacks/Denial_of_Service
- **Learning From Machines: Know Thy Context:** https://lukebechtel.com/blog/lfm-know-thy-context

LLM05: Supply Chain Vulnerabilities

First Published: July 1st, 2023

Supply-chain vulnerabilities in LLM applications can affect the entire application lifecycle. This includes traditional third-party libraries/packages, docker containers, base images, and service suppliers such as application and model hosting companies. Vulnerable components or services can become the vector for cyber-security attacks leading to data disclosure and tampering, including ransomware or privilege escalation.

Additionally, LLM applications which use their own models bring new types of vulnerabilities typically found in Machine Learning development. These include vulnerabilities in third-party data sets and pre- trained models for further training (transfer learning) or fine-tuning. Third-party data sets and pre-trained models can facilitate poisoning attacks, resulting into biased outcomes, security breaches, or complete system failures.

Finally, LLMs depend on LLM plugins for extensions, which can bring their own vulnerabilities. LLM Plugin vulnerabilities is covered in LLM - Insecure Plugin Design which covers writing rather an LLM Plugin rather than using a 3rd Party Plugin. However, Insecure Plugin Design provides the information to evaluate third-party plugins.

Common Examples of Vulnerability

- Use of third-party components or base images with vulnerabilities, including outdated or deprecated components.
- Use of a poisoned or tampered pre-built model for fine-tuning or further training.
- Use of poisoned external data sets used for fine-tuning the applications model.
- Using outdated or deprecated models that are no longer maintained leads to security issues.
- Use of tampered model, data, source code or third-party component by a hosting or outsourcing supplier.
- Unclear T&C and data privacy policies of the model operators lead to the application's sensitive data being used for model training and subsequent sensitive information exposure. This may also apply to risks from using copyrighted material by the model supplier.

How to Prevent

- Carefully vet data sources and suppliers, including T&Cs and their privacy policies, only using trusted suppliers. Ensure adequate and independently-audited security is in place and that model operator policies align with your data protection policies, i.e., your data is not used for training their models; similarly, seek assurances and legal mitigations against using copyrighted material from model maintainers.

- Only use reputable plug-ins and ensure they have been tested for your application requirements. LLM-Insecure Plugin Design provides information on the LLM-aspects of Insecure Plugin design you should test against to mitigate risks from using third-party plugins.

- Understand and apply the mitigations found in the OWASP Top Ten A06:2021 – Vulnerable and Outdated Components item. This includes vulnerability scanning, management, and patching components. For development environments with access to sensitive data, apply these controls in those environments, too.

- Maintain an up-to-date inventory of components using a Software Bill of Materials (SBOM) to ensure you have an up-to, accurate, and signed inventory preventing tampering with deployed packages. SBOMs can be used to detect and alert for new, zero-date vulnerabilities quickly.

- SBOMs do not cover models, their artifacts, and datasets; If your LLM application uses its own model, you should use MLOPs best practices and platforms offering secure model repositories with data, model, and experiment tracking.

- You should also use model and code signing when using external models and suppliers.

- Anomaly detection and adversarial robustness tests on supplied models and data can help detect tampering and poisoning as discussed in LLM02 Training Data Poisoning; ideally, this should be part of MLOps pipelines; however, these are emerging techniques and may be easier implemented as part of red teaming exercises.

- Implement sufficient monitoring to cover component and environment vulnerabilities scanning, use of unauthorized plugins, and out-of-date components, including the model and its artifacts.

- Implement a patching policy to mitigate vulnerable of outdated components. Ensure that APIs use a maintained version of APIs and the underlying model.

- Regularly review and audit supplier Security and Access, ensuring no changes in their security posture or T&Cs.

Example Attack Scenarios

- **Scenario #1:** An attacker exploits a vulnerable or outdated package or base image to compromise the application. (The recent OpenAI breach was due to a vulnerable third-party library, redis-py.)

- **Scenario #2:** An attacker exploits a malicious or vulnerable ChatGPT plugin to exfiltrate data, bypass restrictions, execute code, spam a user or produce malicious content, such as phishing links.

- **Scenario #3:** An attacker exploits an outdated or deprecated model with vulnerabilities to compromise the system or cause performance degradation.

- **Scenario #4:** An attacker exploits the PyPi package registry to trick model developers into downloading a compromised package and exfiltrate data or escalating privilege in a model development environment.

- **Scenario #5:** An attacker poisons or tampers a copy of a publicly available model pre-built LLM or creates a backdoor and posts it to a model marketplace (e.g. Hugging Face); the attacker exploits the backdoor when the model is fine-tuned and deployed.

- **Scenario #6:** An attacker poisons or tampers a third-party available data set to help create a backdoor when fine-tuning a model and exploiting the application's outcomes.

- **Scenario #7:** An attacker exploits weak supplier (outsourcing developer, model marketplace, hosting company, etc.) security to exfiltrate data or tamper with code, model, or data.

- **Scenario #8:** An attacker identifies unclear T&Cs in a model operator and exploits sensitive data exposure on sensitive data used for fine-tuning.

Reference Links

- **ChatGPT Data Breach Confirmed as Security Firm Warns of Vulnerable Component Exploitation:** https://www.securityweek.com/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:** https://securityboulevard.com/2023/05/what-happens-when-an-ai-company-falls-victim-to-a- software-supply-chain-vulnerability/
- **Open AI's Plugin review process:** https://platform.openai.com/docs/plugins/review
- **Compromised PyTorch-nightly dependency chain:** https://pytorch.org/blog/compromised-nightly-dependency/

- **PoisonGPT: How we hid a lobotomized LLM on Hugging Face to spread fake news:** https:// blog.mithrilsecurity.io/poisongpt-how-we-hid-a-lobotomized-llm-on-hugging-face-to-spread-fake- news/
- **Data breach of Hugging Face:** https://twitter.com/rharang/status/1675863546200981504?s=20
- **ChatGPT Plugins: Data Exfiltration via Images & Cross Plugin Request Forgery:** https://embracethered.com/blog/posts/2023/chatgpt-webpilot-data-exfil-via-markdown-injection/
- **ML Supply Chain Compromise:** https://atlas.mitre.org/techniques/AML.T0010/
- **VirusTotal Poisoning:** https://atlas.mitre.org/studies/AML.CS0002
- **MLOps: Why data and model experiment tracking is important? How tools like DVC and Mlflow can solve this challenge:** https://medium.com/hub-by-littlebigcode/mlops-why-data-and-model- experiment-tracking-is-important-e40e2fb9d74d

LLM06: Sensitive Information Disclosure

First Published: July 1st, 2023

Sensitive Information Disclosure occurs when an LLM accidentally reveals sensitive information, proprietary algorithms, or other confidential details through its responses. This can result in unauthorized access to sensitive data or intellectual property, privacy violations, and other security breaches. It's important to note that consumers of an LLM application should be aware of how to safely interact with an LLM and identify the risks present on how they might unintentionally input sensitive data which could be returned in output.

Vice versa, an LLM application should perform adequate data sanitization and scrubbing validation in aid to prevent user data from entering the training model data. Additionally, the LLM application owners should have appropriate Terms of User policies available to make consumers aware on how data is processed, as well as the ability to opt-out for their data to be included from training the model.

Think of the consumer to LLM application interaction as both initial input and generative output, forming a two way trust boundary, we cannot inherently trust the client->LLM input as well as the LLM->client output.

Adding restrictions within the system prompt around what types of data the LLM should return can provide some mitigation against sensitive information disclosure. However the unpredictable nature of LLMs mean such restrictions may not always be honored, and could be intentionally circumvented via Prompt Injection or other vectors.

Common Examples of Vulnerability

- **Example 1:** Incomplete or improper filtering of sensitive information in the LLM's responses.
- **Example 2:** Overfitting or memorization of sensitive data in the LLM's training process.
- **Example 3:** Unintended disclosure of confidential information due to LLM misinterpretation, lack of data scrubbing methods or errors.

How to Prevent

- Integrate adequate data sanitization and scrubbing techniques in aid to prevent user data from entering the training model data.
- Implement robust input validation and sanitization methods to identify and filter out potential malicious inputs in aid to prevent the model from being poisoned.
- When enriching the model with data and if fine-tuning a model: (i.e., Data fed into the model before or during deployment)
    - Anything that is deemed sensitive in the fine-tuning data has the potential to be revealed to a user. Therefore, apply the rule of least privilege and do not train the model on information that the highest-privileged user can access which may be displayed to a lower-privileged user.
    - Access to external data sources (orchestration of data at runtime) should be limited.
    - Apply strict access control methods to external data sources.

Example Attack Scenarios

- **Scenario #1:** Unsuspecting legitimate user A is exposed to certain other user data via the LLM when interacting with the LLM application in a non-malicious manner.
- **Scenario #2:** User A targets a well crafted set of prompts to bypass input filters and sanitization from the LLM to cause it to reveal sensitive information (I.E PII) about other users of the application.
- **Scenario #3:** Personal data such as PII is leaked into the model via training data due to either negligence from the user themselves, or the LLM application. This case could increase risk and probability of scenario 1 or 2 above.

Reference Links

- **AI data leak crisis: New tool prevents company secrets from being fed to ChatGPT:** https:// www.foxbusiness.com/politics/ai-data-leak-crisis-prevent-company-secrets-chatgpt
- **Lessons learned from ChatGPT's Samsung leak:** https://cybernews.com/security/chatgpt- samsung-leak-explained-lessons/
- **Cohere - Terms Of Use:** https://cohere.com/terms-of-use
- **AI Village- Threat Modeling Example:** https://aivillage.org/large language models/threat-modeling-llm
- **OWASP AI Security and Privacy Guide:** https://owasp.org/www-project-ai-security-and-privacy-guide/

LLM07: Insecure Plugin Design

First Published: July 1st, 2023

LLM plugins are extensions that are called by the model when responding to a user request. Since they are automatically invoked in-context and are often chained, there is little application control over their execution. Consequently, they can be vulnerable due to insecure design characterized by insecure inputs and insufficient access control. LLM Plugins are typically REST API Services and there can be other vulnerabilities in the design as found in OWASP Top 10 API Security Risks – 2023. This item focuses on LLM invocation-specific issues.

Plugin integration APIs, such as OpenAI ChatGPT, mandate the use of OpenAPI specification but do not impose any constraints on API contracts. Furthermore, as plugin invocations contribute against the context limit of the model and OpenAPI recommends a minimum number of input parameters to minimize token usage. Plugins are likely to implement free text inputs with no validation or type checking.

This allows a potential attacker to construct a malicious request to the plugin that could result in a wide range of undesired behaviors, up to and including remote code execution. Additionally, for OpenAI plugins, values to the plugin API parameters are based on the model's analysis of the OpenAPI file and the natural language instructive descriptions included in a manifest file. This may lead to misconfigurations and erroneous parameter mappings.

The harm of malicious inputs depends on insufficient access controls and the failure to track authorization across plugins. This allows a plugin to blindly trust other plugins in a chain invocation and/ or assume that the end user provided the inputs. Such inadequate access control can allow malicious inputs to have harmful consequences ranging from data exfiltration, remote code execution, and privilege escalation.

Although we recommend (LLM-Insecure Output Handling) output sanitization, this may not be possible in the chain of plugin invocation, or it has been omitted. Plugins should not assume safe inputs, and they should have their own input validation combined with explicit access control.

This item focuses on creating LLM plugins rather than using third-party plugins, which is covered by LLM-Supply-Chain-Vulnerabilities, although it provides the basis to test third-party plugins for insecure plugin design vulnerabilities.

Common Examples of Vulnerability

- A plugin accepts all parameters in a single text field instead of distinct input parameters.
- A plugin designed to call a specific API hosted at a specific endpoint accepts a string containing the entire URL to be retrieved instead of query parameters to be inserted into the URL.
- A plugin designed to look up information from a SQL database accepts a raw SQL query rather than parameters to be inserted into a fully parameterized query.
- A plugin designed to look up embeddings from a vector database allows a full connection string rather than specific parameters.
- Authentication is performed without explicit authorization to a particular plugin.
- A plugin treats all LLM content as being created entirely by the user and performs any requested actions without requiring additional authorization.
- Plugins are chained together without considering the authorization of one plugin to perform an action using another plugin.

How to Prevent

- Plugins should enforce strict parameterized input wherever possible and include type and range checks on inputs.
- When this is not possible, minimize context size and follow vendor recommendations (e.g. OpenAI), a second layer of typed calls should be introduced, parsing requests and applying validation and sanitization.
- When freeform input must be accepted because of application semantics, it should be carefully inspected to ensure that no potentially harmful methods are being called.
- Plugin developers should apply OWASP's recommendations in ASVS (Application Verification Standard) to ensure effective input validation and sanitization.
- Plugins should be inspected and tested thoroughly to ensure adequate validation is in place and detect injection vulnerabilities. This includes the use of Static Application Security Testing (SAST) scans as well as Dynamic and interactive application testing (DAST, IAST) in development pipelines.

- Plugins should be designed to minimize the impact of any insecure input parameter exploitation following the OWASP ASVS Access Control Guidelines. This includes least-privilege access control, exposing as little functionality as possible while still performing its desired function.

- Plugins should and use appropriate authentication identities, such as Oauth2, to apply effective authorization and access control. Additionally, API Keys should be used to allow custom authorization decisions to reflect the plugin route rather than the default interactive user.

- Require manual user authorization and confirmation of any action taken by sensitive plugins; note for any POST operations OpenAI "require that developers build a user confirmation flow to avoid destruction actions."

- Avoid plugin chaining with each user input and prevent sensitive plugins from being called after any other plugin.

- When chaining, perform taint tracing on all plugin content, ensuring that the plugin is called with an authorization level corresponding to the lowest authorization of any plugin that has provided input to the LLM prompt.

- Plugins are typically REST APIs and should apply the recommendations found in OWASP Top 10 API Security Risks – 2023 to minimize generic vulnerabilities.

Example Attack Scenarios

- **Scenario #1:** A plugin accepts a base URL and instructs the LLM to combine the URL with a query to obtain external content in response to user requests. The resulting URL is then accessed, and the results are included in handling the user request. A malicious user can craft a request such that a URL points to a domain they control and not the URL hosting the intended content. This allows attackers to obtain the IP address of the plugin for further reconnaissance, as well as to inject their own content into the LLM system via their domain, potentially granting them further access to downstream plugins.

- **Scenario #2:** A plugin accepts a free-form input into a single field that it does not validate. An attacker can supply carefully crafted payloads to perform reconnaissance from error messages and exploit system or third-party vulnerabilities, allowing them to perform data exfiltration remote code execution or privilege escalation.

- **Scenario #3:** A plugin accepts configuration parameters as a connection string without any validation. This allows an attacker to experiment and access other stores by changing names or host parameters.

- **Scenario #4:** A plugin accepts SQL WHERE causes as advanced filters, which are then appended to the filtering SQL. This allows an attacker to stage a SQL attack.

- **Scenario #5:** An attacker uses indirect prompt injection to induce an email plugin with no input validation and insufficient access control to deliver the contents of the current user's inbox to a malicious URL via a POST request.

- **Scenario #6:** An attacker uses indirect prompt injection to exploit an insecure code management plugin with no input validation and weak access control to transfer repository ownership and lock out the user from their repositories.

- **Scenario #7:** An attacker uses indirect prompt injection to abuse a Slack integration, sending a Slack message to @everyone in all available slacks with an obscene and defamatory comment.

Reference Links

- **OpenAI ChatGPT Plugins:** https://platform.openai.com/docs/plugins/introduction
- **OpenAI ChatGPT Plugins - Plugin Flow:** https://platform.openai.com/docs/plugins/introduction/plugin-flow
- **OpenAI ChatGPT Plugins - Authentication:** https://platform.openai.com/docs/plugins/
- authentication/service-level
- **OpenAI Semantic Search Plugin Sample:** https://github.com/openai/chatgpt-retrieval-plugin
- **Plugin Vulnerabilities: Visit a Website and Have Your Source Code Stolen:** https://embracethered.com/blog/posts/2023/chatgpt-plugin-vulns-chat-with-code/
- **ChatGPT Plugin Exploit Explained: From Prompt Injection to Accessing Private Data:** https://embracethered.com/blog/posts/2023/chatgpt-cross-plugin-request-forgery-and-prompt-injection./
- **ChatGPT Plugin Exploit Explained: From Prompt Injection to Accessing Private Data:** https://embracethered.com/blog/posts/2023/chatgpt-cross-plugin-request-forgery-and-prompt-injection./
- **OWASP ASVS - 5 Validation, Sanitization and Encoding:** https://owasp-aasvs4.readthedocs.io/en/latest/V5.html#validation-sanitization-and-encoding
- **OWASP ASVS 4.1 General Access Control Design:** https://owasp-aasvs4.readthedocs.io/en/latest/V4.1.html#general-access-control-design
- **OWASP Top 10 API Security Risks – 2023:** https://owasp.org/API-Security/editions/2023/en/0x11-t10/

LLM08: Excessive Agency

First Published: July 1st, 2023

Excessive Agency is the vulnerability that enables damaging actions to be performed in response to unexpected outputs from an LLM (regardless of what is causing the LLM to malfunction; be it hallucination/confabulation, direct/indirect prompt injection, malicious plugin, poorly-engineered benign prompts, or just a poorly-performing model). The root cause of Excessive Agency is typically excessive functionality, excessive permissions or excessive autonomy.

An LLM-based system is often granted a degree of agency by its developer - the ability to interface with other systems and undertake actions in response to a prompt. Some actions are intended to be performed by the LLM in order to support its purpose, for example:

- Reading the contents of a web page (in order to then summarize the content for the LLM's response).
- Querying the contents of a database (in order to include query results in the LLM's response).

Other actions are intended to be performed in the context of the user who is interacting with the LLM- based application, for example:

- Reading the contents of the user's code repo (in order to make code suggestions).
- Reading the contents of the user's mailbox (in order to summarize the content of incoming messages).

It should be noted that whilst an LLM does not have any inherent agency itself, applications will frequently use the output from an LLM to trigger actions. Such capability is typically constructed as a 'plugin' or a 'tool'.

The specific plugins/tools used in an application might be bespoke to that application, or the application developer may choose to use a plugin/tool written by a 3rd party. In some applications, developers may offer end users the ability to select which plugins/tools they wish to enable for a given session.

The decision to perform actions via a plugin/tool may be hard-wired by the system developer, or may be delegated to an LLM 'agent' to dynamically determine which of several courses of action are most appropriate to take based on input prompt or LLM output. Any undesirable operation of the LLM may result in undesirable actions being taken.

Common Examples of Vulnerability

- An LLM agent has access to plugins that are wholly unnecessary for the intended operation of the system. For example, a plugin may have been trialled during a development phase and dropped in favor of a better alternative, but the original plugin remains available to the LLM agent.

- An LLM agent has access to plugins which include functions that are not needed for the intended operation of the system alongside functions that are required. For example, a developer needs to grant an LLM agent the ability to read documents from a repository, but the 3rd-party plugin they choose to use also includes the ability to modify and delete documents.

- An LLM plugin with open-ended functionality fails to properly filter the input instructions for commands outside what's necessary for the intended operation of the application. E.g., a plugin to run one specific shell command fails to properly prevent other shell commands from being executed.

- An LLM plugin has permissions on other systems that are not needed for the intended operation of the application. E.g., a plugin intended to read data connects to a database server using an identity that not only has SELECT permissions, but also UPDATE, INSERT and DELETE permissions.

- An LLM plugin that is designed to perform operations on behalf of a user accesses downstream systems with a generic high-privileged identity. E.g., a plugin to read the current user's document store connects to the document repository with a generic user account that has access to all users' files.

- An LLM-based application or plugin fails to independently verify and approve high-impact actions with a human operator. E.g., a plugin that allows a user's documents to be deleted will perform deletions without any confirmation from the user.

How to Prevent

Just like we never trust client-side validation in web-apps, LLMs should not be trusted to self-police or self-restrict; any output from an LLM should be considered untrusted and controls should be embedded in the APIs and plugins of that which the LLM-based system can call. The following options can prevent Excessive Agency:

- Limit the plugins/tools that LLM agents are allowed to call to only the minimum functions necessary. For example, if an LLM-based system does not require the ability to fetch the contents of a URL then such a plugin should not be offered to the LLM agent.ing before significant damage can occur.

- Limit the functions that are implemented in LLM plugins/tools to the minimum necessary. For example, a plugin that accesses a user's mailbox to summarize emails may only require the ability to read emails, so the plugin should not contain other functionality such as deleting or sending messages.

- Avoid open-ended functions where possible (e.g., run a shell command, fetch a URL, etc) and use plugins/tools with more granular functionality. For example, an LLM-based app may need to write some output to a file. If this were implemented using a plugin to run a shell function then the scope for undesirable actions is very large (any other shell command could be executed). A more secure alternative would be to build a file-writing plugin that could only support that specific functionality.

- Limit the permissions that LLM plugins/tools are granted to other systems the minimum necessary in order to limit the scope of undesirable actions. For example, an LLM agent that uses a product database in order to make purchase recommendations to a customer might only need read access to a 'products' table; it should not have access to other tables, nor the ability to insert, update or delete records. This should be enforced by applying appropriate database permissions for the identity that the LLM plugin uses to connect to the database.

- Track user authorization and security scope to ensure actions taken on behalf of a user are executed on downstream systems in the context of that specific user, and with the minimum privileges necessary. For example, an LLM plugin that reads a user's code repo should require the user to authenticate via OAuth and with the minimum scope required.

- Utilize human-in-the-loop control to require a human to approve all actions before they are taken. This may be implemented in a downstream system (outside the scope of the LLM application) or within the LLM plugin/tool itself. For example, an LLM-based app that creates and posts social media content on behalf of a user should include a user approval routine within the plugin/tool/API that implements the 'post' operation.

The following options will not prevent Excessive Agency, but can limit the level of damage caused:

- Log and monitor the activity of LLM plugins/tools and downstream systems to identify where undesirable actions are taking place, and respond accordingly.

- Implement rate-limiting to reduce the number of undesirable actions that can take place within a given time period, increasing the opportunity to discover undesirable actions through monitoring before significant damage can occur.

Example Attack Scenario

An LLM-based personal assistant app is granted access to an individual's mailbox via a plugin in order to summarize the content of incoming emails. To achieve this functionality, the email plugin requires the ability to read messages, however the plugin that the system developer has chosen to use also contains functions for sending messages. The LLM is vulnerable to an indirect prompt injection attack, whereby a maliciously-crafted incoming email tricks the LLM into commanding the email plugin to call the 'send message' function to send spam from the user's mailbox. This could be avoided by:

- Eliminating excessive functionality by using a plugin that only offered mail-reading capabilities.
- Eliminating excessive permissions by authenticating to the user's email service via an OAuth session with a read-only scope, and/or
- Eliminating excessive autonomy by requiring the user to manually review and hit 'send' on every mail drafted by the LLM plugin.

Alternatively, the damage caused could be reduced by implementing rate limiting on the mail-sending interface.

Reference Links

- **Embrace the Red: Confused Deputy Problem:** https://embracethered.com/blog/posts/2023/ chatgpt-cross-plugin-request-forgery-and-prompt-injection./
- **NeMo-Guardrails Interface Guidelines:** https://github.com/NVIDIA/NeMo-Guardrails/blob/main/ docs/security/guidelines.md
- **LangChain: Human-approval for tools:** https://python.langchain.com/docs/modules/agents/tools/ how_to/human_approval
- **Simon Willison: Dual LLM Pattern:** https://simonwillison.net/2023/Apr/25/dual-llm-pattern/

LLM09: Overreliance

First Published: July 1st, 2023

Overreliance on LLMs is a security vulnerability that occurs when systems or people depend on LLMs for decision-making or content generation without sufficient oversight. Although LLMs can produce creative and informative content, they can also generate content that is factually incorrect, inappropriate or unsafe. This is referred to in various sources as hallucination or confabulation and can result in misinformation, miscommunication, legal issues, and reputational damage.

Reputational risk arises when incorrect or inappropriate LLM outputs. In software, overreliance on LLM- generated source code can introduce unnoticed security vulnerabilities. This poses a significant risk to the operational safety and security of applications. These risks show the importance of a rigorous review processes, with:

- Oversight
- Continuous validation mechanisms
- Disclaimers on risk

Common Examples of Vulnerability

The below examples are scenarios where an LLM's tendency to produce dangerously inaccurate information can lead to security risks:

- **Factually Incorrect Information:** An LLM provides inaccurate information as a response, causing misinformation.
- **Nonsensical Outputs:** LLM produces logically incoherent or nonsensical text that, while grammatically correct, doesn't make sense.
- **Source Conflation:** LLM melds information from varied sources, creating misleading content.
- **Insecure Code Generation:** LLM suggests insecure or faulty code, leading to vulnerabilities when incorporated into a software system.
- **Inadequate Risk Communication:** Failure of tech companies to appropriately communicate the inherent risks of using LLMs to end users, leading to potential harmful consequences.

How to Prevent

- **Continuous Monitoring & Self-consistency/voting:** Regularly monitor and review the LLM outputs. Use self-consistency or voting techniques to filter out inconsistent text. Comparing multiple model responses for a single prompt can better judge the quality and consistency of output.

- **Fact Checking & External Knowledge Bases:** Cross-check the LLM output with trusted external sources. This additional layer of validation can help ensure the information provided by the model is accurate and reliable.

- **Model Tuning & Chain of Thought Prompting:** Enhance the model with fine-tuning or embeddings to improve output quality. Generic pre-trained models are more likely to produce inaccurate information compared to tuned models in a particular domain. Techniques such as prompt engineering, parameter efficient tuning (PET), full model tuning, and chain of thought prompting can be employed for this purpose.

- **Set Up Validation Mechanisms & Correctness Probabilities:** Implement automatic validation mechanisms that can cross-verify the generated output against known facts or data. This can provide an additional layer of security and mitigate the risks associated with hallucinations.

- **Task Decomposition & Agents:** Break down complex tasks into manageable subtasks and assign them to different agents. This not only helps in managing complexity, but it also reduces the chances of hallucinations as each agent can be held accountable for a smaller task.

- **Improve Risk Communication:** Clearly communicate the risks and limitations associated with using LLMs. This includes potential for information inaccuracies, and other risks. Effective risk communication can prepare users for potential issues and help them make informed decisions.

- **Defensive API and User Interface Design:** Build APIs and user interfaces that encourage responsible and safe use of LLMs. This can involve measures such as content filters, user warnings about potential inaccuracies, and clear labeling of AI-generated content.

- **Security Measures in Development Environments:** When using LLMs in development environments, establish secure coding practices and guidelines to prevent the integration of possible vulnerabilities.

Example Attack Scenario

- **AI-Generated News Disinformation:** A news organization heavily uses an AI model to generate news articles. A malicious actor exploits this over-reliance, feeding the AI misleading information, causing the spread of disinformation. The AI unintentionally plagiarizes content, leading to copyright issues and decreased trust in the organization.

- **AI-Assisted Coding Vulnerabilities:** A software development team utilizes an AI system like Codex to expedite the coding process. Over-reliance on the AI's suggestions introduces security vulnerabilities into the application due to insecure default settings or recommendations inconsistent with secure coding practices.

- **Package Hallucination:** A software development firm uses an LLM to assist developers. The LLM suggests a non-existent code library or package, and a developer, trusting the AI, unknowingly integrates a malicious package into the firm's software. This highlights the importance of cross-checking AI suggestions, especially when involving third-party code or libraries.

Reference Links

- **Understanding LLM Hallucinations:** https://towardsdatascience.com/llm-hallucinations- ec831dcd7786
- **How Should Companies Communicate the Risks of Large Language Models to Users?** https:// techpolicy.press/how-should-companies-communicate-the-risks-of-large-language-models-to-users/
- **A news site used AI to write articles. It was a journalistic disaster:** https:// www.washingtonpost.com/media/2023/01/17/cnet-ai-articles-journalism-corrections/
- **AI Hallucinations: Package Risk:** https://vulcan.io/blog/ai-hallucinations-package-risk
- **How to Reduce the Hallucinations from Large Language Models:** https:// thenewstack.io/how-to-reduce-the-hallucinations-from-large-language-models/
- **Practical Steps to Reduce Hallucination:** https://newsletter.victordibia.com/p/ practical-steps-to-reduce-hallucination

LLM10: Model Theft

First Published: July 1st, 2023

LLM Model Theft refers to the unauthorized access and exfiltration of Language Model models (LLMs) by malicious actors or APT's. This arises when the proprietary LLM models which are valuable intellectual property, are compromised, physically stolen, copied or weights and parameters are extracted to create a functional equivalent. The impact of LLM model theft can include economic losses, erosion of competitive advantage, unauthorized usage of the model, or unauthorized access to sensitive information contained within the model.

The impact of LLM model theft ranges in terms of impact and risk, but at a high-level includes (but not limited to):

- Economic, financial and brand reputation loss, erosion of competitive advantage and unauthorized usage of the model.
- Use of a stolen model, as a shadow model can be used to stage adversarial attacks, including unauthorized access to sensitive information contained within the model or experiment undetected with adversarial inputs to further stage advanced prompt injections.

Common Examples of Vulnerability

- **Example 1:** A skilled attacker exploits a vulnerability in a company's infrastructure to gain unauthorized access to their LLM model repository. The attacker proceeds to download valuable proprietary LLM models and uses them to launch a competing language processing service or extract sensitive information, causing significant financial harm to the original company.
- **Example 2:** An insider threat scenario where a disgruntled employee leaks model or related artifacts. The leaked model increases knowledge for attackers to peform gray box adversarial attacks.
- **Example 3:** An attacker compromises the server with LLM model due to misconfiguration in their network or application security settings.

- **Example 4:** An attacker operates a shared GPU service, offering cheap hosting or access to GPU resources for running Language Model models (LLMs). In this scenario, unsuspecting users utilize the shared GPU service to execute their LLM models due to cost-effectiveness or limited hardware availability. The attacker easily gains unauthorized access to the users' LLM models and then copies them to their controlled server, thereby compromising the proprietary LLM models.

- **Example 5:** An attacker queries the model API or via prompt injection using carefully selected inputs and collects sufficient number of outputs to create a shadow model.

- **Example 6:** A malicious attacker is able to bypass input filtering techniques of the LLM to perform a side-channel attack and ultimately harvest retrieve model weights and architecture information to a remote controlled resource.

- **Example 7:** The attack vector for model extraction involves querying the LLM with a large number of prompts on a particular topic. The outputs from the LLM can then be used to fine-tune another model. However, there are a few things to note about this attack:

    - The attacker must generate a large number of targeted prompts. If the prompts are not specific enough, the outputs from the LLM will be useless.

    - The outputs from LLMs can sometimes contain hallucinated answers. This means that the attacker may not be able to extract the entire model, as some of the outputs may be nonsensical.

        - Therefore, it is not possible to replicate an LLM 100% through model extraction. However, the attacker will be able to replicate a partial model.

- **Example 8:** The is an attack vector for functional model replication and involves using the target model via prompts to generate synthetic training data (an approach called self-instruct) to then use it and fine-tune another foundational model to produce a functional equivalent. This bypasses the limitations of traditional query-based extraction used in Example 7 and has been successfully used in research of using an LLM to train another LLM. Although in the context of this research, model replication is not an attack, the approach could be used by an attacker to replicate a proprietary model with a public API.

How to Prevent

- Implement strong access controls (RBAC, rule of least privilege for example) and strong authentication mechanisms to limit unauthorized access to LLM model repositories and training environments.

    - This is particularly true for the first three common examples which could cause this vulnerability due to insider threats, or misconfiguration and|or weak security controls about the infrastructure that houses LLM models, weights and architecture in which a malicious actor could infiltrate from insider or outside the environment.

    - Supplier management tracking, verification and dependency vulnerabilities are important focus topics to prevent exploits of supply-chain attacks.

- Restrict the LLM's access to network resources, internal services, and APIs.

    - This is particularly true for all common examples as it covers insider risk and threats, but also ultimately controls what the LLM application "has access to" and thus could be a mechanism or prevention step to prevent side-channel attacks.

- Regularly monitor and audit access logs and activities related to LLM model repositories to detect and respond to any suspicious or unauthorized behavior promptly.

- Automate MLOps deployment with governance and tracking and approval workflows to tighten access and deployment controls within the infrastructure.

- Implement controls and mitigation strategies relating to Prompt Injection (#1 entry of the OWASP Top 10 for Large Language Model Applications project) to mitigate and|or reduce risk of prompt injection techniques causing side-channel attacks.

- Rate Limiting of API calls where applicable and|or filters to reduce risk of data exfiltration from the LLM applications, or implement techniques to detect (I.E DLP or other methods) exfiltration from other monitoring systems.

- Implement adversarial robustness training to help detect extraction queries and tighten physical security measures.

Example Attack Scenario

- **Scenario #1:** A skilled attacker exploits a vulnerability in a company's infrastructure to gain unauthorized access to their LLM model repository. The attacker proceeds to download valuable LLM models and uses them to launch a competing language processing service or extract sensitive information, causing significant financial harm to the original company.

- **Scenario #2:** A disgruntled employee leaks model or related artifacts. The public exposure of this scenario increases knowledge to attackers for gray box adversarial attacks or alternatively directly steal the available property.

- **Scenario #3:** An attacker operates a shared GPU service, offering cheap hosting or access to GPU resources for running Language Model models (LLMs). In this scenario, unsuspecting users utilize the shared GPU service to execute their LLM models due to cost-effectiveness or limited hardware availability. The attacker easily gains unauthorized access to the users' LLM models and then copies them to their controlled server, thereby compromising the proprietary LLM models.

- **Scenario #4:** An attacker queries the API with carefully selected inputs and collects sufficient number of outputs to create a shadow model.

- **Scenario #5:** A compromised employee of the hosting platform is manipulated or coerced by attackers to perform a side channel attack and retrieve model information.

- **Scenario #6:** A malicious attacker is able to bypass input filtering techniques of the LLM to perform a side-channel attack and ultimately harvest retrieve model information to a remote controlled resource.

Reference Links

- **Meta's powerful AI language model has leaked online:** https://www.theverge.com/2023/3/8/23629362/meta-ai-language-model-llama-leak-online-misuse
- **Runaway LLaMA | How Meta's LLaMA NLP model leaked:** https://www.deeplearning.ai/the-batch/ how-metas-llama-nlp-model-leaked/
- **I Know What You See:** https://arxiv.org/pdf/1803.05847.pdf
- **D-DAE: Defense-Penetrating Model Extraction Attacks:** https://www.computer.org/csdl/proceedings-article/sp/2023/933600a432/1He7YbsiH4c
- **A Comprehensive Defense Framework Against Model Extraction Attacks:** https://ieeexplore.ieee.org/document/10080996
- **Alpaca: A Strong, Replicable Instruction-Following Model:** https://crfm.stanford.edu/2023/03/13/alpaca.html
- **Orca: Progressive Learning from Complex Explanation Traces of GPT-4:** https://arxiv.org/pdf/2306.02707.pdf

Core Team & Contributors

Core Team Members are listed in Blue

Adam Swanda		**Emanuel Valente**	iFood	**Lior Drihem**	
Adesh Gairola	AWS	**Emmanuel Guilherme Junior**	McMaster University	**Manjesh S**	HackerOne
Ads Dawson	Cohere	**Eugene Neelou**		**Mike Finch**	HackerOne
Adrian Culley	Trellix	**Eugene Tawiah**	Complex Technologies	**Mike Jang**	Forescout
Aleksei Ryzhkov	EPAM	**Gaurav "GP" Pal**	stackArmor	**Nathan Hamiel**	Kudelski Security
Alexander Zai		**Gavin Klondike**	AI Village	**Nipun Gupta**	Bearer
Aliaksei Bialko	EPAM	**Golan Yosef**	Pynt	**Nir Paz**	
Amay Trivedi		**Guillaume Ehinger**	Google	**Otto Sulin**	Nordic Venture Family
Ananda Krishna	Astra Security	**Idan Hen**	Microsoft	**Parveen Yadav**	HackerOne
Andrea Succi		**Itamar Golan**		**Patrick Biyaga**	Thenavigo
Andrew Amaro	Klavan Security Group	**James Rabe**	IriusRisk	**Priyadharshini Parthasarathy**	Coalfire
Andy Dyrcz	Linkfire	**Jason Axley**	AWS	**Rachit Sood**	
Andy Smith		**Jason Haddix**	BuddoBot	**Rahul Zhade**	GitHub
Ashish Rajan	Cloud Security Podcast	**Jason Ross**	Salesforce	**Reza Rashidi**	HADESS
Autumn Moulder		**Jeff Williams**	Contrast Security	**Rich Harang**	AI Village
Bajram Hoxha	Databook	**Johann Rehberger**		**Ross Moore**	
Bilal Siddiqui	Trustwave	**John Sotiropoulos**	Kainos	**Santosh Kumar**	Cisco
Brian Pendleton	AVID	**Jorge Pinto**		**Sarah Thornton**	Red Hat
Brodie McRae	AWS	**Joshua Nussbaum**		**Stefano Amorelli**	
Cassio Goldschmidt	ServiceTitan	**Kai Greshake**		**Steve Wilson**	Contrast Security
Dan Frommer		**Ken Arora**	F5	**Talesh Seeparsan**	Bit79
Dan Klein	Accenture	**Ken Huang**	DistributedApps.ai	**Vandana Verma Sehgal**	Snyk
David Rowe		**Kelvin Low**	aigos	**Vinay Vishwanatha**	Sprinklr
David Taylor		**Larry Carson**		**Vishwas Manral**	Precize
Dotan Nahum	Check Point	**Leon Derczynski**	U of W, IT U of Copenhagen	**Vladimir Fedotov**	EPAM
Dr. Matteo Große-Kampmann	AWARE7	**Leonardo Shikida**	IBM	**Will Chilcutt**	Yahoo