OWASP Top 10 for LLM

VERSION 1.0

Published: *August 1, 2023*

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

Introduction

The frenzy of interest of Large Language Models (LLMs) following of mass-market pre-trained chatbots in late 2022 has been remarkable. Businesses, eager to harness the potential of LLMs, are rapidly integrating them into their operations and client facing offerings. Yet, the breakneck speed at which LLMs are being adopted has outpaced the establishment of comprehensive security protocols, leaving many applications vulnerable to high-risk issues.

The absence of a unified resource addressing these security concerns in LLMs was evident. Developers, unfamiliar with the specific risks associated with LLMs, were left scattered resources and OWASP's mission seemed a perfect fit to help drive safer adoption of this technology.

Who is it for?

Our primary audience is developers, data scientists and security experts tasked with designing and building applications and plug-ins leveraging LLM technologies. We aim to provide practical, actionable, and concise security guidance to help these professionals navigate the complex and evolving terrain of LLM security.

The Making of the List

The creation of the OWASP Top 10 for LLMs list was a major undertaking, built on the collective expertise of an international team of nearly 500 experts, with over 125 active contributors. Our contributors come from diverse backgrounds, including AI companies, security companies, ISVs, cloud hyperscalers, hardware providers and academia.

Over the course of a month, we brainstormed and proposed potential vulnerabilities, with team members writing up 43 distinct threats. Through multiple rounds of voting, we refined these proposals down to a concise list of the ten most critical vulnerabilities. Each vulnerability was then further scrutinized and refined by dedicated sub-teams and subjected to public review, ensuring the most comprehensive and actionable final list.

Each of these vulnerabilities, along with common examples, prevention tips, attack scenarios, and references, was further scrutinized and refined by dedicated sub-teams and subjected to public review, ensuring the most comprehensive and actionable final list.

Relating to other OWASP Top 10 Lists

While our list shares DNA with vulnerability types found in other OWASP Top 10 lists, we do not simply reiterate these vulnerabilities. Instead, we delve into the unique implications these vulnerabilities have when encountered in applications utilizing LLMs.

Our goal is to bridge the divide between general application security principles and the specific challenges posed by LLMs. This includes exploring how conventional vulnerabilities may pose different risks or might be exploited in novel ways within LLMs, as well as how traditional remediation strategies need to be adapted for applications utilizing LLMs.

The Future

This first version of the list will not be our last. We expect to update it on a periodic basis to keep pace with the state of the industry. We will be working with the broader community to push the state of the art, and creating more educational materials for a range of uses. We also seek to collaborate with standards bodies and governments on AI security topics. We welcome you to join our group and contribute.

Steve Wilson

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

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

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 can 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. This lack of application control makes them easier to exploit and can result in 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

Prompt Injection Vulnerability occurs when an attacker manipulates a large language model (LLM) through crafted inputs, causing the LLM to unknowingly execute the attacker's intentions. This can be done directly by "jailbreaking" the system prompt or indirectly through manipulated external inputs, potentially leading to data exfiltration, social engineering, and other issues.

- **Direct Prompt Injections**, also known as "jailbreaking", occur when a malicious user overwrites or reveals the underlying systemprompt. This may allow attackers to exploit backend systems by interacting with insecure functions and data stores accessible through the LLM.

- **Indirect Prompt Injections** occur when an LLM accepts input from external sources that can be controlled by an attacker, such as websites or files. The attacker may embed a prompt injection in the external content hijacking 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. Additionally, indirect prompt injections do not need to be human-visible/readible, as long as the text is parsed by the LLM.

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 advanced attacks, the LLM could be manipulated to mimic a harmful persona or interact with plugins in the user's setting. This could result in leaking sensitive data, unauthorized plugin use, or social engineering. In such cases, the compromised LLM aids the attacker, surpassing standard safeguards and keeping the user unaware of the intrusion. In these instances, the compromised LLM effectively acts as an agent for the attacker, furthering their objectives without triggering usual safeguards or alerting the end user to the intrusion.

Common Examples of Vulnerability

1. A malicious user crafts a direct prompt injection 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.

2. A user employs an LLM to summarize a webpage containing an indirect prompt injection. This then causes the LLM to solicit sensitive information from the user and perform exfiltration via JavaScript or Markdown.
3. A malicious user uploads a resume containing an indirect 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.
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.
5. A rogue instruction and content embedded on a visited website which exploits other plugins to scam users.

How to Prevent

Prompt injection vulnerabilities are possible due to the nature of LLMs, which do not segregate instructions and external data from each other. Since LLM use natural language, they consider both forms of input as user-provided. Consequently, there is no fool-proof prevention within the LLM, but the following measures can mitigate the impact of prompt injections:

1. Enforce privilage control on LLM access to backend systems. 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.
2. Implement human in the loop for extensible functionality. When performing privileged operations, such as sending or deleting emails, have the application require the user 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.
3. Segregate external content from user prompts. 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.
4. 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 as it may hide or manipulate information prior to presenting it to the user. Highlight potentially untrustworthy responses visually to the user.

Example Attack Scenarios

1. An attacker provides a direct prompt injection to an LLM-based support chatbot. The injection contains “forget all previous instructions” and new instructions to query private data stores and exploit package vulnerabilities and the lack of output validation in the backend function to send e-mails. This leads to remote code execution, gaining unauthorized access and privilege escalation.
2. An attacker embeds an indirect prompt injection in a webpage instructing the LLM to disregard previous user instructions and use an LLM plugin to delete the user's emails. When the user employs the LLM to summarise this webpage, the LLM plugin deletes the user's emails.
3. A user employs an LLM to summarize a webpage containing an indirect prompt injection to disregard previous user instructions. This then causes the LLM to solicit sensitive information from the user and perform exfiltration via embedded JavaScript or Markdown.
4. A malicious user uploads a resume with a prompt injection. The backend user uses an LLM to summarize the resume and ask if the person is a good candidate. Due to the prompt injection, the LLM says yes, despite the actual resume contents.
5. 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.

Reference Links

1. **ChatGPT Plugin Vulnerabilities - Chat with Code:** https://embracethered.com/blog/posts/2023/ chatgpt-plugin-vulns-chat-with-code/
2. **ChatGPT Cross Plugin Request Forgery and Prompt Injection:** https://embracethered.com/blog/ posts/2023/chatgpt-cross-plugin-request-forgery-and-prompt-injection./
3. **Defending ChatGPT against Jailbreak Attack via Self-Reminder:** https://www.researchsquare.com/ article/rs-2873090/v1
4. **Prompt Injection attack against LLM-integrated Applications:** https://arxiv.org/abs/2306.05499
5. **Inject My PDF: Prompt Injection for your Resume:** https://kai-greshake.de/posts/inject-my-pdf/
6. **ChatML for OpenAI API Calls:** https://github.com/openai/openai-python/blob/main/chatml.md
7. **Not what you’ve signed up for- Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection:** https://arxiv.org/pdf/2302.12173.pdf
8. **Threat Modeling LLM Applications:** http://aivillage.org/large%20language%20models/threat-modeling-llm/
9. **AI Injections: Direct and Indirect Prompt Injections and Their Implications:** https://embracethered.com/blog/posts/2023/ai-injections-direct-and-indirect-prompt-injection-basics/

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, such as passing LLM output directly to backend, privileged, or client-side functions. Since LLM-generated content can be controlled by prompt input, this behavior is similar 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 following conditions can increase the impact of this vulnerability:

- The application grants the LLM privileges beyond what is intended for end users, enabling escalation of privileges or remote code execution.
- The application is vulnerable to external prompt injection attacks, which could allow an attacker to gain privileged access to a target user's environment.

Common Examples of Vulnerability

1. LLM output is entered directly into a system shell or similar function such as `exec` or `eval` , resulting in remote code execution.
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

1. Treat the model as any other user and apply proper input validation on responses coming from the model to backend functions. Follow the OWASP ASVS (Application Security Verification Standard) guidelines to ensure effective input validation and sanitization.
2. Encode model output back to users to mitigate undesired code execution by JavaScript or Markdown. OWASP ASVS provides detailed guidance on output encoding.

Example Attack Scenarios

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.
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 user's conversation. From there the LLM can encode the sensitive data and send it out to an attacker-controlled server
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.
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 URL parameter. 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 user's browser.

Reference Links

1. **Snyk Vulnerability DB- Arbitrary Code Execution:** https://security.snyk.io/vuln/SNYK-PYTHON- LANGCHAIN-5411357
2. **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./
3. **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
4. **Don't blindly trust LLM responses. Threats to chatbots:** https://embracethered.com/ blog/posts/ 2023/ai-injections-threats-context-matters/
5. **Threat Modeling LLM Applications:** https://aivillage.org/large language models/threat-modeling-llm/
6. **OWASP ASVS - 5 Validation, Sanitization and Encoding:** https://owasp-aasvs4.readthedocs.io/en/latest/V5.html#validation-sanitization-and-encoding

LLM03: Training Data Poisoning

The starting point of any machine learning approach is training data, simply “raw text”. To be highly capable (e.g., have linguistic and world knowledge), this text should span a broad range of domains, genres and languages. A large language model uses deep neural networks to generate outputs based on patterns learned from training data.

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. Poisoned information may be surfaced to users or create other risks like performance degradation, downstream software exploitation and reputational damage. Even if users distrust the problematic AI output, the risks remain, including impaired model capabilities and potential harm 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. Naturally, external data sources present higher risk as the model creators do not have control of the data or a high level of confidence that the content does not contain bias, falsified information or inappropriate content.

Common Examples of Vulnerability

1. A malicious actor, or a competitor brand intentionally creates inaccurate or malicious documents which are targeted at a model’s training data.
   - The victim model trains using falsified information which is reflected in outputs of generative AI prompts to it's consumers.
2. A model is trained using data which has not been verified by its source, origin or content.
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.

Whether a developer, client or general consumer of the LLM, it is important to understand the implications of how this vulnerability could reflect risks within your LLM application when interacting with a non-proprietary LLM.

Example Attack Scenarios

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.
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.
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.
4. The vulnerability Prompt Injection could be an attack vector to this vulnerability if insufficient sanitization and filtering is performed when clients of the LLM application input is used to train the model. I.E, if malicious or falsified data is input to the model from a client as part of a prompt injection technique, this could inherently be portrayed into the model data.

How to Prevent

1. 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.
2. Verify the correct legitimacy of targeted data sources and data contained obtained during both the training and fine-tuning stages.
3. 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.
4. Ensure sufficient sandboxing is present to prevent the model from scraping unintended data sources which could hinder the machine learning output.
5. 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.
6. Adversarial robustness techniques such as federated learning and constraints to minimize the effect of outliers or adversarial training to be vigorous against worst-case perturbations of the training data.
   a. An "MLSecOps" approach could be to include adversarial robustness to the training lifecycle with the auto poisoning technique.
   b. 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.

7. 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.
    a. Monitoring and alerting on number of skewed responses exceeding a threshold.
    b. Use of a human loop to review responses and auditing.
    c. Implement dedicated LLM's to benchmark against undesired consequences and train other LLM's using reinforcement learning techniques.
    d. Perform LLM-based red team exercises or LLM vulnerability scanning into the testing phases of the LLM's lifecycle.

Reference Links

1. **Stanford Research Paper:** https://stanford-cs324.github.io/winter2022/lectures/data/
2. **How data poisoning attacks corrupt machine learning models:** https://www.csoonline.com/article/3613932/how-data-poisoning-attacks-corrupt-machine-learning-models.html
3. **MITRE ATLAS (framework) Tay Poisoning:** https://atlas.mitre.org/studies/AML.CS0009/
4. **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/
5. **Inject My PDF: Prompt Injection for your Resume:** https://kai-greshake.de/posts/inject-my-pdf/
6. **Backdoor Attacks on Language Models:** https://towardsdatascience.com/backdoor-attacks-on-language-models-can-we-trust-our-models-weights-73108f9dcb1f
7. **Poisoning Language Models During Instruction:** https://arxiv.org/abs/2305.00944
8. **FedMLSecurity:** https://arxiv.org/abs/2306.04959
9. **The poisoning of ChatGPT:** https://softwarecrisis.dev/letters/the-poisoning-of-chatgpt/

LLM04: Model Denial of Service

An attacker interacts with a LLM 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

1. Posing queries that lead to recurring resource usage through high-volume generation of tasks in a queue, e.g. with LangChain or AutoGPT.
2. Sending queries that are unusually resource-consuming, perhaps because they use unusual orthography or sequences.
3. 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.
4. Repetitive long inputs: The attacker repeatedly sends long inputs to the LLM, each exceeding the context window.
5. Recursive context expansion: The attacker constructs input that triggers recursive context expansion, forcing the LLM to repeatedly expand and process the context window.
6. 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 aims to exploit any inefficiencies in processing variable-length inputs, straining the LLM and potentially causing it to become unresponsive.

Example Attack Scenarios

1. 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.
2. 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.
3. An 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.
4. An 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.
5. An 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.
6. An 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

1. Implement input validation and sanitization to ensure user input adheres to defined limits and filters out any malicious content.
2. Cap resource use per request or step, so that requests involving complex parts execute more slowly.
3. Enforce API rate limits to restrict the number of requests an individual user or IP address can make within a specific timeframe.
4. Limit the number of queued actions and the number of total actions in a system reacting to LLM responses.
5. Continuously monitor the resource utilization of the LLM to identify abnormal spikes or patterns that may indicate a DoS attack.

6. Set strict input limits based on the LLM's context window to prevent overload and resource exhaustion.
7. Promote awareness among developers about potential DoS vulnerabilities in LLMs and provide guidelines for secure LLM implementation.

Reference Links

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

LLM05: Supply Chain Vulnerabilities

The supply chain in LLMs can be vulnerable, impacting the integrity of training data, ML models, and deployment platforms. These vulnerabilities can lead to biased outcomes, security breaches, or even complete system failures. Traditionally, vulnerabilities are focused on software components, but Machine Learning extends this with the pre-trained models and training data supplied by third parties susceptible to tampering and poisoning attacks.

Finally, LLM Plugin extensions can bring their own vulnerabilities. These are described in LLM - Insecure Plugin Design, which covers writing LLM Plugins and provides information useful to evaluate third-party plugins.

Common Examples of Vulnerability

1. Traditional third-party package vulnerabilities, including outdated or deprecated components.
2. Using a vulnerable pre-trained model for fine-tuning.
3. Use of poisoned crowd-sourced data for training.
4. Using outdated or deprecated models that are no longer maintained leads to security issues.
5. Unclear T&Cs 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

1. 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.
2. 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.

3. Understand and apply the mitigations found in the OWASP Top Ten's A06:2021 – Vulnerable and Outdated Components. This includes vulnerability scanning, management, and patching components. For development environments with access to sensitive data, apply these controls in those environments, too.
4. Maintain an up-to-date inventory of components using a Software Bill of Materials (SBOM) to ensure you have an up-to-date, accurate, and signed inventory preventing tampering with deployed packages. SBOMs can be used to detect and alert for new, zero-date vulnerabilities quickly.
5. At the time of writing, SBOMs do not cover models, their artefacts, 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.
6. You should also use model and code signing when using external models and suppliers.
7. Anomaly detection and adversarial robustness tests on supplied models and data can help detect tampering and poisoning as discussed in 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.
8. Implement sufficient monitoring to cover component and environment vulnerabilities scanning, use of unauthorised plugins, and out-of-date components, including the model and its artefacts.
9. Implement a patching policy to mitigate vulnerable or outdated components. Ensure that the application relies on a maintained version of APIs and the underlying model.
10. Regularly review and audit supplier Security and Access, ensuring no changes in their security posture or T&Cs.

Example Attack Scenarios

1. An attacker exploits a vulnerable Python library to compromise a system. This happened in the first Open AI data breach.
2. An attacker provides an LLM plugin to search for flights which generates fake links that lead to scamming plugin users.
3. An attacker exploits the PyPi package registry to trick model developers into downloading a compromised package and exfiltrating data or escalating privilege in a model development environment. This was an actual attack.
4. An attacker poisons a publicly available pre-trained model specialising in economic analysis and social research to create a backdoor which generates misinformation and fake news. They deploy it on a model marketplace (e.g. HuggingFace) for victims to use.
5. An attacker poisons publicly available data set to help create a backdoor when fine-tuning models. The backdoor subtly favours certain companies in different markets.
6. A compromised employee of a supplier (outsourcing developer, hosting company, etc) exfiltrates data, model, or code stealing IP.

7. An LLM operator changes its T&Cs and Privacy Policy so that it requires an explicit opt-out from using application data for model training, leading to memorization of sensitive data.

Reference Links

1. **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/
2. **Open AI's Plugin review process:** https://platform.openai.com/docs/plugins/review
3. **Compromised PyTorch-nightly dependency chain:** https://pytorch.org/blog/compromised-nightly-dependency/
4. **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/
5. **Army looking at the possibility of 'AI BOMs:** https://defensescoop.com/2023/05/25/army-looking-at-the-possibility-of-ai-boms-bill-of-materials/
6. **Failure Modes in Machine Learning:** https://learn.microsoft.com/en-us/security/engineering/failure-modes-in-machine-learning
7. **ML Supply Chain Compromise:** https://atlas.mitre.org/techniques/AML.T0010/
8. **Transferability in Machine Learning: from Phenomena to Black-Box Attacks using Adversarial Samples:** https://arxiv.org/pdf/1605.07277.pdf
9. **BadNets: Identifying Vulnerabilities in the Machine Learning Model Supply Chain:** https://arxiv.org/abs/1708.06733
10. **VirusTotal Poisoning:** https://atlas.mitre.org/studies/AML.CS0002

LLM06: Sensitive Information Disclosure

LLM applications have the potential to reveal sensitive information, proprietary algorithms, or other confidential details through their output. This can result in unauthorized access to sensitive data, intellectual property, privacy violations, and other security breaches. It is important for consumers of LLM applications to be aware of how to safely interact with LLMs and identify the risks associated with unintentionally inputting sensitive data that it may be returned by the LLM in output elsewhere.

To mitigate this risk, LLM applications should perform adequate data sanitization to prevent user data from entering the training model data. LLM application owners should also have appropriate Terms of Use policies available to make consumers aware of how their data is processed and the ability to opt-out of having their data included in the training model.

The consumer-LLM application interaction forms a two-way trust boundary, where we cannot inherently trust the client->LLM input or the LLM->client output. It is important to note that this vulnerability assumes that certain pre-requisites are out of scope, such as threat modeling exercises, securing infrastructure, and adequate sandboxing. Adding restrictions within the system prompt around the types of data the LLM should return can provide some mitigation against sensitive information disclosure, but the unpredictable nature of LLMs means such restrictions may not always be honoured and could be circumvented via prompt injection or other vectors.

Common Examples of Vulnerability

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

How to Prevent

1. Integrate adequate data sanitization and scrubbing techniques to prevent user data from entering the training model data.
2. Implement robust input validation and sanitization methods to identify and filter out potential malicious inputs to prevent the model from being poisoned.
3. When enriching the model with data and if fine-tuning a model: (I.E, data fed into the model before or during deployment)

a. 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.
b. Access to external data sources (orchestration of data at runtime) should be limited.
c. Apply strict access control methods to external data sources and a rigorous approach to maintaining a secure supply chain.

Example Attack Scenarios

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.
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 (PII) about other users of the application.
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

1. **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
2. **Lessons learned from ChatGPT's Samsung leak:** https://cybernews.com/security/chatgpt-samsung-leak-explained-lessons/
3. **Cohere - Terms Of Use:** https://cohere.com/terms-of-use
4. **AI Village- Threat Modeling Example:** https://aivillage.org/large%20language%20models/threat-modeling-llm/
5. **OWASP AI Security and Privacy Guide:** https://owasp.org/www-project-ai-security-and-privacy-guide/

LLM07: Insecure Plugin Design

LLM plugins are extensions that, when enabled, are called automatically by the model during user interactions. They are driven by the model, and there is no application control over the execution. Furthermore, to deal with context-size limitations, plugins are likely to implement free-text inputs from the model with no validation or type checking. This allows a potential attacker to construct a malicious request to the plugin, which could result in a wide range of undesired behaviors, up to and including remote code execution.

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

This item focuses on the creation of LLM plugins rather than using third-party plugins, which is covered by LLM-Supply-Chain-Vulnerabilities.

Common Examples of Vulnerability

1. A plugin accepts all parameters in a single text field instead of distinct input parameters.
2. A plugin accepts configuration strings, instead of parameters, that can override entire configuration settings.
3. A plugin accepts raw SQL or programming statements instead of parameters.
4. Authentication is performed without explicit authorization to a particular plugin.
5. A plugin treats all LLM content as being created entirely by the user and performs any requested actions without requiring additional authorization.

How to Prevent

1. Plugins should enforce strict parameterized input wherever possible and include type and range checks on inputs. When this is not possible, 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.
2. Plugin developers should apply OWASP's recommendations in ASVS (Application Security Verification Standard) to ensure effective input validation and sanitization.

3. Plugins should be inspected and tested thoroughly to ensure adequate validation. Use Static Application Security Testing (SAST) scans as well as Dynamic and Interactive application testing (DAST, IAST) in development pipelines.
4. 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.
5. Plugins should use appropriate authentication identities, such as OAuth2, to apply effective authorization and access control. Additionally, API Keys should be used to provide context for custom authorization decisions which reflect the plugin route rather than the default interactive user.
6. Require manual user authorization and confirmation of any action taken by sensitive plugins.
7. Plugins are, typically, REST APIs, so developers should apply the recommendations found in OWASP Top 10 API Security Risks – 2023 to minimize generic vulnerabilities

Example Attack Scenarios

1. A plugin accepts a base URL and instructs the LLM to combine the URL with a query to obtain weather forecasts which are included in handling the user request. A malicious user can craft a request such that the URL points to a domain they control, which allows them to inject their own content into the LLM system via their domain.
2. A plugin accepts a free-form input into a single field that it does not validate. An attacker supplies carefully crafted payloads to perform reconnaissance from error messages. It then exploits known third-party vulnerabilities to execute code and perform data exfiltration or privilege escalation.
3. A plugin used to retrieve embeddings from a vector store accepts configuration parameters as a connection string without any validation. This allows an attacker to experiment and access other vector stores by changing names or host parameters and exfiltrate embeddings they should not have access to.
4. A plugin accepts SQL WHERE clauses as advanced filters, which are then appended to the filtering SQL. This allows an attacker to stage a SQL attack.
5. 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.

Reference Links

1. **OpenAI ChatGPT Plugins:** https://platform.openai.com/docs/plugins/introduction
2. **OpenAI ChatGPT Plugins - Plugin Flow:** https://platform.openai.com/docs/plugins/introduction/plugin-flow
3. **OpenAI ChatGPT Plugins - Authentication:** https://platform.openai.com/docs/plugins/authentication/service-level
4. **OpenAI Semantic Search Plugin Sample:** https://github.com/openai/chatgpt-retrieval-plugin
5. **Plugin Vulnerabilities: Visit a Website and Have Your Source Code Stolen:** https://embracethered.com/blog/posts/2023/chatgpt-plugin-vulns-chat-with-code/
6. **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./
7. **OWASP ASVS - 5 Validation, Sanitization and Encoding:** https://owasp-aasvs4.readthedocs.io/en/latest/V5.html#validation-sanitization-and-encoding
8. **OWASP ASVS 4.1 General Access Control Design:** https://owasp-aasvs4.readthedocs.io/en/latest/V4.1.html#general-access-control-design
9. **OWASP Top 10 API Security Risks – 2023:** https://owasp.org/API-Security/editions/2023/en/0x11-t10/

LLM08: Excessive Agency

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. The decision over which functions to invoke may also be delegated to an LLM 'agent' to dynamically determine based on input prompt or LLM output.

Excessive Agency is the vulnerability that enables damaging actions to be performed in response to unexpected/ambiguous 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 one or more of: excessive functionality, excessive permissions or excessive autonomy.

Excessive Agency can lead to a broad range of impacts across the confidentiality, integrity and availability spectrum, and is dependent on which systems an LLM-based app is able to interact with.

Common Examples of Vulnerability

1. Excessive Functionality: An LLM agent has access to plugins which include functions that are not needed for the intended operation of the system. 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. Alternatively, a plugin may have been trialled during a development phase and dropped in favour of a better alternative, but the original plugin remains available to the LLM agent.
2. Excessive Functionality: 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.
3. Excessive Permissions: 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.
4. Excessive 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 privileged account that has access to all users' files.

5. Excessive Autonomy: An LLM-based application or plugin fails to independently verify and approve high-impact actions. E.g., a plugin that allows a user's documents to be deleted performs deletions without any confirmation from the user.

How to Prevent

The following actions can prevent Excessive Agency:

1. 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.
2. 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.
3. 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.
4. 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.
5. 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.
6. 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.
7. Implement authorization in downstream systems rather than relying on an LLM to decide if an action is allowed or not. When implementing tools/plugins enforce the complete mediation principle so that all requests made to downstream systems via the plugins/tools are validated against security policies.

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

1. Log and monitor the activity of LLM plugins/tools and downstream systems to identify where undesirable actions are taking place, and respond accordingly.
2. 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 summarise 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: (a) eliminating excessive functionality by using a plugin that only offered mail-reading capabilities, (b) eliminating excessive permissions by authenticating to the user's email service via an OAuth session with a read-only scope, and/or (c) 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

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

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

1. LLM provides inaccurate information as a response, causing misinformation.
2. LLM produces logically incoherent or nonsensical text that, while grammatically correct, doesn't make sense.
3. LLM melds information from varied sources, creating misleading content.
4. LLM suggests insecure or faulty code, leading to vulnerabilities when incorporated into a software system.
5. Failure of provider to appropriately communicate the inherent risks to end users, leading to potential harmful consequences.

How to Prevent

1. 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.
2. 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.
3. 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 partiular domain. Techniques such as prompt engineering, parameter efficient tuning (PET), full model tuning, and chain of thought prompting can be employed for this purpose.

4. 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.
5. 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.
6. 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.
7. 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.
8. When using LLMs in development environments, establish secure coding practices and guidelines to prevent the integration of possible vulnerabilities.

Example Attack Scenario

1. 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.
2. 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.
3. 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

1. **Understanding LLM Hallucinations:** https://towardsdatascience.com/llm-hallucinations- ec831dcd7786
2. **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/
3. **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/
4. **AI Hallucinations: Package Risk:** https://vulcan.io/blog/ai-hallucinations-package-risk
5. **How to Reduce the Hallucinations from Large Language Models:** https:// thenewstack.io/how-to-reduce-the-hallucinations-from-large-language-models/
6. **Practical Steps to Reduce Hallucination:** https://newsletter.victordibia.com/p/ practical-steps-to-reduce-hallucination

LLM10: Model Theft

This entry refers to the unauthorized access and exfiltration of LLM models by malicious actors or APTs. This arises when the proprietary LLM models (being 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 and brand reputation loss, erosion of competitive advantage, unauthorized usage of the model or unauthorized access to sensitive information contained within the model.

The theft of LLMs represents a significant security concern as language models become increasingly powerful and prevalent. Organizations and researchers must prioritize robust security measures to protect their LLM models, ensuring the confidentiality and integrity of their intellectual property. Employing a comprehensive security framework that includes access controls, encryption, and continuous monitoring is crucial in mitigating the risks associated with LLM model theft and safeguarding the interests of both individuals and organizations relying on LLM.

Common Examples of Vulnerability

1. An attacker exploits a vulnerability in a company's infrastructure to gain unauthorized access to their LLM model repository via misconfiguration in their network or application security settings.
2. An insider threat scenario where a disgruntled employee leaks model or related artifacts.
3. An attacker queries the model API using carefully crafted inputs and prompt injection techniques to collect a sufficient number of outputs to create a shadow model.
4. A malicious attacker is able to bypass input filtering techniques of the LLM to perform a side-channel attack and ultimately harvest model weights and architecture information to a remote controlled resource.
5. 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 meaning the attacker may not be able to extract the entire model as some of the outputs can be nonsensical.
    - It is not possible to replicate an LLM 100% through model extraction. However, the attacker will be able to replicate a partial model.

6. The attack vector for functional model replication 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 5 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.

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.

How to Prevent

1. Implement strong access controls (E.G., RBAC and rule of least privilege) and strong authentication mechanisms to limit unauthorized access to LLM model repositories and training environments.
   a. This is particularly true for the first three common examples, which could cause this vulnerability due to insider threats, 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.
   b. Supplier management tracking, verification and dependency vulnerabilities are important focus topics to prevent exploits of supply-chain attacks.
2. Restrict the LLM's access to network resources, internal services, and APIs.
   a. 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.
3. Regularly monitor and audit access logs and activities related to LLM model repositories to detect and respond to any suspicious or unauthorized behavior promptly.
4. Automate MLOps deployment with governance and tracking and approval workflows to tighten access and deployment controls within the infrastructure.
5. Implement controls and mitigation strategies to mitigate and|or reduce risk of prompt injection techniques causing side-channel attacks.
6. 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 (E.G., DLP) extraction activity from other monitoring systems.
7. Implement adversarial robustness training to help detect extraction queries and tighten physical security measures.
8. Implement a watermarking framework into the embedding and detection stages of an LLMs lifecyle.

Example Attack Scenario

1. An attacker exploits a vulnerability in a company's infrastructure to gain unauthorized access to their LLM model repository. The attacker proceeds to exfiltrate 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.
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.
3. An attacker queries the API with carefully selected inputs and collects sufficient number of outputs to create a shadow model.
4. A security control failure is present within the supply-chain and leads to data leaks of proprietary model information.
5. A malicious attacker bypasses input filtering techniques and preambles of the LLM to perform a side-channel attack and retrieve model information to a remote controlled resource under their control.

Reference Links

1. **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
2. **Runaway LLaMA | How Meta's LLaMA NLP model leaked:** https://www.deeplearning.ai/the-batch/ how-metas-llama-nlp-model-leaked/
3. **I Know What You See:** https://arxiv.org/pdf/1803.05847.pdf
4. **D-DAE: Defense-Penetrating Model Extraction Attacks:** https://www.computer.org/csdl/proceedings-article/sp/2023/933600a432/1He7YbsiH4c
5. **A Comprehensive Defense Framework Against Model Extraction Attacks:** https://ieeexplore.ieee.org/document/10080996
6. **Alpaca: A Strong, Replicable Instruction-Following Model:** https://crfm.stanford.edu/2023/03/13/alpaca.html
7. **How Watermarking Can Help Mitigate The Potential Risks Of LLMs?**: https://www.kdnuggets.com/2023/03/watermarking-help-mitigate-potential-risks-llms.html

Core Team & Contributors

Core Team Members are listed in ***Blue***

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