JANUARY 21, 2020

Linux Commands Most Used by Attackers

TOXICPTR

SHARE

This article explains which are the **Linux commands most used by attackers** and **how to defend against these attacks**. Although the solutions are based on Linux they work on other Unix-like systems as well.

The commands were obtained from a study by D. Ramsbrock, R. Berthier and M. Cukie from the University of Maryland, and **Kippo SSH logs** from miscellaneous sources.Linux CommandsOnce an attacker compromises a host, he performs tasks that can be classified in stages or states: Check software configuration, install a program, download a file, etc.These 'states' are **related with the commands** below. Some commands retrieve system information, other commands download files from external sources, other commands can change the system configuration, etc.

cat [/etc/* | .bash_history | /proc/cpuinfo]. Concatenate files and print on the standard output.

chmod. Change file mode bits.

cp. Copy files and directories.

curl. Transfer a URL.

export. Bash builtin command to set environment variables.

ftp. Internet file transfer program.
history. GNU History Library
id. Print real and effective user and group IDs
ifconfig. Configure a network interface.
kill. Send a signal to a process.
last. Show listing of last logged in users.
lwp-download. Fetch large files from the web.
mail. Send and receive Internet mail.
mkdir. Make directories
mv. Move (rename) files.
nano. Nano's ANOther editor, an enhanced free Pico clone.
passwd. Change user password.
perl. Practical Extraction and Report Language.
php. PHP Command Line Interface 'CLI'.
pico. Simple text editor in the style of the Alpine Composer.
ps. Report a snapshot of the current processes.
python. An interpreted, interactive, object-oriented programming language.
rm. Remove files or directories.
sshd. OpenSSH SSH daemon.
tar. GNU 'tar' saves many files together into a single tape or disk archive, and can restore individual files from the archive.
uname. Print system information
unzip. List, test and extract compressed files in a ZIP archive
uptime. Tell how long the system has been running.
useradd. Create a new user or update default new user information.
userdel. Delete a user account and related files.
vi. Screen-oriented text editor.
vim. Vi IMproved, a programmers text editor.
wget. The non-interactive network downloader.
whoami. Print effective userid.
w. Show who is logged on and what they are doing.

Countermeasures

How can we protect our system against the execution of sensitive commands? Obviously, we cannot restrict execution to all the commands above. Some of them are **essential** for daily tasks (cd, mv, cp, ps, etc.). Nevertheless, other commands are potentially **harmful** (useradd, userdel, kill).As we will see later, the **command-line editors** are also important (vi, vim, pico, nano). Keep in mind that the combination of other commands can also be used as a primitive editor (id >> victim-info.txt, cat /etc/passwd >> victim-info.txt).But let's focus on the **critical commands**. We can restrict their execution and apply other

security measures to obstruct the attacker's activities.

Restrict Execution with Permissions

The easiest method to limit the execution is to play with **UNIX file permissions**.For example, to disable execution of wget:

$ chgrp root:root /usr/bin/wget$ chmod 0710 /usr/bin/wget

Now, only root or someone in the sudo group can run wget.If we need to restrict execution of wget command to a special user group we can do something like:

$ groupadd spusers$ usermod -aG spusers bob$ usermod -aG spusers alice$ chgrp spusers /usr/bin/wget$ chmod 0710 /usr/bin/wget

Restricted Shell

Another solution is to use **Restricted Shells**. For example, a bash restricted shell provides an **additional layer of security** to bash by disallowing some features, such as:

- Changing directories with the cd builtin.
- Setting or unsetting the values of the SHELL, PATH, ENV, or BASH_ENV variables.
- Specifying command names containing slashes.
- Specifying a filename containing a slash as an argument to the . builtin command.
- Redirecting output using the >, >|, <>, >&, &>, and >> redirection operators.
- Adding or deleting builtin commands with the -f and -d options to the enable builtin.

Unfortunately, **it is not secure**. A determined user can bypass it if he finds out a way to spawn another shell without restrictions. For example, using vi:

$ vi:set shell=/bin/bash:shell

Hardening Restricted Shell We can **fix the limitations** of the restricted shells by controlling the access to certain commands, such as shells and vi editor. But we may end up using some esoteric solution as:

$ chsh --shell /bin/rbash bob$ mkdir -p /usr/local/rbash/bin$ find /bin -exec ln -s {} /usr/local/rbash{} ;$ grep -v "^#" /etc/shells | while read shell; do rm /usr/local/rbash"${shell}"; done$ echo "export PATH=/usr/local/bin:/usr/local/rbash/bin:/usr/bin"

>> /home/bob/.bash_profile$ chmod 750 /usr/bin/{vi,vim*}

Certainly, **this is not the most recommended way to restrict command execution**, but it may work as an easy temporary solution.

Other

Technologies such as SELinux or Virtualization Technologies (KVM, QEMU, VirtualBox, etc.) are **safer solutions** than the above, but their installation and use are beyond the scope of this document. Please, refer to their official documentation.They take more effort than the other approaches but they may worth it. It only **depends on the value** of the system to protect.

Auditing Command Execution

The next step in hardening our system against these attacks is to deploy security auditing. **The Linux Audit system** provides a way to track security-relevant information on the system.First, we install auditd package:

$ apt-get install auditd

The configuration files are stored in /etc/audit. The **rules** are defined in /etc/audit/audit.rules.For example, if we want to monitor the execution of wget and curl add:

-w /usr/bin/wget -p x -k crit_execs-w /usr/bin/ftp -p x -k crit_execs

Then, restart auditd service to reload the rules:

$ service auditd restart

Now we can **review the audited activity** with ausearch:

$ ausearch -k crit_execs -itype=PATH
msg=audit(11/22/2016 00:28:21.709:16) : item=1 name=(null) inode=61 dev=fe:00 mode=file,755 ouid=root ogid=root rdev=00:00type=PATH
msg=audit(11/22/2016 00:28:21.709:16) : item=0 name=/usr/bin/wget inode=8311 dev=fe:03 mode=file,755 ouid=root ogid=root rdev=00:00type=CWD msg=audit(11/22/2016 00:28:21.709:16) : cwd=/home/bobtype=EXECVE msg=audit(11/22/2016 00:28:21.709:16) : argc= (null) a0=wgettype=SYSCALL msg=audit(11/22/2016 00:28:21.709:16) : arch=x86_64 syscall=execve success=yes exit=0 a0=20fb228 a1=20fd668 a2=20f8e08 a3=0 items=2 ppid=6847 pid=6854 auid=bob uid=bob gid=bob euid=bob suid=bob fsuid=bob egid=bob

```
sgid=bob fsgid=bob tty=tty2 ses=5 comm=wget
exe=/usr/bin/wget key=crit_execs----
type=PATH msg=audit(11/22/2016
00:28:25.865:18) : item=1 name=(null) inode=61
dev=fe:00 mode=file,755 ouid=root ogid=root
rdev=00:00type=PATH msg=audit(11/22/2016
00:28:25.865:18) : item=0 name=/usr/bin/wget
inode=8311 dev=fe:03 mode=file,755 ouid=root
ogid=root rdev=00:00type=CWD
msg=audit(11/22/2016 00:28:25.865:18) :
cwd=/home/alicetype=EXECVE
msg=audit(11/22/2016 00:28:25.865:18) : argc=
(null) a0=wgettype=SYSCALL
msg=audit(11/22/2016 00:28:25.865:18) :
arch=x86_64 syscall=execve success=yes exit=0
a0=1125488 a1=11233c8 a2=1121e08 a3=0
items=2 ppid=6857 pid=6862 auid=alice
uid=alice gid=alice euid=alice suid=alice
fsuid=alice egid=alice sgid=alice fsgid=alice
tty=tty2 ses=6 comm=wget exe=/usr/bin/wget
key=crit_execs
```

Or parse it with aureport:

```
$ ausearch -k crit_execs --raw | aureport --
summary --user -iUser Summary
Report==========================total
auid==========================1 bob1
alice
```

Additional Security Measures

Besides to restrict execution and system auditing, there are a few **additional security measures** that can be applied to limit these attacks.

Securing temporary partitions

Sometimes, the attackers download and execute files from **temporary directories**. We can disable the execution by mounting these directories with noexec option.Open /etc/fstab and configure it to mount /tmp with tmpfs:

```
tmpfs /tmp tmpfs
nodev,nosuid,noexec,noatime,size=2G 0 0
```

Another temporary directory used is /var/tmp. The problem is that according to the FHS:

> *The files and directories located in /var/tmp must not be deleted when the system is booted.*

Here we have two options: dedicate one partition for /var/tmp or create a file and mount it as /var/tmp. If we go with the latter, simpler one, we can proceed as follows:

$ dd if=/dev/zero of=/var/tmp.fs bs=1 count=0 seek=4G$ mkfs.ext4 /var/tmp.fs$ mount -o loop,rw,nodev,nosuid,noexec /var/tmp.fs /var/tmp$ chmod 1777 /var/tmp

Then we edit /etc/fstab to make it permanent:

/var/tmp /var/tmp.fs ext4 rw,nodev,nosuid,noexec,noatime 0 0

Drop Outgoing Connections

After breaking into a system, the attacker nearly always opens **an outbound network connection**. Why? Sometimes because they need their precious hacking tools, generally compressed files with network scanners, password crackers, IRC bots, etc. Another reason is because they send data from their victims back to their infrastructure.Not always is possible, but if the system does not need to open outbound connections **we can block them with a firewall**. The following example drops any new connections made from the system. Is a dangerous option that must be applied with caution.

$ iptables -A OUTPUT -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT$ iptables -P OUTPUT DROP

We can also narrow down the scope of the outbound connections. These rules only allow DNS requests and HTTP/S connections from the system:

$ iptables -A OUTPUT -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT$ iptables -A OUTPUT -p udp --dport 53 -m conntrack --ctstate NEW -j ACCEPT$ iptables -A OUTPUT -p tcp --dport 53 -m conntrack --ctstate NEW -j ACCEPT$ iptables -A OUTPUT -p tcp --dport 80 -m conntrack --ctstate NEW -j ACCEPT$ iptables -A OUTPUT -p tcp --dport 443 -m conntrack --ctstate NEW -j ACCEPT$ iptables -P OUTPUT DROP

An ideal scenario would combine the **ports with ranges of trusted IP addresses** for better security.This will be always **better than having unrestricted output traffic**. We just have to adjust the rules to our policies.

Conclusion

We have described some techniques to **restrict command execution on Unix-like systems** and how to detect abnormal activities with security auditing.These measures will not stop all attacks. As security analysts, **we will have security incidents**, sooner or later. We need **to be prepared** and to have a disaster recovery plan.

The Open Worldwide Application Security Project (OWASP) is a community-led organization and has been around for over 20 years and is largely known for its Top 10 web application security risks (check out our course on it). As the use of generative AI and large language models (LLMs) has exploded recently, so too has the risk to privacy and security by these technologies. OWASP, leading the charge for security, has come out with its Top 10 for LLMs and Generative AI Apps this year. In this blog post we'll explore the Top 10 risks and explore examples of each as well as how to prevent these risks.

LLM01: Prompt Injection

Those familiar with the OWASP Top 10 for web applications have seen the injection category before at the top of the list for many years. This is no exception with LLMs and ranks as number one. Prompt Injection can be a critical vulnerability in LLMs where an attacker manipulates the model through crafted inputs, leading it to execute unintended actions. This can result in unauthorized access, data exfiltration, or social engineering. There are two types: Direct Prompt Injection, which involves "jailbreaking" the system by altering or revealing underlying system prompts, giving an attacker access to backend systems or sensitive data, and Indirect Prompt Injection, where external inputs (like files or web content) are used to manipulate the LLM's behavior.

As an example, an attacker might upload a resume containing an indirect prompt injection, instructing an LLM-based hiring tool to favorably evaluate the resume. When an internal user runs the document through the LLM for summarization, the embedded prompt makes the LLM respond positively about the candidate's suitability, regardless of the actual content.

How to prevent prompt injection:

1. Limit LLM Access: Apply the principle of least privilege by restricting the LLM's access to sensitive backend systems and enforcing API

token controls for extended functionalities like plugins.

2. Human Approval for Critical Actions: For high-risk operations, require human validation before executing, ensuring that the LLM's suggestions are not followed blindly.
3. Separate External and User Content: Use frameworks like ChatML for OpenAI API calls to clearly differentiate between user prompts and untrusted external content, reducing the chance of unintentional action from mixed inputs.
4. Monitor and Flag Untrusted Outputs: Regularly review LLM outputs and mark suspicious content, helping users to recognize potentially unreliable information.

LLM02: Insecure Output Handling

Insecure Output Handling occurs when the outputs generated by a LLM are not properly validated or sanitized before being used by other components in a system. Since LLMs can generate various types of content based on input prompts, failing to handle these outputs securely can introduce risks like cross-site scripting (XSS), server-side request forgery (SSRF), or even remote code execution (RCE). Unlike Overreliance (LLM09), which focuses on the accuracy of LLM outputs, Insecure Output Handling specifically addresses vulnerabilities in how these outputs are processed downstream.

As an example, there could be a web application that uses an LLM to summarize user-provided content and renders it back in a webpage. An attacker submits a prompt containing malicious JavaScript code. If the LLM's output is displayed on the webpage without proper sanitization, the JavaScript will execute in the user's browser, leading to XSS. Alternatively, if the LLM's output is sent to a backend database or shell command, it could allow SQL injection or remote code execution if not properly validated.

How to prevent Insecure Output Handling:

1. Zero-Trust Approach: Treat the LLM as an untrusted source, applying strict allow list validation and sanitization to all outputs it generates, especially before passing them to downstream systems or functions.
2. Output Encoding: Encode LLM outputs before displaying them to end users, particularly when dealing with web content where XSS risks are prevalent.

3. Adhere to Security Standards: Follow the OWASP Application Security Verification Standard (ASVS) guidelines, which provide strategies for input validation and sanitization to protect against code injection risks.

LLM03: Training Data Poisoning

Training Data Poisoning refers to the manipulation of the data used to train LLMs, introducing biases, backdoors, or vulnerabilities. This tampered data can degrade the model's effectiveness, introduce harmful biases, or create security flaws that malicious actors can exploit. Poisoned data could lead to inaccurate or inappropriate outputs, compromising user trust, harming brand reputation, and increasing security risks like downstream exploitation.

As an example, there could be a scenario where an LLM is trained on a dataset that has been tampered with by a malicious actor. The poisoned dataset includes subtly manipulated content, such as biased news articles or fabricated facts. When the model is deployed, it may output biased information or incorrect details based on the poisoned data. This not only degrades the model's performance but can also mislead users, potentially harming the model's credibility and the organization's reputation.

How to prevent Training Data Poisoning:

1. Data Validation and Vetting: Verify the sources of training data, especially when sourcing from third-party datasets. Conduct thorough checks on data integrity, and where possible, use trusted data sources.
2. Machine Learning Bill of Materials (ML-BOM): Maintain an ML-BOM to track the provenance of training data and ensure that each source is legitimate and suitable for the model's purpose.
3. Sandboxing and Network Controls: Restrict access to external data sources and use network controls to prevent unintended data scraping during training. This helps ensure that only vetted data is used for training.
4. Adversarial Robustness Techniques: Implement strategies like federated learning and statistical outlier detection to reduce the impact of poisoned data. Periodic testing and monitoring can identify unusual model behaviors that may indicate a poisoning attempt.

5. Human Review and Auditing: Regularly audit model outputs and use a human-in-the-loop approach to validate outputs, especially for sensitive applications. This added layer of scrutiny can catch potential issues early.

LLM04: Model Denial of Service

Model Denial of Service (DoS) is a vulnerability in which an attacker deliberately consumes an excessive amount of computational resources by interacting with a LLM. This can result in degraded service quality, increased costs, or even system crashes. One emerging concern is manipulating the context window of the LLM, which refers to the maximum amount of text the model can process at once. This makes it possible to overwhelm the LLM by exceeding or exploiting this limit, leading to resource exhaustion.

As an example, an attacker may continuously flood the LLM with sequential inputs that each reach the upper limit of the model's context window. This high-volume, resource-intensive traffic overloads the system, resulting in slower response times and even denial of service. As another example, if an LLM-based chatbot is inundated with a flood of recursive or exceptionally long prompts, it can strain computational resources, causing system crashes or significant delays for other users.

How to prevent Model Denial of Service:

1. Rate Limiting: Implement rate limits to restrict the number of requests from a single user or IP address within a specific timeframe. This reduces the chance of overwhelming the system with excessive traffic.
2. Resource Allocation Caps: Set caps on resource usage per request to ensure that complex or high-resource requests do not consume excessive CPU or memory. This helps prevent resource exhaustion.
3. Input Size Restrictions: Limit input size according to the LLM's context window capacity to prevent excessive context expansion. For example, inputs exceeding a predefined character limit can be truncated or rejected.
4. Monitoring and Alerts: Continuously monitor resource utilization and establish alerts for unusual spikes, which may indicate a DoS attempt. This allows for proactive threat detection and response.
5. Developer Awareness and Training: Educate developers about DoS vulnerabilities in LLMs and establish guidelines for secure model deployment. Understanding these risks enables

teams to implement preventative measures more effectively.

LLM05: Supply Chain Vulnerabilities

Supply Chain attacks are incredibly common and this is no different with LLMs, which, in this case refers to risks associated with the third-party components, training data, pre-trained models, and deployment platforms used within LLMs. These vulnerabilities can arise from outdated libraries, tampered models, and even compromised data sources, impacting the security and reliability of the entire application. Unlike traditional software supply chain risks, LLM supply chain vulnerabilities extend to the models and datasets themselves, which may be manipulated to include biases, backdoors, or malware that compromises system integrity.

As an example, an organization uses a third-party pre-trained model to conduct economic analysis. If this model is poisoned with incorrect or biased data, it could generate inaccurate results that mislead decision-making. Additionally, if the organization uses an outdated plugin or compromised library, an attacker could exploit this vulnerability to gain unauthorized access or tamper with sensitive information. Such vulnerabilities can result in significant security breaches, financial loss, or reputational damage.

How to prevent Supply Chain Vulnerabilities:

1. Vet Third-Party Components: Carefully review the terms, privacy policies, and security measures of all third-party model providers, data sources, and plugins. Use only trusted suppliers and ensure they have robust security protocols in place.
2. Maintain a Software Bill of Materials (SBOM): An SBOM provides a complete inventory of all components, allowing for quick detection of vulnerabilities and unauthorized changes. Ensure that all components are up-to-date and apply patches as needed.
3. Use Model and Code Signing: For models and external code, employ digital signatures to verify their integrity and authenticity before use. This helps ensure that no tampering has occurred.
4. Anomaly Detection and Robustness Testing: Conduct adversarial robustness tests and anomaly detection on models and data to catch signs of tampering or data poisoning.

Integrating these checks into your MLOps pipeline can enhance overall security.

5. Implement Monitoring and Patching Policies: Regularly monitor component usage, scan for vulnerabilities, and patch outdated components. For sensitive applications, continuously audit your suppliers' security posture and update components as new threats emerge.

LLM06: Sensitive Information Disclosure

Sensitive Information Disclosure in LLMs occurs when the model inadvertently reveals private, proprietary, or confidential information through its output. This can happen due to the model being trained on sensitive data or because it memorizes and later reproduces private information. Such disclosures can result in significant security breaches, including unauthorized access to personal data, intellectual property leaks, and violations of privacy laws.

As an example, there could be an LLM-based chatbot trained on a dataset containing personal information such as users' full names, addresses, or proprietary business data. If the model memorizes this data, it could accidentally reveal this sensitive information to other users. For instance, a user might ask the chatbot for a recommendation, and the model could inadvertently respond with personal information it learned during training, violating privacy rules.

How to prevent Sensitive Information Disclosure:

1. Data Sanitization: Before training, scrub datasets of personal or sensitive information. Use techniques like anonymization and redaction to ensure no sensitive data remains in the training data.
2. Input and Output Filtering: Implement robust input validation and sanitization to prevent sensitive data from entering the model's training data or being echoed back in outputs.
3. Limit Training Data Exposure: Apply the principle of least privilege by restricting sensitive data from being part of the training dataset. Fine-tune the model with only the data necessary for its task, and ensure high-privilege data is not accessible to lower-privilege users.
4. User Awareness: Make users aware of how their data is processed by providing clear

Terms of Use and offering opt-out options for having their data used in model training.

5. Access Controls: Apply strict access control to external data sources used by the LLM, ensuring that sensitive information is handled securely throughout the system

LLM07: Insecure Plugin Design

Insecure Plugin Design vulnerabilities arise when LLM plugins, which extend the model's capabilities, are not adequately secured. These plugins often allow free-text inputs and may lack proper input validation and access controls. When enabled, plugins can execute various tasks based on the LLM's outputs without further checks, which can expose the system to risks like data exfiltration, remote code execution, and privilege escalation. This vulnerability is particularly dangerous because plugins can operate with elevated permissions while assuming that user inputs are trustworthy.

As an example, there could be a weather plugin that allows users to input a base URL and query. An attacker could craft a malicious input that directs the LLM to a domain they control, allowing them to inject harmful content into the system. Similarly, a plugin that accepts SQL "WHERE" clauses without validation could enable an attacker to execute SQL injection attacks, gaining unauthorized access to data in a database.

How to prevent Insecure Plugin Design:

1. Enforce Parameterized Input: Plugins should restrict inputs to specific parameters and avoid free-form text wherever possible. This can prevent injection attacks and other exploits.
2. Input Validation and Sanitization: Plugins should include robust validation on all inputs. Using Static Application Security Testing (SAST) and Dynamic Application Security Testing (DAST) can help identify vulnerabilities during development.
3. Access Control: Follow the principle of least privilege, limiting each plugin's permissions to only what is necessary. Implement OAuth2 or API keys to control access and ensure only authorized users or components can trigger sensitive actions.
4. Manual Authorization for Sensitive Actions: For actions that could impact user security, such as transferring files or accessing private repositories, require explicit user confirmation.
5. Adhere to OWASP API Security Guidelines: Since plugins often function as REST APIs,

apply best practices from the OWASP API Security Top 10. This includes securing endpoints and applying rate limiting to mitigate potential abuse.

LLM08: Excessive Agency

Excessive Agency in LLM-based applications arises when models are granted too much autonomy or functionality, allowing them to perform actions beyond their intended scope. This vulnerability occurs when an LLM agent has access to functions that are unnecessary for its purpose or operates with excessive permissions, such as being able to modify or delete records instead of only reading them. Unlike Insecure Output Handling, which deals with the lack of validation on the model's outputs, Excessive Agency pertains to the risks involved when an LLM takes actions without proper authorization, potentially leading to confidentiality, integrity, and availability issues.

As an example, there could be an LLM-based assistant that is given access to a user's email account to summarize incoming messages. If the plugin that is used to read emails also has permissions to send messages, a malicious prompt injection could trick the LLM into sending unauthorized emails (or spam) from the user's account.

How to prevent Excessive Agency:

1. Restrict Plugin Functionality: Ensure plugins and tools only provide necessary functions. For example, if a plugin is used to read emails, it should not include capabilities to delete or send emails.
2. Limit Permissions: Follow the principle of least privilege by restricting plugins' access to external systems. For instance, a plugin for database access should be read-only if writing or modifying data is not required.
3. Avoid Open-Ended Functions: Avoid functions like "run shell command" or "fetch URL" that provide broad system access. Instead, use plugins that perform specific, controlled tasks.
4. User Authorization and Scope Tracking: Require plugins to execute actions within the context of a specific user's permissions. For example, using OAuth with limited scopes helps ensure actions align with the user's access level.
5. Human-in-the-Loop Control: Require user confirmation for high-impact actions. For instance, a plugin that posts to social media

should require the user to review and approve the content before it is published.

6. Authorization in Downstream Systems: Implement authorization checks in downstream systems that validate each request against security policies. This prevents the LLM from making unauthorized changes directly.

LLM09: Overreliance

Overreliance occurs when users or systems trust the outputs of a LLM without proper oversight or verification. While LLMs can generate creative and informative content, they are prone to “hallucinations” (producing false or misleading information) or providing authoritative-sounding but incorrect outputs. Overreliance on these models can result in security risks, misinformation, miscommunication, and even legal issues, especially if LLM-generated content is used without validation. This vulnerability becomes especially dangerous in cases where LLMs suggest insecure coding practices or flawed recommendations.

As an example, there could be a development team using an LLM to expedite the coding process. The LLM suggests an insecure code library, and the team, trusting the LLM, incorporates it into their software without review. This introduces a serious vulnerability. As another example, a news organization might use an LLM to generate articles, but if they don’t validate the information, it could lead to the spread of disinformation.

How to prevent Overreliance:

1. Regular Monitoring and Review: Implement processes to review LLM outputs regularly. Use techniques like self-consistency checks or voting mechanisms to compare multiple model responses and filter out inconsistencies.
2. Cross-Verification: Compare the LLM’s output with reliable, trusted sources to ensure the information’s accuracy. This step is crucial, especially in fields where factual accuracy is imperative.
3. Fine-Tuning and Prompt Engineering: Fine-tune models for specific tasks or domains to reduce hallucinations. Techniques like parameter-efficient tuning (PET) and chain-of-thought prompting can help improve the quality of LLM outputs.
4. Automated Validation: Use automated validation tools to cross-check generated outputs against known facts or data, adding an extra layer of security.

5. Risk Communication: Clearly communicate the limitations of LLMs to users, highlighting the potential for errors. Transparent disclaimers can help manage user expectations and encourage cautious use of LLM outputs.
6. Secure Coding Practices: For development environments, establish guidelines to prevent the integration of potentially insecure code. Avoid relying solely on LLM-generated code without thorough review.

LLM10: Model Theft

Model Theft refers to the unauthorized access, extraction, or replication of proprietary LLMs by malicious actors. These models, containing valuable intellectual property, are at risk of exfiltration, which can lead to significant economic and reputational loss, erosion of competitive advantage, and unauthorized access to sensitive information encoded within the model. Attackers may steal models directly from company infrastructure or replicate them by querying APIs to build shadow models that mimic the original. As LLMs become more prevalent, safeguarding their confidentiality and integrity is crucial.

As an example, an attacker could exploit a misconfiguration in a company's network security settings, gaining access to their LLM model repository. Once inside, the attacker could exfiltrate the proprietary model and use it to build a competing service. Alternatively, an insider may leak model artifacts, allowing adversaries to launch gray box adversarial attacks or fine-tune their own models with stolen data.

How to prevent Model Theft:

1. Access Controls and Authentication: Use Role-Based Access Control (RBAC) and enforce strong authentication mechanisms to limit unauthorized access to LLM repositories and training environments. Adhere to the principle of least privilege for all user accounts.
2. Supplier and Dependency Management: Monitor and verify the security of suppliers and dependencies to reduce the risk of supply chain attacks, ensuring that third-party components are secure.
3. Centralized Model Inventory: Maintain a central ML Model Registry with access controls, logging, and authentication for all production models. This can aid in governance, compliance, and prompt detection of unauthorized activities.
4. Network Restrictions: Limit LLM access to internal services, APIs, and network resources.

This reduces the attack surface for side-channel attacks or unauthorized model access.

5. Continuous Monitoring and Logging: Regularly monitor access logs for unusual activity and promptly address any unauthorized access. Automated governance workflows can also help streamline access and deployment controls.
6. Adversarial Robustness: Implement adversarial robustness training to help detect extraction queries and defend against side-channel attacks. Rate-limit API calls to further protect against data exfiltration.
7. Watermarking Techniques: Embed unique watermarks within the model to track unauthorized copies or detect theft during the model's lifecycle.

Wrapping it all up

As LLMs continue to grow in capability and integration across industries, their security risks must be managed with the same vigilance as any other critical system. From Prompt Injection to Model Theft, the vulnerabilities outlined in the OWASP Top 10 for LLMs highlight the unique challenges posed by these models, particularly when they are granted excessive agency or have access to sensitive data. Addressing these risks requires a multifaceted approach involving strict access controls, robust validation processes, continuous monitoring, and proactive governance.

For technical leadership, this means ensuring that development and operational teams implement best practices across the LLM lifecycle starting from securing training data to ensuring safe interaction between LLMs and external systems through plugins and APIs. Prioritizing security frameworks such as the OWASP ASVS, adopting MLOps best practices, and maintaining vigilance over supply chains and insider threats are key steps to safeguarding LLM deployments. Ultimately, strong leadership that emphasizes security-first practices will protect both intellectual property and organizational integrity, while fostering trust in the use of AI technologies.