OWASP | GenAI SECURITY PROJECT

# OWASP Top 10 for LLM Applications 2026

Version 2026

August 4th, 2026

## License and Usage



## Revision History

- 2023-08-01 Version 1.0 Release
- 2023-10-16 Version 1.1 Release
- 2024-11-18 Version 2025 Release
- [2026 release date] Version 2026 Release

# Table of Content

# Letter from the Project Leads

Stop trying to build a model that cannot be fooled. Build the system around it, so that when the model is fooled, and it will be, nothing important breaks. That posture runs through all ten entries here. This year, for the first time, we can show you the evidence instead of asking you to take it on faith.

Every version of this list before now was built on judgment. Hundreds of practitioners weighed in on what matters most. That vote is still the spine of the list, and it belongs there. This year, we did something the project has never done. We tested the vote against a record of what has actually gone wrong. We pulled together a corpus of 7,714 real incidents from public vulnerability databases and an AI-harm database, and we built classifiers that read them and placed the 6,639 that carried enough detail to sort. Then we asked one blunt question. Does what practitioners fear match what the incident record shows?

The answer was no, not always, and that was the surprise. The vote and the data parted ways in specific, useful places, and those disagreements taught us more than the agreements did.

Prompt injection is the clearest case. Practitioners rank it the number one risk. Rank the categories by the raw incident record instead, and it falls out of the top 10 entirely. That gap is a defense effect. Teams fight injection hard, so fewer clean exploits reach a public database, and the public count understates the risk that mature teams already spend real money holding off. The surface still sits everywhere a model reads untrusted input, which is to say everywhere. It stays at number one. Since you cannot close that surface, you build for the day it is used against you.

Misinformation runs the other direction, and it is the entry I want you to slow down on. Voters placed it near the bottom. The incident record placed it near the top, the widest gap in the direction that actually hurts, where the vote sits low and the evidence sits high. The list still seats Misinformation in the middle: the vote holds the greater weight, but the evidence pulled it up. The disagreement is the point. When a model's fluent, confident output drives a decision or a tool call, a wrong answer turns into a wrong action, and the record says that failure lands more often than the vote assumes.

The community vote carries three-quarters of the weight. The incident data covers the remaining quarter. We deliberately weighed the vote heavily. This list is a consensus product, and one noisy year of data does not get to overturn the judgment of the people doing the work. A quarter weight is enough to move an entry a tier when the gap between belief and evidence runs wide. It is not enough to let imperfect data rewrite the list on its own. That balance decided every final slot in the Top 10.

## What's New in the 2026 Top 10

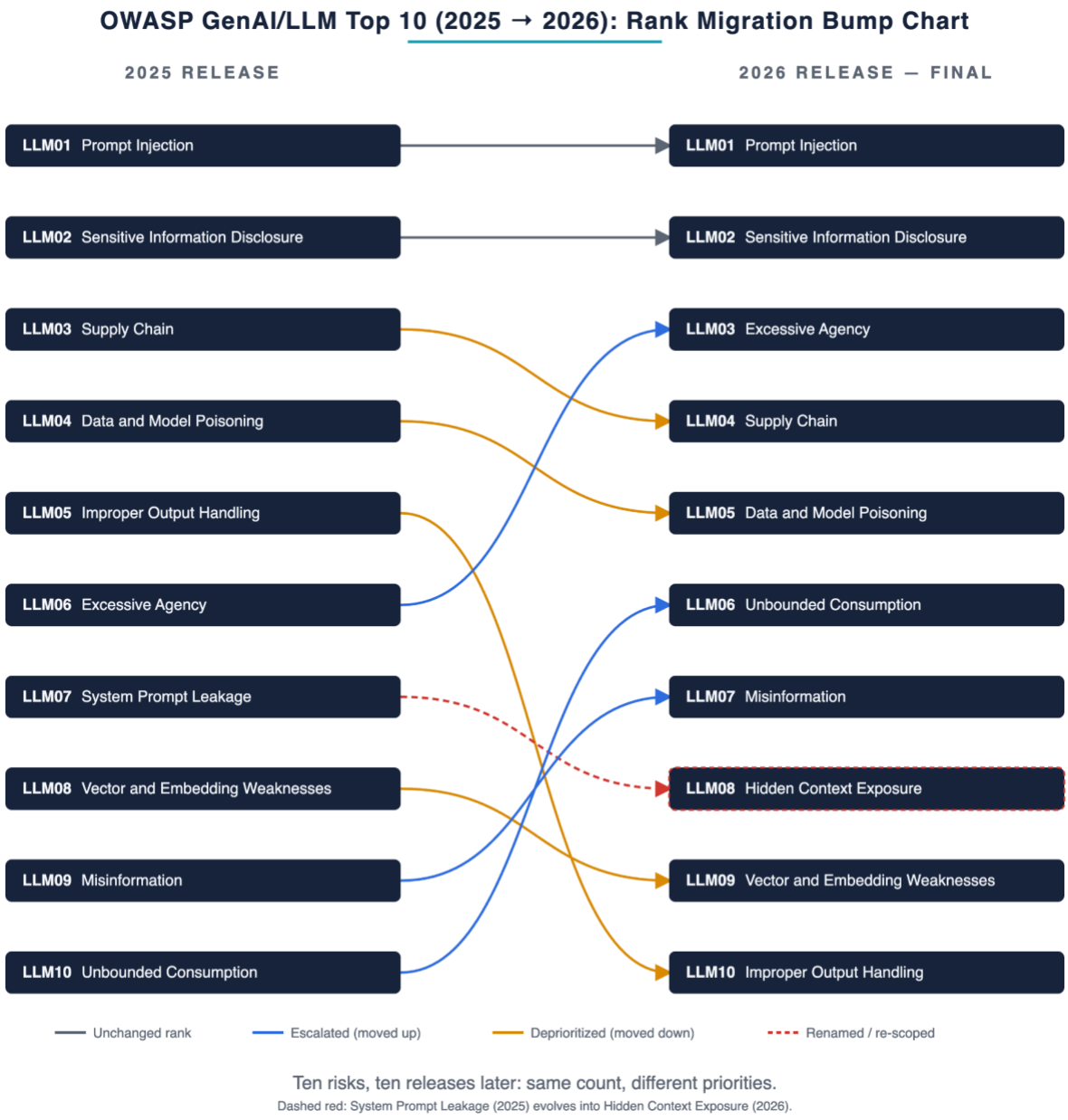



*Figure 1: Rank migration from the 2025 to the final 2026 OWASP GenAI/LLM Top 10, color-coded by movement (steady, escalated, deprioritized, or renamed/re-scoped).*

The order moved more than in past years, and the moves track that gap between belief and evidence. Excessive Agency climbed to third, the most consequential move on the list, because the vote and the record agree that agentic deployments are where the damage is landing. Unbounded Consumption rose four places, carried by practitioners who weigh resource and cost exhaustion higher than in its old rank. Improper Output Handling fell the furthest, from fifth to tenth. Prompt Injection held the top spot on the strength of the vote and the defense effect behind it. Sensitive Information Disclosure held second, the one place at the top where belief and evidence simply agree, and where our confidence is highest. What used to be System Prompt Leakage is now Hidden Context Exposure, a broader framework for the same failure to trust information that should have stayed out of reach.

Several entries also grew. We folded newer, sharper risks into the entries that already own them. Standing up thin new categories would have splintered the list for no gain. Prompt Injection now covers cross-modal attacks, the kind that hide instructions inside an image or an audio track. Supply Chain now accounts for the trust failure when a promoted model artifact is not what it claims to be. Data and Model Poisoning now absorbs fine-tuning subversion. Improper Output Handling now spans the insecure code that assistants generate at scale. The incidents behind those risks were always real. Now they count where they belong.

One boundary matters more every year. This list owns the risk when the model is a component inside your application. The moment that model becomes an actor, with tools it can call, memory it carries between sessions, and consequences it sets in motion downstream, the risk moves to the OWASP Agentic Top 10. Many of the incidents we read sit right on that boundary. Read an entry here for the model-as-component failure. When your model starts acting on its own, pair it with the Agentic list, because neither one covers that ground alone.

## Moving Forward

You can trust this list and act on it today. It brings together the judgment of the practitioners who attack and defend these systems along with the record of what has actually gone wrong in the field, each checked against the other. That grounding is what earlier versions could only promise. Work all ten, start at the top, and build each of them for the day the model is turned against you. Do that, and you are covering both what the field fears most and what it has already been burned by.

Like the technology it covers, this list is a product of the community. It has been shaped by developers, data scientists, and security practitioners who brought their judgment and, this year, their incident records to the work. We are grateful to everyone who contributed, argued, and pushed us to check our beliefs against the evidence. We hope it helps you defend what you are building.


***Steve Wilson***

Project Lead, OWASP Top 10 for LLM Applications
LinkedIn: https://www.linkedin.com/in/wilsonsd/

***Rock Lambros***

Co-Lead, OWASP Top 10 for LLM Applications
LinkedIn: https://www.linkedin.com/in/rocklambros

# LLM Top 10 at a Glance

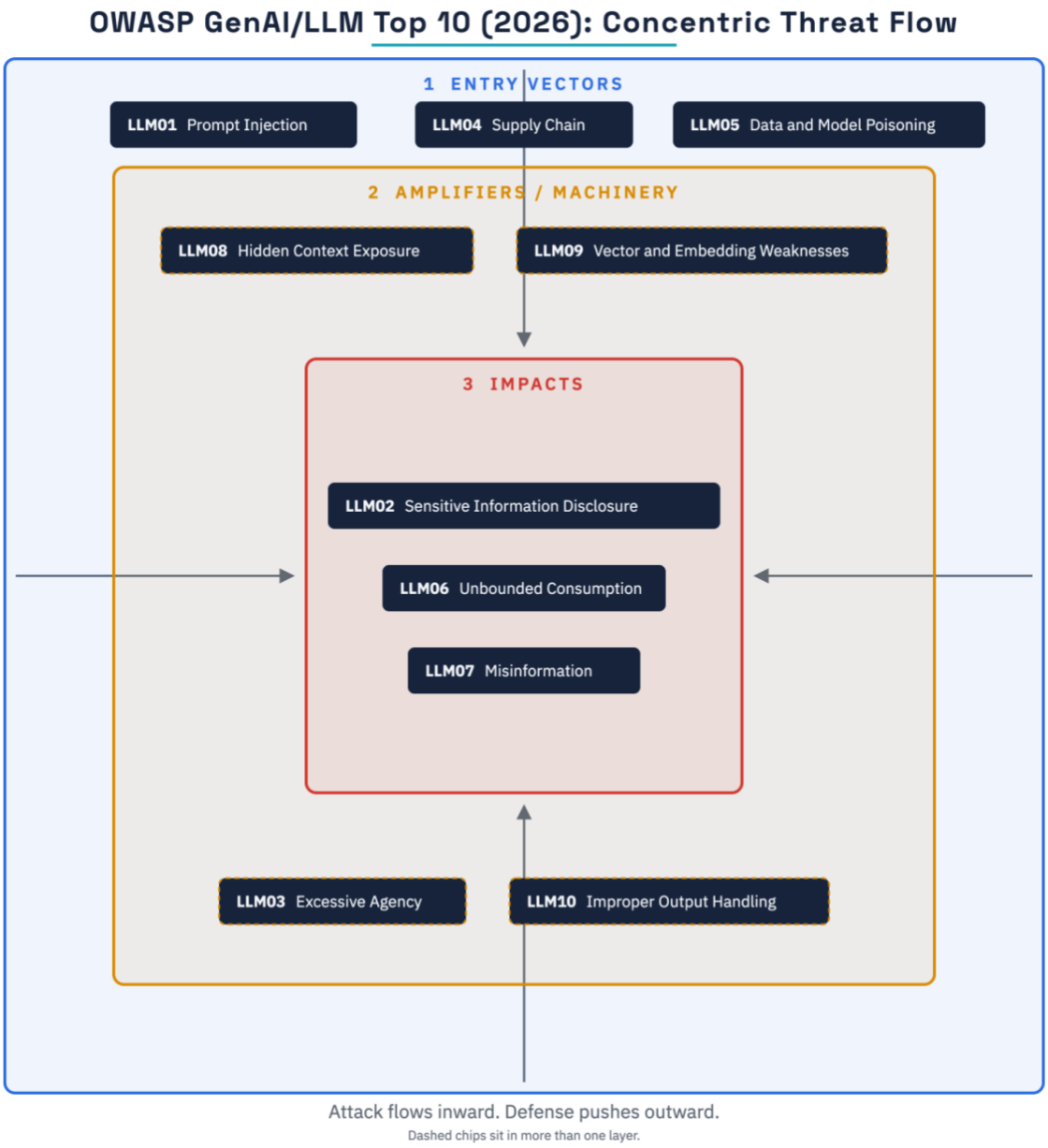



*Figure 2: Attack surface visualized as a bullseye. Entry vectors on the perimeter converge through amplifying machinery to three core impacts. Note: Not every LLM04 attack flows inward: some deliver impact directly.*

# LLM01:2026 Prompt Injection

## Description

A **prompt-injection vulnerability** occurs when input to a large language model (LLM), whether direct user input, retrieved content, tool output, image, audio, or video content, intermediate reasoning, or persistent memory, alters the model's behavior in ways the application developer did not intend. LLMs make no architectural distinction between "instructions" and "data" (both are tokens on the same stream), so there is no clean equivalent to parameterized queries (NCSC, 2025). Inputs need not be human-readable, need not arrive directly from a user, and need not be visible in the rendered interface to influence the model.

Prompt-injection vulnerabilities exist in how models process input and how that input can force the model to pass data or instructions incorrectly to other parts of the system. Three deployment-time properties make this worse. First, **context-window pooling**: the model treats system prompt, user input, retrieved documents, tool outputs, conversation history, and memory as a single token stream, with no enforced trust boundary. Second, **memory persistence**: an injection that writes to long-term memory, a RAG corpus, a vector store, or a hosted memory service taints every subsequent session that reads from that store. Third, **agentic execution**: when the model's output drives tool calls (file system, shell, email, cloud APIs, MCP servers, sub-agents), the blast radius extends from the chat surface to whatever the agent's tools can reach, and tool outputs re-enter the context window, enabling chained or self-replicating effects.

A prompt-injection anatomy can be characterized along three axes. **Delivery surface** is how it reaches the model (direct input, retrieved content, tool output, tool connection channel, or persistent memory). **Propagation behavior** is how it spreads across time and boundaries (single-shot, multi-step kill-chain, cross-session through memory or RAG, or self-replicating across agents). **Encoding** is how the malicious instructions are represented in tokens or pixels (plain text, base64 or other obfuscation, invisible Unicode, multimodal or steganographic, low-resource language). Decomposing a scenario along these axes is a useful threat-modeling step before selecting which mitigations apply.

The severity and nature of a successful prompt injection vary with the business context the model operates in and the agency with which it is architected. Prompt injection can lead to outcomes that include but are not limited to:

- Disclosure of sensitive information, system-prompt content, retrieved private documents, or infrastructure details.
- Manipulation of model output to produce biased, harmful, or attacker-chosen content that downstream systems or users act on.

- Unauthorized invocation of tools the agent is permitted to call, escalating to arbitrary command execution and destructive actions where the agent has shell, file-system, or cloud-API access.
- Data exfiltration via image-URL channels, hidden Unicode characters in rendered output, or covert tool-logging side channels.
- Persistent compromise of agent behavior across sessions through memory or RAG corpus poisoning.

*Note: prompt injection differs from LLM02:2026 Sensitive Information Disclosure, which addresses what the model leaks through its outputs, including reasoning-channel content, and from LLM03:2026 Excessive Agency, which addresses the consequences of model output reaching privileged actions. This entry concerns the input boundary itself. Sanitization and validation of model outputs before they reach downstream components is covered by LLM10:2026 Improper Output Handling.*

# Types of Prompt Injection

## Direct Prompt Injection

A user, or an attacker with the user's access path, supplies input that changes model behavior in undesired ways. Direct injection can be **intentional** (a malicious user crafting a jailbreak) or **unintentional** (a legitimate user pasting content that happens to contain conflicting instructions, or a user who relies on an LLM to help them and inadvertently optimizes their input against an unrelated downstream LLM, as in Scenario #3).

Jailbreaking is the subset of prompt injection where the attacker's goal is to make the model violate its safety protocols. Application-level safeguards help contain it, but effective prevention requires ongoing updates to the model's training and safety mechanisms.

## Indirect Prompt Injection

The model ingests content from an external source (a web page, a document, an email, a tool response, a retrieved RAG passage, an image, an MCP server's output, a database row, or an issue title) that contains data which acts as prompt injection. The user did not supply or see those instructions. The trust profile of the delivery surface determines what defenses are practical:

- **Untrusted surfaces.** Public web pages, emails from unknown senders, search results. Defenders must treat anything from these sources as suspicious. Most prompt-injection research has focused here.
- **Semi-trusted surfaces.** Issue titles in a public bug tracker, package READMEs and changelogs, third-party API responses: content the user chose to retrieve but did not author. The user trusts the platform but not necessarily individual contributors.

- **Trusted surfaces.** The developer's own repositories, databases, internal documents, and mail. The developer may not realize an attacker has placed content here, perhaps via an unrelated upstream vector such as a public bug-report form.

The shared structure: the attacker does not need to compromise the backend directly. They place text where the developer's LLM will read it, and the LLM, operating with the developer's privileges, does the work. Defenses that focus only on the chat surface miss this entirely.

Indirect prompt injection increasingly turns the user's own LLM instance into the weapon against the user's own backend. The pattern: an attacker submits text into a trusted-by-the-user location through a low-privilege channel (a public form, a customer ticket, a community pull request) and waits for the user's MCP-connected agent or developer assistant to read that text while operating under the user's elevated credentials. The agent, not the attacker, performs the privileged action (see Common Example #3; Scenario #9 walks through the production proof-of-concepts).

## Common Examples of Risk

1. **Direct prompt-input override:** a user message overrides the system prompt's role and capability limits, making the model disclose, generate, or act outside its intended scope. Intentional and unintentional inputs both count.
2. **Indirect injection through retrieved content:** attacker instructions ride in a RAG passage, web page, document, or email and run when the content reaches the context window (for example, EmailGPT; INCIBE-CERT, 2024).
3. **Trusted-surface indirect injection:** text planted in a low-privilege but trusted channel (issue tracker, feedback form, support ticket) makes the user's LLM act under its own elevated credentials, exfiltrating repositories, dumping databases, or modifying IDE config, actions the attacker could not perform directly (Invariant Labs, 2025; General Analysis, 2025; Rehberger, 2025a).
4. **Multimodal and steganographic injection:** sub-perceptual perturbations in images, audio, or video are extracted by the encoder (Clusmann et al., 2025; see Scenario #6).
5. **Invisible-character injection and exfiltration:** tag-block, variation-selector, and zero-width Unicode carry instructions or exfiltrate bytes inside benign-looking text. The August 2024 M365 Copilot ASCII-smuggling proof of concept exfiltrated a Slack MFA code (Rehberger, 2024).
6. **Cross-session memory and RAG corpus poisoning:** one tainted entry in persistent memory or a RAG corpus reaches every future session that reads it (W. Zou et al., 2025; see Scenario #4).

7. **Fine-tuning interface as gradient oracle ("fun-tuning"):** an attacker reads per-example loss from a vendor's fine-tuning API to optimize a payload (65% to 82% attack success on Gemini), bringing white-box-style optimization to closed-weight models (Labunets et al., 2025).
8. **Multilingual, encoded, or low-resource-language payloads:** low-resource and code-mixed inputs raise attack success and evade classifiers not trained on the scheme, and Base64, ROT13, or emoji encodings bypass filters that never saw the encoding (Hackett et al., 2025).

## Prevention and Mitigation Strategies

Prompt injection is intrinsic to current generative AI: LLMs make no architectural distinction between instructions and data, and their behavior is stochastic, so no reliable prevention mechanism exists today, a position consistent with NIST (2025), NCSC (2025), and Debenedetti et al. (2025). Defense is therefore architectural rather than interceptive. Design the surrounding system on the explicit assumption that the model's instruction boundary will eventually be bypassed, and constrain what the model is permitted to do, and what its outputs are permitted to reach, so a successful injection does not translate into a successful exploit.

Most high-impact prompt-injection incidents on record became severe because the injection landed inside a system whose tools, scopes, or output-rendering capabilities let the compromised model act on the attacker's behalf at the user's privilege level (see Scenarios #7 through #9). This is the operational relationship between this entry and **LLM03:2026 Excessive Agency**: prompt injection is the input-side compromise, and excessive functionality, permissions, or autonomy are what give that compromise consequences outside the chat window. Simon Willison's "lethal trifecta" (2025) restates the same structural diagnosis as a pre-deployment check: an agent that can simultaneously access private data, ingest untrusted content, and communicate externally has the conditions for high-impact exploitation, and removing any one leg removes them.

Apply the controls below as defense-in-depth, since no single control is sufficient. Some reduce injection success and are expected to degrade against adaptive attackers. Others bound the blast radius once injection succeeds, and these are what survive against attackers who can probe the system. For agentic deployments, the least-privilege and capability-budgeting controls (#4, #8) are load-bearing, and **LLM03:2026** carries the full agency-side treatment.

1. **Constrain the model's role and capabilities in the system prompt.** Use declarative allow and deny statements ("assist with X only, do not access Y, do not forward output to external addresses"), not open-ended grants. This is a partial control only: an attacker who infers the prompt can bypass it (Nasr et al., 2025), so pair it with the privilege controls in #4.

2. **Define a strict output schema and validate every response in trusted application code** before any downstream system acts on it, using structural validation rather than a second LLM call. This catches format violations, not semantic manipulation: a schema-valid response can still carry a malicious SQL query or an exfiltration-formatted email body.
3. **Filter at every modality boundary (text, image, audio, structured data), not just text.** Run modality-specific classifiers, OCR over images, and transcription over audio, then apply text filters to the extracted content. Semantic filters are evadable by rephrasing or encoding, and low-resource and code-mixed inputs degrade their accuracy (Hackett et al., 2025).
4. **Hold credentials and state-change capability in application code, not the model, and grant least privilege per operation.** Route privileged calls through a deterministic policy engine that re-validates intent and arguments at execution time. NIST AI 100-2 E2025 and the CISA and Five Eyes OT joint guidance (CISA et al., 2025) frame this deterministic mediation as a baseline procurement expectation. Broad "convenience" permissions and multi-agent hops re-introduce the risk downstream.
5. **Strip tag-block (U+E0000 to E007F), variation-selector (U+FE00 to FE0F), and zero-width (U+200B, U+200C, U+200D, U+2060) characters at every ingest and render boundary.** These are invisible in normal rendering and smuggle instructions or exfiltration bytes (see Common Example #5), and variation-selector variants (Rehberger, 2025c) smuggle arbitrary bytes invisibly. Stripping does not stop visible-text payloads or future steganographic classes.
6. **Pass external content through a structurally separate, provenance-labeled channel** so the model can distinguish data from instructions (S. Chen et al., 2025; Microsoft Research, 2025). This reduces attack success in non-adaptive tests only: an attacker who knows the marking scheme can mimic it, and StruQ was bypassed under adaptive attack (Nasr et al., 2025).
7. **Require explicit human confirmation before any privileged, irreversible, or externally visible action**, surfacing the exact rendered action rather than a summary to the reviewer. Invisible-character smuggling can make the displayed action differ from the executed one (#5), and approval fatigue degrades reviewer judgment at volume.
8. **Budget agent capabilities with the Rule of Two as a floor** (Meta AI, 2025). Treat simultaneous access to (A) untrusted input, (B) sensitive data, and (C) state change or external communication as high-risk: any [A,B,C] agent needs per-action human approval, and [A,B] or [A,C] configurations need an explicit residual-risk assessment (see Scenario #8). NIST AI 100-2 E2025 and the CISA, FBI, NSA, and ACSC OT guidance (CISA et al., 2025) endorse the rule, which is silent on autonomy depth (Noma Security, 2025).
9. **Treat agent memory writes as privileged operations.** Log the causing prompt, classify writes for instruction or role-modification content, and require approval before instruction-bearing memories persist across sessions. A February 2025 Gemini PoC (Rehberger, 2025b) poisoned memory via

delayed tool invocation (MITRE, n.d.). Factual entries shade into instructions, and incremental writes can evade per-write classification.

10. **Pin, sign, and verify every MCP server and third-party tool package, audit tool descriptions for hidden instructions, and monitor tool composition.** Treat these as a software supply-chain surface, covered by **LLM04:2026 Supply Chain** for third-party tool packages and by ASI04 Agentic Supply Chain Vulnerabilities for MCP servers and tool registries (see Scenario #9). Pinning does not stop a payload shipped in the pinned version or tool-description poisoning that leaves the version unchanged.
11. **Test against adaptive attackers who have read the deployed defense, and reject static-only attack-success claims.** Baseline with AgentDojo (Debenedetti et al., 2024) and JailbreakBench (Chao et al., 2024), then red-team with the full defense specification disclosed to the testers. Nasr et al. (2025) found static attack success near zero while adaptive attack success exceeded 90% for most of 12 recent defenses (see also LLMail-Inject, Microsoft Security Response Center, 2025).

## Example Attack Scenarios

### *Scenario #1: Direct Injection*

An attacker prompts a customer-support chatbot to ignore its guidelines, query private data stores, and send emails, leading to unauthorized access and privilege escalation. **Anatomy:** (a) direct user input, (b) single-shot, (c) plain text

### *Scenario #2: Indirect Injection via Retrieved Web Content*

A user asks an assistant to summarize a web page containing hidden instructions. The model inserts a markdown image whose URL exfiltrates the private conversation to an attacker-controlled domain. The user sees only the rendered image, never the instruction. **Anatomy:** (a) retrieved web content (indirect), (b) single-shot with image-URL exfiltration, (c) plain text hidden in page source

### *Scenario #3: Unintentional Injection*

A job-description PDF embeds an AI-detection instruction. An applicant unknowingly uses an LLM to optimize their resume against it, the model surfaces the instruction, and the recruiting system flags the candidate, a prompt injection with neither party acting maliciously. **Anatomy:** (a) indirect (document / PDF), (b) single-shot, (c) plain text

### *Scenario #4: RAG Repository Poisoning*

An attacker contributes poisoned documents to a corpus the application retrieves over. A matching query returns the modified content, and its instructions alter the output. As few as five poisoned documents have

reached roughly 90% attack success against a knowledge base of millions of texts (W. Zou et al., 2025). **Anatomy:** (a) retrieved content (RAG corpus), (b) cross-session / cross-user, (c) plain text

### *Scenario #5: Payload Splitting*

An attacker splits malicious instructions across multiple resume fields (header, body, attachment) so no single field looks malicious to a per-field classifier. The LLM recombines them at evaluation, and its recommendation is manipulated. **Anatomy:** (a) direct user input split across fields, (b) single-shot recombined at evaluation, (c) plain text fragmented

### *Scenario #6: Multimodal Steganographic Injection*

An attacker embeds an instruction in an image below the human visual threshold. A multimodal model's vision encoder extracts the payload, behavior changes, and the model produces harmful output or an unauthorized tool invocation. This was demonstrated against four frontier vision-language models in oncology imaging (Clusmann et al., 2025) and against general-purpose models via combined visual perturbation and text steering (R. Chen et al., 2025). **Anatomy:** (a) image input (indirect / multimodal), (b) single-shot, (c) steganographic / pixel-level encoding

### *Scenario #7: Zero-Click Document-Borne Agentic Exfiltration*

A crafted email triggers an LLM-powered productivity assistant to exfiltrate organizational data with no user interaction. Aim Security demonstrated this against Microsoft 365 Copilot (Reddy & Gujral, 2025), bypassing both the deployed prompt-injection classifier and the link-redaction filter. **Anatomy:** (a) email / document (indirect), (b) single-shot with tool invocation, (c) plain text with invisible-Unicode exfiltration channel

### *Scenario #8: Agentic Destructive Command Execution*

Two July 2025 events show the same class of impact via different vectors. An attacker committed a destructive system prompt to the Amazon Q VS Code extension repository before AWS reverted it, though the committed code failed to execute due to a syntax error (Amazon Web Services, 2025a). Separately, a runtime injection caused Amazon Q to execute arbitrary code (Amazon Web Services, 2025b). An agent with shell, file-system, or cloud-API access amplifies an injection into a host-impacting incident. **Anatomy:** (a) supply-chain / compromised system prompt, or runtime indirect injection, (b) persistent cross-session, or single-shot with shell tool execution, (c) plain text

### *Scenario #9: Trusted-Backend Indirect Injection through MCP*

An attacker plants text in a low-privilege channel (a public GitHub issue, a support ticket, or a malicious npm package), and the developer's LLM reads it under elevated credentials. Invariant Labs (2025) exfiltrated private repositories via a poisoned GitHub issue, General Analysis (2025) dumped a production database through Cursor's Supabase MCP server running `service_role` and bypassing row-level security, and the malicious `postmark-mcp` package (Koi Security, 2025; Toulas, 2025) BCC'd email to an attacker across an

estimated 300 organizations. **Anatomy:** (a) trusted-surface indirect (MCP channel: issue, ticket, npm package), (b) multi-step tool-chain, (c) plain text

# LLM02:2026 Sensitive Information Disclosure

## Description

Sensitive information disclosure occurs when an LLM-integrated system exposes confidential, regulated, privileged, or proprietary data through a channel the data subject, controller, or system owner did not authorize. The channel is not only the final answer: tool-call arguments, reasoning traces, retrieved chunks, multimodal output, logs, telemetry, embeddings, and observable inference properties (timing, token length, log-probabilities, confidence, cache-hit behavior) are all disclosure surfaces. Treat each as an output subject to the same classification and redaction rules.

Disclosure arises across four phases of the LLM lifecycle:

1. **Training-time.** A model, fine-tune, or LoRA adapter memorizes corpus content and later reproduces it verbatim or in recoverable form. Memorization scales log-linearly with capacity, duplication, and context length. Narrow adapters memorize rare examples with high fidelity, a targeted extraction surface distinct from the base model.
2. **Inference-time.** The model discloses live context (system prompt, RAG chunks, files, tool outputs, memory, or another session's data), often because summarization, translation, or extraction surfaces more than was asked, including visually-redacted spans.
3. **Pipeline-time.** Fine-tuning, distillation, synthetic-data generation, gradients, SDKs, and observability move sensitive data into derived artifacts.
4. **Observation-time.** Adversaries infer facts from externally measurable properties (token length under TLS, latency, log-probabilities, confidence, cache-hit signals) without receiving content.

Protected information includes PII, PHI, financial data, credentials, API keys, trade secrets, model weights, privileged communications, classified or export-controlled material, and biometric and genomic identifiers. Two structural failures drive most incidents. First, **oversharing upstream**: unscoped drives, legacy permissions, and knowledge bases feed RAG with sensitive data the model then retrieves as designed. The fix is the data surface, not the model (DSGAI01). Second, **persistence**: once data influences weights, embeddings, or adapters it stays extractable after source deletion, straining GDPR Article 17 and CCPA §1798.105 erasure obligations. **Open-weights** deployments cannot rely on rate limits, since extraction, membership inference, and inversion run offline at unbounded rates.

Severity should turn on what the recipient can learn, not on whether the leak looked like natural language. Applicable regimes include the EU AI Act (Regulation (EU) 2024/1689, high-risk obligations from August 2026), GDPR, HIPAA, CCPA/CPRA, ISO/IEC 42001, and NIST AI 600-1. This entry covers the LLM as an application *component*. Where it acts as an autonomous actor (cross-session memory, tool choice, multi-step exfiltration), amplified risk is owned by ASI, with deeper controls in DSGAI.

## Common Examples of Risk

1. **Training-data memorization and extraction.** The November 2023 "poem" divergence attack drove `gpt-3.5-turbo` to emit more than 10,000 unique memorized examples for roughly USD 200 (Nasr et al., 2023). Vendor patches have been bypassed repeatedly. Fine-tuned models and their LoRA adapters are more extractable than base models of the same scale, a targeted extraction surface distinct from the base model. Litigation exhibits in NYT v. OpenAI and Getty v. Stability AI include examples of verbatim reproduction, with the Getty case centered on watermark reproduction.
2. **Inference-time context and output disclosure.** The March 2023 ChatGPT Redis bug exposed payment PII for 1.2% of Plus subscribers. More than 4,500 shared conversations were indexed by Google in 2025 through missing `noindex` directives. Clinical-embedding vector stores sit in HIPAA audit-control scope that most teams have not operationalized, and retrieval-layer authorization is widely under-implemented. Treat reasoning traces and tool arguments as outputs, not debugging leftovers. The trace channel also enables reasoning-trace-coercion model extraction. Regex and blocklist filters fall to cross-lingual, base64, and hex encodings. Aggregation across individually-permitted sources (budget + hiring + diligence → a pending M&A target) is a disclosure when policy prohibits the synthesized conclusion.
3. **Embedding and representation disclosure.** Modern inversion reconstructs plaintext from leaked or exported vectors, so an "embeddings-only" backup is a source-document breach. Cosine similarity does not respect ACLs. Authorize before retrieval, because post-generation filtering cannot undo a chunk already supplied to the model. Mechanisms are owned by LLM09:2026 and DSGAI13. This entry owns the regulatory consequence.
4. **Multimodal disclosure.** Vision models OCR credentials and PII from screenshots, notifications, and PDF metadata. Generators reproduce watermarks and identifiable faces (the Getty / Stable Diffusion watermark case is canonical). Cross-modal transformation (text rendered as image, image OCR'd to text) bypasses single-modality DLP.
5. **Inference-time side channels.** SPV-MIA raised membership-inference AUC to 0.9 against fine-tuned targets (Fu et al., 2024), enough for a regulatory breach determination about a named individual. Whisper Leak (McDonald & Bar Or, 2025) classified conversation topics at greater than 98% AUPRC across 28 production models from encrypted traffic. Weiss et al. (2024) reconstructed 29% of response content and inferred topic for 55% via token length. Wu et al. (2025) demonstrated prompt leakage through KV-cache sharing in multi-tenant serving. Dong et al. (2025) inverted a

4,112-token medical prompt from a middle layer at an F1 of 0.8688 (token matching). Carlini et al. (2024) recovered a production model's projection layer through a logit-bias channel.

6. **Training-pipeline disclosure.** Gradient inversion by a malicious server (Boenisch et al., 2021), distillation, and synthetic-data carryover move examples into derived models. Iterative query attacks against a DP-protected model refine LLM-generated queries and home in on confidence spikes to re-identify individuals, so fixed-epsilon DP is necessary but not sufficient without rate-limiting, query-pattern detection, and per-user budgets.
7. **Platform and ecosystem disclosure.** Observability platforms (Langfuse, LangSmith, Datadog LLM Observability) log full prompts, completions, chunks, and traces by default. Two representative incidents: DeepSeek's January 2025 ClickHouse exposure of more than one million rows of logs and API keys (Wiz, 2025), and Check Point's 2026 disclosure of ChatGPT exfiltration through a hidden outbound channel in the code-execution runtime, where one crafted prompt turned the runtime into a silent DNS channel while the visible answer stayed benign (Check Point Research, 2026).

## Prevention and Mitigation Strategies

Mitigations follow the DSGAI tiered structure for a graduated implementation path.

### Tier 1: Foundational (every deployment)

1. Govern corpora: provenance, classification, and deduplication across near-duplicates, transliterations, and format variants. Scrub PII at ingest. Deduplication reduces but does not eliminate memorization.
2. Minimize context: send only task-required fields to external providers. Disable auto-context (`customer_360`, full-record append) unless justified per template.
3. Authorize before retrieval: enforce document- and chunk-level authorization inside the index query, not at the application layer after retrieval. Isolate per-tenant indexes for high-sensitivity workloads.
4. System-prompt hygiene: never store secrets, credentials, or regulated data in system prompts.
5. Sanitize with classifiers, not regex alone: pattern matching plus NER plus trained classifiers, because regex fails on encoded and cross-lingual output.
6. Budget queries per user and per session on sensitive endpoints to disrupt enumeration and membership probing.
7. Operational hygiene: restrict and scrub logs and traces before APM ingestion, encrypt in transit and at rest, and technically enforce no-train/no-retain rather than policy text alone.

### Tier 2: Hardening (regulated / high-sensitivity)

1. DP-SGD calibrated to sensitivity and cardinality with overfitting monitored as a memorization proxy. Pair with detection, because fixed budgets degrade under adaptive querying.

2. Vector-store protection: encryption, ACLs separate from document ACLs, restricted export APIs, minimum-scope k-NN, and embedding-space probing detection.
3. Gate log-probabilities, confidence, and explanations on production endpoints.
4. Classify and redact reasoning traces as first-class output. Never log raw traces to unrestricted observability.
5. Side-channel defenses: random padding and token batching for streaming, segregation of high-sensitivity tenants on dedicated prefix caches, and partitioned KV caches under co-tenancy.
6. Format-preserving encryption for structured identifiers, with strict internal-versus-external routing separation and field allowlists on the external path.
7. AI-aware audit logging into SIEM, continuous DLP and AI-SPM, and a documented domain inventory with an enforced join policy so individually-permitted sources cannot combine into prohibited conclusions.

### Tier 3: Advanced (regulated, classified, high-target)

1. Confidential computing (Intel TDX, AMD SEV-SNP, AWS Nitro Enclaves) or emerging privacy-preserving inference (the AloePri covariant-obfuscation framework, Lin et al., 2026) where the threat model justifies the utility and latency cost.
2. Verifiable erasure across raw data, embeddings, checkpoints, and adapters, validated by post-unlearning extraction and membership-inference probes.
3. Disclosure red-teaming as a release gate: extraction, membership inference, embedding inversion, internal-state inversion, side channels, and LoRA extractability, measured quantitatively and aligned to MITRE ATLAS.
4. Audit synthetic data against extractors. Resist distillation with probing detection, rate limits, and watermarking. Budget aggregate analytics.
5. Exercise a disclosure incident-response playbook: scope by data class and affected subject or session, then assess and meet applicable breach and serious-incident notification obligations across relevant regimes (for example GDPR, HIPAA, and EU AI Act Article 73). Follow with unlearning, retraining, or withdrawal, vector and cache cleanup, vendor notice, and a persistent-memory audit.

## Example Attack Scenarios

#### *Scenario #1*

Divergence prompts make a production model emit memorized PII, URLs, and live credentials at scale, triggering GDPR Article 33 notification.

#### *Scenario #2*

A shared-inference-state defect leaks one user's medical-letter prompt into another user's reasoning trace. HIPAA 60-day notification applies.

### *Scenario #3*

Extended-thinking traces logged verbatim to a shared APM project expose retrieved PII to hundreds of engineers while the answer stays sanitized.

### *Scenario #4*

Prompt injection makes a support bot print its system prompt and an embedded vendor API key.

### *Scenario #5*

A shared legal RAG index crosses firm boundaries, synthesizing one client's privileged strategy into another's answer, an attorney-client waiver event.

### *Scenario #6*

A leaked "embeddings-only" vector backup is reclassified as a source-document breach after inversion, restarting the 72-hour clock.

### *Scenario #7*

Whisper Leak topic inference on encrypted streaming identifies users querying medical, legal, or political topics without decryption.

### *Scenario #8*

Membership inference against a clinical fine-tune identifies training-set patients at high AUC without extracting any record, a HIPAA-reportable determination.

### *Scenario #9*

A model summarizes PII hidden beneath a black-rectangle PDF redaction layer rendered over unmodified text.

### *Scenario #10*

An injected "diagnostic check" makes a code runtime encode spreadsheet content into DNS queries while the visible statistical summary stays benign.

# LLM03:2026 Excessive Agency

## Description

An LLM-based system is often granted a degree of agency by its developer: the ability to call functions or interface with other systems via tools (also called extensions, plugins, or skills by different vendors) to undertake actions in response to a prompt. An LLM agent may also select which tool to invoke dynamically, based on the input prompt or prior LLM output. Agent-based systems will typically make repeated calls to an LLM using output from previous invocations to ground and direct subsequent invocations.

Excessive Agency is the vulnerability that enables damaging actions to be performed in response to unexpected, ambiguous or manipulated outputs from an LLM, regardless of what is causing the LLM to malfunction. Common triggers include:

- hallucination/confabulation caused by poorly-engineered benign prompts, or just a poorly-performing/misaligned model,
- direct/indirect prompt injection from a malicious user, an earlier invocation of a malicious/compromised tool, or (in multi-agent/collaborative systems) a malicious/compromised peer agent.

The root cause of Excessive Agency is typically one or more of:

- excessive functionality,
- excessive permissions,
- 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. Within the context of agentic systems, Excessive Agency can manifest as ASI02: Tool Misuse & Exploitation, ASI03: Identity & Privilege Abuse and ASI08: Cascading Failures.

Note: Excessive Agency differs from Improper Output Handling which is concerned with insufficient scrutiny of LLM outputs. Sanitization of model inputs and outputs is not a root control for Excessive Agency and is covered by LLM01:2026 Prompt Injection for inputs and LLM10:2026 Improper Output Handling for outputs.

## Common Examples of Risk

**1. Excessive Functionality**
An LLM agent has access to tools 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 third-party tool they choose to use also includes the ability to modify and delete documents.

**2. Excessive Functionality**
A tool may have been trialed during a development phase and dropped in favor of a better alternative, but the original tool remains available to the LLM agent.

**3. Excessive Functionality**
An LLM tool 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 tool to run one specific shell command fails to properly prevent other shell commands from being executed.

**4. Excessive Permissions**
An LLM tool has permissions on downstream systems that are not needed for the intended operation of the application. E.g., a tool 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.

**5. Excessive Permissions**
An LLM tool that is designed to perform operations in the context of an individual user accesses downstream systems with a generic high-privileged identity. E.g., a tool to read the current user's document store connects to the document repository with a privileged account that has access to files belonging to all users.

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

## Prevention and Mitigation Strategies

The following actions can prevent Excessive Agency:

**1. Minimize tools**
Limit the tools that LLM agents are allowed to call to only the minimum necessary. For example, if an LLM-based system does not require the ability to fetch the contents of a URL then such a tool should not be offered to the LLM agent.

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

**3. Avoid open-ended tools**
Avoid the use of open-ended tools where possible (e.g., run a shell command, fetch a URL, etc.) and use 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 tool 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 specific file-writing tool that only implements that specific functionality. Tools should define a strict schema for any input parameters, and validate contents prior to use.

**4. Minimize tool permissions**
Limit the permissions that LLM tools are granted to other systems to 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 tool uses to connect to the database.

**5. Execute tools in user's context**
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 tool that reads a user's code repo should require the user to authenticate via OAuth and with the minimum scope required. In delegated or multi-agent workflows, preserve the original user context and authorization scope across chained tool or agent calls, rather than relying only on the permissions of the calling agent or service identity.

**6. Require user approval**
Use human-in-the-loop control to require a human to approve high-impact actions before they are taken. This may be implemented in a downstream system (outside the scope of the LLM application) or within the LLM 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 tool that implements the 'post' operation.

**7. Complete mediation**
Implement authorization in logic rather than relying on an LLM to decide if an action is allowed or not. Enforce the complete mediation principle so that all requests made to downstream systems are validated against security policies by the tool, by an independent pre-execution policy decision point between the tool and the downstream system, or by the downstream system itself. Such policies can help manage cases

where an agent's nominally-permitted action is contextually unsafe. A graduated enforcement policy (audit, warn, block, escalate) permits low-consequence or easily reversible actions to auto-approve, while high-consequence or irreversible ones route to human review. For example, a customer service chatbot can auto-process a refund as store credit (which is recoverable), while an irreversible action such as an external payout routes to human approval.

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

**8. Monitor tool use**
Log and monitor the activity of LLM tools and downstream systems to identify where undesirable actions are taking place, and respond accordingly.

**9. Rate limiting**
Establish thresholds around the invocation of tools and implement circuit breakers that halt, rate-limit or escalate for human review if those thresholds are exceeded. Simple thresholds could be based on the number of invocations, whereas context-aware thresholds could be based on the cumulative value of an input parameter to a tool.

## Example Attack Scenarios

### *Scenario #1: Hijacked Email Assistant*

An LLM-based personal assistant app is granted access to an individual's mailbox via a tool in order to summarize the content of incoming emails. To achieve this functionality, the tool requires the ability to read messages, but the tool the system developer has chosen to use also contains functions for sending messages. The app is also vulnerable to an indirect prompt injection attack, whereby a maliciously-crafted incoming email tricks the LLM into commanding the agent to scan the user's inbox for sensitive information and forward it to the attacker's email address. This could be avoided by:

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

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

# LLM04:2026 Supply Chain

## Description

LLM supply chains are susceptible to vulnerabilities that affect the integrity of training data, models, adapters, conversion pipelines, and deployment platforms, resulting in biased outputs, security breaches, or system failures. While traditional software vulnerabilities focus on code flaws and dependencies, in ML the risks extend to third-party pre-trained models, datasets, and model artifacts, which can be manipulated through tampering, poisoning, or malicious artifact replacement.

Creating LLMs is specialized work that depends on third-party models, datasets, and reusable adapters built with parameter-efficient fine-tuning (PEFT) techniques such as LoRA (Low-Rank Adaptation) and shared on platforms like Hugging Face. The supply chain now includes model artifacts, provenance, and conversion/merge workflows as first-class attack surfaces, and on-device LLMs widen it further.

Some of the risks covered here are also discussed in LLM05:2026 Data and Model Poisoning. This entry focuses on their supply-chain aspect. Supply-chain risks specific to agentic applications, including MCP servers and tool registries, are covered by ASI04 Agentic Supply Chain Vulnerabilities in the OWASP Top 10 for Agentic Applications (OWASP GenAI Security Project, 2026), and MITRE ATLAS catalogs the corresponding adversary techniques under AML.T0010 AI Supply Chain Compromise (MITRE, n.d.).

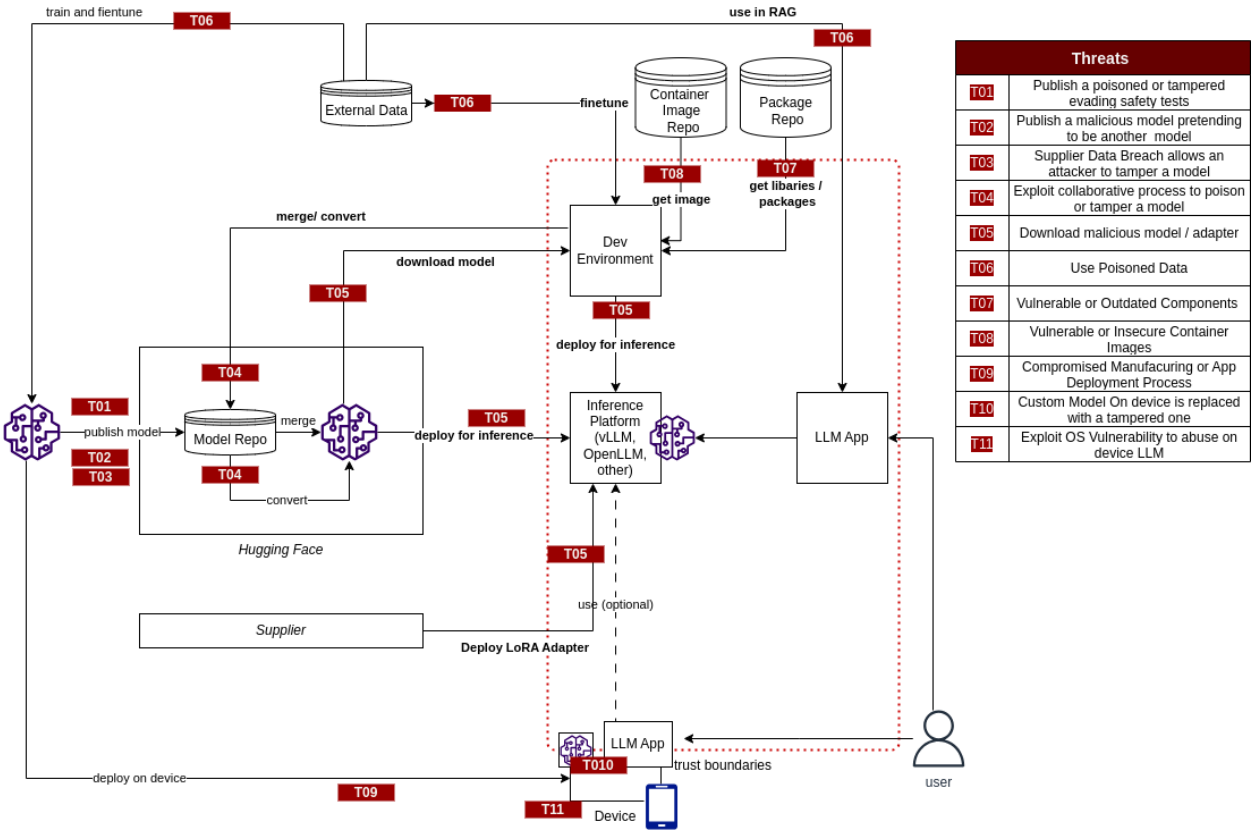



*Figure 3: A simple LLM supply-chain threat model illustrates supply chain attack surfaces.*

# Common Examples of Risk

**1. Vulnerable or Outdated Third-Party Components and Models**

Outdated or deprecated components, including packages, serving frameworks, and models themselves, can be exploited to compromise LLM applications. This is similar to A06:2021 Vulnerable and Outdated Components with increased risks when components are used during model development, fine-tuning, or inference. LLM coding assistants add a new variant: they hallucinate plausible but nonexistent package names at scale (Spracklen et al., 2025), which attackers register in advance ("slopsquatting") so that unverified AI-suggested dependencies resolve to malicious code.

**2. Licensing Risks**

AI development involves diverse software and dataset licenses that impose different usage, distribution, and commercialization requirements. If not properly managed, they create legal and compliance exposure.

**3. Vulnerable or Tampered Pre-Trained Models**
Model artifacts are difficult to inspect comprehensively, and static analysis alone cannot establish behavioral safety. A pre-trained model can contain hidden biases or backdoors introduced through poisoned datasets or direct tampering. Migrating away from unsafe serialization formats such as Python pickle, which can execute arbitrary code on load, reduces but does not eliminate this risk: a backdoor can be embedded directly in a model's computational graph and persist in formats widely considered safe, such as ONNX, and a crafted model file can exploit memory-corruption bugs in a format's native parser, as shown by heap overflows in llama.cpp's GGUF parsing (CVE-2024-23496) (Cisco Talos, 2024).

**4. Weak Provenance and Unsigned Model Artifacts**
Published models carry no strong provenance assurances: Model Cards document a model but do not prove its origin, and a compromised or lookalike supplier account can publish a malicious model under a trusted name. When models, adapters, datasets, fine-tuned checkpoints, and conversion outputs are not signed or hash-pinned, an attacker can replace or silently alter artifacts in transit, in storage, or at the promotion boundary where automated pipelines accept artifacts into trusted environments, especially when pipelines resolve artifacts by a mutable reference (for example, a `latest` tag) instead of an immutable digest.

**5. Vulnerable Adapters and Compromised Conversion, Merge, and Quantization Workflows**
LoRA adapters make fine-tuning modular and efficient, but a malicious adapter can compromise the integrity of the pre-trained base model, whether in collaborative model merge environments or on inference platforms that download and apply adapters to a deployed model. Model conversion and merge services can introduce malicious changes during transformation between formats, bypassing review controls. Quantization is a related transformation risk: model weights can be crafted so the full-precision model evaluates benignly while the quantized artifact exhibits attacker-chosen behavior (Egashira et al., 2025), so full-precision assurances do not transfer to the deployed quantized artifact.

**6. On-Device LLM Supply-Chain Vulnerabilities**
LLMs shipped on devices add compromised manufacturing processes, exploitation of device OS or firmware vulnerabilities, and re-packaged applications with tampered models to the attack surface, making device integrity and firmware trust part of the LLM supply chain.

**7. Unclear T&Cs and Data Privacy Policies**
Unclear terms and conditions (T&Cs) and data privacy policies of model operators can lead to sensitive application data being used for model training and subsequent exposure, and may create copyright risk from material provided by the model supplier.

## Prevention and Mitigation Strategies

1. Carefully vet data sources and suppliers, including T&Cs and privacy policies, and only use trusted suppliers. Regularly review and audit supplier security and access, and re-assess on changes in their security posture or T&Cs.
2. Apply the mitigations in the OWASP Top Ten's A06:2021 Vulnerable and Outdated Components: vulnerability scanning, management, and a patching policy that keeps the application on maintained versions of components, APIs, and underlying models. Apply the same controls to development environments with access to sensitive data, and verify that AI-suggested dependencies exist and are the intended package before adopting them.
3. Apply AI red teaming and evaluations when selecting third-party models, focused on the use cases in scope, and continue in production with anomaly detection and adversarial robustness testing in MLOps and LLM pipelines to detect tampering and poisoning.
4. Maintain an up-to-date, signed inventory of components using a Software Bill of Materials (SBOM), extended to models, adapters, and datasets with AI BOMs (AIBOMs) and ML SBOMs, evaluating options such as the OWASP CycloneDX ML-BOM (OWASP CycloneDX, n.d.) and the OWASP AIBOM project (OWASP GenAI Security Project, 2025). Track licenses in the same inventory and audit it regularly for compliance and transparency.
5. Only use models from verifiable sources and compensate for weak provenance with third-party integrity checks, signing, and file hashes. Cryptographic model signing backed by a transparency log (for example, the OpenSSF Model Signing project and Sigstore) binds a model artifact to a signer identity (Open Source Security Foundation, 2025). Reproducible builds are not guaranteed for model training, and the Coalition for Secure AI (CoSAI) provides a maturity model for signing ML artifacts (OASIS Open, 2025). Signing proves integrity and origin, not safety (a validly signed model from a compromised or trusted-but-malicious supplier can still be backdoored), so combine it with immutable artifact references, provenance policy, policy-based release gates (for example, SLSA) (Open Source Security Foundation, n.d.), behavioral evaluation, and continuous validation of upstream model integrity. Similarly, use code signing for externally supplied code.
6. Strictly monitor and audit collaborative model development environments, and treat model conversion and merge services as high-risk promotion points.
7. Encrypt models deployed at the edge with integrity checks, use vendor attestation APIs to prevent tampered apps and models, and reject unrecognized firmware and untrusted device states.

## Example Attack Scenarios

### *Scenario #1: Compromised Packages and Serving Frameworks*

A compromised dependency reaches a model development or inference environment, as in the December 2022 PyTorch supply-chain attack, where a malicious `torchtriton` package on the PyPI registry shadowed the legitimate PyTorch-nightly dependency and exfiltrated data (PyTorch Foundation, 2022). The serving stack is part of the same attack surface: the ShadowRay attacks exploited the disputed CVE-2023-48022 (unauthenticated dashboards on production Ray servers) in the wild, later escalating into a self-propagating botnet across exposed clusters (Lumelsky & Elbaz, 2025), and in Ollama, CVE-2024-37032 allowed remote code execution through a malicious model manifest pulled from a registry (Wiz, 2024).

### *Scenario #2: Tampered Model Published to a Hub*

An attacker publishes a tampered model under a trusted-looking name, as demonstrated by the PoisonGPT proof-of-concept, in which a model with surgically modified parameters spread misinformation while evading detection by standard benchmark evaluation (Huynh & Hardouin, 2023). The same trust gap covers attacker fine-tunes that strip a popular model's safety features while preserving its performance on benign tasks (Zhan et al., 2023), any pre-trained model deployed without verification, and shared AI-as-a-service platforms, where researchers escaped an inference container via a malicious model to reach other customers' models and data (Tamari & Tzadik, 2024).

### *Scenario #3: Compromised Supplier LoRA Adapter*

An attacker infiltrates a third-party supplier and subtly alters a LoRA adapter that is later merged into a deployed LLM through a model-merge workflow. Once merged, the adapter provides a covert entry point into the system.

### *Scenario #4: Hijacked Model Conversion or Merge Service*

An attacker stages an attack through a model merge or format conversion service to compromise a publicly available model and inject malicious behavior, as shown by HiddenLayer's research on hijacking the Safetensors conversion bot on Hugging Face (HiddenLayer, 2024).

### *Scenario #5: Model Namespace Reuse*

An organization deploys a model from a public hub by referencing it solely by its `Author/ModelName` identifier. The original author deletes or transfers the account, freeing the namespace, and an attacker re-registers the same name and publishes a malicious model under the original path. Pipelines and managed model catalogs that resolve the model by name alone then pull the attacker's model, leading to remote code execution (Saraf & Balassiano, 2025).

### *Scenario #6: Scanner, Safe-Loader, and Safe-Format Bypass*

An organization gates third-party models behind a malware scanner and a loader option documented as safe against code execution. An attacker defeats both. Corrupted or compression-wrapped pickle streams execute their payload before the scanner reaches the broken byte, as in the nullifAI models found on Hugging Face (Zanki, 2025). Model scanners and safe-loader flags have had bypasses assigned CVEs, such as the PickleScan zero-days (Cohen, 2025) and `torch.load` with `weights_only` (CVE-2025-32434)(GitHub Security Advisories, 2025). A ShadowLogic-style backdoor in the computational graph (Wickens et al., 2024) of a "safe" format like ONNX attaches no executable code for a serialization scanner to flag. Treat scanners and safe-loader flags as defense-in-depth layers rather than guarantees, alongside provenance verification and patched loaders and parsers.

### *Scenario #7: Compromised Build Pipeline for Model Artifacts*

An attacker compromises the CI/CD pipeline an organization uses to fine-tune and publish models, through a malicious build-workflow dependency, a stolen artifact-registry credential, or cache poisoning, as in the Ultralytics attack, where GitHub Actions cache injection published trojanized PyPI releases of a flagship AI library (Python Package Index, 2024), including a compromised "fix" release. This is the same build-time substitution seen in classical incidents such as the xz-utils backdoor and the Codecov breach. Because the backdoored artifact is built and signed by the organization's own release infrastructure, it passes downstream provenance checks, internal attestation, and supply-chain scanners that only flag externally sourced components.

### *Scenario #8: Reverse-Engineered Mobile App*

An attacker reverse-engineers a mobile app to replace the on-device model with a tampered version that leads users to scam sites, distributing the re-packaged app through social engineering.

# LLM05:2026 Data and Model Poisoning

## Description

Data and Model Poisoning describes a class of attacks and failures where an adversary (or unsafe process) manipulates data or model artifacts to embed harmful behavior, bias, or exploitable weaknesses into an AI system. In modern GenAI environments, poisoning is not limited to "training data" in the traditional sense. It can occur anywhere data is ingested, transformed, retrieved, or reused, including during pre-training, fine-tuning, embedding creation, retrieval augmentation (RAG), and model distribution. The result is an AI system that may still appear functional but behaves in ways that undermine trust, safety, and security.

Data poisoning occurs when pre-training, fine-tuning, or embedding data is tampered with to introduce vulnerabilities, backdoors, or biases. This can happen intentionally (malicious poisoning) or unintentionally (poor data hygiene, contaminated sources). The manipulation compromises model integrity. The model learns the wrong patterns, internalizes malicious correlations, or is conditioned to behave incorrectly. The consequences include harmful outputs, impaired capabilities, and degraded reliability.

The key idea: poisoning targets the model's "learning process," not a single runtime bug. Unlike typical software vulnerabilities that can be patched by fixing code, poisoning can require data revalidation, retraining, model replacement, or pipeline redesign, which is expensive and operationally disruptive.

Poisoning can occur across multiple stages of the LLM lifecycle:

- **Pre-training:** Maliciously crafted or contaminated corpora cause the model to absorb harmful patterns, unsafe instructions, or skewed representations.
- **Fine-tuning:** Manipulated datasets introduce domain-specific failure modes or hidden triggers.
- **Embeddings and vectorization:** Poisoning targets stored vectors to influence retrieved content, resulting in steered answers or subtle misinformation.
- **Transfer learning / model reuse:** Compromised source models pass that compromise to downstream systems.
- **Continuous learning pipelines:** Automated ingestion without sufficient validation allows attackers to gradually shape model behavior.

The data poisoning surface expands because organizations increasingly rely on external datasets, RAG pipelines, shared model repositories, and agentic workflows. Models distributed through shared repositories can carry risks through bundled non-weight artifacts, including malicious deserialization (e.g., pickle files) and tampering of chat templates, tokenizer configs, LoRA/PEFT adapters, and quantization artifacts, any of which can execute harmful code or alter model behavior when loaded. Such backdoors may leave model behavior untouched until a trigger causes it to change, creating the opportunity for a model to become a sleeper agent.

In agentic deployments, poisoning risks extend to tool integrations, persistent memory stores, and RLHF feedback loops. These attack surfaces are covered in depth in the OWASP Top 10 for Agentic Applications.

This entry covers durable corruption of persistent data or model behavior. Prompt instructions delivered through retrieved content at inference time are covered by LLM01:2026 Prompt Injection, and attacks that exploit embedding geometry by LLM09:2026 Vector and Embedding Weaknesses.

## Common Examples of Risk

1. **Training and Fine-Tuning Data Poisoning:** Attackers inject biased or malicious content into datasets. A targeted variant deliberately erodes refusal behaviors while preserving general accuracy, making degradation undetectable through standard evaluation.
2. **Financial Model Data Poisoning:** Attackers inject mislabeled transaction data into fraud detection models, labeling fraud as legitimate. The model learns to ignore real threats, enabling fraud bypass and undermining trust in AI-driven financial systems.
3. **Open-Source Dataset Supply Chain Poisoning:** Attackers contribute malicious data to widely used datasets. Trigger phrases inserted into a shared dataset can propagate into the many downstream models that fine-tune on it, forcing costly retraining once the backdoor is discovered.
4. **Low-Volume High-Impact Backdoor Poisoning:** As few as 250 poisoned documents compromise models from 600M to 13B parameters regardless of dataset size (Souly et al., 2025). Strategic minimal manipulation is sufficient for significant impact.
5. **AI Recommendation / Memory Poisoning:** Attackers embed hidden instructions in web content to manipulate AI memory or recommendations without detection, demonstrating risks in agent-based systems and persistent memory.
6. **RAG Knowledge Base Poisoning:** A single optimized poisoned text injected per targeted query can override accurate content in a retrieval corpus, and the attack retains high success against paraphrasing, instructional-prevention, and detection-based defenses (Zhang et al., 2025).

7. **Agent / Multi-System Poisoning:** Poisoned inputs in multi-agent workflows influence behavior and data access across entire AI-driven ecosystems, not just individual models.
8. **Healthcare Model Poisoning:** Minimal poisoning of medical training data significantly alters model outputs while passing standard evaluations, creating unsafe recommendations in safety-critical domains.
9. **Malicious AI Models in Supply Chain:** Attackers distribute compromised models via public repositories with embedded backdoors. Organizations downloading these models unknowingly inherit hidden triggers or system compromise.

## Prevention and Mitigation Strategies

1. Track dataset and model lineage using SBOM/ML-BOM (e.g., CycloneDX), enforce signing and verification, and continuously validate data integrity across lifecycle stages.
2. Establish strict validation for all incoming data, vet third-party vendors, and compare outputs against trusted sources to detect bias or adversarial manipulation early.
3. Protect RAG systems by enforcing trust boundaries, filtering retrieved content, applying source scoring, and isolating system instructions from external data.
4. Use sandboxing and strict isolation controls to limit model interaction with unverified data, tools, or external systems.
5. Apply statistical and AI-based anomaly detection across training, embedding, and inference pipelines, and monitor training loss, outputs, and behavior for drift or anomalies against defined thresholds to detect subtle poisoning effects over time.
6. Use curated domain-specific datasets for fine-tuning to reduce exposure to untrusted data and cross-domain contamination.
7. Enforce least-privilege access, network segmentation, and strict data access controls to prevent unauthorized data injection.
8. Use data version control (e.g., DVC) to track dataset changes, maintain version history, and enable rollback and forensic analysis when poisoning is detected.
9. Control automated retraining and feedback loops by validating incoming data, requiring human oversight, and applying rate limits against gradual poisoning through manipulated preference signals.
10. Continuously red team models with adversarial inputs and trigger-based prompts to identify hidden backdoors. Do not assume safety alignment removes backdoors. Dedicated trigger-probing is required after every alignment cycle (Hubinger et al., 2024).

11. Implement grounding techniques with validation layers ensuring retrieved content is verified before influencing outputs.
12. Treat inference artifacts, including chat templates, tokenizer configs, LoRA/PEFT adapters, and quantization artifacts, as security-relevant code. Enforce signing, hash verification, diff checks, and static analysis before deployment.

## Example Attack Scenarios

### *Scenario #1*

An attacker inserts manipulated documents into an internal knowledge repository. Poisoned documents surface in responses, leading to incorrect recommendations, manipulated business decisions, or reputational damage.

### *Scenario #2*

An attacker embeds hidden instructions in webpages summarized by AI tools. When ingested into RAG or memory systems, the instructions bias the model to recommend specific products, enabling financial manipulation and loss of trust in AI outputs.

### *Scenario #3*

An attacker submits crafted inputs into an automated retraining feedback loop. No infrastructure access is required, only standard user-interface access. The result is slow model drift toward degraded accuracy, biased outputs, or unsafe recommendations.

### *Scenario #4*

A malicious insider injects mislabeled transaction data into a training dataset. The model fails to detect fraud, resulting in financial losses, compliance violations, and regulatory breach.

### *Scenario #5*

An attacker uploads poisoned pre-trained weights to a public repository. Standard safety training fails to remove embedded backdoors (Hubinger et al., 2024). Organizations face system compromise, data leakage, or targeted manipulation at scale.

### *Scenario #6*

An attacker modifies a model's chat template (for example in a GGUF package or tokenizer config) with trigger-activated conditional instructions. Redistributed via a public hub, the model behaves normally under benign inputs. Validated across 18 models and 4 inference runtimes: factual accuracy drops from 90% to 15% under trigger conditions, and URL emission exceeds an 80% success rate (Fogel et al., 2026).

### *Scenario #7*

A developer loads a third-party model using unsafe serialization (e.g., pickle). Embedded malicious code executes during loading, enabling host compromise, lateral movement, and infrastructure breach.

### *Scenario #8*

In a shared AI environment, one tenant injects adversarial data into shared embeddings or memory layers, influencing other tenants' responses and causing cross-tenant contamination and privacy risk.

### *Scenario #9*

An attacker injects malicious instructions into an AI agent's persistent memory over multiple sessions. The agent begins prioritizing attacker-controlled logic, resulting in long-term workflow manipulation and hidden persistence.

# LLM06:2026 Unbounded Consumption

## Description

Unbounded Consumption occurs when an LLM application allows excessive and uncontrolled inferences, enabling attackers to disrupt service availability, inflict unsustainable financial costs, or steal intellectual property through model cloning, all by exploiting a common class of vulnerability: the absence of adequate controls over how resources are consumed.

The high computational demands of LLMs, particularly in cloud and pay-per-token environments, make them inherently susceptible to resource exploitation and unauthorized usage. A defining characteristic of this threat is cost asymmetry. Attackers can trigger disproportionately expensive computation at negligible cost to themselves, whether through crafted prompts, stolen credentials, or manipulated workflows.

This risk is compounded by the growing adoption of extended-thinking and reasoning models with large or insufficiently constrained output budgets, multimodal models that substantially increase per-request compute costs, agentic architectures and tool-use protocols (such as MCP) that amplify a single request into cascading downstream operations, and shared inference infrastructure that introduces new supply-chain attack surfaces. Traditional request-rate limiting alone is no longer sufficient. Effective defense demands token-aware cost controls, hard spending caps, agent-level circuit breakers, and continuous cost-attribution monitoring.

## Common Examples of Risk

**1. Variable-Length Input Flood and Output Explosion**
Attackers can overload the LLM with numerous inputs of varying lengths, exploiting processing inefficiencies. This can deplete resources and potentially render the system unresponsive, significantly impacting service availability. This also includes output explosion via fine-tuning poisoning, where a single malicious training sample breaks the model's end-of-sequence behavior and pushes output to the maximum length on every request (Gao et al., 2024).

**2. Denial of Wallet (DoW)**
By initiating a high volume of operations, attackers exploit the cost-per-use model of cloud-based AI services, leading to unsustainable financial burdens on the provider and risking financial ruin.

**3. Large-Context Abuse**
Repeated near-limit requests, context accumulation, and application-side rechunking consume disproportionate compute and memory. Many APIs reject inputs that exceed the context window outright, so the durable risk comes from requests that stay just within limits while inflating per-request cost.

**4. Reasoning-Loop and Thinking-Token Exhaustion**
Attackers craft short, benign-looking prompts that result in resource exhaustion by forcing extended-thinking models into prolonged or non-terminating reasoning loops, consuming massive thinking-token budgets while bypassing input-size filters (Li et al., 2025). Because these prompts are small and appear legitimate, standard input validation provides no protection.

**5. Adversarial Inputs Optimized for Resource Overconsumption**
Attackers use optimization techniques to craft inputs that maximize computational cost. This is distinct from simply asking the model to perform a resource-intensive task and includes sponge examples (Shumailov et al., 2020) and adversarial visual perturbations. It covers optimization of adversarial input with gradient-based and gradient-free techniques. Unlike reasoning-loop attacks, these require explicit optimization over the input space rather than prompt design alone.

**6. Multimodal Inputs and Outputs**
Multimodal models convert images, audio, and video into large numbers of tokens, so a single request can cost substantially more than a comparable text-only request. The exact overhead varies with the model, provider, resolution, media duration, and preprocessing pipeline.

**7. Model Extraction and Distillation Theft**
Attackers query the model API with crafted inputs to collect sufficient outputs to replicate a partial model or fine-tune a functional equivalent. Exposure of logits and log-probabilities significantly accelerates extraction (Carlini et al., 2024). Side-channel extraction of model weights or architecture through timing or shared-infrastructure observation is covered by LLM02:2026 Sensitive Information Disclosure.

**8. Agent-Tool Interactions Flooding Model Resources**
Attackers can publish tools that overuse LLM resources by forcing an LLM-based application into recursive or infinite tool-calling loops. This could cause seemingly legitimate tool actions to result in financial burdens or compromise quality of service. When one tool call fans out into a much larger number of actions, the LLM may need to manage hundreds of calls spawned from a single task, driving token overuse.

**9. Inference Infrastructure Exploitation**
Attackers target vulnerabilities in LLM serving frameworks (vLLM, TensorRT-LLM, SGLang, Triton, Ollama) to crash services or exhaust model resources through unsafe deserialization flaws, special-token injection, and injected chat templates.

## Prevention and Mitigation Strategies

**1. Rate Limiting & Input Size Validation**
Apply rate limiting and user quotas to restrict the number of requests a single source entity can make in a given time period. Move beyond requests per second to enforce limits on tokens-per-minute, tokens-per-day, and estimated cost per request. Use pre-flight token estimation to reject requests before inference begins. This includes validation to ensure inputs do not exceed reasonable size limits.

**2. Hard Spending Caps**
Set non-overridable budget ceilings per API key, user, team, and cloud account. These must be enforcement mechanisms that halt inference when exceeded, rather than alerting thresholds that fast-accumulating workloads can outpace. Spending caps should also account for cost differences between modalities and tool protocols.

**3. Resource Allocation Management**
Monitor and manage resource allocation dynamically to prevent any single user or request from consuming excessive resources.

**4. Sandbox Techniques**
Restrict the LLM's access to network resources, internal services, and APIs. Constraining what the application can reach limits an attacker's ability to exfiltrate extracted model information or data to an external destination.

**5. Graceful Degradation**
Design the system to degrade gracefully under heavy load, maintaining partial functionality rather than complete failure.

**6. Limit Queued Actions and Scale Robustly**
Implement restrictions on the number of queued actions and total actions, while incorporating dynamic scaling and load balancing to handle varying demands and ensure consistent system performance.

**7. Scan for Adversarial Perturbations**
Scan model inputs, particularly visual inputs to LVLMs (large visual language models), for evidence of adversarial perturbations that could cause model resource overconsumption.

**8. Detect Resource-Intensive Tool Interactions**
Monitor agent-tool interactions to identify if a particular session appears to be causing a recursive or resource-intensive action without a clear end state. Establish baselines of normal tool behavior in order to detect if a particular tool is deviating from standard token consumption patterns.

**9. Agentic Circuit Breakers**
Enforce step limits, recursion depth limits, time limits, and per-run cost ceilings on all agent executions. Use state hashing to detect recursive loops.

**10. Inference Infrastructure Hardening**
Keep serving frameworks updated. Disable unsafe deserialization, restrict special-token passthrough, and enforce authentication on all inference endpoints.

## Example Attack Scenarios

### *Scenario #1: Uncontrolled Input Size*

An attacker submits an unusually large input to an LLM application that processes text data, resulting in excessive memory usage and CPU load, potentially crashing the system or significantly slowing down the service.

### *Scenario #2: Repeated Requests*

An attacker transmits a high volume of requests to the LLM API, causing excessive consumption of computational resources and making the service unavailable to legitimate users.

### *Scenario #3: Resource-Intensive Queries*

An attacker crafts specific inputs designed to trigger the LLM's most computationally expensive processes, leading to prolonged GPU usage and potential system failure.

### *Scenario #4: Denial of Wallet (DoW)*

An attacker generates excessive operations to exploit the pay-per-use model of cloud-based AI services, causing unsustainable costs for the service provider.

### *Scenario #5: Functional Model Replication*

An attacker uses the LLM's API to generate synthetic training data and fine-tunes another model, creating a functional equivalent and bypassing traditional model extraction limitations.

### *Scenario #6: Perturbations in LVLM Image Input*

An attacker crafts adversarial image inputs that include perturbations optimized to cause an LVLM to overconsume tokens in its output (Gao et al., 2025).

### *Scenario #7: Multi-turn Tool Calling Loops and Tool Call Fan-Out*

The attacker can publish a malicious tool (e.g. via a Claude Skill on an open-source repository) that instructs an agent to perform recursive cyclical tasks, or tasks that require a large number of tool calls. Developers incorporating that tool into their agents then risk causing excessive token consumption and service instability.

### *Scenario #8: Growing LLM Context in Agentic Sessions*

An attacker or a benign user maintains an open agentic session, gradually injecting content so that each inference re-processes the full accumulated context. Per-turn cost climbs as the context grows, from roughly $0.001 on the first turn to about $0.50 by turn 100. No single request triggers rate limits because each stays individually within budget, yet the aggregate across many concurrent or long-lived sessions reaches hundreds of dollars.

# LLM07:2026 Misinformation

## Description

Misinformation occurs when an LLM or LLM-enabled application produces incorrect, incomplete, unsupported, or misleading information that appears credible enough to influence a human decision, an automated workflow, or an agent action. The core risk is that the incorrect output is trusted and acted upon.

In modern systems, model outputs drive tool calls, generate code, infer system state, authorize actions, and coordinate across agents. This makes misinformation a system-level failure that can lead to financial loss, security incidents, safety risks, or operational disruption.

In agentic systems, misinformation often manifests as incorrect state, reasoning, or evidence that is consumed by downstream components, leading directly to unintended actions.

Misinformation can arise from hallucination, incomplete or stale context, weak grounding, ambiguous prompts, biased or corrupted data, misleading summaries, or unvalidated tool outputs. It can also be deliberately induced by attackers. Where the root cause is prompt injection, poisoning, or supply chain compromise, those risks should be referenced separately. The execution and handling of unsafe generated code is covered by LLM10:2026 Improper Output Handling, and the registration of hallucinated package names as a supply-chain vector is covered by LLM04:2026 Supply Chain. This entry focuses on the resulting failure mode: a false representation that drives a harmful decision or action.

Overreliance remains a key factor. Humans and systems often treat fluent, confident, or well-structured outputs as authoritative. In agentic architectures, this overreliance is frequently embedded in system design.

## Common Examples of Risk

1. Unsupported or False Decision Support: Incorrect or unsupported information influences business, legal, healthcare, financial, or operational decisions.
2. Incorrect State Inference in Workflows: An LLM infers that a condition has been met when it has not, triggering unintended actions.
3. Incorrect or Fabricated Code and Dependencies: The model produces incorrect code recommendations or references non-existent (hallucinated) packages (Spracklen et al., 2025).

4. Misleading Summaries and Critical Omissions: Summaries omit key constraints, exceptions, timestamps, or risks.
5. Adversarially Induced Misinformation: Attackers craft inputs that cause false claims or omission of critical facts.
6. Cross-Agent Misinformation Propagation: Incorrect outputs propagate across agents and workflows.
7. Forged or Misattributed Evidence: Fabricated or manipulated content is presented as authoritative evidence.

## Prevention and Mitigation Strategies

1. Ground Claims Before Action: Require outputs to be grounded in authoritative and current sources.
2. Implement Claim-Check-Act Patterns: Separate generation from execution and verify claims before acting.
3. Validate Tool Calls: Check arguments, authorization, preconditions, and current state before execution.
4. Use Verification Signals (Not Just Confidence): Incorporate groundedness and consistency checks.
5. Enforce Runtime Verification for High-Impact Actions: Introduce approval workflows and system checks.
6. Detect and Prevent Omission Failures: Require structured outputs with mandatory fields.
7. Limit Blast Radius: Apply least privilege, sandboxing, and rate limits.
8. Monitor and Test for Misinformation: Log claims, evidence, and outcomes, and test adversarial scenarios.
9. Calibrate Human and System Trust: Distinguish verified facts from assumptions.
10. Adversarial Evaluation and Continuous Testing: Regularly test workflows against misleading scenarios.

## Example Attack Scenarios

### *Scenario #1: Hallucinated Dependency Recommendation*

A coding assistant recommends a plausible but non-existent package, which an attacker has pre-registered under the hallucinated name, so a developer who trusts the suggestion installs attacker-controlled code (Spracklen et al., 2025).

### *Scenario #2: Incorrect Policy Decision by Agent*

A customer-service agent misreads a policy and approves a refund that violates the terms, resulting in financial loss.

### *Scenario #3: Omission in Safety-Critical Summary*

A clinical summary omits a drug contraindication, and a clinician acts on the incomplete recommendation.

### *Scenario #4: Adversarially Induced False Reasoning*

An attacker seeds a support forum with false remediation steps that a troubleshooting agent retrieves and repeats as a trusted recommendation.

### *Scenario #5: False Alert Triggers Automated Response*

A security agent misclassifies normal traffic as an intrusion and automatically blocks a production network segment, causing an outage.

### *Scenario #6: Cross-Agent Trust Failure*

A retrieval agent reports a customer as identity-verified when it is not, and a downstream payment agent trusts that state and releases funds.

### *Scenario #7: Fabricated Task Completion*

An agent reports that a nightly database backup completed when it never ran, and a later restore fails because no backup exists.

# LLM08:2026 Hidden Context Exposure

## Description

Hidden Context Exposure is the unauthorized extraction, inference, or reconstruction of hidden, non-user-facing system instructions or operational context placed in a model's context. It becomes security-relevant when that hidden context contains or reveals secrets, policy logic, tools, trust boundaries, workflow criteria, proprietary behavior, or other sensitive implementation details that materially increase attacker capability.

In an LLM application, hidden context typically includes the system prompt, developer instructions, retrieved policy text (from RAG knowledge bases, configuration stores, or user-profile services), the schemas of tools and functions the application exposes to the model, and other rules, directives, and materials the application assembles into the model's context window. The common thread is that this hidden context is not intended to be visible to end users but is accessible to the model.

Practitioners should design under the assumption that hidden context is discoverable and that any contents of the context should not be considered a secret. Application developers should ensure that disclosure of hidden context has little or no direct security impact. Sensitive data such as credentials, connection strings, and tokens should not be embedded in it, nor should hidden context be solely relied upon as a security boundary for authorization, privilege separation, policy enforcement, or content filtering.

Severity tracks what is placed in hidden context and how the application relies on it. Findings range from **informational** (no secrets, no security-relevant logic, no reliance on confidentiality) through **medium** (internal rules, filtering criteria, role descriptions, or workflow logic that meaningfully aids an attacker but does not gate critical decisions) to **high** (embedded credentials or tokens, or reliance on hidden-context secrecy for authorization or content policy) and **critical** (where disclosure chains to remote code execution, broad data exfiltration, or privilege escalation in a connected system).

While Hidden Context Exposure introduces risks on its own, it also frequently amplifies risks in adjacent categories:

- Disclosed rules or logic enable more targeted prompt injection (LLM01:2026).
- Embedded credentials constitute sensitive information disclosure (LLM02:2026).
- Revealed tool permissions and schemas expand the surface for excessive agency (LLM03:2026).
- Leaked output-formatting rules can facilitate improper output handling (LLM10:2026).

In summary, LLM08 covers the foundational risk that hidden LLM control context is exposed, inferred, or reconstructed in a way that materially increases attacker capability. LLM08 does not cover:

- The leakage of regulated user or training data (LLM02 Sensitive Information Disclosure).
- The agentic amplifications of this risk, e.g., persistent memory, inter-agent channels, tool configuration persistence, and multi-step agent compromise (the OWASP Top 10 for Agentic Applications).
- Generic application-security concerns inherited by LLM-integrated systems, e.g., server-side log leakage, client-side bundle inspection, and infrastructure-layer side channels.

## Common Examples of Risk

**1. Exposure of Sensitive Functionality, Tool and Function Schemas**
The system prompt or hidden context of the application may reveal sensitive information or functionality that is intended to be kept confidential. This may include sensitive system architecture, available tools and functions, API keys, database credentials, or user tokens. While exposure of these would likely cause harm, the real risk is that sensitive credentials are placed in the hidden context in the first place.

**2. Exposure of Behavioral Control Logic**
The context of the application includes information on internal decision-making processes that should be kept confidential. This information allows attackers to gain insights into how the application works, which they could use to exploit weaknesses or bypass controls in the application.

**3. Reverse Engineering of Safety and Refusal Mechanisms**
System prompts can define the conditions under which a model should refuse or filter content. When these instructions are exposed through system prompt leakage, attackers gain visibility into the rules that govern refusal behavior. While a typical user may only observe responses such as "Sorry, I can't do that," leakage reveals the underlying triggers, conditions, and exceptions that led to that decision. This allows attackers to craft inputs that intentionally avoid known refusal patterns or exploit gaps in enforcement, increasing the likelihood of obtaining responses that would otherwise be restricted.

**4. Disclosure of Permissions and User Roles**
Instruction context could include directives or information related to authorization and permissions. For example, a tool description provided through an internal-facing MCP server may indicate that a user must have the developer role in order to use it, or that a user with a certain role can access a list of documents to search with RAG. The disclosure of such information could invite other types of probing through directed conversation and prompt injection (LLM01:2026) and could potentially reveal additional sensitive information (LLM02:2026).

**5. Exposure of Output Structure and Formatting Rules**
System prompts frequently define how responses should be structured, including required formats such as JSON schemas, templates, or validation constraints. When these instructions are exposed, attackers gain insight into how outputs are constructed and what assumptions downstream systems rely on. This knowledge can be used to generate responses that conform to expected formats while embedding unintended or manipulated values, potentially leading to incorrect parsing or unintended system behavior.

## Prevention and Mitigation Strategies

**1. Do Not Put Sensitive Data in Hidden Context**
Do not embed credentials, secrets, or security-critical configuration directly in the system prompts or hidden context. Assume all context available to the LLM could also be available to users. Instead, externalize such information to the systems that the model does not directly access and avoid letting the model handle sensitive data itself.

**2. Use Deterministic Methods and Guardrails for Validation and Behavior Control**
Because LLMs can be vulnerable to attacks such as prompt injection, hidden context should not be relied on as the primary mechanism for controlling model behavior. Specialized fine-tuning or further training of a model may decrease the risk of disclosure, though it provides no consistent guarantee and may have other unintended consequences. Enforce critical behaviors through independent and deterministic systems outside the model. For example, harmful content detection and prevention should be handled by external safeguards rather than by instructions embedded in hidden context.

**3. Enforce Authorization and Access Control Independently from the LLM**
Critical controls such as privilege separation, authorization bounds checks, and similar must not be delegated to the LLM, whether through the system prompt or another mechanism. These controls should be enforced in a deterministic and auditable manner, which LLMs are not well suited to provide. Where tasks require different levels of access, separate them by authorization context and grant each only the privileges it requires.

## Example Attack Scenarios

### *Scenario #1: Credential Leakage via System Prompt*

An LLM has a system prompt that contains a set of credentials used for a tool that it has been given access to. The system prompt is leaked to an attacker, who is then able to use these credentials for other purposes.

### *Scenario #2: Tool Schema via Context Extraction*

An attacker extracts the hidden context that contains the tool list and parameter schemas through conversational probing and uses the information to craft inputs that steer the application toward specific tool calls. No credential is disclosed and no policy is overtly bypassed, but the attacker now has concrete targets for subsequent prompt injection attempts and reconnaissance for downstream action chaining.

### *Scenario #3 Bypassing Restrictions via Guardrail Disclosures*

An LLM has a system prompt prohibiting the generation of offensive content, external links, and code execution. An attacker extracts this system prompt and uses the disclosed restrictions to craft a prompt injection attack that bypasses them.

# LLM09:2026 Vector and Embedding Weaknesses

## Description

Vector and embedding weaknesses present security risks in any LLM application that converts text, images, code, or audio into numerical representations and uses similarity search to decide what the model sees. Retrieval-Augmented Generation (RAG) is the most familiar case, but the same machinery underlies vector-backed agent memory, semantic caches, and deduplication pipelines. Whenever similarity search sits between a data source and the prompt, the embedding layer becomes part of the application's trust boundary.

These weaknesses are distinct from prompt injection. They exploit the geometry of the embedding space and the mechanics of similarity search rather than the model's instruction-following behavior. Many succeed even when the retrieved content carries no malicious instructions at all. A useful frame: poisoning makes the system wrong, inversion makes it leak, jamming makes it silent, and access-control failure makes it indiscriminate.

This entry covers attacks that depend on the embedding layer to succeed. Indirect prompt injection through retrieved content is covered in LLM01:2026 Prompt Injection, training-time poisoning of the embedding model in LLM05:2026 Data and Model Poisoning, serialization flaws in vector-store libraries in LLM04:2026 Supply Chain, and agent-memory attacks that do not rely on embedding geometry in ASI06:2026 Memory and Context Poisoning (OWASP Top 10 for Agentic Applications). Vectorless retrieval systems (BM25-only, LLM-native tree navigation) inherit the non-geometric risks but have no LLM09 attack surface.

## Common Examples of Risk

**1. Cross-Tenant Leakage via Shared Similarity Search**

In multi-tenant deployments, similarity search frequently runs across the full index before access control is applied at the application layer. An attacker can probe the index with crafted queries and infer the existence, topic, and approximate volume of other tenants' documents from result counts, score distributions, and timing, without ever seeing the documents. The attack succeeds even when every document is correctly tagged and every API call authenticated, because the access-control decision happens after the embedding-space search has run. Conventional auth flaws in vector-DB software, for example CVE-2025-64513 (Milvus, forged sourceID header bypassing authentication, CVSS 9.3) and CVE-2025-69286 (RAGFlow, predictable

token derivation enabling account takeover, CVSS 9.3), are out of scope here but compound the geometric risk: because a vector-store leak is recoverable to source documents via inversion (Risk #2), an auth bug in a vector database carries higher consequence than the same bug in a document database or key-value store.

**2. Embedding Inversion**
Stored embeddings can be inverted to recover source text. Reported recovery rates range from roughly 50–70% of words from sentence embeddings to 92% exact reconstruction of short 32-token inputs with the Vec2Text method (Morris et al., 2023), which trains an inversion model per encoder. ZSInvert (C. Zhang et al., 2025) and Zero2Text (Kim et al., 2026) operate zero-shot with no encoder-specific training, work in cross-domain and black-box settings, and remain effective against differential-privacy noise added at storage. Operationally: vector-database backups, embeddings shipped to third-party services, and embeddings exposed through misconfigured cloud storage should be treated as equivalent to a leak of the underlying documents. Under GDPR and similar regimes, breach notification depends on risk to data subjects, and because modern embeddings can be inverted, that risk is real.

**3. Retrieval-Time Data Poisoning**
An attacker who can write to the corpus, via public scraping pipelines, file uploads, partner feeds, or compromised internal sources, can craft content whose embedding lands close to a target query. When a user submits that query, the attacker's content is retrieved and fed to the LLM as trusted context. Published attacks reliably achieve high success rates with a handful of poisoned documents, even in corpora of millions and against black-box systems. A successful attack requires two conditions simultaneously: the poisoned content must be retrieved (geometric) and must steer the response (generation). Defenders can intervene at either layer. MITRE ATLAS catalogs this class as AML.T0070 (RAG Poisoning) under the Persistence tactic. Training-time poisoning of the embedding model itself is LLM05:2026.

**4. Retrieval Jamming**
Attackers can take a RAG system off the air by inserting a "blocker" document, content engineered to be retrieved for a specific query and to cause the LLM to refuse to answer or claim it lacks information. Unlike poisoning, the blocker carries no malicious instructions. It exploits retrieval mechanics and LLM safety behavior. A single blocker document, generated through black-box optimization without access to the target embedding model or LLM, is sufficient. This is an availability attack on the retrieval layer.

**5. Membership Inference via Similarity Search**
The attacker wants to know whether a specific document (a medical record, a legal filing, an HR complaint) exists in the index, not what it says. If the application returns raw similarity scores or distances to the client, the index becomes a direct membership oracle with no LLM involved. If it returns only generated answers, the attacker can still infer membership by submitting partial documents or perturbed queries and analyzing responses. Membership information can itself be sensitive even when content remains protected.

**6. Semantic Cache and Deduplication Poisoning**

Semantic caches and near-duplicate detection use a cosine-similarity threshold to decide two pieces of content are "the same." An attacker who can craft content landing just above or below that threshold can poison a cache entry so it serves attacker text to all semantically equivalent queries, bypass deduplication with near-duplicates of poisoned content, or force legitimate new content to be silently dropped as a duplicate. Wu et al. (2026) demonstrate semantic cache poisoning end-to-end across AWS, Azure, and Alibaba deployments. Z. Zhang et al. (2026) show black-box key-collision attacks that use surrogate embedding models to engineer threshold-straddling vectors without access to the target encoder. All three failure modes depend on embedding-space geometry and are invisible to document-level controls.

**7. Multimodal Embedding Poisoning**
Cross-modal encoders such as CLIP and ColPali map images, audio, code, and text into one vector space. An attacker who can contribute non-text content can craft an image whose embedding sits close to a sensitive text query. When a user submits that query, the image is retrieved as trusted context. MM-PoisonRAG (Ha et al., 2025) and Poisoned-MRAG (Liu et al., 2025) demonstrate local and global poisoning across multimodal RAG pipelines. "One Pic is All it Takes" shows a single image suffices for targeted and universal VD-RAG poisoning. To a human reviewer the image appears unremarkable, and text-based content scanning does not catch it because the payload is not text.

## Prevention and Mitigation Strategies

**1. Permission and Access Control**
Enforce tenant scoping inside the index query, not as a post-retrieval filter, and validate it server-side. A client-supplied scope is a suggestion, not a control. Authenticate embedding and similarity-search endpoints as first-class APIs with per-tenant rate limits. For high-sensitivity workloads, use physically separated indexes per tenant or trust zone. Apply access control at the chunk level: a mostly-public document can contain a confidential paragraph.

**2. Data Validation, Source Authentication, and Provenance**
Normalize content before embedding: strip zero-width characters, white-on-white text, and Unicode homoglyphs at extraction. Track provenance for every embedding (source, ingestion time, trust tier, pipeline version) so compromised batches can be invalidated and audited. Apply human review to externally sourced content destined for sensitive indexes. Vet the embedding model itself. A backdoored encoder corrupts the geometry of everything ingested.

**3. Data Segregation by Trust Tier**
Mixed-trust content (external web data, internal confidential documents, partner data) must not share an index without hard isolation. Index-level segregation beats classification tags on a shared index because it removes the misconfiguration path.

**4. Anomaly Detection at Ingest and Retrieval**
Flag new vectors that sit unusually close to a wide range of common queries, the signature of retrieval-hijacking poisoning. Watch for queries returning too many high-similarity matches, unusual volume on embedding endpoints (a precursor to query-based inversion), and clusters growing faster than expected after ingest. Do not return raw similarity scores to clients, add noise and diversification at the retrieval-ranking layer, and rate-limit endpoints that could be queried as oracles. Cross-encoder re-ranking raises attack cost but does not replace provenance and ingest controls. Modern attacks target retrieval and ranking jointly.

**5. Storage Lifecycle Controls**
Delete embeddings within a bounded time when the source document is deleted, and verify with reconciliation audits. Treat vector-database backups at the same sensitivity tier as source documents. Encrypt embeddings at rest with keys managed separately from the application layer. When rotating the embedding model, re-embed the corpus rather than mixing old and new vectors. Heterogeneous embeddings create exploitable gaps in similarity behavior. Treat embedding-API keys as secrets. A leaked key gives an attacker query access to your exact encoder.

**6. Monitoring, Logging, and Incident Response**
Keep immutable logs of retrieval activity (tenant scope, query, returned IDs, similarity scores). Monitor for tenant-filter bypass attempts, cross-tenant retrieval anomalies, and abnormal embedding-API consumption. Update incident-response playbooks so "embeddings only" leaks are treated as source-data leaks for breach assessment and notification under GDPR Article 33 and analogous regimes.

## Example Attack Scenarios

### *Scenario #1: Embedding Similarity Attack on a Public Ingestion Pipeline*

A company's RAG system scrapes public documentation and forum posts on a schedule. An attacker publishes posts engineered so their embeddings land near specific internal queries, such as "what is our Q3 revenue projection." When an employee asks that question, the attacker's content is retrieved and fed to the LLM. The same text pasted into a chat would have no effect. The attack works only because the attacker can place content near a target query in embedding space.

### *Scenario #2: Cross-Tenant Inference in a Shared Vector Index*

A multi-tenant SaaS product uses one shared vector index with tenant filtering at the application layer. Tenant A submits probing queries. Similarity search runs across every embedding, including Tenant B's, before the filter is applied. A never sees B's documents, but timing differences, result counts, and score-distribution gaps reveal the existence and approximate topic of B's content. Over many queries, A builds a useful map of B's data. Real-world incidents in this category include CVE-2025-69286 (RAGFlow), where

predictable token generation enabled cross-account compromise in a widely deployed open-source RAG engine.

### *Scenario #3: Embedding Inversion from a Leaked Vector Store*

A cloud misconfiguration exposes a backup of a production vector database. The underlying documents, customer conversation logs containing PII, are encrypted separately and not exposed, so the incident is initially classified as low-severity: "only the embeddings leaked." The attacker runs a zero-shot inversion attack against the stolen embeddings and reconstructs a substantial fraction of the source content, including PII, without access to the original encoder. The incident is reclassified as equivalent to a source-document breach and notification obligations are reassessed. "Embeddings only" is not a safe-harbor classification.

# LLM10:2026 Improper Output Handling

## Description

Improper Output Handling refers specifically to insufficient validation, sanitization, and handling of the outputs generated by large language models before they are passed downstream to other components and systems. Since LLM-generated content can be controlled by prompt input, this behavior is similar to providing users indirect access to additional functionality. Improper Output Handling concerns unsafe use of model output before it is passed downstream, while LLM07:2026 Misinformation concerns output that is incorrect or misleading. Validation and sanitization of model inputs is covered by LLM01:2026 Prompt Injection. Successful exploitation of an Improper 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:

- Excessive application privileges granted to the LLM, enabling privilege escalation or remote code execution.
- Susceptibility to indirect prompt injection that can grant an attacker privileged access to a target user's environment.
- Unvalidated inputs from third-party tools.
- Missing context-specific output encoding (for example, HTML, JavaScript, SQL).
- Insufficient monitoring and logging of LLM outputs.
- Missing rate limiting or anomaly detection for LLM usage.
- Terminal, log, or IDE sinks that render model output without neutralizing control characters such as ANSI escape sequences.
- Client renderers (browser, chat UI, IDE, terminal) that automatically fetch external resources referenced in model output (for example, Markdown images, link previews, iframes), enabling exfiltration of context data through outbound requests.

## Common Examples of Risk

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.

3. LLM-generated SQL queries are executed without proper parameterization, leading to SQL injection.
4. LLM output is used to construct file paths without proper sanitization, potentially resulting in path traversal vulnerabilities.
5. LLM-generated content is used in email templates without proper escaping, potentially leading to phishing attacks.
6. LLM output containing ANSI escape sequences or other control characters is written to a terminal, log viewer, or IDE pane that interprets them, enabling visual spoofing, clipboard hijacking (for example, OSC 52), or exploitation of known terminal emulator vulnerabilities (Rehberger, 2024b).
7. The chat UI auto-renders Markdown images or link previews referenced in model output, allowing an attacker who controls part of the model context to exfiltrate conversation data via the image URL's hostname or query string (Rehberger, 2024a).

## Prevention and Mitigation Strategies

1. Treat the model as any other user, adopting a zero-trust approach, and apply proper input validation on responses coming from the model to backend functions.
2. Follow the OWASP ASVS (Application Security Verification Standard) guidelines to ensure effective input validation and sanitization (OWASP, n.d.).
3. Encode model output back to users to mitigate undesired code execution by JavaScript or Markdown. OWASP ASVS provides detailed guidance on output encoding.
4. Implement context-aware output encoding based on where the LLM output will be used (for example, HTML encoding for web content, JavaScript encoding for browser script contexts).
5. Use parameterized queries or prepared statements for all database operations involving LLM output.
6. Employ strict Content Security Policies (CSP) to mitigate the risk of XSS attacks from LLM-generated content.
7. Implement logging and monitoring systems to detect unusual patterns in LLM outputs that might indicate exploitation attempts.
8. Sanitize control characters (ANSI escape sequences, BEL, OSC, backspace, carriage return) and other non-printable bytes from model output before it is written to terminals, log files, or other interpreting sinks. Encode them visibly when they must be preserved.
9. In client renderers (chat UIs, IDEs, email clients, mobile apps), prevent model output from triggering automatic outbound requests to attacker-controlled endpoints. Disable auto-rendering of Markdown images, link previews, iframes, and similar elements by default. Where rendering is

required, restrict fetches to an explicit allowlist of origins or proxy them through a server-side fetcher that strips data-bearing query parameters.

## Example Attack Scenarios

### *Scenario #1*

An application uses an LLM tool to generate responses for a chatbot feature. The tool also offers a number of administrative functions accessible to another privileged LLM. The general-purpose LLM directly passes its response, without proper output validation, to the tool, causing the tool to shut down for maintenance.

### *Scenario #2*

A user utilizes a website summarizer tool powered by an 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, without any output validation or filtering, to an attacker-controlled server.

### *Scenario #3*

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

### *Scenario #4*

A web app uses an LLM to generate content from user text prompts without output sanitization. An attacker could submit a crafted prompt causing the LLM to return an unsanitized JavaScript payload, leading to XSS when rendered on a victim's browser. Insufficient validation and encoding of the model output enabled this attack.

### *Scenario #5*

An LLM is used to generate dynamic email templates for a marketing campaign. An attacker manipulates the LLM to include malicious JavaScript within the email content. If the application doesn't properly sanitize the LLM output, this could lead to XSS attacks on recipients who view the email in vulnerable email clients.

### *Scenario #6*

An application automatically compiles and deploys LLM-generated code without human review or security testing. Because the output is trusted and executed without validation, insecure code reaches production and is exploited.

# Appendix A: Related Framework Mappings

This appendix consolidates the mappings from the ten OWASP Top 10 for LLM Applications (2026) risk entries to nine external security frameworks and taxonomies. It replaces the per-entry *Related Frameworks and Taxonomies* sections, which have been removed in favor of this single, version-pinned reference.

### How to read this appendix

The **coverage matrix** shows, at a glance, which frameworks map to each risk. The **by-framework sections** give the specific element mappings and why they apply. Mappings are at each framework's coarse level (tactics, pillars/weaknesses, risk categories, control domains); every element is drawn from the pinned framework version listed in **Framework Sources & Versions**. Each cell shows the primary mappings plus the most relevant supporting ones.

**Legend:** ● primary (a central line of defense or description of the risk) · ○ supporting (contributes but is not the center of gravity) · — no applicable mapping.

## Coverage matrix

| Risk | ASI | DSGAI | ATLAS | ATT&CK | CWE | 600-1 | RMF | AICM | AIVSS |
|---|---|---|---|---|---|---|---|---|---|
| **LLM01** Prompt Injection | ● | ● | ● | ● | ● | ● | ○ | ● | ● |
| **LLM02** Sensitive Information Disclosure | ● | ● | ● | ● | ● | ● | ○ | ● | — |
| **LLM03** Excessive Agency | ● | ● | ● | ● | ● | ● | ○ | ● | ● |
| **LLM04** Supply Chain | ● | ● | ● | ● | ● | ● | ● | ● | ○ |

| Risk | ASI | DSGAI | ATLAS | ATT&CK | CWE | 600-1 | RMF | AICM | AIVSS |
|---|---|---|---|---|---|---|---|---|---|
| **LLM05** Data and Model Poisoning | ● | ● | ● | ○ | ● | ● | ○ | ● | ○ |
| **LLM06** Unbounded Consumption | ● | ● | ● | ● | ● | ● | ○ | ● | ○ |
| **LLM07** Misinformation | ● | ● | ● | ○ | ● | ● | ● | ● | ● |
| **LLM08** Hidden Context Exposure | ● | ○ | ● | ○ | ● | ● | ○ | ● | ○ |
| **LLM09** Vector and Embedding Weaknesses | ● | ● | ● | ○ | ● | ● | ○ | ● | ○ |
| **LLM10** Improper Output Handling | ● | ● | ● | ● | ● | ● | – | ● | ○ |

## OWASP Top 10 for Agentic Applications (ASI) — 2026 (announced 2025-12-09)

*Relocated verbatim from the 2026 entries; maps each LLM risk to the OWASP Top 10 for Agentic Applications (ASI) risks.*

| Risk | Element | Relevance |
|---|---|---|
| **LLM01** Prompt Injection | ● ASI01 — Agent Goal Hijack | Injected input overrides the system-prompt role/capability constraints, redirecting agent goals. |
| | ● ASI02 — Tool Misuse & Exploitation | Injected input drives unauthorized tool invocation (the "lethal trifecta" conditions). |

| Risk | Element | Relevance |
|---|---|---|
| | ● ASI03 — Identity & Privilege Abuse | Agent acts under the user's elevated credentials, performing actions the attacker could not. |
| | ● ASI05 — Unexpected Code Execution (RCE) | Shell / file-system / cloud-API access turns injection into arbitrary command execution. |
| | ● ASI06 — Memory & Context Poisoning | Memory and RAG poisoning taint every future session reading the store. |
| | ● ASI08 — Cascading Failures | Tool outputs re-enter the context window, enabling chained or self-replicating effects. |
| | ● ASI09 — Human-Agent Trust Exploitation | Injection bypasses human-in-the-loop confirmation. |
| **LLM02** Sensitive Information Disclosure | ● ASI06 — Memory & Context Poisoning | Persistent agent memory corruption leading to cross-session data leakage |
| **LLM03** Excessive Agency | ● ASI01 — Agent Goal Hijack | Prompt injection or hallucination diverting an agent from its intended task is the trigger mechanism for Excessive Agency ("hallucination/confabulation... or direct/indirect prompt injection") |
| | ● ASI02 — Tool Misuse & Exploitation | A manifestation of Excessive Agency, where extensions/tools carry functionality beyond what the task needs |
| | ● ASI03 — Identity & Privilege Abuse | A manifestation of Excessive Agency, where extensions connect with generic, over-privileged identities |
| | ● ASI05 — Unexpected Code Execution (RCE) | A shell-command extension that fails to filter out unintended commands, and a coding agent whose excessive autonomy and permissions destroyed production infrastructure |
| | ● ASI07 — Insecure Inter-Agent Communication | A malicious/compromised peer agent in multi-agent/collaborative systems is a trigger, and user authorization scope must be preserved "across chained extension or agent calls" |

| Risk | Element | Relevance |
|---|---|---|
| | ● ASI08 — Cascading Failures | A manifestation of Excessive Agency, countered by rate limiting and circuit breakers to halt runaway extension invocation |
| | ● ASI09 — Human-Agent Trust Exploitation | Excessive autonomy (failing to seek confirmation before high-impact actions) and human-in-the-loop approval both concern trust placed in unsupervised agent action |
| **LLM04** Supply Chain | ● ASI04 — Agentic Supply Chain Vulnerabilities | This entry's own scope note defers agentic-specific supply-chain risk to ASI04, leaving this entry to cover the non-agentic model, dataset, and artifact supply chain |
| **LLM05** Data and Model Poisoning | ● ASI04 — Agentic Supply Chain Vulnerabilities | Poisoned pre-trained models and malicious deserialization distributed via public repositories, and tampered inference-time artifacts such as chat templates/GGUF |
| | ● ASI06 — Memory & Context Poisoning | Persistent agent memory and recommendation poisoning, and long-term manipulation of agent decisions via injected memory |
| | ● ASI08 — Cascading Failures | Poisoned inputs propagating across multi-agent and enterprise workflows into unintended data exposure, and cross-tenant contamination via shared embeddings/memory |
| **LLM06** Unbounded Consumption | ● ASI02 — Tool Misuse & Exploitation | Agent-tool interaction loops and MCP tool-call fan-out let a single request drive recursive or high-volume tool calls that exhaust budget and availability |
| | ● ASI08 — Cascading Failures | Agentic architectures and tool-use protocols such as MCP "amplify a single request into cascading downstream operations," illustrated by the fan-out of one task into 50 calls |
| **LLM07** Misinformation | ● ASI08 — Cascading Failures | Incorrect state or evidence produced by one agent and trusted by another |

| Risk | Element | Relevance |
|---|---|---|
| | | propagates misinformation across a multi-agent workflow into a compounding, high-impact failure (Cross-Agent Misinformation Propagation; Cross-Agent Trust Failure) |
| | ● ASI09 — Human-Agent Trust Exploitation | Humans and downstream systems that treat fluent, confident model output as authoritative are the exploited trust surface this entry's overreliance framing describes (Unsupported or False Decision Support) |
| | ● ASI10 — Rogue Agents | An agent that falsifies task completion or fabricates evidence to mislead downstream trust matches this entry's forged-evidence and false-completion failure modes (Forged or Misattributed Evidence; Fabricated Task Completion) |
| **LLM08** Hidden Context Exposure | ● ASI06 — Memory & Context Poisoning | LLM08's scope explicitly defers "the agentic amplifications of this risk, e.g., persistent memory, inter-agent channels, tool configuration persistence, and multi-step agent compromise" to the OWASP Top 10 for Agentic Applications; ASI06 is the pointer for persistent-memory poisoning building on exposed hidden context, not an equivalent risk. |
| | ● ASI07 — Insecure Inter-Agent Communication | Same scope carve-out names "inter-agent channels" as an out-of-scope amplification; ASI07 is the pointer for hidden-context material propagating across agent-to-agent messaging, not an equivalent risk. |
| **LLM09** Vector and Embedding Weaknesses | ● ASI04 — Agentic Supply Chain Vulnerabilities | Embedding-model supply chain: the embedding model itself must be vetted, since "a backdoored embedding model corrupts the geometry of everything ingested" |
| | ● ASI06 — Memory & Context Poisoning | Agent-memory poisoning that does not rely on embedding geometry is out of scope here and referred to ASI06 |

| Risk | Element | Relevance |
|---|---|---|
| **LLM10** Improper Output Handling | ● ASI02 — Tool Misuse & Exploitation | 2026 addition, justified against this entry's own content: a general-purpose LLM passes its response to a privileged extension without output validation, causing the extension to be misused and shut down. |
| | ● ASI05 — Unexpected Code Execution (RCE) | Foundational crosswalk mapping; matches this entry's core risk of unvalidated LLM output reaching a shell, `exec`, or `eval`, resulting in remote code execution. |
| | ● ASI09 — Human-Agent Trust Exploitation | Canonical crosswalk baseline mapping — ASI09 lists Improper Output Handling as a contributing LLM risk. The content match is loose: this entry's scenarios are machine-to-machine (unvalidated output reaching a shell, browser, or database), while ASI09 concerns deceiving a human operator; the shared thread is unvalidated model output reaching a downstream trust decision. |

# OWASP GenAI Data Security 2026 (DSGAI) — v1.0 (2026-03-17)

*Each row maps an LLM Top 10 risk to OWASP GenAI Data Security 2026 (DSGAI) risk categories, where primary elements name the closest data-security peer and supporting elements name the most relevant adjacent data risks.*

| Risk | Element | Relevance |
|---|---|---|
| **LLM01** Prompt Injection | ● DSGAI01 — Sensitive Data Leakage | A successful injection makes disclosure and exfiltration the dominant outcome, spilling system-prompt content, retrieved private documents, and infrastructure details out of the model. |
| | ● DSGAI06 — Tool, Plugin & Agent Data Exchange Risks | Injection rides through MCP servers and third-party tool packages, where poisoned tool descriptions and unpinned connections let a compromised model act across connected systems. |
| | ○ DSGAI09 — Multimodal Capture & Cross-Channel Data Leakage | Steganographic image and invisible-Unicode payloads deliver instructions the human never sees, and markdown image-URL rendering carries the stolen data back out. |
| | ○ DSGAI16 — Endpoint & Browser Assistant Overreach | Browser, IDE, and email assistants that auto-summarize untrusted pages become the indirect-injection channel, from zero-click document exfiltration to a flipped IDE configuration flag. |
| | ○ DSGAI04 — Data, Model & Artifact Poisoning | One tainted entry in a RAG corpus or persistent memory taints every later session that reads it, making poisoning a cross-session propagation path for injection. |
| **LLM02** Sensitive Information Disclosure | ● DSGAI01 — Sensitive Data Leakage | The direct peer: unauthorized exposure of confidential, regulated, or proprietary data through any output channel is the whole risk. |
| | ● DSGAI18 — Inference & Data Reconstruction | Membership inference, embedding inversion, and internal-state inversion reconstruct protected data from |

| Risk | Element | Relevance |
|---|---|---|
| | | measurable model behavior without receiving the content directly. |
| | ○ DSGAI15 — Over-Broad Context Windows & Prompt Over-Sharing | Unscoped drives, legacy permissions, and auto-appended full records feed the model more sensitive data than the task needs, the upstream over-sharing that drives most incidents. |
| | ○ DSGAI08 — Non-Compliance & Regulatory Violations | The disclosure carries the regulatory consequence, triggering EU AI Act, GDPR, HIPAA, and CCPA obligations and their breach-notification clocks. |
| | ○ DSGAI14 — Excessive Telemetry & Monitoring Leakage | Observability platforms log full prompts, completions, retrieved chunks, and reasoning traces by default, exposing sensitive content to anyone with dashboard access. |
| **LLM03** Excessive Agency | ● DSGAI02 — Agent Identity & Credential Exposure | Extensions granted broad write access or a shared high-privilege account let a hijacked agent act far beyond its intended scope, so binding actions to user-scoped credentials across chained calls is the core control. |
| | ● DSGAI06 — Tool, Plugin & Agent Data Exchange Risks | The whole risk concerns extensions, tools, and plugins that retain unneeded capabilities an attacker can trigger, such as a mail extension that keeps its send function. |
| | ○ DSGAI01 — Sensitive Data Leakage | Excess agency lets a compromised agent scan a mailbox for sensitive information and forward it to an attacker. |
| **LLM04** Supply Chain | ● DSGAI04 — Data, Model & Artifact Poisoning | Tampered or backdoored models, adapters, and artifacts enter the pipeline through the supply chain, as with a maliciously edited open model or a poisoned format-conversion service. |
| | ● DSGAI05 — Data Integrity & Validation Failures | Unsigned, unverified artifacts and mutable references resolved at build or promotion time let tampered |

| Risk | Element | Relevance |
|---|---|---|
| | | components pass without integrity checks or signing. |
| | ○ DSGAI03 — Shadow AI & Unsanctioned Data Flows | Unclear supplier terms can silently route application data into a provider's training set, so vetting supplier terms and privacy policies is required. |
| | ○ DSGAI08 — Non-Compliance & Regulatory Violations | Unclear model and dataset licensing and copyright status create legal and compliance exposure that demands license tracking and auditing. |
| **LLM05** Data and Model Poisoning | ● DSGAI04 — Data, Model & Artifact Poisoning | The direct peer covering poisoning across pre-training, fine-tuning, embeddings, RAG, transfer learning, and distributed non-weight artifacts. |
| | ○ DSGAI05 — Data Integrity & Validation Failures | Strict, continuous validation of training and retraining inputs is the front-line defense against injected poison. |
| | ○ DSGAI21 — Disinformation & Integrity Attacks via Data Poisoning | Poisoning a knowledge base surfaces adversary-controlled content as authoritative output, an integrity attack driven by corrupted data. |
| | ○ DSGAI07 — Data Governance, Lifecycle & Classification | Dataset and model lineage tracked through an SBOM or ML-BOM plus version history enables rollback and forensics after a poisoning event. |
| **LLM06** Unbounded Consumption | ● DSGAI20 — Model Exfiltration & IP Replication | Functional model cloning and extraction through high-volume querying steal the model's intellectual property. |
| | ● DSGAI17 — Data Availability & Resilience Failures in AI Pipelines | Resource-exhausting requests and output-explosion attacks render the service unresponsive, the availability failure that resilience controls target. |
| | ○ DSGAI04 — Data, Model & Artifact Poisoning | A single poisoned fine-tuning sample can break end-of-sequence behavior and drive runaway output generation as a denial-of-service vector. |

| Risk | Element | Relevance |
|---|---|---|
| **LLM07** Misinformation | ● DSGAI21 — Disinformation & Integrity Attacks via Data Poisoning | Deliberately injected content makes the system surface false or forged material as authoritative, the core disinformation-via-poisoning mechanism. |
| | ○ DSGAI05 — Data Integrity & Validation Failures | Biased or corrupted source data and unvalidated ingestion are root causes that let misinformation enter and spread. |
| | ○ DSGAI06 — Tool, Plugin & Agent Data Exchange Risks | Unvalidated tool outputs and cross-agent exchange propagate false information, so tool calls need semantic validation. |
| **LLM08** Hidden Context Exposure | ○ DSGAI18 — Inference & Data Reconstruction | The entry is defined by extraction, inference, and reconstruction of hidden context, the same reconstruction attacks that recover a system prompt from model behavior. |
| | ○ DSGAI15 — Over-Broad Context Windows & Prompt Over-Sharing | The central preventive principle is to keep sensitive data out of hidden context and assume all context is discoverable. |
| | ○ DSGAI02 — Agent Identity & Credential Exposure | API keys and tokens embedded in hidden context are harvested once an attacker extracts that context. |
| **LLM09** Vector and Embedding Weaknesses | ● DSGAI13 — Vector Store Platform Data Security | Vector-store platform security is the entry's literal subject, covering encryption, access control, and export limits on embedding stores. |
| | ● DSGAI18 — Inference & Data Reconstruction | Embedding inversion that reconstructs plaintext and membership-inference oracles turn stored vectors back into their source data. |
| | ○ DSGAI11 — Cross-Context & Multi-User Conversation Bleed | Shared similarity search that applies the tenant filter only after retrieval leaks one tenant's chunks into another's results. |
| | ○ DSGAI04 — Data, Model & Artifact Poisoning | Retrieval-time poisoning, semantic-cache poisoning, and multimodal embedding poisoning steer the system to attacker-controlled content. |

| Risk | Element | Relevance |
|---|---|---|
| | ○ DSGAI17 — Data Availability & Resilience Failures in AI Pipelines | A single crafted blocker document can dominate retrieval and take the RAG system off the air, an availability attack on the retrieval layer. |
| **LLM10** Improper Output Handling | ● DSGAI12 — Unsafe Natural-Language Data Gateways (LLM-to-SQL/Graph) | LLM-generated SQL executed without parameterization or scrutiny reaches the database directly and can delete entire tables. |
| | ○ DSGAI06 — Tool, Plugin & Agent Data Exchange Risks | Unvalidated model output crosses into a downstream tool or extension for execution. |
| | ○ DSGAI01 — Sensitive Data Leakage | Unescaped output rendered by a client encodes and sends conversation data to an attacker-controlled server, as in image-URL exfiltration. |
| | ○ DSGAI16 — Endpoint & Browser Assistant Overreach | Client renderers such as chat UIs, IDEs, and email clients auto-fetch external resources from raw model output, executing the exfiltration. |

## MITRE ATLAS — content v2026.06 (format-version 6.0.0)

*Read each row as an OWASP LLM risk mapped to the MITRE ATLAS adversary tactics (AML.TAxxxx) an attack exploiting it traverses, where primary tactics carry the core adversary objective and supporting tactics the enabling steps.*

| Risk | Element | Relevance |
|---|---|---|
| **LLM01** Prompt Injection | ● AML.TA0004 Initial Access | Injected instructions, whether typed directly into the prompt or hidden in content the model later retrieves, are the foothold that lets an attacker seize control of the model's behavior. |
| | ● AML.TA0005 Execution | A successful injection drives the agent to invoke tools and run adversary-chosen commands, including arbitrary code execution on the host. |

| Risk | Element | Relevance |
|---|---|---|
| | ● AML.TA0006 Persistence | Injected instructions written into long-term memory or a retrieval corpus survive across sessions and re-fire whenever the poisoned entry is recalled. |
| | ● AML.TA0007 Defense Evasion | Attackers hide the injection with invisible Unicode, Base64 or ROT13 encoding, and low-resource languages, and use jailbreak phrasing to slip past safety filters. |
| | ● AML.TA0010 Exfiltration | Injection redirects the model to leak private conversation, database, or repository contents through image-URL, tool, or invisible-character channels. |
| | ○ AML.TA0001 AI Attack Staging | Adversaries optimize the injection payload in advance, using gradient-oracle fine-tuning and payload-splitting to raise attack success rates. |
| | ○ AML.TA0011 Impact | A runtime injection can trigger destructive action, such as wiping a developer's local files. |
| | ○ AML.TA0012 Privilege Escalation | An injected agent running on a trusted backend acts under the user's elevated credentials, yielding access beyond the attacker's own rights. |
| **LLM02** Sensitive Information Disclosure | ● AML.TA0010 Exfiltration | Membership inference, model inversion, model extraction, and API-based data theft all move protected data out of the system, sometimes through covert DNS or image channels. |
| | ○ AML.TA0013 Credential Access | An embedded vendor API key placed in the system prompt can be coaxed out of the model, exposing a live credential. |
| | ○ AML.TA0000 AI Model Access | Extraction, membership inference, and inversion run offline at unbounded rates against open-weights deployments, so the access mode itself is the attack surface. |

| Risk | Element | Relevance |
|---|---|---|
| | ○ AML.TA0007 Defense Evasion | Cross-lingual, Base64, and hex encodings defeat regex and blocklist data-loss filters, and cross-modal transformation slips data past single-modality inspection. |
| **LLM03** Excessive Agency | ● AML.TA0005 Execution | An over-broad extension, such as one that runs unfiltered shell commands, lets the agent execute damaging operations, including destructive commands against production systems. |
| | ● AML.TA0011 Impact | Excessive permissions turn an agent action into confidentiality, integrity, and availability harm, from deleting a user's email to destroying production databases and their snapshots. |
| | ○ AML.TA0010 Exfiltration | An indirect injection can make an over-permissioned mail extension forward sensitive inbox contents to an attacker-controlled address. |
| | ○ AML.TA0012 Privilege Escalation | A confused-deputy or weaponized over-privileged agent lets an attacker act with the agent's standing high privileges. |
| **LLM04** Supply Chain | ● AML.TA0004 Initial Access | Compromised models, packages, adapters, and training pipelines are the foothold by which a tampered artifact enters the victim's environment. |
| | ○ AML.TA0003 Resource Development | Attackers publish tampered models, pre-register confabulated (slopsquatted) package names, and re-register abandoned namespaces to stage malicious dependencies. |
| | ○ AML.TA0005 Execution | Malicious model manifests, pickle deserialization, and format-parser overflows yield remote code execution when an artifact is loaded. |
| | ○ AML.TA0006 Persistence | A backdoor embedded in a LoRA adapter or a model's computational |

| Risk | Element | Relevance |
|---|---|---|
| | | graph survives in the deployed model, in a format that looks safe. |
| **LLM05** Data and Model Poisoning | ● AML.TA0003 Resource Development | Adversaries publish poisoned datasets and models and inject tainted samples into training data that others will consume. |
| | ● AML.TA0006 Persistence | Backdoors and sleeper triggers survive safety alignment and retraining, staying dormant in the model until a specific input activates them. |
| | ● AML.TA0011 Impact | Poisoning erodes model and dataset integrity, degrading accuracy, weakening refusals, and steering the model toward harmful outputs. |
| | ○ AML.TA0004 Initial Access | Organizations that pull compromised models or datasets from public repositories inherit the hidden triggers along with the artifact. |
| | ○ AML.TA0001 AI Attack Staging | The attacker backdoors the model and crafts trigger or adversarial data before the model is deployed. |
| | ○ AML.TA0005 Execution | Loading an unsafe artifact executes attacker code, as with pickle deserialization on model load or chat-template injection in a model file. |
| **LLM06** Unbounded Consumption | ● AML.TA0011 Impact | Input floods deny the ML service to legitimate users, and cost-harvesting attacks run up the victim's bill in a denial-of-wallet. |
| | ● AML.TA0010 Exfiltration | Crafted API queries and side-channel weight harvesting extract the model, stealing the intellectual property embodied in its parameters. |
| | ○ AML.TA0000 AI Model Access | Both extraction and resource-exhaustion attacks operate through inference-API access, which is the prerequisite for the technique. |

| Risk | Element | Relevance |
|---|---|---|
| **LLM07** Misinformation | ● AML.TA0011 Impact | False output that is trusted and acted upon causes financial loss, security incidents, safety risks, and operational disruption, as when a fabricated alert derails operations. |
| | ○ AML.TA0003 Resource Development | Attackers publish malicious packages under names the model is known to hallucinate, staging a supply-chain trap keyed to the confabulation. |
| | ○ AML.TA0004 Initial Access | A hallucinated dependency, once recommended and installed by a trusting developer, becomes an AI supply-chain foothold. |
| | ○ AML.TA0001 AI Attack Staging | Adversaries craft inputs that steer the model toward specific false claims, an induced-misinformation staging step. |
| **LLM08** Hidden Context Exposure | ● AML.TA0008 Discovery | The attack extracts and reconstructs the system prompt, tool list, roles, and refusal logic, discovering the AI system's hidden configuration. |
| | ● AML.TA0010 Exfiltration | Leaking that hidden context across the trust boundary is model data leakage, moving configuration the operator meant to keep private into the attacker's hands. |
| | ○ AML.TA0002 Reconnaissance | Extracted context gives the attacker concrete targets for follow-on prompt injection and reconnaissance for downstream action chaining. |
| | ○ AML.TA0013 Credential Access | Credentials embedded in the system prompt are obtained when the hidden context leaks, handing the attacker usable secrets. |
| **LLM09** Vector and Embedding Weaknesses | ● AML.TA0006 Persistence | Poisoned corpus and cache entries stay in the vector store and re-trigger on matching queries, a durable foothold in the retrieval layer. |

| Risk | Element | Relevance |
|---|---|---|
| | ● AML.TA0010 Exfiltration | Embedding inversion, cross-tenant inference, and membership inference pull private data and documents out through the shared index. |
| | ○ AML.TA0001 AI Attack Staging | Attackers craft content whose embedding lands near a target query and use surrogate encoders to engineer threshold-straddling vectors. |
| | ○ AML.TA0011 Impact | Retrieval jamming floods the index as an availability attack, and retrieval poisoning erodes the integrity of returned answers. |
| | ○ AML.TA0008 Discovery | Cross-tenant probing infers the existence, topic, and rough volume of other tenants' documents, and membership inference reveals whether a specific document is indexed. |
| **LLM10** Improper Output Handling | ● AML.TA0005 Execution | Unvalidated model output passed to a shell, browser, or SQL interpreter executes adversary-controlled code, yielding command execution, cross-site scripting, or SQL injection. |
| | ○ AML.TA0010 Exfiltration | Model output encoded and sent to an attacker server, or embedded in a Markdown-image URL, carries sensitive data out of the application. |
| | ○ AML.TA0011 Impact | Unhandled output reaching a privileged sink can delete database tables or force a downstream service offline. |
| | ○ AML.TA0012 Privilege Escalation | When the model holds privileges beyond the end user, unvalidated output reaching a privileged sink escalates those privileges to the attacker. |

## MITRE ATT&CK — v19.1

*Each row maps an OWASP LLM risk to the MITRE ATT&CK v19.1 Enterprise tactics (TA IDs) an adversary traverses when exploiting it, with primary tactics marking the risk's central attack objective and supporting tactics the adjacent stages.*

| Risk | Element | Relevance |
|---|---|---|
| **LLM01** Prompt Injection | ● TA0001 Initial Access | A compromised IDE extension and a malicious MCP npm package deliver injected instructions into the model's context, making the software supply chain the entry point for the attack. |
| | ○ TA0002 Execution | Injected instructions drive arbitrary command execution and destructive actions on the host or connected systems through an agent's shell-tool access. |
| | ○ TA0010 Exfiltration | Injected content exfiltrates data over covert channels such as markdown image URLs, hidden Unicode, and tool-logging side channels while the visible response stays benign. |
| | ○ TA0005 Stealth | Attackers smuggle instructions past human and model review using zero-width and variation-selector Unicode, tag-block encoding, hidden page-source text, and sub-perceptual image steganography. |
| **LLM02** Sensitive Information Disclosure | ● TA0010 Exfiltration | A hidden markdown-image or webhook channel carries sensitive data outbound while the model's visible answer stays innocuous. |
| | ○ TA0009 Collection | Side-channel observation of encrypted LLM traffic reconstructs conversation topics and token content, as in topic inference exceeding 98% AUPRC and token-length reconstruction. |
| | ○ TA0006 Credential Access | Prompt injection prints a system prompt carrying an embedded vendor API key, and exposed logging stores |

| Risk | Element | Relevance |
|---|---|---|
| | | leak API keys an adversary can harvest. |
| **LLM03** Excessive Agency | ● TA0002 Execution | An open-ended shell-command extension that fails to filter unintended commands lets an agent execute arbitrary code, as when a coding agent destroyed production infrastructure. |
| | ● TA0040 Impact | Excessive autonomy leads to destructive actions such as deleting emails without confirmation or wiping a production database and years of snapshots. |
| | ○ TA0010 Exfiltration | An indirect injection commands the agent to scan the user's inbox for sensitive information and forward it to an attacker's email address. |
| | ○ TA0004 Privilege Escalation | Over-privileged or generic high-privilege agent identities and confused-deputy conditions raise the attacker's effective privilege to that of the agent. |
| **LLM04** Supply Chain | ● TA0001 Initial Access | Compromised packages and build pipelines, from a poisoned PyPI dependency to the xz-utils and Codecov compromises, deliver attacker code into the victim environment. |
| | ○ TA0042 Resource Development | Attackers register malicious packages under hallucinated names (slopsquatting) and inject CI/CD caches to stage trojanized releases for distribution. |
| | ○ TA0002 Execution | Malicious model manifests, namespace reuse, and pickle deserialization on model load produce remote code execution once the artifact is pulled and run. |

| Risk | Element | Relevance |
|---|---|---|
| | ○ TA0003 Persistence | A merged LoRA adapter or backdoored model graph provides a covert entry point that survives across deployments and passes downstream provenance checks. |
| **LLM05** Data and Model Poisoning | ○ TA0040 Impact | Poisoned training data or weights manipulate model integrity to produce harmful or degraded outputs, such as bypassing fraud detection or dropping accuracy from 90% to 15% when a trigger fires. |
| | ○ TA0001 Initial Access | Malicious models with embedded backdoors distributed through public repositories reach the victim when pulled into a pipeline. |
| | ○ TA0002 Execution | Embedded malicious code executes during unsafe pickle model loading, and a tampered chat template carries trigger-activated instructions. |
| **LLM06** Unbounded Consumption | ● TA0040 Impact | Resource-exhausting inputs crash or slow the service and denial-of-wallet attacks harvest compute cost, disrupting availability and inflating spend. |
| | ○ TA0010 Exfiltration | Model-extraction queries harvest model information to a remote resource under the attacker's control, enabling intellectual-property theft through cloning. |
| **LLM07** Misinformation | ○ TA0001 Initial Access | Attackers publish malicious packages under names that coding assistants hallucinate (slopsquatting), turning fabricated recommendations into a software supply-chain compromise. |
| **LLM08** Hidden Context Exposure | ○ TA0006 Credential Access | Hidden context such as system prompts holding API keys, database credentials, and user tokens leaks to an attacker who then reuses the exposed credentials. |

| Risk | Element | Relevance |
|---|---|---|
| **LLM09** Vector and Embedding Weaknesses | ○ TA0009 Collection | Cross-tenant probing of a shared vector index infers other tenants' data held in that information repository. |
| | ○ TA0010 Exfiltration | Zero-shot embedding inversion reconstructs source documents and PII from a leaked embedding backup, taking data outside the boundary. |
| | ○ TA0040 Impact | Retrieval jamming degrades availability of the retrieval layer and retrieval-time poisoning manipulates the context fed to the model. |
| **LLM10** Improper Output Handling | ● TA0002 Execution | Passing unvalidated model output into shell, eval, or database interfaces yields remote code execution, cross-site scripting, and SQL injection on backend systems. |
| | ○ TA0010 Exfiltration | Model output encodes sensitive conversation data and sends it to an attacker-controlled server, including via markdown-image URLs. |
| | ○ TA0040 Impact | Unscrutinized model-generated SQL deletes database tables and a privileged extension is driven to shut down, causing data destruction and loss of availability. |
| | ○ TA0004 Privilege Escalation | When the application grants the model privileges beyond those intended for end users, unhandled output enables escalation of privileges. |

## MITRE CWE (Common Weakness Enumeration) — 4.20

*Each OWASP LLM Top 10 2026 entry maps to the CWE weaknesses that name its root cause, with primary CWEs capturing the definitional weakness and supporting CWEs the top contributing failure modes.*

| Risk | Element | Relevance |
|---|---|---|
| **LLM01** Prompt Injection | ● CWE-1427 Improper Neutralization of Input Used for LLM Prompting | Untrusted input from prompts, retrieved documents, tool output, or memory alters model behavior because the model draws no boundary between instructions and data, the definitional prompt-injection weakness. |
| | ○ CWE-707 Improper Neutralization (Pillar) | The pillar-level root cause is a failure to separate and neutralize attacker instructions from data on a single token stream. |
| | ○ CWE-349 Acceptance of Extraneous Untrusted Data With Trusted Data | Context-window pooling merges untrusted retrieved content and tool output with trusted instructions under no enforced trust boundary. |
| | ○ CWE-693 Protection Mechanism Failure (Pillar) | Jailbreaks and adaptive guardrail evasion degrade prompt-injection classifiers and link filters, driving defenses to high adversarial success rates. |
| **LLM02** Sensitive Information Disclosure | ● CWE-200 Exposure of Sensitive Information to an Unauthorized Actor | The entry defines disclosure as exposing confidential, regulated, or privileged data through an unauthorized channel. |
| | ● CWE-359 Exposure of Private Personal Information | PII and PHI disclosure sits at the center of the entry, anchored to GDPR and HIPAA obligations across its examples. |
| | ● CWE-212 Improper Removal of Sensitive Information Before Storage or Transfer | Redaction and removal failures surface sensitive spans, such as text hidden under a black-rectangle redaction layer resurfaced by summarization. |

| Risk | Element | Relevance |
|---|---|---|
| | ● CWE-532 Insertion of Sensitive Information into Log File | Observability pipelines log full prompts, completions, and reasoning traces, exposing sensitive data through log and telemetry stores. |
| | ○ CWE-285 Improper Authorization | Retrieval runs before any authorization check, so cosine similarity returns documents the requester has no right to see. |
| | ○ CWE-732 Incorrect Permission Assignment for Critical Resource | Unscoped drives, legacy permissions, and over-permissive knowledge bases feed the retrieval corpus with sensitive data. |
| | ○ CWE-201 Insertion of Sensitive Information Into Sent Data | Tool-call arguments and outbound requests to external providers carry more sensitive fields than the task actually requires. |
| **LLM03** Excessive Agency | ● CWE-285 Improper Authorization | Authorization must be enforced at a policy decision point in application logic rather than left to the model, defending against confused-deputy and privilege-abuse behavior. |
| | ● CWE-732 Incorrect Permission Assignment for Critical Resource | Database identities and service accounts granted write and delete rights or broadly high privileges beyond what the task needs are a named root cause of excessive agency. |
| | ○ CWE-284 Improper Access Control (Pillar) | The pillar umbrella covers the entry's permission and access-control failures, including missing mediation and lost user context. |
| | ○ CWE-770 Allocation of Resources Without Limits or Throttling | Thresholds and circuit breakers on invocation count or cumulative parameter value are needed to halt runaway tool calls. |
| | ○ CWE-1427 Improper Neutralization of Input Used for LLM Prompting | Direct and indirect prompt injection, such as a crafted email, is the |

| Risk | Element | Relevance |
|---|---|---|
| | | trigger that turns excess permissions into harmful action. |
| **LLM04** Supply Chain | ● CWE-494 Download of Code Without Integrity Check | Models and adapters are pulled by mutable tags or resolved by name without signature or hash verification, letting namespace reuse deliver code execution. |
| | ● CWE-829 Inclusion of Functionality from Untrusted Control Sphere | The entry centers on including third-party models, adapters, and packages from untrusted hubs and registries, including slopsquatting and tampered models. |
| | ○ CWE-349 Acceptance of Extraneous Untrusted Data With Trusted Data | A malicious LoRA adapter merged into a trusted base model blends poisoned artifacts into an otherwise trusted merge-and-convert pipeline. |
| | ○ CWE-664 Improper Control of a Resource Through its Lifetime (Pillar) | Insecure deserialization of model weights and native-parser memory corruption are lifetime-control failures over the loaded model resource. |
| **LLM05** Data and Model Poisoning | ● CWE-349 Acceptance of Extraneous Untrusted Data With Trusted Data | Attacker-controlled untrusted data is mixed into trusted training corpora, retrieval stores, and continuous-learning feeds, the canonical poisoning weakness. |
| | ● CWE-829 Inclusion of Functionality from Untrusted Control Sphere | Loading untrusted models, LoRA and PEFT adapters, tokenizer configs, and chat templates from public repositories pulls poisoned functionality into the pipeline. |
| | ○ CWE-494 Download of Code Without Integrity Check | Signing and hash verification of model artifacts is prescribed because unverified model downloads carry tampered weights. |
| | ○ CWE-1427 Improper Neutralization of Input Used for LLM Prompting | Poisoned retrieval documents and hidden web instructions reach the prompt without neutralization, the |

| Risk | Element | Relevance |
|---|---|---|
| | | indirect-injection facet of poisoning. |
| **LLM06** Unbounded Consumption | ● CWE-400 Uncontrolled Resource Consumption | Every input-flood, context-overflow, thinking-token, and GPU-or-memory exhaustion pattern in the entry is a form of uncontrolled resource consumption. |
| | ● CWE-770 Allocation of Resources Without Limits or Throttling | The defining root cause is the absence of rate, token, cost, and queue limits, with pre-flight estimation and hard spending caps as the missing controls. |
| | ○ CWE-664 Improper Control of a Resource Through its Lifetime (Pillar) | The pillar ancestor of the consumption weaknesses covers management of the compute and memory resource across its lifetime. |
| | ○ CWE-691 Insufficient Control Flow Management (Pillar) | Reasoning loops that burn thinking tokens and recursive or infinite tool-calling loops need recursion-depth limits and loop detection. |
| | ○ CWE-829 Inclusion of Functionality from Untrusted Control Sphere | A malicious third-party tool or skill pulled from an open-source repository drives the resource-exhausting loops. |
| **LLM07** Misinformation | ● CWE-1426 Improper Validation of Generative AI Output | Incorrect, unsupported, or misleading model output is trusted and acted upon without verification, the entry's core weakness. |
| | ○ CWE-349 Acceptance of Extraneous Untrusted Data With Trusted Data | A downstream agent accepts fabricated or misattributed upstream output as trusted evidence, a cross-agent trust failure. |
| | ○ CWE-494 Download of Code Without Integrity Check | A hallucinated package name is installed without any integrity or authenticity check, the slopsquatting failure. |

| Risk | Element | Relevance |
|---|---|---|
| | ○ CWE-1427 Improper Neutralization of Input Used for LLM Prompting | Adversarially crafted inputs induce false or misleading output, the input-side manipulation behind adversarially induced misinformation. |
| **LLM08** Hidden Context Exposure | ● CWE-200 Exposure of Sensitive Information to an Unauthorized Actor | The entry is defined by unauthorized extraction, inference, or reconstruction of hidden system instructions and operational context. |
| | ○ CWE-798 Use of Hard-coded Credentials | API keys, tokens, and connection strings embedded in the system prompt are the highest-severity credential exposure. |
| | ○ CWE-693 Protection Mechanism Failure (Pillar) | Refusal and behavior-control instructions placed in hidden context are reverse-engineered and bypassed rather than enforced. |
| | ○ CWE-285 Improper Authorization | Authorization and access control must be enforced independently of the model, since roles and permissions carried in tool descriptions leak and fail. |
| **LLM09** Vector and Embedding Weaknesses | ● CWE-200 Exposure of Sensitive Information to an Unauthorized Actor | Embedding inversion recovers source text and cross-tenant leakage exposes other customers' data, the core confidentiality failure. |
| | ● CWE-285 Improper Authorization | The access-control decision happens after the embedding-space search runs, so tenant scoping must be enforced server-side inside the index query. |
| | ○ CWE-359 Exposure of Private Personal Information | Customer PII is reconstructed from a leaked embedding backup, creating GDPR data-subject risk. |
| | ○ CWE-732 Incorrect Permission Assignment for Critical Resource | Mis-scoped chunk-level ACLs and a cloud misconfiguration exposing a |

| Risk | Element | Relevance |
|---|---|---|
| | | vector-database backup are access-control misassignments. |
| | ○ CWE-829 Inclusion of Functionality from Untrusted Control Sphere | Retrieval poisoning causes attacker content from an untrusted source to be retrieved and fed to the model as trusted context. |
| **LLM10** Improper Output Handling | ● CWE-1426 Improper Validation of Generative AI Output | Model output is passed downstream with insufficient validation, sanitization, or handling, the entry's exact subject. |
| | ● CWE-116 Improper Encoding or Escaping of Output | Missing context-aware output encoding across HTML, JavaScript, SQL, and terminal sinks is the root cause. |
| | ● CWE-79 Cross-site Scripting (XSS) | Cross-site scripting is the most-repeated concrete sink, addressed by content-security-policy and output-encoding defenses. |
| | ○ CWE-707 Improper Neutralization (Pillar) | The neutralization pillar covers control-character sanitization of ANSI, BEL, OSC, backspace, and carriage-return sequences and SQL escaping and parameterization. |
| | ○ CWE-200 Exposure of Sensitive Information to an Unauthorized Actor | Unhandled output leaks sensitive conversation and website content to an attacker. |
| | ○ CWE-201 Insertion of Sensitive Information Into Sent Data | Sensitive data is smuggled into the hostname or query string of an auto-fetched Markdown-image request. |

## NIST AI 600-1 (Generative AI Profile) — v1.0 (July 2024)

*Each row maps an OWASP LLM risk to the NIST AI 600-1 Generative AI Profile risk categories it engages; primary categories name the entry's core risk, supporting categories name secondary facets.*

| Risk | Element | Relevance |
|---|---|---|
| **LLM01** Prompt Injection | ● Information Security | Prompt injection expands the attack surface, letting adversarial text hijack the model to exfiltrate data or trigger unauthorized actions across connected systems. |
| | ○ Information Integrity | Injected instructions manipulate outputs into attacker-chosen content and poison retrieved or remembered context, corrupting the information the system produces. |
| | ○ Data Privacy | Successful injection can exfiltrate private conversation history, uploaded documents, and repository contents to an attacker. |
| | ○ Human-AI Configuration | Human-in-the-loop confirmation degrades under approval fatigue, and benign content can carry unintended injected instructions, both matters of human-oversight configuration. |
| **LLM02** Sensitive Information Disclosure | ● Data Privacy | The entry centers on privacy leakage and de-anonymization of PII, PHI, biometric, and genomic data, including membership inference that confirms a named individual and defeats erasure obligations. |
| | ● Information Security | It concerns confidentiality of the system and its data through model and data exfiltration, credential exposure, and extraction attacks. |
| | ○ Intellectual Property | Trade secrets and model weights are protected assets at risk, and |

| Risk | Element | Relevance |
|---|---|---|
| | | models can reproduce copyrighted or watermarked training material verbatim. |
| | ○ Value Chain and Component Integration | Third-party observability platforms, SDKs, and integrated browser components disclose sensitive data through the surrounding ecosystem. |
| **LLM03** Excessive Agency | ● Information Security | Excessive agency lets an agent exfiltrate data, perform unauthorized writes, and take destructive actions across the downstream systems it can reach. |
| | ○ Human-AI Configuration | Excessive autonomy acting without confirmation is a root cause, and human-in-the-loop approval is the primary control, both configuration of human oversight. |
| | ○ Confabulation | Hallucinated or confabulated reasoning can trigger the agent to take damaging real-world actions. |
| | ○ Data Privacy | An over-privileged agent can forward a user's private email content to an attacker, a personal-data breach. |
| **LLM04** Supply Chain | ● Value Chain and Component Integration | The entry is entirely about the integrity of third-party models, datasets, adapters, and integrated components across the AI value chain. |
| | ● Information Security | Supply-chain compromise produces backdoors, poisoning, remote code execution, and other security breaches. |
| | ○ Intellectual Property | Unclear licensing for use, distribution, and commercialization, plus copyright |

| Risk | Element | Relevance |
|---|---|---|
| | | exposure from supplier material, put intellectual property at risk. |
| | ○ Information Integrity | Tampered or poisoned models emit biased outputs and misinformation, and a tampered on-device model can steer users to scam sites. |
| | ○ Confabulation | Coding assistants confabulate nonexistent package names that attackers pre-register, the slopsquatting vector. |
| **LLM05** Data and Model Poisoning | ● Information Integrity | Poisoning steers answers, injects subtle misinformation, and manipulates recommendations and downstream business decisions. |
| | ● Value Chain and Component Integration | Third-party datasets, shared model repositories, and bundled artifacts introduce compromise through the GenAI value chain. |
| | ● Information Security | Data and model poisoning, embedded backdoors, and unsafe-artifact code execution are core security risks. |
| | ○ Dangerous, Violent, or Hateful Content | Targeted poisoning erodes refusal behaviors, as when a public chatbot was manipulated into producing offensive content. |
| **LLM06** Unbounded Consumption | ● Information Security | Unbounded consumption threatens availability through denial of service and confidentiality through model extraction and side-channel theft of weights and architecture. |
| | ○ Intellectual Property | Model extraction and functional cloning steal the model as intellectual property and trade secret. |

| Risk | Element | Relevance |
|---|---|---|
| | ○ Value Chain and Component Integration | Shared inference infrastructure and third-party serving frameworks add supply-chain attack surface. |
| **LLM07** Misinformation | ● Information Integrity | The entry concerns GenAI producing false or misleading information that degrades human decisions, the definition of this misinformation risk. |
| | ● Confabulation | Confabulation, NIST's term for confidently stated but false output, is named as a primary source of the misinformation. |
| | ● Human-AI Configuration | Automation bias and overreliance are called a key factor, addressed by calibrating human and system trust. |
| | ○ Information Security | Adversarially induced misinformation and false alerts can cause security incidents and operational disruption. |
| | ○ Value Chain and Component Integration | Hallucinated dependencies and unvalidated third-party tool outputs are value-chain integration risks. |
| **LLM08** Hidden Context Exposure | ● Information Security | System-prompt and context extraction, an expanded attack surface, and lowered barriers to targeted exploitation are the core security risks. |
| | ○ Intellectual Property | Exposed hidden context can reveal proprietary behavior and sensitive implementation details, such as a leaked system prompt. |
| **LLM09** Vector and Embedding Weaknesses | ● Data Privacy | Embedding inversion, membership inference, and cross-tenant leakage recover PII and infer data-subject membership, triggering breach obligations. |

| Risk | Element | Relevance |
|---|---|---|
| | ● Information Security | Access-control failure, unauthorized cross-tenant retrieval, oracle and endpoint abuse, and availability jamming are core security concerns. |
| | ● Information Integrity | Retrieval poisoning corrupts the context the model relies on, degrading the integrity of retrieved and generated information. |
| | ○ Value Chain and Component Integration | A backdoored third-party embedding model corrupts the geometry of everything ingested, a value-chain integrity risk. |
| **LLM10** Improper Output Handling | ● Information Security | Unhandled model output flows into downstream systems as remote code execution, XSS, SQL injection, SSRF, CSRF, and privilege escalation. |
| | ○ Data Privacy | Unsanitized output can leak sensitive conversation data to an attacker. |
| | ○ Confabulation | Generated code can reference hallucinated nonexistent software packages, a confabulation facet of unsafe output. |
| | ○ Value Chain and Component Integration | Third-party extensions that fail to validate inputs and hallucinated-package installs implicate component integration and supply chain. |

## NIST AI RMF (AI 100-1) — v1.0 (2023)

*Each row maps an OWASP LLM risk to the NIST AI RMF (AI 100-1) Categories whose outcomes it most directly exercises, with "primary" marking a near-verbatim fit and "supporting" a partial or governance-level fit.*

| Risk | Element | Relevance |
|---|---|---|
| **LLM01** Prompt Injection | ○ MEASURE 1 (methods & metrics) | The entry rejects static-only attack-success-rate claims and mandates adaptive-attack metrics, baselining injection defenses with AgentDojo and JailbreakBench because static scores run near zero while adaptive success exceeds 90 percent. |
| | ○ MEASURE 2 (evaluate trustworthiness) | Defenses are red-teamed against attackers who have read the full defense specification, evaluating the deployed system's injection resilience rather than trusting a metric selected in isolation. |
| | ○ MAP 4 (map risks across all components) | Decomposing an injection along delivery surface, propagation behavior, and encoding is presented as a threat-modeling step that maps risk across components including RAG corpora, MCP servers, and third-party tool packages. |
| **LLM02** Sensitive Information Disclosure | ○ MAP 5 (characterize impacts on people) | Disclosure is framed as impact to data subjects, where membership inference yields a regulatory-breach determination about a named individual under GDPR or HIPAA. |
| | ○ MEASURE 2 (evaluate trustworthiness) | A release gate requires disclosure red-teaming that quantitatively measures extraction, membership inference, embedding inversion, internal-state inversion, side channels, and LoRA extractability. |
| | ○ MANAGE 4 (risk treatment, response & recovery) | The disclosure incident-response playbook sets breach-notification |

| Risk | Element | Relevance |
|---|---|---|
| | | timelines and treats leaks with unlearning, retraining, withdrawal, and vector and cache cleanup, plus vendor notice. |
| **LLM03** Excessive Agency | ○ MANAGE 4 (risk treatment, response & recovery) | Logging and monitoring of extension and downstream activity is paired with circuit breakers that halt, rate-limit, or escalate agent actions for human review. |
| **LLM04** Supply Chain | ● GOVERN 6 (third-party & supply-chain policy) | The entire entry is LLM supply chain, covering supplier vetting, terms-and-conditions review, and component-patching policy for third-party software, data, and models. |
| | ● MAP 4 (map risks across all components) | A signed SBOM, AIBOM, and ML-SBOM component inventory together with component and supplier vetting maps risk across every third-party model, dataset, and dependency. |
| | ● MANAGE 3 (manage third-party risks) | Vetting and re-auditing suppliers and treating model conversion or merge services as high-risk promotion points is direct third-party-entity risk management. |
| | ○ MEASURE 2 (evaluate trustworthiness) | AI red teaming and evaluation when selecting third-party models, plus in-production adversarial-robustness testing and anomaly detection, assess acquired components. |
| | ○ MEASURE 3 (track risks over time) | Continuous validation of upstream model integrity, in-production anomaly detection, and regular BOM and supplier re-audits track supply-chain risk over time. |
| **LLM05** Data and Model Poisoning | ○ MEASURE 3 (track risks over time) | Outputs, training loss, and behavioral patterns are monitored for drift against defined thresholds |

| Risk | Element | Relevance |
|---|---|---|
| | | to detect subtle poisoning that emerges over time. |
| | ○ MEASURE 2 (evaluate trustworthiness) | Continuous red-teaming with adversarial and trigger-based prompts, with dedicated trigger-probing required after every alignment cycle, evaluates the model for hidden backdoors. |
| | ○ GOVERN 6 (third-party & supply-chain policy) | Vendor vetting, model and data lineage tracking, and controls against open-source dataset supply-chain poisoning and malicious models address third-party poisoning vectors. |
| **LLM06** Unbounded Consumption | ○ MANAGE 1 (prioritize & respond to risks) | Rate limits, hard spending caps, agentic circuit breakers, and graceful degradation form an impact-prioritized response informed by cost-attribution monitoring and baselines. |
| | ○ MEASURE 2 (evaluate trustworthiness) | Adversarial-perturbation scanning and resource-consumption red-teaming, supported by tool-behavior baselines, evaluate the system for the secure-and-resilient availability characteristic. |
| **LLM07** Misinformation | ● MEASURE 2 (evaluate trustworthiness) | The entry's defense is evaluation-centric, using groundedness and consistency checks, misinformation monitoring and testing, and continuous adversarial evaluation to measure the validity, reliability, and accuracy that misinformation threatens. |
| | ○ MEASURE 1 (methods & metrics) | Verification signals such as groundedness and consistency checks replace naive model confidence as the metric for whether a claim can be trusted before action. |

| Risk | Element | Relevance |
|---|---|---|
| | ○ MEASURE 3 (track risks over time) | Claims, evidence, and outcomes are logged and adversarial scenarios tested so misinformation is tracked over time. |
| | ○ MANAGE 1 (prioritize & respond to risks) | Runtime verification and approval workflows for high-impact actions, together with blast-radius limits, are impact-prioritized response controls. |
| **LLM08** Hidden Context Exposure | ○ MEASURE 2 (evaluate trustworthiness) | Detecting hidden-context exposure is a security-evaluation activity, red-teaming for prompt and context extraction grounded in RL-based red-teaming for privacy leakage and system-prompt-extraction research. |
| **LLM09** Vector and Embedding Weaknesses | ○ MAP 4 (map risks across all components) | The embedding layer is established as part of the application's trust boundary, directing teams to vet the embedding model and treat third-party embedding services and backups as in-scope components. |
| | ○ MAP 5 (characterize impacts on people) | Impact to data subjects is characterized, since embedding inversion recovers PII and an embeddings-only leak is reclassified as a source-document breach triggering breach notification. |
| | ○ MANAGE 4 (risk treatment, response & recovery) | Immutable retrieval logs and updated incident-response playbooks treat embeddings-only leaks as source-data breaches with GDPR Article 33 notification. |

## CSA AI Controls Matrix (AICM) — v1.1 (2026-06-22)

*Each row maps an OWASP LLM risk to the CSA AI Controls Matrix v1.1 control domains that address it, where primary domains carry the core control and supporting domains add defense in depth.*

| Risk | Element | Relevance |
|---|---|---|
| **LLM01** Prompt Injection | ● AIS Application & Interface Security | Constrains the injection surface at the interface through system-prompt role separation, output-schema validation, cross-modality input filtering, invisible-Unicode stripping, and provenance-labeled channels that mark untrusted content. |
| | ○ IAM Identity & Access Management | Contains a successful injection by granting each tool call least privilege, running actions in the user's scoped context, and requiring human approval before privileged or irreversible operations. |
| | ○ TVM Threat & Vulnerability Management | Requires adaptive red-teaming by adversaries who have already read the defenses, rejecting static-only test suites that miss evolving injection techniques. |
| | ○ STA Supply Chain Management, Transparency, and Accountability | Treats connected tools, plugins, and MCP servers as a software supply-chain surface that must be pinned, signed, and verified before an agent will call them. |
| **LLM02** Sensitive Information Disclosure | ● DSP Data Security and Privacy Lifecycle Management | Governs classification, minimization, retention, encryption, and verifiable erasure of sensitive records across the four-phase data lifecycle the entry is organized around. |
| | ● IAM Identity & Access Management | Enforces authorize-before-retrieval with document- and chunk-level access control inside the index query and per-tenant isolation, since vector similarity |

| Risk | Element | Relevance |
|---|---|---|
| | | alone does not respect access-control lists. |
| | ○ CEK Cryptography, Encryption & Key Management | Applies encryption in transit and at rest, format-preserving encryption for structured identifiers, and vector-store encryption so sensitive data stays unreadable if exposed. |
| | ○ MDS Model Development Security | Reduces training-data memorization through differential-privacy training calibrated to sensitivity, near-duplicate deduplication, verifiable erasure across checkpoints and adapters, and resistance to distillation. |
| | ○ AIS Application & Interface Security | Gates log-probabilities, confidence scores, and explanations on production endpoints, sanitizes outputs with classifiers, separates internal from external routing, and defends streaming side channels. |
| **LLM03** Excessive Agency | ● IAM Identity & Access Management | Minimizes agent permissions, executes tool actions inside the user's OAuth-scoped context, and applies complete mediation so an over-privileged identity cannot exceed its authorization. |
| | ○ AIS Application & Interface Security | Hardens the extension and tool interface with strict input schemas, parameter validation, avoidance of open-ended functionality, and application-security testing. |
| | ○ LOG Logging and Monitoring | Logs and monitors the activity of extensions and downstream systems so undesirable or unauthorized actions are detected quickly. |

| Risk | Element | Relevance |
|---|---|---|
| | ○ TVM Threat & Vulnerability Management | Runs static, dynamic, and interactive application-security testing within the development pipeline to find vulnerabilities in agent extensions before release. |
| **LLM04** Supply Chain | ● STA Supply Chain Management, Transparency, and Accountability | Directly matches the domain with SBOM/AIBOM/ML-SBOM inventory, supplier vetting, and provenance backed by cryptographic signing and transparency logs. |
| | ● MDS Model Development Security | Counters model tampering and backdoors and preserves integrity across adapter merges, format conversion, and quantization through model signing and behavioral evaluation. |
| | ○ TVM Threat & Vulnerability Management | Scans for vulnerable or outdated components in serving frameworks and models and enforces a patching policy across the dependency chain. |
| | ○ CEK Cryptography, Encryption & Key Management | Uses cryptographic model signing, Sigstore and transparency-log entries, file hashes, and edge-model encryption with integrity checks to verify artifact authenticity. |
| | ○ CCC Change Control and Configuration Management | Enforces promotion boundaries with immutable references, policy-based release gates aligned to SLSA, and build-pipeline integrity across CI/CD. |
| **LLM05** Data and Model Poisoning | ● MDS Model Development Security | Secures training and fine-tuning data and model artifacts against poisoning across the development lifecycle. |
| | ● DSP Data Security and Privacy Lifecycle Management | Validates incoming data, checks integrity, and versions datasets |

| Risk | Element | Relevance |
|---|---|---|
| | | so tampered inputs are caught across the data lifecycle. |
| | ○ STA Supply Chain Management, Transparency, and Accountability | Establishes model and dataset lineage through SBOM/ML-BOM records, artifact signing, and vendor vetting to counter distributed poisoned models. |
| | ○ LOG Logging and Monitoring | Applies statistical and AI-based anomaly detection and behavioral-drift monitoring across training, embedding, and inference to surface poisoning effects. |
| | ○ TVM Threat & Vulnerability Management | Runs continuous adversarial red-teaming and trigger-probing to expose hidden backdoors planted during poisoning. |
| **LLM06** Unbounded Consumption | ● AIS Application & Interface Security | Applies front-line interface controls including rate limiting, token- and cost-aware caps, input-size validation, pre-flight token estimation, and authenticated inference endpoints. |
| | ● BCR Business Continuity Management and Operational Resilience | Preserves availability through graceful degradation and dynamic scaling with load balancing when demand or abuse spikes. |
| | ○ IVS Infrastructure & Virtualization Security | Manages resource allocation and hardens inference infrastructure, including disabling unsafe deserialization and reducing the shared-inference side-channel surface. |
| | ○ LOG Logging and Monitoring | Provides continuous cost-attribution monitoring and detection of resource-intensive tool interactions against normal-behavior baselines. |

| Risk | Element | Relevance |
|---|---|---|
| | ○ IAM Identity & Access Management | Scopes quotas and hard spending caps per API key, user, team, and account and authenticates inference endpoints to blunt stolen-credential abuse. |
| **LLM07** Misinformation | ● AIS Application & Interface Security | Separates claim from action, validates tool calls semantically, requires verification signals and structured outputs with mandatory fields, and verifies model assertions at runtime. |
| | ○ TVM Threat & Vulnerability Management | Runs adversarial evaluation and continuous testing against deliberately misleading scenarios to catch misinformation before it reaches users. |
| | ○ LOG Logging and Monitoring | Logs claims, supporting evidence, and outcomes and monitors for misinformation patterns over time. |
| | ○ STA Supply Chain Management, Transparency, and Accountability | Addresses hallucinated-dependency attacks and unsafe generated code by verifying that packages and dependencies the model names actually exist and are trusted. |
| **LLM08** Hidden Context Exposure | ● AIS Application & Interface Security | Treats hidden context as an interface-exposure risk by curating what enters the context window, applying app-layer guardrails, and enforcing independent controls rather than trusting the prompt. |
| | ○ IAM Identity & Access Management | Enforces authorization and access control independently of the model and splits privileges across agents so exposed context cannot grant capability. |
| | ○ CEK Cryptography, Encryption & Key Management | Keeps credentials, tokens, and connection strings out of the |

| Risk | Element | Relevance |
|---|---|---|
| | | prompt by externalizing secrets to a managed store. |
| **LLM09** Vector and Embedding Weaknesses | ● IAM Identity & Access Management | Scopes tenant identity inside the index query, validates server-side, authenticates per-tenant endpoints, and enforces chunk-level access control. |
| | ● DSP Data Security and Privacy Lifecycle Management | Tracks provenance, segregates trust tiers, deletes embeddings with their source, and audits retention, encryption at rest, and backup sensitivity across the embedding lifecycle. |
| | ● LOG Logging and Monitoring | Applies anomaly detection at ingest and retrieval and keeps immutable retrieval logs monitored for tenant-filter bypass and cross-tenant access. |
| | ○ CEK Cryptography, Encryption & Key Management | Encrypts embeddings at rest with keys managed separately from the application layer and treats embedding-API keys as secrets. |
| | ○ AIS Application & Interface Security | Authenticates embedding and similarity-search endpoints as first-class APIs, applies per-tenant rate limits, and withholds raw similarity scores. |
| | ○ STA Supply Chain Management, Transparency, and Accountability | Vets the embedding model against backdoored-encoder risk and tracks provenance of the encoder and the indexed content. |
| **LLM10** Improper Output Handling | ● AIS Application & Interface Security | Validates model responses as untrusted input and applies context-aware output encoding, parameterized queries, and content-security policy before outputs reach downstream interpreters. |

| Risk | Element | Relevance |
| --- | --- | --- |
| | ○ LOG Logging and Monitoring | Logs and monitors model outputs so exploitation patterns in downstream handling are detected. |
| | ○ IAM Identity & Access Management | Treats the model as any other user under a zero-trust posture so its outputs never inherit privileges beyond the end user. |

## OWASP AIVSS (AI Vulnerability Scoring System) — v0.8

*Each row lists the OWASP AIVSS Agentic AI Core Security Risks that the LLM Top 10 entry can produce or feed, with primary marking the risks the entry most directly enables and supporting marking secondary paths; AIVSS then scores the severity of those agentic risks.*

| Risk | Element | Relevance |
|---|---|---|
| **LLM01** Prompt Injection | ● AIVSS-10 Agent Goal and Instruction Manipulation | A crafted message overrides the system prompt's role and capability limits, redirecting the model to act outside its intended scope. |
| | ● AIVSS-1 Agentic AI Tool Misuse | Injected input drives unauthorized invocation of tools the agent is permitted to call, from file system and shell to email and cloud APIs. |
| | ● AIVSS-2 Agent Access Control Violation | The agent reads attacker-planted text while operating under the user's elevated credentials and performs privileged actions the attacker could not reach directly. |
| | ● AIVSS-6 Agent Memory and Context Manipulation | An injection written to long-term memory or a RAG corpus taints every later session that reads the poisoned store, persisting the compromise across conversations. |
| | ○ AIVSS-7 Insecure Agent Critical Systems Interaction | Where the agent holds shell, file-system, or cloud-API access, an injection escalates into arbitrary command execution and destructive actions on the host. |
| | ○ AIVSS-3 Agent Cascading Failures | Tool outputs re-enter the shared context window, letting one injection chain into further tool calls or self-replicate across steps. |
| | ○ AIVSS-4 Agent Orchestration and Multi-Agent Exploitation | An injection can self-replicate across agents, spreading from |

| Risk | Element | Relevance |
|---|---|---|
| | | one compromised agent to its peers in a multi-agent workflow. |
| **LLM03** Excessive Agency | ● AIVSS-1 Agentic AI Tool Misuse | Extensions carry functions beyond what the task needs, such as read-only access bundled with modify, delete, or send, so a malfunctioning model invokes capabilities it was never meant to use. |
| | ● AIVSS-2 Agent Access Control Violation | Extensions connect to downstream systems with over-broad or generic high-privileged identities, letting the agent act far beyond the user's own authorization scope. |
| | ● AIVSS-7 Insecure Agent Critical Systems Interaction | An open-ended shell-command extension that fails to filter unintended commands, or an over-autonomous coding agent, can execute damaging actions and destroy production infrastructure. |
| | ○ AIVSS-3 Agent Cascading Failures | Without rate limits or circuit breakers, runaway extension invocation compounds into cascading downstream failures. |
| | ○ AIVSS-4 Agent Orchestration and Multi-Agent Exploitation | A malicious or compromised peer agent can trigger damaging actions, and failing to preserve user context across chained agent calls widens the blast radius. |
| | ○ AIVSS-10 Agent Goal and Instruction Manipulation | Direct or indirect prompt injection, such as a crafted incoming email, is the trigger that redirects the agent into unwanted privileged actions. |
| **LLM04** Supply Chain | ○ AIVSS-8 Agent Supply Chain and Dependency Risk | Compromised third-party packages, pre-trained models, LoRA adapters, and conversion or |

| Risk | Element | Relevance |
|---|---|---|
| | | merge pipelines are the dependency substrate an agent inherits, so a tampered model-supply-chain artifact becomes the agent's supply chain too. |
| **LLM05** Data and Model Poisoning | ○ AIVSS-6 Agent Memory and Context Manipulation | Attackers embed hidden instructions in web content that poison an agent's persistent memory or recommendations across sessions, so the agent begins prioritizing attacker-controlled logic. |
| | ○ AIVSS-8 Agent Supply Chain and Dependency Risk | Backdoored models and unsafe-serialization artifacts distributed through public repositories carry hidden triggers that a downstream agent inherits when it loads them. |
| | ○ AIVSS-3 Agent Cascading Failures | Poisoned inputs propagate across multi-agent and enterprise workflows, and shared embeddings or memory layers spread contamination from one tenant to others. |
| **LLM06** Unbounded Consumption | ○ AIVSS-1 Agentic AI Tool Misuse | A malicious tool forces an agent into recursive or infinite tool-calling loops, driving token consumption and cost far beyond the task's needs. |
| | ○ AIVSS-3 Agent Cascading Failures | Agentic and MCP protocols amplify a single request into cascading downstream operations, so one task fans out into many tool calls that exhaust budget and availability. |
| | ○ AIVSS-8 Agent Supply Chain and Dependency Risk | An attacker publishes a malicious tool, such as a skill on an open-source repository, that developers incorporate as an agent dependency, driving token exhaustion once integrated. |

| Risk | Element | Relevance |
|---|---|---|
| **LLM07** Misinformation | ● AIVSS-3 Agent Cascading Failures | Incorrect state or evidence produced by one agent and trusted by another propagates across a multi-agent workflow into a compounding, high-impact failure. |
| | ○ AIVSS-1 Agentic AI Tool Misuse | Because model outputs drive tool calls, an incorrect state inference triggers unintended actions unless invocations are validated against real-world state and intent. |
| | ○ AIVSS-7 Insecure Agent Critical Systems Interaction | A false conclusion, such as wrongly approving a refund or firing a false alert, drives an automated high-impact action that causes financial loss or operational disruption. |
| **LLM08** Hidden Context Exposure | ○ AIVSS-2 Agent Access Control Violation | Disclosed permission rules and user-role directives from hidden context reveal the authorization model, inviting probing that bypasses controls the application should enforce independently of the LLM. |
| | ○ AIVSS-1 Agentic AI Tool Misuse | Extracting the hidden tool list and parameter schemas gives an attacker concrete targets to steer the application toward specific tool calls. |
| **LLM09** Vector and Embedding Weaknesses | ○ AIVSS-6 Agent Memory and Context Manipulation | Retrieval-time poisoning manipulates the geometry of vector-backed agent memory so attacker content is retrieved as trusted context, while non-geometric memory poisoning is deferred to the agentic framework. |
| | ○ AIVSS-8 Agent Supply Chain and Dependency Risk | A backdoored embedding model corrupts the geometry of everything ingested, and |

| Risk | Element | Relevance |
|---|---|---|
| | | vulnerable vector-database dependencies compound the risk since a leaked index is recoverable to source documents. |
| | ○ AIVSS-2 Agent Access Control Violation | Similarity search runs across the full shared index before tenant access control is applied, so an attacker infers or reaches other tenants' content that authorization should have blocked. |
| **LLM10** Improper Output Handling | ○ AIVSS-7 Insecure Agent Critical Systems Interaction | Unvalidated model output reaching a shell, exec/eval, SQL, or file-path sink yields remote code execution, SQL injection, or path traversal on backend systems. |
| | ○ AIVSS-1 Agentic AI Tool Misuse | A general-purpose LLM's unvalidated response passed to a privileged extension causes the extension to be misused, for example forced into an unintended shutdown. |

## Framework Sources & Versions

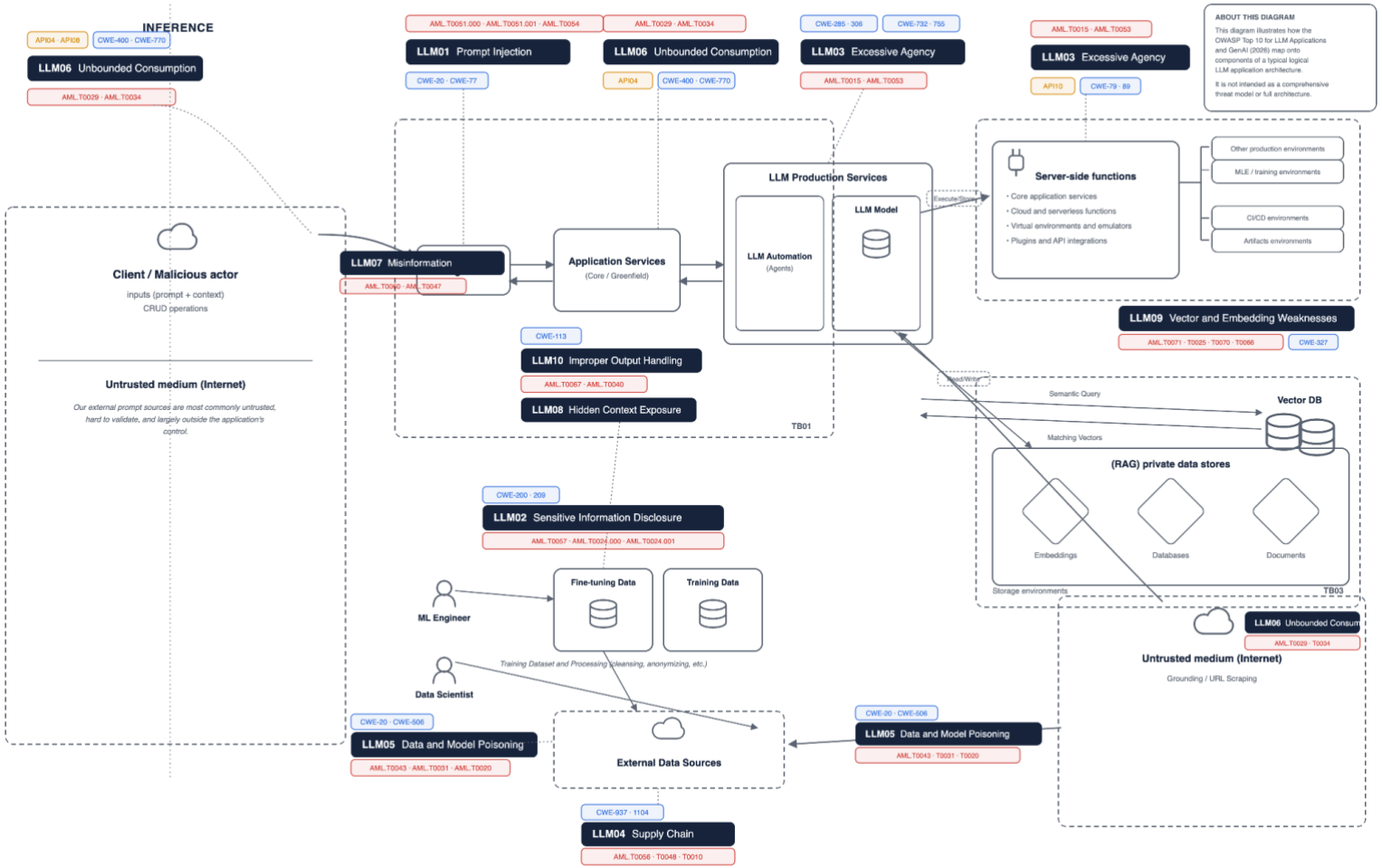
| Framework | Version | Source |
|---|---|---|
| OWASP Top 10 for Agentic Applications (ASI) | 2026 (announced 2025-12-09) | https://genai.owasp.org/resource/owasp-top-10-for-agentic-applications-for-2026/ |
| OWASP GenAI Data Security 2026 (DSGAI) | v1.0 (2026-03-17) | https://genai.owasp.org/resource/owasp-genai-data-security-risks-mitigations-2026/ |
| MITRE ATLAS | content v2026.06 (format-version 6.0.0) | https://raw.githubusercontent.com/mitre-atlas/atlas-data/main/dist/v6/ATLAS-2026.06.yaml |
| MITRE ATT&CK | v19.1 | https://attack.mitre.org/versions/v19/tactics/enterprise/ |
| MITRE CWE (Common Weakness Enumeration) | 4.20 | https://cwe.mitre.org/ |
| NIST AI 600-1 (Generative AI Profile) | v1.0 (July 2024) | https://nvlpubs.nist.gov/nistpubs/ai/NIST.AI.600-1.pdf |
| NIST AI RMF (AI 100-1) | v1.0 (2023) | https://airc.nist.gov/airmf-resources/airmf/5-sec-core/ |
| CSA AI Controls Matrix (AICM) | v1.1 (2026-06-22) | https://cloudsecurityalliance.org/artifacts/ai-controls-matrix-v1-1 |
| OWASP AIVSS (AI Vulnerability Scoring System) | v0.8 | https://aivss.owasp.org/assets/publications/AIVSS%20Scoring%20System%20For%20OWASP%20Agentic%20AI%20Core%20Security%20Risks%20v0.8.pdf |

# Appendix B: LLM Application Architecture and Threat Modeling

# OWASP Top 10 for LLM Applications and Generative AI — 2026 Version

Example LLM Application and Basic Threat Modeling

Original Source: Ads Dawson (GreenTemperTantrum)




Ten risks mapped to where they surface in a typical LLM application.

MITRE ATLAS technique references follow the OWASP LLM Top 10 × ATLAS crosswalk, current as of 2026-Q2.

TB = Trust Boundary

# References

References for the OWASP Top 10 for LLM Applications (2026), grouped by risk entry and formatted in APA-7. A source cited by more than one entry appears in full under each, with an `[also cited in ...]` note.

## LLM01: Prompt Injection


- Amazon Web Services. (2025a, July). *AWS-2025-015: Amazon Q VS Code extension supply-chain incident* [Security bulletin]. https://aws.amazon.com/security/security-bulletins/AWS-2025-015/
- Amazon Web Services. (2025b, July). *AWS-2025-019: Amazon Q runtime injection* [Security bulletin]. https://aws.amazon.com/security/security-bulletins/AWS-2025-019/
- Chao, P., Debenedetti, E., Robey, A., Andriushchenko, M., Croce, F., Sehwag, V., Dobriban, E., Flammarion, N., Pappas, G. J., Tramèr, F., Hassani, H., & Wong, E. (2024). *JailbreakBench: An open robustness benchmark for jailbreaking large language models*. arXiv. https://arxiv.org/abs/2404.01318
- Chen, R., Cui, S., Huang, X., Pan, C., Huang, V. S.-J., Zhang, Q., Ouyang, X., Zhang, Z., Wang, H., & Huang, M. (2025). JPS: Jailbreak multimodal large language models with collaborative visual perturbation and textual steering. In *Proceedings of the 33rd ACM International Conference on Multimedia (MM '25)*. Association for Computing Machinery. https://dl.acm.org/doi/10.1145/3746027.3754561
- Chen, S., Piet, J., Sitawarin, C., & Wagner, D. (2025). StruQ: Defending against prompt injection with structured queries. In *Proceedings of the 34th USENIX Security Symposium (USENIX Security 25)*. USENIX Association. https://www.usenix.org/system/files/usenixsecurity25-chen-sizhe.pdf
- Clusmann, J., Ferber, D., Wiest, I. C., Schneider, C. V., Brinker, T. J., Foersch, S., Truhn, D., & Kather, J. N. (2025). Prompt injection attacks on vision language models in oncology. *Nature Communications, 16*, Article 1239. https://www.nature.com/articles/s41467-024-55631-x
- Cybersecurity and Infrastructure Security Agency, Federal Bureau of Investigation, National Security Agency, & Australian Cyber Security Centre. (2025, December). *Principles for the secure integration of AI in operational technology*. https://www.cisa.gov/sites/default/files/2025-12/joint-guidance-principles-for-the-secure-integration-of-artificial-intelligence-in-operational-technology-508c.pdf
- Debenedetti, E., Shumailov, I., Fan, T., Hayes, J., Carlini, N., Fabian, D., Kern, C., Shi, C., Terzis, A., & Tramèr, F. (2025). *Defeating prompt injections by design*. arXiv. https://arxiv.org/abs/2503.18813
- Debenedetti, E., Zhang, J., Balunović, M., Beurer-Kellner, L., Fischer, M., & Tramèr, F. (2024). *AgentDojo: A dynamic environment to evaluate prompt injection attacks and defenses for LLM agents*. arXiv. https://arxiv.org/abs/2406.13352
- General Analysis. (2025, July). *Supabase MCP can leak your entire SQL database* [Blog post]. https://generalanalysis.com/blog/supabase-mcp-blog

- Greshake, K. (2023). *Inject my PDF: Prompt injection for your resume* [Blog post]. https://kai-greshake.de/posts/inject-my-pdf
- Greshake, K., Abdelnabi, S., Mishra, S., Endres, C., Holz, T., & Fritz, M. (2023). *Not what you've signed up for: Compromising real-world LLM-integrated applications with indirect prompt injection*. arXiv. https://arxiv.org/abs/2302.12173
- Hackett, W., Birch, L., Trawicki, S., Suri, N., & Garraghan, P. (2025). *Bypassing LLM guardrails: An empirical analysis of evasion attacks against prompt injection and jailbreak detection systems*. arXiv. https://arxiv.org/abs/2504.11168
- INCIBE-CERT. (2024). *CVE-2024-5184: EmailGPT* [Vulnerability advisory]. https://www.incibe.es/en/incibe-cert/early-warning/vulnerabilities/cve-2024-5184
- Invariant Labs. (2025, May). *GitHub MCP server vulnerability* [Blog post]. https://invariantlabs.ai/blog/mcp-github-vulnerability
- Koi Security. (2025, September). *Postmark-MCP npm malicious backdoor: Email theft* [Blog post]. https://www.koi.ai/blog/postmark-mcp-npm-malicious-backdoor-email-theft
- Labunets, A., Pandya, N. V., Hooda, A., Fu, X., & Fernandes, E. (2025). *Fun-tuning: Characterizing the vulnerability of proprietary LLMs to optimization-based prompt injection attacks via the fine-tuning interface*. arXiv. https://arxiv.org/abs/2501.09798
- Meta AI. (2025, October 31). *Agents rule of two: A practical approach to AI agent security* [Blog post]. https://ai.meta.com/blog/practical-ai-agent-security/ `[also cited in LLM03]`
- Microsoft Research. (2025). *Defending against indirect prompt injection attacks with spotlighting*. https://www.microsoft.com/en-us/research/publication/defending-against-indirect-prompt-injection-attacks-with-spotlighting/
- Microsoft Security Response Center. (2025, March). *Announcing the winners of the Adaptive Prompt Injection Challenge — LLMail-Inject* [Blog post]. https://www.microsoft.com/en-us/msrc/blog/2025/03/announcing-the-winners-of-the-adaptive-prompt-injection-challenge-llmail-inject/
- MITRE. (n.d.). *AI agent context poisoning: Memory* (AML.T0080.000) [ATLAS technique]. MITRE ATLAS. Retrieved July 11, 2026, from https://atlas.mitre.org/techniques/AML.T0080.000
- Nasr, M., Carlini, N., Sitawarin, C., Schulhoff, S. V., Hayes, J., Ilie, M., Pluto, J., Song, S., Chaudhari, H., Shumailov, I., Thakurta, A., Xiao, K. Y., Terzis, A., & Tramèr, F. (2025). *The attacker moves second: Stronger adaptive attacks bypass defenses against LLM jailbreaks and prompt injections*. arXiv. https://arxiv.org/abs/2510.09023
- National Cyber Security Centre. (2025, December). *Prompt injection is not SQL injection* [Blog post]. https://www.ncsc.gov.uk/blog-post/prompt-injection-is-not-sql-injection
- National Institute of Standards and Technology. (2025). *Adversarial machine learning: A taxonomy and terminology of attacks and mitigations* (NIST AI 100-2e2025). https://csrc.nist.gov/pubs/ai/100/2/e2025/final
- National Institute of Standards and Technology. (n.d.). *CVE-2025-32711 detail*. National Vulnerability Database. Retrieved July 11, 2026, from https://nvd.nist.gov/vuln/detail/CVE-2025-32711
- Noma Security. (2025). *Why the Rule of Two can't protect your agents* [Blog post]. https://noma.security/blog/mcp-servers-agentic-risk-and-the-framework-that-protects-it/

- OWASP GenAI Security Project. (2025). *OWASP Top 10 for LLM Applications — LLM01:2025 Prompt injection*. https://genai.owasp.org/llmrisk/llm01-prompt-injection/
- Reddy, P., & Gujral, A. S. (2025). *EchoLeak: The first real-world zero-click prompt injection exploit in a production LLM system*. arXiv. https://arxiv.org/abs/2509.10540
- Rehberger, J. (2024, August). *M365 Copilot prompt injection, tool invocation and data exfil using ASCII smuggling* [Blog post]. Embrace The Red. https://embracethered.com/blog/posts/2024/m365-copilot-prompt-injection-tool-invocation-and-data-exfil-using-ascii-smuggling/
- Rehberger, J. (2025a, August). *GitHub Copilot: Remote code execution via prompt injection (CVE-2025-53773)* [Blog post]. Embrace The Red. https://embracethered.com/blog/posts/2025/github-copilot-remote-code-execution-via-prompt-injection/ `[also cited in LLM10]`
- Rehberger, J. (2025b, February). *Hacking Gemini's memory with prompt injection and delayed tool invocation* [Blog post]. Embrace The Red. https://embracethered.com/blog/posts/2025/google-gemini-memory-persistence-prompt-injection/
- Rehberger, J. (2025c). *Sneaky bits & ASCII smuggler updates* [Blog post]. Embrace The Red. https://embracethered.com/blog/posts/2025/sneaky-bits-and-ascii-smuggler/
- Toulas, B. (2025, September 25). *Unofficial Postmark MCP npm package silently stole users' emails*. BleepingComputer. https://www.bleepingcomputer.com/news/security/unofficial-postmark-mcp-npm-silently-stole-users-emails/
- Willison, S. (2025, June 16). *The lethal trifecta for AI agents: Private data, untrusted content, and external communication* [Blog post]. https://simonwillison.net/2025/Jun/16/the-lethal-trifecta/
- Zou, A., Wang, Z., Carlini, N., Nasr, M., Kolter, J. Z., & Fredrikson, M. (2023). *Universal and transferable adversarial attacks on aligned language models*. arXiv. https://arxiv.org/abs/2307.15043
- Zou, W., Geng, R., Wang, B., & Jia, J. (2025). PoisonedRAG: Knowledge corruption attacks to retrieval-augmented generation of large language models. In *Proceedings of the 34th USENIX Security Symposium (USENIX Security 25)*. USENIX Association. https://www.usenix.org/system/files/usenixsecurity25-zou-poisonedrag.pdf `[also cited in LLM07, LLM09]`

## LLM02: Sensitive Information Disclosure

- BeyondScale. (2026, May). *Vector database security: RAG compliance & monitoring*. https://beyondscale.tech/blog/vector-database-security-rag-compliance-monitoring
- Boenisch, F., Dziedzic, A., Schuster, R., Shamsabadi, A. S., Shumailov, I., & Papernot, N. (2021). *When the curious abandon honesty: Federated learning is not private*. arXiv. https://arxiv.org/abs/2112.02918
- Carlini, N., Paleka, D., Dvijotham, K. D., Steinke, T., Hayase, J., Cooper, A. F., Lee, K., Jagielski, M., Nasr, M., Conmy, A., Yona, I., Wallace, E., Rolnick, D., & Tramèr, F. (2024). *Stealing part of a production language model*. arXiv. https://arxiv.org/abs/2403.06634 `[also cited in LLM06]`
- Check Point Research. (2026, February). *ChatGPT data leakage via a hidden outbound runtime channel*. https://research.checkpoint.com/2026/chatgpt-data-leakage-via-a-hidden-outbound-channel-in-the-code-execution-runtime

- Dong, T., Meng, Y., Li, S., Chen, G., Liu, Z., & Zhu, H. (2025). *Depth gives a false sense of privacy: LLM internal states inversion*. arXiv. https://arxiv.org/abs/2507.16372
- Fu, W., Wang, H., Gao, C., Liu, G., Li, Y., & Jiang, T. (2024). *Membership inference attacks against fine-tuned large language models via self-prompt calibration* [Conference poster]. Neural Information Processing Systems (NeurIPS 2024). https://neurips.cc/virtual/2024/poster/95327
- The Guardian. (2026, April 1). *Anthropic Claude Code source leak*. https://www.theguardian.com/technology/2026/apr/01/anthropic-claudes-code-leaks-ai
- InstaTunnel. (2026, February). *Differential privacy reversal via LLM feedback*. https://instatunnel.my/blog/differential-privacy-reversal-via-llm-feedback-the-silent-killer-of-data-anonymization
- Jiang, C., Pan, X., Hong, G., Bao, C., Chen, Y., & Yang, M. (2024). *Feedback-guided extraction of knowledge base from retrieval-augmented LLM applications*. arXiv. https://arxiv.org/abs/2411.14110
- Lakshmanan, R. (2026, March 26). *Claude extension flaw enabled zero-click XSS prompt injection via any website*. The Hacker News. https://thehackernews.com/2026/03/claude-extension-flaw-enabled-zero.html
- Lin, Y., Zhang, Q., Ruan, W., Zhang, D., Hong, J., Wu, Y., Xia, H., Mao, Y., & Zhong, S. (2026). *Towards privacy-preserving LLM inference via covariant obfuscation (Technical report)*. arXiv. https://arxiv.org/abs/2603.01499
- McDonald, G., & Bar Or, J. (2025). *Whisper Leak: A side-channel attack on large language models*. arXiv. https://arxiv.org/abs/2511.03675
- Nasr, M., Carlini, N., Hayase, J., Jagielski, M., Cooper, A. F., Ippolito, D., Choquette-Choo, C. A., Wallace, E., Tramèr, F., & Lee, K. (2023). *Scalable extraction of training data from (production) language models*. arXiv. https://arxiv.org/abs/2311.17035
- OWASP GenAI Security Project. (2026). *OWASP GenAI data security 2026* (Version 1.0). https://genai.owasp.org/
- Panda, A., Choquette-Choo, C. A., Zhang, Z., Yang, Y., & Mittal, P. (2024). *Teach LLMs to phish: Stealing private information from language models*. arXiv. https://arxiv.org/abs/2403.00871
- Unit 42. (2026, March). *Gemini Live in Chrome hijacking (CVE-2026-0628)*. Palo Alto Networks. https://unit42.paloaltonetworks.com/gemini-live-in-chrome-hijacking
- Weiss, R., Ayzenshteyn, D., Amit, G., & Mirsky, Y. (2024). *What was your prompt? A remote keylogging attack on AI assistants*. arXiv. https://arxiv.org/abs/2403.09751
- Wiz. (2025, January). *Wiz Research uncovers exposed DeepSeek database leak*. https://www.wiz.io/blog/wiz-research-uncovers-exposed-deepseek-database-leak
- Wu, G., Zhang, Z., Zhang, Y., Wang, W., Niu, J., Wu, Y., & Zhang, Y. (2025). *I know what you asked: Prompt leakage via KV-cache sharing in multi-tenant LLM serving* [Conference paper]. Network and Distributed System Security Symposium (NDSS 2025). https://www.ndss-symposium.org/ndss-paper/i-know-what-you-asked-prompt-leakage-via-kv-cache-sharing-in-multi-tenant-llm-serving/

## LLM03: Excessive Agency


- Bort, J. (2026, February 23). *A Meta AI security researcher said an OpenClaw agent ran amok on her inbox*. TechCrunch. https://techcrunch.com/2026/02/23/a-meta-ai-security-researcher-said-an-openclaw-agent-ran-amok-on-her-inbox/
- Ferreira, B. (2026, March 7). *Claude Code deletes developers' production setup, including its database and snapshots — 2.5 years of records were nuked in an instant*. Tom's Hardware. https://www.tomshardware.com/tech-industry/artificial-intelligence/claude-code-deletes-developers-production-setup-including-its-database-and-snapshots-2-5-years-of-records-were-nuked-in-an-instant
- Kovacs, E. (2026, April 1). *Google addresses Vertex security issues after researchers weaponize AI agents*. SecurityWeek. https://www.securityweek.com/google-addresses-vertex-security-issues-after-researchers-weaponize-ai-agent/
- Kundel, D. (2024, October 10). *Rogue agents: Stop AI from misusing your APIs*. Twilio. https://www.twilio.com/en-us/blog/rogue-ai-agents-secure-your-apis
- Lucas, J. (2024, December 16). *Sandboxing agentic AI workflows with WebAssembly*. NVIDIA Developer Blog. https://developer.nvidia.com/blog/sandboxing-agentic-ai-workflows-with-webassembly/
- Meta AI. (2025, October 31). *Agents rule of two: A practical approach to AI agent security* [Blog post]. https://ai.meta.com/blog/practical-ai-agent-security/ `[also cited in LLM01]`
- NVIDIA. (n.d.). *NeMo-Guardrails: Interface guidelines*. GitHub. https://github.com/NVIDIA/NeMo-Guardrails/blob/main/docs/security/guidelines.md
- Rehberger, J. (2023, May 28). *ChatGPT plugin exploit explained: From prompt injection to accessing private data*. Embrace The Red. https://embracethered.com/blog/posts/2023/chatgpt-cross-plugin-request-forgery-and-prompt-injection./


## LLM04: Supply Chain


- Cisco Talos. (2024). *llama.cpp GGUF library gguf_fread_str heap-based buffer overflow vulnerability (TALOS-2024-1913)*. https://talosintelligence.com/vulnerability_reports/TALOS-2024-1913
- Cohen, D. (2025, December 2). *PyTorch users at risk: Unveiling 3 zero-day PickleScan vulnerabilities*. JFrog. https://jfrog.com/blog/unveiling-3-zero-day-vulnerabilities-in-picklescan/
- Egashira, K., Staab, R., Vero, M., He, J., & Vechev, M. (2025). *Mind the gap: A practical attack on GGUF quantization* (arXiv:2505.23786). arXiv. https://arxiv.org/abs/2505.23786
- GitHub Security Advisories. (2025, April 17). *PyTorch torch.load weights_only bypass (CVE-2025-32434, GHSA-53q9-r3pm-6pq6)*. https://github.com/pytorch/pytorch/security/advisories/GHSA-53q9-r3pm-6pq6
- HiddenLayer. (2024, February 21). *Hijacking Safetensors conversion on Hugging Face*. https://hiddenlayer.com/research/silent-sabotage/
- Huynh, D., & Hardouin, J. (2023, July 9). *PoisonGPT: How we hid a lobotomized LLM on Hugging Face to spread fake news*. Mithril Security. https://blog.mithrilsecurity.io/poisongpt-how-we-hid-a-lobotomized-llm-on-hugging-face-to-spread-fake-news

- Lumelsky, A., & Elbaz, G. (2025, November 18). *ShadowRay 2.0: Attackers turn AI against itself in global campaign that hijacks AI into self-propagating botnet*. Oligo Security. https://www.oligo.security/blog/shadowray-2-0-attackers-turn-ai-against-itself-in-global-campaign-that-hijacks-ai-into-self-propagating-botnet
- MITRE. (n.d.). *AI supply chain compromise (AML.T0010)*. MITRE ATLAS. Retrieved July 11, 2026, from https://atlas.mitre.org/techniques/AML.T0010
- OASIS Open. (2025, November 18). *Coalition for Secure AI releases two actionable frameworks for AI model signing and incident response*. https://www.oasis-open.org/2025/11/18/coalition-for-secure-ai-releases-two-actionable-frameworks-for-ai-model-signing-and-incident-response/
- Open Source Security Foundation. (2025, April 4). *Launch of Model Signing v1.0: OpenSSF AI/ML Working Group secures the machine learning supply chain*. https://openssf.org/blog/2025/04/04/launch-of-model-signing-v1-0-openssf-ai-ml-working-group-secures-the-machine-learning-supply-chain/
- Open Source Security Foundation. (n.d.). *Supply-chain Levels for Software Artifacts (SLSA)*. Retrieved July 11, 2026, from https://slsa.dev/
- OWASP CycloneDX. (n.d.). *Machine learning bill of materials (AI/ML-BOM)*. Retrieved July 11, 2026, from https://cyclonedx.org/capabilities/mlbom/ [also cited in LLM05]
- OWASP GenAI Security Project. (2025, December 18). *Evolving AI transparency: The journey of the AIBOM Generator and its new home at OWASP*. https://genai.owasp.org/2025/12/18/evolving-ai-transparency-the-journey-of-the-aibom-generator-and-its-new-home-at-owasp/
- OWASP GenAI Security Project. (2026). *OWASP Top 10 for Agentic Applications (2026)*. https://genai.owasp.org/resource/owasp-top-10-for-agentic-applications-for-2026/
- Python Package Index. (2024, December 11). *Supply-chain attack analysis: Ultralytics*. PyPI Blog. https://blog.pypi.org/posts/2024-12-11-ultralytics-attack-analysis/
- PyTorch Foundation. (2022, December 31). *Compromised PyTorch-nightly dependency chain between December 25th and December 30th, 2022*. https://pytorch.org/blog/compromised-nightly-dependency/
- Saraf, I., & Balassiano, O. (2025, September 3). *Model namespace reuse: An AI supply-chain attack exploiting model name trust*. Unit 42, Palo Alto Networks. https://unit42.paloaltonetworks.com/model-namespace-reuse/
- Spracklen, J., Wijewickrama, R., Sakib, A. H. M. N., Maiti, A., Viswanath, B., & Jadliwala, M. (2025). We have a package for you! A comprehensive analysis of package hallucinations by code generating LLMs. In *Proceedings of the 34th USENIX Security Symposium (USENIX Security 25)*. USENIX Association. https://www.usenix.org/system/files/usenixsecurity25-spracklen.pdf [also cited in LLM07]
- Tamari, S., & Tzadik, S. (2024, April 4). *Wiz Research finds architecture risks that may compromise AI-as-a-Service providers*. Wiz. https://www.wiz.io/blog/wiz-and-hugging-face-address-risks-to-ai-infrastructure
- Wickens, E., Schulz, K., & Bonner, T. (2024, October 10). *ShadowLogic: Persistent no-code backdoors in AI computational graphs*. HiddenLayer. https://hiddenlayer.com/innovation-hub/shadowlogic/
- Wiz. (2024). *Probllama: Ollama remote code execution vulnerability (CVE-2024-37032)*. https://www.wiz.io/blog/probllama-ollama-vulnerability-cve-2024-37032

- Zanki, K. (2025, February 6). *Malicious ML models discovered on Hugging Face platform (nullifAI)*. ReversingLabs. https://www.reversinglabs.com/blog/rl-identifies-malware-ml-model-hosted-on-hugging-face
- Zhan, Q., Fang, R., Bindu, R., Gupta, A., Hashimoto, T., & Kang, D. (2023). *Removing RLHF protections in GPT-4 via fine-tuning* (arXiv:2311.05553). arXiv. https://arxiv.org/abs/2311.05553

## LLM05: Data and Model Poisoning

- OWASP CycloneDX. (n.d.). *Machine learning bill of materials (AI/ML-BOM)*. Retrieved July 11, 2026, from https://cyclonedx.org/capabilities/mlbom/ [also cited in LLM04]
- DVC. (n.d.). *DVC: Data version control* [Documentation]. https://dvc.org/doc
- F5. (2025). *Data and model poisoning* [OWASP Top 10 for LLM applications guide (2025)]. https://my.f5.com/manage/s/article/K000149782
- Fogel, A., Hofman, O., Cohen, E., & Vainshtein, R. (2026). *Inference-time backdoors via chat templates: From LLM supply chains to agentic system compromise* (arXiv:2602.04653). arXiv. https://arxiv.org/abs/2602.04653
- GitHub. (2026, March 26). *Giskard Agents have server-side template injection via ChatWorkflow.chat() using non-sandboxed Jinja2 Environment* (GHSA-frv4-x25r-588m). GitHub Advisory Database. https://github.com/advisories/GHSA-frv4-x25r-588m
- Hubinger, E., Denison, C., Mu, J., Lambert, M., Tong, M., MacDiarmid, M., Lanham, T., Ziegler, D. M., Maxwell, T., Cheng, N., Jermyn, A., Askell, A., Radhakrishnan, A., Anil, C., Duvenaud, D., Ganguli, D., Barez, F., Clark, J., Ndousse, K., … Perez, E. (2024). *Sleeper agents: Training deceptive LLMs that persist through safety training* (arXiv:2401.05566). arXiv. https://arxiv.org/abs/2401.05566
- JFrog Security Research. (n.d.). *GGUF-SSTI*. https://research.jfrog.com/model-threats/gguf-ssti/
- MITRE. (n.d.). *MITRE ATLAS: Adversarial threat landscape for artificial-intelligence systems*. https://atlas.mitre.org
- National Institute of Standards and Technology. (2023). *Artificial intelligence risk management framework (AI RMF 1.0)* (NIST AI 100-1). https://www.nist.gov/itl/ai-risk-management-framework
- Souly, A., Rando, J., Chapman, E., Davies, X., Hasircioglu, B., Shereen, E., Mougan, C., Mavroudis, V., Jones, E., Hicks, C., Carlini, N., Gal, Y., & Kirk, R. (2025). *Poisoning attacks on LLMs require a near-constant number of poison samples* (arXiv:2510.07192). arXiv. https://arxiv.org/abs/2510.07192
- Zhang, B., Chen, Y., Liu, Z., Nie, L., Li, T., Liu, Z., & Fang, M. (2025). *Practical poisoning attacks against retrieval-augmented generation* (arXiv:2504.03957). arXiv. https://arxiv.org/abs/2504.03957

## LLM06: Unbounded Consumption

- Akkus, A. (2024, December 27). *You wouldn't download an AI: Extracting AI models from mobile apps*. Altay's Blog. https://altayakkus.substack.com/p/you-wouldnt-download-an-ai
- Anthropic. (n.d.). *Vision* [Build with Claude documentation]. Retrieved July 11, 2026, from https://docs.anthropic.com/en/docs/build-with-claude/vision

- Awan, A. A. (2023, March 24). *How watermarking can help mitigate the potential risks of LLMs?* KDnuggets. https://www.kdnuggets.com/2023/03/watermarking-help-mitigate-potential-risks-llms.html
- Carlini, N., Paleka, D., Dvijotham, K. D., Steinke, T., Hayase, J., Cooper, A. F., Lee, K., Jagielski, M., Nasr, M., Conmy, A., Yona, I., Wallace, E., Rolnick, D., & Tramèr, F. (2024). *Stealing part of a production language model* (arXiv:2403.06634). arXiv. https://arxiv.org/abs/2403.06634 `[also cited in LLM02]`
- DeepLearning.AI. (2023, March 15). *Runaway LLaMA: How Meta's LLaMA NLP model leaked*. The Batch. https://www.deeplearning.ai/the-batch/how-metas-llama-nlp-model-leaked/
- Dong, B., Feng, H., & Wang, Q. (2026). *Clawdrain: Exploiting tool-calling chains for stealthy token exhaustion in OpenClaw agents* (arXiv:2603.00902). arXiv. https://arxiv.org/abs/2603.00902
- Gao, H., Zhang, Y., Zhou, Z., Jiang, L., Meng, F., Xiao, Y., Sun, L., Wang, K., Liu, Y., & Feng, J. (2025). *Resource consumption red-teaming for large vision-language models* (arXiv:2507.18053). arXiv. https://arxiv.org/abs/2507.18053
- Gao, K., Pang, T., Du, C., Yang, Y., Xia, S.-T., & Lin, M. (2024). *Denial-of-service poisoning attacks against large language models* (arXiv:2410.10760). arXiv. https://arxiv.org/abs/2410.10760
- Jiang, W., Li, H., Xu, G., Zhang, T., & Lu, R. (2024). A comprehensive defense framework against model extraction attacks. *IEEE Transactions on Dependable and Secure Computing*, *21*(2), 685–700. https://ieeexplore.ieee.org/document/10080996
- Li, Y., Wang, J., Zhu, H., Lin, J., Chang, S., & Guo, M. (2025). *ThinkTrap: Denial-of-service attacks against black-box LLM services via infinite thinking* (arXiv:2512.07086). arXiv. https://arxiv.org/abs/2512.07086
- moohax, & monoxgas. (2023, March 31). *Proof Pudding (CVE-2019-20634)* (AVID-2023-V009). AI Vulnerability Database. https://avidml.org/database/avid-2023-v009/
- Nevo, S., Lahav, D., Karpur, A., Bar-On, Y., Bradley, H. A., & Alstott, J. (2024). *Securing AI model weights: Preventing theft and misuse of frontier models* (RR-A2849-1). RAND Corporation. https://www.rand.org/content/dam/rand/pubs/research_reports/RRA2800/RRA2849-1/RAND_RRA2849-1.pdf
- Radosevich, B., & Halloran, J. (2025). *MCP safety audit: LLMs with the Model Context Protocol allow major security exploits* (arXiv:2504.03767). arXiv. https://arxiv.org/abs/2504.03767
- Sev, C. (2023, August 30). *Security update: Incident involving unauthorized admin access*. Sourcegraph. https://about.sourcegraph.com/blog/security-update-august-2023
- Shumailov, I., Zhao, Y., Bates, D., Papernot, N., Mullins, R., & Anderson, R. (2020). *Sponge examples: Energy-latency attacks on neural networks* (arXiv:2006.03463). arXiv. https://arxiv.org/abs/2006.03463
- Taori, R., Gulrajani, I., Zhang, T., Dubois, Y., Li, X., Guestrin, C., Liang, P., & Hashimoto, T. B. (2023, March 13). *Alpaca: A strong, replicable instruction-following model*. Stanford Center for Research on Foundation Models. https://crfm.stanford.edu/2023/03/13/alpaca.html

## LLM07: Misinformation


- National Institute of Standards and Technology. (2024). *Artificial intelligence risk management framework: Generative artificial intelligence profile* (NIST AI 600-1). U.S. Department of Commerce. https://nvlpubs.nist.gov/nistpubs/ai/NIST.AI.600-1.pdf
- OpenAI. (2025, September 5). *Why language models hallucinate*. https://openai.com/index/why-language-models-hallucinate/
- OWASP Gen AI Security Project. (2025). *LLM09:2025 Misinformation*. https://genai.owasp.org/llmrisk/llm092025-misinformation/
- Spracklen, J., Wijewickrama, R., Sakib, A. H. M. N., Maiti, A., Viswanath, B., & Jadliwala, M. (2025). We have a package for you! A comprehensive analysis of package hallucinations by code generating LLMs. In *Proceedings of the 34th USENIX Security Symposium (USENIX Security 25)*. USENIX Association. https://www.usenix.org/system/files/usenixsecurity25-spracklen.pdf `[also cited in LLM04]`
- Zou, W., Geng, R., Wang, B., & Jia, J. (2025). PoisonedRAG: Knowledge corruption attacks to retrieval-augmented generation of large language models. In *Proceedings of the 34th USENIX Security Symposium (USENIX Security 25)*. USENIX Association. https://www.usenix.org/system/files/usenixsecurity25-zou-poisonedrag.pdf `[also cited in LLM01, LLM09]`


## LLM08: Hidden Context Exposure


- Cao, B., Li, C., Cao, Y., Ge, Y., Wang, T., & Chen, J. (2025). *You can't steal nothing: Mitigating prompt leakages in LLMs via system vectors*. arXiv. https://arxiv.org/abs/2509.21884
- Das, B. C., Amini, M. H., & Wu, Y. (2025). *System prompt extraction attacks and defenses in large language models*. arXiv. https://arxiv.org/abs/2505.23817
- Goffin, J. (2025, July). *Proof of concept: Dangers of system prompt leakage*. Resk. https://resk.fr/pdf/resk-enterprise-ai-security-guide.pdf
- Hui, B., Yuan, H., Gong, N., Burlina, P., & Cao, Y. (2024). *PLeak: Prompt leaking attacks against large language model applications*. arXiv. https://arxiv.org/abs/2405.06823
- jujumilk3. (2025). *Leaked system prompts* [GitHub repository]. GitHub. https://github.com/jujumilk3/leaked-system-prompts
- LeakHub. (n.d.). *LeakHub* [Website]. Retrieved July 11, 2026, from https://leakhub.ai/
- Li, Z., Guo, J., & Cai, H. (2025). *System prompt poisoning: Persistent attacks on large language models beyond user injection*. arXiv. https://arxiv.org/abs/2505.06493
- Nie, Y., Wang, Z., Yu, Y., Wu, X., Zhao, X., Guo, W., & Song, D. (2024). *LeakAgent: RL-based red-teaming agent for LLM privacy leakage*. arXiv. https://arxiv.org/abs/2412.05734
- Zheng, X., Wu, Y., Huang, H., Li, Y., Ma, X., Li, B., Jiang, Y.-G., & Wang, C. (2026). *Just ask: Curious code agents reveal system prompts in frontier LLMs*. arXiv. https://doi.org/10.48550/arXiv.2601.21233

## LLM09: Vector and Embedding Weaknesses


- Chaudhari, H., Severi, G., Abascal, J., Suri, A., Jagielski, M., Choquette-Choo, C. A., Nasr, M., Nita-Rotaru, C., & Oprea, A. (2024). *Phantom: General backdoor attacks on retrieval augmented language generation* (arXiv:2405.20485). arXiv. https://arxiv.org/abs/2405.20485
- Chen, Y., Xu, Q., & Bjerva, J. (2025). ALGEN: Few-shot inversion attacks on textual embeddings via cross-model alignment and generation. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics. https://aclanthology.org/2025.acl-long.1185/
- Chen, Z., Xiang, Z., Xiao, C., Song, D., & Li, B. (2024). *AgentPoison: Red-teaming LLM agents via poisoning memory or knowledge bases* (arXiv:2407.12784). arXiv. https://arxiv.org/abs/2407.12784
- Ha, H., Zhan, Q., Kim, J., Bralios, D., Sanniboina, S., Peng, N., Chang, K.-W., Kang, D., & Ji, H. (2025). *MM-PoisonRAG: Disrupting multimodal RAG with local and global poisoning attacks* (arXiv:2502.17832). arXiv. https://arxiv.org/abs/2502.17832
- Kim, D., Kang, D., Lee, K., Baek, H., & Kang, B. B. (2026). *Zero2Text: Zero-training cross-domain inversion attacks on textual embeddings* (arXiv:2602.01757). arXiv. https://arxiv.org/abs/2602.01757
- Li, H., Xu, M., & Song, Y. (2023). *Sentence embedding leaks more information than you expect: Generative embedding inversion attack to recover the whole sentence* (arXiv:2305.03010). arXiv. https://arxiv.org/abs/2305.03010
- Liu, Y., Yuan, Z., Tie, G., Shi, J., Zhou, P., Sun, L., & Gong, N. Z. (2025). *Poisoned-MRAG: Knowledge poisoning attacks to multimodal retrieval augmented generation* (arXiv:2503.06254). arXiv. https://arxiv.org/abs/2503.06254
- MITRE. (n.d.). *RAG poisoning* (AML.T0070) [Technique]. MITRE ATLAS. https://atlas.mitre.org/techniques/AML.T0070
- Morris, J. X., Kuleshov, V., Shmatikov, V., & Rush, A. M. (2023). *Text embeddings reveal (almost) as much as text* (arXiv:2310.06816). arXiv. https://arxiv.org/abs/2310.06816
- Shafran, A., Schuster, R., & Shmatikov, V. (2025). Machine against the RAG: Jamming retrieval-augmented generation with blocker documents. In *34th USENIX Security Symposium (USENIX Security 25)*. USENIX Association. https://www.usenix.org/conference/usenixsecurity25/presentation/shafran
- Shereen, E., Ristea, D., McFadden, S., Hasircioglu, B., Mavroudis, V., & Hicks, C. (2025). *One pic is all it takes: Poisoning visual document retrieval augmented generation with a single image* (arXiv:2504.02132). arXiv. https://arxiv.org/abs/2504.02132
- Song, C., & Raghunathan, A. (2020). *Information leakage in embedding models* (arXiv:2004.00053). arXiv. https://arxiv.org/abs/2004.00053
- Tan, X., Luan, H., Luo, M., Sun, X., Chen, P., & Dai, J. (2025). RevPRAG: Revealing poisoning attacks in retrieval-augmented generation through LLM activation analysis. In *Findings of the Association for Computational Linguistics: EMNLP 2025*. Association for Computational Linguistics. https://aclanthology.org/2025.findings-emnlp.698/
- Wang, F., Wan, X., Sun, R., Chen, J., & Arık, S. Ö. (2024). *Astute RAG: Overcoming imperfect retrieval augmentation and knowledge conflicts for large language models* (arXiv:2410.07176). arXiv. https://arxiv.org/abs/2410.07176

- Wu, G., Wang, T., Zhang, Y., Zhang, Z., Niu, J., Wu, Y., & Zhang, Y. (2026). When cache poisoning meets LLM systems: Semantic cache poisoning and its countermeasures. In *Proceedings of the 2026 Network and Distributed System Security Symposium (NDSS)*. Internet Society. https://www.ndss-symposium.org/ndss-paper/when-cache-poisoning-meets-llm-systems-semantic-cache-poisoning-and-its-countermeasures/
- Xue, J., Zheng, M., Hu, Y., Liu, F., Chen, X., & Lou, Q. (2024). *BadRAG: Identifying vulnerabilities in retrieval augmented generation of large language models* (arXiv:2406.00083). arXiv. https://arxiv.org/abs/2406.00083
- Zhang, C., Morris, J. X., & Shmatikov, V. (2025). *Universal zero-shot embedding inversion* (arXiv:2504.00147). arXiv. https://arxiv.org/abs/2504.00147
- Zhang, Z., Liu, Z., Xie, Y., Huang, Q., & She, D. (2026). *From similarity to vulnerability: Key collision attack on LLM semantic caching* (arXiv:2601.23088). arXiv. https://arxiv.org/abs/2601.23088
- Zou, W., Geng, R., Wang, B., & Jia, J. (2025). PoisonedRAG: Knowledge corruption attacks to retrieval-augmented generation of large language models. In *Proceedings of the 34th USENIX Security Symposium (USENIX Security 25)*. USENIX Association. https://www.usenix.org/system/files/usenixsecurity25-zou-poisonedrag.pdf `[also cited in LLM01, LLM07]`

## LLM10: Improper Output Handling

- diskordia. (2025, July 24). *Inside CVE-2025-32711 (EchoLeak): Prompt injection meets AI exfiltration*. Hack The Box. https://www.hackthebox.com/blog/cve-2025-32711-echoleak-copilot-vulnerability
- Klondike, G. (2023, June 6). *Threat modeling LLM applications*. AI Village. https://aivillage.org/blog/threat-modeling-llm/
- The MITRE Corporation. (2026). *CWE-116: Improper encoding or escaping of output* (Common Weakness Enumeration List Version 4.20). https://cwe.mitre.org/data/definitions/116.html
- OWASP. (2026). *C7: Model behavior, output control and safety assurance* (AI Security Verification Standard Version 1.0). https://github.com/OWASP/AISVS/blob/main/1.0/en/0x10-C07-Model-Behavior.md
- OWASP. (n.d.). *V5: Validation, sanitization and encoding* (Application Security Verification Standard 4.0). Retrieved July 11, 2026, from https://owasp-aasvs4.readthedocs.io/en/latest/V5.html#validation-sanitization-and-encoding
- Porat, Y. (2025, December 25). *All I want for Christmas is your secrets: LangGrinch hits LangChain Core (CVE-2025-68664)*. Cyata. https://cyata.ai/blog/langgrinch-langchain-core-cve-2025-68664/
- PortSwigger. (n.d.). *Lab: Exploiting insecure output handling in LLMs*. Web Security Academy. Retrieved July 11, 2026, from https://portswigger.net/web-security/llm-attacks/lab-exploiting-insecure-output-handling-in-llms
- Rehberger, J. [wunderwuzzi]. (2024a, June 14). *GitHub Copilot Chat: From prompt injection to data exfiltration*. Embrace The Red. https://embracethered.com/blog/posts/2024/github-copilot-chat-prompt-injection-data-exfiltration/

- Rehberger, J. [wunderwuzzi]. (2024b, December 6). *Terminal DiLLMa: LLM-powered apps can hijack your terminal via prompt injection*. Embrace The Red. https://embracethered.com/blog/posts/2024/terminal-dillmas-prompt-injection-ansi-sequences/
- Rehberger, J. [wunderwuzzi]. (2025, August 12). *GitHub Copilot: Remote code execution via prompt injection (CVE-2025-53773)*. Embrace The Red. https://embracethered.com/blog/posts/2025/github-copilot-remote-code-execution-via-prompt-injection/ `[also cited in LLM01]`
- Willison, S. (n.d.). *Markdown exfiltration* [Tag archive]. Simon Willison's Weblog. Retrieved July 11, 2026, from https://simonwillison.net/tags/markdown-exfiltration/

# Acknowledgements

The 2026 OWASP Top 10 for LLM Applications is the work of dozens of contributors from across the security and AI engineering communities. There are too many to name individually, and the document is better for every one of them. To everyone who drafted, reviewed, debated, voted, tested an entry against a real incident, or refined a mitigation until it held, thank you. This release is yours as much as ours.

We owe particular thanks to the project leads and the entry leads, who carried this list from proposal to publication.

**Project Leads**

- Steve Wilson, Project Lead OWASP Top 10 for LLM Applications
- Rock Lambros, Co-Lead OWASP Top 10 for LLM Applications, Director of AI Standards and Governance, Zenity | Founder, RockCyber

**Entry Leads**

| Entry | Lead(s) |
|---|---|
| LLM01:2026 Prompt Injection | Rachel James |
| LLM02:2026 Sensitive Information Disclosure | Emmanuel Guilherme ; Ken Huang |
| LLM03:2026 Excessive Agency | Andy Smith |
| LLM04:2026 Supply Chain | John Sotiropoulos ; Stefano Amorelli, |
| LLM05:2026 Data and Model Poisoning | Sumeet Jeswan, ; Mark Roxberry, ; Anitha Dakamarri, |
| LLM06:2026 Unbounded Consumption | Sahil Mehta ; Venkata Sai Kishore Modalavalasa, |
| LLM07:2026 Misinformation | Steve Wilson, |
| LLM08:2026 Hidden Context Exposure | Alex Leung ; Vinnie Giarrusso ; Sonu Kumar |
| LLM09:2026 Vector and Embedding Weaknesses | Savio Dsouza; Arshi Chadha, |
| LLM10:2026 Improper Output Handling | Rico Komenda; Gavin Klondike |

# OWASP GenAI Security Project Sponsors

We appreciate our Project Sponsors, funding contributions to help support the objectives of the project and help to cover operational and outreach costs augmenting the resources provided by the OWASP.org foundation. The OWASP GenAI Security Project continues to maintain a vendor neutral and unbiased approach. Sponsors do not receive special governance considerations as part of their support.

Sponsors do receive recognition for their contributions in our materials and web properties. All materials the project generates are community developed, driven and released under open source and creative commons licenses. For more information on becoming a sponsor, visit the Sponsorship Section on our Website to learn more about helping to sustain the project through sponsorship.

## Project Sponsors:

GenAI Security Project — Gold Project Sponsor — Top 10 for LLM and Generative AI

WITNESS AI · HIDDENLAYER · FUJITSU · paloalto NETWORKS · Straiker · Mend.io · snyk · f5 · ByteDance · TREND MICRO

GenAI Security Project — Silver Project Sponsor — Top 10 for LLM and Generative AI

ALICE · GT cyber · zscaler · evoke SECURITY · CROWDSTRIKE · PROTECTO · SentinelOne · PROMPTARMOR · proofpoint. · Cobalt · securiti · apiiro · Capsule · Synack · akto · Starseer · NeuralTrust · CHECK POINT · TROJ.AI · LASSO

**Sponsors list, as of publication date. Find the full sponsor list here.**

# Project Supporters

Project supporters lend their resources and expertise to support the goals of the project.

Accenture
AddValueMachine Inc
Aeye Security Lab Inc.
AI informatics GmbH
AI Village
aigos
Aon
Aqua Security
Astra Security
AVID
AWARE7 GmbH
AWS
BBVA
Bearer
BeDisruptive
Bit79
Blue Yonder
BroadBand Security, Inc.
BuddoBot
Bugcrowd
Cadea
Check Point
Cisco
Cloud Security Podcast
Cloudflare
Cloudsec.ai
Coalfire
Cobalt
Cohere
Comcast
Complex Technologies
Credal.ai
Databook
DistributedApps.ai
DreadNode
DSI
EPAM
Exabeam
EY Italy
F5
FedEx
Forescout
GE HealthCare
Giskard
GitHub
Google
GuidePoint Security
HackerOne
HADESS
IBM
iFood
IriusRisk
IronCore Labs
IT University Copenhagen
Kainos
KLAVAN
Klavan Security Group
KPMG Germany FS
Kudelski Security
Lakera
Lasso Security
Layerup
Legato
Linkfire
LLM Guard
LOGIC PLUS
MaibornWolff
Mend.io
Microsoft
Modus Create
Nexus
Nightfall AI
Nordic Venture Family
Normalyze
NuBinary
Palo Alto Networks
Palosade
Praetorian
Preamble
Precize
Prompt Security
PromptArmor
Pynt
Quiq
Red Hat
RHITE
SAFE Security
Salesforce
SAP
Securiti
See-Docs & Thenavigo
ServiceTitan
SHI
Smiling Prophet
Snyk
Sourcetoad
Sprinklr
stackArmor
Tietoevry
Trellix
Trustwave SpiderLabs
U Washington
University of Illinois
VE3
WhyLabs
Yahoo
Zenity

**Supporters list, as of publication date. Find the full supporter list here.**