

Architectures of Trust: A 2H 2025 Report on Security and Governance for Autonomous LLM Agent Systems

The Evolved Threat Landscape for Agentic Systems

The proliferation of Large Language Models (LLMs) has entered a new phase, marked by a transition from passive, conversational interfaces to autonomous, action-oriented Multi-Agent Systems (MAS). This evolution fundamentally alters the threat landscape. Where security for early-generation LLMs centered on data-centric risks like inaccurate outputs or training data leakage, the security paradigm for 2025 must address *action-oriented* threats. Agentic systems, with their capacity to plan, orchestrate, and execute tasks with minimal human supervision, transform vulnerabilities into direct vectors for system compromise, unauthorized API calls, and financial fraud.¹ The "blast radius" of a security incident is no longer confined to a single model's output but extends to every system an agent can touch.⁴

This section deconstructs the modern threat landscape for agentic systems, using the OWASP Top 10 for LLM Applications (v2025) as a structural guide. Each risk is re-evaluated through the lens of autonomous agents to provide a current threat model for enterprise architects and security practitioners.

The Paradigm Shift: From Data Risks to Action Risks

The core security challenge of 2025 is that agentic systems convert potential data risks into kinetic action risks. A successful prompt injection is no longer just a matter of generating inappropriate content; it is a mechanism for hijacking an agent's control flow to execute unauthorized commands.¹ Similarly, sensitive information disclosure is not merely about leaking training data but involves agents with excessive permissions exfiltrating live production data from integrated systems.⁶ Consequently, security strategies must evolve from

sanitizing prompts to governing agent actions and their real-world consequences.

This shift introduces a "semantic attack surface." Traditional security models focus on syntactic attacks, such as malformed SQL strings or code snippets. Agentic systems, however, are vulnerable to attacks based on the *meaning* of natural language. An instruction like "Find the latest quarterly financial report and summarize the key findings" is a benign and useful command when issued by a CFO. The exact same command becomes a severe data exfiltration vector if an attacker, having compromised a low-level entry point, can issue it to an agent with access to a corporate file share. This demonstrates that conventional Web Application Firewalls (WAFs) and pattern-matching filters, which are blind to intent and context, are fundamentally insufficient for securing this new attack surface.⁸

Deconstructing the OWASP LLM Top 10 (v2025) for Agentic Systems

The OWASP Top 10 for LLM Applications provides a crucial framework for understanding these evolved threats.⁶

LLM01: Prompt Injection as a Control-Flow Hijack Vector

In agentic systems, prompt injection is not a content generation issue but a critical vulnerability that enables control-flow hijacking.⁶ This threat manifests in several sophisticated forms:

- **Indirect Injection:** An agent ingests malicious instructions from an external, untrusted data source it is tasked with processing, such as a webpage, a PDF report, or an email. The agent, lacking a clear boundary between trusted instructions and untrusted data, executes the malicious commands. The **EchoLeak** vulnerability (CVE-2025-32711) serves as a potent real-world example of this, where a crafted email caused Microsoft 365 Copilot to exfiltrate data without any user interaction.⁸
- **Cross-Agent Injection:** In a multi-agent system, a compromised agent (e.g., a web-scraping agent that ingests a malicious site) can poison the data it passes to a subsequent, more privileged agent (e.g., a code-executing agent), effectively using the first agent as a proxy to attack the second.⁵

LLM06: Excessive Agency as a Gateway to System Compromise

Excessive Agency is arguably the most critical risk category for agentic systems. It arises when an agent is granted functionality, permissions, or autonomy beyond the minimum required for its designated task.¹ This creates direct pathways to system compromise.

Examples include:

- An agent equipped with a file system tool that has delete and write permissions when its role only requires read access.¹⁴
- An agent whose API key for a database has write privileges to all tables, when it only needs to query a specific, non-sensitive view.¹
- An autonomous financial agent that can execute trades above a certain threshold without requiring mandatory human-in-the-loop approval.⁶

LLM07 & MASLeak: System Prompt and Compositional IP Leakage

System Prompt Leakage occurs when an attacker extracts an agent's core instructions, revealing its capabilities, limitations, and any embedded (but ill-advised) sensitive data like API keys.⁶ This allows the attacker to craft more effective bypasses for its guardrails.

A more advanced and insidious threat in multi-agent systems is **MASLeak**. This novel attack framework allows an adversary, with only black-box API access, to reverse-engineer an entire MAS architecture. By sending carefully crafted queries that propagate through the agent graph, an attacker can map the system topology, determine the number of agents, and extract the individual system prompts and toolsets of each agent.¹⁶ This is a compositional risk; the interaction between agents reveals far more about the system's intellectual property than any single agent would in isolation, providing a complete blueprint for future attacks.¹⁸

LLM03: Supply Chain Vulnerabilities in the Agent Ecosystem

The attack surface of an agentic system extends to its entire dependency and integration ecosystem. Key vectors include:

- **Unvetted Plugins and Connectors:** Third-party tools can contain vulnerabilities or malicious backdoors, and unchecked access can expose critical infrastructure.⁵
- **Malicious Model Context Protocol (MCP) Servers:** The MCP standard, designed to streamline tool integration, introduces a new trust boundary. An agent connecting to a malicious MCP server can be compromised, as the server can inject hidden instructions

into the tool descriptions it provides to the agent, bypassing standard prompt filters.⁵

- **Poisoned Datasets:** Datasets used for fine-tuning agents can be poisoned to introduce specific biases or backdoors.⁶

LLM08: Vector and Embedding Weaknesses

For agents that use Retrieval-Augmented Generation (RAG), the vector database is a critical component of the supply chain. An attacker can poison the RAG database by injecting malicious or misleading information. When the agent retrieves this corrupted context to inform its reasoning or actions, its behavior can be manipulated, representing a form of indirect data poisoning that subverts the agent's decision-making process.⁶

The total risk of a multi-agent system is often greater than the sum of its parts. A vulnerability in a single, seemingly low-privilege agent, such as a web scraper, can serve as the initial entry point. This agent might ingest malicious content and pass it, now considered "trusted" internal data, to a downstream agent with higher privileges, like one that can execute code or access a database. This pattern, known as "Tool Chaining Abuse," allows for privilege escalation across the agent graph.⁵ This necessitates a shift in threat modeling: the security of any single agent cannot be assessed in isolation. It must be evaluated in the context of its position within the agentic workflow and the capabilities of all agents it can communicate with, demanding a graph-based approach to security analysis.

OWASP LLM Risk (2025)	Description	Amplified Threat in Agentic/Multi-Agent Systems
LLM01: Prompt Injection	Malicious inputs manipulate the LLM's behavior.	Becomes a control-flow hijacking vector. Indirect and cross-agent injection can lead to unauthorized tool use and Remote Code Execution (RCE).
LLM02: Sensitive Info Disclosure	LLM inadvertently reveals confidential data.	Agents with excessive permissions can actively exfiltrate live data from connected systems (databases, APIs, file

		shares), not just training data.
LLM03: Supply Chain	Compromised third-party components, data, or models.	Attack surface expands to include unvetted plugins, malicious MCP servers, and third-party agent frameworks with their own vulnerabilities.
LLM04: Data & Model Poisoning	Manipulation of training or fine-tuning data to create backdoors.	RAG databases become a key vector. Poisoned context can manipulate an agent's reasoning and lead to incorrect or malicious actions.
LLM05: Improper Output Handling	Failure to sanitize LLM outputs before passing to downstream systems.	Agents generating code, API calls, or shell commands without strict validation can directly cause RCE, SQL injection, or SSRF in backend systems.
LLM06: Excessive Agency	Overly broad functionality, permissions, or autonomy.	The core agentic risk. Over-privileged tools and lack of human oversight can turn a simple flaw into a full system compromise.
LLM07: System Prompt Leakage	Exposure of confidential instructions and configurations.	Enables attackers to bypass guardrails. In MAS, can escalate to MASLeak , revealing the entire system architecture and IP.
LLM08: Vector & Embedding Weaknesses	Attacks targeting the vector database in RAG systems.	Poisoned embeddings can manipulate agent behavior by corrupting retrieved

		context, leading to flawed decision-making or malicious actions.
LLM09: Misinformation	Generation of false or misleading information.	An agent acting on hallucinated "facts" can take erroneous and harmful real-world actions (e.g., ordering wrong parts, filing incorrect reports).
LLM10: Unbounded Consumption	LLM resource usage is not properly limited.	Autonomous agents stuck in loops can trigger cascading API calls or intensive computations, leading to massive financial costs (Denial of Wallet) or system-wide DoS.

Table synthesized from.⁶

Foundational Security Architectures for Agentic AI

To counter the evolved threats posed by agentic systems, security must be an architectural principle, not a feature. Reactive defenses are insufficient; a proactive posture requires building systems on foundations of predictability, control, and explicit trust boundaries. This section outlines two such foundational patterns: Control-Flow Integrity via Plan-then-Execute (P-t-E) and Zero-Trust Principles for Non-Human Identities.

Control-Flow Integrity via Plan-then-Execute (P-t-E) Patterns

A cornerstone of secure agent design is the **Plan-then-Execute (P-t-E)** architectural pattern. This model enforces a deliberate separation between strategic planning and tactical execution, offering significant security advantages over the more common and vulnerable

ReAct (Reason-Act) pattern.²⁰

In the P-t-E model, the system is decoupled into two components:

1. **The Planner:** A powerful, high-reasoning LLM that receives the user's objective and generates a complete, multi-step, machine-readable plan to achieve it. This planning phase occurs *before* any interaction with external tools or data sources.
2. **The Executor:** A simpler, more deterministic component (which can be a smaller LLM or even non-LLM code) that executes the pre-defined plan step-by-step.

The primary security benefit of this decoupling is **control-flow integrity**. The ReAct pattern operates in a tight loop: it reasons about the next step, takes an action (e.g., calls a tool), observes the result, and feeds that observation directly back into the next reasoning step. This makes ReAct agents highly susceptible to hijacking. A malicious tool output or a piece of poisoned data retrieved from a website can alter the agent's "thought" process and divert its next action.²⁰

In contrast, the P-t-E pattern is inherently more resilient. Because the entire plan is generated upfront, the executor's trajectory is fixed and predictable. It is not designed to dynamically alter its course based on tool outputs, thus mitigating the risk of its objective being hijacked mid-execution.²⁰ This predictability is not merely an operational benefit; it is a powerful security feature. The generated plan serves as a static, auditable artifact of the agent's intent. This transforms the security challenge from reactively monitoring the unpredictable behavior of a black-box agent to proactively validating a transparent, intended course of action. This enables a new, more robust security workflow:

Plan-Validate-Execute. In this workflow, a human or an automated security layer can inspect and approve the entire plan *before* the executor is invoked, a control that is architecturally impossible in a pure ReAct system.¹

Implementing Zero-Trust Principles for Non-Human Identities

The second foundational principle is to treat every agent as a distinct, non-human identity within the enterprise security fabric, subject to the principles of Zero Trust.¹ An agent is not merely a feature of an application; it is an actor that initiates actions across distributed systems.

This approach requires a fundamental shift in how access control is managed:

- **Identity and Least Privilege:** Each agent must be assigned a unique, verifiable service identity (e.g., via SPIFFE or a cloud IAM service account). This identity must be granted

the absolute minimum set of permissions required for its function, using short-lived, narrowly scoped access tokens. Tool access and API permissions must not be inherited from the host application's environment but must be explicitly granted to the agent's identity.¹

- **Continuous Verification:** Every significant action an agent takes—particularly tool calls and API interactions—must be authenticated and authorized at the point of execution. This requires comprehensive logging of agent actions, tool inputs, and outputs to create a forensic trail for monitoring and anomaly detection.¹
- **Assume Breach:** All data flowing into an agent must be treated as untrusted and potentially hostile. This includes not only direct user prompts but also content retrieved from RAG databases and outputs from other tools or agents. Every input is a potential vector for an injection attack and must be sanitized and validated accordingly.¹

In traditional security, the network was the perimeter. In the cloud era, identity became the new perimeter. For agentic AI, the perimeter shrinks further: it is the **agent's own identity and its scoped permissions**. Securing the agent is not about hardening the server it runs on; it is about rigorously controlling the credentials it holds and the permissions it is granted across every integrated service. This means that enterprise Identity and Access Management (IAM) policies, API gateway authorizers, and database roles must become "agent-aware," capable of enforcing granular policies on these new non-human actors.

Technical Defense-in-Depth: Implementation Blueprints

Building on a secure architecture requires implementing multiple layers of technical controls. This section provides practical blueprints for three critical defense-in-depth layers: input/output guardrails, high-isolation sandboxing for code execution, and automated governance through Policy-as-Code (PaC). All examples are oriented towards a Python and JSON-based development stack.

Input and Output Guardrails with Schema Enforcement

A foundational security pattern for any agentic system is the "Input-Guard -> Agent -> Output-Guard" workflow.¹² This ensures that data is validated and sanitized both before the agent processes it and before the agent's output is acted upon. The most effective and

deterministic way to implement these guards is through strict schema enforcement.

Tool Input Validation with JSON Schema

Before an agent is allowed to execute any tool, its proposed inputs must be validated against a predefined JSON Schema. This prevents a wide range of attacks, including parameter injection and the use of malicious defaults, by ensuring the input is structurally sound and conforms to expected constraints.⁵

For example, a tool that writes content to a file (`file_write`) is a high-risk operation. A strict JSON Schema can enforce critical safety rules, such as preventing path traversal attacks (`../`) and limiting content length.

Example: JSON Schema for a `file_write` tool

JSON

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "FileWriteToolInput",
  "description": "Input schema for a tool that writes content to a file.",
  "type": "object",
  "properties": {
    "filepath": {
      "description": "The relative path of the file to write. Must not contain '..' or start with '/'.",
      "type": "string",
      "pattern": "^(?!.*\\.\\.)(?!/).*$"
    },
    "content": {
      "description": "The content to write to the file.",
      "type": "string",
      "maxLength": 10240
    }
  },
  "required": ["filepath", "content"],
  "additionalProperties": false
}
```

This schema can be validated in Python using the `jsonschema` library before the tool's function is ever called, providing a deterministic check that is far more reliable than relying on the LLM to behave safely.²²

Structured Output Validation

Similarly, the output from an LLM agent must be validated before it is passed to other systems or returned to a user. This is crucial for preventing downstream vulnerabilities like XSS, SQL injection, or RCE.⁶ Frameworks like

Guardrails AI allow developers to define output structures using Pydantic models. The framework then ensures the LLM's response conforms to this structure, re-prompting the model if necessary, before returning a validated and typed object.²³

Secure Code Execution with High-Isolation Sandboxing

For agents that generate and execute code (e.g., for data analysis or software development tasks), running that code directly on the host system is an unacceptable risk, tantamount to an unauthenticated RCE vulnerability.⁵ All untrusted code execution must occur within a high-isolation sandbox.

A robust and scalable architecture for this involves decoupling the agent from the execution environment. An agent can make a request to a dedicated, sandboxed execution service via a REST API (e.g., built with FastAPI). This service then runs the code inside a secure environment, such as a Jupyter kernel within a container, and returns only the output (stdout, stderr, generated files) to the agent.²⁵

For maximum security, the container itself should be run with a user-space kernel like **gVisor**, which intercepts system calls from the containerized application and handles them in a sandboxed environment. This provides a much stronger isolation boundary than standard Linux containers, protecting the host kernel from potential container escape exploits.²⁵

Example: Docker Configuration for gVisor (runsc) Runtime

To use gVisor with Docker, the `daemon.json` file is configured to add `runsc` as a new runtime. Key flags can restrict network access and select the most performant underlying platform.

JSON

```
// /etc/docker/daemon.json
{
  "runtimes": {
    "runsc": {
      "path": "/usr/local/bin/runsc",
      "runtimeArgs": [
        "--network=none",
        "--platform=kvm",
        "--gso=false"
      ]
    }
  }
}
```

With this configuration, a sandboxed container can be launched with a simple command:

Bash

```
docker run --runtime=runsc --rm python:3.11-slim python -c "print('Hello from a gVisor sandbox')"
```

This approach is becoming an industry standard, with frameworks like AutoGen now defaulting to Docker-based execution environments to enhance security out-of-the-box.²⁸

Sandboxing Technology	Isolation Level	Performance Overhead	Startup Time	Resource Footprint	Primary Use Case for LLM Agents
Docker (LXC)	Medium (Shared Kernel)	Low	Fast (<1s)	Low	Running trusted, internally developed

					tools with limited permissions .
Docker + gVisor	High (User-space Kernel)	Medium	Fast (~1s)	Medium	Executing untrusted, LLM-generated code or handling potentially malicious user inputs. The state-of-the-art for security/performance balance.
Firecracker/VMs	Very High (Hardware Virtualization)	High	Slow (seconds)	High	High-stakes , multi-tenant environments where maximum isolation is required and performance overhead is acceptable.

Table synthesized from.²⁵

Automated Governance with Policy-as-Code (PaC)

Policy-as-Code (PaC) provides a scalable and auditable framework for enforcing governance rules on agent actions. Using tools like **Open Policy Agent (OPA)** and its declarative language, **Rego**, security and compliance rules are expressed as code and managed in a central repository. This decouples policy logic from the agent's application code, allowing security teams to update rules without requiring developer intervention.²⁹

In a typical implementation, an agent's request to call a tool is intercepted by a **Policy Enforcement Point (PEP)**, such as an API gateway or a service mesh proxy. The PEP queries a **Policy Decision Point (PDP)**—an OPA instance—with the context of the request (e.g., agent identity, tool being called, parameters). The PDP evaluates the request against its loaded policies and returns an "allow" or "deny" decision, which the PEP then enforces.²⁹

Example: Rego Policy for Agent API Authorization

This Rego policy enforces role-based access control. It denies any agent without the "admin" role from using tools whose names begin with "delete_". This prevents a lower-privileged "researcher" agent from performing destructive actions, even if it is compromised.

Fragment kodu

```
package agent.authz

default allow = false

# Admins are allowed to perform any action
allow {
  input.agent.role == "admin"
}

# Non-admins are allowed to use non-destructive tools
allow {
  input.agent.role != "admin"
  not startswith(input.tool.name, "delete_")
}
```

This policy provides a fine-grained, context-aware control that can be updated and deployed instantly across an entire fleet of agents.³⁰

Securing Modern Agent Orchestration Frameworks

The architectural principles and technical controls previously discussed must be applied within the context of specific agent orchestration frameworks. The most popular frameworks—LangGraph, AutoGen, and CrewAI—provide immense flexibility for building powerful agents, but they do not offer comprehensive, out-of-the-box security suites. Security is a developer responsibility that must be built into the application logic.¹⁴ The architectural pattern of the chosen framework significantly influences the security posture and the ease of implementing specific controls.

Hardening LangGraph Applications

LangGraph's core strength is its use of an explicit, stateful graph to define agent workflows. This structure provides natural and intuitive points for inserting security controls.³⁵

- **Securing State:** The graph's shared state object is a powerful feature for maintaining context but also a potential vector for data leakage or state corruption. It is critical to implement validation logic on state transitions to ensure that sensitive information is not improperly persisted or passed between nodes.³⁷
- **Human-in-the-Loop (HITL) Checkpoints:** LangGraph's architecture is exceptionally well-suited for implementing HITL. A node can be explicitly defined to interrupt the graph's execution and wait for human approval before proceeding. This is a crucial control for high-stakes actions like deploying code, authorizing a financial transaction, or sending external communications. Execution can be paused, the agent's proposed action and reasoning can be reviewed, and only upon explicit approval does the graph continue to the next node.³⁶
- **Tool Security:** LangGraph itself does not provide native tool security features.¹⁴ Developers are entirely responsible for securing each tool integrated into the graph. This means applying the principles from Section 3 is non-negotiable: every tool must have a strict input schema, operate with the least possible privilege, and execute in a sandboxed environment if it handles untrusted code.¹⁴

Securing AutoGen Conversations

AutoGen enables multi-agent collaboration through a conversational paradigm. Its dynamic

and often emergent nature makes it powerful for complex problem-solving but presents unique security challenges, as the exact sequence of operations is not known in advance.³⁵

- **Secure Code Execution:** Given that AutoGen agents frequently generate and execute code, using a secure executor is paramount. The framework provides a `DockerCommandLineCodeExecutor`, which should be considered the mandatory default for any production use case. This ensures that all code execution is isolated within a container, significantly reducing the risk of host compromise.²⁴

Example: Configuring a UserProxyAgent with a Docker Executor

Python

```
from autogen.agentchat import UserProxyAgent
from autogen.coding import DockerCommandLineCodeExecutor
import os

# Ensure a directory for the code exists
os.makedirs("coding_dir", exist_ok=True)

# Create a Docker-based executor
docker_executor = DockerCommandLineCodeExecutor(
    image="python:3.12-slim", # A minimal, trusted image
    timeout=60,             # Set a timeout for execution
    work_dir="coding_dir"   # Mount a local directory into the container
)

# Configure the agent to use the secure executor
user_proxy = UserProxyAgent(
    name="user_proxy_secure",
    code_execution_config={"executor": docker_executor},
    human_input_mode="ALWAYS" # Require human confirmation before execution
)
```

Code synthesized from.²⁴

- **Preventing Control-Flow Hijacking:** Research has shown that AutoGen agents are highly vulnerable to control-flow hijacking when processing malicious content from local files or external sources.¹³ Mitigating this requires rigorous input sanitization before data is passed to any agent and ensuring memory isolation between different user sessions to prevent cross-contamination.⁵
- **Observability for Anomaly Detection:** The unpredictable nature of conversational workflows makes comprehensive monitoring essential. AutoGen's native integration with OpenTelemetry should be used to trace agent interactions, tool calls, and message flows. This detailed telemetry is critical for security teams to establish baseline behaviors and detect anomalies that could indicate a compromise.⁴¹

The choice between a framework like LangGraph and one like AutoGen is also a choice of security model. LangGraph's explicit, predictable graph is better suited for high-stakes, auditable workflows where deterministic checkpoints are required. AutoGen's emergent, conversational model is more powerful for open-ended, creative tasks but demands stronger external "boundary" controls, such as robust sandboxing and runtime policy enforcement, because its internal control flow is less predictable.

Security Feature	LangGraph	AutoGen	CrewAI
Native Code Sandboxing	Developer-Implemented	Configurable (Docker Recommended)	Developer-Implemented
State Management Security	Developer-Implemented (via state validation)	Developer-Implemented (via memory isolation)	Developer-Implemented
Human-in-the-Loop Support	Native & Granular (via graph nodes)	Configurable (via human_input_mode)	Configurable (via HITL tasks)
Observability/Tracing	LangSmith Integration	OpenTelemetry Integration	Langfuse/Phoenix Integration
Overall Security Posture	High Control: Explicit graph enables fine-grained, deterministic security checkpoints. Security is highly configurable but requires significant developer effort.	High Risk/High Flexibility: Conversational model is powerful but less predictable. Relies heavily on robust sandboxing and runtime monitoring.	High-Level Abstraction: Focuses on role-based delegation. Security relies on proper tool definition and trusting external components (e.g., MCP servers).

Table synthesized from.²⁸

Anatomy of Recent Failures: Case Studies and Mitigations

Analyzing recent, real-world security failures provides invaluable lessons for defending future systems. The following case studies from 2025 demonstrate how the theoretical risks discussed previously manifest in production environments and highlight the critical importance of specific defensive layers.

Case Study 1: RCE in Langflow (CVE-2025-3248)

In early 2025, a critical unauthenticated Remote Code Execution (RCE) vulnerability was discovered and actively exploited in Langflow, a popular open-source UI for building agentic workflows.⁴³

- **Vulnerability Analysis:** The flaw, assigned CVE-2025-3248 with a CVSS score of 9.8, existed in the `/api/v1/validate/code` API endpoint. This endpoint was designed to validate Python code snippets but did so by directly passing the user-provided code to a Python `exec` function without proper authentication or sanitization.⁴⁴
- **Exploitation:** An unauthenticated attacker could send a crafted HTTP request to this endpoint containing arbitrary Python code (e.g., a reverse shell payload). The Langflow server would execute this code with the privileges of the server process, leading to a full system compromise.⁴³ With over 500 internet-exposed instances identified, the vulnerability was quickly exploited in the wild.⁴³
- **Mitigation:** The official patch in Langflow version 1.3.0 remediated the issue by placing the vulnerable endpoint behind an authentication wall, ensuring that only authenticated users could access it.⁴⁴
- **Primary Lesson:** This incident underscores that the attack surface of an agentic system is the *entire application stack*, not just the LLM or the agent's reasoning loop. Even with a perfectly secure code execution sandbox for the agent itself, a classic web application vulnerability (in this case, an unauthenticated API endpoint leading to code injection) in the surrounding framework provides a direct path to compromise. AI security teams must possess deep expertise in traditional application security (AppSec) principles and cannot afford to focus solely on LLM-specific risks.

Case Study 2: Zero-Click Exfiltration via EchoLeak (CVE-2025-32711)

Disclosed in June 2025, EchoLeak was a sophisticated, multi-stage attack against Microsoft 365 Copilot that achieved zero-click data exfiltration. It stands as the first publicly documented case of indirect prompt injection being weaponized to cause a concrete data leak in a major production AI system.⁸

- **Vulnerability Analysis:** The attack exploited Copilot's ability to process content from multiple sources, including untrusted external emails, within the same context as a user's trusted internal documents and commands. The core vulnerability was a failure to maintain a hard separation between these trust domains.⁴⁶
- **Exploitation Chain:** The attack required no user interaction beyond Copilot's normal background processing of data.
 1. **XPIA Classifier Bypass:** The attacker sent an email containing a malicious prompt. The prompt was carefully phrased in natural language to appear as a benign request to a human recipient, which allowed it to evade Microsoft's Cross-Prompt Injection Attack (XPIA) classifier designed to detect malicious instructions.⁸
 2. **Link Filter Bypass:** The hidden prompt instructed Copilot to find sensitive information within the user's context and exfiltrate it via a URL. To bypass filters that redacted standard Markdown hyperlinks ([text](url)), the attacker used a lesser-known "reference-style" link syntax ([text][ref]), which the filter failed to recognize.⁸
 3. **Zero-Click Trigger:** To make the exfiltration automatic, the malicious link was embedded within a Markdown image tag (![alt text][ref]). The Copilot chat interface, upon receiving the agent's response, would automatically attempt to fetch and render the "image," thereby sending a request to the attacker-controlled URL containing the sensitive data as a parameter.⁸
- **Mitigation:** Microsoft deployed a server-side patch that reportedly included stricter partitioning of prompts to isolate untrusted external content from trusted internal data, alongside enhanced input filtering mechanisms.⁴⁶
- **Primary Lesson:** The zero-click nature of EchoLeak proves that detection and response are insufficient security strategies for agentic threats. Since no user action is required to trigger the exploit, the system must be architecturally designed to *prevent* the attack chain from executing in the first place. This highlights the critical need for preventative controls like **Strict Prompt Partitioning**, which creates an unbreakable logical boundary between trusted user instructions and untrusted external data, ensuring the latter can never be interpreted as a command.⁴⁷

An Enterprise Operating Model for Agent Governance

Deploying and managing autonomous agents at scale requires more than just technical controls; it demands a robust operating model that integrates governance into the entire AI lifecycle. A successful program establishes clear lines of accountability, standardizes security practices, and empowers teams to innovate safely. A common failure mode is the creation of a parallel "AI ethics" program that is disconnected from the organization's core risk and security functions.¹ To be effective, agent governance must be embedded directly into the existing enterprise risk stack, treating agents not as a new category of risk, but as a powerful new source of existing operational, financial, and reputational risks.¹

A Multi-Tiered Governance Framework

A mature governance program operates on three distinct but interconnected tiers, ensuring that strategic goals, tactical standards, and operational controls are aligned.¹

- **Tier 1: Enterprise Oversight:** This strategic layer is owned by an **AI Risk Committee**, typically chaired by a C-level executive such as the Chief Data & AI Officer (CDAO). This body is responsible for setting the organization-wide risk appetite for AI. Its mandate includes defining high-level policies on acceptable data sources, establishing authority thresholds for autonomous actions (e.g., financial transaction limits), and ensuring all agentic systems comply with external regulations like the EU AI Act and federal mandates such as the U.S. OMB M-25-21.⁴
- **Tier 2: Program Management:** This tactical layer is executed by a central **AI Center of Excellence (CoE)**. The CoE's role is to translate the high-level policies from the oversight committee into practical, reusable standards and templates. Key deliverables include maintaining a comprehensive inventory of all AI agents, their owners, and capabilities; creating standardized "go-live bundles" that package required artifacts like data lineage documentation, red team test results, and rollback plans; and managing a central registry of approved tools and data sources.¹
- **Tier 3: Builder Controls:** This operational layer consists of the technical controls and automated checks embedded directly into the developer workflow. This includes mandating automated security scans for vulnerabilities and secrets in CI/CD pipelines, requiring peer reviews for any new agent tool, enforcing the use of standardized and hardened sandbox environments, and ensuring all agent actions are captured by a centralized logging and monitoring platform.⁴

This structured approach, contrary to being a blocker, often acts as a velocity enabler. An Accenture survey found that organizations with mature AI governance frameworks reduced their time-to-value for AI initiatives by a year or more.⁴ Clear rules, automated guardrails, and

pre-approved patterns reduce ambiguity and rework, allowing development teams to build with confidence and speed.


Phased Rollout Strategy for Autonomous Agents

Introducing highly autonomous systems into an enterprise should be a gradual, risk-managed process. A pragmatic four-phase strategy allows an organization to build capabilities, demonstrate value, and mature its governance practices incrementally.¹

- **Phase 1 — Pilot with Guardrails (4–8 weeks):**
 - **Objective:** Scope one or two high-value use cases with a contained "blast radius."
 - **Controls:** Implement the most stringent security posture. Enforce full logging of all actions, use a default-deny network egress policy, and require mandatory Human-in-the-Loop (HITL) approval for any action that writes data or has external effects. A manual red team review is conducted before release.
- **Phase 2 — Expand with Patterns (1–2 quarters):**
 - **Objective:** Scale adoption by creating reusable, certified patterns for common agentic tasks (e.g., a secure RAG pattern, a pre-approved tool registry).
 - **Controls:** Transition from mandatory HITL to risk-based HITL, where approval is triggered only for actions exceeding a defined risk threshold. Implement automated input/output scanning for prompt injection and PII.
- **Phase 3 — Standardize and Certify (next 2–3 quarters):**
 - **Objective:** Formalize the agent lifecycle with defined gates for promoting agents to production.
 - **Controls:** Integrate agent governance into enterprise GRC (Governance, Risk, and Compliance) tools. Implement canary deployments and automated rollback capabilities based on performance and security monitoring. Conduct periodic internal and external audits against standards like ISO/IEC 42001.
- **Phase 4 — Optimize and Federate (ongoing):**
 - **Objective:** Empower business units to build and deploy agents using self-service templates and certified components, while central oversight is maintained for high-risk categories.
 - **Metrics:** Track business impact (e.g., cycle time reduction), security incident rates, red team finding closure times, and indicators of model or agent drift.

By following this phased approach, an enterprise can harness the transformative potential of autonomous agents while systematically managing the associated risks, building a culture of secure and responsible AI innovation.

Cytowane prace

1. Risks & Governance for AI Agents in the Enterprise (2025) - Skywork.ai, otwierano: września 22, 2025, <https://skywork.ai/blog/ai-agent-risk-governance-best-practices-2025-enterprise/>
2. Agentic AI: Everything You Need to Know - SOCRadar® Cyber Intelligence Inc., otwierano: września 22, 2025, <https://socradar.io/agentic-ai-everything-you-need-to-know/>
3. The agentic threat. - Djimit van data naar doen., otwierano: września 22, 2025, <https://djimit.nl/the-agentic-threat/>
4. Your New, Multi-Tiered Approach to Agent Governance - Dataiku blog, otwierano: września 22, 2025, <https://blog.dataiku.com/new-approach-to-agent-governance>
5. AI Agent Framework Security: LangChain, LangGraph, CrewAI & More - SecureLayer7, otwierano: września 22, 2025, <https://blog.securelayer7.net/ai-agent-frameworks/>
6. OWASP Top 10 LLM, Updated 2025: Examples & Mitigation Strategies, otwierano: września 22, 2025, <https://www.oligo.security/academy/owasp-top-10-llm-updated-2025-examples-and-mitigation-strategies>
7. Securing Agentic AI: Preventing Access and IP Leaks - SPIRL, otwierano: września 22, 2025, <https://www.spirl.com/blog/when-ai-knew-too-much-a-cautionary-tale-about-agentic-systems-without-guardrails>
8. EchoLeak: The First Real-World Zero-Click Prompt Injection Exploit in a Production LLM System - arXiv, otwierano: września 22, 2025, <https://arxiv.org/html/2509.10540v1>
9. EchoLeak: The First Real-World Zero-Click Prompt Injection Exploit in a Production LLM System - arXiv, otwierano: września 22, 2025, <https://arxiv.org/html/2509.10540v1/>
10. OWASP Top 10 Risks for Large Language Models: 2025 updates - Barracuda Blog, otwierano: września 22, 2025, <https://blog.barracuda.com/2024/11/20/owasp-top-10-risks-large-language-models-2025-updates>
11. OWASP Top 10 2025 for LLM Applications: What's new? Risks, and Mitigation Techniques, otwierano: września 22, 2025, <https://www.confident-ai.com/blog/owasp-top-10-2025-for-llm-applications-risks-and-mitigation-techniques>
12. LLM Guardrails: Securing LLMs for Safe AI Deployment - WitnessAI, otwierano: września 22, 2025, <https://witness.ai/blog/llm-guardrails/>
13. Multi-Agent Systems Execute Arbitrary Malicious Code - arXiv, otwierano: września 22, 2025, <https://arxiv.org/html/2503.12188v2>
14. Security Policy |  LangChain, otwierano: września 22, 2025, <https://python.langchain.com/docs/security/>
15. LLM07:2025 System Prompt Leakage - OWASP Gen AI Security Project, otwierano: września 22, 2025, <https://genai.owasp.org/llmrisk/llm072025-system-prompt-leakage/>

16. IP Leakage Attacks Targeting LLM-Based Multi-Agent Systems - arXiv, otwierano: września 22, 2025, <https://arxiv.org/abs/2505.12442>
17. IP Leakage Attacks Targeting LLM-Based Multi-Agent Systems - arXiv, otwierano: września 22, 2025, <https://arxiv.org/html/2505.12442v1>
18. The Sum Leaks More Than Its Parts: Compositional Privacy Risks and Mitigations in Multi-Agent Collaboration - arXiv, otwierano: września 22, 2025, <https://arxiv.org/html/2509.14284v1>
19. MCP Security 101: A New Protocol for Agentic AI - Protect AI, otwierano: września 22, 2025, <https://protectai.com/blog/mcp-security-101>
20. Architecting Resilient LLM Agents: A Guide to Secure Plan ... - arXiv, otwierano: września 22, 2025, <https://arxiv.org/pdf/2509.08646>
21. Mastering LLM Guardrails: Complete 2025 Guide | Generative AI Collaboration Platform, otwierano: września 22, 2025, <https://orq.ai/blog/llm-guardrails>
22. Creating your first schema - JSON Schema, otwierano: września 22, 2025, <https://json-schema.org/learn/getting-started-step-by-step>
23. guardrails-ai/guardrails: Adding guardrails to large ... - GitHub, otwierano: września 22, 2025, <https://github.com/guardrails-ai/guardrails>
24. Secure Code Execution in AI Agents | by Saurabh Shukla - Medium, otwierano: września 22, 2025, <https://saurabh-shukla.medium.com/secure-code-execution-in-ai-agents-d2ad84cbec97>
25. Setting Up a Secure Python Sandbox for LLM Agents - dida Machine Learning, otwierano: września 22, 2025, <https://dida.do/blog/setting-up-a-secure-python-sandbox-for-llm-agents>
26. Code Sandboxes for LLMs and AI Agents | Amir's Blog, otwierano: września 22, 2025, <https://amirmalik.net/2025/03/07/code-sandboxes-for-llm-ai-agents>
27. What is gVisor?, otwierano: września 22, 2025, <https://gvisor.dev/docs/>
28. Code execution is now by default inside docker container | AutoGen 0.2, otwierano: września 22, 2025, <https://microsoft.github.io/autogen/0.2/blog/2024/01/23/Code-execution-in-docker/>
29. Agent Governance at Scale: Policy-as-Code Approaches in Action - NexaStack, otwierano: września 22, 2025, <https://www.nexastack.ai/blog/agent-governance-at-scale>
30. Open Policy Agent, otwierano: września 22, 2025, <https://openpolicyagent.org/>
31. Everyone Loves Policy as Code, No One Wants to Write Rego - Permit.io, otwierano: września 22, 2025, <https://www.permit.io/blog/no-one-wants-to-write-rego>
32. Open Policy Agent (OPA) Rego Language Tutorial - Spacelift, otwierano: września 22, 2025, <https://spacelift.io/blog/open-policy-agent-rego>
33. AutoGen Studio - Microsoft Open Source, otwierano: września 22, 2025, <https://microsoft.github.io/autogen/dev/user-guide/autogenstudio-user-guide/index.html>
34. MCP Security Considerations - CrewAI, otwierano: września 22, 2025, <https://docs.crewai.com/mcp/security>

35. LangGraph vs AutoGen vs CrewAI: Complete AI Agent Framework ..., otwierano: września 22, 2025,
<https://latenode.com/blog/langgraph-vs-autogen-vs-crewai-complete-ai-agent-framework-comparison-architecture-analysis-2025>
36. LangGraph - LangChain, otwierano: września 22, 2025,
<https://www.langchain.com/langgraph>
37. LangGraph Tutorial: A Comprehensive Guide to Building Advanced AI Agents, otwierano: września 22, 2025,
https://dev.to/aragorn_talks/langgraph-tutorial-a-comprehensive-guide-to-building-advanced-ai-agents-l31
38. Use a LangGraph agent | Generative AI on Vertex AI - Google Cloud, otwierano: września 22, 2025,
<https://cloud.google.com/vertex-ai/generative-ai/docs/agent-engine/use/langgraph>
39. A Developer's Guide to Multi-Agent Frameworks: CrewAI, AutoGen, and LangGraph | by Nishant Gupta | AiGenVerse | Aug, 2025 | Medium, otwierano: września 22, 2025,
<https://medium.com/aigenverse/a-developers-guide-to-multi-agent-frameworks-crewai-autogen-and-langgraph-15531c0c7dfe>
40. Code Executors | AutoGen 0.2 - Microsoft Open Source, otwierano: września 22, 2025, <https://microsoft.github.io/autogen/0.2/docs/tutorial/code-executors/>
41. Tracing and Observability — AutoGen - Microsoft Open Source, otwierano: września 22, 2025,
<https://microsoft.github.io/autogen/stable//user-guide/agentchat-user-guide/tracing.html>
42. AutoGen - Microsoft Research, otwierano: września 22, 2025,
<https://www.microsoft.com/en-us/research/project/autogen/>
43. CVE-2025-3248 - RCE flaw in Langflow framework for building AI ..., otwierano: września 22, 2025,
<https://insights.integrity360.com/cve-2025-3248-rce-flaw-in-langflow-framework-for-building-ai-agents-exploited-by-attackers>
44. Unsafe at Any Speed: Abusing Python Exec for Unauth RCE in Langflow AI | Horizon3.ai, otwierano: września 22, 2025,
<https://horizon3.ai/attack-research/disclosures/unsafe-at-any-speed-abusing-python-exec-for-unauth-rce-in-langflow-ai/>
45. Inside CVE-2025-32711 (EchoLeak): Prompt injection meets AI exfiltration - HackTheBox, otwierano: września 22, 2025,
<https://www.hackthebox.com/blog/cve-2025-32711-echoleak-copilot-vulnerability>
46. Preventing Zero-Click AI Threats: Insights from EchoLeak - Trend Micro, otwierano: września 22, 2025,
https://www.trendmicro.com/en_id/research/25/g/preventing-zero-click-ai-threats-insights-from-echoleak.html
47. (PDF) EchoLeak: The First Real-World Zero-Click Prompt Injection Exploit in a Production LLM System - ResearchGate, otwierano: września 22, 2025,

- https://www.researchgate.net/publication/395526362_EchoLeak_The_First_Real-World_Zero-Click_Prompt_Injection_Exploit_in_a_Production_LLM_System
48. Evolving Power Platform Governance for AI Agents - Microsoft, otwierano: września 22, 2025, <https://www.microsoft.com/en-us/power-platform/blog/2025/07/31/evolving-power-platform-governance-for-ai-agents/>
49. April 3, 2025 M-25-21 MEMORANDUM FOR THE HEADS OF EXECUTIVE DEPARTMENTS AND AGENCIES FROM: R~ssell T. Vought \ 1 \ Director \ J - The White House, otwierano: września 22, 2025, <https://www.whitehouse.gov/wp-content/uploads/2025/02/M-25-21-Accelerating-Federal-Use-of-AI-through-Innovation-Governance-and-Public-Trust.pdf>
50. Self-host open-source LLM agent sandbox on your own cloud ..., otwierano: września 22, 2025, <https://blog.skypilot.co/skypilot-llm-sandbox/>
51. Introduction - CrewAI Documentation, otwierano: września 22, 2025, <https://docs.crewai.com/introduction>