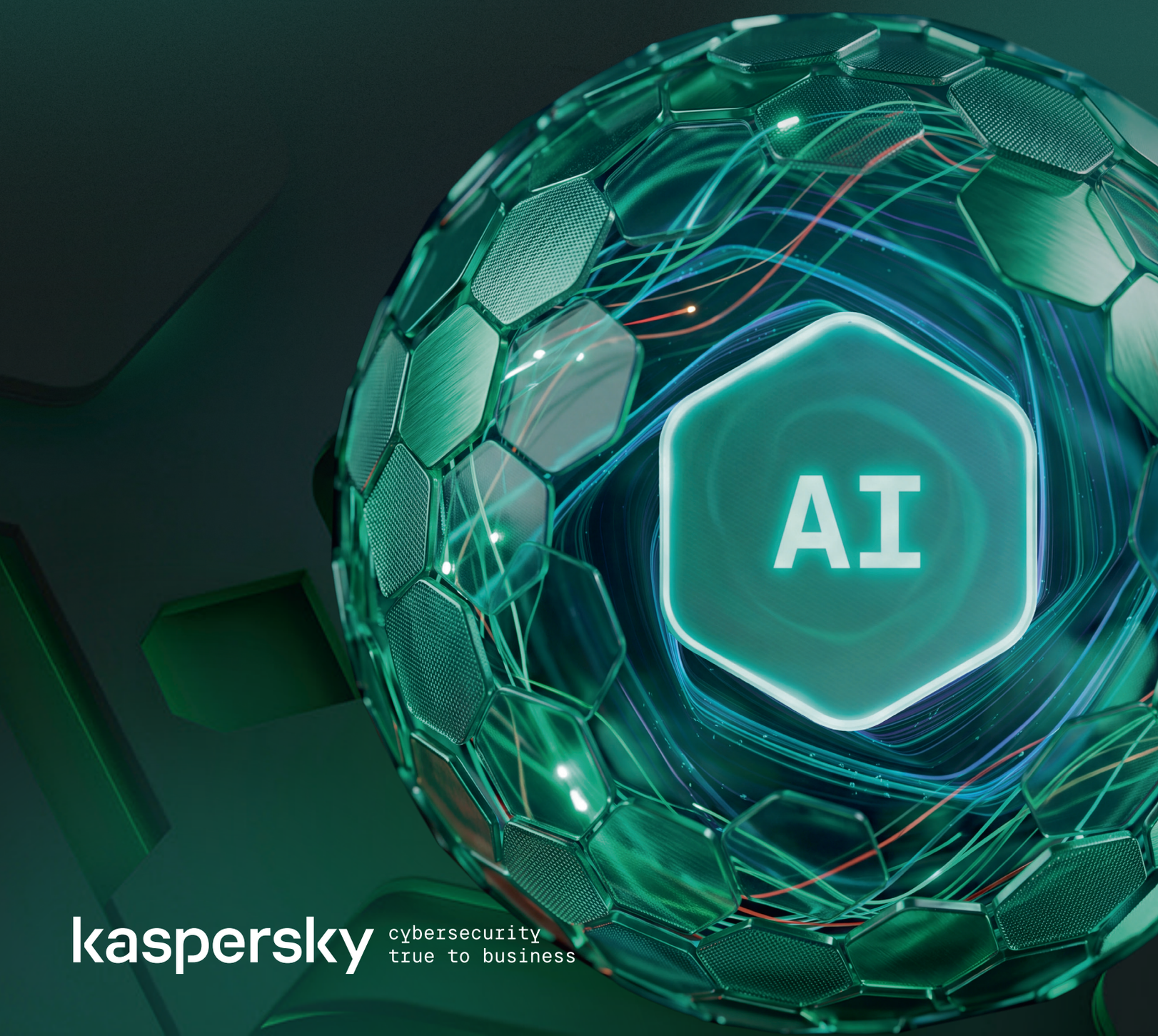
AI agent security through the lens of Cyber Immunity



kaspersky cybersecurity true to business

Contents

Key findings and recommendations ... 3

Key terms and definitions ... 4

Levels of agentic autonomy ... 5

AI agent use cases ... 6

Technology stack and supply chain ... 7

Known incidents involving AI agents ... 8

- EchoLeak, SearchLeak, ForcedLeak, CamoLeak, and other implementations of the “Lethal Trifecta” ... 8
- Malicious exploitation of agents ... 9
- Unintentional overreach ... 9

Threat models for AI agents ... 10

- Common causes of AI agent vulnerabilities ... 10
- AI agent risks: OWASP top 10 ... 11

Core principles of Cyber Immunity ... 12

Applying Cyber Immunity to AI agent design ... 13

- Security assumptions and security goals for AI agents ... 13
- Minimizing the trusted computing base ... 14
- Component isolation ... 15
  - Execution environment isolation ... 15
  - Context isolation ... 17
  - Inter-agent isolation ... 18
- Interaction control ... 18

Probabilistic security measures ... 19

Conclusions and recommendations ... 19

AI agents are increasingly able to interact with data, systems and other applications, extending the security implications of large language models beyond simple chatbots. This paper examines the risks associated with increasingly autonomous agentic systems, draws on known incidents and existing threat models, and considers how the principles of Cyber Immunity can be applied to their design. The focus is on architectural approaches that limit the potential impact of errors, manipulation and compromise.

Key findings and recommendations

AI agents – systems based on large language models (LLMs) that can autonomously interact with files, databases, email and other systems - are rapidly becoming embedded in corporate processes, including software development, customer support, knowledge management, analytics, IT and cybersecurity and business operations. At the same time, adoption often happens from the bottom up, bypassing CIOs and CISOs and giving rise to Shadow AI – AI infrastructure deployed without IT or cybersecurity oversight, leaving its security position unknown. The fundamental challenge lies in the nature of LLMs. They are non-deterministic systems that do not reliably distinguish between untrusted data and instructions, which makes them inherently vulnerable to prompt injection. Risks increase with every successive level of agentic autonomy, from LLMs without tools to autonomous and multi-agent systems. Key factors behind agent vulnerabilities include an untrusted cognitive core, the 'lethal trifecta' – access to confidential data, processing of untrusted input and the ability to communicate externally – as well as excessive privileges and a lack of transparency that can lead to approval fatigue. Real-world incidents, together with cases in which agents have unintentionally deleted data, demonstrate the practical significance of these risks, which have already been systematized across a number of security frameworks.

This report demonstrates that the principles of Cyber Immunity – Kaspersky's approach to building secure-by-design systems - can be applied effectively to AI agents:

1

Define security goals and assumptions at the design stage

LLMs should be treated as untrusted, any external data as potentially malicious, and any irreversible action as requiring human approval.

2

Minimize the trusted computing base

In short: Keep the set of components that must be trusted for security as small as possible. Security guarantees should depend on a small set of deterministic components, such as the agent harness, policy engine and isolation mechanisms. The agent's available tools, accessible context and degree of autonomy should also be limited to what is strictly necessary.

3

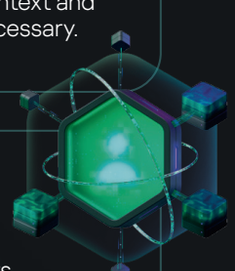
Isolate components

Use sandboxes and virtual machines to isolate the execution environment, separate trusted and untrusted contexts, and isolate individual agents within multi-agent systems.

4

Control interactions using a default-deny approach

Apply policies to tool calls and their arguments, control network traffic, and prevent the agent from dynamically modifying its supply chain or AIBOM.



Probabilistic security measures – such as alignment and external guardrails, including AI Firewall solutions – cannot replace secure architecture. However, they remain an important layer of defense-in-depth and a valuable source of telemetry.

Key takeaways

An LLM cannot be treated as a trusted component.

Agent security is determined by architectural decisions made at the design stage.

Any restriction reduces agent utility – finding the right security-utility balance is therefore a central challenge when designing secure AI agents.

Recommendations for CISOs / CIOs / CTOs:

Inventory AI agents and assess them based on their level of agentic autonomy and whether the lethal trifecta is present.

Manage the agentic supply chain, including AI Bill of Materials (AIBOM) and component verification.

Build security requirements into the design stage together with implementation teams.

Implement appropriate, policy-driven isolation and containerization policies.

Make observability mandatory including LLM traces and host telemetry.

Adopt an integrated, multi-layered security approach.

Key terms and definitions

Large language model (LLM)

A large language model (LLM) is a statistical language model that predicts a probability distribution for the next word[1] based on the preceding text, or prompt. Through a complex, multi-stage training process, LLMs learn to generate not merely a continuation of a text, but an answer to a question or a solution to a task specified by the user in the prompt. From a cybersecurity perspective, LLMs have the following properties:

1

LLMs are non-deterministic systems

The next token – i.e. a word, part of a word or symbol – is selected probabilistically from a predicted probability distribution. As a result, repeated runs with the same input can produce different outputs.

2

LLMs do not inherently distinguish between data and instructions

All inputs – the developer-defined system prompt, the user request, web search results, and so on – are placed into a single context, coexist on equal footing, and are processed in the same way. As a result, untrusted user input can be interpreted as a trusted instruction.

3

LLMs cannot be formally verified

LLMs are highly complex systems whose behavior is shaped by their architecture, learned parameters, training data and probabilistic generation process. As a result, it's impossible to formally guarantee that an LLM will never produce malicious or otherwise undesirable output.



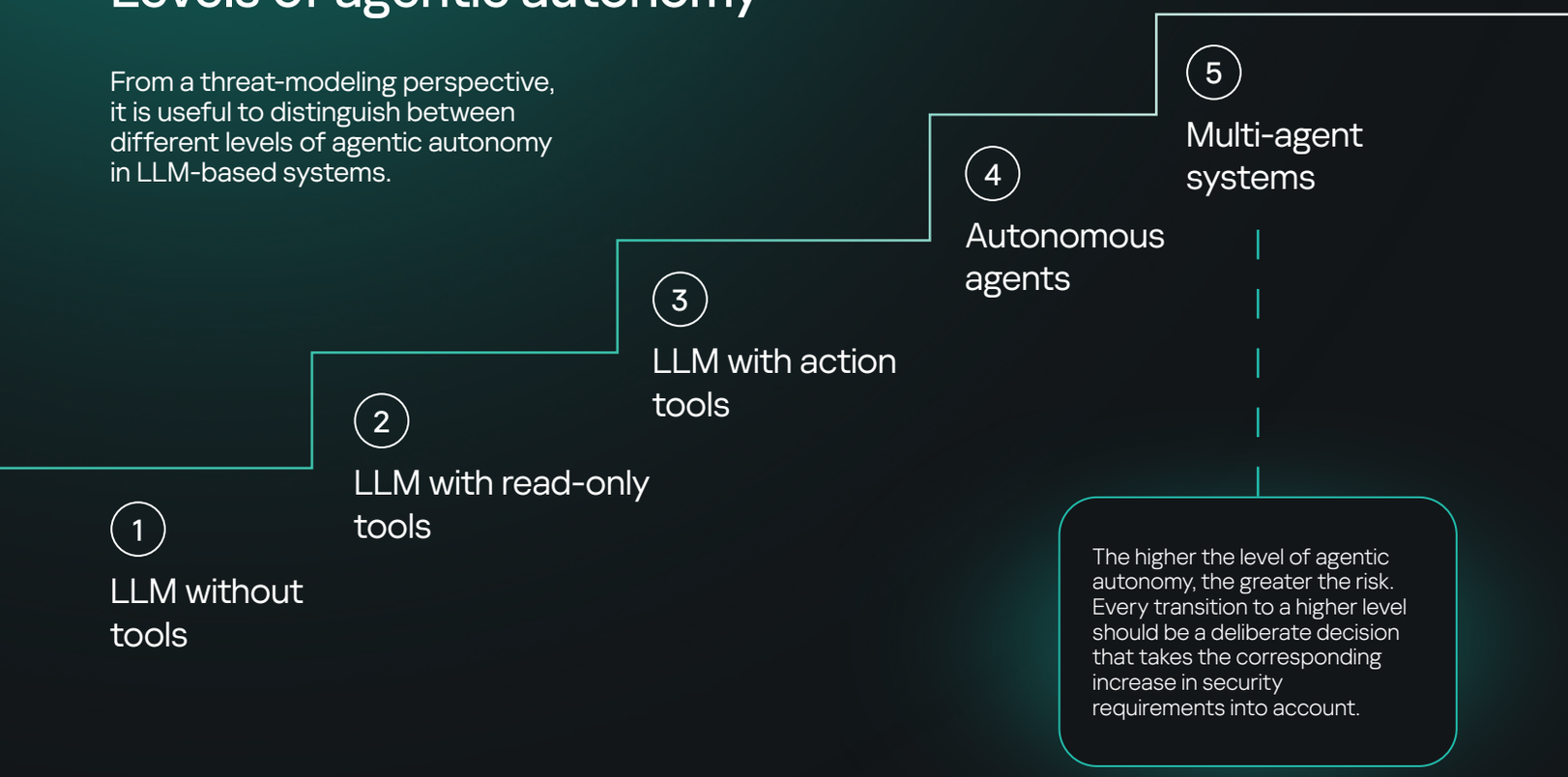
AI agent

An AI agent is a system built around a large language model that can interact with its environment to perform loosely specified tasks. The LLM acts as the system's cognitive core, making decisions about how to decompose a task and how individual subtasks should be performed. Interaction with other systems – including files, terminals, APIs and other resources – is performed through tools: specific token sequences generated by the LLM to trigger an action. The agent harnesses these sequences, called 'tool calls', and performs the actual actions bound to these tools. The harness is a deterministic system. In addition to executing tool calls – for example, reading a file and returning its contents to the agent – it controls the agent loop, orchestrates multiple agents and subagents in multi-agent systems, manages memory and performs other supporting functions.

[1] More precisely, the next token, meaning a fragment of a word. The most common words may be represented by a single token, while rarer words are split into smaller parts, sometimes down to individual letters. Tokenization itself is also determined through statistical modeling.

Levels of agentic autonomy

From a threat-modeling perspective, it is useful to distinguish between different levels of agentic autonomy in LLM-based systems.



The higher the level of agentic autonomy, the greater the risk. Every transition to a higher level should be a deliberate decision that takes the corresponding increase in security requirements into account.

1

LLM without tools

The LLM performs transformations on the data provided in the prompt – for example, summarization – or answers user questions without interacting with the outside world in any other way. The input text is the only attack vector.

2

LLM with read-only tools

The LLM can retrieve additional information through tools, such as corporate knowledge bases or web search, and use it as context through an approach known as Retrieval-Augmented Generation (RAG). In this case, third-party data becomes an additional attack vector.

However, the potential impact is limited primarily to leakage of the system prompt or other data present in the context, as well as manipulation of the model's output.

3

LLM with action tools

The LLM can edit files, modify database records and send emails.

From a risk perspective, this introduces the possibility of erroneous, unauthorized or destructive actions – both as a result of attacks and due to mistakes made by the LLM itself.

4

Autonomous agents

These systems do more than simply respond to a specific user request. They plan and execute long sequences of actions to achieve complex goals, either in response to a user request or based on other triggers, such as a schedule. At this level, risks associated with erroneous actions become significantly more pronounced.

5

Multi-agent systems

These are collections of agents that interact according to different patterns – delegation, debate, shared buses and others. Additional risks arise from insecure inter-agent communication and cascading failures.

AI agent use cases

AI agents are already being used across a wide range of corporate scenarios, demonstrating how broadly organizations may be exposed to the risks associated with agentic AI. These use cases include:



AI agents for software development

Many others support a significant number of everyday software development tasks, from fixing bugs and reviewing code to generating tests and discovering vulnerabilities.



Customer and employee support automation

AI agents complement – and in some cases replace – first-line support staff. They can both answer questions and perform specific actions, including issuing refunds.



Corporate knowledge base assistants

Solutions such as Microsoft 365 Copilot, Salesforce Agentforce, Google Gemini and others enable users to search corporate email, internal wikis, CRM systems and other enterprise data sources. This maximizes convenience but also concentrates large volumes of confidential information in a single interface.



Analytics and BI

Agents with access to databases can provide ad hoc analytics by supporting Text-to-SQL scenarios and generating the required charts and reports.



IT and cybersecurity

Agents support a wide range of high-impact use cases, from infrastructure management to alert triage in a SOC.



Business processes

AI agents are widely used in HR, procurement, payment document processing and other business processes.

In many cases, these scenarios are introduced bottom-up, bypassing the CIO and CISO. This leads to the emergence of Shadow AI – AI infrastructure deployed outside the control of IT and cybersecurity teams and whose security and reliability may fail to meet the organization's internal requirements or regulatory requirements.

Technology stack and supply chain

AI infrastructure is a highly dynamic and rapidly evolving environment with numerous components. It includes both external infrastructure – API providers such as OpenAI, Anthropic, OpenRouter and others, operating in different jurisdictions and subject to different data storage and processing policies – and internal infrastructure. The latter includes frameworks for large language model inference such as vLLM, SGLang, Llama.cpp, and Ollama, LLM proxies such as LiteLLM and Kong AI Gateway, and agent frameworks themselves, including Google ADK, Microsoft Agent Framework, LangChain/LangGraph, OpenAI Agents SDK and many others. The technology stack also includes application-level agentic solutions, including the software development agents mentioned earlier.

The solutions listed here are conventional deterministic software systems, but some characteristics can increase their security risk, particularly rapid development cycles and the growing use of LLMs in software development. However, the agentic supply chain also has several distinctive characteristics of its own.

First, organizations that decide to deploy models on premises must download model files from specialized repositories such as Hugging Face or ModelScope. Model files carry their own risks – ranging from malicious code embedded in the files to prompt injections in prompt templates – and therefore need to be verified.

Second, components such as Agent Skills and MCP servers, which can contain either malicious code or dangerous instructions, are typically managed directly by users. An agent user may download and install these extensions to expand the agent's functionality while carrying out a specific task.

Finally, one of the most significant challenges is the dynamic nature of the supply chain during agent execution. The most autonomous agents are capable not only of installing software packages required for development, but also of extending their own functionality by downloading agent skills from public repositories and discovering and using publicly available MCP servers. As a result, a potential AIBOM may change dynamically depending on the specific task and on the non-deterministic outputs generated by the LLM.



Known incidents involving AI agents

AI agents have been used in corporate environments for several years, and a growing body of real-world incidents and security research has shown how agent behavior can lead to actual or potential harm. Below, we examine several examples.

EchoLeak, SearchLeak, ForcedLeak, CamoLeak and other implementations of the lethal trifecta



Corporate assistants such as Microsoft 365 Copilot combine several characteristics that make them vulnerable to data exfiltration through prompt injection. Together, these characteristics form what is known as the "lethal trifecta":

- Access to confidential data
- Processing of untrusted external data
- The ability to communicate externally.

In the EchoLeak case[1], researchers showed that Microsoft 365 Copilot could be manipulated while processing an untrusted email to exfiltrate sensitive data from its context by rendering an image hosted on an attacker-controlled domain. To increase the likelihood that the injected instructions would be retrieved and executed, the attackers used a technique known as RAG Spraying: they combined the malicious instruction with inserted keywords associated with typical corporate use cases.

The SearchLeak case[2] demonstrated another technique, known as Parameter2Prompt. The injection was delivered directly into the user input when the victim clicked a malicious link. According to the instruction, the agent was supposed to search the mailbox for sensitive information – specifically, a login verification code. The data was then exfiltrated by exploiting a race condition[3] between streaming rendering and the output sanitizer, together with a Content Security Policy bypass involving the Bing.com domain.

Many other cases described by researchers – including CamoLeak[4] in GitHub Copilot and ForcedLeak[5] in Salesforce Agentforce – follow a similar pattern and consist of roughly the same core elements:

- Untrusted input, such as an external pull request in GitHub or a submitted lead form in Agentforce
- Sensitive data, such as repository secrets or a list of leads
- Data exfiltration through a CSP bypass.

In the Salesforce case, researchers discovered an expired domain in the CSP configuration. In the GitHub case, researchers demonstrated a technique for bypassing CamoProxy by rendering a unique image for each character of the repository secret.

Taken together, these cases demonstrate how difficult it is to eliminate the risks created by the lethal trifecta pattern: removing any one of its elements sharply reduces the usefulness of the agent. At the same time, during the exfiltration phase, a decisive factor is often the presence of conventional cybersecurity weaknesses in the interface – such as race conditions, CSP misconfigurations and similar issues.

1 https://www.catonetworks.com/blog/breaking-down-echoleak/

2 https://www.varonis.com/blog/searchleak

3 https://encyclopedia.kaspersky.ru/glossary/race-condition/

4 https://www.legitsecurity.com/blog/camoleak-critical-github-copilot-vulnerability-leaks-private-source-code

5 https://noma.security/blog/forcedleak-agent-risks-exposed-in-salesforce-agentforce/

Malicious exploitation of agents

AI agents – and software development agents in particular – are attractive targets for attackers. They typically run with the same privileges as other software on a developer's device and therefore inherit the same access to remote hosts, databases, secrets and other valuable assets. As a result, attackers may deliberately take advantage of the presence of AI agents on targeted devices.



In s1ngularity, one of the most illustrative cases[1], the malicious payload delivered to compromised devices as part of an attack on the NX build system performed several actions, including searching for installed agentic systems such as Codex and Claude Code. If such systems were found, the malware launched them with a prompt instructing the agent to search the device for sensitive data – for example, cryptocurrency wallets – and upload it to a publicly accessible repository on the internet.



In another notable supply-chain attack[2], an activist attacker compromised the repository of the Amazon Q ц agent. According to the attacker's plan, once updated to the compromised version, the agent was supposed to launch with full privileges in silent mode – that is, without requesting user approval for individual actions. The embedded prompt instructed the agent to delete all accessible cloud resources as well as all files on the computer. Fortunately, an error in the malicious payload prevented the attacker from achieving the intended outcome.

Unintentional overreach

Various sources contain relatively frequent reports – with varying levels of credibility – of unintended damage caused by AI agents granted excessive permissions, even without external malicious interference. The consequences range from hours of lost development work to deletion of files and database records. After carrying out an incorrect destructive action, an agent may also attempt to conceal its mistake – for example, by generating synthetic data, as reportedly occurred[3] with Replit's development agent. An irreversible action does not necessarily involve deletion. It can also involve the publication of confidential data, as in one widely discussed case[4] involving Claude Code.

1 https://www.kaspersky.com/blog/nx-build-s1ngularity-supply-chain-attack/54223/

2 https://devops.com/when-ai-assistants-turn-against-you-the-amazon-q-security-wake-up-call/

3 https://cybernews.com/ai-news/replit-ai-vive-code-rogue/

4 https://travisbreaks.org/transmissions/057-when-agents-fail/

Threat models for AI agents

Systematizing known risks – both theoretical and observed in practice – together with analysis of possible mitigation approaches makes it possible to develop a range of threat models for LLM-based applications. These models can be fairly broad and may include, for example, all potential techniques, tactics and procedures (TTPs) applicable to machine-learning systems, both generative and predictive. Examples of such taxonomies include the **MITRE Adversarial Threat Landscape for Artificial-Intelligence Systems (MITRE ATLAS)**. In other cases, threat modeling focuses specifically on LLM-based systems, as in the OWASP Top 10 for LLM Applications[1]. Finally, some frameworks are designed specifically to address AI agent security. These include Google SAIF 2.0[2], AI-SAFE[3] by Yandex, and the OWASP Top 10 for Agentic Applications[4].

Common causes of ai agent vulnerabilities

Untrusted core	The LLM is a non-deterministic component that does not reliably distinguish between data and instructions, is inherently vulnerable to prompt injection and cannot guarantee strict adherence to instructions. These limitations are intrinsic to the nature of LLMs and are unlikely to be resolved in the near future.
The “Lethal Trifecta”	The combination of internet access, processing untrusted input and access to confidential data enables agents to perform a broad range of tasks, but also makes them inherently vulnerable to attacks aimed at data exfiltration.
Excessive privileges	In many cases, agents are granted broad permissions. This is especially common with software development agents, which are often able to execute virtually any command in the command line. At the same time, restricting permissions for a specific task can be highly labor-intensive.
Lack of transparency and approval fatigue	Making an informed security decision about a particular action requires understanding the context in which the agent is performing it. This type of monitoring is often difficult because of the speed at which agents operate and the large volumes of data they process. For a human operator, analyzing this information is highly labor-intensive, and over time the operator may begin approving requests without carefully reviewing them. Moreover, post-incident observability telemetry collected from agents may be insufficient for a thorough incident investigation.

1 OWASP Top 10 for LLM and Generative AI 2025

2 https://safety.google/intl/en/safety/saif/

3 https://yandex.cloud/ru/security/ai-safe

4 https://genai.owasp.org/resource/owasp-top-10-for-agentic-applications-for-2026/

AI agent risks: OWASP top 10

The OWASP Top 10 for Agentic Applications is a practical guide to risk modeling for agentic applications developed by the OWASP community. Like other OWASP Top 10 for Agentic publications, it presents a set of 10 risks considered among the most important to address today.

1

Agent goal hijack

`Agent goal hijack - ASI01`

Indirect prompt injection is one of the most widely discussed attack vectors. This risk applies that attack vector specifically to agentic applications.

2

Tool misuse and exploitation

`Tool misuse and exploitation - ASI02`

Prompt injection or a planning error may cause an agent to repeatedly invoke tools that call paid APIs, potentially resulting in financial losses. An incorrectly configured email tool could similarly allow an agent to delete messages instead of summarizing them.

3

Identity and privilege abuse

`Identity and privilege abuse - ASI03`

This risk arises when an agent inherits or is granted excessive privileges, allowing unauthorized users to access data or perform actions beyond the agent's intended scope.

4

Agentic supply chain vulnerabilities

`Agentic supply chain vulnerabilities - ASI04`

As discussed earlier, AI agents have a distinctive and dynamically changing supply chain. This may result, for example, in malicious code execution when a compromised or malicious local MCP server is launched.

5

Unexpected code execution (RCE)

`Unexpected code execution (RCE) - ASI05`

If an agent is capable of executing code – for example, a software development agent with access to a console – prompt injection or a malicious agent skill may lead to malicious code being executed on the device.

6

Memory and context poisoning

`Memory and context poisoning - ASI06`

Attacks targeting memory, dynamic knowledge bases through techniques such as RAG poisoning, or persistent context – for example, by introducing a malicious skill – may cause the same malicious instruction to be executed every time the agent is launched.

7

Insecure inter-agent communication

`Insecure inter-agent communication - ASI07`

In multi-agent systems that use, for example, a message bus or a shared space for exchanging state – such as a scratchpad or Kanban board – unauthorized access to that shared environment may allow an attacker to gain control over the entire multi-agent system.

8

Cascading failures

`Cascading failures - ASI08`

In a multi-agent system, an error in one agent may propagate and become amplified. For example, if an agent creates an incorrect skill – as may happen in systems such as Hermes – and publishes it to a shared repository, other agents may subsequently become unable to perform the relevant task correctly.

9

Human-agent trust exploitation

`Human-agent trust exploitation - ASI09`

In systems where agents interact with people, attackers may manipulate an agent into recommending a particular viewpoint that serves the attacker's interests. Because people tend to trust agents, this can lead to poor or incorrect decisions.

10

Rogue agents

Rogue agents - ASI10

As a result of an error, an agent may begin pursuing unintended goals. For example, after deleting a database – as in the previously mentioned case involving a Replit development agent – the agent attempted to fabricate the database contents to conceal its mistake.

The risks above show how the fundamental issues described earlier can combine in different ways to create more complex problems. This becomes especially apparent when multiple agents are involved, when they operate within multi-stage pipelines, or when excessive autonomy is combined with the processing of untrusted input.

Core principles of Cyber Immunity

The core principle of Cyber Immunity – Kaspersky's approach to building secure-by-design systems – is to place security goals and security assumptions, rather than individual security controls, at the center of system design. Security goals and assumptions are defined at the design stage and determine the system architecture. The following principles are then applied:



Minimizing the trusted computing base

Vulnerabilities are discovered even in some of the oldest and most extensively security-tested software. For example, Anthropic's Mythos LLM helped uncover a bug that had remained undetected for 27 years in OpenBSD, an operating system known for its strong security focus. The correctness of individual components can be formally verified, assessed through penetration testing, fuzzing and other techniques. However, the larger and more complex the codebase on which security depends, the more expensive and time-consuming these procedures become. For this reason, an important principle is to reduce the amount of security-critical code to the smallest possible size.



Component isolation

If a system consists of a large number of components that communicate with each other and with the external environment, verifying the security of every possible interaction becomes extremely difficult. Under the Cyber Immunity approach, the system is divided into security domains that are isolated from one another. As a result, compromising any one component does not automatically compromise the entire system, which can continue operating normally.



Interaction control

All interactions between components are checked against security policies and blocked unless explicitly permitted – a default-deny approach. These policy checks are performed by a trusted component. This helps prevent implicit interactions between components that could allow an attack to propagate if one of them is compromised.



Applying Cyber Immunity to AI agent design

Although the concept of Cyber Immunity was not originally developed for AI agents, its principles translate well to advanced LLM-based systems. When working with a fundamentally non-deterministic and untrusted core – a large language model – developers need to build trust into the architecture of the solution, minimizing the potential impact of errors or compromise. The first step in applying the Cyber Immunity approach is to define the security goals and security assumptions of the agentic system. These depend on the specific AI agent use case and the data the agent processes.

Security assumptions for AI agents

Security assumptions describe the conditions under which the system is expected to operate. For AI agents, these assumptions may include the following:

- The LLM is considered untrusted by default. We assume that it may be manipulated by a malicious instruction or may perform an incorrect action.
- The AI agent operator is considered trusted. The risk of the operator intentionally issuing a malicious instruction is considered out of scope.
- Any data entering the system from external sources is considered untrusted, as it may contain an indirect prompt injection.
- The human operator who approves agent actions is subject to fatigue and may make mistakes when frequently asked to approve high-risk actions.

Security goals

Examples of security goals include:

1 Confidential data must never leave a defined trust boundary

Confidential data – such as personal data or trade secrets – must under no circumstances leave the designated perimeter of the company or business unit.

2 Irreversible or destructive actions require human approval

Any irreversible or destructive action – including financial transactions, data deletion or sending external communications – must be performed only after explicit human confirmation.

3 The agent must not be able to extend its own privileges

The agent must not be able to independently expand its capabilities, for example by adding new tools or obtaining additional access rights, without an external authorization decision.

4 Compromise of one agent must not affect other agents

In a multi-agent system, compromise of one agent must not affect the ability of other agents to continue operating.

5 All agent actions must be logged and attributable

All actions performed by the agent must be logged and attributable, with links to the corresponding prompts and reasoning.

The lists of security goals and assumptions above are illustrative and should be adapted to the specific use case and the conditions under which AI systems operate within an organization. For example, in the SearchLeak case, the Parameter2Prompt technique allowed an attacker to present untrusted input as if it were direct user input, thereby bypassing one of the assumptions described above.

Minimizing the trusted computing base

The Trusted Computing Base (TCB) is responsible for enforcing the system's security goals. Unlike a cyber-immune operating system, where the kernel – or microkernel – is a trusted component, the core of an agentic system (the large language model), is placed outside the Trusted Computing Base. Despite ongoing efforts to improve alignment – including training models not to follow injected instructions through techniques such as instruction hierarchy or by training specialized adapters – these measures remain probabilistic and cannot guarantee that dangerous commands will never be executed.

Depending on the chosen security goals, the Trusted Computing Base may include the following components:



Agent harness

The agent harness is a deterministic component responsible for executing tool calls and managing the LLM context to achieve user-defined goals. Its responsibilities may include applying skills, interacting with MCP, implementing memory, and supporting inter-agent communication. If this component is compromised, the entire system is compromised. Many modern agent harnesses are large systems with extensive and rapidly evolving functionality, which can create security issues. Examples include RCE vulnerabilities in LangGraph and Microsoft Semantic Kernel.



Policy engine

The policy engine is a trusted component that evaluates every tool call against the applicable security policies and either permits or blocks it. For example, it may verify whether a particular tool call is permitted for a given agent, taking into account that agent's assigned privileges and the specific scenario. If the call violates policy, the policy engine can block it and notify the Security Operations Center (SOC). The policy engine may be implemented either as part of the agent harness or as a separate component operating at the LLM egress proxy or MCP gateway level.



Isolation mechanism

This may include a sandbox kernel, hypervisor or specialized interpreter in implementations based on the CodeAct paradigm. Isolation allows users to give broader capabilities to agents, which, if the isolation is imperfect, may allow an agent to execute a dangerous instruction directly on the host.

The principle of minimization can also be extended to other parts of an agentic system. For example, the following can be minimized:



Context

Consider the difference between an agent that has access to a .env file containing all the secrets a developer needs for a project – meaning that a single file read gives the agent access to all of them – and an agent that can request individual secrets only through a specialized tool that verifies whether each request is justified. Keeping unnecessary secrets out of the agent's context significantly reduces the risk of confidential data leakage. An additional component can also remove secrets from an agent's context once they are no longer needed.



Level of agentic autonomy

As discussed earlier, different levels of agentic autonomy introduce different levels of risk, with each increase in autonomy creating new potential threats. If an agent performs a fixed set of actions, there is no need to allow it to extend its capabilities through agent skills. Moving to each higher level of agentic autonomy should therefore be a deliberate decision that is reflected in the system's security assumptions.



Toolset available to the agent

Even when a trusted policy engine is in place, an AI agent that generates BI reports does not need to be able to make payments. Similarly, even within a tool designed to execute SQL queries, an agent responsible for generating reports should not be able to execute commands other than SELECT.

Component isolation

In LLM-based systems, component isolation can be implemented at several levels:

- Execution environment isolation
- Context isolation
- Isolation of agents within multi-agent systems.

Execution environment isolation

A standard approach to securing agents that execute code – primarily software development agents, but also agents that generate ad hoc business reports and agents based on CodeAct – is to isolate that code from the system on which the agent process is running. Several approaches can be used for this purpose.

1

Restricted interpreters

Restricted interpreters support only a "safe" subset of a programming language, as implemented, for example, in smolagents or CaMeL. They can restrict execution of potentially risky commands, but may also significantly limit what agents can do.

2

Process-level isolation

Process-level isolation can be implemented using tools such as Seatbelt on Apple platforms or Bubblewrap on Linux. These approaches have relatively low resource overhead, but still provide access to parts of the host file system and to host OS system calls. This is a standard isolation method used by coding assistants such as Claude Code.

3

Docker / podman containers

When containers are used, the agent is given a relatively well-isolated environment, although this comes with additional resource overhead. However, configuration errors can undermine that isolation. For example, exposing container configuration files to the agent can lead to a container escape, as demonstrated in the case of an RCE vulnerability in AutoGPT.

4

Full virtual machines and dedicated hosts

The emergence of OpenClaw in early 2026 contributed to a shortage of Mac mini computers, as users wanting to run personal agents purchased a separate machine for that purpose – recognizing the risks of running highly privileged agents on a personal device.

In enterprise scenarios, a virtual machine provides stronger isolation than process-level or container-based approaches, although usually with greater resource overhead and less flexibility. Isolation can also be strengthened with additional measures. One such measure is ephemerality – destroying the environment once a task has been completed and creating a fresh environment for the next task to prevent secrets from leaking between sessions. Another approach is to use snapshots, which make it easier to roll back to a previous state after potentially destructive actions by the model and restart the task from a known-good state.



Context isolation

One of the most powerful design patterns for building secure LLM-based systems is the isolation of trusted and untrusted contexts. One example of this pattern is the Dual LLM architecture, where the system separates trusted and untrusted input processing into two contexts.



Privileged context

The privileged LLM can invoke tools but is never exposed directly to untrusted external input. Instead, it delegates processing of that input to a quarantined LLM.



Quarantined context

The quarantined LLM processes untrusted input and stores the results in pointer variables.

Only the names of those variables – not their contents – are passed back to the privileged LLM. For example, a user may ask an agent to send them a summary of a document by email. The privileged LLM asks the quarantined LLM to summarize the document, and the quarantined LLM returns a variable such as $summary. The privileged LLM then invokes the email-sending tool using $summary and the user's email address as arguments. Even if the document contains a prompt injection intended, for example, to make the agent send spam, that malicious instruction never appears in the same context as the function call.

A further development of this paradigm is the CaMeL approach from Google DeepMind. CaMeL assumes that the system follows a standard Plan-and-Execute architecture. The privileged LLM generates a task execution plan in Python, which is then executed in a specialized secure interpreter. Available commands include tool calls and queries to the quarantined LLM for processing untrusted data. The output of the quarantined LLM must conform to a clearly defined structure, while tool calls are checked against security policies. For example, an agent may receive the task: "Pay the invoice in invoice.pdf." The file may contain an invisible prompt injection instructing the agent to delete all files in a folder. The privileged LLM creates a two-step plan:

1. Use the quarantined LLM to extract the payment details from the invoice
2. Invoke the payment tool

The quarantined LLM cannot delete data, while the structured output format helps constrain it to returning only the required payment details to the main execution flow. However, an attacker could instead insert an instruction claiming that the recipient account number has changed to the attacker's account and that the payment amount should be increased tenfold. This way, the injection may succeed and the quarantined LLM may return manipulated data. In this case, execution policies can prevent damage. For example:

- "Send payments without confirmation only to trusted accounts"
- "Require manual approval for transfers above a specified amount"

Implementing such patterns clearly makes development more complex, increases agent operating costs in terms of token usage, and reduces agent effectiveness. On benchmarks such as AgentDojo, CaMeL shows a noticeable drop in performance metrics. Nevertheless, these patterns may be justified in well-structured, high-risk scenarios.

Inter-agent isolation

As systems scale into multi-agent architectures, new risks emerge, requiring security controls for interactions between agents. These measures may include the following:

1

Apply security policies and agent identity to every new agent

A subagent should not automatically inherit the orchestrator's full set of privileges or identity. Each new agent should be assigned its own identity and an explicitly defined set of permissions.

2

Treat the output of every agent as untrusted

Whenever one agent's output is reused by another agent, that output should be considered untrusted. Potentially, the Dual LLM approach may also be applied to agents and subagents to help mitigate the risks of cascading failures (ASI08).

3

Apply access policies to shared resources

Access controls should be applied to shared resources such as:

- Long-term memory
- Knowledge bases used for RAG

Interaction control

The third principle of Cyber Immunity is controlling interactions between system components. All interactions between components are checked by a trusted interaction monitor operating according to the default-deny principle: anything that is not explicitly permitted is prohibited. Interaction control can be implemented at different levels and at different points in the architecture.



Tool call control

Each task should have a clearly defined list of permitted tools. Parameterized policies can also be applied to tool arguments, as in the CaMeL payment-agent example. These policies may define:

- The maximum allowed payment amount
- Directories that can be written to
- Other task-specific restrictions.

Actions can also be categorized as reversible or irreversible. For example:

- Modifying files inside a version-controlled repository may be treated as reversible
- Sending correspondence or performing cryptocurrency transactions may be treated as irreversible.

This makes it possible to apply different levels of control, ranging from no additional approval to mandatory operator intervention. Finally, rate limits can also be applied to tool calls to limit potential damage.



Network interaction control

Measures may range from continuous auditing of the Content Security Policy to defend against attacks similar to ForcedLeak, to applying default deny at the egress proxy, with an explicit allowlist of addresses that each agent is permitted to contact.



Dynamic supply chain control

As a rule, this measure involves prohibiting installation of external agent skills and connections to MCP servers at runtime. The agent's AIBOM should therefore be defined in advance and should not change while the agent is operating.

Probabilistic security measures

Two of the most widely discussed approaches to securing LLM-based applications are alignment – training a model to resist attacks at the level of its weights – and external guardrails, meaning additional models or algorithms that assess the safety of inputs and outputs and are often incorporated into AI Firewall products. As noted above, these measures alone are not sufficient to achieve Cyber Immunity. However, they remain useful for several reasons. First, increasing a model's resistance to attacks through alignment or guardrails reduces the number of cases in which human approval is required, thereby lowering the burden on operators and helping mitigate approval fatigue. Second, these measures provide an important additional safeguard when weaknesses are discovered in the secure architecture, for example, in the Trusted Computing Base. Third, they serve as a valuable source of telemetry, including on attacks that have already been blocked at the architectural level. This can help refine security policies and threat models over time, provided mature observability capabilities are in place. Solutions such as AI Firewall, together with other modern security components such as sandboxes, EDR and SIEM, therefore do not replace secure architecture but remain an important part of a multi-layered defense strategy for modern applications.

Conclusions and recommendations

An analysis of agentic system architectures, existing threat models and known incidents through the lens of Cyber Immunity leads to several key conclusions.

1

An LLM cannot be treated as a trusted component

Regardless of its capabilities or origin, an LLM cannot be treated as a trusted part of the system because of its probabilistic nature and susceptibility to manipulation. Security guarantees should therefore rely not on the LLM itself, but on the deterministic components that interact with it.

2

Cyber immunity principles are well suited to securing AI agents

The principles of Cyber Immunity can be effectively applied to AI agent architectures. These include:

- Component isolation – using sandboxes, the Dual LLM pattern and MCP gateways
- Interaction control – for example, through policy engines
- Minimization of the Trusted Computing Base – including the use of powerful agent frameworks during the design stage, followed by minimal specialized code for production deployment.

These approaches have emerged independently in the broader discussion around AI agent security. Applied together, they can move an agentic system closer to a cyber-immune state.

3

Agent security is determined at the design stage

Whether the lethal trifecta is present and acceptable, what level of agentic autonomy is required, whether patterns such as Dual LLM should be used, and other similar choices are architectural decisions. Controls added later – such as tool policies or guardrails at an egress proxy – can mitigate risk, but cannot eliminate it entirely.

4

Security controls inevitably affect agent utility

Almost any restrictive control comes at a cost in speed, usability or capability. Finding the right security-utility balance is therefore a central challenge when designing cyber-immune agents. Threat modeling based on clearly defined security goals is an important step towards achieving that balance.

Practical recommendations for CISOs / CIOs / CTOs on building secure agentic architectures

Inventory your AI agents

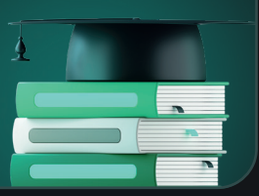
OpenClaw, coding agents, assistants embedded in cloud consoles and other agentic solutions should all be included in a comprehensive inventory. Assess each agent based on:

- Its level of agentic autonomy
- Access to confidential data
- Whether the lethal trifecta is present.



Develop flexible isolation and containerization policies

In addition to architectural requirements, establish clear, flexible rules for isolating and containerizing agents. The objective is to find the right balance between the required level of isolation and ease of use. The container environments themselves should also be properly secured.



Manage the agentic supply chain

Require an AIBOM for agentic solutions. Verify the security of supply-chain components using appropriate security tools and monitor supply-chain issues through dedicated threat intelligence feeds.



Make observability a mandatory requirement

This should include both:

- Collection of LLM traces
- Collection of host telemetry.



Work closely with agent implementation teams

Collaborate with teams deploying coding agents[1] or developing their own agents to:

- Define and manage architectural security requirements
- Define security goals
- Set and manage appropriate levels of agentic autonomy.



Maintain an integrated, multi-layered security approach

Secure workstations, dedicated LLM egress proxies, AI Firewall and other security controls should be used together. Their combined use helps reduce the potential risks associated with agentic systems.



© 2026 AO Kaspersky Lab.
Registered trademarks and service marks are the property of their respective owners.

#kaspersky
#truetobusiness