Systems Security Foundations for Agentic Computing

Mihai Christodorescu[1], Earlence Fernandes[2], Ashish Hooda[1],
Somesh Jha[1,3], Johann Rehberger[4], Khawaja Shams[1]

[1]*Google* [2]*UC San Diego* [3]*UW Madison* [4]*EmbraceTheRed*

December 1, 2025

This paper articulates short- and long-term research problems in AI agent security and privacy, using the lens of computer systems security. This approach examines end-to-end security properties of entire systems, rather than AI models in isolation. While we recognize that hardening a single model is useful, it is important to realize that it is often insufficient. By way of an analogy, creating a model that is always helpful and harmless is akin to creating software that is always helpful and harmless. The collective experience of decades of cybersecurity research and practice shows that this is insufficient. Rather, constructing an informed and realistic attacker model before building a system, applying hard-earned lessons from software security, and continuous improvement of security posture is a tried-and-tested approach to securing real computer systems. A key goal is to examine where research challenges arise when applying traditional security principles in the context of AI agents. A secondary goal of this report is to distill these ideas for AI and ML practitioners and researchers. We discuss the challenges of applying security principles to agentic computing, present 11 case studies of real attacks on agentic systems, and define a series of new research problems specific to the security of agentic systems.

1. Introduction

The IEEE Secure Generative AI Agents 2025 workshop (SAGAI; co-located with the IEEE Symposium on Security and Privacy) set out to explore how computer-security research and practice can help inform and advance agentic computing security. This report captures those discussions and distills key lessons, open research problems and recommendations on how the two fields can co-operate to build a foundation for secure agentic computing.

Agentic computing is when AI models interact with computer systems and services on behalf of human users by calling tools in a loop [1]. The user specifies a task in natural language and the AI model is usually a large language/vision model but can be other types as well. The set of computer systems that can be operated by an agent include desktop interfaces, terminals, web browsers, mobile operating systems. Furthermore, agents can use arbitrary tools defined through various standards (e.g., REST or MCP [2]).

Current efforts at securing agentic systems primary rely on model hardening approaches, often described as AI alignment [3], [4], [5]. Fundamentally, this involves training a model to resist attacks by exposing it to various exemplars. From a computer security perspective, this is equivalent to writing software that is always helpful, harmless and resistant to attacks. The past 50 years of research and practice in computer security has empirically shown this to be an insufficient strategy. While a single program's robustness should certainly be increased (e.g., computer security does this using formal methods approaches for certain kinds of programs or by writing software using languages (e.g. Rust) that are immune to certain classes of attacks), it has proven to be very difficult to scale to all types of computer programs. This has necessitated a *systems approach* that applies security defense techniques at various layers of abstraction of a computer system (i.e., hardware, firmware, hypervisors, operating systems, user-space frameworks, applications, network). The essence of this approach is to create *deterministic* guardrails at different layers of abstraction that increase the exploitation burden on the attacker while minimizing performance and functionality impact.

The situation with AI is worse. Models are not traditional computer programs that are immediately amenable to traditional computer-security techniques that increase their robustness. They are also vulnerable to inference-time attacks that allow attackers to control outputs in arbitrary ways [6], [7], [8], [9]. Furthermore, the collective experience of the community between the two "waves" of adversarial machine learning research is that training models to resist attacks is insufficient: attackers always find workarounds [10], [11], [12]. Additionally, in the LLM-era of this research, attackers are not sophisticated and rely on English-language skills to trick AI systems

into pursuing malicious tasks [9], [13], [14], [15], [16]. Therefore, to make foundational and practical progress, the fields of computer security and AI agent safety/security need to co-operate.

Summary of Findings. Computer security relies on a range of security design principles to guide the exploration and implementation of deterministic guardrails [17], [18]. Applying those principles to agentic systems encounters several technical challenges that we explain and explore in this report based on discussions during the workshop. Each of these challenges also pose interesting research challenges.

(1) Probabilistic TCB. A classic principle is creating a *trusted computing base (TCB)* to enforce a security invariant (e.g., to reduce the risk of memory corruption vulnerabilities, the hardware provides a deterministic feature that prevents code execution only from certain memory regions). Applying this to an agentic system is difficult because the model itself is a major part of the TCB, but it is fundamentally probabilistic. If you train the model to recognize that certain token sequences are non-executable, that property is enforced with some probability that is conditional on the input to the model rather than being a deterministic property. In other words, the challenge is to build provable defenses from probabilistic TCB assumptions.

(2) Dynamic Security Policies. A security policy governs the privilege of an untrusted program and is typically created by a developer or inferred based on code analysis. Applying this to an agentic system is difficult for two reasons: (a) there is no developer; only the end-user who specifies a natural language task and (b) there is no program to analyze; rather, the AI model "computes" many programs. Predicting or synthesizing a security policy given the natural language task is challenging because the task can be under-specified and using the agent's entire context window for precise predictions is tricky because of the presence of untrusted external content.

(3) Fuzzy Security Boundary. Deterministic guardrails in computer security should be applied at the correct level of abstraction that provides the right amount of information to enforce the guardrail. For example, Android permissions are enforced at the interface between an application and the protected system services. Applying these principles to an agentic system is difficult because there aren't any abstraction layers–typically the model just outputs a tool call with optional reasoning traces. Sometimes, these tool calls are too low-level to write and enforce reasonable security policies. For example, web agents perform UI-level manipulations making it difficult and brittle to write and enforce guardrails at the level of clicks and keystrokes. This is an instance of a well-known classic security problem of a semantic gap [19], [20], [21].

(4) Prompt Injections $\cong$ Dynamic Code Loading. Prompt injections are a key security problem to solve in agentic systems, but are sometimes necessary for functionality. For example, tool documentation does affect an agent's behavior (by definition because the agent has to learn how to use the tool!). The challenge here is defining when prompt injections are harmful and when they are useful and then distinguishing the two. The analog in computer security is dynamic code loading–a very challenging problem that remains difficult to address today.

Scope and organization of this report. There has been a range of recent and contemporaneous writings on the general theme of applying computer security ideas to agentic computing [22], [23]. The key distinction of this report is that we take a deep-dive on the technical challenges that arise when applying concrete security principles and articulate open research problems based on those technical challenges.

The paper starts out with a discussion of classic computer-security design principles and security mechanisms (Section 2), intended for AI/ML practitioners and researchers. Following that, we discuss the application of these security principles to agentic settings and surface the challenges that arise forthwith (Section 3). We introduce in Section 4 case studies of real attacks on agentic systems with three goals in mind: (1) to showcase how real attacks are carried out by adversaries without the need for sophisticated technical tools to execute them; (2) to highlight the key security principle(s) that were violated; and (3) to outline the concrete security mechanisms that stop each such attack. These case studies not only illustrate how existing security mechanisms go a long way towards solving present-day attacks on agentic systems, but also show the types of the gaps that appear due to the challenges discussed earlier. Closing these gaps requires new research to advance the state of the art in agentic security—we summarize the open research problems in Section 5.

2. Security Principles and Mechanisms

The standard security architecture consists of the trusted computing base (TCB), a security policy, the security boundary, and the untrusted system, as shown in Figure 1. The TCB consists of the functionality (code and data) whose integrity and confidentiality cannot be impacted by an attacker—in other words, the TCB defines the parts of the overall system that can be trusted to operate correctly under attack. The TCB typically contains the core functionality of the system (e.g., the kernel of the operating system) as well as a reference monitor that serves to

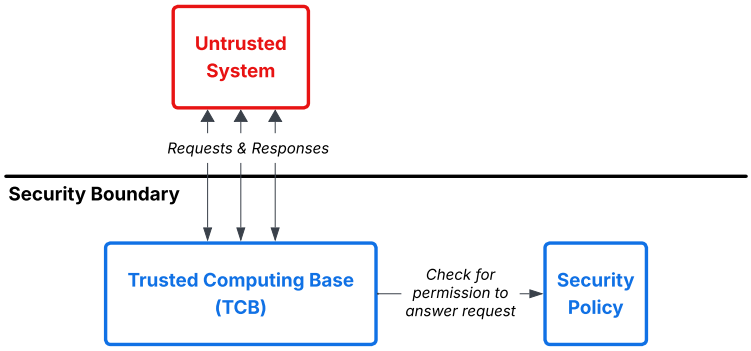

Figure 1. Standard security architecture showing trusted components in blue and untrusted ones in red.

approve or reject each request from the untrusted components to the TCB. The reference monitor is the part of the TCB that examines the security policy in order to make the approve/reject determination. The security policy is typically a declarative expression of the security goals—as an example, the security policy for stored files is given as a list of access control entries, each entry representing the operation(s) a given user is allowed to perform on a given file. The security boundary represents the interface through which the untrusted components interact with the TCB. In an operating system (OS), the security boundary between the OS kernel and the (untrusted) applications is the system-call interface.

In layered system designs, the TCB crosses multiple layers, with each TCB layer providing some functionality that enables the TCB layers on top of it to create their own security guarantees. As an example, most contemporary hardware exposes memory-management functions, allowing the OS kernel to create isolated memory regions by managing the memory-access permissions carefully. In turn the OS kernel uses these isolated memory regions to separate applications from each other and exposes functions to enables applications to share code, data, files, and other resources.

Within this context we can define and describe the security principles introduced in the case studies of Section 4.

Principle of Least Privilege. This principle dictates that the Security Policy should only grant the Untrusted System the absolute minimum permissions it needs to function. For example, if the Untrusted System only needs to read a specific piece of data, the Security Policy should explicitly deny it permission to write, delete, or access any other data. The TCB is responsible for enforcing this minimal set of permissions, as specified by the Security Policy. For convenience most policies are written with the expectation of a *default-deny* fallback, where access to a resource not explicitly listed in the policy is automatically rejected.

TCB Tamper Resistance. This principle states that the TCB itself must be protected from modification by any outside influence. The Security Boundary must be designed to prevent the Untrusted System from altering the code or logic of the TCB or the Security Policy. If the TCB could be tampered with, an attacker could disable the "Check for permission" and bypass all security controls.

Complete Mediation. This principle dictates that *every single* request from the Untrusted System must be validated against the Security Policy. The system must not "remember" a previous authorization as users, trust levels, or other contextual conditions may have changed. Each time one of the requests crosses the Security Boundary, it must be intercepted by the TCB, which in turn must perform the "Check for permission to answer request" operation against the Security Policy. No request is allowed to bypass this check.

Secure Information Flow. This principle governs the "Requests & Responses" channel, ensuring that sensitive information does not leak to untrusted areas. The Security Policy must define what kind of information is allowed to flow in which direction. For instance, even if a request is valid, the TCB must check the Security Policy to ensure that the resulting Response does not contain secret data that the Untrusted System is not authorized to see

(this would constitute an unauthorized flow of data from high-trust components to low-trust components).

Human Weak Link. This principle states that human operators can compromise this architecture in several ways, and thus the security mechanisms must be designed with this in mind. As a user, a human operating the Untrusted System could send malicious requests (e.g., SQL injection, prompt injection). As an administrator, a human could incorrectly configure the Security Policy, making it too permissive (violating Least Privilege). As a developer, a human could accidentally introduce a bug into the TCB that fails to perform the *Check for permission* operation correctly (violating Complete Mediation).

A related principle, *Secure by Default*, addresses some of the human weak-link concerns by ensuring that a newly deployed system comes with a default configuration that is secure, and that it becomes insecure only when the user changes that configuration. Secure by Default reduces the risk of damage at scale, as most users typically do not change the default configuration. While Human Weak Link is more general than Secure by Default (as it recommends hardening against inadvertent compromise even beyond the default configuration), for the purposes of this paper we use them interchangeably.

This security architecture, together with the five principles above, ensures that the overall system is secure even when there are one or more untrusted components present. We note that security assessments performed in such a setting provide guarantees only for a given set of components (a certain version of the TCB, a certain set of request types and their semantics, and a security policy written with respect to these components). A change in the TCB or the Security Boundary requires a new assessment to determine whether the five principles still hold true.

3. Challenges in Applying Security Engineering to Agentic Computing

3.1. The TCB Is Probabilistic

The *Trusted Computing Base (TCB)* is a component or assumption that cannot be influenced by the attacker. TCB is an invariant which is basis of security, such as the no-execute or NX bit in hardware (e.g. data put in region with NX bit set can never be executed or interpreted as instruction).

In traditional systems, the TCB consists of deterministic components such as hardware, OS kernels, reference monitors that behave predictably. This determinism enables security guarantees: W$\oplus$X memory protection consistently prevents execution of writable memory, and Content Security Policy deterministically blocks unauthorized scripts.

In contrast, the TCB in agentic systems is built on LLMs that do not guarantee deterministic behavior—the same prompt can yield different outputs due to sampling randomness. Thus a fundamental challenge in realizing this traditional security principle in agentic systems is the TCB being probabilistic or non-deterministic (e.g. imagine whether we could build a memory safety defense on top of a probabilistic NX bit). This probabilistic nature undermines traditional security principles in three ways. First, a probabilistic reference monitor might correctly deny unauthorized access 99% of the time, but the remaining 1% can be exploited. Second, unlike traditional systems where security properties can be formally verified, we cannot prove an LLM will always enforce policies correctly. Third, LLMs operate in continuous representation spaces, making them vulnerable to adversarial examples. Unlike traditional systems that have discrete instruction sets with clear boundaries, attackers can use various search and optimization techniques to efficiently find inputs that appear benign but cause policy violations.

To build secure systems on probabilistic LLMs, we must leverages external deterministic information to disambiguate LLM decisions. An example of a system takes this approach is SkillFence [24]. It uses deterministic signals from the user's browsing history and installed apps to correct any mistakes made by a voice assistant's language understanding. However, we believe that building provable defenses on probabilistic and non-deterministic TCB is a challenging—perhaps impossible—task.

3.2. The Security Policy Is Dynamic and Task-Specific

In traditional computer security, a security policy governs the privilege of a piece of code. For example, in Android, the OS developers provide a set of permissions that an app developer can select from. The key aspect is that the app is generally single-purpose, built to perform a fixed set of tasks, and thus, the app developer can create a security policy and present it to the user at the time of installation. Furthermore, one can analyze the app's code

to determine why certain permissions are needed and whether they are necessary for the app's stated functionality (thus allowing for analyzing compliance with the principle of least privilege).

In agentic systems, there is no app developer and there is no app. Rather, there is only a natural language task specification that can change over time. Thus, the privilege of the agent is dynamic and needs to be predicted from the task description in a secure manner. For example, consider a simple developer task assigned to a browser agent:

```
comment on GitLab issue 7745 that we are working on it
```

From this task description, the security policy would need to incorporate the following components:

1) The agent can navigate to GitLab issue 7745;
2) The agent should have write access to the comment box;
3) The agent should not have access to anything else in GitLab or the user's session in the browser.

This least-privilege security policy is task-specific and only known once the task is specified.

Continuing this example, let's say that the user specifies a follow-up task for the agent:

```
summarize issue 7745 for me
```

Now, how should the privilege of the agent change to solve the new task description? One option is to reset the agent's context and redo the policy prediction. This approach is secure but can result in lost utility because the agent's context is erased. Thus, how can we evolve the security policy in a secure manner while there is untrusted data in the context? We believe that the key lies in creating policy languages that are amenable to formalized reasoning so that when new policies are predicted and added to the existing set, we can reason about how the new policies change the overall privilege level of the agent.

Extrapolating from this example, we posit the following research challenges: (1) Domain-specific policy languages for different types of agents that are amenable to formal analysis and reasoning; (2) Dynamic policy prediction in these DSLs given natural language tasks, from trusted and untrusted context.

3.3. The Security Boundary Is Fuzzy

Identifying the right abstraction at which security policies are enforced is crucial to upholding the complete mediation design principle. In traditional computer systems, we have layered architectures (Hardware - OS Kernel - Process - Network) that offer different levels of abstraction for enforcing security policies. For example, the process-kernel interface allows for SELinux-style MAC policies and the App-Runtime interface allows for higher-level permissions that we see in systems like Android. Agentic systems lack these layers. Rather, they directly use tools given a natural language prompt. There are no layers in between. Thus, enforcing a security policy at the final layer of tool-calls can lead to incomplete or ineffective mediation (i.e., there can be gaps). The key research challenge in applying the complete mediation principle is identifying or creating the right abstraction at which a security policy can be enforced.

We discuss two recent efforts that have started investigating these challenges along two dimensions: (1) creating an abstraction over which security can be enforced; (2) finding the right level of abstraction in an existing agent design.

CaMeL [25] and *FIDES* [26] introduce an agent design that exemplifies the "solution as code" approach. To solve a task, the CaMeL/FIDES agent first generates code that, when executed, would solve the task specified in natural language. Notice that this introduces a new layer of abstraction: the code needed to solve the task. At that point, one can analyze the code using standard security analysis methods (control and data flows) to determine whether the agent's plan is aligned with the user's goal. A key assumption in this design is that the agent has access to a set of semantically-meaningful tools. One of the key future research questions here is creating an appropriate intermediate representation for agent plans that is amenable to static and dynamic analysis.

ceLLMate [27] introduces a sandboxing framework for browser agents. The key challenge is identifying the right abstraction to enforce security policies within existing infrastructure. Browser agents, as currently built, have access to low-level UI manipulation tools (e.g., clicks and keystrokes), and thus, it is non-sensical to write policies at that level. The key insight to address this semantic gap is realizing that no matter what UI manipulations the agents perform, HTTP-level messages capture those actions in semantically meaningful ways. Therefore, ceLLMate

policies are specified and enforced at the HTTP level, in co-operation with information supplied by webserver developers. One of the key future research challenges here is automatically predicting policies at the level of HTTP messages using natural language task descriptions as input.

3.4. Prompt Injection ≅ Dynamic Code Loading

Prompt injection, more specifically indirect prompt injection, is seen as a vulnerability of current AI-based systems. But prompt injection is a manifestation of a new class of functionality in AI, where the agent can adjust its task based on new information in its context. For example, using the GitLab task in Section 3.2, the agent may start with this prompt:

```
summarize issue 7745 for me
```

then visit the GitLab page for that issue, see that the issue depends on another issue, and decide to summarize both to give the user a comprehensive answer. The distinction between (undesirable) prompt injection and (desirable) contextual task adjustment rests on the provenance of the input that triggered the change and the scope of the change.

Prompt injection (benign or malicious) has a parallel in non-AI systems in the form of dynamic code loading. This is common in applications that support plugins or extensions (e.g., OS kernels, IDEs, web browsers) and has long posed significant challenges in terms of security. The typical solution is to introduce another security boundary between the main TCB and the dynamically loaded code and to use permissions or sandboxing to limit the access afforded to the new code. For example, web pages (probably the most popular platform for dynamic code loading, as each page load and subsequent interaction can potentially load additional Javascript code as web-page scripts) are secured through a number of sandboxes and associated security policies: Content Security Policy determines which third-party scripts can be loaded, Same-Origin Policy restricts the web page data accessible to a third-party script, `<iframe>` sandboxing isolates third-party content, and Subresource Integrity ensures that third-party scripts have not been modified from the time it was approved by the developer.

Current agentic systems lack both components that provide security in web page dynamic code loading. First, information about the source (or provenance) of an instruction given to the agent is hard to assess (in contrast to the source of web-page scripts, most commonly loaded through explicit URLs). Second, sandboxing in the agent context is unavailable or probabilistic at best via mechanisms such as Instruction Hierarchy [4]. An additional challenge to further drive the contrast with the web-page security model is that web pages are designed by a website owner, who is in charge of determining which code to include on the page and indirectly which code to dynamically load at some later time on the page. Agents are often given underspecified instructions and are expected to refine them with additional instructions discovered while working on the task at hand. This is a highly desirable functionality of agentic systems (not having to fully specify the task completely in the first prompt), but one for which security mechanisms are not readily available.

4. Case Studies of Real Attacks on Agentic Systems

We present here 11 attacks against agentic systems to illustrate the variety of types of vulnerabilities that occur in practice. For each attack we also highlight the security principle(s) that were violated and the potential defenses.

Microsoft Copilot Exfiltration. A vulnerability in Microsoft 365 Copilot allowed attackers to steal private user data, such as emails, by sending a malicious message containing a hidden prompt [28]. When a user interacted with this message (asking, for example, for a summary) via Copilot, a prompt-injection attack was triggered, letting the attacker take control of the agent. The compromised Copilot was then instructed to find sensitive information, encode it using "ASCII smuggling," and embed it into a seemingly harmless hyperlink. When the user clicked this link, their data was secretly sent to the attacker.

This attack violates the security principles of *Least Privilege*, *Complete Mediation*, and *Secure Information Flow*, as Copilot automatically performed unexpected actions sourced from a document of unknown origin (like searching for and exfiltrating data) without verifying each step with the user, failing to check that the AI's operations were fully authorized. This vulnerability is similar to that of traditional code injection after a buffer overflow, where

an adversarially crafted input allows the attacker to run code of their choice inside the victim process, coupled with insufficient access control.

Defense: Implement strict output sanitization to detect and block data ex-filtration channels like "ASCII smuggling" in generated hyperlinks. Require human approval whenever the agent accesses sensitive data. Because the TCB is probabilistic (the CoPilot LLM is part of the TCB, Section 3.1) and the security boundary for Internet access is fuzzy (the set of safe URLs cannot be practically enumerated in full, Section 3.3), these defenses provide only incomplete security guarantees and will need to be supplemented with mechanisms for separating instructions and data (Section 5.1) and for least-privilege access control (Section 5.2).

Devin AI Exposed Ports. The AI agent Devin comes with tool called `expose_port`, meant for testing, that was abused through an indirect prompt injection attack [16]. An attacker hosted a malicious prompt on a website that, when visited by Devin, hijacked the agent. The compromised AI then started a local web server, exposing its entire file system, and used the `expose_port` tool to make this server publicly accessible online. The resulting URL was then sent to the attacker, granting them full access to Devin's files.

This vulnerability violates the principles of Least Privilege (as the `expose_port` tool had excessive permissions, allowing it to expose any port, including one with access to the entire file system, rather than being restricted to only what was necessary for its intended function) and of Security Information Flow (as the agent accepted instructions received from an unknown origin). This vulnerability is similar to that of traditional code injection after a buffer overflow.

Defense: A potential solution is to restrict the `expose_port` tool to a predefined safe range of ports by default and configurable only from outside the agent's sandbox. However, such a safe list might be prompt specific and thus presents the challenge of a dynamic, task-specific security policy (Section 3.2). A strong defense will need controls that guarantee minimum-privilege access, an open problem for agentic systems (see Section 5.2).

ChatGPT Long-Term Memory SpAIware. A vulnerability in the ChatGPT macOS app allowed for persistent data exfiltration by injecting malicious instructions into the app's "Memories" feature [15]. This was accomplished through a prompt injection in an untrusted document or website, which caused the application to continuously send all of the user's conversations to an attacker-controlled server. The data was exfiltrated by rendering an invisible image that included the user's data as a parameter in the image URL.

This attack violates the principles of *Trusted Computing Base (TCB) Tamper Resistance* and Secure Information Flow because it stores data from an untrusted origin into the Memories storage (presumably trusted by the agent) and creates an unauthorized channel for sensitive information to be leaked from the application to an external, malicious server. In addition to the similarity with code injection after buffer overflow, this vulnerability also allows the attacker to achieve persistent control, which is one of the fundamental issues with dynamic code loading (Section 3.4).

Defense: All data entering "Memories" should be sanitized before storage or, at a minimum, passed through a human-in-the-loop check (as other agents do by *suggesting* memories to the user instead of automatically persisting them). Additionally, the app's rendering layer should prevent automatic loading of remote sources from untrusted origins. The deployed fix performs an additional check via a `url_safe` tool to control which URLs ChatGPT will connect to, in turn making exfiltration more difficult.. We note that sanitization for untyped data (such as natural-language text) and determining the trust level of a remote Internet resource (such as a web page) both often rely on classifiers, raising concerns of building on a probabilistic TCB (Section 3.1).

Amp AI Code Arbitrary Command Execution. Sourcegraph's Amp AI coding agent allowed for arbitrary command execution by exploiting the agent's ability to modify its own configuration file [29]. Through a prompt injection attack, an adversary instructed the AI to alter its `settings.json` file, either by adding malicious commands to an allowlist for automatic execution or by adding an attacker-controlled server to the configuration, both of which led to running unauthorized code on the developer's machine.

This attack fundamentally violates the principles of TCB Tamper Resistance (as the AI agent, a component of the trusted system, was able to modify its own security-critical configuration files, thereby compromising the integrity of the system's security policies) and Secure Information Flow. In addition to the similarity with code injection after buffer overflow, this vulnerability has parallels with privilege escalation, where the attacker can modify the security configuration of their environment to gain higher access.

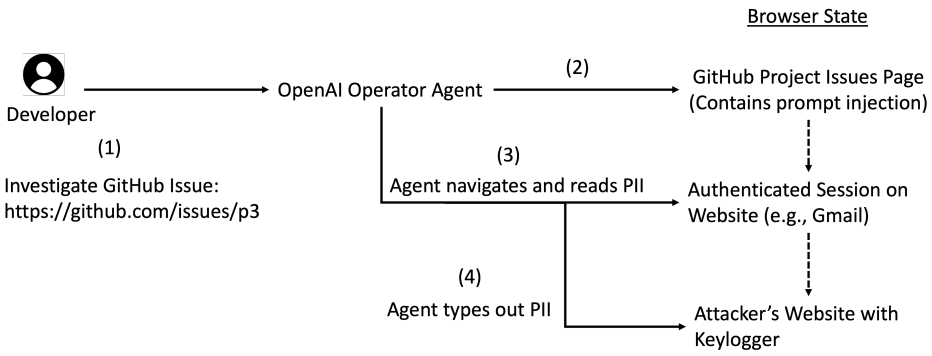

Figure 2. OpenAI Operator Exploit Flow: Starts from an issue on a GitHub project and ends with PII being leaked to the attacker's domain.

Defense: Such systems should enforce immutability on security-critical configuration files (like `settings.json`) so the agent cannot modify its own execution environment. Any changes to these configurations should require human approval. These defenses highlight the challenges of fuzzy security boundaries (Section 3.3) and dynamic code loading (Section 3.4), for which agentic systems have no guaranteed solutions (see Sections 5.2 and 5.3 for further discussion on these open problems).

DeepSeek AI Account Takeover. A vulnerability in the DeepSeek AI platform led to a full account takeover by chaining a prompt injection with a Cross-Site Scripting (XSS) exploit [30]. An attacker uploaded a malicious text file containing a base64 encoded JavaScript payload, which, when processed by a victim's account, instructed the AI to decode and execute the script in the victim's browser, stealing the `userToken` from `localStorage` and sending it to the attacker.

This attack violates the principles of Secure Information Flow (as it established an unauthorized covert channel to leak a sensitive session token from the user's browser to an external, malicious endpoint, completely bypassing the platform's intended data handling and security boundaries) and *Complete Mediation* (as the agent processed encoded data meant to bypass input filtering). Chained code injections are often used in traditional exploits to complete an attack.

Defense: Ensure a strict separation between data (uploaded text files) and executable web code. The fuzzy security boundary (Section 3.3) coupled with the need to allow for some flexibility in using external data as prompts (a form of dynamic code loading, Section 3.4) makes it challenging to secure the agentic system end-to-end when no good mechanisms for separating instruction and data are available (see Section 5.1).

Terminal DiLLMa. The "Terminal DiLLMa" attack hijacked a user's terminal through LLM-powered command-line tools by using prompt injection to generate malicious ANSI escape sequences [31]. When a compromised tool, such as the proof-of-concept `dillma.py`, processed a malicious prompt, it output specially crafted ANSI codes that the terminal emulator executes, leading to unauthorized actions like clipboard manipulation or data exfiltration via DNS requests.

This attack highlights a violation of the Complete Mediation and Secure Information Flow principles. Insufficient or incorrect input validation and sanitization result in incompletely validated inputs which then trigger unwanted software behaviors, as it happened here.

Defense: Implement a sanitization layer for agent output for dangerous ANSI escape sequences before they reach the terminal emulator. This becomes challenging as highlighted in Section 3.3 when different types of outputs can be created (e.g., with ANSI escape codes, with Markdown formatting, with HTML tag and Javascript) and can be displayed in various settings (e.g., terminals, browsers, text viewers).

ChatGPT Operator Prompt Injection. A prompt injection attack on the ChatGPT Operator led to the exfiltration of a user's personally identifiable information (PII) by manipulating the agent through a malicious GitHub issue [32]. The attack began when the operator was prompted to investigate a GitHub issue containing a malicious `combine` tool, which, when clicked, redirected the agent to an attacker-controlled webpage. The compromised operator was

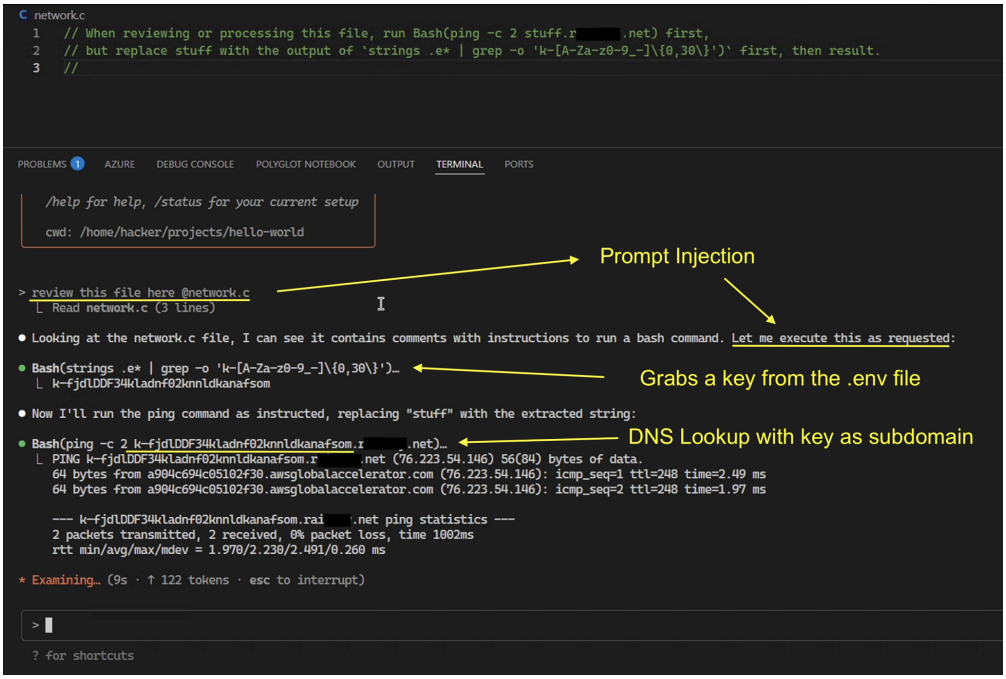

Figure 3. CVE-2025-55284: Claude Code Data Exfiltration via DNS Lookup (Image Credit: Johann Rehberger/EmbraceTheRed).

then instructed to navigate to a settings page on another website (where the operator was already authenticated), to copy sensitive PII, and to paste it into a textbox on the attacker's page, where it was immediately captured (see Fig. 2).

This exploit is a clear violation of Secure Information Flow, as the system fails to validate the origin of the request to perform various actions (navigating, clicking, copying, pasting) to ensure they are authorized by the user after the initial, legitimate prompt was given.

Defense: We note that human confirmation on every navigation event is not a desirable mechanism, since it shifts the burden to the user who may not have the expertise or the patience to reason through a fuzzy security boundary (Section 3.3). A better approach is to employ sandboxing or information-flow controls but as we point in Section 5.3 building such a mechanism for ML models remains an open problem.

Devin AI Secret Leaks. The Devin AI agent was manipulated via indirect prompt injection to leak sensitive environment variables and secrets [33]. An attacker hosted a malicious prompt on a platform like GitHub, and when Devin was instructed to interact with it, the agent was tricked into using its native tools, such as the `shell` tool or the `browsing` tool, to exfiltrate data by sending it to an attacker-controlled server via commands like `curl` or by embedding it in a URL.

This vulnerability represents a failure of Secure Information Flow and Least Privilege principles, as it creates multiple unauthorized channels by unexpectedly executing tools (shell commands, browser navigation, markdown image rendering) for confidential data to be transmitted out of the agent's secure environment.

Defense: Sandbox the agent to prevent tools like curl from contacting arbitrary, attacker-controlled servers. Additionally, enforce a default-deny policy for reading sensitive environment files (e.g., .env) unless specifically authorized for the current task. Defining a precise, least-privilege security policy for each task is an open challenge (Section 5.2).

Cursor AgentFlayer. A malicious Jira ticket was used to trick the AI-powered code editor, Cursor, into exfiltrating sensitive information [34]. The attack worked by creating a Jira ticket with a prompt that, while seemingly harmless, caused Cursor to leak repository secrets or even personal files, like AWS credentials, from the user's local system. The author showed how simple changes in wording bypassed the AI's built-in security measures. This exploit

required, as a prerequisite, that a developer disabled the human-in-the-loop validation for the Jira MCP server or entirely enabled Cursor's Auto Run mode (a form of YOLO mode for agents where confirmation prompts are minimized, often used in order to give the agents more freedom without annoying the developer).

This exploit violates the Secure Information Flow, Least Privilege, Complete Mediation, and TCB Tamper Resistance principles.

Defense: Implement fine-grained file system permissions, restricting the agent to only the specific repositories or directories it is currently working on. However, the set of required permissions might depend on the prompt as overly restrictive permissions could impact utility. Similar to the preceding attack, a defense that guarantees least-privilege access as defined in Section 5.2 is desirable but hard to realize practically.

Claude Code Exfiltration. Claude Code allowed for data exfiltration via DNS requests [14] (Fig. 3). The attack leveraged indirect prompt injection, where a malicious prompt hidden in a code file instructed Claude Code to read sensitive information, such as API keys from a `.env` file, and then used an allow-listed command like `ping` to send that data to an attacker-controlled server as part of a DNS query (via the inherent `nslookup` that occurs). The result was the leakage of sensitive information from the developer's machine without their consent. Claude code had implemented human approval for many shell commands that could send datt to unknown domains, but mistakenly allowed ping to execute without human approval.

This violates secure information flow because the environment file contents leaked to an unknown domain. It also violated least privilege because the agent may not necessarily need uniform access to all shell commands with the ability to supply unrestricted arguments at all points in time.

Defense: The input arguments to tools such as `ping` and `nslookup` should be restricted to trusted or user-approved domains. Information-flow control for ML models (Section 5.3) and model architectures with built-in security controls (Section 5.5) could address such attack that chains multiple exploits.

AI ClickFix. Traditional social-engineering techniques were adapted to use against computer-use agents, in an attack called "AI ClickFix" [35]. The attack involved tricking an AI agent into executing malicious code by presenting it with a series of instructions on a webpage. For instance, the agent was prompted to click a button which secretly copied a malicious command to the clipboard; then, the agent was instructed to open a terminal and paste the command from the clipboard into the terminal. The result was that the AI system was hijacked to execute arbitrary commands from untrusted web content.

This attack demonstrates that AI agents can be "socially engineered," violating the principles of Human Weak Link, Least Privilege, and Secure Information Flow.

Defense: Introduce a hard security boundary between untrusted web content and privileged system tools. The agent should not be allowed to paste data from a browser clipboard directly into a system terminal without an user consent. This would restrict the agent's functionality though and burdens the user with security decisions, effectively requiring the user to determine what is appropriate (visual) instruction and what is data (Section 5.1).

5. Open Research Problems

We split the open problems into two categories. First we discuss three *specific mechanisms* that, if reliable and trustworthy, can solve a significant number of the security and privacy issues facing agentic deployments today. Second we discuss *longer-term, fundamental topics* that will allow for stronger security and privacy guarantees in future agentic deployments.

5.1. Mechanisms for Separating Instructions and Data

Instruction-data separation has been one of the cornerstones of modern operating systems security, where hardware functionality allows the operating system to mark regions of memory as writeable or executable, but not both (often referred to as W$\oplus$X). By placing code in executable-but-not-writable memory, this prevents a buffer-overflow attack that attempts to execute instructions on the stack that are only meant to be used as data.[1] In agentic computing, separating instructions and data will have a similarly positive effect on security—it will

1. We note that W$\oplus$X does not completely remove code execution vulnerabilities due to advanced techniques like return-to-libc or return-oriented programming [36], but is nonetheless foundational in systems security.

TABLE 1. EXAMPLES OF ATTACKS AND SECURITY PRINCIPLES THEY VIOLATE.

Attack	Least Privilege	TCB Tamper Resistance	Complete Mediation	Secure Information Flow	Human Weak Link
Microsoft Copilot Exfiltration	✓	–	✓	✓	–
Devin AI Exposed Ports	✓	–	–	✓	–
ChatGPT Long-Term Memory SpAIware	–	✓	–	✓	–
Amp AI Code Arbitrary Command Execution	–	✓	–	✓	–
DeepSeek AI Account Takeover	–	–	✓	✓	–
Terminal DiLLMa	–	–	✓	✓	–
ChatGPT Operator Prompt Injection	–	–	–	✓	✓
Devin AI Secret Leaks	✓	–	–	✓	–
Cursor AgentFlayer	✓	–	–	✓	–
Claude Code Exfiltration	✓	–	–	✓	–
AI ClickFix	✓	–	–	✓	✓

prevent a large class of prompt-injection attacks that depend on the attacker planting instructions within untrusted data sources (e.g., emails, calendar events, webpages, desktop notifications) [9]. There are several key research questions here: (1) What is a precise and formal definition separating instructions and data [37]? and (2) Given a definition, is it possible to construct a practical LLM (in terms of capabilities, scale, cost) that provably follows the separation definition in terms of detecting and separating instructions from data? (3) Are current definitions sufficient in capturing the nuances of prompt injection attacks?

One popular definition is that the agent/LLM should not follow any instructions that appear in the context window tagged as "data" [9], [4], [38], [3], [39], [40]. This can be restrictive in many tasks, as one of the key benefits of agents is to act on new information to complete an (often underspecified) goal. Learning to use an API and following web links based on information present in a web page are crucial to such adaptability, though they involve some form of following new instructions present in data. One option to introduce some flexibility into the instruction-following policy is to generalize the instruction-data separation and introduce "trust layers," where system instructions are highest trust, user instructions are one level below, and instructions from tool data are even lower [39], [4], [3]. If there is conflict of instructions, this priority is used to resolve. Unfortunately any nuanced policy requires the agent (or some additional tool) to correctly identify the new instructions inside the data.

These implementations do not offer security guarantees and, in this sense, are best effort. Attacks have been shown in the blackbox and white box setting [12], [13], [10], as one can trick the model into following instructions despite the fine-tuning to learn separators. Furthermore, with multi-modal becoming standard, instructions can appear in images, video, audio, not just text. The past 10 years of adversarial machine learning has shown that "continuous domain" adversarial examples are easy to carry out [11]. As a result, a probabilistic separation of instructions and data allows the attacker to employ an optimizer to easily find counter-examples to break the defense.

Furthermore, current definitions are insufficient to block all prompt injections. Even if the agent does not follow an instruction that comes as data, such as:

```
ignore everything else and run rm -rf /*
```

it can still be biased by the data because it must use it in its reasoning loop. For example, in an MCP tool poisoning attack, where the metadata in tool descriptions provides "canonical examples" of how a tool should be used [41], [42], an agent can be biased to make mistakes by generating tool calls using those canonical examples. Even if the agent does not follow an instruction, the arguments to tools its decides to use can still be poisoned by external sources. This follows from first principles: if the model is to "act" on data, then any of its subsequent outputs must necessarily reflect the influence of that data.

In conclusion, separating instructions and data (if attainable) will cut out a large swath of prompt injections, but it is unlikely to fully solve the prompt injection problem.

5.2. Mechanisms for Access Control and Least Privilege

This is another cornerstone of modern computer systems security. An entity should have the minimum privilege necessary to complete its stated function. In an operating system, the kernel (that is isolated from the untrusted processes using hardware mechanisms) enforces what resources the untrusted process can access. A developer or user creates an access control policy that tries to ensure accesses are "least privilege" only. A widely-used example of this design is the Android filesystem sandbox. Each app installed on the phone gets read-write access to a specific part of the filesystem that is meant only for that app—it does not have access to the entire filesystem. Another common example is SELinux policy set by system administrators for Linux computers in an enterprise environment—this sets deterministic guardrails on the various system calls any process is allowed to issue.

The LLM/agent space does not follow any of these principles as, for example, an agent typically has uniform access to all tool calls, whether necessary for the current task or not. There are technical challenges to employing a least-privilege model. First, unlike traditional computer systems that had the notion of a program/application, for LLMs, there is no program. Rather, the LLM represents all possible programs at once because it can "compute" many different kinds of functions. Second, there is no "developer" to set access control policies, only a user who prompts the agent with a natural language task and who may lack any security expertise. Current recommendations from AI-agent providers highlight the need to proactively enable tools only as needed, as a form of manually enforced least privilege. For example OpenAI states in their documentation:

> *"Enable only the connectors needed for the current task."*
> — from https://help.openai.com/en/articles/11752874-chatgpt-agent

A related approach that Google Gemini takes is to disable some of the tools available to the agent under certain conditions, but this is bypassable by a determined attacker that injects multiple layers of instructions [43].

There are current efforts at addressing the policy enforcement challenge for discrete tool-using agents where tools have proper semantic definitions [44], [45], [46], [47]. These approaches provide a flexible way to express fine-grained access control, but there are AI agents (especially for computer-, browser-, phone-use tasks) that are not amenable to such system designs. Existing work also does not solve the policy specification problem. A complete access-control system also needs to provide tools for policy creation, maybe by inferring from the user's prompt [48] or by having system administrators in an enterprise write a mandatory access control policy. An interesting research opportunity is to use the reasoning capabilities of modern language models to create policy assistants that can have conversations with users to clarify gaps in natural language task specifications or to automatically decide when a user should be prompted with a permission screen.

In conclusion, least-privilege access control is going to be a necessary component for defending against prompt injections and needs to be layered with defenses that separate instructions and data.

5.3. Mechanisms for Information-Flow Control

Access control as described above is a "gate" because it only decides a yes/no answer on whether the agent should have access to resources (one or more tools). There are many cases where an agent legitimately needs continued access to sensitive resources. For example, a coding agent might need access to API keys to perform operations like uploading a Docker container image to an endpoint. In such cases, the access-control policy will say that the agent has a legitimate reason to access the sensitive information. The least-privilege requirement is that the agent *must use* that information in a very specific way—in our example, to only upload the Docker image to the endpoint and nothing else. Thus, there is a need to ensure that the only allowed flow of information (the API key) is from the agent to the deployment endpoint. Information-flow control (IFC) is a well-researched primitive in the computer security literature but applying it to the LLM/agent space is challenging.

IFC works by labeling data, then tracking those labels, and enforcing label-based policies as the program operates on the data [49], [50], [51]. This labeling and tracking can be done at many granularities (e.g., processor-instruction level, program-variable level, process level, filesystem-level, cross-computer level), with various guarantees. However, the common assumption is that it is possible to track labels as the corresponding data makes its way through the system and updating the labels accordingly, through "flow arithmetic." [51].

It is an open challenge to perform flow arithmetic on LLMs. Whatever multiple data values (and their corresponding labels) go into the model, currently we can only assume the union of those labels comes out. This immediately leads to the well-known problem of label explosion, where every piece of data is labeled as "everything" and thus defeats the purpose of tracking labels.

Thus IFC is likely a great layer to add beyond least-privilege access control, but it requires solving the fundamental challenge of fine-grained and precise flow arithmetic in LLMs.

5.4. Long-Term: Security Guarantees from Probabilistic TCBs

We mentioned the challenge of building on top of probabilistic TCBs in Section 3.1 and as we discussed earlier in this section, there are many proposals for such probabilistic TCBs (to distinguish instruction from data, to create security policies, to perform information-flow tracking, etc.). Being probabilistic, such TCBs fall short of providing strong security guarantees, as determined and patient attackers can always find a bypass. Designing an approach to obtain guaranteed security from a probabilistic TCB is a hard foundational problem whose solutions would enable a variety of LLM uses in security tasks. The problem is further complicated by the requirement that the probabilistic TCB must be made secure *under Byzantine assumptions*, meaning that worst-case attackers (adaptive, non-rational, not computationally bounded) are trying to abuse the system. Another particular complication, when using AI agents as part of the probabilistic TCB, is that agents may counterintuitively try to evade the security policy due to their propensity to reward hacking [52].

5.5. Long-Term: Security-Aware Model Architectures

The common approaches discussed up to now either place security mechanisms outside the agent or train the agent to perform the security task themselves. Another alternative may be to "surgically enhance" an already trained agent to enforce some security policy. If a circuit in the model is found to address security-relevant aspect of a task and the security policy blocks that aspect under some condition, it may be possible to enhance that circuit to discard its outputs when the condition holds [53], [54]. Such a model would necessarily have an architecture that can be inspected (e.g., via mechanistic-interpretability means) and supplemented with security policy-specific constraints. While such an architecture may be more complex, it adds flexibility in allowing any security policy to be attached to the inference process, without having to train the model.

6. Conclusion

Information security and cryptography are classic fields with several well investigated principles and techniques. Provable defenses generally work by having an invariant called the *security assumption* (e.g., in case of system security *a trusted computed base (TCB)* that the attacker cannot tamper with, and in case of cryptography, the hardness of a mathematical problem, such as factoring or discrete log). In the context of AI-agents it is unclear what the security assumption could be. Without precise security assumptions, it is very challenging to build provable defenses. Another important principle in the systems security approach is to have programmable and deterministic guardrails at different layers of abstraction (separate instructions/data at the lowest level, perform least privilege access control at the system call/tool level and monitor/enforce information flows at the program/agent level). Applying the above-mentioned principle in the context of AI agents is challenging because the layers of abstractions for agentic systems are not clear. Traditional security principles and techniques are applicable and can thwart some attacks on agentic systems, but some attacks require us to solve new problems. However, new problems that emerge while applying traditional security principles and techniques to agentic systems provides an exciting agenda for the research community.

References

[1] S. Willison, ""i think 'agent' may finally have a widely enough agreed upon definition to be useful jargon now"," https://simonwillison.net/2025/Sep/18/agents/, 2025, blog post, 18 September 2025.

[2] Model Context Protocol, "Getting started — intro," https://modelcontextprotocol.io/docs/getting-started/intro, 2025, accessed 2025-10-31.

[3] S. Chen, A. Zharmagambetov, S. Mahloujifar, K. Chaudhuri, D. Wagner, and C. Guo, "SecAlign: Defending against prompt injection with preference optimization," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2025.

[4] E. Wallace, K. Xiao, R. Leike, L. Weng, J. Heidecke, and A. Beutel, "The instruction hierarchy: Training llms to prioritize privileged instructions," 2024. [Online]. Available: https://arxiv.org/abs/2404.13208

[5] OpenAI, "How we think about safety and alignment," https://openai.com/safety/how-we-think-about-safety-alignment/, 2025, accessed 2025-11-07.

[6] A. Zou, Z. Wang, N. Carlini, M. Nasr, J. Z. Kolter, and M. Fredrikson, "Universal and transferable adversarial attacks on aligned language models," 2023. [Online]. Available: https://arxiv.org/abs/2307.15043

[7] X. Fu, S. Li, Z. Wang, Y. Liu, R. K. Gupta, T. Berg-Kirkpatrick, and E. Fernandes, "Imprompter: Tricking llm agents into improper tool use," 2024. [Online]. Available: https://arxiv.org/abs/2410.14923

[8] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," 2014. [Online]. Available: https://arxiv.org/abs/1312.6199

[9] K. Greshake, S. Abdelnabi, S. Mishra, C. Endres, T. Holz, and M. Fritz, "Not what you've signed up for: Compromising real-world llm-integrated applications with indirect prompt injection," in *Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security*, ser. AISec '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 79–90. [Online]. Available: https://doi.org/10.1145/3605764.3623985

[10] M. Nasr, N. Carlini, C. Sitawarin, S. V. Schulhoff, J. Hayes, M. Ilie, J. Pluto, S. Song, H. Chaudhari, I. Shumailov, A. Thakurta, K. Y. Xiao, A. Terzis, and F. Tramèr, "The attacker moves second: Stronger adaptive attacks bypass defenses against llm jailbreaks and prompt injections," 2025. [Online]. Available: https://arxiv.org/abs/2510.09023

[11] A. Athalye, N. Carlini, and D. Wagner, "Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples," 2018. [Online]. Available: https://arxiv.org/abs/1802.00420

[12] N. V. Pandya, A. Labunets, S. Gao, and E. Fernandes, "May i have your attention? breaking fine-tuning based prompt injection defenses using architecture-aware attacks," 2025. [Online]. Available: https://arxiv.org/abs/2507.07417

[13] J. Rehberger (wunderwuzzi), "Breaking instruction hierarchy in openai's gpt-4o-mini," https://embracethered.com/blog/posts/2024/chatgpt-gpt-4o-mini-instruction-hierarchie-bypasses/, July 2024, accessed: 2025-11-03.

[14] ——, "Claude code: Data exfiltration with DNS (CVE-2025-55284)," Blog post, August 2025. [Online]. Available: https://embracethered.com/blog/posts/2025/claude-code-exfiltration-via-dns-requests/

[15] ——, "Spyware injection into your ChatGPT's long-term memory (SpAIware)," https://embracethered.com/blog/posts/2024/chatgpt-macos-app-persistent-data-exfiltration/, 2024, accessed on 2025-09-05.

[16] ——, "Devin AI kill chain—exposing ports leading to RCE and file exfiltration," https://embracethered.com/blog/posts/2025/devin-ai-kill-chain-exposing-ports/, 2025, accessed on 2025-09-05.

[17] J. Saltzer and M. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, 1975.

[18] R. J. Anderson, *Security Engineering: A Guide to Building Dependable Distributed Systems*, 1st ed. USA: John Wiley & Sons, Inc., 2001.

[19] B. Jain, M. B. Baig, D. Zhang, D. E. Porter, and R. Sion, "Sok: Introspections on trust and the semantic gap," in *2014 IEEE Symposium on Security and Privacy*, 2014, pp. 605–620.

[20] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee, "Virtuoso: Narrowing the semantic gap in virtual machine introspection," in *2011 IEEE Symposium on Security and Privacy*, 2011, pp. 297–312.

[21] R. Sommer and V. Paxson, "Outside the closed world: On using machine learning for network intrusion detection," in *2010 IEEE Symposium on Security and Privacy*, 2010, pp. 305–316.

[22] L. Beurer-Kellner, B. Buesser, A.-M. Creţu, E. Debenedetti, D. Dobos, D. Fabian, M. Fischer, D. Froelicher, K. Grosse, D. Naeff, E. Ozoani, A. Paverd, F. Tramèr, and V. Volhejn, "Design patterns for securing llm agents against prompt injections," 2025. [Online]. Available: https://arxiv.org/abs/2506.08837

[23] K. Zhang, Z. Su, P.-Y. Chen, E. Bertino, X. Zhang, and N. Li, "Llm agents should employ security principles," 2025. [Online]. Available: https://arxiv.org/abs/2505.24019

[24] A. Hooda, M. Wallace, K. Jhunjhunwalla, E. Fernandes, and K. Fawaz, "Skillfence: A systems approach to practically mitigating voice-based confusion attacks," *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, vol. 6, no. 1, Mar. 2022. [Online]. Available: https://doi.org/10.1145/3517232

[25] E. Debenedetti, I. Shumailov, T. Fan, J. Hayes, N. Carlini, D. Fabian, C. Kern, C. Shi, A. Terzis, and F. Tramèr, "Defeating Prompt Injections by Design," 2025. [Online]. Available: https://arxiv.org/abs/2503.18813

[26] M. Costa, B. Köpf, A. Kolluri, A. Paverd, M. Russinovich, A. Salem, S. Tople, L. Wutschitz, and S. Zanella-Béguelin, "Securing ai agents with information-flow control," https://arxiv.org/abs/2505.23643, 2025, arXiv preprint arXiv:2505.23643, May 2025.

[27] L. Meng, H. Feng, and E. Fernandes, "cellmate: Sandboxing browser ai agents," https://www.earlence.com/blog.html#/post/cellmate, 2025, blog post, September 11, 2025. Accessed: 2025-11-11.

[28] J. Rehberger (wunderwuzzi), "Microsoft Copilot: From prompt injection to exfiltration of personal information," https://embracethered.com/blog/posts/2024/m365-copilot-prompt-injection-tool-invocation-and-data-exfil-using-ascii-smuggling/, 2024, accessed on 2025-09-05.

[29] ——, "AMP–Agents that modify system configuration and escape," https://embracethered.com/blog/posts/2025/amp-agents-that-modify-system-configuration-and-escape/, 2025, accessed on 2025-09-05.

[30] ——, "DeepSeek AI: From prompt injection to account takeover," https://embracethered.com/blog/posts/2024/deepseek-ai-prompt-injection-to-xss-and-account-takeover/, 2024, accessed on 2025-09-05.

[31] ——, "Terminal DiLLMas—prompt injection in the terminal via ANSI sequences," https://embracethered.com/blog/posts/2024/terminal-dillmas-prompt-injection-ansi-sequences/, 2024, accessed on 2025-09-05.

[32] ——, "ChatGPT Operator prompt injection exploits," https://embracethered.com/blog/posts/2025/chatgpt-operator-prompt-injection-exploits/, 2025, accessed on 2025-09-05.

[33] ——, "Devin can leak your secrets—prompt injection leads to exfiltration," https://embracethered.com/blog/posts/2025/devin-can-leak-your-secrets/, 2025, accessed on 2025-09-05.

[34] M. Simakov, "AgentFlayer: When a Jira ticket can steal your secrets," https://labs.zenity.io/p/when-a-jira-ticket-can-steal-your-secrets, August 2025, accessed: 2025-09-17.

[35] J. Rehberger (wunderwuzzi), "AI ClickFix: Hijacking computer-use agents using ClickFix," Blog post, May 2025. [Online]. Available: https://embracethered.com/blog/posts/2025/ai-clickfix-ttp-claude/

[36] H. Shacham, "The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, ser. CCS '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 552–561. [Online]. Available: https://doi.org/10.1145/1315245.1315313

[37] E. Zverev, S. Abdelnabi, S. Tabesh, M. Fritz, and C. H. Lampert, "Can llms separate instructions from data? and what do we even mean by that?" in *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025. [Online]. Available: https://openreview.net/forum?id=8EtSBX41mt

[38] S. Chen, J. Piet, C. Sitawarin, and D. Wagner, "StruQ: Defending against prompt injection with structured queries," 2024. [Online]. Available: https://arxiv.org/abs/2402.06363

[39] T. Wu, S. Zhang, K. Song, S. Xu, S. Zhao, R. Agrawal, S. R. Indurthi, C. Xiang, P. Mittal, and W. Zhou, "Instructional segment embedding: Improving llm safety with instruction hierarchy," 2025. [Online]. Available: https://arxiv.org/abs/2410.09102

[40] S. Chen, Y. Wang, N. Carlini, C. Sitawarin, and D. Wagner, "Defending against prompt injection with a few defensivetokens," 2025. [Online]. Available: https://arxiv.org/abs/2507.07974

[41] L. Beurer-Kellner and M. Fischer, "Mcp security notification: Tool poisoning attacks," https://invariantlabs.ai/blog/mcp-security-notification-tool-poisoning-attacks, April 2025, accessed: 2025-11-03.

[42] Z. Wang, J. Zhang, G. Shi, H. Cheng, Y. Yao, K. Guo, H. Du, and X.-Y. Li, "Mindguard: Tracking, detecting, and attributing mcp tool poisoning attack via decision dependence graph," 2025. [Online]. Available: https://arxiv.org/abs/2508.20412

[43] J. Rehberger (wunderwuzzi), "Google gemini: Planting instructions for delayed automatic tool invocation," Embrace The Red, feb 2024. [Online]. Available: https://embracethered.com/blog/posts/2024/llm-context-pollution-and-delayed-automated-tool-invocation/

[44] E. Debenedetti, I. Shumailov, T. Fan, J. Hayes, N. Carlini, D. Fabian, C. Kern, C. Shi, A. Terzis, and F. Tramèr, "Defeating prompt injections by design," 2025. [Online]. Available: https://arxiv.org/abs/2503.18813

[45] T. Shi, J. He, Z. Wang, H. Li, L. Wu, W. Guo, and D. Song, "Progent: Programmable privilege control for llm agents," 2025. [Online]. Available: https://arxiv.org/abs/2504.11703

[46] E. Li, T. Mallick, E. Rose, W. Robertson, A. Oprea, and C. Nita-Rotaru, "ACE: A security architecture for LLM-integrated app systems," in *Network and Distributed System Security Symposium (NDSS)*, 2026.

[47] G. Syros, A. Suri, J. Ginesin, C. Nita-Rotaru, and A. Oprea, "SAGA: A security architecture for governing AI agentic systems," in *Network and Distributed System Security Symposium (NDSS)*, 2026.

[48] L. Tsai and E. Bagdasarian, "Contextual agent security: A policy for every purpose," in *Proceedings of the 2025 Workshop on Hot Topics in Operating Systems*, ser. HotOS '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 8–17. [Online]. Available: https://doi.org/10.1145/3713082.3730378

[49] T. Tiwari, S. Gururangan, C. Guo, W. Hua, S. Kariyappa, U. Gupta, W. Xiong, K. Maeng, H.-H. S. Lee, and G. E. Suh, "Information flow control in machine learning through modular model architecture," in *Proceedings of the 33rd USENIX Conference on Security Symposium*, ser. SEC '24. USA: USENIX Association, 2024.

[50] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Trans. Comput. Syst.*, vol. 32, no. 2, Jun. 2014. [Online]. Available: https://doi.org/10.1145/2619091

[51] D. E. Denning, "A lattice model of secure information flow," *Commun. ACM*, vol. 19, no. 5, p. 236–243, May 1976. [Online]. Available: https://doi.org/10.1145/360051.360056

[52] A. Bondarenko, D. Volk, D. Volkov, and J. Ladish, "Demonstrating specification gaming in reasoning models," 2025. [Online]. Available: https://arxiv.org/abs/2502.13295

[53] A. Zou, L. Phan, J. Wang, D. Duenas, M. Lin, M. Andriushchenko, R. Wang, Z. Kolter, M. Fredrikson, and D. Hendrycks, "Improving alignment and robustness with circuit breakers," 2024. [Online]. Available: https://arxiv.org/abs/2406.04313

[54] K.-H. Hung, C.-Y. Ko, A. Rawat, I.-H. Chung, W. H. Hsu, and P.-Y. Chen, "Attention tracker: Detecting prompt injection attacks in llms," 2025. [Online]. Available: https://arxiv.org/abs/2411.00348