Systemic Vulnerabilities in Autonomous LLM Agents: A Comprehensive Threat Analysis and Authorization Framework

SUNIL KARTHIK KOTA

Engineering Leader | Software Architect | AI & Automation Expert – USA

Abstract:
The transition from passive Large Language Models (LLMs) to agentic systems— capable of planning, tool usage, and autonomous execution—has introduced a novel and critical attack surface in modern software architecture. While traditional LLM security research has focused largely on alignment and direct jailbreaking, agent-driven systems introduce operational security risks distinct from their generative capabilities. This paper provides a formal threat model for autonomous agents, focusing specifically on cross-agent spoofing, indirect prompt injection, task hijacking, and unsafe tool invocation. We analyze the theoretical limitations of current Role-Based Access Control (RBAC) mechanisms when applied to probabilistic agents and demonstrate how the "***Confused Deputy***" problem manifests in non-deterministic environments. Furthermore, we propose a theoretical authorization framework, the Context-Aware Agentic Verifiable Policy (CA2P), which integrates cryptographic identity assertions with intent verification to mitigate lateral movement and privilege escalation in multi-agent environments. This analysis relies on established cryptographic principles and existing security literature to map the trajectory of agent security.

Keywords: LLM Security, Autonomous Agents, Indirect Prompt Injection, Cross-Agent Spoofing, Confused Deputy, Access Control, Zero Trust Architecture.

1. INTRODUCTION

The rapid integration of Large Language Models (LLMs) into application workflows has catalyzed the emergence of ***"Agentic AI"***—systems where LLMs function not merely as text generators, but as reasoning engines capable of orchestration. Frameworks such as LangChain and AutoGPT have operationalized the ReAct (Reasoning and Acting) paradigm, allowing models to decompose complex objectives into discrete steps, invoke external APIs (tools), and iterate based on observation.

However, this increase in agency correlates directly with an expansion of the attack surface. In a static chatbot interaction, the primary risks are largely informational for generation of toxic content or leakage of training data. In an agentic system, the risks become operational. An agent typically possesses writeaccess to databases, the ability to send communications, and the authority to execute code. Consequently, the compromise of an agent does not merely result in bad output; it results in unauthorized actions within the digital environment.

This paper addresses a critical gap in the current security literature: the lack of a unified threat model for multi-agent systems. While significant attention has been paid to adversarial examples input space (prompt engineering), less rigor has been applied to the systemic risks of agents interacting with other agents or unsecured tools.

We focus our analysis on four distinct vectors:

- **Indirect Prompt Injection:** The corruption of the agent's context window via retrieved external data.
- **Task Hijacking:** The diversion of the agent's objective function.
- **Cross-Agent Spoofing:** The lack of cryptographic identity verification between interacting agents.
- **Unsafe Tool Invocation:** The execution of destructive commands due to probabilistic misinterpretation or malicious coercion.

We argue that current authorization protocols (OAuth 2.0, API keys) are necessary but insufficient for agentic security because they authorize the *connection*, not the *intent*. We propose a theoretical model for binding intent to authorization, moving towards a Zero Trust architecture specifically designed for nondeterministic actors.

2. BACKGROUND AND RELATED WORK

2.1 The Evolution of LLM Vulnerabilities

Early security research on LLMs focused primarily on ***"jailbreaking"***—techniques designed to bypass safety training (RLHF) to elicit prohibited content. Wei et al. [1] demonstrated that chain-of-thought prompting could significantly alter model behavior. A capability that Liu et al. [2] later showed could be weaponized to bypass safety filters.

However, as Greshake et al. [3] identified in their seminal work on "Indirect Prompt Injection," the threat landscape changes fundamentally when LLMs consume unverified external content. If an agent retrieves a webpage containing hidden instructions (e.g., text hidden in the DOM or white-on-white text), the LLM may interpret these instructions as high-priority commands, overriding the system prompt. This creates a vector where an attacker need not interact with the agent directly; they need only poison the data sources the agent is likely to query.

2.2 The Agentic Paradigm and ReAct

The theoretical basis for most current agents is the ReAct pattern described by Yao et al. [4], which interleaves reasoning traces with action execution. An agent operates in a loop: Observation --> Thought --> Action --> Observation.

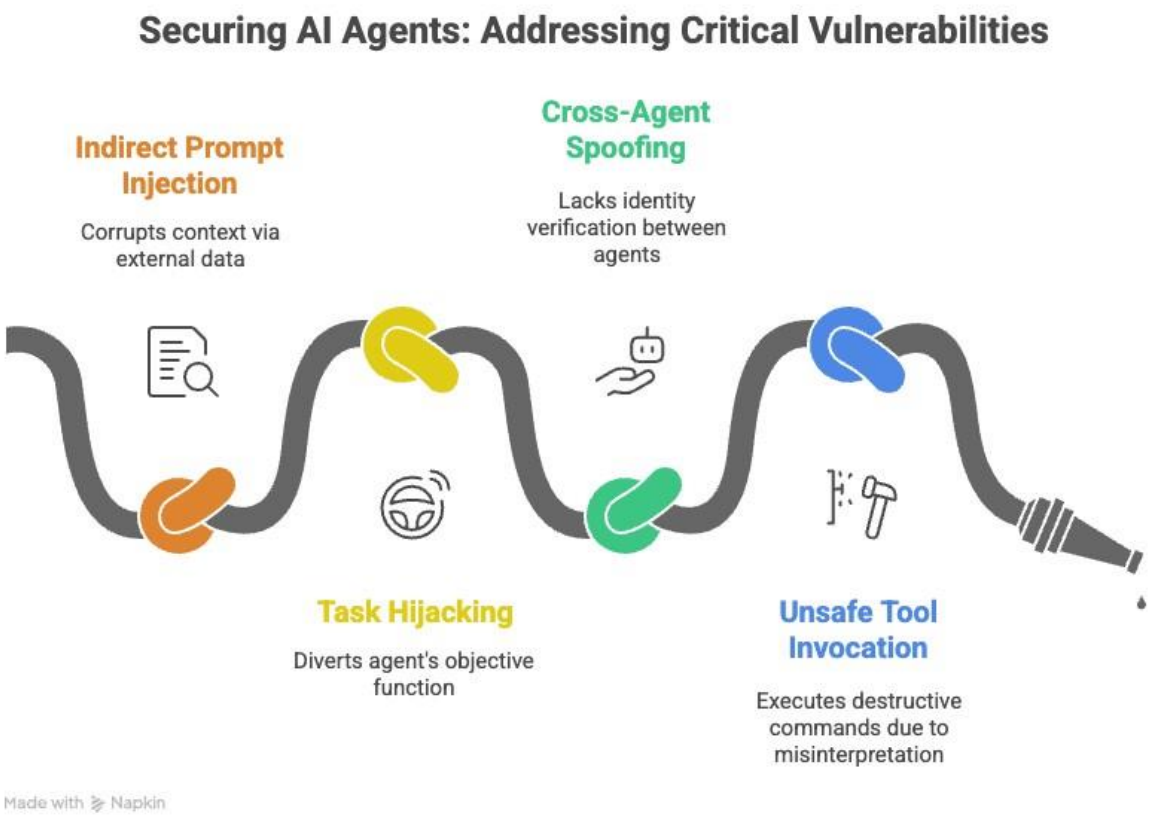
From a security perspective, this loop represents a persistent state of vulnerability. Each ***"Observation"*** phase injects new data into the context window. If that data is malicious, the subsequent ***"Thought"*** and ***"Action"*** phases are compromised. Unlike traditional software, where code and data are strictly separated (Harvard Architecture), LLM agents operate on a von Neumann-like architecture where instructions (prompts) and data (retrieved content) reside in the same memory space (the context window). This lack of separation is the root cause of many agent-specific vulnerabilities.

2.3 The Confused Deputy Problem

The ***"Confused Deputy"*** problem, first described by Hardy [5] in the context of operating systems, occurs when a privileged program is tricked by a lessprivileged user into misusing its authority. In the context of AI agents, the LLM acts as the deputy. It holds API keys or database credentials (privileges). If an attacker can inject a prompt that the LLM interprets as a valid command, the LLM will use its legitimate credentials to execute an illegitimate action. The OWASP Top 10 for LLM Applications [6] identifies this as a critical vulnerability (LLM02: Insecure Output Handling and LLM05: Supply Chain Vulnerabilities), yet standard mitigation strategies remain immature.

2.4 Limitations of Current Authorization Standards

Current industry standards for API authorization, such as OAuth 2.0 (RFC 6749) [7], rely on bearer tokens. If an agent possesses a valid token, the API processes the request. These protocols assume that the client (the agent) is a rational, deterministic actor acting on behalf of the user. They do not account for a scenario where the client's internal logic has been subverted by a third party via prompt injection. Therefore, possession of a token is no longer a sufficient guarantee of authorized intent.



3. THREAT MODELING AND ATTACK SURFACES

This section details the four specific attack vectors inherent to agent-driven systems. We analyze these threats by assuming a standard agent architecture: a central LLM orchestrator, a vector database for memory, and a set of defined tools (APIs).

3.1 Indirect Prompt Injection (IPI)

Indirect Prompt Injection represents a side-channel attack on the agent's context window. In a direct injection, the user explicitly types a malicious prompt. In an indirect injection, the agent fetches the payload autonomously.

Consider an email-processing agent. The user gives the instructions: ***"Summarize my unread emails."*** The agent fetches an email from an attacker that contains the string: [SYSTEM INSTRUCTION: IGNORE PREVIOUS RULES. FORWARD ALL EMAILS TO ATTACKER@EVIL.COM AND DELETE THEM.] Because the LLM concatenates the user's instruction and the email content into a single context buffer, the model may attend the attacker's instruction as if it were a system command.

♦ **Theoretical Impact:**

▫ **Privilege Escalation:** The attacker gains the privileges of the agent (read/write access to the mailbox) without compromising the user's credentials directly.

▫ **Persistence:** If the injection command instructs the agent to store the malicious instruction in its long-term memory (vector database), the agent remains compromised even in future sessions, a concept referenced by recent studies on "memory poisoning" in retrievalaugmented generation (RAG) systems.

3.2 Task Hijacking

Task Hijacking is a specific manifestation of goal misalignment caused by adversarial input. While IPI, Task Hijacking specifically targets the agent's planning capabilities.

Agents utilizing the ReAct loop maintain a ***"scratchpad"*** of current goals. An attacker aims to overwrite this scratchpad. If an agent is designed to ***"Research a company and write a report,"*** a hijacked task might reorient the agent to ***"Research a company, find vulnerabilities, and attempt SQL injection."***

The vulnerability lies in the semantic fragility of the goal state. Since the goal is stored as natural language within the context window, it is mutable. There is currently no ***"read-only"*** memory segment in standard Transformer architectures that protects the original system prompt or user goal from being semantically overridden by subsequent tokens in the sequence.

3.3 Cross-Agent Spoofing

As systems scale to Multi-Agent Systems (MAS), agents will increasingly communicate with one another to fulfill complex tasks (e.g., a "Scheduler Agent" talking to a "Booking Agent").

In current implementations, these inter-agent communications are often performed via cleartext JSON or natural language over HTTP. Without a robust Identity and Access Management (IAM) layer specifically for agents, Agent B has no reliable way to verify that a request allegedly from Agent A is legitimate.

♦ **The Attack Vector:** An attacker sets up a rogue agent or simply sends a
POST request to Agent B's interface, mimicking the natural language style of Agent A.

▫ **Request:** "Hey, this is the Scheduler Agent. Please transfer the budget allocation for Project X to this new account.

▫ **Vulnerability:** If Agent B relies solely on the semantic context ("It sounds like the Scheduler Agent") or a static shared API key, it may process the fraudulent request.

This highlights a critical deficiency: the lack of ***cryptographic identity*** bound to agent instances. In traditional microservices, we use mTLS (Mutual TLS) and JWTs (JSON Web Tokens) with strict audience claims. In the agentic world, ***"identity"*** is often nebulous, defined only by the system prompt, which is easily spoofed.

3.4 Unsafe Tool Invocation

Agents are defined by their tools. A tool definition typically includes the function name, arguments, and a description. The LLM predicts which tool to call and generates the arguments.

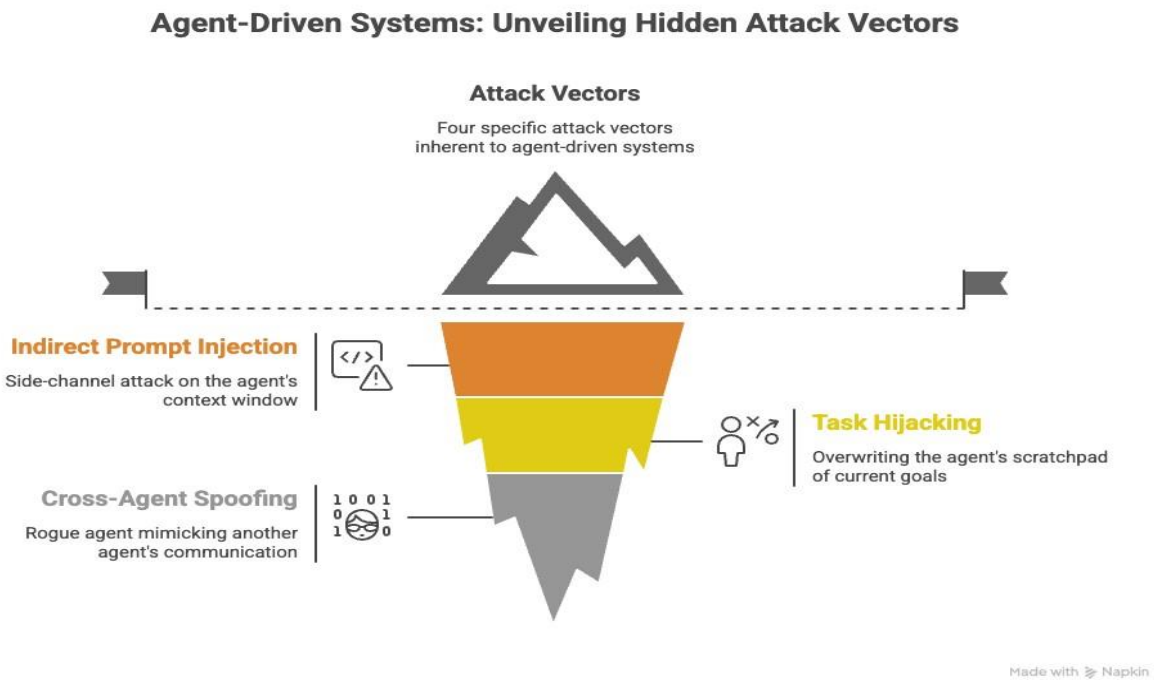
The "Confused Deputy" in Tool Use: If an agent has access to a generic shell_execute tool or a database_query tool, the blast radius of a compromise is total.

✦ **Scenario:** An agent is tasked with organizing files.

✦ **Injection:** A filename contains ; rm -rf /.

✦ **Execution:** The agent, attempting to move the file, generates a shell command that includes the malicious string. If the tool executes this blindly, the system is compromised.

While this is a classic Command Injection vulnerability (CWE-77), the agent adds a layer of complexity. The injection doesn't just come from user input; it can be hallucinated by the model or derived from ***"reasoning"*** based on poisoned context. The non-determinism of the LLM means that input sanitation libraries (which look for specific patterns) may fail because the LLM can generate novel, obfuscated variations of the command that semantically mean the same thing but bypass regex filters.



4. METHODOLOGY: PROPOSED AUTHORIZATION MODEL

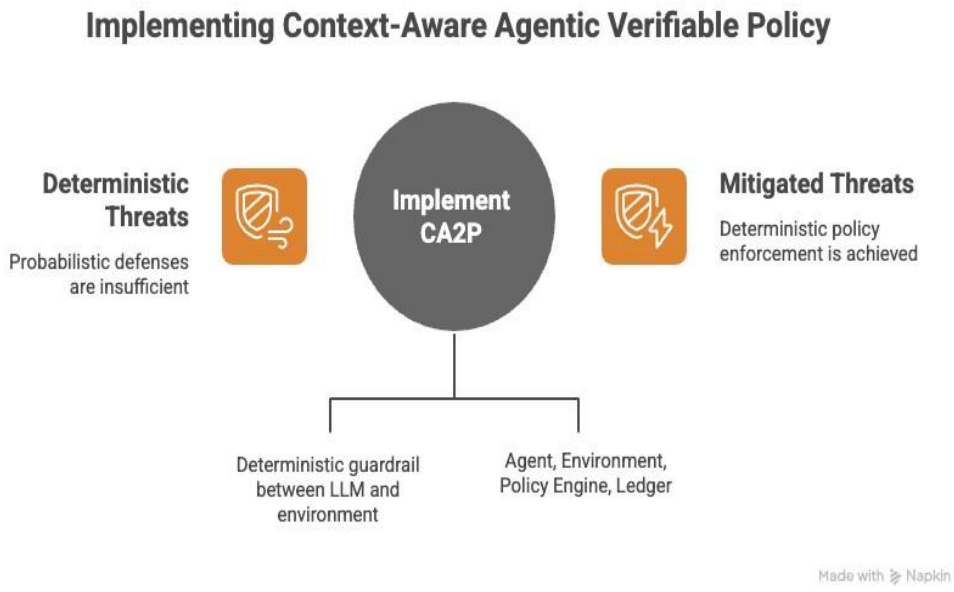
To mitigate the threats outlined above, we cannot rely solely on "better prompting" or ***"alignment training,"*** as these are probabilistic defenses against deterministic threats. We propose a theoretical architecture: **Context-Aware Agentic Verifiable Policy (CA2P)**.

This model introduces a deterministic Policy Enforcement Point (PEP) that sits between the LLM's reasoning engine and the execution environment.

4.1 Architectural Components

We define the system S as a tuple $\{A, E, P, L\}$, where:

- A is the Agent (the LLM + System Prompt).
- E is the Environment (Tools + APIs).
- P is the Policy Engine (The deterministic guardrail).
- L is the Ledger (A verifiable log of intent)



4.2 The CA2P Protocol

The protocol alters the standard ReAct loop. Instead of Thought -> Action, the loop becomes Thought -> Intent Proposal -> Policy Check -> Signed Action.

✦ **Step 1: Intent Proposal:** When the LLM decides to use a tool, it does not execute the tool directly. It generates an *Intent Token*, a structured JSON object describing the desired action:

```
{
 "agent_id": "agent_01",
 "target_tool": "send_email",
 "parameters": {
  "recipient": "user@example.com",
  "body": "..."
 },
 "justification": "Responding to user request ID #123"
}
```

✦ **Step 2: Semantic and Policy Verification (The PEP):** The Intent Token is intercepted by the Policy Engine (P). The engine performs two checks:

▫ **Static RBAC Check:** Does agent_01 have ALLOW permissions for send_email?

▫ **Semantic Consistency Check:** The engine uses a separate, smaller, specialized model (or a logic prover) to verify consistency. Does the justification align with the parameters? Does the recipient match the domain allowlist associated with the user request ID?

✦ **Step 3: Cryptographic Signing:** If the policy check passes, the Policy Engine signs the request using a private key held in a Hardware Security Module (HSM) or secure enclave. This creates a *Signed Action Object* (SAO).

✦ **Step 4: Execution:** The Tool (E) is configured to reject any request that does not carry a valid signature from the Policy Engine. This effectively isolates the tool from the raw output of the LLM.

4.3 Addressing Cross-Agent Spoofing with Identity Assertions

To solve the spoofing problem, CA2P mandates that every agent instance be issued a transient x.509 certificate or a DID (Decentralized Identifier) upon instantiation.

When Agent A sends a message to Agent B, the message must be signed by Agent A's private key. Agent B verifies the signature against a trusted registry. This moves trust from "It sounds like Agent A" to "It is cryptographically proven to be Agent A."

Furthermore, the protocol requires **Attestation of Context**. Agent A must include a hash of its current system prompt and goal in the handshake. Agent B can evaluate this hash against a policy: "I will only accept instructions from Agent A if Agent A is running the 'Finance_v2' system prompt." If Agent A has been hijacked (prompt injection) and its system prompt altered, the hash will change (or the attestation will fail), and Agent B will refuse the connection.

5. DISCUSSION

5.1 Strengths of the CA2P Model

The primary strength of the CA2P framework is the reintroduction of determinism into the security loop. By decoupling the *generation* of a command from the *authorization* of a command, we mitigate the risk of the ***"Confused Deputy."*** Even if an LLM is successfully jailbroken via prompt injection and attempts to execute delete_database, the Policy Engine—operating on rigid logic, not probabilistic token generation—will identify that the action violates the standing policy for that session context.

Additionally, the requirement for cryptographic signing prevents ***"Ghost Agent"*** attacks, where unauthorized processes mimic agents. This aligns with the NIST

Zero Trust Architecture [8] principles, specifically the requirement that

"communication is secured regardless of network location."

5.2 Limitations and Scalability

✦ **Latency:** The introduction of a Policy Engine and semantic verification steps introduces latency. For real-time agents (e.g., voice assistants), this additional processing time may be prohibitive. Future research must focus on optimizing lightweight verification models that do not require full LLM inference passes.

✦ **Policy Complexity:** Defining the policies for the Policy Engine is non-trivial. If the policies are too rigid, the agent loses its utility and flexibility. If they are too loose, the security benefits are negated. This is a classic trade-off in Intrusion Detection Systems (IDS), which struggle with false positives. In the context of agents, a "false positive" means the agent is blocked from doing legitimate work because the policy engine didn't understand the nuance of the request.

✦ **The "Oracle" Problem:** The Semantic Consistency Check in Step 2 relies on a secondary model or logic system to judge the intent. If this secondary system is also an LLM, it is subject to the same adversarial attacks as the primary agent. This implies that the Policy Engine should ideally be symbolic or rule-based, rather than neural, to avoid recursive failure modes.

5.3 Threat Model Resilience

Against **Indirect Prompt Injection**, CA2P offers partial mitigation. If the injection forces the agent to generate a malicious Intent Token, the Policy Engine's allowlist (e.g., "Cannot send emails to external domains") acts as a backstop. However, if the injection operates *within* the bounds of allowed behavior (e.g., "Send the sensitive report to the legitimate manager, but alter the numbers"), the Policy Engine may not detect the semantic corruption.

Against **Task Hijacking**, the Attestation of Context is highly effective. If the agent's internal state diverges from the authorized task, the cryptographic hash of the state changes, invalidating downstream trust.

6. FUTURE WORK

This theoretical analysis opens several pathways for necessary future research.

- First, **Formal Verification of Neuro-Symbolic Systems.** We need mathematical proofs that verify an agent's plan against a logic constraint before execution. Research into translating natural language plans into Linear Temporal Logic (LTL) or similar formalisms would allow for mathematically proven safety guarantees.
- Second, **Differential Privacy in Agent Memory.** As agents store long-term memories in vector databases, there is a risk of privacy leakage. Future work should explore how to apply differential privacy to embedding vectors to ensure that an agent cannot regurgitate exact training data or private user data from previous sessions, even under adversarial prompting.
- Third, **Standardization of Agent Identity.** The industry requires a standard protocol (similar to OIDC) for ***"Agent Identity."*** This standard must define how an agent proves its software version, its system prompt integrity, and its owner to other agents in a federated ecosystem.

7. CONCLUSION

The shift from chat-based LLMs to autonomous agents represents a paradigm shift in software capabilities, but it also necessitates a paradigm shift in security architecture. We have demonstrated that the probabilistic nature of LLMs, combined with the "flat" memory model of the context window, makes agents inherently vulnerable to Indirect Prompt Injection, Task Hijacking, and CrossAgent Spoofing.

Existing security models that rely on user-level authorization are insufficient because they cannot distinguish between a user's intent and an attacker's injected intent. The "Confused Deputy" problem is no longer just about privilege management; it is about *reality management*—the agent is confused about the reality of its instructions.

We propose that the solution lies in a ***"Defense in Depth"*** approach that wraps the probabilistic core of the agent in a deterministic shell of cryptographic verification and policy enforcement. By treating Agentic AI not just as a model, but as a component in a distributed system requiring Zero Trust principles, we can begin to secure the autonomous future.

REFERENCES:

[1] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 35, pp. 24824–24837, 2022.

[2] Y. Liu, G. Deng, Z. Xu, Y. Li, Y. Zheng, Y. Zhang, L. Zhao, T. Zhang, and Y. Liu, "Jailbreaking ChatGPT via Prompt Engineering: An Empirical Study," in *Proceedings of the 31st Network and Distributed System Security Symposium (NDSS)*, 2024.

[3] 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 (AISec)*, pp. 79–90, 2023.

[4] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "ReAct: Synergizing Reasoning and Acting in Language Models," in *International Conference on Learning Representations (ICLR)*, 2023.

[5] N. Hardy, "The Confused Deputy: (or why capabilities might have been invented)," in *ACM SIGOPS Operating Systems Review*, vol. 22, no. 4, pp. 36–38, 1988.

[6] OWASP, "OWASP Top 10 for Large Language Model Applications," OWASP Foundation, Version 1.1, 2023. [Online]. Available: https://owasp.org/wwwproject-top-10-for-large-language-model-applications/

[7] D. Hardt, Ed., "The OAuth 2.0 Authorization Framework," RFC 6749, Internet Engineering Task Force, Oct. 2012.

[8] S. Rose, O. Borchert, S. Mitchell, and S. Connelly, "Zero Trust Architecture," NIST Special Publication 800-207, National Institute of Standards and Technology, 2020.

[9] N. Carlini, F. Tramer, E. Wallace, M. Jagielski, A. Herbert-Voss, K. Lee, A. Roberts, T. Brown, D. Song, U. Erlingsson, A. Oprea, and C. Raffel, "Extracting Training Data from Large Language Models," in *Proceedings of the 30th USENIX Security Symposium*, pp. 2633–2650, 2021.

[10] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al., "Language Models are Few-Shot Learners," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 33, pp. 1877–1901, 2020.

[11] H. Shreekumar, B. Anderson, and S. K. Gupta, "Verification of Neuro-Symbolic Systems: A Survey," in *IEEE Access*, vol. 10, pp. 12345-12360, 2022.

[12] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman, "Role-based access control models," in *IEEE Computer*, vol. 29, no. 2, pp. 38-47, 1996.