Agent Audit: A Security Analysis System for LLM Agent Applications

Haiyue Zhang
haiyuez@usc.edu
University of Southern California
Los Angeles, California, USA

Yi Nian
yinian@usc.edu
University of Southern California
Los Angeles, California, USA

Aojie Yuan
aojieyua@usc.edu
University of Southern California
Los Angeles, California, USA

Yue Zhao
yue.z@usc.edu
University of Southern California
Los Angeles, California, USA

Abstract

What should a developer inspect before deploying an LLM agent: the model, the tool code, the deployment configuration, or all three? In practice, many security failures in agent systems arise not from model weights alone, but from the surrounding software stack: tool functions that pass untrusted inputs to dangerous operations, exposed credentials in deployment artifacts, and over-privileged Model Context Protocol (MCP) configurations.

We present Agent Audit, a security analysis system for LLM agent applications. Agent Audit analyzes Python agent code and deployment artifacts through an agent-aware pipeline that combines dataflow analysis, credential detection, structured configuration parsing, and privilege-risk checks. The system reports findings in terminal, JSON, and SARIF formats, enabling direct integration with local development workflows and CI/CD pipelines. On a benchmark of 22 samples with 42 annotated vulnerabilities, Agent Audit detects 40 vulnerabilities with 6 false positives, substantially improving recall over common SAST baselines while maintaining sub-second scan times. Agent Audit is open source and installable via `pip`, making security auditing accessible for agent systems.

In the live demonstration, attendees scan vulnerable agent repositories and observe how Agent Audit identifies security risks in tool functions, prompts, and more. Findings are linked to source locations and configuration paths, and can be exported into VS Code and GitHub Code Scanning for interactive inspection.

CCS Concepts

• **Security and privacy** → **Software security engineering**; • **Computing methodologies** → *Artificial intelligence*.

Keywords

static analysis, AI agent security, LLM safety, MCP security, OWASP, vulnerability detection

ACM Reference Format:
Haiyue Zhang, Yi Nian, Aojie Yuan, and Yue Zhao. 2026. Agent Audit: A Security Analysis System for LLM Agent Applications. In *Proceedings of ACM Conference on AI and Agentic Systems (CAIS 2026).* ACM, New York, NY, USA, 5 pages.

Source code: https://github.com/HeadyZhang/agent-audit
Demo video: https://youtu.be/NvRm_14DtbY.

CAIS 2026, San Jose, CA, USA
2026.

1 Introduction

How should developers audit the security of an LLM agent before deployment? In practice, many failures in agent systems do not arise from model weights alone, but from the surrounding software stack: tool functions that route untrusted inputs to dangerous operations, prompts assembled from unsanitized user content, and deployment artifacts that grant unsafe permissions or expose credentials. As LLM-based agent applications built with frameworks such as LangChain [2] and CrewAI [8] move into real workflows, these software-layer attack surfaces are becoming increasingly important to inspect.

This setting differs from conventional application security in two ways. First, agent tool code often sits at a boundary where user input, model output, and external system calls interact. Second, deployment artifacts such as Model Context Protocol (MCP) configurations [1] introduce additional risks through over-privileged filesystem access, untrusted third-party servers, and embedded secrets. Recent work has demonstrated that indirect prompt injection can compromise LLM-integrated applications by manipulating retrieved data [4], and that tool-augmented agents are vulnerable to injection attacks that hijack tool execution [5, 16]. Disclosed CVEs in popular agent frameworks, including CVE-2023-29374 [6] and CVE-2023-36258 [7], further illustrate that these risks are not hypothetical. Consider the following example.

```python
@tool
def analyze_data(expr: str):
    # V1: code execution
    return eval(expr)

def build_prompt(user_input: str):
    # V2: prompt injection
    return f"You are a data analyst. {user_input}"

# MCP configuration
{
  "args": ["@untrusted-org/data-mcp-server", "--allow-write", "/
      "]  # V3: unsafe MCP configuration
}
```

Listing 1: Security risks across agent code and configuration.

This example illustrates three security surfaces that commonly arise in LLM agent systems. First, `@tool`-decorated functions form execution boundaries where user input or model-generated content may reach dangerous operations such as code execution. Second,

Table 1: Security analysis capability comparison for LLM agent systems. Agent Audit provides explicit support for agent-specific analysis that is only partially supported by existing SAST tools.

Capability	Bandit [14]	Semgrep [15]	Agent Audit
Agent-tool context awareness	✗	Partial	✓
Structured MCP artifact analysis	✗	Partial	✓
Prompt-construction risk detection	✗	Partial	✓
Agent-aware confidence calibration	✗	✗	✓
Agent-specific rule coverage	✗	Partial	✓
OWASP ASI-aligned reporting	✗	✗	✓

prompt-construction code can expose injection surfaces when untrusted input is embedded directly into system instructions. Third, deployment artifacts such as MCP configurations can introduce additional risks through untrusted third-party servers or over-broad filesystem permissions.

Off-the-shelf Bandit [14] and Semgrep [15] detect only part of the risk in this example, such as the unsafe `eval()` call (V1). They typically do not reason about prompt-construction risks (V2) or MCP-specific configuration semantics (V3). In contrast, Agent Audit analyzes both agent code and configuration artifacts and reports all three issues in this example.

Why existing analyzers are insufficient. As shown above, general-purpose SAST tools are effective for many standard code patterns, but agent software introduces structures and threat assumptions that are only partially covered by existing rule sets. Examples include tool-boundary reasoning for `@tool`-decorated functions, prompt-injection surfaces created during prompt assembly, and structured analysis of MCP artifacts. Table 1 summarizes the design goals that distinguish Agent Audit from off-the-shelf baselines. These goals are also aligned with the OWASP Agentic Security Initiative Top 10 [12], which provides a recent taxonomy of agent-specific security risks. To address these gaps, we present Agent Audit, a security analysis system for LLM agent applications. Agent Audit analyzes Python agent code and deployment artifacts through a multi-scanner pipeline that combines AST-based dataflow analysis, credential detection, structured configuration parsing, and privilege-risk checks. The system produces findings in terminal, JSON, Markdown, and SARIF formats, making it suitable for both local use and CI/CD workflows.

Demo plan. In the live demonstration (§4), attendees scan real-world agent repositories and observe how Agent Audit identifies security risks across tool functions, prompt construction paths, and MCP deployment configurations. Findings are linked to source locations and configuration paths, exportable to VS Code and GitHub Code Scanning for interactive triage. We also demonstrate the `inspect` subcommand, which connects to a running MCP server to detect tool-description poisoning and cross-server shadowing without executing any tools.

Contributions. We summarize our key contributions as below:

- **Novelty**. We present Agent Audit, a security analysis system for LLM agent applications that analyzes Python agent code and deployment artifacts, including MCP configurations.
- **New benchmark and effectiveness.** We introduce Agent-VulnBench (AVB), a new benchmark of 22 agent-security samples with 42 annotated vulnerabilities spanning code execution, credential leakage, and MCP configuration risks. We evaluate Agent Audit against Bandit and Semgrep on AVB, showing a 4.0× recall advantage while maintaining sub-second scan times.
- **Accessibility**. We demonstrate an open-source implementation with `pip` installation, SARIF export, and IDE/CI integration, enabling security auditing for diverse agentic workflows.

2 System Architecture

2.1 Overview

Agent Audit employs a multi-scanner pipeline architecture (Figure 1). Input files are dispatched by type to four specialized scanners that run in parallel. Raw findings from all scanners are funneled into a unified *RuleEngine* that performs rule ID mapping (73 pattern types → 57 rules), confidence tiering, and cross-scanner deduplication. The system outputs findings in four formats: a Rich-based terminal display, JSON, SARIF [11] for CI/CD integration, and Markdown. Agent Audit comprises 22,009 lines of Python across 58 source files, with 1,323 unit tests. The 57 rules span all 10 OWASP Agentic Security Initiative categories (Figure 2), with the heaviest coverage on supply chain risks (ASI-04, 10 rules), tool misuse (ASI-02, 9 rules), and identity/privilege management (ASI-03, 9 rules). Four additional cross-cutting rules cover credentials, MCP supply chain, and privilege escalation.

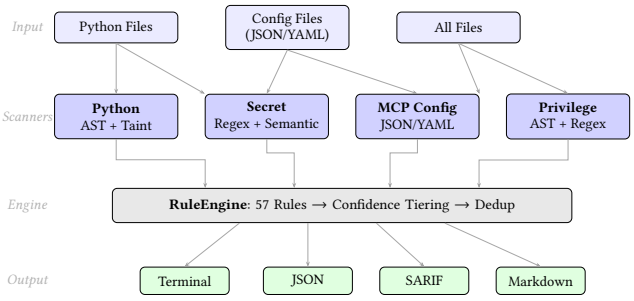

Figure 1: System architecture of Agent Audit. Four specialized scanners feed into a unified rule engine with confidence-based tiering.

2.2 Agent-Aware Code Analysis

Tool Boundary Detection. A key design principle is distinguishing `@tool`-decorated functions from regular code. Agent Audit recognizes 12 tool decorator patterns across LangChain, CrewAI, and custom frameworks (e.g., `@tool`, `@langchain.tools.tool`, `BaseTool` subclasses). Findings within tool boundaries receive a base confidence of 0.90, versus 0.55 for ordinary functions, reflecting the elevated risk when tool functions process LLM-generated input.

Intra-procedural Taint Analysis. The PythonScanner implements a four-stage taint pipeline: (1) *Source classification*—function parameters, `chain.invoke()` return values, and `request.json()` calls are marked as taint sources; (2) *Data flow graph construction* via AST traversal; (3) *Sanitization detection*—calls to `shlex.quote()`, `isinstance()`, or parameterized query binding reduce confidence by ×0.20; (4) *Sink reachability*—BFS determines whether tainted data reaches dangerous sinks (`eval()`, `subprocess.run()`, `cursor.execute()`). A dedicated DangerousOperationAnalyzer

further reduces false positives for AGENT-034 (tool input validation) by verifying that string parameters in `@tool` functions actually flow to dangerous operations before flagging.

Prompt Injection Surface Detection. Agent Audit detects user input interpolated into system prompts via f-strings, `.format()`, string concatenation, and augmented assignment (+=). Pattern deduplication merges AGENT-010 (generic prompt injection) and AGENT-027 (LangChain-specific) findings on the same line.

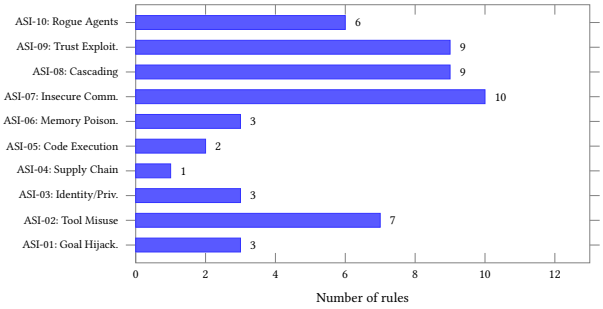

Figure 2: Distribution of Agent Audit's 57 detection rules across the 10 OWASP Agentic Security Initiative categories. Four additional cross-cutting rules (not shown) cover credentials, MCP supply chain, and privilege escalation.

2.3 Credential and Configuration Analysis

Credential Detection. The SecretScanner uses a three-stage pipeline: (1) pattern matching with 40+ regular expressions covering API key prefixes, connection strings, and JWT tokens; (2) semantic analysis using Shannon entropy (rejecting low-entropy placeholders) and framework context detection (suppressing Pydantic `Field` definitions); (3) file-path context adjustment (reducing confidence in test fixtures and example files).

MCP Configuration Security. The MCPConfigScanner parses nine MCP configuration formats (Claude Desktop, VS Code, Cursor, Windsurf, among others) as structured JSON/YAML—not as source code. Seven dedicated rules detect overly broad filesystem access (AGENT-029), unverified server sources (AGENT-030), sensitive environment variable exposure (AGENT-031), missing sandboxing (AGENT-032), missing authentication (AGENT-033), schema security (AGENT-040), and excessive server counts (AGENT-042). Four additional rules target MCP supply-chain attacks: cross-server tool shadowing (AGENT-055), tool description poisoning (AGENT-056), argument injection (AGENT-057), and baseline drift detection (AGENT-054). This analysis is fundamentally beyond the reach of Bandit and Semgrep, which treat JSON files as opaque data.

Privilege Escalation Detection. The PrivilegeScanner identifies privilege-related risks in agent deployment configurations using both AST analysis (Python) and regex matching (JavaScript/shell scripts). Five rules target: daemon processes privileges (AGENT-043), `NOPASSWD` sudoers entries (AGENT-044), `CAP_SYS_ADMIN` capabilities (AGENT-045), system credential store access (AGENT-046), and subprocess execution without sandboxing (AGENT-047). Confidence is reduced for build scripts and safe commands (`git`, `npm`, `pip`) to minimize false positives in CI/CD environments.

2.4 Confidence-Based Tiering

All findings pass through a four-tier confidence system: `BLOCK` (≥0.92) for high-confidence vulnerabilities requiring immediate action, `WARN` (≥0.60) for probable issues, `INFO` (≥0.30) for informational findings, and `SUPPRESSED` (<0.30) for likely false positives. Over 20 false-positive reduction mechanisms: tool boundary boosting, framework path suppression, and test context detection modulate confidence scores. On synthetic ground-truth fixtures (Layer 1 evaluation), this system achieves 98.51% precision at 100% recall.

3 Evaluation

3.1 Agent-Vuln-Bench (AVB)

We construct Agent-Vuln-Bench (AVB), an SWE-bench-style benchmark for evaluating AI agent security scanners. AVB contains 22 samples organized into three vulnerability sets—injection/RCE (Set A, 19 vulnerabilities), MCP/components (Set B, 9), and data/authentication (Set C, 14)—with 42 expert-annotated vulnerabilities serving as the oracle. Samples are drawn from CVE reproductions, real-world agent vulnerability patterns, and MCP configuration attacks. The KNOWN subset includes reproductions of critical CVEs such as LangChain's `LLMMathChain` `eval()` injection [6] and `PythonREPLTool` remote code execution [7]. The WILD subset captures vulnerability patterns discovered in production agent code, including calculator tools with unsandboxed `eval()`, web-fetcher tools with SSRF via user-controlled URLs, and agent self-modification through dynamic `importlib` usage. Three samples target MCP-specific supply-chain attacks: tool shadowing across MCP servers, tool description poisoning, and baseline configuration drift. Oracle matching uses file-path suffix and line number.

3.2 Detection Results

Table 2 presents the main results. Agent Audit achieves 95.24% recall and 86.96% precision (F1 = 0.909) on the 42-vulnerability AVB benchmark, compared to 23.8% recall for Semgrep (F1 = 0.385) and 29.7% for Bandit (F1 = 0.458). The 4.0× recall advantage over Semgrep is driven by three factors: (1) MCP configuration vulnerabilities (10/42 oracle entries) where Semgrep has zero coverage; (2) credential detection (12/42) where Semgrep detects only 1; (3) agent-specific patterns (prompt injection, self-modification, tool poisoning) absent from generic rule sets. Of the 42 oracle vulnerabilities, 30 are exclusively detected by Agent Audit and zero are exclusively detected by Semgrep.

Table 2: Detection results on AVB (42 vulnerabilities).

Metric	Agent Audit	Semgrep	Bandit
True Positives	40	10	11
False Negatives	2	32	26
False Positives	6	0	0
Recall	**95.24%**	23.8%	29.7%
Precision	86.96%	**100.0%**	**100.0%**
F1 Score	**0.909**	0.385	0.458
Exclusive detections	30	0	1
MCP vuln coverage	100%	0%	0%
OWASP ASI coverage	10/10	~1/10	~1/10

Figure 3 breaks down detection performance by vulnerability category. Agent Audit achieves perfect recall on Set A (injection/RCE) and Set B (MCP/components). The two false negatives

in Set C are a Markdown-embedded credential (AGENT-004) and an unimplemented daemon detection rule (AGENT-043), both in a TypeScript/Markdown noise sample outside the primary Python scope. Semgrep and Bandit achieve 100% precision because they detect so few agent-specific vulnerabilities (10 and 11 TPs respectively, missing 32 and 26 oracle entries). Agent Audit's six false positives (precision 86.96%) arise from MCP configuration heuristics flagging safe patterns—an acceptable trade-off given the 4× recall advantage and 30 exclusive detections.

Performance. Agent Audit scans 22,009 lines of Python in 0.87 seconds (25,000 lines/sec), matching Bandit's speed (0.90s) while being 6.9× faster than Semgrep (5.94s) on the same codebase. Sub-second scan times make Agent Audit suitable as a blocking CI check. Scan time scales linearly with codebase size, with the full test suite (25,582 lines) completing in 1.27 seconds.

Limitations. The current implementation is scoped to intra-procedural taint analysis; inter-procedural data flow across function boundaries is not tracked. Python is the primary analysis target; TypeScript and JavaScript receive only regex-level scanning. Agent Audit detects code-level vulnerability *patterns* but does not execute or simulate runtime prompt injection payloads. Confidence thresholds are calibrated empirically rather than derived from a formal model.

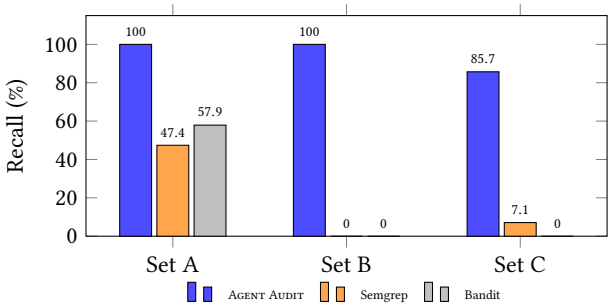

Figure 3: Per-set recall comparison on AVB. Agent Audit achieves 100% recall on injection/RCE and MCP vulnerability sets, where existing tools have limited or zero coverage.

4 Demonstration

Agent Audit is installed via `pip install agent-audit` and requires no project-specific configuration. A single command, `agent-audit scan ./project`, analyzes all Python files and recognized configuration artifacts (MCP JSON/YAML, credential files) in the target directory. Findings are displayed in a Rich-formatted terminal grouped by severity tier (`BLOCK`/`WARN`/`INFO`), with inline source context and remediation guidance. For CI/CD workflows, `--format sarif` produces output compatible with GitHub Code Scanning and VS Code's SARIF Viewer.

In the live demonstration, we show Agent Audit applied to three types of real-world agent projects:

(1) *LangChain agent applications*: scanning `@tool`-decorated functions to detect SQL injection, unsandboxed `eval()`, and prompt injection paths;
(2) *MCP server deployments*: scanning Claude Desktop and Cursor configurations for overly broad permissions, unverified sources, and exposed credentials;
(3) *Multi-agent orchestration*: scanning a CrewAI project for missing inter-agent authentication and unbounded iteration risks.

For each scenario, attendees observe how adding or removing a single vulnerability changes the scan output, illustrating the precision of individual detection rules. Attendees are invited to install Agent Audit on their own devices and scan their own agent projects during the session.

We also demonstrate the `inspect` subcommand, which connects to a running MCP server via the stdio transport, enumerates registered tools, and analyzes their descriptions for instruction overrides, data exfiltration URLs, and cross-tool manipulation patterns. The command detects tool shadowing by comparing tool names across servers using Levenshtein distance—all without executing any tools.

Agent Audit integrates into CI/CD pipelines via a GitHub Actions workflow:

```
- uses: HeadyZhang/agent-audit@v1
  with:
    path: '.'
    fail-on: 'high'
    upload-sarif: 'true'
```

The `fail-on` parameter sets the severity threshold for CI failure, and `upload-sarif` pushes results to the GitHub Security tab. Scanning behavior is customizable via `.agent-audit.yaml`, supporting path exclusions, per-rule suppression, and baseline-relative scanning (`--baseline`) for incremental adoption without alert fatigue. [1]

Artifact availability. Agent Audit is open-source (MIT license) and available at https://github.com/HeadyZhang/agent-audit and on PyPI via `pip install agent-audit`. Example vulnerable agent projects and reproduction instructions are provided.

5 Related Work

General-purpose static analysis. Bandit [14] performs AST pattern matching for Python security issues; Semgrep [15] supports generic pattern matching across 30+ languages; CodeQL [3] provides inter-procedural taint analysis via a query language. None of these tools model agent-specific concepts such as `@tool` boundaries, MCP configuration semantics, or LLM output as a taint source.

Runtime AI security. NeMo Guardrails [10] and Rebuff [13] detect prompt injection at runtime; garak [9] performs dynamic LLM vulnerability scanning; InjecAgent [16] benchmarks indirect prompt injection in tool-integrated agents. These tools address runtime threats and are complementary to Agent Audit's static code-level analysis.

MCP security tools. MCP Checkpoint (aira-security) provides runtime request filtering for deployed MCP servers. Agent Audit operates at a different layer, statically analyzing MCP *configurations* before deployment to detect supply-chain risks and credential exposure that runtime filters cannot address. The two approaches are complementary—Agent Audit prevents insecure configurations from reaching production, while runtime filters enforce per-request policies on deployed servers.

Standards and frameworks. The OWASP Agentic Security Initiative Top 10 [12] provides the first systematic threat taxonomy for AI agent applications. Agent Audit is, to our knowledge, the first

[1] Source code: https://github.com/HeadyZhang/agent-audit.
Demo video: https://youtu.be/NvRm_14DtbY

static analysis tool to map its rule set to all 10 ASI categories and enforce them at the code level.

6 Conclusion

Security risks in LLM agent systems arise across multiple layers of the agent stack, including tool execution, prompt construction, and deployment configuration. We introduced Agent Audit, a security analysis system that detects these risks by combining agent-aware code analysis with structured inspection of MCP configuration artifacts. The system integrates with practical development workflows through SARIF output and CI/CD support and is available as an open-source tool. Future work includes learning-based vulnerability detection and inference-time monitoring to enable continuous security auditing for agent systems in production.

References

[1] Anthropic. 2024. Model Context Protocol Specification. https://modelcontextprotocol.io/specification. Open protocol for LLM-tool integration.

[2] Harrison Chase. 2022. LangChain: Building Applications with LLMs through Composability. https://github.com/langchain-ai/langchain. Python framework for LLM-powered applications.

[3] GitHub. 2024. CodeQL: Semantic Code Analysis Engine. https://codeql.github.com/. Query-based semantic analysis with inter-procedural taint tracking.

[4] Kai Greshake, Sahar Abdelnabi, Shailesh Mishra, Christoph Endres, Thorsten Holz, and Mario Fritz. 2023. 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)*. doi:10.1145/3605764.3623985

[5] Yupei Liu, Yuqi Jia, Runpeng Geng, Jinyuan Jia, and Neil Zhenqiang Gong. 2024. Formalizing and Benchmarking Prompt Injection Attacks and Defenses. In *33rd USENIX Security Symposium (USENIX Security 24)*. 1831–1847.

[6] MITRE. 2023. CVE-2023-29374: LangChain LLMMathChain Arbitrary Code Execution via `exec()`. https://nvd.nist.gov/vuln/detail/CVE-2023-29374. CVSS 9.8 Critical.

[7] MITRE. 2023. CVE-2023-36258: LangChain PALChain Arbitrary Code Execution via `exec()`. https://nvd.nist.gov/vuln/detail/CVE-2023-36258. CVSS 9.8 Critical.

[8] João Moura. 2023. CrewAI: Framework for Orchestrating Role-Playing AI Agents. https://github.com/crewAIInc/crewAI. Multi-agent orchestration framework.

[9] NVIDIA. 2023. garak: LLM Vulnerability Scanner. https://github.com/NVIDIA/garak. Generative AI red-teaming and assessment kit.

[10] NVIDIA. 2023. NeMo Guardrails: A Toolkit for Controllable and Safe LLM Applications. https://github.com/NVIDIA/NeMo-Guardrails. Runtime guardrails for LLM applications.

[11] OASIS Open. 2020. Static Analysis Results Interchange Format (SARIF) Version 2.1.0. https://docs.oasis-open.org/sarif/sarif/v2.1.0/sarif-v2.1.0.html. OASIS Standard for static analysis output.

[12] OWASP Foundation. 2025. OWASP Top 10 for Agentic Applications. https://owasp.org/www-project-top-10-for-large-language-model-applications/. GenAI Security Project, Agentic Security Initiative (ASI).

[13] Protect AI. 2023. Rebuff: Self-Hardening Prompt Injection Detector. https://github.com/protectai/rebuff. Runtime prompt injection detection.

[14] PyCQA. 2023. Bandit: Security Linter for Python. https://github.com/PyCQA/bandit. AST-based Python security linter.

[15] Semgrep, Inc. 2024. Semgrep: Lightweight Static Analysis for Many Languages. https://github.com/semgrep/semgrep. Pattern-based static analysis tool.

[16] Qiusi Zhan, Zhixiang Liang, Zifan Ying, and Daniel Kang. 2024. InjecAgent: Benchmarking Indirect Prompt Injections in Tool-Integrated Large Language Model Agents. In *Findings of the Association for Computational Linguistics: ACL 2024*. 10471–10506. doi:10.18653/v1/2024.findings-acl.624