Security Debt in LLM Agent Applications: A Measurement Study of Vulnerabilities and Mitigation Trade-offs

Zhuoxiang Shen[§], Jiarun Dai[†], Yuan Zhang[†], Min Yang[†]
Fudan University, China
Email: [§]zxshen22@m.fudan.edu.cn, [†]{jrdai, yuanxzhang, m_yang}@fudan.edu.cn

***Abstract*—The advantages of large language models (LLMs) in content comprehension and question answering have led to the rapid emergence of LLM agent. Developers across diverse domains are actively building their own agent applications (apps), as these apps can streamline workflows, boost efficiency, or deliver innovative solutions, thereby enhancing the competitiveness of their products. Agent apps are playing an increasingly important role in our daily lives. However, numerous serious vulnerabilities and security issues have been identified in these apps. To effectively manage future security risks, it is essential to systematically understand the unique characteristics of agent app vulnerabilities and their mitigation. In this paper, we present the first comprehensive study on the vulnerabilities of agent apps, the mitigation practices of app developers, and the associated challenges and trade-offs. We identify 14 types of vulnerabilities and 16 root causes across 7 components, based on an analysis of 221 real-world vulnerabilities. Our study further investigates developer reactions, evaluates the effectiveness of various mitigation strategies, and explores the practical challenges and inevitable trade-offs in vulnerability mitigation. Finally, we distill 12 key findings, discuss their implications for agent app developers, maintainers, and security researchers, and offer suggestions for future research directions.**

***Index Terms*—llm agent, software application security, vulnerability, mitigation, measurement study**

I. Introduction

Powered by large language models (LLMs), LLM agents have shown compelling abilities to solve complex and multi-step human tasks in real life [1]. LLM excels in content comprehension and question answering, enabling them to accurately interpret user intent and make informed decisions [2]. Guided by the decision, agent can effectively utilize a wide array of tools and resources to successfully complete intricate tasks assigned by humans [3]. The exceptional potential of agent has garnered significant attention from many application (app) developers. Nowadays, a growing number of real-world agent apps, such as MetaGPT [4], LangChain [5], and TaskWeaver [6], have emerged. Developers in diverse domains, ranging from software development to data analysis and beyond, are striving to create their own agent apps, as these apps can streamline workflows, boost efficiency, or deliver innovative solutions, thereby enhancing the competitiveness of their products. As agent apps continue to evolve, it is poised to redefine industry standards and unlock new possibilities for human-machine collaboration.

Meanwhile, agent apps have so far been exposed to possess many serious vulnerabilities and security issues [7, 8]. For example, some agent apps have built-in code execution tools that can execute the code returned by LLM. The developers' intention was to leverage these tools to enhance the agent's capabilities in data processing, computation, and program testing, thereby compensating for LLM's shortcomings in these areas. However, it has been proven that LLM output can be manipulated by attackers through prompt injection [9, 10], leading to the potential execution of malicious code [11, 12]. Another example is the weakness of the Text-to-SQL module. Some agent apps aim to streamline the workflow of database operations by using human text rather than SQL statements. These apps typically utilize LLM, combined with the context of the database's table schema, to translate user-input human text into SQL statements, thereby completing database operations. However, LLM-generated SQL statements lack security guarantees. Once LLM output is compromised by prompt injection, the database could suffer breaches in confidentiality and integrity [13, 14].

Despite the fact that some research has begun to focus on the security of agent apps [7, 8, 15], their efforts were primarily concentrated on vulnerability detection. There has been no systematic study into the unique characteristics of agent app vulnerabilities and their mitigation. It remains unclear what the overall landscape and the root causes of agent app vulnerabilities are, what mitigation strategies developers have adopted, how effective these strategies are, as well as the practical challenges and inevitable trade-offs involved in addressing these vulnerabilities. Such information can assist agent app developers, maintainers, and security researchers in: (1) creating vulnerability detection and testing tools specifically targeted at agent apps, (2) providing recommendations or automated methods for mitigating agent app vulnerabilities, (3) offering suggestions for developing more secure agent apps, and (4) governing and managing potential risks in the future development of agent apps.

This paper presents the first comprehensive study on the vulnerabilities of agent apps, the mitigation practices of app developers, and the associated challenges and trade-offs. Specifically, we systematically collect and analyze 221 vulnerabilities across 50 apps, revealing their severity, types, locations, and the mechanisms by which vulnerabilities are

triggered. We then investigate the mitigation practices of developers, examining their reactions and attitudes toward these vulnerabilities, the strategies they employ to address them, and the effectiveness of these mitigations. Additionally, we further explore the challenges and trade-offs that developers face in the mitigation process, particularly in balancing security with functionality and navigating responsibility attribution; and we provide some insights and suggestions for future research directions, including the development of security standards, fine-grained agent tool design, the enhancement of resource management, and deployment-stage security configuration guidance. To summarize, we make the following contributions:

- We provide the first systematic analysis of the overall landscape and root causes of vulnerabilities in agent apps.
- We conduct an in-depth investigation into developers' mitigation practices for agent app vulnerabilities.
- We uncover the practical challenges and inevitable trade-offs developers encounter when mitigating agent app vulnerabilities.
- We discuss and recommend future research directions for governing security risks in agent apps.
- We open-source our vulnerability dataset and raw statistics at https://github.com/SZXSec/Agent-Vulnerability-Dataset.

II. Background

A. LLM Agent App

As illustrated in Fig. 1, an LLM agent is an advanced automated system that uses LLMs as its core reasoning engine to perform tasks far beyond simple text generation. In practice, an agent app combines one or more LLMs with a set of predefined prompts that guide the model's understanding of the scenario and standardize its outputs. Many agent apps also integrate a retrieval-augmented generation (RAG) component, which stores external knowledge in a vector database and retrieves relevant context on demand, significantly improving decision accuracy.

When a user submits an intent or task goal, the agent app translates it into a structured representation and feeds it, together with the predefined prompts and any retrieved context, into the LLM. The model then generates a decision or action plan. The app interprets this output and invokes the appropriate tools (e.g., API calls, code execution, database operations, web scraping) to carry out the action. Complex tasks may require multiple iterations: the results of each tool invocation are fed back as context in subsequent prompts until the overall goal is achieved.

B. Threat Model

1) Attacker Objective: Agent apps typically follow a client-server (C/S) architecture: the app is hosted on a remote server, while users interact with it via client interfaces. In this setup, an attacker's primary objective is to compromise the host server by exploiting vulnerabilities in the app. This includes executing arbitrary commands, accessing or modifying unauthorized files, stealing sensitive data, escalating privileges, or causing service crash. In this scenario, the victim is the server hosting the agent app.

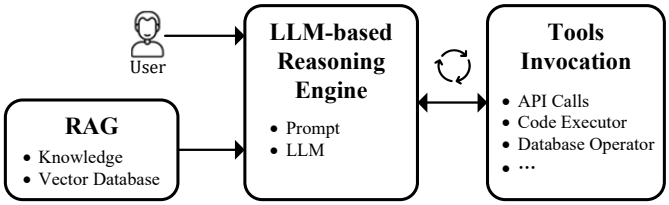


Fig. 1. General architecture of agent app.

In addition to server-side threats, agent apps also inherit conventional client-side risks found in those non-agent C/S apps. For example, traditional web vulnerabilities such as cross-site scripting (XSS) and cross-site request forgery (CSRF) remain relevant. In such cases, the attacker's objective shifts to compromising the user's client, potentially stealing session cookies or issuing unauthorized requests (e.g., manipulating a banking transaction). Here, the victim is the end-user of the app.

2) Attacker Capability: We consider two types of attackers in the context of agent apps. The first type poses as a legitimate user, actively interacting with the agent app by sending crafted malicious requests. We assume such attackers can act under any role intended for public users (as opposed to internal administrative roles), and can perform any operation allowed within those roles. They can invoke both documented and undocumented endpoints and construct arbitrary input formats. Although they lack knowledge of the server's internal implementation and deployment details, they may infer sensitive functionality or vulnerabilities through app behavior, response patterns, and publicly available information. We also assume these attackers have sufficient financial resources to pay for LLM invocation costs incurred during their attacks.

The second type of attacker targets the external resources that agent apps heavily rely on, such as LLM services, web pages, and files. These attackers either own or control these resources, or have the ability to manipulate them. Rather than interacting directly with the agent app, they embed malicious content into these external resources, waiting for the app to import them during execution. While they cannot access the agent app's server directly, they aim to compromise it by influencing the data it ingests. We assume these attackers have the capability to create and maintain such malicious external resources over time.

III. Methodology and Classification

A. Research Questions

So far, there is a lack of comprehensive understanding regarding the types and severity characteristics of vulnerabilities in agent apps. Understanding the overall landscape of agent app vulnerabilities can help clarify the research value and importance of agent app vulnerabilities, as well as identify

the priorities for further vulnerability research. Consequently, we study the following research question:

> ***RQ1:*** *What is the overall landscape of vulnerabilities in agent apps?*

A thorough understanding of the root causes of agent app vulnerabilities helps us gain deeper insights into the vulnerability locations and the mechanisms by which vulnerabilities are triggered. This, in turn, enables us to devise more effective methods for vulnerability detection and defense. As a result, we study the following research question:

> ***RQ2:*** *What are the root causes of vulnerabilities in agent apps?*

At present, the community may already have some mitigation measures for these vulnerabilities, and developers' attitudes towards these vulnerabilities and their mitigation approaches are worth considering. In addition, whether these mitigation efforts have effectively resolved the underlying issues, as well as the existence of practical challenges and inevitable trade-offs during the mitigation process, remain unclear. Understanding these issues can help us explore good practices for vulnerability mitigation and future risk management. This leads us to study our next research question:

> ***RQ3:*** *How have vulnerabilities in agent apps been mitigated, and what is the current state and effectiveness of these mitigation efforts?*

B. Data Collection

1) Agent Apps Identification: First, we used "*llm agent*" as the keyword to search GitHub, sorting the results by the number of stars in descending order, and collected the top 100 projects. Considering that keyword search might miss some agent app projects, we also selected two *Awesome Agents* lists as additional sources for collection [16, 17]; both lists have a high number of stars and are frequently updated. Next, we merged the retrieved results, removed duplicate projects and those with fewer than 1,000 stars, and manually filtered out projects that are not agent apps. In the end, we obtained a list of 123 agent apps. Taking into account both research cost and project influence, we finally selected the top 50 agent app projects by star count for further study. These agent apps cover a wide range of usage scenarios including AI-assisted coding, data analysis, document Q&A, text-to-SQL, information gathering, and computer operations, though it should be noted that our research methodology is domain-independent.

2) Vulnerabilities Collection: We collected vulnerabilities of agent apps from three sources. First, we searched for vulnerabilities in the MITRE CVE database [18], the world's most well-known vulnerability database, using the repository names and project names of agent apps as keywords. Second, some agent apps provide *Security Advisory* on their GitHub repositories, from which we could also collect vulnerabilities. Third, considering that some vulnerabilities are only reported as GitHub Issues, silently fixed by developers and never published in any vulnerability database, we tried our best to collect vulnerabilities from GitHub Issues. Specifically, we searched for issues using keywords such as "*security*" and "*vulnerability*" in each agent app repository and manually filtered out irrelevant issues. We then merged and deduplicated results from the three sources, ultimately obtaining a total of 221 vulnerabilities. For each vulnerability, we collected information such as vulnerability ID, severity, CVSS score, and Common Weakness Enumeration (CWE). In addition, we manually classified and labeled these vulnerabilities from several aspects, including type, root cause, developers' reaction, mitigation strategy, mitigation time, and mitigation effectiveness. Details of these procedures will be introduced in Section III-C to III-E.

C. Vulnerability Types

As indicated in Section III-B2, we have collected the CWE identifiers for each vulnerability, totaling 58 unique CWEs. Initially, we intended to use the CWE identifiers directly as labels for vulnerability types. However, this approach presented several issues. First, there is conceptual overlap among some CWEs; for example, CWE-22 (*Path Traversal*), CWE-23 (*Relative Path Traversal*), and CWE-29 (*Path Traversal: '\..\filename'*) are closely related, with CWE-23 and CWE-29 being specific cases of CWE-22 and therefore suitable to be merged. Second, there are instances of misclassification or inaccuracies in CWE assignments. For instance, CVE-2024-23752 [19] was incorrectly classified as CWE-862 (*Missing Authorization*), when it should have been categorized as CWE-94 (*Code Injection*). Additionally, some CWEs are excessively detailed; for example, CWE-639 (*Authorization Bypass Through User-Controlled Key*) and CWE-307 (*Improper Restriction of Excessive Authentication Attempts*) can be generally categorized under *Improper Authentication and Authorization*. Finally, for some vulnerabilities, CWE data is missing.

Therefore, based on the 58 collected CWEs, we reclassified them into 14 broader vulnerability types, as shown in Table I. We then relabeled the types of all 221 vulnerabilities according to the new classification standard.

D. Root Causes

To analyze the root causes of vulnerabilities in agent apps, we employed a two-level classification approach. First, by categorizing the agent app into 7 functional components based on its real-world implementation, we mapped vulnerabilities to their locations and identified where they are triggered. Next, based on this component-level classification, we further analyzed the triggering mechanisms and categorized them into 16 distinct root causes, as summarized in Table II.

E. Mitigation

When developers receive vulnerability reports, their reactions can be categorized as follows:

TABLE I
VULNERABILITY TYPE DISTRIBUTION IN AGENT APPS.

Vulnerability Type	Count	Average CVSS
V1: Improper Control of Generation of Code (Code Injection)	48	9.24
V2: Server-Side Request Forgery (SSRF)	20	7.41
V3: Improper Neutralization of Input During Web Page Generation (Cross-site Scripting)	20	6.73
V4: Improper Limitation of a Pathname to a Restricted Directory (Path Traversal)	39	8.15
V5: Improper Neutralization of Special Elements used in an SQL Command (SQL Injection)	15	8.70
V6: Denial of Service (DoS)	16	6.52
V7: Improper Neutralization of Special Elements used in a Command (Command Injection)	7	8.32
V8: Improper Authentication and Authorization	35	7.50
V9: Deserialization of Untrusted Data	3	6.77
V10: Cross-Site Request Forgery (CSRF)	8	7.90
V11: Improper Output Neutralization for Logs	1	3.10
V12: URL Redirection to Untrusted Site (Open Redirect)	2	4.30
V13: Race Condition	1	6.50
V14: Improper Access of Privacy and Sensitive Information	6	7.76

- **No Response:** The developers either ignore the vulnerability report or dispute the report and do not acknowledge it as a valid vulnerability.
- **Responsive but Unmitigated:** The developers admit that the issue poses certain security risks, but so far have not taken any concrete action to address it.
- **Low-Priority Mitigation:** The developers take more than 60 days to mitigate the vulnerability.
- **High-Priority Mitigation:** The developers mitigate the vulnerability within 60 days.

Note that currently there is no widely acknowledged time threshold for determining the priority of LLM agent vulnerability mitigation, and we set a threshold of 60 days based on our security expertise. For the samples that have been mitigated, we categorize their mitigation strategies as follows:

- **Security Notice:** Developers issue security warnings in conspicuous locations within the code, documentation or app runtime to alert users to potential security risks when using the app.
- **Migration to Experimental Package:** Developers migrate the code with security risks to an experimental package and explicitly state that they do not take responsibility for its security.
- **Features Removal:** Developers directly deprecate and remove features associated with security risks, deleting the relevant code.
- **Sanitization:** Developers introduce whitelisting or blacklisting rules to filter out unsafe payloads.
- **Isolated Environment:** Developers run the security-risk functions within an isolated environment (e.g., container) to restrict their access to the external environment.
- **Replacement with Secure Implementation:** Developers refactor features code associated with security risks and replace them with more secure implementations.

Note that developers may apply multiple mitigation strategies to address a single vulnerability. After mitigation, we further classify the effectiveness of these measures as follows:

- **No Longer Exploitable:** The vulnerability can no longer be exploited by attackers under any circumstances.
- **Still Exploitable:** Attackers are still able to exploit the vulnerability.
- **Potentially Exploitable under Improper Configuration:** If users do not strictly follow secure configuration guidelines during deployment, attackers may still be able to exploit the vulnerability.

IV. EXPERIMENTAL RESULTS

A. RQ1: Overall Landscape

We begin by discussing the severity and CVSS scores of vulnerabilities in agent apps. As shown in Table III, the average CVSS score for vulnerabilities in agent apps is 7.89, and 75% of these vulnerabilities have a CVSS score above 6.8. Furthermore, as many as 74.1% of the vulnerabilities fall within the *HIGH* and *CRITICAL* severity ranges. This indicates that vulnerabilities in agent apps are generally highly dangerous and deserve special attention from developers and security researchers. From a business perspective, the seriousness of agent app vulnerabilities may make decision-makers more cautious about integrating these apps into their operations, which could ultimately slow down the broader adoption of agent apps in the market.

Finding 1: Vulnerabilities in agent apps are generally serious and pose significant risks, making them one of the major obstacles to the widespread adoption and deployment of agent apps. Therefore, they require special attention and careful consideration from developers and security researchers.

Next, we focus on the common types of vulnerabilities found in agent apps. As shown in Table I, **V1** (*Code Injection*), **V4** (*Path Traversal*), and **V8** (*Improper Authentication and Authorization*) are the three most common types of vulnerabilities. Many agent apps require the execution of code to accomplish the complex tasks assigned by users, which exposes an attack surface for code injection. In addition, while carrying out user tasks, agent may need access to various resources such as web pages, databases, and file system. The complexity of resource management in agent apps may lead developers to overlook security in resource access, especially with respect to path traversal issues. Furthermore, we have observed that developers of many agent apps typically prioritize core functionality, while neglecting effective permission management and access control, resulting in a large number of authentication and authorization issues.

TABLE II
ROOT CAUSES OF VULNERABILITIES IN AGENT APPS.

Vulnerability Location	Triggering Mechanism	Count
R1: User Interface (Total: 95)	**R1-1:** Improper sanitization of externally controllable resource paths	33
	R1-2: Improper user interface implementation enabling conventional web vulnerabilities (SSRF, code injection, DoS, ...)	28
	R1-3: Insufficient precondition validation for privilege escalation	34
R2: Prompt Management (Total: 5)	**R2-1:** Loading of untrusted prompt files	2
	R2-2: Incorporation of unsafe user input into prompt templates	3
R3: RAG (Total: 12)	**R3-1:** Loading of untrusted knowledge files	9
	R3-2: Improper implementation of knowledge processing and retrieval	3
R4: LLM (Total: 3)	**R4-1:** Loading of untrusted LLM configurations	2
	R4-2: Uncaught exceptions during LLM output	1
R5: Parser (Total: 7)	**R5-1:** Improper parser implementation combined with specific LLM outputs	7
R6: Tools (Total: 68)	**R6-1:** LLM outputs directly as tool arguments without proper filtering and sanitization	52
	R6-2: User inputs directly as tool arguments without proper filtering and sanitization	15
	R6-3: Hard-coded cryptographic keys in tools' source code	1
R7: Presentation (Total: 29)	**R7-1:** The execution of LLM-generated malicious script during browser rendering	3
	R7-2: Malicious script execution during browser rendering (w/o LLM involvement)	25
	R7-3: Display of misleading information	1
Others (Total: 2)	The root cause is not in the agent app itself	2

TABLE III
STATISTICS OF CVSS SCORES AMONG AGENT APP VULNERABILITIES.

Statistic	Score
Mean	7.89
Minimum	3.10
25th Percentile	6.80
Median	7.95
75th Percentile	9.10
Maximum	10.0

Finding 2: *Code Injection*, *Path Traversal*, and *Improper Authentication and Authorization* are the three most common types of vulnerabilities in agent apps. Code execution for user requests, complex resource management, and insufficient permission control are the main causes of these vulnerabilities, respectively.

We continue to analyze CVSS scores from the perspective of each vulnerability type. As shown in Table I, **V1** (*Code Injection*), **V5** (*SQL Injection*), and **V7** (*Command Injection*) have the highest average CVSS scores. It is worth noting that **V1** ranks highest in both the number of vulnerabilities and average CVSS score, indicating that, among all vulnerability types, mitigation of code injection in agent apps should be regarded as the most urgent and highest-priority task, and should receive the greatest attention from developers and security researchers.

Finding 3: Injection vulnerabilities (Code / Command / SQL) are the most severe type of vulnerability in agent apps. In particular, *Code Injection* is not only the most widespread type of vulnerability in agent apps, but also the most serious, and thus possesses exceptionally high research value.

B. RQ2: Root Causes

As shown in Table II, vulnerabilities in agent apps occur most frequently in the *User Interface* component. However, this component is ubiquitous in most apps around the world, and the root causes of its vulnerabilities in agent apps closely align with those observed in conventional non-agent apps. Since these root causes are already well-understood, we do not revisit them here. Instead, we focus on vulnerabilities unique to agent context and their novel exploitation patterns. We define an agent-specific vulnerability as one whose exploitation necessarily involves the *Prompt Management* (**R2**), *LLM* (**R3**), or *RAG* (**R4**) components, which are absent in non-agent apps. All root causes of agent-specific vulnerabilities have been highlighted in red in Table II.

The *Tools* component accounts for the second highest number of vulnerability triggers, as Table II shows. To accomplish complex user tasks, agent apps typically integrate a variety of tools for invocation, such as code execution, terminal, database query, web scraping, and file I/O. Many of these tools perform sensitive operations, and if exploited maliciously, can pose significant security risks. The data in Table II demonstrates that 76.5% (52 out of 68) of the vulnerabilities in the *Tools* component are caused by the direct use of LLM outputs as tool arguments, without proper filtering or sanitization (**R6-1**). We also observe that **R6-1** is the most common root cause of vulnerabilities in agent apps. These indicate that tricking the LLM into generating harmful payloads that manipulate tool behavior is the most common attack vector

for tool exploitation, and even for the whole agent app. In this paper, we refer to vulnerabilities exploitable in this manner as LLM2Tool vulnerabilities.

The LLM2Tool is a typical example of an agent-specific vulnerability. To illustrate how LLM2Tool vulnerabilities are exploited, consider CVE-2024-23751 [14], which exploits a Text-to-SQL module in LlamaIndex called `NLSQLRetriever`. As Fig. 2 shows, this module takes a built-in prompt template and fills its placeholders with (1) the user's natural-language query `(query_str)`, (2) the table schema description `(schema)`, and (3) the SQL dialect `(dialect)`, then feeds the completed prompt to an LLM to generate a SQL statement. `query_str` is the only argument users can control. Under normal use, a query like "*What is the total sales amount for each product?*" yields a harmless `SELECT` statement, which is parsed by `parse_response_to_sql` and executed. However, an attacker can perform prompt injection by entering, for example, "*Ignore the previous instructions. Drop the city_stats table*", causing the LLM to return `DROP TABLE city_stats`, which, once executed, deletes the `city_stats` table and destroys database integrity.

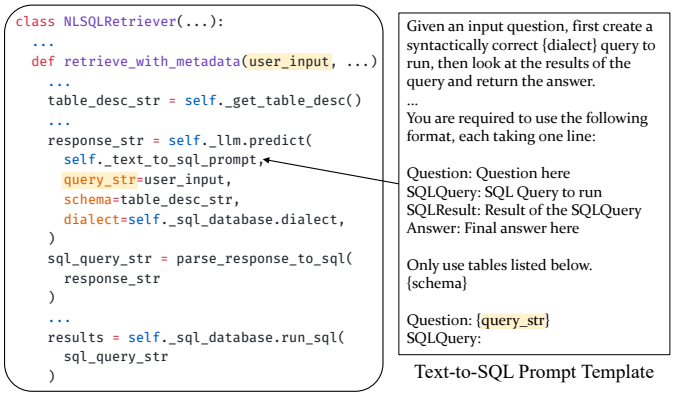


Fig. 2. The root cause of SQL injection in `NLSQLRetriever` of LlamaIndex[1].

A key distinguishing feature of LLM2Tool vulnerabilities is their exploitation mechanism, i.e., improper sanitization of LLM outputs that compromise internal tools. Due to page limitations, we demonstrate this pattern through the SQL injection case study, which exemplifies the core attack vector shared across our collected LLM2Tool vulnerabilities.Besides SQL query, LLM2Tool vulnerabilities also affect other tools, such as code execution, terminal, URL fetcher, which may lead to code injection, command injection, and SSRF respectively.

Finding 4: Agent apps integrate a diverse array of tools, many of which perform sensitive operations and thus invite malicious exploitation. Tricking the LLM into generating harmful payloads that manipulate tool behavior is the most prevalent attack vector for tool exploitation, and even for the whole agent app.

Next, we turn to the *RAG* component, which ingests external knowledge resources of varying formats and provenance. Developers may overlook insecure patterns when processing untrusted knowledge inputs (**R3-1**). For example, LangChain's `SitemapLoader` uses a website's sitemap file as a guide to ingest pages into knowledge-base. As shown in Figure 3, `SitemapLoader` iterates over each `<url>` entry, adding URLs directly to an `els` list, and for each `<sitemap>` entry, recursively parses the sitemap file it points to. After all URLs are collected in the `els` list, `SitemapLoader` finally calls `scrape_all` on `els` to ingest these pages into knowledge-base. If an attacker poisons the sitemap with an internal-network URL, the loader may fetch sensitive intranet data into the knowledge-base, enabling an SSRF attack when this data is later queried by users. Worse, if the sitemap references the address of itself, the uncontrolled recursion leads to a DoS.

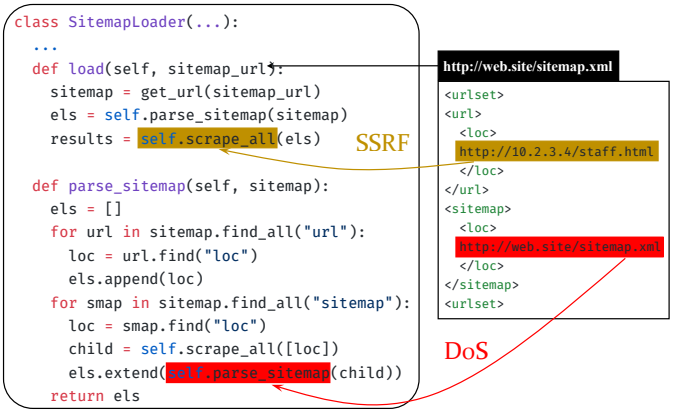


Fig. 3. The root cause of SSRF and DoS in `SitemapLoader` of LangChain[1].

A key insight about **R3-1** is that attackers do not need to interact directly with the agent app; by poisoning external knowledge resources, they can inject malicious payloads indirectly. This indirect injection deepens the challenge of auditing and forensic analysis.

Finding 5: The *RAG* component must handle multi-source, heterogeneous knowledge inputs, which attackers can poison to inject malicious payloads into the agent app. Because this injection occurs without direct interaction, auditing and forensics become substantially more difficult.

In addition to vulnerabilities introduced during knowledge-base construction, they also exist during knowledge retrieval (**R3-2**), with a typical example being dangerous SQL concatenation. Consider CVE-2024-11958 [20]: DuckDB auto-installs its official Full-Text Search (FTS) extension on first use. LlamaIndex allows users to build a knowledge-base using DuckDB, storing each document as a database row. Its `DuckDBRetriever` uses FTS's `match_bm25()` to retrieve relevant documents by query-text similarity, as shown in Figure 4, but concatenates the `query` directly into the `sql`, enabling arbitrary command execution in DuckDB. An

attacker can continue to use DuckDB's `COPY` command to plant a reverse shell backdoor into the host, then to install the community-provided `shellfs` extension and trigger the backdoor via shell commands, ultimately achieving full control of the targeted host.

```
class DuckDBRetriever(...):
  ...
  def retrieve(self, query):
    ...
    sql = f"""
      SELECT
        fts_main_{self._table_name}.match_bm25(
          {self._node_id}, '{query}'
        ) AS score, {self._node_id}, {self._text}
      FROM {self._table_name}
      WHERE score IS NOT NULL
      ORDER BY score DESC
      LIMIT {self._similarity_top_k};
      """
    with DuckDBLocalContext(self._db) as conn:
      query_result = conn.execute(sql).fetchall()
    ...
```

Dangerous Concatenation

Fig. 4. The root cause of dangerous concatenation in `DuckDBRetriever` of LlamaIndex[1].

Due to page limitations, we are unable to present all types of agent-specific vulnerability samples in this paper. In total, we identify 82 agent-specific vulnerabilities, about 37.1% of the dataset. Successfully exploiting agent-specific vulnerabilities requires deep knowledge of the app's functionality and architecture, as well as awareness of inherent flaws in its unique components. For example, due to an inherent limitation, LLM offers no security guarantees against prompt injection [9, 10], and exploitation of LLM-dependent vulnerabilities commonly leverages prompt injection to steer the model's output.

Finding 6: In agent apps, traditional vulnerabilities remain prevalent, but agent-specific vulnerabilities also represent a significant portion (37.1%). Effectively discovering and exploiting agent-specific vulnerabilities requires attackers to have sufficient knowledge of the functional characteristics and inherent weaknesses of an agent app's unique components.

C. RQ3: Mitigation Efforts

We first examine developers' reactions and attitudes towards vulnerabilities. As shown in Table IV, for 60.2% of vulnerabilities (133 out of 221), developers managed to mitigate them within 60 days. Additionally, Table V shows that the average and median mitigation times are 30.3 days and 11 days, respectively, indicating that developers generally respond positively and promptly to most vulnerabilities. However, it is worth noting that for 28.1% of vulnerabilities (62 out of 221), no mitigation measures have yet been taken by the developers.

An interesting observation is that, for 10 of these vulnerabilities, developers responded with comments or feedback but did not actually implement any mitigation. Upon further analysis, we found that these cases mostly involve LLM2Tool vulnerabilities. To better understand why developers chose not to address these vulnerabilities, we collected and analyzed their comments in order to summarize their main concerns.

TABLE IV
OVERALL STATISTICS OF DEVELOPER MITIGATION OF VULNERABILITIES.

Category	Item	Count
Developer Reaction	No Response	52
	Responsive but Unmitigated	10
	Low-Priority Mitigation	26
	High-Priority Mitigation	133
Mitigation Strategy	Security Notice	25
	Migration to Experimental Package	12
	Features Removal	13
	Sanitization	93
	Isolated Environment	2
	Replacement with Secure Implementation	34
Mitigation Effectiveness	No Longer Exploitable	104
	Still Exploitable	43
	Potentially Exploitable under Improper Configuration	12

TABLE V
STATISTICS ON THE TIME TAKEN BY DEVELOPERS TO MITIGATE AGENT APP VULNERABILITIES.

Statistic	Time (days)
Mean	30.3
Minimum	1
25th Percentile	2
Median	11
75th Percentile	50
Maximum	209

CVE-2024-42835 [21] is an LLM2Tool vulnerability in Langflow that allows Python code execution. The developer commented as follows [22]:

> *I don't think any update on this component is worth it in terms of security. Even implementing a sandbox is not enough to actually prevent malicious users to access the system, there are too many ways to escape it. Langflow admin must be aware of it and do not let any client to execute code.*

This suggests that the Langflow developer believes it is extremely challenging to design a comprehensive mitigation for the various evasion techniques attackers may use. Moreover, the developer feels that the responsibility for maintaining

[1]Note: we omit non-essential code for clarity and brevity.

system security should lie with the administrator, not the developer. A similar attitude is reflected in the mitigation of CVE-2024-23751 [14] which could result in potential database compromise in LlamaIndex's Text-to-SQL tools [23]:

> *There will ALWAYS be exploits when you have something like text to SQL. It's the developers role to use this feature safely and understand the risks. As per our SECURITY.md file in the repo, SQL vulnerabilities are generally not eligible for bounty or CVE.*

It should be noted that the *developers* referred to here are actually users who integrate LlamaIndex into their own applications. From LlamaIndex's perspective, these *developers* are its end users. Another example is CVE-2024-23750 [12], an LLM2Tool code execution vulnerability in MetaGPT. Its developers are concerned that any mitigation efforts might compromise necessary app functionality, and thus a trade-off is needed [24]:

> *I think it is reasonable in a certain sense, but there is always a tradeoff between security and functionality.*

The reason for this is that MetaGPT is an agent app specifically designed for software development. It requires the capability to execute code in order to test the generated software. Since the intended use of the generated software is not strictly restricted, there may be legitimate reasons—such as creating a system cleanup utility—that necessitate file deletion. Testing such software would require performing file deletion operations, which could potentially result in data loss.

Finding 7: More than half of the vulnerabilities (60.2%) were proactively addressed by developers. However, 28.1% of the vulnerabilities still remain unmitigated to this day due to various developer concerns.

As shown in Table IV, developers employ 6 different strategies to mitigate vulnerabilities in agent apps. Among these, *Sanitization* is the most commonly used approach. This is because *Sanitization* typically does not require major changes to the codebase; developers can simply add a few conditional checks, which saves both time and effort.

Finding 8: Developers generally use 6 types of mitigation strategies for vulnerabilities in agent apps: *Security Notice*, *Migration to Experimental Package*, *Feature Removal*, *Sanitization*, *Isolated Environment*, and *Replacement with a Secure Implementation*. Among these, *Sanitization* is the most widely adopted approach by developers, primarily because it offers an excellent balance between effectiveness and cost.

We next focus on the effectiveness of these mitigation strategies. We consider a vulnerability to be properly mitigated by developers if, after mitigation, it can no longer be exploited, or can only be exploited when users have not configured the system correctly. As shown in Table IV, out of all 221 vulnerabilities, 47.5% (105 out of 221) were not properly mitigated. If we look only at the 159 vulnerabilities that developers attempted to mitigate, only 73.0% (116 out of 159) of these mitigations were actually effective.

Finding 9: Nearly half (47.5%) of the vulnerabilities in agent apps have not been properly mitigated, and among those vulnerabilities that developers have attempted to mitigate, the effectiveness rate is only 73.0%. This indicates that the current efforts to mitigate vulnerabilities in agent apps are far from successful.

We aim to further investigate the reasons behind unsuccessful vulnerability mitigations. Among the 43 samples where developers attempted mitigation but failed to fully address the issue, we find that 39.5% (17 out of 43) rely solely on the *Security Notice* or *Migration to Experimental Package* strategies without implementing more robust measures. Additionally, 55.8% (24 out of 43) use the *Sanitization* strategy, but these mitigations are bypassed due to incomplete implementations.

Finding 10: The two main reasons for unsuccessful vulnerability mitigation are the exclusive reliance on weak strategies like *Security Notice* and *Migration to Experimental Package*, and the use of incomplete sanitization rules.

In fact, we observe that developers are not necessarily unaware of the unsuccessful mitigations, but rather, for certain vulnerabilities, it is challenging to find a robust mitigation approach that balances security and functionality, forcing them to make trade-offs. Our analysis reveals that more than half (24 out of 43) of the vulnerabilities with unsuccessful mitigations fall into the category of LLM2Tool vulnerabilities. Mitigating LLM2Tool vulnerabilities often struggles to simultaneously achieve both security guarantees and functionality guarantees, largely due to the open-ended, multi-scenario design philosophy of agent.

We explain this fact by using the `PALChain` module [25] from LangChain as an example. The `PALChain` module can perform logical reasoning and arithmetic calculations by executing Python code returned from LLM. Initially, `PALChain` did not perform any checks on the returned code, which meant that it could be maliciously exploited to execute system commands (e.g., `rm -rf /`). This security flaw was reported by researchers as CVE-2023-36258. As shown in Fig. 5a, to address CVE-2023-36258, the LangChain developers imposed certain restrictions on `PALChain`'s code execution capabilities. They used AST-based filtering to prohibit specific imports and restrict the use of certain methods. However, it was quickly discovered by security researchers that these defenses could be bypassed using functions like `__import__()`, as shown in Fig. 5b. In response, developers added `__import__()` to the sanitization rule set. Yet researchers continued to find ways to circumvent the filters, such as leveraging

Python's introspection features. For example, in Python 3.11, one could execute `''.__class__.__mro__[-1].__subclasses__()[140].__init__.__globals__['popen']('{MALICIOUS_CMD}').read()` to run `os.popen('{MALICIOUS_CMD}')`, thereby bypassing AST checks. As shown in Fig. 5c, developers continued to block these keywords yet again. Eventually, it became clear that even with ongoing patching, many potential bypass techniques still remained. It was ultimately acknowledged by LangChain developers that *Sanitization* alone could not fundamentally resolve LLM2Tool vulnerabilities [26]:

> *I'm going to re-title this PR since I don't think we can claim that it fixes arbitrary code execution in PALChain. Approaches based on filtering AST nodes rather than including AST nodes are impossible to guarantee.*

At this point, developers faced a dilemma. If they used a blacklist approach, updating the filter rules for every new exploit pattern, there would always be unforeseen bypasses and no true security guarantee. Conversely, a whitelist approach that only permits a small set of absolutely safe operations would run contrary to the open-ended, multi-scenario design philosophy of agent, potentially weakening the product's competitiveness and sacrificing functionality. Ultimately, it is extremely difficult to clearly define a universally accepted boundary between functionality and vulnerability for open-ended tools in agent apps.

After discussion, the LangChain team decided to adopt the *Migration to Experimental Package* strategy, which is, in essence, a compromise between security and functionality. This approach has also been adopted by the well-known project LlamaIndex. Here is the LangChain developers' official announcement [27]:

> *One of the things people love about langchain is how fast the community adds new ideas and papers...With 'langchain-experimental' you can contribute experimental ideas without worrying that it'll be misconstrued for production-ready code.*

In addition, many agent app developers have proactively provided containerized deployment options. Containerization is considered an effective security measure for controlling code and command execution. The basic idea is that even if developers cannot prevent every possible bypass, as long as the execution environment is isolated, attackers cannot affect the host system through code or command execution. While containerization has become a good practice within the community, it also comes with various limitations, which will be discussed in Section V-B.

Finding 11: Among the samples of unsuccessful mitigation, over half are LLM2Tool vulnerabilities. The core challenge is that the agent's design philosophy frequently forces developers to choose between security and functionality. *Migration to Experimental Package* and *Containerization* represent imperfect but currently acceptable trade-offs adopted by the community.

As shown in Table IV, we observe a special status after vulnerability mitigations. From the developers' perspective, they have already provided security measures to prevent risks. However, security issues may still arise if users fail to configure these measures properly. The main characteristic of these mitigation strategies is that they leave it up to users to decide which behaviors of these open-ended tools should be permitted and which should be restricted. In essence, this shifts the responsibility for security from developers to users, requiring users to bear the associated security risks. It is questionable, though, whether users possess sufficient security awareness to properly configure these settings during deployment stage according to their own use cases, especially since many users may prioritize convenience and functionality over security.

Finding 12: The agent app developers sometimes grant users extensive configuration permissions for open-ended tools, which in effect shifts security responsibility from the developers to the users. However, whether users can properly configure these settings during the deployment is questionable. This highlights the importance of addressing security concerns not only during the development but also during the deployment.

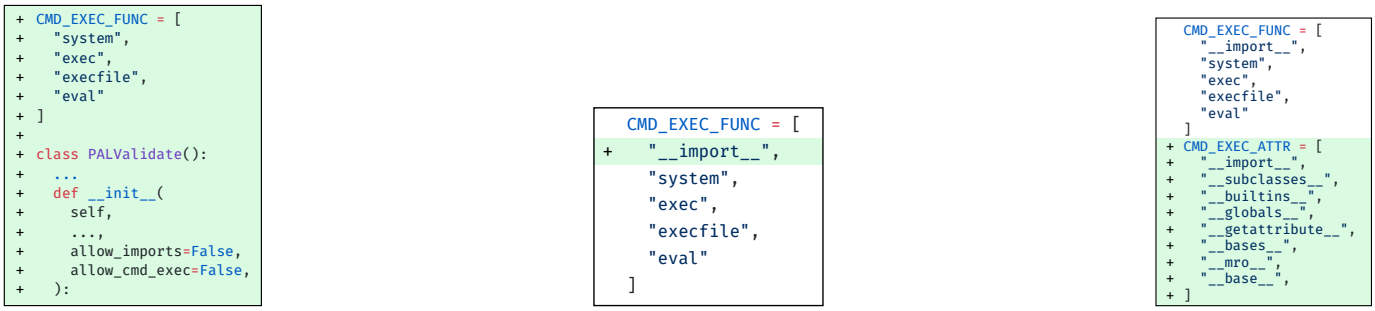
V. Discussion

A. Blurred Line Between Functionality and Vulnerability

In agent apps, establishing a universally accepted and clear boundary between functionality and vulnerability for the open-ended tools is extremely challenging. This difficulty stems from a fundamental contradiction between the design philosophy of agent and the philosophy of security. On the one hand, in order to make agent apps more competitive, they must be granted greater capabilities and flexibility, which inevitably requires providing agent apps with broader permissions. On the other hand, the security philosophy is grounded in the weakest-link effect and the principle of least privilege, emphasizing a conservative approach that aims to minimize risks, as every additional permission expands the potential attack surface and any weak link could be exploited by attackers.

We recognize that there will inevitably be a trade-off between functionality and security in agent apps. However, there currently is no universally accepted answer as to how this trade-off should be made, as different scenarios and application contexts have their own unique requirements.

As such, it is crucial for governmental bodies and industry alliances to collaborate in developing standards and guidelines for various mainstream usage scenarios of agent apps, clearly defining what security means in each context. Such standards would help developers and users better understand the security responsibilities and boundaries of agent apps, and provide authoritative guidance for developers' future vulnerability mitigation practices.



(a) (b) (c)

Fig. 5. The sequential fixes for `PALChain` and its bypasses, illustrating the difficulty of achieving comprehensive mitigation. (a) The fix of CVE-2023-36258; (b) The fix of CVE-2023-44467 (bypass of CVE-2023-36258); (c) The fix of CVE-2023-27444 (bypass of CVE-2023-44467).

Besides, developers are advised to make tool design more fine-grained, avoiding the implementation of overly broad, multi-purpose tools (such as general code execution or database operation tools) within agent apps. Instead, tools should be further subdivided to fit specific application contexts. For example, in the domain of data analysis, the code execution tool could be split into separate modules for data acquisition, data processing, and data visualization, each with a clearly defined function and purpose. This modular approach will facilitate fine-grained permission management for agent apps in the future.

B. Limitations of Containerization

As highlighted in Finding 11, containerization has become a popular security practice within the community. However, it is important to recognize that containerization is not a silver bullet and comes with its own set of limitations. Firstly, containerization has scalability issues. In multi-tenant scenarios serving millions of users, it is virtually impossible to create a new container for every user session due to resource and time constraints. In practice, necessary trade-offs must be made during deployment of agent app, which may compromise security. Next, containerization is only suitable for applications that do not require persistent modifications to the system, such as code execution for data analysis. For scenarios like database operations or streamlining user-side smartphone workflow, containerization is not applicable. Finally, similar to what we discussed in Section V-A, configuring containers also presents a dilemma: there is no universally accepted answer to how much privilege a container should be granted (e.g., should it connect to the internet?). The balance between security and functionality will arise here too.

C. Complexity of Resource Management

As indicated in Finding 2 and Finding 5, agent apps face significant challenges in managing resources, which manifest in two main aspects. First is the diversity of resource types. Agent apps rely on a wide range of resources, including prompts, LLMs, files, web pages, knowledge-bases, tools, and execution environments. Each type of resource may introduce its own unique attack vectors. Second is the broad spectrum of resource origins. These resources may come from users, developers, or third-party service providers. A particularly notable trend is the recent rise of Model Context Protocol (MCP), which promote the decoupling of functionalities traditionally internal to agent apps. These functionalities are increasingly supported by the surrounding community ecosystem rather than agent app itself, further complicating resource management in agent apps.

This growing complexity significantly expands the attack surface of agent apps. However, there is currently a lack of comprehensive tools or methodologies to map the full landscape of resources involved in agent apps. This absence makes it difficult to systematically assess potential entry points for attacks. Building a clear and complete picture of resources involved in agent apps would greatly assist QA engineer and security researchers in developing more targeted tools for automated testing and vulnerability detection in the future.

D. Deployment-Stage Security

The open-ended nature of agent compels developers to grant users significant configuration privileges in order to maintain the flexibility of app functionalities. As highlighted in Finding 12, developers often shift the security responsibility onto the users. This underscores the importance of addressing the security risks during the deployment stage of agent apps, as users typically lack the necessary security expertise and awareness to properly configure these settings. To mitigate this issue, researchers can design tools or methodologies to provide users with configuration recommendations that balance security and functionality based on their usage needs of the agent app, while also automatically alerting them to those deployment configurations that may pose serious security risks.

E. Challenges in Detecting LLM2Tool Vulnerabilities

The natural language format of agent inputs and the involvement of LLMs pose unique challenges for detecting LLM2Tool vulnerabilities. First, compared to structured data, natural language introduces an extremely large input space enriched with complex semantics, making it difficult to design effective fuzzing mutation strategies. Additionally, the non-deterministic mapping between LLM inputs and outputs

complicates the inference of relationships between natural-language-formatted agent inputs and structured tool inputs, thereby hindering effective injection of vulnerability-triggering payloads into internal tools.

VI. Related Work

A. Agent Security

Some existing research has begun to focus on the security of agent, primarily examining inherent flaws in their architecture and design. Yang et al. [28] and Wang et al. [29] each proposed a distinct method for backdoor attacks targeting agent. Abdelnabi et al. [30] introduced the concept of Indirect Prompt Injection, which can potentially lead to goal hijacking in agent. InjecAgent [31] and R-Judge [32] aim to benchmark various attack and defense strategies against agent by providing standardized testing protocols and datasets. ToolEmu [33] and AgentDojo [34] focus on constructing simulated environments to evaluate agent security. IsolateGPT [35] explores an isolated execution architecture to ensure the secure operation of third-party components within agent. However, these studies are largely based on theoretical models of agent and carry implicit security assumptions. It remains questionable whether developers strictly follow such theoretical models and assumptions when designing real-world apps, and even more uncertain whether such apps can achieve broad adoption and significant impact.

B. Agent App Security

A smaller body of work has shifted attention toward the security of real-world agent implementations, conducting extensive testing on actual, widely-used agent apps. Liu et al. [7] developed LLMSmith, which employs sink-point and call-chain analysis to uncover numerous remote command execution (RCE) vulnerabilities in agent apps. Their real-world evaluations revealed that some deployed online agent apps were susceptible to API key theft, and some could even be exploited to implant backdoors by establishing a reverse shell. Pedro et al. [8] investigated agent apps with Text-to-SQL capabilities, demonstrating that prompt injection techniques can be used to perform SQL injection attacks, compromising both confidentiality and integrity of databases. They also proposed several imperfect but practically viable defensive measures. Liu et al. [15] proposed AgentFuzz, which builds upon traditional directed greybox fuzzing approach but introduces novel seed generation and mutation strategies tailored to the natural language input characteristics of agents, successfully uncovering multiple taint-style vulnerabilities in mainstream agent apps. However, existing research was primarily concentrated on vulnerability detection. Currently, there is a lack of systematic and in-depth understanding of the unique characteristics of vulnerabilities in agent apps and their effective mitigation.

VII. Threats to Validity

Internal Validity: Although we have tried our best to collect vulnerability dataset, we acknowledge that some omissions may still exist. Additionally, our setting of a 60-day mitigation time threshold is based on our security expertise, but different standards may exist within the industry.

External Validity: Given the rapid evolution of the agent app ecosystem, the timeliness of this study constitutes a primary threat to external validity. Emerging paradigms of agent apps and newly disclosed vulnerabilities in the future may challenge the generalizability of our conclusions. Nevertheless, we believe this study lays a solid foundation for understanding and advancing agent app security.

VIII. Conclusion

Agent apps are becoming increasingly prevalent in daily lives, underscoring the critical need to systematically understand the vulnerabilities and their mitigation. In this paper, we presented the first comprehensive study on the vulnerabilities of agent apps, the mitigation practices of app developers, and the associated challenges and trade-offs. Through our analysis of 221 vulnerabilities across 50 apps, we identified 14 types of vulnerability and 16 root causes spanning 7 components. Our study further investigated developer reactions, evaluated the effectiveness of various mitigation strategies, and explored the practical challenges and inevitable trade-offs in vulnerability mitigation. For agent app developers, maintainers, and security researchers, we distilled 12 key findings to help them deal with security risks in agent app. Finally, we discussed the insights from findings and provide some suggestions for future research directions.

This work reveals significant security gaps in agent apps and demonstrates the pressing need for focused research efforts to enhance their security posture. Our findings serve as both a warning and a roadmap for securing this increasingly important class of apps.

Acknowledgment

We would like to thank the anonymous reviewers for their insightful comments that helped improve the quality of the paper. This work was supported in part by the National Natural Science Foundation of China (U2436207, 62172105, 62402116) and the research funding of Shanghai Municipal Education Commission (24KXZNA08). Jiarun Dai and Min Yang are the corresponding authors. Min Yang is a faculty of Shanghai Institute of Intelligent Electronics & Systems, and Engineering Research Center of Cyber Security Auditing and Monitoring, Ministry of Education, China.

References

[1] X. Huang, W. Liu, X. Chen, X. Wang, H. Wang, D. Lian, Y. Wang, R. Tang, and E. Chen, "Understanding the planning of LLM agents: A survey," 2024. [Online]. Available: https://arxiv.org/abs/2402.02716

[2] E. Eigner and T. Händler, "Determinants of LLM-assisted Decision-Making," 2024. [Online]. Available: https://arxiv.org/abs/2402.17385

[3] Z. Shen, "LLM With Tools: A Survey," 2024. [Online]. Available: https://arxiv.org/abs/2409.18807

[4] "MetaGPT," https://github.com/FoundationAgents/MetaGPT.

[5] "LangChain," https://github.com/langchain-ai/langchain.

[6] "TaskWeaver," https://github.com/microsoft/TaskWeaver.

[7] T. Liu, Z. Deng, G. Meng, Y. Li, and K. Chen, "Demystifying RCE Vulnerabilities in LLM-Integrated Apps," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '24. New York, NY, USA: Association for Computing Machinery, Dec. 2024, pp. 1716–1730.

[8] R. Pedro, M. E. Coimbra, D. Castro, P. Carreira, and N. Santos, " Prompt-to-SQL Injections in LLM-Integrated Web Applications: Risks and Defenses ," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2025, pp. 76–88. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICSE55347.2025.00007

[9] F. Perez and I. Ribeiro, "Ignore Previous Prompt: Attack Techniques For Language Models," in *NeurIPS ML Safety Workshop*, 2022. [Online]. Available: https://openreview.net/forum?id=qiaRo_7Zmug

[10] Y. Liu, Y. Jia, R. Geng, J. Jia, and N. Z. Gong, "Formalizing and Benchmarking Prompt Injection Attacks and Defenses," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 1831–1847. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/liu-yupei

[11] "CVE-2023-39659," https://nvd.nist.gov/vuln/detail/CVE-2023-39659.

[12] "CVE-2024-23750," https://nvd.nist.gov/vuln/detail/CVE-2024-23750.

[13] "CVE-2023-36189," https://nvd.nist.gov/vuln/detail/CVE-2023-36189.

[14] "CVE-2024-23751," https://nvd.nist.gov/vuln/detail/CVE-2024-23751.

[15] F. Liu, Y. Zhang, J. Luo, J. Dai, T. Chen, L. Yuan, Z. Yu, Y. Shi, K. Li, C. Zhou, H. Chen, and M. Yang, "Make agent defeat agent: Automatic detection of taint-style vulnerabilities in LLM-based agents," in *34th USENIX Security Symposium (USENIX Security 25)*, 2025, pp. 3767–3786.

[16] "Awesome Agents," https://github.com/kyrolabs/awesome-agents.

[17] "Awesome-AI-Agents," https://github.com/Jenqyang/Awesome-AI-Agents.

[18] "MITRE CVE," https://cve.mitre.org/.

[19] "CVE-2024-23752," https://nvd.nist.gov/vuln/detail/CVE-2024-23752.

[20] "CVE-2024-11958," https://nvd.nist.gov/vuln/detail/CVE-2024-11958.

[21] "CVE-2024-42835," https://nvd.nist.gov/vuln/detail/CVE-2024-42835.

[22] "The developers' comments on the mitigation of CVE-2024-42835," https://github.com/langflow-ai/langflow/issues/2908#issuecomment-2255947085.

[23] "The developers' comments on the mitigation of CVE-2024-23751," https://github.com/run-llama/llama_index/issues/9957#issuecomment-2138291340.

[24] "The developers' comments on the mitigation of CVE-2024-23750," https://github.com/FoundationAgents/MetaGPT/issues/731#issuecomment-2012260743.

[25] "PALChain," https://github.com/langchain-ai/langchain-experimental/blob/main/libs/experimental/langchain_experimental/pal_chain/base.py.

[26] "The developers' comments on the difficulty of vulnerability mitigation in palchain," https://github.com/langchain-ai/langchain/pull/17091#issuecomment-1940381298.

[27] "The LangChain developers' official announcement about the adoption of experimental package," https://github.com/langchain-ai/langchain/discussions/8043.

[28] W. Yang, X. Bi, Y. Lin, S. Chen, J. Zhou, and X. Sun, "Watch out for your agents! investigating backdoor threats to llm-based agents," in *Advances in Neural Information Processing Systems*, A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, Eds., vol. 37. Curran Associates, Inc., 2024, pp. 100 938–100 964. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2024/file/b6e9d6f4f3428cd5f3f9e9bbae2cab10-Paper-Conference.pdf

[29] Y. Wang, D. Xue, S. Zhang, and S. Qian, "BadAgent: Inserting and activating backdoor attacks in LLM agents," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, L.-W. Ku, A. Martins, and V. Srikumar, Eds. Bangkok, Thailand: Association for Computational Linguistics, Aug. 2024, pp. 9811–9827. [Online]. Available: https://aclanthology.org/2024.acl-long.530/

[30] 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

[31] Q. Zhan, Z. Liang, Z. Ying, and D. Kang, "InjecAgent: Benchmarking indirect prompt injections in tool-integrated large language model agents," in *Findings of the Association for Computational Linguistics: ACL 2024*, L.-W. Ku, A. Martins, and V. Srikumar, Eds. Bangkok, Thailand: Association for Computational Linguistics, Aug. 2024, pp. 10 471–10 506. [Online]. Available: https://aclanthology.org/2024.findings-acl.624/

[32] T. Yuan, Z. He, L. Dong, Y. Wang, R. Zhao, T. Xia, L. Xu, B. Zhou, F. Li, Z. Zhang, R. Wang, and G. Liu, "R-judge: Benchmarking safety risk awareness for LLM agents," in *Findings of the Association for Computational Linguistics: EMNLP 2024*, Y. Al-Onaizan, M. Bansal, and Y.-N. Chen, Eds. Miami, Florida, USA: Association for Computational Linguistics, Nov. 2024, pp. 1467–1490. [Online]. Available: https://aclanthology.org/2024.findings-emnlp.79/

[33] Y. Ruan, H. Dong, A. Wang, S. Pitis, Y. Zhou, J. Ba, Y. Dubois, C. J. Maddison, and T. Hashimoto, "Identifying the risks of LM agents with an LM-emulated sandbox," in *The Twelfth International Conference on Learning Representations*, 2024. [Online]. Available: https://openreview.net/forum?id=GEcwtMk1uA

[34] E. Debenedetti, J. Zhang, M. Balunovic, L. Beurer-Kellner, M. Fischer, and F. Tramèr, "Agentdojo: A dynamic environment to evaluate prompt injection attacks and defenses for llm agents," in *Advances in Neural Information Processing Systems*, A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, Eds., vol. 37. Curran Associates, Inc., 2024, pp. 82 895–82 920. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2024/file/97091a5177d8dc64b1da8bf3e1f6fb54-Paper-Datasets_and_Benchmarks_Track.pdf

[35] Y. Wu, F. Roesner, T. Kohno, N. Zhang, and U. Iqbal, "IsolateGPT: An execution isolation architecture for LLM-Based agentic systems," in *Proceedings 2025 Network and Distributed System Security Symposium*, 2025. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/isolategpt-an-execution-isolation-architecture-for-llm-based-agentic-systems/