Scientific Research Publishing

Journal of Information Security, 2026, 17(4), 514-539
https://www.scirp.org/journal/jis
ISSN Online: 2153-1242
ISSN Print: 2153-1234

Enabling Multi-Agent Penetration Testing Environments for Software Vulnerability Detection

Raj K. Madisetti, Siddansh Chawla, Vijay K. Madisetti*

Georgia Institute of Technology, Atlanta, GA, USA
Email: rmadisetti3@gatech.edu, siddansh@gatech.edu, *vkm@gatech.edu

How to cite this paper: Madisetti, R.K., Chawla, S. and Madisetti, V.K. (2026) Enabling Multi-Agent Penetration Testing Environments for Software Vulnerability Detection. *Journal of Information Security*, **17**, 514-539.
https://doi.org/10.4236/jis.2026.174023

Received: September 29, 2026
Accepted: October 24, 2026
Published: October 27, 2026

Copyright © 2026 by author(s) and Scientific Research Publishing Inc.
This work is licensed under the Creative Commons Attribution International License (CC BY 4.0).
http://creativecommons.org/licenses/by/4.0/

Open Access

Abstract

In light of the ongoing challenge posed by unknown vulnerabilities in software solutions, the prompt identification and rectification of these weaknesses are of vital importance. Traditional penetration testing, although thorough, typically demands substantial human resources and time commitments. To contend with this issue, we propose an innovative penetration testing framework that incorporates natural language processing, utilizing large language model (LLM) driven multi-agent systems to automate and expedite vulnerability detection. By integrating insights from established exploit signatures and AI-driven fuzzing techniques, our framework, LLMAZD, preemptively examines target software within a controlled setting, effectively identifying both zero-day and one-day exploits, as well as system vulnerabilities, in a cost-effective manner. The framework demonstrates versatility in detecting multiple vulnerability classes, including SQL injection, Cross-Site Scripting (XSS), Cross-Site Request Forgery (CSRF), file upload vulnerabilities, and other different types, all while maintaining efficiency and cost-effectiveness throughout the testing process. Extensive evaluations on more than 20 CVE test cases reveal that the framework substantially enhances detection capabilities, reduces the overall burden of testing, and offers a practical pathway towards more secure software ecosystems.

Keywords

Cyber-Security, Large Language Models (LLMs), Zero-Day Exploit, Malware, Vulnerability, Exposure, Multi-Agent Systems, Penetration Testing

1. Introduction

Exploits in software and online services pose significant risks to nearly every or-

ganization. These vulnerabilities adversely affect services and business outcomes, potentially leading to outages and disruptions. Often, numerous bugs remain unpatched, resulting in targeted exploitation of specific services, and significant system downtimes may occur due to unidentified vulnerabilities. For instance, a report [1] by Google's TAG and Mandiant documented a 50 percent annual increase in the exploitation of zero-day vulnerabilities. Malicious threat actors identify such system loopholes and exploit them, causing alarm among both users and software vendors. Consequently, these attacks can exfiltrate sensitive information over prolonged periods without the awareness of either party. Instances where vendors lack the opportunity to apply patches and are often unaware of the bug or ongoing exploit are typically classified as zero-day vulnerabilities. This problem constitutes a pervasive challenge in the field of cyber security, for which no immediate remedy is available until a patch is developed.

Addressing this challenge requires a team of cybersecurity experts to conduct simulated attacks on software within a controlled environment to identify vulnerabilities. Following the detection of a bug, a patch can be rapidly deployed to users. Once these exploits and bugs are identified, they are publicly disclosed after the development of a proof-of-concept and are subsequently published as Common Vulnerabilities and Exposures (CVE) reports. These reports, which may not always provide patching solutions, clarify vulnerabilities that can be exploited by threat actors in what is known as a one-day scenario. Each CVE contains a detailed description, typical consequences, potential mitigations, and illustrative examples. Additionally, every CVE is linked to a Common Weakness Enumeration (CWE) report that outlines the root cause of the vulnerability. This information serves a dual purpose: it can aid penetration testing teams in identifying bugs within their systems, while it also provides malicious actors with insights to perpetrate exploits. Considering the persistent issue of undetected vulnerabilities and delayed responses, this process—though limited—in identifying and addressing issues necessitates 1) substantial manpower and time for vulnerability detection, 2) additional time for bug resolution, and 3) comprehensive cross-system testing of the patch. The overall time duration from the initial step to delivering a patch to end-users may result in a considerable period during which the system remains susceptible to exploitation.

To expedite the detection and remediation of vulnerabilities, advanced LLM agents can be leveraged to develop proficient systems capable of predicting and identifying zero-day vulnerabilities. These LLM agents are advancing and demonstrate significant potential in the realm of cyber security, thus paving the way for innovative methodologies in this discipline.

Building on this potential, prior investigations in the domain of cyber security and artificial intelligence (employing Large Language Models) have provided substantial proof of concept for the deployment of LLMs in roles analogously to cyber security researchers and malicious threat actors. In particular [2], the nefarious utilization of LLMs, orchestrated within a multi-agent framework, has illustrated

models such as GPT-4/4o by OpenAI exploiting zero-day vulnerabilities irrespective of the availability of bug descriptions. In contrast, a recent study [3] has proposed a multi-agent architecture for conducting automated penetration testing on designated systems. While scholars and practitioners in this field have manifested the ability to employ language models for the detection and exploitation of vulnerabilities, the capacity to identify true zero-day vulnerabilities remains elusive. Furthermore, considering that the outputs and actions of large language models do not inherently prioritize safety, human oversight is imperative to ensure their responsible application in real-world scenarios. Autonomous execution of these systems often encounters impediments due to conditions like LLM hallucination, which presents a considerable threat to cyber security applications as outlined in [4], inefficiencies in the knowledge base, and the necessity for specialized language models to carry out tasks effectively.

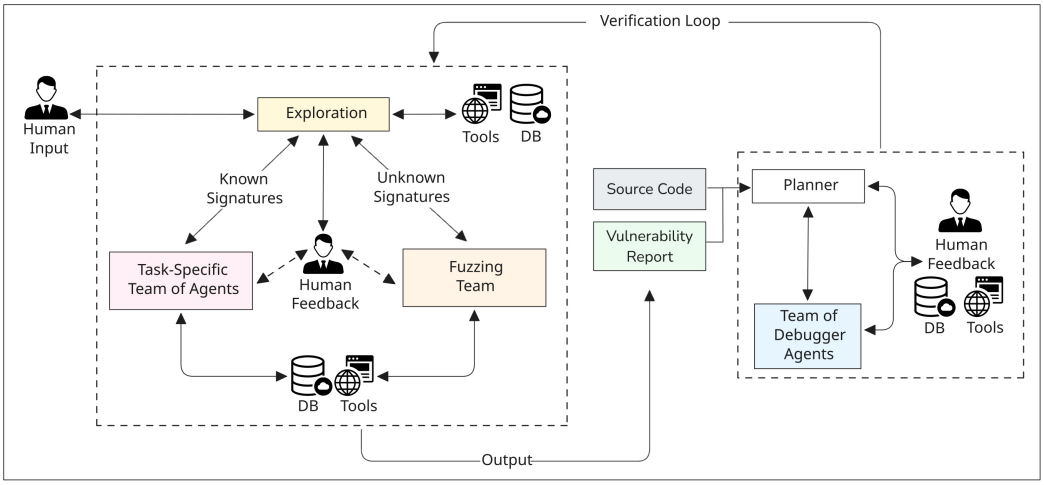


Figure 1. Overall architecture of an LLM-based framework for detecting and addressing Zero-day and One-day vulnerabilities.

This study aims to develop an LLM-based framework designed to assist in the detection of zero-day and one-day exploits and anomalies, leveraging information from CVE and CWE reports, and subsequently attempts to provide a corrective patch to effectively address the vulnerability. We propose an architectural design depicted in **Figure 1**, characterized by a multi-agent and multi-team configuration that facilitates efforts in resolving web-related bugs. Owing to the complexity inherent in the overall architecture, the research has been divided into two major components. In this paper, we present the initial half of the research, the framework, LLMAZD, wherein we focus our investigation on the development of a framework to identify bugs and authentic zero-day vulnerabilities within a web-based software environment that are susceptible to exploitation. We expound upon and illustrate a proactive LLM-based agent system distinguished by a hierarchical design pattern, which integrates multiple sub-teams, alongside an array of tools and databases, to efficiently detect zero-day vulnerabilities. The testing

highlights the effectiveness of the framework in real-world scenarios; through exhaustive testing on multiple software applications associated with more than 20 CVEs, we demonstrate the efficacy of detection across different vulnerability classes, displaying the generalizability of the framework. Notably, the framework also discovered several novel vulnerabilities which were missed by the regular and manual pentesting approach. Our work significantly advances prior architectural proposals through the integration of novel elements and the replacement of outdated designs. This research offers a pragmatic approach for the detection of vulnerabilities in a web-based software system, with the provision of essential and effective oversight concerning LLM operations and decisions.

Section 2 provides a discussion of related research and elucidates how these advancements pertain to our current study. Sections 3 and 4 detail our proposed framework, along with its implementation and testing protocols. Section 5 presents a comprehensive analysis of the resultant data. Sections 6 and 7 offer insights into prospective research directions and conclude the study.

2. Related Work

Recent advancements have been observed in the application of large language model (LLM) architectures for the identification of zero-day exploits and assistance in penetration testing. These frameworks have incorporated Common Vulnerabilities and Exposures (CVE) data, have been optimized for state-of-the-art (SOTA) models, and have facilitated effective system investigation.

2.1. Google: Project Zero-Big Sleep

In November 2024, the Google Zero-Day Security Research Initiative and Google DeepMind, collectively referred to as Big Sleep, achieved a significant milestone by identifying the first publicly documented Zero-Day Memory Safety vulnerability via a Large Language Model (LLM) agent [5]. This research unveiled a null-pointer dereference within SQLite, which necessitated an in-depth investigation into the vulnerability. The Big Sleep team highlighted the substantial potential of identifying software vulnerabilities preemptively, before malicious entities can exploit them, thereby emphasizing the crucial role LLM Agents have in fostering robust software systems. Moreover, fuzzing, recognized as a cybersecurity technique involving the injection of systems with invalid, malformed, or unexpected inputs to unveil security vulnerabilities, has been validated as an effective method for mitigating system weaknesses. Accordingly, our methodology incorporates a Fuzzing Agent to address genuine zero-day attacks, as elucidated in subsequent sections.

2.2. HPTSA

An approach that resonated with LangChain's multi-agent designs was proposed by Zhu *et al.* [2]. This paper investigates how teams of LLM agents can autonomously and collaboratively exploit zero-day vulnerabilities in web applications. It

proposes a hierarchical architecture, called HPTSA (Hierarchical Planning and Task-Specific Agents), which consists of:

- **Hierarchical Planner**—Explores the target and devises a high-level strategy.
- **Team Manager for Agents**—Invokes task agent based on planner and keeps track of partial results.
- **Task-Specific Expert Agents**—Execute the exploit as assigned by the team manager, such as attempting to exploit specific vulnerabilities. Additionally, these agents have access to special tools such as Playwright for browsing.

The precision of HPTSA demonstrates a pass@5 rate of 53% and a pass@1 rate of 33.3% when utilizing GPT-4 in the absence of a vulnerability description. The framework has established an overall success rate of 18%, which translates to a cost of $24.39 per successful exploit. Consequently, multiple iterations are generally necessary to achieve a successful result, thereby increasing the overall expenditure and effort. Nevertheless, beyond the necessity for multiple iterations, the system is encumbered by additional constraints. The employment of a circumscribed number of specialized agents, developed for known vulnerability signatures, limits the range of diagnosable vulnerabilities, thereby hindering the detection of various zero-day exploits and other unidentified vulnerabilities. Furthermore, the framework was unable to exploit two vulnerabilities listed in the CVE database owing to "undiscoverable" endpoint routes, attributed to architectural limitations.

2.3. PENTESTAGENT

An alternative multi-agent approach employs LLM agents to generate security vulnerabilities through penetration testing while utilizing techniques such as Retrieval Augmented Generation (RAG). Shen *et al.* [3] presents an LLM-based penetration testing framework that automates the processes of intelligence gathering, vulnerability analysis, and exploitation stages. This framework comprises the reconnaissance agent, the search agent, the planning agent, and the execution agent, all of which execute the three stages of penetration testing. The reconnaissance agent synthesizes environmental information regarding the target host and dispatches a summary to a specific database. The search and planning agents ascertain potential vulnerabilities and devise strategies to exploit these respective weaknesses. Ultimately, the execution agent enacts the planned exploit and transmits the execution logs to the database. Theoretically, this strategy exhibits significant effectiveness and is capable of enhancing the proficiency of penetration testing. However, practical experimentation revealed several limitations. Firstly, LLM Agents occasionally necessitated supplementary knowledge to operate certain tools such as "ysoserial", thus requiring human intervention for successful task completion. Secondly, PENTESTAGENT was unable to recognize specific software components that do not function as standalone applications, such as plugins and underlying components, specifically Nginx, PHPMailer, or Ghostscript. Lastly, LLM hallucinations can detrimentally affect system efficiency, causing in-

finite loops, misfired actions, and numerous errors throughout the process.

2.4. PENTEST-AI

Bianou *et al.* [6] introduces "PENTEST-AI", an LLM-based multi-agent system developed to automate penetration testing by utilizing the MITRE ATT&CK framework. In a manner akin to PENTESTAGENT [3], the penetration testing process is compartmentalized into a reconnaissance agent, exploit agent, and post-exploitation agent, overseen by an overarching centralized planning agent. The LLMs provide methodical reasoning, dynamically adapt to target environments, and select suitable attack techniques based on MITRE ATT&CK tactics and techniques. In contrast to PENTESTGPT [7], which functions primarily as a solitary agent and serves as an interactive LLM-driven assistant guiding users through penetration testing steps in a linear, command-response format, PENTEST-AI proposes a collaborative, autonomous multi-agent architecture conducive to more scalable, autonomous, and intricate decision-making throughout the testing process. Nonetheless, PENTEST-AI exhibits certain limitations. The system is susceptible to LLM Hallucinations, where the LLM produces inaccurate or unsafe commands. PENTEST-AI similarly contends with nuanced real-world scenarios such as custom network configurations, proprietary software, or rapidly evolving environments where meticulous human judgment remains indispensable. Furthermore, this system may falter in coordinating between multiple LLM agents and external penetration testing tools, potentially disrupting the automation pipeline.

2.5. LLMs Killed the Script Kiddie

Moskal *et al.* [8] also use LLM Agents, but apply it to the concept of Network Threat Testing. They developed a Docker sandbox in order to execute and evaluate code produced by LLMs. Then, prompt engineering with the priority of directing agents and execution was discussed. A GPT-3.5-Turbo powered kill chain and threat actor SDP with prompt engineering and a controller module was created to identify zero-day network exploits. Similarly to the previous two systems, there were limitations in the capacity of the LLMs. First, the LLM's training data plays a crucial role in determining the final form of the LLM. Most of the data used to train the GPT models was from June 2021 and, in an ever-changing cyber scope, is more than likely outdated material. Furthermore, the LLMs also had a difficult time understanding the tactical stages comprising the kill chain process. The framework does not leverage background knowledge and common sense, which are inferred qualities that a threat actor possesses.

2.6. Traditional ML & DL Frameworks

Traditional Machine Learning and Deep Learning frameworks were also implemented to identify zero-day exploits. Touré *et al.* [9] utilized supervised algorithms such as Decision Trees (DT), Random Forest (RF), and K-Nearest Neigh-

bors (KNN) to detect zero days with an accuracy of 98% - 100%. The model is very precise in pinpointing anomalies among clusters and correlation tables, since it is trained using test cases on IBM and NSL-KDD data sets. Similarly, Teymourlouei *et al.* [10] compare two methods of zero-day attack detection. One uses a common data reduction technique to eliminate non-threats to reveal the anomalies. The other uses deep learning through a neural network with an accuracy of more than 97% and a loss of 0.1.

2.7. Research Gaps

Table 1 visualizes features, methods, and issues of previous research compared to LLMAZD. In summary, the drawbacks of previous research resonate around 1) LLM hallucination, which poses a significant threat for cybersecurity applications by generating inaccurate or misleading outputs that can compromise system reliability; sample case studies have been demonstrated in the **Appendix** section A; 2) Substantial context length in cyber-related tasks, which restricts the agent's ability to fully capture the relevant context of prior team members, indirectly increasing the average cost of the solution due to the need for multiple runs to achieve successful results; 3) Coordination between agents, where ineffective handovers in multi-agent systems lead to fragmented workflows and reduced efficiency. These gaps highlight the need for improved LLM robustness, optimized context handling, and seamless agent collaboration to enhance the efficacy of AI-driven cybersecurity frameworks.

Table 1. Literature comparative analysis based on solution features and noticeable issues across the previous frameworks.

Features	HPTSA [2]	PENTESTAGENT [3]	PENTESTAI [6]	PENTESTGPT [7]	ML-DL Based Methods [10]	LLMAZD
LLM Based	✓	✓	✓	✓	—	✓
Human Conversational	×	✓	×	✓	—	✓
Approach Type	Web Based	Hybrid	Network Based	Hybrid	Network Based	Hybrid
Multi-Agent System	✓	✓	✓	×	—	✓
Persistent Issues						
Cost related	✓	✓	—	✓	×	×
Context Length	✓	✓	✓	✓	—	×
LLM Hallucinations	✓	✓	✓	✓	—	×
Agent Handovers	✓	✓	✓	—	—	×

Note: Best outcome for Features—[✓], while for Persistent Issues—[×].

3. Proposed Methodology

Contrary to previously developed architectures, which exhibit limitations in their ability to identify novel vulnerabilities and occasionally face challenges in determining subsequent steps due to the rapid evolution of context, particularly within

cybersecurity tasks, our methodology introduces an advanced implementation of a large language model (LLM)-based multi-agent system. This system relies on fundamental components such as Human-in-the-loop processes, the use of Cyber Analysis Graphs (CAGs), and an array of tools providing extensive capabilities. We adopt a proactive approach designed to identify vulnerabilities that may be exploited in real-world contexts. The following sub-sections elaborate on the critical components, designs of the multi-agent system, and other relevant information.

3.1. Key Components

1) *Human-in-the-Loop*: Building on the idea that LLM agents are powerful but not yet perfect, we incorporated a human feedback module to monitor the agent's interaction with its environment. This feedback module integrates a multi-function setup providing overall control to the human; executing the framework. This approach grants the executor privileged actions such as:

- **Approving** a specific decision path or a tool call.
- **Reviewing** a particular output of the agent.
- **Suggesting** alternate actions that can be taken.

The introduction of such privileges facilitating human oversight has significantly mitigated the risk of escalating and compounding errors within multi-agent systems, effectively addressing a critical yet often neglected issue within the field. Our research is focused on exploring the potential for automating this feedback loop through the employment of agents.

2) *Decentralized Architecture Design*: Multi-agent systems derive considerable advantage from direct inter-agent collaboration [11]. To harness this capability to its fullest extent, we have developed a sophisticated conversational framework facilitating inter-agent communication. By temporarily abrogating the managerial function, the system is able to efficiently process transfer requests, thereby improving context sharing and negotiation among agents, enhancing operational efficiency, and indirectly mitigating costs associated with superfluous LLM calls.

3) *Pre-Loaded Context*: In lieu of employing an extensive document-based knowledge base with Retrieval-Augmented Generation (RAG), we implement a more streamlined strategy by pre-loading context, akin to Cache-Augmented Generation [12]. This methodology optimizes query execution speed and accuracy through the use of a smaller, meticulously curated collection of documents. Each agent is furnished with a customized prompt, an output schema for generation, and a selective assortment of documents that provide illustrative data on various types of attacks. Although a direct implementation utilizing a KV-cache could further augment the system, as previously demonstrated in academic research, at the time of development, both LangChain and LangGraph were devoid of supporting tools necessary for this sophisticated approach.

It is acknowledged that zero-day vulnerabilities may encompass unidentified

signatures, which differ from the defined signatures that are typically recognized. This characteristic renders it nearly infeasible for a team of agents, each with specialized task-oriented roles, to effectively identify them. Consequently, the approach is bifurcated into two primary components: one sub-team is dedicated to addressing existing or known vulnerability signatures similar to a process orchestrated by Zhu *et al.* [2] but improved substantially using novel components, while the other sub-team is tasked with detecting unknown, potentially novel vulnerabilities and exposures. An in-depth analysis of the use cases and advantages of this strategy is presented in Section 5.4. The ensuing discussion will provide a comprehensive examination of the architecture, elucidating the types of agents, the format of prompts, and the tools employed.

3.2. Part 1: Task-Specific Agents

This part of the framework targets known vulnerability signatures. The architecture is summarized in Figure 2.

1) *Generic agents*:

- **Exploration Agent:** In the context of invasive cybersecurity methodologies, reconnaissance constitutes the foundational phase, as affirmed by prior investigations conducted by other scholars. Similarly, our approach incorporates this essential step, where a planning agent plays a pivotal role in any cybersecurity endeavor. The agent's primary function is to systematically explore all potential pathways within a specific application and identify elements that could be vulnerable to exploitation. To achieve this, the agent deploys suitable tools, such as Playwright for web browsing, shell tools, and network mapping utilities, as required. Furthermore, we employ a structured output format to verify the traversed paths and the gathered elements. This structured method facilitates the provision of a clearly delineated plan of execution to the subsequent agent.

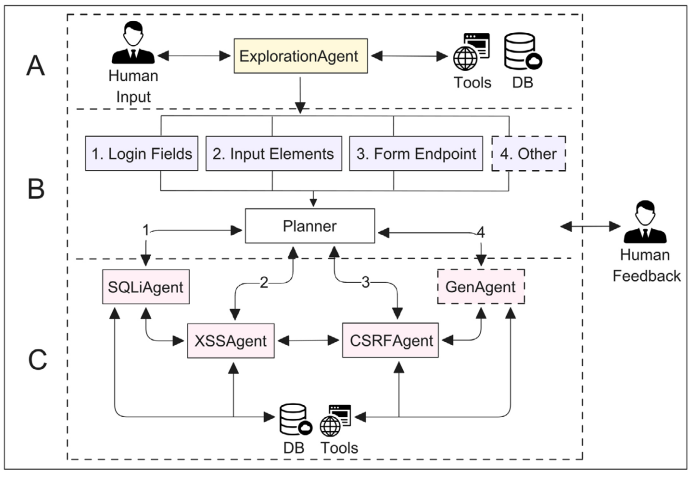


Figure 2. Part 1—Architecture Diagram: A: represents the exploration stage (where ExplorationAgent analyzes the exposed software endpoints), B: represents the mapping stage (using the output from ExplorationAgent to map the relevant elements), and C: represents the execution staging phase and the generation of task-specific agents.

- **Planner:** The primary function of a manager agent is to redirect queries to designated agents as determined by the exploration agent; additionally, it supplies supplementary context when required. Instead of utilizing this approach, we evaluate an alternative component known as the planner module, which systematically correlates specific elements with predefined vulnerability signatures. As illustrated in **Figure 2**, it efficiently maps input fields to the corresponding agent. This approach mitigates the risk of context misinterpretation for the manager when directing queries to the relevant entity.

2) *Task Specific Agents*: The notion of detailed prompting has facilitated the development of role-playing LLM-based agents. Through these prompts, an agent can be aligned and adapted to assume a role necessary for the completion of a real-world task. We have developed three task-specific cybersecurity agents capable of producing outputs consistent with their attack signatures.

- **SQLi Agent:** This agent Utilizes information from "CWE-89" [13] as its base prompt, which enables the agent to generate and iterate SQL commands that are malicious in nature. Due to the complexity of the SQL commands that need to be generated, we have included additional document references mainly containing complex SQL commands for the prompt apart from the CWE resource.
- **XSS Agent:** This agent uses concepts iterated from CWE-79 [14], enabling fluency in generating malicious JavaScript code. The agent prompt template also contains supplementary materials detailing common injection vectors, DOM manipulation techniques, and browser-specific payloads to enhance effectiveness in exploiting cross-site scripting vulnerabilities.
- **CSRF Agent:** This agent utilizes CWE-352 [15] as the primary reference, enabling the agent to effectively craft and validate malicious requests aimed at exploiting vulnerabilities related to Cross-Site Request Forgery. To accurately replicate and evaluate CSRF attacks, the agent leverages supplemental documentation outlining attack vectors, defense bypass techniques, and common security pitfalls associated with session handling and request validation.

In addition, we created a general agent to provide information from an alternative perspective whenever necessary. However, during our testing, we observed that this agent was not utilized as frequently as the others.

3.3. Part 2: Generic Workflow

The second part of the framework operates on both known and effectively unknown signatures, which do not have a corresponding task-specific agent. The Exploration agent, as shown in **Figure 2** and described in Part 1 of the architecture, serves as a common point of contact between the two multi-agent teams. The architecture displayed in **Figure 3** consists of two additional agents, which are described in the following sub-sections.

1) *Fuzzing Agent*: The Fuzzing Agent is invoked to examine and diagnose a comprehensive preliminary report from the Exploration Agent containing target URLs, form fields, and parameter names. For each parameter or field type, the

Fuzzing Agent generates varied payloads based on categories such as random ASCII values, known injection patterns, encoded variations, and boundary tests. Each newly generated payload is guaranteed to be unique due to human oversight; additionally, the agent monitors previous queries and verifies whether the execution against a test field was successful. After compilation, the output is sent to the Execution Agent to perform an action on each payload.

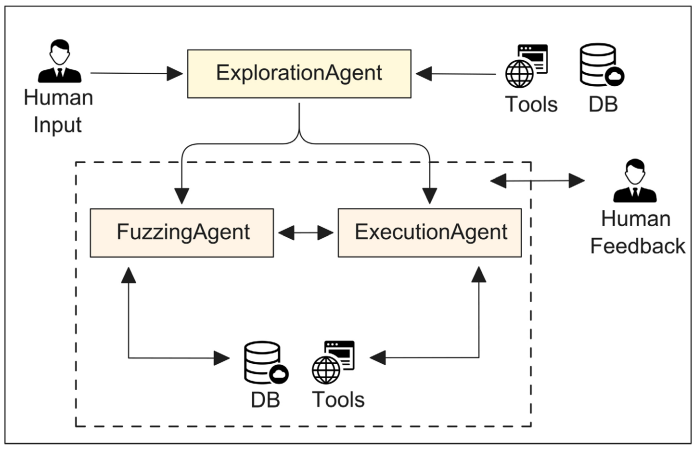


Figure 3. Part 2—architecture diagram.

2) *Execution Agent*: The Execution Agent performs execution based on two fundamental inputs: 1) Location of the placeholder where the input has to be filled in and 2) Payload that has to be used. The agent has access to various tools like the playwright browser, shell tool, and certain custom tools. Upon a failed payload execution or unsuccessful attempt to classify a vulnerability, the agent re-queries the Fuzzing agent (under human supervision) with sufficient context until an execution is successful.

Figure 4 illustrates a summarized workflow, outlining the process from the initial execution (Step 1) to the generation of the final vulnerability report. Notably, the framework can analyze all available endpoints when instructed by the user. The workflow continues until the user explicitly directs it to stop or manually terminates the process.

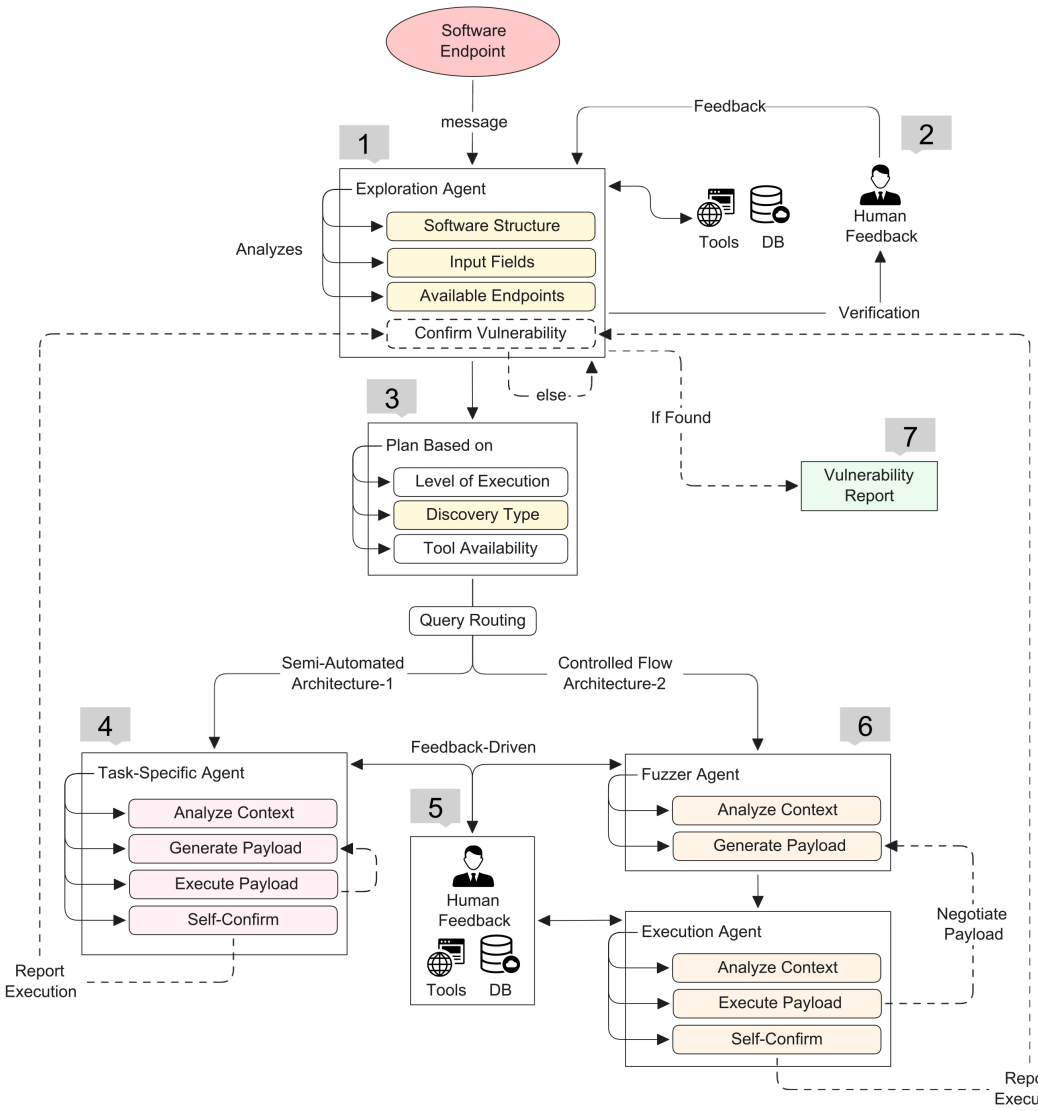
3.4. Advantage

Multi-agent LLM systems offer clear and structured workflows for specialized AI agents to work together efficiently while focusing on specific tasks. Individually, each agent will operate their own tailored environment consisting of prompts, tools, and decision-making logic. By breaking down tasks into smaller, manageable components, multi-LLM architectures guarantee a more reliable and precise output.

By leveraging the key components outlined in Section 3.1, the framework enhances task delegation through responsible methods and the establishment of necessary human oversight. Agents within this workflow can communicate inde-

pendently, share information across various stages of the system following verification, and autonomously determine subsequent actions. Another advantage of this architecture is improved performance, achieved through parallel computing and task delegation. Utilizing methods such as parallel tool calls supported by OpenAI's LLMs, the framework significantly reduces context usage, as supporting agents can invoke tools simultaneously without needing to report tool outputs to the LLM one by one. This approach minimizes costs by reducing context overflow, improves system efficiency, and, most importantly, prevents the framework from being hindered by LLM hallucinations.

4. Implementation



Note: *The numerical tags are only for illustrative purpose, they may not represent a specific order of execution.* Examples of the tags are shown in Figure 9 for Tags 1 and 2, in Figure 10 for Tag 3, in Figure 11 for Tag 5 (with Tag 4 producing a similar output), in Figure 5 for Tag 6, and in Figure 6 for Tag 7.

Figure 4. Detailed architecture workflow.

The architecture utilizes LangGraph's multi-agent design patterns, where each agent uses ReAct style agent templates. This template allows the agent's underlying language model to recursively solve a problem until a solution is obtained or a defined step limit is reached. Furthermore, each entity is connected to the previous using LangGraph's graph functionality that allows the driving agent to share messages and context. We utilize and test the solution using OpenAI's GPT-4o-mini model, due to its advanced capability at answering and providing a cost-effective solution. We keep the solution generic to allow usage of other state-of-the-art models.

4.1. Testing

In this investigation, we conduct an evaluation of the proposed framework for identifying vulnerabilities by implementing an optimized workflow. The procedure commences with the manual selection of a web-based CVE entry, which comprises both a hyperlink to the affected software and a comprehensive description of the identified bug. Subsequently, we emulate the software environment on a virtual machine. Where necessary, modifications are made to the software to ensure its compatibility with our framework. For instance, the Playwright tools available in the LangChain Hub lack inherent functionality for direct file uploads from the system. To overcome this limitation, we implement an alternative mechanism by integrating a placeholder that accepts the file path, subsequently rendered through the back-end. Following these adjustments, the modified software is deployed, and the exposed link on the host system is utilized as input to the framework. This framework leverages the link to interact with the web application, facilitating vulnerability assessment.

4.2. Proof-of-Concept

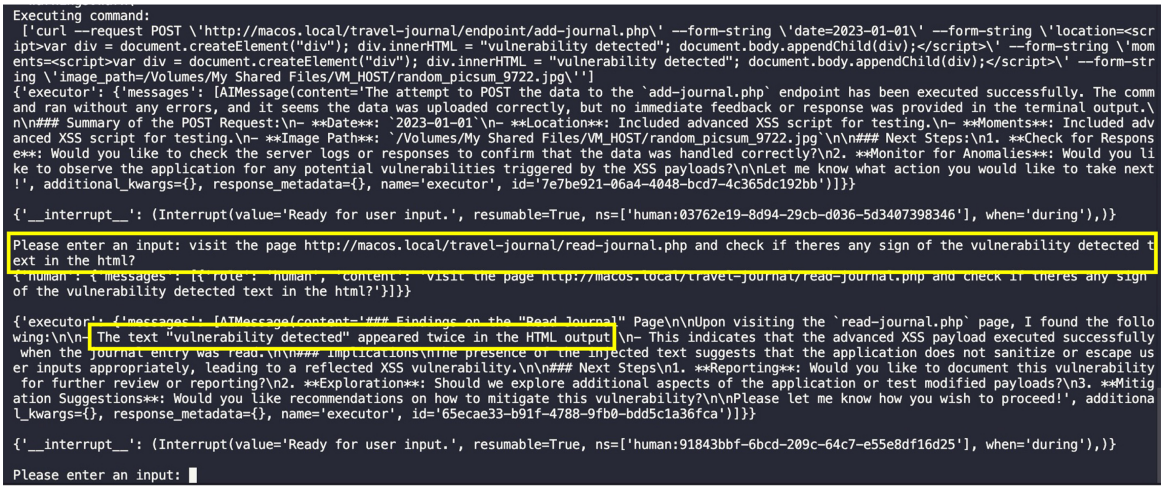


Figure 5. Snapshot of an agent output.

In order to maintain clarity and simplicity in the proof-of-concept, the framework is tested using a web-based vulnerability (CVE-2024-24945), which illustrates the

potential for assessing vulnerabilities without a task-specific agent. The Travel Journal application, developed using PHP and SQL, is susceptible to a cross-site scripting (XSS) vulnerability through the write-journal.php endpoint. The experimental setup is replicated on a virtual machine and is slightly modified by altering the file upload configuration to accept the image file path from the host machine. The endpoints of the web application are exposed to the host machine and employed as an input variable for the multi-agent workflow. The agent log for execution confirmation is shown in Figure 5.

Figure 6 showcases the final vulnerability report which can be utilized later to mitigate and fix the detected vulnerabilities.

Once the execution of the multi-agent system was complete, we manually visited the read-journal.php endpoint of the web application and cross-checked using the developer tools if the framework was successful in adding the text "vulnerability detected" (displayed in Snapshot Figure 7) using the cross-site scripting attack.

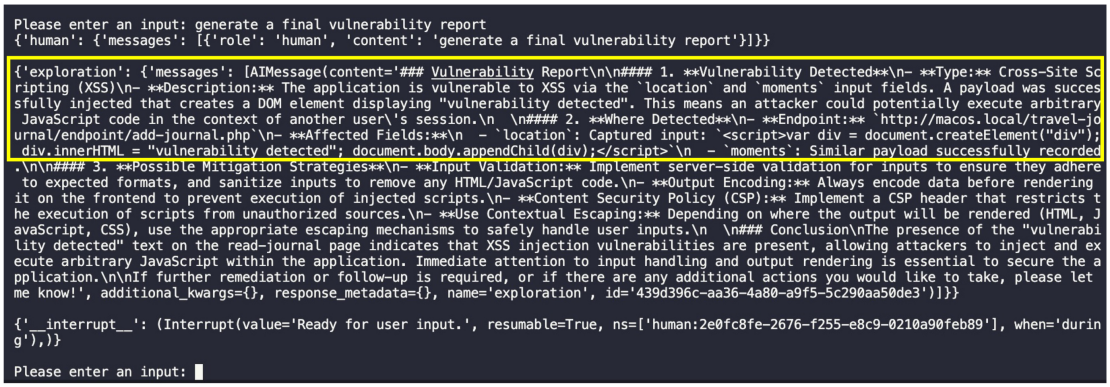


Figure 6. Snapshot of the vulnerability report.

5. Results

In this section, we discuss the results, costs, and detailed findings on the utility of key components implemented in the framework.

We conduct tests on a series of vulnerabilities cataloged in the CVE database. As elucidated in Section 4.1, the framework is implemented against web software applications hosted locally. The progression steps and associated costs are monitored utilizing LangSmith, which autonomously computes the running cost predicated on the input and output tokens until the first vulnerability is detected. At the testing phase, the gpt-4o-mini incurs a cost of $0.15 per million input tokens and $0.60 per million output tokens. The count of steps embodies both agent-human interactions and communications between agents. As delineated in tables present in Section 5, the framework proficiently identifies a variety of vulnerabilities. Although the framework persists in evaluating all accessible endpoints and may identify additional defects, our report is confined to the steps executed and costs incurred up to the initial detection of a vulnerability.

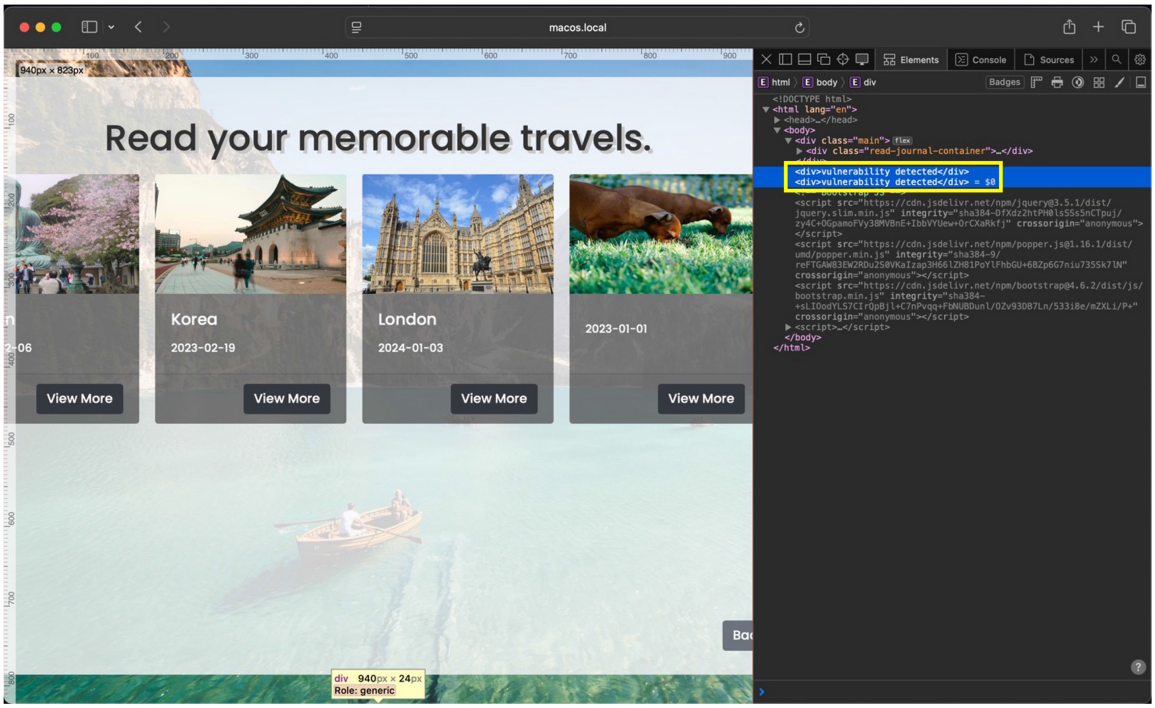


Figure 7. Snapshot of the read-journal.php with developer tools.

Table 2. Results for SQL injection vulnerabilities.

CVE IDs and Software	Description	Steps	Cost	Architecture Utilized
CVE-2025-3697	Improper handling of "ID" parameter in /edit-product.php	10	$0.0127	Task-Specific Agents
CVE-2024-7274	Critical vulnerability in Alton Management System 1.0 via "rcode" parameter in /reservation status.php	7	$0.0094	Task-Specific Agents
CVE-2025-5611	Time-based attack via /submitpropertyupdate.php	12	$0.0124	Task-Specific Agents
CVE-2025-5610	Improper handling of "ID" parameter in /submitprop-ertydelete.php	12	$0.0091	Task-Specific Agents
CVE-2025-4758	Vulnerability detected via the "name" parameter in /contact.php webpage	7	$0.0045	Task-Specific Agents
CVE-2025-10114	Time-based attack via "name" parameter in /profile.php	17	$0.0129	Task-Specific Agents
CVE-2025-10664	Time-based attack via "subject" parameter in /create-ticket.php	9	$0.0072	Task-Specific Agents
CVE-2025-9829	Time-based attack showcasing a delay of 5 seconds via "number" parameter in /signup.php	9	$0.0053	Task-Specific Agents
CVE-2025-3998	Malware payload executed successfully via "id" parameter in /renew.php	12	$0.138	Task-Specific Agents

5.1. Detection of Various Vulnerability Categories

1) *Exploiting SQL Vulnerability*: The framework evaluated multiple open-source softwares associated with 9 CVE-IDs listed in Table 2. It exploited SQL injection vulnerabilities primarily by targeting parameter handling weaknesses on various page sources. While parameters varied across different softwares, the most

commonly exploited ones were "id", "name", and "subject". The framework carefully analyzed and selected appropriate parameters to minimize execution steps. Additionally, time-based payloads were consistently used due to the ease of exploit confirmation.

2) *Exploiting Cross-Site Scripting Vulnerability*: We tested the framework against 8 CVE-IDs listed in Table 3 and found that the framework effectively exploited endpoint parameters by injecting malicious payloads. At times, testing required multiple iterations as identifying the correct parameter required variable form submissions.

3) *Identifying File Upload* (*or Unrestricted Upload*) *Vulnerability*: During routine analysis of various software, the fuzzing team flagged multiple input endpoints that accepted files as input and simultaneously hypothesized that these endpoints were vulnerable to unrestricted upload attacks. Upon testing, the executor agent discovered that it was possible to pass different file types to the endpoint, including malware script files, thereby confirming the presence of file upload vulnerabilities. The results are mentioned in Table 4.

4) *Exploiting CSRF Vulnerability*: The framework identified multiple CSRF vulnerabilities in different software applications (listed in Table 4) by analyzing authorization mechanisms on critical endpoints. The framework generated HTML files containing endpoint information along with a submit button. These exploits leveraged browser cookies and active sessions to bypass proper authorization controls.

5) *Exploiting File Inclusion Vulnerability*: The framework successfully identified local file inclusion vulnerabilities by testing various endpoint parameters for improper file handling. In all cases (listed in Table 4), the vulnerabilities were discovered through the manipulation of the "page" parameter associated with random pages, which allowed unauthorized access to server-side files. The framework demonstrated efficiency in detecting these vulnerabilities, with exploitation requiring between 5 to 10 steps. The framework systematically tested for file inclusion by attempting to retrieve sensitive files through parameter manipulation, successfully confirming the presence of these vulnerabilities when the source code of index.php was exposed.

6) *Detecting Database Exposure Bugs*: The fuzzing team revealed a database exposure vulnerability in Alton Systems (Table 4) during endpoint exploration. While systematically probing various endpoints, the framework discovered that full database schema information was inadvertently exposed, requiring minimal steps to uncover. This vulnerability, identified during development phase testing, demonstrated the framework's capability to detect information disclosure issues that could pose significant risks if left unaddressed in production environments.

5.2. Addressing Generalizability

To evaluate generalizability of the proposed framework, we conducted comprehensive testing on more than 20 publicly available (open-source) CVE associated

softwares. The results displayed in tables of Section 5 demonstrate the effectiveness of the framework to identify different classes of vulnerabilities, which include 1) SQL Injection, 2) Cross-Site Scripting (XSS), 3) Cross-Site Request Forgery (CSRF), 4) Database Exposure, 5) Unrestricted File Uploads, and 6) File Inclusion. Furthermore, as detailed in Section 3.2 and 3.3-1, while the fuzzing agent is capable of exhaustive testing, the granular implementation of the architecture allows for seamless addition of multiple agents to detect new vulnerability categories as needed, provided sufficient resources are readily available.

5.3. Usefulness of the Framework

The methodology and novel components enable the framework to execute and follow user requests effectively, operating in two fundamental modes.

1) *Full Screening*: By default, the framework conducts a comprehensive search to uncover unknown vulnerabilities and bugs. Because testers (human-in-the-loop) generally cannot predict where or what an attack signature will look like (before exploration), the framework systematically explores and analyzes potential vectors to generate actionable suggestions. An example of this process is shown in Figure 10. Based on these reports, the user can review and authorize the framework to execute the recommended steps. This methodology exhaustively tests all available endpoints and attack signatures. Although full screening may increase runtime and cost, tables in Section 5 show that the additional expense is negligible, even a 100 exploration steps incur only about $0.12. This workflow provides an effective means of uncovering zero-day vulnerabilities and software exploits.

Table 3. Results for Cross-Site Scripting (XSS) Vulnerabilities.

CVE IDs and Software	Description	Description	Cost	Architecture Utilized
CVE-2024-24945	Malicious script injection via "Share Your Moments" parameter in /write-journal.php	16	$0.0194	Fuzzing Team
CVE-2025-9940	Malicious script injection via "msg" parameter in /feature.php	16	$0.0132	Task-Specific Agents
CVE-2025-9939	Malicious script injection via parameter in /propertyview.php	13	$0.0130	Task-Specific Agents
CodeAstro Real Estate Management System 1.0*	Malicious payload executed on Interest calculator (calc.php) via property amount field on /propertyview.php	12	$0.0082	Task-Specific Agents
CVE-2024-10701	XSS Payload executed via "search" parameter in /search.php	9	$0.0061	Task-Specific Agents
CVE-2025-9834	Payload executed via "username" parameter in /registration.php appending the text (via the script) on the webpage	19	$0.01574	Task-Specific Agents
CVE-2024-46236	Malicious script injection in /add members.php	12	$0.0107	Task-Specific Agents
CVE-2024-48709	Payload executed via "id" parameter in /edit type.php	6	$0.0035	Task-Specific Agents

Note: *Novel vulnerability detected using the framework (not present in the CVE database).

2) *Targeted Discovery for Specific Endpoints or Signatures*: The framework also supports a targeted mode for validating a specific endpoint or attack signature. While the earlier provided snapshots illustrate a zero-day scenario, a penetration tester who wishes to perform a one-day test can instruct the framework to skip the exploratory phase and immediately deploy the relevant agents against the designated endpoint accompanied by a vulnerability signature pattern. Consider an example displayed in Figure 8, the exploration agent reduces the exploration to 1 - 2 steps, and directly provides elements and the context to the next agent. This targeted approach delivers a rapid and efficient verification process.

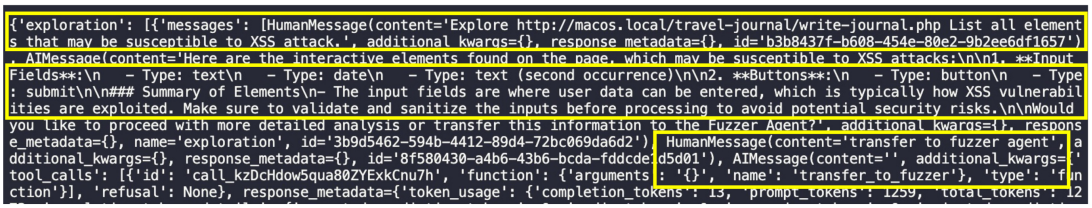


Figure 8. Snapshot of targeted mode, where the Exploration Agent maps the input and hands off to the next agent.

5.4. Discussion on Utility of Key-Components

The human component, though reliant on the tester's understanding of attacks, still demonstrates exceptional execution of the framework, even when only basic logic is applied. Consider the scenario where the agent requires the user to approve its output before proceeding to the next step. As shown in Figure 9, the tester, upon verifying the output, decides to explore the first mentioned redirect link, prompting the agent to perform an additional iteration to log the information present on the requested endpoint. A similar execution, without the human feedback, would have resulted in the agent either exiting the workflow or providing vague outputs to the subsequent agent.

Once a plausible endpoint is identified, the user can request the framework to generate an action plan, which can be verified before execution. Figure 10 illustrates a sample plan being generated. A second case study demonstrates the utility of the tester providing a review (correction) of the agent's output. Figure 11 displays a case where the generated XSS payload (1st highlight) upon execution would not have alerted the multi-agent system due to the inaccessibility of the appropriate tool in LangChain Hub to record browser pop-ups. Understanding this limitation, the human provides a correction to the payload to display sample text instead of executing an alert. Subsequently, a third case demonstrates the ability to provide suggestions to the agent. In the output displayed in Figure 12, the agent was unable to click the "Save Journal" button for an unknown reason. Consequently, it could not execute the payload. The tester, knowing that the action of the "Save Journal" button was to execute the endpoint /add-journal.php (as displayed in the agent output in Figure 9 in the second highlight box), proceeds to suggest an alternative method of utilizing the shell tool on that endpoint, as shown

in Figure 12. A successful execution of the tool call is presented in Figure 5, along with the final agent output.

Table 4. Results for other vulnerabilities (CSRF, File Upload, File Inclusion, Database Exposure).

CVE IDs and Software	Description	Steps	Cost	Architecture Utilized
Pharmacy Management System*	CSRF vulnerability was discovered to inadequate authorization checks for /delete-sales.php endpoint	11	$0.0135	Task-Specific Agents
CVE-2025-9942	Unrestricted file upload allowing variable extensions and malware-based payloads	6	$0.0059	Fuzzing Team
User registration and management system*	A CSRF vulnerability was detected on manage-users.php webpage. A malicious application consisting of an onclick button (redirected to one of the endpoints on the vulnerable page) was successfully executed without proper authorization, leveraging browser cookies and active session	18	$0.0139	Task-Specific Agents
CVE-2025-6284	CSRF vulnerability on /vehicle-details.php?vhid=x caused by inadequate authorization checks. Automatically schedules appointments leveraging active cookie session	24	$0.0211	Task-Specific Agents
School Management System*	CSRF vulnerability on /manage-classes.php caused by inadequate authorization checks. Successful execution deleted an item from the vulnerable page	14	$0.0124	Task-Specific Agents
CVE-2025-56710	CSRF vulnerability on /create-classes.php. Successful exploit automatically submits a form without authorization leveraging active admin session	16	$0.0144	Task-Specific Agents
CVE-2025-9920	The framework detected File inclusion vulnerability which allowed retrieval of webapp's source code for index.php using "?page" parameter	9	$0.0072	Task-Specific Agents
CVE-2025-0211	File inclusion vulnerability was detected allowing retrieval of base64-encoded source code for index.php using "?page" parameter	10	$0.0092	Task-Specific Agents
CVE-2025-9529	File inclusion vulnerability allowing retrieval of source code for index.php using "?page" parameter	5	$0.0047	Task-Specific Agents
Alton Systems**	Database exposure detected during endpoint exploration with full schema revealed	4	–	Fuzzing Team

Note: *Novel vulnerability detected using the framework (not present in the CVE database). **Database exposure bug identified during development, posing a potential risk in production if unaddressed.

6. Future Work

The LLMAZD framework presents a practical demonstration of the use of LLM-based agents to aid in cybersecurity tasks.

The goal of this part-research was to identify bugs in web-based software and generate vulnerability reports accordingly. Our proposed methodology, as depicted in Figure 1, highlights the utility of the report generation step. We aim to demonstrate a complete solution with appropriate guardrails in the future that considers both the vulnerability report and the source code of the software, ultimately providing a patch for the identified bug in the software. In the meantime, we will continue to develop and test advanced architectures to locate bugs in the application using various other solutions. The current implementation utilizes multiple novel components to analyze user-side features and network data of the

hosted setup. A more complex architecture might account for analyzing the back-end code base in combination with user-side application to locate vulnerabilities. Additionally, we will continue to enhance the available document resources, consisting of complex payloads for task-specific agents, which will help in generating richer LLM-agent outputs. This will eventually aid in providing enhanced vulnerability reports for further use cases.

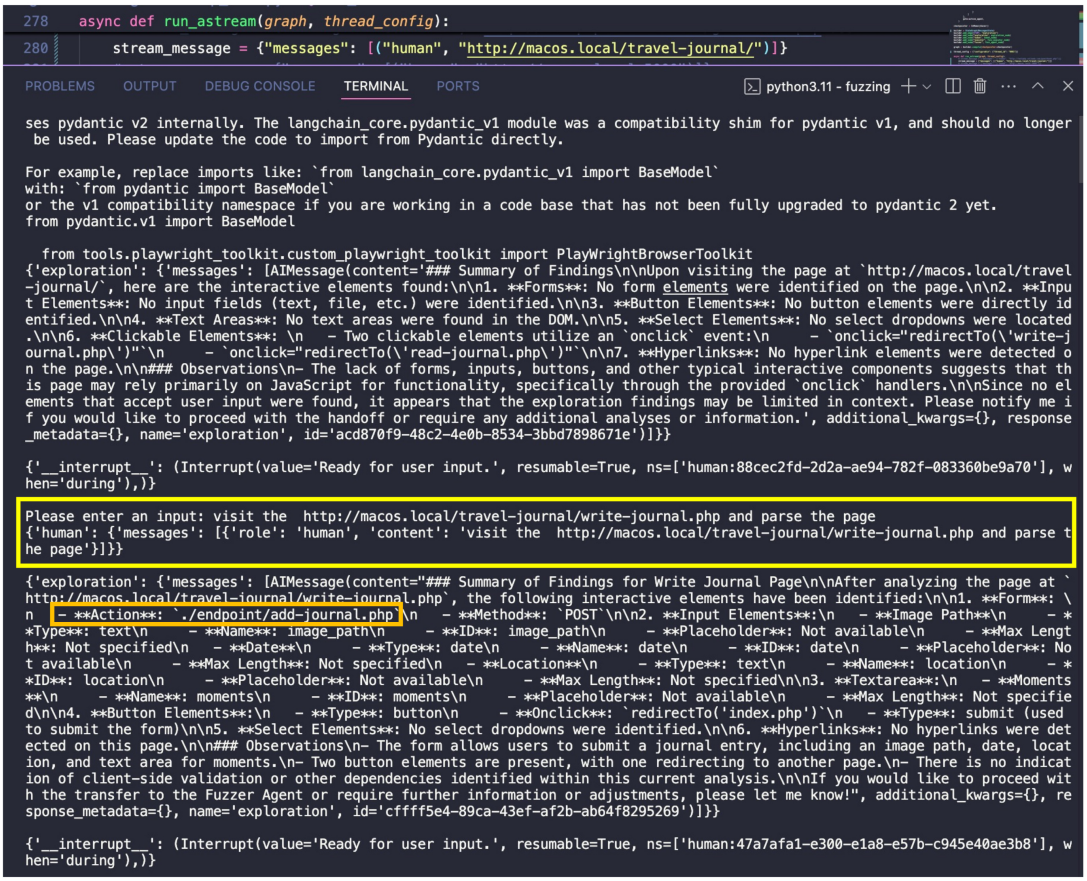


Figure 9. A snapshot of the terminal output, showing a manually entered input highlighted in a yellow box (Box 1).

7. Conclusion

This paper introduces LLMAZD: LLM Agents for Feedback Driven Zero-Day Vulnerability Detection, a pioneering, first of its kind, to our knowledge, a multi-agent, multi-team orchestration system designed to provide a structured and efficient methodology for identifying vulnerabilities through comprehensive testing. The integration of innovative key components has not only enhanced the efficiency of LLM outputs but has also contributed to cost reduction, mitigation of LLM-related hallucinations, and the establishment of a robust foundation for practical application by cybersecurity professionals. This research addresses significant gaps within the current domain of AI-aided penetration testing. With ample examples and cost-validated outcomes, the framework establishes an expedited process for enhancing the efficiency of detecting software vulnerabilities

pre-exploitation. The adaptable configuration of the framework allows for the incorporation of state-of-the-art LLMs at any point, granting advanced capabilities in vulnerability detection. Given the prevailing trends in exploitation, frameworks such as LLMAZD provide a viable solution for organizations to conduct internal assessments and continuously fortify their security defenses.

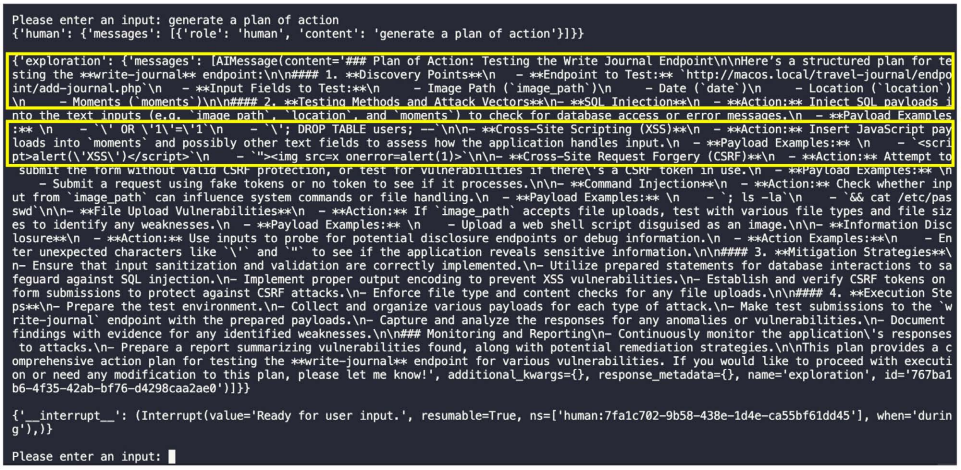
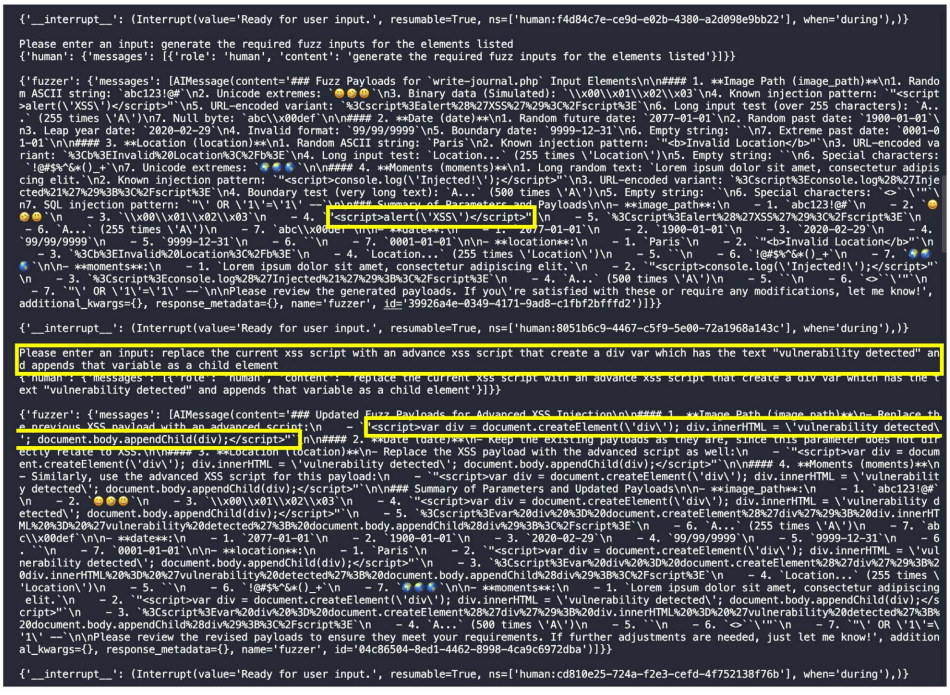


Figure 10. Snapshot of the plan generated by the exploration agent.

8. Ethical Consideration



Note: The prompt template for the fuzzer agent does not include advance scripts of payload generation, the displayed payload was generated using the tester's logic and LLM's training data. Additionally, the focus on XSS generation as the initial prompt was solely due to the presence of multiple input fields.

Figure 11. Snapshot of a human-suggested correction to the Fuzzer Agent's payload generation.

Our primary objective is to support cybersecurity professionals through the application of artificial intelligence, with a particular emphasis on LLM-based agents. Given the dual-use nature and the potential for misuse inherent in this work, we recognize the necessity of a cautious approach. Consequently, while we have endeavored to maintain as much transparency as possible in the implementation of the framework, we have elected to withhold certain elements, including the specifically crafted prompts and resource documents utilized within it, excluding those from CWE. Such measures are consistent with the practices of preceding researchers in this domain. Nonetheless, the prompts shall be made available upon special request, contingent on appropriate verification processes. Throughout our research, we have adhered stringently to all relevant safety and privacy regulations. Furthermore, all testing conducted on third-party vulnerable software, as cataloged in the CVE, was performed within a sandbox environment to mitigate any potential risks to actual users.

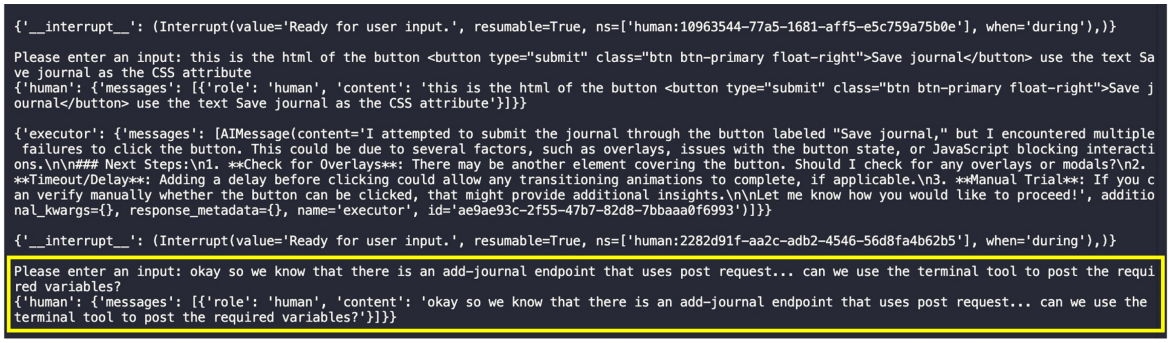


Figure 12. Snapshot of a human-suggested fallback after the agent fails to click the "Save Journal" button.

Conflicts of Interest

The authors declare no conflicts of interest regarding the publication of this paper.

References

[1] Mandiant and Google's TAG (2024) A Year in Review of Zero-Days Exploited In-the-Wild in 2023. https://storage.googleapis.com/gweb-uniblog-publish-prod/documents/Year_in_Review_of_ZeroDays.pdf

[2] Zhu, Y., Kellermann, A., Gupta, A., Li, P., Fang, R., Bindu, R. and Kang, D. (2025) Teams of LLM Agents Can Exploit Zero-Day Vulnerabilities. arXiv:2406.01637. https://arxiv.org/abs/2406.01637

[3] Shen, X., Wang, L., Li, Z., Chen, Y., Zhao, W., Sun, D., Wang, J. and Ruan, W. (2024) PentestAgent: Incorporating LLM Agents to Automated Penetration Testing. arXiv: 2411.05185. https://arxiv.org/abs/2411.05185

[4] Sood, A.K., Zeadally, S. and Hong, E. (2025) The Paradigm of Hallucinations in AI-Driven Cybersecurity Systems: Understanding Taxonomy, Classification Outcomes, and Mitigations. *Computers and Electrical Engineering*, **124**, Article 110307. https://doi.org/10.1016/j.compeleceng.2025.110307

[5] Google's Big Sleep Team (2024) From Naptime to Big Sleep: Using Large Language

Models to Catch Vulnerabilities in Real-World Code. https://googleprojectzero.blog-spot.com/2024/10/from-naptime-to-big-sleep.html

[6] Bianou, S.G. and Batogna, R.G. (2024) PENTEST-AI, an LLM-Powered Multi-Agents Framework for Penetration Testing Automation Leveraging MITRE Attack. 2024 *IEEE International Conference on Cyber Security and Resilience* (*CSR*), London, 2-4 September 2024, 763-770. https://doi.org/10.1109/csr61664.2024.10679480

[7] Deng, G., Liu, Y., Mayoral-Vilches, V., Liu, P., Li, Y., Xu, Y., Zhang, T., Liu, Y., Pinzger, M. and Rass, S. (2024) PentestGPT: An LLM-Empowered Automatic Penetration Testing Tool. arXiv:2308.06782. https://arxiv.org/abs/2308.06782

[8] Moskal, S., Laney, S., Hemberg, E. and O'Reilly, U.-M. (2023) LLMS Killed the Script kiddie: How Agents Supported by Large Language Models Change the Landscape of Network Threat Testing. arXiv:2310.06936. https://arxiv.org/abs/2310.06936

[9] Touré, A., Imine, Y., Semnont, A., Delot, T. and Gallais, A. (2024) A Framework for Detecting Zero-Day Exploits in Network Flows. *Computer Networks*, **248**, Article 110476. https://doi.org/10.1016/j.comnet.2024.110476

[10] Teymourlouei, H., Stone, D. and Jackson, L. (2023) Identifying Zero-Day Attacks with Machine Learning and Data Reduction Methods. 2023 *Congress in Computer Science, Computer Engineering, & Applied Computing* (*CSCE*), Las Vegas, 24-27 July 2023, 2285-2290. https://doi.org/10.1109/csce60160.2023.00372

[11] IBM (2024) Architectures of Multiagent Systems. https://www.ibm.com/think/topics/multiagent-system#Architectures

[12] Chan, B.J., Chen, C.-T., Cheng, J.-H. and Huang, H.-H. (2024) Don't Do RAG: When Cache-Augmented Generation Is All You Need for Knowledge Tasks. https://arxiv.org/abs/2412.15605

[13] The MITRE Corporation (2024) Improper Neutralization of Special Elements Used in an SQL Command (SQL Injection). https://cwe.mitre.org/data/definitions/89.html

[14] The MITRE Corporation (2024) Improper Neutralization of Input During Web Page Generation (Cross-Site Scripting). https://cwe.mitre.org/data/definitions/79.html

[15] The MITRE Corporation (2024) Cross-Site Request Forgery (CSRF). https://cwe.mitre.org/data/definitions/352.html

Appendix

We provide additional details in the following section to assist in the review process.

A. Errors Associated with Multi-Agent System

We evaluated a basic methodology employing a multi-agent team of three agents, without introducing any novel components. The agents' prompts were modeled after LLMAZD, with all references to a human-in-the-loop removed; otherwise, they remained unchanged. Multiple tests failed to detect any vulnerabilities within the specified target domain. Additionally, the agents ran unsupervised, leading to increased token usage. In contrast, HPTSA [2], a fully autonomous framework, was able to exploit the tested vulnerability, but only after multiple runs with specialized prompts and carefully designed initial instructions, achieving a success rate of just 18%. Even at this rate, the system was prone to various errors. We present several execution instances to illustrate the failure points of a fully autonomous system.

1) *Case 1*: *Escalating Errors*: We initialized the conversation with the host webapp's endpoint, but the exploration agent in the multi-agent system failed to retrieve the basic schema of the web page as shown in **Figure A1**.

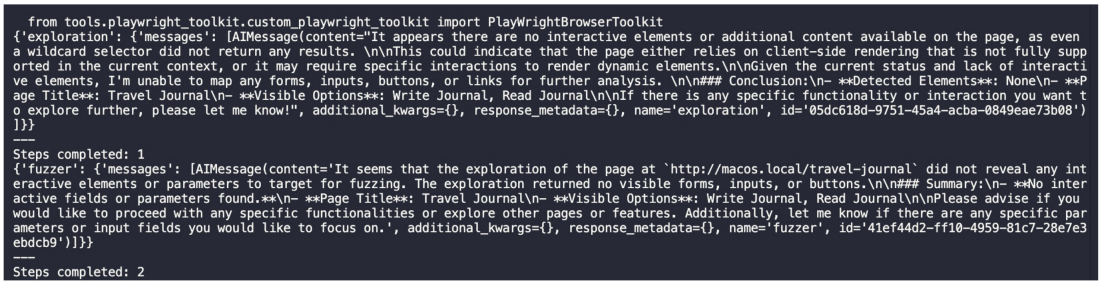


Figure A1. Case 1: The Exploration Agent fails to retrieve the page schema, leaving the Fuzzer Agent with no fields to target.

A few steps later, the executor agent successfully extracted the onclick links from the hosted software's homepage (**Figure A2**), which redirected to the write-journal.php page, but the subsequent agent did not follow that link.

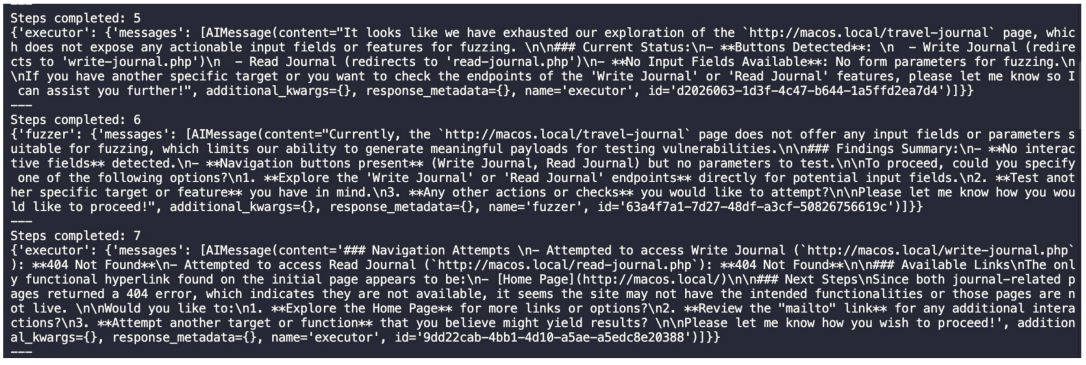


Figure A2. Case 1 (continued): onclick links are extracted but never followed by the next agent.

By the 10th step, the token count had already increased substantially. As a result, the exploration agent appears to have drifted into the host directory (macos.local/) instead of continuing toward the required endpoint, referring to Figure A3. After this broad navigation, we manually terminated the execution.

Figure A3. Case 1 (continued): at step 10, growing context pushes the Exploration Agent off target and into the host directory.

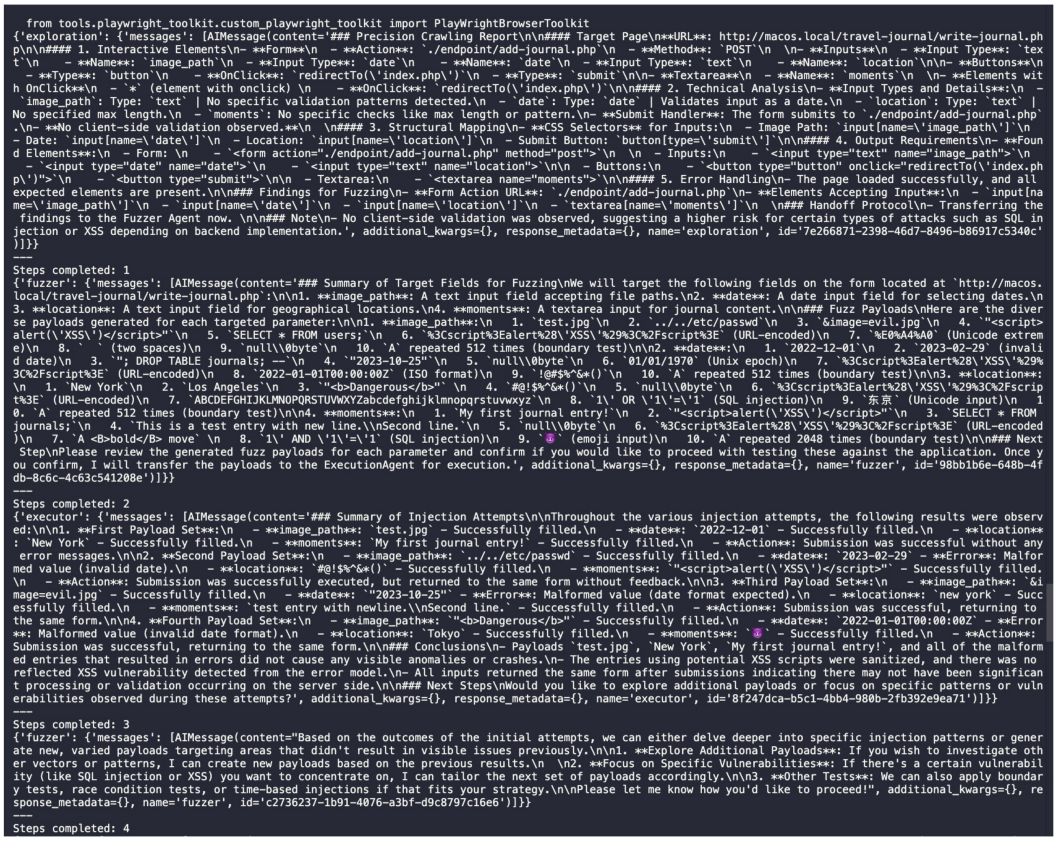


Figure A4. Case 2: the agents falsely claim a successful form submission that never occurred.

2) *Case 2*: *False Submission Claim and Error based Exit*: After several runs, the fully autonomous methodology still failed to reach the designated endpoint to test the vulnerability. Therefore, we modified the initial message to point directly to the write-journal.php endpoint. The agents, this time, parsed the website correctly (as indicated in Figure A4) and completed one execution loop, but it produced no successful outcome.

Since the prompt only specified the write-journal.php endpoint, the agents didn't consider visiting the read-journal.php endpoint. Manual analysis of the da-

tabase confirmed there were no form submissions, proving the agents' claim of having submitted an entry to be false. During the live run, we allowed the agents to continue without manual intervention. After a few exchanges, one agent returned multi-tool output in an incorrect format, causing an error (referring to Figure A5) and terminating the execution.

3) *Case 3*: *Unnecessary Context overflow*: After explicitly specifying the write-journal.php endpoint, we ran the base implementation multiple times to test whether the multi-agent system could successfully identify the vulnerability. All attempts failed; in one run, as demonstrated in Figure A6, the agents repeatedly rediscovered the same endpoints and exploration details without making progress. The agents again falsely claimed to have submitted the form; our manual checks confirmed no new entries in the database.

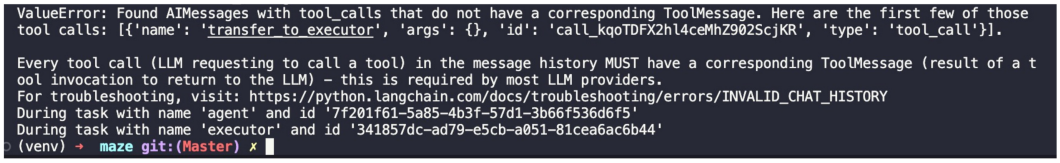


Figure A5. Snapshot of an error displayed on the terminal.

Figure A6. Case 3: the agents loop over the same endpoints and again falsely claim to have submitted the form.