Autonomy Comes with Costs: Detecting Denial-of-Service Vulnerabilities Caused by Resource Abusing in LLM-based Agents

Jiaqi Luo[†], *Jiarun Dai*[†], *Fengyu Liu*[†], *Songyang Peng*[†], *Youkun Shi*[†], *Tong Bu*[†], *Geng Hong*[†], *Xudong Pan*[†‡], *Yuan Zhang*[†]

[†]*Fudan University* [‡]*Shanghai Innovation Institute*

Abstract

LLM-based agents have recently attracted significant attention. By leveraging the semantic understanding capabilities of large language models (LLMs), these agents can autonomously perform complex tasks according to user requests, such as downloading files and summarizing content. However, the lack of comprehensive resource governance renders them susceptible to abuse, potentially leading to resource exhaustion and denial-of-service (DoS) conditions.

In this work, we present the first systematic security study of resource management in LLM-based agents. We identify three representative patterns of resource lifecycle management, each of which enables distinct avenues for DoS exploitation. Building on these insights, we propose `AgentDoS`, a novel directed grey-box fuzzing framework designed to detect DoS vulnerabilities arising from resource exhaustion. `AgentDoS` first analyzes the resource lifecycle within the agent and then leverages an LLM to generate functionality-specific seed prompts in natural language that drive the agent toward excessive resource consumption. We evaluated `AgentDoS` on 20 widely used open-source LLM-based agents and discovered 36 zero-day vulnerabilities affecting 16 agents, 15 of which have over 10,000 stars on GitHub. To date, 15 CVE IDs have been assigned for these vulnerabilities.

1 Introduction

LLM-based agents have recently attracted significant attention. Leveraging the powerful semantic understanding capabilities of large language models (LLMs), these agents can interpret natural language instructions and autonomously perform complex tasks, such as downloading files and summarizing content, in real-world scenarios [72, 74]. Mainstream platforms, such as OpenAI GPTs and Coze, currently host a wide variety of agents [10, 15], attracting millions of users [84].

Unlike conventional software systems with well-defined business logic, LLM-based agents autonomously execute complex, resource-intensive tasks from natural language instructions. However, these characteristics also make them highly vulnerable to **Denial-of-Service (DoS)** caused by resource exhaustion. To investigate the emerging resource management practices in agents and their associated risks, we present the first study of resource management in LLM-based agents. Our observations reveal three representative resource lifecycle patterns. First, during single-task execution, agents are often permitted to consume substantial resources to preserve autonomy in handling diverse operations. Although these resources are released upon task completion, a single malicious prompt (e.g., loading of a large file) can immediately deplete them. Second, while some developers impose limits on resources consumed within a single turn, certain resources persist across multiple turns to maintain conversational coherence. These resources may be cleared at the end of a session, making their lifecycle coincide with the chat session. An attacker can repeatedly interact within the same session, progressively accumulating retained data and eventually exhausting resources. Finally, some resources are never released once allocated. Attackers need only repeatedly interact with the agent to gradually consume all resources and induce a DoS condition.

Currently, fuzzing techniques have been widely applied to detecting DoS vulnerabilities [34, 36, 37, 47, 81], valued for their high precision and ability to generate proofs of concept (PoCs) for vulnerability verification. Specifically, fuzzers generate initial seeds that conform to the target application's input format and then mutate them (e.g., via bit-flip [1] strategies) until resource exhaustion is triggered. However, DoS in agents requires semantically precise prompts that both activate specific functionality (e.g., a download command) and induce heavy resource usage (e.g., retrieving a 50 GB file). Existing fuzzers cannot operate at this semantic level [49]. While AgentFuzz [49] makes the first attempt to generate prompts with functional semantics, its focus is limited to taint-style vulnerabilities, remaining ineffective in driving excessive resource consumption. Moreover, prior work has not considered the resource lifecycle in agents, specifically, whether resources must be exhausted within a single interaction or accumulated over multiple turns within the same chat session, resulting in potential false negatives.

To this end, we are motivated to develop a resource-lifecycle-based approach for detecting DoS vulnerabilities in LLM-based agents. The basic idea is to first identify data-storing instructions and determine their corresponding resource lifecycle, and then iteratively refine prompts and dispatch prompts to the agent using lifecycle-aware strategies (e.g., issuing them in a new chat session or within the same session) until a DoS condition is triggered. However, realizing this detection approach is non-trivial, given the complexity of agents. We summarize the key challenges as follows:

- **Challenge I: How to Identify the Lifecycle of Resources?** Agents often consume substantial resources, yet their management is rarely documented in detail, making it difficult to determine whether a given resource is released after a single chat or retained across multiple interactions as part of the conversational context.
- **Challenge II: How to Generate Prompts that Effectively Trigger Resource Abuse?** Triggering DoS vulnerabilities in LLM-based agents requires semantically precise prompts that not only activate specific functionalities (e.g., issuing a download command) but also induce substantial resource consumption (e.g., retrieving a 50 GB file). However, existing approaches fall short of generating such prompts, making it difficult to expose DoS vulnerabilities in agents.

In this paper, we propose a directed grey-box fuzzing (DGF) [33]-based approach, named `AgentDoS`. Our design is motivated by several key insights that directly address the aforementioned challenges. First, the implementation logic of agents provides valuable cues for distinguishing different resource lifecycles. Specifically, if a resource is allocated solely for handling a single prompt and is neither referenced by other variables nor retained in subsequent logic, its lifecycle is limited to that individual interaction. In contrast, certain resources may persist beyond a single turn. Second, agents rely on various components (e.g., tools) to process actions, and their functionalities are often further detailed in the system prompt. The class and method names, together with the source code of these components and the system prompt, reveal both their functional purpose and the semantics of resource-intensive operations. Moreover, high-quality prompts typically align with these semantics while simultaneously driving substantial resource consumption. Such information can therefore guide the generation of prompts that both activate specific functionalities and induce excessive resource usage.

Based on these insights, we design `AgentDoS` with two main phases. In the first stage, `AgentDoS` employs static analysis to locate resource-consuming operations and determine their corresponding lifecycles according to the agent's execution logic. It then conducts intra-procedural analysis to exclude operations unaffected by user input or strictly constrained, retaining only exploitable candidates. For each, `AgentDoS` extracts call chains and relevant code snippets for downstream analysis. In the second phase, `AgentDoS` parses the agent's system prompt and, together with the information extracted in the previous phase, leverages an LLM to generate functionality-specific seed prompts. Each seed is then executed and evaluated based on its semantic alignment with the target functionality as well as the resources it consumes. Two mutators then refine high-quality seeds from complementary perspectives, i.e., functional semantics and resource-intensive behavior, and dispatch them with lifecycle-aware strategies. Finally, `AgentDoS` monitors agent responsiveness to determine whether a DoS vulnerability is triggered.

We evaluated `AgentDoS` on 20 widely used open-source agent applications that provide web services, which are particularly susceptible to attacks due to their exposure and the potential impact of exploitation. As a result, `AgentDoS` achieved a resource lifecycle identification accuracy of 95.2% and successfully uncovered 36 zero-day DoS vulnerabilities, 33 of which had not been detected by existing state-of-the-art approaches [49], achieving 100% precision and 94.7% recall. These vulnerabilities affect 16 open-source agents, 15 of which have received more than 10k stars on GitHub. We promptly reported all findings to the respective developers. To date, 15 of them have been assigned CVE IDs.

We summarize the contributions of this work as follows:

- To the best of our knowledge, we present the first systematic security study of resource management in LLM-based agents, uncovering the root causes of resource exhaustion vulnerabilities in agent system design and implementation.
- We design and implement a novel detection tool, `AgentDoS`, which effectively identifies DoS vulnerabilities caused by resource abusing in real-world agent applications.
- We evaluate `AgentDoS` on 20 widely used LLM-based agent applications, where it successfully discovers 36 zero-day vulnerabilities. As of now, these vulnerabilities have been assigned 15 CVE IDs.

2 Problem Statement

In this section, we first provide an overview of LLM-based agents and introduce a taxonomy of resource lifecycles within agent systems (in §2.1). We then present a real-world DoS vulnerability caused by resource abuse and present why existing works are ineffective (in §2.2). Finally, we delve into the root causes of resource-abusing vulnerabilities (in §2.3) and introduce our threat model (in §2.4).

2.1 Background

First, we outline the background of LLM-based agents and their resource management strategies.

LLM-based Agent. The advancement of LLMs has accelerated the development of sophisticated LLM-based agents [3, 11, 18, 72]. These agents leverage the natural language understanding capabilities of LLMs to interpret user prompts

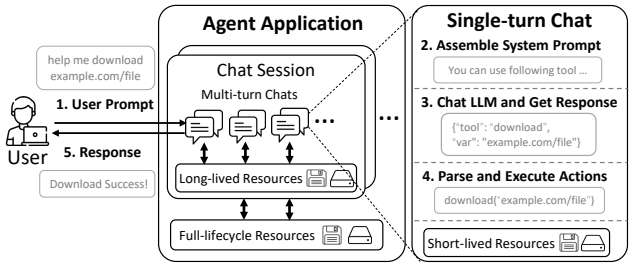

Figure 1: Simplified workflow of LLM-based agents.

and autonomously perform complex tasks that exceed the capabilities of traditional LLMs, such as web browsing, by integrating a suite of components (e.g., external tools).

Figure 1 illustrates a simplified workflow for a typical agent. (1) Initially, the user initiates a task request within a chat session, for instance, "help me download example.com/file." (2) The agent then augments this prompt with a predefined system prompt, which instructs the LLM on which components can be invoked and provides a brief description of them. (3) The combined prompt is sent to the LLM. (4) The agent parses the LLM's response to determine the planned action and executes it using the relevant component (e.g., a download tool). (5) Finally, the agent delivers the outcome of the executed action to the user. (6) The user may then continue in the same session to process previous outputs (e.g., summarizing the file) or start a new session for a different task.

Lifecycle of Resources in Agents. During execution, LLM-based agents frequently consume substantial system resources (e.g., memory, disk space). If these resources are not properly released after use, they can accumulate over time, leading to redundant artifacts and eventual resource exhaustion. However, to the best of our knowledge, there is currently no systematic analysis of resource management strategies in agent systems. To bridge this gap, we take an initial step toward understanding how LLM-based agents manage system resources, with a particular focus on the timing of resource release. Specifically, we selected 20 representative agent applications from popular open-source repositories on GitHub [14] and manually analyzed their source code and documentation (detailed in §6.2). Our investigation reveals that while many resources are promptly released after a single interaction, others persist beyond the scope of a single turn. In particular, we classify resource lifecycles in agents into three types based on their release timing:

1. *Short-lived resources* are allocated during a single-turn interaction and are subsequently released, either by garbage collection [13] or explicitly by the agent, once the interaction completes. For this type of resource, a malicious attacker must exhaust all resources within a single prompt.
2. *Long-lived resources* refer to resources that persist across multiple turns within the same chat session and are released only when the session terminates (e.g., upon reaching a predefined interaction limit). For such resources, an attacker can repeatedly issue interactions within the same session to progressively accumulate consumption until exhaustion.
3. *Full-lifecycle resources* refer to data that, once allocated, are never released and remain accessible for the entire lifespan of the agent system. For this type of resource, an attacker can exhaust resources through repeated interactions across multiple sessions.

2.2 Motivation Example

Then, we introduce the resource-abusing vulnerabilities caused by improper resource management and discuss why existing techniques fail to detect them in LLM-based agents.

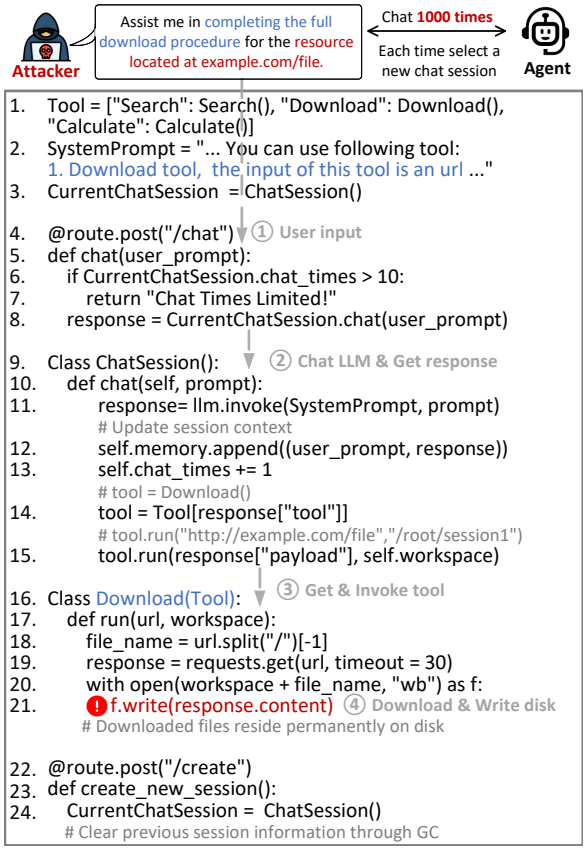

Figure 2: A real-world DoS vulnerability example from a popular open-source agent.

A Real-World Example. Figure 2 illustrates a simplified code snippet of a zero-day vulnerability we discovered (CVE-2025-4**90, anonymized for ethical reasons) on a mainstream open-source agent platform. Specifically, this agent is designed to autonomously download arbitrary files for tasks such as literature reviews. However, due to improper management of downloaded resources, attackers can exploit this feature to

launch DoS attacks by exhausting all disk space, rendering the agent's functionality unavailable.

Exploitation of Resource Abusing Vulnerability. To exploit this vulnerability, the attacker performs the following steps: ❶ The attacker issues a crafted prompt to the agent via its single-chat web API (lines 4–5). ❷ The agent forwards this prompt to the LLM (line 11), which plans the next action by selecting a tool from the system prompt. ❸ The agent parses the LLM's response and invokes the selected tool (lines 14–15). In this case, the `Download` tool with the parameter `example.com/file`. ❹ The tool retrieves the file and stores it locally (lines 19–21). ❺ While the agent enforces per-session resource limits (e.g., a download timeout at line 23 and a restriction on the number of turns per session at lines 6–7), files generated in previous sessions persist on disk (lines 23–24). An attacker can therefore repeatedly create new sessions (lines 22–23) and replay the above steps, progressively accumulating files until system resources are exhausted.

Limitations of Existing Detection Methods. Although DoS vulnerability detection has been extensively studied, we found that existing techniques are ineffective to identifying the above vulnerability. First, most existing approaches [45, 64, 69, 78] primarily target CPU exhaustion and thus cannot address memory and disk resource exhaustion in agents. Second, while a few studies have explored memory exhaustion [36, 47, 48, 81] and memory leaks [34, 37, 65, 71], agent systems typically rely on customized frameworks rather than well-defined ecosystems (e.g., beans in Spring Boot) to manage resource lifecycles. As a result, existing work provides insufficient guidance for identifying resource lifecycles within agents. More importantly, these approaches are unable to generate semantically valid prompts. Finally, the most relevant work is `AgentFuzz` [49], but it focuses exclusively on taint-style vulnerabilities and fails to induce excessive resource consumption. As shown in our evaluation (detailed in §6.4), `AgentFuzz` can detect only 7.9% of the resource-abusing vulnerabilities identified in our benchmark.

2.3 Problem Understanding

To address this gap, we delve into the root causes of resource-abusing vulnerabilities. We found that the main reasons for the resource abuse problem in the agent are twofold:

- *Mismanaged Resource Lifecycle:* Agents typically retain conversational context across multiple interactions to preserve coherence. Since they autonomously execute tasks based on user input, developers cannot predict which intermediate resources may be required later. To avoid disrupting functionality, developers therefore tend to preserve most intermediate data. For instance, as shown in Figure 2, files downloaded during a chat session are retained indefinitely and never deleted. This design flaw allows attackers to craft prompts that trigger unnecessary allocations and repeated interactions, causing redundant resources to accumulate over time and eventually exhausting system resources.
- *Lack of Resource Size Limits:* A straightforward defense against resource exhaustion is to restrict the amount of resources an agent can consume. However, existing size limits are often incomplete. For instance, many agents provide a download tool that allows the LLM to fetch arbitrary files with configurable parameters such as timeout and file size. An attacker can exploit this functionality by downloading files that exceed the remaining disk space, directly exhausting storage. Moreover, as shown in Figure 2, a 30-second timeout implicitly caps the size of a single download at approximately 300 MB. Yet, without proper checks on overall resource availability, an attacker can repeatedly download smaller files across multiple sessions, bypassing per-interaction limits and gradually consuming all available disk space. Conversely, overly strict limits can undermine the agent's ability to autonomously complete tasks. For example, during a literature review task, an agent may fail to download all necessary documents under restrictive resource constraints, preventing task completion.

2.4 Threat Model

In our threat model, we assume that both the agents and their developers are benign and that the agents' execution environments are not compromised. The adversary is a remote attacker who can interact with the target agent under normal usage conditions, such as by issuing prompts through a publicly accessible web interface. This interaction pattern is common in mainstream LLM-based agent systems [3, 11, 18]. The attack is executed by sending specially crafted malicious prompts to the agent. When the agent processes these prompts, it is induced to consume excessive system resources (e.g., memory or disk), ultimately leading to a Denial-of-Service (DoS) condition.

3 Overall Ideas

In this section, we first discuss the challenges of identifying resource-abusing vulnerabilities in LLM-based agents and present our key insights (in §3.1). We then provide an overview of our proposed framework and demonstrate its application to the motivating example (in §3.2).

3.1 Challenges & Key Insights

As previously discussed, the resource lifecycle plays a critical role in shaping the attacker's exploitation strategy. Specifically, for resources released after a single turn, an attacker must maximize consumption within a single prompt. Conversely, for resources that persist across multiple turns, an attacker can incrementally increase consumption over successive interactions within a session to eventually cause exhaus-

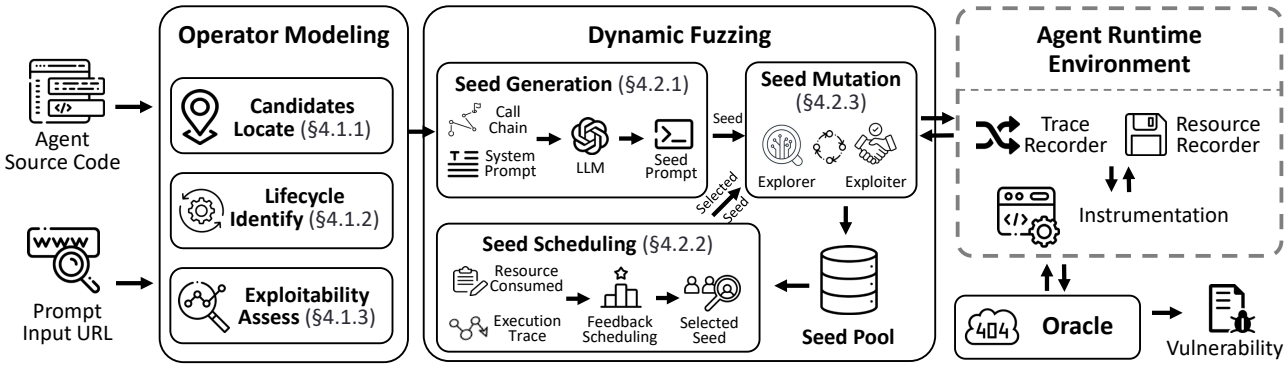

Figure 3: Architecture of AgentDoS.

tion. However, exploiting such vulnerabilities presents two key challenges stemming from complex resource lifecycles and the unique inputs of LLM-based agents.

Challenge I: How to Identify the Lifecycle of Resources? During execution, LLM-based agents interact with various system resources, such as memory and disk storage. Unlike traditional web applications, which manage resource lifecycles through well-defined frameworks, agents often rely on customized agent frameworks that are rarely documented in detail. As a result, it is difficult to determine whether a specific resource is released after a single chat (i.e., a short-lived resource), retained across multiple interactions and released only when the chat session ends (i.e., a long-lived resource), or persisted even after the chat session is terminated (i.e., a full-lifecycle resource).

Key Insight I: Agent Implementation Logic-Driven Lifecycle Determination. Considering that our resource lifecycle classification in §2.1 is closely related to the agent's business logic, we leverage the agent's handling of single conversations and its chat session deletion logic to assist in determining resource lifecycles. Specifically, if a resource is allocated only during a single conversation, is not referenced by other variables, and is explicitly released in subsequent logic, we classify it as a short-lived resource. In contrast, certain resources may persist beyond a single conversation. By inspecting the deletion logic of chat sessions, we can determine whether such resources are released when the session ends. If they are released, they are classified as long-lived resources; otherwise, we consider them to remain resident in the agent, constituting full-lifecycle resources.

Challenge II: How to Generate Prompts that Effectively Trigger Resource Abuse? To expose DoS vulnerabilities in agents, a prompt must encode precise semantic intent that both activates the vulnerable component (e.g., issuing a command such as "`Download ...`") and simultaneously drives excessive resource consumption (e.g., requesting the download of a 50GB file). However, existing DoS detection approaches [36, 71, 81] primarily rely on low-level mutations (e.g., bit flips [1]), which cannot capture or manipulate the high-level semantics required in natural language prompts. Although AgentFuzz [49] makes a first attempt to generate prompts with functional semantics, its scope is limited to taint-style vulnerabilities, and it remains ineffective in inducing substantial resource consumption.

Key Insight II: LLM-Assisted Resource-Consuming Prompt Generation. The key insight is that component class and method names, together with the system prompt and surrounding code, convey both functional intent and the semantics of resource-intensive operations. As shown in Figure 2, the component name `"Download"` and its description explicitly indicate its functionality, while the enclosing function suggests that exhausting disk resources requires downloading a large file. Moreover, the semantic similarity between a prompt and the vulnerable component, combined with observed runtime resource consumption, reflects the prompt's potential to trigger DoS. We therefore leverage LLMs' natural language understanding to interpret these semantics and iteratively generate high-quality prompts that both activate vulnerable components and maximize resource usage.

3.2 Approach Overview

Drawing on the above insights, we propose AgentDoS, a novel directed grey-box fuzzing approach for efficiently detecting DoS vulnerabilities caused by resource abusing in LLM-based agent. As illustrated in Figure 3, AgentDoS consists of two stages: operator modeling and dynamic fuzzing.

In the first stage, we apply static analysis to identify resource-consuming sink callsites and determine their resource lifecycles. We use intra-procedural taint analysis to filter out operations that are not affected by user input or are subject to strict resource limits. For each remaining potentially vulnerable sink, we extract the associated call chains and enclose function code snippets to provide rich contextual information for downstream analysis.

In the second stage, we parse the agent's system prompt and, together with the extracted call chains, use an LLM to generate functionality-specific seed prompts. Each seed is ex-

ecuted and stored in a seed pool, where it is evaluated on functionality relevance, resource consumption, and selection frequency. The top-scoring seed is selected for mutation, and two complementary mutators iteratively refine it from functional and resource-intensive perspectives. Mutated prompts are dispatched according to the sink's lifecycle: short-lived and full-lifecycle resources are fuzzed across sessions, whereas long-lived resources are fuzzed within the same session. AgentDoS continuously monitors the agent's web interface to detect DoS conditions. Upon detection, it outputs the final prompt as the exploit (EXP) for short-lived resources, and the complete dialogue sequence for long-lived and full-lifecycle resources.

Here, we illustrate how AgentDoS detects the vulnerability in the example shown in Figure 2. First, AgentDoS locates the disk write operation (line 21). By performing forward control-flow analysis from this sink callsite, it determines that the written content is never deleted in subsequent logic. Next, AgentDoS examines the logic for deleting the current chat session (lines 22–24) and find that the written content also persists after session deletion. Therefore, we classify it as a full-lifecycle resource. Further intra-procedural analysis reveals that the written content is controllable through the function input and lacks size checks, indicating a potentially vulnerable operator. In the dynamic fuzzing stage, AgentDoS extracts the system prompt and generates an initial seed. Since the resource is full-lifecycle, each prompt is dispatched through a new chat session. Based on execution feedback, AgentDoS determines whether the sink has been triggered and selectively applies different mutators to iteratively refine the prompt, guiding it toward activating the vulnerable component and driving excessive resource consumption. This process is repeated iteratively until the agent eventually becomes unresponsive, at which point the vulnerability is confirmed.

4 AgentDoS Design

In this section, we present the design of AgentDoS. It consists of two main modules. The *Vulnerable Operator Modeling* module is responsible for identifying all resource-consuming operators, analyzing their lifecycles, and selecting those that are potentially vulnerable (in §4.1). The *Dynamic Fuzzing* module manages the seed generation, scheduling, and mutation processes to detect these vulnerabilities (in §4.2).

4.1 Vulnerable Operator Modeling

In this phase, AgentDoS leverages static analysis to first locate resource-consuming operators (in §4.1.1) and determine the lifecycle of the resources they manipulate (in §4.1.2), and finally assess the exploitability of these operators (in §4.1.3).

4.1.1 Vulnerability Candidates Locating

AgentDoS primarily focuses on two types of resources: memory and disk. Regarding memory resources, motivated by prior work [48, 81], we focus on two key factors for exploiting data storage to induce resource exhaustion. First, data structures used to store injected data can grow without bound, requiring container types (e.g., lists, dictionaries, sets) that hold references to user-controlled inputs. We thus consider Python's built-in containers and their insertion operations as defined in the official documentation [20]. Second, previous studies [7, 8, 23, 32] report that certain parsing operators, such as `BeautifulSoup` [21], may cause excessive memory consumption when handling deeply nested HTML structures. Despite official guidelines [5, 6] addressing these issues, many developers overlook them, so we include such scenarios in our analysis. For disk resources, we focus primarily on disk write operations, as they directly increase disk space consumption. We list all predefined sinks in Table 5 of Appendix A.

Subsequently, we leverage static analysis to locate all sink callsites. Notably, if the operator manipulates a class member variable, we trace back to the location of its caller rather than the operator itself to obtain a more accurate context. For example, in the case shown in Figure 2, where `self.memory.append` is invoked at line 12, we attribute the sink callsite to its caller's location at line 8, rather than to the operator's own location. As discussed in §3.1, class and method names often convey semantic cues about functionality, and surrounding code snippets frequently reflect patterns indicative of high resource usage, such as repetitive or large-scale data operations. To capture these, we perform iterative backward traversal from each sink, enumerating all possible call paths until no further callers exist in the call graph. Finally, we extract the function containing the sink callsite as the associated code snippet for analysis. To mitigate the limitations of static analysis, particularly the difficulty of resolving implicit calls, we adopt the rule set proposed by LLMSmith [50], which is designed to enhance the precision and completeness of control flow graphs (CFGs) extraction in agents.

4.1.2 Resource Lifecycle Identifying

After locating the sink callsite, the next step is to determine the lifecycle of the resource it manipulates. As outlined in §3.1, we leverage the agent's logic for handling single-turn conversations and deleting chat sessions to aid in resource lifecycle classification. Specifically, we first manually identify the API entry points responsible for processing single-turn chats and for deleting chat sessions, and then construct forward CFGs starting from these entry points.

For memory resources, we analyze the scope of the variable associated with the operator. If the variable is initialized within the same function, i.e., its scope is limited to the current function, we classify it as a short-lived resource, as it will be automatically reclaimed by the GC once the function

Table 1: List of vulnerable operator features.

Name	Description
Operator	The name of the potentially vulnerable operator
Location	The source code location (file location and line number)
Type	The type of resource manipulated by the operator (Memory or Disk)
Lifecycle	The lifecycle of resource (Short-lived, Long-lived, or Full-lifecycle)
Snippet	The code snippet associated with the operator
Call Chain	The call chain to the operator
Controllable	Whether it is potentially influenced by user input (True or False)
Constraints	Whether capacity limitations exist (True or False)

scope ends. If the variable is referenced by a global variable, we further analyze the CFG of the chat session deletion process. Specifically, if the global variable is reassigned during session deletion, we classify the resource as long-lived, since the reassignment removes the reference and enables GC to reclaim the memory. In contrast, if the variable is neither locally scoped nor reassigned during session deletion, we categorize it as a full-lifecycle resource, as it may persist throughout the entire agent lifecycle unless explicitly cleared.

For disk resources, once the operator location has been identified, we extract the file path being written to and analyze its subsequent CFG. If a deletion instruction targeting this file path is found within the subsequent CFG, the resource is classified as short-lived, since it is released immediately after the interaction. If no such deletion occurs within the subsequent CFG, we further analyze the chat-session deletion logic. If the file is removed upon session termination, the resource is classified as long-lived, as it persists throughout the session but is released when the session ends. If the disk resource is never deleted, we categorize it as a full-lifecycle resource, which persists beyond the entire agent lifecycle unless explicitly cleaned up by the developer. Finally, due to the limitations of static analysis (e.g., dynamic path construction), we conduct manual dynamic validation for cases where the resource lifecycle cannot be determined statically.

4.1.3 Exploitability Assessing

To reduce the overhead of subsequent dynamic analysis, we perform a static exploitability assessment. This assessment consists of two key dimensions: (1) whether the operator can be influenced by user input, and (2) whether the resource usage is subject to explicit size constraints. To improve scalability, we employ intra-procedural taint analysis to filter out operators that are not exploitable.

Specifically, for short-lived in-memory resources, we adopt different heuristics based on the type of operator. For parser-like operators (e.g., `BeautifulSoup`), if any function argument can be propagated into the operator, we consider it potentially exploitable. For append-like operations (e.g., `list.append`), we assess whether they reside inside loops whose iteration count depends on user input [48]; if so, we flag them as posing a potential resource-exhaustion risk. For long-lived and full-lifecycle in-memory resources, we conservatively assume potential vulnerability if any user input can propagate into the operator, regardless of structural conditions. For disk resources, we mark an operator as potentially exploitable if any function input can influence the data being written to disk. To identify explicit bounds on resource usage for size-expandable data, we conduct a backward analysis from each operator to locate conditional checks on size-related variables. If these checks restrict the sink's execution, we consider them as evidence of an enforced size limit.

Finally, in this stage, `AgentDoS` collects the information summarized in Table 1. `AgentDoS` retain only those operators that are potentially controllable by the user and impose no explicit constraints on data size (i.e., `Controllable` = True and `Constraints` = False).

4.2 Dynamic Fuzzing

In this phase, `AgentDoS` first generates functionality-specific initial seeds (in §4.2.1), then evaluates their quality based on execution feedback (in §4.2.2), and subsequently iteratively mutates the high-quality seeds until a DoS condition is triggered (in §4.2.3).

4.2.1 Seed Generation

PROMPT 1. Seed Generation Prompt.

To guide the agent in invoking the target component: **First**, infer the component's functionality from the class and function names in the call chain. **Second**, create a natural language prompt whose semantic intent aligns with that functionality.
@EXAMPLE:
INPUT:
<call chain>: calculator→result.append()
<system prompt>: You can use the following tools: 1. **Calculator**: You can input a mathematical expression as a string parameter, and the tool will return the computed result. 2. ...
OUTPUT:
<prompt>: Please use the calculator to evaluate the following expression: 3 * (4 + 5).

As discussed in §3.1, the system prompt provides crucial context enabling LLMs to infer component semantics. However, statically extracting this prompt is challenging due to its fragmented storage across multiple files and a lack of structural markers. For example, in Langflow [17], the agent's role and component descriptions reside in separate modules and are dynamically combined just before the LLM call, complicating static reconstruction. To overcome this, we capture the system prompt dynamically at runtime by hooking the third-party LLM invocation API (e.g., OpenAI API [2]). We trigger a full query by sending a fixed input (i.e., `"hi"`) and record the API parameters, extracting the system role field that contains the assembled system prompt. Since the system prompt does not change with input, it only needs to be extracted once at the start of fuzzing for each agent.

Subsequently, we provide the extracted call chains and system prompt to the LLM for seed prompt generation. Following [49], we adopt a one-shot learning [67] strategy with Chain-of-Thought (CoT) [70] reasoning to guide the LLM in producing seed prompts. Specifically, we first guide the LLM through step-by-step reasoning and supply an example consisting of a call chain leading to a known sink, the corresponding system prompt, a user prompt that successfully triggers the sink, and an explanation linking the prompt to the call chain and contextual information. Through this process, the LLM learns to infer semantic intent from new call chains and system prompts, and is able to generate functionality-specific seed prompts that are likely to activate vulnerable components. An example prompt is shown in PROMPT 1.

4.2.2 Seed Scheduling

Feedback Collection. As noted in §3.1, a successful DoS prompt must simultaneously encode component invocation and resource consumption semantics. Accordingly, we evaluate the quality of each seed from these two perspectives.

1) Functionality Score. Similar to [49], we evaluate the functional semantics of seed prompts using both CFG distance and LLM-based assessment. Specifically, we record the execution trace and compute its CFG distance to the target sink. Additionally, we adopt the LLM-as-a-Judge paradigm [85] to assess the semantic similarity between the prompt and the intended functionality of the component. The LLM is provided with the seed prompt, the execution trace, which contains rich semantic signals reflecting the tools invoked and actions taken by the agent, as well as the component's system prompt and call chain. It assigns scores from 0 to 5 across three dimensions: clarity, conciseness, and functional consistency. The sum of these scores serves as the seed's semantic score, with higher values indicating stronger alignment with the target component. The scoring prompt is illustrated in PROMPT 2. Finally, the functionality score is computed as:

$$F_s = x^{-k} + L_s \tag{1}$$

where x is the shortest distance from the execution trace to the sink callsite, k is the hyper-parameter that adjusts the weight, and L_s denotes the semantic score assigned by the LLM.

PROMPT 2. Functionality Score Prompt.

Your task is to score user prompts based on how well their semantic intent aligns with the target component.
For each prompt, assess the following three aspects and assign a score from 1 (poor) to 5 (excellent):
1.Clarity: Is the prompt presented in a straightforward and precise manner, making it easy to understand?
2.Conciseness: Is the prompt formulated succinctly, while avoiding unnecessary redundancy or irrelevant details?
3.Relevance: Is the semantic intent of the prompt closely related to the intended component or function?

2) Resource Consumed Score. At this stage, we leverage the resource usage incurred by the agent during prompt execution to evaluate the resource consumption semantics of a prompt. Considering that different resource lifecycles lead to exhaustion through different mechanisms, we adopt tailored evaluation strategies. For short-lived resources, exhaustion typically occurs during prompt execution. Hence, we measure the *peak resource usage* observed during the execution of the agent. For long-lived and full-lifecycle resources, exhaustion tends to result from persistent accumulation across multiple prompts. Therefore, we evaluate the *residual resource consumption* after the completion of a single chat session, which reflects the degree to which resources are retained by the agent beyond prompt execution.

The resource consumed score is calculated as follows:

$$R_s = \begin{cases} (m_m - m_i) \;/\; m_t \;*\; 100 & \text{if } \textit{short-lived resource} \\ (m_e - m_i) \;/\; m_t \;*\; 100 & \text{otherwise} \end{cases} \tag{2}$$

where m_m denotes the maximum amount of resources consumed on the agent server during the execution of the user request, while m_i represents the initial resource usage prior to the chat. m_e captures the resource usage after the prompt has been executed. m_t refers to the total amount of system resources. For each operator that manipulates a specific type of resource, we track and record the usage of the corresponding resource type accordingly.

Seed Selection. This step selects high-quality seeds for fuzzing the LLM-based agent. Generally, `AgentDoS` favors seeds that either reach targets more effectively or trigger higher memory consumption. Furthermore, to avoid premature convergence to suboptimal local solutions, we introduce a penalty term that reduces the score of seeds that have been selected frequently. The final score is computed as follows:

$$S_s = \alpha F_s + \beta R_s - N \tag{3}$$

where N is the number of times the seed has been selected and α and β are hyperparameters that adjust the weights of these factors. We then select the highest-scoring seed for the next round of mutation.

4.2.3 Seed Mutation

A seed prompt may fail to invoke the target component or lack sufficient resource consumption semantics, thereby preventing the successful triggering of a DoS condition. To address this limitation, we propose a reflexion-based mechanism [62] that iteratively mutates and refines prompts to narrow the semantic gap between the prompt and both the target component's functionality and its associated resource consumption behaviors. This mechanism incorporates contextual feedback from previous executions, which is provided to the language model in subsequent iterations. Such feedback enables the model to efficiently learn from prior failures and reinforce

successful patterns, ultimately leading to the generation of high-quality seed prompts. Building upon this idea, AgentDoS maintains an independent history for each seed, preserving a dedicated memory of all mutation attempts associated with that seed. This memory records key information, including the newly generated prompt, the corresponding reasoning process, the execution trace, and the feedback score obtained from prior iterations. To further enhance the mutation process, we introduce two specialized mutators:

Explorer. By default, we apply this mutator to iteratively refine the selected prompt until it successfully reaches the target sink. Given that the primary goal of the exploration phase is to guide the prompt toward invoking the intended component, we provide the LLM with seed memory enriched with component-relevant semantics. This memory includes the associated call chain, system prompt, previously generated prompts, execution traces, and functionality scores. Similar to [49], the LLM is guided to refine its reasoning through structured, step-by-step inference. The LLM continues to iteratively refine the prompt and update the seed memory, progressively improving the quality of future mutations. An example prompt of explorer is provided in PROMPT 3.

PROMPT 3. Explorer Prompt.

Your task is to iteratively refine user prompts by deeply analyzing the agent's behavior.
First, evaluate the seed prompt's intent using its execution trace, and infer the target component's functionality from the call chain and system prompt.
Second, pinpoint which prompt segment caused the agent to invoke an incorrect component.
Third, use the history of generated prompts to inform your reasoning and avoid repeating previous errors.
Finally, revise the prompt to better align its semantics with the intended call chain and system prompt.

Exploiter. If a selected seed prompt successfully triggers the target sink, AgentDoS transitions to the mutator phase, which is designed to further optimize the prompt's resource consumption semantics. In this stage, the LLM is provided with a resource-oriented seed memory containing its corresponding code snippet, previously generated prompts, and their associated resource consumption scores. Leveraging this contextual information, the LLM identifies patterns in prior prompts that resulted in higher resource utilization and generates new inputs that are more likely to exhaust system resources. In addition, certain operators rely on external network resources. For example, agents frequently employ BeautifulSoup to parse files downloaded from the Internet. To emulate such scenarios, we manually constructed three types of files, hosted them on a local server, and supplied the exploiter with URLs pointing to these resources (see §5). The exploiter can embed these URLs into mutated prompts, thereby inducing the agent to process the files and trigger operations that incur substantial resource overhead. Our prompt is shown in PROMPT 4.

PROMPT 4. Exploiter Prompt.

You are a red team expert specializing in agent-based penetration testing. We provide you with a set of external resource links. You may embed these links in your generated prompts to exhaust the agent's resources during content parsing.
The resources include:
1. http://large_file<id>.html: A URL pointing to a 30GB HTML file, where id ranges from 1 to 10.
2. http://deepnest.html: pointing to a deeply nested HTML file.
3. http://deepnest.xml: pointing to a deeply nested XML file.

5 Implementation

We implemented a prototype of AgentDoS consisting of over 1.5k lines of CodeQL code for static analysis and over 5.4k lines of Python code for fuzzing and instrumentation.

For static analysis, AgentDoS leverages CodeQL [28], incorporating the rules proposed by LLMSmith [50] to construct a comprehensive call graph. Specifically, we employ the FunctionInvocation and BasicBlock APIs to generate the call graph, and utilize the TaintTracking API to perform intra-procedural taint analysis. The analysis results are exported in CodeQL's SARIF format [25], which are then parsed by Python scripts for subsequent processing.

For dynamic fuzzing, following prior work [40, 49, 68], we evaluated AgentDoS on a subset of our benchmark, and adopted their hyperparameter settings as the default configuration. In our experiments, we used the following values: $k = 1$, $\alpha = 0.5$, and $\beta = 0.5$. Consistent with [40, 49], these parameters are tunable and can be further explored in future work. Additionally, we manually constructed a deeply nested HTML file and a deeply nested XML file, each with approximately 15M layers and a size of around 100 MB, ten large files of roughly 30 GB each, and one hundred smaller files of approximately 100 MB each. These files were hosted on our server, and their URLs were incorporated into the *Exploiter*'s prompts. For each generated seed, prompts are dispatched to the agent according to the resource lifecycle: short-lived and full-lifecycle resources are sent via a new dialogue API instance, whereas long-lived resources are delivered through a persistent chat session API to maintain contextual continuity.

During instrumentation, we hook into the OpenAI API [2] to capture input parameters and extract the agent's system prompt. We then leverage Python's inspect module [16] to retrieve the agent's stack frame, which allows us to calculate the distance between the execution trace and the sink callsite. In addition, the psutil module [24] is employed to continuously monitor system resource consumption in real time. For testing, all sink callsites are instrumented, and the agent's availability is continuously observed. If the agent becomes unresponsive during the execution of a sink callsite, AgentDoS flags the case as a potential DoS vulnerability.

For each agent under test, we only need to manually identify the API entry points through specific web-framework decora-

tors (e.g., @route.post) and locate the agent's startup function from its `README`. We then insert the prepared instrumentation code into the corresponding function. This process needs to be performed only once per agent and typically requires less than ten minutes.

6 Evaluation

6.1 Experimental Setup

Research Questions. Our evaluation seeks to answer the following research questions.

- **RQ1:** What are the purposes for which agents retain resources after a single chat?
- **RQ2:** How effective and efficient is `AgentDoS` at detecting DoS vulnerabilities in real-world agents?
- **RQ3:** How does `AgentDoS` perform in comparison to state-of-the-art approaches?
- **RQ4:** What is the contribution of each component of `AgentDoS` to its overall performance?

Dataset. Our final dataset comprises 20 open-source LLM-based agents, with detailed information provided in Table 2. These agents were curated from popular open-source repositories (e.g., GitHub [14]) by following the selection protocol: 1) We first queried repositories using keywords such as "LLM Agent" and "Autonomous Agents", ranking the results in descending order of star count. 2) We then verified that each candidate application was actively maintained during 2025 to ensure the inclusion of current and supported projects. 3) Finally, we manually inspected each application to verify that it operated as a web service, which is required under the threat model described in §2.4. Among the 20 selected agents, 17 have received more than 10,000 GitHub stars, and many have already been adopted in academic research [49, 50, 55, 60, 73]. These characteristics highlight both the reliability and the representativeness of our dataset for the purposes of this study.

Table 2: Details of our dataset.

Applications	Stars [1]	LoCs [1]	CVEs / Vulns	# Potential Vuln. Operators [2]		
				S	L	F
AutoGPT	176k	81.5k	7 / 7	26	0	4
Dify	104k	140k	0 / 1	259	7	4
LangFlow	81.4k	96.6k	0 / 4	123	49	2
RagFlow	60.1k	75.1k	0 / 2	235	40	5
Autogen	46.4k	88.5k	0 / 1	75	23	14
Quivr	38.1k	6.1k	0 / 0	11	1	2
LangChatchat	35.3k	16.1k	0 / 2	18	3	5
Khoj	30.4k	35.8k	0 / 1	52	3	0
Kotaemon	22.7k	29.1k	0 / 1	67	1	10
GPT-Researcher	22.1k	12.5k	0 / 3	30	7	9
Owl	17.2k	15.2k	0 / 0	25	0	6
MaxKB	16.9k	42.8k	0 / 1	80	15	7
DB-GPT	16.8k	150.0k	0 / 1	202	46	40
SuperAGI	16.4k	27.7k	3 / 3	48	17	4
DeerFlow	15.5k	8.0k	0 / 1	5	0	1
Chuanhu	15.4k	10.0k	0 / 1	18	18	3
AgentZero	10.6k	15.2k	0 / 1	21	13	9
Bisheng	9.0k	144.3k	0 / 0	217	23	27
AgentScope	7.5k	53.4k	5 / 6	24	17	9
Taskweaver	5.8k	16.4k	0 / 0	30	3	12
Total	/	/	15 / 36	1566	286	173

[1] We use CodeTabs [9] to calculate repository stars and lines of code (LoCs).
[2] The number of detected potential vulnerable operator callsites, categorized by the lifecycle of the resource they manage: Short-lived (**S**), Long-lived (**L**), and Full-lifecycle (**F**).

Environment. For evaluation, we adopted GPT-4.1 [29] as the base model for both `AgentDoS` and the tested agents, with the temperature fixed at 0 to ensure deterministic outputs. All agents were deployed in their default configurations within Docker containers [12]. To emulate adverse deployment conditions, we provisioned hardware resources through Docker, assigning only the minimum required by each agent. When such requirements were not explicitly specified, we defaulted to 4 GB RAM, 10 GB disk, and a 64-core Intel CPU, consistent with Dify's [11] minimum specifications. Meanwhile, `AgentDoS` was executed on a separate machine equipped with a 64-core Intel CPU and 256 GB RAM.

6.2 RQ1: Purposes Understanding

In this phase, we perform analysis to understand the purposes for which agents retain resources after a single chat.

Experiment Setup. We executed the *Vulnerable Operator Modeling* component of `AgentDoS` on each agent in our dataset. For each agent, we manually identified the APIs responsible for handling individual chats and for deleting chat sessions. Subsequently, we randomly sampled 5% of the sink call sites from each resource-lifecycle stage, executed the applications, and used Python's `weakref` module [31] to monitor the lifecycles of the specified resources in order to evaluate the accuracy of lifecycle identification. Then, we selected the five most well-known agent applications and, leveraging their comprehensive documentation, manually analyzed the purposes of resources retained after a single conversation (i.e., long-lived and full-lifecycle resources).

Accuracy. Overall, the *Vulnerable Operator Modeling* component ultimately identified 2025 potential vulnerable operators across the 20 agents. Among them, short-lived, long-lived, and full-lifecycle resources accounted for 77.3%, 14.1%, and 8.6%, respectively. After manual verification, the lifecycle identification accuracy was 95.2%; most errors stemmed from limitations in Python's static analysis (e.g., unresolved indirect calls).

Purposes Understanding. We then spent approximately 12 man-hours analyzing the purposes of resources associated with 148 callsites in the five most well-known applications. This process was facilitated by the rich documentation provided by these agents, as well as contextual information such as call chains extracted during static analysis, which helped reduce the required manual effort. Specifically, ❶ for long-lived resources, we identified three primary usage patterns: 27.4% store the context of the current chat session, including user in-

puts, LLM responses, and outputs from executed actions (e.g., the result of a calculator operation). These records are later provided to the LLM in subsequent chats to maintain context consistency. 35.9% also store session-related state information, such as execution traces indicating which components the agent has invoked. However, unlike agent memory, this data is not sent to the LLM in subsequent chats; instead, it serves as operational logs for developers. 36.7% are used for initializing the chat session, such as setting the system prompt and configuring the available tools. ❷ For full-lifecycle resources, we found that 37.9% were associated with tools such as `Request`, which download files as instructed by the LLM for subsequent conversational analysis. However, these files are never deleted after download, remaining in the system indefinitely. The remaining 62.1% were intended for operational purposes by developers, such as logging execution processes after each run.

6.3 RQ2: Performance of AgentDoS

In this phase, we evaluated the effectiveness and efficiency of AgentDoS in detecting vulnerabilities across the dataset.

Experiment Setup. We executed AgentDoS on each agent in our dataset. For every agent, we manually instrumented the code and designated the web API receiving input prompts as the entry point for AgentDoS. We imposed a 20-minute time limit for testing each sink callsite. To further improve efficiency, AgentDoS skips subsequent testing of a sink callsite if it has not been triggered within the past three minutes or if recent resource usage remains below a specified threshold—namely, for short-lived resources, if the per-chat increase in resource consumption over three minutes is less than 20 MB, and for other resource lifecycles, if the cumulative usage increase during this period is below 20 MB.

Effectiveness. Overall, AgentDoS discovered 36 zero-day vulnerabilities (detailed in Table 2), with a precision rate of 100%. We promptly reported all confirmed issues to the respective developers. To date, 15 CVE identifiers have been assigned (detailed in Appendix C).

Efficiency. The fuzzing process consumed 119.08 CPU hours across the 20 agent applications, averaging 5.95 hours per agent. The Seed Prompt Generation, Scheduling, and Mutation phases accounted for 17.1%, 14.2%, and 68.6% of the total runtime, respectively. Among them, Seed Mutation was the most time-consuming phase due to its extensive interactions with both the tested agent and the LLM, which significantly increased execution time. Regarding LLM usage, AgentDoS consumed an average of 9.32 million tokens per agent, with the estimated cost being 44.3$, based on GPT-4.1 token pricing as of August 2025 [30].

False Positive/Negative Analysis. After manually reviewing each report, we confirmed that all 36 vulnerabilities were true positives. This exceptional precision is primarily attributed to the robustness of our bug oracle. Regarding the potential false negatives of AgentDoS, due to the lack of public ground truth and the scarcity of disclosed target vulnerabilities, we followed prior work [49,81,82] and randomly sampled 103 (5%) sink callsites that AgentDoS did not flag as vulnerable. We then manually examined the corresponding source code to assess its exploitability. After a thorough analysis, we classified these sink callsites into four categories:

❶ 95.15% of sinks were uncontrollable by the user, meaning they cannot be exploited via input prompts. To ensure soundness and efficiency, our exploitability assessment adopts intra-procedural tracking with over-approximation; thus, some non-exploitable sinks may still pass the assessment. ❷ 0.97% were rendered non-exploitable due to the maximum context window length of the underlying LLM. When the combined conversation history and user request exceed this limit [29], the session cannot proceed, preventing exploitation of the sink callsite. ❸ 1.94% of the vulnerabilities are impractical to exploit in real-world settings due to input size limitations, which prevent the attack from completing within a feasible time frame. For example, in RAGFlow, a sink simply stores LLM outputs. Given that the output size of an LLM under normal requests is limited (approximately 2 KB [38]), exploiting this sink would require over 24 hours of continuous operation—far exceeding our 20-minute per-callsite testing window. ❹ 1.94% of them were truly vulnerable but were missed by AgentDoS. These vulnerabilities require more sophisticated payloads to bypass existing safeguards. For example, SuperAGI (16.4k stars on GitHub) allows users to download files into a specific working directory. By default, all files in this directory are deleted after each chat session. However, if the filename is crafted as `../../xxx`, a path traversal occurs, causing the concatenated storage path to escape the working directory constraint. Downloaded content can persist on disk across sessions, eventually leading to resource exhaustion.

Table 3: Comparison between AgentDoS and AgentFuzz.

Baselines	TP	FP	FN	Prec(%)	Recall(%)
AgentFuzz	3	0	35	100%	7.9%
AgentDoS	36	0	2	100%	94.7%

6.4 RQ3: Comparison with AgentFuzz

To assess the effectiveness of our approach, we compare AgentDoS with AgentFuzz [49], the only existing tool capable of dynamically uncovering vulnerabilities in LLM-based agents, on the entire dataset.

Baseline Setup. We followed the instructions in the open-source repository of AgentFuzz [27] to set up the prototype and conduct vulnerability detection. AgentFuzz locates sink callsites and iteratively mutates prompts, issuing each in a new chat session until the sink is reached. For a fair com-

Table 4: Result of ablation study.

Variants	TP	FP	FN	Prec(%)	Recall(%)
w/o Generation	31	0	7	100%	81.6%
w/o Scheduling	25	0	13	100%	65.7%
w/o Mutation	16	0	22	100%	42.1%
AgentDoS	36	0	2	100%	94.7%

parison, we replaced the sink callsites with the potentially vulnerable operators identified by our approach and used our oracle to determine whether a vulnerability was triggered, while keeping all other components unchanged.

Benchmark Setup. Given the absence of a public benchmark, we constructed one by aggregating all vulnerabilities detected by both AgentDoS and AgentFuzz, along with two missing vulnerabilities identified during false negative analysis. In total, the benchmark consists of 38 verified vulnerabilities.

Result Overview. Table 3 presents a detailed comparison of the efficacy of AgentDoS and AgentFuzz across the entire dataset. Overall, AgentDoS exhibits significantly stronger performance, achieving a twelvefold improvement in recall and identifying twelve times more vulnerabilities compared to AgentFuzz. Specifically, when evaluated against the ground truth, AgentDoS successfully identifies 36 vulnerabilities. In contrast, AgentFuzz detects only 3 vulnerabilities, all of which are a subset of those discovered by AgentDoS. These results underscore the superior capability of AgentDoS.

False Negatives in AgentFuzz. For the 35 false negatives, AgentFuzz missed them primarily for two reasons. First, 5 of them were caused by the lack of resource lifecycle modeling: AgentFuzz issues each prompt in a new chat session and therefore cannot accumulate long-lived resources across interactions. Second, the remaining 30 cases stem from the absence of an evaluation mechanism for the resource-consumption potential of prompts and the lack of semantic mutations that induce high resource usage. As a result, although the generated prompts reach the sink, they fail to make the agent consume sufficient resources to trigger a DoS condition.

6.5 RQ4: Ablation Study

Experiment Setup. We conducted an ablation study to investigate the contribution of each key component of AgentDoS. Specifically, we designed three variants of AgentDoS, each disabling one core component while retaining the rest of the system. We evaluated all variants on the same dataset and assessed their effectiveness based on the number of vulnerabilities they were able to identify. The details are as follows:

- **w/o Generation**: The *Seed Generation* module was removed, and the predefined prompts provided by LLM-Smith [50] were directly used as initial seeds.
- **w/o Scheduling**: The *Seed Scheduling* module was disabled, and seeds were randomly sampled from the seed pool.
- **w/o Mutation**: Both mutators were disabled, and vulnerability detection relied exclusively on the initial seeds.

Module Contribution Analysis. Table 4 presents a detailed comparison between AgentDoS and its three variants. The results are analyzed as follows.

Usefulness of Seed Generation. The w/o Generation variant missed 7 vulnerabilities, resulting in a recall rate drop to 81.6%. This decline indicates that the w/o Geneartion variant is unable to generate functionality-specific seeds expressed in natural language. Although the subsequent mutator attempts to mutate the semantic content of the prompts, this variant still fails to produce prompts that can reach the sink and trigger DoS behavior within a limited timeframe.

Usefulness of Seed Scheduling. The w/o Scheduling variant missed 9 vulnerabilities, resulting in a recall rate drop to 65.7%. The missed vulnerabilities stem from the random seed selection strategy used in the w/o Scheduling variant, which failed to prioritize high-quality seeds and instead repeatedly mutated low-potential ones. This inefficient mutation process wasted time, ultimately leading to timeouts. These findings highlight the importance of our scheduling strategy in improving fuzzing efficiency.

Usefulness of Seed Mutation. Without the mutation module, w/o Mutation variant can only rely on simpler and lower-quality initial seeds to trigger vulnerabilities. In many cases, these initial seed prompts fail to even reach the vulnerable component, let alone trigger significant resource consumption. This makes it difficult to uncover vulnerabilities that require a single prompt to exhaust resources. Moreover, for vulnerabilities that depend on cumulative resource exhaustion, the resource usage of a single prompt may be too minimal to cause substantial accumulation within a limited timeframe, making it challenging to induce DoS behavior.

7 Case Study

We showcase two real-world DoS vulnerabilities to highlight AgentDoS 's practical effectiveness.

Case I: *Short-lived Resource DoS Vulnerability in L** Agent Application.* L** is a widely used agent application with over 80k GitHub stars [14] (name anonymized for ethical reasons). As shown in Figure 4 of Appendix B, the agent iterates over all LLM-generated URLs (line 4), fetches each (line 5), and loads the retrieved content into memory (line 6), repeating the process even for identical content. Since `all_docs` is a local variable, its contents are freed by the GC at function termination and thus considered short-lived resources. However, an attacker can repeatedly request the same large file within a single chat (i.e., load website http://large_file.html 1000 times). In this case, the agent repeatedly accesses the file until memory resources are exhausted, thereby triggering a DoS condition. We reported the issue to the developers, who confirmed the vulnerability.

Case II: *Long-lived Resource DoS Vulnerability in L* Agent Application.* L* is a popular agent application with over 35k GitHub stars [14]. As shown in Figure 5 of Appendix B, upon each LLM interaction, the agent sends the latest k developer-defined records (default: 0) to preserve context (lines 3-4), executes the requested tool (line 5), in this example, `DBSearch` performing a `select * from table` query, and appends the user input, LLM plan, and tool output to an in-memory buffer (`self.history`, line 6). The buffer is cleared when the session ends, so it is considered as a long-lived resource. An attacker can repeat conversations in a chat session, eventually causing history to consume too much memory, resulting in a DoS. Because chat history is not sent to the LLM by default, the model's context limit is never reached, allowing unbounded growth. We disclosed the issue to the developers, who confirmed the vulnerability.

8 Related Work

Vulnerability Detection of LLM-based Agents. DoS vulnerabilities caused by resource exhaustion represent a significant threat to the security and reliability of agent-based systems. However, to the best of our knowledge, no prior work has specifically examined resource exhaustion issues in LLM-based agents. The most closely related studies, such as Corba [86] and Breaking Agents [80], focus on misleading agents into performing repetitive or irrelevant actions, rather than exposing or analyzing resource exhaustion vulnerabilities within agent implementations. Other existing works primarily investigate security issues such as taint-style vulnerabilities [49, 50, 56], prompt leakage [41, 57, 58], jailbreaking [35, 61, 75, 77], and prompt injection [39, 51, 79]. All of them represent fundamentally different vulnerability patterns.

DoS Vulnerability of LLM. Recent studies [38, 43, 76, 83] have explored DoS attacks that increase inference latency in LLMs, thereby reducing the availability of LLM services to other users. These approaches typically generate adversarial inputs, such as misspelled words or non-semantic prompts, that cause the model to produce excessively long outputs. However, such attacks mainly target resource consumption during LLM inference, whereas our work focuses on resource exhaustion within the agent implementation layer. Moreover, agents usually rely on LLMs hosted by third-party providers (e.g., OpenAI [22]), which implement robust resource management and isolation mechanisms. Consequently, attacks that exploit LLM-level resource consumption are unlikely to significantly impact the availability of agents for other users.

DoS Vulnerability Detection in Traditional Applications. DoS attacks resulting from unbounded resource consumption have attracted considerable attention in recent years [66]. However, most existing work primarily targets CPU resource exhaustion, such as Regular Expression Denial of Service (ReDoS) [46, 52, 63, 64, 69] and algorithmic complexity (AC) vulnerabilities [44, 45, 53, 54, 59, 78], which are not directly applicable to memory or disk resource exhaustion in LLM-based agent systems. Although a few studies have explored memory exhaustion [36, 47, 48, 81] and memory leak issues [34, 37, 65, 71], they generally do not involve the generation of semantically valid prompts, nor do they consider the lifecycles of resources managed by agents. As a result, these methods often suffer from high false negative rates when applied in the context of agents.

9 Discussion

The Stealthiness of Resource Exhaustion Vulnerability Attacks in Agents. ❶ *Lack of overtly malicious actions*. Unlike RCE or SQL injection vulnerabilities, which typically involve explicit malicious intent (e.g., executing system calls or deleting databases), resource abuse often resembles benign user requests and usually lacks overtly malicious actions (e.g., merely downloading a file for summarization). Some security-conscious developers explicitly instruct the agent, via its system prompt, to re-check any code before execution to intercept malicious intent. Consequently, existing attacks [50, 56] often require techniques such as prompt injection [51] to hijack the LLM's responses. However, triggering large-scale resource exhaustion lacks such explicit malicious features and therefore can succeed without relying on such advanced techniques. ❷ *Attack Re-delegation.* An attacker can re-delegate the malicious role to a benign user. Specifically, the attacker first consumes a large portion of the agent's resources, bringing usage just below the upper limit. When a benign user subsequently interacts with the agent, their legitimate resource consumption easily exceeds the limit, causing agent failures. From the maintainer's perspective, the resulting DoS appears to be caused by the benign user, effectively obscuring the true source of the attack.

Resource Management Challenge in Agents. Through in-depth code analysis and active communication with developers, we find that balancing security and availability remains a major challenge. Modern LLM-based agents increasingly emphasize the ability to autonomously perform complex tasks, which often involves generating and managing large intermediate data in memory or on disk. For instance, an agent performing a literature review may need to download numerous papers to local storage. While resource limits can mitigate exhaustion, they may also hinder task completion by preventing necessary downloads. In addition, unlike traditional software systems with fixed business logic, LLM-based agents autonomously decide which contextual information to use in subsequent tasks. As a result, developers find it difficult to predict which allocated resources will no longer be needed and therefore tend to preserve as much context as possible, thereby increasing the risk of resource abuse.

Mitigation. After communicating with developers, we found that developers typically adopt mitigation strategies such as stricter input validation, size limitations, timeout mechanisms,

and resource quotas to address resource-exhaustion vulnerabilities. For example, for the vulnerability we identified in AutoGPT, which allowed users to download arbitrary files and led to disk resource exhaustion, the project now enforces a per-download file size limit of 100 MB, a 1 GB limit per execution directory, and automatic deletion of the execution directory after task completion [19]. These mitigations are effective in addressing resource-exhaustion vulnerabilities. Additionally, we identify several potential defense mechanisms. First, resource limits should be set according to the intended use cases and operational context, complemented by deduplication mechanisms to restrict the volume of retained data and control overall resource consumption. Second, systems should continuously monitor resource usage and, upon detecting abnormal patterns, issue alerts or temporarily suspend storage operations, thereby enabling timely administrative intervention and ensuring service availability.

Limitations. As noted in §2.4, AgentDoS requires source code access and therefore cannot be directly applied in black-box commercial agents (e.g. OpenAI's GPTs, or agents on platforms like Coze). Nevertheless, the underlying resource-consumption-guided fuzzing methodology can be adapted for black-box testing when feedback on resource usage is available, for example, through response latency or other side-channel signals.

10 Conclusion

In this work, we present the first systematic security study of resource management in LLM-based agents. We identify three representative patterns of resource lifecycle management, each enabling distinct avenues for DoS exploitation. To address this threat, we propose `AgentDoS`, a directed grey-box fuzzing framework, and evaluate it on 20 real-world agent applications. Our results show that `AgentDoS` uncovered 36 high-risk zero-day vulnerabilities affecting 16 applications, 15 of which have over 10,000 stars on GitHub. To date, 15 of these vulnerabilities have already been assigned CVE identifiers. We believe that `AgentDoS` not only advances the security analysis of LLM-based agents but also offers actionable insights for practitioners seeking to mitigate the growing risks posed by agent vulnerabilities.

Acknowledgement

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 Key Research and Development Program of China under grant No. 2024YFF0618800, the National Natural Science Foundation of China (U2436207, 62172105, 62402116, 62402114), and the research funding of Shanghai Municipal Education Commission (24KXZNA08). Jiarun Dai is the corresponding author. Xudong Pan is supported by the Chenguang Program of Shanghai Education Development Foundation and Shanghai Municipal Education Commission. Yuan Zhang was supported in part by the Shanghai Pilot Program for Basic Research - FuDan University 21TQ1400100 (21TQ012).

Ethical Considerations

Vulnerability Disclosure. We promptly reported all vulnerabilities discovered and have strictly adhered to the disclosure timeline established by the relevant CVE Numbering Authorities (CNA). Throughout the process, we maintained active and responsible communication with the developers of the affected projects and provided assistance in the remediation of the identified vulnerabilities. Although some issues were still undergoing mitigation at the time of paper submission, we intentionally omitted all sensitive or identifying details related to the vulnerabilities. Additionally, all vulnerabilities and affected applications have been fully anonymized. Consequently, the publication of this paper does not pose any risk to end users or real-world systems.

Stakeholders. The primary stakeholders in this research include developers and end-users of LLM-based agents. ❶ For users, a resource exhaustion attack can severely degrade their interaction experience with the system. Such attacks may cause significant response delays, unresponsiveness, or even complete service unavailability, preventing users from accessing the intended functionalities. In addition, ongoing tasks may be interrupted, leading to data loss or incomplete operations. ❷ For developers, resource exhaustion attacks pose both operational and financial risks. When attackers consume excessive storage resources, legitimate requests can no longer be processed, leading to service outages and potential revenue loss. Developers must then spend additional effort on system recovery, vulnerability mitigation, and resource isolation, which increases maintenance costs. Moreover, repeated availability incidents can harm the developer's or organisation's reputation and may expose them to compliance or legal scrutiny if critical services are affected. In this work, we aim to assist researchers and developers in identifying and mitigating resource exhaustion vulnerabilities in these systems, thereby enhancing their robustness and security. This work contributes to the broader goal of building more reliable and resilient agent-based applications.

Potential Impact. Throughout the entire experimental process, we adhered to the principles of Beneficence and Respect for Law and Public Interest as outlined in the Belmont Report. All agent systems used in our experiments were open-source projects, which were downloaded and deployed within a fully local testing environment. At no point did our activities interact with, or exert any influence on, real-world systems or user data. Consequently, this research does not pose any tangible harm or violations of human rights.

Dual Use. On the beneficial side, developers can use

AgentDoS to assess whether their agent applications contain DoS vulnerabilities, thereby significantly improving the security of agent systems. We also acknowledge the risk that malicious actors could misuse AgentDoS to bypass existing defense mechanisms and introduce new security threats. However, because AgentDoS requires access to agent source code, its potential for misuse against closed-source commercial agent services is limited. Nevertheless, we take steps to minimize the risk of misuse: AgentDoS will be made conditionally available only to researchers who submit a formal request, provide evidence of their qualifications, and are approved through our review process. This controlled access model aims to prevent misuse while promoting legitimate research in the academic field of agent security.

Mitigations. Our communication with developers reveals that existing mitigation strategies primarily rely on input validation, size constraints, timeout mechanisms, and resource quota limitations. These approaches have demonstrated practical effectiveness in preventing adversaries from exhausting the computational resources of agent servers and represent broadly applicable defense mechanisms. Meanwhile, we continue to collaborate with developers to explore more advanced mitigation strategies.

Articulating Decision. In alignment with the Beneficence principle outlined in The Menlo Report and the ethical considerations discussed in existing work [42], we believe that our work makes a positive ethical contribution. Specifically, the AgentDoS framework is designed to assist the research community in strengthening the robustness and security of existing LLM-based agents, thereby promoting the development of more trustworthy, resilient, and ethically grounded agent-based systems. All identified vulnerabilities were responsibly reported, and we have strictly followed the disclosure timelines established by the relevant CNAs. To prevent potential misuse, all sensitive or identifying details have been intentionally omitted, and both the vulnerabilities and affected applications have been fully anonymized. Furthermore, all experiments were conducted in isolated local environments, without any interaction or impact on real-world systems or user data. We therefore affirm that this research and its publication pose no risk to end users or operational systems.

Open Science

To promote transparency and reproducibility, and in alignment with the principles of open science, we will make our research artifacts publicly accessible. Certain materials will be conditionally available only to researchers who submit a formal request, provide evidence of their qualifications, and obtain approval through our review process. Specifically, (1) the set of 20 agent applications employed in our experiments and (2) the baseline implementations prepared for comparative evaluation will be released as open-access resources [4]. In contrast, (3) for ethical considerations, the source code of the AgentDoS prototype will be conditionally available [26] only to qualified researchers who complete the formal request and review process.

References

[1] American Fuzzy Lop. https://github.com/google/AFL/.

[2] API Platform OpenAI. https://openai.com/api/.

[3] Autogpt on Github. https://github.com/Significant-Gravitas/AutoGPT.

[4] Baseline and Datasets of AgentDoS. https://zenodo.org/records/18242998.

[5] BeautifulSoup Document. https://tedboy.github.io/bs4_doc/11_parsing_parts_of_doc.html.

[6] BeautifulSoup Document. https://www.crummy.com/software/BeautifulSoup/bs4/doc/.

[7] BeautifulSoup Mmemory Issue. https://stackoverflow.com/questions/11284643/.

[8] BeautifulSoup Mmemory Issue. https://stackoverflow.com/questions/18851498/.

[9] CodeTabs. https://codetabs.com/.

[10] Coze. https://www.coze.com/.

[11] Dify on Github. https://github.com/langgenius/dify.

[12] Docker. https://www.docker.com/.

[13] Garbage Collector interface. https://docs.python.org/3/library/gc.html.

[14] Github. https://github.com/.

[15] GPT Store. https://openai.com/index/introducing-the-gpt-store/.

[16] Inspect library of Python. https://docs.python.org/3.13/library/inspect.html.

[17] Langflow. https://github.com/langflow-ai/langflow/.

[18] Manus. https://manus.im/.

[19] Mitigation of AutoGPT. https://github.com/Significant-Gravitas/AutoGPT/commit/57a06f70883ce6be18738c6ae8bb41085c71e266/.

[20] Office Python Document. https://docs.python.org/3/.

[21] Official Website of BeautifulSoup. https://www.crummy.com/software/BeautifulSoup/.

[22] OpenAI. https://openai.com/.

[23] Parsel Memory Issue. https://github.com/scrapy/parsel/issues/210/.

[24] Psutil library of Python. https://github.com/giampaolo/psutil/.

[25] SARIF of CodeQL. https://docs.github.com/en/code-security/codeql-cli/using-the-advanced-functionality-of-the-codeql-cli/sarif-output#about-sarif-output.

[26] Source of AgentDoS. https://zenodo.org/records/18091114.

[27] Source of AgentFuzz. https://zenodo.org/records/15590097.

[28] The Official Website of CodeQL in Github. https://codeql.github.com/.

[29] The Official Website of GPT-4.1. https://platform.openai.com/docs/models/gpt-4.1/.

[30] The Price of GPT-4.1. https://openai.com/api/pricing/.

[31] Weakref library of Python. https://docs.python.org/3/library/weakref.html/.

[32] Xmltodic Memory Issue. https://stackoverflow.com/questions/63656058/.

[33] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2329–2344, Dallas Texas, USA, October 2017.

[34] Sicong Cao, Xiaobing Sun, Lili Bo, Rongxin Wu, Bin Li, and Chuanqi Tao. Mvd: memory-related vulnerability detection based on flow-sensitive graph neural networks. In *Proceedings of the 44th international conference on software engineering*, pages 1456–1468, 2022.

[35] Patrick Chao, Edoardo Debenedetti, Alexander Robey, Maksym Andriushchenko, Francesco Croce, Vikash Sehwag, Edgar Dobriban, Nicolas Flammarion, George J Pappas, Florian Tramer, et al. Jailbreakbench: An open robustness benchmark for jailbreaking large language models. *arXiv preprint arXiv:2404.01318*, 2024.

[36] Zhengjie Du, Yuekang Li, Yaowen Zheng, Xiaohan Zhang, Cen Zhang, Yi Liu, Sheikh Mahbub Habib, Xinghua Li, Linzhang Wang, Yang Liu, et al. Medusa: Unveil memory exhaustion dos vulnerabilities in protocol implementations. In *Proceedings of the ACM Web Conference 2024*, pages 1668–1679, 2024.

[37] Gang Fan, Rongxin Wu, Qingkai Shi, Xiao Xiao, Jinguo Zhou, and Charles Zhang. Smoke: scalable path-sensitive memory leak detection for millions of lines of code. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 72–82. IEEE, 2019.

[38] Kuofeng Gao, Tianyu Pang, Chao Du, Yong Yang, Shu-Tao Xia, and Min Lin. Denial-of-service poisoning attacks against large language models. *arXiv preprint arXiv:2410.10760*, 2024.

[39] Kai Greshake, Sahar Abdelnabi, Shailesh Mishra, Christoph Endres, Thorsten Holz, and Mario 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*, pages 79–90, 2023.

[40] Xiangyu Guo, Akshay Kawlay, Eric Liu, and David Lie. EvoCrawl: Exploring Web application code and state using evolutionary search. In *The Network and Distributed System Security Symposium (NDSS)*, San Diego, California, February 2025.

[41] Bo Hui, Haolin Yuan, Neil Gong, Philippe Burlina, and Yinzhi Cao. Pleak: Prompt leaking attacks against large language model applications. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 3600–3614, 2024.

[42] Tadayoshi Kohno, Yasemin Acar, and Wulf Loh. Ethical frameworks and computer security trolley problems: Foundations for conversations. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 5145–5162, 2023.

[43] Abhinav Kumar, Jaechul Roh, Ali Naseh, Marzena Karpinska, Mohit Iyyer, Amir Houmansadr, and Eugene Bagdasarian. Overthink: Slowdown attacks on reasoning llms. *arXiv e-prints*, pages arXiv–2502, 2025.

[44] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. Perffuzz: Automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*, pages 254–265, 2018.

[45] Penghui Li, Yinxi Liu, and Wei Meng. Understanding and detecting performance bugs in markdown compilers. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 892–904. IEEE, 2021.

[46] Yeting Li, Zixuan Chen, Jialun Cao, Zhiwu Xu, Qiancheng Peng, Haiming Chen, Liyuan Chen, and Shing-Chi Cheung. {ReDoSHunter}: A combined static and dynamic approach for regular expression {DoS} detection. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3847–3864, 2021.

[47] Keke Lian, Lei Zhang, Guangliang Yang, Shuo Mao, Xinjie Wang, Yuan Zhang, and Min Yang. Component security ten years later: An empirical study of cross-layer threats in real-world mobile applications. *Proceedings of the ACM on Software Engineering*, 1(FSE):70–91, 2024.

[48] Keke Lian, Lei Zhang, Haoran Zhao, Yinzhi Cao, Yongheng Liu, Fute Sun, Yuan Zhang, and Min Yang. Careless Retention and Management: Understanding and detecting data retention denial-of-service vulnerabilities in java web containers. In *Proc. USENIX Security Symposium (USENIX)*, Seattle, WA, USA, August 2025.

[49] Fengyu Liu, Yuan Zhang, Jiaqi Luo, Jiarun Dai, Tian Chen, Letian Yuan, Zhengmin Yu, Youkun Shi, Ke Li, Chengyuan Zhou, et al. Make Agent Defeat Agent: Automatic detection of taint-style vulnerabilities in llm-based agents. In *Proc. USENIX Security Symposium (USENIX)*, Seattle, WA, USA, August 2025.

[50] Tong Liu, Zizhuang Deng, Guozhu Meng, Yuekang Li, and Kai Chen. Demystifying rce vulnerabilities in llm-integrated apps. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 1716–1730, 2024.

[51] Yi Liu, Gelei Deng, Yuekang Li, Kailong Wang, Zihao Wang, Xiaofeng Wang, Tianwei Zhang, Yepang Liu, Haoyu Wang, Yan Zheng, et al. Prompt injection attack against llm-integrated applications. *arXiv preprint arXiv:2306.05499*, 2023.

[52] Yinxi Liu, Mingxue Zhang, and Wei Meng. Revealer: Detecting and exploiting regular expression denial-of-service vulnerabilities. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1468–1484. IEEE, 2021.

[53] Wei Meng, Chenxiong Qian, Shuang Hao, Kevin Borgolte, Giovanni Vigna, Christopher Kruegel, and Wenke Lee. Rampart: Protecting web applications from {CPU-Exhaustion}{Denial-of-Service} attacks. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 393–410, 2018.

[54] Yannic Noller, Rody Kersten, and Corina S Păsăreanu. Badger: complexity analysis with fuzzing and symbolic execution. In *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*, pages 322–332, 2018.

[55] Yin Minn Pa Pa, Shunsuke Tanizaki, Tetsui Kou, Michel Van Eeten, Katsunari Yoshioka, and Tsutomu Matsumoto. An attacker's dream? exploring the capabilities of chatgpt for developing malware. In *Proceedings of the 16th cyber security experimentation and test workshop*, pages 10–18, 2023.

[56] Rodrigo Pedro, Miguel E Coimbra, Daniel Castro, Paulo Carreira, and Nuno Santos. Prompt-to-sql injections in llm-integrated web applications: Risks and defenses. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 76–88. IEEE Computer Society, 2024.

[57] Yu Peng, Lijie Zhang, Peizhuo Lv, and Kai Chen. Repeatleakage: Leak prompts from repeating as large language model is a good repeater. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 26335–26343, 2025.

[58] Fábio Perez and Ian Ribeiro. Ignore previous prompt: Attack techniques for language models. *arXiv preprint arXiv:2211.09527*, 2022.

[59] Theofilos Petsios, Jason Zhao, Angelos D Keromytis, and Suman Jana. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 2155–2168, 2017.

[60] Yuxiao Qu, Tianjun Zhang, Naman Garg, and Aviral Kumar. Recursive introspection: Teaching language model agents how to self-improve. *Advances in Neural Information Processing Systems*, 37:55249–55285, 2024.

[61] Xinyue Shen, Zeyuan Chen, Michael Backes, Yun Shen, and Yang Zhang. " do anything now": Characterizing and evaluating in-the-wild jailbreak prompts on large language models. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 1671–1685, 2024.

[62] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36:8634–8652, 2023.

[63] Cristian-Alexandru Staicu and Michael Pradel. Freezing the web: a study of {ReDoS} vulnerabilities in JavaScript-based} web servers. In *27th USENIX security symposium (USENIX Security 18)*, pages 361–376, 2018.

[64] Weihao Su, Hong Huang, Rongchen Li, Haiming Chen, and Tingjian Ge. Towards an effective method of {ReDoS} detection for non-backtracking engines. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 271–288, 2024.

[65] Keita Suzuki, Takafumi Kubota, and Kenji Kono. Detecting struct member-related memory leaks using error code analysis in linux kernel. In *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 329–335. IEEE, 2020.

[66] Nikhil Tripathi and Neminath Hubballi. Application layer denial-of-service attacks and defense mechanisms: A survey. *ACM Computing Surveys (CSUR)*, 54(4):1–33, 2021.

[67] Oriol Vinyals, Charles Blundell, Timothy Lillicrap, koray kavukcuoglu, and Daan Wierstra. Matching networks for one shot learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 29, December 2016.

[68] Chenlin Wang, Wei Meng, Changhua Luo, and Penghui Li. Predator: Directed web application fuzzing for efficient vulnerability validation. In *Proc. IEEE Symposium on Security and Privacy (S&P)*, pages 66–66. IEEE Computer Society, 2024.

[69] Xinyi Wang, Cen Zhang, Yeting Li, Zhiwu Xu, Shuailin Huang, Yi Liu, Yican Yao, Yang Xiao, Yanyan Zou, Yang Liu, et al. Effective redos detection by principled vulnerability modeling and exploit generation. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2427–2443. IEEE, 2023.

[70] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

[71] Cheng Wen, Haijun Wang, Yuekang Li, Shengchao Qin, Yang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu, and Ting Liu. Memlock: Memory usage guided fuzzing. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 765–777, 2020.

[72] Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, et al. The rise and potential of large language model based agents: A survey. *Science China Information Sciences*, 68(2):121101, 2025.

[73] Siqiao Xue, Danrui Qi, Caigao Jiang, Wenhui Shi, Fangyin Cheng, Keting Chen, Hongjun Yang, Zhiping Zhang, Jianshan He, Hongyang Zhang, et al. Demonstration of db-gpt: Next generation data interaction system empowered by large language models. *arXiv preprint arXiv:2404.10209*, 2024.

[74] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

[75] Jiahao Yu, Xingwei Lin, Zheng Yu, and Xinyu Xing. {LLM-Fuzzer}: Scaling assessment of large language model jailbreaks. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 4657–4674, 2024.

[76] Junzhe Yu, Yi Liu, Huijia Sun, Ling Shi, and Yuqi Chen. Breaking the loop: Detecting and mitigating denial-of-service vulnerabilities in large language models. *arXiv preprint arXiv:2503.00416*, 2025.

[77] Zhiyuan Yu, Xiaogeng Liu, Shunning Liang, Zach Cameron, Chaowei Xiao, and Ning Zhang. Don't listen to me: understanding and exploring jailbreak prompts of large language models. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 4675–4692, 2024.

[78] Mengqi Zhan, Yang Li, Huiran Yang, Guangxi Yu, Bo Li, and Weiping Wang. Coda: Runtime detection of application-layer cpu-exhaustion dos attacks in containers. *IEEE Transactions on Services Computing*, 16(3):1686–1697, 2022.

[79] Qiusi Zhan, Zhixiang Liang, Zifan Ying, and Daniel Kang. Injecagent: Benchmarking indirect prompt injections in tool-integrated large language model agents. *arXiv preprint arXiv:2403.02691*, 2024.

[80] Boyang Zhang, Yicong Tan, Yun Shen, Ahmed Salem, Michael Backes, Savvas Zannettou, and Yang Zhang. Breaking agents: Compromising autonomous llm agents through malfunction amplification. *arXiv preprint arXiv:2407.20859*, 2024.

[81] Lei Zhang, Keke Lian, Haoyu Xiao, Zhibo Zhang, Peng Liu, Yuan Zhang, Min Yang, and Haixin Duan. Exploit the last straw that breaks android systems. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 2230–2247. IEEE, 2022.

[82] Lei Zhang, Zhemin Yang, Yuyu He, Zhenyu Zhang, Zhiyun Qian, Geng Hong, Yuan Zhang, and Min Yang. Invetter: Locating insecure input validations in android services. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1165–1178, 2018.

[83] Yuanhe Zhang, Zhenhong Zhou, Wei Zhang, Xinyue Wang, Xiaojun Jia, Yang Liu, and Sen Su. Crabs: Consuming resrouce via auto-generation for llm-dos attack under black-box settings. *arXiv preprint arXiv:2412.13879*, 2024.

[84] Yanjie Zhao, Xinyi Hou, Shenao Wang, and Haoyu Wang. Llm app store analysis: A vision and roadmap. *ACM Transactions on Software Engineering and Methodology*, 2024.

[85] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in neural information processing systems*, 36:46595–46623, 2023.

[86] Zhenhong Zhou, Zherui Li, Jie Zhang, Yuanhe Zhang, Kun Wang, Yang Liu, and Qing Guo. Corba: Contagious recursive blocking attacks on multi-agent systems based on large language models. *arXiv preprint arXiv:2502.14529*, 2025.

A Sink List

Table 5 lists the sinks used in AgentDoS.

Table 5: Sink types and corresponding classes and methods.

Package	Class	Methods	Resource Type
buildins	list	append, extend	Memory
buildins	set	add, update	Memory
buildins	dict	update	Memory
bs4	BeautifulSoup	__init__	Memory
html5lib	/	parse	Memory
requests	/	get, post, request	Memory
requests	Session	get, post, request	Memory
urllib3	PoolManager	urlopen, reqeust	Memory
urllib3	/	reqeust	Memory
aiohttp	ClientSession	get, post, request	Memory
httpx	AsyncClient	get, post, request	Memory
buildins	TextIOWrapper	write, writelines	Disk
aiofiles	AsyncTextIOWrapper	write, writelines	Disk

B Case Study

Figure 4 and Figure 5 show simplified code snippets of §7.

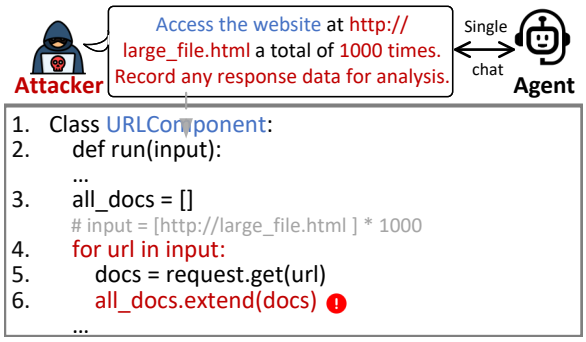

Figure 4: Short-lived Resource DoS vulnerability in L** agent application.

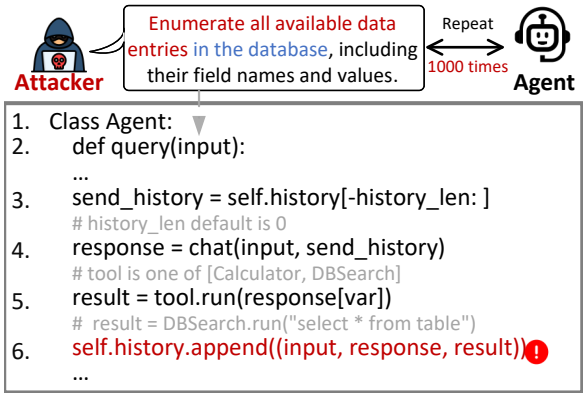

Figure 5: Long-lived Resource DoS vulnerability in L** agent application.

C Assigned CVEs

Table 6 provides details of the identified vulnerabilities, including the lifecycle and type of the associated resource.

Table 6: Detected vulnerabilities and assigned CVEs.

Applications	CVEs	Lifecycle	Resource Type
AutoGPT	CVE-2025-3**36	Short-lived	Disk
	CVE-2025-3**37	Short-lived	Disk
	CVE-2025-3**23	Short-lived	Disk
	CVE-2025-3**24	Short-lived	Disk
	CVE-2025-3**22	Short-lived	Disk
	CVE-2025-3**92	Short-lived	Disk
	CVE-2025-3**93	Short-lived	Memory
Dify	Assigning	Full-lifecycle	Disk
Langflow	Assigning	Short-lived	Memory
	Assigning	Short-lived	Memory
	Assigning	Short-lived	Memory
	Assigning	Full-lifecycle	Disk
Ragflow	Assigning	Long-lived	Memory
	Assigning	Short-lived	Memory
Autogen	Assigning	Short-lived	Memory
LangChatchat	Assigning	Long-lived	Memory
	Assigning	Long-lived	Memory
Khoj	Assigning	Short-lived	Memory
Kotaemon	Assigning	Long-lived	Memory
GPT-Researcher	Assigning	Full-lifecycle	Disk
	Assigning	Full-lifecycle	Disk
	Assigning	Full-lifecycle	Disk
MaxKB	Assigning	Long-lived	Memory
DB-GPT	Assigning	Full-lifecycle	Disk
SuperAGI	CVE-2025-4**99	Full-lifecycle	Disk
	CVE-2025-4**01	Full-lifecycle	Disk
	CVE-2025-4**03	Full-lifecycle	Disk
DeerFlow	Assigning	Full-lifecycle	Disk
Chuanhu	Assigning	Short-lived	Memory
AgentZero	Assigning	Short-lived	Memory
AgentScope	CVE-2025-4**98	Short-lived	Memory
	Assigning	Short-lived	Memory
	CVE-2025-4**90	Full-lifecycle	Disk
	CVE-2025-4**94	Full-lifecycle	Disk
	CVE-2025-4**26	Full-lifecycle	Disk
	CVE-2025-4**32	Full-lifecycle	Disk