PentestAgent: Incorporating LLM Agents to Automated Penetration Testing

Xiangmin Shen
Northwestern University
Evanston, Illinois, USA
xiangminshen2019@u.northwestern.edu

Lingzhi Wang
Northwestern University
Evanston, Illinois, USA
lingzhiwang2025@u.northwestern.edu

Zhenyuan Li
Zhejiang University
Hangzhou, Zhejiang, China
lizhenyuan@zju.edu.cn

Yan Chen
Northwestern University
Evanston, Illinois, USA
ychen@northwestern.edu

Wencheng Zhao
Ant Group
Hangzhou, Zhejiang, China
wencheng.zwc@antgroup.com

Dawei Sun
Ant Group
Hangzhou, Zhejiang, China
david.sdw@antgroup.com

Jiashui Wang
Zhejiang University
Hangzhou, Zhejiang, China
12221251@zju.edu.cn

Wei Ruan
Zhejiang University
Hangzhou, Zhejiang, China
ruanwei@zju.edu.cn

Abstract

Penetration testing is a critical technique for identifying security vulnerabilities, traditionally performed manually by skilled security specialists. This complex process involves gathering information about the target system, identifying entry points, exploiting the system, and reporting findings. Despite its effectiveness, manual penetration testing is time-consuming and expensive, often requiring significant expertise and resources that many organizations cannot afford. While automated penetration testing methods have been proposed, they often fall short in real-world applications due to limitations in flexibility, adaptability, and implementation.

Recent advancements in large language models (LLMs) offer new opportunities for enhancing penetration testing through increased intelligence and automation. However, current LLM-based approaches still face significant challenges, including limited penetration testing knowledge and a lack of comprehensive automation capabilities. To address these gaps, we propose PentestAgent, a novel LLM-based automated penetration testing framework that leverages the power of LLMs and various LLM-based techniques like Retrieval Augmented Generation (RAG) to enhance penetration testing knowledge and automate various tasks. Our framework leverages multi-agent collaboration to automate intelligence gathering, vulnerability analysis, and exploitation stages, reducing manual intervention. We evaluate PentestAgent using a comprehensive benchmark, demonstrating superior performance in task completion and overall efficiency. This work significantly advances the practical applicability of automated penetration testing systems.

CCS Concepts

• **Security and privacy → Penetration testing**; • **Computing methodologies → Multi-agent systems**.

Keywords

Penetration Testing, Large Language Model, Agent

1 Introduction

Penetration testing is a widely adopted technique for proactively identifying security vulnerabilities. This process involves gathering information about the target system (reconnaissance), identifying possible entry points, attempting to exploit the system, and reporting the findings. [13] Traditionally, penetration testing has been a complex manual process requiring highly skilled security specialists with extensive experience. Testers typically write their own exploits, master public domain tools, and perform numerous tedious and time-consuming tasks. [42] According to Rapid7's Under the Hoodie report, penetration testing takes an average of 80 hours, with significant outliers taking several hundred hours. [6] Consequently, manual penetration testing often necessitates large, diverse teams, which most organizations cannot afford.

Although automated penetration testing has been a concept for over a decade, a significant gap remains between the proposed methods and their real-world application. Early works [4, 29, 35] primarily modeled attack planning as an attack graph problem [2] in a deterministic and fully observable world. However, such an approach imposes limitations: it assumes complete observability from the defenders' standpoint and lacks the flexibility and adaptability required for dynamic environments.

Later efforts [5, 14, 18, 19, 36–38, 48] addressed these shortcomings by introducing uncertainty into planning methodologies, treating attack planning as a Markov Decision Process (MDP), which model the world as states and actions as transitions between states, with a reward function encoding the "reward" for moving from one state to another. As extensions to MDP-based approaches, the subsequent works employ partially observable Markov decision process (POMDP) [36, 37] and reinforcement learning algorithms [5, 18] to account for further uncertainty in the environment and action outcomes. These advancements better align with real-world conditions where attackers possess limited knowledge of the target systems. Nevertheless, these probabilistic models focus on establishing a theoretical model for automated pentesting planning and lack the implementation aspect.

Large language models (LLMs) are rapidly evolving, showcasing impressive capabilities in a wide range of tasks, including text summarizing, data analysis, and question-answering. The powerful LLMs have gained significant attention in security applications, leading to a shift towards LLM-based security solutions that offer enhanced intelligence and automation capabilities compared to existing methods, making it possible to address the implementation gap in automated penetration testing.

Recent attempts to utilize LLMs for automating penetration testing [11, 17, 47] have shown some promising initial results. However, two crucial gaps still need to be addressed for practical use:
1) Limited pentesting knowledge: These methods heavily rely on pre-trained language models for generating actionable items. However, the training datasets for these models often lack comprehensive coverage of penetration testing techniques. This results in a limited state space and an outdated action space, reducing the effectiveness and relevance of the generated actions.
2) Insufficient Automation: Existing approaches lack the automated capabilities to adapt to various environments, including validating and debugging the suggested procedures and dynamically acquiring and applying new pentesting techniques.
Without addressing these challenges, current methods become error-prone brute force techniques with limited effectiveness in practice.

Table 1: Comparison of LLM-based pentesting systems

System	State&Action Space	Online Search Augmentation	Validation& Debugging Capability
PentestAgent	Large	Auto	Auto
AutoAttacker [47]	Unknown[1]	Manual	Manual
PentestGPT [11]	Unknown[1]	Manual	Manual
Happe et al. [17]	Small	No	No

[1] AutoAttacker and PentestGPT solely rely on LLMs to provide reconnaissance and attack techniques, which can be limited and outdated.

To overcome these challenges, we propose a novel LLM-based automated penetration testing framework PentestAgent. Our framework aims to enhance penetration testing knowledge by continuously integrating new techniques and updating the framework's knowledge base with the assistance of LLMs. Additionally, PentestAgent establishes a robust automated penetration testing pipeline utilizing LLM techniques, incorporating validation and debugging mechanisms to ensure the effectiveness and relevance of generated actions in specific target environments. By bridging these gaps, we aim to significantly improve the practical applicability and reliability of automated penetration testing frameworks.

PentestAgent adopts a multi-agent design. Each agent within the framework is equipped with a set of tools and assumes responsibility for a specific task in the penetration testing process. These agents are highly adaptable, as their toolsets can be customized to suit various tasks, making them flexible and extensible across different scenarios.

Besides LLM agents, PentestAgent incorporates Retrieval Augmented Generation (RAG) [20] into its framework. RAG enhances the capabilities of LLMs by enabling them to leverage additional data for response synthesis and thus serves as a potent augmentation tool for LLM agents, enabling them to produce more informed and contextually relevant outputs. RAG also allows customized control of the supplementary context in the communication, ensuring efficient use of the context window. Table 1 shows a comparison among existing automated pentesting systems.

PentestAgent comprises four major components: a reconnaissance agent, a search agent, a planning agent, and an execution agent. These components collaborate seamlessly to automate the three primary stages of penetration testing: intelligence gathering, vulnerability analysis, and exploitation.

The reconnaissance agent initiates the process by gathering environmental data upon receiving the target. It generates and executes reconnaissance commands to collect comprehensive information about the target host. This data is then analyzed and stored in an environmental information database for further reference.

In the vulnerability analysis stage, the search agent queries the environmental database to identify exposed services and applications. It identifies potential attack surfaces and procedures, cataloging them separately. Concurrently, the planning agent employs Retrieval Augmented Generation (RAG) techniques to refine potential attack surfaces and selects suitable exploits tailored to the target environment.

During exploitation, the execution agent attempts to execute planned attacks on the target host. It retrieves necessary operational details from the environmental database, debugs execution errors, and logs all activities for comprehensive penetration testing reports.

This comprehensive approach promises to mitigate the reliance on manual intervention and enhance the scalability and adaptability of automated pentesting systems. To sum up, we make the following contributions:

- We design PentestAgent, a LLM-based automated pentesting system that operates with minimal human intervention. PentestAgent integrates multi-agent design and Retrieval Augmented Generation (RAG) techniques to enhance penetration testing knowledge and automate various tasks.
- We design a comprehensive penetration testing benchmark based on the leading open-source collection of pre-built vulnerable docker environments VulHub. This benchmark spans various levels of difficulty and encompasses a wide range of common weaknesses and vulnerabilities, providing a comprehensive and practical framework for evaluating penetration testing tools.
- We design experiments and metrics to evaluate PentestAgent on our benchmark. The results demonstrate PentestAgent's superior performance in automatically completing the entire penetration testing process, as well as in individual penetration tasks.

We will make our benchmark datasets and framework publicly available to facilitate further research in automated penetration testing.

2 Background and Related Work

2.1 Penetration Testing

Penetration testing, or pentesting, is a multi-stage, labor-intensive process designed to identify security vulnerabilities in systems under test.

2.1.1 Workflow. According to the Penetration Testing Execution Standard (PTES) [41], penetration testing consists of three main stages: intelligence gathering, vulnerability analysis, and exploitation. Existing tools typically focus on individual tasks within these stages. For instance, Nmap[28] specializes in information gathering by collecting response data from a target through direct interaction. Nessus [43] and OpenVAS [15] are dedicated to vulnerability analysis, providing comprehensive scanning capabilities through their integrated services and tools. Metasploit [34] focuses on exploitation, offering various exploits with customizable payloads once a vulnerability is identified. While these tools excel in their specific tasks, mastering their use and integrating them into a cohesive attack plan requires significant expertise in penetration testing and substantial manual effort.

Recent advancements in artificial intelligence have led to the development of more sophisticated penetration testing frameworks based on machine learning and Markov Decision Process (MDP) algorithms [5, 18, 48]. For example, Chen et al. [5] designed a reinforcement learning-based framework for automated attack planning. This framework incorporates expert knowledge into state-action pairs and employs a reward function to train the system to execute actions with the highest success rate. Although these frameworks can generate reasonable attack plans, they lack the dynamic implementation aspects of penetration testing. They are unable to react to potential failures and adjust the plan in real time.

The rise of LLM-based applications has further advanced the automation of penetration testing tasks such as text analysis, task planning, code modification, and execution debugging. However, the existing LLM-based penetration testing frameworks still lack comprehensive coverage of the stages and automation for practical use. AutoAttacker [47] focuses on constructing post-breach attacks, neglecting the pre-compromise stages. PentestGPT [11], while implicitly considering multiple stages through its "pentesting task tree," still relies on human decision to proceed with a certain branch of tasks, leading to inefficiency and ineffectiveness. For example, PentestGPT may overly focus on one task while neglecting others, resulting in an unbalanced approach. Moreover, PentestGPT and AutoAttacker depend on the LLM's pre-trained knowledge and human analysis to gather additional information about the target, discover and validate vulnerabilities, and select the next steps from the task tree. These tasks still demand considerable manual effort.

Our objective is to develop a comprehensive and automated penetration testing framework that integrates all stages into a coherent and effective workflow. By leveraging the advanced capabilities of LLMs, we aim to significantly enhance the level of automation in penetration testing, reducing dependency on human expertise and effort while ensuring seamless integration across all stages of the penetration testing process.

2.1.2 Scope. Depending on the scope, penetration testing can be roughly divided into two groups: external assessments and internal assessments. [6] In external assessments, the penetration tester starts off on the internet and targets the client's web applications, services, and other internet-facing assets. The tests include social engineering engagement, red team attack simulation, and external network compromise. In internal assessments, the tester has access to the internal network, source code, or physical access to a device under test. The tests include code review and internal network compromise. Several works have focused on improving internal assessments using LLM-based frameworks (e.g., ChatAFL [25] and FuzzGPT [12] for fuzzing, LLift [22] and LATTE [24] for program and binary analysis), while the external assessments have gained less attention. This paper aims to fill the gap to improve external assessments with LLM-based methods. According to Rapid7's latest report on penetration testing [7], external assessments have been the most demanding ones, in which external network compromise alone takes more than 80% of the external assessments performed. Thus, this paper will illustrate the effectiveness of PentestAgent with web penetration testing.

2.2 Challenges of Applying LLM to Pentesting

Although LLM-based systems have demonstrated excellent capabilities in various tasks, we identify the following challenges in using LLM in penetration testing tasks.

C1. Limited Pentesting Knowledge. The LLM has basic knowledge about vulnerabilities and penetration testing to help get the penetration testing started. However, it requires the user to manually search for additional information, such as actual CVE numbers, analyze related information about the CVE like the vulnerability type and relevant exploits, manually set up the exploitation tools, search for the appropriate exploit to run, and configure the exploit. All these additional efforts require penetration testing expertise and significant human effort. For example, here is an example conversation with GPT-4 on how to conduct penetration testing on ActiveMQ 5.17.3.

Limited Pentesting Knowledge Example

User: What may be the vulnerabilities of ActiveMQ 5.17.3? And how can I test if the vulnerabilities exist?
GPT-4:

(1) Keep Software Updated ...
(2) Check CVEs ...
(3) Security Scanning Tools ...
(4) Configuration Review ...
(5) Penetration Testing ...
(6) Log Analysis ...

User: Could you give detailed steps for pentesting?
GPT-4:

(1) Identify Known CVEs ...
(2) Set Up Metasploit ...
(3) Search for ActiveMQ Modules ...
(4) Configure and Run Exploit ...
(5) Manual Testing ...

C2. Short-term Memory. The limitation of models' context windows, leading to the short-term memory problem, becomes particularly challenging during long-lasting tasks such as penetration testing, which requires continuous memory across a prolonged time period. For instance, in vulnerability analysis, information gathered during intelligence gathering is crucial for identifying vulnerabilities and searching for corresponding exploits. Similarly, in the exploitation stage, information from the intelligence gathering stage aids in selecting and configuring appropriate exploits. The short-term memory limitations can lead to several issues throughout the penetration testing process.

1) Repetition of Tasks: Due to the restricted context window, the model may forget previously gathered information or actions taken, leading to redundant tasks being performed. For example, LLM may repeat the information collection process that was already performed earlier.

Repetition of Tasks Example

Intelligence Gathering

LLM: Use Nmap to perform a comprehensive scan of all ports on the target host to identify open ports and services.
User: {Nmap scan results}

Vulnerability Analysis

LLM: Use Nmap to perform a comprehensive scan of all ports on the target host to identify open ports and services.
User: {Nmap scan results}

2) Loss of Context: As the model's context shifts with each interaction or stage transition, it may lose the contextual understanding necessary for making informed decisions or executing sequential tasks effectively. This can result in suboptimal exploitation attempts or misalignment with the overall penetration testing objectives. For example, LLM may fail to provide detailed instructions on how to execute an exploit due to context loss.

Loss of Context Example

Intelligence Gathering

{Information collection steps}…
LLM: The target OS is Linux and the target IP is 192.168.238.129.

Exploitation

User: How do I execute this exploit?
LLM: The target OS and IP are needed to configure the exploit. For investigation of the unknown OS and IP, do the following: …

C3. Workflow integration. In the context of penetration testing, which involves a multi-stage pipeline of interconnected tasks, integrating an LLM introduces several challenges related to output quality control and stateful working memory management.

1) Output Quality Control: Ensuring that the LLM's output is formatted in a way that downstream modules can parse easily is crucial for the smooth operation of the entire penetration testing pipeline. This requires the LLM to generate output in a structured format that adheres to predefined standards or protocols, making it easier for subsequent modules to process and utilize the information effectively. Additionally, maintaining high content quality is essential. Before passing its output to downstream modules, the LLM should conduct validation checks to ensure the accuracy, completeness, and relevance of the generated information. LLMs may suffer from the hallucination problem, producing irrelevant or incorrect answers. Implementing robust quality control is necessary to mitigate the risk of propagating errors or misleading data through the pipeline, thereby reducing the likelihood of a single point failure disrupting the entire testing process.

2) Stateful Working Memory Management: Each stage of penetration testing often requires different sets of stateful working memory, encompassing information such as discovered vulnerabilities, selected exploits, target environment details, and ongoing session contexts. The challenge lies in enabling smooth transitions of this working memory between tasks and sessions. If the LLM cannot retain and switch between continuous stateful memory throughout the penetration testing process, it can disrupt the flow and coherence of the testing sequence. For example, if the LLM fails to retain the progress made in exploit execution after obtaining necessary information from the target environment details working memory to proceed, it may lead to restarting the exploit execution from the beginning. This redundancy can delay progress and impact the overall thoroughness and effectiveness of the testing. However, current LLMs do not inherently support such working memory management within and between sessions, posing a significant challenge in achieving seamless integration across the penetration testing pipeline.

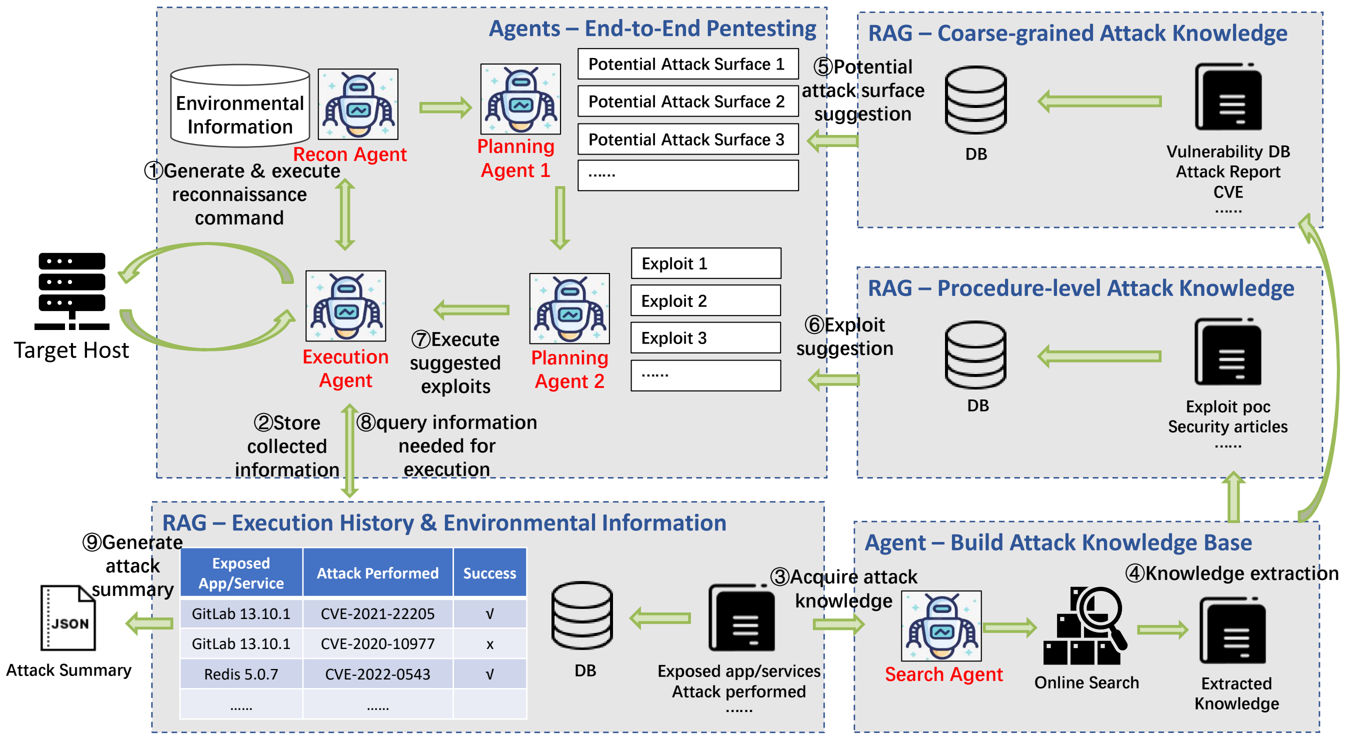
2.3 LLM Techniques for Overcoming Challenges

The rapid advancement of LLM studies has introduced a new level of intelligence and automation capabilities, significantly enhancing penetration testing performance. Various LLM techniques can be applied to different stages of pentesting to improve efficiency and effectiveness, addressing the challenges mentioned in §2.2.

LLM agents, which are LLMs equipped with additional tools, extend the functionalities of traditional models. These agents can be beneficial in all stages of pentesting by performing tasks that traditionally required human intervention, such as text analysis and code debugging. With the right tools, an LLM agent can search for and learn penetration testing knowledge online, thus addressing the challenge of limited pentesting knowledge (**C1**). To fully leverage an LLM agent's capabilities, it is essential to provide an appropriate system message that defines the agent's basic profile, including its capabilities, limitations, output format, and additional specifications [26].

Retrieval-Augmented Generation (RAG) enhances LLMs by allowing them to utilize external data for generating responses. This technique involves three main stages: indexing, retrieval, and response synthesis. Initially, the dataset is indexed for efficient retrieval. Upon receiving a query, RAG retrieves relevant information from the indexed dataset and combines it with the original query before sending it to the LLM for response synthesis. RAG effectively addresses the challenges of short-term memory (**C2**) and stateful working memory management (**C3.2**) by enabling users to maintain long-term memories that can be dynamically queried and stored.

The chain-of-thought (CoT) technique significantly improves the ability of large language models to perform complex reasoning [46]. By guiding the LLM to follow a logical sequence of steps, this method enhances the model's problem-solving capabilities.


Agents – End-to-End Pentesting
Environmental Information
Recon Agent
①Generate & execute reconnaissance command
Planning Agent 1
Potential Attack Surface 1
Potential Attack Surface 2
Potential Attack Surface 3
......
Target Host
Execution Agent
⑦Execute suggested exploits
Planning Agent 2
Exploit 1
Exploit 2
Exploit 3
......
②Store collected information
⑧query information needed for execution
RAG – Coarse-grained Attack Knowledge
⑤Potential attack surface suggestion
DB
Vulnerability DB
Attack Report
CVE
......
RAG – Procedure-level Attack Knowledge
⑥Exploit suggestion
DB
Exploit poc
Security articles
......
RAG – Execution History & Environmental Information
⑨Generate attack summary
JSON
Attack Summary
DB
Exposed app/services
Attack performed
......
Agent – Build Attack Knowledge Base
③Acquire attack knowledge
④Knowledge extraction
Search Agent
Online Search
Extracted Knowledge

Exposed App/Service	Attack Performed	Success
GitLab 13.10.1	CVE-2021-22205	√
GitLab 13.10.1	CVE-2020-10977	x
Redis 5.0.7	CVE-2022-0543	√
......	......	

Figure 1: An overview of the components in PentestAgent

Role-playing [21] ask the LLM to impersonate an imaginary character, allowing LLM to operate with clear objectives and boundaries, thereby enhancing their efficiency and effectiveness.

Self-reflection techniques, where the LLM summarizes its past mistakes into long-term memory to avoid similar errors in subsequent communications, have proven useful for learning complex tasks over a handful of trials [40].

Structured output techniques can save time spent on iterative prompt testing and ad-hoc parsing, reducing overall LLM inference costs and latency, as well as developers' effort. Additionally, structured outputs ensure smooth integration with downstream processes and workflows [23].

Together, chain-of-thought, role-playing, self-reflection, and structured output techniques significantly improve the quality of LLM output, effectively addressing the output quality control challenge (**C3.1**).

3 System Design

3.1 System Overview

As shown in Fig. 1, PentestAgent comprises four major components: the **reconnaissance agent**, the **search agent**, the **planning agent**, and the **execution agent**. These agents collaborate to perform the three main stages of penetration testing.

Intelligence Gathering: ① Upon receiving user input specifying the target, the reconnaissance agent initiates the penetration testing process by gathering environmental information about the target host. The reconnaissance agent generates and executes reconnaissance commands, aiming to collect comprehensive environmental data from the target host. ② The reconnaissance agent then analyzes the execution results and compiles a summary of the target environment, which is stored in a designated environmental information database.

Vulnerability Analysis: Next, the search and planning agents work together to perform the vulnerability analysis. ③ The search agent queries the environmental information database to retrieve a list of services and applications exposed on the target host. ④ Guided by these services and applications, the search agent searches for potential attack surfaces and procedures and saves them in separate databases. ⑤ The planning agent first leverages the RAG techniques to find a list of potential attack surfaces. ⑥ Subsequently, the planning agent uses these identified attack surfaces to determine suitable exploits for the target environment.

Exploitation: ⑦ Finally, the execution agent attempts to execute these attack plans on the target host. ⑧ The execution agent communicates with the environmental information database to obtain the necessary information for executing the exploits. It also debugs any execution errors by modifying the code or executing additional commands to gather more information. ⑨ All execution history is stored in a database and can be used to generate a comprehensive penetration testing report.

This structured and automated framework aims to streamline the penetration testing process, enhancing efficiency and reducing the manual effort required.

3.2 Reconnaissance Agent

The reconnaissance agent takes a specified target as input and interacts with it to collect detailed information, ultimately generating

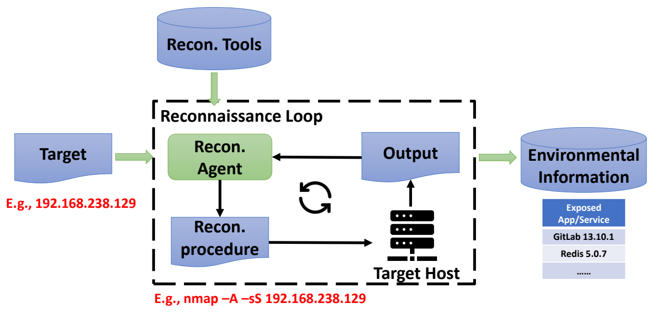


Figure 2: Reconnaissance agent workflow

Reconnaissance System Message (Simplified)

Role-play

You're an excellent cybersecurity penetration tester assistant. Guide the tester . . .

Chain-of-Thought

Use Nmap to identify exposed ports, then use relevant tools in Nmap to analyze these ports on the target host . . .

RAG

You should use your query tool to learn about available reconnaissance tools . . .

Structured Output

You should always respond in valid JSON format with the following fields: {FORMAT SPEC.} . . .

a summary of the environmental information as the output. As illustrated in Fig. 2, the process begins when a target is provided to the reconnaissance agent. The agent operates in a self-iterating loop, generating reconnaissance commands to gather information from the target and analyzing the results of these commands until the best efforts have been made. Once the reconnaissance loop concludes, the agent summarizes its findings and stores them in a database.

The reconnaissance agent adheres to a general workflow defined with expert knowledge to perform the reconnaissance task. It determines specific procedures or tools to use with the help of external knowledge supported by the RAG framework. To achieve our desired workflow, we carefully design the system messages and prompts for the reconnaissance agent, implementing the following techniques to overcome the challenges mentioned in §2.2.

Role-playing has proven effective in bypassing the safety policies enforced by the LLM [10]. Thus, we ask the reconnaissance agent to act as a penetration tester assistant to validate its reconnaissance behaviors.

We use Chain-of-Thought (CoT) to break down complex tasks into several sub-tasks and construct an effective reconnaissance workflow to reduce hallucination. Since the reconnaissance workflow involves a self-iterating loop, it is important to specify a stop condition to avoid the agent getting into an infinite loop. Using CoT effectively enforces the stop condition by specifying the tasks to complete before stopping.

Retrieval-Augmented Generation (RAG) allows the reconnaissance agent to retrieve relevant information from a database containing documentation of various reconnaissance tools, enabling it to use up-to-date tools for effective information collection. For example, it can use web application fingerprinting tools with open-source fingerprinting databases like ObserverWard [1] to aid in reconnaissance. Furthermore, RAG allows the reconnaissance agent to store collected environmental information in a database for later use, addressing the short-term memory issue.

The reconnaissance agent analyzes previous execution results and generates the next command to execute in each communication. To enforce adherence to the penetration testing pipeline and ensure a smooth transition to subsequent steps, we use structured output, asking the reconnaissance agent to respond using a specified format.

After the reconnaissance agent determines that it should stop the reconnaissance loop, it summarizes the reconnaissance results and stores them in a database to make the short-term reconnaissance memory persistent. The following prompt generates a structured output of the reconnaissance summary. Specifying the output structure and providing a comprehensive example guides the agent to output relevant information and reduces hallucination.

3.3 Search Agent

The search agent takes target services and applications as input and stores relevant attack knowledge into databases as output. As illustrated in Fig. 3, the search agent performs two rounds of hierarchical online search for relevant information. In the first round, it searches and analyzes the results to extract potential attack surfaces relevant to the target. In the subsequent round, it uses the identified potential attack surfaces as a guide to search and analyze procedure-level attack knowledge. The potential attack surfaces and procedure-level attack knowledge are stored in two separate databases for future use.

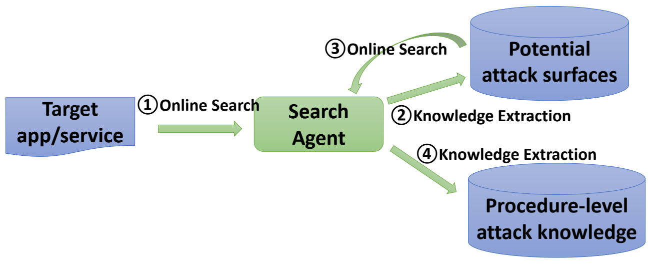


Figure 3: Search agent workflow

The online search module is customizable and extensible. We have implemented several search functions, including general searches on Google, vulnerability-specific searches on databases like Snyk [39] and AVD [8], and searches in exploit code repositories such as GitHub and ExploitDB. In our hierarchical search workflow, we use Google and vulnerability database searches to identify potential attack surfaces in the first round and then employ Google and code repository searches to find exploit implementation details in the second round.

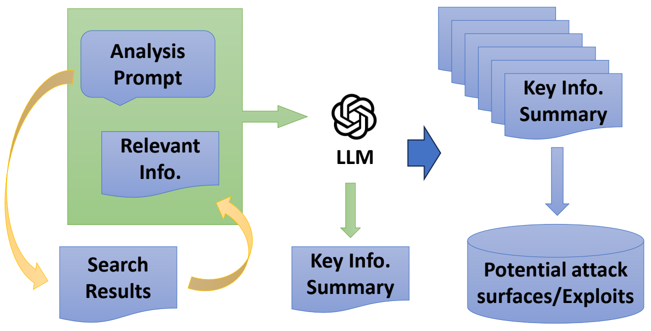


Figure 4: RAG workflow for search result summarization. The yellow arrows denote the retrieval process, and the green arrows denote the generation process.

After each round of online searches, the search agent analyzes the results. However, indexing and storing information from raw search results is inefficient. Therefore, we leverage RAG-based question-answering to extract key information from the raw search results and use the extracted knowledge to build a more relevant and concise database. As elucidated in Fig. 4, given the analysis prompt, the RAG framework will first retrieve relevant segments of information from the search results. Then, it sends the analysis prompt with the retrieved information as context to LLM as the question, and the LLM will analyze the information in the context to help answer the queries in the analysis prompt and generate a comprehensive summary for the search results containing the key information we are looking for. Finally, the summaries of individual documents are gathered to build a potential attack surface or exploit database in Fig. 3.

For the first round of searching for potential attack surfaces, we use the following prompt to extract knowledge from individual search results. Specifically, we ask for relevancy and key information about vulnerabilities, such as CVE numbers, as well as other keywords or URLs that can lead to more detailed information. We also ask the search agent to output the analysis results in a structured format for subsequent processing.

Potential Attack Surface Analysis Prompt (Simplified)

RAG & CoT

Generate a concise summary of the document to answer the following questions:
1) Does this document describe vulnerabilities targeting a particular service or app; if so, what is the relevant service/app version?
2) Provide information that can be used to search for the exploit of the vulnerabilities.

Structured Output

You should always respond in valid JSON format with the following fields: {FORMAT SPEC.} …
For example, the response looks like this: {OUTPUT FORMAT EXAMPLE}

After analyzing all individual search results, the search agent summarizes them into a structured output for subsequent parsing and storing.

Search Results Summary Prompt

List ALL CVE numbers, URLs, keywords, and their applicable version relevant to exploit the vulnerabilities of {APP}. The results should be presented in valid JSON format with the following fields: {FORMAT SPEC.} …

Similarly, for the second round of searching for procedure-level exploit details, the search agent analyzes individual search results using RAG and CoT. First, it checks whether the repository contains a relevant exploit. Then, it extracts key information such as applicable service or application versions and prerequisites for running the exploit. While the first round of analysis mainly focuses on the LLM's text summarization capability, the second round relies on the LLM's code analysis capability to determine whether the code functions as an exploit and the dependencies required to execute it.

From our initial attempts, we found that the LLM is not familiar with software versioning. Therefore, we added a paragraph containing descriptions and examples to demonstrate how to handle software versions as few-shot learning. We use the following prompt to extract the desired information.

Exploit Procedure Analysis Prompt (Simplified)

RAG & CoT

Give a concise summary of the entire repository to answer the following questions:
1) whether this repository contains an exploit targeting a particular service or app;
2) What effect does the exploit have? Use one phrase to summarize the effect (e.g., remote command execution);
3) What relevant service/app version can this exploit be applied to?

Few-shot Learning

Note the app version is typically formatted as x.y.z. Explicitly state the version with the following formats …
4) what are the requirements to run this exploit? (e.g., OS, library dependencies, etc.)

Structured Output

You should always respond in valid JSON format with the following fields: {FORMAT SPEC.} …
For example, the response looks like this: {OUTPUT FORMAT EXAMPLE}

After the penetration testing knowledge is extracted by the search agent, it is stored in a hierarchical tree structure as shown in Fig. 5. The hierarchical tree-structured penetration testing knowledge base allows efficient searching and systematic management of penetration testing knowledge.

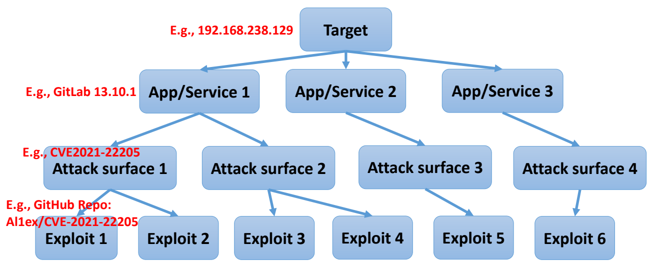


Figure 5: Hierarchical-tree structured pentesting knowledge database

3.4 Planning Agent

The planning agent takes detected services and applications from the reconnaissance agent as input and generates an exploitation plan as output. As shown in Fig. 1, the planning agent leverages RAG and the pentesting knowledge base (Fig. 5) to first generate a list of potential attack surfaces relevant to the services and applications. Then, the planning agent follows a similar process to generate a list of exploits.

The planning agent uses the service or application as a key to find the relevant database for potential attack surfaces and retrieves these from the database according to the service or application version and vulnerability types. The planning agent makes suggestions for attack surfaces based on the application version and categorizes the attack surfaces by vulnerability types. We designed the following prompt to generate a list of potential attack surfaces given a particular service or application.

Attack Surface Suggestion Prompt (Simplified)

List out all vulnerabilities ranked by confidence that can be used to exploits …
The results should be presented in valid JSON format with the following fields: {FORMAT SPEC.} …

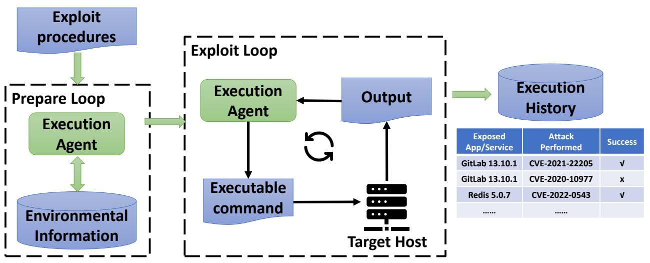
The planning agent uses the attack surface to find the relevant database for exploits and retrieves exploit details from the database according to the service or application version and exploit effects (e.g., remote code execution, authentication bypass). The planning agent then makes suggestions for exploits based on the application version and categorizes the exploits by exploit effects. We designed the following prompt to generate a list of exploits for each potential attack surface.

Exploit Suggestion Prompt (Simplified)

List out paths of all relevant repositories ranked by the confidence that contain exploits … The results should be presented in valid JSON format with the following fields: {FORMAT SPEC.} …

3.5 Execution Agent

The execution agent takes the exploit details as input and attempts to execute the exploits on the target automatically, ultimately generating an exploitation summary as output. The execution agent follows the order suggested by the planning agent. As illustrated in Fig. 6, each exploit execution can be divided into two stages: the preparation stage and the exploitation stage.



Exposed App/Service	Attack Performed	Success
GitLab 13.10.1	CVE-2021-22205	√
GitLab 13.10.1	CVE-2020-10977	x
Redis 5.0.7	CVE-2022-0543	√
……	……	

Figure 6: Execution agent workflow

During the preparation stage, the execution agent analyzes the exploit details to determine the requirements for successful execution, such as the parameters needed. It then queries the database containing environmental information to obtain the necessary information. The following system message guides the execution agent in fulfilling its duty. To bypass the safety mechanisms that may prevent the process while the execution agent tries to execute the exploit, we use role-playing to make it act as a cybersecurity penetration tester.

To effectively find the information needed to execute the exploit, we employ the CoT technique to guide the execution agent to first identify all the parameters and then determine the information needed for each parameter. During this analysis, the execution agent retrieves relevant information from the exploit details using RAG to provide the context. Finally, the execution agent outputs the required information in a structured JSON format, allowing it to proceed to the subsequent step of requesting information from the environmental information database.

Execution Preparation System Message (Simplified)

Role-play

You're an excellent cybersecurity pentesting assistant …

CoT

Analyze the exploit to answer the following questions:
1) What parameters do you need to fill in to execute this exploit successfully?
2) What information do you need to fill in these parameters?

RAG

You should use your query tool to learn the details about the exploit …

Structured Output

You should always respond in valid JSON format with the following fields: {FORMAT SPEC.} …

After obtaining a list of needed information, the execution agent uses the following prompt to query the environmental information database to fill in the information.

Execution Information Query Prompt (Simplified)

Based on the known information, try to provide the information listed here. {INFO NEEDED ...}

CoT

You should examine the information needed one by one. For each piece of information needed, you should ...

RAG

You should use your query tool to learn about the target environment ...

Structured Output

The results should be presented in valid JSON format with the following fields: {FORMAT SPEC.} ...

After getting the response containing the requested information, the execution agent's system message is updated to transition to the exploitation stage.

Execution Exploit System Message (Simplified)

Your next task is to provide step by step guide for executing the exploit and debugging the errors encountered ...

RAG

You should use the tool to learn the code and README of the exploit to figure out how to properly execute it.

Specifications

Avoid commands that require user interactions ...

Self-reflection

When the results indicate an error, you should ...

Structured Output

You should always respond in valid JSON format with the following fields: {FORMAT SPEC.} ...

During the exploitation stage, the execution agent uses RAG to obtain details of the code execution, breaks down the execution plan, and generates a step-by-step execution guide. Similar to the reconnaissance agent, the execution agent engages in iterative loops to execute the exploit.

When errors are encountered during exploit execution, proper error handling is required. To guide the execution agent in debugging errors, we employ the self-reflection technique. The execution agent analyzes and fixes errors based on the code and error message while concurrently documenting the error history for future reference to avoid repeating the error. This iterative process ensures continual refinement and optimization of our automated pentesting system.

4 Evaluation

In this section, we present the benchmark established for evaluating automated penetration testing frameworks and discuss the evaluation results. We address the following research questions (RQs) in our evaluation:

RQ1. Effectiveness. What's the success rate of finishing the whole penetration testing process automatically?

RQ2. Completion level. What's the completion level of individual penetration testing stages that can be automatically finished?

RQ3. Efficiency. How much time and API cost are needed for PentestAgent to complete a penetration testing task?

4.1 Evaluation Setup

4.1.1 Benchmark Dataset. The benchmark dataset should be easily accessible and include a diverse set of tasks with varying difficulty levels to evaluate the automated penetration testing framework. Accessibility is essential for a good benchmark; otherwise, it prevents the whole community from using it. The tasks in the benchmark should involve exploiting various vulnerabilities targeting different services and applications to mimic real-world penetration testing scenarios. More importantly, the tasks should have appropriate difficulty labels to reflect how well the system under test can handle tasks of different difficulty levels, helping researchers identify the strengths and weaknesses of the system.

Several platforms can serve as the dataset of the benchmark, such as HackTheBox [16], OWASP Benchmark [32], VulnHub [45], and VulHub [44]. However, HackTheBox lacks accessibility to the public, requiring a VIP subscription to access most of its test machines, which creates a burden for using the benchmark. OWASP Benchmark and VulnHub contain thousands of target testing environments, covering a wide range of real-world penetration testing scenarios. However, setting up these environments for testing requires significant human effort. Furthermore, they do not provide a difficulty level reference for their test cases, necessitating manual effort to determine the difficulty level for each test case.

Finally, we chose VulHub as our benchmark dataset. VulHub provides an open-source collection of over a hundred pre-built vulnerable Docker environments, which has been widely recognized and utilized in penetration testing practices. The container-based platform supports infrastructure as code (IaC), making it easy to set up the testing environments. Besides, Docker containers provide sufficient isolation for penetration testing. Moreover, most vulnerable environments in VulHub are constructed to reproduce a particular Common Vulnerabilities and Exposures (CVE) [27]. Each vulnerable environment is associated with a CVE number, which allows us to use metrics associated with CVE numbers to learn about the properties of each vulnerable environment. Specifically, we learn about the difficulty of vulnerability exploits through the Common Vulnerability Scoring System (CVSS)[9] and learn about how realistic the vulnerable environment is via the Exploit Prediction Scoring System (EPSS)[31]. We elaborate on how we construct the benchmark dataset in §A.1 in the appendix.

As a result, we compiled a benchmark comprising 67 penetration testing targets, spanning 32 CWE (Common Weakness Enumeration) categories as shown in Fig.12 in the appendix. Within our benchmark, there are 50 targets with easy exploitability difficulty, 11 with medium exploitability difficulty, and 6 with hard exploitability difficulty. This diverse and realistic collection of vulnerable environments ensures a comprehensive assessment.

4.1.2 Metric. To answer our research question, we design metrics to evaluate the effectiveness and efficiency of PentestAgent. These metrics are essential for assessing the performance of the automated penetration testing framework.

We measure the effectiveness of PentestAgent by determining whether all three stages of penetration testing are completed successfully and automatically. We define successful completion as follows: given a target IP, PentestAgent can automatically perform a functional exploit on the vulnerable environments.

Some penetration tests may be partially successful and require human assistance. However, failure in a previous penetration testing stage will affect the subsequent stages. To better understand the effectiveness of each component in PentestAgent, we measure the completion level at the stage level. This involves assessing the penetration testing stages that can be automatically completed, assuming the preceding stages have been successful. The completion criteria for each stage are defined as follows. The information gathering stage is considered complete if the target application is successfully identified by PentestAgent. The vulnerability analysis stage is marked as complete when PentestAgent identifies functional exploits based on the target application. We manually verify whether the discovered exploits are effective. The exploitation stage is completed if PentestAgent can automatically and successfully execute the exploit. This stage-level evaluation provides a granular understanding of PentestAgent 's autonomy and effectiveness in progressing through the penetration testing process with minimal human assistance.

Furthermore, we measure the efficiency of PentestAgent using the time taken and the API cost incurred to complete penetration tests. The time metric evaluates the duration required for PentestAgent to complete an entire penetration test cycle, from initial reconnaissance to exploit execution. the API cost metric quantifies the computational resources consumed by the framework during the testing process. These metrics provide insights into the system's resource consumption and operational speed, which are critical for practical deployment and scalability.

4.1.3 Environment setup. The simulated vulnerable applications are hosted on a virtual machine with 2 CPU cores and 8 GB RAM, running Ubuntu 22.04 LTS. To avoid interference with the testing process, we have disabled all services that require listening on ports, such as SSH.

The attacker machine is also hosted on a virtual machine with 16 CPU cores and 16 GB RAM, running Kali Linux 2024.1. The attacker machine includes all the pre-installed tools available in Kali Linux, with no additional tools installed.

The victim machine and the attacker machine maintain network connectivity via NAT. The vulnerable containers on the victim machine are created with the network parameter set to the victim machine's IP, allowing the attacker machine to directly access the vulnerable environments hosted in the victim machine's containers. This setup ensures the attacker can simulate real-world network conditions when attempting to exploit the vulnerabilities.

4.1.4 LLM models. We utilize the OpenAI GPT-3.5 and GPT-4 models, representing state-of-the-art LLM technology. These models were accessed via the OpenAI APIs. Specifically, we used the gpt-3.5-turbo-0125 model, which has a context window size of 16,385 tokens and is trained with data up to September 2021. The pricing for this GPT-3.5 model is \$0.50 / 1M input tokens and \$1.50 / 1M output tokens. Additionally, we used the GPT-4o model, which features a context window size of 128,000 tokens and is trained with data up to October 2023. The pricing for this GPT-4 model is \$5.00 / 1M input tokens and \$15.0 / 1M output tokens.

Our automated penetration testing framework does not extensively rely on LLMs' inherent capabilities and knowledge to plan and perform most of the tasks. Instead, we designed a robust working pipeline for the penetration testing process, using LLMs as tools for specific and well-defined tasks, such as text summarization and code analysis. Given this design, we do not experiment with many other LLM models with our framework. This approach ensures consistency and reliability, as the framework's performance is less dependent on the specific strengths and learned knowledge of different LLM models. By including GPT-3.5 and GPT-4, we aim to demonstrate the variability and invariability of our framework when utilizing models with varying capabilities, ensuring it is effective across different LLM configurations. In §5.2, we discuss the compatibility of PentestAgent with various LLMs.

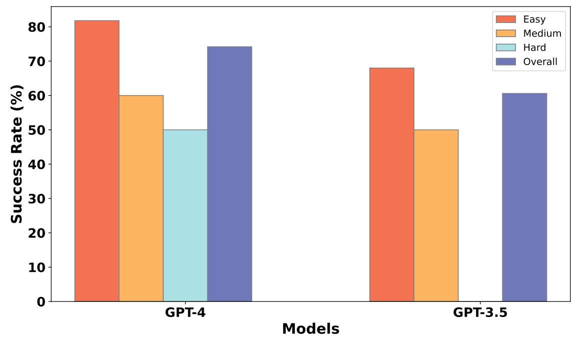


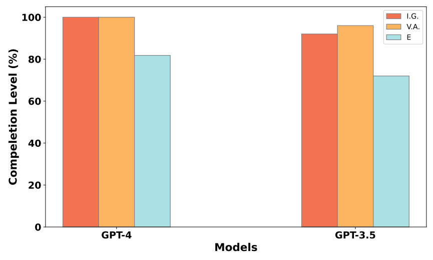
Figure 7: Success rate on penetration testing tasks

4.2 Effectiveness of the Entire Framework

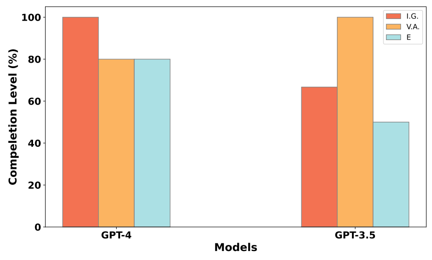
We investigate the effectiveness of PentestAgent by its success rates of completing the penetration testing process. Fig. 7 shows the success rates of exploiting vulnerabilities categorized by difficulty levels and overall performance across different models. The GPT-4 model demonstrated a 74.2% overall success rate in completing automated penetration testing tasks, outperforming the GPT-3.5 model, which achieved a 60.6% success rate. Both models consistently achieved success rates above 60%, affirming the effectiveness of PentestAgent in establishing an automated penetration testing pipeline.

While the GPT-4 model showed a higher overall success rate compared to GPT-3.5, the difference between their performances was not substantial. This suggests that our framework does not rely heavily on LLMs' general knowledge and capabilities alone.

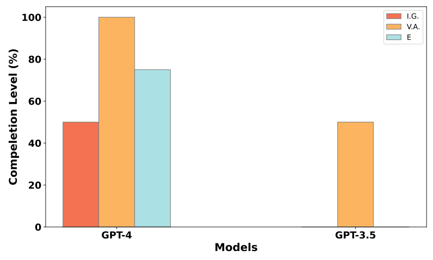
Notably, the GPT-3.5 model struggled particularly with hard penetration testing tasks, achieving no success in the hardest category. This disparity likely stems from the inherent differences in context window size and learned knowledge between the models, impacting their ability to handle complex reasoning required for challenging tasks.



(a) Easy tasks



(b) Medium tasks



(c) Hard tasks

Figure 8: Completion level of penetration testing stages on different difficulty of tasks. I.G. denotes the intelligence gathering stage, V.A. denotes the vulnerability analysis stage, and E denotes the exploitation stage.

4.3 Completion level of Penetration Testing Stages

We further delve into PentestAgent by examining its completion level on individual penetration testing stages. Fig. 8 illustrates the completion levels of penetration testing stages across various difficulty categories and models. The PentestAgent with GPT-4 model exhibited robust performance across all stages and difficulty levels, demonstrating its ability to handle challenging tasks. In easy tasks, GPT-4 model achieved full completion in the intelligence gathering and vulnerability analysis stages, maintaining a high exploitation stage completion rate of 81.8%. In medium difficulty tasks, GPT-4 continued its strong performance in all stages, despite some drop in the vulnerability analysis stage. However, in harder tasks, the GPT-4 model encountered challenges, with completion rates dropping to 50% in the intelligence gathering stage, indicating potential limitations in handling complex scenarios that require better reconnaissance tools and advanced reasoning capabilities.

Conversely, the GPT-3.5 model demonstrated varying degrees of success across different difficulty levels. It achieved very good performance in easier tasks, achieving high completion rates in intelligence gathering (92%) and vulnerability analysis (96%) stages, although with a slightly lower completion rate in exploitation (72%). In medium difficulty tasks, while maintaining a perfect record in vulnerability analysis (100%), it encountered difficulties in both intelligence gathering (66.7%) and exploitation (50%) stages, indicating challenges in complex reasoning required for effective reconnaissance and exploitation. Notably, in hard tasks, the GPT-3.5 model struggled significantly, achieving no completions in the intelligence gathering and exploitation stages, underscoring limitations in handling advanced penetration testing tasks that demand extensive contextual understanding and reasoning.

Overall, the findings suggest that while both models can automate substantial portions of penetration testing tasks, the GPT-4 model consistently outperforms the GPT-3.5 model, especially in more challenging scenarios.

4.4 Efficiency

To evaluate PentestAgent's efficiency, we measure the time spent and cost needed to perform penetration tests. Fig. 9 presents an overview of the average time and cost of conducting penetration testing tasks using the GPT-4 and GPT-3.5 models. Across all stages (intelligence gathering, vulnerability analysis, and exploitation), the GPT-4 model required, on average, 346.7 seconds for intelligence gathering, 780.9 seconds for vulnerability analysis, and 52.3 seconds for exploitation. The cumulative time spent on all tasks totaled 1164.7 seconds, with an average cost of $2.66 per task. In contrast, the GPT-3.5 model exhibited lower time requirements and cost: 212.9 seconds for intelligence gathering, 698.8 seconds for vulnerability analysis, and 58.6 seconds for exploitation, resulting in a total average time of 1009.8 seconds and a cost of $1.09 per task.

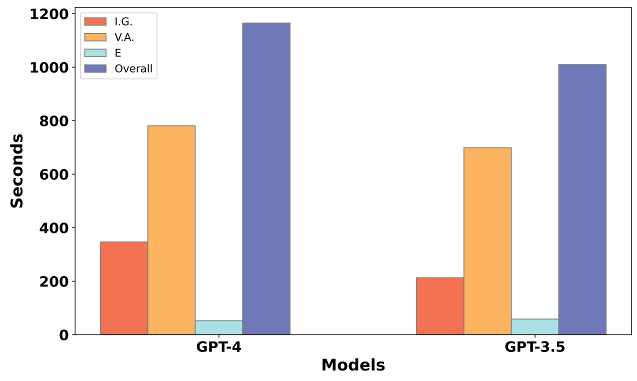


Figure 9: Average time spent on penetration testing tasks

The analysis underscores the trade-off between efficiency and performance. While the GPT-4 model demonstrates superior capabilities in automated penetration testing, it does so at a higher operational cost.

4.5 Failure Analysis

We analyzed failure cases encountered during our evaluation and identified a few representative failure scenarios. As illustrated in Fig. 8, most failures occurred during the intelligence gathering and exploitation stages.

In the intelligence gathering stage, PentestAgent occasionally fails to recognize services or applications with the appropriate level of granularity. For instance, our evaluation revealed that PentestAgent struggled to detect components like PHPMailer, PHPUnit, and Ghostscript. These are not standalone applications but rather plugins or components running on web servers. Tools like Nmap can identify the underlying web server frameworks, such as Nginx, but fail to enumerate these components. To address this limitation, PentestAgent allows integration of additional web component fingerprinting tools and specialized libraries to more accurately detect and categorize such web components.

At the exploitation stage, PentestAgent can encounter failures due to several challenges: requiring additional knowledge, needing user interaction, or experiencing LLM hallucinations.

Requiring Additional Knowledge: Certain exploits demand a level of domain-specific knowledge that may exceed the capabilities of an LLM agent. For example, exploiting Samba server 4.6.3 (CVE-2017-7494) assumes the attacker has prior knowledge of credentials (username and password) to establish an SMB connection. Moreover, exploiting JBoss (CVE-2017-12149) requires expertise in using the "ysoserial" tool to craft payloads for exploiting unsafe Java object deserialization. These limitations can be overcome by integrating a human-in-the-loop design, where human experts can provide the additional knowledge or context required. Thanks to PentestAgent's modular structure and its task-decomposition pipeline, human experts can easily intervene at any point in the testing process to assist with complex tasks.

Requiring User Interaction: Some exploits require user interactions that are typically performed manually, such as file uploads via web user interfaces. For instance, exploiting elFinder (CVE-2021-32682), an open-source file manager for web environments, involves manually creating and uploading an archive file. Similar to the mitigation method in the previous scenario, PentestAgent allows the human user to step in at any penetration testing stage to assist tasks requiring user interaction. Furthermore, the recent advancements in intelligent agents like AutoGPT [3] offer a promising solution by mimicking human actions for complex tasks. By integrating such intelligent agents, PentestAgent could automate these user interactions, significantly enhancing its capabilities in handling tasks traditionally performed by human testers.

LLM Hallucination: Another challenge is LLM hallucination, where the model generates incorrect or misleading information. This issue can be particularly problematic during the exploitation phase, as one hallucination can lead to a cascade of errors in subsequent steps. For example, if the execution agent fails to generate the correct commands or input parameters, it may mistakenly assume the exploit has bugs, leading it down an incorrect debugging path that will never succeed. We employ several strategies to mitigate hallucinations. First, we reduce the randomness of LLM outputs by setting the model's temperature to zero and attempting to execute the exploit multiple times. We also implement several stop conditions to prevent unintended consequences of hallucination, such as getting stuck in infinite loops or executing unintended actions. These stop conditions include hard-coded limits on the number of execution attempts and prompt-based conditions like "stop when you see the same error again." Additionally, the attack knowledge base usually contains multiple exploits for the same vulnerability, allowing PentestAgent to attempt different approaches until a functional exploit is found.

4.6 Comparison with PentestGPT

We conducted a comparison of the effectiveness and efficiency of PentestAgent against PentestGPT. Unlike PentestAgent, PentestGPT requires human involvement for feedback and decision-making throughout the penetration testing process. Thus, we compare their performance using case studies. To create a fair evaluation, we randomly selected five vulnerabilities, including two categorized as easy, two as medium, and one as hard. We enlisted an undergraduate student with limited penetration testing experience to act as the human component required by PentestGPT. The student followed PentestGPT's guidance without applying external knowledge for decision-making or task completion.

Under this testing condition, PentestGPT was unable to fully exploit any of the five vulnerabilities. In contrast, PentestAgent successfully completed exploitation in three out of the five cases. To provide a further comparison, we examined the performance of PentestGPT in individual penetration testing stages. PentestGPT was able to recognize the target application in only one of the five cases, whereas PentestAgent correctly identified the target application in four of the five cases. In the information gathering stage, PentestGPT spent an average of 826.25 seconds and required 7.4 rounds of interaction between the tester and the system. By comparison, PentestAgent completed information gathering in under 400 seconds on average, with no need for human-tester interaction. Given the correct target application information, PentestGPT successfully guided the exploitation of only one vulnerability. PentestAgent, on the other hand, automatically exploited four vulnerabilities, including the hard case, once the target application was identified.

These results demonstrate that PentestAgent significantly outperforms PentestGPT in both effectiveness and efficiency, accomplishing penetration testing tasks autonomously without requiring human assistance.

5 Discussion

5.1 Benchmark Coverage

Our evaluation was conducted using a benchmark dataset comprising known vulnerabilities, which raises questions about the practicality in real-world scenarios. Firstly, it is important to recognize that known vulnerabilities pose significant risks. Many organizations and institutions struggle with timely patching practices, contributing to vulnerable and outdated components ranking 6th on the OWASP Top 10 Web Application Security Risks. [33] Additionally, while our benchmark dataset features known vulnerabilities, we selected environments based on their Exploit Prediction Scoring System (EPSS) scores. These scores reflect the likelihood of a vulnerability being exploited in real-world scenarios. The dataset's mean EPSS score is 79.58, with a median of 97.19, indicating that the vulnerabilities represented are highly likely to exist and be exploitable in practical settings. Moreover, finding open datasets containing zero-day or even one-day vulnerable environments remains challenging. By focusing on known vulnerabilities with high EPSS scores, our evaluation ensures that PentestAgent operates within a realistic and credible context, assessing its effectiveness in addressing vulnerabilities that pose genuine risks to cybersecurity.

5.2 Compatibility with Other LLMs

In light of data privacy concerns, PentestAgent has been designed to be compatible with a range of large language models (LLMs). Beyond commercial models like OpenAI's pre-trained versions, we have also experimented with open-source LLMs such as Mistral and Llama 3. Our experiments confirm that PentestAgent functions effectively with SOTA open-source models, although the system's efficiency can vary depending on the response time of the specific model in use. As we discussed in §4.5, the failure cases of PentestAgent are rarely caused by the capabilities of the LLM. Therefore, we believe that the overall effectiveness of PentestAgent is unlikely to be significantly impacted by the choice of LLM.

6 Conclusion

This paper presents PentestAgent, a novel LLM-based framework for automated penetration testing designed to address the limitations of existing frameworks: limited pentesting knowledge and insufficient automation. By leveraging a multi-agent architecture and incorporating various LLM techniques like Retrieval Augmented Generation (RAG), PentestAgent enhances the penetration testing process through improved knowledge integration and automation.

Our comprehensive benchmark, based on VulHub's vulnerable Docker environments, provided a comprehensive test bed of PentestAgent. The evaluation results demonstrate that PentestAgent achieves satisfying performance in task completion and overall efficiency.

References

[1] 0x727. 2024. ObserverWard. https://github.com/0x727/ObserverWard

[2] Paul Ammann, Duminda Wijesekera, and Saket Kaushik. 2002. Scalable, graph-based network vulnerability analysis. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*. 217–224.

[3] AutoGPT. 2024. AutoGPT. https://github.com/Significant-Gravitas/AutoGPT

[4] Mark S Boddy, Johnathan Gohde, Thomas Haigh, and Steven A Harp. 2005. Course of Action Generation for Cyber Security Using Classical Planning.. In *ICAPS*. 12–21.

[5] Jinyin Chen, Shulong Hu, Haibin Zheng, Changyou Xing, and Guomin Zhang. 2023. GAIL-PT: An intelligent penetration testing framework with generative adversarial imitation learning. *Computers & Security* 126 (2023), 103055.

[6] Rapid7 Global Consulting. 2019. Under the Hoodie: Lessons from a Season of Penetration Testing. https://www.rapid7.com/research/reports/under-the-hoodie-2019/ Accessed: 2024-06-19.

[7] Rapid7 Global Consulting. 2020. Under the Hoodie: Lessons from a Season of Penetration Testing. https://www.rapid7.com/research/reports/under-the-hoodie-2020/ Accessed: 2024-06-27.

[8] Alibaba Could. 2024. Vulnerability DB. https://avd.aliyun.com/

[9] National Vulnerability Database. 2024. Common Vulnerability Scoring System Calculator. https://nvd.nist.gov/vuln-metrics/cvss/v3-calculator

[10] Gelei Deng, Yi Liu, Yuekang Li, Kailong Wang, Ying Zhang, Zefeng Li, Haoyu Wang, Tianwei Zhang, and Yang Liu. 2023. Jailbreaker: Automated jailbreak across multiple large language model chatbots. *arXiv preprint arXiv:2307.08715* (2023).

[11] Gelei Deng, Yi Liu, Víctor Mayoral-Vilches, Peng Liu, Yuekang Li, Yuan Xu, Tianwei Zhang, Yang Liu, Martin Pinzger, and Stefan Rass. 2024. PentestGPT: Evaluating and Harnessing Large Language Models for Automated Penetration Testing. In *33rd USENIX Security Symposium (USENIX Security 24)*. 847–864.

[12] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. 2024. Large language models are edge-case generators: Crafting unusual programs for fuzzing deep learning libraries. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.

[13] Matthew Denis, Carlos Zena, and Thaier Hayajneh. 2016. Penetration testing: Concepts, attack methods, and defense strategies. In *2016 IEEE Long Island Systems, Applications and Technology Conference (LISAT)*. IEEE, 1–6.

[14] Karel Durkota and Viliam Lisỳ. 2014. Computing Optimal Policies for Attack Graphs with Action Failures and Costs.. In *STAIRS*. 101–110.

[15] GreenBone. 2024. GreenBone OpenVAS. https://www.openvas.org/

[16] HackTheBox. 2024. Hackthebox: Hacking training for the best. https://www.hackthebox.com/

[17] Andreas Happe and Jürgen Cito. 2023. Getting pwn'd by ai: Penetration testing with large language models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2082–2086.

[18] Zhenguo Hu, Razvan Beuran, and Yasuo Tan. 2020. Automated penetration testing using deep reinforcement learning. In *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, 2–10.

[19] Leanid Krautsevich, Fabio Martinelli, and Artsiom Yautsiukhin. 2013. Towards modelling adaptive attacker's behaviour. In *Foundations and Practice of Security: 5th International Symposium, FPS 2012, Montreal, QC, Canada, October 25-26, 2012, Revised Selected Papers 5*. Springer, 357–364.

[20] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems* 33 (2020), 9459–9474.

[21] Guohao Li, Hasan Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. 2023. Camel: Communicative agents for" mind" exploration of large language model society. *Advances in Neural Information Processing Systems* 36 (2023), 51991–52008.

[22] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2023. The Hitchhiker's Guide to Program Analysis: A Journey with Large Language Models. *arXiv preprint arXiv:2308.00245* (2023).

[23] Michael Xieyang Liu, Frederick Liu, Alexander J Fiannaca, Terry Koo, Lucas Dixon, Michael Terry, and Carrie J Cai. 2024. " We Need Structured Output": Towards User-centered Constraints on Large Language Model Output. In *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*. 1–9.

[24] Puzhuo Liu, Chengnian Sun, Yaowen Zheng, Xuan Feng, Chuan Qin, Yuncheng Wang, Zhi Li, and Limin Sun. 2023. Harnessing the power of llm to support binary taint analysis. *arXiv preprint arXiv:2310.08275* (2023).

[25] Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. 2024. Large language model guided protocol fuzzing. In *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)*.

[26] Microsoft. 2024. System message framework and template recommendations for Large Language Models (LLMs). https://learn.microsoft.com/en-us/azure/ai-services/openai/concepts/system-message

[27] MITRE. 2024. CVE. https://cve.mitre.org/

[28] nmap. 2024. nmap. https://nmap.org/

[29] Jorge Lucangeli Obes, Carlos Sarraute, and Gerardo Richarte. 2013. Attack planning in the real world. *arXiv preprint arXiv:1306.4044* (2013).

[30] Forum of Incident Response and Inc. Security Teams. 2024. Common Vulnerability Scoring System v3.0: Specification Document. https://www.first.org/cvss/specification-document

[31] Forum of Incident Response and Inc. Security Teams. 2024. Exploit Prediction Scoring System (EPSS). https://www.first.org/epss/

[32] OWASP. 2024. OWASP Benchmark. https://owasp.org/www-project-benchmark/

[33] OWASP. 2024. Top 10 Web Application Security Risks. https://owasp.org/www-project-top-ten/

[34] Rapid7. 2024. Rapid7 Metasploit. https://www.metasploit.com/

[35] Mark Roberts, Adele Howe, Indrajit Ray, Malgorzata Urbanska, Zinta S Byrne, and Janet M Weidert. 2011. Personalized vulnerability analysis through automated planning. In *Working Notes for the 2011 IJCAI Workshop on Intelligent Security (SecArt)*. 50.

[36] Carlos Sarraute, Olivier Buffet, and Jörg Hoffmann. 2012. POMDPs make better hackers: Accounting for uncertainty in penetration testing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 26. 1816–1824.

[37] Carlos Sarraute, Olivier Buffet, and Jörg Hoffmann. 2013. Penetration testing== POMDP solving? *arXiv preprint arXiv:1306.4714* (2013).

[38] Carlos Sarraute, Gerardo Richarte, and Jorge Lucángeli Obes. 2011. An algorithm to find optimal attack paths in nondeterministic scenarios. In *Proceedings of the 4th ACM workshop on Security and artificial intelligence*. 71–80.

[39] Snyk Security. 2024. Snyk Vulnerability Database. https://security.snyk.io/

[40] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2024. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems* 36 (2024).

[41] The Penetration Testing Execution Standard. 2024. PTES Technical Guidelines. http://www.pentest-standard.org/index.php/PTES_Technical_Guidelines

[42] Yaroslav Stefinko, Andrian Piskozub, and Roman Banakh. 2016. Manual and automated penetration testing. Benefits and drawbacks. Modern tendency. In *2016 13th international conference on modern problems of radio engineering, telecommunications and computer science (TCSET)*. IEEE, 488–491.

[43] Tenable. 2024. Tenable Nessus. https://www.tenable.com/products/nessus

[44] Vulhub. 2024. Vulhub. https://github.com/vulhub/vulhub

[45] VulnHub. 2024. VulnHub. https://www.vulnhub.com/

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

[47] Jiacen Xu, Jack W Stokes, Geoff McDonald, Xuesong Bai, David Marshall, Siyue Wang, Adith Swaminathan, and Zhou Li. 2024. AutoAttacker: A Large Language Model Guided System to Implement Automatic Cyber-attacks. *arXiv preprint arXiv:2403.01038* (2024).

[48] Tian-yang Zhou, Yi-chao Zang, Jun-hu Zhu, and Qing-xian Wang. 2019. NIG-AP: A new method for automated penetration testing. *Frontiers of Information*

Technology & Electronic Engineering 20, 9 (2019), 1277–1288.

A Appendix

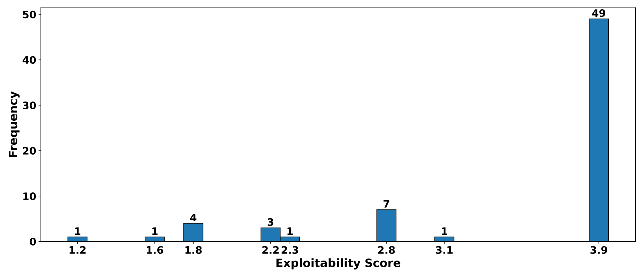


Figure 10: Distribution of exploitability scores

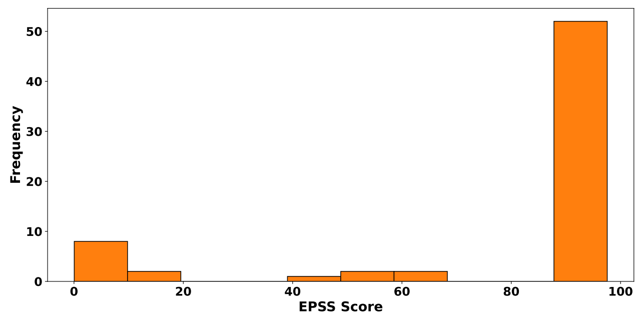


Figure 11: Coverage of EPSS scores

A.1 Benchmark Construction

We use CVSS and EPSS scores to determine the difficulty of exploiting vulnerabilities. CVSS provides a numerical score reflecting the properties of vulnerabilities. Since most of the CVEs on VulHub adopt CVSS version 3.x metrics, we use this as our reference to assign difficulty levels. The numerical score is made of two parts: exploitability and impact. For our penetration testing purpose, we use the exploitability metric as the reference to assign difficulty levels. The exploitability score reflects the ease and technical means by which the vulnerability can be exploited [30]. A higher exploitability score indicates that the vulnerability is easier to exploit. We studied the distribution of exploitability scores, as shown in Fig.10. We found that most exploitability scores are above 3.0, and exploitability scores of 2.0 and 3.0 make natural cutoffs for easy, medium, and hard difficulties. Some vulnerable applications or services have more than one CVE number. We select the CVE to use based on the EPSS score. The EPSS scores measure how likely a vulnerability will be exploited in the wild. A higher EPSS score indicates the vulnerability is more likely to be exploited, making it more realistic for penetration tasks. Fig. 11 shows the distribution of the EPSS scores of the CVEs in our benchmark dataset.

In addition, we remove the Docker images that are not associated with a CVE number and do not have CVSS 3.x scores. Additionally, some vulnerable applications are removed from the dataset due to complicated setup processes, such as requiring a license key from a service provider.

To maintain integrity and fairness in our evaluations, we strictly prohibit PentestAgent from directly accessing any content from VulHub repository, thereby preventing any advantage or bias in our testing methodology.

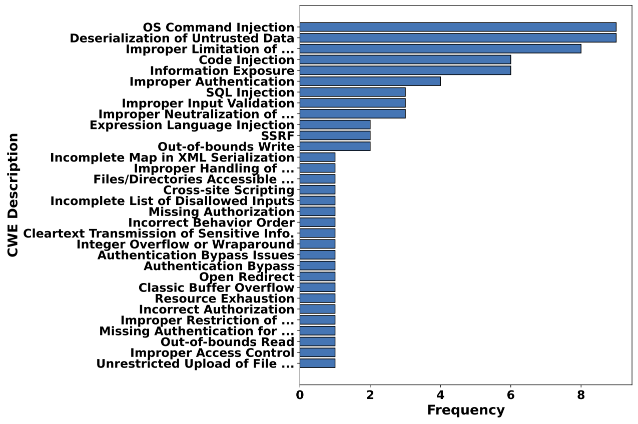


Figure 12: Coverage of CWE

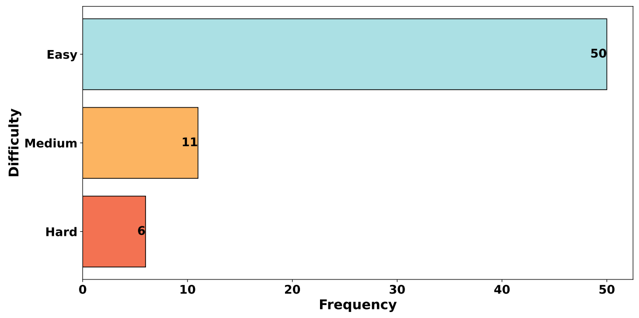


Figure 13: Distribution of exploitation difficulty ratings

Fig. 13 shows the difficulty rating distribution of our benchmark dataset.