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

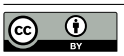
Wei Ruan
Zhejiang University
Hangzhou, Zhejiang, China
ruanwei@zju.edu.cn

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 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 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.

*Both authors contributed equally to this work.



ASIA CCS '25, Hanoi, Vietnam

ACM ISBN 979-8-4007-1410-8/2025/08
https://doi.org/10.1145/3708821.3733882

CCS Concepts

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

Keywords

Penetration Testing, Large Language Model, Agent

ACM Reference Format:
Xiangmin Shen, Lingzhi Wang, Zhenyuan Li, Yan Chen, Wencheng Zhao, Dawei Sun, Jiashui Wang, and Wei Ruan. 2025. PentestAgent: Incorporating LLM Agents to Automated Penetration Testing. In *ACM Asia Conference on Computer and Communications Security (ASIA CCS '25), August 25–29, 2025, Hanoi, Vietnam.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3708821.3733882

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. [14] 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. [43] According to Rapid7's Under the Hoodie report, penetration testing takes an average of 80 hours, with significant outliers taking several hundred hours. [7] 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 [5, 30, 36] primarily modeled attack planning as an attack graph problem [3] 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 [6, 15, 19, 20, 37–39, 49] 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) [37, 38] and reinforcement learning algorithms [6, 19] 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 [12, 18, 48] 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, including validating and debugging the suggested procedures and dynamically acquiring and applying new pentesting techniques.

Table 1: Comparison of LLM-based pentesting systems

System	State&Action Space	Online Search Augmentation	Validation& Debugging Capability
PentestAgent	Large	Auto	Auto
AutoAttacker [48]	Unknown[1]	Manual	Manual
PentestGPT [12]	Unknown[1]	Manual	Manual
Happe et al. [18]	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 employs a multi-agent design where each agent is responsible for a specific task in the penetration testing process. This flexible architecture enables the customization of toolsets across different tasks, enhancing the system's adaptability. In addition to LLM agents, PentestAgent integrates Retrieval Augmented Generation (RAG) [21] to leverage supplementary data during response synthesis and manage the context efficiently. This combination enhances the quality of the generated outputs and further reduces the need for manual intervention.

Table 1 provides a comparative overview of existing automated pentesting systems, illustrating that while frameworks like AutoAttacker and PentestGPT rely on manual or limited approaches, PentestAgent offers a fully automated solution with a larger state and action space and advanced online search augmentation and debugging capabilities.

Moreover, our contributions extend beyond the framework itself. We introduce a comprehensive penetration testing benchmark based on VulHub's collection of vulnerable Docker environments and Capture The Flag (CTF) challenges on HackTheBox. This benchmark spans various difficulty levels and encompasses a wide range of vulnerabilities, addressing a critical gap in current research by providing a practical and accessible evaluation framework.

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 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 make our benchmark datasets and framework publicly available to facilitate further research in automated penetration testing.[1]

2 Background and Related Work

2.1 Penetration Testing

Penetration testing (pentesting) is a structured, multi-stage process designed to identify security vulnerabilities in systems. According to the Penetration Testing Execution Standard (PTES) [42], pentesting consists of three main stages: intelligence gathering, vulnerability analysis, and exploitation. Penetration testing is broadly divided into external and internal assessments [7]. External assessments focus on assets exposed on the Internet, such as web applications, online services, and external networks, using techniques like social engineering, red teaming, and, in particular, web penetration testing.

[1] https://github.com/nbshenxm/pentest-agent

In contrast, internal assessments target the organization's internal network, source code, or physical devices, typically involving code reviews and internal network compromises.

While several recent works have enhanced internal assessments using LLM-based frameworks (e.g., ChatAFL [26], FuzzGPT [13], LLift [23], and LATTE [25]), external assessments, especially web penetration testing, remain underexplored. According to Rapid7's latest report [8], external network compromise constitutes over 80% of penetration testing tasks. This paper addresses this gap by demonstrating how PentestAgent automates web penetration testing, thereby enhancing the overall applicability and efficiency of automated pentesting in real-world scenarios.

Existing Tools. While automated tools exist for individual tasks within these stages, integrating them into a seamless and effective workflow remains a significant challenge. Existing tools specialize in specific penetration testing tasks. For instance, Nmap[29] is widely used for intelligence gathering, allowing testers to analyze network configurations through direct interaction with targets. Nessus[44] and OpenVAS[16] focus on vulnerability analysis, scanning systems for known weaknesses using extensive vulnerability databases. Metasploit[35] is commonly used for exploitation, providing a range of exploits and payloads to execute attacks on identified vulnerabilities. Although these tools are effective alone, their effective use requires expert knowledge, manual decision making, and significant effort to coordinate workflow.

AI-Driven Penetration Testing. 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 [6, 19, 49]. For example, Chen et al. [6] 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.

LLM-Based Penetration Testing. 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 [48] focuses on constructing post-breach attacks, neglecting the pre-compromise stages. PentestGPT [12], 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.

2.2 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.

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 [27].

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 [47]. By guiding the LLM to follow a logical sequence of steps, this method enhances the model's problem-solving capabilities.

Role-playing [22] asks 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 [41].

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,

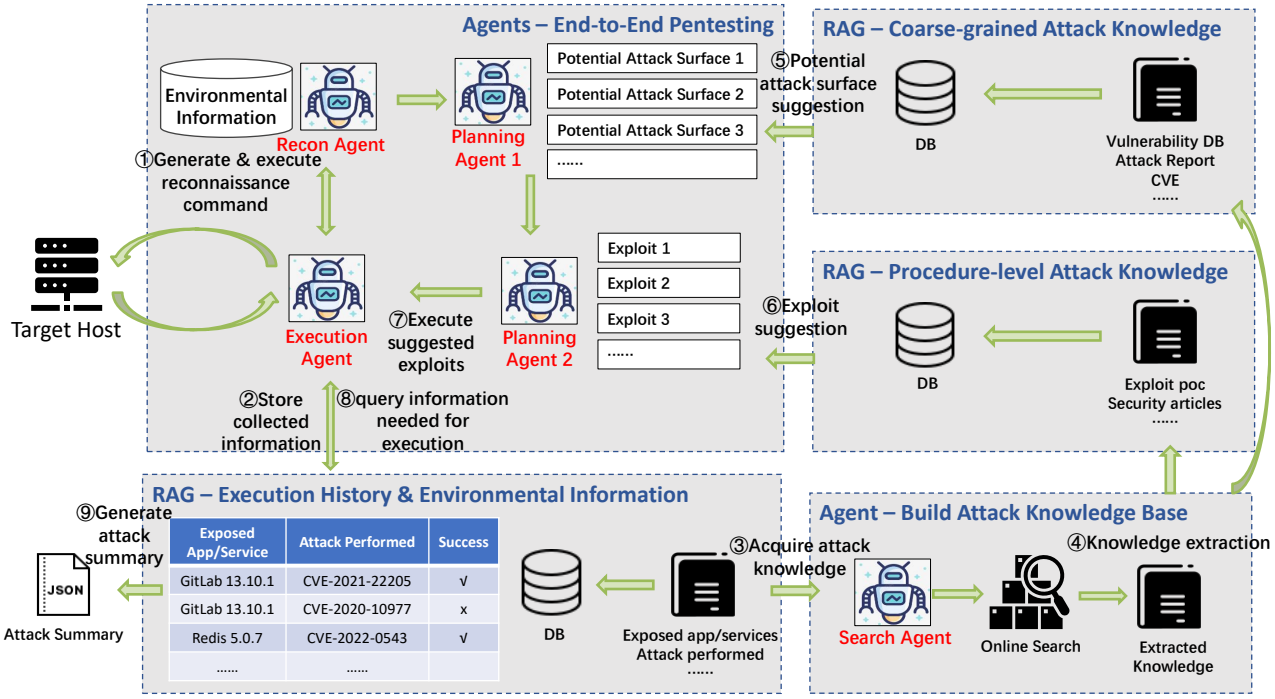


Figure 1: An overview of the components in PentestAgent

structured outputs ensure smooth integration with downstream processes and workflows [24].

Together, these 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

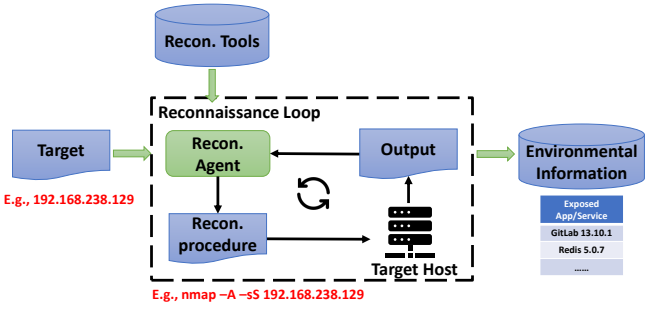


Figure 2: Reconnaissance agent workflow

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

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.} …

Role-playing has proven effective in bypassing the safety policies enforced by the LLM [11]. 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.

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.

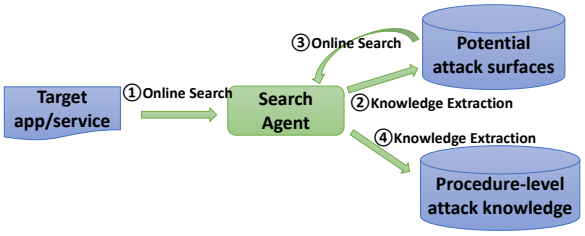


Figure 3: Search agent workflow

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 [40] and AVD [9], 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.

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 the 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

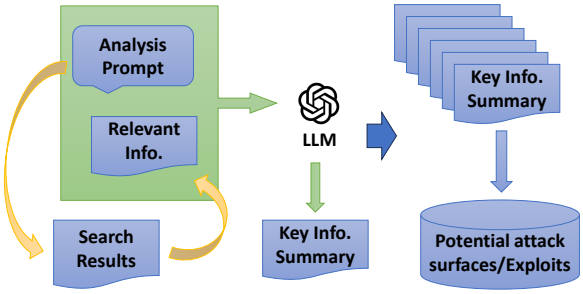


Figure 4: RAG workflow for search result summarization. The yellow arrows denote the retrieval process, and the green arrows denote the generation process.

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}

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.

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.

3.4 Planning Agent

The planning agent takes the 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 services and applications. Then, the planning agent follows a similar process to generate a list of exploits.

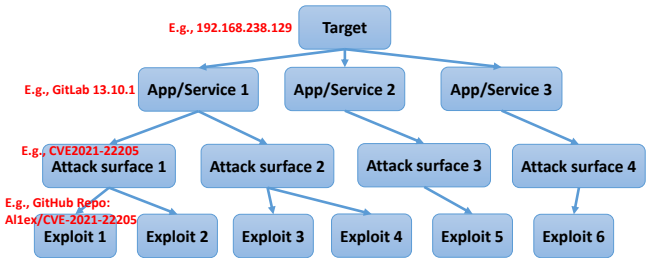


Figure 5: Hierarchical pentesting knowledge database

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 version of the service or application and the types of vulnerabilities. The planning agent makes suggestions for attack surfaces based on the application version and categorizes attack surfaces by vulnerability types.

The planning agent then 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.

3.5 Execution Agent

The execution agent takes the details of the exploit as input and attempts to execute the exploit 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.

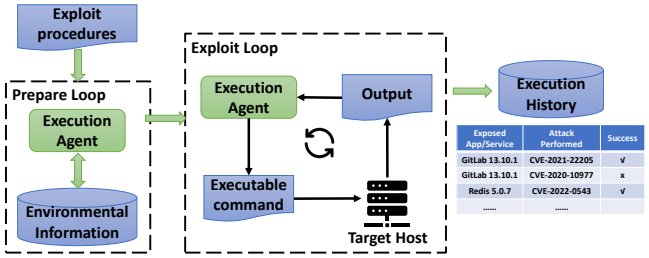


Figure 6: Execution agent workflow

During the preparation stage, the execution agent analyzes the details of the exploit 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 are required for executing this exploit?
2) What information do you need to execute the exploit?

RAG

Use your query tool to learn the details about …

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 queries the environmental information database to fill in the information. After receiving the response containing the requested information, the execution agent's system message is updated as follows to transition to the exploitation stage.

Execution Exploit System Message (Simplified)

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

RAG

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

Self-reflection

When the results indicate an error, you should analyze the error and try to fix it …

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 [17], OWASP Benchmark [33], VulnHub [46], and VulHub [45]. 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 and HackTheBox 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) [28]. 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)[10] and learn about how realistic the vulnerable environment is via the Exploit Prediction Scoring System (EPSS)[32]. We elaborate on how we construct the benchmark dataset in §B.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.13 in the appendix. These vulnerabilities cover eight security risks in OWASP Top 10 vulnerability [34]. Within our benchmark, there are 50 targets with easy exploitability difficulty, 11 with medium exploitability difficulty, and

6 with hard exploitability difficulty. In addition, we incorporated 11 Capture The Flag (CTF) challenges from HackTheBox to simulate more challenging and realistic scenarios. These challenges are used in Section 4.5 for a practicality study and in Section 4.6 for the comparative evaluation with PentestGPT. 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. For the HackTheBox targets, our focus is on obtaining initial access to the target host. In this context, a successful exploit is one that grants access to the target system.

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 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.

We evaluate our framework using a mix of commercial and open-source LLMs. Table 2 summarizes their key properties.

4.2 Effectiveness of the Entire Framework

We investigate the effectiveness of PentestAgent by its success rates in 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.

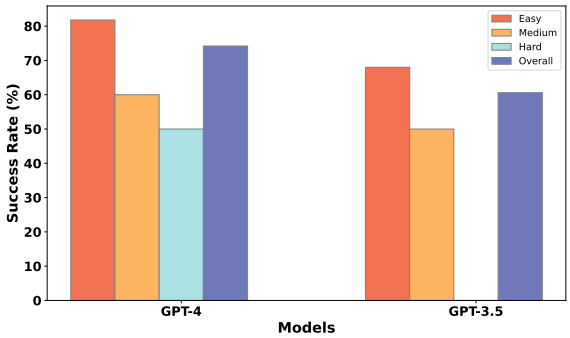


Figure 7: Success rate on penetration testing tasks

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 the complex reasoning required for challenging tasks.

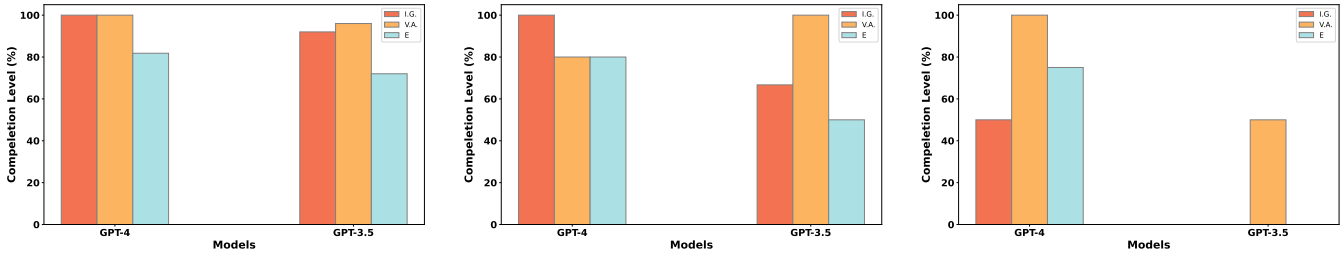
4.3 Completion level of Penetration Testing Stages

To further evaluate PentestAgent, we analyze its performance across individual penetration testing stages. Fig. 8 presents the completion rates for intelligence gathering, vulnerability analysis, and exploitation across different difficulty levels and LLM backbones.

GPT-4 demonstrated robust performance across all stages and difficulty levels, effectively handling a range of penetration testing tasks. In easy tasks, it achieved full completion in both intelligence gathering and vulnerability analysis, with an 81.8% completion rate in exploitation. For medium-difficulty tasks, GPT-4 maintained high completion rates across all stages, with a minor drop in vulnerability

Table 2: Summary of LLM models used in our evaluation. Input/output costs are based on pricing at the time of testing.

Model	Context Window	Knowledge Cutoff	Input Cost	Output Cost
GPT-3.5-turbo-0125	16,385 tokens	Sep 2021	$0.50/1M tokens	$1.50/1M tokens
GPT-4o	128,000 tokens	Oct 2023	$5.00/1M tokens	$15.00/1M tokens
o1-mini	128,000 tokens	Oct 2023	$1.10/1M tokens	$4.40/1M tokens
Llama 3.1-8B-Instruct	128,000 tokens	Dec 2023	Open-source	Open-source



(a) Easy tasks **(b) Medium tasks** **(c) Hard tasks**

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

analysis. However, in hard tasks, performance declined, particularly in intelligence gathering (50%), suggesting limitations in handling complex reconnaissance tasks that require advanced reasoning and adaptive strategies.

In contrast, GPT-3.5 exhibited more variability across difficulty levels. It performed well in easy tasks, with 92% completion in intelligence gathering and 96% in vulnerability analysis, though its exploitation stage completion rate (72%) was slightly lower. For medium tasks, while maintaining a 100% completion rate in vulnerability analysis, its performance declined in intelligence gathering (66.7%) and exploitation (50%), indicating challenges in navigating complex reconnaissance and execution scenarios. Notably, in hard tasks, GPT-3.5 struggled significantly, failing to complete both intelligence gathering and exploitation, highlighting its limitations in reasoning and contextual understanding required for advanced penetration testing.

Overall, both models effectively automate significant portions of penetration testing, but GPT-4 consistently outperforms GPT-3.5, particularly in more complex scenarios. These results emphasize the importance of advanced reasoning capabilities and enhanced reconnaissance strategies in achieving higher success rates in automated penetration testing.

4.4 Ablation Study

To assess how different LLM backbones influence PentestAgent's performance, we performed an ablation study on a subset of VulHub targets. Due to hardware limitations, specifically insufficient GPU resources to efficiently run the Llama 3.1 model on the full dataset, we selected a smaller subset comprising six easy, five medium, and two hard targets. Fig. 9a and Fig. 9b summarize the completion levels and overhead results, respectively. The results show that while all models perform consistently in vulnerability analysis (with 100% completion), differences emerge in the other stages. For the intelligence gathering stage, GPT-4 and o1-mini achieve higher completion levels compared to GPT-3.5 and Llama 3.1-8B-Instruct, suggesting that certain models are more effective in integrating context and managing complex reasoning. In the exploitation stage, o1-mini outperforms the others, indicating its strength in executing detailed attack procedures, whereas GPT-3.5 and Llama 3.1-8B-Instruct fall behind.

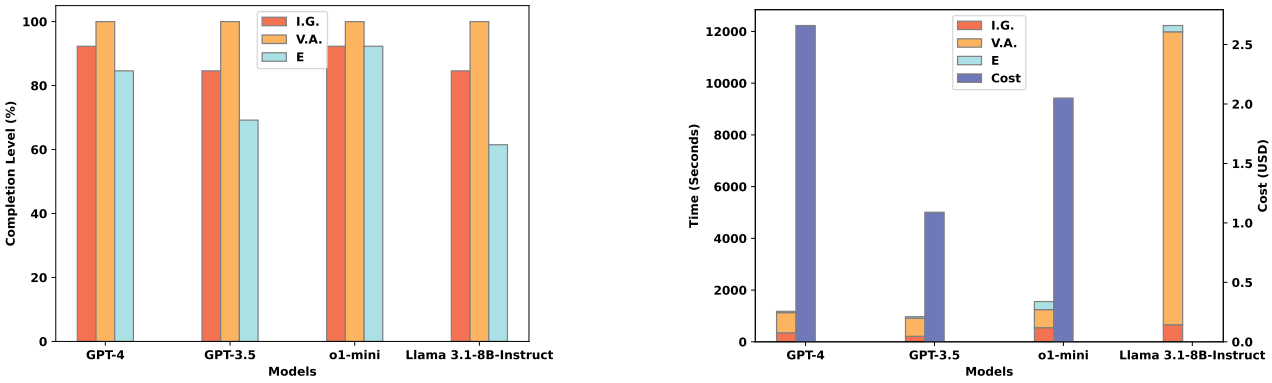
Overhead measurements further highlight trade-offs between processing time and cost. Although GPT-3.5 is faster in intelligence gathering and is the most cost-effective, its lower exploitation performance suggests a potential compromise in handling complex tasks. In contrast, while o1-mini delivers the best exploitation completion, it incurs longer processing times during vulnerability analysis. The Llama 3.1-8B-Instruct model, despite having no cost, suffers from significant time overhead and lower exploitation performance, which may limit its practical use.

4.5 Practicality Study

To evaluate PentestAgent's effectiveness in realistic settings, we deployed it to solve HackTheBox challenges. Unlike standardized benchmarks, these challenges simulate dynamic, real-world penetration testing tasks by presenting diverse vulnerabilities across various CWEs. In this study, we selected 11 HackTheBox machines, nine labeled as easy, one as medium, and one as hard, to provide a broad assessment of the framework's practical utility.

Fig. 10 shows the completion level and overhead of PentestAgent on HackTheBox challenges. Table 3 in Appendix C further details PentestAgent's performance on these challenges by reporting the number of completed testing stages. While PentestAgent successfully exploited six machines, others like *Pilgrimage* achieved only partial completion, with only the vulnerability analysis stage being successful.

Overall, these results indicate that PentestAgent can address a range of real-world scenarios. However, the variability in stage completion underscores the need for further improvements to achieve more consistent, full-stage automation. We discuss the failed cases in detail in Section 4.7.



(a) Completion level on targets

(b) Average time spent and cost on targets

Figure 9: Completion level and overhead of different LLM Backbones.

4.6 Comparison with PentestGPT

We conducted a comparison of the effectiveness and efficiency of PentestAgent against PentestGPT. Unlike PentestAgent, PentestGPT requires human participation for feedback and decision-making throughout the penetration testing process. Thus, we compare their performance using case studies. We randomly selected ten vulnerabilities from VulHub (five easy, three medium, and two hard) and included 11 HackTheBox challenges (nine easy, one medium, and one hard). Two evaluators with different skill levels conducted the tests: an undergraduate student with limited penetration testing experience evaluated PentestGPT on the VulHub targets, while a PhD student with more experience assessed the HackTheBox challenges. Both systems were configured to use the GPT-3.5 model to ensure a fair comparison.

Fig. 10 shows the completion level and overhead comparison between PentestAgent and PentestGPT on HackTheBox targets. Our results indicate that PentestAgent achieves higher exploitation success and overall efficiency. For instance, PentestAgent completes intelligence gathering in 220 seconds compared to 1199 seconds for PentestGPT, and finishes exploitation in 172 seconds versus 364 seconds. Although PentestGPT is slightly faster in vulnerability analysis, its slower performance in other stages due to human involvement reduces its overall efficiency. Additional results on VulHub targets and evaluation details can be found in the Appendix C. These findings imply that PentestAgent can deliver more consistent and timely penetration testing, highlighting the benefits of minimizing human intervention in real-world security assessments.

4.7 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.

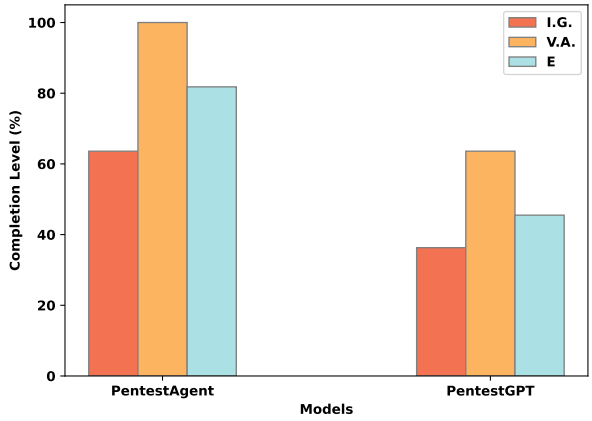
In the intelligence gathering stage, PentestAgent occasionally fails to recognize services or applications with the appropriate level of granularity. For instance, our evaluation revealed that PentestAgent struggled to detect components like PHPMailer, PHPUnit, and Ghostscript. These are not standalone 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.

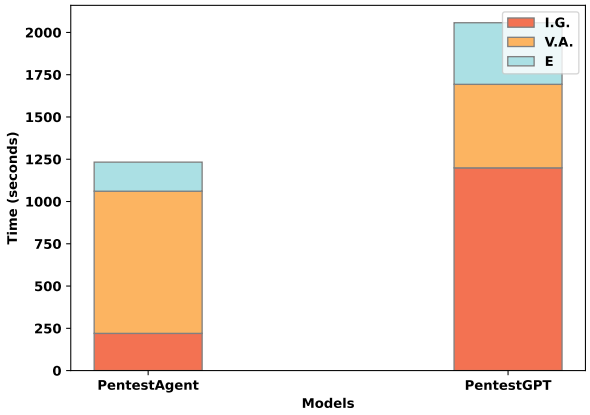
Requiring Additional Knowledge: Certain exploits demand a level of domain-specific knowledge that may exceed the capabilities of an LLM agent. For example, exploiting Samba server 4.6.3 (CVE-2017-7494) assumes the attacker has prior knowledge of credentials (username and password) to establish an SMB 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 [4] 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



(a) Comparison of completion levels on HackTheBox targets



(b) Comparison of average time spent on HackTheBox targets

Figure 10: Completion level and overhead comparison on HackTheBox targets.

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.

5 Discussion

5.1 Comparison with Existing Frameworks

While our primary comparison in this work is with PentestGPT, several emerging frameworks address related challenges in automated penetration testing. Notably, AutoAttacker and Enigma offer complementary perspectives that highlight different strengths and limitations relative to PentestAgent.

AutoAttacker [48] focuses exclusively on the post-breach stage of an attack. It is designed to automate the "hands-on-keyboard" exploitation phase once a system has been compromised. In contrast, PentestAgent addresses the entire penetration testing pipeline—from reconnaissance and vulnerability analysis to exploitation—providing a more comprehensive solution. This broader scope is critical for real-world scenarios, where early-stage tasks are just as vital as post-breach actions for assessing and improving security.

Enigma [2], on the other hand, extends the SWE-agent framework by integrating interactive tools that support solving Capture The Flag challenges. Its design primarily targets challenges in the crypto and reverse engineering domains and still relies on human-in-the-loop interactions. Although Enigma's interactive interfaces are effective for guiding the agent through specific problem domains, its reliance on manual intervention limits full automation. In contrast, PentestAgent is engineered to operate autonomously across diverse attack stages, reducing the need for human feedback and thus enabling more consistent performance in automated penetration testing.

Overall, these comparisons illustrate that while AutoAttacker and Enigma contribute valuable insights and capabilities, PentestAgent distinguishes itself by offering an end-to-end automated solution.

5.2 Limitations on Performing Sophisticated Pentesting

Our system, PentestAgent, focuses on exploiting individual vulnerable applications and services to help identify and mitigate these vulnerabilities. However, more sophisticated attack planning may be required in more complex penetration testing scenarios, such as red team simulations. These scenarios often involve combining several vulnerabilities to achieve a more challenging yet impactful exploit. For example, an SSRF vulnerability could be used as an intermediary step to exploit an internal application, eventually leading to obtaining root privileges.

While addressing such sophisticated attack strategies is beyond the scope of this paper, our framework, PentestAgent, can still be valuable in these complex scenarios. Our system can identify and validate exposed vulnerabilities, such as SSRF, which can serve as starting points for further exploitation. This initial identification and validation process can significantly contribute to the overall penetration testing workflow, providing a foundation upon which more advanced exploitation techniques can be built.

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 and chain-of-thought, PentestAgent enhances the penetration testing process through improved knowledge integration and automation.

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

References

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

[2] Talor Abramovich, Meet Udeshi, Minghao Shao, Kilian Lieret, Haoran Xi, Kimberly Milner, Sofija Jancheska, John Yang, Carlos E Jimenez, Farshad Khorrami, et al. 2024. EnIGMA: Enhanced Interactive Generative Model Agent for CTF Challenges. *arXiv preprint arXiv:2409.16165* (2024).

[3] 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.

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

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

[11] 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).

[12] 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.

[13] 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.

[14] 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.

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

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

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

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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).

[24] 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.

[25] 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).

[26] 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)*.

[27] 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

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

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

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

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

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

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

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

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

[36] 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.

[37] 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.

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

[39] 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.

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

[41] 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).

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

[43] 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.

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

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

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

[47] 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.

[48] 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).

[49] Tian-yang Zhou, Yi-chao Zang, Jun-hu Zhu, and Qing-xian Wang. 2019. NIGAP: A new method for automated penetration testing. *Frontiers of Information Technology & Electronic Engineering* 20, 9 (2019), 1277–1288.

A Prompts

This section specifies more prompts used in the PentestAgent pipeline.

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.

Reconnaissance Summary Prompt (Simplified)

Provide a summary of all reconnaissance findings …
The summary of findings should be presented in valid JSON format with the following fields: {FORMAT SPEC.}
For example, {OUTPUT FORMAT EXAMPLE}

The following prompt summarizes the search results 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.} …
For example, {OUTPUT FORMAT EXAMPLE}

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}

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 exploit {app} {version} and provide the details about the vulnerabilities and the reasons to support each selection …
The details should include …
Make the selections by checking whether {version} is within the applicable version of the exploit and the vulnerability types …
The results should be presented in valid JSON format with the following fields: {FORMAT SPEC.} …
For example, {OUTPUT FORMAT EXAMPLE}

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 …applicable to {app} {version} and provide the details about the exploit and reasons to support each selection …
The details should include …
Make the selections by checking whether {version} is within the applicable version of the exploit and the execution effects …
The results should be presented in valid JSON format with the following fields: {FORMAT SPEC.} …
For example, {OUTPUT FORMAT EXAMPLE}

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.} …
For example, the response looks like this: {OUTPUT FORMAT EXAMPLE}

B Benchmark Construction

B.1 VulHub 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 [31]. A higher exploitability score indicates that the vulnerability is easier to exploit. We studied the distribution of exploitability scores, as shown in Fig.11. 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. 12 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.

Fig. 14 shows the difficulty rating distribution of our VulHub benchmark dataset.

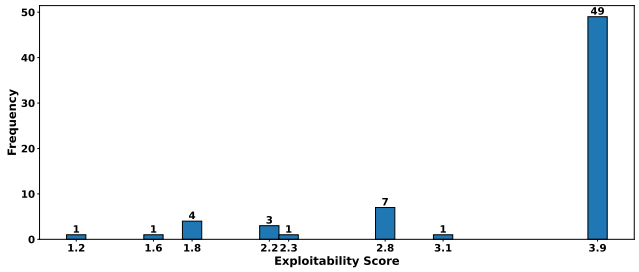


Figure 11: Distribution of exploitability scores

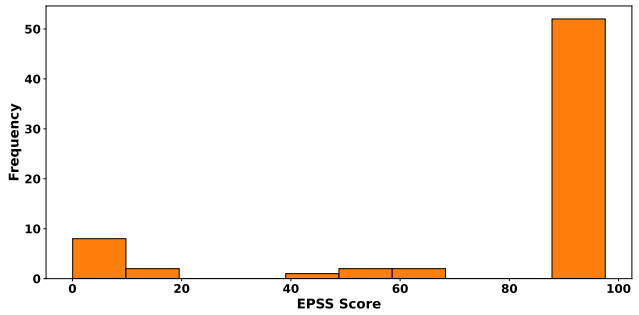


Figure 12: Coverage of EPSS scores

B.2 HackTheBox Benchmark Construction

In selecting HackTheBox (HTB) challenges for evaluating PentestAgent, we aimed to create a diverse and representative set of targets that reflect real-world penetration testing scenarios. Our selection focused on key aspects such as operating system diversity, vulnerability relevance, and difficulty levels to ensure a comprehensive assessment.

To evaluate PentestAgent 's adaptability across different environments, we included both Linux and Windows machines. This diversity reflects the variety of systems encountered in real-world engagements and ensures that PentestAgent is tested across different exploitation techniques. The chosen machines also incorporate well-known vulnerabilities spanning a decade, allowing us to assess PentestAgent 's ability to handle both historical and contemporary exploits. For instance, Blue features the EternalBlue

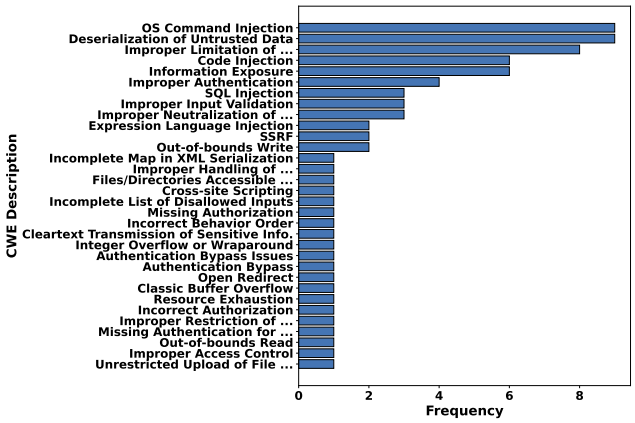


Figure 13: Coverage of CWE

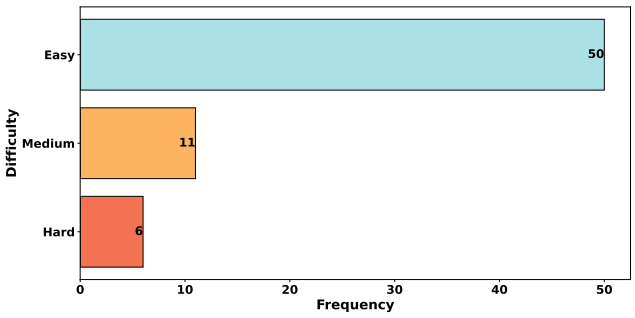


Figure 14: Distribution of exploitation difficulty ratings

vulnerability (CVE-2017-0144), a widely known Windows SMB exploit, while Legacy includes MS08-067 (CVE-2008-4250), another critical SMB-based attack.

We selected machines across easy, medium, and hard difficulty levels to analyze PentestAgent 's performance in different attack scenarios. Easier challenges, such as Lame and Optimum, test fundamental exploitation techniques, while medium and hard machines, such as Stratosphere (CVE-2017-5638) and Reel (CVE-2017-0199), require deeper reconnaissance, multi-step attacks, and more advanced reasoning. This progression ensures that PentestAgent is evaluated not only on basic automation tasks but also on its effectiveness in handling complex, real-world pentesting challenges.

This carefully selected set of HTB challenges, as shown in Table 3, allows for a thorough assessment of PentestAgent 's automation capabilities, performance across different vulnerability types, and effectiveness in progressively complex penetration testing scenarios.

B.3 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,

Table 3: PentestAgent's performance among HackTheBox CTF challenges

Machine	Difficulty	Completed Stage
Sau	Easy	2/3 (I.G, V.A)
Pilgrimage	Easy	1/3 (V.A)
Lame	Easy	3/3
Topology	Easy	3/3
PC	Easy	3/3
Blue	Easy	3/3
Shocker	Easy	2/3 (V.A., E)
Optimum	Easy	3/3
Legacy	Easy	3/3
Stratosphere	Medium	2/3 (V.A., E)
Reel	Hard	2/3 (V.A, E.)

Table 4: PentestGPT's performance among HackTheBox CTF challenges

Machine	Difficulty	Completed Stage
Sau	Easy	2/3 (I.G, V.A)
Pilgrimage	Easy	1/3 (V.A)
Lame	Easy	2/3 (V.A., E)
Topology	Easy	2/3 (V.A., E)
PC	Easy	0/3
Blue	Easy	2/3 (V.A., E)
Shocker	Easy	0/3
Optimum	Easy	3/3
Legacy	Easy	3/3
Stratosphere	Medium	0/3
Reel	Hard	1/3 (I.G.)

contributing to vulnerable and outdated components ranking 6th on the OWASP Top 10 Web Application Security Risks. [34] 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.

C Additional Evaluation Results

The detailed pentesting performance of both systems on HackTheBox challenges are available in Table 3 and Table. 4.

In addition to the HackTheBox evaluation, we compare PentestAgent and PentestGPT on VulHub targets, analyzing both completion levels and overhead, as shown in Fig. 15 and Fig. 16.

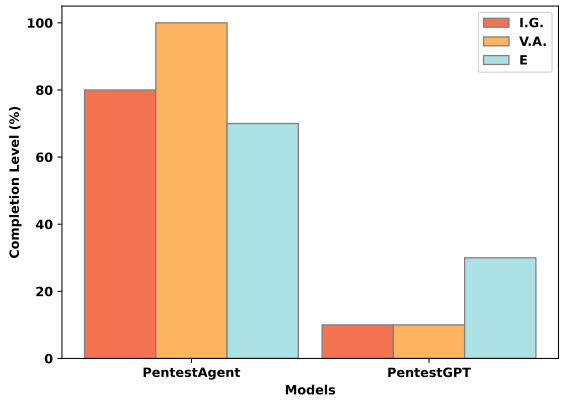


Figure 15: Comparison of completion levels on VulHub targets

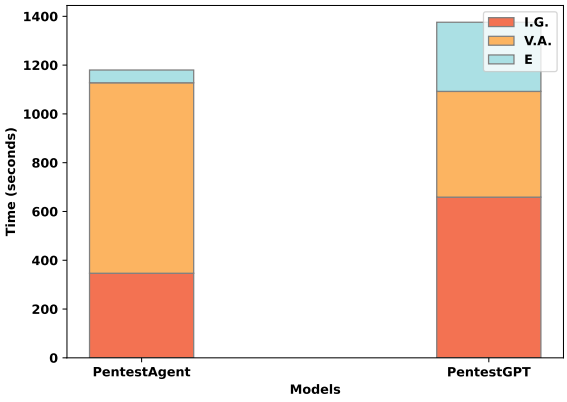


Figure 16: Comparison of average time spent on VulHub targets

PentestAgent significantly outperformed PentestGPT across all penetration testing stages. In intelligence gathering, PentestAgent achieved an 80% completion rate, compared to only 10% for PentestGPT, demonstrating its superior ability to extract relevant target information. In vulnerability analysis, PentestAgent completed 100% of the tasks, whereas PentestGPT again achieved just 10%, highlighting its limited capability in identifying and assessing vulnerabilities. During exploitation, PentestAgent successfully completed 70% of tasks, more than double PentestGPT 's 30%, confirming its stronger execution capabilities.

Efficiency is crucial in penetration testing automation, and PentestAgent consistently required less time than PentestGPT in key stages. It completed intelligence gathering in 212.9 seconds, whereas PentestGPT took 658.7 seconds, over three times longer. Similarly, in exploitation, PentestAgent finished in 58.6 seconds, compared to 283.5 seconds for PentestGPT, demonstrating its more streamlined attack execution. While PentestAgent took

longer in vulnerability analysis (698.8 seconds vs. 433.5 seconds), this additional time contributed to its higher success rate, ensuring a more accurate and actionable assessment.

The results confirm that PentestAgent is both more effective and more efficient than PentestGPT. It achieves higher completion rates across all stages, particularly in intelligence gathering and vulnerability analysis, which are critical for successful exploitation. Moreover, its lower overhead in intelligence gathering and exploitation makes it a more scalable and practical solution for real-world penetration testing. While its vulnerability analysis takes slightly longer, this trade-off results in more reliable and successful attack execution, solidifying PentestAgent as a robust and efficient automated pentesting framework.