PENTESTGPT: An LLM-empowered Automatic Penetration Testing Tool

Gelei Deng[1], Yi Liu[1], Víctor Mayoral-Vilches[2,3] , Peng Liu[4], Yuekang Li[5], Yuan Xu [1], Tianwei Zhang[1], Yang Liu[1], Martin Pinzger[2], and Stefan Rass[6]

[1]Nanyang Technological University, [2]Alpen-Adria-Universität Klagenfurt, [3]Alias Robotics, [4]Instituite for Infocomm Research, A*STAR, [5]University of New South Wales, [6]Johannes Kepler University Linz

{*gelei.deng, yi009, xu.yuan, tianwei.zhang, yangliu*}*@ntu.edu.sg*, *victor@aliasrobotics.com*

liu_peng@i2r.a-star.edu.sg, Martin.Pinzger@aau.at, stefan.rass@jku.at

***Abstract*—Penetration testing, a crucial industrial practice for ensuring system security, has traditionally resisted automation due to the extensive expertise required by human professionals. Large Language Models (LLMs) have shown significant advancements in various domains, and their emergent abilities suggest their potential to revolutionize industries. In this research, we evaluate the performance of LLMs on real-world penetration testing tasks using a robust benchmark created from test machines with platforms. Our findings reveal that while LLMs demonstrate proficiency in specific sub-tasks within the penetration testing process, such as using testing tools, interpreting outputs, and proposing subsequent actions, they also encounter difficulties maintaining an integrated understanding of the overall testing scenario.**

In response to these insights, we introduce PENTESTGPT, an LLM-empowered automatic penetration testing tool that leverages the abundant domain knowledge inherent in LLMs. PENTESTGPT is meticulously designed with three self-interacting modules, each addressing individual sub-tasks of penetration testing, to mitigate the challenges related to context loss. Our evaluation shows that PENTESTGPT not only outperforms LLMs with a task-completion increase of 228.6% compared to the GPT-3.5 model among the benchmark targets but also proves effective in tackling real-world penetration testing challenges. Having been open-sourced on GitHub, PENTESTGPT has garnered over 4,700 stars and fostered active community engagement, attesting to its value and impact in both the academic and industrial spheres.

***Index Terms*—security, offensive, cybersecurity, pentesting**

1. Introduction

Guaranteeing a system's immunity to potential attacks is a formidable challenge. Offensive security methods, such as penetration testing (pen-testing) or red teaming, have become essential in the security lifecycle. As detailed by Applebaum [1], these methods require security teams to attempt breaches of an organization's defenses to uncover vulnerabilities. They offer marked advantages over traditional defensive mechanisms, reliant on incomplete system knowledge and modeling. Guided by the principle *"the best defense is a good offense"*, this study focuses on offensive strategies, particularly *penetration testing*.

Penetration testing [2] is a proactive offensive technique aiming at identifying, assessing, and mitigating as many security vulnerabilities as possible. This involves executing targeted attacks to confirm diverse flaws (e.g., erratic behaviors) and is efficacious in creating a comprehensive inventory of vulnerabilities complemented by actionable enhancement recommendations. As a widely-employed practice for security appraisal, penetration testing empowers organizations to discern and neutralize potential vulnerabilities in their networks and systems before exploitation by malicious entities. Despite its significance, the industry often leans on manual techniques and specialized knowledge [3], making it labor-intensive. This has generated a gap in responding to the escalating demand for adept and efficient security evaluations.

Recently Large Language Models (LLMs) [4], [5] are making striking progress, exhibiting an increasingly nuanced understanding of human-like text and effectively executing various tasks across diverse domains. One intriguing aspect of LLMs is their emergent abilities [6], which are not explicitly programmed but arise during the training process. These abilities enable LLMs to perform complex tasks such as reasoning, summarization, question-answering, and domain-specific problem-solving without requiring specialized training. Such capabilities indicate the transformative potential of LLMs across various sectors, including cybersecurity. A critical question thus emerges: can LLMs be leveraged in cybersecurity, particularly for performing automated penetration testing?

Motivated by this question, we set out to evaluate the capabilities of LLMs on real-world penetration testing tasks. Unfortunately, the current benchmarks for penetration testing [7], [8] are not comprehensive and fail to assess

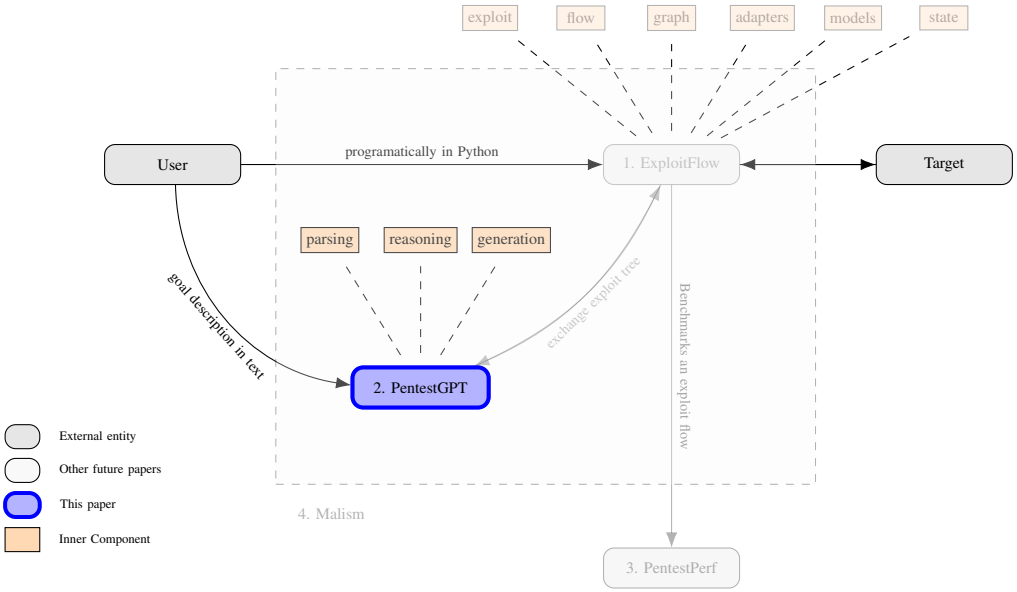


Figure 1: Architecture of our framework to develop a fully automated penetration testing tools, MALISM. Figure depicts the various interaction flows that an arbitrary `User` could follow using MALISM to pentest a given `Target`. **1.** Corresponds with EXPLOITFLOW, a modular library to produce security exploitation routes (*exploit flows*) that caputures the state of the system being tested in a flow after every discrete action. **2. (this paper)** Corresponds with PENTESTGPT, a testing tool that leverages the power of LLMs to produce testing guidance (heuristics) for every given discrete state. **3.** PENTESTPERFis a comprehensive penetration testing benchmark to evaluate the performances of penetration testers and automated tools across a wide array of testing targets. **4.** captures MALISM, our framework to develop fully automated penetration testing tools which we name *cybersecurity cognitive engines*.

progressive accomplishments fairly during the process. To address this limitation, we construct a robust benchmark that includes test machines from HackTheBox [9] and VulnHub [10]—two leading platforms for penetration testing challenges. Comprising 13 targets with 182 sub-tasks, our benchmark encompasses all vulnerabilities appearing in OWASP's top 10 vulnerability list [11]. Also, it offers a more detailed evaluation of the tester's performance by monitoring the completion status for each sub-task.

Armed with this benchmark, we conduct an exploratory study using GPT-3.5 [12], GPT-4 [13], and Bard [14] as representative LLMs. We interactively test these models by guiding them to complete the penetration tasks against our benchmark targets. This interaction involves setting a penetration testing goal for the LLM, soliciting it for the appropriate operation to execute, implementing it in the testing environment, and feeding the test outputs back to the LLM for next-step reasoning (Figure 2). By repeating this cycle, we derive the final penetration testing results. To evaluate the performance of the LLMs, we compare their results against baseline solutions provided by official walkthroughs and solutions from certified penetration testers. By analyzing similarities and differences in their problem-solving approaches, we aim to better understand LLMs' penetration testing capabilities and discern how their problem-solving strategies diverge from those of human experts.

Our investigation yields intriguing insights into the capabilities and limitations of LLMs in penetration testing. We discover that LLMs demonstrate proficiency in managing specific sub-tasks within the testing process, such as utilizing testing tools, interpreting their outputs, and suggesting subsequent actions. Compared to human experts, LLMs are especially adept at executing complex commands and options with testing tools, while models like GPT-4 excel in comprehending source code and pinpointing vulnerabilities. Furthermore, LLMs can craft appropriate test commands and accurately describe graphical user-interface operations needed for specific tasks. Leveraging their vast knowledge base, they can design inventive testing procedures to unveil potential vulnerabilities in real-world systems and CTF challenges. However, we also note that LLMs have difficulty in maintaining a coherent grasp of the overarching testing scenario, a vital aspect for attaining the testing goal. As the dialogue advances, they may lose sight of earlier discoveries and struggle to apply their reasoning consistently toward the final objective. Additionally, LLMs might overemphasize recent tasks in the conversation history, regardless of their vulnerability status. As a result, they tend to neglect other potential attack surfaces exposed in prior tests and fail to complete the penetration testing task.

The outcomes of our empirical study are promising, re-

vealing that LLMs possess the necessary domain knowledge to perform penetration testing tasks. In particular, they are great at providing *an intuition* of what to do in a given networking scenario. However, what they lack is effective guidance to carry out these tasks independently and maintain a cohesive grasp of the testing scenario. On the other hand, as investigated in a prior research publication [] focused on capturing the exploitation route (or flow) for automation. Given the complexity of the (network) state space, the state itself is not enough to reason about what are the best actions to pentest. It rapidly becomes evident that a heuristic is needed to support autonomous pentesting which helps pick actions to achieve given goals. With this understanding, we aim to contribute unlocking the potential of modern machine learning approaches and develop a fully automated penetration testing framework that helps produce cybersecurity cognitive engines. Our overall architecture is depicted in Figure 1, which shows our current work so far and near future planned contributions. Our proposed framework, MALISM, is designed to enable a user without in-depth security domain knowledge to produce its own cybersecurity cognitive engine that helps conduct penetration testing over an extensive range of targets. This framework comprises three primary components:

1) EXPLOITFLOW []: A modular library to produce cyber security exploitation routes (*exploit flows*). EXPLOITFLOW aims to combine and compose exploits from different sources and frameworks, capturing the state of the system being tested in a flow after every discrete action which allows learning attack trees that affect a given system. EXPLOITFLOW's main motivation is to facilitate and empower Game Theory and Artificial Intelligence (AI) research in cyber security. It provides a unique representation of the exploitation process that encodes every facet within it. Its representation can be effectively integrated with various penetration testing tools and scripts, such as Metasploit [15] to perform end-to-end penetration testing. Such representation can be further visualized to guide the human experts for the reproduction of the testing process.
2) PENTESTGPT (**this paper**): An automated penetration testing system that leverages the power of LLMs to produce testing guidance and intuition at every given discrete state. It functions as the core component of the MALISM framework, guiding the LLMs to efficiently utilize their domain knowledge in real-world testing scenarios.
3) PENTESTPERF: A comprehensive penetration testing benchmark developed to evaluate the performances of penetration testers and automated tools across a wide array of testing targets. It offers a fair and robust platform for performance comparison.

The harmonious integration of these three components forms an automated, self-evolving penetration testing framework capable of executing penetration tests over various targets, MALISM. This framework to develop fully automated penetration testing tools, which we name *cybersecurity cognitive engines*, aims to revolutionize the field of penetration testing by significantly reducing the need for domain expertise and enabling more comprehensive and reliable testing.

Building on our insights into LLMs' capabilities in penetration testing, we present PENTESTGPT, an interactive system designed to enhance the application of LLMs in this domain. Drawing inspiration from the collaborative dynamics commonly observed in real-world human penetration testing teams, PENTESTGPT is particularly tailored to manage large and intricate projects. It features a tripartite architecture comprising Reasoning, Generation, and Parsing Modules, each reflecting specific roles within penetration testing teams. The Reasoning Module emulates the function of a lead tester, focusing on maintaining a high-level overview of the penetration testing status. We introduce a novel representation, the Pentesting Task Tree (PTT), based on the cybersecurity attack tree [16]. This structure encodes the testing process's ongoing status and steers subsequent actions. Uniquely, this representation can be translated into natural language and interpreted by the LLM, thereby comprehended by the Generation Module and directing the testing procedure. The Generation Module, mirroring a junior tester's role, is responsible for constructing detailed procedures for specific sub-tasks. Translating these into exact testing operations augments the generation process's accuracy. Meanwhile, the Parsing Module deals with diverse text data encountered during penetration testing, such as tool outputs, source codes, and HTTP web pages. It condenses and emphasizes these texts, extracting essential information. Collectively, these modules function as an integrated system. PENTESTGPT completes a complex penetration testing task by bridging high-level strategies with precise execution and intelligent data interpretation, thereby maintaining a coherent and effective testing process.

We evaluate PENTESTGPT using our benchmark to showcase its efficacy. Specifically, our system achieves remarkable performance gains, with 228.6% and 58.6% increases in sub-task completion compared to the direct usage of GPT-3.5 and GPT-4, respectively. We also apply PENTESTGPT to the HackTheBox active penetration testing machines challenge [17], completing 4 out of the 10 selected targets at a total OpenAI API cost of 131.5 US Dollars, ranking among the top 1% players in a community of over 670,000 members. This evaluation underscores PENTESTGPT's practical value in enhancing penetration testing tasks' efficiency and precision. The solution has been made publicly available on GitHub[1], receiving widespread acclaim with over 4,700 stars to the date of writing, active community engagement, and ongoing collaboration with multiple industrial partners.

In summary, we make the following contributions:

- **Development of a Comprehensive Penetration Testing Benchmark.** We craft a robust and representative penetration testing benchmark, encompassing a multitude of test

1. For anonymity during the review process, we have created an anonymous repository to open-source our solution [18].

machines from leading platforms such as HackTheBox and VulnHub. This benchmark includes 182 sub-tasks covering OWASP's top 10 vulnerabilities, offering fair and comprehensive evaluation of penetration testing.
- **Empirical Evaluation of LLMs for Penetration Testing Tasks.** By employing models like GPT-3.5, GPT-4, and Bard, our exploratory study rigorously investigates the strengths and limitations of LLMs in penetration testing. The insights gleaned from this analysis shed valuable light on the capabilities and challenges faced by LLMs, enriching our understanding of their applicability in this specialized domain.
- **Development of an Innovative LLM-powered Penetration Testing System.** We engineer PENTESTGPT, a novel interactive system that leverages the strengths of LLMs to carry out penetration testing tasks automatically. Drawing inspiration from real-world human penetration testing teams, PENTESTGPT integrates a tripartite design that mirrors the collaborative dynamics between senior and junior testers. This architecture optimizes LLMs' usage, significantly enhancing the efficiency and effectiveness of automated penetration testing.

2. Background & Related Work

2.1. Penetration Testing

Penetration testing, or "pentesting", is a critical practice to enhance organizational systems' security. In a typical penetration test, security professionals, known as penetration testers, analyze the target system, often leveraging automated tools. The standard process is divided into seven phases [19]: Reconnaissance, Scanning, Vulnerability Assessment, Exploitation, and Post Exploitation (including reporting). These phases enable testers to understand the target system, identify vulnerabilities, and exploit them to gain access.

Despite substantial efforts [8], [20], [21] in the field, a fully automated penetration testing pipeline remains elusive. The challenges in automating the process arise from the comprehensive knowledge needed to understand and manipulate various vulnerabilities and the demand for a strategic plan to guide subsequent actions. In practice, penetration testers often use a combined approach integrating depth-first and breadth-first search techniques [19]. They begin by obtaining an overarching understanding of the target environment (utilizing a breadth-first approach) before focusing on specific services and vulnerabilities (employing a depth-first approach). This strategy ensures a thorough system analysis while prioritizing promising attack vectors, relying heavily on individual experience and domain expertise. Additionally, penetration testing requires many specialized tools with unique features and functions. This diversity adds complexity to the automation process. Therefore, even with the support of artificial intelligence, creating a fully unified solution for automated penetration testing remains a formidable challenge.

2.2. Large Language Models

Large Language Models (LLMs), including OpenAI's GPT-3.5 and GPT-4, are prominent tools with applications extending to various cybersecurity-related fields, such as code analysis [22] and vulnerability repairment [23]. These models are equipped with wide-ranging general knowledge and the capacity for elementary reasoning. They can comprehend, infer, and produce text resembling human communication, aided by a training corpus encompassing diverse domains like computer science and cybersecurity. Their ability to interpret context and recognize patterns enables them to adapt knowledge to new scenarios. This adaptability, coupled with their proficiency in interacting with systems in a human-like way, positions them as valuable assets in enhancing penetration testing processes. Despite inherent limitations, LLMs offer distinct attributes that can substantially aid in the automation and improvement of penetration testing tasks. The realization of this potential, however, requires the creation and application of a specialized and rigorous benchmark.

3. Penetration Testing Benchmark

3.1. Motivation

The fair evaluation of Large Language Models (LLMs) in penetration testing necessitates a robust and representative benchmark. Existing benchmarks in this domain [7], [8] have several limitations. First, they are often restricted in scope, focusing on a narrow range of potential vulnerabilities, and thus fail to capture the complexity and diversity of real-world cyber threats. For instance, the OWASP benchmark *juiceshop* [24] is commonly adopted for evaluating web vulnerability testing. However, it does not touch the concept of privilege escalation, which is an essential aspect of penetration testing. Second, existing benchmarks may not recognize the cumulative value of progress through the different stages of penetration testing, as they tend to evaluate only the final exploitation success. This approach overlooks the nuanced value each step contributes to the overall process, resulting in metrics that might not accurately represent actual performance in real-world scenarios.

To address these concerns, we propose the construction of a comprehensive penetration testing benchmark that meets the following criteria:

Task Variety. The benchmark must encompass diverse tasks, reflecting various operating systems and emulating the diversity of scenarios encountered in real-world penetration testing.

Challenge Levels. To ensure broad applicability, the benchmark must include tasks of varying difficulty levels suitable for challenging novice and expert testers.

Progress Tracking. Beyond mere success or failure metrics, the benchmark must facilitate tracking of incremental progress, thereby recognizing and scoring the value added at each stage of the penetration testing process.

3.2. Benchmark Design

Following the criteria outlined previously, we develop a comprehensive benchmark that closely reflects real-world penetration testing tasks. The design process progresses through several stages.

Task Selection. Our first step is to meticulously select tasks from HackTheBox [9] (HTB) and VulnHub [10]. These platforms are widely recognized and frequently utilized for penetration testing practice. Our selection process is guided by a desire to incorporate a diverse and challenging set of tasks. Capture The Flag (CTF) exercises and real-world testing scenarios have been included. The targets are drawn from various operating systems and encompass a broad spectrum of vulnerabilities. This approach ensures a wide representation of real-world penetration testing tasks. To account for different skill levels, the selected tasks cover a broad range of difficulty. While HTB and VulnHub offer reference difficulty levels, we further validate these with input from three certified penetration testers[2], including the authors of this work. This collaborative process yields a consensus on the final difficulty rating for each target, aligning with the conventional categorization [10] of penetration testing machines into *easy*, *medium*, and *hard* levels. It is worth noting that our benchmark does not explicitly include benign targets for evaluating false positives. This is because the iterative and exploratory nature of penetration testing inherently involves investigating services within the target that may ultimately be deemed benign. In this process, our primary focus is successfully identifying genuine vulnerabilities.

Task Decomposition. We further parse the testing process of each target into a series of sub-tasks, following the standard solution commonly referred to as the "walkthrough" in penetration testing. Each sub-task corresponds to a unique step in the overall process. Specifically, a sub-task may represent a micro-step involving the use of a particular penetration testing tool (e.g., performing port scanning with nmap [25]) or exploiting a unique vulnerability identified in the Common Weakness Enumeration (CWE) [26] (e.g., exploiting SQL injection). To standardize decomposition, we arrange the sub-tasks into a two-layer structure. Initially, we categorize each sub-task according to the five phases of penetration testing, as described in Section 2. Then, we label the sub-task with either the corresponding CWE item it targets or the specific tools employed. These two steps enable us to formulate an exhaustive list of sub-tasks for every benchmark target. We include this list in Appendix 6, and the complete sub-task information is accessible on our anonymous open-source project [18].

Benchmark Validation. The final stage of our benchmark development involves rigorous validation. This step ensures that our benchmark accurately reflects real-world penetration testing scenarios and offers reproducibility. During validation, three certified penetration testers independently attempt the penetration testing targets, refining the sub-tasks as needed. We adjust our task decomposition accordingly because some targets may have multiple valid solutions.

By the end, we compile a benchmark of 13 penetration testing targets with 182 sub-tasks in 25 categories. The benchmark includes all types of vulnerabilities as listed in the OWASP [11] Top 10 Project. Detailed information on the included categories is listed in the Appendix Section 6. To contribute to community development, we have made this benchmark publicly available online at our anonymous project website [18].

4. Exploratory Study

We conduct an exploratory study to assess the capabilities of LLMs in penetration testing. Our primary objective is determining how well LLMs can adapt to the real-world complexities and challenges associated with penetration testing tasks. Specifically, we aim to address the following two research questions:

RQ1 (Capability): To what extent can LLMs perform penetration testing tasks?

RQ2 (Comparative Analysis): How do the problem-solving strategies of human penetration testers and LLMs differ?

We utilize the benchmark described in Section 3 to evaluate the performance of LLMs on penetration testing tasks. In the following, we first delineate our testing strategy for this study. Subsequently, we present the testing results and an analytical discussion to address the above research questions.

4.1. Testing Strategy

LLMs cannot perform penetration tests directly. Their capabilities are primarily text-based, responding to queries and providing suggestions. However, penetration testing often involves operations with user interfaces (UI) and understanding graphical information, such as website images. This necessitates a bridge between the test machine and the LLM to facilitate task completion.

We introduce an interactive loop structure to evaluate the LLM's abilities in penetration testing within our benchmark. This process, depicted in Figure 2, consists of the following stages: ❶ We present the target information to the LLM and request recommendations for penetration testing actions. This initiates a looped testing procedure. ❷ We implement the actions suggested by the LLM, which encompass both terminal commands and graphical interactions. ❸ We gather the results of the actions. Text-based output, such as terminal responses or source code, is recorded directly. Human penetration testers provide concise summaries and descriptions for non-textual results (e.g., images). The summarized information is returned to the LLM to inform subsequent actions. ❹ This cycle continues until we identify a solution or reach a standstill. We compile a record of the testing procedures, encompassing successful tasks, ineffective actions, and any reasons for failure, if applicable.

2. Our penetration testers are all Offensive Security Certified Professionals (OSCP).

TABLE 1: Overall performance of LLMs on Penetration Testing Benchmark.

	Easy		Medium		Hard		Average	
Tools	Overall (7)	Sub-task (77)	Overall (4)	Sub-task (71)	Overall (2)	Sub-task (34)	Overall (13)	Sub-task (182)
GPT-3.5	1 (14.29%)	24 (31.17%)	0 (0.00%)	13 (18.31%)	0 (0.00%)	5 (14.71%)	1 (7.69%)	42 (23.07%)
GPT-4	4 (57.14%)	52 (67.53%)	1 (25.00%)	27 (38.03%)	0 (0.00%)	8 (23.53%)	5 (38.46%)	87 (47.80%)
Bard	2 (28.57%)	29 (37.66%)	0 (0.00%)	16 (22.54%)	0 (0.00%)	5 (14.71%)	2 (15.38%)	50 (27.47%)
Average	2.3 (33.33%)	35 (45.45%)	0.33 (8.33%)	18.7 (26.29%)	0 (0.00%)	6 (17.64%)	2.7 (20.5%)	59.7 (32.78%)

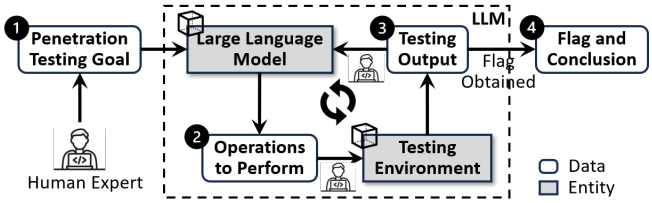


Figure 2: Overview of strategy to use LLMs for penetration testing.

4.2. Evaluation Settings

We proceed to assess the performances of various LLMs in penetration testing tasks using the strategy mentioned above.

Model Selection. Our study focuses on three cutting-edge LLMs that are currently accessible: GPT-3.5 and GPT-4 from OpenAI and LaMDA [27] from Google. These models are selected based on their prominence in the research community and consistent availability. To interact with the LLMs mentioned above, we utilize chatbot services provided by OpenAI and Google, namely ChatGPT [28] and Bard [14]. For this paper, the terms GPT-3.5, GPT-4, and Bard will represent these three LLMs.

Experimental Setup. We conduct our experiments in a local environment where the target and testing machines are part of the same private network. The testing machine operates on Kali Linux [29], version 2023.1. Several measures are implemented to validate the effectiveness of our testing procedures. First, we repeat the tests to account for inherent variability in the LLM outputs. In particular, we test each target with each LLM five times. We performed 195 tests in total, i.e., 5 repetitions * 3 models * 13 targets. In this process, a sub-task is considered successful if it succeeds in at least one trial, and a penetration task is considered successful as long as one trial succeeds. Second, we make the best efforts to translate UI operations and graphical information into natural languages accurately. In addition, we ensure the precise execution of the instructions provided by the LLMs. Third, we maintain the integrity of the testing process by strictly limiting the tester's role to executing actions and reporting results without adding expert knowledge or guidance. Finally, the testing and target machines are rebooted after each test to reset their states, ensuring a consistent starting point for each test.

Tool Usage. Our study aims to assess the innate capabilities of LLMs without reliance on automated vulnerability scanners such as Nexus [30] and OpenVAS [31]. Consequently, we explicitly instruct the LLMs to refrain from using these tools. However, we follow the LLMs' recommendations for utilizing other tools designed to validate specific vulnerability types (e.g., sqlmap [32] for SQL injections). Occasionally, versioning discrepancies may lead the LLMs to provide incorrect instructions for tool usage. In such instances, our penetration testing experts evaluate whether the instructions would have been valid for a previous version of the tool. They then make any necessary adjustments to ensure the tool's correct operation.

4.3. Capability Evaluation (RQ1)

To study **RQ1**, we begin by assessing the overall performance of three prominent LLMs: GPT-4, Bard, and GPT-3.5. The results of these evaluations are compiled in Table 1. The experimental results show that the three LLMs completed at least one end-to-end penetration testing task. This achievement underscores their ability to conduct a broad spectrum of testing operations, particularly within environments of less complexity. Among the models, GPT-4 stands out with superior performance, achieving success with 4 targets of easy difficulty and 1 of medium difficulty. Bard and GPT-3.5 also demonstrate commendable capabilities, completing 2 and 1 targets of easy difficulty, respectively. When examining sub-tasks, GPT-4 accomplishes 52 of 77 on easy difficulty targets and 27 out of 71 on medium ones, underlining its potential for significant contributions to more complex penetration testing scenarios. Though not as proficient as GPT-4, GPT-3.5 and Bard still show promise, completing 13 (18.31%) and 16 (22.54%) of sub-tasks on medium difficulty targets, respectively. However, the performance of all three models noticeably diminishes when challenged with hard difficulty targets. While each model can complete the initial reconnaissance phase on these targets, they fall short in exploiting the identified vulnerability. This outcome is not entirely unexpected, as the hard difficulty machines are deliberately crafted to be exceedingly difficult. They often include services that appear vulnerable but are, in fact, non-exploitable—a trait commonly referred to as *rabbit holes* [33]. Additionally, the routes to successfully exploiting these machines are typically inventive and unforeseeable, making them resistant to straightforward replication by automated tools. For instance, the benchmark target *Falafel* involves deliberately crafted SQL injection vulnerabilities, which are resistant to *sqlmap* and can only be exploited through manually designed payloads. Existing LLMs do

not exhibit the capability to solve them solely without the guidance of human experts.

Finding 1: Large Language Models (LLMs) have shown proficiency in conducting end-to-end penetration testing tasks but struggle to overcome challenges presented by more difficult targets.

TABLE 2: Top 10 Types of Sub-tasks completed by each tool.

Sub-Tasks	Walkthrough	GPT-3.5	GPT-4	Bard
General Tool Usage	25	4	10	7
Port Scanning	9	9	9	9
Web Enumeration	18	4	8	4
Code Analysis	18	4	5	4
Shell Construction	11	3	7	4
Directory Exploitation	11	1	7	1
General Privilege Escalation	8	2	4	3
Flag Capture	8	1	5	2
Passowrd/Hash Cracking	8	2	4	2
Network Exploitation	7	1	3	2

We further examine the detailed sub-task completion performances of the three LLMs, as presented in Table 2. Analyzing the completion status, we identify several areas where LLMs excel. First, they adeptly utilize common penetration testing tools to interpret the corresponding outputs, especially in enumeration tasks correctly. For example, all three evaluated LLMs successfully perform all nine *Port Scanning* sub-tasks. They can configure the widely-used port scanning tool, *nmap* [25], comprehend the scan results, and formulate subsequent actions. Second, the LLMs reveal a deep understanding of prevalent vulnerability types, connecting them to the services on the target system. This understanding is evidenced by the successful completion of sub-tasks related to various vulnerability types. Finally, LLMs demonstrate their effectiveness in code analysis and generation, particularly in the tasks of *Code Analysis* and *Shell Construction*. These tasks require the models to read and generate codes in different programming languages, essential in penetration testing. This often culminates in identifying potential vulnerabilities from code snippets and crafting the corresponding exploits. Notably, GPT-4 outperforms the other two models regarding code interpretation and generation, marking it the most suitable candidate for penetration testing tasks.

Finding 2: LLMs can efficiently use penetration testing tools, identify common vulnerabilities, and interpret source codes to identify vulnerabilities.

4.4. Comparative Analysis (RQ2)

To address **RQ2**, we examine the problem-solving strategies that LLMs employ, contrasting them with human penetration testers. In each penetration testing trial, we concentrate on two main aspects: (1) Identifying the unnecessary operations that LLMs prompt, which are not conducive to successful penetration testing, as compared to a standard walkthrough; and (2) Understanding the specific factors that prevent LLMs from successfully executing penetration tests.

TABLE 3: Top Unnecessary Operations Prompted by LLMs on the Benchmark Targets

Unnecessary Operations	GPT-3.5	GPT-4	Bard	Total
Brute-Force	75	92	68	235
CVE Study	29	24	28	81
SQL Injection	14	21	16	51
Command Injection	18	7	12	37

TABLE 4: Top causes for failed penetration testing trials

Failure Reasons	GPT3.5	GPT4	Bard	Total
Session context lost	25	18	31	74
False Command Generation	23	12	20	55
Deadlock operations	19	10	16	45
False Scanning Output Interpretation	13	9	18	40
False Source Code Interpretation	16	11	10	37
Cannot craft valid exploit	11	15	8	34

We analyze the unnecessary operations prompted by LLMs by breaking down the recorded testing procedures into sub-tasks. We employ the same method to formulate benchmark sub-tasks, as Section 3 outlines. By comparing this to a standard walkthrough, we identify the primary sub-task trials that fall outside the standard walkthrough and are thus irrelevant to the penetration testing process. The results are summarized in Table 3. We find that the most prevalent unnecessary operation prompted by LLMs is brute force. For all services requiring password authentication, LLMs typically advise brute-forcing it. This is an ineffective strategy in penetration testing. We surmise that many hacking incidents in enterprises involve password cracking and brute force. LLMs learn these reports from accident reports and are consequently considered viable solutions. Besides brute force, LLMs suggest that testers engage in CVE studies, SQL injections, and command injections. These recommendations are common, as real-world penetration testers often prioritize these techniques, even though they may not always provide the exact solution.

We further investigate the reasons behind the failure of penetration testing trials. We manually categorize the causes of failure for the 195 penetration testing trials, with the results documented in Table 4. This table reveals that the predominant cause of failure is the loss of session context. The three examined models face difficulties in maintaining long-term conversational memory uniformly, frequently forgetting previous test results as the dialogue progresses. This lack of retention may be attributable to the limited token size within the LLM conversation context. Given the intricate nature of penetration testing—where a tester must skillfully link minor vulnerabilities across different services to develop a coherent exploitation strategy—this loss of context substantially undermines the models' effectiveness.

Finding 3: LLMs struggle to maintain long-term memory, which is vital to link vulnerabilities and develop exploitation strategies effectively.

Secondly, LLMs strongly prefer the most recent tasks, adhering rigorously to a depth-first search approach. They concentrate on exploiting the immediate service, rarely deviating to a new target until all potential paths for the current one have been pursued. This can be attributed to the attention of LLMs focusing more on the beginning and end of the prompt, as revealed in [34]. Experienced penetration testers generally assess the system from a broader standpoint, strategizing the subsequent steps likely to provide the most substantial results. When combined with the aforementioned memory loss issue, this tendency causes LLMs to become overly fixated on a specific service. As the test progresses, the models completely forget previous findings and reach a deadlock.

Finding 4: LLMs strongly prefer recent tasks and a depth-first search approach, often resulting in an overfocus on one service and forgetting previous findings.

Lastly, LLMs have inaccurate result generation and hallucination issues, as noted in [35]. This phenomenon ranks as the second most frequent cause of failures and is characterized by the generation of false commands. In our study, we observe that LLMs frequently identify the appropriate tool for the task but stumble in configuring the tools with the correct settings. In some cases, they even concoct non-existent testing tools or tool modules.

Finding 5: LLMs may generate inaccurate operations or commands, often stemming from inherent inaccuracies and hallucinations.

Our exploratory study of three LLMs within penetration testing reveals their potential for executing end-to-end tasks. Nevertheless, challenges arise in maintaining long-term memory, devising a testing strategy beyond a depth-first approach, and generating accurate operations. In the following section, we elucidate how we address these challenges and outline our strategy for designing our LLM-powered penetration testing tool.

5. Methodology

5.1. Overview

In light of the challenges identified in the preceding section, we present our proposed solution, PENTESTGPT, which leverages the synergistic interplay of three LLM-powered modules. As illustrated in Figure 3, PENTESTGPT incorporates three core modules: the **Reasoning Module**, the **Generation Module**, and the **Parsing Module**. Each module reserves one LLM session with its conversation and context. The user interacts seamlessly with PENTESTGPT, where distinct modules process different types of messages. This interaction culminates in a final decision, suggesting the subsequent step of the penetration testing process that the user should undertake. In the following sections, we elucidate our design reasoning and provide a detailed breakdown of the engineering processes behind PENTESTGPT.

5.2. Design Rationale

Our central design considerations emerged from the three challenges observed in the previous Exploratory Study (Section 4): The first challenge (*Finding 3*) pertains to the issue of penetration testing context loss due to memory retention. LLMs in their original form struggle to maintain such long-term memory due to token size limits. The second obstacle (*Finding 4*) arises from the LLM chatbots' tendency to emphasize recent conversation content. In penetration testing tasks, this focuses on optimizing the immediate task. This approach falls short in the complex, interconnected task environment of penetration testing. The third obstacle (*Finding 5*) is tied to the inaccurate results generation by LLMs. When tasked to produce specific operations for a step in penetration testing directly, the outputs are often imprecise, sometimes even leading to

PENTESTGPT has been engineered to address these challenges, rendering it more apt for penetration testing tasks. We drew inspiration from the methodologies employed by real-world penetration testing teams, where a director plans overarching procedures, subdividing them into subtasks for individual testers. Each tester independently performs their task, reporting results without an exhaustive understanding of the broader context. The director then determines the following steps, possibly redefining tasks, and triggers the subsequent round of testing. Essentially, the director manages the overall strategy without becoming entrenched in the minutiae of the tests. This approach is mirrored in PENTESTGPT's functionality, enhancing its efficiency and adaptability in conducting penetration tests. Our strategy divides penetration testing into two processes: identifying the next task and generating the concrete operation to complete the task. Each process is powered by one LLM session. In this setup, the LLM session responsible for task identification retains the complete context of the ongoing penetration testing status. At the same time, the generation of detailed operations and parsing of information is managed by other sessions. This division of responsibilities fosters effective task execution while preserving the overarching context.

To assist LLMs in effectively carrying out penetration testing tasks, we design a series of prompts that align with user inputs. We utilize the Chain-of-Thought (CoT) [36] methodology during this process. As CoT reveals, LLMs' performance and reasoning capabilities can be significantly enhanced using the *input*, *chain-of-thought*, *output* prompting format. Here, the *chain-of-thought* represents a series of intermediate natural language reasoning steps leading to the outcome. We dissect the penetration testing tasks into micro-steps and design prompts with examples to guide LLMs through processing penetration testing information

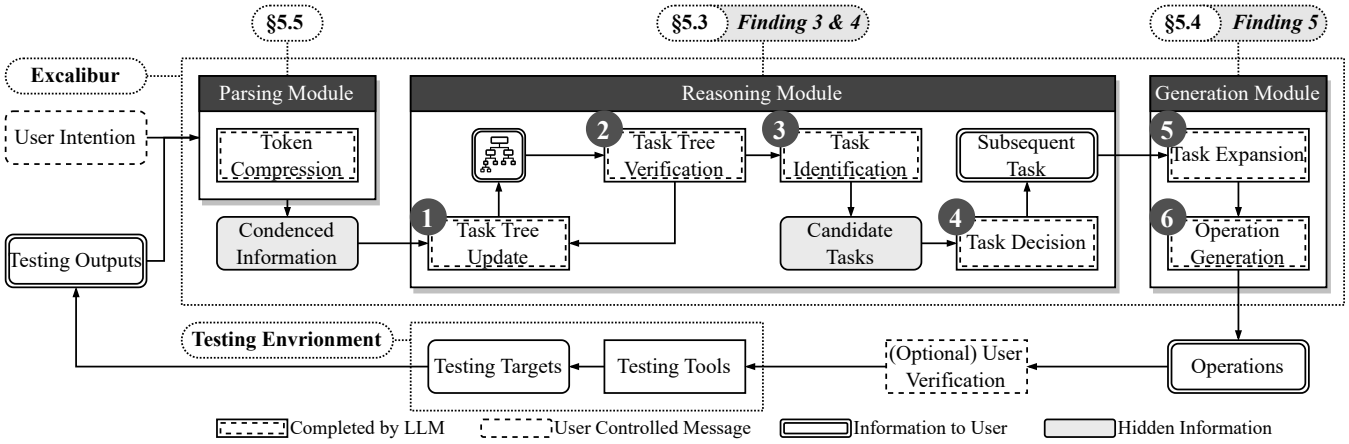


Figure 3: Overview of PENTESTGPT.

step-by-step, ultimately leading to the desired outcomes. The complete prompts are available at our anonymized open-source project [18].

5.3. Reasoning Module

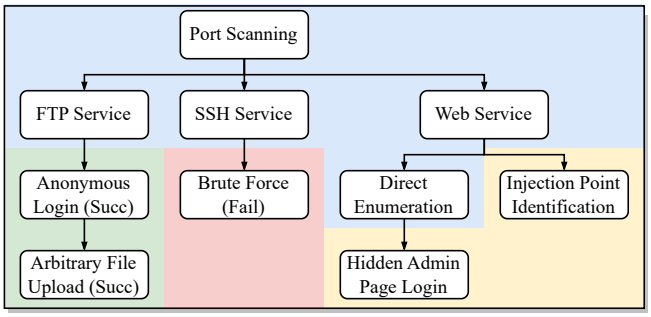
The **Reasoning Module** plays a pivotal role in our system, analogous to a team lead overseeing the penetration testing task from a macro perspective. It obtains testing results or intentions from the user and prepares the testing strategy for the next step. This testing strategy is passed to the generation module for further planning.

To effectively supervise the penetration testing process and provide precise guidance, it is crucial to translate the testing procedures and outcomes into a natural language format. Drawing inspiration from the concept of an attack tree [37], which is often used to outline penetration testing procedures, we introduce the notion of a *pentesting task tree* (PTT). This novel approach to testing status representation is rooted in the concept of an *attributed tree* [38]:

Definition 1 (Attributed Tree). A *attributed tree* is an edge-labeled, attributed polytree $G = (V, E, \lambda, \mu)$ where V is a set of nodes (or vertices), E is a set of directed edges, $\lambda : E \to \Sigma$ is an edge labeling function assigning a label from the alphabet Σ to each edge and $\mu : (V \cup E) \times K \to S$ is a function assigning key(from K)-value(from S) pairs of properties to the edges and nodes.

Given the definition of *attributed tree*, PTT is defined as follows:

Definition 2 (Pentesting Task Tree). An PTT T is a pair (N, A), where: (1) N is a set of nodes organized in a tree structure. Each node has a unique identifier, and there is a special node called the root that has no parent. Each node, other than the root, has exactly one parent and zero or more children. (2) A is a function that assigns to each node $n \in N$ a set of attributes $A(n)$. Each attribute is a pair (a, v), where a is the attribute name and v is the attribute value. The set of attributes can be different for each node.



a) PTT Representatoin

```
Task Tree:
1. Perform port scanning (completed)
  - Port 21, 22 and 80 are open.
  - Services are FTP, SSH, and Web Service.
2. Perform the testing
  2.1 Test FTP Service
    2.1.1 Test Anonymous Login (success)
      2.1.1.1 Test Anonymous Upload (success)
  2.2 Test SSH Service
    2.2.1 Brute-force (failed)
  2.3 Test Web Service (ongoing)
    2.3.1 Directory Enumeration
      2.3.1.1 Find hidden admin (to-do)
    2.3.2 Injection Identification (todo)
```

b) PTT Representation in Natural Language

Figure 4: Pentesting Task Tree in a) visualized tree format, and b) natural language format encoded in LLM.

As outlined in Figure 3, the Reasoning Module's operation unfolds over four key steps operating over the PTT. ❶ Initially, the module absorbs the user's intentions to construct an initial PTT in the form of natural language. This is achieved by carefully instructing the LLM with examples and definitions of PPT using meticulously crafted prompts. The LLM outputs are parsed to confirm that the tree structure is accurately formatted. Note that due to the nature of the tree structure, it can be represented in the natural language format through layered bullets, as illustrated in Figure 4. The Reasoning Module effectively

overcomes the memory-loss issue by maintaining a task tree that encompasses the entire penetration testing process. ❷ After updating the tree information, a verification step is conducted on the newly updated PTT to ascertain its correctness. This process checks explicitly that only the leaf nodes of the PTT have been modified, aligning with the principle that atomic operations in the penetration testing process should only influence the status of the lowest-level sub-tasks. This step confirms the correctness of the reasoning process, safeguarding against any potential alterations to the overall tree structure due to hallucination by the LLM. If discrepancies arise, the information is reverted to the LLM for correction and regeneration. ❸ With the updated PTT, the Reasoning Module evaluates the current tree state and pinpoints viable sub-tasks that can serve as candidate steps for further testing. ❹ Finally, the module evaluates the likelihood of these sub-tasks leading to successful penetration testing outcomes. It then recommends the top task as the output. The expected results of this task are subsequently forwarded to the Generation Module for an in-depth analysis. This is feasible, as demonstrated in the exploratory study, since LLMs, particularly GPT-4, can identify potential vulnerabilities when provided with system status information. This procedural approach enables the Reasoning Module to address one of the inherent limitations of LLMs, precisely their tendency to concentrate solely on the most recent task. Note that in cases where the tester identifies that the correct task is incorrect or not completed in a preferred way, he could also manually revise the PTT through the interactive handle further discussed in Section 5.6.

We devise four sets of prompts to sequentially guide the Reasoning Module through the completion of each stage. To bolster the reproducibility of our results, we optimize these prompts further with a technique known as hint generation [39]. From our practical experience, we observe that LLMs are adept at interpreting the tree-structured information pertinent to penetration testing and can update it accurately in response to test outputs.

5.4. Generation Module

The Generation Module translates specific sub-tasks from the Reasoning Module into concrete commands or instructions. Each time a new sub-task is received, a fresh session is initiated in the Generation Module. This strategy effectively isolates the context of the overarching penetration task from the immediate task under execution, enabling the LLM to focus entirely on generating specific commands.

Instead of directly transforming the received sub-task into specific operations, our design employs the CoT strategy [36] to partition this process into two sequential steps. This design decision directly addresses the challenges associated with model inaccuracy and hallucination by enhancing the model's reasoning capability. In particular, ❺ upon the receipt of a concise sub-task from the Reasoning Module, the Generation Module begins by expanding it into a sequence of detailed steps. Notably, the prompt associated with this sub-task requires the LLM to consider the possible tools and operations available within the testing environment. ❻ Subsequently, the Generation Module transforms each of these expanded steps into precise terminal commands ready for execution or into detailed descriptions of specific Graphical User Interface (GUI) operations to be carried out. This stage-by-stage translation eliminates potential ambiguities, enabling testers to follow the instructions directly and seamlessly. Implementing this two-step process effectively precludes the LLM from generating operations that may not be feasible in real-world scenarios, thereby improving the success rate of the penetration testing procedure.

By acting as a bridge between the strategic insights provided by the Reasoning Module and the actionable steps required for conducting a penetration test, the Generation Module ensures that high-level plans are converted into precise and actionable steps. This transformation process significantly bolsters the overall efficiency of the penetration testing procedure.

An Illustrative Example. We utilize a real-world running example to illuminate how the Reasoning Module and the Generation Module collaboratively operate to complete penetration testing tasks. Figure 5 illustrates a single iteration of PENTESTGPT working on the HackTheBox machine Carrier [40], a medium-difficulty target. As depicted in a-1), the PTT, in natural language format, encodes the testing status, revealing the open ports (21, 22,80) on the target machine. The Reasoning Module is subsequently instructed to identify the available tasks. As highlighted in red, service scanning is the only available task on the leaf node of the PTT. This task is therefore chosen and forwarded to the Generation Module for command generation. The generated command is executed in the testing environment, and the execution result is conveyed to the Reasoning Module to update the PTT. In a-2), the Reasoning Module integrates the previous scanning result into the PTT, cross-referencing it with the earlier PTT to update only the leaf nodes. It then looks for the available tasks to execute. In this case, two tasks emerge: scanning the web service on port 80 and checking the SSH service for known vulnerabilities. The LLM evaluates which task is more promising and chooses to investigate the web service, often seen as more vulnerable. This task is passed to the Generation Module. The Generation Module turns this general task into a detailed process, employing *nikto* [41], a commonly used web scanning script. The iterative process continues until the tester completes the penetration testing task.

5.5. Parsing Module

The **Parsing Module** operates as a supportive interface, enabling effective processing of the natural language information exchanged between the user and the other two core modules. Two needs can primarily justify the existence of this module. First, security testing tool outputs are typically verbose, laden with extraneous details, making it computationally expensive and unnecessarily redundant to feed

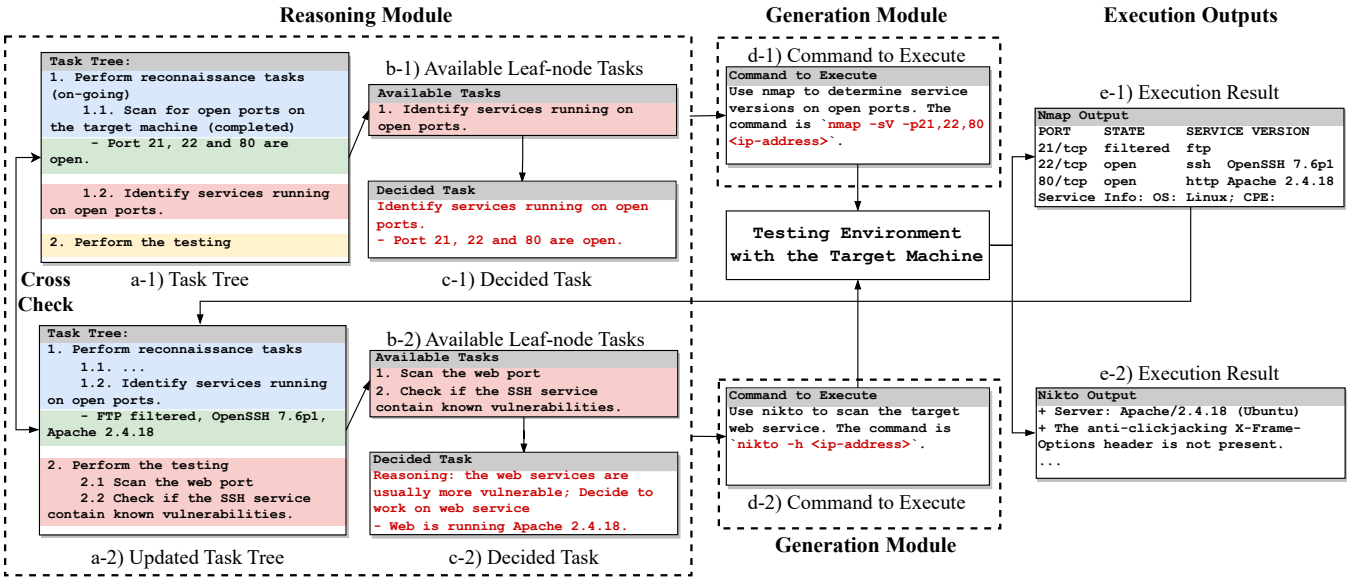


Figure 5: A demonstration of the task-tree update process on the testing target *HTB-Carrier*

these extended outputs directly into the LLMs. Second, users without specialized knowledge in the security domain may struggle to extract key insights from security testing outputs, presenting challenges in summarizing crucial testing information. Consequently, the Parsing Module is essential in streamlining and condensing this information.

In PENTESTGPT, the Parsing Module is devised to handle four distinct types of information: (1) *user intentions*, which are directives provided by the user to dictate the next course of action, (2) *security testing tool outputs*, which represent the raw outputs generated by an array of security testing tools, (3) *raw HTTP web information*, which encompasses all raw information derived from HTTP web interfaces, and (4) *source codes* extracted during the penetration testing process. Users must specify the category of the information they provide, and each category is paired with a set of carefully designed prompts. For source code analysis, we integrate the GPT-4 code interpreter [42] to execute the task.

5.6. Active Feedback

While LLMs can produce insightful outputs, their outcomes may sometimes require revisions. To facilitate this, we introduce an interactive handle in PENTESTGPT, known as active feedback, which allows the user to interact directly with the Reasoning Module. A vital feature of this process is that it does not alter the context within the Reasoning Module unless the user explicitly desires to update some information. The reasoning context, including the PTT, is stored as a fixed chunk of tokens. This chunk of tokens is provided to a new LLM session during an active feedback interaction, and users can pose questions regarding them. This ensures that the original session remains unaffected, and users can always query the reasoning context without making unnecessary changes. If the user believes it necessary to update the PTT, they can explicitly instruct the model to update the reasoning context history accordingly. This provides a robust and flexible framework for the user to participate in the decision-making process actively.

5.7. Discussion

We explore various design alternatives for PENTESTGPT to tackle the challenges identified in Exploratory Study. We have experimented with different designs, and here we discuss some key decisions.

Addressing Context Loss with Token Size: a straightforward solution to alleviate context loss is the employment of LLM models with an extended token size. For instance, GPT-4 provides versions with 8k and 32k token size limits. This approach, however, confronts two substantial challenges. First, even a 32k token size might be inadequate for penetration testing scenarios, as the output of a single testing tool like *dirbuster* [43] may comprise thousands of tokens. Consequently, GPT-4 with a 32k limit cannot retain the entire testing context. Second, even when the entire conversation history fits within the 32k token boundary, the API may still skew towards recent content, focusing on local tasks and overlooking broader context. These issues guided us in formulating the design for the Reasoning Module and the Parsing Module.

Vector Database to Improve Context Length: Another technique to enhance the context length of LLMs involves a vector database [44], [45]. By transmuting data into vector embeddings, LLMs can efficiently store and retrieve information, practically creating long-term memory. Theoretically, penetration testing tool outputs could be archived

in the vector database. In practice, though, we observe that many results closely resemble and vary in only nuanced ways. This similarity often leads to confused information retrieval. Solely relying on a vector database fails to overcome context loss in penetration testing tasks. Integrating the vector database into the design of PENTESTGPT is an avenue for future research.

Precision in Information Extraction: Precise information extraction is crucial for conserving token usage and avoiding verbosity in LLMs. Rule-based methods are commonly employed to extract diverse information. However, rule-based techniques are engineeringly expensive given natural language's inherent complexity and the variety of information types in penetration testing. We devise the Parsing Module to manage several general input information types, a strategy found to be both feasible and efficient.

Limitations of LLMs: LLMs are not an all-encompassing solution. Present LLMs exhibit flaws, including hallucination [46] and outdated knowledge. Our mitigation efforts, such as implementing task tree verification to ward off hallucination, might not completely prevent the Reasoning Module from producing erroneous outcomes. Thus, a human-in-the-loop strategy becomes vital, facilitating the input of necessary expertise and guidance to steer LLMs effectively.

6. Evaluation

In this section, we assess the performance of PENTESTGPT, focusing on the following four research questions:

RQ3 (Performance): How does the performance of PENTESTGPT compare with that of native LLM models and human experts?

RQ4 (Strategy): Does PENTESTGPT employ different problem-solving strategies compared to those utilized by LLMs or human experts?

RQ5 (Ablation): How does each module within PENTESTGPT contribute to the overall penetration testing performance?

RQ6 (Practicality): Is PENTESTGPT practical and effective in real-world penetration testing tasks?

6.1. Evaluation Settings

We implement PENTESTGPT with 1,700 lines of Python3 code and 740 prompts, available at our anonymized project website [18]. We evaluate its performance over the benchmark constructed in Section 3. In this evaluation, we integrate PENTESTGPT with GPT-3.5 and GPT-4 to form two working versions: PENTESTGPT-GPT-3.5 and PENTESTGPT-GPT-4. Due to the lack of API access, we do not select other LLM models, such as Bard. In line with our previous experiments, we use the same experiment environment setting and instruct PENTESTGPT to only use the non-automated penetration testing tools.

6.2. Performance Evaluation (RQ3)

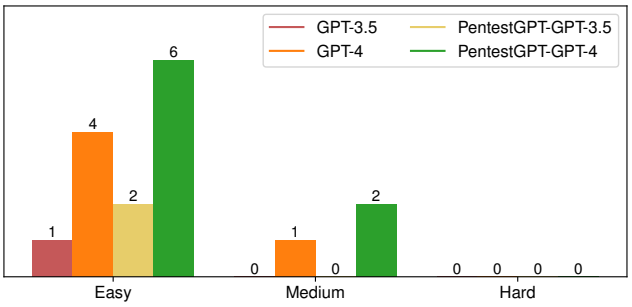
The overall task completion status of PENTESTGPT-GPT-3.5, PENTESTGPT-GPT-4, and the naive usage of LLMs is illustrated in Figure 6a. As the Figure shows, our solutions powered by LLMs demonstrate superior penetration testing capabilities compared to the naive application of LLMs. Specifically, PENTESTGPT-GPT-4 surpasses the other three solutions, successfully solving 6 out of 7 easy difficulty targets and 2 out of 4 medium difficulty targets. This performance indicates that PENTESTGPT-GPT-4 can handle penetration testing targets ranging from easy to medium difficulty levels. Meanwhile, PENTESTGPT-GPT-3.5 manages to solve only two challenges of easy difficulty, a discrepancy that can be attributed to GPT-3.5 lacking the knowledge related to penetration testing found in GPT-4.

The sub-task completion status of PENTESTGPT-GPT-3.5, PENTESTGPT-GPT-4, and the naive usage of LLM is shown in Figure 6b. As the Figure illustrates, both PENTESTGPT-GPT-3.5 and PENTESTGPT-GPT-4 perform better than the standard utilization of LLMs. It is noteworthy that PENTESTGPT-GPT-4 not only solves one more medium difficulty target compared to naive GPT-4 but also accomplishes 111% more sub-tasks (57 vs. 27). This highlights that our design effectively addresses context loss challenges and leads to more promising testing results. Nevertheless, all the solutions struggle with hard difficulty testing targets. As elaborated in Section 4, hard difficulty targets typically demand a deep understanding from the penetration tester. To reach testing objectives, they may require modifications to existing penetration testing tools or scripts. Our design does not expand the LLMs' knowledge of vulnerabilities, so it does not notably enhance performance on these more complex targets.

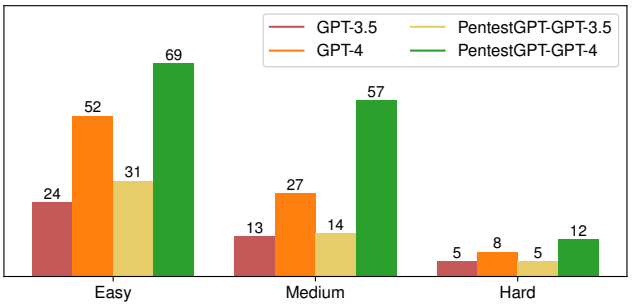
6.3. Strategy Evaluation (RQ4)

We then investigate the problem-solving strategies employed by PENTESTGPT, contrasting them with those of LLMs and human experts. By manually analyzing the penetration testing process of PENTESTGPT, we synthesize its underlying approaches to problem-solving. We surprisingly find that PENTESTGPT decomposes the penetration testing task in a manner akin to human experts, successfully achieving the overall goal. Instead of focusing solely on the most recently discovered task, PENTESTGPT can pinpoint potential sub-tasks likely to lead to successful outcomes.

Figure 7 provides an illustrative example, demonstrating the strategic differences between GPT-4 and PENTESTGPT while handling the VulnHub machine, *Hackable II* [47]. This target comprises two vulnerable services: an FTP service allowing arbitrary file uploads and a web service enabling file viewing through FTP. A successful exploit necessitates exploiting both services by uploading a malicious PHP shell via the FTP service and triggering it through the web service. As depicted in the figure, GPT-4 begins by enumerating the FTP service and successfully identifies the file upload vulnerability (❶-❸). However, it fails to correlate



(a) Overall completion status.



(b) Subtask completion status.

Figure 6: The performance of GPT-3.5, GPT-4, PENTESTGPT-GPT-3.5, and PENTESTGPT-GPT-4 on overall target completion and sub-task completion.

this with the web service, resulting in an incomplete exploit in the following steps. Conversely, PENTESTGPT follows a more holistic approach, toggling between enumerating the FTP service and browsing the web service. In particular, PENTESTGPT firstly ❶ enumerates the FTP service and ❷ web service to understand the general situation. It then ❸ prioritizes the FTP service, and ❹ eventually discovers the file upload vulnerability. More importantly, in this process, PENTESTGPT identifies that files available on FTP are the same as those on the web service. By connecting these findings, PENTESTGPT guides the tester to ❺ perform a shell upload, ❻ leading to a successful reverse shell. This strategy aligns with the walkthrough solution and highlights PENTESTGPT's comprehensive understanding of the penetration testing process and its ability to make effective decisions on the optimal sub-task to pursue next. This reveals PENTESTGPT's strategic thinking and ability to integrate different aspects of the testing process.

Our second observation is that although PENTESTGPT behaves more similarly to human experts, it still exhibits some strategies that humans will not apply. For instance, PENTESTGPT still prioritizes brute-force attacks before vulnerability scanning. This is obvious in cases where PENTESTGPT always tries to brute-force the SSH service on target machines.

We then analyze the failed penetration testing cases to understand the limitations of PENTESTGPT. Beyond the absence of some advanced penetration testing techniques, two primary issues emerge. First, PENTESTGPT struggles

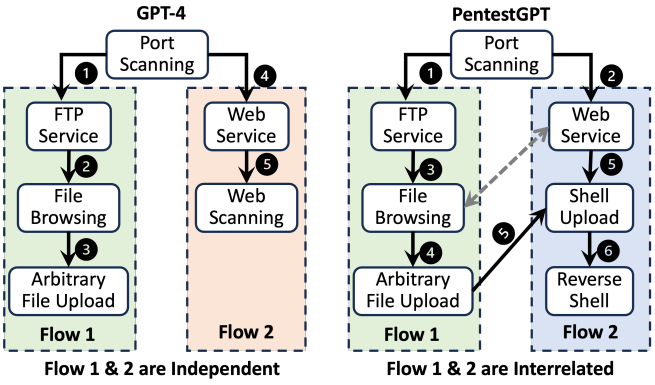


Figure 7: Penetration testing strategy comparison between GPT-3.5 and PENTESTGPT on *VulnHub-Hackable II*.

to interpret images. LLMs are limited to text comprehension, so they cannot accurately process images. This issue might be addressed by developing large multimodal models to understand text and visual data. Second, it cannot grasp certain social engineering tricks and subtle cues. For instance, real-world penetration testers often create brute-force wordlists using information gathered from the target service. Though PENTESTGPT can retrieve a list of names from a web service, it fails to instruct the use of tools to create a wordlist from those names. These limitations underline the necessity for improvement in areas where human insight and intricate reasoning are still more proficient than automated solutions.

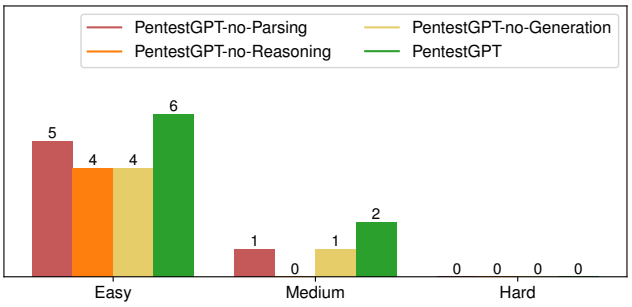
6.4. Ablation Study (RQ5)

We perform an ablation study on how the three modules: Reasoning Module, Generation Module, and Parsing Module, contribute to the performance of PENTESTGPT. We implement three variants:

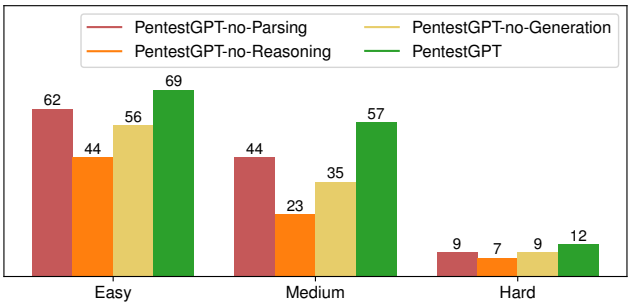
1) PENTESTGPT-NO-PARSING: the Parsing Module is deactivated, causing all data to be directly fed into the system.
2) PENTESTGPT-NO-GENERATION: the Generation Module is deactivated, leading to the completion of task generation within the Reasoning Module itself. The prompts for task generation remain consistent.
3) PENTESTGPT-NO-REASONING: the Reasoning Module is desabled. Instead of PTT, this variant adopts the same methodology utilized with LLMs for penetration testing, as delineated in the Exploratory Study.

All the variants are integrated with GPT-4 API for testing.

The results of the three variants tested on our penetration testing benchmarks are depicted in Figure 8. In general, PENTESTGPT demonstrates superiority over the three ablation baselines regarding overall target and sub-task completion. Our key findings are as follows: (1) In the absence of the Parsing Module, PENTESTGPT-NO-PARSING attains marginally lower performance in overall task and sub-task completion relative to the full configuration. While parsing information is advantageous in penetration testing,



(a) Overall completion status



(b) Sub-task completion status

Figure 8: The performance of PENTESTGPT, PENTESTGPT-NO-ANNOTATION, PENTESTGPT-OPERATION-ONLY, and PENTESTGPT-PARAMETER-ONLY on both normalized average code coverage (μLOC) and bug detection.

the 32k token size limit often suffices for various outputs. Given the Reasoning Module's inherent design to maintain the entire testing context, the lack of the Parsing Module does not substantially impair the tool's performance. (2) PENTESTGPT-NO-REASONING fares the worst, completing only 53.6% of the sub-tasks achieved by the full solution, an outcome even inferior to the naive application of GPT-4 in testing. We attribute this to the Generation Module adding supplementary sub-tasks to the LLM context. Since the prompts are not tailored for scenarios without the Reasoning Module, the resulting outputs are irrelevant for the naive LLM without the Generation Module. Furthermore, the extended generation output obscures the original context, hindering the LLM's ability to concentrate on the task, thus failing the test. (3) PENTESTGPT-NO-GENERATION realizes performance slightly above that of GPT-4 employed naively. This occurs because, without the Generation Module, the testing procedure closely resembles the usage of LLMs. Notably, the Generation Module is principally intended to guide the tester in executing precise penetration testing operations. Without this module, the tester may depend on supplementary information to operate the tools or scripts essential for completing the test.

6.5. Practicality Study (RQ6)

We demonstrate that PENTESTGPT exhibits practicality for real-world penetration testing beyond the crafted benchmark. For this purpose, we engage PENTESTGPT in the HackTheBox active machine challenges, a series of penetration testing objectives open to global testers. Each challenge consists of two components: a user flag, retrievable upon initial user access, and a root flag, obtainable after gaining root access. Our evaluation encompasses five targets of easy difficulty and five of medium difficulty. During this exercise, PENTESTGPT, utilizing GPT-4's 32k token API, conducts up to five tests on each target. Success is defined solely by the capture of the root flag. Table 5 details the performance of PENTESTGPT in these challenges[3]. Ultimately, PENTESTGPT completes three easy and five medium challenges. The total expenditure for this exercise amounts to 131.5 USD, averaging 21.92 USD per target. This cost is markedly lower than employing human penetration testers and falls within an acceptable range. Our evaluation, therefore, underscores PENTESTGPT's capability to yield viable penetration testing results in real-world settings at an efficient cost, thereby highlighting its potential as a practical tool in the cybersecurity domain.

TABLE 5: PENTESTGPT performance over the active HackTheBox Challenges.

Machine	Difficulty	Completion	Completed Users	Cost (USD)
Sau	Easy	✓	4798	15.2
Pilgramage	Easy	✓	5474	12.6
Topology	Easy	✗	4500	8.3
PC	Easy	✓	6061	16.1
MonitorsTwo	Easy	✓	8684	9.2
Authority	Medium	✗	1209	11.5
Sandworm	Medium	✗	2106	10.2
Jupiter	Medium	✗	1494	6.6
Agile	Medium	✓	4395	22.5
OnlyForYou	Medium	✗	2296	19.3
Total	-	6	-	131.5

7. Discussion

We recognize that the penetration testing walkthrough might have been part of the training material for the tested LLMs, potentially biasing the results. To mitigate this, we take two measures. First, we manually verify that the LLM does not have prior knowledge of the target machine. We do this by prompting the LLMs if the tested machine is within their knowledge base. Second, we include penetration testing target machines released after 2021 in our benchmark, which falls outside the training data of OpenAI models. The practicality study on the most recent HackTheBox challenges also demonstrates that PENTESTGPT can solve challenges without prior knowledge of the target.

The rapidly evolving nature of LLMs and inconsistencies in available APIs could invalidate PENTESTGPT's designed prompts. We strive to make prompts general and suitable for various LLMs. However, due to their hacking nature, some LLMs resist generating specific penetration testing content, such as concrete reverse shell scripts. Our prompts include jailbreak techniques [48] to guide the LLM to generate penetration-testing-related information. How to generate

3. **Completed Users** denotes the number of users globally who have completed the target as of the manuscript submission time. Note that HackTheBox boasts over 670,000 active users.

reproducible outcomes is an important direction we are working towards.

We identify hallucination in Large Language Models [46] as a significant challenge where the model's outputs diverge from its training data. This affects the reliability of our automatic penetration testing tool. We are actively exploring various techniques [49] to reduce hallucination and enhance our tool's performance. As an ongoing work, we believe such an attempt will lead to a more robust and effective automatic penetration testing tool.

8. Conclusion

In this work, we explore the capabilities and limitations of Large Language Models (LLMs) in the context of penetration testing. By developing and implementing a novel benchmark, we provide critical insights into how LLMs perform in this intricate domain. We find that LLMs handle fundamental penetration testing tasks and utilize testing tools competently, but they also suffer from context loss and attention issues inherent to their design.

Building on these findings, we introduce PENTESTGPT, a specialized tool that simulates human-like behavior in penetration testing. Drawing inspiration from the structure of real-world penetration testing teams, PENTESTGPT features Reasoning, Generation, and Parsing Modules. This design enables a divide-and-conquer approach to problem-solving. Our thorough evaluation of PENTESTGPT reveals its potential and highlights areas where human expertise continues to outpace current technology. Overall, the contributions of this study serve as a valuable resource and offer a promising direction for continued research and development in the essential field of cybersecurity.

References

[1] A. Applebaum, D. Miller, B. Strom, H. Foster, and C. Thomas, "Analysis of automated adversary emulation techniques," in *Proceedings of the Summer Simulation Multi-Conference*. Society for Computer Simulation International, 2017, p. 16.

[2] B. Arkin, S. Stender, and G. McGraw, "Software penetration testing," *IEEE Security & Privacy*, vol. 3, no. 1, pp. 84–87, 2005.

[3] G. Deng, Z. Zhang, Y. Li, Y. Liu, T. Zhang, Y. Liu, G. Yu, and D. Wang, "Nautilus: Automated restful api vulnerability detection."

[4] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong *et al.*, "A survey of large language models," *arXiv preprint arXiv:2303.18223*, 2023.

[5] Y. Liu, T. Han, S. Ma, J. Zhang, Y. Yang, J. Tian, H. He, A. Li, M. He, Z. Liu *et al.*, "Summary of chatgpt/gpt-4 research and perspective towards the future of large language models," *arXiv preprint arXiv:2304.01852*, 2023.

[6] J. Wei, Y. Tay, R. Bommasani, C. Raffel, B. Zoph, S. Borgeaud, D. Yogatama, M. Bosma, D. Zhou, D. Metzler *et al.*, "Emergent abilities of large language models," *arXiv preprint arXiv:2206.07682*, 2022.

[7] N. Antunes and M. Vieira, "Benchmarking vulnerability detection tools for web services," in *2010 IEEE International Conference on Web Services*. IEEE, 2010, pp. 203–210.

[8] P. Xiong and L. Peyton, "A model-driven penetration test framework for web applications," in *2010 Eighth International Conference on Privacy, Security and Trust*. IEEE, 2010, pp. 173–180.

[9] "Hackthebox: Hacking training for the best." [Online]. Available: http://www.hackthebox.com/

[10] [Online]. Available: https://www.vulnhub.com/

[11] "OWASP Foundation," https://owasp.org/.

[12] "Models - openai api," https://platform.openai.com/docs/models/, (Accessed on 02/02/2023).

[13] "Gpt-4," https://openai.com/research/gpt-4, (Accessed on 06/30/2023).

[14] Google, "Bard," https://bard.google.com/?hl=en.

[15] Rapid7, "Metasploit framework," 2023, accessed: 30-07-2023. [Online]. Available: https://www.metasploit.com/

[16] S. Mauw and M. Oostdijk, "Foundations of attack trees," vol. 3935, 07 2006, pp. 186–198.

[17] [Online]. Available: https://app.hackthebox.com/machines/list/active

[18] A. Authors, "Excalibur: Automated penetration testing," https://anonymous.4open.science/r/EXCALIBUR-Automated-Penetration-Testing/README.md, 2023.

[19] G. Weidman, *Penetration testing: a hands-on introduction to hacking*. No starch press, 2014.

[20] F. Abu-Dabaseh and E. Alshammari, "Automated penetration testing: An overview," in *The 4th International Conference on Natural Language Computing, Copenhagen, Denmark*, 2018, pp. 121–129.

[21] J. Schwartz and H. Kurniawati, "Autonomous penetration testing using reinforcement learning," *arXiv preprint arXiv:1905.05965*, 2019.

[22] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? assessing the security of github copilot's code contributions," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 754–768.

[23] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Examining zero-shot vulnerability repair with large language models," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 2339–2356.

[24] "OWASP Juice-Shop Project," https://owasp.org/www-project-juice-shop/, 2022.

[25] [Online]. Available: https://nmap.org/

[26] MITRE, "Common Weakness Enumeration (CWE)," https://cwe.mitre.org/index.html, 2021.

[27] E. Collins, "Lamda: Our breakthrough conversation technology," May 2021. [Online]. Available: https://blog.google/technology/ai/lamda/

[28] "New chat," https://chat.openai.com/, (Accessed on 02/02/2023).

[29] "The most advanced penetration testing distribution." [Online]. Available: https://www.kali.org/

[30] S. Inc., "Nexus vulnerability scanner." [Online]. Available: https://www.sonatype.com/products/vulnerability-scanner-upload

[31] S. Rahalkar and S. Rahalkar, "Openvas," *Quick Start Guide to Penetration Testing: With NMAP, OpenVAS and Metasploit*, pp. 47–71, 2019.

[32] B. Guimaraes and M. Stampar, "sqlmap: Automatic SQL injection and database takeover tool," https://sqlmap.org/, 2022.

[33] J. Yeo, "Using penetration testing to enhance your company's security," *Computer Fraud & Security*, vol. 2013, no. 4, pp. 17–20, 2013.

[34] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2023.

[35] Y. Bang, S. Cahyawijaya, N. Lee, W. Dai, D. Su, B. Wilie, H. Lovenia, Z. Ji, T. Yu, W. Chung *et al.*, "A multitask, multilingual, multimodal evaluation of chatgpt on reasoning, hallucination, and interactivity," *arXiv preprint arXiv:2302.04023*, 2023.

[36] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," 2023.

[37] H. S. Lallie, K. Debattista, and J. Bal, "A review of attack graph and attack tree visual syntax in cyber security," *Computer Science Review*, vol. 35, p. 100219, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1574013719300772

[38] K. Barbar, "Attributed tree grammars," *Theoretical Computer Science*, vol. 119, no. 1, pp. 3–22, 1993. [Online]. Available: https://www.sciencedirect.com/science/article/pii/030439759390337S

[39] H. Sun, X. Li, Y. Xu, Y. Homma, Q. Cao, M. Wu, J. Jiao, and D. Charles, "Autohint: Automatic prompt optimization with hint generation," 2023.

[40] Sep 2018. [Online]. Available: https://forum.hackthebox.com/t/carrier/963

[41] "Nikto web server scanner." [Online]. Available: https://github.com/sullo/nikto

[42] [Online]. Available: https://openai.com/blog/chatgpt-plugins#code-interpreter

[43] KajanM, "Kajanm/dirbuster: a multi threaded java application designed to brute force directories and files names on web/application servers." [Online]. Available: https://github.com/KajanM/DirBuster

[44] J. Wang, X. Yi, R. Guo, H. Jin, P. Xu, S. Li, X. Wang, X. Guo, C. Li, X. Xu *et al.*, "Milvus: A purpose-built vector data management system," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 2614–2627.

[45] R. Guo, X. Luan, L. Xiang, X. Yan, X. Yi, J. Luo, Q. Cheng, W. Xu, J. Luo, F. Liu *et al.*, "Manu: a cloud native vector database management system," *Proceedings of the VLDB Endowment*, vol. 15, no. 12, pp. 3548–3561, 2022.

[46] M. Zhang, O. Press, W. Merrill, A. Liu, and N. A. Smith, "How language model hallucinations can snowball," *arXiv preprint arXiv:2305.13534*, 2023.

[47] [Online]. Available: https://www.vulnhub.com/entry/hackable-ii,711/

[48] Y. Liu, G. Deng, Z. Xu, Y. Li, Y. Zheng, Y. Zhang, L. Zhao, T. Zhang, and Y. Liu, "Jailbreaking chatgpt via prompt engineering: An empirical study," *arXiv preprint arXiv:2305.13860*, 2023.

[49] P. Manakul, A. Liusie, and M. J. Gales, "Selfcheckgpt: Zero-resource black-box hallucination detection for generative large language models," *arXiv preprint arXiv:2303.08896*, 2023.

TABLE 6: Summarized 26 types of sub-tasks in the proposed penetration testing benchmark.

Task	Description
General Tool Usage	Utilize various security tools for scanning, probing, and analyzing vulnerabilities in the target system.
Port Scanning	Identify the open ports and related information on the target machine.
Web Enumeration	Gather detailed information about the target's web applications, including directory structure, available services, and underlying technologies.
Code Analysis	Review the target's source code to find vulnerabilities that may lead to unauthorized access or other malicious activities.
Shell Construction	Craft and utilize shell codes to manipulate the target system, often enabling control or extraction of data.
Directory Exploitation	Traverse and manipulate directories to discover sensitive files, misconfigurations, or hidden information on the target system.
General Privilege Escalation	Identify and exploit weaknesses in permissions to gain higher-level access to systems or data.
Flag Capture	Locate and retrieve specific data markers ("flags") often used in Capture The Flag (CTF) challenges to prove that a system was successfully penetrated.
Password/Hash Cracking	Utilize tools and techniques to decipher or crack passwords and cryptographic hash values for unauthorized authentication.
Network Exploitation	Identify and exploit vulnerabilities within the network infrastructure to gain unauthorized access or disrupt services.
Command Injection	Inject arbitrary commands to be run on a host machine, often leading to unauthorized system control.
User Access Management	Manipulate user access controls to escalate privileges or gain unauthorized access to resources.
Credential Discovery	Locate and extract authentication credentials such as usernames and passwords within the system.
FTP Exploitation	Exploit vulnerabilities in FTP (File Transfer Protocol) services to gain unauthorized access, file manipulation, or data extraction.
CronJob Analysis	Analyze and manipulate scheduled tasks (cron jobs) to execute unauthorized commands or disrupt normal operations.
SQL	Exploit SQL (Structured Query Language) vulnerabilities like SQL injection to manipulate databases and extract sensitive information.
Windows Domain Exploitation	Target Windows-based networks to exploit domain-level vulnerabilities, often gaining widespread unauthorized access.
Deserialization	Exploit insecure deserialization processes to execute arbitrary code or manipulate object data.
Brute Force	Repeatedly try different authentication credentials to gain unauthorized access to systems or data.
XSS (Cross-Site Scripting)	Inject malicious scripts into web pages viewed by others, allowing for unauthorized access or data theft.
PHP Exploit	Utilize or create exploits targeting PHP applications, leading to unauthorized access or code execution.
Custom Password	Create and utilize custom-crafted passwords based on gathered information, aiding in unauthorized access attempts.
XXE (XML External Entity)	Exploit vulnerabilities in XML parsers to perform unauthorized reading of data, denial of service, or execute remote requests.
SSH Exploit	Target SSH (Secure Shell) services to gain unauthorized access or command execution on remote systems.
CVE Study	Research known vulnerabilities in the Common Vulnerabilities and Exposures (CVE) database to understand and potentially exploit weaknesses in target systems.
Others	Other engagements in additional exploratory testing and other methods to uncover vulnerabilities not identified by standard procedures.