Enhancing Cybersecurity: Design of an Automated Penetration Testing Framework for Common Vulnerabilities and Exposures (CVE)

Nur Rohman Rosyid[1*], Anni K. Fauziyyah[2], Yoan N. Ananda[3]

[1-3]*Internet Engineering Technology, Department of Electrical Engineering and Informatics, Vocational College, Universitas Gadjah Mada, Yogyakarta, Indonesia*

*corr-author: nrohmanr@ugm.ac.id

Abstract— The progression of digital transformation has increased cybersecurity concerns, primarily due to the growing prevalence of system vulnerabilities. Penetration testing (pentesting) is an essential technique for identifying and assessing vulnerabilities; however, conventional methods are labor-intensive and heavily reliant on expert participation. This study proposes the development of an automated penetration testing framework that utilizes Common Vulnerabilities and Exposures (CVE) to enhance efficiency and reduce reliance on manual processes. The framework utilizes software engineering design patterns, namely the Template Method and Abstract Factory, to guarantee modularity, scalability, and maintainability. The implementation and evaluation reveal the system's capacity to reliably perform CVE-based penetration testing activities with consistent performance across multiple iterations. Comparative testing demonstrates that the suggested framework attains superior consistency in execution time and resource utilization compared to monolithic solutions. In conclusion, the established methodology offers a dependable basis for automated CVE-based security evaluations and facilitates continuous adaptation to forthcoming cybersecurity issues.

Keywords: automated penetration testing, cybersecurity, design patterns, CVE

I. INTRODUCTION

CYBERSECURITY has become an increasingly critical domain in today's digital era. The pervasive and easily accessible digital interconnectivity has not only accelerated global business processes but also created an attractive surface for cybercriminals. Reports from institutions such as [1-4] consistently highlight a year-on-year increase in the scale and complexity of cybersecurity threats. According to [1], there has been a notable rise in cybersecurity threats, with 34 new adversary groups, a 110% increase in attacks on cloud infrastructure, and a 75% year-over-year growth in cloud intrusions. Alarmingly, these evolving threats have not been adequately anticipated or mitigated by many security leaders. In [4], emphasizes that cybersecurity risks within business ecosystems are becoming increasingly problematic, with third-party partners being both valuable assets and potential sources of risk. Recent studies highlight predictive cyber defence. The deep learning-based LSTM model described by [5] can predict cyberattacks by type, handling action, and severity. They found that cyber dangers are intensifying and that anticipatory framework for proactive mitigation is needed. This highlights the need for scalable, automated, and data-driven vulnerability assessment and penetration testing. Specifically, 41% of organizations experiencing cybersecurity incidents identified third parties as the source, and 54% lacked visibility into supply chain vulnerabilities. Ironically, 64% of security executives who believed their organization's cyber resilience met minimum operational requirements admitted to being unaware of these supply chain vulnerabilities. Nonetheless, there is consensus on the urgent need for cohesive action across the cybersecurity ecosystem. Implementing such actions, however, demands significant cybersecurity talent and investment [6]. As a solution, KPMG recommends full support for security teams through automation to enable rapid response to incidents [3].

Cybersecurity automation has become increasingly essential, given the scale and complexity of threats targeting critical infrastructures that support vital societal and governmental functions. Manual approaches to cybersecurity mitigation are no longer sufficient in the face of rapidly evolving technologies and attack tactics. Thus, AI-driven and adaptive automated defences have emerged as strategic solutions for achieving early detection, swift response, and layered protection across interconnected networks and systems [7]. Examples of automated security solutions include correcting firewall misconfigurations [8] and reducing virtual network firewall rule complexity [9].

Regular and periodic evaluation of an organization's cybersecurity posture is vital for maintaining high levels of security. A key aspect of this evaluation is vulnerability management, which encompasses various testing methodologies, including penetration testing or ethical hacking. Penetration testing serves as a fundamental component of a cybersecurity strategy and lifecycle, assessing the effectiveness of security tools and policies [10], while also reinforcing organizational defences against emerging threats [11].

Penetration testing simulates cyberattacks against a system, requiring the ethical hacker to possess knowledge of the target, necessary resources, formal authorization, and adherence to ethical standards [12]. In the context of web applications, comprehensive studies have explored fundamental concepts, tool selection strategies [13], and effective penetration testing procedures [14]. The penetration testing process involves a sequence of technically demanding phases including reconnaissance, network scanning, enumeration, vulnerability analysis, exploitation, and result documentation. Due to the time-intensive nature of these phases and their potential operational impact, automation is considered an effective alternative for improving testing efficiency [15].

Automated penetration testing can even be executed on lightweight platforms such as Single-Board Computers (SBCs). An experiment conducted by [16] utilized a Raspberry Pi SBC, leveraging tools such as Nmap, OpenVAS, and Metasploit for the automation workflow. Nmap was used for reconnaissance, OpenVAS for vulnerability analysis, and Metasploit for exploit execution. Due to network and system heterogeneity, selecting the right exploit poses a significant challenge. This has been addressed through machine learning approaches for optimal exploit selection, effectively reducing decision-making time [17,18,19]. Additionally, unpredictable scenarios during the penetration testing phases further complicate the automation process. To address this, techniques such as Bayesian Decision Networks have been proposed to mimic human decision-making during penetration testing [20].

Various automation strategies and specializations have been explored in literature. Infrastructure and server vulnerability testing using tools like Lynis and Ansible have proven effective for automated assessments across network servers [21]. Penetration testing does not end with initial access; subsequent steps must assess internal system vulnerabilities, including the search for sensitive assets. Early-stage post-breach penetration testing automation has been demonstrated using Python scripts for directory traversal and the identification of sensitive files [22]. Automation in information gathering has also been addressed using a range of software tools [23].

While several prior studies, such as [24], have demonstrated the effectiveness of automating penetration testing workflows using Bash shell scripts, these approaches often rely on procedural scripting and tight coupling between tools and logic, which limit extensibility and maintainability. For instance, the automation presented in [24] focuses primarily on simplifying reconnaissance activities and organising scan results through hardcoded Bash workflows. While efficient in repetitive scenarios, these scripting methods tend to be less scalable, more challenging to maintain, and harder to extend when dealing with diverse and evolving CVE structures.

To overcome these limitations, this research proposes a modular, extensible, and maintainable framework for CVE-driven automated penetration testing, designed with software engineering best practices. The framework employs design pattern methodologies—specifically, template method and abstract factory patterns—to enable separation of concerns between CVE modules, scanning logic, and reporting layers.

To guide the purpose, this research formulates the following research questions:

1. *RQ-1*: How can CVE-based penetration testing activities be automated in a modular and scalable manner using design pattern architectural structures?
2. RQ-2: How does the proposed framework compare to a monolithic implementation in terms of execution time consistency, CPU usage behaviour, and extensibility?

By addressing these questions, this study aims to provide not only a functional tool for penetration testing automation but also contributes a scalable, maintainable, and report-integrated architecture to the field of cybersecurity automation.

II. METHOD

This study details the development and implementation of an automated penetration testing framework specifically designed to assess CVE, aimed at enhancing efficiency and accessibility in vulnerability assessment. The software development process utilizes pts software design patterns to create a flexible and adaptable code structure that supports continuous development and enhancement of penetration testing modules.

Software design patterns serve as a solution to recurring design problems in software engineering. Rather than providing ready-to-use code, design patterns describe reusable solutions and strategies for organizing code in a maintainable and comprehensible manner [25,26]. These patterns are fundamental techniques in object-oriented software development, focusing on object creation, structural relationships, and inter-object behavioural coordination. The use of design patterns promotes loose coupling between software components, fostering modularity and minimizing interdependencies.

This research employs two distinct design patterns to construct the proposed automated penetration testing framework. The Template Design Pattern, classified as a behavioural design pattern, is used to define the communication flow and execution sequence among objects. It outlines the skeleton of the penetration testing algorithm and dictates the sequence of phases to be executed. The responsible objects for each testing phase—including information gathering, network scanning, enumeration, vulnerability analysis, exploitation, and reporting—are organized using the template structure. Subclasses are allowed to override specific methods without altering the overall algorithmic structure.

To ensure extensibility and flexibility in module selection, the Abstract Factory Design Pattern is incorporated. This behavioral pattern enables developers to instantiate specific CVE-based penetration testing algorithms by referencing abstract concepts, without modifying or disrupting the existing software architecture. The proposed software model shows a class diagram of an auto-penetration testing application frameworks in Fig. 1. This framework consists of two primary layers: the template layer, which governs the sequencing of penetration testing phases using the Template Design Pattern, and the functional implementation layer, which employs the Abstract Factory Design Pattern to manage the implementation of each penetration testing phase.

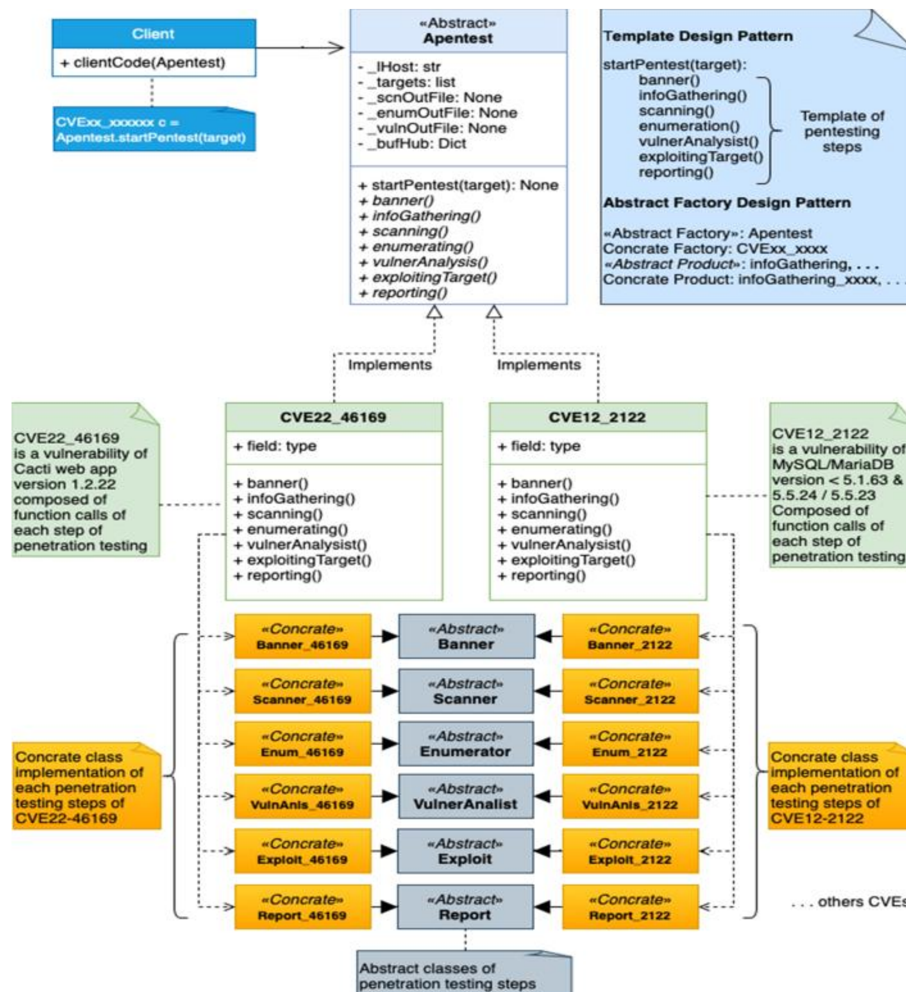

Fig. 1 Class diagram of the automated penetration testing framework

The standard protocol and service penetration testing phases referenced in [15]—namely, information gathering, network scanning, enumeration, vulnerability analysis, target exploitation, and reporting—serve as the basis for the template pattern design. An abstract class named **Apentest** defines a set of abstract methods that outline the penetration testing process. This class contains a single template method, **startPentesting(target)**, and several abstract methods, including **infoGathering()**, **scanning()**, **enumerating()**, **vulnerAnalysis()**, **exploitingTarget()**, and **reporting()**. Concrete classes tailored to specific CVEs implement these methods.

The Abstract Factory Design Pattern enables flexibility in selecting CVE modules. As shown in Fig. 1, the components follow typical Abstract Factory terminology:

1) *Abstract Products:* Interfaces that declare the expected structure for each penetration testing phase and its supporting components. These include interfaces such as Banner (for displaying exploit information), Scanner, Enumerator, VulnerAnalyst, Exploit, and Report.

2) *Concrete Products:* Implementations of the abstract product interfaces. These classes are grouped and named according to their associated CVE (Common Vulnerabilities and Exposures) codes. For instance, **Scanner_46169** implements the Scanner interface specifically for CVE-2022-46169, a vulnerability involving unauthenticated remote command execution in the Cacti open-source network monitoring system.

3) *Abstract Factory:* An interface that declares a set of methods for generating the abstract products. In this framework, the **Apentest** abstract class also acts as the abstract factory.

4) *Concrete Factory:* Implements the abstract factory methods to instantiate specific CVE modules. These classes are named after the corresponding CVEs (e.g., CVE22_46169) and are responsible for producing only the related components. For example, the **banner()** method generates a **Banner_46169** object that implements the **Banner** interface.

This design ensures that all objects created by the concrete factories conform to the interfaces defined in the abstract factory, promoting loose coupling and enabling seamless integration of new CVE modules.

To evaluate the framework's efficiency, memory usage tests were conducted using a comparative baseline, a monolithic version of the automated penetration testing application implemented as a single file. Memory allocation was tracked using Python's built-in Tracemalloc library, which captures runtime memory snapshots at various points during execution. This allows developers to monitor memory usage and detect potential leaks [27,28,29].

III. RESULT AND DISCUSSION

This section presents the results of implementing, evaluating, and analysing the proposed CVE-based automated penetration testing framework. The discussion is divided into three parts: the implementation details of the automated penetration testing framework, a comparative analysis of execution time and CPU resource usage, and an evaluation of memory efficiency. Each subsection provides information about practical feasibility, performance stability, and resource optimization. For comparative analysis, a monolithic (conventional) version of the automated penetration testing application was developed. The primary difference between the monolithic version and the proposed framework lies in their programming approaches. The monolithic version consolidates all functions into a single Python file, whereas the proposed framework adopts a design pattern-based methodology.

A. *Implementation of the Automated Penetration Testing Framework*

The automated penetration testing application was implemented using the Python programming language. Fig. 2 illustrates the logical structure of the CVE-based automated penetration testing implementation. The main project directory is named Pentest, which contains four subdirectories: **AutoPentest**, **CVE**, **Metode**, and **Utils**. These subdirectories contain the core classes of the automated penetration testing framework.

The abstract class **Apentest**, which acts as both an abstract factory and a template, is located in the **AutoPentest** directory and defined in the file **APentest.py**. All abstract product classes responsible for each penetration testing phase—such as **Banner**, **Scanner**, **Enumeration**, **VulnerAnalist**, **Exploit**, and **Report**—are grouped in the **Metode** directory. The CVE directory organizes concrete factory classes and concrete product classes, placing individual subdirectories for each CVE. For example, CVE22-46169 for a vulnerability in Cacti web application and CVE12-2122 for a MySQL/MariaDB vulnerability. Each subdirectory contains a concrete factory class (e.g., **CVE12_2122.py**) and concrete product classes such as **Banner_2122.py**, **Scanner_2122.py**, and others. The **Utils** directory contains utility programs supporting the framework.

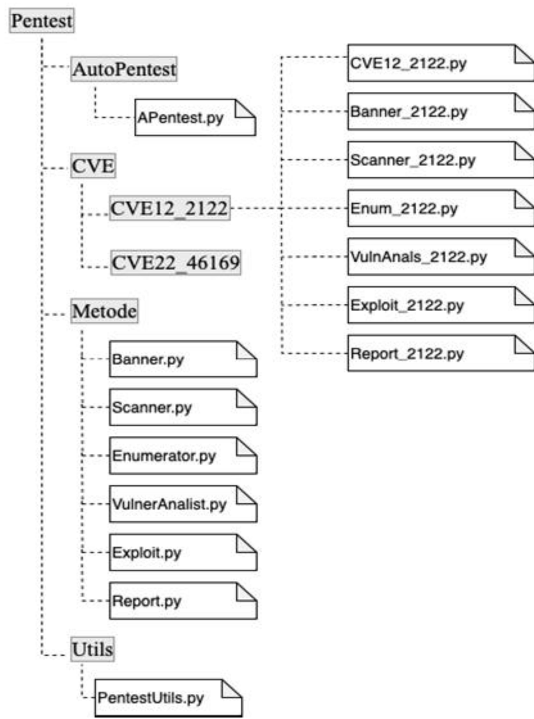

Fig. 2 Folder tree structure of the automated penetration testing framework implementation

Executing the program requires a minimum of two parameters: the target IP address and the type of CVE to be executed. The user calls the **client_code()** function with these parameters, after which the program selects the desired penetration test module using the Abstract Factory design pattern (DP). Once the CVE module is invoked, the penetration testing process manages its sequence using the Template design pattern, as illustrated in the activity diagram of the application shown in Fig. 3.

The invocation of penetration testing phase components follows the defined template order, starting from the banner and continuing through the report. Each CVE module defines only the components necessary for its execution, skipping any undefined components during runtime. Execute the command using the command-line interface (CLI) terminal by typing: **python3 Client.py**. Fig. 4 shows the complete output, and the first line under the banner prompts the user to set the target hosts in MySQL unauthenticated password bypass vulnerability (CVE-2012-2122). As the program executes, progress messages are displayed. The process concludes with a message that confirms the successful generation of the penetration testing report. Fig. 5 provides a sample excerpt from the penetration testing report.

A critical component of this automated penetration testing process is the generation of a structured report document. The reporting procedure leverages CVE descriptions, which serve as a standardized reference for sharing information on known vulnerabilities and exposures. These descriptions are widely used in vulnerability and cybersecurity risk analysis processes [30,31,32]. The report generation can be categorized as semi-automated, as it combines hard-coded content within the source code with dynamically generated sections based on the outcomes of each penetration testing phase.

The hard-coded sections are derived from official CVE descriptions and other authoritative sources relevant to the associated CVE (for example, CVE-2012-2122 — a vulnerability in the MySQL/MariaDB Databases). In contrast, each stage of the penetration testing process elaborates on specific findings using dynamic variables embedded in the program. The *Proof of Concept* (PoC) demonstrating successful exploitation of identified vulnerabilities constitutes one such variable, which is programmatically inserted into the report following a predefined template. Additional supporting information—such as Nmap scanning results, enumerated IP addresses and port numbers, and other pertinent data—is also included to enrich the report content.

Based on the implementation and testing results, it is evident that the use of the Template and Abstract Factory design patterns successfully supports a flexible and modular application architecture. Future expansion of CVE-based penetration testing modules can be achieved by implementing class diagrams, as shown in Fig. 1, and adhering to the directory structure in Fig. 2, by adding new CVE-specific subdirectories under the CVE folder. Furthermore, the structured organization simplifies maintenance and enhances the manageability of CVE-specific modules.

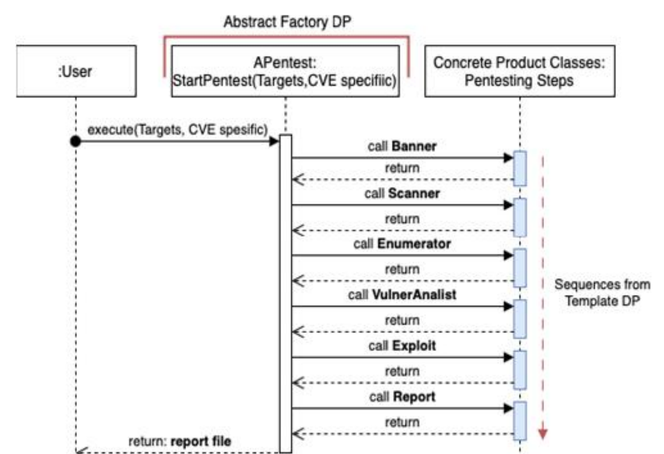

Fig. 3 Activity diagram of automated penetration testing application framewor

```
┌──(rosyid㉿rosyid)-[~/Documents/Pentest]
└─$ python3 Client.py

Set the target IP Address: 10.33.102.225
Auto Pentester says I'm scanning target ['10.33.102.225'] from 10.33.102.151
target IP: ['10.33.102.225']
Auto Pentester says I'm enumerating network target
Auto Pentester says I'm analysing vulnerability network target
Processing Check Vulnerability: 10.33.102.225 3306 MySQL 5.5.23
Target 10.33.102.225 is seem to be vulnerable :)
{'vulnList': [['10.33.102.225', '3306']]}
Auto Pentester says I'm exploiting network target
Please wait ... AutoPentest is attempting a password bypass.

[!!!] VULNERABILITY DETECTED: Login succeeded without a valid password.
[+] Target IP: 10.33.102.225
[+] Password bypass: vFNsgPGC
[+] Number of attempts: 288
[+] MySQL Output:
Database
information_schema
mysql
performance_schema
test

Pentesting report generated successfully
```

Fig. 4 Final results of executing the automated penetration testing application for CVE-2012-2122

B. Execution Time and CPU Resource Utilization Analysis

The execution time was measured by recording timestamps before and after running the application, repeating the process 11 times. Fig. 6 shows a line chart comparing the execution time of the automated penetration testing framework and a monolithic version. Both versions exhibit a generally similar trend across iterations, although some notable differences emerge in terms of execution time stability.

Quantitatively, the average execution time for the monolithic version was 97.48 seconds, while the framework version recorded a slightly faster average of 95.75 seconds. This suggests a marginal performance gain in overall runtime by the framework. However, the standard deviation for the monolithic version was 3.22 seconds. In contrast, the framework showed a higher variation of 5.84 seconds, indicating that while the framework is faster on average, it tends to be less predictable in some early iterations.

In iterations 1 through 4, the framework showed higher execution times than the monolithic version (e.g., iteration 1: framework = 111.89s vs. monolithic = 99.19s), indicating that the framework incurs additional overhead during initial setup. This pattern is typical in modular systems that require pre-execution dependency resolution or dynamic loading of CVE-specific modules.

However, in iteration 6, the monolithic version experienced a significant spike in execution time, reaching 107.31 seconds, which is significantly above its typical range. This anomaly likely reflects a one-off fluctuation in system resources or internal queuing processes in the application. While the framework remained stable at 98.83 seconds in the same iteration, continuing a trend of more consistent performance in the subsequent runs.

Furthermore, the framework exhibited relatively minor variations within a narrow band between 92.11 and 98.83 seconds—especially from iterations 7 through 11—indicating greater consistency in execution time and suggesting that early-stage overhead diminishes after initial executions. In contrast, the monolithic version exhibited greater temporal fluctuation, with time values ranging from 91.56 to 99.37 seconds. This fluctuation likely results from lower internal modular control and less efficient resource handling in certain conditions.

More noticeable fluctuations, particularly in iteration 6 for the monolithic version (103.31s) and iteration 1 for the framework version (111.89s). These outliers elevate the standard deviation in both cases and should be considered when interpreting stability. Nevertheless, when excluding these extremes, the framework achieves a more uniform execution profile, reinforcing its reliability under repeat testing conditions.

To summarize the discussion on execution time, the monolithic version exhibits greater temporal volatility but achieves a slightly superior average execution time. The framework version, on the other hand, exhibits better consistency and scalability potential—a crucial benefit for automated and repeatable penetration testing environments—despite having a minor startup overhead. A significant difference in design philosophy between monolithic and modular automated security systems is highlighted by the trade-off between consistency and raw performance.

4. Key Findings

Upon successful exploitation, the following databases were enumerated: information _schema, mysql, performance_schema, test.

Target Host	10.33.102.225
Open Port	3306/tcp
Service	MySQL
Detected Version	5.5.23
Vulnerability Status	Confirmed Exploitable
Login Attempt Count	409 attempts before successful login

5. Analysis

The target was identified as running MySQL version 5.5.23, which matches the list of vulnerable versions for CVE-2012-2122. Exploitation results show that an attacker can gain root access through random brute-force attempts, without a valid password, and successfully execute SQL commands.

6. Risk Impact

An attacker who successfully exploits this vulnerability can gain administrative access to the database system. This can result in:
- Unauthorized access and modification of sensitive data.
- Lateral movement within the internal network.
- System compromise and data exfiltration.

Given the ease of exploitation and potential damage, this vulnerability is classified as High Risk.

7. Rekomendasi Remediasi and Conclution

To mitigate this vulnerability, we recommend the following actions:
1. Immediately upgrade MySQL to version 5.5.24 or later:
   - MySQL ≥ 5.5.24
   - MariaDB ≥ 5.5.24
2. Restrict remote access to MySQL (port 3306) to trusted IP addresses only.
3. Enable detailed logging and monitor for unusual access patterns.
4. Implementation firewall or fail2ban for blocking brute-force login attempts.
5. Posibly implement socket authentitation (unix_socket) in local system.

The presence of CVE-2012-2122 on the MySQL service of the assessed host poses a significant

Fig. 5 Sample excerpt from the automated penetration testing report

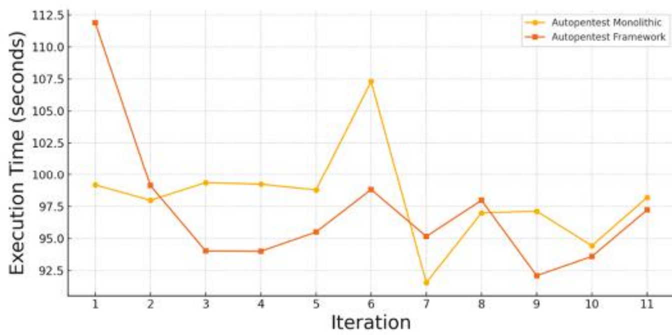

Fig. 6 Comparison of execution time between the proposed automated penetration testing framework and the monolithic version

In a subsequent test, both execution time and CPU usage were measured concurrently, as shown in Fig. 7. The chart confirms the previous findings: the framework version maintained consistent execution time, while the monolithic version showed continued variation. Regarding CPU utilization, the framework version demonstrated usage fluctuations ranging from -11.70% to 25.10%. A CPU usage drop in the first iteration (-11.70%) may suggest low computational demand or interference from concurrent processes, such as the execution of Nmap during network scanning. The monolithic version exhibited similar CPU usage behavior, with fluctuations ranging from -11.90% to 22.40%.

While both versions experienced CPU spikes, the framework version tended to consume slightly more resources (e.g., 25.10% in iteration 3), implying more intensive computation. From the perspective of both execution time and CPU usage, the framework version demonstrated superior stability in timing, albeit at the cost of higher CPU and memory utilization. In contrast, the monolithic version proved more efficient in terms of memory usage but lacked consistency in time execution, which may affect performance under certain conditions.

C. *Memory Usage Efficiency Analysis*

Memory profiling evaluated the efficiency of memory allocation and usage across both versions of the application: the automated penetration testing framework and the monolithic version. Key memory-consuming components—such as **importlib**, **docx/image**, and others—were analysed. Results are visualized using bar charts to facilitate comparison between the two implementations.

Fig. 8 presents the comparative memory usage for major components, including modules such as **importlib._bootstrap**, **docx/image/image.py**, **zipfile**, and others. The **importlib._bootstrap** module had the highest memory usage (1714 KiB), followed by **docx/image/image.py** (1582 KiB). These components significantly contribute to overall memory consumption due to their roles in runtime module loading and image processing for generating PDF report. Other components, such as **cryptography/bindings/openssl/binding.py**, **pathlib**, and **pyfiglet**, consumed less than 200 KiB, reflecting a minimal impact on total memory usage.

The results show that both the automated penetration testing framework, and the monolithic version exhibit similar performance in terms of memory consumption. There are no significant differences in the memory usage of the evaluated components. This equivalence is expected, given that the architectural modification in the framework version is primarily at the logical level (design patterns), with no changes to the underlying program logic that interacts with the operating system. The libraries and memory-consuming operations remain consistent across both versions.

The primary contributors to memory usage are the image processing and module loading operations, while cryptographic and auxiliary modules, such as **pyfiglet,** exhibit low and stable memory usage. This suggests that these supporting components are memory-efficient and do not introduce overhead, reinforcing the framework's efficiency from a memory management perspective.

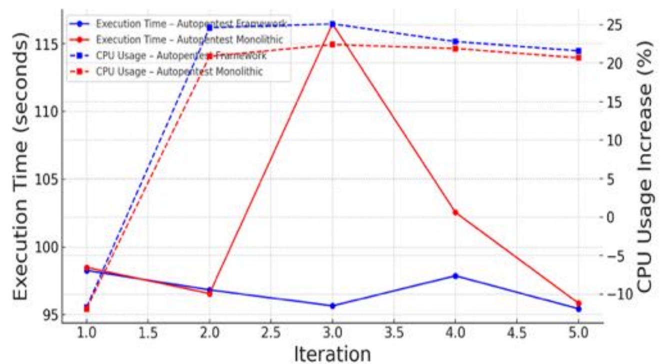

Fig. 7 Comparison of execution time and CPU usage between the proposed automated penetration testing framework and the monolithic version

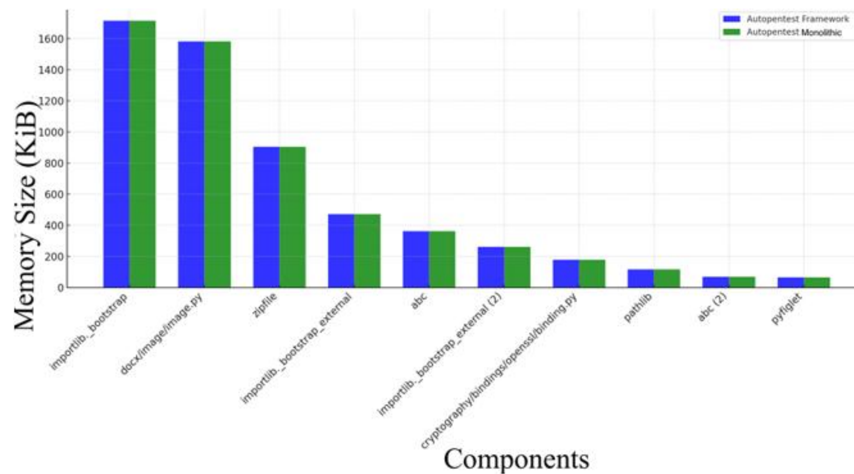

Fig. 8 Comparison of memory usage between the proposed automated penetration testing framework and the monolithic version

IV. CONCLUSION

This study presents a structured and extensible approach to automating penetration testing activities through a CVE-based modular framework. The proposed framework introduces a layered architecture that separates exploitation modules, scanning logic, and reporting components. By employing design pattern methodologies, this framework supports long-term maintainability, facilitates the integration of new exploit modules, and enables automated, template-based reporting. Empirical evaluations demonstrate that, although the framework incurs a slightly higher average execution time in early iterations due to initial module loading and orchestration, it consistently outperforms the monolithic version in terms of stability and consistency. The design also enables developers and cybersecurity professionals to scale the testing process more efficiently, adapt to diverse CVE structures, and present findings in a structured format for stakeholder review and analysis. Overall, this research represents a significant improvement over traditional script-based automation by offering not only automation but also a robust architecture capable of evolving with future needs in penetration testing, particularly following updates to CVEs.

REFERENCES

[1] Kurtz George, "Crowdstrike Global Treat Report," 2024. Accessed: Aug. 12, 2025. [Online]. Available: https://go.crowdstrike.com/rs/281-OBQ-266/images/GlobalThreatReport2024.pdf

[2] Horowitz Maya, "Cyber Security Report 2024," 2024. Accessed: Aug. 13, 2025. [Online]. Available: https://engage.checkpoint.com/2024-forrester-wave-enterprise-firewall-solutions-report/featured/2024-cyber-security-report?fw=a372d

[3] P. Doshi, "Cybersecurity Considerations 2024: Supercharge Security with Automation," 2024. Accessed: Apr. 02, 2024. [Online]. Available: https://assets.kpmg.com/content/dam/kpmg/xx/pdf/2024/01/cyber-considerations-report.pdf

[4] G. Bueermann and M. Rohrs, "Global Cybersecurity Outlook 2024," Jan. 2024. Accessed: Aug. 12, 2025. [Online]. Available: https://www3.weforum.org/docs/WEF_Global_Cybersecurity_Outlook_2024.pdf

[5] J. Siswanto, I. Sembiring, A. Setiawan, and I. Setyawan, "Number of Cyber Attacks Predicted With Deep Learning Based LSTM Model," *JUITA: Jurnal Informatika*, vol. 12, no. 1, pp. 39–48, May 2024, doi: 10.30595/juita.v12i1.20210.

[6] HAYS, "Hays 2024 Global Cyber Security Report," 2024. Accessed: Apr. 02, 2024. [Online]. Available: https://www.hays.co.uk/market-insights/global-cyber-security-report

[7] R. Mazzolin and A. Madni, "An Overview of Cyber Security Considerations and Vulnerabilities in Critical Infrastructure Systems and Potential Automated Mitigation - A Review," *Journal of Engineering Research and Sciences*, vol. 1, no. 4, pp. 9–21, 2022, doi: 10.55708/js0104002.

[8] N. B. Y. Ben Souayeh and A. Bouhoula, "A Fully Automatic Approach for Fixing Firewall Misconfigurations," in *2011 IEEE 11th International Conference on Computer and Information Technology*, 2011, pp. 461–466. doi: 10.1109/CIT.2011.84.

[9] D. Bringhenti, G. Marchetto, R. Sisto, F. Valenza, and J. Yusupov, "Automated Firewall Configuration in Virtual Networks," *IEEE Trans Dependable Secure Comput*, vol. 20, no. 2, pp. 1559–1576, 2023, doi: 10.1109/TDSC.2022.3160293.

[10] H. M. Z. Al Shebli and B. D. Beheshti, "A study on penetration testing process and tools," in *2018 IEEE Long Island Systems, Applications and Technology Conference (LISAT)*, 2018, pp. 1–7. doi: 10.1109/LISAT.2018.8378035.

[11] Y. Khera, D. Kumar, Sujay, and N. Garg, "Analysis and Impact of Vulnerability Assessment and Penetration Testing," in *2019 International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COMITCon)*, 2019, pp. 525–530. doi: 10.1109/COMITCon.2019.8862224.

[12] M. Bishop, "About penetration testing," *IEEE Secur Priv*, vol. 5, no. 6, 2007, doi: 10.1109/MSP.2007.159.

[13] E. A. Altulaihan, A. Alismail, and M. Frikha, "A Survey on Web Application Penetration Testing," 2023. doi: 10.3390/electronics12051229.

[14] J. Creasey, "A guide for running an effective Penetration Testing programme," *Crest*, no. April, 2017, Accessed: Aug. 13, 2025. [Online]. Available: https://www.crest-approved.org/wp-content/uploads/2022/04/CREST-Penetration-Testing-Guide-1.pdf

[15] F. Abu-Dabaseh and E. Alshammari, "Automated Penetration Testing : An Overview," 2018. doi: 10.5121/csit.2018.80610.

[16] A. Akkiraju, D. Gabay, H. B. Yesilyurt, H. Aksu, and S. Uluagac, "Cybergrenade: Automated Exploitation of Local Network Machines via Single Board Computers," in *2017 IEEE 14th International Conference on Mobile Ad Hoc and Sensor Systems (MASS)*, IEEE, Oct. 2017, pp. 580–584. doi: 10.1109/MASS.2017.95.

[17] O. Valea and C. Oprisa, "Towards Pentesting Automation Using the Metasploit Framework," in *Proceedings - 2020 IEEE 16th International Conference on Intelligent Computer Communication and Processing, ICCP 2020*, 2020. doi: 10.1109/ICCP51029.2020.9266234.

[18] A. AlMajali, L. Al-Abed, K. M. A. Yousef, B. J. Mohd, Z. Samamah, and A. A. Shhadeh, "Automated Vulnerability Exploitation Using Deep Reinforcement Learning," *Applied Sciences*, 2024, [Online]. Available: https://api.semanticscholar.org/CorpusID:273445441

[19] A. U. H, B. S. Anavi, B. Goyal, S. P. Kasturi, and P. Agarwal, "Advanced Reinforcement Learning Based Penetration Testing," in *2024 International Conference on Electronics, Computing, Communication and Control Technology (ICECCC)*, 2024, pp. 1–6. doi: 10.1109/ICECCC61767.2024.10593902.

[20] G. M. Roberts and G. L. Peterson, "Automated Computer Network Exploitation with Bayesian Decision Networks," in *Proceedings of the International Florida Artificial Intelligence Research Society Conference, FLAIRS*, 2022. doi: 10.32473/flairs.v35i.130610.

[21] C. P. Varun and R. Agarwal, "Automation of Server Security Assessment," in *4th International Conference on Circuits, Control, Communication and Computing, I4C 2022*, 2022. doi: 10.1109/I4C57141.2022.10057759.

[22] S. Chaudhary, A. O'Brien, and S. Xu, "Automated Post-Breach Penetration Testing through Reinforcement Learning," in *2020 IEEE Conference on Communications and Network Security, CNS 2020*, 2020. doi: 10.1109/CNS48642.2020.9162301.

[23] V. R. Saraswathi, I. S. Ahmed, S. M. Reddy, S. Akshay, V. M. Reddy, and S. M. Reddy, "Automation of Recon Process for Ethical Hackers," in *2022 International Conference for Advancement in Technology, ICONAT 2022*, 2022. doi: 10.1109/ICONAT53423.2022.9726077.

[24] A. K. Singh and G. Kumar, "AUTOMATION IN MANUAL PENETRATION TESTING USING BASH SHELL SCRIPT," *International Research Journal of Modernization in Engineering Technology and Science*, 2023, doi: 10.56726/irjmets40392.

[25] P. Gahlyan and S. Narayan Singh, "Analysis of Catalogue of GoF Software Design Patterns," in *2018 8th International Conference on Cloud Computing, Data Science & Engineering (Confluence)*, 2018, pp. 814–818. doi: 10.1109/CONFLUENCE.2018.8442878.

[26] D. Budgen, "Design Patterns: Magic or Myth?," *IEEE Softw*, vol. 30, no. 2, pp. 87–90, 2013, doi: 10.1109/MS.2013.26.

[27] P. Nawalramka, "Memory profiling in Python with tracemalloc," Apr. 2022. [Online]. Available: https://www.red-gate.com/simple-talk/development/python/memory-profiling-in-python-with-tracemalloc/

[28] C. Bala Priya, "How To Trace Memory Allocation in Python," Sep. 2024. [Online]. Available: https://www.kdnuggets.com/how-to-trace-memory-allocation-in-python

[29] Sunny Solanki, "tracemalloc - How to Profile Memory Usage By Python Code." [Online]. Available: https://coderzcolumn.com/tutorials/python/tracemalloc-how-to-trace-memory-usage-in-python-code

[30] X. Wu, W. Zheng, X. Chen, F. Wang, and D. Mu, "CVE-assisted large-scale security bug report dataset construction method," *Journal of Systems and Software*, vol. 160, p. 110456, 2020, doi: https://doi.org/10.1016/j.jss.2019.110456.

[31] Y. Wei, L. Bo, X. Sun, B. Li, T. Zhang, and C. Tao, "Automated event extraction of CVE descriptions," *Inf Softw Technol*, vol. 158, p. 107178, 2023, doi: https://doi.org/10.1016/j.infsof.2023.107178.

[32] A. N. Kia, F. Murphy, B. Sheehan, and D. Shannon, "A cyber risk prediction model using common vulnerabilities and exposures," *Expert Syst Appl*, vol. 237, p. 121599, 2024, doi: https://doi.org/10.1016/j.eswa.2023.121599.