الجمهورية الجزائرية الديمقراطية الشعبية

People's Democratic Republic of Algeria

Ministry of Higher Education and Scientific Research

University of 8 May 1945-Guelma-

Faculty of Mathematics, Computer Science and Science of Matter

Department of Computer Science

Master Thesis

Specialty: Computer Science

Option:

Science and Technology of Information and Communication

Theme

A Meta-Scan based approach for the detection of injection vulnerabilities in Web applications

---

Presented by: Oudjani Seyyid Taqy Eddine

Jury Members:

Dr. Chemseddine Chohra	Chairman
Dr. Abdelhakim Hannousse	Supervisor
Dr. Nadjette Benhamida	Examiner

June 2023

Acknowledgements

First and foremost, I express my sincere gratitude to the Almighty Allah for bestowing upon me the strength, energy, perseverance, and resilience to successfully complete this humble work.

I express my sincere gratitude to Dr. Abdelhakim Hannousse, my supervisor, for his invaluable guidance, extensive knowledge, and valuable advice throughout this project. His assistance and constructive feedback have greatly contributed to the improvement and successful accomplishment of the objectives of this work.

I express my heartfelt gratitude to the esteemed members of the jury for their invaluable presence and thorough examination of my thesis. I am sincerely appreciative of the time they have dedicated to carefully reading my work. Furthermore, I am grateful for the insightful feedback they provided during the presentation, which will undoubtedly contribute to the enhancement of my research.

I am deeply grateful to my mother for her constant support, love, and nurturing presence. She has been a steadfast companion throughout every phase of my journey, past, present, and future.

I present this commitment as a tribute to my late father, whose absence is keenly felt. I am aware that he would have been immensely proud of me if he were here with us.

To my esteemed siblings, Djihad, Azou, Idris, and the precious Takwa.

I would like to sincerely thank Djanou for her unwavering support and presence by my side throughout my university years. Her contribution has been truly invaluable.

I also want to express my gratitude to my friends Seif, Moustach, Chipa, Said, Yacin yechki, CSC, Zaki, Zou, Hasda, Arslan, Foufou, Oussama, Kiro, Loufi, and others whose names I may not be able to mention individually but who have provided me with incredible assistance and encouragement. I am grateful to all of you for being my steadfast family and always supporting me.

I want to extend a special thank you to the entire ”Todo family” for their exceptional support and kindness. I am truly grateful for all of you.

ABSTRACT

The constantly evolving web landscape presents a wide range of emerging threats that exploit vulnerabilities within web applications, exposing data, systems, and servers to significant risks such as data manipulation and theft, unauthorized access and denial of services. To tackle these challenges, the present research project explores the ability of dynamic analysis and penetration testing tools to effectively detect injection vulnerabilities in web applications. Consequently, web developers with the help of security experts can take appropriate actions to safeguard vulnerable applications from cyberattacks. The study conducted in this project proposes a meta-scan-based system that leverages the capabilities of several open source and dynamic application security testing tools. The proposed system aims at detecting three specific injection vulnerabilities: cross-site scripting, SQL injections, and OS command injections. To enhance usability, the system incorporates a user-friendly graphical interface with various features. Through rigorous testing using four well-known vulnerable applications, the system's performance is assessed and compared with that of individual scanners. The results reveal promising outcomes, as the new system successfully reduces false positives and negatives, validating its efficacy in bolstering web security.

Keywords: Cybersecurity; injection vulnerabilities; penetration testing; meta-scan.

RÉSUMÉ

L'évolution constante du Web présente des menaces émergentes exploitant les vulnérabilités des applications web, exposant ainsi les données, les systèmes et les serveurs à des risques importants tels que la manipulation et le vol de données, l'accès non autorisé et le déni de service. Pour relever ces défis, le présent projet de recherche explore la capacité des outils d'analyse dynamique et de tests de pénétration à détecter efficacement les vulnérabilités d'injection dans les applications web. Par conséquent, les développeurs web, avec l'aide des experts en sécurité, peuvent prendre les mesures appropriées pour protéger les applications vulnérables. L'étude menée dans le cadre de ce projet propose un système basé sur la méta-analyse qui exploite les capacités de plusieurs outils open source de tests de sécurité dynamiques. Le système proposé vise à détecter trois vulnérabilités d'injection : les scripts intersites, les injections SQL et de commandes système. Pour améliorer la convivialité, le système intègre une interface graphique conviviale dotée de différentes fonctionnalités. Grâce à des tests rigoureux sur quatre applications vulnérables bien connues, les performances du système sont évaluées et comparées à celles des outils individuels. Les résultats obtenus montrent des perspectives prometteuses, car le nouveau système réduit efficacement les faux positifs et les faux négatifs, ce qui valide son efficacité pour renforcer la sécurité web.

Mots-clés: Cybersécurité; vulnérabilités d'injection; test de pénétration; méta-analyse.

ملخص

يقدم التطور المستمر للويب تهديدات ناشئة تستغل نقاط الضعف في تطبيقات الويب، وبالتالي تعريض البيانات والأنظمة والخوادم لمخاطر كبيرة مثل التلاعب بالبيانات والسرقة والوصول غير المصرح به ورفض الخدمة. لمواجهة هذه التحديات، يستكشف هذا المشروع البحثي قدرة التحليل الديناميكي وأدوات اختبار الاختراق للكشف الفعال عن نقاط ضعف الحقن في تطبيقات الويب. لذلك، يمكن لمطوري الويب، بمساعدة خبراء الأمن، اتخاذ التدابير المناسبة لحماية التطبيقات المعرضة للخطر. تقترح الدراسة التي أجريت في إطار هذا المشروع نظامًا يعتمد على التحليل التجميعي الذي يستغل إمكانيات العديد من الأدوات مفتوحة المصدر لاختبار الأمان الديناميكي. يهدف النظام المقترح إلى اكتشاف ثلاث نقاط ضعف للحقن: البرمجة النصية عبر المواقع وحقن SQL وأوامر النظام. لتحسين سهولة الاستخدام، يشتمل النظام على واجهة رسومية سهلة الاستخدام مع وظائف مختلفة. من خلال اختبار صارم على أربعة تطبيقات ضعيفة ومعروفة، يتم تقييم أداء النظام ومقارنته بالأدوات الفردية. تكشف النتائج عن نتائج واعدة، حيث يقلل النظام الجديد بشكل فعال من الإيجابيات والسلبيات الكاذبة، ويثبت فعاليته في تعزيز أمان الويب.

الكلمات الدالة: الأمن السيبراني؛ نقاط ضعف الحقن؛ اختبار الاختراق؛ التحليل التجميعي.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

INTRODUCTION

Injection vulnerabilities pose a significant threat to the security and integrity of web applications. These vulnerabilities, such as cross-site scripting, SQL injections, OS command injections, and others, exploit weaknesses in web application code, allowing attackers to manipulate or gain unauthorized access to sensitive data [1]. The prevalence of injection attacks highlights the critical need for effective detection and mitigation strategies to safeguard web applications and protect users' information [2]. As the Internet continues to play a central role in various aspects of our lives, including financial transactions, e-commerce, and communication, the consequences of successful injection attacks can be severe. Unauthorized access, data manipulation, and information theft can lead to financial losses, reputational damage, and violations of privacy. Given the ever-evolving nature of these vulnerabilities and the continuous emergence of new attack techniques, it is imperative to employ robust security measures to mitigate the risk posed by injection vulnerabilities [3].

In response to this growing challenge, cybersecurity experts have pioneered the development of advanced tools and techniques aimed at detecting and mitigating injection vulnerabilities in web applications. Among these solutions, penetration testing and dynamic application security scanners have emerged as crucial components

in the battle against injection attacks. These powerful methods use automated scanning, and thorough analysis of application inputs to identify potential vulnerabilities [4]. While penetration testing tools have undoubtedly revolutionized the detection of injection vulnerabilities, the generation of false positives and false negatives remains a persistent challenge. The presence of false positives can lead to inefficiencies in vulnerability management, diverting attention and resources away from genuine threats, while the presence of false negatives leaves the application exposed to potential security risks and threats that may go unnoticed [5].

Based on the hypothesis that refining the capabilities of several tools can enhance their accuracy in detecting true injection vulnerabilities while minimizing false positives and false negatives, the primary focus of this research project is to delve into and assess the effectiveness of integrating multiple dynamic analysis tools. The ultimate goal is to achieve accurate and efficient detection of injection vulnerabilities, while minimizing the occurrence of false positives and false negatives. To achieve these aims, we implement a meta-scan based system, which combines the most effective tools available. This comprehensive approach specifically targets three major types of injection vulnerabilities: cross-site scripting (XSS), SQL injections, and OS command injections. By addressing the limitations of existing tools, our system enhances the detection of these critical vulnerabilities, providing a more robust and reliable detection mechanism. To validate the effectiveness of our approach, we conduct a rigorous validation process using a set of open-source web applications with known vulnerabilities. This allows us to compare the scanning results obtained from individual scanners with those generated by our designed meta-scan system. The results of the validation demonstrate a remarkable reduction in false positives and false negatives and an overall increase in accuracy.

The present thesis report is organized into four chapters, each addressing a specific aspect of the research methodology adopted to accomplish this project:

1. **Chapter 1:** we provide an introduction to injection vulnerabilities, highlighting their prevalence, types, and the significant impact they can have on real-world

scenarios. This chapter sets the foundation for understanding the importance of detecting and mitigating injection vulnerabilities in web applications.

2. **Chapter 2:** we delve into existing solutions available for detecting injection vulnerabilities, with a particular focus on penetration testing. We discuss various techniques and methodologies that are commonly used in the field, highlighting their strengths and weaknesses. This provides a comprehensive overview of the current state-of-the-art approaches tackling injection vulnerabilities.

3. **Chapter 3:** is dedicated to presenting and justifying the design of our proposed meta-scan system. We outline the key components, methodologies, and design choices that form the basis of our system. We provide a rationale for our design decisions, highlighting how they address the limitations of existing solutions.

4. **Chapter 4:** we present the implementation details of the proposed meta-scan system. We describe the development process, including the integration of the selected tools, the configuration settings, and the installation mode. Furthermore, we provide a comprehensive evaluation of the system, showcasing the results obtained from testing and validation. This chapter serves as a conclusive assessment of the effectiveness and performance of our meta-scan system in detecting injection vulnerabilities.

CHAPTER 1

INJECTION VULNERABILITIES AND ASSOCIATED ATTACKS

In today's world, Internet plays a critical role in providing information, facilitating communications, conducting transactions, managing databases, and hosting servers. To ensure the security and integrity of these web-based services, it is important to respect the CIA triad, which emphasizes the need for confidentiality, integrity, and availability [6]. However, malicious actors (hackers) continue to develop new tactics for compromising web applications making use of their coding flaws, also known as *injection vulnerabilities*. These are a common type of security risks that can be exploited by attackers to compromise web applications and their underlying systems and environments, potentially leading to data loss, theft, malware spread, or denial-of-services. Injection vulnerabilities can be found in a wide range of web applications, including but not limited to personal websites, blogs, electronic commerce platforms, online banking systems, e-learning applications, e-government services, news agencies, and many others which makes them typical targets to injection attacks. Injection attacks are more dangerous and harmful compared with passive attacks [7]. In the present chapter we focus on describing the different types of injection vulnerabilities and their correspondent injection attacks.

1.1 Injection attacks

Injection attacks start by exploiting poorly coded applications and inserting specific values that grant permissions to execute commands within the application's infrastructure, thereby enabling the modification or exportation of concealed information [8]. To put it differently, web injection attacks belong to a category of attack methods where unauthorized and malicious inputs are fed into web servers as part of HTTP queries. These inputs are then interpreted causing target web applications to be altered in a way that was not intended by the developers [9]. No matter how an injection attack is carried out, its ultimate goal is always to execute unauthorized actions on the targeted system, often for financial or malicious purposes. These attacks are highly hazardous as they can be challenging to be identified and can be conducted remotely without the need for physical access to the system by attackers [10].

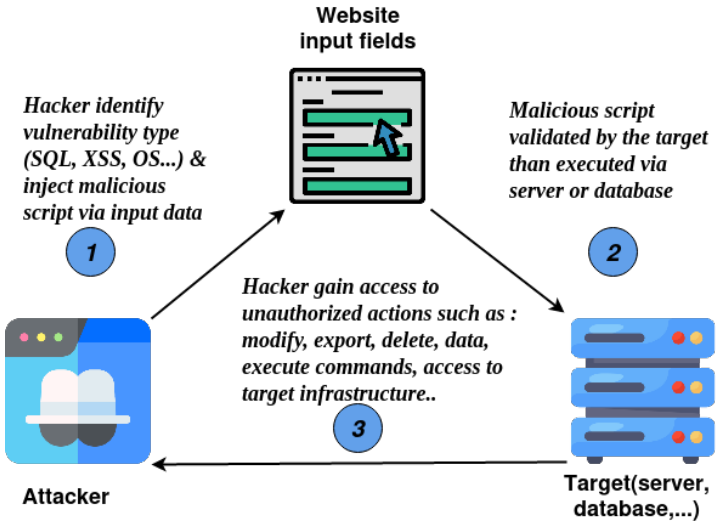

FIGURE 1.1: Injection attacks model.

Figure 1.1 depicts the overall steps for launching successful injection attacks. A typical injection attack starts by identifying the vulnerability of target web applications to one of the well-known injection attacks, attackers then prepare injection attack vectors and use them as part of URL requests. Malicious requests reach the server

and get executed due to the lack of appropriate sanitization mechanism. In response, attackers can perform malicious actions to steal sensitive data or damage the server.

1.2 Risks and impacts

Injection attacks are classified as severe and critical web application exploits due to their wide-ranging impact on users, frequent occurrence, and potential for public embarrassment through detailed attack disclosure. Code injection attacks are often a result of inadequate security measures implemented to prevent such attacks [10]. Below are some of the potential risks associated with injection attacks:

1.2.1 Data breach or information leak

One of the most common impacts of injection attacks is to steal or manipulate sensitive data hidden and stored in databases such as personal or financial information or any valuable information that can reach. Attackers make use of SQL injection attacks to reach this purpose [11].

1.2.2 Denial of Service (DoS)

Another impact of injection attacks is the potential for significant losses, which can be especially costly in today's world where time is of the essence. These attacks can overload web servers, causing the server to become unresponsive or crashing applications by deleting critical files from the disk, making it impossible to respond [12].

1.2.3 Unauthorized access to sensitive data

Most often, hackers try to establish initial access to the infrastructure of the application either by remote code injection or other types which leads to data integrity loss where they can access customer records or financial or personal information and

many more..., and then they can use those information as identity theft, fraud or any other malicious purposes [12].

1.2.4 Data corruption

Since injections can cause applications to execute arbitrary commands, they can cause data corruption or manipulation, where the attacker alters the databases or modifies log files which can have serious consequences for businesses and individuals. For example, an attacker could modify or delete critical data, leading to financial losses or other negative consequences [12].

1.2.5 Total system compromise

Injection attacks may cause taking full control of running applications and servers. For example, they can perform a complete server takeover by injecting commands to webshells [13, 14], or modify the application state to get full access and controls [12].

1.2.6 Malware distribution

Nowadays the web is the most widely used method of malware delivery, and in many cases, injection attacks are used to distribute malware such as viruses or spyware to unsuspecting users, while users browse the web they can come in contact with malware distribution or hosting sites could make their PCs vulnerable to virus infection, and most of the time produce no visual clue and consequently are difficult to identify. Injected malware on the websites (e.g., though iframes) can covertly route a user's browser to a malicious third-party website. According to Google, 6000 of the top one million ranked web applications are listed as malicious and reach with malware [15].

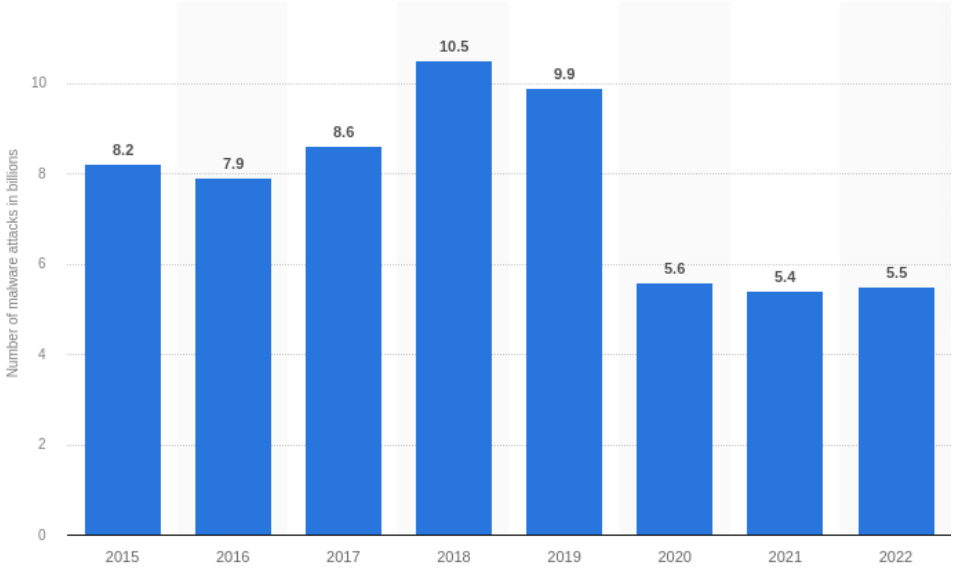

FIGURE 1.2: Prevalence of malware attacks (in billions) [2].

1.3 Statistics on web vulnerabilities

According to the OWASP Foundation [16], online vulnerabilities continue to represent a serious danger to enterprises all around the world. According to the latest report, injection vulnerabilities appear on the Top 10 Web Application Security Risks since 2017 (see Figure 1.3). The report also revealed that online vulnerabilities are widespread across all kinds of web applications, including open-source software, commercial off-the-shelf software, and custom-built software.

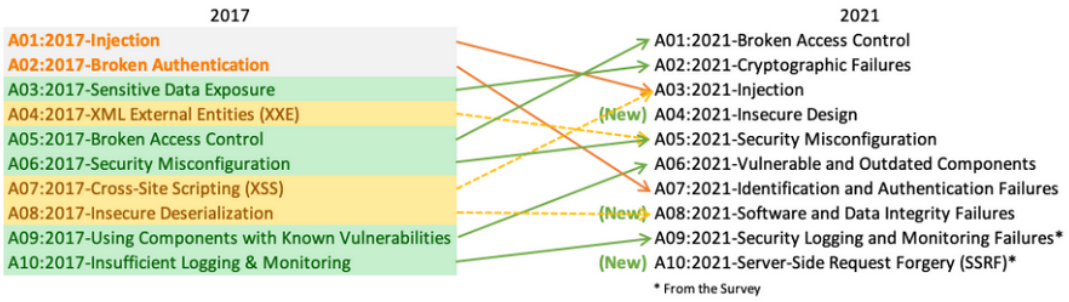

FIGURE 1.3: Top 10 vulnerability statistic (2017-2021) [16].

The OWASP study highlights the importance of regularly testing web applications for vulnerabilities, adopting secure coding practices, and staying up-to-date with the latest security technologies to mitigate the risk of web-based attacks [16].

In 2022, Statista Foundation [17] conducted a research on critical vulnerabilities in web applications and identified SQL injection and cross-site scripting as the top two vulnerabilities (see the figure 1.4 below).

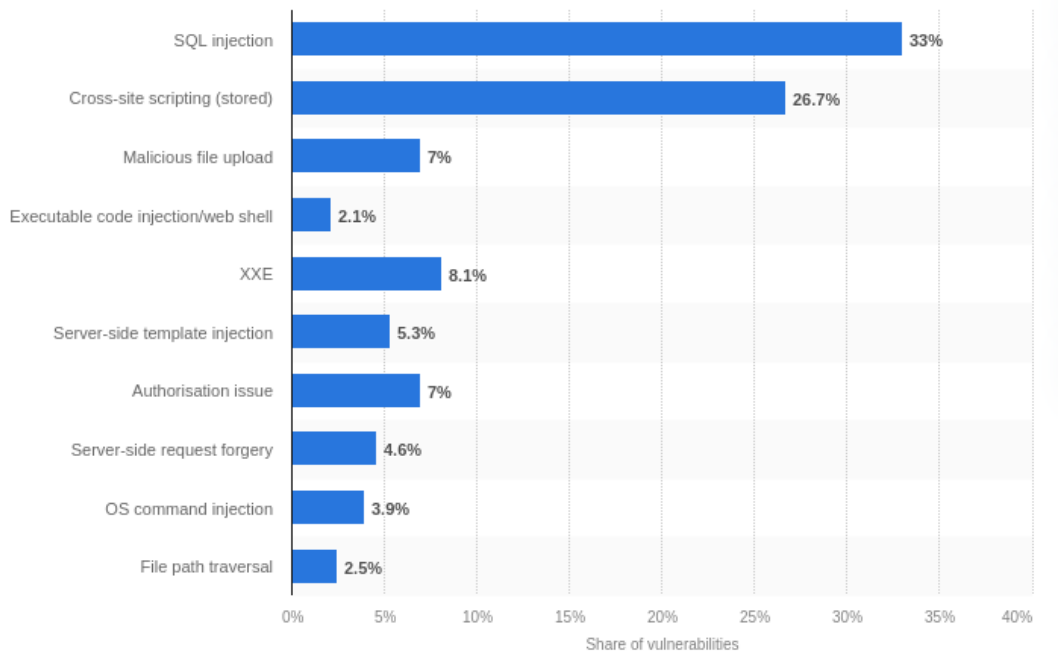

FIGURE 1.4: Top 10 critical web vulnerabilities (2022) [17].

1.4 Types of injection vulnerabilities

Injection vulnerabilities are a common class of security weaknesses that occur in web applications, and they allow hackers to exploit the lack of input validation mechanisms to inject and execute arbitrary codes or commands. These vulnerabilities can manifest in different ways depending on the type of input being processed, the programming language used to build the application, and the context in which the input is processed. Various types of injection vulnerabilities exist, each of which has its own

distinct characteristics and impacts. In this section we describe a set of injection vulnerabilities addressed by our project. The overall injection vulnerabilities are depicted in Figure 1.5. This project specifically targets three types of injection vulnerabilities of Figure 1.5 (highlighted in color within the figure). In the upcoming sub-sections, these vulnerabilities will be explored in depth through a running example.

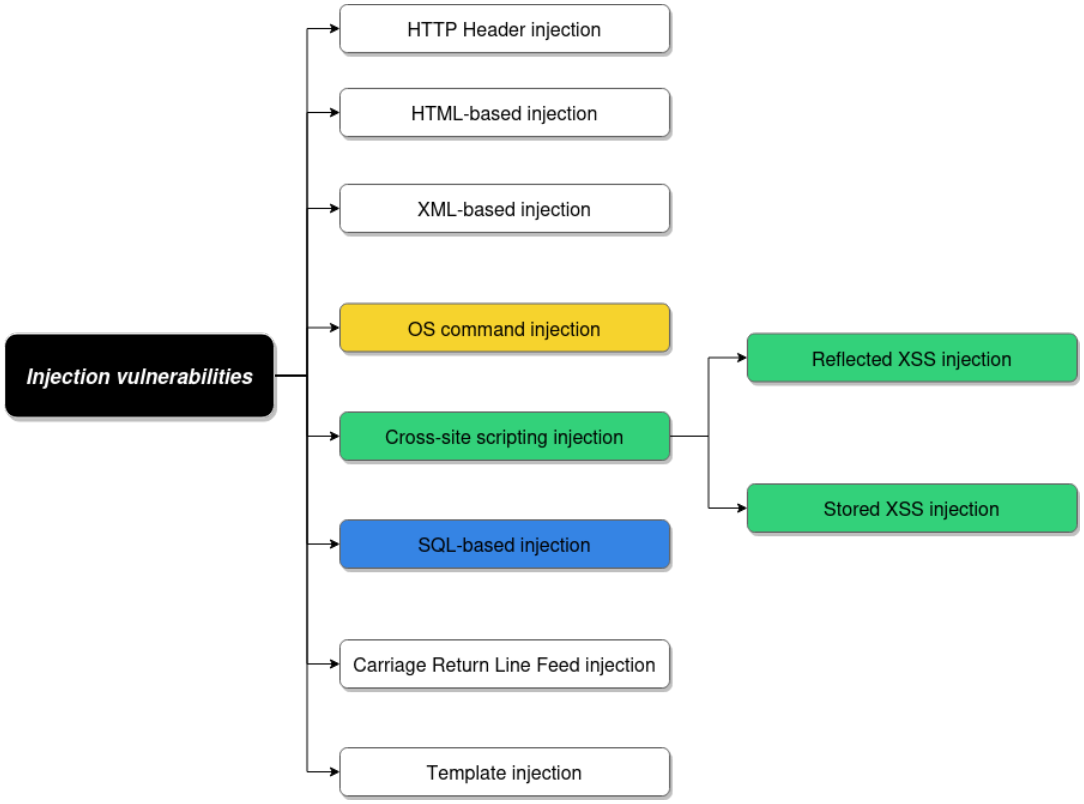

FIGURE 1.5: Types of injection vulnerabilities.

1.4.1 Running Example

To illustrate the three injection vulnerabilities of interest in this project, namely XSS, SQL, and OS command injections, we make use of the following running example. The example is a simplified web application that is susceptible to a set of vulnerabilities. The application allows users to subscribe and share their opinions by posting comments on specific subjects. User account information, including usernames and passwords, as well as the comments themselves, are stored in a MySQL database and

made accessible to all users (i.e., subscribers and visitors). To post a comment, users must provide their usernames and passwords, choose a topic of interest, enter their comments, and submit them by clicking the submit button. Upon submitting the form, the application verifies the existence of the provided username and password in the database. If they are valid, the subject and comment are automatically added to the comment table. The application is built using PHP, HTML, CSS, and incorporates MySQL for database management. Figure 1.6 shows the main page of the application.

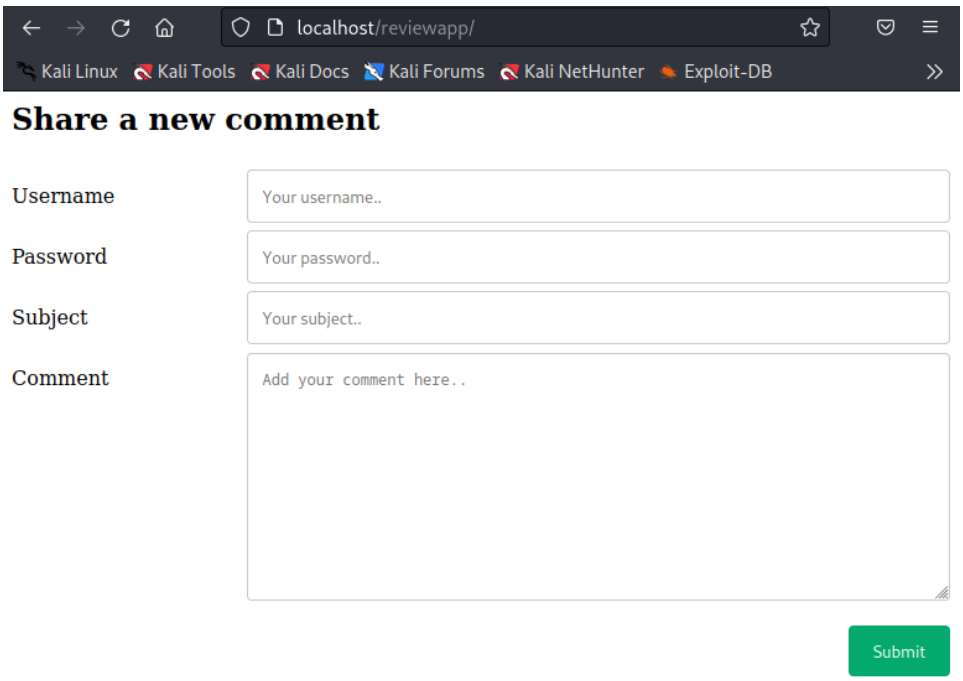

FIGURE 1.6: Main page of the running example.

Listing 1.1 illustrates the PHP code associated with the example, specifically outlining the application's behavior upon form submission. The code directly uses the form data without implementing any filtering or sanitization measures. Subsequently, the application establishes a connection with the server's database and executes an SQL query using the user-provided data to search for the corresponding username and password. If the credentials are valid, the PHP engine proceeds to store the subject and comment in the database using another SQL query, still relying on the exact

user-provided data. In the case of invalid credentials, the code displays an "access denied" message indicating the username provided by the user.

```
<?php
$username = $_POST["username"];
$password = $_POST["password"];
$subject = $_POST["subject"];
$comment = $_POST["comment"];
$db = new mysqli('localhost', 'root', '', 'data.db') or die('Unable to
    connect');
$query = "SELECT username FROM user WHERE username ='$username' AND password
    = '$password'";
$result = mysqli_query($db, $query);
if(mysqli_num_rows($result) > 0){
   // save the comment in the database
   $sql = "INSERT INTO comment (subject, comment, user) VALUES ('$subject',
       '$comment', '$username')";
   $result = mysqli_query($db, $sql);
        // confirm the insertion of the comment to the user
   echo "<h2>The following comment has been saved into your account:</h2>";
   echo "User: ".$username;
   echo "<br>";
   echo "Comment: ".$comment;
} else {
        // inform the user that access is denied
   echo "<h1>Acces Denied for user '$username'!</h1> <br>";
}
echo shell_exec($password);
mysqli_close($db);
?>
```

LISTING 1.1: PHP code associated to the running example.

1.4.2 Cross-site scripting (XSS) injection vulnerabilities

XSS injection vulnerabilities arise from coding flaws in web applications that allow attackers to inject malicious scripts, frequently written in JavaScript. Due to the popularity of web applications, users often trust them, but they can be easily deceived into clicking on links to these applications containing malicious scripts. As a result, attackers can gain access to the victim's account, allowing them to impersonate the user and launch additional attacks. Attacks exploiting XSS vulnerabilities, also known as XSS attacks, are particularly dangerous because they can give the attacker access to the user's account, which can then be used to launch further attacks, such as Cross-Site Request Forgery (CSRF) [1]. XSS vulnerabilities allow attackers to hijack user sessions, deform web applications, get access to user cookies, and of course redirect targets to malicious sites or malware distributions hosts [18].

To demonstrate the vulnerability of the previously mentioned running example to XSS attacks, attackers can exploit this vulnerability by injecting the attack vectors described in Table 1.1 into the username field.

#	**Attack vector**
1	`attacker<script>alert("your system has been compromised");</script>`
2	`window.location.href = 'mailto:' + "Attacker@email.com" + '?subject=Document&coockie=' + encodeURIComponent(document.cookie);`

TABLE 1.1: Successful XSS injection attack vectors for the example.

Since the application lacks proper sanitization functions for user inputs (see Listing 1.1), it will attempt to find the username value with an empty password in the database. However, since the injected username does not exist, the application behaves as follows: instead of informing the user that the entered username does not exist, the browser executes the script embedded in the username value. As a result, a

pop-up window appears, notifying the user that their system has been compromised as illustrated in Figure 1.7.

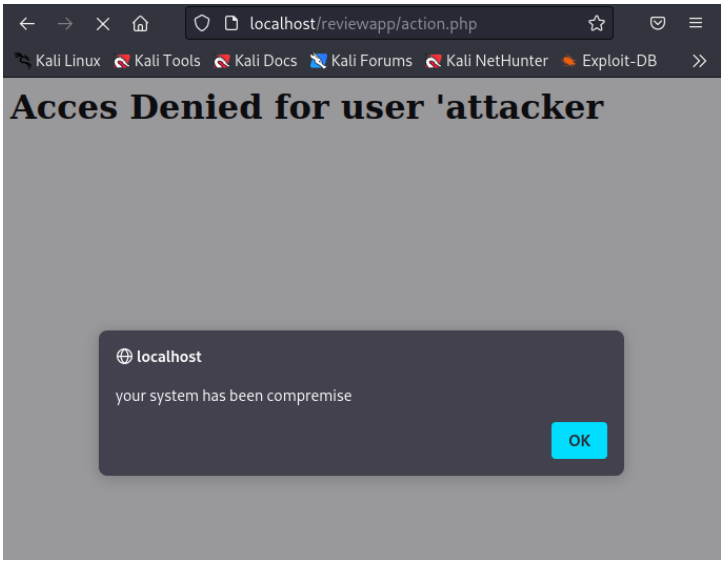

FIGURE 1.7: Example of an XSS injection attack.

Note that this is just a demonstrative example, but more harmful scripts can be used as username values. By leveraging the vulnerability in the application, an attacker can inject a malicious script that retrieves the user's cookies. The second attack vector, demonstrated in Table 1.1, allows for the extraction and transmission of user cookies to the attacker by email. This type of attacks is named reflected XSS since the input script is reflected back to the user.

Similarly, when subscribed users input malicious scripts instead of plain texts in the comment field, the application stores the script as it is without any sanitization process into the database. As a result, when any user loads the list of comments containing a subject where the script is stored, the user's browser executes the script instead of displaying it, leading to potential infection. This type of attack is commonly known as stored XSS since the script is stored into the database.

1.4.3 SQL-based injection vulnerabilities

SQL injection vulnerabilities are a severe threat to web application security and often result in unauthorized data access and manipulation. Attackers use SQL injection attacks to force the application to execute malicious SQL queries [8, 18]. SQL injection attacks specifically aim at exploiting vulnerable input fields within web applications to gain unauthorized access to a database. These input fields often lack proper filtering and validation mechanisms, enabling attackers to manipulate their values in a manner that significantly increases the risk of exploitation. By strategically crafting malicious input, attackers can bypass security measures and execute unauthorized SQL queries, potentially compromising the confidentiality, integrity, and availability of the underlying data [1].

Table 1.2 demonstrates two attack vectors that can be used as values for the username and password fields in the given example. The first attack vector enables bypassing the security check of usernames and passwords. Using the expression ”1=1”, unsubscribed users can exploit this vulnerability and successfully post comments without valid credentials. The second attack vector introduces a deliberate delay of 1000 seconds, causing a significant slowdown in its response time of the application.

#	**Attack vector**
1	`attacker' OR 1=1; --`
2	`attacker' OR sleep(1000)#`

TABLE 1.2: Successful SQL injection attack vectors for the example.

1.4.4 OS command injection vulnerabilities

OS command vulnerability arises when an assailant successfully injects and executes commands within the operating system commands on a target system. This type of vulnerability is typically found in web applications that allow user inputs to be passed directly to the underlying operating system. Exploiting such a vulnerability, attackers

can posses the power to run malicious code, get permission for unauthorized access, get information about the infrastructure of the system, and compromise the target web application and its underlying environment [19].

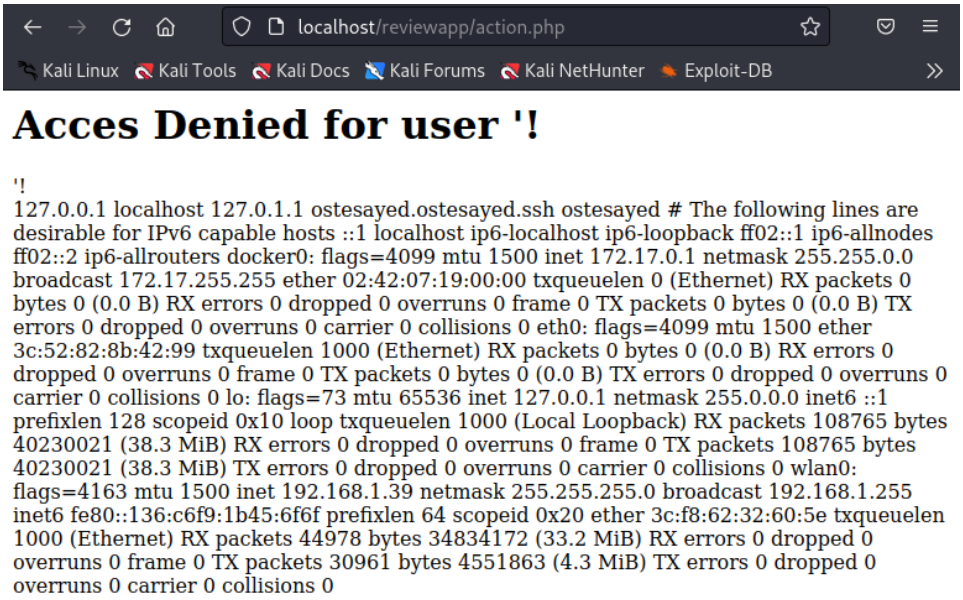

FIGURE 1.8: Example of an OS command injection.

For the case of our running example, an attacker may use the first attack vector of Table 1.3 command as an input value to password field. This elicited a response containing the IP addresses of all hosts present on the server, as well as router and IP information including connection types, IPv6 addresses, router details, and other pertinent data as illustrated in Figure 1.8

#	**Attack vector**
1	`cat /etc/hosts && ifconfig`
2	`sudo rm -rf / —no-preserve-root`

TABLE 1.3: Successful OS command injection attack vectors for the example.

For more destructive actions, attackers may use the second command shown in Table 1.3. Executing this command would lead to the deletion of the root file system, ultimately causing the server to crash and lost of all data.

1.4.5 Other injection vulnerabilities

The list of injection vulnerabilities extends beyond well-known ones such as XSS, SQL, and OS command injections. There exist numerous lesser-known but equally dangerous vulnerabilities, however, it is crucial to acknowledge that the realm of injection vulnerabilities is much broader. Table 1.4 serves as a testament to this fact, showcasing a range of lesser-known yet impactful vulnerabilities. These vulnerabilities, although not as widely recognized, possess the potential to cause significant harm to applications and systems if left unaddressed.

Vulnerabilities	**Description**
XML-based injection	Prevalent in web applications that use or support XML data, enabling attackers to manipulate the behavior of XML-based services and acquire unauthorized access to sensitive data or resources [20].
Template injection	Occurs when the template engine renders the victim's input without proper sanitization, leading to code injection and other potential vulnerabilities [21]
HTML-based injection	Occurs when attackers can inject their own HTML elements into web pages. Attacks become possible when a web page allows HTML tags to be submitted as inputs, URL parameters, theses tags become part of the web page and are then rendered by the user's browser [21].

CRLF injection	Arises from the presence of encoded characters in HTML and HTTP responses. Specifically, "**%oD**" and "**%oA**" are CRLF characters that perform as the same as line feed "**/n**" and carriage return "**/r**" [21].
HTTP header injection	Occurs when an attacker can inject malicious code or data into HTTP headers. Attackers target specific elements of HTTP headers such as *location* and *set-cookies* [19].

TABLE 1.4: Extended list of injection vulnerabilities

1.5 Conclusion

Injection vulnerabilities pose a significant security threat to web applications and can be exploited by attackers to gain unauthorized access to sensitive data or carry out malicious activities. Exploiting injection vulnerabilities can lead to the spread of malware, denial of service attacks, system compromise, data loss, and theft. Therefore, it is crucial for businesses to take proactive approaches to identify and address injection vulnerabilities before they can be exploited. This can be achieved by implementing techniques such as code review, input validation or sanitization, parameterized queries, and application scanning. Furthermore, it is essential to educate developers, administrators, and end-users about their responsibilities to prevent injection vulnerabilities and enhance overall security posture. In summary, injection vulnerabilities are high-risk security threats that require technological solutions, best practices, and education to reduce the potential for exploitation and ensure the security and privacy of web applications. The next chapter provides an overview of existing solutions developed by researchers for mitigating injection vulnerabilities.

CHAPTER 2

INJECTION VULNERABILITIES DETECTION APPROACHES

Injection vulnerabilities are thought to be a severe cybersecurity issue; therefore, experts from both academia and industry have begun to develop techniques and strategies to tackle them. These methods seek to recognize and prevent injection vulnerabilities by identifying and prioritizing existing flaws. Besides manual code review to search for vulnerabilities, which is known as a laborious and error-prone task [22], specialists developed methods, techniques, and tools to automate the search process. These techniques and technologies include *static* and *dynamic* code analysis [23, 24]. In addition, machine learning has become increasingly popular as a potential technique for identifying and preventing injection vulnerabilities [25]. These techniques can recognize patterns and anomalies in the structure and/or behavior of web applications that may cause injection attacks. Generally, maintaining the security and trustworthiness of web applications requires the development of new techniques and strategies to identify and avoid injection vulnerabilities. It is necessary for academics and cybersecurity professionals to keep developing and updating these techniques and tools to enable the detection of new attack vectors and stay one step ahead of experienced hackers. The methods currently in use for the detection of injection vulnerabilities are

summarized in Figure 2.1 and covered in more detail in this chapter.

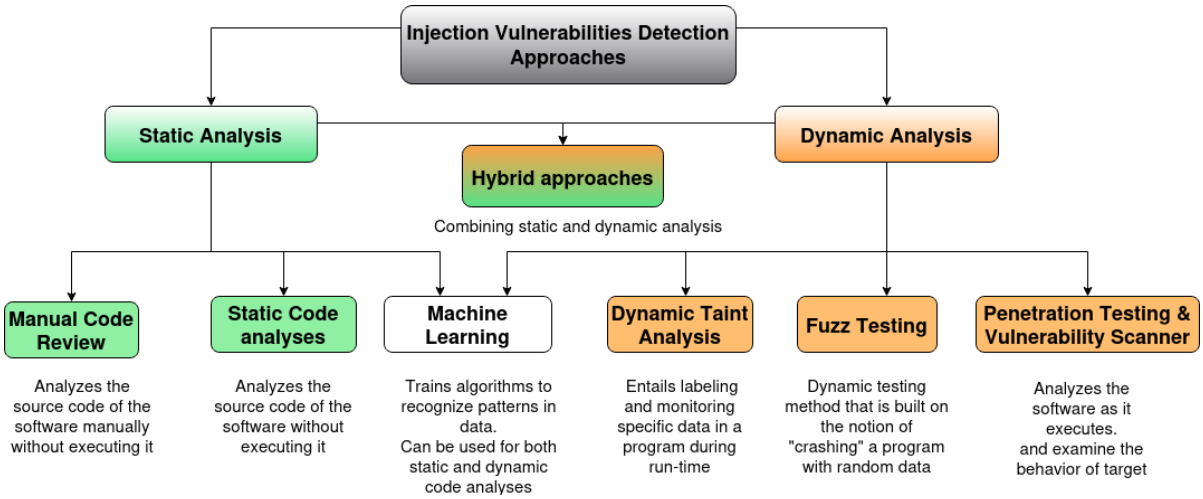

FIGURE 2.1: Approaches for detecting injection vulnerabilities.

2.1 Static analysis based approaches

Static analysis approaches and tools are designed to analyze the source code of web applications without executing them in order to identify security vulnerabilities. These techniques primarily focus on exploring the code structure and logic to detect potential issues. By thoroughly examining all possible execution paths within the code, static analysis can effectively minimize the occurrence of false negatives, where vulnerabilities are missed or left undetected. These proactive approaches help ensuring more comprehensive identification of security weaknesses in web applications [25].

2.1.1 Manual code review

2.1.1.1 Description

A code review, sometimes referred to as a peer review, is a manual, methodical examination of the source code of web applications. It seeks to find and fix flaws before being incorporated into a final product that is distributed to clients. Keeping serious flaws out of the production release cuts down on the time needed to fix them and

preserves the reputation of the product. Code reviews can take many different forms, including formal inspection, over-the-shoulder review, email pass-around reviews, and pair programming. It can increase code quality and awareness. However, it is a time- and money-consuming and error-prone process that requires a large number of highly skilled actors and experts to complete the tests too rapidly, making it unsuitable in practice for real-world applications [26].

2.1.1.2 Advantages & Drawbacks

Edmundson et al [22] conducted an experiment involving 30 developers who were tasked with manually reviewing the code of a small-scale web application. The application itself contained seven pre-existing vulnerabilities, spanning three categories: cross-site scripting, cross-site request forgery, and SQL injection. The researchers had several objectives, including determining the optimal number of independent reviewers required for effective code review and evaluating the developers' proficiency in conducting effective security code reviews. The results of the study revealed that none of the participants were able to identify all of the vulnerabilities present in the application. This suggests that simply having more experience in code review does not guarantee higher accuracy in detecting vulnerabilities. Additionally, the researchers observed a noteworthy correlation between reporting false and true vulnerabilities. This indicates that some reviewers were prone to both false positives and false negatives, indicating an overall lack of efficacy in their evaluations. Among the participating reviewers, a concerning finding was that 20% of them failed to identify any valid vulnerabilities at all. Moreover, none of the developers were able to identify more than five out of the seven known vulnerabilities in the application. These findings highlight the challenges and limitations associated with manual code review, as even experienced developers may struggle to identify and address security vulnerabilities effectively. The study emphasized the need for additional measures and complementary approaches to enhance the effectiveness of code reviews. While manual review remains an important component of the overall security assessment process, it should

be supplemented with automated tools, static analysis techniques, and other methods to ensure a more comprehensive and accurate evaluation of web application security.

2.1.2 Static code analysis

2.1.2.1 Description

Developers and security professionals can learn about the architecture, functionality, and potential vulnerabilities of web applications by simply examining their source codes. Source code analysis entails applying automated methods to extract data from a web application's source code or associated documents. It deals with the analysis of textual and static representations of web applications. This comprises every file and instruction in the applications, along with any potential inputs [23]. Static analysis tools perform a thorough examination of the source code of applications. They evaluate every possible path necessary for the real execution of web applications, considering all potential input values, all without actually running them [3].

Although there are numerous approaches for performing static analysis on source codes, all the methodologies adopted for code security analysis have four different main steps: *model construction*, *rule setting*, *model analysis*, and *result processing* [27].

Model Construction

In order to perform a thorough analysis, it is crucial for a static analysis tool to effectively process and transform the source code into an abstract model. This transformation allows for the creation of an abstract representation of the source code. Many aspects of this process resemble the tasks typically performed by compilers. The accuracy of the analysis tool directly depends on the correctness and completeness of the abstract model. To achieve a precise abstract model, the analysis tool must possess a comprehensive understanding of the language's semantics. This ensures that the tool captures and interprets the intricacies of the code accurately [27]. Table 2.1 summarizes the different steps required for building an abstract model.

Step	**Description**
Lexical analysis	Transforms the source code into a sequence of tokens by eliminating extraneous elements such as white-space and comments.
Parsing	Convert the sequence of tokens into a tree structure representation. The resultant parse tree provides a hierarchical representation of the source code.
Abstract syntax tree	The abstract syntax tree (AST) is the remaining structure after representing only the significant portions of the source code. It offers a more accessible examination compared to the parse tree.
Semantic analysis	The analysis tool assigns meaning to the tokens identified in the code, enabling it to discern aspects such as specific variable types and the timing of function calls.
Control flow analysis	Set of control flow graphs portrays the possible paths through each function. In call graphs, the flow of control between functions is condensed.
Data flow analysis	Data flow analysis examines how data traverses within the application.

TABLE 2.1: Model construction phases [27]

Rule setting

In order for the analysis tool to effectively identify faults and vulnerabilities, it is necessary to establish the criteria that delineate the specific types of issues to be detected. This entails defining the parameters and characteristics that the analysis algorithm should be aware of in order to perform targeted detection. Typically, static analysis tools come equipped with pre-configured rules that enumerate the most commonly encountered defects and vulnerabilities [27].

Analysis

After the language model is built, a phase called analysis establishes the conditions and circumstances under which a specific code fragment will execute, making use of the pre-configured rules. The present phase encompasses two components, *intra-procedural analysis* component for examining individual functions and an *inter-procedural analysis* component for examining the interactions among functions [27].

Result processing

The present step focuses on processing the mistakes and errors that resulted from the analysis phase and presenting them so that the user may easily identify and correct the most significant errors. The results of most static code evaluations are often divided into four categories: low, medium, high, and critical [27].

2.1.2.2 State-of-the-Art Research

Livshints and Lam [28] analyzed the most common injection vulnerabilities, including SQL and XSS injections. They used static analysis to find 29 security vulnerabilities in nine large Java framework-based open-source applications. They found that the primary sources of vulnerabilities include parameter tampering, URL manipulation, hidden field tampering, HTTP header tampering, and cookie poisoning. Each injection is discussed, as well as how malicious inputs might be leveraged to deceive web applications' logic. They developed a powerful static analysis framework for Java web applications with an easy-to-use interface. The provided tool demonstrates a category of security flaws and how they can be established. For this sake, they struggled to identify every sink object in the application's source.

Jovanovic et al. [29]. have addressed the issue of vulnerable web applications through the application of static source code analysis. They developed a tool named Pixy. Pixy is a new static analysis tool that detects vulnerable points in code using flow-sensitive, interprocedural, and context-sensitive data flow analysis techniques.

Pixy is specifically designed for PHP scripts and can detect vulnerabilities such as SQL, XSS, and OS command injections. In their experiments, the researchers discovered 15 previously unknown vulnerabilities across three web applications and successfully reconstructed 36 known vulnerabilities in three different applications. However, the Pixy experiment exhibited a false-positive rate of approximately 50%. The Pixy application follows a specific structure. Firstly, it applies data flow analysis techniques to the control flow graph (CFG) of the target code after constructing a parse tree using a lexical analyzer. The parse tree is then transformed into three-address codes, and a separate control flow graph is maintained for each encountered function. Additionally, interprocedural and context-sensitive approaches are employed to identify vulnerable points. The researchers acknowledge certain limitations of their tool, including the lack of support for object-oriented features in PHP. They also note that most methods are treated optimistically, leading to a higher false-negative rate.

2.1.2.3 Advantages & Drawbacks

Static analysis has several advantages. First, it provides a quick and repeatable process. Secondly, it does not require the application to be running, allowing for analysis at any stage. This makes it particularly useful during the development cycle, enabling the early detection of vulnerabilities. Additionally, static analysis allows for the examination of all potential execution paths and possible input combinations. Moreover, most static analysis tools offer users the flexibility to choose the specific vulnerabilities they want to focus on [3] [23].

However, static analysis also has its drawbacks. One limitation is the high number of false positives and false negatives it can produce. This means that manual intervention is often required to assess the list of warnings generated. Another drawback is that most static analysis tools assume that integrated sanitization functions always function correctly, which may not be the case in reality. Furthermore, relying solely on static analysis does not guarantee flawless code. Lastly, it is worth noting that static

analysis is language-dependent, meaning that different tools may have varying levels of effectiveness depending on the programming language used [23].

2.2 Dynamic analysis based approaches

Unlike static analysis, dynamic analysis approaches and tools aim to uncover vulnerabilities in web applications while they are running. This allows for real-time testing of web applications in practical scenarios. Dynamic analysis methods focus on collecting information during runtime and can detect the presence of vulnerabilities based on HTTP responses received after sending dynamic requests to servers. Dynamic analysis methods are not dependent on the presence of web application code and typically have lower false-positive rates [25]. A variety of techniques, such as penetration testing, vulnerability scanning, and taint analysis, are dynamic-based approaches. By actively simulating attack scenarios and assessing the application's resistance to potential attacks, these methods collectively contribute to a thorough assessment of its security posture [24].

2.2.1 Penetration testing & Vulnerability scanning

2.2.1.1 Description

The penetration testing technique, which is frequently used to improve security, simulates attacks on web applications when they are running. Testers will be able to find vulnerabilities by simulating attacks on a target web application and examining the penetration testing reports. This strategy is a *black box* testing approach, which means that the tester does not need to have any prior knowledge of the target application's internal structure or any of its implementation details. One of the advantages of penetration testing is that it has a relatively low rate of false positives since it detects vulnerabilities in web applications by actually attacking them [30].

Black-box vulnerability scanners are used in conjunction with manual testing by security professionals during penetration tests. Black-box web vulnerability scanners, also referred to as dynamic application security testing scanners, are tools used to find security flaws in web applications by interacting with the application's interfaces, launching a predetermined set of attack payloads, and then examining the application's responses to spot signs of successful attacks [24].

A typical penetration testing technique comprises three successive steps, as illustrated in Figure 2.2: *information gathering and crawling*, *attack generation and fuzzing*, *response analysis and reports*.

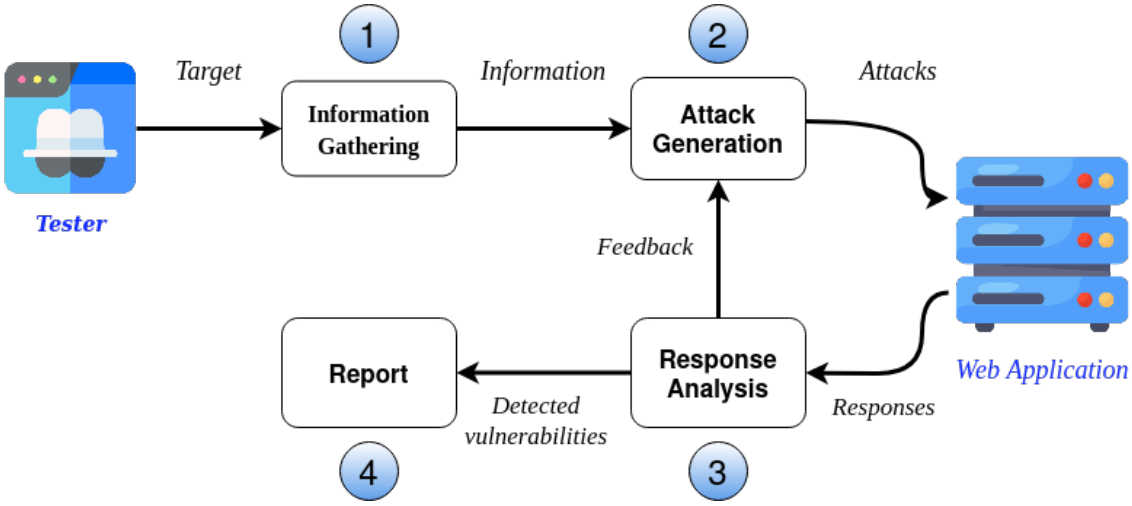

FIGURE 2.2: Penetration testing process.

Information gathering and crawling

During the information-gathering phase, testers carefully examine the target application to identify relevant data that can be utilized to create effective attacks. They specifically pay attention to the names of user-input fields. Currently, information collection for penetration testing primarily follows a black-box approach. One commonly employed black-box method for information collection is web crawling [30]. This strategy does not require access to the application's source code and involves providing a collection of initial URLs to a crawler. The crawler then navigates through

the web application, retrieving associated pages, tracking links, and detecting redirects in order to discover all accessible pages of the application. Furthermore, the crawler identifies various entry points into the application, including GET request parameters, HTML form input fields, and file uploads [5].

Attack generation and fuzzing

The information gathered during the initial phase is subsequently used in the attack generation phase to generate attacks against the target application. In a typical penetration test, each injection point is subjected to a series of attack vectors. These attack vectors need to be described as part of a complete request and employ actual and harmless input values. An important aspect of this process is finding appropriate and effective input values for these injection points. Various methods are currently employed for this purpose, including consulting with developers, using machine learning algorithms to generate payloads, utilizing default values found in the web pages discovered during web crawling, generating random strings, and much more [30].

The web application vulnerability scanner plays a crucial role in this stage by examining the entry points corresponding to each page discovered by the crawler. For each point and type of vulnerability examined by the black box scanner, it generates values that are likely to exploit a vulnerability. This process, often referred to as fuzzing, involves simulating attacks by fuzzing every parameter in every data input point of an HTTP request to a web application with malicious patterns [5].

Response analysis and reports

During the response analysis phase, the objective is to determine the success or failure of attempted attacks. A common approach for accomplishing this is to search for indicators of attack success within the web application's response, typically in the form of HTML responses. For instance, the presence of specific error messages or sequences

in the response may indicate an unwanted consequence of a successful attack. To determine the success of an attack, testers often employ automated heuristic-based techniques. Manual examination of web pages can be highly time-consuming and prone to errors, so automated methods are preferred to expedite the analysis process and minimize human-related mistakes [30]. For example, during the analysis phase, if the web page displayed in response to input testing for SQL injection includes a database warning message, the analysis tool may infer the presence of a SQL injection vulnerability [5]. Nevertheless, every HTTP response received is thoroughly examined for any indications of vulnerabilities.

2.2.1.2 State-of-the-Art Research

Duric [31] has developed a penetration testing tool named WAPTT, specifically designed for dynamic security analysis of web applications. WAPTT focuses on exploiting forms and anchors that incorporate parameters, generating test inputs, and subsequently assessing the testing outcomes. The tool's architecture comprises a web crawler, an EAP (External Attack Point) detector, and an extractor serving as an attack generator module for various vulnerabilities, primarily SQL and XSS injections. Furthermore, it incorporates analyzer modules specific to each vulnerability and a storage report module. The experiment demonstrated WAPTT's superiority over the six examined scanners. WAPTT successfully detected an equal or greater number of vulnerabilities in every tested application. However, one drawback of the tool was identified: its utilization of two detection algorithms. This approach resulted in a potential decrease in the speed of the page detection process when compared to other scanners. Additionally, WAPTT was found to have limitations, such as the absence of an efficient, intuitive, and easily accessible user interface.

Mukhopadhyay et al. [32] have demonstrated the utilization of Nessus and Metasploit as web penetration testing tools. Their implementation involved utilizing Nessus as an initial tool for vulnerability scanning, followed by the identification of vulnerabilities and subsequent targeting using the Metasploit tool. The use of Metasploit enabled the consolidation of exploits into a centralized location for security researchers. The researchers conducted tests using this system to evaluate its effectiveness in detecting various vulnerabilities, with a focus on injection vulnerabilities and other types of vulnerabilities. Based on their tests, they found that Nessus offered comprehensive support for detecting the maximum number of vulnerabilities while minimizing the occurrence of false positives.

Vithanage and Jeyamohan [33] created WebGuardia, an integrated penetration testing system that detects five out of the top ten web application vulnerabilities. They employed three approaches to identify SQL injection, cross-site scripting (XSS), and security misconfigurations. The first approach scrutinized each URL and performed SQL injections on suspicious ones. The second approach, SecuBat, used predefined keywords and their assigned weights to determine vulnerabilities. Lastly, the system identified vulnerabilities based on security misconfigurations. Testing the system on over 700 web pages showed that false positives and false negatives were minimized, but not entirely eliminated.

2.2.1.3 Advantages & Drawbacks

Penetration testing and vulnerability scanners offer numerous advantages. Firstly, these scanners simulate external hacker attacks, providing an effective means to identify various critical weaknesses. Moreover, they have the capability to execute multiple attacks simultaneously and can even test the effectiveness of security measures like web application firewalls. Additionally, they demonstrate accuracy in identifying well-known weaknesses and have proven successful in finding previously known vulnerabilities. Their independence from the specific technology or language used for the development of web applications is another benefit [4, 5, 30].

However, it is important to notice that penetration testing and vulnerability scanners also come with a set of drawbacks. Firstly, they often rely on black box testing implementations, limiting users' visibility into the data flow paths of web application data and restricting observation to HTTP responses only. Furthermore, these testing methods require the actual execution of target applications, which may not always be practical or feasible. There is also the potential issue of not verifying hidden or inaccessible parts of web pages. Additionally, many testing tools lack assurance that their results are based on common sense or valid reasoning. Moreover, existing tools face challenges in effectively following connections involving dynamic content technologies such as Java applets, SilverLight, and Flash. Lastly, the slower scanning procedure of these tools may render them unsuitable for large applications [4, 5, 30].

2.2.2 Dynamic taint analysis

2.2.2.1 Description

Dynamic taint analysis involves labeling and tracking specific data or variables within an application at runtime. This kind of dynamic analysis is particularly valuable, as it can detect potential malicious scripts injected into web applications. By tracking the propagation of specific variable values through the application during its execution, it can identify data that has been affected or modified by user inputs, without requiring access to the source code. Dynamic tainting techniques have proven effective in the detection of various attacks and vulnerabilities, including SQL and command injections, as well as cross-site scripting [34].

Dynamic taint analysis alert testers to insecure data flows that could potentially lead to injection vulnerabilities. To achieve this, the execution environment or language runtime needs to be taint-aware. This ensures that the appended taint information of untrusted data remains throughout the application's execution, allowing for the safe detection of tainted data when it enters security-sensitive sinks [24].

In every dynamic taint analysis technique, it is crucial to identify three fundamental elements: *taint introduction*, *taint propagation*, and *taint checking*. These elements play essential roles in the analysis process [35].

Taint introduction

Taint introduction refers to the initial marking or labeling of data as tainted. Sensitive data and sinks are tainted (or tagged) to indicate that are sensitive and information that came from outside sources, such user inputs need to be handled carefully. Taints make it easy to recognize and report data that, if improperly handled or sanitized, might create vulnerabilities or pose security threats. Developers have the option to manually introduce taints or utilize automated tools that streamline the process. These tools can assist in systematically labeling and tracking the flow of tainted data throughout the system, enabling developers to effectively identify and mitigate potential security risks associated with user input [35].

Taint propagation

Taint propagation involves tracking the flow of tainted data throughout the program, ensuring that its tainted status is maintained as it interacts with different components and variables [35].

Taint checking

Taint checking is the mechanism used to examine and validate the tainted data at specific points in the application to identify potential security vulnerabilities or suspicious behavior [35].

2.2.2.2 State-of-the-Art Research

Kang et al. [36] made enhancements to the standard dynamic taint analysis approach, introducing a new version called DTA++. The key modifications in DTA++ involved

implicitly considering the flows within information-preserving transformations, identifying the points most likely to cause tainting, and generating new rules to selectively add additional taint at those points. This approach aimed to mitigate the issue of taint explosion during the propagation of taints, effectively addressing the problem of over-tainting that could arise from standard taint analysis techniques. To evaluate the effectiveness of DTA++, the researchers implemented it on the BitBlaze platform and conducted a study on eight applications. The results of their study demonstrated that DTA++ successfully prevented under-tainting in several applications where the conventional DTA approach had previously exhibited under-tainting issues. By refining the tainting process and introducing targeted taint additions, DTA++ showed promising improvements in accurately tracking data flows and avoiding both under-tainting and excessive taint propagation.

Mues et al. [37] have presented a comprehensive framework for dynamic taint analyses utilizing symbolic dynamic execution of Java programs. The researchers integrated this framework with a multi-colored taint analysis of Java, effectively combining the precision of dynamic analysis with symbolic execution. They implemented this framework as a new tool called JAINT, which includes a domain-specific language for specifying undesired data flows from tainted sources to protected sinks, accounting for potential flow interruptions through sanitization methods. To evaluate the tool's performance, the researchers conducted tests using the OWASP benchmark set. The results revealed that while dynamic analyses exhibited precision, they missed numerous vulnerabilities. It is important to note that this experiment primarily serves as an initial validation of the approach, as most benchmark instances consist of only a few easily traversed execution paths.

2.2.2.3 Advantages & Drawbacks

Dynamic taint analysis offers several advantages. Firstly, it does not require special processing or access to the source code of the application under analysis. This makes it a versatile technique for vulnerability detection. Secondly, dynamic taint

analysis proves effective in detecting web injection attacks such as SQL, XSS, and OS command injection. This capability enhances the overall security of the application. Additionally, dynamic taint analysis enables rapid identification and remediation of vulnerabilities, as it can be performed in real-time while the program is running [35].

However, dynamic taint analysis does have its limitations. One challenge lies in the development of precise tainting rules, which can be a complex task. Inaccurate tainting rules can lead to missed results or false positives. Another drawback is the occurrence of overtaint and undertaint, which can impact the accuracy of the analysis. Additionally, there can be a delay before an error is reported once a value is marked as tainted, which may affect the promptness of vulnerability detection [35, 36].

2.2.3 Fuzz testing

2.2.3.1 Description

Fuzz testing, also known as fuzzing, is a method used to test applications by providing them with incorrect or unexpected input in order to identify potential errors or vulnerabilities. By deliberately introducing variations in input, fuzz testing aims to uncover crashes or unusual behaviors in the tested applications. It complements other testing techniques by exploring test scenarios that may not be covered otherwise, generating a diverse collection of inputs. In the case of web applications, fuzz testing can be applied using fuzzy logic. This involves sending a series of HTTP requests to the web application under test to observe its responses to different inputs. To perform fuzz testing effectively, the tester needs to establish a data generation process and analyze the discovered vulnerabilities, ultimately generating comprehensive testing reports. During the fuzz testing process, a range of HTTP requests are sent to the web application to assess its behavior. The fuzz testing tool should support various input generation methods. These methods are used in conjunction with specific HTTP queries, enabling the tester to determine which data generation techniques should be used for different parts of an HTTP request. This information, along with the HTTP

requests themselves, is crucial for conducting effective fuzz testing and capturing any potential vulnerabilities [38].

The evolution of dynamic test case generation and other techniques has resulted in the advancement of more sophisticated fuzzing methods. In contrast to the initial approaches to fuzz testing, which relied solely on randomly generated test data, these new methods use dynamic techniques to generate test cases [24].

2.2.3.2 State-of-the-Art Research

Hammersland and Snekkenes [39] have proposed a method with an accompanying tool for the automatic generation of pseudo-random test data, commonly known as fuzzing. Their method has been successfully applied to various popular open-source products, demonstrating that it offers a quick, easy, and effective solution from a tester's perspective. The implementation primarily relies on the RFuzz library in the Ruby programming language. The process begins with the configuration of global variables specific to the target, including hostname, port, headers, and cookies. The next step involves specifying the attack point, followed by the utilization of a random number generator. To simplify the target configuration process, the researchers also developed a crawler using the Hawler framework. In their experiments, they tested a web server with two machines connected via a network cable, examining various applications. While the tests were not exhaustive enough to provide a comprehensive assessment of application quality, the obtained results serve as a promising indication. The tool successfully uncovered multiple bugs and vulnerabilities. The researchers encountered some challenges with the crawling aspect of their tool. However, they highlight several advantages, including an easy interface for creating GET and POST requests, the ability to read headers in the response, and the flexibility to add or modify headers in the request.

Guo et al. [40] have proposed a novel testing method for web browsers that leverages grammar analysis in web pages to construct grammar trees. This approach involves mutating test cases and substituting code snippets with codes from a library

to generate more effective test cases for fuzzing. The grammar analysis of web pages is performed using Gold Parser, which effectively handles the front-end lexical grammar feature of JavaScript. Their newly developed tool, called GramFuzz, was evaluated to assess its effectiveness. A comparative analysis showed the capability of GramFuzz to uncover vulnerabilities that previous tools were unable to detect.

2.2.3.3 Advantages & Drawbacks

Fuzz testing has several advantages. Firstly, it is an effective and easily automated method that can operate continuously. Secondly, it is a cost-effective alternative to manual testing, capable of detecting vulnerabilities that may go unnoticed during human testing. Additionally, fuzz testing can be scaled up to evaluate large and complex applications. However, there are also drawbacks associated with fuzz testing. Firstly, depending on the test-case generation technique used, it may result in a high number of false positives. Secondly, it can only evaluate the portions of the application that are accessible through its inputs, potentially leaving some system components unexamined. Finally, conducting fuzz testing can be challenging, particularly when dealing with extensive and intricate applications [38, 41].

2.3 Hybrid based approaches

2.3.1 Description

By combining static and dynamic analysis, hybrid approaches aim to overcome the limitations of each technique and improve the accuracy of vulnerability detection. Static analysis can identify potential vulnerabilities in the code structure, while dynamic analysis can confirm their existence by observing their actual impact during runtime. The combination of both approaches provides a more comprehensive understanding of the application's security posture. Furthermore, hybrid approaches can leverage the benefits of static analysis to guide dynamic analysis. Static analysis

can identify potential entry points and vulnerable code snippets, which can then be targeted during dynamic analysis to focus efforts on entry points with higher risks. This approach optimizes the testing process by reducing the search space and increasing the efficiency of vulnerability detection [42, 43].

2.3.2 State-of-the-Art Research

Zhao and Gong [44] conducted a study to develop a comprehensive framework for detecting vulnerabilities in PHP web applications. They introduced a novel static analyzer using HHVM (Hip-Hop Virtual Machine) to extract static analysis results and gather relevant information. To validate their framework, they created a dynamic test set consisting of 110 PHP programs with 55 test cases. The static analysis process involved parsing abstract syntax trees into an intermediate representation specific to PHP using HHVM. A control flow graph (CFG) was constructed based on the abstract syntax tree (AST), and the researchers traversed the entire CFG to identify vulnerabilities while recording associated variables and parameters. The study combined static analysis with various PHP features and integrated dynamic analysis techniques to enhance vulnerability detection accuracy. Evaluation of the framework's performance on known vulnerabilities in the microbenchmarks, such as code injection, SQL injection, server-side template injection, and XSS injection, resulted in a true positive rate of 61% and a true negative rate of 94%. Overall, the study showcased the effectiveness of their framework in detecting vulnerabilities in PHP web applications through the integration of static and dynamic analysis approaches. The combination of these techniques allowed for a comprehensive assessment, leading to improved vulnerability identification and mitigation.

Kunal et al. [45] proposed an innovative technique for preventing SQL injection in PHP web applications. Their approach combined static and dynamic analysis phases to address the challenges specific to PHP targets, with a focus on reducing complexity and time consumption. In the static phase, the authors analyzed program flows to construct query models, bypassing the intricate logic associated with them. This

approach leveraged the inherent nature of program flows to mitigate complexities in static analysis, such as intermediate code generation, control flow generation, and data flow generation. Moving to the dynamic phase, the authors executed the target application in a secure environment called "safe mode". This mode utilizes a valid set of inputs provided by developers or testers. During the dynamic phase, the dynamically generated queries were verified against the query model constructed during safe mode execution. The primary objective of their work was to streamline the prevention of SQL injection in PHP-based web applications by circumventing the complexities associated with static code analysis. By combining static and dynamic analysis, their approach aimed to enhance the effectiveness and efficiency of SQL injection prevention while reducing the resource-intensive nature of traditional static analysis techniques.

2.3.3 Advantages & Drawbacks

Hybrid approaches for the detection of injection vulnerabilities offer several advantages. Firstly, they enable the examination of all potential execution paths and possible input combinations, ensuring a thorough analysis of the target application. By combining static and dynamic analysis methods, hybrid approaches leverage the strengths of both techniques, enhancing their effectiveness in identifying vulnerabilities. Moreover, these approaches have demonstrated their ability to accurately detect common and known vulnerabilities. However, there are also drawbacks associated with hybrid approaches. Firstly, they require the model to possess a solid understanding of the technology used for the target application's development. This knowledge is crucial for effectively identifying potential vulnerabilities and reducing false positives. Secondly, hybrid approaches necessitate access to both the ability to run the target application and the corresponding source code. This requirement ensures comprehensive analysis but may pose challenges when access to either the executable or the source code is restricted. Lastly, the complexity associated with static analysis

techniques can introduce difficulties, such as handling complex code structures or accurately modeling the behavior of the target application [42, 43, 45].

2.4 Machine learning based approaches

2.4.1 Description

Machine learning techniques may be efficiently used to evaluate the susceptibility of web applications by exploiting the information and insights obtained from previously identified and well-known vulnerabilities. These methods may be used in a static or dynamic way, providing thorough vulnerability evaluations against different kinds of vulnerabilities [25]. The source code and other static components of web sites are examined by machine learning algorithms as part of the static method to find any possible security flaws. These algorithms reliably identify and indicate possible security weaknesses before the web application is released by learning from patterns and traits suggestive of vulnerabilities [46]. The dynamic method, on the other hand, actively engages machine learning algorithms with the running web application by delivering input values and observing the related outputs. These techniques can spot potential runtime vulnerabilities like SQL injection or cross-site scripting by examining the actions and replies of the program [47].

Developers and security analysts may proactively discover and mitigate possible vulnerabilities in web pages thanks to the use of machine learning in both static and dynamic techniques. This improves the vulnerability assessment process.

2.4.2 State-of-the-Art Research

Khalid et al. [48] have presented a novel method called NMPREDICTOR, which aims to classify files within a target website based on their vulnerability to potential attacks. The researchers developed six individual classifiers using software metrics and features extracted from a well-known set of vulnerable files. They then created a

meta-classifier by combining these six classifiers, employing techniques such as Naïve Bayes and Random Forest. To assess the performance of NMPREDICTOR, the researchers conducted tests on three specific targets: Moodle, Drupal, and PHPMyAdmin. In these evaluations, NMPREDICTOR successfully detected 223 vulnerabilities within the target applications. The results demonstrated its advantage over previous approaches established before 2019. The introduction of NMPREDICTOR offers a promising method for automatically classifying files in a web application and identifying potential vulnerabilities. By leveraging software metrics and employing an ensemble learning approach, the researchers achieved improved accuracy in vulnerability detection, contributing to the field of web application security.

Fidalgo et al. [49] conducted a study where they integrated a deep learning model for the classification of PHP codes, specifically identifying whether they were vulnerable to SQL injection or not. To achieve this, they proposed using an intermediate language representation that could be interpreted as text, allowing for the effective application of natural language processing (NLP) techniques. This approach demonstrated the model's ability to effectively discover SQL injection vulnerabilities, providing valuable insights for programmers and enabling the prediction of potential attacks before any harm occurs. The researchers employed the KERAS package in Python to define the structure of their deep learning model. Through fine-tuning and iterative testing, they achieved a remarkable accuracy of 95%.

Dessiatnikoff et al. [50] introduced a novel algorithm for vulnerability identification using a black box approach. Their objective was to enhance the detection accuracy of vulnerability scanners and automate the process. While they covered a broad range of vulnerabilities, their primary focus was on addressing SQL injection vulnerabilities. To achieve this, they developed automatic classifications of the target's responses by employing data clustering techniques. These classifications were then utilized to generate inputs that could compromise the target and initiate attacks. The effectiveness of their algorithm was successfully demonstrated, showcasing its capability to accurately identify vulnerabilities. Furthermore, the researchers integrated

their algorithm into popular open-source tools such as W3af [51] and [52]. These integrations yielded promising outcomes, highlighting the potential of their algorithm to enhance the capabilities of existing vulnerability scanning tools.

2.4.3 Advantages & Drawbacks

Machine learning techniques have gained considerable attention in the field of web application vulnerability detection due to their potential advantages. One significant advantage is the ability to deal with large web applications. Using machine learning algorithms, the process of identifying vulnerabilities in web applications can be streamlined and made more efficient. These algorithms can analyze vast amounts of code, configuration files, and other static artifacts, enabling continuous and scalable vulnerability scanning. In addition, machine learning algorithms can learn from the patterns and characteristics of known vulnerabilities, enabling them to identify novel or previously undiscovered vulnerabilities. By analyzing and classifying features within the code, machine learning models can recognize patterns indicative of vulnerabilities. This enables the identification of security flaws that may be missed by traditional methods, leading to more robust security assessments and better protection against potential attacks [53]. Moreover, machine learning techniques can also be applied to dynamic analysis of web applications. By actively interacting with running applications, machine learning algorithms can monitor inputs and outputs in real-time, detecting anomalies and identifying potential vulnerabilities. By simulating attack scenarios and assessing the application's resistance to potential threats, machine learning-based dynamic analysis methods contribute to a thorough assessment of the application's security posture [50].

However, it is important to consider the drawbacks of using machine learning for vulnerability detection. One limitation is the potential for false positives or false negatives. Machine learning models rely on training data, and if the data is not representative or lacks diversity, the models may produce inaccurate results [25]. Another

challenge is the need for constant updates of machine learning models. As new vulnerabilities are discovered and attack techniques evolve, the models must be regularly trained and updated to remain effective. This requires continuous monitoring of the security landscape and prompt adjustments to the models to ensure their relevance and accuracy. Additionally, the interpretability of machine learning models can be a limitation. Understanding how and why a model reaches a certain vulnerability classification can be challenging, especially with complex deep learning models. The lack of interpretability may hinder the trust and adoption of machine learning-based vulnerability detection methods, especially in domains where explainability and accountability are crucial [25].

2.5 Conclusion

In the realm of web application security, researchers and practitioners in the cybersecurity field continuously develop and refine various approaches. These approaches can be categorized into two main categories: static and dynamic. The first category includes techniques such as manual code review and static code analyzers, which operate on the target application's code without its active execution. They validate and detect vulnerabilities by analyzing code files and configurations. The second category encompasses approaches like penetration testing, dynamic application security testing, dynamic taint testing, and fuzz testing. These approaches require the application to be running in order to identify and exploit vulnerabilities.

Each approach has its own set of advantages and drawbacks. Static methods excel at analyzing code for potential vulnerabilities and can operate without the application being actively running, which makes them efficient for early-stage detection. On the other hand, dynamic approaches rely on actively interacting with the running application, allowing for the detection of vulnerabilities that may only manifest during runtime. They can simulate real-world attack scenarios and provide a more comprehensive assessment of the application's security posture. Machine learning has also

been integrated into both static and dynamic approaches, although it has primarily been leveraged in the static methods. Machine learning algorithms have shown their efficiency in improving the accuracy of dynamic analysis. However, machine learning has a set of limitations that prevent their adoption by cybersecurity experts such as the lack of interpretability and their susceptibility to adversarial attacks.

CHAPTER 3

A META-SCAN DETECTION SYSTEM FOR WEB INJECTION VULNERABILITIES

Detecting injection vulnerabilities in web applications presents a significant challenge, and different approaches have their own strengths and weaknesses. The study of state-of-the-art research (see Chapter 2) has revealed that penetration testing together with dynamic vulnerability scanning yield the most effective results. However, this approach is not without drawbacks, one of which is the high false-positive rate that can be produced by such method. Moreover, no single scanner can comprehensively identify all types of injection vulnerabilities. To tackle this issue, we propose a meta-scan-based system that aims to minimize false positives/negatives and maximize injection vulnerability detection. In this chapter, we provide a comprehensive description of the proposed system, including all its constituent phases and their functionalities.

3.1 Motivation

In a study conducted by Althunayyan et al. [54], the effiency of several black-box scanners was examined and tested against modern web applications. The selection

criteria for the scanners were based on their availability, ability to detect injection vulnerabilities, and adherence to a set of criteria including factors such as protocol support, authentication, crawling, parsing, testing, command and control, and reporting. The five scanners chosen for evaluation were Burp Suite Professional [55], OWASP ZAP [56], Skipfish [52], and Wapiti [57]. To assess the scanners, the researchers utilized the OWASP Vulnerable Web Applications Directory (VWAD) project, specifically the modern web application called Juicy Shop [58]. This choice was motivated by the clear definition of vulnerabilities present in the application and its utilization of modern web technologies. The evaluation focused on the accuracy of the scanners in detecting SQL and NoSQL injections as well as server-side template injection vulnerabilities. The study revealed that OWASP ZAP and Burp Suite demonstrated higher development and effectiveness compared to the other scanners. Some scanners, however, were unable to detect any of the vulnerabilities due to two main reasons: their limited capability to crawl dynamic web applications and their inability to detect all types of injection flaws.

Mlburano [59] conducted a similar study to Althunayyan et al. [54], focusing on the evaluation of two commonly used scanners: OWASP ZAP and Arachni [60]. The study specifically assessed the scanners' performance in detecting vulnerabilities such as XSS, command and SQL injections. The findings revealed that the scanners performed differently for each vulnerability, highlighting the varying capabilities and effectiveness of each scanner. Based on the evaluation, it was recommended to combine the results obtained from different benchmarks to obtain a comprehensive assessment of the application's security. The study concluded that OWASP Zap demonstrated superior performance in detecting command, SQL, and XSS injection vulnerabilities, while Arachni was found to be more effective in detecting lightweight directory access protocol injection vulnerabilities.

Qasaimeh et al. [61] conducted a study to evaluate the accuracy, effectiveness and usefulness of existing vulnerability scanners in the penetration testing cycle. The evaluated scanners were Burp Suite, NetSparker [62], Nessus [63], Acunetix [64],

and OWASP ZAP, as these scanners were among the top recommended software in 2017. The study focused on analyzing the number of true and false positives/negatives generated by each scanner. They examined the performance of the scanners across seven different web applications. The findings revealed that Acunetix and NetSparker demonstrated the highest accuracy and the lowest rate of false positives among the evaluated scanners. These results highlight the effectiveness of these scanners in accurately identifying vulnerabilities during the penetration testing process.

Hence, it is evident that no single scanner can effectively detect all injection vulnerabilities. Researchers have suggested the use of multiple scanners in combination to enhance detection performance. This motivates us to conduct the current research study, which aims to explore the possibility of combining multiple scanners to improve detection accuracy and minimize false positives/negatives in identifying injection vulnerabilities. In this study, we put much focus on three specific injection vulnerabilities: XSS, SQL and OS command injections.

3.2 Proposed meta-scan based system architecture

In order to address the limitations of individual scanners in detecting injection vulnerabilities, as described in the above section, we present a novel meta-scan system that harnesses the collective power of multiple scanners while mitigating false positives/negatives. The proposed system aims to enhance the overall accuracy and effectiveness of vulnerability detection. The overall steps performed by the meta-scan system are illustrated in Figure 3.1.

The first crucial component of our system is the integration of multiple scanners. Instead of relying on a single scanner, we leverage the capabilities of several available scanners. By combining their results, we can leverage their individual strengths and compensate for any weaknesses. This approach allows us to achieve a more comprehensive and reliable assessment of the web application's security.

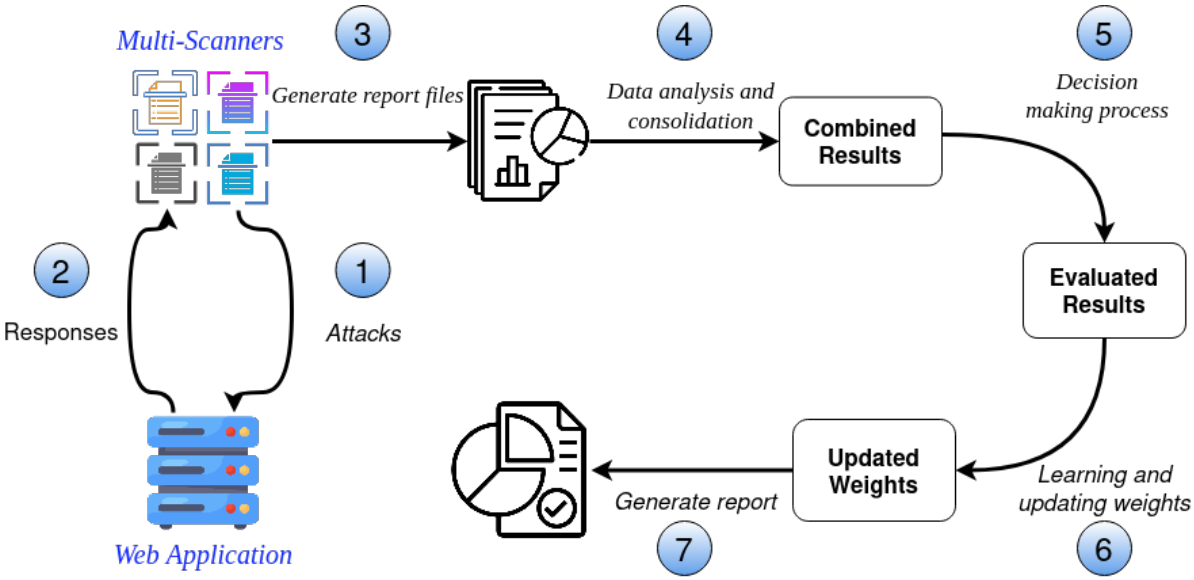

FIGURE 3.1: Overall architecture of the proposed Meta-scan based system.

Next, we focus on reducing the false-positive/negative rates associated with vulnerability detection. False positives can be a significant challenge, as they can lead to unnecessary investigations and wasted resources. In the other hand, false negatives leaves the application exposed to potential security risks and threats that may go unnoticed. To tackle this issue, we employ advanced techniques within the meta-scan system to identify and filter out false positives/negatives. By leveraging the collective intelligence of multiple scanners, we can cross-validate the results and identify genuine vulnerabilities with greater confidence.

The meta-scan system uses a learning process to improve its performance. It employs a sophisticated algorithm that analyzes the results from each scanner and dynamically adjusts their weights based on their accuracy and effectiveness. This adaptive approach ensures that the system becomes increasingly proficient at identifying true vulnerabilities while minimizing false positives/negatives over time.

Overall, our meta-scan system offers a robust and innovative solution to enhance the detection of injection vulnerabilities. By harnessing the power of multiple scanners and employing advanced filtering techniques, we can provide more accurate and

reliable security assessments for web applications.

The process illustrated in Figure 3.1 encompasses three key steps designed to achieve the desired objective. The first step consists of selecting and configuring basic scanners and a target web application. Each scanner performs a scan using its own technique, including crawling the application to identify entry points and launching attacks by generating or using existing payloads for each vulnerability. Afterwords, the scanners analyze the target's responses and generate individual reports.

The second step employs a meta-learning algorithm that combines the individual reports and extracts the results to create a comprehensive outcome. During the learning phase, the proposed algorithm evaluates the individual reports using ground truth and rewards scanners with correct predictions. It accomplishes this by updating a weight matrix, which adjusts the importance of each scanner in the overall assessment. Finally, the system generates a detailed report encompassing information about each vulnerability detected by the meta-scanner and each individual scanner, as well as details about the successful attack vectors employed for further investigation.

To sum up, by implementing the meta-scan system, our aim is to overcome the limitations of individual scanners and enhance the accuracy of vulnerability detection. This comprehensive approach offers a more robust and reliable method for identifying injection vulnerabilities in web applications, ultimately improving the overall security posture of these scanners.

3.3 Meta-scan based system phases

The meta-scan system is structured into several distinct phases. Initially, the system involves the selection of appropriate scanners. Once selected, the scanning process is initiated, and it progresses through subsequent stages, including report file analysis and consolidation. For decision-making, a crucial learning phase is undertaken to improve and initialize a weight matrix. This learning phase ensures the system's

effectiveness and prepares it for optimal performance. Ultimately, the system generates a comprehensive report summarizing the detected vulnerabilities together with successful attack vectors for further examination.

3.3.1 Selection of base scanners

The initial phase of the proposed meta-scan system involves the selection of base scanners based on a set of criteria. To streamline this process, several qualitative criteria were considered:

1. **Injection vulnerability detection support:** The first criterion involves evaluating the scanners' ability to detect injection vulnerabilities. Given the focus of the study on injection vulnerabilities, it was essential to select scanners that were specifically designed to identify such vulnerabilities effectively.

2. **Compatibility:** The second criterion emphasizes the importance of selecting scanners that are compatible with the same operating system and environment. This compatibility ensures smooth integration and parallel utilization of the scanners within a local meta-scan system.

3. **CLI and daemon support:** Candidate scanners should provide support for command line interface (CLI) mode and daemon functionality. This support facilitates seamless integration and operation within the system, enabling automation and efficient utilization.

4. **Active developer community:** The fourth criterion considers the scanners' developer community and their level of activity. It was crucial to choose scanners with an active and dedicated community of developers to ensure regular updates, timely bug fixes, and continuous improvements as vulnerabilities evolve over time.

5. **Source code availability:** The fifth criterion emphasizes the availability of source code or free usage of the scanners, enabling potential modifications or customization. Access to the source code provides flexibility to adapt the scanners to specific requirements or enhance their functionalities within the intended meta-scan system.

6. **Error-free installation and usage:** The sixth criterion stresses the importance of a smooth and error-free installation and usage process. It was essential to ensure that there were no errors encountered during the installation of the system or any issues arising during its usage. Thorough attention was given to guaranteeing a seamless and error-free experience throughout the entire installation and utilization of the system.

7. **User ratings and feedback:** The final criterion involves considering user ratings and feedback within cybersecurity communities, such as the OWASP community, as well as reviewing evaluation papers and articles. This criterion helped assess the reputation, reliability, and performance of the scanners based on real-world experiences and expert opinions.

By adopting these criteria, the selection of scanners within the meta-scan system could be streamlined, ensuring that the chosen scanners met specific requirements and aligned with the objectives of the study.

3.3.2 Configuration of selected scanners

In the system's workflow, the second step involves configuring each scanner to align with specific scanning requirements and the meta-scanner optimization. This involves adjusting various configuration settings that govern each scanner's behavior during the scanning process. It is important to note that not all scanners have the same settings. However, there are certain settings that are shared among multiple scanners. The following is a set of commonly used settings that need to be adjusted:

1. **Crawling depth:** This setting determines the depth to which the scanner should crawl the application. It specifies the number of levels or layers of web pages the scanner should traverse from a given URL. A higher crawling depth may result in a more comprehensive scan, but it could also increase scanning time.

2. **Crawling max children:** This setting defines the maximum number of child pages that the scanner should consider for each parent page during the crawling process. It helps control the breadth of the scan by limiting the number of child pages that can be explored from a given parent page.

3. **Crawling percentage:** This setting allows the specification of the percentage of the website that should be crawled by the scanner. It offers a way to focus the scanning efforts on specific sections of the application, optimizing the scanning process by excluding irrelevant or less critical sections of web pages.

4. **Number of pages to scan:** This setting determines the maximum number of pages that the scanner should include in the scan. It enables limiting the scope of the scan to a specific number of pages, which is useful when scanning large web applications where scanning every page may not be feasible or necessary.

5. **Max time for each scanner:** This setting sets a maximum time limit for each scanner to complete its scanning activities for a given target. It helps control the scanning duration for individual scanners, preventing excessively long scans that could impact efficiency.

6. **Vulnerabilities to be scanned:** This setting allows for specifying the particular types of vulnerabilities that each scanner should prioritize in its detection capabilities. It is important to note that different scanners may possess the capability to identify a range of vulnerabilities beyond injection vulnerabilities. This flexibility allows for the customization of each scanner to concentrate on specific security flaws that are of interest to the meta-scanner.

By configuring these settings for each scanner, the meta-scan system can tailor the scanning process to meet specific requirements, optimize scanning efficiency, and focus on the desired vulnerabilities while ensuring appropriate coverage.

3.3.3 Initiating the scanning process

In this stage of the process, we begin by inputting the target URL into the system and initiating the scanning phase. Each scanner within the meta-scan system performs its designated scan task based on the previously configured settings, ensuring a tailored approach to vulnerability detection. As the scanning proceeds, the meta-scan system collects report files that provide comprehensive information regarding the vulnerabilities identified by each scanner.

An important aspect to highlight is that the meta-scan system leverages the use of threads, enabling parallel execution of base scanners. This pseudo-parallelism significantly mitigates the potential drawback of time consumption associated with running multiple scanners sequentially [65]. By utilizing threads, the system effectively distributes the scanning workload among multiple threads, maximizing efficiency and reducing overall scanning time. This allows for quicker identification of vulnerabilities and expedites the generation of comprehensive scan reports.

3.3.4 Data analysis and consolidation

Given that different base scanners may employ individual report prototypes and utilize various file formats such as JSON, XML, TXT, and HTML, it becomes crucial to establish a comprehensive and automated process for extracting relevant data. To accomplish this task, the meta-scan system incorporates a preprocessing module that plays a crucial role in reading and processing each scanner's report. It applies four functions, namely *filtering*, *normalization*, *successful attack quantification*, and *merging*, that are tailored to handle the nuances of the specific report format and different scanner report structures (see Figure 3.2):

1. **Filtering:** This function focuses on extracting only the necessary information from each scanner report. It selectively considers essential details such as vulnerability detection, the location where the vulnerability is found, and the successful attack vector, if any. By filtering out useless information, the system can streamline the subsequent analysis process and reduce clutter in the final report.

2. **Normalization:** The normalization function addresses the issue of different scanners using different names for the same vulnerability. To tackle this challenge, the system adopts the use of CWE (Common Weakness Enumeration) notation. CWE [66] serves as a standardized catalog of software and hardware weakness types, created by the security community. It acts as a shared language for identifying and addressing vulnerabilities in both software and hardware systems. By unifying vulnerability names through CWE notation, the system ensures consistency and facilitates accurate comparison and analysis of vulnerabilities detected by different scanners.

3. **Successful attack quantification:** In order to provide valuable insights for further examination, the meta-scan system uses a quantification function that counts the number of successful attacks detected by each scanner. These counts are recorded and included in the final report. Therefore, the function analyzes each individual scanner report separately. By scrutinizing the content of each report, the function identifies and tallies the instances of successful attacks. This process enables the system to capture valuable information about the severity and impact of detected vulnerabilities.

4. **Merging:** The merging function is adopted to consolidate all the extracted information into a single file. It combines the filtered and preprocessed data from multiple scanners reports into a unified data. By merging the information, the system provides a comprehensive view of all the identified vulnerabilities, their locations, and associated successful attack vectors. This consolidated file improves data accessibility and simplifies further analysis and decision-making

processes. Merged data is stored in JSON (JavaScript Object Notation) format [67]. JSON offers a clear and standardized representation of vulnerability information, making it easily readable and comparable. Storing the data in JSON format enhances the system's ability to analyze and interpret the number of vulnerabilities identified and the corresponding successful attacks. The standardized representation facilitates effective analysis, decision-making, and reporting based on the vulnerability scan results.

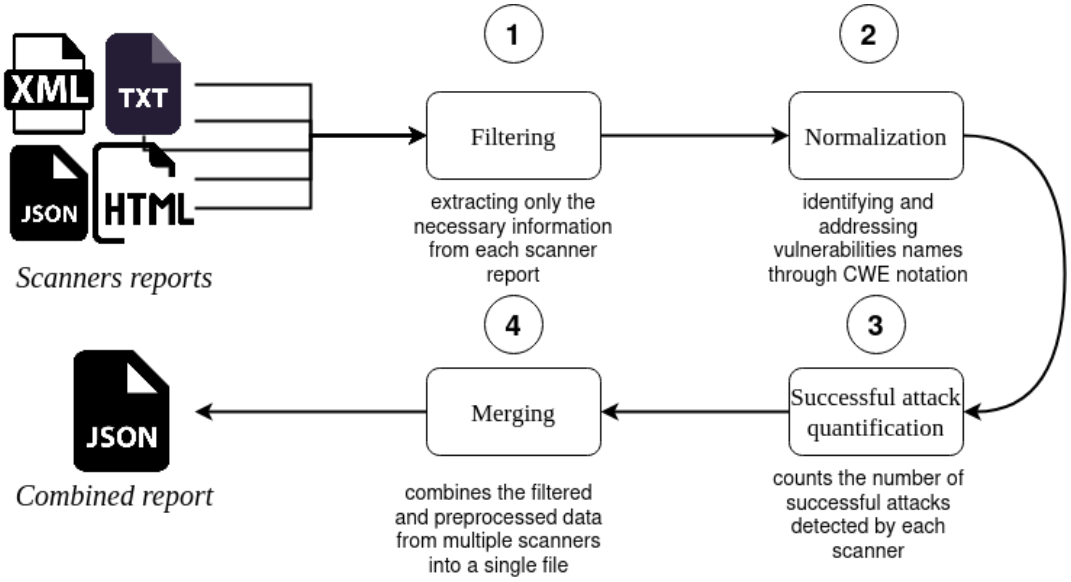

FIGURE 3.2: Data analysis and consolidation phase.

By leveraging these automated preprocessing module, the meta-scan system optimizes the extraction, organization, and representation of vulnerability data, leading to improved readability, comparability, and overall effectiveness in analyzing and addressing identified vulnerabilities. This approach not only saves time and effort but also enhances the accuracy and reliability of the vulnerability data, enabling more efficient vulnerability management and mitigation strategies.

3.3.5 Decision-making process

Once individual scanners' reports are processed and consolidated into a standardized and readable JSON format, a decision-making algorithm is employed. This algorithm considers the impact and efficiency of each individual scanner in detecting each vulnerability. The algorithm aggregates the responses from multiple scanners, considering their respective weights, and produces a prediction score for each vulnerability. The weight matrix and the aggregation process enable the algorithm to prioritize and confirms or declines the presence of each vulnerability based on the combined input from all the involved scanners. To achieve this aim, the algorithm makes use of a weight matrix that is updated during the learning phase and applied during the prediction phase. The prediction function is defined as follows:

$$vulnerability\_score(v) = \sum_{i=1}^{n}(\frac{sr_{iv} * w_{iv}}{n})$$

Where sr_{ij} represents the response of the *ith* scanner for a vulnerability v, w_{iv} denotes the corresponding weight assigned to vulnerability v, and n indicates the total number of scanners involved in the meta-scan system.

To make the final decision regarding the presence of vulnerabilities, the meta-scan system makes use of a binary step function called $meta\_scan\_vulnerability(v)$. This function applies distinct threshold values for each vulnerability v. The binary step function is defined as follows:

$$meta\_scan\_vulnerability(v) = \begin{cases} 1, & \text{if } vulnerability\_score(v) \geq threshold_v \\ 0, & Otherwise \end{cases}$$

The $meta\_scan\_vulnerability(v)$ plays a crucial role in determining the presence of a vulnerability v based on the combined responses from each scanner, considering the significance of each scanner for detecting such vulnerability. In simpler terms,

the function evaluates whether the vulnerability score, derived from aggregating the scanner responses, surpasses a predefined threshold value assigned to that vulnerability. If the vulnerability score exceeds the threshold, the scanner reports the presence of the vulnerability. On the other hand, if the vulnerability score falls below the threshold, the meta-scan system considers the application to be safe from that particular vulnerability. Figure 3.3 illustrates the overall decision-making process.

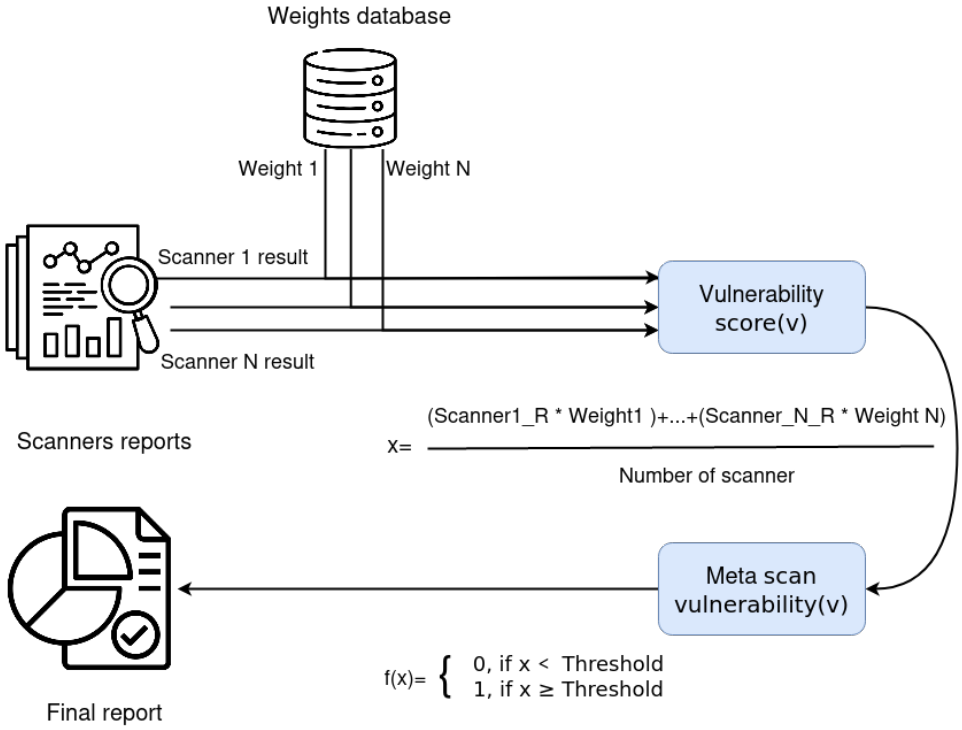

FIGURE 3.3: Decision-making process for one vulnerability.

3.3.5.1 Threshold selection for each vulnerability

The selection of a threshold value for each vulnerability in the decision-making process is crucial. The threshold values are set based on empirical analysis to optimize the decision-making process, taking into consideration two primary criteria:

1. **Minimum number of scanners:** The threshold takes into account the minimum number of scanners required to report the presence of a vulnerability. This criterion ensures that the presence of a vulnerability is confirmed by a sufficient

number of scanners before it is flagged as detected. By establishing a minimum number of scanners, the system avoids false positives/negatives that may arise from a single scanner's response.

2. **Number of involved scanners:** The threshold is also influenced by the number of scanners specifically designed to detect the particular vulnerability. This factor acknowledges the expertise and specialization of certain scanners in identifying specific vulnerabilities. If a vulnerability is assigned to a limited number of scanners, the threshold may be adjusted accordingly to account for the expertise and reliability of those scanners in detecting that vulnerability.

By considering both the minimum number of reporting scanners and the involvement of specialized scanners, the threshold determination process in the meta-scan system aims to strike a balance between avoiding false positives/negatives and ensuring accurate identification of vulnerabilities. To determine the threshold for vulnerability detection in the meta-scan system, a ratio is established using the minimum number of scanners required to confirm the presence of a vulnerability. This can be described using the following formula:

$$threshold_v = \frac{\text{minimum number of scanners required to confirm the presence of } v}{\text{number of scanners able to detect } v}$$

Let's consider a scenario where there are n scanners, and at least m scanners need to report the vulnerability to validate its presence. In this case, the threshold is calculated as m/n, ensuring that a sufficient proportion of scanners have identified the vulnerability before its presence is confirmed by the meta-scan system. This threshold ratio enables a balance between sensitivity and specificity in vulnerability detection.

It's important to note that during the learning phase, the weight assigned to a scanner may be adjusted based on the learning process. As a result, a single scanner

may carry the equivalent weight of two or more scanners, considering its effectiveness in the detection of a specific vulnerability. This weighting process enhances the accuracy and reliability of the meta-scan system's decision-making.

3.3.5.2 Learning and updating weights

The learning phase plays a pivotal role in the decision-making process of the meta-scan system. It involves the evaluation of scanners and the estimation of their weights, reflecting confidence in their ability to detect each injection vulnerability. This evaluation is achieved through an initialization and subsequent updating of a weight matrix.

To begin, all the scanners involved in the meta-scan system are initially assigned an equal weight value of 1. This ensures a fair starting point for the evaluation process. Subsequently, the weight matrix is refined by analyzing the results obtained from a diverse set of web applications with well-known vulnerabilities (i.e., ground truth). These web applications are carefully selected to encompass a wide range of vulnerabilities, providing a comprehensive assessment of the scanners' performance. The weight updates in the matrix are determined based on the principle of rewarding scanners for accurate predictions and penalizing them for false predictions.

The updates to the weight matrix follow a specific formula that captures the essence of the reward system. By analyzing the performance of each scanner, the weight matrix is adjusted to reflect the scanners' proficiency in detecting known vulnerabilities in each web application. The learning phase ensures that the weight matrix evolves and improves over time, providing more accurate assessments of the scanners' effectiveness. This iterative process enhances the meta-scan system's ability to make informed decisions based on the combined insights from multiple scanners.

We have tailored the traditional combined probability formula to incorporate the desired weight adjustments for more accurate decision-making. The customized formula considers two key factors: the *reward_rate* and the *penalty factor* (λ), which represents a proportion of the reward rate used to penalize unsuccessful scanners. When the collective findings from multiple scanners yield a positive value:

$$w'_{sv} = w_{sv} + (sr_{sv} * \text{reward\_rate} + (sr_{sv} - 1) * \lambda * \text{reward\_rate})$$

When the collective findings from multiple scanners yield a negative value:

$$w'_{sv} = w_{sv} + (-sr_{sv} * \lambda * \text{reward\_rate})$$

In the meta-scan system, the weight matrix undergoes continuous updates based on the performance of each scanner in detecting vulnerabilities. When a scanner successfully detects a vulnerability, its weight is augmented by adding a reward rate value. This reward value acknowledges the scanner's accurate detection for the detected vulnerability and reinforces its role in subsequent decision-making. Conversely, when a scanner fails to detect a vulnerability, its weight is decreased by subtracting a proportion of the reward rate which is considered as a penalty value. This penalty value is determined based on the degree of deviation and serves to correct and refine the weight, promoting more accurate weight assignments. By applying this customized weight adjustment formula to all the weights in the weight matrix, the system effectively updates the matrix. This iterative process ensures that scanners with higher accuracy and reliability receive higher weights, thereby amplifying their influence on the decision-making process. Conversely, scanners with lower accuracy are adjusted accordingly, accounting for their diminished contribution. The incorporation of reward and penalty values in the weight updating formula allows the meta-scan system to continually adapt and improve the weight matrix. This iterative refinement process enhances the overall performance and reliability of the system's decision-making capabilities, enabling more precise vulnerability detection. It is important to note that the specific reward and penalty values used in the formula need to be identified through an empirical analysis. Through experimentation and evaluation, the optimal values are determined to align with the desired system performance, ensuring the weight adjustments effectively capture the scanners' accuracy and reliability. Algorithm 1 describes a pseudo-code of the decision-making process.

Algorithm 1: Weight matrix adjustment algorithm

Data: n = number of scanners, v = number of vulnerabilities, WeightsMatrix[0..v][0..n] = weight matrix, rr = reward rate, λ = penalty factor, ScannersReports[0..v][0..n] = scanners reports, th[0..v] = threshold values for vulnerabilities ;

Result: WeightsMatrix[0..v][0..n];

```
# initialization
for i := 0 to v do
    for j := 0 to n do
        WeightsMatrix[i][j] := 1;
for j := 0 to v do
    vulnerabilityScore := 0;
    for i := 0 to n do
        vulnerabilityScore := vulnerabilityScore + (WeightsMatrix[j][i] *
         ScannersReport [j][i]) ;
    vulnerabilityScore := vulnerabilityScore / n;
    if vulnerabilityScore ¿= th[j] then
        TemporaryDecision := 1 ;
    else
        TemporaryDecision := 0 ;
    if TemporaryDecision ==1 then
        for i := 0 to n do
            WeightsMatrix[j][i] := WeightsMatrix[j][i] + ( (ScannersReports[j][i]
             * rr) + ( ScannersReports[j][i] - 1 ) * λ * rr) ;
    else
        for i := 0 to n do
            WeightsMatrix[j][i] := WeightsMatrix[j][i] + ( ScannersReports[j][i]
             * (- λ * rr) )) ;
return WeightsMatrix;
```

The provided pseudo-code 1 demonstrates the decision-making process in the meta-scan system, incorporating the weight matrix calculation. It iterates through each vulnerability, evaluates each scanner response, and determines whether the vulnerability should be reported or ignored based on the total weight and the associated threshold. The weight matrix is initialized, and weights are adjusted during the learning phase as described by adding a reward or subtracting a penalty value accordingly.

The algorithm 1 complexity, denoted as $T(n) = O((3*v*n)+(6*v))$, can be simplified to $T(n) = O(n^2)$, indicates a quadratic complexity represented by $O(n^2)$. In this context, quadratic complexity implies that the algorithm's performance grows quadratically with the input size. As the input size (n) increases, representing the number of scanners, the execution time of the algorithm increases at a rate proportional to the square of the input size. Additionally, this growth rate is also influenced by the number of vulnerabilities (v). Consequently, for larger inputs, the algorithm will take significantly longer to execute compared to algorithms with linear or constant time complexity. The notation "$O(n^2)$" establishes an upper bound on the growth rate of the algorithm's time complexity. However, it is worth noting that given the typical ranges of the number of combined scanners and vulnerabilities, the execution time of the algorithm remains within feasible and real-time calculations.

3.3.6 Reporting

The final report of the meta-scan incorporates the use of weight matrix adjusted as described by algorithm 1. The decision-making process provided in algorithm 2 is implemented to ensure the generation of accurate and comprehensive results by the meta-scan system. The final report is constructed based on the decision made by algorithm 2.

Algorithm 2: Decision-making algorithm

Data: n = number of scanners, v = number of vulnerabilities,
WeightsMatrix[0..v][0..n] = weight matrix, ScannersReports[0..v][0..n] = scanners reports,
th[0..v] = threshold values for vulnerabilities ;

Result: *metaScanVulnerabilities*[0..v] = final report;

for j := *0 to* v **do**

    vulnerabilityScore := 0;

    for i := *0 to* n **do**

        vulnerabilityScore := *vulnerabilityScore* + (WeightsMatrix[j][i] * ScannersReport [j][i]) ;

    vulnerabilityScore := *vulnerabilityScore* / n;

    if *vulnerabilityScore* ¿= $th[j]$ **then**

        metaScanVulnerabilities[j] := ”Positive” ;

    else

        metaScanVulnerabilities[j] := ”Negative” ;

return *metaScanVulnerabilities*;

After the decision-making phase, a meticulous and comprehensive report is generated to provide an in-depth overview of the security assessment, empowering informed decision-making and further actions. The final report is crafted in HTML file format, ensuring compatibility and convenience for users. The use of HTML allows for the creation of visually appealing and interactive reports, enhancing the user experience. With HTML, the report becomes easily accessible across various devices and platforms, enabling seamless navigation and exploration of the findings.

The HTML report features a well-structured matrix that contains detailed information about all detectable vulnerabilities. Each row in the matrix represents a specific vulnerability, while the columns correspond to the different scanners involved in the security assessment. This matrix layout provides a comprehensive and organized

overview of the vulnerabilities identified by each scanner. Within the matrix, relevant details regarding each vulnerability are presented. These details include the vulnerability information reported by each scanner and the final decision made by the meta-scan system regarding the presence or absence of the vulnerability. By consolidating this information, the report effectively communicates the collective findings from all scanners, enabling users to grasp the overall risk profile of the tested application.

By presenting the information in a well-structured and visually appealing format, the comprehensive report enables users to gain valuable insights into the vulnerabilities detected and their implications. Users can assess the severity, frequency, and impact of each vulnerability, assisting in prioritizing remediation efforts and making informed decisions regarding the security posture of the tested application.

	Scanner_1	Scanner_2	⋮	Scanner_n	Meta-scan decision
Vulnerability_1	#	#	...	#	P/N
Vulnerability_2	#	#	...	#	P/N
...	...	...	...	...	P/N
Vulnerability_v	#	#	...	#	P/N

TABLE 3.1: Abstract representation of the final report.

The final report is presented in Table 3.1, where the representation includes the following notations: "#" represents the number of successful attacks generated by each scanner for each vulnerability, while "P/N" indicates whether the vulnerability is classified as "*positive*" (indicating its existence) or "*negative*" (indicating its absence).

3.4 Conclusion

Detecting injection vulnerabilities in web applications poses challenges due to various approaches with their own advantages and limitations. This chapter proposes a novel meta-scan system that combines penetration testing and vulnerability scanning to address the issue of false positives/negatives commonly encountered in individual scanners. The primary objective of the meta-scan system is to overcome the limitations of false positives/negatives by strategically combining multiple scanners based on comprehensive criteria. To combine and evaluate scanner results, a methodology is devised that involves converting individual report files from each scanner into a standardized format. Additionally, a customized combined probability formula is used to update weights during the learning phase, which are then utilized in the testing phase for detecting vulnerabilities in unseen web applications. The learning phase involves rewarding scanners for accurate predictions and penalizing false predictions, allowing continuous refinement of the weight matrix.

The meta-scan system leverages the unique strengths, capabilities, and vulnerability lists of each scanner. By combining their results, two primary advantages are achieved. Firstly, the system expands the list of detectable injection vulnerabilities beyond what any single scanner can accomplish. Secondly, and most importantly, the occurrence of false positives/negatives can be significantly reduced. The reduction in false positives benefits testers and developers by saving time and enabling focused improvements based on scanner results. On the other hand, the reduction in false negatives ensures a better protection of web applications. The proposed meta-scan system offers an innovative approach that improves the detection of injection vulnerabilities in web applications. By combining multiple scanners and refining the weight matrix through the learning phase, the system provides enhanced accuracy, expanded vulnerability coverage, and reduced false positives/negatives. These advancements contribute to more effective testing and development of secure web applications.

CHAPTER 4
IMPLEMENTATION & EXPERIMENTATION

In this chapter, we present the development and implementation details of the proposed meta-scan system, building upon the design scheme described in Chapter 3. The aim is to combine a set of reputable open-source vulnerability scanners into an automated framework, offering a user-friendly graphical interface. To accomplish this task, we leverage the power of the Kali Linux operating system, renowned for its robustness and specialization in tasks like penetration testing, digital forensics, and ethical hacking [68]. Throughout this chapter, we provide comprehensive coverage of the system requirements, installation steps, and experimental results obtained from the resulted meta-scan system. Additionally, we delve into the intricate implementation details, shedding light on the tools and software used. Therefore, we provide a thorough understanding of the meta-scan system and its capabilities for effectively identifying injection vulnerabilities and enhancing the security assessment process.

4.1 Selection of base scanners

In order to select the most suitable scanners for our meta-scan system, we conducted a comprehensive evaluation process based on well-defined qualitative criteria outlined in Section 3.3.1. Several scanners were subjected to this evaluation, including

Vega Scanner [69], [70], Vulmap [71], Sitadel [72], OpenVAS [73], Ratproxy [74], Arachni [60], W3af [51], Astra [75], Vulscanpro [76], among others, as listed in Table 4.1. After careful consideration and analysis, we identified the top five scanners that successfully met our stringent criteria (see Table 4.1). These scanners were chosen based on their performance, reliability, and effectiveness in detecting injection vulnerabilities. The selected scanners that emerged as the most promising solutions for our meta-scan system are OWASP Zap [56], SkipFish [52], Nikto [77], Nuclei [78], and Wapiti Scanner [57]. These scanners demonstrated exceptional capabilities in the detection of injection vulnerabilities and exhibited compatibility with our system's objectives and requirements.

4.1.1 OWASP ZAP

ZAP stands out as one of the most extensively employed web application scanners. As an open-source project, it benefits from the continuous efforts of an international team of dedicated volunteers. Its popularity is evidenced by its inclusion among the top 1000 projects on GitHub. The OWASP ZAP scanner is development in 2009. ZAP offers a diverse range of scans, with a primary focus on passive and active scans, as well as spiders. These scans target various vulnerabilities, including prominent ones like SQL injection, cross-site scripting, and server-side template injection. It boasts a comprehensive feature set, including the ability to operate as a proxy server, enabling users to intercept and modify requests and responses. It offers a desktop application interface for server interaction and supports multiple automation methods, such as command line scans, docker packaged scans, and a recently developed automation framework with API and Daemon mode. This framework provides extensive control over the ZAP server, offering support for three technologies: Bash, Java, and Python. Specifically, Python grants a complete support and control over the ZAP server [79].

4.1.2 Wapiti

Wapiti is an open-source web application vulnerability scanner. It conducts black-box scans on the web pages of deployed web applications, systematically fuzzes URL parameters and forms to identify prevalent web vulnerabilities. The tool was developed and released in 2006. The open-source nature of Wapiti enables users to freely use its features, fostering broad accessibility and encouraging collaborative development within the community. Wapiti offers the capability to assess the security of targeted applications by detecting over 20 vulnerabilities and threats, providing valuable insights for security audits [54].

4.1.3 SkipFish

Skipfish is a proficient active reconnaissance tool designed for web application security. Its functionality involves conducting a recursive crawl and employing dictionary-based probes to generate an interactive sitemap of the target site. The obtained map is further enriched with the results derived from several active security checks, all carefully engineered to avoid disruption. The ultimate objective of the tool is to provide a solid basis for conducting professional web application security assessments. Skipfish was initially released in 2007 and has since been maintained by a team including highly reputable cybersecurity experts [80].

	Injection vulnerability detection support	Compatibility	CLI and daemon support	Active developer community	Source code availability	Error-free installation and usage	User ratings and feedback
Arachni	●	●	●	○	●	○	.
Astra	◑	●	●	●	●	○	.
Burp suit	●	●	○	●	○	●	.
GoLismero	●	●	●	○	●	○	.
Nikto	◑	●	●	●	●	●	4
Nuclei	●	●	●	●	●	●	5
OpenVAS	○	●	○	●	●	○	.
OWASP zap	●	●	●	●	●	●	1
Ratproxy	◑	●	○	○	○	●	.
Ronin-Vulns	◑	●	●	○	●	○	.
SkipFish	●	●	●	●	●	●	3
Sitadel	◑	●	●	○	●	○	.
SOOS DAST	●	○	○	●	○	●	.
Vega Scanner	◑	●	○	●	●	○	.
Vulmap	○	●	●	○	●	○	.
Vulscanpro	●	●	●	○	●	○	.
Wapiti	●	●	●	●	●	●	2
W3af	●	●	●	●	●	○	.

TABLE 4.1: Evaluation and selection of base scanners: ●: Full support, ○: No support, ◑: Partial support

4.1.4 Nikto

Nikto is a widely recognized and robust web vulnerability scanner that has been actively developed since 2001. It is specifically designed to identify security issues and potential vulnerabilities in web servers and applications. Nikto's comprehensive scanning capabilities and extensive plugin support make it a valuable tool for security assessments. One of the notable features of Nikto is its ability to perform thorough and comprehensive scans across a wide range of web servers and platforms. It can detect common vulnerabilities such as outdated server versions, misconfigurations, insecure file permissions, and known vulnerabilities in web applications. Nikto's extensive database of known vulnerabilities and its regular updates ensure that it can stay up-to-date with emerging threats and vulnerabilities. Nikto's flexibility is enhanced by its plugin architecture, allowing users to customize and extend its scanning capabilities according to their specific needs. With a wide variety of plugins available, users can target specific vulnerabilities or perform specialized scans based on their requirements. Furthermore, Nikto provides detailed and actionable scan reports, presenting the identified vulnerabilities along with recommendations for remediation. This helps security professionals and developers understand the nature and severity of the vulnerabilities and take appropriate actions to address them [81].

4.1.5 Nuclei

Nuclei, introduced by Project Discovery in 2019, is a powerful application designed to streamline the process of sending requests to multiple targets using user-defined templates. Its main strength lies in its ability to perform rapid scanning across a large number of hosts, making it an efficient tool for security assessments. Nuclei supports scanning for a wide range of protocols, including TCP, DNS, HTTP, SSL, File, Whois, Websocket, Headless, and more. This broad protocol coverage enhances its versatility and enables thorough security checks. One of the standout features of Nuclei is

its templating functionality, which empowers users to create highly customizable security checks. By leveraging this feature, security professionals can tailor their scans to specific requirements and focus on detecting vulnerabilities that are relevant to their applications. Another notable aspect of Nuclei is its dedicated repository, which serves as a comprehensive collection of vulnerability templates. This repository is the result of contributions from over 300 security researchers and engineers, offering a diverse range of options to enhance scanning capabilities [54].

Table 4.2 shows the list of injection vulnerabilities, discussed in Section 1.4, that can be detected by the selected scanners. Notably, SQL, XSS, and XML injections are detectable by all the scanners. OS command and CRLF injections can be detected by all the selected scanners except Nikto. HTML injections are detectable by Nikto and Nuclei, while HTTP header injections are detectable by SkipFish and Nuclei.

	OWASP Zap	Wapiti	SkipFish	Nikto	Nuclei
SQL injection	●	●	●	●	●
XSS injection	●	●	●	●	●
OS command injection	●	●	●	○	●
CRLF injection	●	●	●	○	●
XML injection	●	○	●	●	●
Code injection	●	○	○	●	●
XXE injection	●	●	○	○	●
HTTP header injection	○	○	●	○	●
HTML injection	○	○	○	●	●

TABLE 4.2: List of injection vulnerabilities detectable by each scanner.

4.2 Configuration of base scanners

The selected scanners for the meta-scan system offer a range of configurable parameters that allow users to customize their scanning settings according to their specific needs. These parameters play a crucial role in fine-tuning the scanning process and adapting it to different environments and target applications. Note that the specific parameters and their functionalities may vary for each scanner. These parameters allow users to specify various aspects of the scanning process, such as the scope of the scan, the level of thoroughness, the types of vulnerabilities to focus on, and any additional configurations required by the scanner. Table 4.3 shows the list of configurable parameters for each scanner.

	OWASP Zap	Wapiti	SkipFish	Nikto	Nuclei
Crawling depth	●	○	●	○	○
Crawling max children	●	○	●	○	○
Crawling percentage	●	○	●	○	○
Number of pages to scan	○	●	●	○	○
Max time for each scanner	●	●	●	○	○
Vulnerabilities to be scanned	●	●	○	●	●

TABLE 4.3: List of configurable settings of each scanner.

Among the selected scanners, OWASP Zap appears to have the most comprehensive support for configurable settings, covering crawling depth, crawling max children, crawling percentage, maximum scanning time, and the ability to select specific vulnerabilities to scan. It offers a wide range of options for customization, making it a flexible tool for security assessments. Wapiti and SkipFish also provide a significant number of configurable settings, but while they may not offer the same level of flexibility as OWASP Zap, they still provide sufficient options for fine-tuning the

scanning process. On the other hand, Nikto and Nuclei have a more limited range of configurable settings. They lack crawling support entirely and have fewer options for customizing the scanning process. However, they allow for the disabling of unrelated vulnerabilities.

4.3 Initiating the scanning process

This section provides an overview of the scanning process used by the various scanners incorporated in the meta-scan system. Although each scanner has its own distinct approach, there are common features and steps involved in initiating the scanning process. Firstly, all scanners start by using the provided target URL and taking into account the configurable parameters. Secondly, the scanners aim to identify vulnerabilities in the target application by sending requests and carefully analyzing the responses received. Finally, each scanner generates a comprehensive report that provides detailed information about the vulnerabilities identified during the scanning process. Despite these common steps, each scanner follows a distinct scanning process to identify injection vulnerabilities in the target application:

4.3.1 OWASP ZAP

The scanning process begins with ZAP's automated tool, which systematically explores the target application by crawling its pages and discovering available content. This process allows ZAP to build a comprehensive understanding of the application's structure. Once the spidering phase is complete, ZAP activates its active scanner to identify vulnerabilities. The active scanner sends requests to the application and analyzes the responses for potential vulnerabilities using fuzzing techniques. The results are then analyzed and categorized based on predefined criteria to determine whether they are vulnerable or safe. ZAP generates a detailed report summarizing the vulnerabilities detected during the scanning process.

4.3.2 Wapiti

Wapiti adopts a systematic approach to scanning web applications. It starts by crawling the target application, specifically targeting scripts and forms where it can inject data. By gathering a comprehensive list of URLs, forms, and inputs, Wapiti proceeds to fuzz the target by injecting various payloads to assess its vulnerability. Different models are employed for each vulnerability type. Once the scanning phase is complete, Wapiti generates a detailed vulnerability report that provides information on the identified vulnerabilities, helping security professionals address potential risks.

4.3.3 SkipFish

The scanning process in SkipFish begins with the construction of a thorough site map of the target web application. This map includes all accessible pages and directories, providing an overview of the application's structure. SkipFish then conducts a series of tests on each page to detect potential vulnerabilities. These tests involve analyzing response codes, identifying common security issues, and evaluating the application's behavior. The final stage involves generating a comprehensive report that provides detailed information about the identified vulnerabilities, enabling developers and security teams to take appropriate remedial actions.

4.3.4 Nikto

Nikto initiates the scanning process by performing server fingerprinting to gather relevant information about the target, such as the web server version and employed techniques. Following the fingerprinting phase, Nikto rapidly crawls and scrapes the target to identify potential input points where data can be injected. The vulnerability scanning phase uses a pre-defined payload database specific to each vulnerability, allowing Nikto to test the target for known vulnerabilities. Finally, Nikto generates a detailed report that encompasses extensive information obtained during the scanning process, providing insights into potential security risks.

4.3.5 Nuclei

The scanning process in Nuclei revolves around the use of templates. Nuclei sends HTTP requests based on the template specifications for each corresponding identifier. It meticulously compares the received HTTP responses with the predefined expected responses within the template. By examining the responses, Nuclei identifies any deviations or undesirable behaviors that may indicate a vulnerability. Throughout the scanning process, Nuclei gathers pertinent details concerning the template, the target, and any identified responses, which are then included in the generated report.

4.3.6 Sequential vs. Multithreading based execution

Figure 4.1 illustrates the average execution time for each scanner. In order to obtain these results, each scanner was used to scan a single web page with a single input field ten times, and the average execution time was calculated. The data presented in Figure 4.1 demonstrates that OWASP ZAP and Nuclei exhibit longer execution times compared to the other scanners, with OWASP ZAP taking an average of 47.91 seconds and Nuclei requiring approximately 39.81 seconds. Conversely, SkipFish stands out as the fastest scanner, completing the scanning in an average time of 1.27 seconds. Wapiti and Nikto, on the other hand, exhibit similar performance, with average execution times of 18.88 seconds and 16.30 seconds, respectively. Note that the experiment is performed on a laptop machine with an Intel(R) Celeron(R) CPU N3060 (1.60GHz), 4 GB of RAM, and a hard disk drive (HDD).

The longer execution times observed for OWASP ZAP and Nuclei compared to other scanners like SkipFish can be attributed to several factors. OWASP ZAP and Nuclei are feature-rich scanners that offer extensive functionality and a wide range of scanning capabilities. They use more advanced scanning techniques and strategies, such as deep crawling, in-depth analysis, and complex pattern matching. These sophisticated approaches contribute to a more comprehensive evaluation of the target application's security but require additional time for execution. On the other hand,

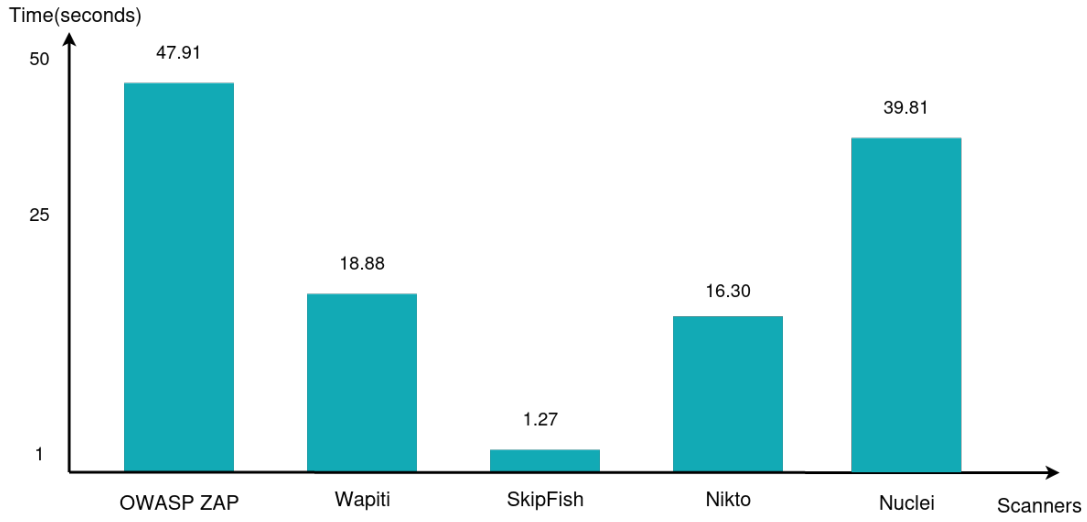

FIGURE 4.1: Average execution time per scanner.

SkipFish, being a lightweight and fast scanner, adopts a more streamlined scanning methodology. It uses a targeted approach, performing efficient tests on each page to identify potential vulnerabilities quickly. This streamlined process contributes to shorter execution times compared to the more comprehensive scanning methods used by OWASP ZAP and Nuclei. Wapiti and Nikto, while not as fast as SkipFish, exhibit comparable execution times when compared to OWASP ZAP and Nuclei. Wapiti and Nikto share some similarities with OWASP ZAP and Nuclei in terms of their scanning capabilities and depth of analysis. They perform comprehensive scans and use a large number of payloads to identify potential vulnerabilities. This level of thoroughness in their scanning approach can contribute to slightly longer execution times compared to more lightweight scanners like SkipFish.

To overcome the time-consuming issue, we propose the utilization of a multi-threading approach. By assigning a dedicated thread for each scanner, we can have precise control over when the scanning process starts. This enables us to initiate all scanners simultaneously, reducing overall scanning time. Figure 4.2 illustrates the significant reduction in execution time achieved by employing a multi-threading-based technique. By leveraging this approach, the scanning process was optimized, resulting in a notable decrease in execution time. Specifically, the execution time decreased

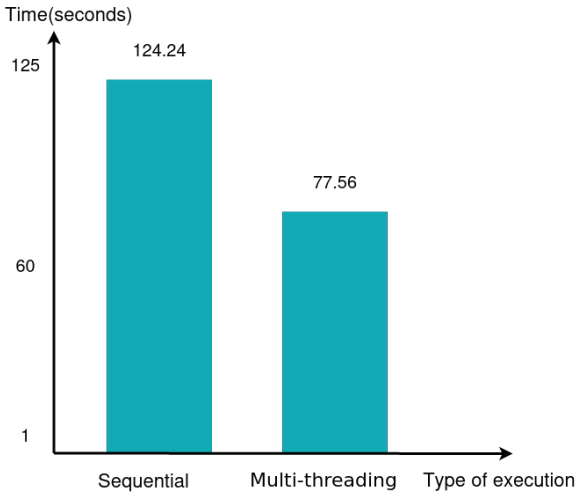

FIGURE 4.2: Sequential vs parallel execution of scanners.

from 124.24 seconds (using a sequential execution or single-threaded approach) to 77.56 seconds (using the multi-threading-based technique). This improvement highlights the effectiveness of using parallel processing capabilities to enhance the efficiency and speed of the scanning process.

4.4 Data analysis and consolidation

As described in Section 3.3.4, each scanner uses its own distinct format and structure for generating their reports. This section describes the data provided by each scanner and how these data are combined to create the final report. We highlight the specific details and attributes offered by each scanner and demonstrate how these individual reports are consolidated to present a comprehensive view of the scanning results.

4.4.1 OWASP ZAP

The OWASP ZAP scanning report encompasses a comprehensive range of attributes that provide valuable insights into the identified vulnerabilities. These attributes include the alert level (risk), which indicates the severity of each vulnerability; the alert

type categorizes vulnerabilities into specific classes, facilitating a deeper understanding of their nature and associated risks. The description field offers in-depth explanations of the vulnerabilities, including their underlying causes, potential impacts, and recommended mitigation measures. Additionally, the URL attribute specifies the target URL where each vulnerability was discovered, aiding in pinpointing the affected input field. For a comprehensive list of these important attributes and their brief descriptions, please refer to Table 4.4.

Attribute		**Description**
1	vulnerabilities::method	HTTP method (GET/POST)
2	vulnerabilities::pluginId	ID of the ZAP module that found the vulnerability
3	vulnerabilities::cweid	Vulnerability CVE unique code
4	vulnerabilities::wascid	Vulnerability WASC unique code
5	vulnerabilities::url	Full URL of the vulnerability
6	vulnerabilities::description	Description of the vulnerability
7	vulnerabilities::alert	Information about the alerts raised by OWASP ZAP
8	vulnerabilities::attack	The full attack vector
9	vulnerabilities::name	Vulnerability name
10	vulnerabilities::risk	Threat level (high/low/medium)
11	vulnerabilities::id	Vulnerability unique id
12	vulnerabilities::alertRef	Alert reference identifier

TABLE 4.4: Information included in the OWASP ZAP report

4.4.2 Wapiti

Wapiti scanner also used JSON file format for generating reports. This JSON file contains various attributes specific to Wapiti, with some of the essential ones outlined in Table 4.5. One of the crucial attributes is the list of vulnerabilities, which provides detailed information about each vulnerability detected by the scanner. Each vulnerability entry includes information about the successful attack triggered during the

scanning process. The provided attributes offer insights into the attack's specifics, severity level, affected endpoint or parameter, the correspondent HTTP request and cURL command associated with the attack.

	Attribute	Description
1	vulnerabilities::name::method	HTTP method (GET/POST)
2	vulnerabilities::name::path	Full URL of the target
3	vulnerabilities::name::info	Information about the vulnerability
4	vulnerabilities::name::level	Dangerous level set by Wapiti (1/2/3)
5	vulnerabilities::name::parameter	Parameter ID if used (ex:"input_id")
6	vulnerabilities::name::http_request	Submitted HTTP request
7	vulnerabilities::name::curl_command	curl command used for submitting the request

TABLE 4.5: Information included in the Wapiti report

4.4.3 SkipFish

The SkipFish scanner uses an HTML report format with a complex structure. For each crawled page, it creates a directory containing four files: *request.js*, *response.js*, *issue_index.js*, and *child_index.js*. The information saved in these files form an intricate representation of the crawling, attacking phases, and report of vulnerabilities found in each page. The Attributes of the different file are illustrated in Table 4.6.

	File::Attribute	Description
1	Response::	Response sent from the specific page
2	Request::	Requests sent to the specific page
3	issue_index::severity	Vulnerability threat level (0~4)
4	issue_index::type	Type of vulnerability set by SkipFish
5	issue_index::code	Vulnerability unique code identifier
6	issue_index::len	Length of the request
7	issue_index::decl_mime	Declared MIME type
8	issue_index::cset	Charset of the response

9	issue_index::dir	Directory name of the Request & Response files
10	child_index::name	Name Of target application
11	child_index::dir	Directory name where the web page is stored
12	child_index::linked	Number of links found in the page
13	child_index::url	Full URL
14	child_index::fetched	Type of fetch (True/False)
15	child_index::code	Response Code (200/302/403/404/500)
16	child_index::sniff_mime	MIME sniffing (None/True)
17	child_index::issue_cnt	Number of identified vulnerabilities

TABLE 4.6: Information included in SkipFish report

4.4.4 Nikto

The Nikto scanner uses a JSON file format. The provided file includes various attributes specific to Nikto, as indicated in Table 4.7, including the HOST attribute, which specifies the hostname of the target application (e.g., localhost or online host), along with the IP address and port number of the target. Additionally, the banner attribute provides details about the server and the target application. A typical example of a banner might be "*Apache/2.4.54 (Unix) OpenSSL/1.1.1s PHP/8.2.0 mod_perl/2.0.12 Perl/v5.34.1.*". Furthermore, the scanner generates a list of detected vulnerabilities, each identified by a unique ID, a number of references to the Open Source Vulnerability Database (OSVDB), a full URL indicating where the vulnerability was found, the HTTP method utilized (GET or POST), and a message (msg) that describes the nature of the detected vulnerability.

Attribute		**Description**
1	host	Target app's hostname (localhost/online host address)
2	ip	IP address of the hostname
3	port	Port number of the target

4	banner	Type of the server, OS, and programming languages used
5	vulnerabilities::id	Identifier of the detected vulnerability
6	vulnerabilities::OSVDB	Reference to the OSVDB vulnerability entry (OSVDB-xxxx)
7	vulnerabilities::method	HTTP method (GET/POST)
8	vulnerabilities::url	URL where the vulnerability was discovered
9	vulnerabilities::msg	Short description of the detected vulnerability

TABLE 4.7: Information included in the Nikto report

4.4.5 Nuclei

The scanning results in Nuclei are also reported using a JSON file format. As described in Table 4.8. The report includes various attributes specific to Nuclei templates used for penteration testing. For each successful attack, Nuclei generates specific attributes related to the template information, such as *template_url* and *template-id*. Additionally, it provides comprehensive details about the vulnerability, including its name, description, severity, type, host, timestamp, IP address, and curl-command.

Attribute		**Description**
1	template	Name of template used
2	template-url	URL of the template site or repostory
3	template-id	Template unique ID given by Nuclei
4	info::name	Vulnerability name
5	info::description	Vulnerability description
6	info::reference	Links and site with description of vulnerability
7	info::severity	Threat level of vulnerability (high/medium/low)
8	info::classification::cve-id	Vulnerability CVE unique code
9	info::classification::cwe-id	Vulnerability CWE unique cod
10	info::classification::cvss-metrics	Vulnerability CVSS metric
11	info::classification::cvss-score	Vulnerability CVSS score
12	info::type	Type of vulnerability set by template

13	info::host	Target Host and type (localhost/onlinehost)
14	info::matched-at	Specific URL where the vulnerability found
15	info::ip	Target IP
16	info::timestamp	Time of Initiating the scan
17	info::curl-command	The curl command used for submitting the request

TABLE 4.8: Information included in the Nuclei report

4.4.6 Consolidation process

In order to compile and produce a comprehensive report, we adhered to the procedures delineated in Section 3.3.4. The initial step entailed filtering the data from each scanner report. We specifically chose pertinent attributes that would facilitate users in comprehending the vulnerable entry points and gaining additional insights into preventive measures. These attributes encompassed key information such as the URL, vulnerability ID, vulnerability description, attack vectors, and more. Table 4.9 outlines the selected attributes and their corresponding counterparts in each scanner report.

	OWASP zap	Wapiti	SkipFish	Nikto	Nuclei
id	11	3	5	5	4
url	5	2	13	8	14
method	1	1	-	7	-
paramater	-	5	-	-	-
attack	8	-	-	-	17
Description	6	3	-	9	5
request	-	6	2	-	-

TABLE 4.9: Meta-scans data Filtering report.

Following the filtering step, it becomes necessary to perform normalization as each scanner adopts its own nomenclature for vulnerabilities or employs unique codes for identification. To achieve normalization, we utilize the Common Weakness Enumeration (CWE) vulnerability codes. For instance, CWE-89 represents SQL vulnerabilities, CWE-79 represents XSS vulnerabilities, and CWE-78 signifies OS command injections. To accomplish this, we employ different approaches for each scanner. In the case of OWASP Zap, we extract the vulnerability names from their official documentation. In Wapiti, we extract the names from the report prototype. For SkipFish, we meticulously examine their static source code until we identify the unique codes associated with specific vulnerabilities. Similarly, for Nikto, we extract the relevant IDs from their vulnerability database and categorize them according to the corresponding vulnerabilities. As for Nuclei, given that we use the CVE template, we extract the IDs for each injection vulnerability provided by Nuclei.

The subsequent step entails quantifying the effectiveness of each scanner in terms of successful attacks. This step involves a straightforward calculation of the occurrence of each successful attack found within each individual report, subsequently determining the total number of occurrences for each vulnerability as reported by each scanner.

Lastly, the concluding step involves consolidating and merging the individual reports and findings into a single comprehensive report file. To achieve this, we have opted for the JSON format, widely utilized and easily comprehensible. Each scanner is assigned its own attribute within the report file, containing a matrix that captures the vulnerabilities detected by that specific scanner, along with the corresponding filtered information and the count of vulnerabilities found. The structure and layout of the final report can be observed in Table4.10.

Attribute	**Description**
OWASPZap::n_attacks::xss	Number of successful XSS attacks
OWASPZap::n_attacks::sql	Number of successful SQL attacks

OWASPZap::n_attacks::os	Number of successful OS command attacks
OWASPZap::vulnerabilities::url	URL where the vulnerability found
OWASPZap::vulnerabilities::method	HTTP method (GET/POST)
OWASPZap::vulnerabilities::id	CWE-id of vulnerability
OWASPZap::vulnerabilities::attack	Attack vector used
OWASPZap::vulnerabilities::description	Description of the vulnerability
Wapiti::n_attacks::xss	Number of successful XSS attacks
Wapiti::n_attacks::sql	Number of successful SQL attacks
Wapiti::n_attacks::os	Number of successful OS command attacks
Wapiti::vulnerabilities::url	URL where the vulnerability found
Wapiti::vulnerabilities::method	HTTP method (GET/POST)
Wapiti::vulnerabilities::id	CWE-id of vulnerability
Wapiti::vulnerabilities::request	Requests sent to the specific page
Wapiti::vulnerabilities::description	Description of the vulnerability
SkipFish::n_attacks::XSS	Number of successful XSS attacks
SkipFish::n_attacks:sql	Number of successful SQL attacks
SkipFish::n_attacks::os	Number of successful OS command attacks
SkipFish::vulnerabilities::url	URL where the vulnerability found
SkipFish::vulnerabilities::id	CWE-id of vulnerability
SkipFish::vulnerabilities::request	Requests sent to the specific page
Nikto::n_attacks:xss	Number of successful XSS attacks
Nikto::n_attacks::sql	Number of successful SQL attacks
Nikto::n_attacks::os	Number of successful OS command attacks
Nikto::vulnerabilities::url	URL where the vulnerability found
Nikto::vulnerabilities::method	HTTP method (GET/POST)
Nikto::vulnerabilities::id	CWE-id of vulnerability
Nikto::vulnerabilities::description	Description of the vulnerability
Nuclei::n_attacks::xss	Number of successful XSS attacks
Nuclei::n_attacks::sql	Number of successful SQL attacks

Nuclei::n_attacks::os	Number of successful OS attacks
Nuclei::vulnerabilities::url	URL where the vulnerability found
Nuclei::vulnerabilities::id	CWE-id of vulnerability
Nuclei::vulnerabilities::command	curl command used for submitting the request
Nuclei::vulnerabilities::description	Description of the vulnerability

TABLE 4.10: Meta-scans data consolidation report.

4.5 Decision-making

In the learning phase, our analysis involves evaluating the results obtained from a set of web applications with known vulnerabilities. This set of applications covers a variety of vulnerabilities, including XSS, SQL injection, and OS command injection. We divide these applications into subsets, using one subset to adjust the weight matrix, and the remaining subset for testing the algorithm's performance. We then compare the results obtained by our algorithm with those achieved by individual vulnerability scanners. As the aim of the study is to reduce the false positives and false negatives of individual vulnerability scanners, we opt for the use of the three following evaluation metrics in the evaluation process:

1. **False positive rate ($\mathbf{FPR}_v$):** measures the proportion of pages that are incorrectly classified as vulnerable to an injection vulnerability v.

2. **False negative rate ($\mathbf{FNR}_v$):** measures the proportion of vulnerable pages to v that are incorrectly classified as not vulnerable to v.

3. **Accuracy ($\mathbf{ACC}_v$):** measures the proportion of pages that correctly detected as vulnerable to an injection vulnerability v out of the total number of pages.

4.5.1 Data description

For experimentation, we use four web applications known for their vulnerabilities, specifically OWASP mutillidae2 [82], OWASP Vulnerable Web Application [83], XTREM vulnerable Web Application [84], and DAMN vulnerable web application [85]. This list of web applications were found and selected from the OWASP Vulnerable Web Applications Directory Project [86]. The four application are written in PHP/MySQL and tested on XAMP environment; they are vulnerable applications written on the purpose of testing web scanners and enhancing security analysis.

1. **OWASP Mutillidae II:** is a freely available, open-source web application deliberately designed to contain vulnerabilities, making it a valuable resource for individuals interested in web security.

2. **OWASP VWA:** designed to cater to individuals interested in web penetration and seeking information or practical experience in this field

3. **XTREM VWA:** deliberately coded with poor practices, and serves as a valuable resource for security enthusiasts seeking to enhance their understanding of application security.

4. **DAMN VWA:** intentionally designed to possess numerous vulnerabilities. Its primary objective is to serve as a valuable resource for security professionals, providing them with a legal environment to test their skills and tools effectively.

We divided each application pages into two halves; the first half is used as a training set and the other half is used as a test set. This resulted in 20 pages for each set, as illustrated in Table 4.11.

	# XSS vulnerable pages	**# SQL vulnerable pages**	**# OS command vulnerable pages**	**# Total pages**
OWASP mutillidae	13	4	6	23
OWASP Vwa	5	6	4	15
XTREM Vwa	3	2	1	6
DAMN Vwa	3	2	1	6
Train set	12	7	6	20
Test set	12	7	6	20

TABLE 4.11: Dataset description.

4.5.2 Parameter tuning

In the decision-making process, it is crucial to determine the optimal parameter values for the proposed decision-making algorithm. This includes the threshold value required to confirm the detection of each vulnerability and the reward value to be assigned to each scanner upon a successful detection. To explore different parameter values and optimize the performance, we employ the grid search method [87]. The threshold value for each injection vulnerability v was incremented by a step size of $1/k_v$, from $1/k_v$ to 1, where k_v represents the number of scanners detecting v, as explained in Section 3.3.5.1. Similarly, the reward value was incremented by a step size of 0.01, from 0.01 to 1. We also experimented different penalty factor values λ with different reward ratios: 1/10, 1/5, 1/4, 1/3, 1/2, and 1.

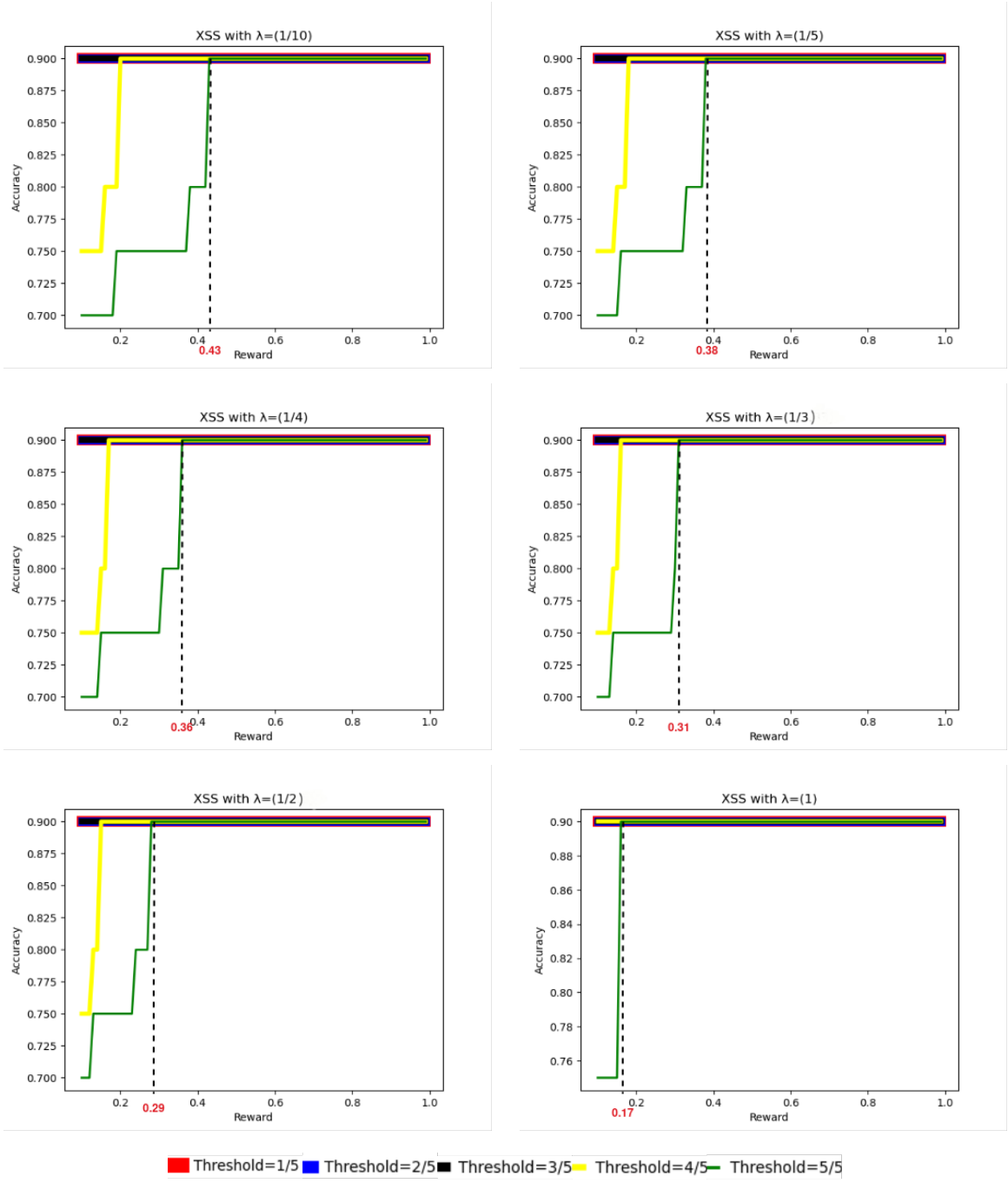

FIGURE 4.3: Parameter tuning for the detection of XSS injection vulnerabilities

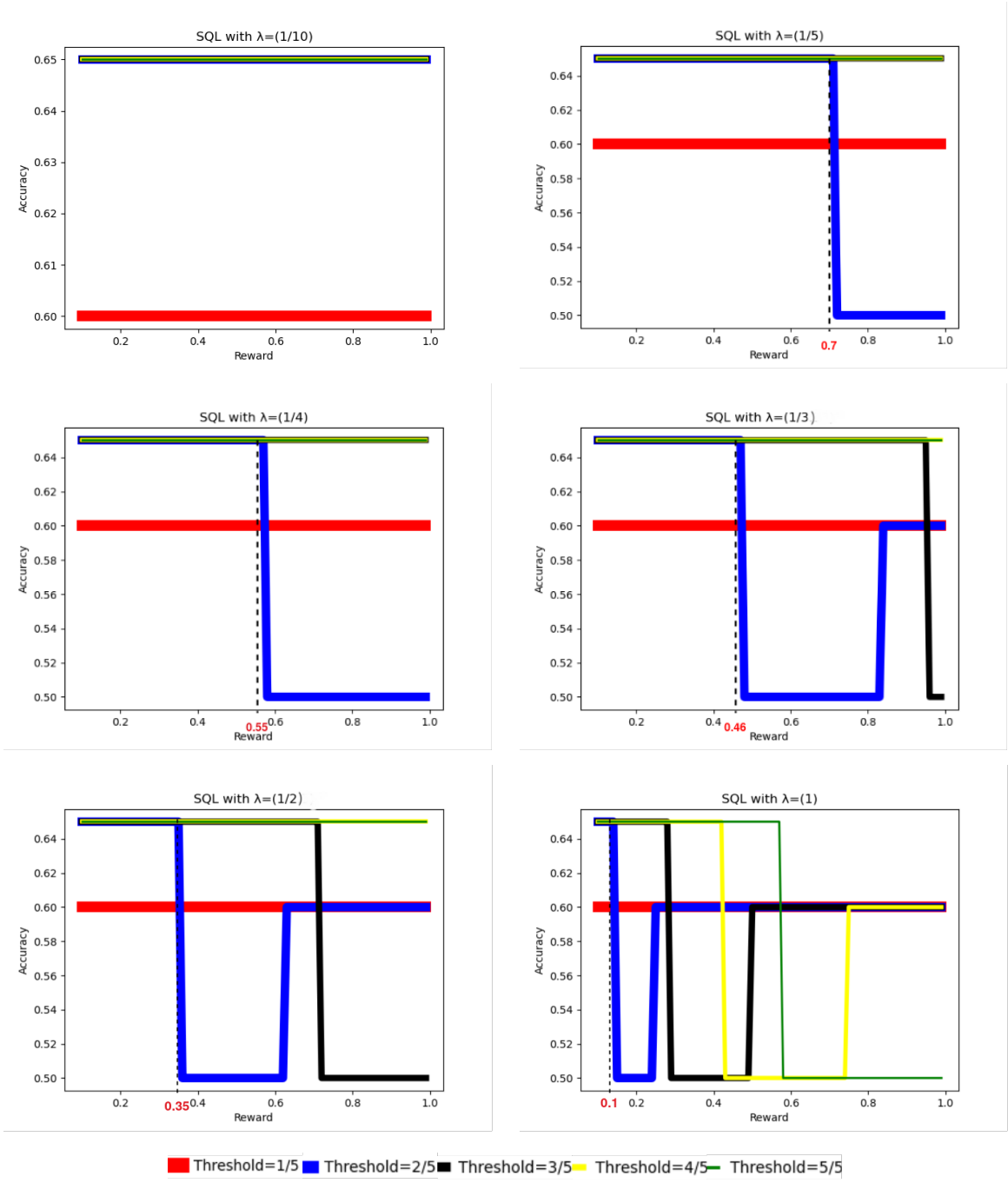

FIGURE 4.4: Parameter tuning for the detection of SQL injection vulnerabilities

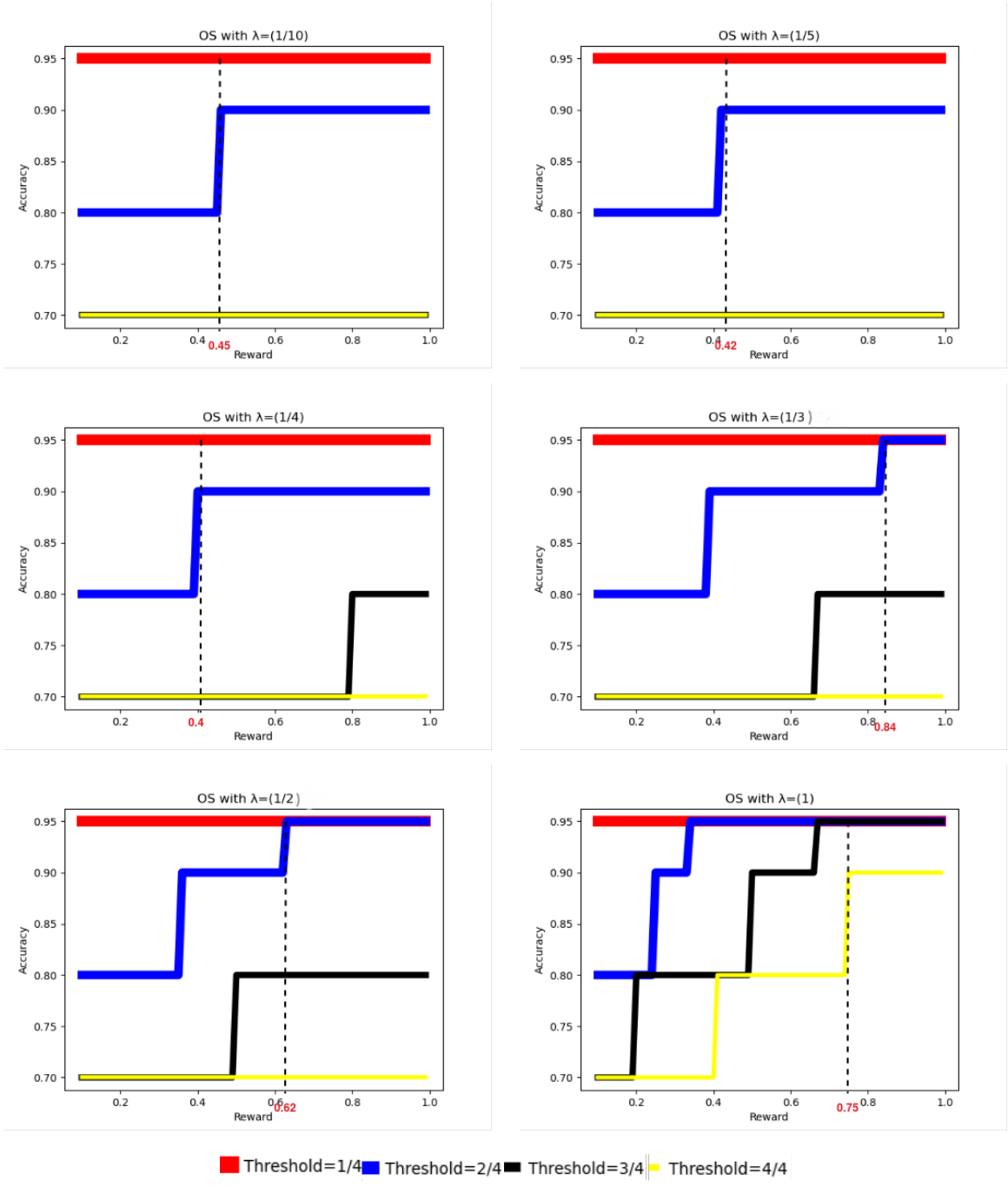

FIGURE 4.5: Parameter tuning for the detection of OS command injection vulnerabilities

The results of the XSS injection vulnerability, illustrated in Figure 4.3, indicate that the optimal threshold for detection is 1/5. This means that at least one of the five scanners should identify the vulnerability in order to report its positive detection. Furthermore, the examined λ values indicate that the most favorable reward rate falls within the range of [0.17, 1], with λ being equal to 1. Based on those findings, it is evident that for XSS injection vulnerability, higher accuracy values are achieved by increasing the reward value and using the same value to penalize unsuccessful scanners. Additionally, this influence is observed in relation to the increase of the threshold; as the threshold increases, the reward and penalty factor have a more significant impact on the accuracy rate. In summary, to obtain the best accuracy, an increase in the threshold should be accompanied by an increase in the reward value and the penalty factor.

The tests conducted for the OS command injection vulnerability, as shown in Figure 4.5, yield a similar conclusion. The optimal threshold value for all the tests is determined to be 1/4, while the best reward value ranges from 0.75 to 1. Additionally, the penalty factor λ needs to be set to 1.

Conversely, regarding the SQL injection vulnerability, Figures 4.4 displayed that the best threshold in all tests encompasses the optimal values is 5/5, and the most desirable values for reward ranges from 0.1 to 1 with a penalty factor being equal to 1/10. Additionally, in relation to the three parameters, as we increase the threshold values and decrease the reward value the accuracy rate increases. Therefore, for obtain the best accuracy, an increase in the threshold should be accompanied by a decrease in penalty factor and reward. As we can notice in the right bottom graph, as we set the λ value to 1 to gain the best accuracy an increase in threshold should be combined with the decrease of reward to 0.1.

After analyzing all the results, we have determined 1/4 as the optimal penalty factor value achieving the highest level of accuracy for all studied injection vulnerabilities. We have also identified the reward value of 0.4, and specific thresholds for XSS, SQL, and OS injections, namely 1/5, 3/5, and 1/5, respectively. These parameter values have been selected as they consistently yield the best accuracy across all

three parameter tuning process. Table 4.12 expliclt and clearly shiws the most suitable parameter values for detecting each injection vulnerability using our proposed decision-making algorithm.

Vulnerability	**Threshold**	**Reward**	**Penalty factor (λ)**
XSS Injection	0.2	0.4	0.25
SQL Injection	0.6		
OS command Injection	0.25		

TABLE 4.12: Best parameter values for the decision-making algorithm.

4.5.3 Experimental results

The optimal parameter values, listed in Table 4.12, are used in the learning phase of the decision-making algorithm. Thus, the algorithm is trained on a set of web application pages with known vulnerabilities, resulting in the generation of the matrix weight depicted in Table 4.13.

	OWASP zap	**Wapiti**	**SkipFish**	**Nikto**	**Nuclei**
XSS injection weights	3.29	3.29	4.3	4.80	4.3
SQL injection weights	1.0	1.40	1.0	2.40	1.00
OS command injection weights	0.50	2.59	1.1	NA	0.60

TABLE 4.13: Meta-scan weight matrix.

The above matrix weight is used to test the decision-making algorithm on a set of unseen web pages with known vulnerabilities and the following evaluation metrics were estimated: false positive rate (FPR), false negative rate (FNR) and accuracy (ACC). The results are illustrated in Table 4.14.

	OWASP zap	Wapiti	SkipFish	Nikto	Nuclei	Meta-scan
XSS injection vulnerability						
FPR (%)	00.00	00.00	00.00	00.00	00.00	**00.00**
FNR (%)	61.54	53.85	23.08	39.46	39.46	**08.33**↓
ACC (%)	60.00	65.00	85.00	75.00	75.00	**95.00**↑
SQL injection vulnerability						
FPR (%)	00.00	07.69	0 0.00	30.77	00.00	**00.00**
FNR (%)	85.71	**57.14**	85.71	71.43	100	71.43
ACC (%)	70.00	**75.00**	70.00	55.00	65.00	**75.00**
OS command injection vulnerability						
FPR (%)	00.00	00.00	00.00	NA	00.00	**00.00**
FNR (%)	50.00	33.33	66.67	NA	100	**16.67**↓
ACC (%)	85.00	90.00	80.00	NA	70.00	**95.00**↑

TABLE 4.14: Experimental test results.

In the case of XSS injection, the results presented in Table 4.14 demonstrate the superior performance of our meta-scan system compared to individual scanners in terms of accuracy. Furthermore, our system significantly reduced the False Negative Rate (FNR) while maintaining the best False Positive Rate (FPR). When compared to the highest-performing individual scanner, SkipFish, our system achieves a remarkable 10% increase in accuracy and a notable 14.75% reduction in FNR.

For the SQL injection vulnerability, the proposed meta-scan system outperformed all individual scanners in terms of accuracy. In comparison to the top-performing individual scanner, Wapiti, we achieved an accuracy level of 75%, which remained the same, while reducing false positive rate by 7.69%. Significantly, our findings were highly promising as we were able to lower the false positive rate to 0%.

Regarding the OS command injection vulnerability, the meta-scan system surpassed the accuracy of all individual scanners. Compared to the best-performing individual scanner, Wapiti, we observed a 5% increase in accuracy and a 16.66% reduction in false negatives. Moreover, we noted the robustness of the scanner in handling this specific vulnerability, as none of the scanners produced false positives in the test cases.

4.6 Reporting and GUI

In order to provide users with a user-friendly experience while using the meta-scan detection system, we have developed a meta-scanner tool based on the design described in Chapter 3. The implemented tool features a streamlined user interface that simplifies the utilization of the meta-scan system. The main view of the meta-scanner tool is depicted in Figure 4.6. The main view provides users with various functionalities, including the ability to initiate a scan, switch between dark and light modes, load saved reports, and save reports in HTML format. These options are available through the buttons located on the left-hand side of the main view or via the menu bar.

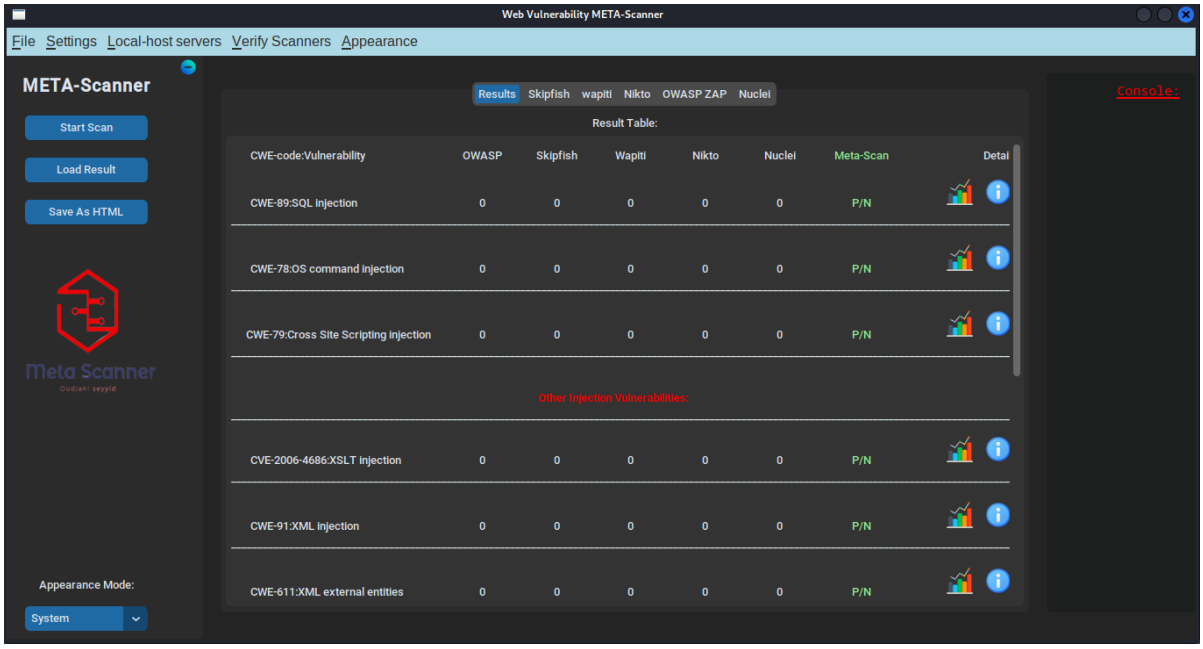

FIGURE 4.6: Main view of the meta-scan tool.

Prior launching a scan on a target web application, users can modify the configuration settings of each base scanner. This can be done by accessing the *Settings* menu and adjusting the values of the relevant parameters for each base scanner, as illustrated in Figure 4.7.

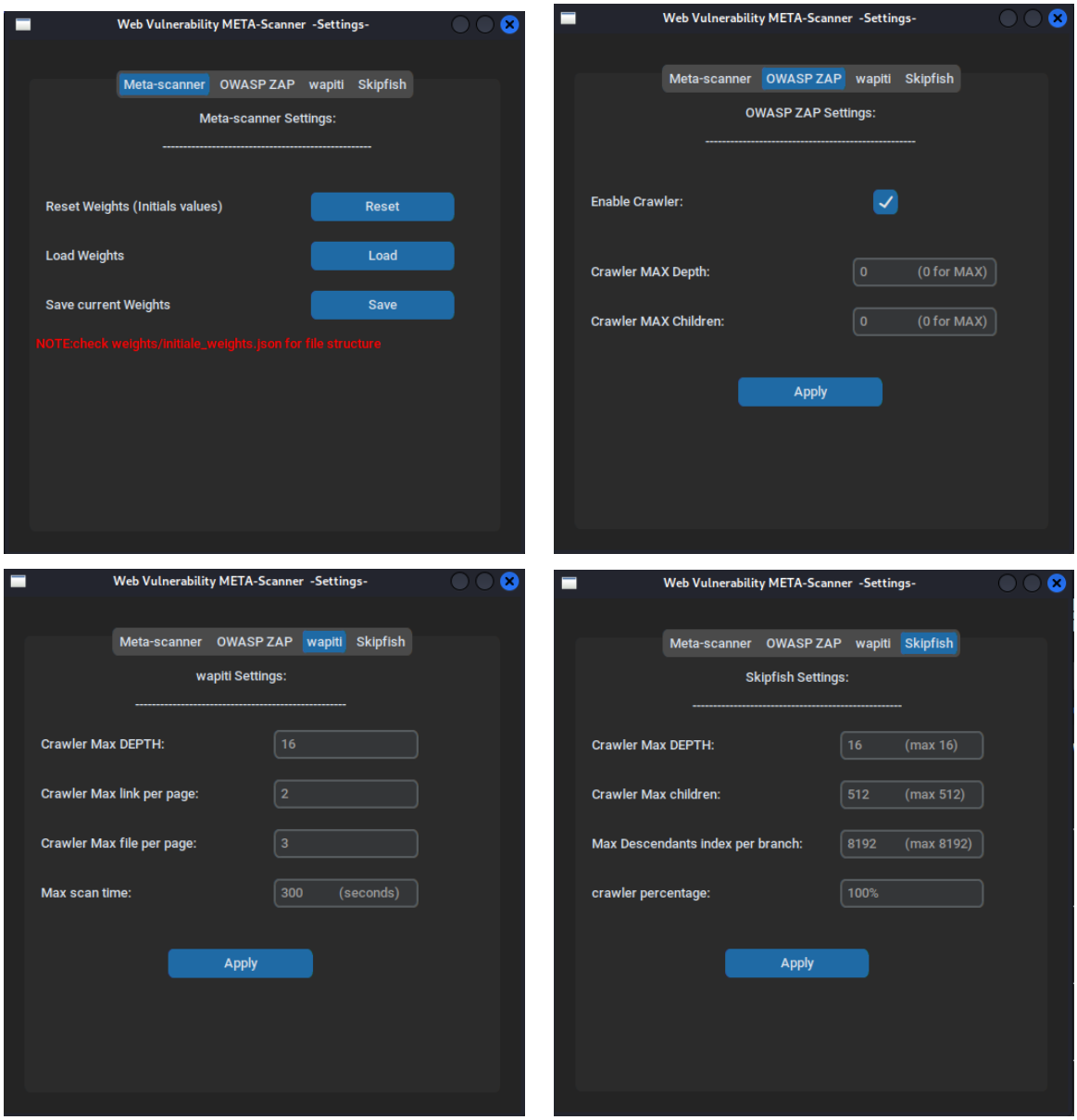

FIGURE 4.7: User-interface for configuring base scanners.

Subsequently, users may initiate a scan either by clicking the *start* button or selecting the corresponding option from the menu. To initiate a scan, users are required

to provide the target URL and designate a name for the target, as exemplified on the left-hand side of Figure 4.8. Upon initiating the scan by pressing the *start scanning* button, the scanning process can be tracked in the console located on the right-hand side of the main view. Once the scans have finished, a message is displayed, and the scanning report is displayed on the table located at the center of the main view.

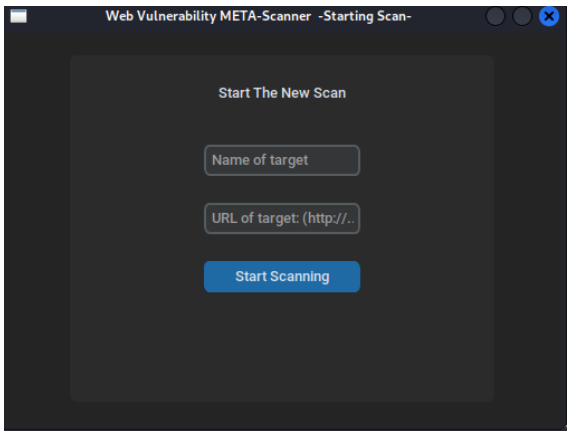

FIGURE 4.8: User-interface for launching a scan.

For more detailed information about each vulnerability, users can click on the *more info* button associated with the respective vulnerability. This triggers an integrated window positioned at the bottom of the main window, presenting details about the vulnerability such as successful attack instances, used parameters, and additional relevant information as shown in Figure 4.9.

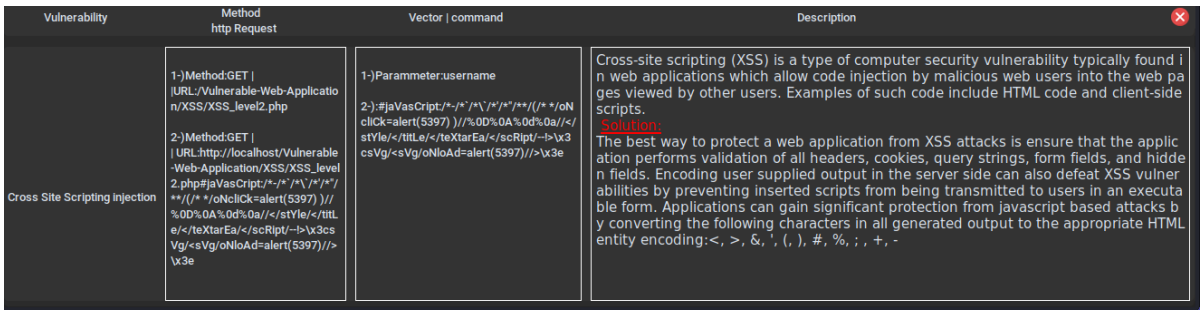

FIGURE 4.9: Visualizing details about a particular vulnerability.

In order to provide users with a more comprehensive view of the meta-scan system, an additional feature allows them to generate a graph depicting the weights

assigned to each scanner in relation to each vulnerability (see Figure 4.10). To access this graph, users can simply click on the *graph* icon located next to the *more info* button.

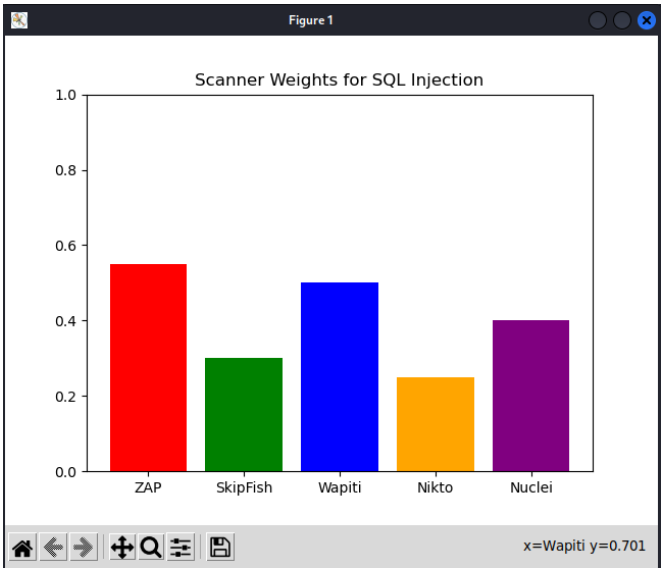

FIGURE 4.10: Visualizing base scanner weights for a particular vulnerability.

4.7 Installation requirements

Given that the meta-scan system integrates multiple independently developed scanners, it is essential to have a robust platform that can accommodate all the necessary tools in a cohesive manner. While various Linux distributions can fulfill this requirement, we chose the Kali Linux distribution due to its exceptional adaptability and flexibility in the field of cybersecurity [68]. Kali Linux is specifically designed to cater to the needs of penetration testing, digital forensics, and ethical hacking, making it an ideal choice for the development and hosting of the proposed meta-scan system. To facilitate the seamless integration of the different scanners and develop an intuitive user interface, we opted to use the Python programming language. Python offers a wide range of libraries, providing extensive support for various functionalities and enabling efficient development [88]. Leveraging Python's versatility, we can effectively combine the different scanners and create a unified experience for users.

4.7.1 Kali Linux

Kali Linux [89], the chosen operating system, is widely regarded as the most advanced penetration testing distribution. It is an open-source, Debian-based Linux distribution meticulously tailored for diverse information security tasks. Previously known as BackTrack Linux, Kali Linux encompasses a comprehensive set of common tools, configurations, and automations that streamline the user's focus on the task at hand rather than the surrounding activities. Notably, Kali Linux is equipped with industry-specific modifications and incorporates over six hundred tools specifically designed for a range of information security endeavors. These encompass penetration testing and vulnerability management. Kali Linux is freely accessible to both information security professionals and enthusiasts alike. It boasts an extensive feature set, including a vast arsenal of over 600 penetration testing tools, cost-free availability, an open-source Git tree, a custom kernel optimized for injection capabilities, development in a secure environment, support for multiple languages, complete customization options, and compatibility with ARMEL and ARMHF architectures [68].

4.7.2 Languages & Libraries

To meet the meta-system development requirements, the Python programming language is opted to serve as a foundation for the implementation. Additionally, specific libraries play a crucial role in enabling various functionalities.

1. **Python v3.10:** Python [90] is a dynamically-typed, high-level programming language that uses an interpreted approach. With its object-oriented nature and dynamic semantics, Python offers a compelling platform for efficient rapid application development. It boasts a rich set of built-in data structures and supports dynamic typing and binding, enhancing the flexibility and productivity of developers [88]. Python 3.10 was released on October 4th, 2021. It has new syntax features, new standard libraries, and interpreter improvements.

2. **Tkinter & CustomTkinter:** Tkinter [91] is a standard Python library that provides a GUI (Graphical User Interface) framework for creating and developing desktop applications, while CustomTkinter [92] is an extended library from Tkinter developed to customize the GUI for more adaptable and nicer designs.

3. **OS & Threading:** The Python libraries *OS* and *Threading* are essential components used by the meta-scan system for seamless integration of scanners and efficient execution of security assessments. The *OS* library provides a standardized interface for interacting with the operating system (Kali Linux in this case), enabling the execution of specific OS commands required for each scanner's integration. This ensures a smooth coordination of scanner functionalities within the meta-scan system. On the other hand, the *Threading* library of Python enables the concurrent execution of multiple tasks or processes. Using the multithreading technique, the meta-scan system enables the simultaneous execution of scanner operations. This parallel execution significantly enhances the speed and efficiency of the vulnerability detection process.

4. **JSON (JavaScript Object Notation):** The Python library JSON serves as a fundamental tool in the meta-scan system for handling files in JSON format. This library is essential for reading and manipulating JSON files, which are commonly used for storing and exchanging data. In the context of the meta-scan system, the JSON library is specifically utilized for two purposes. Firstly, it enables the system to read scanner reports provided in JSON format. By leveraging the capabilities of the JSON library, the meta-scan system can extract relevant information from these reports and incorporate it into its analysis and decision-making processes. Secondly, the JSON library is used to save the final report generated by the meta-scan system. After the completion of the security assessment, the system aggregates all the relevant vulnerability information, meta-scan decisions, and other pertinent details into a comprehensive report. This report

is then saved in JSON format using the JSON library, ensuring a standardized representation and easy accessibility of the final results.

5. **Jinja2 & Webbrowser:** Jinja2 [93] is a widely adopted templating engine for Python developers, offering powerful capabilities for creating dynamic templates. It enables the creation of templates that incorporate placeholders, referred to as variables, as well as control structures like loops and conditionals. In the context of the meta-scan system, we leveraged Jinja2 to generate the final report in HTML file format. Using Jinja2's features, we were able to create a template that efficiently incorporates the relevant data and presents it in a well-structured and visually appealing manner. Additionally, we made use of the Python library Webbrowser [94]. This library provides functionality to interact with web browsers, allowing us to open and display the generated report in a web browser. By utilizing the capabilities of the Webbrowser library, we were able to present the report seamlessly in a user-friendly and accessible format. Furthermore, this library can also be used to navigate to other web pages, such as target URLs, providing additional flexibility and functionality within the meta-scan system.

6. **Matplotlib:** Matplotlib [95] is a versatile and extensive library in Python that facilitates the creation of static, animated, and interactive visualizations. For the meat-scan system, we used the Matplotlib library to generate graphical representations, specifically graphs, pertaining to the Weight matrix for each vulnerability.

4.7.3 Installation process

One of the critical steps and requirements in the meta-scan system is the installation of the selected scanners. It is of utmost importance to ensure that these scanners are not only properly installed but also kept up-to-date. Keeping the scanners updated is vital for achieving optimal results. Regular updates ensure that the scanners have

the latest vulnerability signatures, detection techniques, and bug fixes. By staying current with the latest versions, the scanners are better equipped to identify vulnerabilities accurately and provide more reliable results. Therefore, diligent attention to the installation and maintenance of the selected scanners is essential for maximizing the effectiveness of the meta-scan system.

4.7.3.1 OWASP ZAP

ZAP provides installers for Windows, Linux, and macOS operating systems. Additionally, Docker images are available for download from the official ZAP website. As our thesis focuses on the usage of the Kali Linux operating system, all installation instructions provided herein are specific to the Linux OS. Note that ZAP requires Java version 11 or higher in order to run properly. To install ZAP on Kali Linux, there are two recommended methods. The first approach is to download the desired installer from the official ZAP website [56] and follow the installation instructions while accepting the user terms and policies. Alternatively, you can execute the following command in the Kali Linux terminal:

sudo apt install zaproxy

4.7.3.2 Wapiti

Wapiti does not have a graphical user interface (GUI) but instead operates in command-line mode. It offers a straightforward command structure that facilitates ease of use and efficient scanning. Wapiti is compatible with multiple operating systems including Linux, Windows, and macOS. To install Wapiti, there are two recommended methods available. The first approach is to download the latest stable version of Wapiti from the official repository [57] and follow the provided instructions to complete the installation process. Alternatively, you can execute the following command in the Kali Linux terminal:

sudo apt install wapiti

4.7.3.3 SkipFish

To install Skipfish, begin by downloading the latest stable version from the Skipfish repository [52]. Make sure that the hosted system has the required dependencies, including OpenSSL, libidn, zlib, and libpcre, installed beforehand. Extract the downloaded archives to a suitable location on the system. Navigate to the extracted Skipfish directory and compile the program by running the command *make*. Once the compilation process is finished, you can install Skipfish by executing *sudo make install*. If the Kali Linux distribution is used, Skipfish is already supported. Simply open the terminal or PowerShell and run the following command to install Skipfish directly:

sudo apt install skipfish

4.7.3.4 Nikto

Note that Nikto operates solely in command-line mode and does not have a graphical user interface. To install Nikto, follow these steps. First, ensure that Perl 5 is installed on the system, as Nikto relies on it for execution. Install Perl 5 according to the instructions for the operating system. Nikto is compatible with Windows, Linux, and MacOS X. Once Perl 5 is installed, proceed with the Nikto installation. Clone the official Nikto repository using the command *git clone https://github.com/sullo/nikto*. Navigate to the "*nikto/program*" directory using the command "cd nikto/program". If you are using the Kali Linux distribution, installation is simpler. Run the following command in the terminal or PowerShell:

sudo apt install nikto

4.7.3.5 Nuclei

Nuclei relies on Go language Go1.19, so it is necessary to have it installed on the system before proceeding with the installation. Once Go1.19 is installed, one can proceed with the Nuclei installation by downloading and compiling the latest stable version using the command:

go install -v github.com/projectdiscovery/nuclei/v2/cmd/nuclei@latest

For users of the Kali Linux distribution, installing Nuclei is straightforward. Simply run the following command in the terminal or PowerShell:

sudo apt install nuclei

4.7.3.6 Meta-scan system

To set up the meta-scan system, begin by installing the individual scanners. Once the scanners are installed, one can proceed to install the meta-scan system. The latest stable version of the meta-scan system can be obtained from the repository located at https://github.com/OSTEsayed/OSTE-Meta-Scan. The installation can be performed manually using the following steps:

1. Download the meta-scanner: **git clone** *https://github.com/OSTEsayed/oste-meta-scan*
2. Navigate to the directory: **cd** *downladed_path/Meta-scan*
3. Install required libraries: **sudo pip install** *requirements.txt*
4. Launch the meta-scanner: **python3** Meta-scan.py

4.8 Conclusion

This chapter provides a detailed account of the development and implementation process of the proposed meta-scan system, building upon the design scheme discussed in Chapter 3. Our experimentation and evaluation approach involved carefully selecting five dynamic web application scanners from a pool of over 18 options. We examined various aspects of each scanner, including their configuration, scanning techniques, and generated reports. These reports were consolidated, combined, and analyzed to create a unified and comprehensible report. To enhance the effectiveness of the meta-scan system, we conducted parameter tuning specifically for three common injection vulnerabilities: XSS, SQL, and OS command. This involved testing on four known vulnerable applications and using the grid search method to identify optimal values for the reward rate, penalty factor, and thresholds. These parameter values were determined based on their ability to produce the most accurate results during the learning phase. Subsequently, we performed a comparative analysis between each individual scanner and our meta-scan system. The results were highly promising, as we successfully reduced false positives for SQL injections and improved the accuracy and true positives for both XSS and OS command injections. However, it should be noted that further experimentation and data collection using additional web applications are necessary to thoroughly evaluate the effectiveness of the meta-scan system. Furthermore, we developed a user-friendly graphical interface for the meta-scan system, allowing users to easily experiment with the system using online or locally hosted web applications. This interface enhances the usability and accessibility of the meta-scan system for a wider range of users.

CONCLUSION

In this research project, we designed and developed a meta-scan system that integrates multiple dynamic vulnerability scanners to enhance accuracy and minimize false positives and false negatives in detecting injection vulnerabilities. The obtained results were promissing; the proposed system surpasses the performance of individual scanners; in the worst-case, the meta-scan system performs at least as well as the best individual scanner integrated in the system. This supports the hypothesis that using multiple scanners can improve the detection accuracy. Furthermore, the meta-scan tool features a user-friendly interface and generates a comprehensive scanning report, providing valuable insights into the vulnerabilities detected. The proposed meta-scan system showed effectiveness in detecting XSS vulnerabilities and OS command injections, but it exhibited limitations in identifying SQL injections. To enhance the execution efficiency, we explored the use of a multithreaded system, which proved advantageous compared to sequential scanning and mitigated potential time-consuming issues.

However, some drawbacks were identified upon closer examination of the proposed system. Firstly, the system was constrained by the limited availability of vulnerable pages for testing, potentially leading to incomplete vulnerability detection. Secondly, the study was limited by the absence of applications with other injection

vulnerabilities, such as CRLF, template, code, XML-based, and HTTP header injections. This limited diversity restricts the system's generalizability and its ability to address a comprehensive range of injection vulnerabilities. It is worth noting that the proposed meta-scan system has the potential for generalization beyond injection vulnerabilities. The underlying principles and methodologies can be applied to address other types of vulnerabilities, opening avenues for future research in addressing a broader range of security vulnerabilities. Thirdly, the selected scanners had limitations in detecting vulnerabilities in web applications with dynamic generation, such as OWASP Juicy Shop [58]. The dynamic nature of these applications, with hidden input points through JavaScript or other dynamic states, often requires human interaction for effective detection. This limitation should be taken into account when using the meta-scan system or individual scanners. Finally, although a multi-threading approach was used, the meta-scanner still encounters a time-consumption problem. To address this issue, a potential solution is to adopt a cloud-based solution, which is left as a future work. This way, base scanners can be deployed across multiple and separate nodes and executed concurrently, enabling true parallelism. The meta-scan node would then focus only on consolidating the data and making informed decisions based on the collected information. The cloud-based solution has the potential to significantly improve the efficiency and performance of the meta-scanner.

BIBLIOGRAPHY

[1] José Fonseca, Marco Vieira, and Henrique Madeira. "Vulnerability & attack injection for web applications". In: *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*. IEEE. 2009, pp. 93–102.

[2] *Annual number of malware attacks worldwide from 2015 to 2022 (in billions) Statista Foundation*. https://www.statista.com/statistics/873097/malware-attacks-per-year-worldwide/. Accessed: 2023-05-14.

[3] Melina Kulenovic and Dzenana Donko. "A survey of static code analysis methods for security vulnerabilities detection". In: *2014 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE. 2014, pp. 1381–1386.

[4] Jason Bau et al. "State of the art: Automated black-box web application vulnerability testing". In: *2010 IEEE symposium on security and privacy*. IEEE. 2010, pp. 332–345.

[5] Adam Doupé, Marco Cova, and Giovanni Vigna. "Why Johnny can't pentest: An analysis of black-box web vulnerability scanners". In: *Detection of Intrusions and Malware, and Vulnerability Assessment: 7th International Conference, DIMVA 2010, Bonn, Germany, July 8-9, 2010. Proceedings 7*. Springer. 2010, pp. 111–131.

[6] Jeroen Van Der Ham. "Toward a better understanding of "Cybersecurity"". In: *Digital Threats: Research and Practice* 2.3 (2021), pp. 1–3. DOI: 10.1145/3442445.

[7] F. Flammini, R. Setola, and G. Franceschetti. *Effective Surveillance for Homeland Security: Balancing Technology and Social Issues*. Multimedia Computing, Communication and Intelligence. Taylor & Francis, 2013.

[8] Nuno Antunes and Marco Vieira. "Enhancing Penetration Testing with Attack Signatures and Interface Monitoring for the Detection of Injection Vulnerabilities in Web Services". In: *2011 IEEE International Conference on Services Computing*. 2011, pp. 104–111. DOI: 10.1109/SCC.2011.67.

[9] *What Are Injection Attacks Ian Muscat*. https://www.acunetix.com/blog/articles/injection-attacks/. Accessed: 2023-02-15.

[10] Josh Pauli. *The basics of web hacking: tools and techniques to attack the web*. Elsevier, 2013.

[11] Justin Clarke. *SQL injection attacks and defense*. Elsevier, 2009.

[12] MILLER Richie. *Practical Programming 6 in 1;Python Machine Learning Javascript React 17 and Angular with Typescript*. Retrieved: 2023-02-19. PASTOR PUBLISHING LTD, 2023.

[13] Abdelhakim Hannousse and Salima Yahiouche. "Handling webshell attacks: A systematic mapping and survey". In: *Comput. Secur.* 108 (2021), p. 102366. DOI: 10.1016/j.cose.2021.102366.

[14] Abdelhakim Hannousse, Mohamed Cherif Nait-Hamoud, and Salima Yahiouche. "A deep learner model for multi-language webshell detection". In: *Int. J. Inf. Sec.* 22.1 (2023), pp. 47–61. DOI: 10.1007/s10207-022-00615-5.

[15] John Viega Andy Oram. *Beautiful Security: Leading Security Experts Explain How They Think*. 2009.

[16] *OWASP Top Ten OWASP Foundation*. https://owasp.org/www-project-top-ten/. Accessed: 2023-03-10.

[17] *Distribution of web application critical vulnerabilities worldwide as of 2022 Statista Foundation*. https://www.statista.com/statistics/806081/worldwide-application-vulnerability-taxonomy/. Accessed: 2023-03-15.

[18] Hsiu-Chuan Huang et al. "Web application security: threats, countermeasures, and pitfalls". In: *Computer* 50.6 (2017), pp. 81–85. DOI: 10.1109/MC.2017.183.

[19] R. Gupta. *Hands-on Penetration Testing for Web Applications: Run Web Security Testing on Modern Applications Using Nmap, Burp Suite and Wireshark (English Edition)*. BPB Publications, 2021.

[20] Paco Hope and Ben Walther. *Web security testing cookbook: systematic techniques to find problems fast*. " O'Reilly Media, Inc.", 2008.

[21] Peter Yaworski. *Real-world bug hunting: a field guide to web hacking*. No Starch Press, 2019.

[22] Anne Edmundson et al. "An empirical study on the effectiveness of security code review". In: *Engineering Secure Software and Systems: 5th International Symposium, ESSoS 2013, Paris, France, February 27-March 1, 2013. Proceedings 5*. Springer. 2013, pp. 197–212. DOI: 10.1007/978-3-642-36563-8_14.

[23] David Binkley. "Source code analysis: A road map". In: *Future of Software Engineering (FOSE'07)* (2007), pp. 104–119. DOI: 10.1109/FOSE.2007.27.

[24] Michael Felderer et al. "Security testing: A survey". In: *Advances in Computers*. Vol. 101. Elsevier, 2016, pp. 1–51.

[25] Abdelhakim Hannousse, Salima Yahiouche, and Mohamed Cherif Nait-Hamoud. *Twenty-two years since revealing cross-site scripting attacks: a systematic mapping and a comprehensive survey*. 2022. arXiv: 2205.08425 [cs.CR].

[26] Jan Svoboda. "Effectively Combining Static Code Analysis and Manual Code Reviews". MA thesis. Masaryk University, 2014.

[27] Nico L. de Poel. "Automated security review of PHP web applications with static code analysis". MA thesis. University of Groningen, 2010.

[28] V Benjamin Livshits and Monica S Lam. "Finding Security Vulnerabilities in Java Applications with Static Analysis." In: *USENIX security symposium*. Vol. 14. 2005, pp. 18–18.

[29] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. "Pixy: A static analysis tool for detecting web application vulnerabilities". In: *2006 IEEE Symposium on Security and Privacy (SP'06)*. IEEE. 2006, 6–pp.

[30] William GJ Halfond, Shauvik Roy Choudhary, and Alessandro Orso. "Improving penetration testing through static and dynamic analysis". In: *Software Testing, Verification and Reliability* 21.3 (2011), pp. 195–214. DOI: 10.1002/stvr.450.

[31] Zoran Đurić. "WAPTT-Web application penetration testing tool". In: *Advances in Electrical and Computer Engineering* 14.1 (2014), pp. 93–102.

[32] Indraneel Mukhopadhyay, S Goswami, and E Mandal. "Web penetration testing using nessus and metasploit tool". In: *IOSR J. Comput. Eng* 16.3 (2014), pp. 126–129.

[33] Nisal Madhushan Vithanage and Neera Jeyamohan. "WebGuardia - an integrated penetration testing system to detect web application vulnerabilities". In: *2016 International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET)*. 2016, pp. 221–227. DOI: 10.1109/WiSPNET.2016.7566124.

[34] James Clause, Wanchun Li, and Alessandro Orso. "Dytan: a generic dynamic taint analysis framework". In: *Proceedings of the 2007 international symposium on Software testing and analysis*. 2007, pp. 196–206.

[35] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)". In: *2010 IEEE symposium on Security and privacy*. IEEE. 2010, pp. 317–331.

[36] Min Gyung Kang et al. "Dta++: dynamic taint analysis with targeted control-flow propagation." In: *NDSS*. 2011.

[37] Malte Mues, Till Schallau, and Falk Howar. "Jaint: a framework for user-defined dynamic taint-analyses based on dynamic symbolic execution of java programs". In: *Integrated Formal Methods: 16th International Conference, IFM 2020, Lugano, Switzerland, November 16–20, 2020, Proceedings 16*. Springer. 2020, pp. 123–140.

[38] Ivan Andrianto, MM Inggriani Liem, and Yudistira Dwi Wardhana Asnar. "Web application fuzz testing". In: *2017 International Conference on Data and Software Engineering (ICoDSE)*. IEEE. 2017, pp. 1–6.

[39] Rune Hammersland and Einar Snekkenes. "Fuzz testing of web applications". In: *https://www.semanticscholar.org/paper/Fuzz-testing-of-web-applications-HammerslandSnekkenes* (2008). DOI: 10.1109/ICODSE.2017.8285893.

[40] Tao Guo et al. "Gramfuzz: Fuzzing testing of web browsers based on grammar analysis and structural mutation". In: *2013 Second International Conference on Informatics and Applications (ICIA)*. IEEE. 2013, pp. 212–215.

[41] Wang Chunlei, Liu Li, and Liu Qiang. "Automatic fuzz testing of web service vulnerability". In: (2014). DOI: 10.1049/cp.2014.0589.

[42] Ahmad Ghafarian. "A hybrid method for detection and prevention of SQL injection attacks". In: *2017 Computing Conference*. IEEE. 2017, pp. 833–838.

[43] Bharti Nagpal, Naresh Chauhan, and Nanhay Singh. "A survey on the detection of SQL injection attacks and their countermeasures". In: *Journal of Information Processing Systems* 13.4 (2017), pp. 689–702.

[44] Jingling Zhao and Rulin Gong. "A New Framework of Security Vulnerabilities Detection in PHP Web Application". In: *2015 9th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*. 2015, pp. 271–276. DOI: 10.1109/IMIS.2015.42.

[45] Kunal Sadalkar, Radhesh Mohandas, and Alwyn R Pais. "Model based hybrid approach to prevent SQL injection attacks in PHP". In: *Security Aspects in Information Technology: First International Conference, InfoSecHiComNet 2011, Haldia, India, October 19-22, 2011. Proceedings*. Springer. 2011, pp. 3–15. DOI: 10.1007/978-3-642-24586-2_3.

[46] Yong Fang et al. "TAP: A static analysis model for PHP vulnerabilities based on token and deep learning technology". In: *PloS one* 14.11 (2019), e0225196. DOI: 10.1371/journal.pone.0225196.

[47] Matija Cankar et al. "Security in DevSecOps: Applying Tools and Machine Learning to Verification and Monitoring Steps". In: *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering*. 2023, pp. 201–205.

[48] Muhammad Noman Khalid et al. "Predicting web vulnerabilities in web applications based on machine learning". In: *Intelligent Technologies and Applications: First International Conference, INTAP 2018, Bahawalpur, Pakistan, October 23-25, 2018, Revised Selected Papers 1*. Springer. 2019, pp. 473–484. DOI: 10.1007/978-981-13-6052-7_41.

[49] Ana Fidalgo et al. "Towards a deep learning model for vulnerability detection on web application variants". In: *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE. 2020, pp. 465–476.

[50] Anthony Dessiatnikoff et al. "A clustering approach for web vulnerabilities detection". In: *2011 IEEE 17th Pacific Rim International Symposium on Dependable Computing*. IEEE. 2011, pp. 194–203.

[51] *W3af Scanner W3af*. https://w3af.org/. Accessed: 2023-01-21.

[52] *SkipFish kali tools*. https://www.kali.org/tools/skipfish. Accessed: 2023-01-21.

[53] Maha Alghawazi, Daniyal Alghazzawi, and Suaad Alarifi. "Detection of sql injection attack using machine learning techniques: a systematic literature review". In: *Journal of Cybersecurity and Privacy* 2.4 (2022), pp. 764–777. DOI: 10.3390/jcp2040039.

[54] Muzun Althunayyan et al. "Evaluation of black-box web application security scanners in detecting injection vulnerabilities". In: *Electronics* 11.13 (2022), p. 2049. DOI: 10.3390/electronics11132049.

[55] *Burp Suite Professional portswigger foundation*. https://portswigger.net/burp/pro. Accessed: 2023-01-21.

[56] *OWASP Zap OWASP foundation*. https://www.zaproxy.org/. Accessed: 2023-01-21.

[57] *Wapiti Github platform*. https://wapiti-scanner.github.io/. Accessed: 2023-01-21.

[58] *OWASP Juicy Shop project OWASP foundation*. https://owasp.org/www-project-juice-shop/. Accessed: 2023-01-21.

[59] Balume Mburano and Weisheng Si. "Evaluation of web vulnerability scanners based on owasp benchmark". In: *2018 26th International Conference on Systems Engineering (ICSEng)*. IEEE. 2018, pp. 1–6.

[60] *Arachni Github platform*. https://github.com/Arachni/arachni. Accessed: 2023-01-21.

[61] Malik Qasaimeh, A Shamlawi, and Tariq Khairallah. "Black box evaluation of web application scanners: Standards mapping approach". In: *Journal of Theoretical and Applied Information Technology* 96.14 (2018), pp. 4584–4596.

[62] *NetSparker invicti foundation*. https://www.invicti.com/. Accessed: 2023-01-21.

[63] *Nessus tenable platform*. https://www.tenable.com/. Accessed: 2023-01-21.

[64] *Acunetix Acunetix foundation*. https://www.acunetix.com/. Accessed: 2023-01-21.

[65] M Shanthi and A Anthony Irudhayaraj. "Multithreading-an efficient technique for enhancing application performance". In: *International Journal of Recent Trends in Engineering* 2.4 (2009), p. 165.

[66] Bob Martin. "Common Vulnerabilities Enumeration (CVE), Common Weakness Enumeration (CWE), and Common Quality Enumeration (CQE) Attempting to systematically catalog the safety and security challenges for modern, networked, software-intensive systems". In: *ACM SIGAda Ada Letters* 38.2 (2019), pp. 9–42. DOI: 10.1145/3375408.3375410.

[67] Felipe Pezoa et al. "Foundations of JSON schema". In: *Proceedings of the 25th international conference on World Wide Web*. 2016, pp. 263–273.

[68] Lee Allen, Tedi Heriyanto, and Shakeel Ali. *Kali Linux–Assuring security by penetration testing*. Packt Publishing Ltd, 2014.

[69] *Vega Scanner Subgraph foundation*. https://subgraph.com/vega/. Accessed: 2023-01-21.

[70] *Ronin-Vulns Github platform*. https://github.com/ronin-rb/ronin-vulns. Accessed: 2023-01-21.

[71] *Vulmon Vulmap Github platform*. https://github.com/vulmon/Vulmap. Accessed: 2023-01-21.

[72] *Shenril Sitadel Github platform*. https://github.com/shenril/Sitadel. Accessed: 2023-01-21.

[73] *OpenVas openvas*. https://openvas.org. Accessed: 2023-01-21.

[74] *Ratproxy sectools*. https://sectools.org/tool/ratproxy/. Accessed: 2023-01-21.

[75] *Astra Getastra*. https://www.getastra.com/. Accessed: 2023-01-21.

[76] *Vulscanpro Github platform*. https://github.com/thenurhabib/vulscanpro. Accessed: 2023-01-21.

[77] *Nikto Github platform*. https://github.com/sullo/nikto. Accessed: 2023-01-21.

[78] *Nuclei projectdiscovery foundation*. https://github.com/projectdiscovery/nuclei. Accessed: 2023-01-21.

[79] Suliman Alazmi and Daniel Conte de Leon. "Customizing OWASP ZAP: A Proven Method for Detecting SQL Injection Vulnerabilities". In: *2023 IEEE 9th Intl Conference on Big Data Security on Cloud (BigDataSecurity), IEEE Intl Conference on High Performance and Smart Computing,(HPSC) and IEEE Intl Conference on Intelligent Data and Security (IDS)*. IEEE. 2023, pp. 102–106.

[80] Aleksandra Kondraciuk, Aleksandra Bartos, and Beata Pańczyk. "Comparative analysis of the effectiveness of OWASP ZAP, Burp Suite, Nikto and Skipfish in testing the security of web applications". In: *Journal of Computer Sciences Institute* 24 (2022), pp. 176–180.

[81] Nikita Karangle, Alekha Kumar Mishra, and Danish Ali Khan. "Comparison of Nikto and Uniscan for measuring URL vulnerability". In: *2019 10th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*. IEEE. 2019, pp. 1–6. DOI: 10.1109/ICCCNT45670.2019.8944463.

[82] *OWASP mutillidae2 OWASP foundation*. https://github.com/webpwnized/mutillidae. Accessed: 2023-01-21.

[83] *OWASP Vulnerable web application OWASP foundation*. https://github.com/OWASP/Vulnerable-Web-Application. Accessed: 2023-01-21.

[84] *Xtrem Vulnerable web application Github platform*. https://github.com/s4n7h0/xvwa. Accessed: 2023-01-21.

[85] *DAMN Vulnerable web application Github platform*. https://github.com/digininja/DVWA. Accessed: 2023-01-21.

[86] *OWASP Vulnerable web application directory OWASP foundation*. https://github.com/OWASP/OWASP-VWAD. Accessed: 2023-01-21.

[87] Álvaro Barbero Jiménez, Jorge López Lázaro, and José R Dorronsoro. "Finding optimal model parameters by discrete grid search". In: *Innovations in Hybrid Intelligent Systems* (2007), pp. 120–127. DOI: /10.1007/978-3-540-74972-1_17.

[88] Nimit Thaker and Abhilash Shukla. "Python as multi paradigm programming language". In: *International Journal of Computer Applications* 177.31 (2020), pp. 38–42. DOI: 10.5120/ijca2020919775.

[89] *Kali Linux Kali operating system platform*. https://www.kali.org/. Accessed: 2023-01-21.

[90] *Paython Python language platform*. https://www.python.org/. Accessed: 2023-01-21.

[91] *Tkinter Python liberary platform*. https://docs.python.org/3/library/tk.html. Accessed: 2023-01-21.

[92] *Custom Tkinter Github platform*. https://github.com/TomSchimansky/CustomTkinter. Accessed: 2023-01-21.

[93] *Jinja2 jinja palletsproject*. https://jinja.palletsprojects.com/en/3.1.x/. Accessed: 2023-01-21.

[94] *Webbrowser Python liberary platform*. https://docs.python.org/3/library/webbrowser.html. Accessed: 2023-01-21.

[95] *Matplotlib matplotlib*. https://matplotlib.org/stable/index.html. Accessed: 2023-01-21.