Master

Developing an in house vulnerability scanner for detecting Template Injection, XSS, and DOM-XSS vulnerabilities

MARIUS HAUGER, STIG JENSEN

SUPERVISOR
Nadia Saad Noori

University of Agder, 2023
Faculty of Engineering and Science
Department of Engineering and Sciences

Obligatorisk gruppeerklæring

Den enkelte student er selv ansvarlig for å sette seg inn i hva som er lovlige hjelpemidler, retningslinjer for bruk av disse og regler om kildebruk. Erklæringen skal bevisstgjøre studentene på deres ansvar og hvilke konsekvenser fusk kan medføre. Manglende erklæring fritar ikke studentene fra sitt ansvar.

1.	Vi erklærer herved at vår besvarelse er vårt eget arbeid, og at vi ikke har brukt andre kilder eller har mottatt annen hjelp enn det som er nevnt i besvarelsen.	Ja
2.	**Vi erklærer videre at denne besvarelsen:** • Ikke har vært brukt til annen eksamen ved annen avdeling/universitet/høgskole innenlands eller utenlands. • Ikke refererer til andres arbeid uten at det er oppgitt. • Ikke refererer til eget tidligere arbeid uten at det er oppgitt. • Har alle referansene oppgitt i litteraturlisten. • Ikke er en kopi, duplikat eller avskrift av andres arbeid eller besvarelse.	Ja
3.	Vi er kjent med at brudd på ovennevnte er å betrakte som fusk og kan medføre annullering av eksamen og utestengelse fra universiteter og høgskoler i Norge, jf. Universitets- og høgskoleloven §§4-7 og 4-8 og Forskrift om eksamen §§ 31.	Ja
4.	Vi er kjent med at alle innleverte oppgaver kan bli plagiatkontrollert.	Ja
5.	Vi er kjent med at Universitetet i Agder vil behandle alle saker hvor det forligger mistanke om fusk etter høgskolens retningslinjer for behandling av saker om fusk.	Ja
6.	Vi har satt oss inn i regler og retningslinjer i bruk av kilder og referanser på biblioteket sine nettsider.	Ja
7.	Vi har i flertall blitt enige om at innsatsen innad i gruppen er merkbart forskjellig og ønsker dermed å vurderes individuelt. Ordinært vurderes alle deltakere i prosjektet samlet.	Nei

Publiseringsavtale

Fullmakt til elektronisk publisering av oppgaven Forfatter(ne) har opphavsrett til oppgaven. Det betyr blant annet enerett til å gjøre verket tilgjengelig for allmennheten (Åndsverkloven. §2).
Oppgaver som er unntatt offentlighet eller taushetsbelagt/konfidensiell vil ikke bli publisert.

Vi gir herved Universitetet i Agder en vederlagsfri rett til å gjøre oppgaven tilgjengelig for elektronisk publisering:	Ja
Er oppgaven båndlagt (konfidensiell)?	Nei
Er oppgaven unntatt offentlighet?	Nei

Acknowledgements

We would like to express our heartfelt appreciation to our supervisors, Nadia and William, for their invaluable guidance and support throughout our master's thesis.

Both Nadia and William's guidance and support have been vital to our academic growth and the successful completion of our thesis. We are fortunate to have had the privilege of working under their supervision. Our heartfelt appreciation goes out to them for their invaluable assistance throughout this journey.

Abstract

Web applications are becoming an essential part of today's digital world. However, with the increase in the usage of web applications, security threats have also become more prevalent. Cyber attackers can exploit vulnerabilities in web applications to steal sensitive information or take control of the system. To prevent these attacks, web application security must be given due consideration.

Existing vulnerability scanners fail to detect Template Injection, XSS, and DOM-XSS vulnerabilities effectively. To bridge this gap in web application security, a customized in-house scanner is needed to quickly and accurately identify these vulnerabilities, enhancing manual security assessments of web applications.

This thesis focused on developing a modular and extensible vulnerability scanner to detect Template Injection, XSS, and DOM-based XSS vulnerabilities in web applications. Testing the scanner against other free and open-source solutions on the market showed that it outperformed them on Template injection vulnerabilities and nearly all on XSS-type vulnerabilities. While the scanner has limitations, focusing on specific injection vulnerabilities can result in better performance.

Contents

List of Figures

List of Tables

Acronyms

CSTI	Client Side Template Injection. 5
DOM	Document Object Model. 1
ERB	Embedded Ruby. 7, 47
FN	False Negative. 9, 10
FP	False Positive. 9, 10, 48
RCE	Remote Code Execution. 7
SSTI	Server Side Template Injection. 47
TLD	Top-level Domain. 21
TP	True Positive. 9, 10, 48
XSS	Cross site scripting. iv, 5–7

Chapter 1

Introduction

1.1 Background

In today's digital age, the threat of exploited vulnerabilities is constantly increasing, and it has become essential for security professionals to assess the security of systems regularly[6]. However, performing these assessments can be time-consuming and require a considerable amount of resources to be done correctly. Consequently, conducting a system penetration test has become an expensive service that not all businesses can afford[44].

To overcome this challenge, the RedTeam at Netsecurity has initiated an in-house project to create a modular design vulnerability scanner that will increase the effectiveness of security assessments. By developing an automated vulnerability scanner, it will reduce the reliance on manual testing, which will save time and effort. Moreover, this scanner will be designed to be flexible, allowing it to be easily adapted to different environments.

1.2 Problem statement

The existing vulnerability scanning solutions in the market fail to meet the requirements for detecting Template Injection, XSS, and DOM-XSS vulnerabilities effectively. Tests conducted on current vulnerability scanners have revealed inadequate performance in identifying these specific vulnerabilities, leading to a significant gap in web application security assessment[1][2]. Consequently, there is an urgent need to develop an in-house vulnerability scanner that can be customized to efficiently and accurately detect Template Injection, XSS, and DOM-XSS vulnerabilities. This scanner should be capable of quickly identifying these simple types of vulnerabilities, thereby enabling more effective manual security assessments of web applications.

1.3 Objective

The main objective of this thesis is to develop an in house vulnerability scanner for detecting Template Injection, XSS, and DOM-XSS vulnerabilities. The scanner is designed to be a modular and extensible tool that can be used by security professionals to increase the effectiveness and efficiency of security assessments.

This scanner specifically aims to detect and report three types of web application vulnerabilities: Template Injection, Document Object Model (DOM)-based XSS, and XSS. These vulnerabilities are some of the most commonly exploited, making it essential to detect and patch them as soon as possible[12]. With this scanner, Netsecurity's RedTeam will be better equipped to proactively identify and address these vulnerabilities, enhancing its clients' security posture.

1.4 Sub-objectives

To achieve the main objective, a set of sub-objectives were collaboratively defined with Netsecurity. Each of these sub-objectives focuses on a distinct task, although some are interconnected. These sub-objectives were included to shape the thesis according to Netsecurity's requirements and also to provide guidance on the development of the vulnerability scanner.

- Conduct a literature review of existing vulnerability scanners and their capabilities to identify strengths and weaknesses that can inform the design and development of the proposed scanner.
- Design and develop a modular and extensible vulnerability scanner that can detect Template Injection vulnerabilities in web applications. The scanner should be easy to use and configurable, with the ability to run on different platforms.
- Compile a comprehensive payload library for each vulnerability type that the scanner is designed to detect. The payload library will be used to test the scanner and evaluate its effectiveness.
- Incorporate the ability to detect Cross-Site Scripting (XSS) vulnerabilities, including both reflected and stored XSS, as well as DOM-based XSS, into the scanner's functionality.
- Test the scanner's effectiveness in detecting Template Injection, XSS, and DOM-based XSS vulnerabilities, and compare the results to those of existing open-source scanners.

Overall, the objective of this thesis is to provide a reliable and efficient tool that can be used by security professionals to identify vulnerabilities in web applications. The proposed vulnerability scanner is intended to improve existing tools, focusing on detecting Template Injection vulnerabilities, which are often overlooked by other scanners.

1.5 Report outline

The thesis's outline is provided in the list below, along with a brief synopsis of each chapter.

2. **State of the Art** - This section provides an overview of the current state of vulnerability scanning, as well as an explanation of the technology and tools required to understand the subsequent sections.
3. **Work methodology** - Describes the approach used for the thesis.
4. **Method** - This text provides a description and explanation of the required tools and the process used for development.
5. **Design** - This section explains the current vulnerability scanner, including its functions and components. It also includes UML diagrams and flowcharts.
6. **Results** - Contains the results of our vulnerability scanner as well as a comparison to other open-source scanners.
7. **Discussion** - This discussion will cover the performance of testing applications, vulnerability scanner results, and the issue of testing bias.

8. **Conclusion** - This summarizes the findings of the thesis and evaluates how well they align with the original goals.

Chapter 2

State of the Art

This chapter will provide an overview of the state of the art in web application vulnerability detection, including existing techniques and tools, current trends, and challenges in the field. We will also discuss the importance of vulnerability detection in web applications and its impact on the security and reliability of web-based systems.

2.1 Web application vulnerabilities

Web applications are a critical component of various industries and services on the modern internet. However, with the increasing reliance on these applications, reported vulnerabilities have also increased significantly. While vulnerabilities such as SQL injection, cross-site scripting (XSS), and buffer overflows have been long-standing concerns, variations such as client-side template injection, DOM-based XSS, and server-side request forgery (SSRF) have become more prevalent and pose new challenges to web application security[12].

To address these vulnerabilities, effective methods to detect and prevent them are needed to ensure the security and reliability of web-based systems. Currently, many techniques and tools are available for detecting vulnerabilities in web applications, including manual and automated approaches[19]. Manual techniques like penetration testing and code reviews are time-consuming and resource-intensive, but they provide detailed information about the application's security posture. On the other hand, automated techniques, such as vulnerability scanners, are faster and more scalable, but they may lack the accuracy and depth of manual [21].

Despite the availability of various techniques and tools, detecting vulnerabilities in web applications remains a challenge due to the constantly evolving threat landscape[29]. Attackers are always looking for new ways to exploit vulnerabilities, and developers are always introducing new features and functionalities into web applications, which can introduce new vulnerabilities. To keep up with these challenges, researchers and practitioners are continuously developing new approaches and tools for web application vulnerability detection[43].

2.2 Web application architecture

Web applications have become an integral part of modern-day computing. They enable users to access web-based services and interact with web-based content through a web browser. The architecture of a web application refers to the structure and design of the application, including its components, interactions, and dependencies[47].

Web application architecture is typically divided into three main components: the client-side, server-side, and database. The client-side of a web application is responsible for rendering the user interface and providing the user with a means of interacting with the application. This component is typically implemented using HTML, CSS, and JavaScript and executed in the user's browser. The client-side communicates with the server-side using HTTP requests and responses[47].

2.2.1 Client-side

The client-side of a web application includes the user interface, which is usually implemented using HTML, CSS, and JavaScript. The client-side code is executed on the user's machine, and it communicates with the server-side of the web application using HTTP requests and responses[47].

2.2.2 Server-Side

The server-side of a web application includes the application logic and data processing. It is responsible for receiving client-side requests, processing them, and returning appropriate responses. Server-side applications are usually developed using programming languages such as PHP, Java, Python, and .NET[47].

2.2.3 Database

Web applications often use a database to store and manage data. Databases are used to store user information, transaction data, and application data. Popular databases include MySQL, Oracle, and Microsoft SQL Server[47].

2.3 Client side template injection

Client Side Template Injection (CSTI) is a critical security vulnerability that can occur when web applications use client-side templating engines to render HTML content based on user input without adequate validation and sanitization. This type of vulnerability can have serious consequences, including Cross site scripting (XSS) attacks, information disclosure, or other types of malicious actions[3].

CSTI attacks involve injecting malicious code into web applications by exploiting vulnerabilities in the templating engine or using specially crafted input that bypasses input validation and sanitization checks. Once injected, the malicious code can execute in the context of the victim's browser, potentially leading to various types of attacks.

CSTI vulnerabilities are particularly challenging to prevent and detect, as client-side templating engines execute within the browser, making it easier for attackers to manipulate the rendered HTML. To mitigate the risk of CSTI attacks, it is essential to use trusted templating engines and to avoid using user input in templates as much as possible[3].

2.3.1 Template engines

Template engines are a crucial part of web application development, allowing for the dynamic generation of content based on user input and other data sources. However, the increased reliance on template engines has also resulted in several security vulnerabilities, such as client-side template injection, if not adequately secured. These security concerns pose a significant challenge for web developers and security professionals.

One of the critical functions of a template engine is to provide a framework for rendering web pages dynamically. This framework often includes placeholders or variables in the template that are replaced with server or user input data. While this approach provides flexibility and ease of use for developers, it also introduces potential security vulnerabilities.

For example, an attacker can exploit template engines' vulnerabilities by injecting malicious code into the web application by bypassing input validation and sanitization checks. Once injected, the malicious code can execute in the context of the victim's browser, potentially leading to XSS attacks, information disclosure, or other types of malicious actions.

Furthermore, to ensure the secure use of template engines, developers should follow best practices, such as using well-defined input validation and sanitization routines, avoiding user input in templates, and keeping template files separate from executable code. By adhering to these practices, developers can reduce the likelihood of introducing vulnerabilities to their web [15].

AngularJS:

AngularJS is a JavaScript-based open-source front-end web application framework primarily maintained by Google and by a community of individual developers and corporations. It was released in 2010 as a means of making complex, single-page web applications easier to build and maintain[35].

AngularJS extends HTML vocabulary for web applications by creating custom HTML elements and attributes that can be utilized in a declarative manner. This allows for a more intuitive and less imperative way of developing web applications compared to traditional JavaScript development.

One of the key features of AngularJS is its two-way data binding capability, which allows for automatic synchronization of the model and view components. This results in a more streamlined development process and reduced amount of boilerplate code.

AngularJS also provides a comprehensive set of reusable components and services, including directives for manipulating the DOM, services for making HTTP requests, and pipes for transforming data. These components and services can be easily shared across different parts of an application, leading to a more modular and maintainable code-base[35].

In addition to its features, AngularJS also has a large and supportive community, with many resources available for learning and problem-solving. Its popularity has led to integration with other popular front-end technologies such as TypeScript, allowing for type safety and improved developer experience.

VueJS

Vue.js is an open-source JavaScript framework for building user interfaces and single-page applications. It was created in 2014 by Evan You and has since gained a large following in the web development community due to its simplicity, performance, and versatility.

Vue.js operates on a component-based architecture, where components are modular and reusable pieces of UI that can be easily composed to create complex user interfaces. This allows for a more modular and maintainable codebase compared to traditional JavaScript development.

One of the key features of Vue.js is its reactivity system, which allows for automatic updates to the UI when the underlying data changes[35]. This is achieved through a simple and intuitive syntax for declaring reactive data, eliminating the need for manual DOM updates and reducing the potential for bugs.

Vue.js also provides a set of powerful tools for managing state, including a centralized store for managing shared data and a set of debugging tools for understanding the state of the application. This makes it easier to develop and maintain large-scale applications.

In addition to its features, Vue.js has a strong community, with a wealth of resources and plugins available for learning and problem-solving. Its popularity has led to integration with other popular front-end technologies such as TypeScript and Vue CLI, allowing for type safety and improved developer experience[35].

Embedded Ruby:

Embedded Ruby (ERB) is a templating system used in web development that allows developers to embed Ruby code into HTML or other text-based documents. ERB is a part of the Ruby programming language and is often used in developing Ruby on Rails applications.

ERB provides an easy-to-use but powerful templating system for Ruby. Using ERB, actual Ruby code can be added to any plain text document to generate document information details and flow control.

One of the key features of ERB is its ability to support variables and loops. Variables can be defined in the Ruby code and referenced in the HTML markup. Loops, such as for loops and while loops, can also be used to generate dynamic content.

Another useful feature of ERB is the ability to include partial templates, smaller templates that can be reused in multiple locations throughout a project. This allows developers to modularise their code and avoid duplication[7].

ERB utilises tags to define the rules of the program. The tags are similar to escape characters in other templating engines like Angular. However, ERB has multiple tags with different purposes[8].

If the application is poorly implemented, users can abuse these tag characters like in an XSS payload. Because of ERB's ability to execute ruby code, this can be used for Remote Code Execution (RCE) on the server, leading to a severe vulnerability[46].

2.4 Cross site scripting

Cross site scripting (XSS) is a type of web vulnerability that allows an attacker to inject malicious scripts into a web application, which are then executed in the context of other users' browsers. This type of attack occurs when a web application fails to properly validate and sanitize user-supplied input before including it in dynamically generated web pages. As a result, the injected scripts can be executed in the browser of other users who visit the vulnerable web application, leading to a wide range of attacks such as theft of sensitive information, session hijacking, and defacement of web pages[25].

XSS vulnerabilities are categorized into three main types: Stored XSS, Reflected XSS, and DOM-based XSS. In Stored XSS, the malicious script is stored on the target web application's server and is displayed to other users when they visit the affected page. Reflected XSS occurs when the malicious script is reflected back to the user in the server's response, typically as part of a URL parameter or in a form submission. DOM-based XSS, on the other hand, occurs when the injected script is executed directly within the Document Object Model (DOM) of the web page, without involving the server[26].

2.4.1 Common XSS Attack Vectors

There are several common attack vectors used in XSS attacks, including:

- Script injection in input fields: Attackers can inject malicious scripts into input fields, such as textboxes and forms, which are then rendered as part of the web page without proper sanitization.
- URL parameter injection: Attackers can inject scripts into URLs as parameters, which are then reflected back in the server's response, potentially leading to script execution in the user's browser.
- Cross-Site Script Inclusion (XSSI): Attackers can include malicious scripts from an external domain, which are then executed in the context of the vulnerable web application.
- Cross-Site Request Forgery (CSRF) with XSS: Attackers can leverage XSS vulnerabilities to execute unauthorized actions on behalf of a victim user, such as changing their password or performing actions without their consent[25].

2.5 DOM-Based XSS

DOM-based Cross-Site Scripting (XSS) is a type of XSS vulnerability that poses a significant threat to modern web applications. It is characterized by the injection of malicious code into the DOM of a web page, which can then be executed by the user's browser.

One of the most significant differences between DOM-based XSS and other types of XSS vulnerabilities is that the malicious payload is not directly injected into the server-side response. Instead, it is injected into the client-side JavaScript code and executed within the context of the victim's browser. This makes the vulnerability particularly challenging to detect and mitigate since traditional server-side security measures are often ineffective against it.

The root cause of DOM-based XSS vulnerabilities is the failure of the web application to properly sanitize user input that is used to construct dynamic JavaScript code. This allows an attacker to inject a payload that includes JavaScript code that is executed within the victim's browser context. The attacker can exploit the vulnerability in various ways, such as by tricking the victim into clicking a link, submitting a form, or simply loading a web page that contains the malicious payload.

Once the payload is executed, the attacker can perform a range of actions, such as stealing sensitive user data, modifying the web page's content, or redirecting the victim to a phishing website. This type of attack can be particularly devastating since the malicious payload can execute in the context of a trusted website, making it difficult for the user to detect that they are under attack[26].

2.6 Vulnerability scanning

Vulnerability scanning is a critical component in ensuring the security of digital systems, as it enables organizations to identify and address potential security risks before they can be exploited by attackers. Automated vulnerability scanners are designed to detect vulnerabilities in web applications, and any other digital system by analyzing various aspects of the application's functionality and behavior.

A vulnerability scan of a web application is a systematic assessment of the application's security posture. It involves the use of automated tools and techniques to identify vulnerabilities and weaknesses in the application, which could be exploited by attackers.

The process of a vulnerability scan typically involves the following steps:

- Discovery: The vulnerability scanner identifies the target web application and performs a scan to identify the application's assets, such as web pages, URLs, and parameters.
- Mapping: The vulnerability scanner maps the application's structure and identifies the relationships between the various components. This step involves analyzing the application's navigation flow and identifying the different modules and functions.
- Enumeration: The vulnerability scanner enumerates the application's vulnerabilities and weaknesses by sending various input parameters to the application and analyzing the responses. This step involves using various techniques such as fuzzing, brute-forcing, and injection attacks.
- Analysis: The vulnerability scanner analyzes the results of the enumeration phase and categorizes the vulnerabilities based on their severity, impact, and likelihood of exploitation. This step involves identifying false positives and prioritizing vulnerabilities based on their criticality.
- Reporting: The vulnerability scanner generates a report that summarizes the findings of the scan, including the identified vulnerabilities, their severity, and recommendations for remediation. The report is typically used by developers and security professionals to fix the identified vulnerabilities and improve the application's security posture.

A vulnerability scan of a web application typically focuses on identifying common web application vulnerabilities such as cross-site scripting (XSS), SQL injection, and file inclusion vulnerabilities. It also includes checks for misconfigurations, outdated software components, and weak passwords.

The effectiveness of a vulnerability scanner is directly dependent on its ability to accurately identify vulnerabilities in the web application. This entails minimizing both False Positive (FP) and False Negative (FN). The scanner identifies false positives as vulnerabilities, but in reality, they do not exist in the web application. False negatives, on the other hand, are actual vulnerabilities that go undetected by the scanner. It is critical to reduce false positives and negatives to ensure that the vulnerability scanner provides accurate results.

A True Positive (TP) is an actual vulnerability that is detected and reported by the scanner. The TP rate is a crucial metric for evaluating the effectiveness of a vulnerability scanner, as it measures the percentage of actual vulnerabilities that are correctly identified by the scanner. A higher TP rate indicates that the scanner is more effective at identifying vulnerabilities in the web application.

The performance of a web application vulnerability scanner can be evaluated using statistical performance metrics. Precision, Recall, and F-Measure are among the key metrics used in addition to the number of TP, FP, and FN[1].

- Precision is the ratio of accurately detected vulnerabilities to the total number of detected vulnerabilities and is represented as:

$$\frac{TP}{TP+FP}$$

- Recall, on the other hand, is the ratio of correctly detected vulnerabilities to the total number of existing vulnerabilities, represented as:

$$\frac{TP}{TP+FN}$$

- Finally, F-Measure is the harmonic mean of Precision and Recall. This metric provides a combined score that reflects the balance between Precision and Recall. It is represented as:

$$2 * \frac{Precision * Recall}{Precision + Recall}$$

2.6.1 Types of scanning techniques

Consideration should be given to the type of scanning required for a web application vulnerability assessment. Various methods are available, and it is important to select the appropriate one to obtain accurate results. The following assessment techniques coincide with three scanning methods[18]:

- Black Box assessment: This method does not require an understanding of the application's internal workings and focuses on identifying vulnerabilities through external testing.

- White Box assessment: This method requires comprehensive knowledge of the application's internal structure and focuses on identifying vulnerabilities through internal testing.

- Gray Box assessment: This method integrates aspects of both black box and white box scans and requires some understanding of the application's internal workings.

In addition to the assessment techniques, there are several analysis methodologies. These methodologies outline the process of conducting a scan and the specific components of the software that are analyzed. These methods include dynamic, static, and hybrid analysis. Although these terms are commonly utilized in malware research, they can be applied to any form of software analysis[11][34].

- Dynamic Analysis: This method identifies vulnerabilities by interacting with the application in real time, sending requests, and analyzing the responses.

- Static Analysis: This method identifies vulnerabilities by analyzing the application's source code or compiled binaries without executing the code.

- Hybrid Analysis: This method integrates elements of both dynamic and static scans to provide a more comprehensive evaluation of the application's security posture.

Each of the methods described has its advantages and disadvantages. The choice between scanning and analysis method should be based on the specific requirements of the assessment and the resources available. Similarly, the choice between a black box, white box, or gray box assessment will depend on the knowledge available about the application's internal workings. It is essential to understand the strengths and limitations of each method to ensure that a comprehensive and practical vulnerability assessment is performed[18].

Black box assessment

- **Benefits:**
  - Analysts does not have access to credentials or access-tokens before testing.
  - The scan is performed from the perspective of an attacker, providing a realistic view of the security posture of the application.
  - It serves to uncover any discrepancies or inconsistencies in the specifications.
  - Both the developer and analyst are independent of each other.
- **Disadvantages:**
  - The scan may not identify all vulnerabilities, as internal vulnerabilities only available to signed-in users will be undetected.
  - Some sections of the back-end may not be tested at all.

White box assessment

- **Benefits:**
  - The scan is performed with a detailed knowledge of the application's internal structure, which provides a more comprehensive view of the security posture.
  - The scan can identify vulnerabilities that are not detectable through other methods.
  - Given access to a user account within scope, it can test a larger part of the application, detecting vulnerabilities only available to authenticated users.
- **Disadvantages:**
  - Requires a well defined scope with the customer before conducting.[1]
  - The scan may not be practical for large and complex applications.

Regarding gray box assessment, it offers a combination of the advantages and disadvantages of both black and white box scans. It can be more valuable than only performing black box testing, as it can identify back-end vulnerabilities that black box scans alone cannot. However, gray box scans require a higher level of expertise compared to black box assessments.[18].

Often web application penetration testing consists of a gray-box scan. The attacker is often given access to sign-in information but conducts a test without using these credentials first. This is in the context of web application testing, considered grey-box testing[20].

Dynamic vulnerability scanning of a web application is an active evaluation process to identify potential security weaknesses or vulnerabilities. The scan involves sending requests to

[1]All tests do need a well defined scope, however given access to a user account, the privileges of the account also needs to be considered.

the application and analyzing the responses to gain insights into its security posture. The goal is to uncover security risks by simulating user interactions and evaluating the application's responses in real-time.

For instance, the dynamic scanner may simulate a user submitting data into a web form and evaluate the application's input validation. If the validation process is not correctly implemented, the scanner may identify it as a vulnerability, as the application could be susceptible to attack.

Dynamic scanning can provide a more comprehensive evaluation of the application's security posture as it considers its actual behavior, including how it responds to different inputs and conditions. However, it should be noted that dynamic scans may generate false positive results and, therefore, should be used in conjunction with other scanning methods for a complete security assessment[11].

Compared to dynamic scanning, static scanning of a web application involves analyzing its source code or compiled binaries without executing the application. The static scan focuses on the code itself, where it searches for known security weaknesses and coding practices that could lead to vulnerabilities and other potential security risks.

Examining the code provides a deeper understanding of the application's operation. However, the lack of consideration for the application's actual behavior may result in missed vulnerabilities. Static scanning is usually conducted in conjunction with dynamic scanning to address this issue. It is known as hybrid scanning, giving a more comprehensive evaluation of the application's security posture[11][9].

Dynamic Scan:

- **Benefits**:
  - The scan is performed in real-time and can identify vulnerabilities that are not detectable through other methods.
  - The scan provides a more realistic view of the security posture of the application, as it interacts with the application in the same way as an attacker would.
- **Disadvantages**:
  - The scan may not identify all vulnerabilities, as the internal structure of the application is not taken into account.
  - The scan may cause a performance impact on the application and the network.

Static Scan:

- **Benefits**:
  - The scan is performed without executing the code and can identify vulnerabilities that are not detectable through other methods.
  - The scan can be performed faster and with less resources compared to dynamic scans.
- **Disadvantages**:
  - The scan may not identify all vulnerabilities, as it does not take into account the runtime behavior of the application.
  - The scan may generate false positive results, leading to a waste of time and resources.

Hybrid Scan:

- **Benefits**:
  - The scan provides a comprehensive view of the security posture of the application by combining the benefits of both dynamic and static scans.
  - The scan can identify vulnerabilities that are not detectable through other methods.
- **Disadvantages**:
  - The scan requires a combination of resources and expertise, which may be difficult or expensive to acquire.
  - The scan may still not identify all vulnerabilities, as the internal structure of the application is not fully known.

2.7 Current vulnerability scanners

The current market of web application vulnerability scanners is split between premium paid solutions and free, open-source solutions. The paid solutions provide an up-to-date product, often with more features than the free products. In contrast, the free scanners are rarely updated and lack features. However, many organizations still opt for free, open-source solutions due to budget constraints or a preference for open-source software. While free scanners may have some limitations, they still offer a valuable service to organizations looking to scan their web applications for vulnerabilities.

In addition, it is important to keep in mind that web application vulnerabilities are constantly evolving, and a scanner that was effective last year may not be sufficient today. Regular updates and maintenance of the scanner are crucial to ensure that it remains effective in identifying vulnerabilities in web applications.

There are reports indicating that these scanners do not provide enough support for specific injection vulnerabilities, such as certain types of XSS[2]. Moreover, they fail to address template injection vulnerabilities, which share many similarities with XSS. This highlights a significant gap in the current market and an opportunity for improvement.

With this in mind, a further study of some available free vulnerability scanners has been done. Four scanners have been chosen based on testing done in other reports[1].

2.7.1 OWASP ZAP

OWASP ZAP (Zed Attack Proxy) is a popular open-source vulnerability scanner for web applications that was first released in 2010. It is designed to be easy to use, highly customizable, and suitable for experienced and novice security testers.

ZAP uses a passive scanning approach, meaning it does not send malicious payloads to the tested web application. Instead, it intercepts and analyzes the requests and responses between the client and server, looking for potential vulnerabilities. This approach helps to reduce the risk of accidentally triggering false positives or causing damage to the target application.

ZAP includes a wide range of automated testing features, including the ability to detect common vulnerabilities such as SQL injection, cross-site scripting (XSS), and directory traversal.

It also includes several advanced testing features, such as the ability to test for authentication and session management vulnerabilities and perform active scanning of web applications.

One of the key features of ZAP is its high degree of customizability. For example, it includes a powerful scripting engine that allows security testers to write scripts to extend the tool's functionality or automate repetitive testing tasks. Additionally, ZAP includes a number of built-in plug-ins that can be used to customize the tool for specific testing scenarios.

ZAP also includes a suite of reporting and analysis tools that allow security testers to quickly and easily analyze the results of their testing. In addition, it provides detailed reports on vulnerabilities found, along with suggested remediation steps and links to additional resources for further learning[48].

2.7.2 IronWASP

IronWASP is a free, open-source vulnerability scanner for web applications first released in 2009. It is designed to be highly customizable and can be used to test a wide range of web applications, including those with complex architectures and non-standard configurations.

One of the key features of IronWASP is its ability to detect both common and advanced vulnerabilities, including SQL injection, cross-site scripting (XSS), file inclusion, and server-side request forgery (SSRF). It also supports testing for vulnerabilities in APIs and mobile applications, making it a versatile tool for testing the security of modern web-based applications.

IronWASP uses a combination of automated and manual testing techniques to identify vulnerabilities. It includes a variety of built-in testing modules that can be configured to test for specific vulnerabilities or to perform general security testing. Additionally, it includes a suite of manual testing tools that allow security testers to perform more in-depth testing of web applications. One of the unique features of IronWASP is its support for customized scripting. This allows security testers to create their own testing scripts and automate repetitive testing tasks, further increasing the efficiency of the testing process.

IronWASP also includes a number of reporting and analysis tools that allow security testers to quickly and easily analyze the results of their testing. In addition, it provides detailed reports on vulnerabilities found, along with suggestions for remediation and links to additional resources for further learning[23].

2.7.3 Wapiti

Wapiti is a free and open-source vulnerability scanner designed for web applications. It was developed in 2008 by Nicolas Surribas, and the source code is available on GitHub. The name "Wapiti" comes from the Shoshone word for elk, known for its keen sense of smell and ability to sniff out hidden things - similar to how the Wapiti vulnerability scanner is designed to discover hidden vulnerabilities in web applications.

Wapiti uses a black-box testing methodology, meaning that it does not require access to the source code of the web application being tested. Instead, it sends specially crafted HTTP requests to the application and analyzes the responses to identify potential vulnerabilities. This approach allows Wapiti to test web applications that are already deployed and running, making it a valuable tool for testing the security of publicly accessible web applications.

One of the key features of Wapiti is its ability to detect a wide range of vulnerabilities, including SQL injection, cross-site scripting (XSS), file inclusion, command injection, and more. It does this by using a variety of testing techniques, such as parameter brute-forcing, error-based testing, and time-based testing.

Wapiti is also designed to be highly customizable, with a wide range of configuration options that allow users to tailor the testing process to their specific needs. For example, users can specify which types of vulnerabilities to test for, which HTTP methods to use, and how many requests to send per second. This customization level can help users optimize the testing process and minimize false positives[42].

2.7.4 Vega

Vega is a free and open-source vulnerability scanner for web applications that was first released in 2011. It is designed to be a user-friendly and powerful tool for detecting vulnerabilities in web applications.

One of the key features of Vega is its ability to perform both active and passive scanning of web applications. Active scanning involves sending payloads to the web application being tested to look for potential vulnerabilities. Passive scanning, on the other hand, involves analyzing the requests and responses between the client and server to detect vulnerabilities without sending any malicious payloads.

Vega supports a wide range of automated testing features, including the ability to detect common vulnerabilities such as SQL injection, cross-site scripting (XSS), and file inclusion. It also includes advanced testing features, such as the ability to test for authentication and session management vulnerabilities.

In addition to its automated testing capabilities, Vega includes a suite of manual testing tools that allow security testers to perform more in-depth testing of web applications. These tools include intercepting and modifying requests and responses, analyzing cookies and session data, and performing customized testing tasks.

One of the unique features of Vega is its support for collaborative testing. It includes a built-in proxy server that allows multiple security testers to work on the same testing project simultaneously and to share testing data and results.

Vega also includes a powerful reporting and analysis engine that allows security testers to quickly and easily analyze the results of their testing. In addition, it provides detailed reports on vulnerabilities found, along with suggested remediation steps and links to additional resources for further learning[41].

Chapter 3

Work methodology

This section details the steps taken to accomplish the thesis project, outlining the preliminary work structure and the planning and execution of various sections. Additionally, the section incorporates the attack methodology for how the vulnerability scanner tests with payloads.

The methodology and workflow were designed based on a sequential model, where each stage needed to be completed before moving on to the next. Therefore, the predetermined sections were structured as follows:

- Objective planning
- Literature review of existing research on the subject
- Analysis of existing vulnerability scanners
- Analysis of vulnerabilities?
- Design vulnerability scanner
- Develop vulnerability scanner
- Gather and research payloads
- User feedback?

3.1 Objective planning

During the planning phase of the thesis, the objectives were established with the aim of aligning the project's goals with the expectations of Netsecurity and the practical constraints of the allotted time frame. It was acknowledged that discovering new vulnerabilities may not be a guaranteed outcome, and thus all parties agreed to set realistic expectations for the project.

3.2 Literature review of existing research on the subject

This section of the thesis was designed with the aim of conducting a comprehensive literature review to gain a deeper understanding of the subject matter. It was recognized that this research is essential, as it provides a solid foundation for the remaining sections of the thesis. To this end, a thorough investigation of previous studies on web application exploits was undertaken, focusing on relevant research papers. The process involved carefully analyzing and evaluating the available literature in order to identify key concepts, theories, and findings. The result was a well-researched and evidence-based review that served as an essential guide for the subsequent thesis sections. The comprehensive and meticulous nature of the review ensured that the research was sound and reliable, providing a solid basis for the development of the thesis as a whole[27].

3.3 Analysis of existing vulnerability scanners

Several existing frameworks for vulnerability scanning already exist, but many of them focus on different types of attacks than those targeted by this thesis. However, it was still valuable to study these frameworks to understand how they automated the scanning process and applied techniques such as fuzzing to find vulnerabilities. By exploring these existing frameworks, the development of a more effective and comprehensive vulnerability scanner for template injection attacks, DOM-based XSS, and XSS can be informed and enhanced[4].

3.4 Analysis of vulnerabilities

To develop an effective vulnerability scanner, it is essential to understand the underlying principles behind each of them. CSTI, for example, involves the injection of user input into templates on the client side, which can lead to unintended code execution. On the other hand, DOM-based XSS exploits vulnerabilities in the Document Object Model (DOM) of web pages, allowing attackers to execute malicious scripts within the victim's browser. Finally, XSS is a well-known vulnerability that allows attackers to inject and execute their code on web pages, which can lead to sensitive data theft or unauthorized access. By examining how existing vulnerability scanners approach these types of vulnerabilities and how they can be detected and mitigated, we can develop a more effective and specialized scanner to detect these specific types of attacks. This involves exploring payload generation, pattern matching, and vulnerability correlation techniques to identify and report on CSTI, DOM-based XSS, and XSS vulnerabilities.

3.5 Design vulnerability scanner

The design phase of the vulnerability scanner was initiated by employing the Unified Modeling Language diagram. This technique was chosen as it offered a clear and accurate representation of the various components that would be integrated into the scanner before the development stage commenced. By carefully planning and outlining the scanner's features and functionalities, we ensured that the development phase would progress seamlessly without any significant roadblocks. Furthermore, using a well-structured and comprehensive design plan enabled the team to allocate resources better and prioritize tasks. Ultimately, the UML diagram was crucial for keeping the entire project on track and helped us achieve our goals within the stipulated timeframe.

3.6 Develop vulnerability scanner

The section focused on developing and coding an automated vulnerability scanning framework. The choice of programming language was Python due to its ease of use and extensive scripting support. The development process took several months as it was essential to create a solid foundation to build upon. Extensive research was conducted to identify existing frameworks, and a thorough analysis of the available solutions was carried out to determine the most suitable approach for achieving the project's objectives. Throughout the development process, various iterations were tested and refined until the final framework was robust, reliable, and capable of detecting the targeted vulnerabilities, including client-side template injection, DOM-based XSS, and XSS.

3.7 Construct payloads

Upon completing the design and development of the vulnerability scanner prototype, the next step was creating a payload library to test for actual vulnerabilities. To achieve this, a comprehensive payload library was composed for each type of vulnerability that would be tested, including client-side template injection, DOM-based XSS, and XSS. The process involved identifying the most commonly used attack vectors for each vulnerability type and generating payloads to simulate these attacks. Ensuring that the payloads were thorough and accurate to produce reliable results during testing was crucial. Creating the payload library was an extensive process involving significant research and testing to ensure the scanner could detect a wide range of vulnerabilities accurately.

Chapter 4

Method

4.1 Design specifications

To develop the scanner efficiently and systematically, the following tools have been used to specify the design specifications. Functional Requirements Document (FRD) was chosen because it provides a structured way to define and document the functional requirements of the scanner. By using FRD, the team could define and prioritize the scanner's features and functionalities, which helped ensure that the scanner was developed in a user-driven manner that aligned with the needs of the target users.

Unified Modeling Language (UML) was chosen because it provides a standard language for visualizing, specifying, constructing, and documenting the artifacts of software systems. By using UML, the team could model and design the scanner's architecture, components, and behavior in a structured and consistent way, which helped ensure that the scanner was developed with high quality and maintainability.

In addition, the development process was driven by user feedback and testing to ensure that the scanner met the needs of the target users. User-driven development was crucial to ensuring that the scanner was user-friendly, effective, and easy to use for both technical and non-technical users.

4.1.1 Functional Requirements Document (FRD)

A Functional Requirements Document (FRD) is a formal document that outlines the detailed functionalities, features, and capabilities of a software system or application. It serves as a blueprint for the development team, providing a comprehensive and organized description of what the software is supposed to do, and how it is expected to perform. The FRD acts as a reference for stakeholders, including developers, testers, and project managers, to ensure that the software meets the requirements and expectations of the intended users.

4.1.2 UML Class Diagram

A UML (Unified Modeling Language) Class Diagram is a visual representation of the static structure of a software system or application. It describes the classes, their attributes, relationships, and behavior of the objects in the system. Class diagrams are commonly used during the design and analysis phase of software development to model the structure and organization of the software system or application.

- Classes: Represents the static structure of the software system or application, including the classes or objects that exist in the system. Each class is depicted as a rectangle, with the class name at the top, followed by its attributes and methods.

- Attributes: Describes the properties or characteristics of a class, which define the state of the objects of that class. Attributes are depicted as name-value pairs, and may include data types, visibility (public, private, protected), and multiplicity (cardinality).
- Methods: Represents the behavior or actions that can be performed by the objects of a class. Methods are depicted as name-signature pairs, and may include input parameters, return types, and visibility.
- Relationships: Depict the associations or connections between classes in the system. Common types of relationships include:
  - Association: Represents a generic relationship between classes, indicating that one or more objects of a class are associated with one or more objects of another class. Associations are depicted as lines connecting the classes, with optional labels to indicate the nature of the association.
  - Inheritance/Generalization: Represents an inheritance relationship between classes, indicating that one class inherits attributes, methods, and behavior from another class. Inheritance is depicted as a solid line with a triangular arrowhead pointing towards the superclass.
  - Composition/Aggregation: Represents a whole-part or container-contained relationship between classes, indicating that one class is composed of or aggregated by another class. Composition is depicted as a filled diamond at the container class end, connected to the contained class with a solid line.
- Multiplicity: Specifies the cardinality or number of instances that can be associated with a class in a relationship. Multiplicity is depicted as numbers or ranges near the ends of the association lines, indicating the minimum and maximum number of instances allowed.
- Stereotypes: Represents additional annotations or tags that can be used to further define the characteristics of a class or relationship. Stereotypes are depicted as labels in guillemets (« ») above or below the class or relationship.

The UML Class Diagram serves as a visual representation of the structure and relationships between classes in the software system or application. It helps in understanding the organization of objects and their interactions, and serves as a reference for developers during the implementation phase. It can also be used for documentation, communication, and analysis purposes.

4.2 Choice of technologies and tools

4.2.1 Selenium

The *selenium* library is a popular Python package for automating web browsers. It provides a simple and consistent interface to control web browsers programmatically and perform actions such as filling out forms, clicking buttons, and navigating between pages. selenium can test web applications, scrape website data, or interact with web-based services. It provides an API to interact directly with most browser drivers, such as Firefox, Chrome, and Edge. [17]

Selenium is distributed under the Apache License 2.0, a permissive open-source license allowing for unrestricted use, modification, and distribution of the software, subject to certain conditions. This license ensures that the code is available to the community for personal and commercial use and encourages contributions and improvements to the project. Overall, the *selenium* library is a powerful and flexible tool for automating web browsers in Python and is widely used in various industries, such as software testing and web development.[38]

Selenium wire

Selenium Wire is a Python library that extends Selenium's Python bindings to give you access to the underlying requests made by Selenium WebDriver. This means that you can intercept and modify HTTP requests and responses made by the browser, allowing you to more effectively test web applications and automate tasks.

4.2.2 Requests

The *requests* library is a popular Python package for making HTTP requests to web servers. It provides an easy-to-use interface for sending HTTP/1.1 requests with a variety of methods such as GET, POST, PUT, DELETE, etc. and handling the resulting responses. The library is built on top of the *urllib3* library and supports many advanced features like SSL/TLS verification, timeouts, authentication, session handling, and more.[32] *requests* is also distributed under the Apache License 2.0.[38]

4.2.3 Beautiful Soup

Beautiful Soup is a Python package that is used for web scraping purposes. It allows the parsing and manipulation of HTML and XML documents, providing an easy-to-use interface for extracting data from web pages. *Beautiful Soup* is an open-source package released under the MIT license, which means that it can be freely used and modified for personal and commercial purposes.[33][40]

Beautiful Soup creates a parse tree from the input HTML or XML document. It then provides a variety of search methods for locating specific tags, attributes, or content within the document. *Beautiful Soup* supports a wide range of search techniques, including searching by tag name, attribute, text content, or CSS selector.

4.2.4 tldextract

The tldextract library is a Python package that allows the extraction of the Top-level Domain (TLD), subdomain, and domain name from a given URL. This library is distributed under the BSD 3-Clause License, allowing unrestricted software use, modification, and distribution, subject to certain conditions that do not apply to this project.[24][39]

The library uses a combination of algorithms and a public domain list of TLDs to determine the TLD of a given URL. It first separates the domain into its subdomains and top-level domain, then checks the extracted TLD against a list of known TLDs to ensure that it is valid. If the TLD is not found in the list, the library assumes that the TLD is a second-level domain and continues to extract the domain name.

4.2.5 URLlib

urllib is a package that collects several modules for working with URLs, where this project only uses *urllib.parse*. This module defines a standard interface to break Uniform Resource Locator (URL) strings up into components (addressing scheme, network location, path etc.), combine the parts back into a URL string, and convert a "relative URL" to an absolute URL given a "base URL"[14].

4.2.6 Git

Git is a popular version control system developers use worldwide to manage source code, collaborate on projects, and track changes over time. As a distributed system, Git allows developers to work offline and synchronize their changes with a central repository when they return online. This makes it a powerful tool for teams working on complex projects with many contributors.

Git was the chosen version control system in this project due to its versatility and ability to manage the different development branches. Additionally, Git's robustness in managing code changes ensured that no data was lost during development.

To facilitate collaboration and version control management, GitHub was the primary platform for storing and sharing project repositories. This allowed the code and documentation to be shared quickly.

Multiple repositories were created on GitHub to keep the project organized and segmented into specific components. Each repository was dedicated to a specific component or feature, making it easy to work on individual parts of the project and manage their changes effectively[22].

4.2.7 Pycharm

PyCharm is an integrated development environment (IDE) used for Python programming. It provides developers with tools for writing and debugging code, managing project dependencies, and integrating with version control systems like Git. PyCharm was used extensively in our project as the primary IDE for development and testing. It offered a user-friendly interface and a variety of valuable features, such as code completion, refactoring, and debugging tools. Additionally, PyCharm's integration with Git and GitHub allowed for efficient collaboration among team members and streamlined the development process[30].

4.2.8 Python

Python is a high-level, interpreted programming language that is widely used for various applications such as web development, data analysis, machine learning, and scientific computing. Its simplicity, ease of use, and availability of a vast range of libraries make it a popular choice among developers[31].

4.3 Vulnerability scanner

4.3.1 Development process

Crawler

To begin the development process, we first created a functional web crawler. This tool is crucial for a vulnerability scanner to operate effectively. In traditional web applications, there are multiple pages linked with hyperlinks under the same domain[37]. These pages can have unique vulnerabilities that may not exist on other pages. Using a web crawler, the scanner can locate and index all pages within the web application, allowing for comprehensive vulnerability testing.

To start the development of the crawler, a list of requirements was created:

- Ability to find all hyperlinks on the page
- Not index duplicates
- Limit it to scanning a specified domain

To avoid unnecessary work, a search was conducted to find pre-existing solutions in the form of open-source libraries or plugins for Python. Two options were evaluated: creating a solution from scratch, using code from scrapingbee.com as a foundation, or utilizing Scrapy, a Python framework for web crawling. Ultimately, the decision was made to modify the crawler found on scrapingbee.com, as Scrapy was deemed overly complex for the current project[5][13].

The necessary adjustments were made to ensure the scanner included the desired features. A JSON configuration file was created to address the crawler going beyond the intended domain. This file specified the domain, suffix and subdomain; specifying the subdomain is optional. If any URLs were found that did not fit within the specified range, they were not included in the list of pages to crawl.

JSON is preferred as a scope file over XML because of its concise syntax and personal preference. Furthermore, JSON files are easy to parse, making it the go-to choice for defining the scope and payload library.

During its development, the crawler has remained almost identical to its earliest version, with the only addition being the function to check the page source text for links based on a regex pattern. This functionality was added after Netsecurity requested it.

Input field finder

To create the input finder, the plan was straightforward. First, the beautiful soup library would be used to parse each page discovered by the crawler. This library creates a soup class, allowing the user to loop over all input fields on the page. The next step would involve testing the inputs by sending a dummy payload to the page. The page would then be rechecked for instances of the dummy payload to verify if it gets reflected on the page. Lastly, the pages with a reflection would be listed alongside the input fields for the payload sender to send payloads to. As the input finder already has the code to send payloads, the payload sender would be integrated into the input finder.
The initial plan for creating the input finder seemed straightforward, but it had to be revised before any coding could transpire. For instance, there are 22 different input field types, with most having completely different functions**??**. So it had a sorting system that indexed the fields according to type. It also indexed buttons, as they are often hyperlinks with a predefined payload. By changing the hyperlink payload, one can inject template and XSS payloads.

Secondly, sending payloads to the different input fields differed widely. Therefore one function per field type was planned. However, a complete rewrite was done before all five functions were written. As a result, only payload-sending functions were written for buttons and text fields. Furthermore, the function for text fields utilised Selenium to send the payloads, whereas the other used the requests session. The function for sending payloads to buttons only supported sending GET requests, and this is mostly fine as it is rare to use other methods with buttons. It functioned by altering the URL from the button, using string manipulation to change the payload.

In the first iteration of developing the scanner, we drew from our combined experience in manual web application penetration testing and consulted with experts at Netsecurity. We did, however, have no previous knowledge of web development, leading to over-ambition and misconceptions. This was prevalent in input detection, as hidden fields were treated as typical input fields, not just a container for data that users cannot see or modify when a form is submitted. They are also often required to be submitted together with the form, and because the scanner only had support for one input at the time, this was impossible. In addition, wanting to include selection fields like drop-down menus and checklists made things complex. Because even though one can inject payloads to such inputs, having those inputs on a web page is rare, the impact of not having them is insignificant. These and other issues lead to the following limitations.

The first versions of the input finder had several limitations. One of the main issues was that it could only send a payload to a single input field at a time and ignored the <form> tag, which is standard when grouping inputs in HTML. This made it impossible to determine if two inputs needed to be submitted together. Additionally, the indexing system used a dictionary for each input field type, resulting in four types: text, selection, hidden, and interactive. However, the "interactive" type was never used and only served as a placeholder for future development. This indexing method created a nested dictionary with three layers, as the payloads were also added to a dictionary with the field name as the key. These limitations persisted in the first two versions of the input field finder until a complete rewrite was implemented for version three.

The main structure was kept in the input finder's third version. The main changes were to the indexing and the filtering. Instead of indexing the inputs in a dictionary like the previous version, two new lists were created, one for inputs and one for forms. A new function was created to index forms. This function creates a dictionary containing all the information relevant to the form, action, method, enctype, and inputs. The inputs are a list of all the inputs in the form filtered through the inputs filter.

The new input filter only included seven input types of the 22 available for standard HTML web applications. The seven are text, textarea, search, email, tel, URL, and number. These fields cover all input fields that accept text, and limited fields, like range and radio, are filtered out. Reducing the number of indexed input types helps reduce a lot of complexity in the payload sender. However, by keeping all text fields, it can still detect the most common vulnerabilities. This is considered an acceptable compromise for the limited development timeframe.

Payload-Sender

As previously mentioned, the payload sender was initially integrated into the first input finder. However, in order to increase the flexibility of the scanner, it was decided that the payload sender should be an independent component. Although none of the first version's payload sender was reused, the team implemented the lessons learned from the limitations of the earlier version.

To ensure the scanner's effective operation, a list of required functionalities prior to development was created. From this list, it was determined that a new class was necessary to tie all the components, namely the crawler, input finder, and payload sender, together in a more structured manner. This new class was named Webapp and was responsible for managing all the parts of the scanner and facilitating data flow among them. The development process for the Webapp class is described in the section below.

The payload sender must be capable of:

- Need to be able to send payloads to HTML forms with required inputs.
- Modifying payloads to match the specific pattern requirements of different input fields, such as email addresses
- Be able to check buttons on the page and send payloads to those.
- Check for different types of payload execution, not just alerts.
- Reporting the vulnerabilities found, along with the specific payloads that were used, to assist with effective analysis and resolution.

In the initial stage of development, a mechanism was created to send payloads to the pages. The requests library was chosen to have complete control over the HTTP requests sent from the scanner. To simplify this process, support functions were developed - one for formatting and creating the request data and another for generating a request header.

Next, a method for identifying payload execution was developed. It involved parsing the response of the request using Beautiful Soup and searching for instances of the alert function in the page's script source. However, during testing, it was found to be an imperfect method as it cannot detect DOM-XSS and often produces false positives.

Because of the limitation of payload execution detection, a different approach was needed. Here different approaches were tested, all based on Selenium. The first method tested was using selenium-requests, a library that extends the functionality of Selenium by adding support for sending web requests in a similar fashion to the requests library. The reasoning for testing this first was that it avoided a lot of rewriting of the payload sender, as all that changed were the lines where the web request was sent. However, limitations were discovered with selenium-requests, such as a lack of documented support for POST requests and data parameters, and it did not work as hoped.

Finally, Selenium was chosen as the tool for identifying vulnerabilities in a web application motivated by its ability to mimic the functionality of a standard web browser, enabling the detection of Document Object Model (DOM) based vulnerabilities. However, an extensive rewrite of the payload sender was required for Selenium to work. Because Selenium does not allow for sending requests directly but gives an API to interact with the web application like a human would. In order to submit a form, one input field has to be selected at a time, and the payload "typed" in by Selenium. This methodology imposes constraints on the tool's effectiveness, as it necessitates that each input field be visible to the simulated "user" in order to be interacted with. Furthermore, when all fields are filled, one cannot just send the request. Instead, one has to select the submit button. Automating this is challenging, as web applications are not standardized to the extent that every app uses the same name, location, or Xpath for the submission button. To solve this problem, a total of six try-except statements are used. Even then, it is buggy and tends to click the first button it sees on the page.

Development was halted at this point due to time limitations. Despite a few minor bugs, it was deemed successful as it met all the requirements specified at the start of the development process.

4.4 Webapp

The process of developing the webapp was relatively simple. Initially, the webapp received input from the main function and passed it on to the crawler. Afterward, it checked each page one by one using the input finder and payload sender, iterating over a list of URLs obtained from the crawler. The only modification made during development was the addition of the feature to iterate over the payloads in the payload library. This was achieved by utilizing the JSON library in Python to extract all the payloads categorized as XSS, DOM-XSS, and Template Injection. The webapp then tested for XSS first, followed by DOM-XSS, and lastly, Template Injection while iterating over each category one at a time.

4.4.1 Flow

1. Ask the user to provide a URL to scan.
2. Prompt the user to confirm whether they want to expand the scope, and if they say "y", automatically include the domain, subdomain, and suffix in the scope.
3. Send an HTTP request to the specified URL to crawl its pages.
4. Parse the HTML content of each page to identify any input fields.
5. Initialize Selenium and add the requests session to Selenium, transferring all cookies.
6. Find and test all buttons to check for redirection or changes in the href compared to the URL.
7. Send payloads to all buttons using Selenium.
8. Send payloads to forms using Selenium.
9. Send payloads to single input fields outside of forms.
10. Check whether the payloads execute successfully. If the payload is triggered, indicate that the page is vulnerable.

4.5 Payload list creation

Payload list creation is an essential part of a web application security assessment, specifically for detecting vulnerabilities such as Cross-Site Scripting (XSS), DOM-based XSS, and Template Injection. The payload list is a collection of specially crafted input values that aim to test the application's input validation and sanitization routines, which can help identify vulnerabilities.

4.5.1 XSS

XSS vulnerabilities, payloads are designed to inject JavaScript code that will execute in the victim's browser context. The payload list will include various types of scripts, including those that execute alerts, redirect the victim to a malicious website, or execute arbitrary code. Payloads can be categorized into different types, such as simple payloads, advanced payloads, and edge-case payloads, based on their complexity[25].

4.5.2 DOM-based XSS

DOM-based XSS, the payload list will include inputs that can manipulate the Document Object Model (DOM) to execute JavaScript code in the victim's browser. Unlike stored and reflected XSS, where the malicious code is executed on the server-side, in DOM-based XSS, the code is executed entirely in the browser. The payloads can be designed to manipulate various DOM elements, such as form fields, query strings, or window objects, to execute the malicious code[26].

4.5.3 Template injection

Template Injection, the payload list will contain inputs that aim to inject malicious code into the application's templating engine. The payloads are designed to test the templating engine's functionality and discover vulnerabilities that could be exploited to inject malicious code.

Payload Gathering

Gathering payloads for the vulnerability scanner was essential to effectively test web applications for vulnerabilities. Open-source repositories such as GitHub, GitLab, and Bitbucket are excellent sources of payloads that can be used to test for specific vulnerabilities. For example, extensive lists of payloads for client-side template injection (CSTI) can be found in repositories such as PortSwigger's Web Security Academy and OWASP's Web Security Testing Guide.

Similarly, repositories such as XSS Payloads and PayloadAllTheThings provide a wide variety of payloads that can be used to test for XSS and DOM-based XSS vulnerabilities.

The vulnerability scanner's payload library was constructed from these open-source repositories, as these payloads have been known to work effectively on web applications. Using these payloads was crucial to the effectiveness of the vulnerability scanner in identifying and remedying security issues in web applications.

To measure the effectiveness of the scanner in detecting client-side template injection attacks, we will track the number of successful detections and the amount of time it takes to detect and remediate the attack. We will also compare the performance of the scanner against each of the template engines.

4.6 Testing plan

In order to test the vulnerability scanner, a testing plan will be implemented using vulnerable web applications provided by PortSwigger and their Academy. The chosen applications clearly demonstrate various types of vulnerabilities, and solutions are always provided.
All the scanners mentioned in the state of the Art chapter will be tested in the same labs. To ensure accuracy, only one scanner will be run at a time on the web application. The lab will be reset after each scanner is tested before the next one is run on the application.

The scanner will be run on a set of ten labs that cover different types of vulnerabilities. These labs include:

- DOM XSS in AngularJS expression with angle brackets and double quotes HTML-encoded: Link
- Basic server-side template injection: Link

- Stored DOM XSS: Link
- DOM XSS in document.write sink using source location.search inside a select element: Link
- Reflected XSS into a JavaScript string with angle brackets HTML encoded: Link
- Reflected XSS into attribute with angle brackets HTML-encoded: Link
- Reflected XSS into HTML context with nothing encoded: Link
- Stored XSS into HTML context with nothing encoded: Link
- DOM XSS in document.write sink using source location.search: Link
- DOM XSS in innerHTML sink using source location.search: Link

The scanner's intended functionality can be thoroughly tested by utilising these labs as they cover Stored XSS, Reflected XSS, DOM XSS, and Template Injection. However, it is necessary to note that the Template Injection used in these labs is server-side (SSTI). Therefore, it cannot accurately say that it would detect a Client-side Template Injection (CSTI) vulnerability; however, one of the labs uses Angular JS, a client-side framework[16]. Therefore, using angular gives some idea of the CSTI capabilities of the scanner.

Chapter 5

Design

5.1 UML Class diagrams

5.1.1 Scanner version 1.0

The buildup of the first version consisted of two classes connected by a main function. The scanner only had a single instance of each class, using lists and dictionaries to index and iterate through the web application's pages.

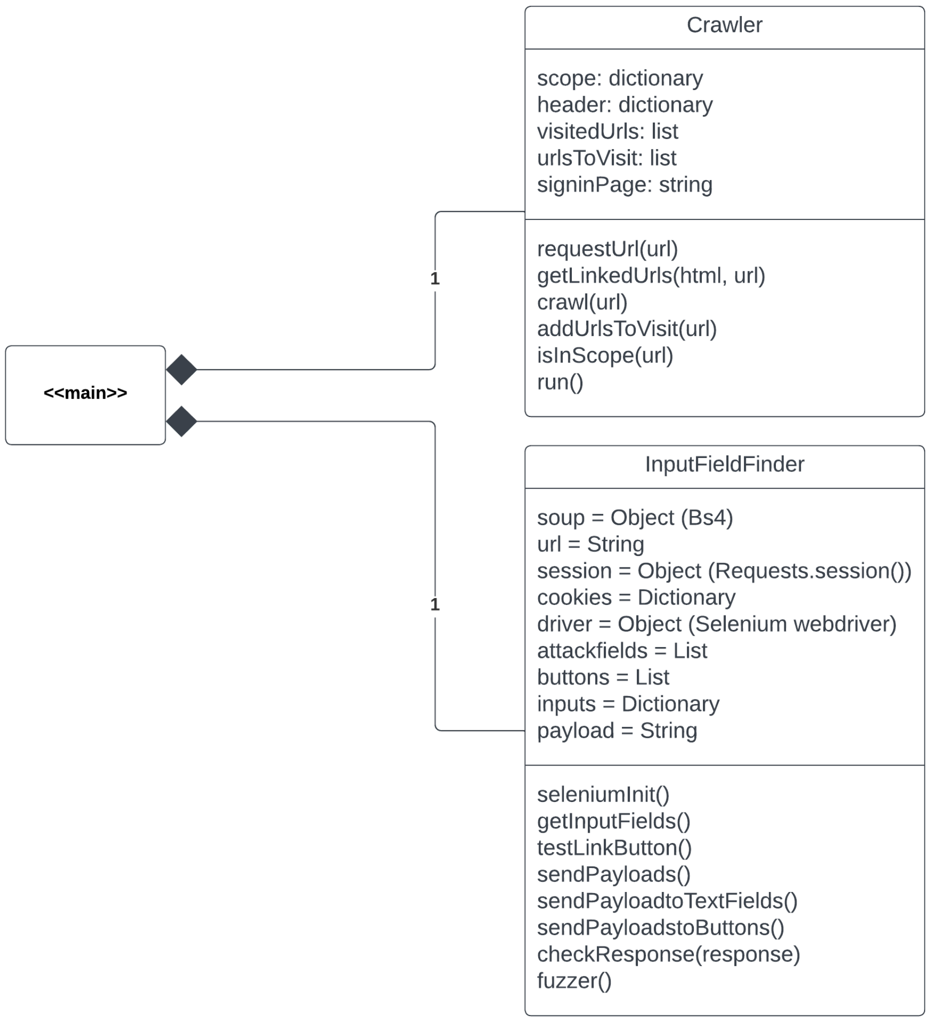


Figure 5.1: First version of the scanner

5.1.2 Scanner version 2.0

The next version divided the payload sender and input finder into two classes. A class was also created and planned to be used if a user signed in to the web app or added an authentication cookie. However, the authentication feature was never developed and removed in the next scanner version. The scanner still used only a single instance of each class, using the same method for page iteration as the first version.

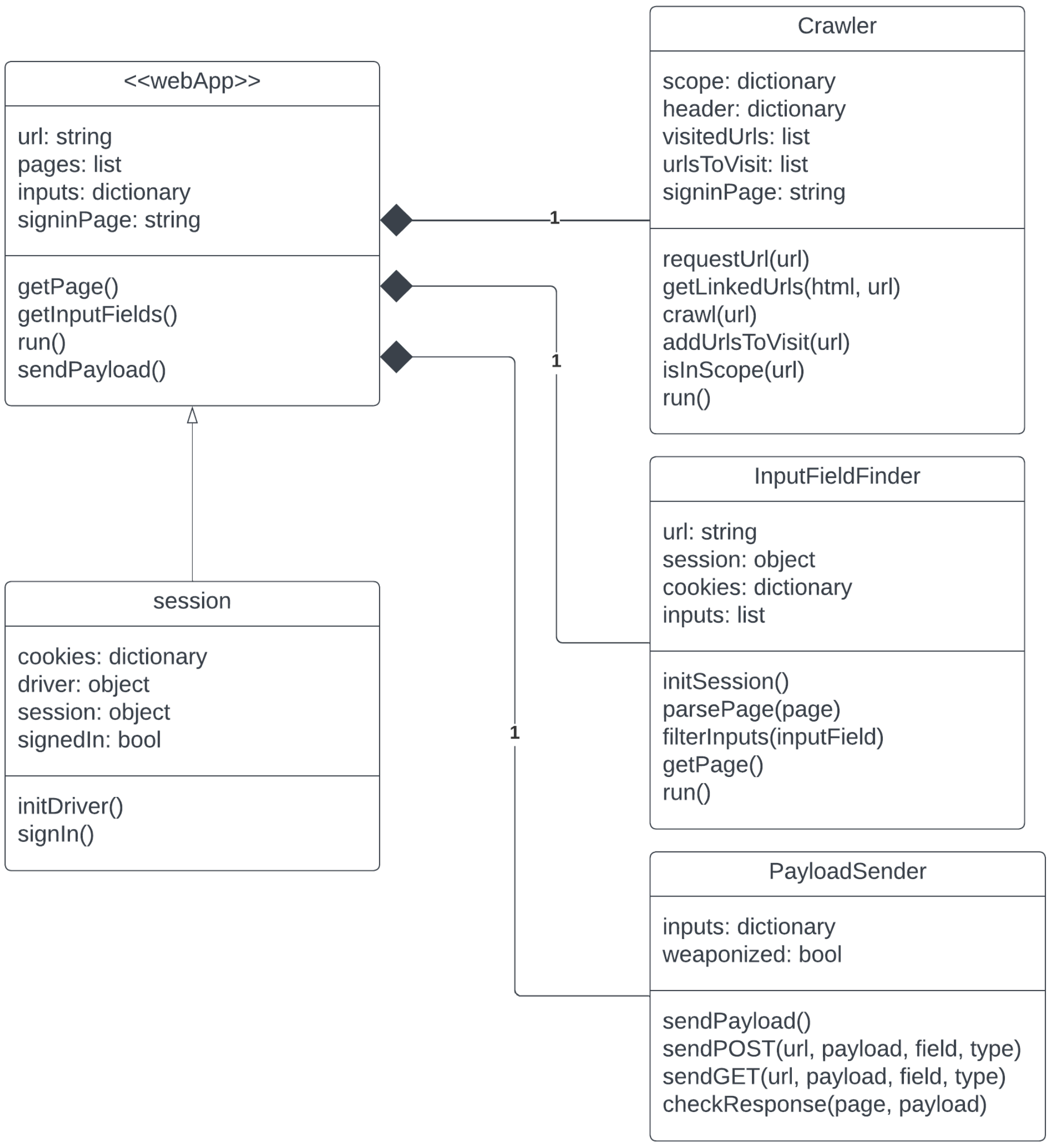


Figure 5.2: Second iteration of the scanner

5.1.3 Final scanner version

The final version of the scanner saw significant changes made to the payload sender and input finder. While the crawler still operates on a single instance and adds each page crawled to a list, the input finder and payload sender classes are now only used once per page. This simplifies the indexing process, as only the inputs and forms of one page need to be stored at any given time. Additionally, the payload sender class is now set up for each payload and each page, further streamlining the process. Although authentication functionality has been removed in this version, some unused functions from earlier versions remain.

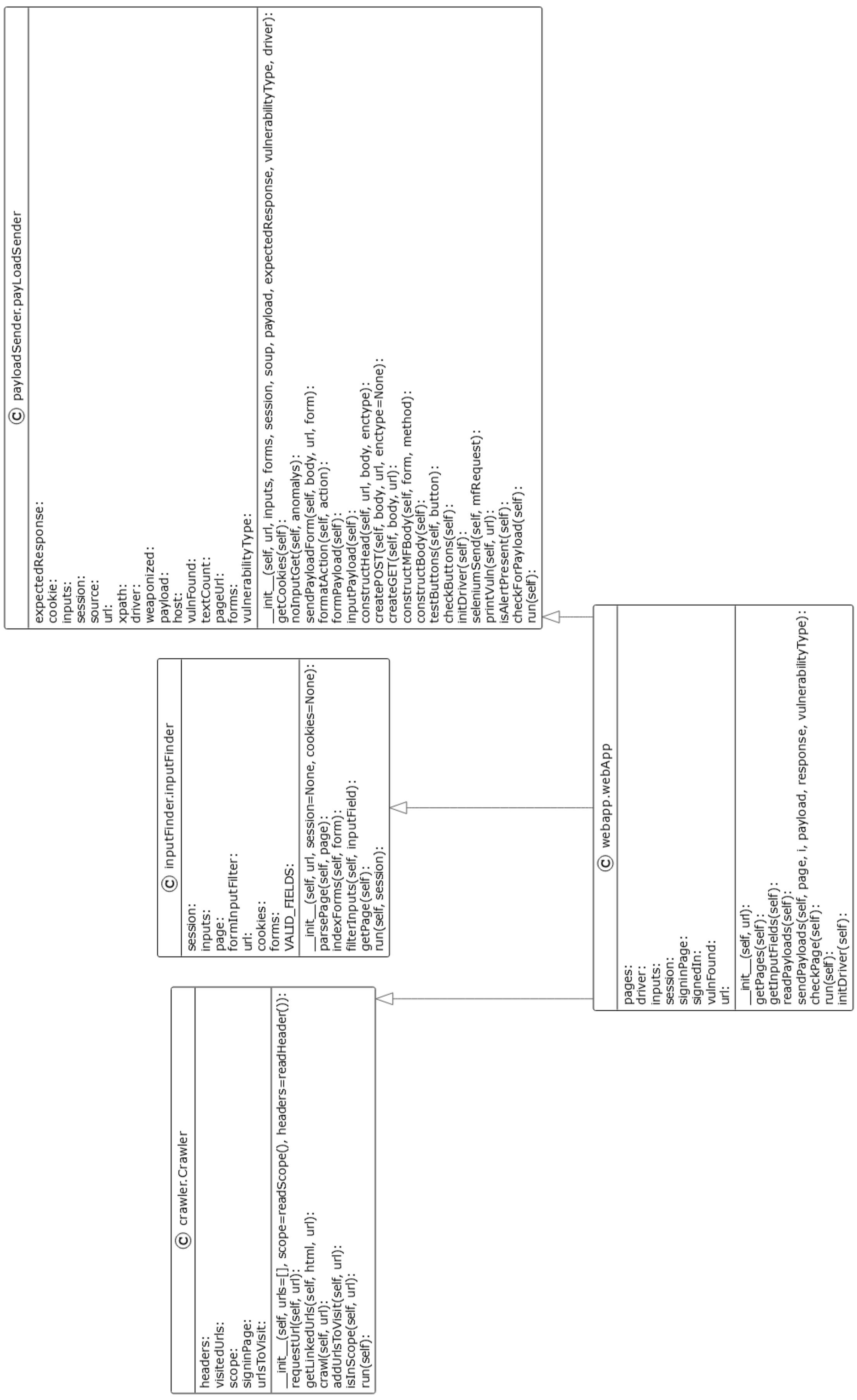


Figure 5.3: Third and final version

5.2 UML Activity Diagrams

5.2.1 Crawler version 1.0

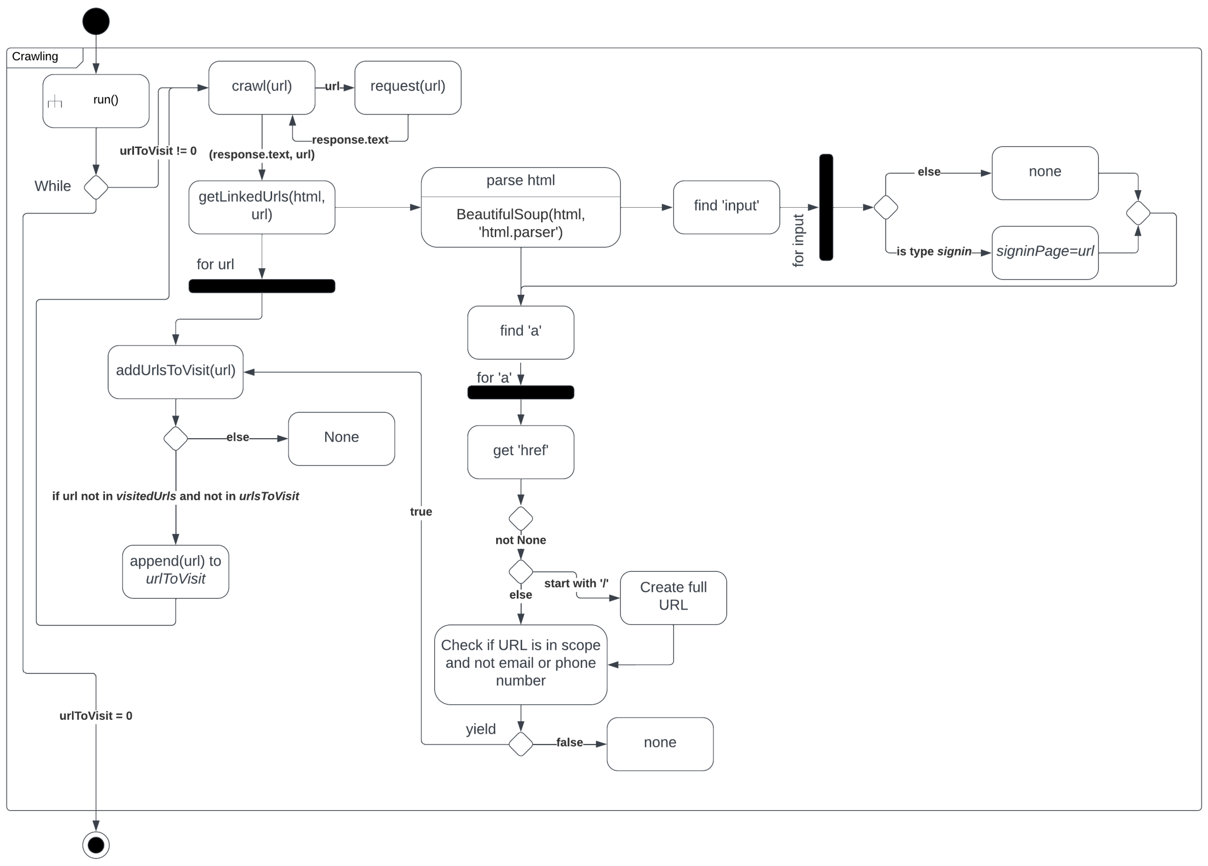


Figure 5.4: Crawler first iteration

5.2.2 Crawler version 2.0

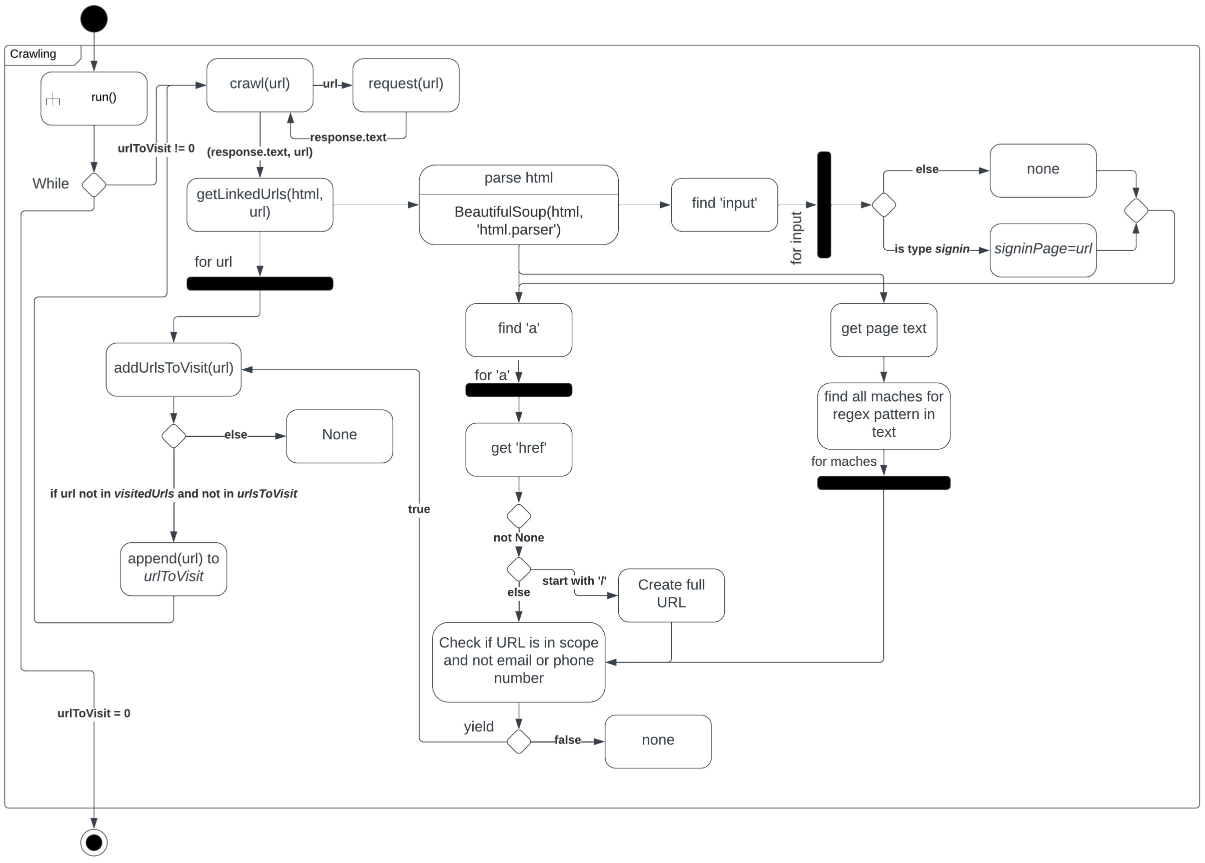


Figure 5.5: Crawler second iteration based on feedback

5.2.3 Changes in v2

In the second iteration of the crawler, one additional feature was added. The crawler now has the functionality to check through the page text to look for additional URLs which are not noted with an anchor tag. This is done using a regex pattern. If any part of the text matches the regex pattern, it is added to the list of URLs to scan. The pattern is imperfect but should capture most URLs in the page text.

5.2.4 Input-field scanner version 1.0

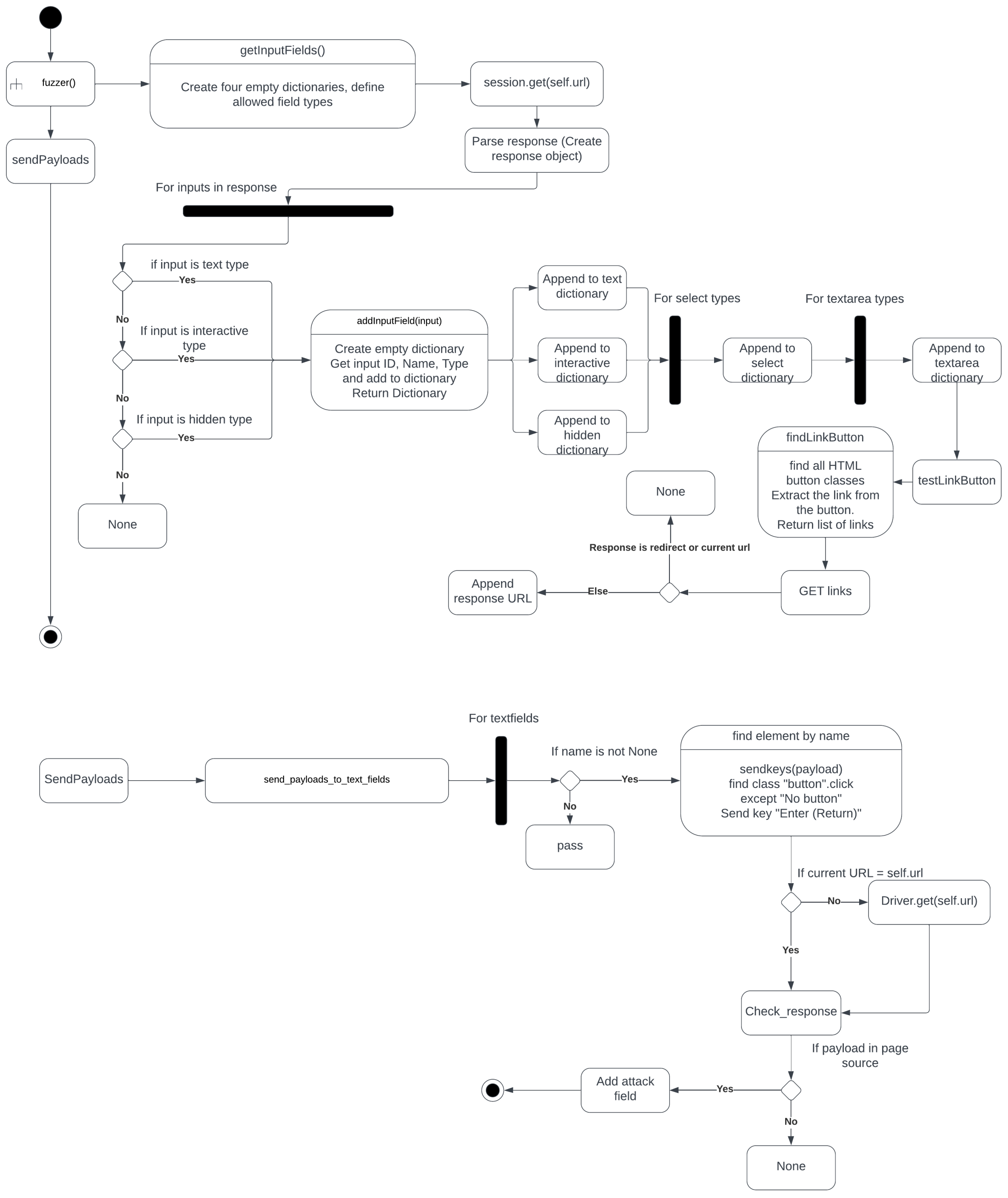
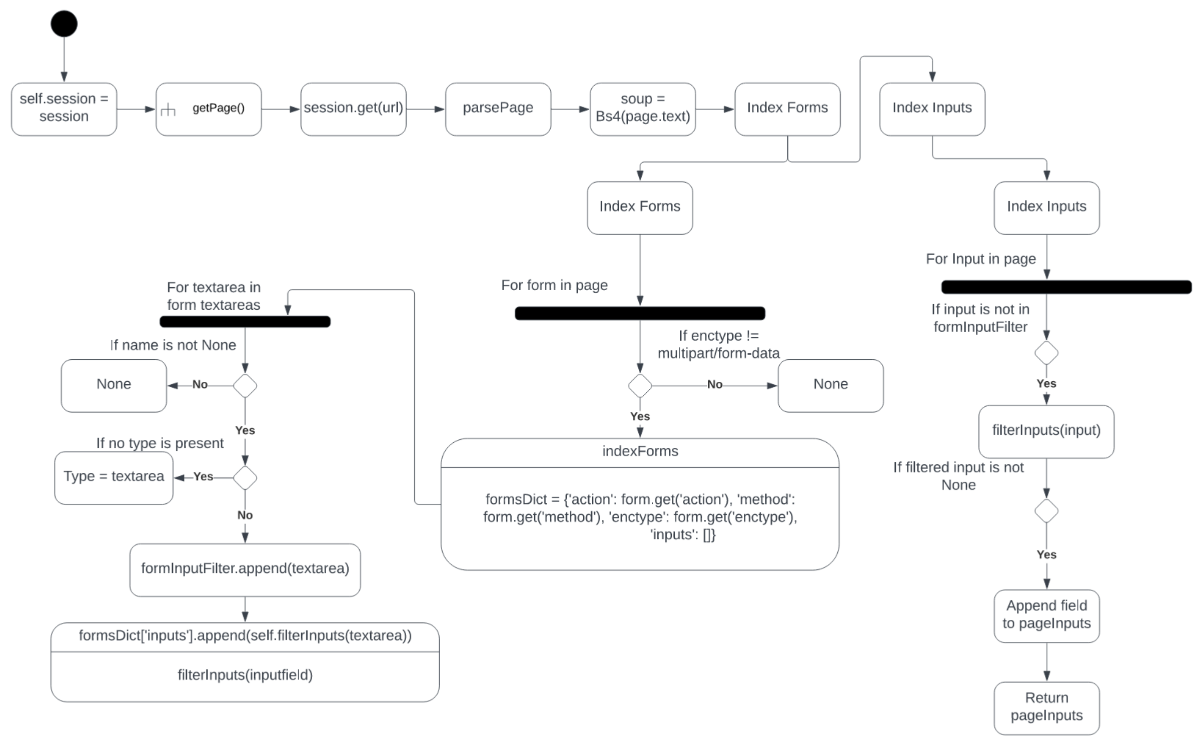

fuzzer()
getInputFields()
Create four empty dictionaries, define allowed field types
session.get(self.url)
Parse response (Create response object)
sendPayloads
For inputs in response
if input is text type
Yes
No
If input is interactive type
Yes
No
If input is hidden type
Yes
No
None
addInputField(input)
Create empty dictionary
Get input ID, Name, Type and add to dictionary
Return Dictionary
Append to text dictionary
Append to interactive dictionary
Append to hidden dictionary
For select types
Append to select dictionary
For textarea types
Append to textarea dictionary
testLinkButton
findLinkButton
find all HTML button classes
Extract the link from the button.
Return list of links
None
Response is redirect or current url
GET links
Else
Append response URL
For textfields
SendPayloads
send_payloads_to_text_fields
If name is not None
Yes
No
pass
find element by name
sendkeys(payload)
find class "button".click
except "No button"
Send key "Enter (Return)"
If current URL = self.url
No
Driver.get(self.url)
Yes
Check_response
If payload in page source
Yes
Add attack field
No
None

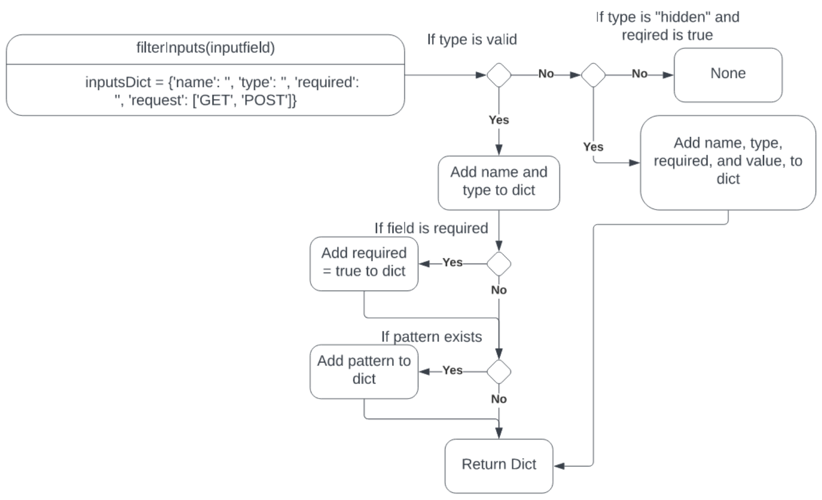
Figure 5.6: First edition of input-field scanner with payload sender

5.2.5 Input-field scanner version 2.0



self.session = session
getPage()
session.get(url)
parsePage
soup = Bs4(page.text)
Index Forms
Index Inputs
Index Forms
Index Inputs
For textarea in form textareas
If name is not None
None
No
Yes
If no type is present
Type = textarea
Yes
No
formInputFilter.append(textarea)
formsDict['inputs'].append(self.filterInputs(textarea))
filterInputs(inputfield)
For form in page
If enctype != multipart/form-data
No
None
Yes
indexForms
formsDict = {'action': form.get('action'), 'method': form.get('method'), 'enctype': form.get('enctype'), 'inputs': []}
For Input in page
If input is not in formInputFilter
Yes
filterInputs(input)
If filtered input is not None
Yes
Append field to pageInputs
Return pageInputs

Figure 5.7: Over arching flow of the Input-field scanner



filterInputs(inputfield)
inputsDict = {'name': '', 'type': '', 'required': '', 'request': ['GET', 'POST']}
If type is valid
No
If type is "hidden" and reqired is true
No
None
Yes
Add name and type to dict
Yes
Add name, type, required, and value, to dict
If field is required
Add required = true to dict
Yes
No
If pattern exists
Add pattern to dict
Yes
No
Return Dict

Figure 5.8: Flow of the input field filtering function

5.2.6 Changes made in v2

The changes done to the input-field scanner were extensive and is explained in section 4.3.1. The flowcharts above represents the final version of the input-field scanner.

5.2.7 Payload sender

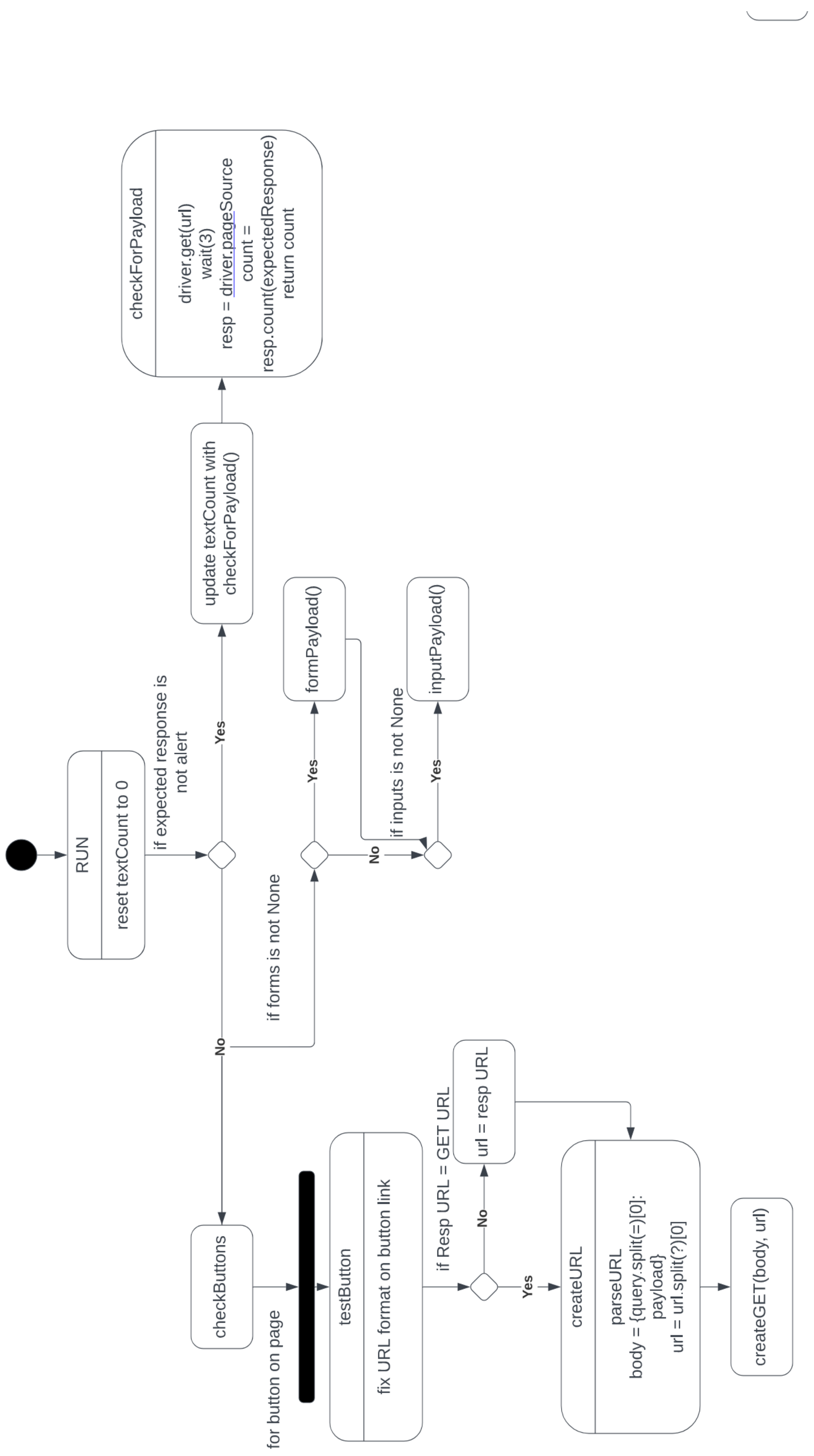


Figure 5.9: Over arching flow of the payload sender

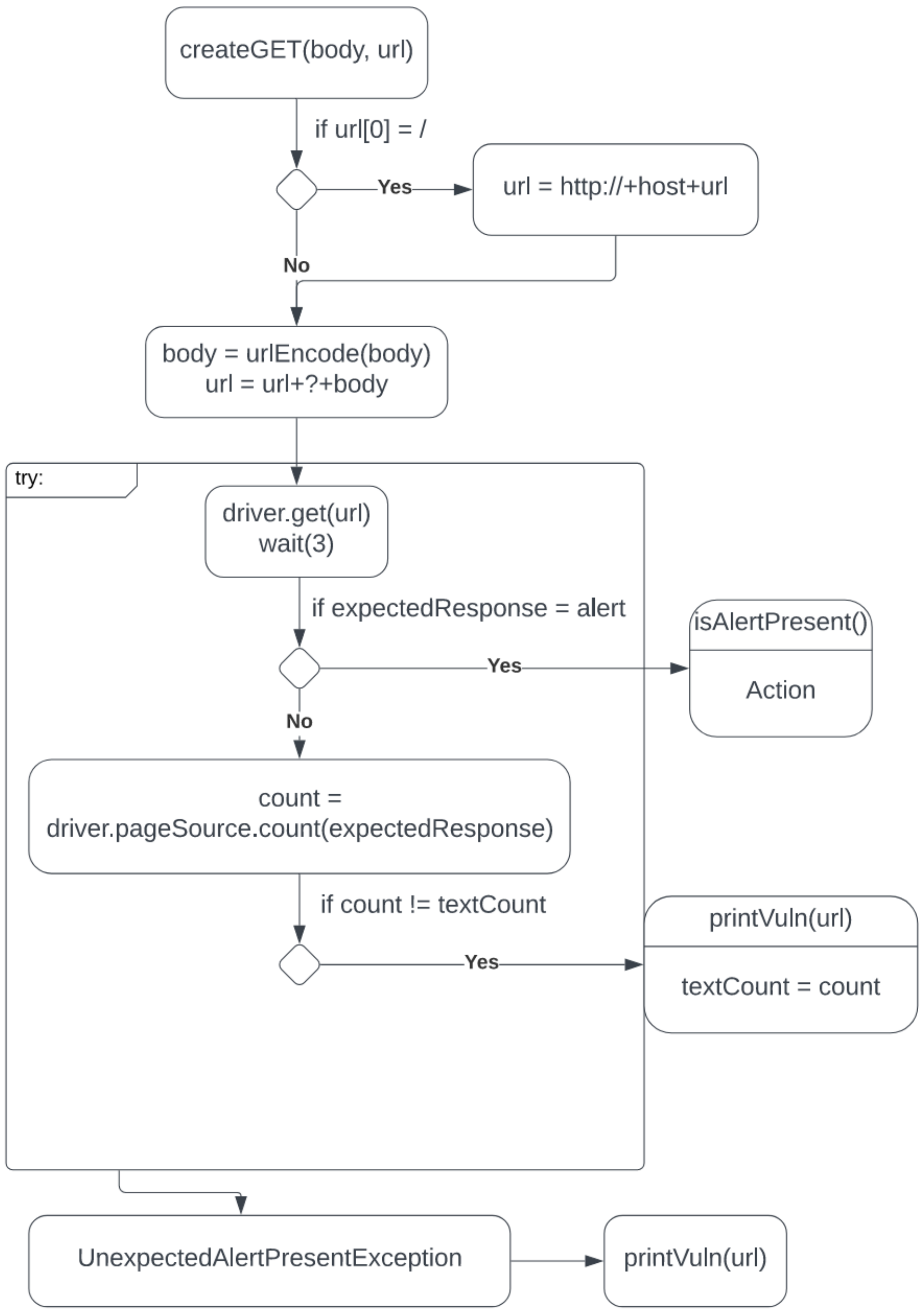
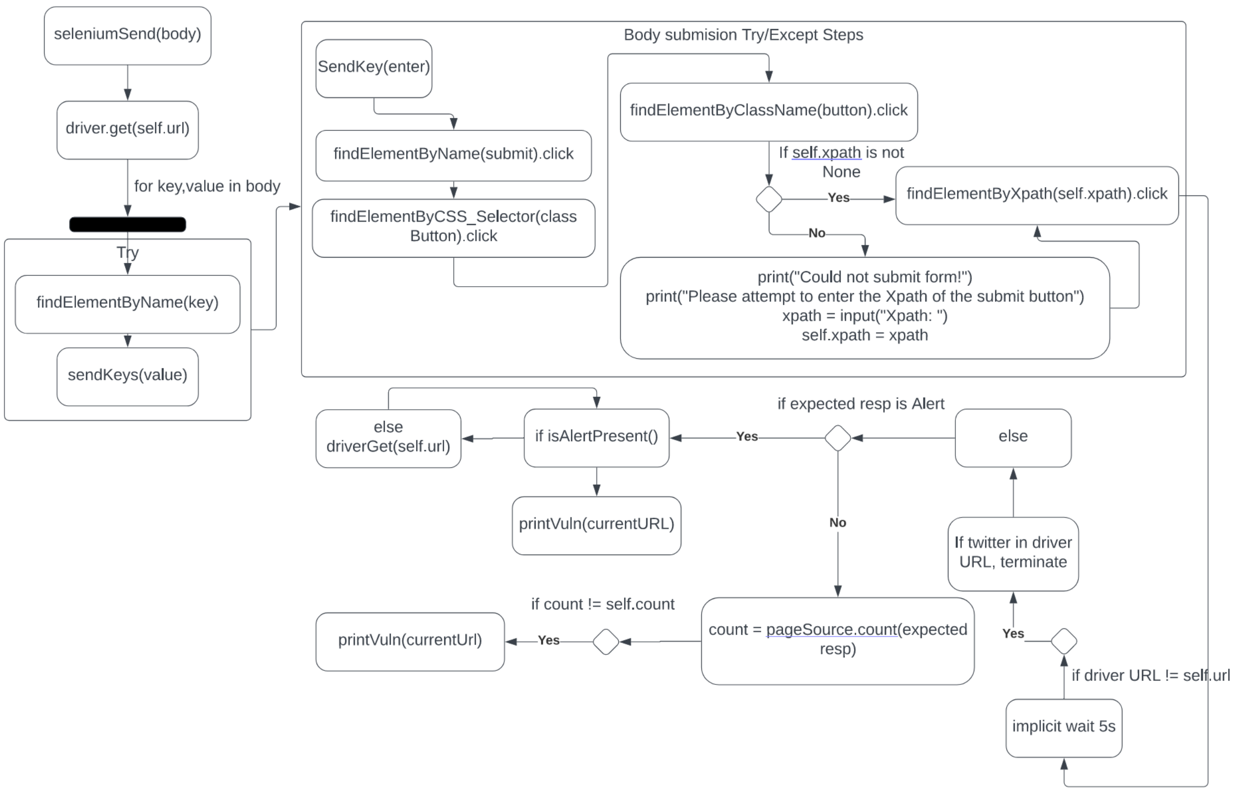


Figure 5.10: Flow of the createGET function



seleniumSend(body)
driver.get(self.url)
for key,value in body
Try
findElementByName(key)
sendKeys(value)
Body submision Try/Except Steps
SendKey(enter)
findElementByName(submit).click
findElementByCSS_Selector(class Button).click
findElementByClassName(button).click
If self.xpath is not None
Yes
findElementByXpath(self.xpath).click
No
print("Could not submit form!")
print("Please attempt to enter the Xpath of the submit button")
xpath = input("Xpath: ")
self.xpath = xpath
else driverGet(self.url)
if isAlertPresent()
Yes
if expected resp is Alert
else
printVuln(currentURL)
No
If twitter in driver URL, terminate
if count != self.count
printVuln(currentUrl)
Yes
count = pageSource.count(expected resp)
Yes
if driver URL != self.url
implicit wait 5s

Figure 5.11: Flow of the seleniumSend function

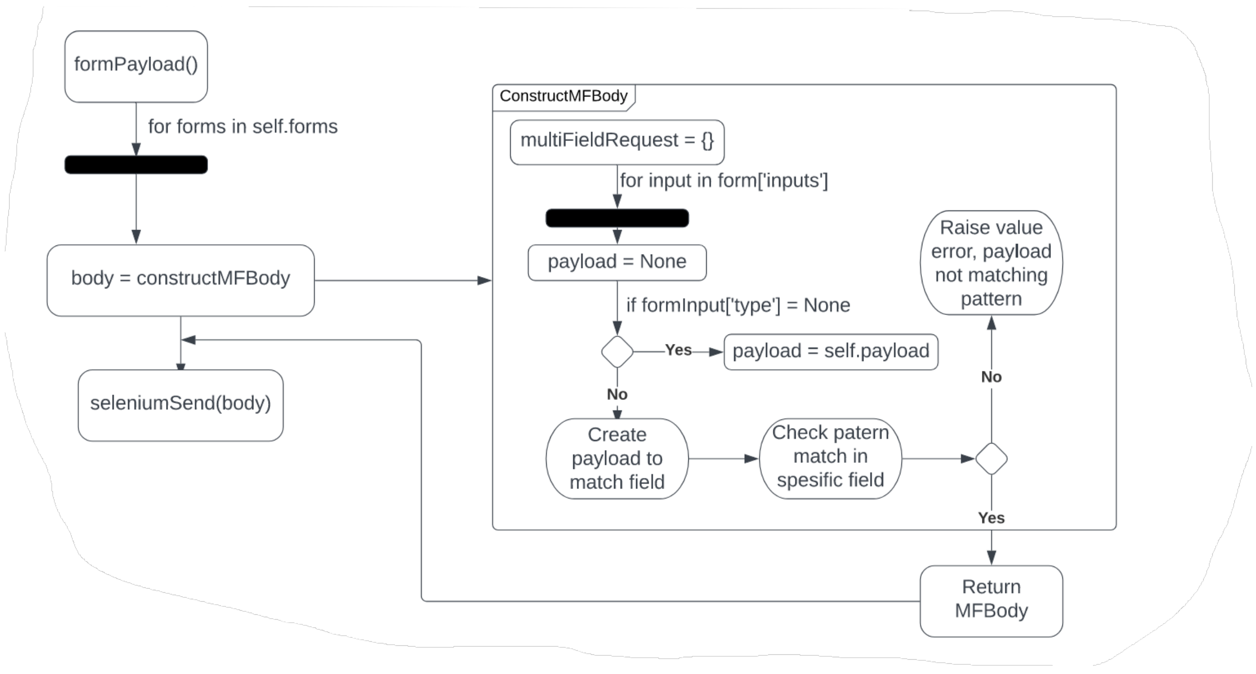


Figure 5.12: Flow of the function for sending payloads to forms

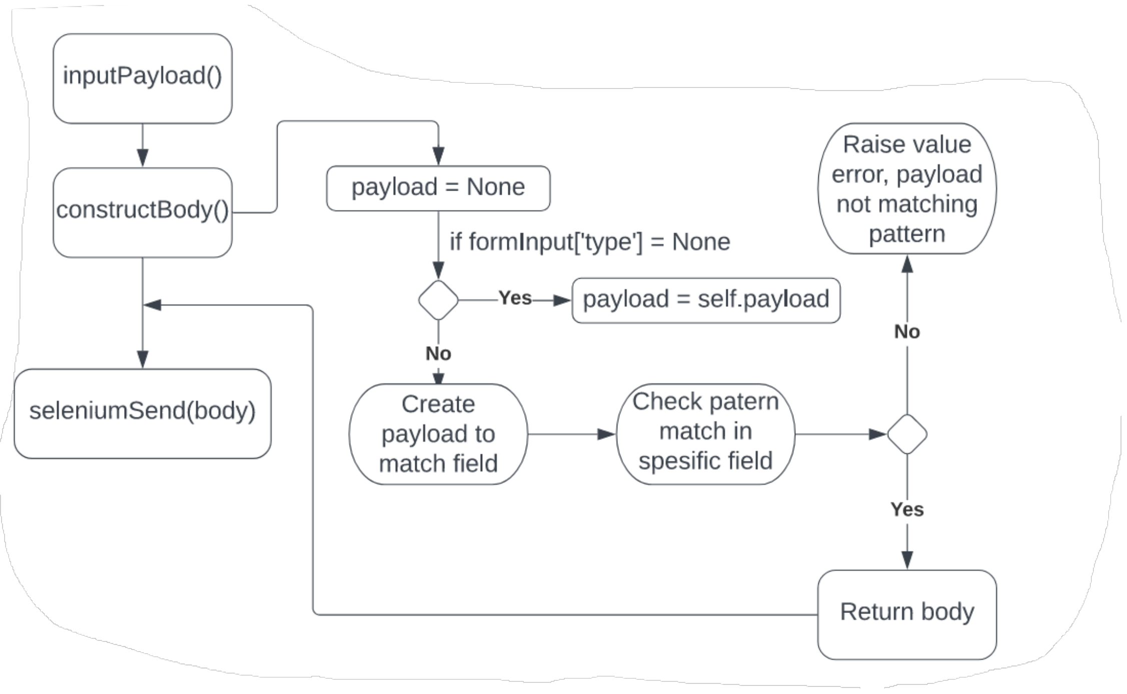
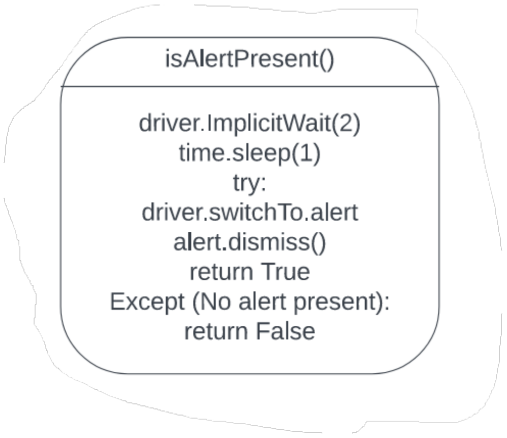

inputPayload()
constructBody()
payload = None
if formInput['type'] = None
Yes
payload = self.payload
No
Raise value error, payload not matching pattern
No
seleniumSend(body)
Create payload to match field
Check patern match in spesific field
Yes
Return body

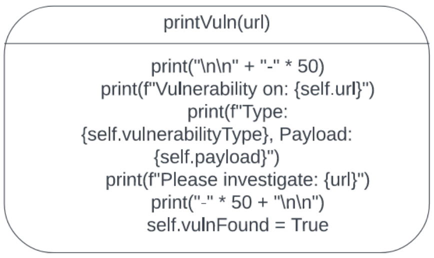
Figure 5.13: Flow of the function for sending payloads to input fields



```
isAlertPresent()

driver.ImplicitWait(2)
time.sleep(1)
try:
driver.switchTo.alert
alert.dismiss()
return True
Except (No alert present):
return False
```

(a) The flow of isAlertPresent function



```
printVuln(url)

print("\n\n" + "-" * 50)
print(f"Vulnerability on: {self.url}")
print(f"Type:
{self.vulnerabilityType}, Payload:
{self.payload}")
print(f"Please investigate: {url}")
print("-" * 50 + "\n\n")
self.vulnFound = True
```

(b) The printVuln function

5.3 Changes made to the payload sender

In section 4.3.1 under Payload Sender, you can find an explanation of the changes made to it. Flow charts for the payload sender were not consistently created during development due to the drastic design changes made in the final stages of the scanner's development. As mentioned in section 4.3.1, various techniques for sending payloads were tested, including the requests library, selenium requests, selenium wire, and using a proxy. The final version of the payload sender uses standard functions from the Selenium library, while the driver from Selenium wire is still utilized.

5.4 Components

5.4.1 Webapp class

The webapp class is responsible for coordinating the scanning process of the web application. It receives the initial URL from the main function and passes it on to the crawler. Once the crawler is executed, the inputfinder and payload sender are systematically run on each page the crawler has discovered. If a vulnerability is detected, the webapp class will display the vulnerable page and the type of vulnerability found.

- `getPages(self)`: Uses a web crawler to visit pages on the website and add them to the list of visited pages. Also identifies the sign-in page if there is one.
- `getInputFields(self)`: Finds all input fields on each visited page and adds them to a dictionary of inputs indexed by a key. Returns the class object.
- `readPayloads(self)`: Reads a JSON file of payloads to be used in testing for vulnerabilities.
- `sendPayloads(self, page, i, payload, response, vulnerabilityType)`: Sends payloads to each input field on a given page and checks the response for the given vulnerability type (e.g. cross-site scripting). Returns True if the vulnerability is found, False otherwise.
- `checkPage(self)`: Runs the payload tests on all visited pages for each vulnerability type, stopping once a vulnerability is found.
- `run(self)`: Runs the main scan by calling `getPages()`, setting up a requests session, calling `getInputFields()`, and calling `checkPage()`.
- `initDriver(self)`: Initializes a web driver (in this case, Firefox) and adds the cookies of the session to it. It returns the web driver object.

5.4.2 Crawler

The vulnerability scanner starts by utilizing the web crawler. The web crawler will use the inputs the user gives to initialize the first steps for the vulnerability scanner.

The Crawler class has several functions, which are:

- `requestUrl(self, url)`: Sends an HTTP GET request to a URL and returns the response text if the request is successful. If the request fails, it logs an error and returns None.
- `getLinkedUrls(self, html, url)`: Parses HTML and extracts all the links and input forms. If a sign-in form is found, it sets the signinPage attribute of the class to the current URL. It yields all the valid links (i.e., links that are in the scope and are not mailto or tel links).
- `crawl(self, url)`: Crawls a URL and adds all the linked URLs to the list of URLs to visit.
- `addUrlsToVisit(self, url)`: Adds a URL to the list of URLs to visit if it has not already been visited or added.
- `isInScope(self, url)`: Extracts the domain, subdomain, and suffix from the URL and checks if the URL is in the same subdomain, domain, and suffix as the scope. If the URL is in the scope, it returns True; otherwise, it returns False.
- `run(self)`: Displays a message with the number of URLs to visit, and while the list of URLs to visit is not empty, it removes the first URL from the list and crawls it. If an error occurs while crawling the URL, it logs the error.

Global functions outside of class:

- `readHeader()`: Reads a JSON file that contains HTTP request headers and returns the headers if the file exists. If the file is not found, it returns None.
- `readScope()`: Reads a JSON file that contains a scope for the crawler and returns the scope if the file exists. If the file is not found, it returns None.

5.4.3 Input finder

The input finder expands upon the data that the web crawler has collected. The input finder scans for all types of inputs in the pages that the web crawler has crawled.

The inputFinder class has the following functions:

- `parsePage()`: This function takes the page content as input and uses BeautifulSoup library to parse the HTML code of the page. It then searches for all the forms and inputs within the HTML code and filters out the unwanted inputs using the `filterInputs` function. It returns a list of inputs that have been filtered.
- `indexForms()`: This function takes a single form as input and indexes its attributes such as action, method, and inputs. It uses the `filterInputs` function to filter out the unwanted inputs, and then appends the filtered inputs to the indexed form in the class. The indexed form is then appended to the list of forms in the class.
- `filterInputs()`: This function takes an input field as input and checks if its type attribute is valid. If the type attribute is valid, it creates a dictionary of the input field's name, type, and required attributes, and then returns the dictionary. If the input field is of type "hidden" and has a "required" attribute, it also adds the input field's value to the dictionary.
- `getPage()`: This function sends a GET request to the URL provided in the constructor and saves the response in the class variable "page". If an error occurs, it prints an error message.
- `run()`: This function takes a session object as input and sets it as the session for the class. It then gets the page content using the `getPage` function, parses the page using the `parsePage` function, and returns the list of filtered inputs.

5.4.4 Payload sender

The payload sender uses the input fields found in the input finder and formats and sends payloads accordingly. The payload sender systematically checks for one vulnerability type at a time and is orchestrated by the webapp class. After sending a payload to a page, the page is checked for payload execution. If the payload is triggered, information about the vulnerable page is printed to the console.
The payload sender class has the following functions:

- `getCookies(self)` : Returns a dictionary of cookies from the session.
- `noInputGet(self, anomalys)` : Sends payloads using GET requests to each of the anomalies in the `anomalys` list. If a payload triggers a vulnerability, information about the vulnerability is printed to the console.
- `sendPayloadForm(self, body, url, form)` : Sends a payload using either a POST or GET request, depending on the form's method attribute. If a payload triggers a vulnerability, information about the vulnerability is printed to the console.

- `formatAction(self, action)` : Takes an action URL and constructs a complete URL using either the page URL or the base URL, depending on the format of the action URL. Returns the complete URL.
- `formPayload(self)` : Sends payloads to each form on the page using `sendPayloadForm()`.
- `inputPayload(self)` : Sends a payload to the input field on the page using `sendPayloadForm()`.
- `constructHead(self, url, body, enctype)` : Constructs and returns a dictionary of headers for a request, using information from the session and the form.
- `createPOST(self, body, url, enctype=None)` : Sends a POST request to a URL with a body containing a comment field with the payload. Returns the response object.
- `createGET(self, body, url)` : Sends a GET request to a URL with a query string containing the payload. If the expected response is an alert, checks for an alert and prints information about the vulnerability if one is found. Otherwise, checks for an increase in the count of a specified text string and prints information about the vulnerability if the count has increased.
- `constructMFBody(self, form, method)` : Constructs and returns a dictionary of form input names and values for a multi-field request, using the form's inputs and the payload.
- `constructBody(self)` : Constructs and returns a dictionary containing a comment field with the payload.
- `seleniumSend(self, body)` : Sends a request using Selenium and checks for a vulnerability using `checkExecution()`.
- `checkButtons(self)`: This function searches the HTML source for any buttons with specific classes and calls `testButtons` on each one.
- `initDriver(self)`: This function initializes a Firefox driver with specified options and adds cookies from a previous session, if applicable.
- `printVuln(self, url)`: This function prints information about a vulnerability found in the HTML source.
- `isAlertPresent(self)`: This function checks if an alert is present on the page and dismisses it if so, calling `printVuln` on the current URL and returning `True` if an alert is found.
- `checkForPayload(self)`: This function loads the page and counts the instances of the expected response in the page source, returning the count.
- `run(self)`: This function runs the payload sender, calling `checkForPayload`, `checkButtons`, `formPayload`, and `inputPayload` as appropriate.

Global functions outside of the class:

- `readWordlist()` : Reads and returns the contents of a file named "wordlist.txt" as a list of payloads.
- `checkExecution(response)` : Takes a response object and checks if it contains a script tag with the string "alert(1)". Returns `True` if it does, `False` otherwise.

5.4.5 Get payloads from JSON sorted by type and expected output

The scanner comes with an extensive payload library that can be used to test web apps for vulnerabilities. The payloads are sorted according to the vulnerabilities they intend to attack. The sorting of the payloads makes it much simpler to identify particular vulnerabilities and target parts of the web application that might be more vulnerable to particular attacks. The payloads' ability to generate a specific reply when a web application is deemed insecure is another crucial feature. It enables the scanner to find vulnerabilities quickly and give the correct feedback.

5.5 Limitations and constraints

The web application vulnerability scanner has several limitations that can impact its effectiveness. Firstly, when using the "Submit" selection, the scanner will only choose the first button on the web page. This can be a significant limitation, as it may prevent the scanner from submitting data if there is more than one button on the page. Additionally, the scanner can only select inputs with a name, which may also impact its ability to test all fields in a web application.

Another issue with the scanner is its lack of fuzzing capabilities. It does not send random or unexpected data to check for vulnerabilities, and it also does not care about the technology being used in the web application. Instead, it tests until a vulnerability is found or all payloads have been tested. While this approach may be effective in identifying some vulnerabilities, it may not be sufficient in all cases.

When testing for Stored Cross-Site Scripting (XSS) vulnerabilities, the scanner can create a lot of garbage content on a page. This is not a flaw in the scanner itself but rather a result of how Stored XSS vulnerabilities work.

The scanner's crawler is also limited in its ability to find links on a web page. It can only find links in the page text that matches the REGEX pattern, which may miss some URLs and is not perfect.

Additionally, the scanner's use of the JSON file format can inhibit the use of Common Vulnerabilities and Exposures (CVE) Proof of Concepts (PoCs) directly. This is because the JSON file format does not allow for multi-line payloads, which may limit the scanner's effectiveness in testing for certain vulnerabilities[10].

Finally, the scanner may falsely identify vulnerabilities on pages with alerts, which can lead to inaccurate results.

Chapter 6

Results

6.1 Scanner

6.1.1 Dynamic scan

The vulnerability scanner follows the principles of dynamic vulnerability scanning. This approach of a dynamic vulnerability scan leads to the steps needed by the vulnerability scanner that this chapter will explain. As mentioned in Chapter 2(cite here later), a dynamic scan involves sending a request to a web application and analyzing the responses that it gives. The present system employs a dynamic scanning technique that involves testing on a live-running web page using a standard web browser, namely Firefox. The selection of Firefox as the browser is not based on any particular reason, and other commonly used browsers such as Chrome, Edge, or Safari could have been used. It is noteworthy, however, that Selenium offers more features and better support for Chrome and Firefox[36][17].

All the testing procedures are executed within the confines of the program-controlled browser, simulating the actions of a typical web application user. The payloads are executed on a platform identical to that in a standard Firefox instance, utilizing the same gecko driver that commercial Firefox uses, thus rendering the scanning process as realistic as possible. This characteristic is highly desirable as it enables the most accurate testing of client-side vulnerabilities such as XSS and CSTI[25].

6.1.2 Configuring the scanner

Before running the scanner, there are two options that need to be configured. Additionally, the scanner has a head.json file that can be modified to change the HTTP header for the Crawler. This is commonly done to ensure that the user agent is identifiable. It is standard practice for web crawlers to have an identifiable user agent[28].

The scanner will prompt for two inputs at the start of each run. The first input is the webpage URL that needs to be scanned. The second input is a question about changing the scope. If the answer is (y), the scanner will prompt the user for the domain, subdomain, and suffix in that specific order. These details are saved in the scope.json file that the Crawler will use. The Crawler will not attempt to connect to any pages outside of this scope limitation. This is an essential measure to prevent the scanner from accessing pages without the owner's permission, which could result in legal action against the scanner user[45].

The user also has the option to edit the payloads.json file to their needs. However, further source code editing will be required to add more payload classes, e.g., SQLi. For the currently implemented classes (XSS, DOM-XSS, and CSTI), one only needs to add more payloads with expected responses. It is worth noting that the payloads must adhere to the restrictions and limitations of the JSON format[10].

6.2 Running the scanners

All the scanners ran with minimal setup; the only manual action required was to input a URL into the scanner, and all tests will run automatically. However, it is worth noting that running Our Scanner needs to be run twice because of a bug where the updated scope does not save before the program terminates, making the Crawler fail on the first attempt.

The run-time for the scanners varied a lot, with Vega being the fastest and Wapiti3 being the slowest by far. In our testing, the scanner's speeds are here ranked from fastest to slowest:

1. Vega
2. Iron Wasp
3. Our Scanner
4. OWASP Zap
5. Wapiti3

6.3 Scanning results

Vulnerable apps	Iron Wasp			Wapiti3			OWASP Zap			Vega			Our Scanner		
	FP	TP	Misses	FP	TP	Misses	FP	TP	Misses	FP	TP	Misses	FP	TP	Misses
DOM XSS in AngularJS expression with angle brackets and double quotes HTML-encoded	0	0	1	3	0	1	5	1	0	0	0	1	0	1	0
Basic server-side template injection	0	0	1	3	0	1	5	0	1	0	0	1	0	1	0
Stored DOM XSS	0	0	1	3	0	1	5	1	0	0	0	1	0	1	0
DOM XSS in document.write sink using source location.search inside a select element	1	0	1	3	0	1	5	1	0	0	0	1	0	0	1
Reflected XSS into a JavaScript string with angle brackets HTML encoded	0	1	0	3	0	1	5	1	0	0	1	0	0	1	0
Reflected XSS into attribute with angle brackets HTML-encoded	0	1	0	3	1	0	5	1	0	0	1	0	0	0	1
Reflected XSS into HTML context with nothing encoded	0	1	0	3	1	0	5	1	0	0	1	0	0	1	0
Stored XSS into HTML context with nothing encoded	0	0	1	3	0	1	5	0	1	0	0	1	0	1	0
DOM XSS in document.write sink using source location.search	0	0	1	3	0	1	5	1	0	0	0	1	0	1	0
DOM XSS in innerHTML sink using source location.search	1	0	1	3	0	1	5	1	0	0	0	1	0	1	0

Table 6.1: Scan results for all 5 scanners tested

Scanner	True Positives Found	Vulnerabilities Missed
Iron Wasp	30%	70%
Wapiti3	20%	80%
OWASP Zap	80%	20%
Vega	30%	70%
Our Scanner	80%	20%

Table 6.2: Vulnerability detection percentage for the web-applications tested

6.4 Scanner functionality results

6.4.1 Template injection

IronWASP	OWASP ZAP	Vega	Wapiti	Our scanner
No functionality	Limited functionality	No functionality	No functionality	Good functionality

In the course of developing an in-house vulnerability scanner for web applications, a test was conducted to assess the effectiveness of several open-source scanners in detecting template injection vulnerabilities. The scanners evaluated in this test were IronWASP, OWASP ZAP, Vega, Wapiti, and the in-house scanner.

The results of the test showed that IronWASP, Vega, and Wapiti did not demonstrate any functionality for detecting template injection vulnerabilities, as they detected no vulnerabilities in the relevant labs, see table 6.1. OWASP ZAP demonstrated limited functionality in detecting such vulnerabilities, being able to detect vulnerable versions of the templating engine in use by the web-application. However, it did not identify a simple Server Side Template Injection (SSTI) in the "Basic server-side template injection" lab as shown in table 6.1. Here the application was running a vulnerable Embedded Ruby (ERB) implementation. Zap was unable to identify the vulnerability. Therefore, it does not get marked as good functionality for template injection.

In contrast, the in-house scanner demonstrated good functionality in detecting template injection vulnerabilities. This suggests that the in-house scanner may be a more effective tool for identifying this particular type of vulnerability in web applications than the open-source scanners tested.

The in-house scanner developed in this study appears to be a promising option for detecting template injection vulnerabilities, but further testing is needed to assess its effectiveness on applications vulnerable to template injection.

6.4.2 XSS

IronWASP	OWASP ZAP	Vega	Wapiti	Our scanner
Limited functionality	Limited functionality	Limited functionality	Limited functionality	Limited functionality

As part of the development of an in-house vulnerability scanner for web applications, a test was conducted to evaluate the effectiveness of several open-source scanners in detecting cross-site scripting (XSS) vulnerabilities. The scanners evaluated in this test were IronWASP, OWASP ZAP, Vega, Wapiti, and the in-house scanner.

The results of the test indicated that all of the scanners had limited functionality in detecting XSS vulnerabilities. Specifically, IronWASP, OWASP ZAP, Vega, Wapiti, and the in-house scanner all demonstrated limited functionality in detecting XSS vulnerabilities.

These findings suggest that additional development and testing may be necessary in order to improve the effectiveness of the in-house scanner and the open-source scanners tested in detecting XSS vulnerabilities. Further research into the development of more effective techniques for detecting XSS vulnerabilities may be necessary to improve the overall effectiveness of vulnerability scanners for web applications.

Overall, these results highlight the ongoing challenge of detecting XSS vulnerabilities in web applications, and the need for ongoing research and development to improve the effectiveness of vulnerability scanning tools.

6.4.3 DOM-XSS

IronWASP	OWASP ZAP	Vega	Wapiti	Our scanner
No functionality	Good functionality	No functionality	No functionality	Limited functionality

The test results indicated that IronWASP, Vega, and Wapiti did not exhibit any functionality in detecting DOM-based XSS vulnerabilities. As a result, these scanners detected no vulnerabilities in the relevant labs, as illustrated in Table 6.1. On the other hand, our in-house scanner demonstrated limited functionality in detecting such vulnerabilities, being able to detect only one of the four apps that were vulnerable to DOM-based XSS in this test.

In contrast, OWASP ZAP demonstrated good functionality in detecting such vulnerabilities, detecting all cases of DOM-based XSS vulnerabilities in our vulnerable apps. These results suggest that our in-house scanner still has room for improvement to enhance its ability to detect DOM-based XSS vulnerabilities.

6.5 Notes about the scanning

All the applications are based on the same framework and function and look almost identical. Therefore the same False Positive (FP) has been identified by Wapiti3 and Zap. This hurts the accuracy ratings of the scanners and is not necessarily a realistic figure. All pages have only one vulnerability; therefore, the True Positive (TP) numbers and misses are always 1 or 0, giving the FP a comparatively high score. Unfortunately, this also skews the results when comparing the scanners statistically from these few tests, making it wholly unrealistic and more a test of the scanner's functionality rather than performance. Because of this skew, it has been decided not to include performance statistics in the report, as this would be unfair and biased toward specific scanners.

Also worth mentioning is that Our Scanner was developed using these types of labs. Therefore, its functions were developed to solve the more basic labs, and payloads to solve more advanced labs were added to the payload list. Some of the payloads are taken from PortSwigger, and might, therefore, be ideally suited to one or more of these labs. This gives a substantial bias in the detection results, as mentioned making it unfair to compare the performance statistics in the conclusion of this report.

Chapter 7

Discussion

In this section, we will cover the final version of the scanner. We will discuss the results of the testing and its performance, along with previous research on vulnerability scanners that are already available. We will also go over the limitations of the final version and explain the reasons for the results. Finally, we will provide suggestions for further improvements to the scanner.

7.1 Results

After testing the scanner against free, open-source scanners, it has become evident that there is room for improvement in the market. Additionally, this highlights the continued need for expertise in conducting thorough manual vulnerability analysis of web applications. While there are several available scanners, including the one we developed, they are limited by certain web design features. For example, only one scanner was able to detect a vulnerability within a select element input. In contrast, all scanners detected a reflected XSS vulnerability in a search field on the page.

We discovered a need for improved functionality in detecting template injection vulnerabilities. Our scanner was the only scanner to identify the two template injection vulnerabilities tested correctly. Only one other scanner was able to detect one of the vulnerabilities. However, it only picked up on the page's outdated and vulnerable library rather than the actual vulnerability itself. These vulnerabilities were not difficult to detect and could be exploited with simple one-line payloads that anyone could copy and paste. The fact that no other scanners besides ours were able to detect these vulnerabilities highlights a significant gap in the vulnerability scanner market, which we have helped to close with our scanner.

Our results coincide with other studies done on the subject of automatic vulnerability scanners. It shows that the current detection level needs to be improved and that there is a large room for improvement regarding different types of vulnerabilities. Other research also displays these issues, getting similar rates of missed vulnerabilities and false positives[1][21].

7.2 The scanner

7.2.1 Crawler

To save time, we used a pre-existing crawler from scapingbee.com as the foundation for our own. We made necessary adjustments, such as implementing support for a scope and detecting URLs within page text, to meet our specific requirements.
The crawler developed for this project is a crucial component of the in-house vulnerability scanner for web applications. It is responsible for identifying all the URLs that need to be

scanned and ensuring that only approved pages are scanned. The crawler uses Beautiful Soup to extract all the URLs linked to anchor tags within an HTML page. This approach has proven to be effective in identifying URLs that are approved for scanning. However, it does have some limitations, mainly with the parsing methods used to extract URLs.

One of the primary limitations of the initial crawler was that it only extracted URLs from anchor tags within a page. This meant that any URLs that were present in the text but not hyperlinked were missed. To address this limitation, an additional feature was added to the crawler in the second iteration. The crawler was modified to include the functionality of searching the page text to identify URLs that are not noted with an anchor tag. This is done using a regex pattern, and any URLs that match the pattern are added to the list of URLs to scan.

While the regex pattern used in the crawler is not perfect, it should capture most URLs present in the page text. This feature is expected to improve the overall effectiveness of the crawler by identifying all URLs on the page, whether hyperlinked or not. However, it is worth noting that this feature may also result in some false positives, which would cause an error while scanning.

7.2.2 Input finder

The input finder was built from the ground up, using requests and beautiful soup for respectively retrieving and parsing pages. Both libraries are the go-to for these tasks in Python, as they are the most extensive and well-supported for their respective areas.

Its main purpose is to expand upon the data collected by the web crawler by scanning for all types of inputs in the web pages crawled. The inputfinder consists of several functions that work together to extract information about input fields in forms and filter them based on their type and required attributes.

However, the input finder is limited by design to ignore some input fields. This is reflected in the results, as it could not discover a vulnerability in a selection field element. The decision to filter out field types like selection fields, radios and checkboxes is made primarily because of time constraints. They would require more functionality added to the payload sender to support all these types.

7.2.3 Payload sender

The payload sender was first built into the input finder and used the selenium library. Then, to simplify the sender, a standalone class was created. This class was first intended to use the requests library to send the payloads, as it gives complete control of the whole request, from method to header and body. However, after implementing and testing this design, a limitation from using the requests library was discovered. The requests library does not support running java-script on the web application. The lack of java-script support is a major issue, as it removes the functionalities to check for DOM-based XSS payloads, as they are stored in the DOM and not in the pages java-script. This method was still functional for normal XSS payloads, as you could look for alerts in the page java-script code. However, it would not know if the code would be executed, leading to possible false positives.

To solve this challenge, different methods were tested. First, we tried using a proxy to redirect all the traffic from the current page to a Selenium instance without success. Then we attempted to use selenium-wire, which has its own proxy to intercept requests before they were sent to the server. However, it was quickly discovered that this implementation needed to be improved, as it did not allow sending POST requests with payloads. It was clear that a different approach was required, so we ended up reverting to a system similar to the first iteration of the payload sender. Finally, we decided to use Selenium again. This was a hard decision, as we thought a lot of code needed to be rewritten. However, a lot of our code could be easily reused, and the implementation of the new system went faster than expected. The new implementation now uses an API-controlled browser with Mozilla's Gecko driver. This means it functions almost identically to any Firefox browser, enabling it to run javascript, which was the main reason for changing from Requests to Selenium. There are still limitations; we would ultimately like complete control of the requests sent. However, as of now, this functionality is not available for Selenium.

7.3 Testing applications

During the development of the vulnerability scanner, the team faced several challenges and made important decisions that shaped the project's direction. For example, one of the initial ideas was to develop vulnerable web applications to test for template injection vulnerabilities. This approach would require web applications that use different template engines, which would enable testing the scanner's effectiveness against various scenarios.

However, the development of these vulnerable web applications was canceled early in the project for several reasons. Firstly, the team needed to gain experience in web development, and learning the necessary skills would require considerable time and resources. Secondly, there are other options for web application testing that already exist, such as using pre-existing vulnerable web applications or using testing platforms like PortSwigger labs.

7.4 Testing bias

The issue of testing bias is an important consideration in the evaluation of our in-house vulnerability scanner. The scanner was thoroughly tested against PortSwigger labs throughout the development process to ensure its functionality. We chose these labs because they offered a wide range of vulnerabilities, were free, and available on demand. In addition, to test the payload execution detection, we added lab solution payloads to the list of payloads. This likely contributed to the scanner's improved performance compared to the other scanners tested. However, it's worth noting that most of the labs utilized during final testing were not part of the testing during the development process.

While our scanner has demonstrated promising results in detecting vulnerabilities in web applications, it is essential to recognize that there may be limitations to its performance when testing against different environments or scenarios. Thus, the scanner's effectiveness in detecting vulnerabilities in real-world scenarios needs to be further evaluated and tested to determine its overall reliability.

7.5 Future research

The first step for future research on the scanner is to improve on the limitations and constraints of the current version, mentioned in 5.5.

- Enhanced Form Submission: Future research can focus on developing algorithms that intelligently identify and interact with multiple buttons, allowing the scanner to submit data even in scenarios where multiple buttons are present. Furthermore, expanding the capability to select inputs without a name attribute can improve the coverage of field testing within web applications.

- Advanced Fuzzing Techniques: Incorporating advanced fuzzing capabilities into the scanner can significantly enhance its ability to detect vulnerabilities. Research efforts can explore the integration of fuzzing techniques that send random or unexpected data to the application, testing for potential vulnerabilities beyond the limited payload set.
- Intelligent Crawler Enhancements: Advanced crawling algorithms that employ advanced text analysis and link extraction techniques can be developed to identify URLs more accurately, thereby enhancing the coverage of the vulnerability scan. In addition, exploring alternative approaches, such as leveraging machine learning algorithms for link identification, can further improve the crawler's efficiency.

Further, one area of future research is developing a more extensive payload library. Currently, the vulnerability scanner can detect common web application vulnerabilities, but a more extensive payload library would increase the likelihood of detecting more advanced vulnerabilities.
Another potential area for future research is the addition of functionality to check the page technology used by the web application. By identifying outdated or vulnerable versions of back-end and front-end frameworks and libraries, the scanner could alert to potential vulnerabilities associated with these outdated or vulnerable technologies.

Expanding the functionality of the payload sender to support all types of input fields could also be an area of further research. Currently, the payload sender is limited to text type fields and buttons. It is possible to have vulnerabilities in selection type fields as well.

Finally, re-designing the payload-sending system to have more control over the request attributes sent to the web application is another area that could be explored in future research. This could potentially allow for more effective testing of input validation vulnerabilities, as well as provide greater flexibility in customizing requests. In order to do this, however, a more advanced library than Selenium is needed.

Chapter 8

Conclusion

This thesis has addressed the need for a vulnerability scanner to detect Template Injection, XSS, and DOM-based XSS vulnerabilities in web applications. This thesis aimed to develop a modular and extensible vulnerability scanner that Netsecurity's RedTeam can use to increase the efficiency and accuracy of security assessments.

Based on the results from testing the scanner against other free and open-source solutions on the market, it can be concluded that it helps close the vulnerability scanner market gap. The results show that the in-house scanner out performed all available scanners on Template injection vulnerabilities while beating nearly all on XSS-type vulnerabilities.

While the scanner has limitations, it has been proven that technology can be improved even under strict time constraints. Additionally, focusing on specific injection vulnerabilities rather than attempting to detect all vulnerability types can result in better performance in certain areas. Although creating a particular scanner, or scanner function for each vulnerability type may require more time and resources, it can significantly enhance accuracy and detection rates.

To better understand the implications of these results, future studies could address the practical applications and benefits of using modular and specialized vulnerability scanners in real-world scenarios. This could include testing the scanner on a larger scale, and in various environments, to determine its effectiveness in identifying and addressing vulnerabilities in different web applications. Additionally, conducting comparative studies on the use of different vulnerability scanners, including commercial and open-source solutions, can provide a better understanding of the strengths and weaknesses of each approach.

Further research could also explore the potential impact of incorporating machine learning algorithms into vulnerability scanners. This could enhance the accuracy and efficiency of the scanner by automatically detecting and classifying new types of vulnerabilities as they emerge. Additionally, investigating ways to integrate vulnerability scanning with other security tools and systems, such as intrusion detection and prevention systems, can help to provide a more comprehensive security solution for web applications.

In conclusion, this thesis has demonstrated the feasibility and benefits of developing a specialized vulnerability scanner for detecting Template Injection, XSS, and DOM-based XSS vulnerabilities in web applications. The results indicate that a modular and specialized approach to vulnerability scanning can improve detection rates and accuracy, and help to close the gap in the vulnerability scanner market. Future studies can build on these findings by exploring the practical applications and benefits of this approach in real-world scenarios, and by investigating ways to further enhance the accuracy and efficiency of vulnerability scanning.

Appendix A

Functional Requirements Document (FRD)

Introduction

This Functional Requirements Document aims to develop a vulnerability scanner that will detect vulnerabilities in web apps, specifically the client-side template injection vulnerability. In addition, this app can also crawl the domain to establish if there are any other pages in the web app that could be vulnerable. The scanner is being developed for Netsecurity's red team and will prompt the user for input if and when it finds sign-in fields. The scanner will be developed using Python and rely on the BeautifulSoup, Selenium, and Requests libraries.

Assumptions and Constraints

The following assumptions and constraints have been identified for the development of the vulnerability scanner:

- The scanner will only be able to detect vulnerabilities in web applications that are vulnerable to client-side template injection.
- The scanner can give the user the oppertunity to sign in manually. This could also be done automatically by the app.
- The scanner will be developed using Python and the BeautifulSoup, Selenium, and Requests libraries.
- The scanner will stay within the given domain. Even if it finds a link when crawling that is outside the given domain, it will not follow it. Its limited to the given scope, not only domain. Defined in the config field. In the config the user can set a header, and scope. The scope cannot limit directories(page directories).
- The scanner's user interface will change values within the code itself for now.
- Stig Jensen and Marius Hauger will develop the scanner, and no other individuals will be involved in the development.

Quality Assurance and Testing

The following quality assurance and testing processes will be implemented for the vulnerability scanner:

- Development meetings with Netsecurity's red team to demonstrate the current status of the scanner and receive feedback on its effectiveness and usability.
- Continuous testing under the development phase to identify and address any issues or bugs as they arise.
- Development of test cases for the scanner will include testing against web applications with different levels of complexity and different vulnerability types.

Stakeholders

The following are the stakeholders involved in the development and implementation of the vulnerability scanner:

- Netsecurity Red Team: The primary users of the vulnerability scanner, who will use it to identify and report vulnerabilities in web applications.
- Stig Jensen and Marius Hauger: These two individuals are responsible for the development of the vulnerability scanner and will be responsible for ensuring that the scanner meets technical requirements and industry standards.
- Netsecurity Red Team Customers: Customers who purchase services from the Netsecurity Red Team may be interested in the scanner's capabilities and effectiveness in identifying vulnerabilities in their own systems and applications.

Technical Requirements

The following are the technical requirements for the vulnerability scanner:

- Programming Language: The scanner will be developed using Python.

- Libraries: The scanner will rely on the following libraries:
    - BeautifulSoup: A Python library for web scraping and parsing HTML and XML documents.
    - Selenium: A suite of tools for automating web browsers. The scanner will use the Selenium WebDriver API to interact with web pages and detect vulnerabilities.
    - Selenium WebDriver: A component of Selenium that provides a programming interface to control web browsers. The scanner will use the WebDriver API to crawl web pages and detect vulnerabilities.
    - Requests: A Python library for making HTTP requests. The scanner will use the Requests library to send HTTP requests to web pages and receive responses.
- System Requirements: The scanner should be compatible with the latest stable version of Python and the libraries listed above. The minimum system requirements for running the scanner will depend on the size and complexity of the web application being scanned.

Deployment

The **deployment** process for the vulnerability scanner will involve the following steps:

- **Installation**: The scanner will be installed as a Python module on the Netsecurity Red Team's systems.
- **System Requirements**: The system requirements for running the scanner will depend on the size and complexity of the web application being scanned.
- **Configuration**: The scanner will be configured to scan the specific domain and web application(s) as specified by the Netsecurity Red Team.

Maintenance and Support

The following information will be provided to ensure that the vulnerability scanner can be easily maintained and supported:

- **Troubleshooting**: Information on how to troubleshoot any issues that may arise when using the scanner will be provided.
- **Updates**: Information on how to update the scanner to the latest version will be provided.

Technical Support: THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Risk Assessment and Management

The following **risks** have been identified for the development and implementation of the vulnerability scanner:

- **Technical Risks**: The scanner may not be able to detect all vulnerabilities in web applications or may produce false positives. To mitigate this risk, the scanner will be tested against a variety of web applications with different levels of complexity and different template engines.
- **Project Risks**: There may be delays in development or unexpected issues that arise during development that could impact the completion of the project by the deadline. To

mitigate this risk, regular development meetings will be held with Netsecurity's red team to ensure that the project is on track and that any issues are addressed in a timely manner.

- **Security Risks**: The scanner may be vulnerable to attacks or could be used maliciously if it falls into the wrong hands. To mitigate this risk, the scanner will only be installed on secure systems within the Netsecurity Red Team's network, and access to the scanner will be restricted to authorized users.

Conclusion

The development of the vulnerability scanner will provide the Netsecurity Red Team with a valuable tool for identifying and reporting vulnerabilities in web applications. The scanner will be developed using Python and rely on the BeautifulSoup, Selenium, and Requests libraries to detect vulnerabilities. Quality assurance and testing processes will be implemented to ensure the scanner is effective and easy to use. The scanner will be installed on the Netsecurity Red Team's systems and will be configured to scan the specific domain and web application(s) as specified by the red team. Information on troubleshooting, updates, and technical support will be provided to ensure the scanner can be easily maintained and supported. Finally, risk assessments and management strategies will be implemented to mitigate potential risks to the project's success.

Appendix B

Webapp Class Source

```
import requests
import json
from selenium.webdriver.firefox.options import Options
from seleniumwire import webdriver
import crawler
import inputFinder
import payloadSender


class webApp:
    def __init__(self, url):
        self.url = url
        self.pages = [""]
        self.signinPage = None
        self.signedIn = False
        self.inputs = {}
        self.session = None
        self.driver = None
        self.vulnFound = False

    def getPages(self):
        application = crawler.Crawler(urls=[self.url])
        application.run()
        if application.signinPage:
            self.signinPage = application.signinPage
        else:
            self.signinPage = None
        self.pages = application.visitedUrls
        self.pages = list(set(self.pages))
        return self

    def getInputFields(self):
        inputField = {}
        K = 0  # Indexing key for inputField
        for page in self.pages:
            inputField = {}
            s = inputFinder.inputFinder(page).run(self.session)
            inputField['url'] = page
            inputField['inputs'] = s.inputs
            self.inputs[K] = inputField
            K += 1
        return self

    def readPayloads(self):
        with open('payloads.json') as f:
            data = json.load(f)
        return data

    def sendPayloads(self, page, i, payload, response, vulnerabilityType):
        if not i.forms and not i.inputs:
            p = payloadSender.payLoadSender(page, None, None, ...
                self.session, i.page, payload, response, ...
                vulnerabilityType, self.driver)
            p.run()
            self.vulnFound = p.vulnFound
            return self.vulnFound
            # No forms or inputs on page, might just remove page from list ...
                in later editions
        elif not i.inputs:
            p = payloadSender.payLoadSender(page, None, i.forms, ...
                self.session, i.page, payload, response, ...
                vulnerabilityType, self.driver)
            p.run()
            self.vulnFound = p.vulnFound
            return self.vulnFound
        elif not i.forms:
            p = payloadSender.payLoadSender(page, i.inputs, None, ...
                self.session, i.page, payload, response, ...
                vulnerabilityType, self.driver)
            # Create logic to handle just inputs
            pass
        else:
            p = payloadSender.payLoadSender(page, i.inputs, i.forms, ...
                self.session, i.page, payload, response, ...
                vulnerabilityType, self.driver)
            pass
    def checkPage(self):
        inputField = {}
        data = self.readPayloads()
        self.driver = self.initDriver()
        for page in self.pages:
            sendingPayloads = True
            while sendingPayloads:
                i = inputFinder.inputFinder(page).run(self.session)
                print("\n\n" + "-" * 50)
                print(f"Checking page: {page} for XSS vulnerabilities")
                print("-" * 50 + "\n\n")
                for payload in data['XSS']:
                    if self.sendPayloads(page, i, payload['payload'], ...
                        payload['response'], vulnerabilityType='XSS'):
                         break
                if self.vulnFound:
                    break
                print("\n\n" + "-" * 50)
                print(f"Checking page: {page} for DOM-XSS vulnerabilities")
                print("-" * 50 + "\n\n")
                for payload in data['DOM_XSS']:
                    if self.sendPayloads(page, i, payload['payload'], ...
                        payload['response'], vulnerabilityType='DOM_XSS'):
                         break
                if self.vulnFound:
                    break
                print("\n\n" + "-" * 50)
                print(f"Checking page: {page} for CSTI vulnerabilities")
                print("-" * 50 + "\n\n")
                for payload in data['CSTI']:
                    if self.sendPayloads(page, i, payload['payload'], ...
                        payload['response'], vulnerabilityType='CSTI'):
                        break
                if self.vulnFound:
                    break
                sendingPayloads = False
        print("Done")

    def run(self):
        self.getPages()
        sess = requests.session()
        sess.get(self.url)
        self.session = sess
        self.getInputFields()
        self.checkPage()

    def initDriver(self):
        options = Options()
        options.headless = False
        driver = webdriver.Firefox(
            options=options,
            seleniumwire_options={'verify_ssl': False}
        )
        if self.session is not None:
            driver.get(self.url)
            driver.delete_all_cookies()
            for cookie in self.session.cookies:
                driver.add_cookie({'name': cookie.name, 'value': ...
                    cookie.value})
            driver.refresh()
        return driver
```

Appendix C

Crawler Source

```
import requests                # Importing the requests module to make HTTP ...
    requests
import tldextract              # Importing the tldextract module to extract ...
    the domain and subdomain from a URL
import logging                 # Importing the logging module for error ...
    handling
from bs4 import BeautifulSoup  # Importing BeautifulSoup for parsing HTML

class Crawler:
    def __init__(self, urls=[], scope=None, headers=None):
        # Initializing a Crawler instance with a list of urls to visit, a ...
            scope for the crawler, and headers for HTTP
        # requests
        self.scope = scope
        self.headers = headers
        self.visited_urls = []
        self.urls_to_visit = urls
        self.signinPage = None

    def request_url(self, url):
        # Sending an HTTP GET request to a URL and returning the response ...
            text if successful
        try:
            response = requests.get(url, headers=self.headers)
            response.raise_for_status()    # Raising an exception for ...
                non-200 status codes
        except requests.exceptions.RequestException as e:
            print(f"An error occurred while making the request: {e}")
            return None
        except Exception as e:
            print(f"An unexpected error occurred: {e}")
            return None
        return response.text

    def get_linked_urls(self, html, url):
        # Parsing HTML and extracting all the links and input forms
        soup = BeautifulSoup(html, 'html.parser')
        signinForm = ['password', 'username', 'login', 'signin', 'email']
        # List of input types to identify sign in forms
        soup.find_all('input')      # Finding all the input tags in the HTML
        for form in soup.find_all('input'):
            if form.get('type') in signinForm:    # Checking if the input ...
                tag is a sign-in form
                 print("Sign in form found")
                 self.signinPage = url
```

```
                    break
            for link in soup.find_all('a'):         # Finding all the anchor ...
                tags in the HTML
                href = link.get('href')
                if href[0] == "/":
                    if self.scope['subdomain'] == "None":
                        href = f"https://www.{self.scope['domain']}.
                        {self.scope['suffix']}{href}"
                    else:
                        href = ...
                            f"https://{self.scope['subdomain']}.{self.scope['domain']}.
                        {self.scope['suffix']}{href}"
                if href and self.isInScope(href) and "@" not in href:
                    yield href         # Yielding the valid links

        def crawl(self, url):
            # Crawling a URL and adding all the linked URLs to the list of ...
                URLs to visit
            html = self.request_url(url)
            for url in self.get_linked_urls(html, url):
                self.add_url_to_visit(url)

        def add_url_to_visit(self, url):
            # Adding a URL to the list of URLs to visit if it hasn't already ...
                been visited or added
            if url not in self.visited_urls and url not in self.urls_to_visit:
                self.urls_to_visit.append(url)

        def isInScope(self, url):
            # Extracts the domain, subdomain, and suffix from the URL
            parsedUrl = tldextract.extract(url)
            if self.scope['subdomain'] == "None":
                # Checks if the URL is in the same domain, and suffix as the scope
                if (parsedUrl.domain == self.scope['domain']) and ...
                    (parsedUrl.subdomain == "www") and (
                        parsedUrl.suffix == self.scope['suffix']):
                    return True
            else:
                # Checks if the URL is in the same subdomain, domain, and ...
                    suffix as the scope
                if (parsedUrl.domain == self.scope['domain']) and ...
                    (parsedUrl.subdomain == self.scope['subdomain']) and (
                        parsedUrl.suffix == self.scope['suffix']):
                    return True
            # Returns False if the URL is not in the scope
            return False

        def run(self):
            # Displays a message with the number of URLs to visit
            print(f'Running crawler with {len(self.urls_to_visit)} urls to ...
                visit...')
            while self.urls_to_visit:
                # Removes the first URL from the list and crawls it
                url = self.urls_to_visit.pop(0)
                print(f'Crawling: {url} ({len(self.urls_to_visit)} urls to ...
                    visit)')
                try:
                    self.crawl(url)
                # Logs an error if crawling the URL fails
                except Exception:
                    logging.exception(f'Failed to crawl: {url}')
                # Adds the URL to the list of visited URLs
                finally:
                    self.visited_urls.append(url)
```

Appendix D

Input Finder Source

```
import requests
from bs4 import BeautifulSoup


class inputFinder:
    VALID_FIELDS = ['text', 'textarea', 'search', 'email', 'tel', 'url', ...
       'number']

    def __init__(self, url, session=None, cookies=None):
        self.url = url
        self.session = session
        self.cookies = cookies
        self.inputs = []
        self.forms = []
        self.formInputFilter = []
        self.page = None

    def parsePage(self, page):
        pageInputs = []
        soup = BeautifulSoup(page.text, 'html.parser')
        for form in soup.find_all('form'):
            if form.get('enctype') != 'multipart/form-data':
                self.indexForms(form)
        for inputField in soup.find_all('input'):
            if inputField not in self.formInputFilter:
                self.filterInputs(inputField)
                if self.filterInputs(inputField) is not None:
                    pageInputs.append(self.filterInputs(inputField))
        return pageInputs

    def indexForms(self, form):
        formsDict = {'action': form.get('action'), 'method': ...
           form.get('method'), 'enctype': form.get('enctype'),
                    'inputs': []}
        for textarea in form.find_all('textarea'):
            if textarea.get('name') is not None:
                if not textarea.get('type'):
                    textarea['type'] = 'textarea'
                self.formInputFilter.append(textarea)
                formsDict['inputs'].append(self.filterInputs(textarea))
        for inputField in form.find_all('input'):
            if inputField.get('name') is not None:
                self.formInputFilter.append(inputField)
                formsDict['inputs'].append(self.filterInputs(inputField))
        self.forms.append(formsDict)
    def filterInputs(self, inputField):
        inputsDict = {'name': '', 'type': '', 'required': '', 'request': ...
            ['GET', 'POST']}
        if inputField.get('type') in self.VALID_FIELDS:
            inputsDict['name'] = inputField.get('name')
            inputsDict['type'] = inputField.get('type')
            if inputField.get('required') is not None:
                inputsDict['required'] = True
            if inputField.get('pattern') is not None:
                inputsDict['pattern'] = inputField.get('pattern')
            return inputsDict
        if inputField.get('type') == 'hidden' and ...
            inputField.get('required') is not None:
            inputsDict['name'] = inputField.get('name')
            inputsDict['type'] = inputField.get('type')
            inputsDict['required'] = True
            inputsDict['value'] = inputField.get('value')
            return inputsDict

    def getPage(self):
        try:
            page = self.session.get(self.url)
            self.page = page
            return page
        except requests.exceptions.RequestException as e:
            print("An error occurred:", e)

    def run(self, session):
        self.session = session
        page = self.getPage()
        self.inputs = self.parsePage(page)
        return self
```

Appendix E

Payload Sender Source

```
import bs4
import requests
import random
import string
import re
import json
import urllib.parse
from bs4 import BeautifulSoup
from selenium.webdriver import Keys
from selenium.common.exceptions import *
from selenium.webdriver.firefox.options import Options
from seleniumwire import webdriver
from selenium.webdriver.common.by import By
import time

def readWordlist():
    payloads = []
    wordlist = open("wordlist.txt", "r")
    for line in wordlist:
        payloads.append(line.strip())
    return payloads


def checkExecution(response):
    # This needs to change, however I need to do the payload parsing logic ...
        first.
    soup = bs4.BeautifulSoup(response.content, 'html.parser')
    alert_script = soup.find('script', string=lambda text: text is not ...
        None and 'alert(1)' in text.lower())
    if alert_script is not None:
        return True
    else:
        return False


class payLoadSender:
    def __init__(self, url, inputs, forms, session, soup, payload, ...
        expectedResponse, vulnerabilityType, driver):
        self.inputs = inputs
        self.driver = driver
        self.forms = forms
        self.weaponized = False
        self.session = session
        self.url = url
        self.host = urllib.parse.urlparse(url).netloc
        self.pageUrl = ""
        self.payload = payload
        self.cookie = self.getCookies()
        self.source = soup
        self.expectedResponse = expectedResponse
        self.vulnerabilityType = vulnerabilityType
        self.textCount = 0
        self.vulnFound = False
        self.xpath = None

    def getCookies(self):
        return self.session.cookies.get_dict()

    def noInputGet(self, anomalys):
        for anomaly in anomalys:
            url = anomaly.split('=')[0]
            payload = urllib.parse.quote(self.payload)
            print(url + "=" + payload)
            response = self.session.get(url + "=" + payload)
            if response:
                if checkExecution(response):
                    print("\n\n" + "-" * 50)
                    print(f"Vulnerability on: {anomaly}")
                    print(f"Type: {self.vulnerabilityType}, Payload: ...
                        {self.payload}")
                    print(f"Action: {url}")
                    print(f"Method: GET")

    def sendPayloadForm(self, body, url, form):
        response = None
        if form['method'] == 'POST':
            try:
                response = self.createPOST(body, url, form['enctype'])
            except requests.exceptions.RequestException as e:
                print(e)
        elif form['method'] == 'GET':
            try:
                response = self.createGET(body, url)
            except requests.exceptions.RequestException as e:
                print(e)
        if response:
            if checkExecution(response):
                print("\n\n" + "-" * 50)
                print(f"Vulnerability on: {self.url}")
                print(f"Type: {self.vulnerabilityType}, Payload: ...
                    {self.payload}")
                print(f"Action: {form['action']}")
                print(f"Method: {form['method']}")
            if response.url != url:
                response = self.session.get(self.url)
            if response.status_code == 200:
                if checkExecution(response):
                    print("\n\n" + "-" * 50)
                    print(f"Vulnerability on: {self.url}")
                    print(f"Type: {self.vulnerabilityType}, Payload: ...
                        {self.payload}")
                    print(f"Action: {form['action']}")
                    print(f"Method: {form['method']}")

    def formatAction(self, action):
        if action[0] == '/':
            url = self.pageUrl + action
        else:
            url = self.url
        return url

    def formPayload(self):
        if self.forms is not None:
            for form in self.forms:
                body = self.constructMFBody(form, form['method'])
                self.seleniumSend(body)

    def inputPayload(self):
                body = self.constructBody()
                self.seleniumSend(body)

    def constructHead(self, url, body, enctype):
        cookie = []
        for key, value in self.cookie.items():
            cookie.append(f"{key}={value}")
        head = {'Host': self.host,
                'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64)',
                'Accept-Language': 'en-US,en;q=0.5',
                'Accept-Encoding': 'gzip, deflate',
                'Content-Type': enctype,
                'Content-Length': str(len(body)),
                'Origin': self.pageUrl,
                'Connection': 'close',
                'Referer': self.url,
                'Cookie': f", ".join(cookie)}
        return head

    def createPOST(self, body, url, enctype=None):
        body['comment'] = self.payload
        if url[0] == '/':
            url = "https://" + self.host + url
        if enctype is not None:
            if enctype == 'application/json':
                head = {'Content-Type': enctype}
                body = json.dumps(body)
                response = self.session.post(url, data=body, headers=head)
                return response
            elif enctype == 'application/x-www-form-urlencoded':
                body = urllib.parse.urlencode(body)
                head = self.constructHead(url, body, enctype)
                response = self.session.post(url, data=body, headers=head)
                return response
            else:
                print(f"Content type {enctype} is not supported!")
                pass
        else:
            pass

    def createGET(self, body, url):
        if url[0] == '/':
            url = "https://" + self.host + url
        body = urllib.parse.urlencode(body)
        url = url + '?' + body #Changed from url + "?" + body to url + "=" ...
            + body
        try:
            self.driver.get(url)
            self.driver.implicitly_wait(3)
            if self.expectedResponse == 'alert':
                if self.isAlertPresent():
                    self.printVuln(url)
            else:
                count = self.driver.page_source.count(self.expectedResponse)
                if count > self.textCount:
                    self.printVuln(url)
                    self.textCount = count
        except UnexpectedAlertPresentException:
            self.printVuln(url)

    def constructMFBody(self, form, method):
        multiFieldRequest = {}
        for formInput in form['inputs']:
            payload = None
            if formInput['type'] == None:
                payload = self.payload
            if formInput['type'] == 'email':
                random_string = ...
                    ''.join(random.choices(string.ascii_letters + ...
                    string.digits, k=5))
                payload = f"{random_string}@{random_string}.{'com'}"
                if 'pattern' in formInput:
                    if not re.match(formInput['pattern'], payload):
                        # payload does not match pattern
                        raise ValueError("Payload does not match pattern" ...
                            + formInput['pattern'])
            elif formInput['type'] == 'tel' and formInput['pattern']:
                payload = random.randint(10000000, 99999999)
                if not re.match(formInput['pattern'], payload):
                    # payload does not match pattern
                    raise ValueError("Payload does not match pattern" + ...
                        formInput['pattern'])
            elif formInput['type'] == 'url' or 'pattern' in formInput and ...
                'http' in formInput['pattern']:
                payload = "http://www." + self.payload + ".com"
                if not re.match(formInput['pattern'], payload):
                    # payload does not match pattern
                    raise ValueError("Payload does not match pattern" + ...
                        formInput['pattern'])
            if payload is not None:
                multiFieldRequest[formInput['name']] = payload
            elif formInput['type'] == 'hidden':
                pass
            else:
                multiFieldRequest[formInput['name']] = self.payload
        return multiFieldRequest

    def constructBody(self):
        inputDict = {}
        payload = None
        for htmlInput in self.inputs:
            if htmlInput['type'] is None:
                payload = self.payload
            if htmlInput['type'] == 'email':
                random_string = ...
                    ''.join(random.choices(string.ascii_letters + ...
                    string.digits, k=5))
                payload = f"{random_string}@{random_string}.{'com'}"
                if 'pattern' in htmlInput:
                    if not re.match(htmlInput['pattern'], payload):
```

```
                    # payload does not match pattern
                    raise ValueError("Payload does not match pattern" ...
                        + htmlInput['pattern'])
            elif htmlInput['type'] == 'tel' and htmlInput['pattern']:
                payload = random.randint(10000000, 99999999)
                if not re.match(htmlInput['pattern'], payload):
                    # payload does not match pattern
                    raise ValueError("Payload does not match pattern" + ...
                        htmlInput['pattern'])
            elif htmlInput['type'] == 'url' or 'pattern' in htmlInput and ...
                'http' in htmlInput['pattern']:
                payload = "http://www." + self.payload + ".com"
                if not re.match(htmlInput['pattern'], payload):
                    # payload does not match pattern
                    raise ValueError("Payload does not match pattern" + ...
                        htmlInput['pattern'])
            if payload is not None:
                inputDict[htmlInput['name']] = payload
            else:
                inputDict[htmlInput['name']] = self.payload
        print(inputDict)
        return inputDict


    def testButtons(self, button):
        if button['href'] is not None:
            if button['href'][0] == '/':
                url = "https://" + self.host + button['href']
                response = requests.get(url)
                if response.url != url:
                    url = response.url
                parsedURL = urllib.parse.urlparse(url)
                queryString = parsedURL.query
                body = {queryString.split('=')[0]: self.payload}
                url = url.split('?')[0]
                self.createGET(body, url)

    def checkButtons(self):
        buttonClasses = ['button',
                         'button is-loading',
                         'button is-text',
                         'button is-white',
                         'button is-black',
                         'button is-light',
                         'button is-dark',
                         'button is-primary',
                         'button is-link',
                         'button is-info',
                         'button is-success',
                         'button is-warning',
                         'button is-danger',
                         'button is-normal',
                         'button is-medium',
                         'button is-large',
                         'button is-fullwidth',
                         'button is-static',
                         'button is-rounded']
        soup = BeautifulSoup(self.source.content, 'html.parser')
        for buttonClass in buttonClasses:
            button_links = soup.find_all('a', {'class': buttonClass})
            for button in button_links:
                self.testButtons(button)
        pass

    def initDriver(self):
        options = Options()
        options.headless = False
        driver = webdriver.Firefox(
            options=options,
            seleniumwire_options={'verify_ssl': False}
        )
        if self.session is not None:
            driver.get(self.url)
            driver.delete_all_cookies()
            for cookie in self.session.cookies:
                driver.add_cookie({'name': cookie.name, 'value': ...
                    cookie.value})
            driver.refresh()
        return driver

    def seleniumSend(self, mfRequest):
        try:
            self.driver.get(self.url)
            for key, value in mfRequest.items():
                try:
                    self.driver.find_element(By.NAME, key).send_keys(value)
                except:
                    print(self.driver.exception)
                    pass
            try:
                self.driver.send_keys(Keys.ENTER)
            except:
                try:
                    self.driver.find_element(By.NAME, 'submit').click()
                except:
                    try:
                        self.driver.find_element(By.CSS_SELECTOR, ...
                            '[class*="button" i]').click()
                    except:
                        try:
                            self.driver.find_element(By.CLASS_NAME, ...
                                "button").click()
                        except:
                            if self.xpath is not None:
                                try:
                                    self.driver.find_element(By.XPATH, ...
                                        self.xpath).click()
                                except:
                                    print("Could not submit form!")
                                    print("Please attempt to enter the ...
                                        Xpath of the submit button")
                                    xpath = input("Xpath: ")
                                    self.xpath = xpath
                                    self.driver.find_element(By.XPATH, ...
                                        xpath).click()
            self.driver.implicitly_wait(5)
            if self.driver.current_url != self.url:
                if 'twitter' in self.driver.current_url:
                    print("TWITTER DETECTED!")
                    print("LAB Solved! Killing session...")
                    self.driver.close()
                    exit()
                else:
                    if self.expectedResponse == 'alert':
                        if self.isAlertPresent():
                            self.printVuln(self.driver.current_url)
                        else:
                            try:
                                self.driver.get(self.url)
                                if self.isAlertPresent():
                                    self.printVuln(self.driver.current_url)
                            except:
                                try:
                                    if self.isAlertPresent():
                                        self.printVuln(self.driver.current_url)
                                except:
                                    print(self.driver.exception)
                    else:
                        count = ...
                            self.driver.page_source.count(self.expectedResponse)
                        if count != self.textCount:
                            self.printVuln(self.driver.current_url)
                            self.textCount = count
                        else:
                            self.driver.get(self.url)
                            count = ...
                                self.driver.page_source.count(self.expectedResponse)
                            if count != self.textCount:
                                self.printVuln(self.driver.current_url)
                                self.textCount = count
            if self.isAlertPresent():
                self.printVuln(self.driver.current_url)
        except UnexpectedAlertPresentException:
            self.printVuln(self.url)

    def printVuln(self, url):
        print("\n\n" + "-" * 50)
        print(f"Vulnerability on: {self.url}")
        print(f"Type: {self.vulnerabilityType}, Payload: {self.payload}")
        print(f"Please investigate: {url}")
        print("-" * 50 + "\n\n")
        self.vulnFound = True

    def isAlertPresent(self):
        self.driver.implicitly_wait(2)
        time.sleep(1)
        try:
            alert = self.driver.switch_to.alert
            alert.dismiss()
            self.printVuln(self.driver.current_url)
            return True
        except NoAlertPresentException:
            return False

    def checkForPayload(self):
        self.driver.get(self.url)
        self.driver.implicitly_wait(3)
        response = self.driver.page_source
        count = response.count(self.expectedResponse)
        return count

    def run(self):
        self.textCount = 0  # Used to check instances of payload in ...
            response, resets when a new page is loaded
        if self.expectedResponse != 'alert':  # Check if expected response ...
            is an alert
            self.textCount = self.checkForPayload()  # Count instances of ...
                expected payload in response
        self.checkButtons()
        if self.forms is not None:
            self.formPayload()
        if self.inputs is not None:
            self.inputPayload()
```

Appendix F

Changelog

F.0.1 January 17th, 2023

The development of the scanner commenced on January 17th, 2023, when the initial commit for the "Full repository for Master thesis 2023" was pushed to a GitHub repository explicitly created for the master thesis project. At this stage, the team conducted preliminary research into the Selenium API for the headless browser, as well as testing the Selenium standalone Docker and successfully opening a web page using the Selenium Firefox module with a basic Python script (../Testing/Selenium1.py).

Additionally, on this day, the team began compiling a list of "Useful links" on the GitHub repository. The progress and updates were documented through comments on GitHub, stating the creation of the master thesis repository, the initial research into Selenium API, the successful testing of Selenium standalone Docker, and the ability to open web pages using the Selenium Firefox module with a Python script.

F.0.2 January 20th, 2023

On this particular day, the team attempted to utilize the "alert" feature of Selenium in the project. This involved a series of testing scripts and a final script named AlertTesting.py, which aimed to determine the feasibility of incorporating this feature into the vulnerability scanner.

Two commits were made to GitHub on this day. The first commit included a "sample" code that utilized the Selenium library to interact with alerts. In contrast, the second commit was customized to align with the specific requirements and scope of the project.

Comment from GitHub:
"Created a working alert checker using Selenium using an except condition if the driver fails to switch to alert. One thing to note is that this might fail if the site has not yet finished loading. A solution would be to add a timeout feature. I also tried to check this alert using ‘expected conditions‘ with the ‘alertIsPresent()‘ function with no success, as it never failed and didn’t stop running. "

F.0.3 March 12th 2023

On March 12th, 2023, the team initiated the second iteration of the client-side template injection (CSTI) module, following the initial presentation of the vulnerability scanner to Netsecurity. In order to streamline the development process and focus on implementing the CSTI module, a new GitHub repository was explicitly created for this purpose. The entire codebase of the vulnerability scanner, including the existing features and the new CSTI module, was uploaded to this new GitHub repository.

The decision to create a new GitHub repository for the second iteration of the CSTI module was driven by the need to separate the CSTI module's development from the scanner's initial version. This allowed for a clear separation of changes and updates made during the second iteration from the previous version, making it easier to track and manage the development progress of the CSTI module.

F.0.4 March 13th 2023

A list of "requirements" was made to make the second iteration of the vulnerability scanner much better.

The list:

- What does an input field need to be tested?
  - Page (URL)
  - Input field type
  - Input field name
- What does an input field need to be tested?
  - text, textarea, search, email, tel, URL, number, range, date, month, week, time, datetime-local, color
- From w3schools: Note: Only form elements with a name attribute will have their values passed when submitting a form. Do we then want to ignore all input fields without a name attribute?

Further pseudo code and code for the "inputfinder.py" was developed this day.

The pseudo-code:

- Inputs: URL, Session, Cookies (URL required, session and cookies optional)
- Check if the URL is valid
- Check if session and cookies are provided
- If session and cookies are provided, do an init with Selenium and requests to apply session
- If session and cookies are not provided, do an init with Selenium and requests to create a new session
- Send a request to the URL and store the response in a variable
- Parse response with BeautifulSoup
- Find all input fields
- Filter input fields by type and name
- Store input fields in a nested dictionary
- Dictionary 1: URL
- Dictionary 2: Input field type, name (URL:inputType:", inputName:"...)
- Return the dictionary

Creating the pseudo-code was to "brainstorm" how it should work and give a clear direction when we started developing.

We also decided to change the layout of how the vulnerability scanner would function. This resulted in segregating certain parts of the code to become a modular design that could be called from the main. These where at this point webapp.py and inputFinder.py

F.0.5 March 14th, 2023

During the development phase, several important updates were made to the payloadSender.py script to enhance the functionality of the web application vulnerability scanner. Firstly, a simple POST functionality was added, enabling the scanner to send HTTP POST requests to web applications for vulnerability testing.

In addition, methods were implemented to input a dictionary to filter out methods that were not applicable to the selected field. This logic was incorporated to improve the efficiency and effectiveness of the scanner by skipping unnecessary methods that do not work for the specific field being tested.

Furthermore, a limited GET functionality was added to the payloadSender.py script, allowing the scanner to send HTTP GET requests to web applications for vulnerability testing, albeit with limitations.

A separate file was added to aid in testing and implementing short scripts, and the README file was updated to reflect the changes in the scanner's functionality. Additionally, a "checkResponse" feature was added, which prints "yaba daba doo" when a reflection is found. This visual indication was designed to alert developers quickly when a reflection vulnerability is detected during testing.

These updates to the payloadSender.py script played a crucial role in enhancing the capabilities of the web application vulnerability scanner. The addition of POST and GET functionalities, filtering of unnecessary methods, and immediate feedback on reflection vulnerabilities through the "checkResponse" feature aimed to improve the accuracy and efficiency of the scanner in detecting web application vulnerabilities. These enhancements were a significant step towards achieving the project's goals and advancing the vulnerability scanner's capabilities.

F.0.6 March 16th, 2023.

The inputFinder.py function progressed significantly, transitioning from a pseudo code to a more complete and functional component of the vulnerability scanner.

The inputFinder class, implemented in Python, is designed to locate and parse input fields, such as form fields, from a web page. This class utilizes the BeautifulSoup library, a popular Python library for parsing HTML and XML documents, to achieve this functionality.

The inputFinder class takes a URL as an input parameter and optionally accepts a session and cookies for making HTTP requests with authentication. This class aims to scan web pages for potential security vulnerabilities associated with input fields to identify and mitigate such vulnerabilities as part of the overall vulnerability scanning process.

The development of the inputFinder class represents a significant step in advancing the vulnerability scanner, as it provides a functional and comprehensive solution for identifying and parsing input fields in web pages. Integrating this class into the vulnerability scanner contributes to the overall effectiveness and accuracy of the scanning process, enhancing the scanner's ability to identify potential vulnerabilities related to input fields and strengthen the security of web applications.

F.0.7 March 21st, 2023.

New logic was added to the code to handle Form GET requests and perform pattern checks for special input fields such as email and URL. This indicates that the code's functionality was enhanced to include handling GET requests and implementing pattern checks for specific input field types.

Implementing this logic involves checking the input field type, such as email or URL, and applying specific processing or validation rules accordingly. For example, for email fields, the logic may involve checking if the input value provided by the user is in a valid email format. Similarly, the logic may involve validating if the input value is in a valid URL format for URL fields.

Adding logic for Form GET requests and pattern checks for special input fields like email and URL can significantly improve the accuracy and effectiveness of the vulnerability scanner in correctly identifying and processing input fields. This enhancement can result in more reliable results and better detection capabilities for identifying vulnerabilities, ultimately contributing to the overall effectiveness of the vulnerability scanning process.

F.0.8 March 22nd, 2023.

On March 22nd, the development to add POST functionality to the vulnerability scanner was initiated. This entailed implementing the capability to send POST requests to web pages as part of the scanning process. However, it was observed that the POST functionality was not functioning as expected and did not yield the intended results. Despite this setback, it was considered a promising starting point for further development of the POST functionality. The groundwork had been laid out, focusing on debugging and resolving the issues to make it functional.

F.0.9 March 27th, 2023.

On March 27th, the issues with the non-working POST functionality in the vulnerability scanner were resolved, and the POST functionality was completed. This involved identifying and fixing coding errors, misconfigurations, or any other issues hindering the POST functionality's proper functioning. This included thorough code debugging, validating the correctness of data payloads being sent in the POST requests, ensuring the proper handling of responses from the web pages, and other related tasks.

With the POST functionality successfully implemented, the vulnerability scanner gained the ability to send payloads or Form data as part of POST requests to the targeted web application during the scanning process. This enabled the scanner to simulate data submission to web forms or other input fields and test for vulnerabilities related to input validation, data handling, and other security issues.

Completing the POST functionality significantly enhanced the vulnerability scanner's capabilities, allowing it to conduct more comprehensive and accurate scans of the targeted web application, as it could now send payloads or Form data to the application in a similar manner to how an actual user interacts with web forms. This improvement increased the scanner's effectiveness in identifying potential vulnerabilities and improved its overall scanning capabilities.

F.0.10 March 28th, 2023

Functionality for printing vulnerability identification was added with the printVuln() function. A new function for sending GET requests was created.

F.0.11 April 18th 2023

The first rewrite of the payload sender from the requests library to Selenium was committed. It still reqiures some work, and the constructBody function must be rewritten to work with Selenium.

F.0.12 April 25th 2023

Switched to Selenium for payload sending and execution detection. Integrated Selenium library for improved payload sending and execution detection. Implemented logic to read the payload JSON file, enabling easier customization and management of payloads. Introduced functionality to check for non-Alert type payloads, enhancing payload diversity and flexibility.

F.0.13 May 2th 2023

Conducted extensive code cleanup, removed unused code and resources from the project. Eliminated redundant functions, variables, and dependencies. Introduced functionality to check for non-Alert type payloads, enhancing payload diversity and flexibility.

F.0.14 May 22th 2023

Implemented a short sleep timer in the alert detection to allow the JavaScript on the page to run before checking for an alert, this improved detection performance.

Bibliography

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

[2] Nuno Antunes and Marco Vieira. "Benchmarking vulnerability detection tools for web services." In: *2010 IEEE International Conference on Web Services*. IEEE. 2010, pp. 203–210.

[3] Artur Avetisyan. "Understanding Template Injection Vulnerabilities." In: (2022). https://www.paloaltonetworks.com/blog/prisma-cloud/template-injection-vulnerabilities/ retrieved 05.01.2023.

[4] Yohan Suryanto Azwar Al Anhar. "Evaluation of Web Application Vulnerability Scanner for Modern Web Application." In: (2021). https://ieeexplore.ieee.org/document/9497831 retrieved 05.05.2023.

[5] Ari Bajo. "Web crawling with Python." In: (2023). https://www.scrapingbee.com/blog/crawling-python/ retrieved 20.01.2023.

[6] Jim Boehm. "Cybersecurity trends: Looking over the horizon." In: (2022). https://www.mckinsey.com/capabilities/risk-and-resilience/our-insights/cybersecurity/cybersecurity-trends-looking-over-the-horizon retrieved 05.01.2023.

[7] Gregory T Brown. *Ruby Best Practices: Increase Your Productivity-Write Better Code.* " O'Reilly Media, Inc.", 2009.

[8] "class ERB." In: (). https://ruby-doc.org/3.2.2/stdlibs/erb/ERB.html retrieved 25.04.2023.

[9] Roddy Correa et al. "Hybrid Security Assessment Methodology for Web Applications." In: *Computer Modeling in Engineering & Sciences* 126.1 (2021), pp. 89–124.

[10] Douglas Crockford and Chip Morningstar. *Standard ECMA-404 The JSON Data In-terchange Syntax.* 2017.

[11] Anusha Damodaran et al. "A comparison of static, dynamic, and hybrid analysis for malware detection." In: *Journal of Computer Virology and Hacking Techniques* 13 (2017), pp. 1–12.

[12] Lior Ben Dayan. "OWASP Top 10 vulnerabilities 2022: what we learned." In: (2023). https://vulcan.io/blog/owasp-top-10-vulnerabilities-2022-what-we-learned/ retrieved 05.01.2023.

[13] Scrapy developers. "Scrapy at a glance." In: (2023). https://docs.scrapy.org/en/latest/intro/overview.html retrieved 20.01.2023.

[14] Python Software Foundation. "urllib.parse — Parse URLs into components." In: (2023). https://docs.python.org/3/library/urllib.parse.html#module-urllib.parse retrieved 25.04.2023.

[15] Fullstakcpython.com. "Template Engines." In: (2022). https://www.fullstackpython.com/template-engines.html retrieved 05.01.2023.

[16] Google. "AngularJS — Superheroic JavaScript MVW Framework." In: (2021). https://angularjs.org/ retrieved 29.04.2023.

[17] Jason Huggins. "selenium 4.8.2." In: (18.02.2023). https://pypi.org/project/selenium/ retrieved 01.03.2023.

[18] Taraq Hussain and Satyaveer Singh. “A comparative study of software testing techniques viz. white box testing black box testing and grey box testing.” In: *IJAPRR), ISSN* (2015), pp. 2350–1294.

[19] GORAN JEVTIC. “17 Best Vulnerability Assessment Scanning Tools.” In: (2020). https://phoenixnap.com/blog/vulnerability-assessment-scanning-tools retrieved 05.01.2023.

[20] JR Johnson. “White Box vs. Black Box Web Application Penetration Testing.” In: (2020). https://www.triaxiomsecurity.com/white-box-vs-black-box-web-application-penetration-testing/ retrieved 29.04.2023.

[21] Kyle Johnson. “Pros and cons of manual vs. automated penetration testing.” In: (2022). https://www.techtarget.com/searchsecurity/feature/Pros-and-cons-of-manual-vs-automated-penetration-testing retrieved 05.01.2023.

[22] kinsta. “What Is GitHub? A Beginner’s Introduction to GitHub.” In: (2022). https://kinsta.com/knowledgebase/what-is-github/ retrieved 05.01.2023.

[23] Lavakumar Kuppan. “IronWASP - Open Source Advanced Web Security Testing Platform.” In: (2015). http://blog.ironwasp.org/ retrieved 05.01.2023.

[24] John Kurkowski. “tldextract 3.4.0.” In: (4.10.2022). https://pypi.org/project/tldextract/ retrieved 01.03.2023.

[25] Jim Manico et al. “Cross Site Scripting (XSS).” In: (2022). https://owasp.org/www-community/attacks/xss/ retrieved 27.04.2023.

[26] OWASP. “DOM Based XSS.” In: (2022). https://owasp.org/www-community/attacks/DOM_Based_XSS retrieved 05.01.2023.

[27] Marco Pautasso. “Ten Simple Rules for Writing a Literature Review.” In: (2013). https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3715443/ retrieved 05.05.2023.

[28] Kien Pham, Aécio Santos, and Juliana Freire. “Understanding website behavior based on user agent.” In: *Proceedings of the 39th International ACM SIGIR conference on Research and Development in Information Retrieval*. 2016, pp. 1053–1056.

[29] ProtectOnce. “What is Web Application Security Testing Why Its Hard to Do Well.” In: (2021). https://protectonce.com/what-is-web-application-security-testing-why-its-hard-to-do-well/ retrieved 05.01.2023.

[30] PyCharm. “The Python IDE for Professional Developers.” In: (2023). https://www.jetbrains.com/pycharm/ retrieved 05.01.2023.

[31] Python. “TWhat is Python? Executive Summary.” In: (2023). https://www.python.org/doc/essays/blurb/ retrieved 05.01.2023.

[32] Kenneth Reitz. “requests 2.28.2.” In: (12.01.2023). https://pypi.org/project/requests/ retrieved 01.03.2023.

[33] Leonard Richardson. “beautifulsoup4 4.11.2.” In: (3.01.2023). https://pypi.org/project/beautifulsoup4/ retrieved 01.03.2023.

[34] Kevin A Roundy and Barton P Miller. “Hybrid analysis and control of malware.” In: *Recent Advances in Intrusion Detection: 13th International Symposium, RAID 2010, Ottawa, Ontario, Canada, September 15-17, 2010. Proceedings 13*. Springer. 2010, pp. 317–338.

[35] Elar Saks. “JavaScript Frameworks: Angular vs React vs Vue.” In: (2019).

[36] selenium.dev. “The Selenium Browser Automation Project.” In: (2023). https://www.selenium.dev/documentation/ retrieved 03.04.2023.

[37] V Solovei, Olga Olshevska, and Y Bortsova. “The difference between developing single page application and traditional web application based on mechatronics robot laboratory onaft application.” In: *Automation of technological and business processes* 10.1 (2018).

[38] spdx.org. “Apache License 2.0.” In: (2018). https://spdx.org/licenses/Apache-2.0.html retrieved 01.03.2023.

[39] spdx.org. “BSD 3-Clause "New" or "Revised" License.” In: (2018). https://spdx.org/licenses/BSD-3-Clause.html retrieved 01.03.2023.

[40] spdx.org. “MIT License.” In: (2018). https://spdx.org/licenses/MIT.html retrieved 01.03.2023.

[41] Subgraph. “Vega helps you find and fix cross-site scripting (XSS), SQL injection, and more.” In: (2023). https://subgraph.com/vega/ retrieved 05.01.2023.

[42] Nicolas Surribas. “Wapiti The web-application vulnerability scanner.” In: (2023). https://wapiti-scanner.github.io/ retrieved 05.01.2023.

[43] Agnes Talalaev. “Website Hacking Statistics You Should Know in 2022.” In: (2021). https://patchstack.com/articles/website-hacking-statistics/ retrieved 05.01.2023.

[44] Andriy Varusha. “What Can You Expect To Pay For Penetration Testing?” In: (2022). https://bsg.tech/blog/what-can-you-expect-to-pay-for-penetration-testing/ retrieved 05.01.2023.

[45] Andrew Whitaker and Daniel P Newman. *Penetration Testing and Network Defense: Penetration Testing _1*. Cisco Press, 2005.

[46] Scott White and Geoff Walton. “Ruby ERB Template Injection.” In: (2017). https://www.trustedsec.com/blog/rubyerb-template-injection/ retrieved 25.04.2023.

[47] William. “Web Application Architecture: The Latest Guide 2022.” In: (2022). https://www.clickittech.com/devops/web-application-architecture/#h-what-is-a-3-tier-architecture retrieved 05.05.2023.

[48] OWASP ZAP. “ZAP.” In: (2023). https://www.zaproxy.org/docs/ retrieved 05.01.2023.