FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Automated Vulnerability Scanning of Web Applications
Student:	Oliver Šmakal
Supervisor:	Ing. Jiří Dostál, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer Security and Information technology
Department:	Department of Computer Systems
Validity:	until the end of summer semester 2022/2023

Instructions

With the growing popularity of web applications, it is necessary to emphasize their information security.
According to the OWASP TOP 10 leader list, code injection and cross-site scripting (XSS) belong among the most serious security vulnerabilities.
These security vulnerabilities can be caused by insufficiently escaping user input that allows attackers to execute malicious SQL payloads (SQL injection). Unlike SQL injection which executes the attack on the database of the server, in the case of XSS, the payload is executed on the client. Various automated tools exist to simplify the detection of these vulnerabilities.
1. Get familiar with SQL injection and XSS vulnerabilities.
2. Research and analyze available tools for the detection and exploitation of these vulnerabilities.
3. Based on your analysis, choose one type of vulnerability and implement a tool or a plugin that will extend the capabilities of its automated detection.
4. For testing purposes investigate intentionally vulnerable web applications.
5. Test the implemented solution.

Electronically approved by prof. Ing. Pavel Tvrdík, CSc. on 24 January 2022 in Prague.

Bachelor's thesis

AUTOMATED VULNERABILITY SCANNING OF WEB APPLICATIONS

Oliver Šmakal

Faculty of Information Technology
Department of Computer Systems
Supervisor: Ing. Jiří Dostál, Ph.D.
January 4, 2023

Czech Technical University in Prague
Faculty of Information Technology
© 2022 Oliver Šmakal. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis: Šmakal Oliver. *Automated Vulnerability Scanning of Web Applications.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022.

Contents

List of Tables

List of code listings

First and foremost, I would like to express my deepest gratitude to my supervisor, Ing. Jiří Dostál, for his invaluable advice, time and patience. I am also thankful for the everlasting support of my family, especially my wife and my son.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis. I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on January 4, 2023 .....

Abstract

The goal of this thesis is to create a new tool or a plugin which will improve capabilities of automated detection of web application vulnerabilities.

To reach this goal the most popular open-source web vulnerability scanners were analyzed and benchmarked for SQLi and XSS detection. Based on this analysis a decision to create a ZAP plugin which would improve its XSS detection capabilities was made.

The newly created plugin successfully reaches this goal by correctly detecting whether an XSS payload is being injected into a properly quoted JavaScript context.

Keywords web vulnerability scanning tool, SQL injection, XSS, DAST, ZAP

Abstrakt

Cílem této práce je vytvořit nástroj či plugin, který rozšíří možnosti skenování zranitelností webových aplikací.

K dosažení tohoto cíle byla provedena analýza nejpopulárnějších open-source nástrojů k automatické detekci zranitelností typu SQL injekce a XSS. Na základě této analýzy bylo učiněno rozhodnutí vytvořit plugin k nástroji ZAP, který rozšíří jeho možnosti detekce XSS zranitelností.

Vyvinutý plugin úspěšně dosahuje tohoto cíle pomocí správného rozpoznání, zda se injektovvaný kód nachází uvnitř uvozovek v JavaScript contextu a zda je injektovaný kód správně ošetřený.

Klíčová slova nástroj pro skenování zranitolností webových aplikací, SQL injekce, XSS, DAST, ZAP

List of abbreviations

BeEF	The Browser Exploitation Framework
bWAPP	Buggy Web Application
CLI	command line interface
CWE	Common Weakness Enumeration
DAST	dynamic application security testing
DBMS	database management system
DDoS	distributed denial of service
DOM	document object model
DoS	denial of service
DVWA	Damn Vulnerable Web Application
GNU	GNU's Not Unix!
GPL	General Public License
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IAST	interactive application security testing
IDS	intrusion detection system
IVWA	intentionally vulnerable web application
JS	JavaScript
JSP	Java Server Pages
mXSS	mutation XSS
OWASP	The Open Web Application Security Project
RFI	remote file inclusion
SAST	static application security testing
SQLIA	SQL injection attack
SQLi	SQL injection
w3af	web application attack and audit framework
WAF	web application firewall
WAVSEP	The Web Application Vulnerability Scanner Evaluation Project
XSS	cross site scripting
XSSer	Cross Site "Scripter"
ZAP	Zed Attack Proxy
TP	True Positive
FP	False Positive
TN	True Negative
Fn	False Negative
TPR	True Positive Rate
FPT	False Positive Rate

Introduction

With rise in popularity of web application in recent years it is increasingly important to put emphasis on their security.

The main contribution of this thesis is a new tool or plugin which will expand possibilities in automated scanning of web applications in scanning for SQL injection or cross site scripting vulnerabilities. This thesis can also be of use for those who implements similar tool or for those who needs a guideline how to test web scanners.

There are far to many types of vulnerabilities that can be found in web applications. That is why the number of investigated vulnerabilities had to be cut down. In the end, SQL injection and cross site scripting were selected. Even though their ranking in OWASP TOP 10 has came down since the previous revision, the potential damage caused by an exploitation of these vulnerabilities still carries immensely negative consequences, which are mostly impacting the users of vulnerable web applications.

Moreover, this thesis is focused on DAST tools. This means that the tool that is to be developed, will scan the web application for vulnerabilities during the run of the application and with no access to the source code of the application.

In the first chapter the main focus will be an analysis of SQL injection and cross site scripting vulnerabilities. In the second chapter an analysis of intentionally vulnerable web applications is carried out. After that, in the third chapter, the focus will shift towards the tools for automated scanning of web applications and they will be tested against the selected subset IVWAs. In the last chapter a plugin or a tool which extends capabilities of XSS or SQLi detection will be implemented and tested.

Objectives

The aim of the first chapter of this thesis is to carry out an analysis of SQL injection and cross site scripting vulnerabilities.

The main objective of the second chapter is to research intentionally vulnerable web applications.

The objective of the third chapter is to conduct a research on and to preform an analysis of tools for an automatic detection and exploitation of previously analyzed vulnerabilities. Subsequently these tools will be tested against a selected subset of IVWAs.

The goal of the last chapter is to create a tool or a plugin, which will expand the possibilities of the tools researched in the previous chapter and to carry out a testing of the implemented solution.

Chapter 1

The Nature of SQL Injection and Cross Site Scripting Vulnerabilities

The main focus of this chapter is to investigate and analyze current state of SQL injection and cross site scripting vulnerabilities and define key concepts.

1.1 SQL Injection

SQL injection is a a very dangerous kind of exploit which can cause leakage of important data from a database such as usernames, passwords, credit card details, addressed and more pieces of personal information.

Even though focus of this paper is on web applications, SQL injection is not exclusive to them and it may be applicable to any case where code accepts input and then the SQL statement is dynamically generated from this input.

As it is described in [1] it is possible to classify SQLIAs firstly by the mechanism that was used to deliver the payload, secondly by the intend of the attack and lastly by the form of the payload.

1.1.1 Injection Mechanisms

When an attacker wants to carry out a successful SQL injection attack the first step is to get the payload of the attack to the server. There are 4 main avenues of how the payload could be presented to the server.

1.1.1.1 Injection through User Input

This vector of attacks is the most simple one. In this case the payload is send to the web application typically via HTTP GET or POST request. These request are often used for handling input forms, thus data from these is likely to be used in formation of SQL query. If the creation of this query is handled incorrectly, application might be vulnerable to some kind of SQL injection. [1]

1.1.1.2 Injection through Cookies

"An HTTP cookie (web cookie, browser cookie) is a small piece of data that a server sends to a user's web browser. The browser may store the cookie and send it back to the same server with later requests." [2]

These cookies are used for storing information about web application state on client's machine. For restoring previous session a web application may retrieve the cookie from the client. It is in this moment, a query might be constructed in an vulnerable way on the server and as the client has complete control over cookies it is possible to construct a cookie with malicious payload. [3]

1.1.1.3 Injection through Server Variables

A collection of variables containing HTTP, network headers and environmental variables — server variables — are used in web applications in multiple ways such as logging usage statistics and identifying browsing trends. Logging these variables incorrectly may lead to a vulnerability to SQLIAs. As attackers are able to forge the values of HTTP and network headers, malicious payloads can be placed directly into the headers and when the web application tries to log this information in an insecure way the payload is executed. [1]

1.1.1.4 Second-order Injection

One of the ways how to prevent SQLIAs is to correctly escape input. This doing causes that the data may contain values with special SQL keywords and syntax elements. This should not be a problem if this is implemented everywhere throughout the web application, but when this mechanism is used incompletely a situation where data already present in database is used to create a malicious query. The principle of using data already stored in the database to bypass, similarly to the example above, is call second-order injection. [3]

1.1.2 Attack Intent

There are many goals an attacker could want to achieve. Some of the following goals are required to fulfill in order to successfully fulfill bigger objectives. For example in order to extract data in which an attacker might be interested, he would need to identify injectable parameters and determine database schema before performing the extraction of data itself.

Identifying Injectable Parameters The goal in this case to discover if any vector of previously mentioned vectors of attack are vulnerable.

Performing Database Finger-printing The objective of the database finger-printing is discover the type and version of the database used by the web application. This is useful as it allows attacker to craft database specific attack. The finger-printing of databases is possible thanks to the fact that to certain types of queries different databases respond differently. [1]

Determining Database Schema In order to correctly extract data from a database, some knowledge of the database schema such as table names, column names, column data types is required. Attacks with this intent are executed to collect data for further exploitation. [1]

Extracting Data In case of these types of attacks, various techniques are used to extract data from the database. Very often this information could be highly sensitive and desired by attackers. This intent is behind most of SQLIAs. [1]

Adding or Modifying Data The goal of these attacks is to add or change information in a database thus corrupting its data. [1]

Performing DoS The objective of these attacks is to deny service of web application to other users. This is performed by shutting down the database, but locking or dropping database tables also fall under this category. [1]

Evading Detection *"This category refers to certain attack techniques that are employed to avoid auditing and detection by system protection mechanisms."* [1]

Bypassing Authentication The goal is to obtain rights and privileges of another application user. To reach this goal, authentication of the web application must by bypassed. [1]

Performing Privilege Escalation Differently from bypassing authentication, the focus of these attacks are shifted toward exploiting the database used privileges. Various implementation errors and logical flaws in the database are being taken advantage of in order to fulfill this goal. [1]

Executing Remote Commands Very often there are different stored procedures and functions available to database users. Executing these arbitrary commands on the database is the goal in this case. [1]

1.1.3 SQLIA Techniques

There are many techniques that can be used for exploiting SQL injection vulnerabilities. Most of the following techniques are used in conjunction to get information about the database, bypass existing defenses and finally exploit the vulnerability.

1.1.3.1 Tautologies

The first method of creating malicious payload is constructing disjunction with a tautology in the WHERE clause of SQL statement. This causes that the statement being executed will effect all rows in the database table. [4], [1]

Another possible use for tautology in where statement is to bypass authentication. This is done in exactly the same way — creating a tautology in a where clause, causing the database to return all the rows. Very often the application cares only if the result is not null and if it is not the application concludes the authentication as successful. [1]

1.1.3.2 Incorrect Queries

This technique of injection is based on constructing the payload in such a way that the executed query is malformed in some way. This often results in response with an error page which describes the nature of the error that was caused by the query. As these error descriptions can be overly descriptive thus this technique is useful for gathering information about the the database. The error can be caused by various sources but the most commonly used ones are syntax, type conversion and logical errors. These type of errors can cause leakage of information about injectable parameters, column data types and names of columns and tables used in the database. [1], [5]

1.1.3.3 Union Query

Goal of this technique is to execute attacker's query. This is reached by abusing the UNION keyword and returning the original query and the attacker's query in one result set. This is done by including `UNION SLECET` in the payload. [4], [1]

1.1.3.4 Stacked Queries

Stacked queries are sometimes also referenced to as piggy-backed queries. If successfully executed, this type of attack can be extremely harmful. Using this technique, attacker can execute any SQL statement including stored procedures[1]. This is carried out by injecting query delimiter (";") at the start of the payload resulting in execution of the original query as well as the injected statement. [1], [4]

1.1.3.5 Using Comments

In order to successfully inject payload into a vulnerable query it is often necessary to get rid of the ending part of the original query. This is the case when this technique became useful. By adding start comment keyword, commonly being --, at the end of the payload, the part behind the injected payload is commented out and will be ignored when the query is executed. [4]

1.1.3.6 Stored Procedures

This technique relies on executing stored procedures. If an attacker knows which database is used, an opportunity to use standard set of stored procedures, which is used to expand upon the functionality of database and its usually build in to the database, arises. [1]

Stored procedures are often written in other scripting languages potentially bringing more vulnerabilities that can be exploited. This fact could result for example in elevating privileges or running arbitrary code using buffer overflow vulnerability. [1], [6]

On top of these build-in procedures there are user-defined procedures. An attacker can create stored and subsequently abuse this procedure. [6]

1.1.3.7 Inference

This concept is based on observing different behavior after executing a statement. The original query is in this case modified so that it contains some sort of condition and so that after evaluation of this condition different action is taken. Techniques implementing this concept can be useful if the targeted application does not give the attacker database error messages as a feedback to successfully executed injection. However by carefully noting the behavior of the application and how it changes upon attempts to inject SQL payload it is possible to find out which parameters are vulnerable to SQLIAs and to deduct additional information about the database. [7], [1]

There are two main techniques using based on the concept of inference and they are blind injection and timing attack.

1.1.3.7.1 Blind Injection This technique depends on the web application to behave differently. The process is following. An attacker inject his true/false condition, if the application behaves ordinarily, just like before the statement was evaluated to true, however if significant change in web application behavior is observed, the injected statement is false. [1]

1.1.3.7.2 Timing Attack Timing attack is very similar to blind injection but instead of observing different behavior of the web application, this technique depends upon measuring the delay of the response of the database. To carry out this attack a statement is placed into the injected payload such that it contains if/else with the predicate in question where one of the branches is made of SQL statement which takes a known amount of time to execute. There are two possible scenarios. Either the predicate will be evaluated as false and the the time measured from request to response will be t1, or the predicate will be evaluated to true, executing the

[1]Stored procedure is a routine stored in the database that can be run. There are two kinds of procedures: user-defined procedure and procedures, which are provided by the database by default.

branch in which there is the statement which takes some time t2 to execute, in this case the attacker will get the response in roughly the time of t1+t2. Therefore by measuring the time, which was taken in waiting for the response, the attacker is able to infer whether the predicate was true or false. [1], [7]

1.1.3.8 Alternate Encoding

The main objective of this technique is to avoid detection of the injected payload by the web application. This technique is used alongside others to carry out successful SQLIAs to evade detection and prevention thus potentially expanding the number of possibilities in which the attacker may exploit the web application. Since the awareness of SQLIAs is quite high the use of this technique is often necessary. [1]

Scanning for bad characters [2] is a commonly implemented in web applications as a defense against SQLIAs however this is often done in insufficient ways. It is very complicated to scan for all possible encoding of strings with special meaning. Adding to the complexity, different layers of web application may handle alternate encoding of stings differently. With the fact that implementing this type of defense in practice is extremely difficult, a great success can be found using this technique. [1]

1.2 Cross Site Scripting

XSS was place among top 10 vulnerabilities in OWASP TOP 10 in years 2017 and 2021. [8] Using XSS vulnerabilities an attacker can do many things including theft of sensitive data of the victim, carry out DDoS and DoS attacks and use the victim's account to brows the web application by stealing his cookies. XSS attacks are based on the idea of forcing the victim's browser to execute malicious JS code crafted by the attacker.

1.2.1 Classification of XSS Vulnerabilities

Every XSS attack can be categorized depending on how the payload was delivered to the victim. Following categories of XSS attacks are mutually disjunctive. This categorization is useful as it provides as with better means to describe the problem and type of vulnerability that can be found on the web application.

1.2.1.1 DOM Based XSS

This kind of XSS is also called type-0 XSS. This XSS vulnerability can occur when client-side JavaScript contains code which writes into HTML such as document.write(). If the website contains such code an attacker will be able to exploit it by creating a link containing malicious JS code. When the victim clicks on this link a request is send to the server and consequently a response is generated.[3] When the response is obtained a malicious code is executed in the victim's browser. [9]

1.2.1.2 Stored XSS

Stored XSS, also known as type-1 XSS, the payload finds its way to the victim through the database. This is done by embedding malicious code into the submissions of the web application.

[2] In this context "bad characters" are referring to such a string of characters which has special meaning in SQL. An example of bad characters would be single quotes `'` or comment operators `--` ,

[3] This response does not contain malicious code, but it can. This type of XSS is know as reflected XSS and will be mentioned later

This submission is then stored in the database. When the victim accesses this submission during the use of the application, the payload is executed. [9]

1.2.1.2.1 Blind XSS This kind of XSS is a special case of stored based XSS. Stored XSS attack becomes blind XSS when the attacker deploys series of payloads on the web application blindly. Thus very often the attacker doesn't even know if the attack was successful , as it is not unusual for the attacker to not have rights to display the injected XSS, until the first victim was attacked. [10]

1.2.1.3 Reflected XSS

Stored XSS, also known as type-2 XSS, is very similar to DOM based XSS in that first the victim must be lured to click a link send by an attacker. The difference is that the malicious code is contained in the response of the server to the client's request. Another difference between these types of XSS is that in DOM based XSS the vulnerability is on the client-side of the web application whereas in case of reflected XSS the server-side is exploited. [9]

1.2.1.4 Mutation XSS

This kind of XSS is quite rare. The main idea in this case is to inject payload which is safe but latter letting it be mutated during interpretation into a malicious form. [11]

One thing that can be used to successfully perform this kind of XSS is interpretation of invalid HTML code by web browsers. Modern browsers tries to make sense of this invalid HTML code and more code is added to form a valid HTML. However depending on how this is handled it may lead to the possibility of creation and subsequent execution of unsafe code. [11]

Thanks to the mutation aspect of this attack it,is easy to bypass server-side and client-side defense mechanisms and filters against XSS as when the payload is checked for executable code there is not any yet. [12]

1.2.2 Risk of XSS Vulnerabilities

Even though in this work the exploitation of the XSS vulnerabilities will not be the main focus it is sill important to recognize why these vulnerabilities are dangerous and why being able to detect these vulnerabilities is truly useful. List of certain exploitations follows to demonstrate importance of XSS vulnerability detection.

1.2.2.1 Phishing Attack

The goal of this kind of attack is to obtain sensitive data about the victim. To carry out this attack, a website that copies the look of the original web application closely is set up. Then, using XSS, an iframe[4] with link to attackers website as src attribute is embedded into the website. When the victim then tries to use the original web application, the attacker's website is used instead through the use of iframe. This can lead to a leak of login information as the user might try to log in to the web application when if fact communicating with the attacker's website. After the victim fills out the login information, the attacker's website will save it and forward the user to the original website making the user not notice that anything is wrong. [9], [14]

1.2.2.2 Theft of Cookies

In this case, when the XSS attack is successful all cookies of the user are send to the attacker. Depending on the type of information these cookies contain, attacker can use it to do multiple

[4] Iframe element is used to display content of other websites [13]

things. For example, if among these cookies was an authentication cookie with its attributes not set correctly, an attacker may attempt to use it to login into the vulnerable web application disguised as the victim. [9]

1.2.2.3 DoS Attack

An attacker can store a JS file on the malicious website and include it in the payload of the attack. If executed this file will inject invalid data into the users cookies. When the victim tries to visit the website for the second time the web application will respond with an error from 400 range as the applications are not be able to process cookies with invalid data. [9]

1.2.2.4 DDoS Attack

This kind of attack can be used to create many requests which will be send to the targeted web application. In this case the payload contain a script which will continuously send request to the targeted website. With rising number of users visiting the part of the vulnerable application, where the payload is embedded, the number of requests received by the targeted website up until the targeted website can no longer respond. [9]

1.2.2.5 Theft of Browser Screenshots

Similarly to previous types of attack a JS file is again stored at the malicious website and contains the payload of the attack. In the payload there is a script which will take a screenshot of the victims browser and send it to the attackers website, which the attacker can later view and obtain sensitive information. [9]

1.2.2.6 XSS Worms

In case of XSS worm, when the payload of the attack reaches user it preforms the XSS again on top of the core of the payload which helps to affect bigger number of users. [9]

Chapter 2

Intentionally Vulnerable Web Applications

For testing purposes, intentionally vulnerable web applications (IVWAs) are needed. They allow for comparison of tools for automated vulnerability scanning of web applications. Most of following IVWAs is contain many vulnerabilities however the vulnerabilities of interest are those related to SQL injection and XSS.

2.1 Damn Vulnerable Web App (DVWA)

DVWA is an open source IVWA build with PHP and MySQL with various levels of difficulty maintained to this day. The goal of DVWA is to aid security professionals to test their skills and tools in a legal environment. There are 3 levels of security: low, medium and high. On high level the IVWA should not be vulnerable, on medium level the vulnerabilities should represent such a vulnerability which could be found in real application and on low level the application is completely vulnerable and its intended use is to teach basic exploitation techniques. On top of the security levels setting, a PHPIDS[1] can be enabled to make the application even harder to exploit. The SQL vulnerabilities in this IVWA are SQL injection through user input and bind SQL injection and the XSS o are reflected, DOM based and stored XSS.

2.2 Buggy Web Application (bWAPP)

This IVWA is made using PHP and MySQL, it is free and open source. In total there are 19 SQL Injection vulnerabilities with 4 of those using Blind SQL injection technique, 14 reflected, 6 stored and 2 other not classified XSS vulnerabilities. Source code is included in the official virtual box instance and it seems that the last update of the application was done in 2014. [15], [16]

2.3 WackoPicko

WackoPicko was introduced in a paper called “Why Johnny Can’t Pentest: An Analysis of Black-box Web Vulnerability Scanners”. Its development stopped at least 4 years ago. The application uses PHP and MySQL and it is published under MIT licence. This application has in total 15

[1] PHPIDS is a PHP intrusion detection system. It is a web application firewall (WAF) which filters user input based on blacklist of potentially malicious code.

intention vulnerabilities, however only 7 of those are XSS or SQL injection vulnerabilities: 2 SQL injection and 5 XSS vulnerabilities. [17], [18]

2.4 WAVSEP

Web Application Vulnerability Scanner Evaluation Project (WAVSEP) is a vulnerable web application build with Java designed to help assessing the features, quality and accuracy of web application vulnerability scanners. It is published under GNU GPL v3 license. It contains following vulnerabilities that would be beneficial for the goal of this thesis:

- XSS via RFI[2]: 108 test cases
- Reflected XSS: 66 test cases
- Error Based SQL injection: 80 test cases
- Blind SQL Injection: 46 cases
- Time Based SQL Injection: 10 test cases

Unfortunately the last update to this IVWA was in 2015. [19], [20]

2.5 OWASP WebGoat

This IVWA is maintained by OWASP designed to teach security of web applications. It is an java application which is maintained to this day. It can be easily run via docker or a jar file and it can even be configured by manipulating environment variables to not include certain sections. But while this IVWA is neat tool for learning about these vulnerabilities it is not meant to be used as a testing platform. [21], [22]

2.6 OWASP Juice Shop

Juice Shop is a IVWA maintained by OWASP foundation. It is published under the MIT licence and it contains vulnerabilities types from OWASP TOP 10 and a few more types. It is build using Angular in the frontend, Node JS in the backend and as a database SQLite is used. This vulnerable application includes 11 injection vulnerabilities (not necessarily SQL injection vulnerabilities) and 9 XSS vulnerabilities. [23], [24]

2.7 OWASP Mutillidae II

OWASP Mutillidae II is a free and open-source IVWA build with PHP and MySQL. It is still maintained to this day and it contains over 40 vulnerabilities. For every category of vulnerabilities mentioned in OWASP Top Ten 2007, 2010, 2013 and 2017 there is a vulnerability of this category included in the application. Regarding SQL Injection and XSS it includes 15 SQL injection vulnerabilities, 21 reflected XSS vulnerabilities, 5 stored XSS vulnerabilities and 2 DOM based XSS vulnerabilities. Many of those vulnerabilities require use of different attack vectors. [25], [26]

[2]RFI is short for Remote File Inclusion

2.8 OWASP Benchmark

OWASP Benchmark is an IVWA made with benchmarking automated scanners in mind. This java application has large number of vulnerabilities. The total number of test cases is 21,041 in v1.1 and 2740 in v2.2. In v1.2 there are 504 SQL injection cases and 455 XSS cases. Every test case there is in this IVWA can be categorized as either a true positive or a false positive CWE[3]. There is a tool included in the project which can automatically score results of tested tools. [27] The IVWA can be quite easily run either in docker container or locally from source.

2.9 Security Crawl Maze

Security crawl maze is not exactly an IVWA. The goal of this application is test the audibility of web application scanner to follow a link embedded in a page. [28] There are many ways in which can a link be embedded thus it is possible for the tools to miss some cases and not scan the whole application.

2.10 Webseclab

"*Webseclab contains a sample set of web security test cases and a toolkit to construct new ones. It can be used for testing security scanners, to replicate or reconstruct issues, or to help with investigations or discussions of particular types of web security bugs.*" [29]

The main focus of webseclab is on XSS vulnerabilities. There is no public website running an instance of this IVWA, but it is quite easy to run locally. Although the documentation is missing the application itself is quite good at showing how to find the vulnerabilities.

2.11 Google Firing Range

This IVWA contains quite a big number of test cases of for various types of vulnerabilities. Running Google Firing Range locally is quite complex and requires having a google account, fortunately there is a public instance running which can be used for testing. [30]

The other inconvenient thing about this IVWA is that it is lacking documentation, which means that distinguishing cases which should not be detected as they are false positive cases is quite involved.

[3]CWE stands for common weakness enumeration and it is a list of hardware and software vulnerabilities

Chapter 3

Tools for Automatic Detection and Exploitation of Web Application Vulnerabilities

In this chapter, tools for an automatic detection and exploration will be presented. In second section of this chapter those tools will be tested on selected IVWA and based on these results these tools will be compared.

There are many solutions to analyze security of software however most of these solutions can be categorized into 3 groups: static application security testing (SAST), dynamic application security testing (DAST) and interactive application security testing (IAST). [31]

SAST, also known as white-box testing, examine the source code of the software. This bring two obstacles. Firstly the application needs to have access to the source code, secondly the application needs to be developed for each technology separately[1]. The advantage over DAST is that SAST can detect vulnerabilities in code that are not exposed to the outside. [31]

DAST, also known as black-box testing, shows no need to have access to the source code of the examined application. The examined application is tested during runtime. This however means that in order to use DAST a functional application is required. Since the application is tested from the outside with no knowledge of the source code it is less dependent on the technologies that were used to create the application. [31]

IAST, also known as grey-box testing, is relatively new compared to SAST and DAST. IAST combines techniques used SAST and DAST to bring even better results. Possible implementation of IAST tool has sensor installed on the backed of the application which sends information about the source code during execution of the application to the DAST concurrently running DAST scanner which uses these information to better scan the application. [31]

3.1 Web Application Scanners

[1] It would be useless to use such a testing software which can detect vulnerabilities in PHP application on an application written in Java, thus separate application/module must be developed

3.1.1 Existing Web Application Scanners

Tool for automatic detection and exploitation can be called web application scanners. They use DAST[2]to find vulnerabilities of web applications. Most of these scanners consists of three main modules. Firstly there is a crawling module responsible for obtaining links to all reachable pages of the application with other critical information such as potentially injectable parameters and form input fields. After the crawling module is done it hands this information to the attack module, which probes these potential attack vectors with a payload. When the response of the web application to the request with the payload is obtained an analysis module examines it for signs of successful attack. [18]

3.1.1.1 OWASP Zed Attack Proxy (ZAP)

ZAP is a widely used open source web application scanner. [32] It can detect many kind of vulnerabilities including SQLi and XSS. This tool works like a main-in-the-middle proxy which stands between the user's browser and the web application. Additional functionalities are available from a variety of add-ons. The SQLi and XSS vulnerabilities which can be scanned for without any non-mandatory add-ons are following: reflected and persistent XSS, SQLi in URL parameters and forms using these techniques: error based and boolean based SQLi, union queries and stacked queries. [33], [34]

3.1.1.2 Skipfish

Skipfish is an active web application security reconnaissance tool. It is an open source web application scanner aimed at being efficient and easy to use. The output of this scanner is a report meant to serve as a foundation for further web application security assessment. When this application is being run, it probes for many vulnerabilities, importantly it checks for following vulnerabilities: SQLi through user input, stored and reflected XSS through vulnerabilities in document body, HTTP redirects and HTTP header splitting. This tool is not designed for exploitation of these vulnerabilities. [35]

3.1.1.3 Sqlmap

This is a tool automating detection and exploitation of SQLi vulnerabilities. It supports 36 database management systems (DBMS) and uses 6 techniques: boolean-based and time-based blind SQLi, error-based, UNION query based, stacked queries and out-of-band SQLi. This web application scanner can be used to exploit theses vulnerabilities automatically in many ways including but not limited to: enumeration of users, password hashes, tables, columns, databases, cracking of retrieved password hashes, dump of the database tables. In case that the attacked web application uses MySQL, PostgreSQL or Microsoft SQL Server as a DBMS more database specific attacks can be launched such as downloading and uploading any file from the database server and execution of arbitrary commands on the underlying operating system. [36]

After testing this too does not seem to be suitable for web scanning. It can be used to exploit found vulnerabilities for sure, but it cannot be used to find them in simple fashion. After running following command on OWASP Benchmark, 0 vulnerabilities were reported by the tool. This tool build in crawling tool is not the best one either with finding only 22 out of 69 URLs and it cannot export its results so it cannot be easily utilized with other tools.

[2]SAST and IAST are not viable in this case as attacker simply does not have access to the source code of the attacked web application and certainly cannot install any kind of sensor on the backend of the application

3.1.1.4 Cross Site "Scripter" (XSSer)

XSSer is a python program published under GNU GPL v3. It is focused is on XSS vulnerabilities and it has many features including: Anti-antiXSS Firewall rules[3], anti-XSS filters bypass[4]. It supports variety of advanced XSS attacks and techniques for example using HTTP headers as an attack vector or using XSS vulnerability to split server response into two and carry out more complex attacks. [37], [38], [39]

3.1.1.5 Arachni

This web application scanner which is free for non-commercial can crawl web application and detect many kinds of vulnerabilities including SQLi and XSS. From the SQLi realm it is error based SQLi, blind SQLi, and multiple variations of XSS. Even though Arachni is not free for commercial purposes its code is still publicly available. It can be deployed in various ways but the one of interest to this work is the CLI scanner. The possibilities of this tool can be extended using plugins. [40], [41]

3.1.1.6 Web Application Attack and Audit Framework (w3af)

This framework consists of plugins which can be of various types but the following types are the most essential ones: crawl, audit and attack. Responsibility of crawl plugins is to find all valid URLs, forms and other possible points of attack. The objective of an audit plugin is to verify whether the the injection points supplied by the crawl plugins are actually vulnerable. The third type of plugins are attack plugins. These plugins exploit vulnerabilities found by the audit plugin. There are audit plugins for detecting SQLi and XSS vulnerabilities and they are capable of detecting following SQLi and XSS vulnerabilities: SQLi through user input, blind SQLi, reflected and stored XSS. [42]

3.1.1.7 Nikto

Nikto is open source web scanner which among other things Nikto searches the web application for Server and bad software configurations and default, insecure or outdated programs. [43] It should be able to detect even detect XSS and SQLi. [44]

Not much success was found with this tool against OWASP Benchmark, only the "X-XSS-Protection header is not defined" message shows up in the reports. This hint that the server does not force the browser to use XSS Protection against reflected XSS. This feature is not supported by all browsers and very often the default setting of the browser is to have this feature enabled. This means that this message has not that much meaning in terms of XSS vulnerability detection. For this reason it won't be compared to other tools as it seems that the tool goal is to detect bad configuration of servers and not to seek vulnerabilities.

3.1.1.8 Wapiti

Wapiti is a command-line application that is able to crawl a website and to detect many types of vulnerabilities including reflected XSS, stored XSS, error based SQLi, boolean based SQLi and time based SQLi. This project is still being maintained to this day. License GNU GPL v2. Written in python.[45]

[3] Additional options can be set so that an attempt to bypass WAF/IDS products can be made

[4] Multiple options can be selected for the payload to be specifically encoded so that the web applications filter is bypassed

3.1.2 Evaluation of Web Application Scanners

For the purpose of testing web application scanners OWASP Benchmark and Security Crawl Maze were selected. OWASP Benchmark was selected because it contains wide range of vulnerabilities which are straightforward to get to for crawler modules. Even though Security Crawl Maze does not contain any vulnerabilities it can provide useful information about the capabilities of the crawler module of the scanner. These two IVWAs should help in independent assessment of the crawler and attacker modules.

3.1.2.1 Crawling Capabilities Testing on Security Crawl Maze

Secutity Crawl Maze contains a total of 69 hidden files which can be identified by the .found appendix. This makes it convenient to evaluate the crawling results of the tested tools and the list of found URLs can be cut down to these interesting easily for example by using the grep command. Some tools however does not provide a list of found URLs.[5] This fortunately is not a big problem as the log of the web application can be used to obtain information about incoming request (provided the application is run locally instead of using the public web instance).

The best scoring tools in this benchmark are ZAP with 37 and Skipfish with 36 found URLs. Interestingly enoough the set of URLs which was discovered by both of these tools is not that big and when these sets are combined they contain 48 URLs compared to the 50 URLs discovered by all tools combined. Other tools results were not as good at 23 discovered URLs for Wapiti, 22 for Sqlmap and only 16 for Arachni.

3.1.2.2 Vulnerability Detection Testing on Owasp Benchmark

The evaluation of the reports generated by the tools when scanning OWASP Benchmark was done following the description in [27].

Every test case has four possible outcomes:

- True Positive (TP): tool correctly reports a real vulnerability
- False Negative (FN): tool does not report a real vulnerability
- True Negative (TN): tool does not report a fake vulnerability
- False Positive (FP): tool reports a fake vulnerability

True positive rate, false positive rate and score are defined over a set of test cases. These metrics can be calculated as follows:

- True Positive Rate (TPR): $TPR = TP/(TP + FN)$
- False Positive Rate (FPR): $FPR = FP/(FP + TN)$
- Score: $Score = TPR - FPR$

Results in SQli and XSS detection for ZAP, Arachni and Wapiti were calculated by the Benchmark utilities. Results for the rest of the tools had to be created manually as they do not support XML output required by the utilities provided by OWASP benchmark. Note that the overall score computed by OWASP Benchmark is not the same as the combined score used in this text since the overall score combines results from all categories whereas the combined score uses only the SQLi and XSS category.

Out of the tested tools the best placed one in both SQLi, XSS and combined detection, scoring at 63.26 %, 89.43 % and 75.68 % in those categories respectively, is ZAP. The second best scoring tool is Arachni which ended up with 50.02 % in SQLi, 63.82 % in XSS and 56.59 % in combined detection. Interestingly All of the results can be seen in tables 3.3, 3.4 and 3.5.

[5] The tools for which this method was used are Sqlmap and Wapiti

Table 3.1 Security Crawl Maze results part 1

Path	Zap	Arachni	Skipfish	Sqlmap	Wapiti
/headers/content-location.found	✗	✗	✗	✗	✗
/headers/link.found	✗	✗	✗	✗	✗
/headers/location.found	✓	✓	✓	✓	✓
/headers/refresh.found	✗	✗	✓	✓	✗
/html/doctype.found	✗	✗	✓	✗	✗
/html/manifest.found	✗	✗	✗	✗	✗
/html/body/background.found	✓	✗	✗	✗	✗
/html/body/a/href.found	✓	✓	✓	✓	✓
/html/body/a/ping.found	✓	✗	✗	✗	✗
/html/body/audio/src.found	✓	✗	✓	✓	✓
/html/body/applet/archive.found	✗	✗	✗	✗	✗
/html/body/applet/codebase.found	✗	✗	✓	✗	✗
/html/body/blockquote/cite.found	✗	✗	✗	✗	✓
/html/body/embed/src.found	✓	✗	✓	✓	✓
/html/body/form/action-get.found	✓	✓	✓	✗	✓
/html/body/form/action-post.found	✓	✓	✓	✗	✓
/html/body/form/button/formaction.found	✓	✗	✗	✗	✓
/html/body/frameset/frame/src.found	✓	✓	✓	✓	✓
/html/body/iframe/src.found	✓	✓	✓	✓	✓
/html/body/iframe/srcdoc.found	✓	✗	✓	✗	✗
/html/body/img/dynsrc.found	✗	✗	✗	✗	✗
/html/body/img/lowsrc.found	✗	✗	✗	✗	✗
/html/body/img/longdesc.found	✗	✗	✗	✗	✗
/html/body/img/src-data.found	✗	✗	✓	✗	✗
/html/body/img/src.found	✓	✗	✓	✓	✓
/html/body/img/srcset1x.found	✓	✗	✗	✗	✓
/html/body/img/srcset2x.found	✗	✗	✓	✗	✓
/html/body/input/src.found	✓	✗	✓	✓	✗
/html/body/isindex/action.found	✗	✗	✗	✗	✗
/html/body/map/area/ping.found	✗	✗	✗	✗	✗
/html/body/object/data.found	✓	✗	✗	✗	✓
/html/body/object/codebase.found	✗	✗	✓	✗	✗
/html/body/object/param/value.found	✗	✗	✗	✗	✗
/html/body/script/src.found	✓	✓	✓	✓	✓
/html/body/svg/image/xlink.found	✓	✗	✓	✗	✗
/html/body/svg/script/xlink.found	✓	✗	✓	✗	✗
/html/body/table/background.found	✓	✗	✗	✗	✗
/html/body/table/td/background.found	✓	✗	✗	✗	✗
/html/body/video/src.found	✓	✗	✓	✓	✓
/html/body/video/poster.found	✓	✗	✗	✗	✗
/html/head/profile.found	✓	✗	✗	✗	✗
/html/head/base/href.found	✗	✗	✓	✓	✓
/html/head/comment-conditional.found	✓	✗	✓	✓	✗
/html/head/import/implementation.found	✗	✗	✗	✗	✗
/html/head/link/href.found	✓	✓	✓	✓	✓
/html/head/meta/content-csp.found	✗	✗	✓	✗	✗
/html/head/meta/content-pinned-websites.found	✗	✗	✗	✗	✗
/html/head/meta/content-reading-view.found	✓	✗	✗	✓	✗
/html/head/meta/content-redirect.found	✓	✓	✓	✓	✓

Table 3.2 Security Crawl Maze results part 2

Path	Zap	Arachni	Skipfish	Sqlmap	Wapiti
/html/misc/url/full-url.found	✓	✓	✓	✓	✓
/html/misc/url/path-relative-url.found	✓	✓	✓	✓	✓
/html/misc/url/protocol-relative-url.found	✓	✓	✓	✓	✓
/html/misc/url/root-relative-url.found	✓	✓	✓	✓	✓
/html/misc/string/dot-dot-slash-prefix.found	✗	✗	✓	✗	✗
/html/misc/string/dot-slash-prefix.found	✗	✗	✓	✗	✗
/html/misc/string/url-string.found	✗	✓	✓	✗	✗
/html/misc/string/string-known-extension.pdf	✗	✗	✗	✗	✗
/javascript/misc/comment.found	✓	✗	✓	✗	✗
/javascript/misc/string-variable.found	✓	✗	✓	✗	✗
/javascript/misc/string-concat-variable.found	✗	✗	✓	✗	✗
/javascript/frameworks/angular/event-handler.found	✗	✗	✗	✗	✗
/javascript/frameworks/angular/router-outlet.found	✗	✗	✗	✗	✗
/javascript/frameworks/angularjs/ng-href.found	✗	✗	✗	✗	✗
/javascript/frameworks/polymer/event-handler.found	✗	✗	✗	✗	✗
/javascript/frameworks/polymer/polymer-router.found	✗	✗	✗	✓	✗
/javascript/frameworks/react/route-path.found	✗	✗	✗	✗	✗
/javascript/frameworks/react/index.html/search.found	✗	✗	✗	✗	✗
/misc/known-files/robots.txt.found	✓	✗	✗	✗	✗
/misc/known-files/sitemap.xml.found	✓	✓	✗	✓	✗

Table 3.3 Benchmark results for SQLi

Tool	TP	FN	TN	FP	TPR	FPR	Score
Wapiti	29	243	232	0	10.66 %	0.00 %	10.66 %
ZAP	192	80	215	17	70.59 %	7.33 %	63.26 %
Skipfish	109	163	232	0	40.07 %	0.00 %	40.07 %
Sqlmap	0	272	232	0	0.00 %	0.00 %	0.00 %
Arachni	142	130	227	5	52.21 %	2.16 %	50.05 %

Table 3.4 Benchmark results for XSS

Tool	TP	FN	TN	FP	TPR	FPR	Score
Wapiti	31	215	209	0	12.60 %	0.00 %	12.60 %
ZAP	220	26	209	0	89.43 %	0.00 %	89.43 %
Skipfish	0	246	209	0	0.00 %	0.00 %	0.00 %
Sqlmap	0	246	209	0	0.00 %	0.00 %	0.00 %
Arachni	157	89	209	0	63.82 %	0.00 %	63.82 %

Table 3.5 Benchmark results combined for XSS and SQLi

Tool	TP	FN	TN	FP	TPR	FPR	Score
Wapiti	60	458	441	0	11.58 %	0 %	11.58 %
ZAP	412	106	424	17	79.54 %	38.55 %	75.68 %
Skipfish	109	409	441	0	21.04 %	0 %	21.04 %
Sqlmap	0	246	209	0	0 %	0 %	0 %
Arachni	299	219	436	5	57.72 %	11.34 %	56.59 %

Chapter 4

Implementation of a ZAP Plugin

This chapter goes through the implementation of a ZAP plugin and documents a few related decisions such as why ZAP was chosen as the platform for the plugin.

First of all it should be explained why ZAP plugin which would improve XSS detection was chosen to be implemented. The option which was abandoned first was implementing a standalone tool. Though this would have come with several advantages such as being able to choose any programming language for creating of the tool. However, in order for this piece of software to bring any benefits over the other tools, it would have to be very specialized.

There is one way in which it could be possible to specialize such tool and that is to specialize in XSS vulnerabilities exploiting deprecated features which are still compatible with old versions of browsers. Though these outdated browsers such as Internet Explorer are not very popular, they sometimes are still used for old enterprise application which rely on they specific behavior.

Unfortunately there are a few problems with this idea. Firstly, even getting old versions of browsers installed may be complicated. The only official archive of old versions that was found was one for Netscape Browser. [46] The second problem worth mentioning is that these vulnerabilities would require creating a new IVWA for testing purposes as there is none which would be well documented with test cases focused at deprecated features. Lastly these outdated web applications relaying on browsers such as Internet Explorer are probably legacy services which are no longer being developed, thus creating such a tool might not contribute to the security of these legacy applications. For this reasons this option was dismissed.

As no other options were know at this time, implementing a plugin for some other tool was a better choice. From the tools benchmarked in this thesis only ZAP and Arachni support plugins and ZAP is according to the Benchmark a better tool. On top of that Zap has better guide for how to develop plugins. The decision to improve XSS detection capabilities of ZAP is based on that on the ZAP official website where there are benchmarks results against several IVWAs. There is also stated, that any failed tests are a good candidate for fixing. [47] None of the benchmark results test for SQLi however two of the result sets include XSS cases in which ZAP fails. [48], [49]. These reasons lead to the decision to implement a ZAP plugin which would focus on improving the capabilities of XSS detection.

4.1 About ZAP

As ZAP was chosen to be the platform for this plugin it deserves better explanation on how the tool behaves and works internally. Though the following information is not by any means exhaustive, it provides necessary information to better explain the context in which the plugin

is being created.

4.1.1 Scanning in ZAP

In order to detect any vulnerabilities there are a few thing must happen in ZAP. Firstly the website will be crawled for all reachable pages. While the website is being crawled passive scan is being run as well. The passive scan goes only through the requests and responses which were generated during the crawling and does not create any new requests towards the web application by itself. This type of scanning can for example reveal an incorrectly set header.

After the passive scan with crawl is done an active scan can be run. Active scan is separated from the passive scan as the nature of active scan can potentially do something harmful to the web application since a new request, through which an attempt to demonstrate a vulnerability, is carried out. This kind of scan is more essential to revealing XSS vulnerabilities as that is something that might not be possible to reveal just by using passive scan since the nature of XSS requires injecting some payload in order to prove the website vulnerable.

Active scan rules and passives scan rules are separated from the main application and each other by being separate plugins which are mandatory. On top of that there are alpha and beta versions of these packages which contain more experimental features.

4.2 Implementation of the Plugin

During the implementation the first thing that was chosen to focus on was the failing Google Firing Range test cases. Looking at the benchmark results at the website [48] there are 32 test cases marked as failing. After that the Webseclab was chosen as there are quite a few test cases failing as well. [49]

4.2.1 Setup of the New Plugin

To create a new plugin there is a dedicated guide in the documentation of ZAP. [50] Along with this source it is also useful to bear in mind information from the article of adding new active scan rules [51].

In ZAP there are 3 official main packages for active scanning, these are the ascanrules and its alpha and beta versions. With regard to XSS there is another crucial plugin called domxss. Each rule is then implemented as a class which extends certain class from a few which are provided by ZAP. The main rules of interest from the point for this thesis could be PersistentXssScanRule, PersistentXssSpiderScanRule, PersistentXssPrimeScanRule, CrossSiteScriptingScanRule located in the ascanrules plugin and DomXssScanRule located in the domxss plugin. [52]

As reflected XSS is the the main focus thanks to the escaped test cases provided by Webseclab are aimed at reflected XSS and escaped test cases of Firing Range also seemingly being of reflected nature, the logic of the CrossSiteScriptingScanRule rule class was closely analyzed.

In this class the most important method is called scan. It this method an initial request with a parameter set to a constant value is made. Then the response is analyzed and the constant value is search for. For each match an information about the context in which the value was found is returned. Then each of those are further analyzed, context aware attacks are attempted and subsequent responses are searched for evidence of a successful injection. As this is proven to work and there weren't any obvious issues or shortcomings with this logic it was also used as a logical base for the new plugin.

For the search of reflected values and analysis of their context a special sub package of ascanrules called httputils is being used. For the context of this work this package will also be used in the plugin. Though this will not guarantee that the code in the new plugin could be

without any changes transferred to the ascanrules plugin it will definitely help to ensure that too much changes are not required.

4.2.2 Firing Range

Regarding the firing range the focus was on the escaped XSS section as the low score of 29 % was strangely low. [48] The ultimate reason why the score was so low is that there were a lot of false positives which are not accounted for partially because the website is out of date with new information and partially because the documentation of the Firing Range is non-existing. It is believed that the a few test cases are not exploitable in modern browsers and as such will be taken for false positive test cases in this thesis. This includes following test cases:

- /escape/serverside/encodeUrl/attribute_name
- /escape/serverside/encodeUrl/attribute_quoted
- /escape/serverside/encodeUrl/attribute_script
- /escape/serverside/encodeUrl/attribute_singlequoted
- /escape/serverside/encodeUrl/attribute_unquoted

4.2.2.1 /escape/serverside/escapeHtml/href

This test case is invalid as unfortunately the link which shows this test case and the IVWA just states that the template for this test case is missing.

4.2.2.2 /escape/serverside/*/css_style*

Any cases, where there the payload in injected into a css context, should not be possible to exploit by using XSS in any modern browser. And with the inability to use tag characters, as they are escaped, it is impossible to get out of the css context. For the purposes of this work this makes all of these test cases false positive ones.

4.2.2.3 /escape/serverside/encodeUrl/*

As can be seen in the source code of the Firing Range, `URLEncoder.encode(value, "UTF-8");` is used to escape the payload. [53] This means that every special character that could help for constructing a payload which would actually do something as for example brackets are encoded. [54] This prevents an attacker from constructing any kind of meaningful attack and thus these test cases will be considered as being false positives.

4.2.2.4 /escape/serverside/escapeHtml/body

In this test case html is escaped and the payload is injected in-between body tags like this `<body>$PAYLOAD</body>`. Without creating a context where JavaScript could be executed, which is impossible without the use of the tags, it is impossible to perform XSS. This then is another false positive case.

4.2.2.5 /escape/serverside/escapeHtml/head

This case is the exact same as in the /escape/serverside/escapeHtml/body case as though they are different in name, their content is the same. Even if the payload would be injected in-between the head tags like this `<head>$PAYLOAD</head>` it would still not be exploitable for the same reasons injection in-between body tags is.

4.2.2.6 /escape/serverside/escapeHtml/textarea

For the same reasons as head and body, XSS cannot be performed on this test case.

4.2.2.7 /escape/serverside/escapeHtml/body_comment

Though some old versions of Internet Explorer could be exploited inside of an html comment it would still require the tag characters to be usable, which they are not and thus this context cannot be escaped as in the above cases. [55]

4.2.2.8 /escape/js/*

Two of the three test cases are exploitable. On both /escape/js/encodeURIComponent and /escape/js/html_escape the vulnerability can be demonstrated for example by injecting `alert(1)`. However the /escape/js/escape test case is not vulnerable as it does encode the brackets. Detection of these vulnerabilities were not implemented as for proper detection of this kind of vulnerability it would require to interpret JS code inside the plugin which is quite complex task and would be out of scope of this thesis as to the authors knowledge there is no java library which would provide required functionality.

4.2.3 Webseclab

There are 7 test cases marked as failing against Webseclab of which there are 3 false positives. [49] These false positives are not possible to be fixed by the extension as to the author's knowledge there is no way in zap to mark an alert caused by another extension as false positive Even if it would be possible it still might be a bad idea to do so as then this extension could mark false positives wrongly outside of the benchmarks, this should be handled inside each extension. In this regard the goal is to trigger as few false positives as possible with the new extension. Thus there are 4 cases in this IVWA in which ZAP fails to detect real vulnerability. Bellow there is a section for each of the cases in which the solution to it is explained in more detail.

4.2.3.1 backslash1

In this test case the server fails to escape Unicode sequences. This means that though payload like `<script>alert(1)</script>` would have been escaped to `\u003c\u0073\u0063\u0072\u0069\u0070\u0074\u003e\u0061\u006c\u0065\u0072\u0074\u0028\u0031\u0029\u003c\u002f\u0073\u0063\u0072\u0069\u0070\u0074\u003e` Server won't escape this sequence and the payload will trigger.

Though working in theory, this is not possible to implement in the ZAP extension. According to the article on how to write active scan rules [51], for all requests toward the target a special method provided by the ZAP itself should be used. This method either need an URL encoded input, or expects it, throwing an exception when it is not. Unfortunately URL encoding UTF-16 payload is breaking the desired effect as the backslash characters are escaped to %5C which ruins the desired outcome. The above stated example would turn to `%5Cu003c%5Cu0073%5Cu0063%5Cu0072%5Cu0069%5Cu0070%5Cu0074%5Cu003e%5Cu0061%5Cu006c%5Cu0065%5Cu0072%5Cu0074%5Cu0028%5Cu0031%5Cu0029%5Cu003c%5Cu002f%5Cu0073%5Cu0063%5Cu0072%5Cu0069%5Cu0070%5Cu0074%5Cu003e` which will prevent the payload from triggering.

4.2.3.2 js3

Test-cases js3 and its counterpart js3_fp checks whether the tested tool is capable of differentiating between injection into quoted JS context and an unquoted one.

An example of a payload that would be triggered against js3 could be `in=1,x:alert(12345)` as is mentioned as proof of concept in the IVWA. When the request is made by the tool it is needed

to locate this exact string in the response. In order to see whether the payload was injected into a quoted JS context or not firstly a naive solution was used.

This naive solution included only searching for the first quote to the left from the injected string and then counting the number of times it is there (left of the injected string) until a `">"` is found. If the number of quotation marks is even the payload is not quoted otherwise it is. This worked for the benchmark but caused problems with the js3_search_fp where the quotation is a bit more complex. Thus proper solution was needed.

The final version of implemented solution detects the quotation context by looking at each character from the end of the opening script tag to the start of the payload. At the beginning of this algorithm the payload is marked as not quoted and the last quote variable is empty. If the char currently being looked at is not a quote nothing happens, if the char is a quote but the last quote is empty the payload is marked as quoted and the last quote variable is assigned the value of the current character. If the character is a quote and the last quote variable is not empty and the current character equals the last quote variable the payload is marked and not quoted. Its implementation, which may be better to read, can be seen in code snippet A.1. This version seems to work great and does not cause any false positives.

4.2.3.3 js3_notags

This test case is based on the same principle as the js3 test case. The only difference is that this time it filters out `"<"` and `">"` tags. This would have been a more complicated to deal with if the js3 case was solved is a way that would depend on the `"<"` and `">"` tags being used, which it is not. This means that this case is solved without doing any extra changes to the quotation detection algorithm. The false positive pair for this test case js3_notage is also not being triggered as the implementation A.1 of above described algorithm is able to detect that the payload is being injected inside the quotes.

4.2.3.4 enc2

The enc2 test case is focusing on the ability to escape quoted context of the tested tool. To demonstrate this let's imagine that the payload is being injected in place of `$FOO` in following JS context `<script>var f = '$FOO';</script>`. It would be possible to inject a payload which starts witch a quote. Let's choose `';alert(xss)//` as such payload. Injecting it in place of `$FOO` creates this script to be handled to the client browser `<script>var f = '';alert(xss)//';</script>`. This allows the payload to escape the original quoted context of JavaScript and to be executed.

Unfortunately this test case escapes the quotation mark by putting backslash before it. This results in situation where if the previous payload is injected the resulting code is `<script>var f = '\';alert(xss)//';</script>` which means that the escaping attempt was not successful and the payload is not being triggered. But with the information that the backslash itself is not being escaped by the server it is still possible to escape the quotation in this case. Adding the backslash before the quotation mark, changing the payload to `\';alert(xss)//`, results that in the response there is `<script>var f = '\\';alert(xss)//';</script>`. The backslash which was added by server to escape the quotation mark is escaped by the backslash in the injection. This means that the injected quotation mark is interpreted as a end of quotation and the rest of the payload will be executed.

Implementing this is quite straightforward. Simply when it is detected that the payload is quoted first the regular quote escape is tried and when it fails the second one with the backslash escape is performed.

In the corresponding false positive test case enc2_fp backslashes are escaped too preventing for the use of this technique. This is solved by incorporating the `\'` escape not to the beginning of the payload but inside of it. This means that when the tool looks for the reflected payload it won't be found as it is mutated inside.

To demonstrate this consider following payload `\';alert(1)//`. In the response then `<script> var f = '\\\';alert(xss)//';</script>`. The problem here is that the original payload is still a substring of the response and thus could be falsely recognized as successful injection. But when the backslash escape in not on the beginning or the end of the payload this is negated. The payload can then look for example like this `1\';alert(1)//` which would result in `<script>var f = '1 \\\';alert(xss)//';</script>` as a response. In this case the payload is no longer a substring of the response.

4.3 Benchmarking the Newly Created Plugin

4.3.1 Firing Range

In the firing range there were no new false positives triggered. Additionally, the newly created plugin had correctly recognized vulnerabilities in following test cases:

- /escape/serverside/escapeHtml/attribute_script
- /escape/serverside/escapeHtml/js_assignment
- /escape/serverside/escapeHtml/js_comment
- /escape/serverside/escapeHtml/js_quoted_string
- /escape/serverside/escapeHtml/js_singlequoted_string
- /escape/serverside/escapeHtml/js_slashquoted_string
- /escape/serverside/escapeHtml/js_eval

4.3.2 Webseclab

Regarding the Webseclab benchmark, the plugin successfully detects three more previously discussed test cases which were not detected by ZAP before while not detecting any of the false positives.

4.3.3 OWASP Benchmark

To be sure that the plugin does not cause any new false positives it was also tested against the OWASP Benchmark and it did not trigger any false positive test cases.

Conclusion

The main goals of this thesis were to develop a a plugin or a tool extending XSS or SQLi detection capabilities. In order to successfully reach this goal, the following had to be done.

Firstly, SQLi and XSS vulnerabilities were researched in section 1. Secondly, a number of IVWAs were analyzed in section 2. The most interesting ones are OWASP Benchmark and Security Crawl Maze and both of them were later chosen in section 3 to benchmark existing web scanners. OWASP Benchmark was chosen as it contains a big number of XSS and SQLi test-cases compared to other IVWAs. On the other hand, Security Crawl Maze was chosen because it focuses on the crawling ability of the tested tool. Thirdly, an analysis of web application scanners was carried out in section 3. ZAP and Arachni were the best performing solutions against the OWASP Benchmark and Security Crawl Maze.

Lastly, a ZAP plugin detecting XSS vulnerabilities was implemented. This plugin now correctly detects whether a payload was injected into a properly escaped and quoted JavaScript context. Thanks to this a better score was reached against Webseclab. Tests of this plugin against Google Firing Range and OWASP Benchmark did not show any new false positives to be incorrectly flagged as vulnerable. Implementation and testing of this plugin is described in section 4.

In conclusion, though there is more to be improved in XSS detection in ZAP, the created plugin now successfully detects previously overlooked vulnerabilities. ZAP and Arachni was established according to the benchmarks to be the best open-source tool in terms of XSS and SQLi detection and together OWASP Benchmark and Security Crawl Maze were determined to be the best way to benchmark a vulnerability detection tool.

Appendix A

Code snippets

Code listing A.1 JS quotation detection algorithm

```
private boolean isInJsQuotes(HtmlContext context) {
    // get start of script
    int scriptStart = 0;
    for (int i = context.getStart() - 1; i > 0; i--) {
        char chr = htmlPage.charAt(i);
        if (chr == '>') {
            scriptStart = i;
            break;
        }
    }
    int scriptOffset = scriptStart;
    char lastQuote = 0;
    boolean isQuoted = false;

    while ( scriptOffset < context.getStart() ){
        char chr = htmlPage.charAt(scriptOffset);
        scriptOffset++;
        if (!isQuote(chr)){
            continue;
        }
        if ( !isQuote(lastQuote) ){
            isQuoted = true;
            lastQuote = chr;
            continue;
        }
        if ( lastQuote != chr ){
            continue;
        }
        lastQuote = 0 ;
        isQuoted = false;

    }
    if (lastQuote != 0){
        context.setJsQuote(String.valueOf(lastQuote));
    }
    return isQuoted;
}
```

Bibliography

1. HALFOND, William G; VIEGAS, Jeremy; ORSO, Alessandro, et al. A classification of SQL-injection attacks and countermeasures. In: *Proceedings of the IEEE international symposium on secure software engineering*. IEEE, 2006, vol. 1, pp. 13–15. Available also from: https://www.cc.gatech.edu/fac/Alex.Orso/papers/halfond.viegas.orso.ISSSE06.pdf.
2. MOZILLA. *Using HTTP cookies* [online]. 2022. [visited on 2022-02-20]. Available from: https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies.
3. CHANDRASHEKHAR, Roshni; MARDITHAYA, Manoj; THILAGAM, Santhi; SAHA, Dipankar. SQL Injection Attack Mechanisms and Prevention Techniques. In: *Proceedings of the 2011 International Conference on Advanced Computing, Networking and Security*. Surathkal, India: Springer-Verlag, 2011, pp. 524–533. ADCONS'11. ISBN 9783642292798. Available from DOI: 10.1007/978-3-642-29280-4_61.
4. BUEHRER, Gregory; WEIDE, Bruce W.; SIVILOTTI, Paolo A. G. Using Parse Tree Validation to Prevent SQL Injection Attacks. In: *Proceedings of the 5th International Workshop on Software Engineering and Middleware*. Lisbon, Portugal: Association for Computing Machinery, 2005, pp. 106–113. SEM '05. ISBN 1595932054. Available from DOI: 10.1145/1108473.1108496.
5. LITCHFIELD, David. *Web Application Dissembly with ODBC Error Messages* [online]. 2002. [visited on 2022-05-03]. Available from: https://www.davidlitchfield.com/WebApplicationDisassemblywithODBCErrorMessages.pdf.
6. FAYÓ, Esteban Martínez. *Advanced SQL injection in Oracle databases* [online]. 2005. [visited on 2022-05-04]. Available from: https://www.orkspace.net/secdocs/Web/SQL%20Injection/Advanced%20SQL%20Injection%20In%20Oracle%20Databases.pdf.
7. ANLEY, Chris. *(more) Advanced SQL Injection* [online]. NGSSoftware Insight Security Research, 2002 [visited on 2022-05-04]. Available from: https://img2.helpnetsecurity.com/dl/articles/more_advanced_sql_injection.pdf.
8. STOCK, Andrew van der; GLAS, Brian; SMITHLINE, Neil; GIGLER, Torsten. *OWASP Top 10:2021* [online]. OWASP Top 10 team, 2021 [visited on 2022-04-04]. Available from: https://owasp.org/Top10/.
9. LIU, Miao; ZHANG, Boyu; CHEN, Wenbin; ZHANG, Xunlai. A Survey of Exploitation and Detection Methods of XSS Vulnerabilities. *IEEE Access*. 2019, vol. 7, pp. 182004–182016. Available from DOI: 10.1109/ACCESS.2019.2960449.
10. SELVARANI, Venkata. A survery on XSS attacks and its counter measures in network security. *Journal of Analysis and Computation*. 2020. Available also from: https://www.ijaconline.com/wp-content/uploads/2020/02/selvarani-paper-1.pdf.

11. NIDECKI, Tomasz Andrzej. *Mutation XSS in Google Search*. 2019. Available also from: https://www.acunetix.com/blog/web-security-zone/mutation-xss-in-google-search/.
12. HEIDERICH, Mario; SCHWENK, Jörg; FROSCH, Tilman; MAGAZINIUS, Jonas; YANG, Edward Z. MXSS Attacks: Attacking Well-Secured Web-Applications by Using InnerHTML Mutations. In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*. Berlin, Germany: Association for Computing Machinery, 2013, pp. 777–788. CCS '13. ISBN 9781450324779. Available from DOI: 10.1145/2508859.2516723.
13. MOZILLA. *¡iframe¿: The Inline Frame element* [online]. 2022. [visited on 2022-04-03]. Available from: https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe.
14. MLT. *A tale of eBay XSS and Shoddy incident response* [online]. 2016. [visited on 2022-04-03]. Available from: https://ret2libc.wordpress.com/2016/01/11/a-tale-of-ebay-xss-and-shoddy-incident-response/.
15. MESELEM, Malik. *bWAPP an extremely buggy web app* [online]. ©2022. [visited on 2022-04-09]. Available from: https://itsecgames.com/index.htm.
16. MESELEM, Malik. *bWAPP - README* [online]. 2018. [visited on 2022-04-09]. Available from: https://sourceforge.net/projects/bwapp/.
17. DOUPE, Adam. *WackoPicko Vulnerable Website* [online]. OWASP, [2017] [visited on 2022-04-04]. Available from: https://github.com/adamdoupe/WackoPicko.
18. DOUPÉ, Adam; COVA, Marco; VIGNA, Giovanni. Why Johnny can't pentest: An analysis of black-box web vulnerability scanners. In: *Detection of Intrusions and Malware, and Vulnerability Assessment - 7th International Conference, DIMVA 2010, Proceedings*. 2010, pp. 111–131. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). ISBN 3642142141. Available from DOI: 10.1007/978-3-642-14215-4_7. Copyright: Copyright 2010 Elsevier B.V., All rights reserved.; 7th GI International Conference on Detection of Intrusions and Malware and Vulnerability Assessment, DIMVA 2010 ; Conference date: 08-07-2010 Through 09-07-2010.
19. *wavsep* [online]. 2015. [visited on 2022-04-09]. Available from: https://sourceforge.net/projects/wavsep/.
20. GOOGLE. *wavsep* [online]. [2014]. [visited on 2022-04-09]. Available from: https://code.google.com/archive/p/wavsep/.
21. OWASP. *WebGoat 8: A deliberately insecure Web Application* [online]. [2022]. [visited on 2022-04-04]. Available from: https://github.com/WebGoat/WebGoat.
22. OWASP. *OWASP WebGoat* [online]. ©2022. [visited on 2022-04-04]. Available from: https://owasp.org/www-project-webgoat/.
23. OWASP. *OWASP Juice Shop* [online]. ©2014–2022. [visited on 2022-04-08]. Available from: https://owasp.org/www-project-juice-shop/.
24. KIMMINICH, Björn et al. *OWASP Juice Shop* [online]. 2022, ©2014–2022. [visited on 2022-04-08]. Available from: https://github.com/juice-shop/juice-shop/.
25. OWASP. *OWASP Mutillidae II* [online]. 2021. [visited on 2022-04-10]. Available from: https://github.com/webpwnized/mutillidae.
26. OWASP. *OWASP Mutillidae II* [online]. ©2022. [visited on 2022-04-10]. Available from: https://www.zaproxy.org/docs/scans/webseclab/#section-reflected.
27. OWASP. *OWASP Benchmark* [online]. ©2022. [visited on 2022-04-10]. Available from: https://owasp.org/www-project-benchmark/.

28. [GOOGLE]. *Security Crawl Maze* [online]. 2019. [visited on 2022-04-15]. Available from: https://github.com/google/security-crawl-maze.
29. SAVINTSEV, Dmitry. *Webseclab* [online]. 2018. [visited on 2022-12-25]. Available from: https://github.com/yahoo/webseclab/blob/master/README.md.
30. GOOGLE INC. *What is Firing Range?* [Online]. ©2014. [visited on 2023-01-02]. Available from: https://github.com/google/firing-range.
31. PAN, Yuanyuan. Interactive Application Security Testing. In: *2019 International Conference on Smart Grid and Electrical Automation (ICSGEA)*. 2019, pp. 558–561. Available from DOI: 10.1109/ICSGEA.2019.00131.
32. ZAP DEV TEAM. *OWASP ZAP – Statistics* [online]. ©2022. [visited on 2022-04-15]. Available from: https://www.zaproxy.org/docs/statistics.
33. ZAP DEV TEAM. *OWASP ZAP – What does ZAP test for?* [Online]. ©2022. [visited on 2022-04-15]. Available from: https://www.zaproxy.org/faq/what-does-zap-test-for.
34. ZAP DEV TEAM. *OWASP ZAP – Active Scan Rules* [online]. ©2022. [visited on 2022-04-15]. Available from: https://www.zaproxy.org/docs/desktop/addons/active-scan-rules.
35. ZALEWSKI, Michal; HEINEN, Niels; ROSCHKE, Sebastian. *Skipfish - web application security scanner* [online]. 2013. [visited on 2022-04-13]. Available from: https://gitlab.com/kalilinux/packages/skipfish.
36. DAMELE, Bernardo A. G.; STAMPAR, Miroslav. *Sqlmap* [online]. ©2006–2022. [visited on 2022-04-13]. Available from: https://sqlmap.org.
37. PSY. *XSSer: Cross Site "Scripter"* [online]. [2021]. [visited on 2022-04-13]. Available from: https://xsser.03c8.net/.
38. PSY. *Xsser* [online]. 2020. [visited on 2022-04-13]. Available from: https://code.03c8.net/epsylon/xsser.
39. LORD EPSYLON. *XSS_Hacking_tutorial* [online]. [2016]. [visited on 2022-04-16]. Available from: https://xsser.03c8.net/xsser/XSS_for_fun_and_profit_SCG09_(english).pdf.
40. ARACHNI. *Home - Arachni - Web Application Security Scanner Framework* [online]. ©2010–2017. [visited on 2022-04-13]. Available from: https://www.arachni-scanner.com.
41. ARACHNI. *Features - Arachni - Web Application Security Scanner Framework* [online]. ©2010–2017. [visited on 2022-04-13]. Available from: https://www.arachni-scanner.com/features/.
42. W3AF. *W3af's documentation* [online]. ©2014. [visited on 2022-04-15]. Available from: https://docs.w3af.org/en/latest/index.html.
43. SULLO, Chris. *Overview & Description* [online]. 2018. [visited on 2023-01-02]. Available from: https://github.com/sullo/nikto/wiki.
44. SULLO, Chris. *Annotated Option List* [online]. 2021. [visited on 2023-01-02]. Available from: https://github.com/sullo/nikto/wiki/Annotated-Option-List.
45. SURRIBAS, Nicolas. *The web-application vulnerability scanner* [online]. 2006-2022©. [visited on 2022-05-17]. Available from: https://wapiti-scanner.github.io/.
46. NETSCAPE COMMUNICATIONS. *Download Netscape* [online]. ©1994-2022. [visited on 2022-12-26]. Available from: http://www.netscape-communications.com/download/.
47. ZAP DEV TEAM. *ZAP Scans* [online]. ©2022. [visited on 2022-12-27]. Available from: https://www.zaproxy.org/docs/scans/.

48. ZAP DEV TEAM. *ZAP vs Firing Range* [online]. ©2022. [visited on 2022-12-27]. Available from: https://www.zaproxy.org/docs/scans/firingrange/#section-escape.

49. ZAP DEV TEAM. *ZAP vs Webseclab* [online]. ©2022. [visited on 2022-12-27]. Available from: https://www.zaproxy.org/docs/scans/webseclab/#section-reflected.

50. ZAP DEV TEAM. *Creating a New Add-on in zap-extensions* [online]. ©2022. [visited on 2023-01-01]. Available from: https://www.zaproxy.org/docs/developer/creating-new-addon-in-zap-extensions/.

51. BENNETTS, Simon. *Hacking ZAP #4 - Active scan rules* [online]. 2014. [visited on 2023-01-01]. Available from: https://www.zaproxy.org/blog/2014-04-30-hacking-zap-4-active-scan-rules/.

52. ZAP DEV TEAM. *zaproxy/zap-extensions* [online]. ©2022. [visited on 2023-01-02]. Available from: https://github.com/zaproxy/zap-extensions.

53. GOOGLE INC. *ServersideEscape.java* [online]. ©2014. [visited on 2023-01-01]. Available from: https://github.com/google/firing-range/blob/master/src/tests/escape/ServersideEscape.java.

54. ORACLE. *URLEncoder* [online]. ©1993-2022. [visited on 2023-01-01]. Available from: https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/net/URLEncoder.html.

55. CHEATSHEETS SERIES TEAM. *XSS Filter Evasion* [online]. ©2021. [visited on 2023-01-01]. Available from: https://cheatsheetseries.owasp.org/cheatsheets/XSS_Filter_Evasion_Cheat_Sheet.html.

Contents of the enclosed SD Card

- `readme.txt` ..... Contents description
- `pract` ..... Everything related to the practical part
  - `impl` ..... Source code of the implementation
- `thesis-src` ..... Source code of the thesis
- `thesis.pdf` ..... PDF version of the thesis