Security Lab 3

Ghadi Ayoub
Malek Kanaan
Submitted to dr.Ayman Tajeddine

March 2024

All the references used are listed in the end of the report.
Mac address of the machine: 34-E1-2D-4B-B0-EA
IP address of the machine: 192.168.204.1
Both records are from the home's WiFi.
IP address from the university: 10.21.140.21

1 Setup

In order to perform the attacks, we need to install a DVWA on a virtual machine. We have decided to download Kali Linux using a vmware from the respective sources:

- https://www.kali.org/get-kali/kali-platforms (For Kali).
- https://www.vmware.com/products/workstation-player/workstation-player-evaluation.html (For the VM).

1.1 Installing a DVWA

After following the instructions provided in the question sheet which include:

1. Cloning the repository.
2. Moving the DVWA into a specific directory using a superuser privilege.
3. Running apache server.
4. Starting maria db and executing the commands which are listed in the tutorial.

Figure 1: Step 1

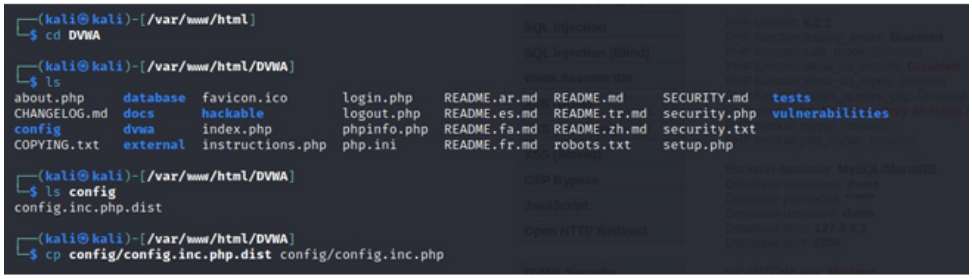


Figure 2: Step 2

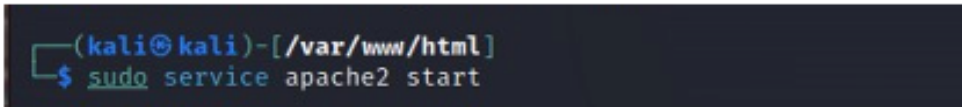


Figure 3: Step 3

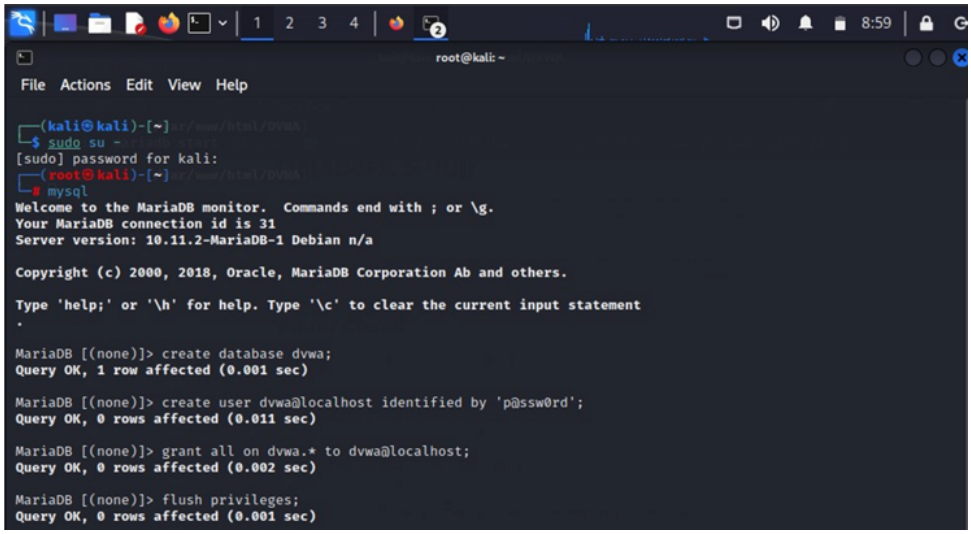


Figure 4: Step 4

We login and get greeted with this interface:

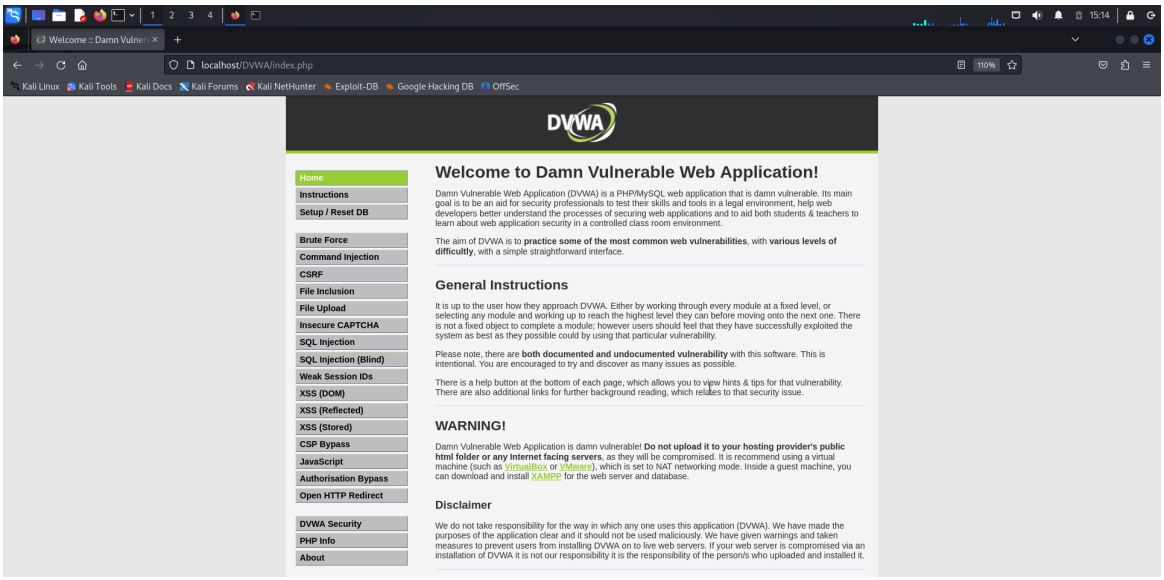


Figure 5: DVWA's interface from Kali linux running on a VM

The installation process was overall smooth, we only had missing resources which we had to download, as explained previously.

2 Command execution and SQL injection challenge

2.1 Low level

2.1.1 Attack methodology

The goal is to remotely find out the user of the web service on the OS, as well as the machines hostname via RCE.

The first strategy is to check if **special characters** are being escaped. We are operating from a bottom up approach, meaning that we start from the simplest to the more complicated input validations. The reason we are starting from this particular point is due to the realization that the source does not sanitize special characters, as indicated in the picture below:

Command Injection Source

vulnerabilities/exec/source/low.php

```php
<?php

if( isset( $_POST[ 'Submit' ]  ) ) {
    // Get input
    $target = $_REQUEST[ 'ip' ];

    // Determine OS and execute the ping command.
    if( stristr( php_uname( 's' ), 'Windows NT' ) ) {
        // Windows
        $cmd = shell_exec( 'ping  ' . $target );
    }
    else {
        // *nix
        $cmd = shell_exec( 'ping  -c 4 ' . $target );
    }

    // Feedback for the end user
    echo "<pre>{$cmd}</pre>";
}

?>
```

Figure 6: Low security command execution code

By inserting any valid IP address with special characters appended to it, one can execute any shell command.

Vulnerability: Command Injection

Ping a device

Enter an IP address: [ ] Submit

```
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.047 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.020 ms
64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.024 ms
64 bytes from 127.0.0.1: icmp_seq=4 ttl=64 time=0.022 ms

--- 127.0.0.1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3050ms
rtt min/avg/max/mdev = 0.020/0.028/0.047/0.010 ms
help
index.php
source
```

Figure 7: Command injection low security level output

This is the result of simply injecting the following command: `170.0.0.1 && ls`.

2.1.2 User Risk

We have managed to leak information by simply appending a simple ls command. A hacker could use more advanced commands and gather important information about the system that could help him greatly into using it for malicious intent. For example, they could use `170.0.0.1 && cat/ect/password`, leaking highly sensitive information.

2.1.3 Potential security prevention

We can add lines to the code that escape special characters using a function like `filter_var()`, this could help us for this kind of attack **only**.

2.2 Medium level

2.2.1 Attack methodology

Moving on to the medium level, the source code gets updated to the one below:

Command Injection Source

vulnerabilities/exec/source/medium.php

```php
<?php

if( isset( $_POST[ 'Submit' ]  ) ) {
    // Get input
    $target = $_REQUEST[ 'ip' ];

    // Set blacklist
    $substitutions = array(
        '&&' => '',
        ';'  => '',
    );

    // Remove any of the characters in the array (blacklist).
    $target = str_replace( array_keys( $substitutions ), $substitutions, $target );

    // Determine OS and execute the ping command.
    if( stristr( php_uname( 's' ), 'Windows NT' ) ) {
        // Windows
        $cmd = shell_exec( 'ping  ' . $target );
    }
    else {
        // *nix
        $cmd = shell_exec( 'ping  -c 4 ' . $target );
    }

    // Feedback for the end user
    echo "<pre>{$cmd}</pre>";
}

?>
```

Figure 8: Command injection medium source code

We can see that in the code, the author is black-listing the special characters we have used earlier. However, since they are not using a library and are manually black-listing symbols, we can still use other symbols and it would still work:

Vulnerability: Command Injection

Ping a device

Enter an IP address: Submit

```
help
index.php
source
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.015 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.019 ms
64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.022 ms
64 bytes from 127.0.0.1: icmp_seq=4 ttl=64 time=0.023 ms

--- 127.0.0.1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3061ms
rtt min/avg/max/mdev = 0.015/0.019/0.023/0.003 ms
```

Figure 9: Command line injection medium output

We have received the same output by just removing one & from the same injection we have previously used.

2.2.2 User Risk

The same risks are imposed on the users. One can append any command without having to compromise with the format of the output.

2.2.3 Potential security prevention

Since the problem remains relatively the same, we can use the same function used earlier to fix it.

2.3 High level

In this security level, the author is checking for all the potential special characters, still statically though. If we look carefully however, it seems that the symbol used for piping is checked with a space after it.

Figure 10: Command line high level security source code

We can use the command in the following form: `170.0.0.1 |ls` (or anything after the piping) and it will work. It seems however, that the user has hidden the content of the directory, we can use another command to still gather information, like `170.0.0.1 |pwd`. We thus get the following output:

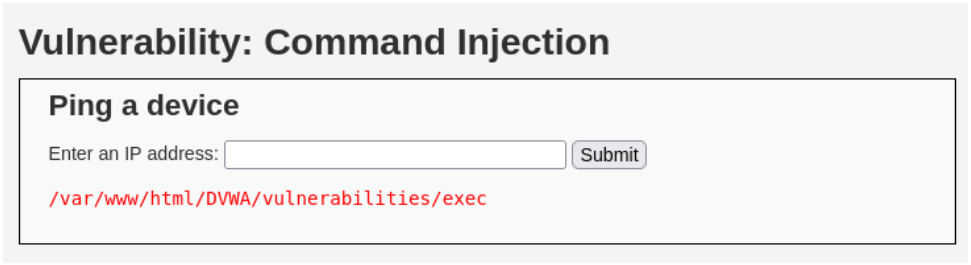


Figure 11: Command line injection high output

2.3.1 User Risk

While the frequency of risks is lower now, hackers can still abuse the symbol used for piping to append commands.

2.3.2 Potential security prevention

To fix this issue, we can simply delete the space or use the function presented earlier in all the other subsections.

2.4 Low level (SQL injection)

SQL injections are commands you use in order to perform read/write/execute operations on the database.

2.4.1 Attack methodology

If we look at the source code, we notice that the `Query` variable does not implement a security measure to piggyback another command, and so, by just inserting `'UNION SELECT first_name, password FROM users;#`, we retrieve the following data:

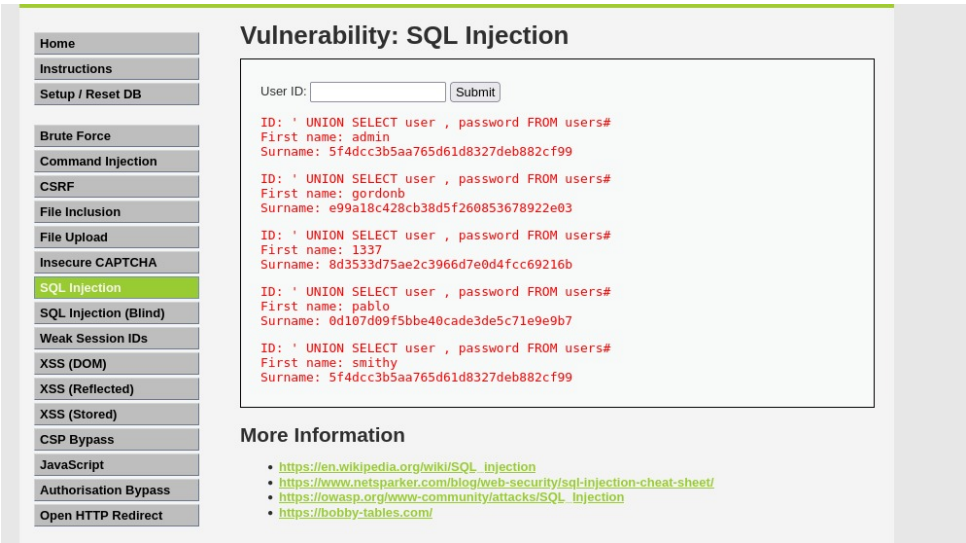


Figure 12: SQL injection low

2.4.2 User risk

While in this example, we have just leaked some information present in the database, we can also insert queries to DOS the device, edit it (loss of integrity) and access higher privileges.

2.4.3 Potential security prevention

Prevention includes parameterized queries (most effective), proper input validation and stored procedures.

2.5 Medium level (SQL injection)

2.5.1 Attack methodology

We first notice that the input format has changed to only allow us to insert specific indices. We can work around this however through the developer console of the web browser:

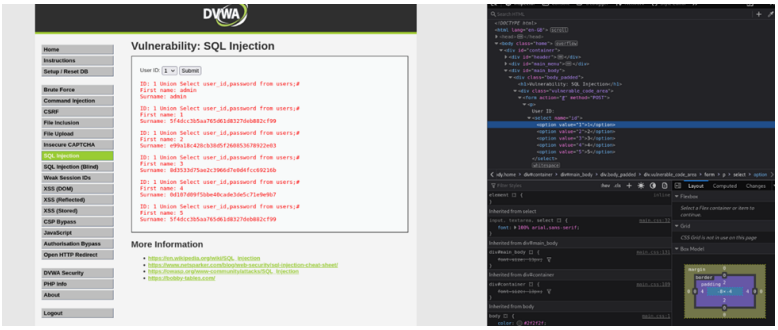


Figure 13: From the browser's console

From there, we inject the same command we did earlier and return a successful response.

2.5.2 User risk

The same risks are involved since we are doing the same attack.

2.5.3 Potential security prevention

The same policies are also involved.

2.6 High level (SQL injection)

2.6.1 Attack methodology

Despite the security level being set to "high", we can use the same command again and it will work as expected, here is the proof:

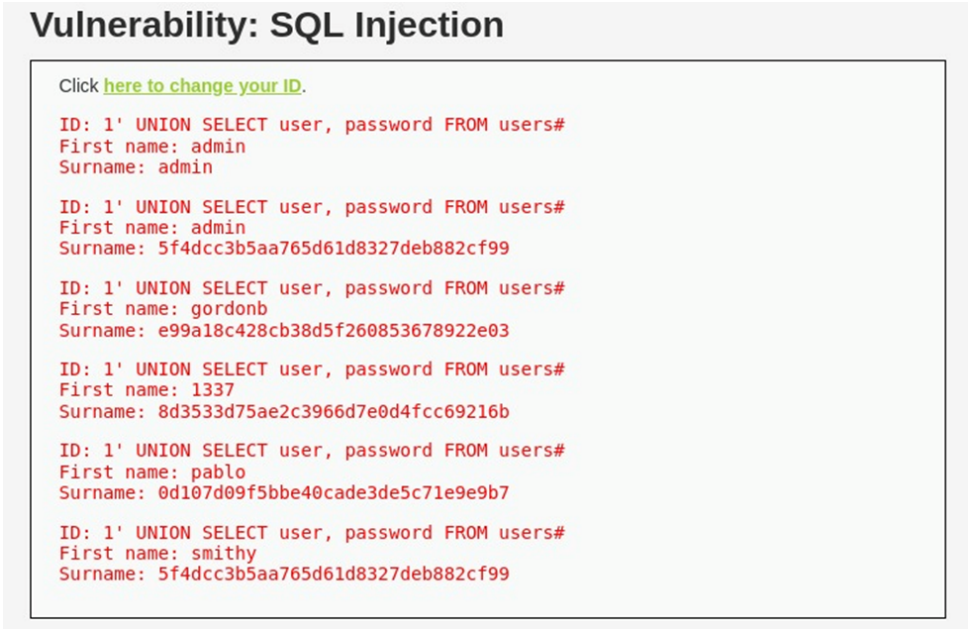


Figure 14: SQL injection high

2.6.2 User risk

The same risks are involved since we are doing the same attack.

2.6.3 Potential security prevention

The same policies are also involved.

2.7 Low level (SQL Blind)

From https://portswigger.net/ : "Blind SQL injection occurs when an application is vulnerable to SQL injection, but its HTTP responses do not contain the results of the relevant SQL query or the details of any database errors. Many techniques such as UNION attacks are not effective with blind SQL injection vulnerabilities."

2.7.1 Attack methodology

Given the former definition, we thus tried to get a response from the website by using the `1' and sleep(5)` query, which resulted in a small delay in loading the page, indicating a positive response and confirming that here forward we can perform structured attacks.

Now, in order to get the version of the database we repeatedly performed the following command: `1' substring(int,int)='int`. If we get an ID that exists, it indicates that the number exist, including the opposite.

Figure 15: Case 1

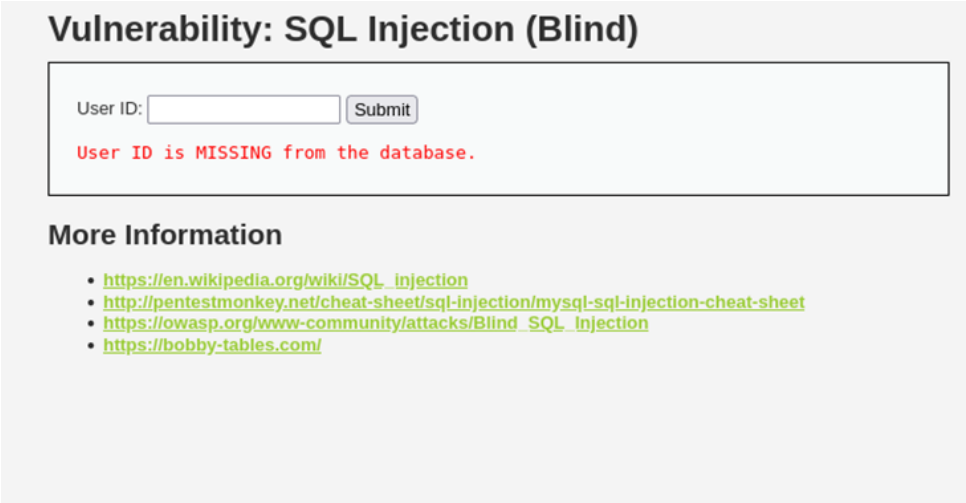


Figure 16: Case 2

After checking that 5 users exist and trying the following commands, we figured out that the version of the database is 10.11:

Command Used	Output
`1' and substring(version(),1,1)='1`	True
`2' and substring(version(),2,1)='0`	True
`3' and substring(version(),3,1)='.'`	True
`4' and substring(version(),4,1)='1`	True
`5' and substring(version(),5,1)='1`	True

2.7.2 User risk

This is simply a subset of what we are have already discussed. We are leaking data that can be used harmfully.

2.7.3 Potential security prevention

The same policies are also involved.

2.8 Medium level (SQL Blind)

2.8.1 Attack methodology

Since the input has also changed here to a similar fashion as the normal SQL's medium level, we had to use the developer's console to inject the same commands we did earlier:

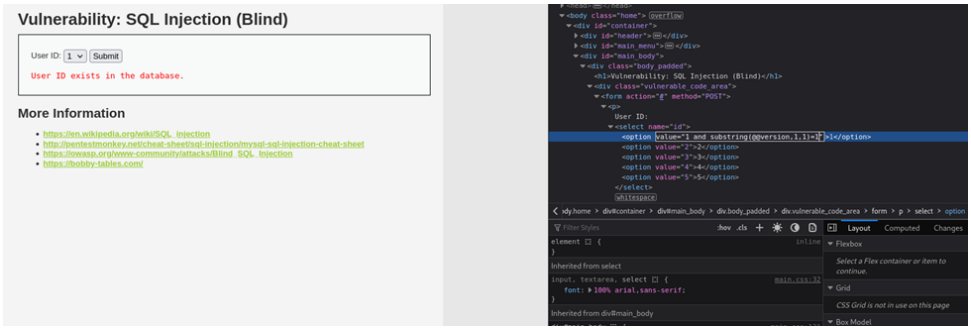


Figure 17: SQL blind medium

2.9 High level (SQL Blind)

The high level involves the same procedure as the low one. This is because the added security does not prevent leakage using the procedure we happened to have used:

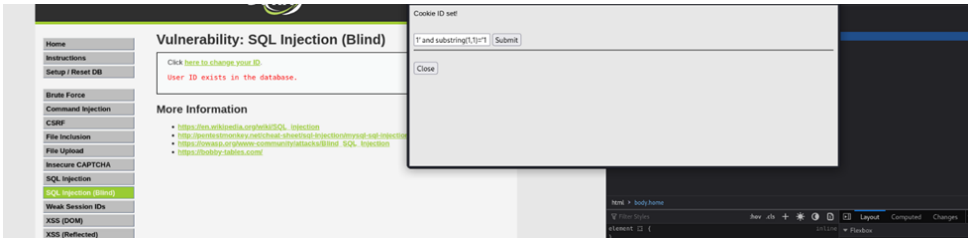


Figure 18: SQL blind high

3 XSS Challenges

Cross side scripting or XSS, is the act of injecting malicious code to hurt subsequent users' output by exploiting vulnerabilities in website that are otherwise benign.

3.1 Low level (Dom)

3.1.1 Attack methodology

If we look at the hint provided, we understand that our starting point is the URL of the page:

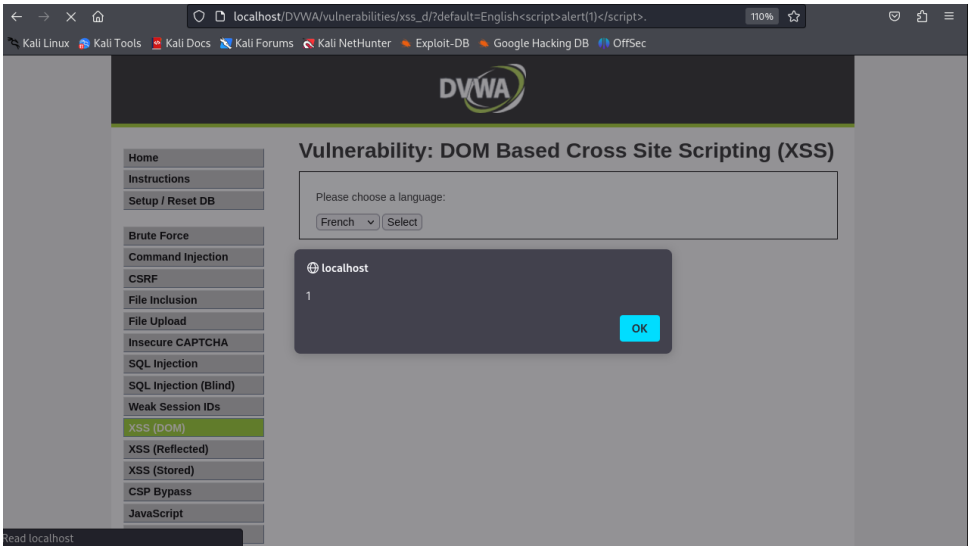


Figure 19: Output using the hint

Our goal will be to exploit this vulnerability to steal the user's cookies.
Replacing the "1" with document.cookie, we get the following output:

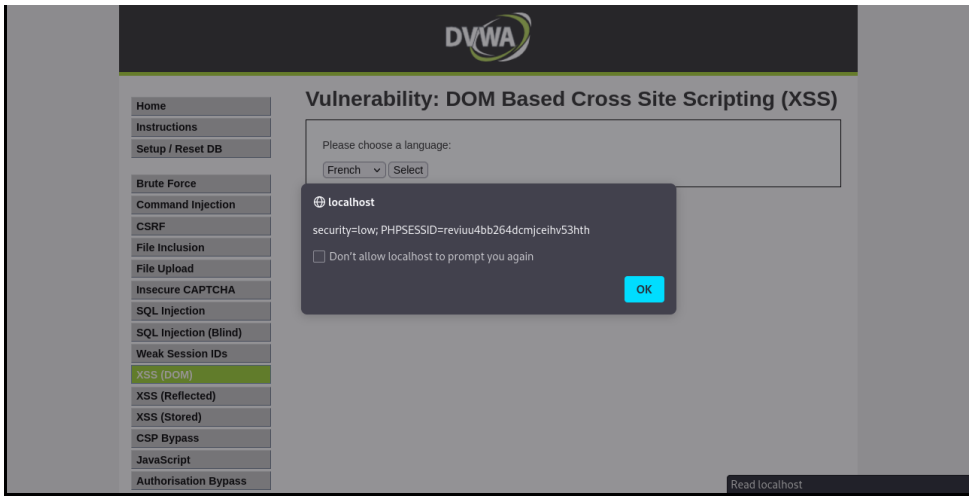


Figure 20: Program output

3.1.2 User risk

Gaining access to the session ID allows one to impersonate the user. Since this ID serves as a token which authenticates the user. This snowballs into a wide open window of opportunities to do harm.

3.1.3 Potential security prevention

Security prevention includes error reporting, HTTP headers implementation, and proper input sanitation.

3.2 Medium level (Dom)

3.2.1 Attack methodology

The user has implemented lines of code to protect themselves against attacks using the script tag. However, using the following URL, we have managed to obtain the previous output:

`http://localhost/DVWA/vulnerabilities/xss default=Englishoption%3E%3C/select%3E%3Cimg%20src=%27x%27 %20onerror=%27alert(document.cookie)%27%3E.`

What is happening here is that we are trying to alter the output by executing a script outside the conventional tag. On error will execute a script without using the actual "script" tag.

3.2.2 User risk

The same risks are involved since the same attack is being done, but using a longer road.

3.2.3 Potential security prevention

Since the issue remains the same, we can use the same security prevention.

3.3 High level (Dom)

3.3.1 Attack methodology

The user here is whitelisting the languages we can use, meaning that we cannot append a new language and inject a script. What we can do however is append a hashtag and then append the same script used earlier like this:
`http://localhost/DVWA/vulnerabilities/xss_d default=English%3`
`Cscript Ealert(document.cookie)C/scriptE`

3.3.2 User risk

The same risks are involved since the same attack is being done, but using a longer road.

3.3.3 Potential security prevention

Since the issue remains the same, we can use the same security prevention.

3.4 Low level (reflected)

Reflected cross side scripting attacks are very similar to the ones where we use the Dom, except in this case, we are injecting the script inside the user input.

3.4.1 Attack methodology

Here, we want to inject a script that could steal the user cookie in the same way we did in the URL earlier, except in the user input. The code does not implement any seccurity policy and will accept any input, thus injecting
`?name=<script>alert(document.cookie);</script>`, we get the following result:

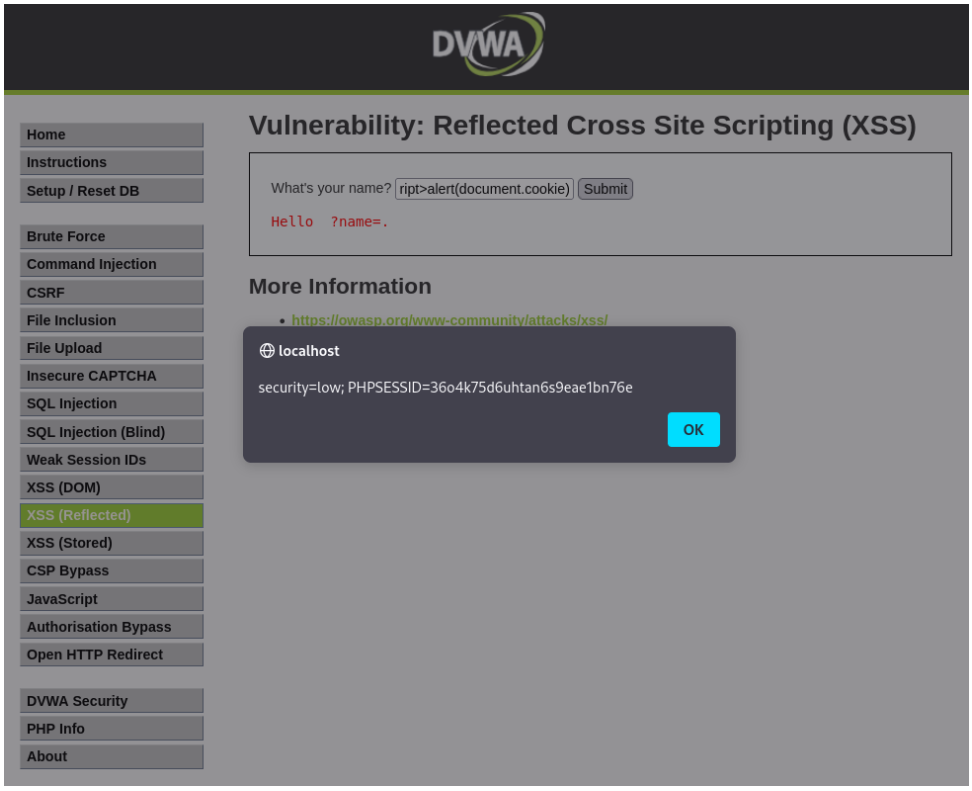


Figure 21: XSS reflected output

3.4.2 User risk

The same user risks are presented as the ones in the previous section.

3.4.3 Potential security prevention

We could whitelist the characters we want, eventually narrowing what the user is allowed to input by a significant margin.

3.5 Medium level (reflected)

3.5.1 Attack methodology

Although the hint points towards using a case sensitive script tag, since the user is checking that we do not use the original one, we decided instead to opt for the same injection we used in the medium level we used earlier:

`option></select><img src='x' on error='alert(document.cookie)'`.
It worked as expected.

3.5.2 User risk

The same risks are involved since the same attack is being done, but using a longer road.

3.5.3 Potential security prevention

Since the issue remains the same, we can use the same security prevention.

3.6 High level (reflected)

In this part, the user is using regular expressions to skip special characters.

3.6.1 Attack methodology

By using the same command we have used earlier, we obtain the same output, since it does not use any of the character that are skipped.

3.6.2 User risk

The same risks are involved since the same attack is being done, but using a longer road.

3.6.3 Potential security prevention

Since the issue remains the same, we can use the same security prevention.

3.7 Low level (Stored)

From the stored word, we understand that the inputs, being the scripts, are being saved somewhere. Usually, they are saved in a database.

3.7.1 Attack methodology

In this section, we have two fields, a name field, with limited characters, and a message field, where we have more room to inject our code. We will use the message field to steal cookies the same way we did earlier. On input `<script>alert(document.cookie)</script>`, we get the following output:

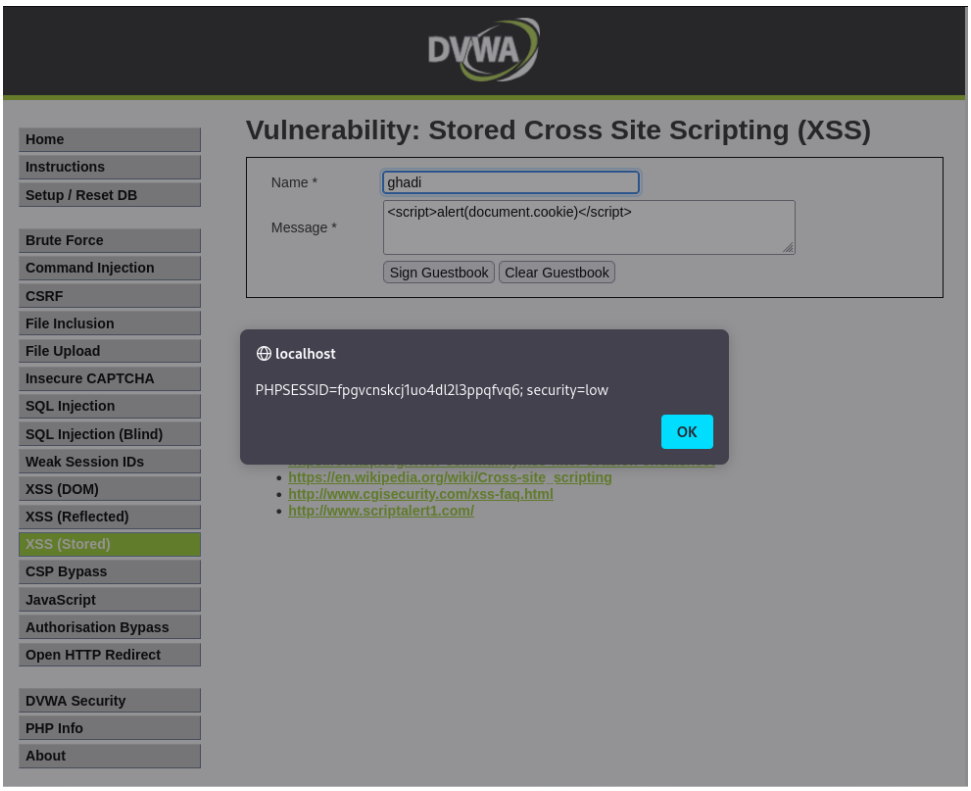


Figure 22: XSS stored output

3.7.2 User risk

The same risks we have explained earlier remain valid.

3.7.3 Potential security prevention

Security prevention include input sanitation, encryption of data in transit, whitelisting characters (global) or simply limiting character length (local) .... pretty much what we have discussed earlier.

3.8 Medium level (Stored)

3.8.1 Attack methodology

In this section, we can see that some security measures were added. The user implements a weak type of sanitation. The user is also handling attacks that use the script tag, however as we have seen from the past section, script tag is case sensitive. However here is the key point, the second policy is applied on the name string without the first one. If we inject something like `<Script>alert(document.cookie)</Script>` in the name placeholder, we can achieve the desired result. We first need to make the input size bigger by editing the code from the client side. Navigating to the table data tag of this particular placeholder, we get the following:

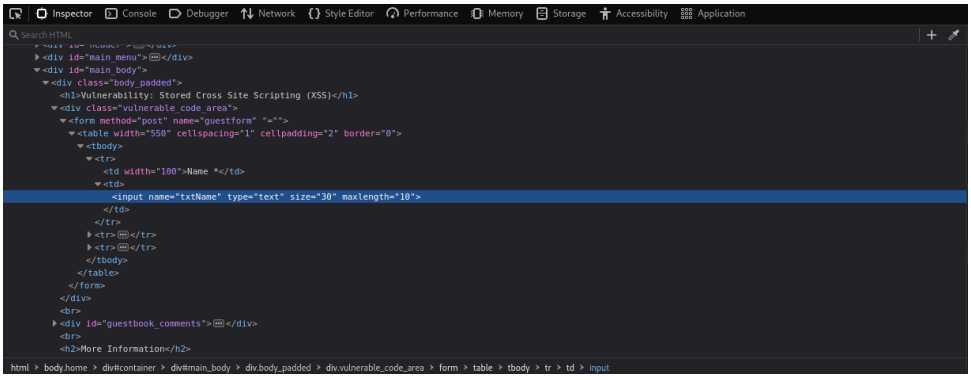


Figure 23: From the developer's console

We simply edit the size to something suitable, and then inject the case sensitive string (`<Script>alert(document.cookie)</Script>`)

3.8.2 User risk

The same risks are involved since the same attack is being done, but using a longer road.

3.8.3 Potential security prevention

Make sure to also sanitize the name placeholder, and fix the size of the size of it using a final variable?

3.9 High level (Stored)

3.9.1 Attack methodology

In this section, the user is implementing the same strong measures for both entries. With one key difference, The name sanitation does not include the htmlspecialchars() step, which would convert special HTML characters to their entity equivalents. We can still use event based techniques to achieve the result we want in the name field. Of course, we would still need to make the size of the name field large enough.
And so, using the following line: `<img src="X" onerror=alert(document.cookie)>` , we manage to steal the session's id.

3.9.2 User risk

The same risks are involved since the same attack is being done, but using a longer road.

3.9.3 Potential security prevention

Make sure to implement the same techniques everywhere and insure consistency in the policies used all across the board. The policies we can use were already discussed.

4 Other Challenges

4.1 Low level (Brute force)

As the term says, brute forcing is the fact of directly and repeatedly trying to access the information of a user. No indirect "clever" tricks are involved.

4.1.1 Attack methodology

In the following section, we will use burp-suite (the one in kali), to check the http requests made and try to craft a replay attack using the password that was inputted.

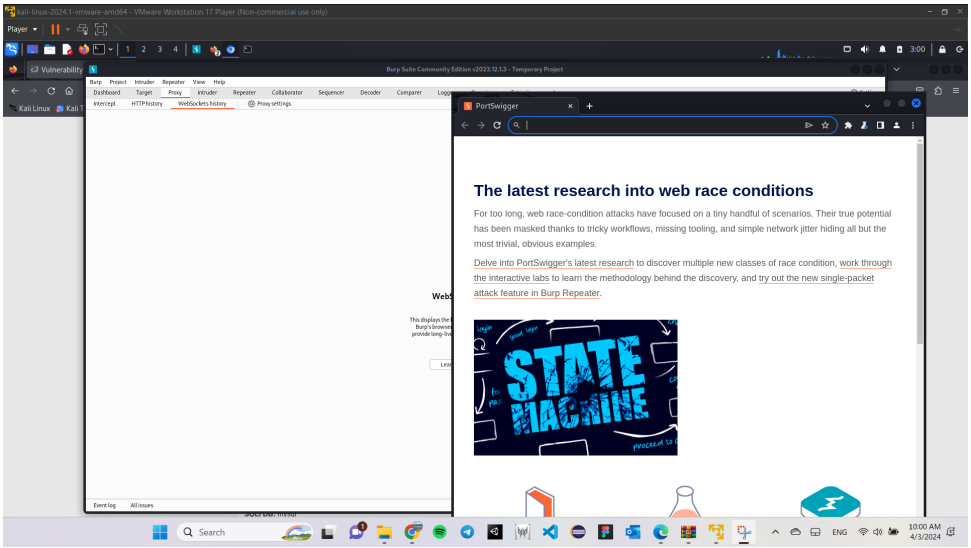


Figure 24: Burp-suite interface

We need to login in to the DVWA website from burp-suite's built-in browser. After doing so, as well as logging in using the admin password, we see the following:

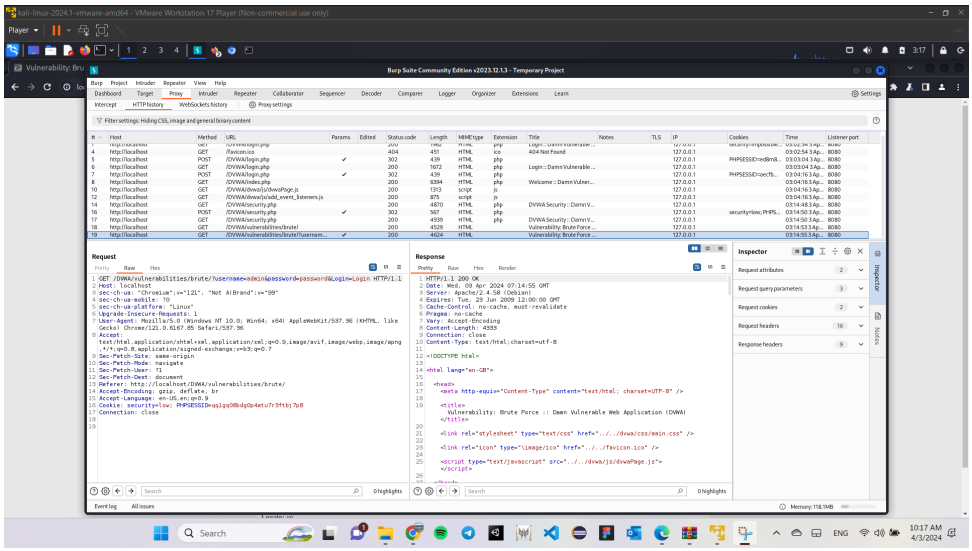


Figure 25: Burp suite http requests

After finding the corresponding request that was performed when we logged in as the admin, we will send it to the intruder. This will allow us to send automated requests with different headers every time.
Next, we will go to the grep-match tab. This is where we will flag each response by mapping them to each request.

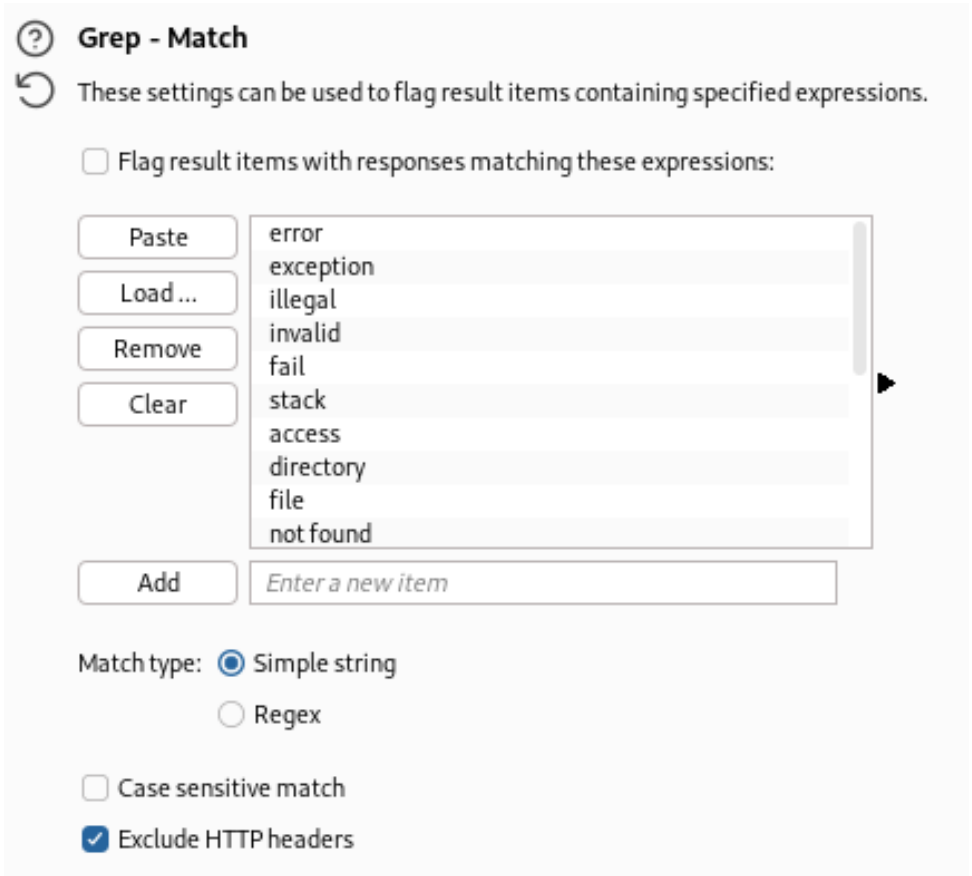


Figure 26: Grep-Match's interface

We will add the following strings: `Welcome to the password protected area` and `Username and/or password incorrect`.

Now in the positions tab, we will keep the sniper option because we only need one set of headers:

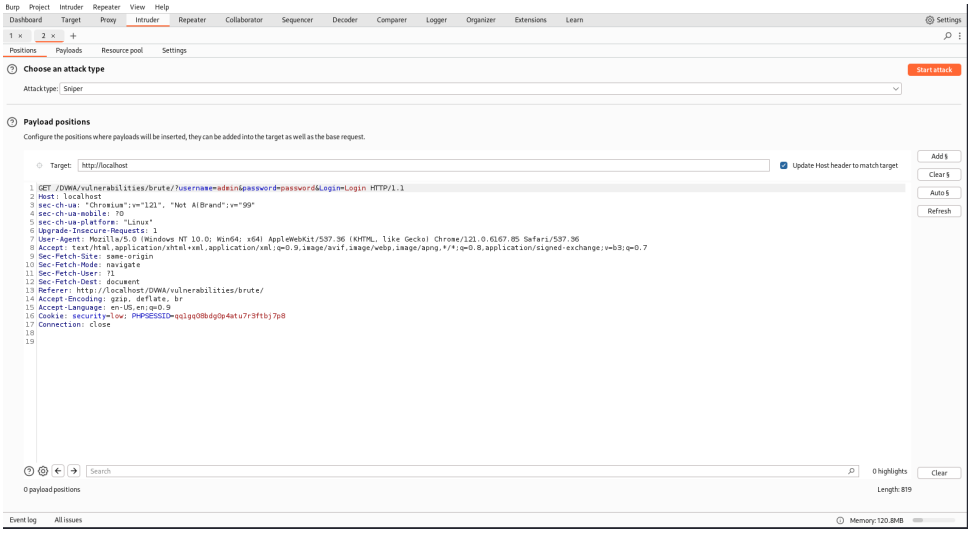


Figure 27: Position tab

We will now need to provide the software with data that will be autonomously requested, you can find the list of inputs in the link below: https://raw.githubusercontent.com/danielmiessler/SecLists/master/Passwords/Common-Credentials/10-million-password-list-top-10000.txt
Here is the list after adding the passwords:

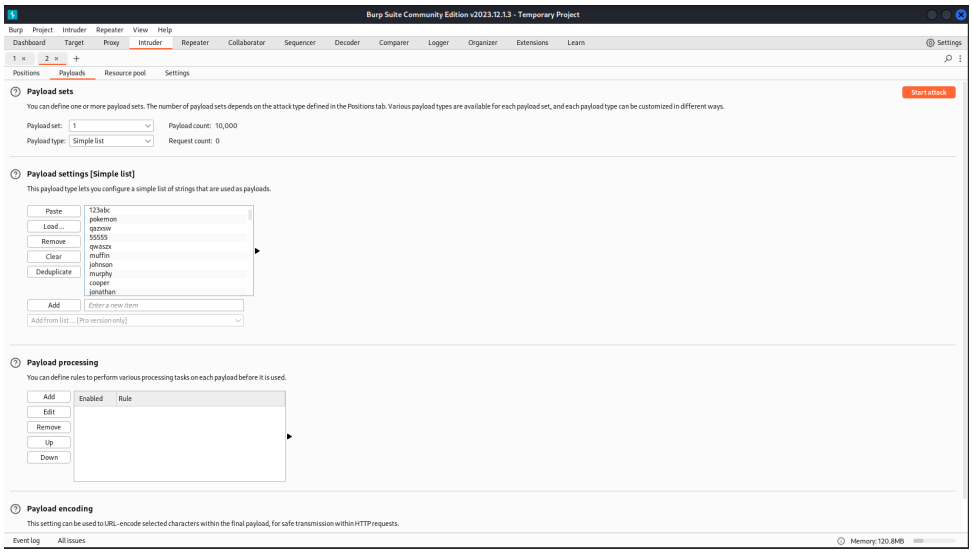


Figure 28: Full list after adding the passwords

Now that everything is set, we can start our attack:

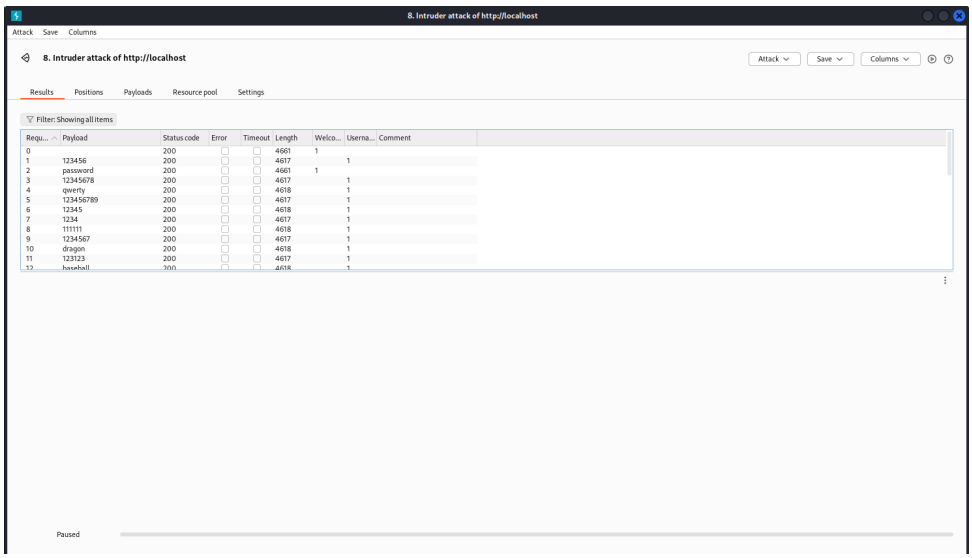


Figure 29: List of trials

Here we can see a list of the passwords that were tried. a 1 under each response text signifies whether the attack was successful or not. Obviously, since the password value is actually `"password"`, the only 1 under the successful message is under this value.
Let us now do the same attack for all the other usernames. First, we need to fetch the actual names and turn them into a list. One of the way we can do so is input their IDs in the sql injection prompt and query their names: Now, simply replace in the header the username with each of the ones in the corresponding tale column.

ID	Name
1	admin
2	gordonb
3	1337
4	pablo
5	smithy

Table 1: User IDs and Names

After replaying the attack, we note the following results:

gordonb's password is **abc123**.(entry 13)
Since 13337's password was taking a lot of time to be found, we decided to put a threshold of 50 entries and shuffle the list on Microsoft excel till we succeed. We knew that it was impossible to know if this would save us time (since it depends on how far the password is), but we decided to do so either way:

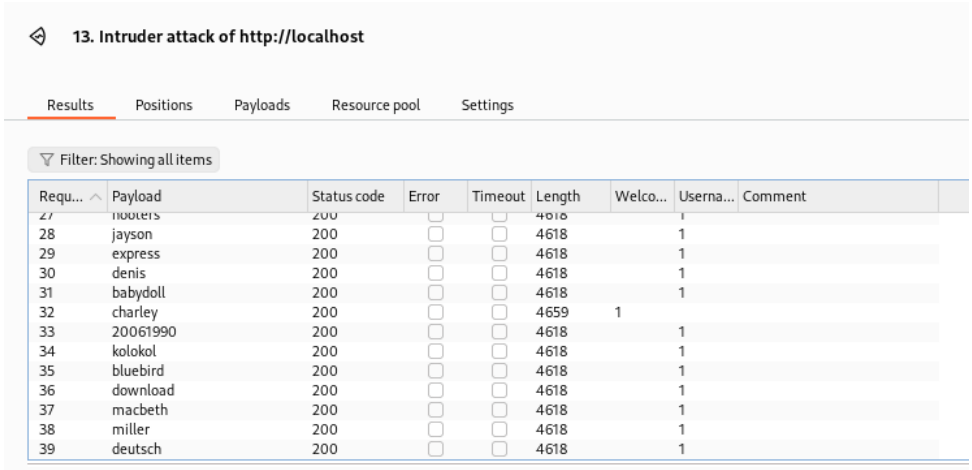
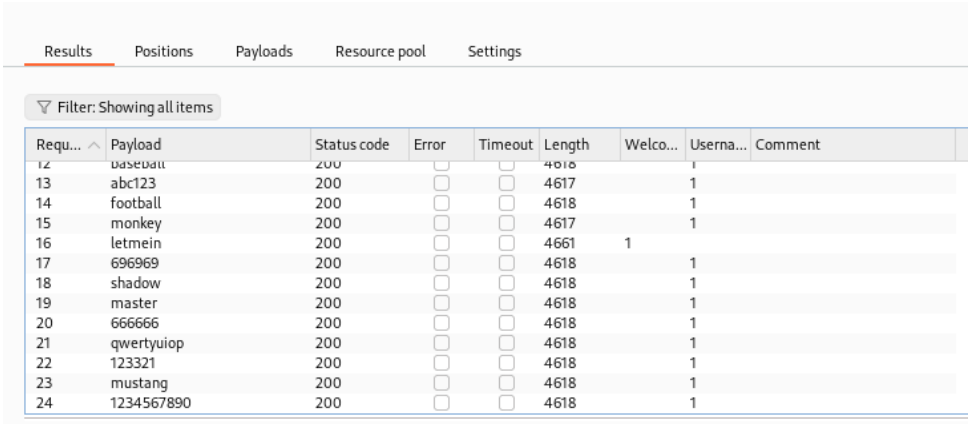


Figure 30: 1337's password is **charley**

Luckily, we got the password around our fourth try. Returning to the original list, we proceed normally and

get the following results:



Requ...	Payload	Status code	Error	Timeout	Length	Welco...	Userna...	Comment
12	baseball	200			4618		1	
13	abc123	200			4617		1	
14	football	200			4618		1	
15	monkey	200			4617		1	
16	letmein	200			4661	1		
17	696969	200			4618		1	
18	shadow	200			4618		1	
19	master	200			4618		1	
20	666666	200			4618		1	
21	qwertyuiop	200			4618		1	
22	123321	200			4618		1	
23	mustang	200			4618		1	
24	1234567890	200			4618		1	

Figure 31: pablo's password is **letmein**

Requ...	Payload	Status code	Error	Timeout	Length	Welco...	Userna...	Comment
0		200			4663	1		
1	123456	200			4618		1	
2	password	200			4663	1		
3	12345678	200			4618		1	
4	qwerty	200			4617		1	
5	123456789	200			4618		1	
6	12345	200			4617		1	
7	1234	200			4618		1	
8	111111	200			4617		1	
9	1234567	200			4618		1	
10	dragon	200			4617		1	
11	123123	200			4618		1	
12	baseball	200			4617		1	

Figure 32: smithy's password is **password**

4.1.2 User risk

Since the attacker gained access to the password, they can impersonate the user and have gained access to all the data linked to this account and perhaps to other accounts, given that users tend to recycle passwords.

4.1.3 Potential security prevention

Limit the amount of tries one can use to login, use better authentication by implementing 2FA, force users to make stronger passwords and force them to change it once in a while.

4.2 Medium level (Brute force)

4.2.1 Attack methodology

Using the same technique, we got the same results. The only difference was that the requests were slower to be processed. Each request took around three seconds longer to be performed, meaning that something like 1337's password would have taken an even more significant amount of time using the list prior to shuffling it.

4.2.2 User risk

The same risks are involved since the same attack is being done.

4.2.3 Potential security prevention

Since the issue remains the same, we can use the same security prevention.

4.3 High level (Brute force)

4.3.1 Attack methodology

Analysing the source code this time around, we notice a particular additional line being added as extra security measure. The `generateSessionToken()` function adds an extra security layer that stops us from using the

same technique to brute force our way. When we mark only the password as a payload, it becomes a random variable, however, the second payload being the token, remains a constant even though the function generates a fresh one every time. We then need to mark the token as a second payload, and mark the attack as pitchfork, since we need to set two distinct payloads:

Figure 33: Form the postion tab

The settings for the first payload will remain the same. As for the second one, we need to do the following:

- Add the initial token in the recursive grep
- Set the range to extract what we want from `value='` to `[/>\r\n]`
- Create a custom resource pool with max concurrent requests = one. The reason for this is because each request will generate a new token, thus we cannot send them in bursts where each ten let's say, will be mapped to one token.

Following those steps, we receive the same results (obviously).

4.3.2 User risk

The same risks are involved since the same attack is being done.

4.3.3 Potential security prevention

We can see in this case that the token has acted in some way as another layer of authentication, and we have still managed to get in. That does not mean that the measure should not be implemented, and even more true, that the other measures should be discarded.

4.4 Low level (File inclusion)

Some applications enable users to input something directly within the scope of their file stream. This allows for potentially malicious file execution.
When trying to proceed with the first file inclusion attack, we ran into an error because `PHP function allow url include` was disabled. To fox this problem, we did the following:

- cd into the directory where the file responsible for allowing this functionality exists.
- Enter the `vim php.init` command.
- Toggle the feature from off to on.

4.4.1 Attack methodology

Looking at the source code, the user is getting the page without implementing any sort of security:

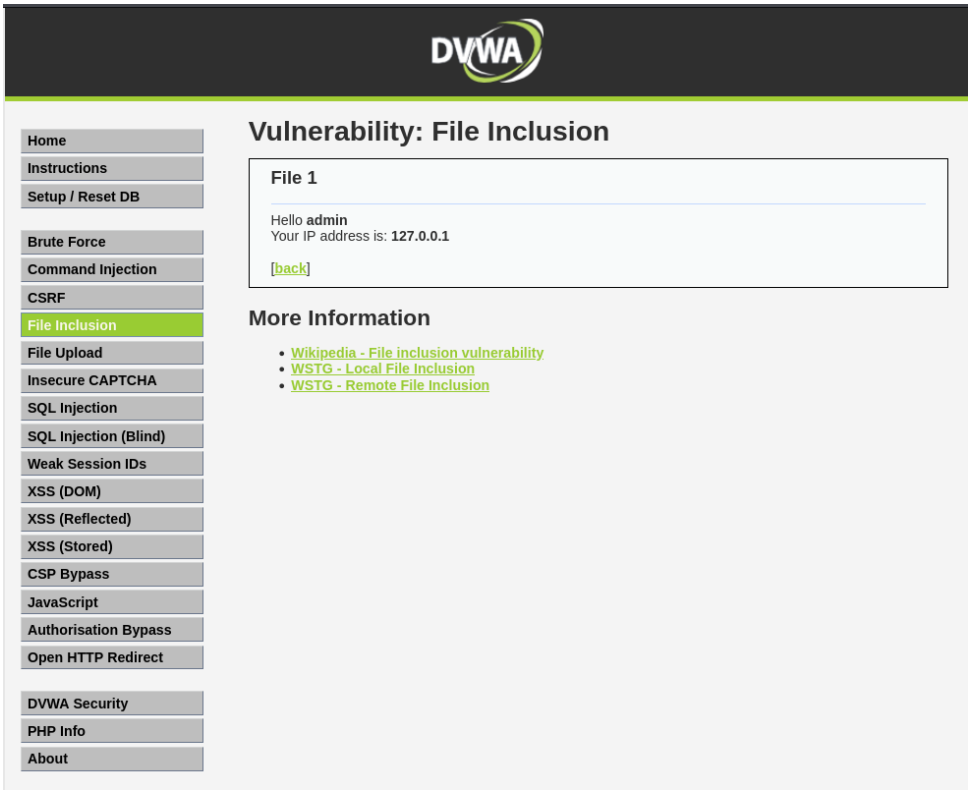


Figure 34: File 1's content.

Thus, if we change the url to something like `/etc/passwd` instead of the original one, we can leak the following information:

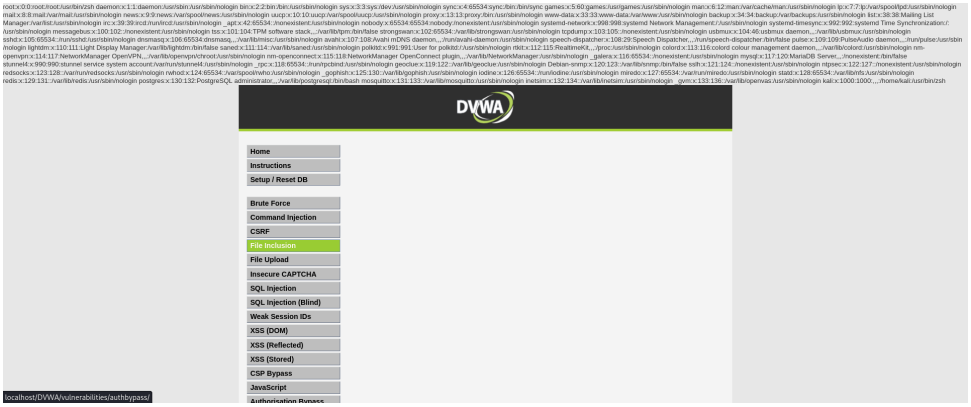


Figure 35: Leaked information

4.4.2 User risk

Many risks are presented, the hacker can execute code on the web server, leak information, DOS the system, etc...

4.4.3 Potential security prevention

Security prevention includes sanitation of input, whitelisting characters, and using relative paths instead of absolute paths.

4.5 Medium level (File inclusion)

4.5.1 Attack methodology

The user here is skipping special characters, not the one we are using however, meaning that if we input the same string in the URL, we will get the same result.

4.5.2 User risk

The same risks are involved since the same attack is being done.

4.5.3 Potential security prevention

Make sure to properly sanitize the string using well trusted built-in libraries instead of manually doing the job.

4.6 High level (File inclusion)

4.6.1 Attack methodology

If we look at the source code now, the user is checking that the file is called `include.php`. Meaning that we cannot just put a random URL and hope that it would work (in theory).
In practice however, we can notice that there is a problem with the logic being used. The user is checking if a hacker injects a path from the file **and** is different from `include.php`, meaning that if we satisify one of the conditions, we will bypass the code block that protects us against this attack.

And so, using the following string: `file:///etc/passwd`, we get the following:

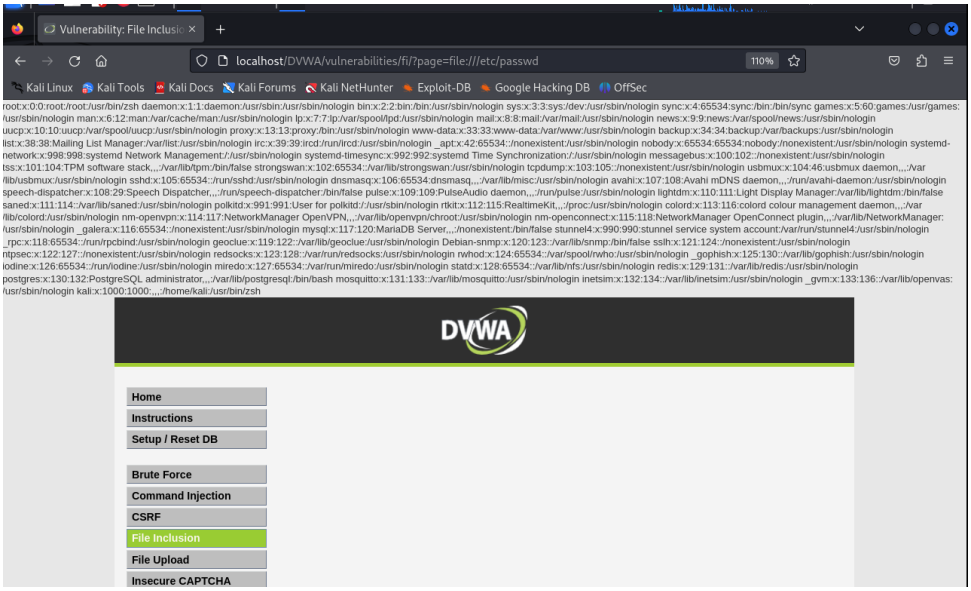


Figure 36: File inclusion high security output

4.6.2 User risk

The same risks are involved since the same attack is being done, but using a longer road.

4.6.3 Potential security prevention

Fix the logic in the code and use the policies we have mentioned earlier.

4.7 Low level (File upload)

One of the many potential breaches that can occur in a software is when one implements a file uploading feature. If not secured, one can exploit the software by uploading a malicious script,

4.7.1 Attack methodology

Since the user is not doing security check. We will write a file that sends an alert message and upload it. The following is the content of the file that we will upload:

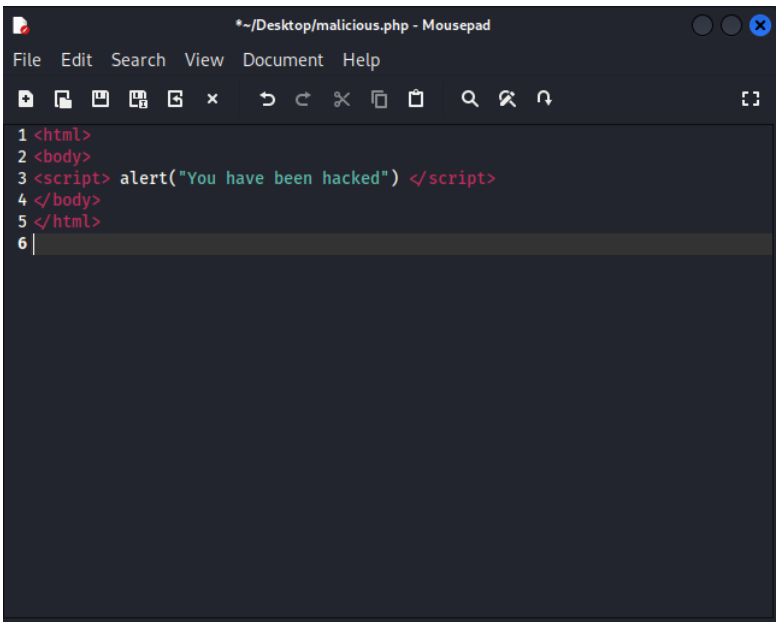


Figure 37: The malicious file

After uploading the file, we input its path and this is what happens:

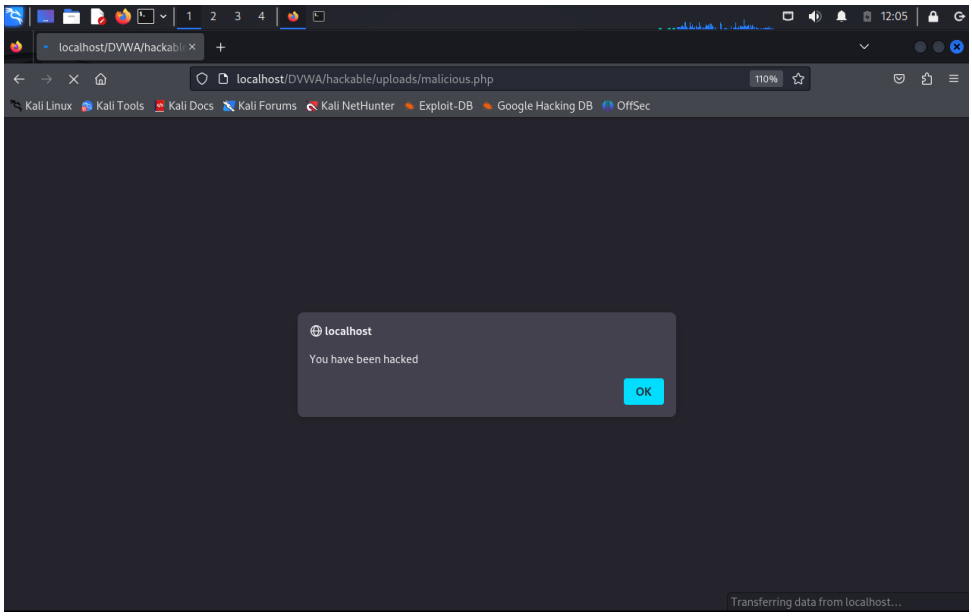


Figure 38: File upload low output

4.7.2 User risk

Using a malicious script, not only can a hacker alter the content of the website (as we have just seen), but they can also vandalize the website's content, steal data, turn the host into a zombie to launch other attacks, etc...

4.7.3 Potential security prevention

Implement access control measures, server sanitation and validation, keep all software up to date.

4.8 Medium level (File upload)

4.8.1 Attack methodology

In this level, the user checks that the file being apploaded can only be a png or jpeg file. We thus need to trick it into thinking we are uploading a picture. To do this , we first change the name of the file to `malicious.php.png`, and upload it:

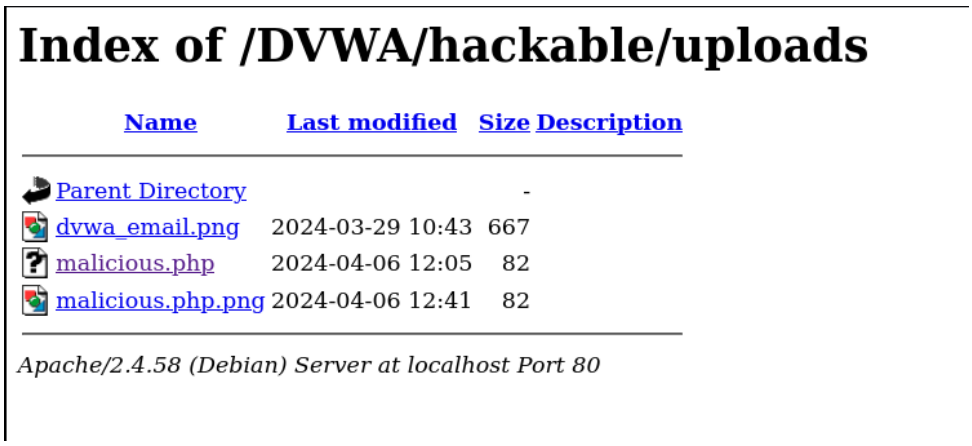


Figure 39: File hierarchy

We now need to find a way to rename the file using the mv command. We can use the medium difficulty command line injection field in order to do it by piping the mv command this way:

```
127.0.0.1 | mv ../../hackable/uploads/malicious.php.png ../../hackable/uploads/malicious2.php
```

We can see now that the file has been renamed back to its original way:

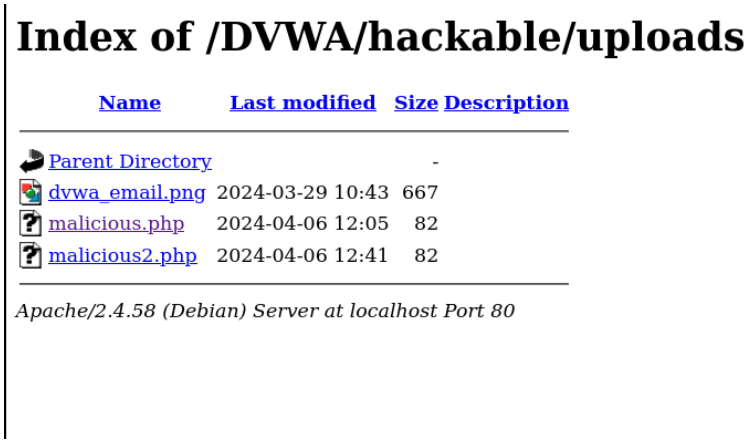


Figure 40: File hierarchy two

Injecting now the path:

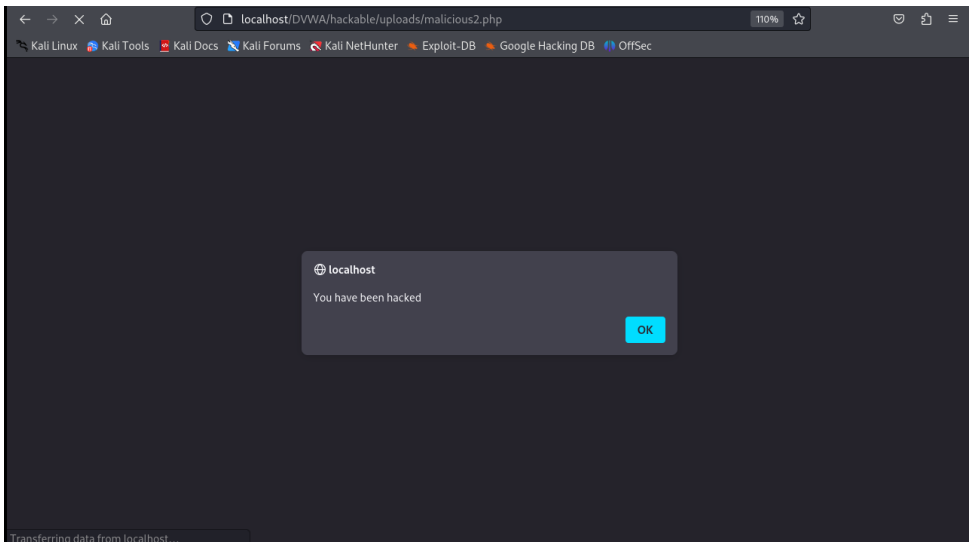


Figure 41: File upload medium result

4.8.2 User risk

The same risks are involved since the same attack is being done, but using a longer road.

4.8.3 Potential security prevention

Implement the same security measures we mentioned used earlier.

4.9 High level (File upload)

4.9.1 Attack methodology

At this specific level, the user is adding an extra security layer by checking for double extensions as well.
We did a bit of research and realized that people were using some proxy to have more control over the file being uploaded. We did not understand why we needed to go through all this process, however. We decided to incrementally try by first changing the file to have a simple .png extension (without the nested trick). It did not work, we received an error that the prompt wanted an actual png.

We thus performed the three following simple steps:

1. Rename the file back to `malicious.php`
2. Edit the code by just adding GIF before the first html tag.
3. Change the extension back to `malicious.png`
4. Upload the file again...It worked.

All we needed now was to repeat what we did last time, but with a space after the piping symbol (remember high security injection). Had this security level taken care of the space, we would have needed to drop it to the middle level to be able to inject our command.

4.9.2 User risk

The same risks are involved since the same attack is being done, but using a longer road.

4.9.3 Potential security prevention

Use the same policies mentioned, but with an emphasis on the sever side sanitation.

4.10 Low level (CSP bypass)

From the definition of provided by DVWA itself, content bypass policy (CSP) "is used to define where scripts and other resources can be loaded or executed from".

Our goal will be to bypass the security policy and execute some JavaScript code.

4.10.1 Attack methodology

We thought of using the file upload section to quickly upload our files, unfortunately however, it uses a different directory than the one we want. We thus used some commands to gain access to the directory and write a small script:

Figure 42: Sequence of commands

We can now include the file we made on the DVWA domain, since it is one of the accepted ones:

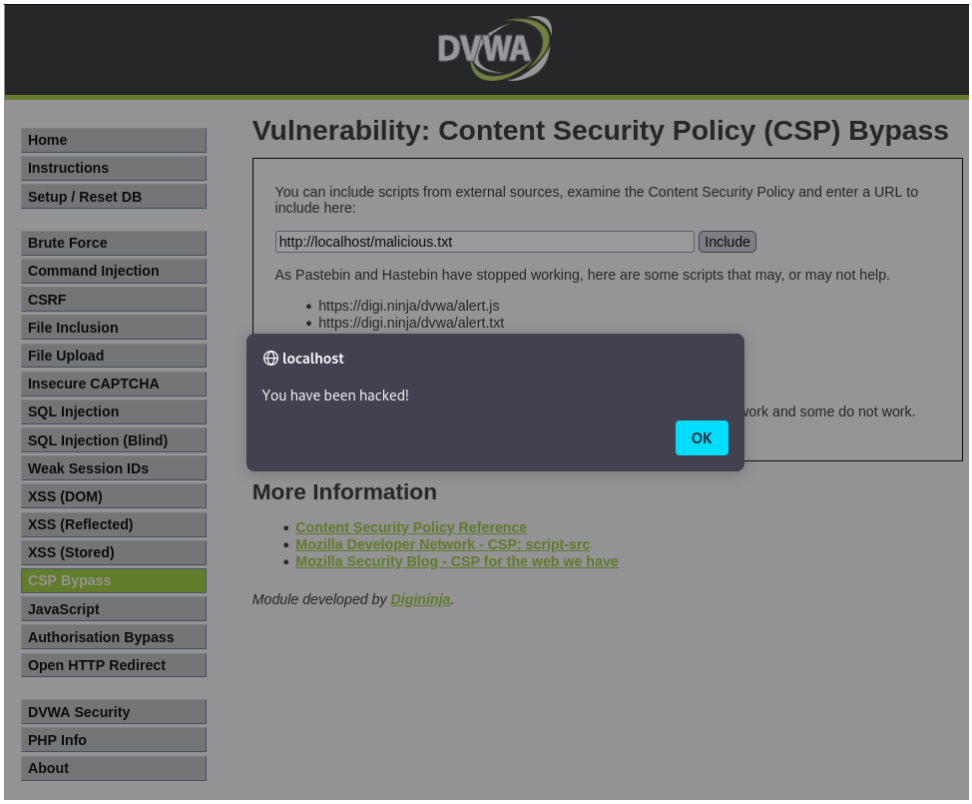


Figure 43: CSP low output

4.11 Medium level (CSP bypass)

In this level, if we look at the source code, the user has added a header `nonce="TmV2ZXIgZ29pbmcgdG8gZ2l2ZSB5b3UgdXA="`. The problem is that the value inside is completely static, meaning that inputting the same header in a script

will work fine:

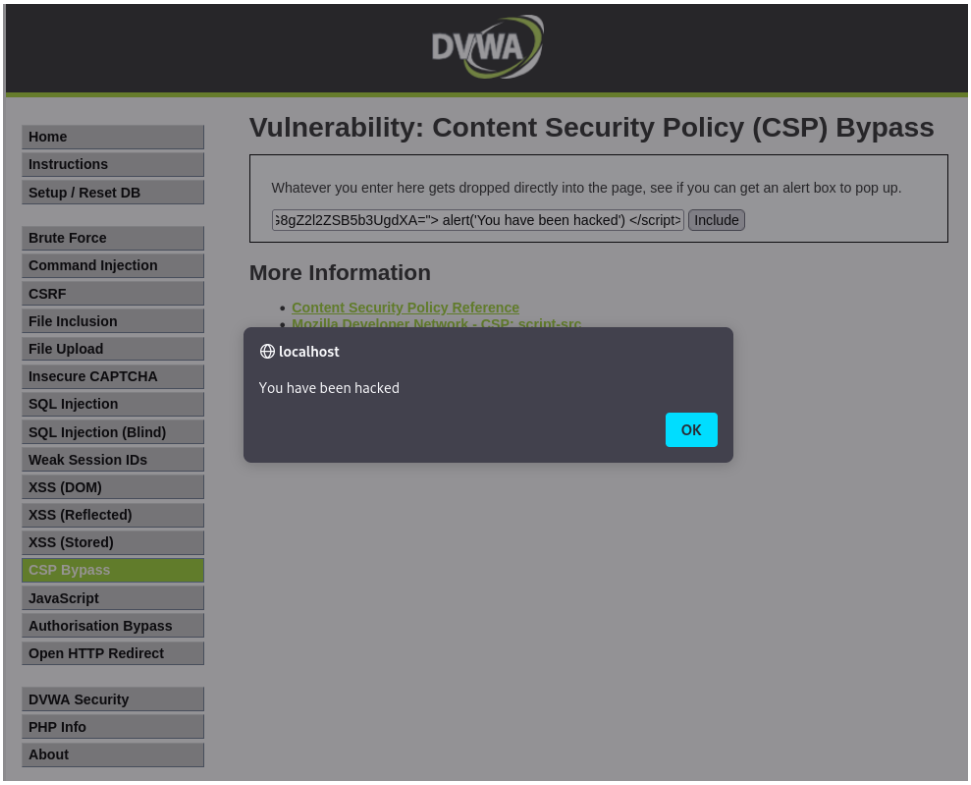


Figure 44: CSP medium output

4.12 High level (CSP bypass)

4.12.1 Attack methodology

In this section, the user hints that they are calling a specific endpoint to request for some code which computes the sum (presumably).

If we intercept the packet using burp-suite, we get the following:

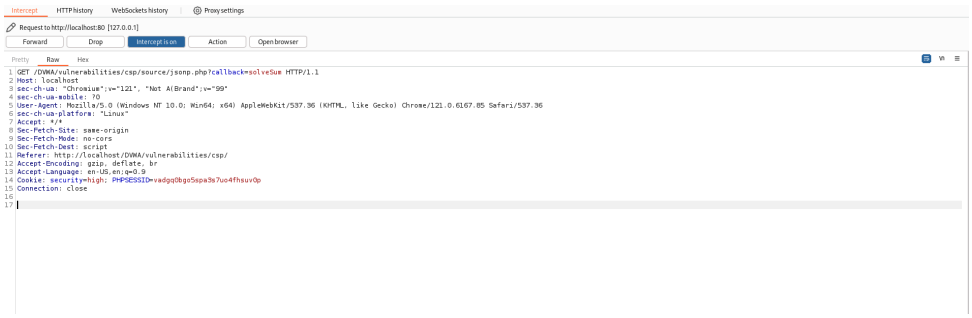


Figure 45: Intercepted message

We notice that there is callback function which is computing the sum, as suspected. We can replace it with a pop-up message to get the results we want and forward the packet.

After doing so, the attack yields a success:

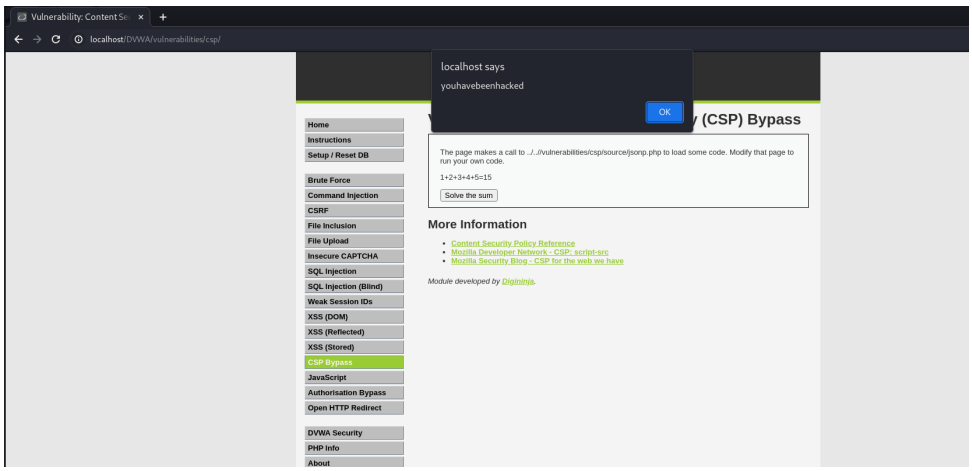


Figure 46: CSP high output

4.12.2 General user risks

As we can see, CSP attacks are notorious for giving an open window for hacker to inject any malicious script. It is not the policy itself that is vulnerable, but the faulty user implementation (like the static nonce in the medium level), or completely not skipping a policy that could hold fatal for the user.

4.12.3 General potential security prevention

Use randomly and dynamically generated nonce. Implement CSPs that cover vulnerabilities across the board. Refrain from using unsafe JavaScript routines, etc..

5 References

- https://www.kali.org/get-kali/kali-platforms
- https://www.vmware.com/products/workstation-player/workstation-player-evaluation.html
- https://raw.githubusercontent.com/danielmiessler/SecLists/master/Passwords/Common-Credentials/10-million-password-list-top-10000.txt
- https://portswigger.net/