IDATT2503 Exercise 01

The assignment:

> Exercise 1
>
> 1. WebGoat
> 2. Start WebGoat using the Docker method on https://github.com/WebGoat/WebGoat
> 3. Open Burpsuite or another interception proxy of your choice (optional but recommended)
> 4. Complete the following exercises:
>   1. A1 Broken Access Control
>   - Hijack a session
>   - Insecure Direct Object References (all parts)
>   2. A3 Injection
>   - SQL injection (intro) (all parts)
>   - SQL injection (advanced) (all parts)
>   - Path traversal (all parts)
> 5. Hacker101
>
> - Log into ctf.hacker101.com
> - Solve all parts of the challenge “Micro-CMS v1”
>
> Tips: Collaborate together with others on these problems. Read about the types of vulnerabilities and attempt the problems before looking for hints.
>
> Deliverable (PDF): Create a short writeup (2-3 sentences including the exploit used) of each problem with a screenshot if necessary.

A1 Broken Access Control

Hijack a session

By sending multiple requests to the page, one notices that the `hijack_cookie` is generated and issued with sequential numbers in the first part of the string with each request sent to the site. The latter part is also increasing, but not as predictable the first. The way it increases, as well as the 'hint' in the lessons suggests this is a Unix time stamp.

```
hijack_cookie=240960387731437039 5-1725366617803
hijack_cookie=240960387731437039 6-1725366617811

hijack_cookie=240960387731437039 8-1725366617819
```

Noticing the missing item in the sequence `(...)97`, suggests this is "taken" by someone else. Brute forcing the time stamp would likely yield a cookie that belongs to that session, which belongs to someone else. If we can generate this cookie ID, we may gain access to their session. We can do this by sending a request as a simple bash script that iterates through the available options. A good place to start is to narrow the interval down based on the time stamps collected by the initial probing.

Exploit: Brute force the session cookie `[ID]-[timestamp]` with a simple bash script at website.

Insecure Direct Object References (all parts)

Using Burp to intercept requests. Identifies the two missing fields `userid` and `role`.

Can request our own profile using a direct object reference in the URL. Can also request out profile page by pointing to out specific user ID.

By this logic, one could potentially find the profile of another user by inputing a different ID in the URL.

Using the Intruder functionality in Burp, we can then issue payloads to try iterating over user IDs, using our own ID as a reference point. Keep sending the payloads until we get back a response with a different users profile details.

Exploit: Directly accessing another users profile by brute forcing their profile ID in request-response on the user profile objects.

A3 Injection

SQL injection (intro) (all parts)

Standard QSL injection. Manipulating SQL entries through input fields.

In some field we can input basic commands, while in others we trick the input by using logical "traps", e.g. where both logical statements evaluate to `true` which results in the statement being evaluated as a SQL query. This can be done by closing the expected input string for name fields, and then appending the logical check. Lesson 10, you close the employee_name field string with a single `'` quote, make a logical claim that is true `or 1=1` and end the input by commenting out `--` the rest of the line: `' or 1=1 --`

In lesson 11, we do the same as in 10, only where we then append a SQL query for updating the record for the employee in question.

SQL injection (advanced) (all parts)

Solution 1: Using a UNION statement: `' UNION SELECT userid, user_name, password, null, null, cookie, null FROM user_system_data; --`

```
You have succeeded:

USERID, FIRST_NAME, LAST_NAME, CC_NUMBER, CC_TYPE, COOKIE, LOGIN_COUNT,
101, jsnow, passwd1, null, null, , null,
102, jdoe, passwd2, null, null, , null,
103, jplane, passwd3, null, null, , null,
104, jeff, jeff, null, null, , null,
105, dave, passW0rD, null, null, , null,
Well done! Can you also figure out a solution, by appending a new SQL Statement?
Your query was: SELECT * FROM user_data WHERE last_name = '' UNION SELECT userid,
user_name, password, null, null, cookie, null FROM user_system_data; --'
```

Solution 2: Appending a new SQL statement: `' OR 1=1; SELECT * FROM user_system_data; --`

```
You have succeeded:

USERID, USER_NAME, PASSWORD, COOKIE,
101, jsnow, passwd1, ,
102, jdoe, passwd2, ,
103, jplane, passwd3, ,
104, jeff, jeff, ,
105, dave, passW0rD, ,
Well done! Can you also figure out a solution, by using a UNION?
Your query was: SELECT * FROM user_data WHERE last_name = '' OR 1=1; SELECT * FROM
user_system_data; --'
```

The Login Form

This one I had to look up many hints for. One

First checked if input fields were vulnerable to injection by using a simple `' OR 1=1; --` logical check. As per hint, the register form was vulnerable. Checking for user `tom` with a password longer than 0: `tom' and length(password)>0; --`

Then test for which character the password starts with

`tom' AND substring(password, 1, 1) = 'a' --`. Getting the message that the user {0} exists indicates wrong password. We then use Burp, or a Python script, to brute force the password by sending a payload that find the length of the password, and then run through every permutation of the password. For each "already exist" we know that the combination is one step closer to correct.

Path traversal (all parts)

By intercepting the upload path, we can work out how to navigate the directory structure. We can then send request to the server with commands used for navigating the directory structure `../`

Exploit: Passing navigation arguments and figuring out the pattern which will bypass the sanitizer checks.

Hacker101 - CFT Micro-CMS v1