CSE 543- Computer Security

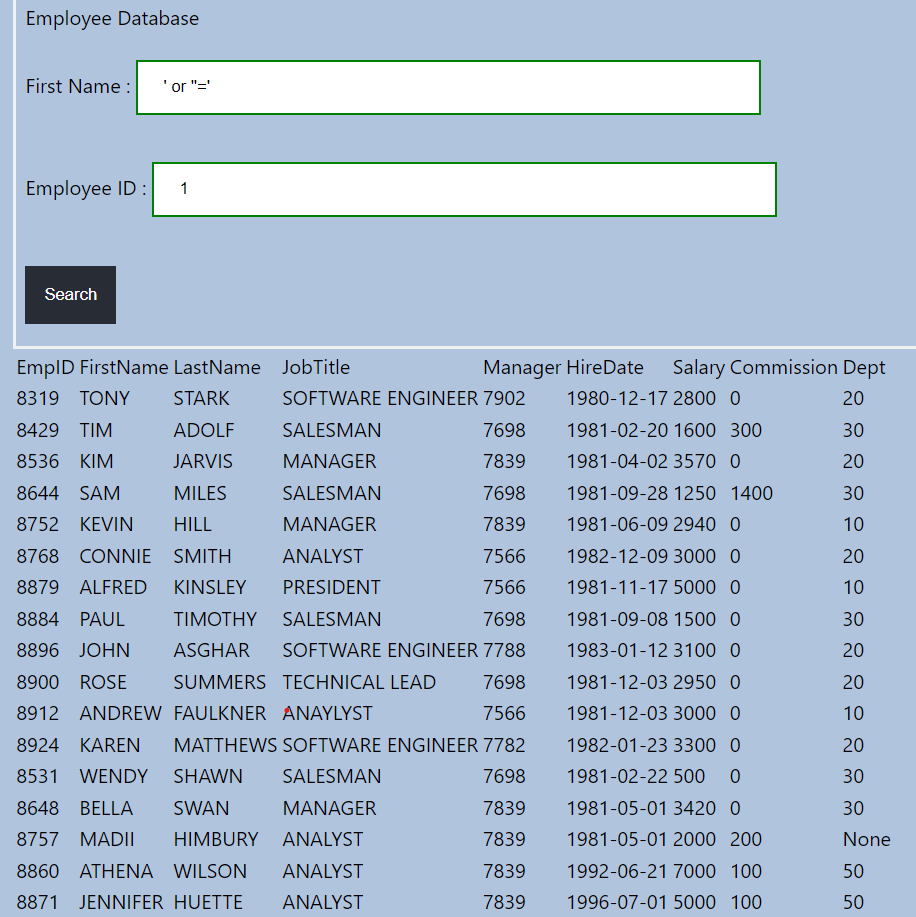


*Project 4: Web Security*

Date: December 7, 2022

Student Name: Md Rafi Ur Rashid

**Task 1.1**



Tried to generate a sql of the following form to conduct the attack:

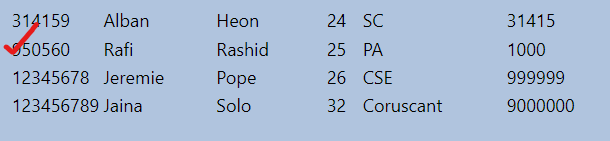
*Select \* from EMPLOYEES where EmpID=1 and FirstName=‘’ or ‘’=‘’*

The *where* condition is true for each record.

**Task 1.2**

Graphical user interface, text, application

Description automatically generated



At first tried to find out how the original query looks like with some trial and error.

Then tried to append a second query for insertion at the end of the existing one. Which finally looks something like-

Select \* from BANK where FirstName= ‘Rafi’ and Age=1 and AccId=1; INSERT INTO BANK VALUES (950560, 'Rafi', 'Rashid', 25, 'PA', 1000)

**Task 1.3**

Graphical user interface, table

Description automatically generated

Tried to create an always true condition and comment out the rest of the query:  
*Select \* from EMPLOYEES where EmpID=EmpID # and FirstName=‘anything’*

**Task 1.4**

Graphical user interface, application, website

Description automatically generated

First of all I figured out the password column name with some trial and error. Then conducted a series of brute force/ intuitive guesses in a character-by-character manner. The crafted query somewhat looks like the following:

*Select \* from LOGIN where username=’* *secret\_user' and left( password, 10) = 'acrazylong' #*

Graphical user interface, text, application, email

Description automatically generated

**Task 2.1**

Graphical user interface, website

Description automatically generated

Pressed the login button and inspected the request payload in the network tab. Password has been passed in plaintext.

**Task 3.1**

Graphical user interface, text, application

Description automatically generated

I found that the value of the street address input is getting embedded as an HTML element (the hidden <div> in the above figure) when I register. So, I tried to utilize the HTML onerror attribute to trigger a sample script, and it seems to work. So, an attacker can execute malicious script (e.g., send user cookie to another origin) through XSS by exploiting this vulnerability.

**Task 3.2**

Timeline

Description automatically generated

I just reused the idea from 3.1 and execute a script that send a GET request to the specified endpoint along with the user cookies.

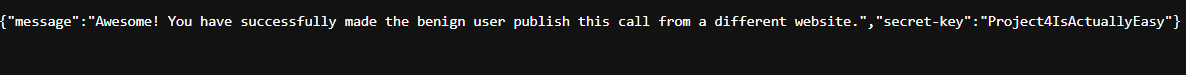
**Graphical user interface

Description automatically generatedTask 4.1**

I took the request URL for registration by inspecting the network tab. Then used this as a form action in the crafted index.html. So the registration is forged from this html not from the original site.

Text

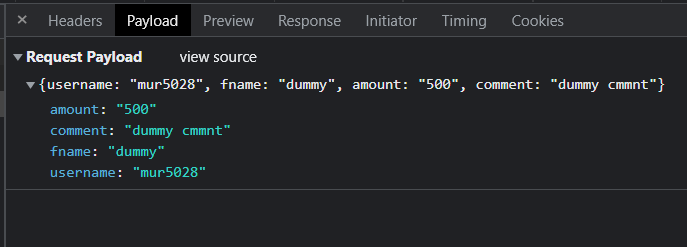
Description automatically generated



Graphical user interface, application, email

Description automatically generated

**Task 4.2.**



Graphical user interface, application

Description automatically generatedText

Description automatically generatedText

Description automatically generatedText

Description automatically generated

Like the previous task, I took the request URL for registration by inspecting the network tab. I also found that unlike the previous task, now the request body is taking json format. So, I observed the content-type in the request header and crafted a form in index.html in a slightly different manner. I tried to make sure that the name and value attributes of the input tag carry the whole json body combined. So, while submitting the form, now it sends a json object in the guise of form-data.