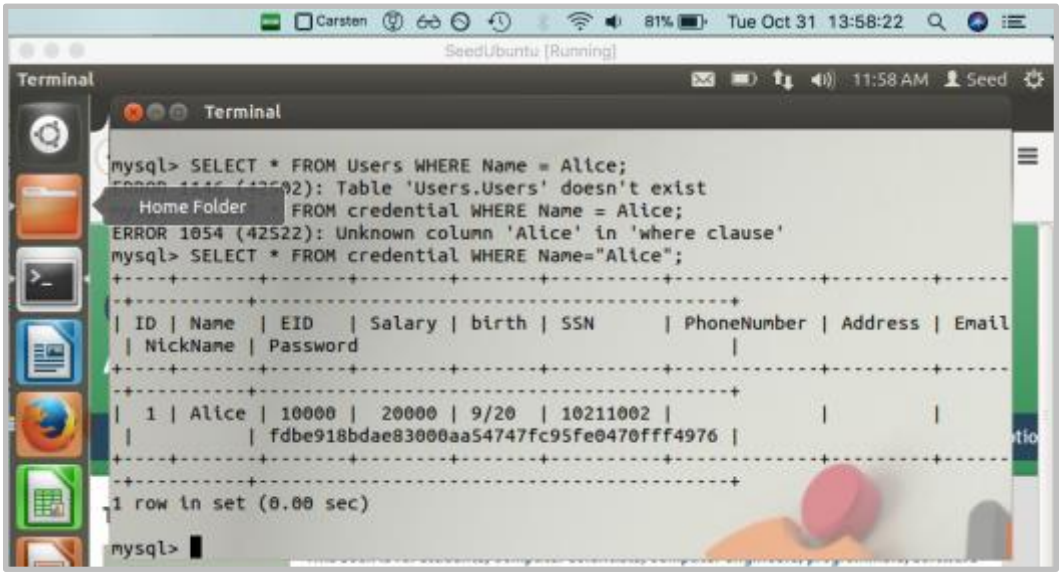
Carsten Hood
UIN: 922009787
CSCE 465-500 Computer & Network Security
Due Friday, November 3, 2017

HW #3: SQL Injection Lab

==========

Task 1. MySQL Console

After setting up the SQL database, we successfully print Alice's profile information by executing the SQL query `FROM credential WHERE Name = "Alice";` as shown in the following screenshot:



==========

Task 2. SQL Injection Attack on SELECT Statement

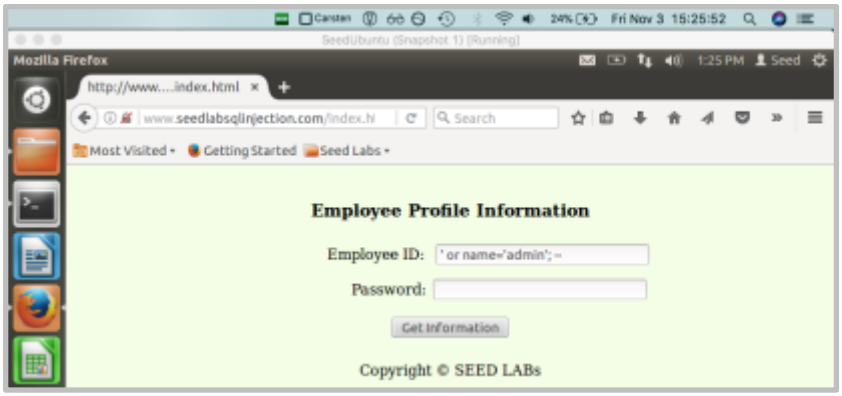
2.1: SQL Injection Attack from webpage.

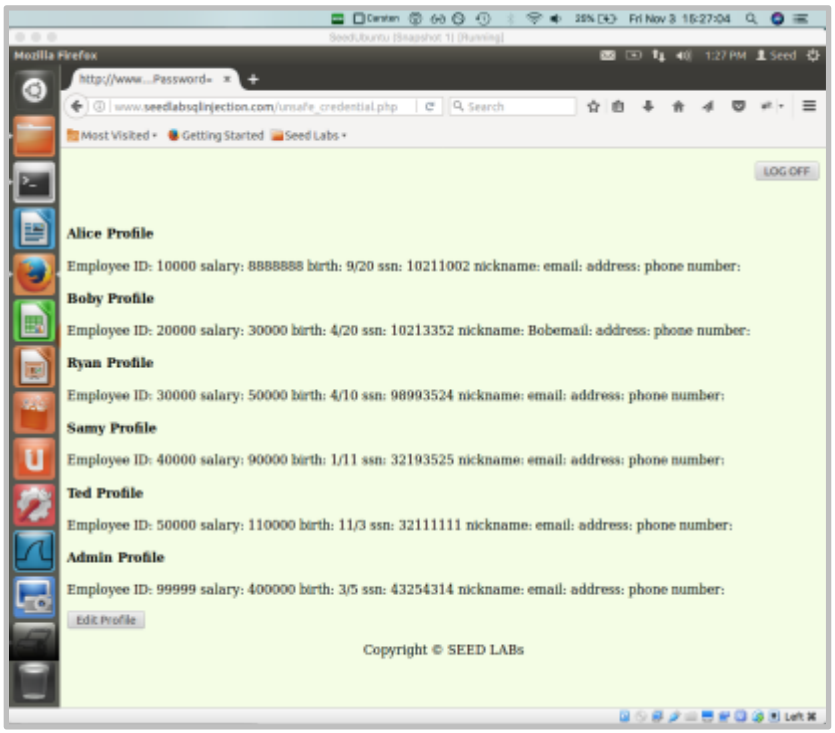
To log in as the admin with knowledge only of admin's name 'admin' we inject the following code via the Employee ID field: `' or name='admin'; --` . Leave the password field blank. Joining the query's name parameter with the `or` keyword means that information with the 'admin' name is returned regardless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

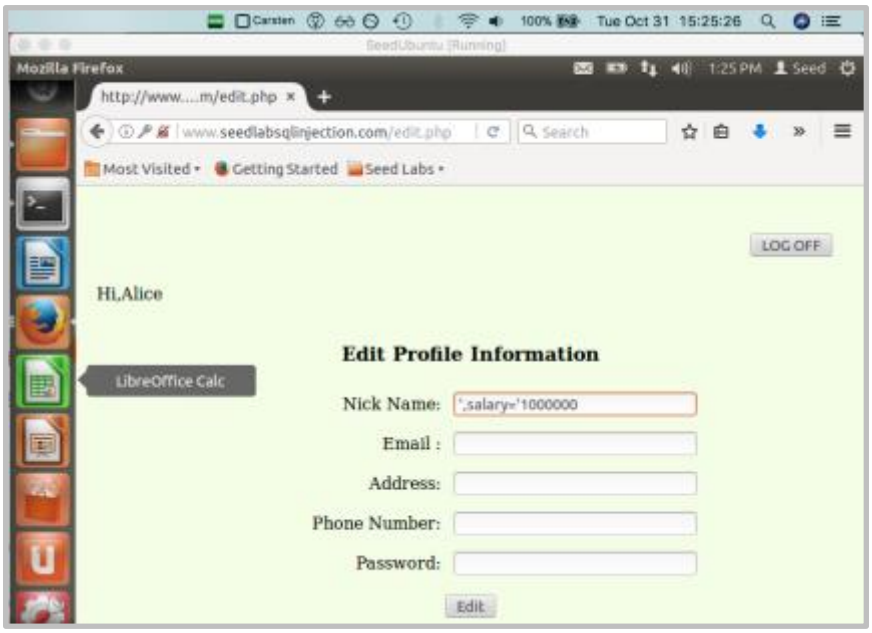
The below screenshot shows successful use of this attack to reveal all database information:

==========

Task 3. SQL Injection Attack on UPDATE Statement

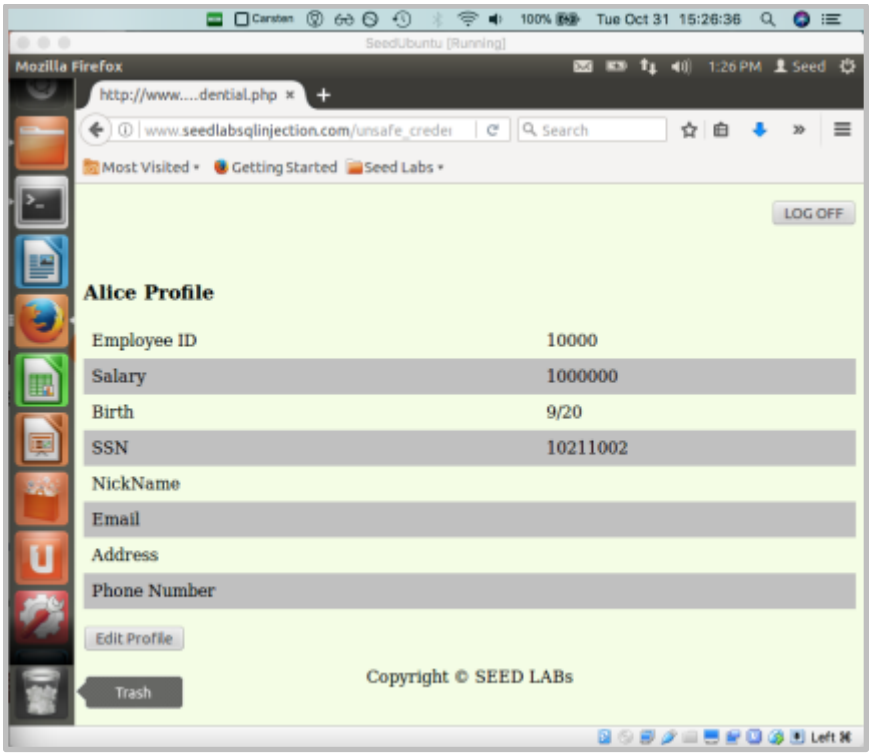
Task 3.1: SQL Injection Attack on UPDATE Statement — modify salary.

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:

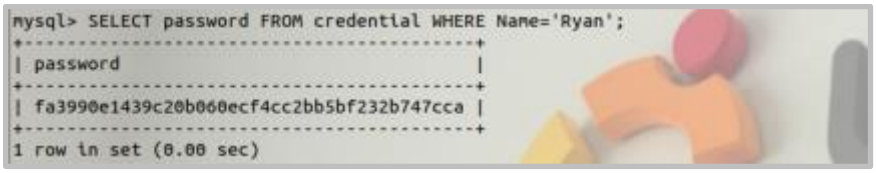
To change Ryan's password while logged in as Alice we go to Alice's "Edit Profile Information" page, enter the desired password (in this case `AliceWasHere`), and enter the text `' WHERE name='Ryan' --` in the Phone Number field as show below:

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:

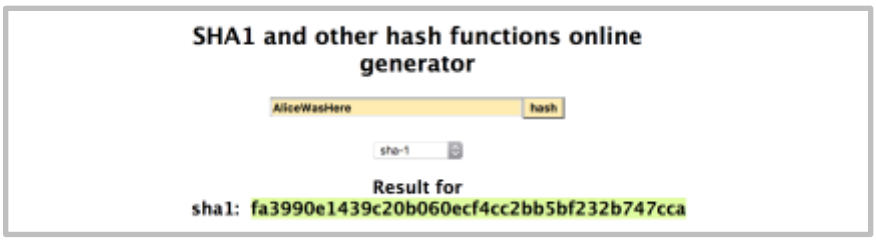
```
If ($input_pwd!='') {
        $input_pwd = sha1(AliceWasHere);
        $sql = "UPDATE credential SET nickname='$input_nickname',
                email='$input_email', address='$input_address',
                Password='$input_pwd', PhoneNumber='' WHERE name='Ryan' -- '
                where ID=$input_id;";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

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:



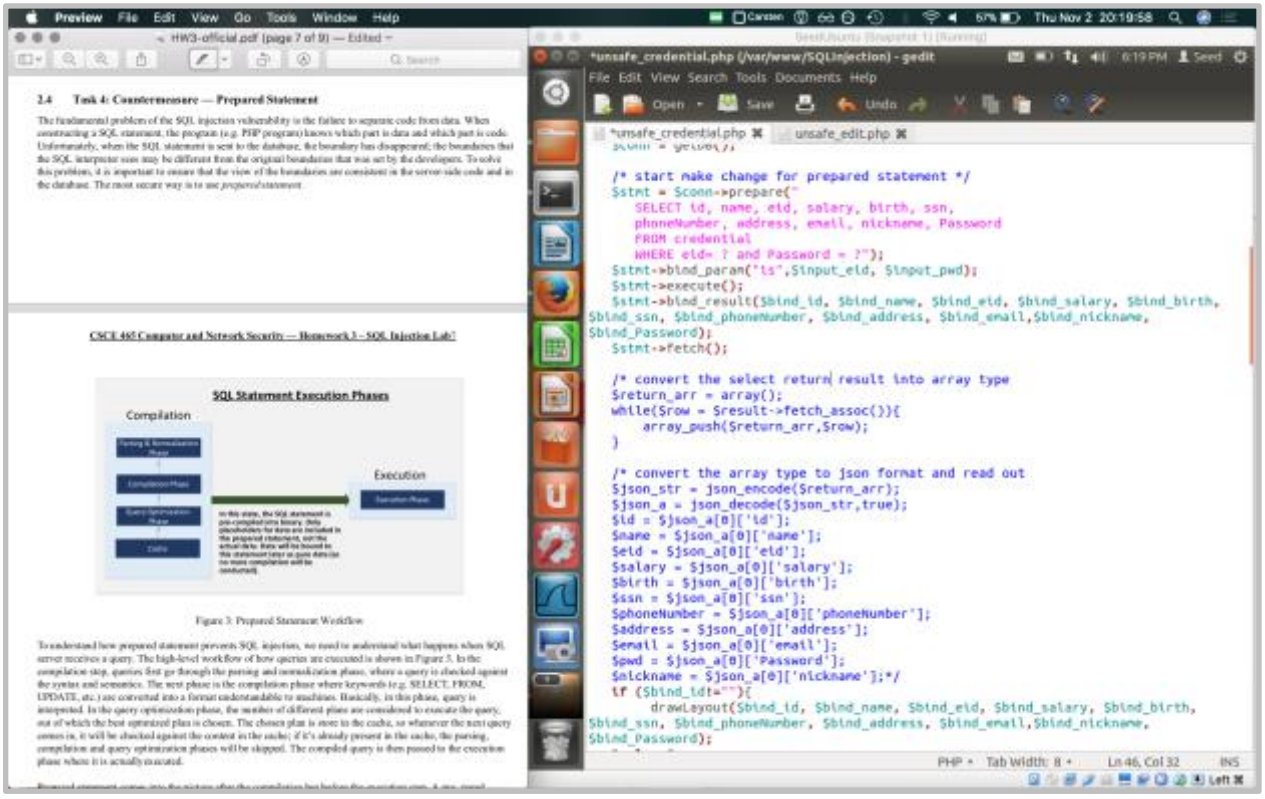
Indeed when we plug our new password (`AliceWasHere`) into an online SHA1 function we see exactly the same hash, confirming that we successfully changed Ryan's password to our own password:



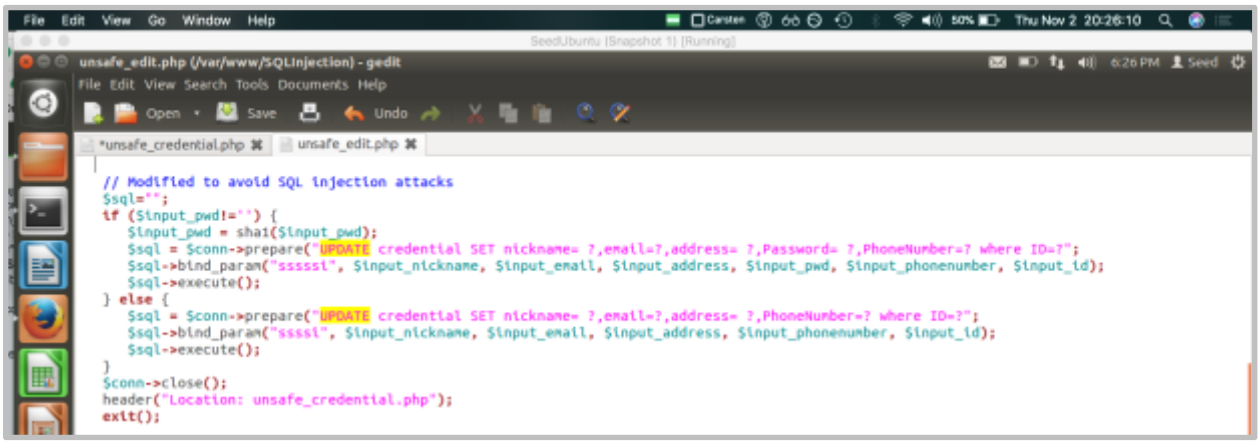
==========

Task 4: Countermeasure – Prepared Statement

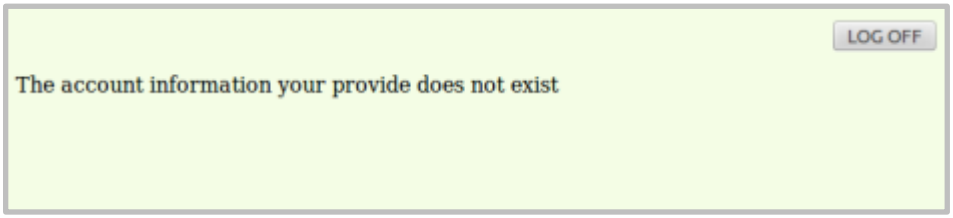
First we update the code in `unsafe_credential.php` using the prepared statement mechanism to make it safe against SQL injection. This includes commenting out the JSON-related code as shown in the code screenshot below:



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



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:



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.