[[1]](#footnote-1)

SQL Injection Attack LabDavid Warren, CS428, dlwarren2@crimson.ua.edu

*Abstract*—This lab is about SQL Injection attacks. SQL injection is a vulnerability between a front facing webpage and its corresponding database. This vulnerability allows malicious users to view information they should not be allowed to see or change data they should not be allowed to change.

# INTRODUCTION and Lab Definition

T

HIS lab is an exploration of the SQL injection attack, how it can be conducted and also countermeasures that can be used against it. It allows the students to walk through how an SQL Injection vulnerability occurs in code, how it is exploited, and finally how it is remedied. This is accomplished first by setting up the VM environment to host a vulnerable webpage, then by conducting SQL injection attacks against the login and edit page. Finally, a prepared statement is used as a countermeasure to prevent the attack [1]. The exploit is achieved from given code in var/www/SQLInjection from the unsafe\_edit.php and unsafe\_credential.php and from the lab instructions.

# Lab Setup

## Lab Environment

The first step in this lab was to create a suitable lab environment to conduct our exploration. To do this, an Ubuntu VM was created using VirtualBox and an Ubuntu image from SEED security labs’ SEEDUbuntu12.04. A setup document was referenced to create the VM from the pre-built VM image in the previous Lab 1 [2][3]. Then, the environment needed to be configured. First, the Apache web server was restarted to ensure it was active.

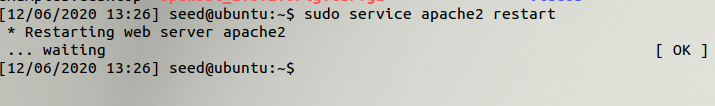


Fig. 1. Restart the Apache Web Server

The DNS and Apache Server were already configured correctly from the VM image. Next, the php magic quotes countermeasure had to be turned off. You can see this done below.

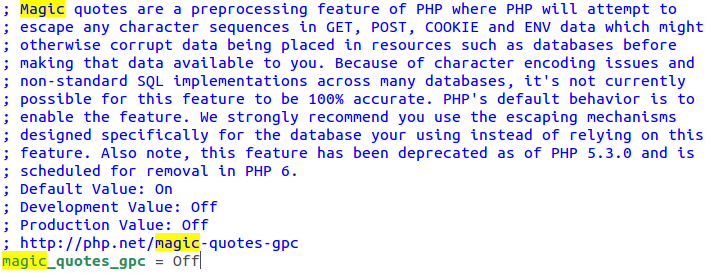


Fig. 2. Turning magic quotes off.

Finally, the patch was run, specifically the bootstrap shell code. The results of this patch can be seen below.

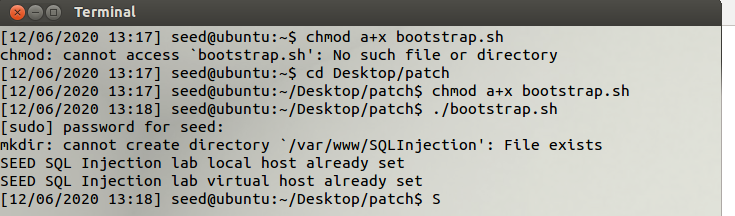


Fig. 3. Patch applied.

Now, we are ready to conduct the lab experiments.

# Task 1

## Summary of task

In this task, our goal is to become familiar with the SQL commands by finding out more about the provided database. Then, we want to print all profile information on a single employee: “Alice.”

## Process of task

To do this, we log in to the database, change the active database to “Users,” and then show the available tables. This can be seen below.

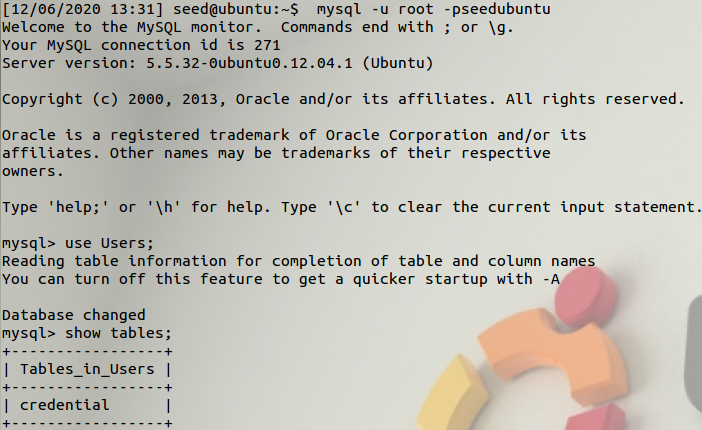


Fig. 4. Log in and show the tables in “Users”

After we found the available tables, we searched the table “credentials” to give us all profile information for “Alice.” That profile information can be seen below.

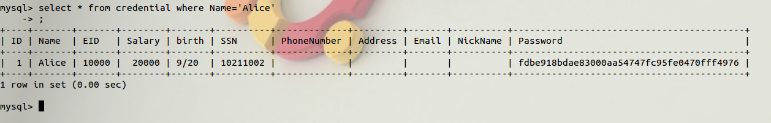


Fig. 5. Find profile information for “Alice.”

## Result of Task

After becoming familiar with MySQL console, “Alice’s” profile information was successfully displayed. We also found out all columns in the database and can make some inferences on certain data types for the table.

# Task 2

## Summary of task

In this task, we will use SQL injection attacks on a select statement. According to the lab document, there is an unsafe .php file running for the login called “unsafe\_credential.php” that can be exploited via a select statement [1]. First, we will exploit this vulnerability from the webpage. Finally, we will exploit this vulnerability from the MySQL command line. Both of these exploits will be attempting to gain access to the administrator account which we know has the name “admin.”

## Process of task

### SQL Injection Attack from webpage

First, we must analyze the SQL statement in the unsafe file. We can see that no prepared statement is being used in the select query. To target this, we will use the EID input field and our knowledge of the user “admin” to conduct this attack. We will close the EID input variable with a quote, then add an “or” statement so we specifically get the admin information (as the EID input will be empty). This statement can be seen below.

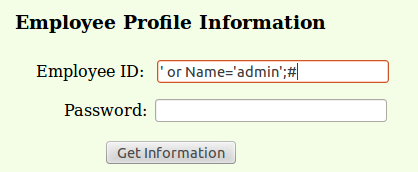


Fig. 6. SQL injection statement

Finally, we comment out the password input so no password check has to be done. This statement succeeded and returned all employee information. Result can be seen below.

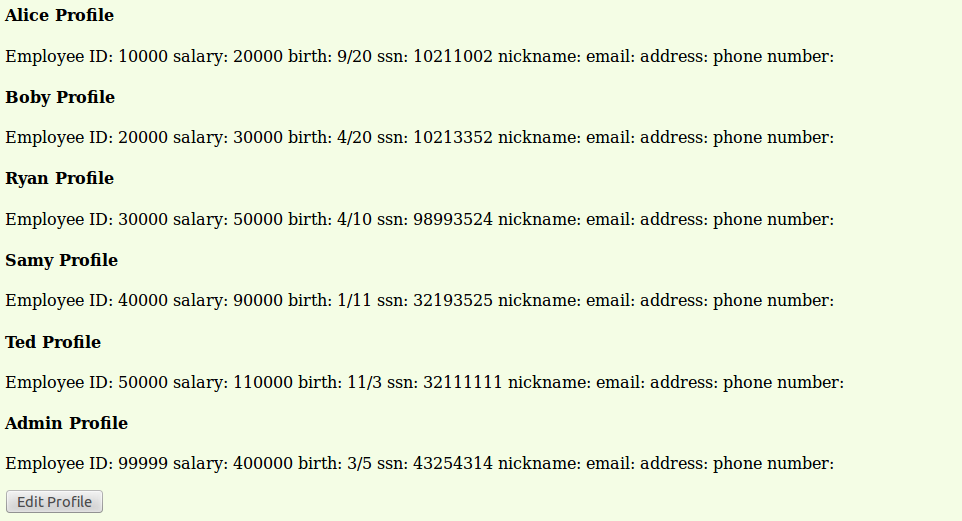


Fig. 7. SQL injection attack results.

### SQL Injection Attack from command line

This attack was very similar. We use the same SQL injection statement except from a curl call. In the EID input, we put our (URL encoded) injection statement from Task 2.1. Special characters were simply replaced with their URL encoded equivalents. You can figure this out by checking the URL parameters after Task 2.1 in conjunction with the information on whitespace and single quotes from the lab documentation [1]. After changing the statement slightly, the SQL injection attack from the command line was a success and the same information was received in HTML form. Statement and results can be seen together, below. URL encoded statement is highlighted in the curl call. The HTML results follow of all employee information.



Fig. 8. Curl call with URL encoded exploit and results.

### This exploit was also a success.

### Append a new SQL statement

In this subtask, we must add a new SQL statement attempting to delete a record after the original exploit. We will chose to delete “Alice” we can see from the Task 2.1 results. To do this, we add a semicolon and then an SQL delete statement for names that are equal to “Alice.” The attempted statement and result can be seen below.

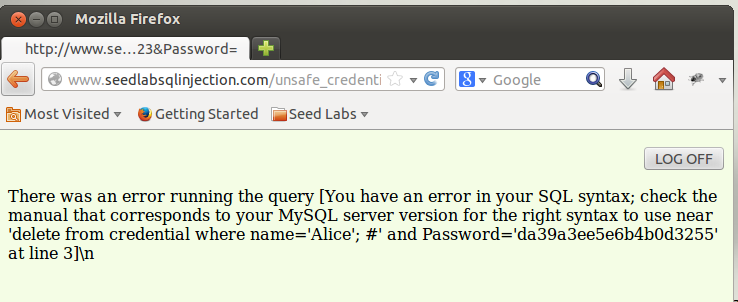


Fig. 9. Second statement fails.

The second statement fails while the first statement has succeeded. In all likelihood, PHP is preventing multiple SQL queries from executing at once and the delete fails.

## Analysis of exploit

The “unsafe\_credentials.php” is extremely vulnerable to SQL injection attacks via the select statement. We were able to masquerade as “admin” to view all employee information. The only place we could not exploit was in adding another SQL statement to delete. Most likely, you would have to specify in the PHP code that you are conducting multiple queries at once for this to be able to work. If that were the case, we could hijack the query again and run as many queries as we like. This would have a devastating effect.

# Task 3

## Summary of task

In this task, we will exploit the update SQL vulnerability from “unsafe\_edit.php” after editing a user’s record. We will attempt two separate updates. First, we will attempt to update a salary. Finally, we will attempt to update another user’s password.

## Process of task

### SQL Injection Attack on UPDATE – modify salary

First, we navigate to the edit profile page. Now, using the provided code from the lab documentation [1] we can identify a vulnerability in the UPDATE statement. There is no prepared statement being used, so we can use SQL injection in the nickname variable. To do this, we will add another single quote to close the nickname parameter, then add the salary parameter and increase it to “30000” from the previous “20000” we saw in Task 2.1. Then we add a semicolon and pound to close the sql statement and comment the rest of the SQL in the php code out. The statement in the webpage can be seen below.

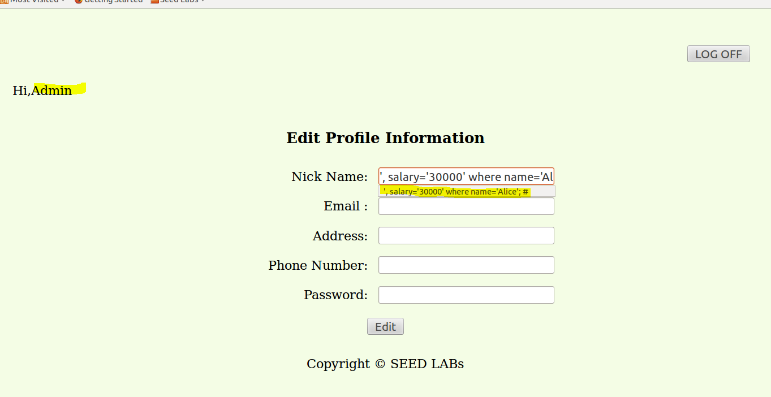


Fig. 9. Full statement highlighted changing salary in nickname.

After submitting this edit, we can see the salary of Alice changed below.

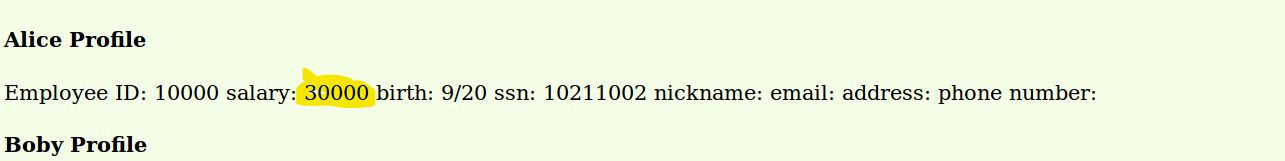


Fig. 10. Salary has changed.

The salary has increased to our desired value of “30000” and this UPDATE SQL injection attack was a success.

### SQL Injection Attack on UPDATE – modify password

Now we will attempt the same exploit to change a user’s password. From the lab documentation, we know that the PHP code is using SHA1 to hash the passwords. So, we hash our desired password with SHA1, and update the password column on our desired record. To do this, we can use openSSL to hash a string. The result of this hash can be seen below.

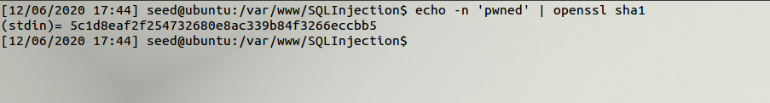


Fig. 11. Hash a desired password with SHA1.

After hashing, we update the password of a desired user (we will use “Ryan”) using the same method from Task 3.1. Attempt can be seen below.

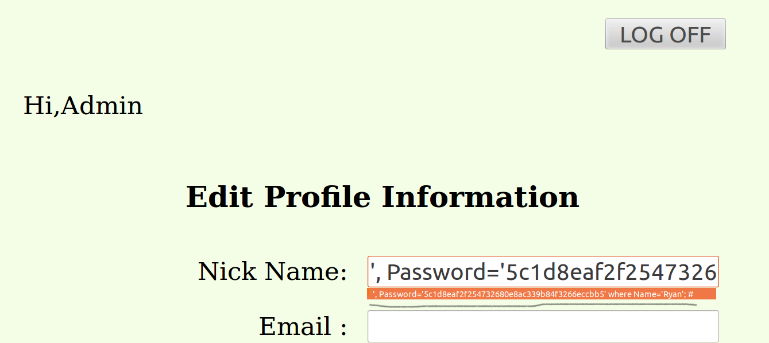


Fig. 12. UPDATE attempt with Ryan’s password.

The update was a success and now we can log into Ryan’s account using the password “pwned” and his EID number we have gathered from Task 2.1. Results of this attempt can be seen below. EID and unhashed password can be seen used in the URL to log in to the account.

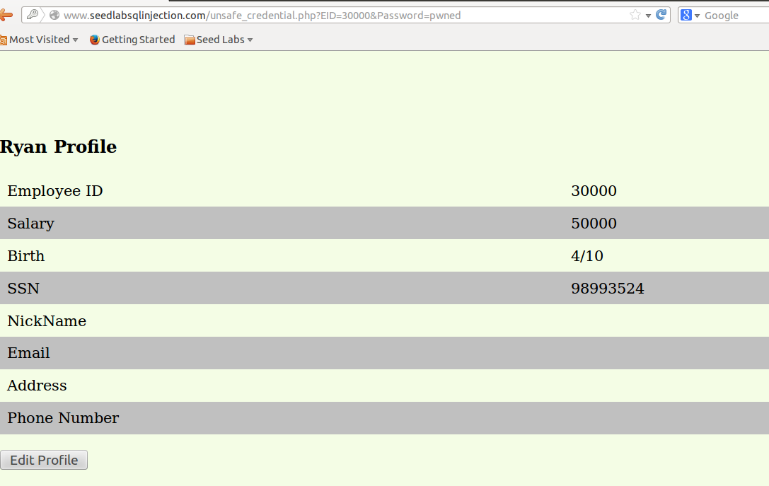


Fig. 13. UPDATE succeeded and we now have access to Ryan’s account with our new password.

This exploit succeeded and we have gained access to Ryan’s account with the new password.

## Analysis of exploit

The UPDATE SQL Injection attack was a success. We were able to change Alice’s salary in Task 3.1 and Ryan’s password in Task 3.2. Isolated UPDATE queries with no prepared statements can be very devastating and can allow malicious users to easily gain access to accounts as well as locking out the real user from their account.

# Task 4

## Summary of task

In this task, we will make use of prepared statements as a countermeasure to SQL injection attacks. I predict this will be very effective at sanitizing input and not allowing any unwanted SQL statements being run in our PHP code.

## Process of task

First, we need to set up this countermeasure in the “unsafe\_ credentials.php” file. We replace all parameters in the SQL string with “?” and then use “$conn->prepare($sql)” to ready the query to be safely parameterized. We bind the input variables in the query with “$stmt->bind\_param()” specifying the type and variable. After executing, we bind the results from the columns to specific variables, and fetch results. After doing this, the unsafe code is now safe. Unwanted SQL queries in the input variables will now be treated as strings and are completely sanitized. The solution in the “unsafe\_credentials.php” file can be seen below.

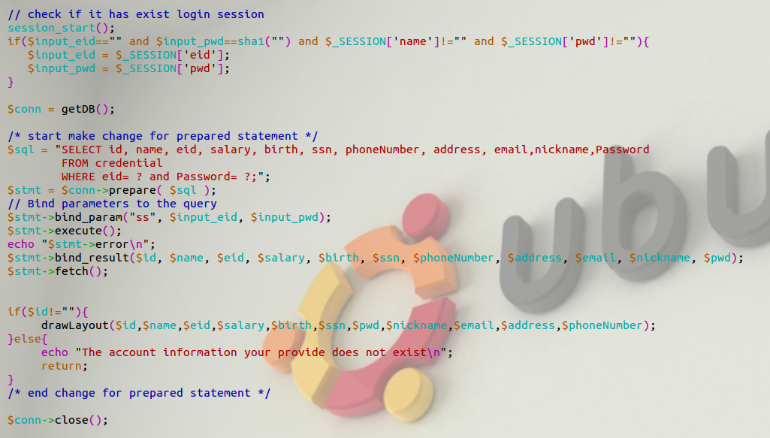


Fig. 14. “unsafe\_credentials.php” fix to add prepared statements.

After adding these countermeasures, we can go to test them using the same process from Task 2.1. Results can be seen below.

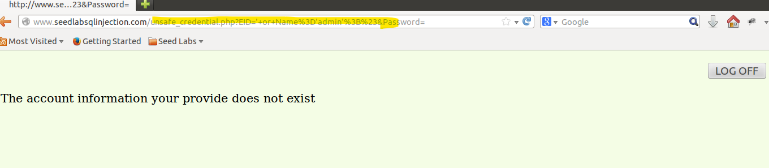


Fig. 15. Prepared statements successfully stop SQL Injection in SELECT query.

This countermeasure successfully stops the previous exploit used in Task 2.1. The same input can be seen in the URL while the page displays that the account information does not exist. Next, we will fix the “unsafe\_edit.php” file in much the same manner. We will exchange the unsafe queries in the edit for prepared statements to prevent SQL code from being executed in the UPDATE query. We simply replace each $sql string with a prepared statement as from before, interchanging the correct input variables and binding the outputs. The fix for the “unsafe\_edit.php” file can be seen below.

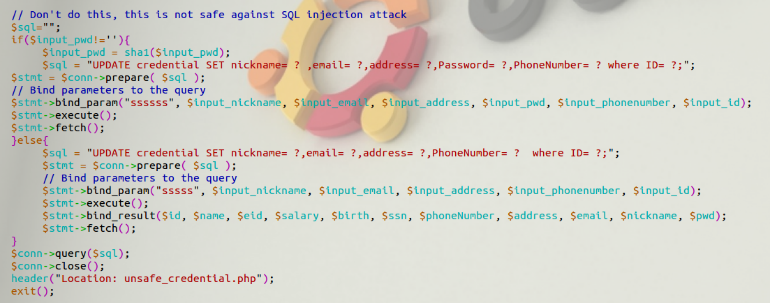


Fig. 16. “unsafe\_edit.php” fix to add prepared statements.

After adding these countermeasures, we can go to test them using the same process from Task 3.1. The results can be seen below.

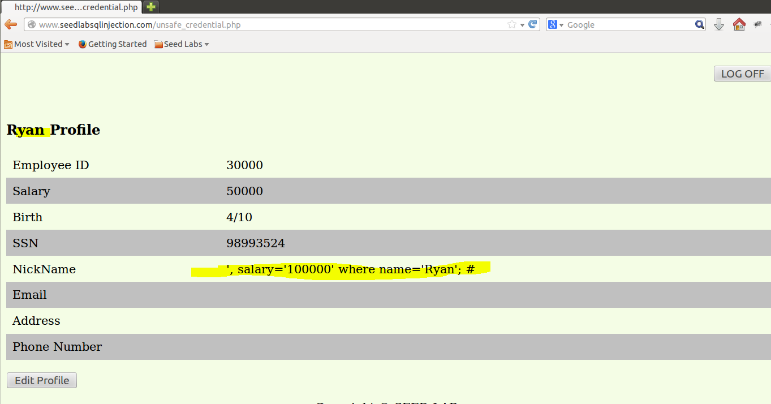


Fig. 17. Prepared statements successfully stop SQL Injection in UPDATE query.

After attempting to raise Ryan’s salary to “100000,” we can see that instead of the statement being altered the nickname entry is treated as a string and Ryan’s nickname is now our exploit code. This countermeasure was again a success and stopped the UPDATE query from updating an unwanted value.

## Analysis of countermeasure

This countermeasure was a success. It successfully prevented SQL code from being modified in both the SELECT and UPDATE queries using the same techniques from Tasks 2 and 3. The countermeasure was easily implemented.

# Results and Conclusion

The SQL injection attack was successful in all instances except one. The SQL injection attack was not able to run multiple queries off of one vulnerability. This is most likely due to the way the PHP code was written. If it were written to execute multiple statements at once, then the exploit would have been successful in every instance. This exploit is very devastating and can lead to massive amounts of information being gained by malicious actors as well as the possible loss of accounts if the attacker has knowledge of how passwords are stored and what hashes are being used. The countermeasures, however, were simple to implement and very successful. They were able to stop all of our previous attacks from Tasks 2 and 3. All programmers should use prepared statements when input is being taken directly or derivatively from user input to prevent these attacks.

References

[1] SQL Injection Attack Lab. Available:

http://www.cis.syr.edu/~wedu/seed/Labs\_12.04/Web/Web\_SQL\_Injection/SQL\_Injection.pdf.

[2] “How to use VirtualBox to Run Our Pre-built VM Image?” Available: http://www.cis.syr.edu/~wedu/seed/Documentation/Ubuntu12\_04\_VM/UseVirtualBox.pdf.

[3] Crypto Lab - Secret-Key Encryption. Available: http://www.cis.syr.edu/~wedu/seed/Labs\_12.04/Crypto/Crypto\_Encryption/.

1. [↑](#footnote-ref-1)