

# Lab 4

## SQL Injection

Brian Prost

SDEV350 6380: Database Security

Prof. Reginald Haseltine

June 28, 2021

## One

After reviewing the functionality of this application, my primary concern is the decision to have sensitive information such as social security numbers, salaries, and residential addresses of employees so easily accessible. I can't imagine being at any workplace where I could lookup this information about one of my coworkers, no matter their place in the organization's hierarchy. The fact that I also have the permissions to update any information on any employee is extremely dangerous.

Let's examine the privacy concerns. First, the problems presented in this application's retrieval. The first obvious problem is that there are some data points that shouldn't even be retrieved through an application like this. What need would any individual have to retrieve a social security number through a web-based form? The only time something like a social security number, salary, or address should be accessed would be for human resource purposes. There is not even a use case where an employee would need to access that information about themselves; they already know it.

Therefore, a different application would need to be provided to persons with those privileges to be used to access these data points, in accordance with the practice of then principle of least privilege.

For both the retrieving and updating data code, it is concerning that the user inputs are not sanitized. We are taking whatever the user enters and throwing that into our application. This isn't an issue for normal use and won't affect the performance of the application, but a malicious actor could inject code that would be executed within the database; aka. SQL Injection.

Another **glaring** problem is the fact that it looks like the password is being stored in plaintext. This should never be done, and is very concerning, especially so because this is one of the data points that a malicious actor is in search of. With

the right code for a SQL injection, every username and password would be returned to them in plain text.

## Two

To better protect passwords, all passwords should be salted and hashed using secure cryptographic algorithms. According to the NIST, password changes should not be automated to be changed by a certain schedule. While the contemporary wisdom before was that since passwords were changed so often, any hacker getting a password could potentially have an old password and not get access. However, we've learned that when password expiration practices are implemented, most people will change their password in a rather predictable pattern. For instance, if their password was "iFreakingHateMyJob" and then the company requires them to change it after 90 days, users won't generate another secure password; they'll just change it to something similar, like "iFr3akingH8MyJ0b2021".

Malicious actors are able to predict this type of behavior, and if they *do* get one of your passwords, they won't have much trouble trying to guess the new one. Therefore, other authentication security implementations should be considered.

As noted previously, for storing the password on the server, the consensus is: don't do that!

Instead, we should salt and hash the passwords with a hashing scheme such as "bcrypt" that prevent malicious actors from figuring out the original password and therefore not getting access to other server functions granted to that user.

Password complexity guidelines should also be instituted, even though the passwords are being hashed. It doesn't matter if we salt and store your password on the server if your password is easy to guess, like "password123". Oracle helps us out here by simply providing us complexity tags when setting up user accounts.

Last, and it cannot be stressed enough, is to enable a form of Two Factor Authentication for access. Though it should be noted that the NIST does not recommend that SMS codes be used as a form of 2FA.

As discussed before, the data points salary, SSN, address, and password should not be even stored on a web-facing server such as this, unless it has an explicit purpose for doing so. For storing extra private information such as social security numbers, it should be stored on a server behind an additional firewall from the open internet. No information has been given about the security of this server, but if it is not in a DMZ, it should be.

I mentioned that some people at a company may need to have ready access to this information. This is not rare but common, especially in a time such as now where work is done remote. Therefore, specific user roles should be created for these employees because it is essential for completing the duties of their job. This does not mean that the data should not be stored and secured as previously discussed, but the mechanisms that are put in place to access that data should only be extended to users who have been granted that role by the system administrator.

Though it isn't given at what point this web form is accessible, it should be stated that this form shouldn't be available for just any user. Since it is requesting data about people at the company, you shouldn't be able to have access to the form until you have already logged in. If you're not logged in, there should be no way to access the form, even if you have a URL for it.

Lastly, then user inputs are not being sanitized. This is problem #1. Any number of attacks and data extraction can be performed if we just let anybody throw any string of data into our application.

None of these recommendations for improvement will hinder the performance of the application, because it will still perform as well as it did before, as long as you are authorized to be performing any usage of it.

And not to be forgotten: document all security implementations.

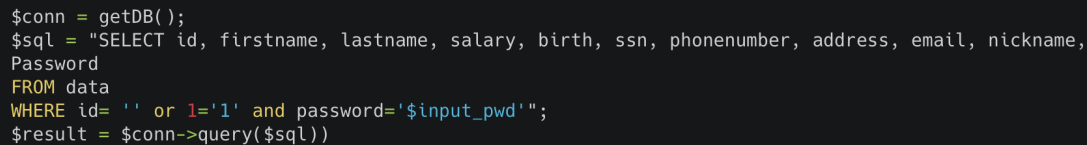
## Three

SQL Injection is 100% a problem presented in this application. Because user inputs are not sanitized, anything entered into our application will be processed. If that input is an SQL statement, processing it actually executes the statement. This is our database; only we get to execute SQL statements!

The only way to prove that it's an issue, though is to demonstrate it. One way is to enter this into the text box in the form where \$input\_id is grabbed from:

' or 1='1

So when our code for retrieval runs, this is what it would look like:



```
$conn = getDB();  
$sql = "SELECT id, firstname, lastname, salary, birth, ssn, phonenumber, address, email, nickname,  
Password  
FROM data  
WHERE id= '' or 1='1' and password='$input_pwd';"  
$result = $conn->query($sql)
```

When the program runs this, it will return a bunch of errors saying “failed to log in as x” where x would equal each and every username in the database. Boom. This simple input just exposed every username in the database.

As seen in the above graphic, \$input\_id was equal to ' or 1='1 which tricked the program into running a SQL select statement that read, “select these columns from the data table where id is blank or 1 is equal to 1.” Well, 1 is always equal to 1, so that equates out to TRUE. So the final version of the SQL statement would read, “select these columns from the data table where username is TRUE.” Note that this doesn't

mean where there is a plaintext value of 'true' but rather when there is a value, any value, in that column.

This makes sense now when we see the program returning a bunch of usernames. Because it tried to log in for each username where the id was equal to TRUE and when it failed (because we didn't know or enter a password) the program exited saying that it failed to log in as that person. And it iterated over every entry.

Not only does this give away user data, it violates another principle of secure software development which is to fail quietly. The program shouldn't print out an error that says "failed to log in as x," it should just print an error that says "operation failed." There is no reason to say that you failed to log in as a certain username, because if you were not a malicious actor, your username would still be in the form and you could examine if you had a typo in your entry.

## Four

To fix this issue, we must validate user inputs by using parameterized queries for code execution.

Instead of this:

```
$conn = getDB();
$sql = "SELECT id, firstname, lastname, salary, birth, ssn, phonenumber, address, email, nickname,
Password
FROM data
WHERE id= '$input_id' and password='$input_pwd'";
$result = $conn->query($sql)
```

We should do this:

```
$conn = getDB();
$sql = "SELECT id, firstname, lastname, salary, birth, ssn, phonenumber, address, email, nickname,
Password
FROM data
WHERE id = ? and password = ?";
$sql->bind_param("ss", $input_id, $input_pwd);
$result = $conn->query($sql);
```

In this new example code, we've fixed the SQL injection vulnerability by preparing the SQL statement ahead of time, we prevent the SQL statement being prepared based on user input. Basically, the SQL statement is prepared, and then the input parameters are **bound** to the statement. Because of this, there is no way to make our input become part of the SQL statement, only part of the variable of which the statement acts upon.

## References

- Berkeley - University of California. (n.d.). *Database Hardening Best Practices* | Information Security Office. University of California - Berkeley : Security. Retrieved June 28, 2021, from <https://security.berkeley.edu/education-awareness/database-hardening-best-practices>
- Grassi, P. A., Fenton, J. L., Newton, E. M., Perlner, R. A., Regenscheid, A. R., Burr, W. E., Richer, J. P., Lefkovitz, N. B., Danker, J. M., Choong, Y. Y., Greene, K. K., & Theofanos, M. F. (2017). Digital identity guidelines: authentication and lifecycle management. *Digital Identity Guidelines - Authentication and Lifecycle Management*. Published. <https://doi.org/10.6028/nist.sp.800-63b>
- HackHappy. (2018, June 22). *SQL Injection Attack Tutorial (2019)*. YouTube. <https://www.youtube.com/watch?v=WFFQw01EYHM>
- PHP MySQL Prepared Statements*. (n.d.). W3Schools. Retrieved June 30, 2021, from [https://www.w3schools.com/php/php\\_mysql\\_prepared\\_statements.asp](https://www.w3schools.com/php/php_mysql_prepared_statements.asp)
- What is SQL Injection? Tutorial & Examples* | Web Security Academy. (n.d.). PortSwigger. Retrieved June 29, 2021, from <https://portswigger.net/web-security/sql-injection>