SQL Injection Using SEED Labs SQLi Platform

James Cogswell
Cyber Security Engineering
George Mason University
Undergraduate
jcogswe@gmu.edu

Robert Weiner
Cyber Security Engineering
George Mason University
Undergraduate
rweiner@gmu.edu

ABSTRACT

Any web form which takes user input likely is connected to a SQL Database of some form. If the code powering the queries to this database does not properly prevent user input from having SQL commands, then the database is vulnerable to SQL Injection Attacks. With this vulnerability, a malicious user can access information they are not authorized to access, change table values, and even open new vulnerabilities on the hosting server to gain root access.

1 INTRODUCTION

Almost every business today has a database that contains terabytes or even petabytes of sensitive and personal information. Typically this type of information is kept on what is called SQL database server in the company's datacenter. In short, SQL databases are relatively easy to use, set up, and they offer a wide range of benefits to businesses that have to maintain large amounts of data. Some of these benefits, such as organization of large amounts of data, can also increase the security risk that the business must take on.

Security risks will always be present and must be of very high importance in any business model, and it is the job of security engineers to ensure these risks are handled efficiently. In medium-to-large organizations, data being compromised can cost anywhere from several millions (or even billions) of dollars to completely shutting down the business altogether. In this writing, some of the various issues and vulnerabilities associated with SQL servers will be explained, although at a relatively low-level as this is not meant to be a comprehensive guide to protecting your business' SQL databases.

2 PROJECT CHALLENGES / LAB SETUP

2.1 Project Challenges

Initially, we were planning on building our own mySQL Server and a simple web form to demonstrate our attack against. However, establishing and configuring a SQL Server, while simple for an enterprise operation, was far too much to accomplish for the two of us in the time we had. Thus, we elected to use the SEED Labs SQLi platform to conduct our attacks. This platform is created to be used only for very specific lab operations, and as such was very restrictive on what attack methods and commands we could use.

2.2 Lab Setup

The lab setup is fairly simple. First, there is a mySQL database that is hosted locally on the SEED Labs 16.04 VM. This database is connected to a web page hosted locally on an Apache2 server. This web page consists of an HTML page which is connected to a PHP script which allows the user's input in the HTML form fields to be fed into a SQL query. The output of the query is fed back into the HTML script to be displayed.

2.2.1 Apache2 Web Server Overview

Apache2 is a commonly used built-in web server component of Linux systems. This web server is what is used to host the sites and, in this case, perform the exploitation of the mySQL Database.

2.2.2 mySQL Database Overview

The VM came with a prebuilt mySQL Database containing only a "Users" database which contains one "credential" table. This table contains a fictional company's employee records, such as SSN, phone number, birthdate, salary, etc.

2.2.3 HTML Web Page Overview

The web page for this fictional corporation is very plain and simple, although this is very often not the case. For example, if you were to go to BestBuy.com, they have several different pages that have access to several different databases, and they're not all necessarily connected to each other. To explain in more detail, the Product database will be separate and different than the database attached to the Careers page. Because of this, you cannot search for careers in the Laptop section of the BestBuy website, for example.

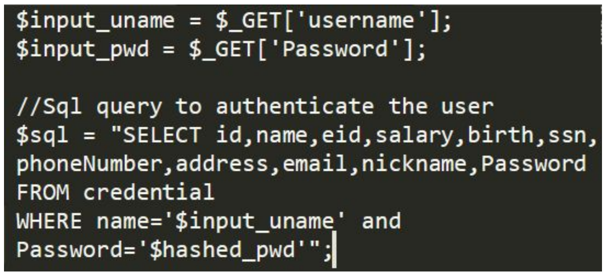
Therefore, any vulnerabilities within these separate databases will need to be secured individually. For this project, we used a simple page just so that we can try to explain a relatively nontrivial exploit in an easy-to-understand yet comprehensive manner.

However, it should be noted that in our example business website, the HTML is very restrictive for the inputs of the drawn tables. The purpose for this was that this web page was written for specific tasks in the SEED Lab, so only very specific values needed to be printed. Therefore, we will be unable to show certain SQL injection attacks such as drop tables.

2.2.4 PHP Code / Vulnerability Overview

PHP is often used to power the backends of web pages as it is built to interface with HTML and databases. This is a scripting, server-side language which allows it to be easily updated and for new updates to be pushed without having to push these updates directly to consumers. The ease of writing a PHP script can oftentimes make the programmer's working on it not notice glaring holes in the security of the script. One such example of this is with the SQL Injection (SQLi) vulnerability.

The SQLi vulnerability comes about from the PHP code taking the raw user input provided by the HTML web form and concatenating that directly into the SQL query being built. By doing this, the user's input will be interpreted literally as a part of the overall string. This means that is a user was enter
`' or 1=1#` then the SQL query in Figure 1 would end with `name='' or 1=1#`. This would make the query search for any row of the table where the `name` column is an empty string, or 1=1. The statement `1=1` always evaluates to `True` so every row will be returned.



```
$input_uname = $_GET['username'];
$input_pwd = $_GET['Password'];

//Sql query to authenticate the user
$sql = "SELECT id,name,eid,salary,birth,ssn,
phoneNumber,address,email,nickname,Password
FROM credential
WHERE name='$input_uname' and
Password='$hashed_pwd'";
```

Figure 1: PHP code vulnerable to SQLi

3 SQLi ATTACK WALKTHROUGH

In this SQLi attack example, we're going to explain a few simple, yet very destructive attacks one can perform, with the purpose of demonstrating the importance of properly handling user input on your web site.

3.1 Steps Before the Attack

There are several steps that must be taken in a real world attack before the real attack can go underway.

3.1.1 Determining if the Web Page is Vulnerable

Before launching the SQLi attack, we must first check to see if the web form is vulnerable to SQLi. To do this, we will send a simple SQLi command (such as `‵ or 1=1#`) to the server and see if it accepts the query. If there is an error stating this command is not valid, or if the HTML is not rendered at all, the server is not vulnerable.

3.1.2 Determining the Version of SQL Running on the Server

To begin issuing advanced commands, we need to know the syntax of the commands that will be accepted. This means we must investigate which version of SQL is running on the server. If we were to guess it is an Oracle SQL Server, we would issue the command

```
‵ UNION SELECT @@version#
```

This command unions the resulting table with the `@@version` keyword that is found in every Oracle SQL Server. In our case, this results in an error. This tells us this server is not running Oracle SQL Server.

In order to guess if it is a mySQL database, we would issue the command

```
‵ UNION SELECT v$version#
```

which unions the returned table with the `v$version` keyword found in mySQL databases. In our case, this will result in an error telling us there is a column mismatch. This means it is a mySQL database as the keyword is found and the error is simply due to the numbers of resulting columns not matching in the `UNION` statement.

3.1.3 Determining the Number of Columns

We now need to determine the number of columns in the table being queried so that our `UNION` statement can be padded with `NULL` values to yield output and not have an error. To do this, we can use the command

```
‵ UNION SELECT NULL, NULL,...#
```

to pad out a new empty `SELECT` statement with `NULL` values until a column mismatch error is not raised. The number of `NULL` values needed is the number of columns in the table being queried.

3.1.4 Finding the Names of Other Tables in the Database

We can now use the number of columns to issue a command to view all tables in the database so that we can expand our attack to getting information from tables we are not supposed to see. This command for a mySQL Database is:

```
‘ UNION SELECT table_name,
Table_type, engine, NULL, NULL,
NULL,...#
```

Where there is the appropriate amount of NULL values used to add up to the total number of columns needed as determined in the previous step.

3.2 Attack on the SELECT Statement

The PHP behind the website does not prevent us from typing in a specific command that allows us to login to whichever profile we choose. For instance, if I wanted to use SQLi to login as the admin for the site, I could type

```
‘ or name=’admin’#
```

into the username field and I will be granted access to the admin profile. See Figure 2 for an example of the execution of this exploit.

Once we have gained access into the administrator's profile, we can now see all of the information for each employee. Figure 3 shows the database access that we now have and it contains critical information such as Salary, Birthday, and SSN. Imagine if your information was readily available to someone with bad intentions. This is only the beginning.

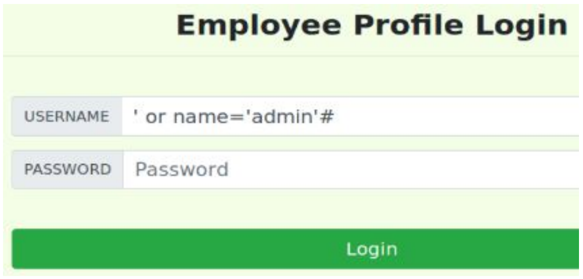
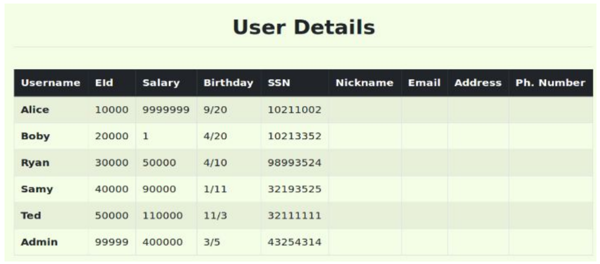


Figure 2: SQLi on Employee Login Screen



User Details

Username	EId	Salary	Birthday	SSN	Nickname	Email	Address	Ph. Number
Alice	10000	9999999	9/20	10211002				
Boby	20000	1	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				

Figure 3: Administrator Account Access

3.3 Attack on the UPDATE Statement

Similar to the previous attack on the SELECT statement from 3.1, the PHP code also doesn't prevent us from using specific commands to update the table values. Here, I'm going to log in as myself (Alice) normally, go to edit my profile, then execute another SQLi to update my own salary from 10000 to 99999. My boss Boby and I have been at odds lately, so perhaps I will change his salary to 1. The commands used to change my own salary as well as Boby's are shown below, respectively.

```
‘ , salary=’99999’ WHERE name=’Alice’;#
‘ , salary=’1’ WHERE name=’Boby’;#
```

See Figure 4 for the execution of this SQLi and Figure 5 for the results.

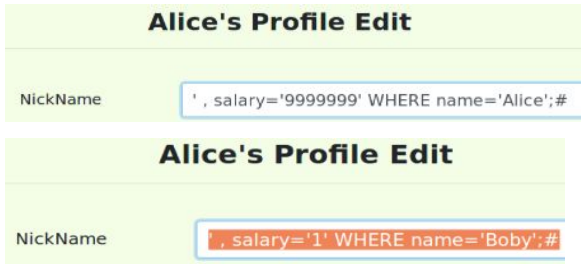
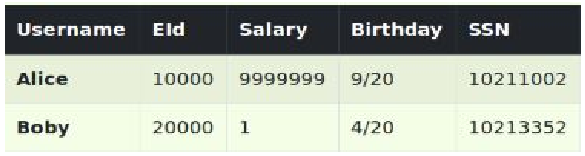

Alice's Profile Edit
NickName
', salary='9999999' WHERE name='Alice';#
Alice's Profile Edit
NickName
', salary='1' WHERE name='Boby';#

Figure 4: SQLi to Update Salary



Username	EId	Salary	Birthday	SSN
Alice	10000	9999999	9/20	10211002
Boby	20000	1	4/20	10213352

Figure 5: SQLi Results

3.4 SQLi to Change Admin Password

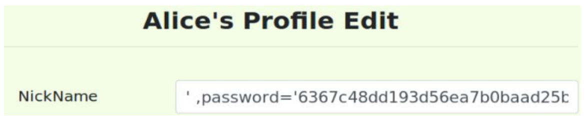
Another powerful tool we have at our disposal with SQLi exploits is that, if we can change the salary of any employee in the database, who's to say that if the passwords are in the same table we can't change the passwords too? We can, but first we need a little information. How are the passwords stored? They're likely not stored in plaintext so we need to first find out what hashing and possibly salting algorithm is used to obfuscate the plaintext passwords. I mention this because it is critical for this type of attack. For simplicity, I was provided with the hashing algorithm that was used - SHA1.

The next thing I did was go into the linux terminal and create a hash of the password 'abc123' which I will use as the new password to the administrator's account. The SQLi command used to do this is below.

```
' ,password='6367c48...5ee' WHERE
        name='Admin';#
```

Note: included '...' to save space.

The value inside the single quotes of the password field is the SHA1 hash of the password 'abc123'. When I inject the command in Figure 6 I can change the admin password and login regularly using that new password.


Alice's Profile Edit

NickName: ' ,password='6367c48dd193d56ea7b0baad25t

Figure 6: SQLi to Update Password

4 SQLi MITIGATIONS

Several mitigations exist for the SQLi vulnerability such as input sanitization, prepared statements, stored procedures, and enforcing least privilege onto the SQL query itself. Two of these are detailed below.

4.1 Mitigation 1: Input Validation

Input validation is used in many more scenarios than just mitigating SQLi risk, but it is also very applicable to this use case. What input validation consists of is either applying whitelist or blacklist rules to the input entered by the user.

In the case of mitigating SQLi, a blacklist is more applicable as you can blacklist all SQL commands that should never be present inside of a username or a valid search bar query for example. A piece of example pseudo-code is provided in Figure 7. This method can be extended even further by applying even more advanced methods of input validation such as training AI to differentiate between SQL statements and valid usernames.

Input validation can also be extended into other areas of the webpage or service itself, such as imposing rules on what is a valid username during creation. This can only go so far though as SQLi is applicable to far more than just login forms. Due to this reason, the next method is generally regarded as a more powerful and versatile method.

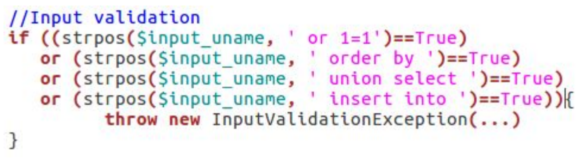

```
//Input validation
if ((strpos($input_uname, ' or 1=1')==True)
    or (strpos($input_uname, ' order by ')==True)
    or (strpos($input_uname, ' union select ')==True)
    or (strpos($input_uname, ' insert into ')==True)){
          throw new InputValidationException(...)
}
```

Figure 7: Example code to validate user input

4.2 Mitigation 2: Prepared Statements

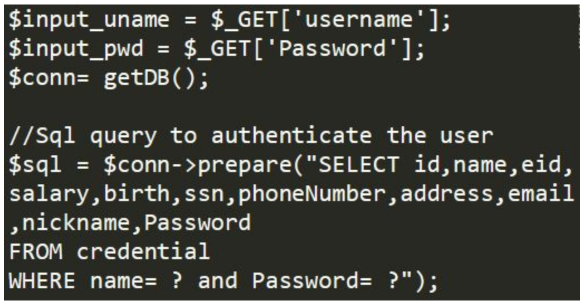
Prepared statements are generally regarded as the strongest defense against a SQLi attack. This method consists of three steps: Creating a prepared SQL query with placeholders for user input, binding the user input to the placeholders, and executing the query with these bound parameters in place.

Here, we will be discussing the implementation in PHP specifically.

The first step is fairly simple. Instead of concatenating the variables holding the user input directly into the SQL query, you instead replace them with question marks and wrap the prepared query string in a call to `prepare("...")` as seen in Figure 8. This sends this skeleton-query to the SQL Database for pre- pre-processing.

The second step is to bind the placeholders in the prepared query to parameters. This is done by making a call to `bind_param("..." , .. , ..)` on the `$sql` variable object. The first parameter of this function is a string holding the data types of the variables being fed in ('s': String, 'i': Integer, etc), in the same order as the variable parameters. The string will have an amount of characters equal to the number of variables being fed in. The rest of the parameters being fed into this function are the variables being placed in for the placeholders. Example code is shown in Figure 9.

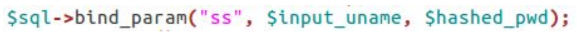
Lastly, a simple call to `execute()` is done on the `$sql` object to tell the database to execute the now fully created query.



```
$input_uname = $_GET['username'];
$input_pwd = $_GET['Password'];
$conn= getDB();

//Sql query to authenticate the user
$sql = $conn->prepare("SELECT id,name,eid,
salary,birth,ssn,phoneNumber,address,email
,nickname,Password
FROM credential
WHERE name= ? and Password= ?");
```

Figure 8: PHP code to prepare a SQL query



```
$sql->bind_param("ss", $input_uname, $hashed_pwd);
```

Figure 9: PHP code showing a bind_param call that would be done after Figure 8

5 CONCLUSION

In conclusion, SQLi can be an extraordinarily devastating attack on any business that values data privacy (which is pretty much all of them). Preventing exploitation of SQLi can be very easy to implement but is often times forgotten. Nothing is ever perfectly secure, so taking several steps to ensure the security of your assets and your user's PII must be a top priority.

What we have shown here is an introduction to SQLi and is by no means meant to be a comprehensive guide to preventing them. It is our hope that a basic understanding of the fundamentals behind SQLi attacks and how simple the attack can truly be will assist - or at least motivate - future security professionals in their duty to engineer secure systems for the business to which they have committed to defending. Data has quickly, yet unexpectedly become arguably the most valuable asset to attackers and certainly to those who want to pay their bills by protecting it. It is a battle that will likely never cease to exist, which is why it is important for security professionals to keep their skills sharp.

6 REFERENCES

https://seedsecuritylabs.org/Labs_16.04/PDF/Web_SQL_Injection.pdf
https://www.youtube.com/watch?v=ciNHn38EyRc&list=PLHInliae5oqXGS_oHLDayC8mmHuY_Ld4z&index=4&t=0s
https://www.imperva.com/learn/application-security/sql-injection-sqli/
http://pentestmonkey.net/cheat-sheet/sql-injection/mysql-sql-injection-cheat-sheet
https://www.acunetix.com/websitesecurity/sql-injection/
https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html
http://www.zbeanztech.com/blog/important-mysql-commands