

## Lab 09 - Web Security

### Objectives

- Web vulnerabilities, both server-side and client-side
- Server-side SQL injection
- Cross-Site Scripting, Cross-Site Request Forgery

### Background

#### SQL Injection

SQL Injection is a server-side code injection vulnerability resulting from improper (unsanitized) input directly concatenated into SQL queries. Typical server queries are built as strings:

```
sql = "SELECT * FROM table WHERE item = " + user_input_variable + " " <other expressions>";
database.query(sql);
```

Note that the user may choose to escape the SQL quotes and alter the SQL statement, e.g.:

```
user_input_variable = " ' OR 1=1 -- "; // example input given by the user
sql = "SELECT * FROM table WHERE item = " + user_input_variable + " " <other expressions>";
```


An SQL injection exploit ultimately depends on the target SQL expression (which is usually unknown to the attacker) and query result behavior (whether the query contents are displayed on screen or the user is blind, errors reported etc.).

#### Make sure to check those cheatsheets out:

- <https://portswigger.net/web-security/sql-injection>
  - <https://github.com/swisskyrepo/PayloadsAllTheThings/tree/master/SQL%20Injection>
- and:
- <https://github.com/swisskyrepo/PayloadsAllTheThings/blob/master/SQL%20Injection/MySQL%20Injection.md>

### Other server-side vulnerabilities

The SQL injection is a popular server-side code injection vulnerability, but there are many mistakes that a website developer / system administrator can make (*expect to find some of them in your homework :P*):

- code injection (LDAP, eval, shell execution etc.);
- broken authentication or access control (authorization);
- sensitive data exposure (e.g., backups / keys forgotten inside web root);
- path traversal;
- server misconfiguration;
-  *and many more*



There are even  free or commercial web vulnerability scanners for testing a server's security!

### Client-side vulnerabilities

Browsers are now among the most targeted pieces of software on the Internet. This is mainly because of the large threat vector resulting from the complexity of the web ecosystem, requiring features such as fancy **HTML+CSS** rendering, animation and even sandboxed, untrusted JavaScript code execution.

Even when the browsers do a good job at protecting against attacks, sometimes trusted websites themselves may contain bugs that directly affect the security of their users.

A major threat, **Cross Site Scripting (XSS)** is a JavaScript code injection vulnerability where an attacker that found a way to post public **HTML** scripts into an unprotected website (e.g., by using comments forms or forum responses). Those scripts, if served to other visitors, will execute with the credentials of their respective users, making it possible for the attacker to scam, exfiltrate personal data or even push malware into the victim's PC.

Another typical client-side vulnerability that the web developers need to protect their websites against is  **Cross-Site Request Forgery (CSRF)**. In this attack, the victim is tricked into opening an attacker-controlled web page which then issues custom requests (either using concealed elements that do external requests - `img`, `form`, or by using JavaScript / AJAX) to another (target) website. The browser will happily make the requests using the target domain's cookies and credentials. If the target website has URLs that execute certain actions (e.g., POST  <https://my-blog/post.php>) without verifying the source of the request, any malicious page can execute them. Note that the attacker cannot see the results of those requests (unless authorized by CORS headers by the target). In practice, any URL endpoint executing sensitive actions needs to be protected using either referer validation or CSRF tokens.

### Setup

You will be using a  OpenStack VM for your tasks.

Remember that the instances have private IPs, `10.9.x.y`, inaccessible from anywhere but the campus network. Since we need to use a local browser to access a web server running inside the VM, we will employ a SSH tunnelling + proxy trick to accomplish this.

You should already have a **SSH keypair** for authenticating with `fep` & OpenStack:

We will be using `ssh`'s Local Port Forwarding feature, requesting it to pass all packets from `localhost:8080` through the SSH tunnel to the destination VM on `8080`:

```
ssh -L "8080:localhost:8080" -J <first.lastname>@fep.grid.pub.ro student@10.9.X.Y
```



### Tasks

#### 1 [20p]. SQL Injection

- Start the web server by using the following sequence:

```
# First, start the MySQL instance in background
docker run -d --rm --name mysql ropubisc/lab08-mysql
# Wait until the MySQL server fully starts:
docker logs mysql -f
# Ctrl+C and continue when it says: 'mysqld: ready for connections.'

# Finally, start the sample web server
docker run -it --link mysql:mysql -p 8080:8080 ropubisc/lab08-web-server
```

- Connect to the application using  `http://localhost:8080/` (assuming you forwarded the port correctly)
- Now: You don't know any user / password for this website. Try to log in using SQL Injection!
- The most common approach when testing for SQL Injection is to input an apostrophe (") in any of the provided fields ( <https://security.stackexchange.com/questions/67972/why-do-testers-often-use-the-single-quote-to-test-for-sql-injection>)
- Hint: Try the apostrophe in one of the login fields, check if it shows an error!
- Note: After examining the error the form prompts, we can assume how the query is being made:

```
'SELECT <some columns> FROM users WHERE username = ' + username + ' AND password = ' + password
<
>
```

- Is pretty obvious that the strings provided are not escaped and we can abuse this misconfiguration. Check the links in the beginning!
- If you ever want to exit the MySQL server:

```
docker kill mysql
```

#### 2 [20p]. Advanced SQL Injection

- Start the web server from the first task again.
- What if I told you there is a hidden **flag** inside the database? Find it!
- Hint: where do you have query feedback inside the application? try to do an UNION hack!
- Note: Since we will be using the same query as the one used in the first exercise, we must first find the exact number of columns provided to the statement. We are aiming at building a query of this format:

```
SELECT col1, col2, ..., colN FROM users WHERE username = '' UNION SELECT col1, col2, ... , colN-
<
>
```

- Note: for UNION to work, you must SELECT exactly the same number of columns as in the original query!
  - After finding out the exact number of columns, we can use `GROUP_CONCAT` technique to extract the available database table names. Check out the cheatsheets from the Background section! (P.S. database schema is `journalapp`)
  - Hint:
- ```
UNION SELECT col1, col2, ... , colN-1, GROUP_CONCAT(<what are we looking for in the schema>) FR
<
>
```

It is not necessary to know the exact names of 'col1', 'col2', ... 'colN-1'. You can replace it with numbers or '@'.


- Got any table that catches your eye? We are going to use `GROUP_CONCAT` again, but this time we are trying to find the name of the columns of our desired table.

Hint:


```
UNION SELECT col1, col2, ... , colN-1, GROUP_CONCAT(<what are we looking for in the table>) FRO
<
>
```

- Got the column name? Good. Now it should be nothing more than a simple select query :)
  - Hint:
- ```
UNION SELECT col1, col2, ... , colN-1, <desired column name> FROM <desired table name>
```


#### 3 [20p]. Cross-Site Scripting

- You can still save the day: cover the monster's mouth (you can use the image at  `http://localhost:8080/images/muzzle.png`)!
- Since **HTML** is allowed, you can also inject a JavaScript alert, example:



```
<script>alert("XSS!");</script>
```

- Hint:** You can use absolute element positioning, e.g.: `<div style="position: absolute; top: -300px;left:100px;"> insert your img here </div>`. Try it with the browser's developer console / inspect element first before injecting it inside a message ;)
-  [https://www.w3schools.com/css/css\\_positioning.asp](https://www.w3schools.com/css/css_positioning.asp)
- Hint: Console can be accessed either by right-clicking and choosing Inspect or by hitting F12 on your keyboard and navigating to 'Console' tab.
- Note: Try appending code to the document's body.

#### 4 [20p]. Cross-Site Request Forgery

- The objective is to fool your victim using an external website (simulated using a local .html page) to post an attacker-controller message into the website.
- On your local station, as the attacker: create a simple **HTML** page that posts a hidden message to  `http://localhost:8080/journal/post`; this will be equivalent to hosting a malicious website;
- Hint: Check the **HTML** of the website for a `<form>` example!
- Hint: Use an input with `type="hidden"`
- Switching sides (you're the victim, now): open that **HTML** page using your web browser. Click a button and the hidden message will be posted ;)
- (Make sure you are logged in inside the web app before doing the attack!)
- For bonus, you can try to do a cross-site AJAX posting the message automatically (without requiring user interaction!).

#### 5 [20p]. Server Reconnaissance

- Can you steal the source code of the server-side code using HTTP only?
- Once you found it, try to find the database credentials!
- Hint: try to guess the path to a  common file that all NodeJS projects have! It may reference the main script's name!
- Also try it by using a tool:  `nikto`, `apt install nikto`

### Feedback

Please take a minute to fill in the  feedback form for this lab.

Search

#### Lectures

- Lecture 01 - Introduction
- Lecture 02 - Cryptography
- Lecture 03 - Hardware Security
- Lecture 04 - Access Control
- Lecture 05 - Authentication and Key Establishment
- Lecture 06 - Application Security
- Lecture 07 - Operating System Security
- Lecture 08 - Network Security
- Lecture 09 - Web Security
- Lecture 10 - Privacy Preserving Technologies
- Lecture 11 - Forensics

#### Labs

- kernel
  - Lab 01 - Introduction
  - Lab 02 - Cryptography
  - Lab 03 - Authentication and Key Establishment Lab
  - Lab 03 - Hardware Security
  - Lab 04 - Access control
  - Lab 05 - Authentication in Linux
  - Lab 06 - Application Security
  - Lab 07 - Operating System Security
  - Lab 08 - Network Security
  - Lab 09 - Web Security
  - Lab 10 - Forensics
  - Lab 11 - Privacy Technologies
  - Lab 12 - Security and Machine Learning

#### Support

- Useful resources
- Virtual Machine

#### Table of Contents

- Lab 09 - Web Security
  - Objectives
  - Background
    - SQL Injection
    - Other server-side vulnerabilities
    - Client-side vulnerabilities
  - Setup
  - Tasks
    - 1 [20p]. SQL Injection
    - 2 [20p]. Advanced SQL Injection
    - 3 [20p]. Cross-Site Scripting
    - 4 [20p]. Cross-Site Request Forgery
    - 5 [20p]. Server Reconnaissance
    - Feedback