Systems Security lab 4: SQL injection

Preparation

Start the seed VM. The collabtive web application is located at *www.sqllabcollabtive.com* (this is an `/etc/hosts` entry pointing to localhost, no internet access required). The default username/password is **`admin/admin`**. From the admin panel you will see that there are also users alice, peter, bob, ted and seed – you can probably guess their passwords.

The application is written in PHP and runs on a locally hosted apache2 server with source files in `/var/www/SQL/Collabtive` and a MYSQL database.

First we will turn off magic quotes, a PHP protection mechanism that escapes quotes in user input (such as form fields). To do this, open the file `/etc/php5/apache2/php.ini` as root and change line 756 to read

```
magic_quotes_gpc = Off
```

Then run the command

```
sudo service apache2 restart
```

You can browse to the application with the firefox browser installed on the lab VM. This has the Tamper Data extension preinstalled. Activate this with Tools/Tamper Data then select the "Start Tamper" button. Whenever your browser now sends a request, you can choose to tamper with the parameters.

For a task to be complete, it has to run on an unmodified version of the lab application unless the task says otherwise. You may want to run a modified version on a second VM however where you can add extra logging – to have a peek at the SQL statements being constructed, you could change the login statement as follows

```
$q = "SELECT ID ...
file_put_contents("/tmp/log", "SQL: " . $q . "\n", FILE_APPEND);
$sel1 = mysql_query($q);
```

This appends the generated SQL to a file that you can watch with "tail -f" in a terminal.

Task 1: SQL injection on the login form

Your first task is to login as admin without knowing the password. The typical SQL injection attack works something like this: if the constructed query is

```
SELECT ID FROM users WHERE name = '$user' AND pass = '$pass'
```

then entering the following values

```
user: admin
pass: ' OR '' = '
```

would interpolate to

```
SELECT ID FROM users WHERE name = 'admin' AND pass = '' OR '' = ''
```

which skips the password check.

This won't work directly for us. The login code starts at line 363 of `/var/www/SQL/Collabtive/include/class.user.php` and looks like this

```
function login($user, $pass)
{
    if (!$user)
    {
        return false;
    }

    $pass = sha1($pass);
    $sel1 = mysql_query("SELECT ID,name,locale,lastlogin,gender
                         FROM user WHERE (name = '$user' OR
                         email = '$user') AND pass = '$pass'");
    $chk = mysql_fetch_array($sel1);
    if ($chk["ID"] != "")
    {
        ... // set up session
        return true;
    }
    else
    {
        return false;
    }
}
```

We can't do anything with $pass because it's hashed. To make things worse, $user lives inside brackets too. *Hint: the # character starts a MYSQL comment until the end of line.*

- Task 1.1 : log into the admin account without knowing the password
- Task 1.2 : explain why you can't use this particular query to overwrite data

For task 1.2, once you've figured out how to get the SQL injection running it would be really tempting to do something like this:

```
(stuff) ... ; UPDATE user SET pass =
'c6dda4f283c9e13682ea3aef2e1732dd80b63cbc'
WHERE name = 'admin';
```

That hex string is the sha1 hash of 'owned'. The semicolon separates SQL statements. Unfortunately this *will not work* – your task 1.2 is to explain why not.

Task 2 – SQL injection on UPDATE

Log in as Peter and go to My Account, Edit (Wrench icon). Submitting this form runs an UPDATE statement (function edit on line 82 of user.class.php). You can see in the source that the programmer applied mysql_real_escape_string to most parameters but "forgot" this for the $company one.

- Task 2.1 : as Peter, overwrite information in Bob's account (try changing his e-mail to peter@evil.com). *But don't change Bob's name!*
- Task 2.2 : still as Peter, change Bob's password to 'peter' (without quotes). Again without changing Bob's name.

Task 3 – Countermeasures

- Task 3.1: Explain what the *magic_quotes_gpc* setting does that I asked you to turn off at the beginning. Show what happens when you attempt your attack from Task 1.1 with magic quotes on (you can log queries as described on page 1). You will have to restart the apache2 service after editing the php.ini file.
- Task 3.2: Explain what *mysql_real_escape_string* does (for extra marks, also compare it to the version without "real_"). Explain what happens to your attack in Task 2.1 with this protection activated for the company field too. (Turn magic quotes *off* again for this task.)
- Task 3.3: Explain how *prepared statements* can secure a database query. How would you use a prepared statement for the login function?