SQL Injection Attack Lab

CMSC 426/626

Based on *SQL Injection Attack Lab — Using Collabtive*
Adapted and published by Christopher Marron, UMBC

1 Overview

SQL injection is a code injection technique that exploits the vulnerabilities in the interface between web applications and database servers. The vulnerability is present when a user's input is not correctly checked within the web application before being sent to the back-end database servers.

Many web applications accept inputs from users and use these inputs to construct SQL queries in order to access or update information in databases. When the SQL queries are not carefully constructed, SQL injection vulnerabilities can result. SQL injection attacks are one of the most frequent attacks on web applications.

For this lab, we modified a web application called `Collabtive`, disabling several countermeasures implemented by `Collabtive`. As a result, we created a version of `Collabtive` that is vulnerable to SQL injection attacks. Although our modifications are artificial, they are representative of mistakes made by many web developers. Your goals in this lab are to exploit the SQL injection vulnerabilities, demonstrate the damage that can be achieved by the attacks, and master techniques that can help defend against such attacks.

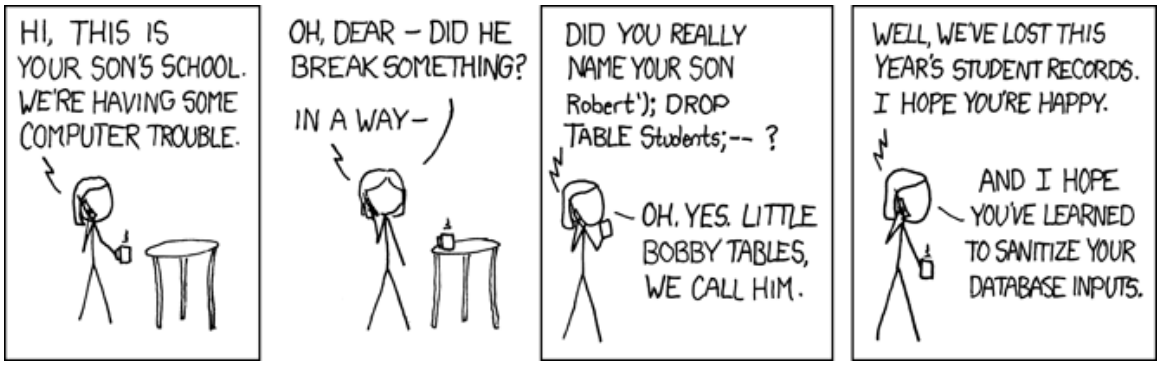

Figure 1: Exploits of a Mom (http://xkcd.com/327)

2 Lab Environment

The name of the VM image that supports this lab is called `SEEDUbuntu12.04`, built in September 2013. If you have a pre-built VM image prior to `SEEDUbuntu12.04`, you must download the new version from the SEED web site (`http://www.cis.syr.edu/~wedu/seed/`).

2.1 Environment Configuration

The lab requires three software packages, all of which are already installed in the provided VM image:

1. The Firefox web browser
2. The Apache web server
3. The `Collabtive` project management web application

In Firefox, you may want to use the `LiveHTTPHeaders` extension to inspect the HTTP requests and responses. The pre-built `Ubuntu` VM image provided to you has the Firefox web browser with the required extensions already installed.

Starting the Apache Server. The Apache web server is also included in the pre-built `Ubuntu` image. However, the web server is not started by default. You need to first start the web server using the following command:

```
% sudo service apache2 start
```

The `Collabtive` Web Application. We use an open-source web application called `Collabtive` in this lab. `Collabtive` is a web-based project management system. It is already installed and configured in the pre-built `Ubuntu` VM image. We have also created several user accounts on the `Collabtive` server. To see all the users' account information, first log in as the admin (username `admin`, password `admin`). Once logged in as admin, other users' account information can be obtained from the User's Account Information Project on the front page.

DNS Configuration. We have configured the `Collabtive` server to use the following URL:

URL	Description	Directory
`http://www.sqllabcollabtive.com`	Collabtive	`/var/www/SQL/Collabtive/`

The URL is only recognized from inside of the virtual machine, because we have modified the `/etc/hosts` file to map the domain name of the URL to the virtual machine's local IP address (`127.0.0.1`). You may map any domain name to a particular IP address by editing `/etc/hosts`. For example you can map `http://www.example.com` to the local IP address by appending the following entry to `/etc/hosts`:

```
127.0.0.1       www.example.com
```

If your web server and browser are running on two different machines, you must modify `/etc/hosts` on the browser's machine to map the domain name to the web server's IP address (*not* `127.0.0.1`).

2.2 Turn Off the Countermeasures

PHP provides a mechanism called ”magic quotes” to automatically defend against SQL injection attacks (this protection method is deprecated after PHP version 5.3.0). We must disable magic quotes for the first two lab tasks:

1. Go to `/etc/php5/apache2/php.ini`.
2. Find the line: `magic_quotes_gpc = On`.
3. Change it to this: `magic_quotes_gpc = Off`.
4. Restart the Apache server by running "`sudo service apache2 restart`".

3 Lab Tasks

3.1 Task 1: SQL Injection Attack on `SELECT` Statements

In this task, you need to log into `Collabtive` at `www.sqllabcollabtive.com`, *without* providing a password. You can achieve this using SQL injection. Normally, before users start using `Collabtive`, they need to login using their user names and passwords. `Collabtive` displays the following login window asking the user to input a `username` and `password`.

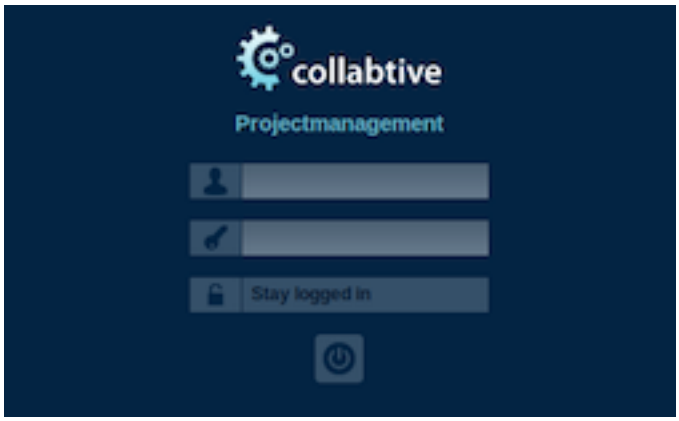

Figure 2: The login window

User authentication is implemented in `include/class.user.php` under the Collabtive root directory (`/var/www/SQL/Collabtive/`). It takes the user-provided login data and determines whether it matches the user name and password fields of any record in the user database. If there is a match, the user has provided a correct user name and password combination and should be allowed to login. Like most web applications, PHP programs interact with their back-end databases using the standard SQL language. In `Collabtive`, the following SQL query is constructed in `class.user.php` to authenticate users:

```
$pass = sha1($pass);

$sel1 = mysql_query ("SELECT ID, name, locale, lastlogin, gender,
    FROM  user
    WHERE (name = '$user' OR email = '$user') AND pass = '$pass'");
```

In the above SQL statement, `user` is the name of the SQL table that holds the user data. The variable `$user` holds the string typed in the `Username` textbox, and `$pass` holds the string typed in the `Password` textbox. User's inputs in these two textboxs are placed directly in the SQL query string. If the SELECT statement returns at least one result, the user is allowed to login.

SQL Injection Attacks on Login: There is a SQL injection vulnerability in the above query. Can you take advantage of this vulnerability to achieve the following objectives?

- **Task 1.1**: Can you log into another person's account without knowing the correct password?
- **Task 1.2**: Can you find a way to modify the database (still using the above SQL query)? For example, can you add a new account to the database, or delete an existing user account? Obviously, the above SQL statement is a query-only statement, and cannot update the database. However, using SQL injection, you can turn the above statement into two statements, with the second one being the update statement. Please try this method, and see whether you can successfully update the database.

  To be honest, we are unable to achieve the update goal. This is because of a particular defense mechanism implemented in MySQL. In the report, you should show us what you have tried in order to modify the database. You should find out why the attack fails and what mechanism in MySQL has prevented such an attack. You may look up evidence (second-hand) from the Internet to support your conclusion. However, first-hand evidence will get more points (use your own creativity to find first-hand evidence). If you find ways to succeed in the attacks, you will be awarded bonus points.

3.2 Task 2: SQL Injection on `UPDATE` Statements

In this task, you need to make an unauthorized modification to the database. Your goal is to modify another user's profile using SQL injection. In `Collabtive`, if users want to update their profiles, they can go to `My account`, click the `Edit` link, and then fill out a form with the new profile information. When a user sends the update request to the server, an `UPDATE` SQL statement is constructed in `include/class.user.php`. The purpose of this statement is to modify the current user's profile information in the `users` table. There is a SQL injection vulnerability in the SQL statement. Find the vulnerability and use it to do the following:

- Change another user's profile without knowing his or her password. For example, if you are logged in as Alice, your goal is to use the vulnerability to modify Ted's profile information, including Ted's password. After the attack, you should be able to log in to Ted's account.

3.3 Task 3: Countermeasures

The fundamental problem of SQL injection vulnerability is the failure to separate code from data. When constructing a SQL statement, the program (e.g. PHP program) knows what part is data and what part is code. Unfortunately, when the SQL statement is sent to the database, the boundary between data and code may be blurred, allowing a malicious user to insert SQL code through a data field. To solve this problem, it is important to ensure that the views of the boundaries are consistent in the server-side code and in the database. There are various ways to achieve this.

- **Task 3.1: Escaping Special Characters using `magic_quotes_gpc`.** In the PHP code, if a data variable is the string type, it needs to be enclosed within a pair of single quote symbols ('). For example, in the SQL query listed above, we see `name = '$user'`. The single quotes surrounding

`$user` are an attempt to separate the data in the `$user` variable from the code. Unfortunately, this separation will fail if the contents of `$user` include any single quote. Therefore, we need a mechanism to tell the database that a single quote in `$user` should be treated as part of the data, not as a special character in SQL. All we need to do is to add a backslash (\) before the single quote.

PHP provides a mechanism to automatically add a backslash before single-quote ('), double quote ("), backslash (\), and NULL characters. If this option is turned on, all these characters in the inputs from the users will be automatically escaped. To turn on this option, go to `/etc/php5/apache2/php.ini`, and add `magic_quotes_gpc = On` (the option is on by default in the VM provided to you; you should have turned it off as part of the lab environment configuration). Remember, if you update `php.ini`, you need to restart the Apache server by running "`sudo service apache2 restart`"; otherwise, your change will not take effect.

Turn the magic quote mechanism on and off and see how it helps protect against SQL injection attacks.

Note that starting in PHP 5.3.0 (the version in our provided VM is 5.3.10), the feature has been deprecated[1] for several reasons including:

- Portability: Assuming it to be on, or off, affects portability. Most code must use the function `get_magic_quotes_gpc()` to check the state and adjust accordingly.
- Performance and Inconvenience: not all user inputs are used for SQL queries, so mandatory escaping of all data not only affects performance, but also become annoying when some data are not supposed to be escaped.

- **Task 3.2: Escaping Special Characters using** `mysql_real_escape_string`. A better way to escape data to defend against SQL injection is to use database specific escaping mechanisms instead of relying upon features like magical quotes. PHP provides an escaping mechanism, called `mysql_real_escape_string()`, which prepends backslashes to a few special characters, including `\x00`, `\n`, `\r`, `\`, `'`, `"` and `\x1A`.

  Use this function to fix the SQL injection vulnerabilities identified in the previous tasks. You should disable the other protection schemes described in the previous tasks before working on this task. *Note:* `mysql_real_escape_string` has been deprecated since PHP 5.5.0.

- **Task 3.3: Prepare Statement.** A more general solution to separating data from SQL logic is to tell the database exactly which part is data and which is code. PHP provides `mysqli` and the `prepare` mechanism for this purpose. `mysqli` is one of two preferred methods for protecting against SQL injection attacks in PHP code (the other is the `PDO_MYSQL` extension).

[1] In the process of authoring computer software, its standards, or documentation, `deprecation` is a status applied to software features to indicate that they should be avoided, typically because they have been superseded. Although deprecated features remain in the software, their use may raise warning messages recommending alternative practices, and deprecation may indicate that the feature will be removed in the future. Features are deprecated — rather than immediately removed — in order to provide backward compatibility, and to give programmers who have used the feature time to bring their code into compliance with the new standard.

```
$db = new mysqli("localhost", "user", "pass", "db");
$stmt = $db->prepare("SELECT ID, name, locale, lastlogin
                      FROM users
                      WHERE name=? AND age=?");
$stmt->bind_param("si", $user, $age);
$stmt->execute();

// The following two functions are used to retrieve the
// results of the SELECT statement
$stmt->bind_result($bind_ID, $bind_name, $bind_locale,
                                    $bind_lastlogin);
$chk=$stmt->fetch();
```

Parameters for the `mysqli()` constructor can be found in `config/standard/config.php`. Using the prepare statement mechanism, we can divide the process of sending a SQL statement to the database into two steps. The first step is to send the code, i.e., the SQL statement without the data. This is the prepare step. The second step is to send the data to the database using `bind_param()`. The database will treat everything sent in this step as data, not code. The first argument of `bind_param` indicates the types of the bound variables; in the example above, `"si"` indicates that the first parameter (`$user`) has a string type, and the second parameter (`$age`) has an integer type. Valid types are "i" (integer),"d" (double), "s" (string), and "b" (blob).

If we wish to retrieve the results of a `SELECT` statement, we also need to bind variables to the selected fields and then fetch the results into the bound variables.

Use the prepare statement mechanism to fix the SQL injection vulnerabilities in the `Collabtive` code.

4 Helpful Hints

Print out debugging information. When we debug traditional programs (e.g. C programs) without using any debugging tool, we often use `printf()` to print out some debugging information. In web applications, whatever is printed out by the server-side program is actually displayed in the web page sent to the users; the debugging printout may mess up the web page. There are several ways to solve this problem. A simple way is to print all the information to a file. For example, the following code snippet can be used by the server-side PHP program to print the value of a variable to a file.

```
$myFile = "/tmp/mylog.txt";
$fh = fopen($myFile, 'a') or die("can't open file");
$Data = "a string";
fwrite($fh, $Data . "\n");
fclose($fh);
```

A useful Firefox Add-on. Firefox has an add-on called `"Tamper Data"`, it allows you to modify each field in the HTTP request before the request is sent to the server. For example, after clicking a button on a web page, an HTTP request will be generated. However, before it is sent out, the `"Tamper Data"` add-on

intercepts the request, and gives you a chance to make an arbitrary change on the request. This tool is quite handy in this lab.

The add-on only works for Firefox versions 3.5 and above. If your Firefox has an earlier version, you need to upgrade it to use this add-on. In our most recently built virtual machine image (SEEDUbuntu12.04), Firefox is already upgraded to version 23.0, and the `"Tamper Data"` add-on is already installed.

5 Submission

You must submit a detailed lab report to describe what you have done and what you have observed. You may include details using `LiveHTTPHeaders`, `Wireshark`, or screen shots. The lab write-up must be typed and well-written, using proper English language and grammar. In particular, your lab write-up must include the following information:

Task 1.1 input to the login screen that results in a successful injection attack; written description of how the injection attack was discovered and why it succeeds.

Task 1.2 user input with which you attempted to modify the database; written description of countermeasures that prevented your attack from succeeding *or* description of why your attack succeeded.

Task 2 input to the profile update screen that results in a successful injection attack; written description of how the injection attack was discovered and why it succeeds.

Task 3.1 results of running your injection attacks from Tasks 1.1 and 2 with "magic quotes" enabled.

Task 3.2 results of running your injection attacks from Tasks 1.1 and 2 using `mysql_real_escape_string`; code excerpts showing your use of the function in the `Collabtive` PHP code.

Task 3.3 results of running your injection attacks from Tasks 1.1 and 2 using `mysqli`.

In addition to the written lab report, you must turn in your code for `class.user.php` implementing the `mysqli` approach from Task 3.3.