SQL INJECTION

SQL injection refers to the act of someone inserting a MySQL statement to be run on your database without your knowledge. Injection usually occurs when you ask a user for input, like their name, and instead of a name they give you a MySQL statement that you will unknowingly run on your database.

sql injection example

Below is a sample string that has been gathered from a normal user and a bad user trying to use SQL Injection. We asked the users for their login, which will be used to run a SELECT statement to get their information.

MySQL & PHP Code:

```
// a good user's name

$name = "timmy";

$query = "SELECT * FROM customers WHERE username = '$name'";

echo "Normal: " . $query . "<br />";

// user input that uses SQL Injection

$name_bad = " ' OR '1 ";

// our MySQL query builder, however, not a very safe one

$query_bad = "SELECT * FROM customers WHERE username = '$name_bad'";

// display what the new query will look like, with injection

echo "Injection: " . $query_bad;
```

Display:

```
Normal: SELECT * FROM customers WHERE username = 'timmy'
Injection: SELECT * FROM customers WHERE username = '' OR 1''
```

The normal query is no problem, as our MySQL statement will just select everything from customers that has a username equal to *timmy*.

However, the injection attack has actually made our query behave differently than we intended. By using a single quote (') they have ended the string part of our MySQL query

- username = ' '

and then added on to our WHERE statement with an OR clause of 1 (always true).

- username = ' ' **OR 1**

This OR clause of 1 will always be *true* and so **every single entry** in the "customers" table would be selected by this statement!

more serious sql injection attacks

Although the above example displayed a situation where an attacker could possibly get access to a lot of information they shouldn't have, the attacks can be a lot worse. For example an attacker could empty out a table by executing a *DELETE* statement.

MySQL & PHP Code:

```
$name_evil = "'; DELETE FROM customers WHERE 1 or username = '";

// our MySQL query builder really should check for injection

$query_evil = "SELECT * FROM customers WHERE username = '$name_evil'";

// the new evil injection query would include a DELETE statement

echo "Injection: " . $query_evil;
```

Display:

```
SELECT * FROM customers WHERE username = ' '; DELETE FROM customers WHERE 1 or username = ' '
```

If you were run this query, then the injected DELETE statement would completely empty your "customers" table. Now that you know this is a problem, how can you prevent it?

injection prevention - mysql_real_escape_string()

What *mysql_real_escape_string* does is take a string that is going to be used in a MySQL query and return the same string with all SQL Injection attempts safely escaped. Basically, it will replace those troublesome quotes(') a user might enter with a MySQL-safe substitute, an escaped quote \'.

Lets try out this function on our two previous injection attacks and see how it works.

MySQL & PHP Code:

```
//NOTE: you must be connected to the database to use this function!

// connect to MySQL

$name_bad = "' OR 1'";
```

```
$name_bad = mysql_real_escape_string($name_bad);

$query_bad = "SELECT * FROM customers WHERE username = '$name_bad'";

echo "Escaped Bad Injection: <br />" . $query_bad . "<br />";

$name_evil = "'; DELETE FROM customers WHERE 1 or username = '";

$name_evil = mysql_real_escape_string($name_evil);

$query_evil = "SELECT * FROM customers WHERE username = '$name_evil'";

echo "Escaped Evil Injection: <br />" . $query_evil;
```

Display:

```
Escaped Bad Injection:
SELECT * FROM customers WHERE username = '\' OR 1\''
Escaped Evil Injection:
SELECT * FROM customers WHERE username = '\'; DELETE FROM customers WHERE 1 or username
= \''
```

Notice that those evil quotes have been escaped with a backslash \, preventing the injection attack. Now all these queries will do is try to find a username that is just completely ridiculous:

- Bad: \' OR 1\'
- Evil: \'; DELETE FROM customers WHERE 1 or username = \'

And I don't think we have to worry about those silly usernames getting access to our MySQL database. So please do use the handy *mysql_real_escape_string()*function to help prevent SQL Injection attacks on your websites. You have no excuse not to use it after reading this lesson!

Cross-Site Request Forgery (CSRF)

Cross-Site request forgery is a client-side vulnerability that allows an attacker to make requests on the user's behalf. Although, most CSRF exploits require a user to be authenticated to the susceptible site to be successful, this is not always the case.

An Example of Cross-Site Request Forgery

A user (victim) opens their browser and logs on to their online banking application located at http://bank.com

After checking their balance they browse away from the site (without logging off) and start reading web pages about olives from Madagascar. One of these olive sites is owned by an attacker. The attacker's website has the following <img> tag:

```
<img src="http://bank.com/transfer.asp?to_acct=445544&amp;amount=1000">
```

When the victim's browser loads the malicious page that contains this img tag, the victims browser makes the transfer request/transfer.asp?to_acct=445544&amount=1000 to bank.com using the **authenticated** cookie from the earlier session. Upon making this request, the bank then transfers $1,000 from the victim's account to account 445544. The attacker has now successfully executed a cross-site request forgery attack against a user of bank.com

How Do You Prevent Cross-Site Request Forgery

Any **sensitive** request that is generated by the user should force the user to "re-authenticate." A simple example is that of change password functionality. You always want to verify the user knows the old password before changing their password, even if they are currently authenticated.

If you determine that "re-authentication" may be an inconvenience for the user or if all of your requests are considered sensitive then the application developers should include a random token that is unique to the user session. This token **should not** be present in the cookie, but rather as a hidden field in the HTML and then appended to the URL during any form submission.

When the attacker attempts to trick the users browser into making a request, the web application will look for this random token. The random token will not exist for the request, and the request will be denied. This prevents the CSRF attack from being successful.

Note: Having SSL does not protect your application from CSRF vulnerabilities.

References:

http://bretthard.in/2009/05/xss-cross-site-scripting/

http://www.tizag.com/mysqlTutorial/mysql-php-sql-injection.php