24 March 2013

SQL Injection

SQL injection is an interesting sort of software vulnerability. It is relatively common in websites, and I've actually found a few vulnerabilities myself that were based around SQL injection.

Let me start by describing SQL. SQL, of Structured Query Language, is a language used to store and retrieve data in databases. Each database can have multiple named tables. Tables have multiple named columns. Tables can them have entries (rows) inserted that use the columns as their definition.

For a website, I might have a table called "users" with columns "username", "email", and "password". "password" would likely not be stored in plain text, but rather in a hash. for more information about hashes, refer to MP2 page 13.

To insert a new user into the "users" table, I'll write:

```
1     INSERT INTO 'users' VALUES('george', 'george@gmail.com', 'a5de69f4');
```

When a website gets a hit for the "create user" page, it might run some code like this:

```
1     userName = userInput("username")
2     passWord = userInput("password").hash()
3     email = userInput("email")
4
5     query = "INSERT INTO 'users' VALUES('" + userName + "', '" + email + "',
'" + password + "');"
6     mysql_execute(query)
```

This would take user input and shove it into the database. Remember this code example, because it is important.

To obtain data from a database, I write:

```
1     SELECT * FROM 'users' WHERE username='george';
```

Which will return with a row that contains George's username, email, and hashed password. I can use similar code as before to dynamically form any SQL query.

Another thing I should mention now is the concept of SQL 'comments'. When the combination of characters "--" is located somewhere in an SQL statement, everything after the "--" is not executed. This is useful for writing comments in SQL without throwing an error. This:

```
1     SELECT * FROM 'users'; hi I like tacos
```

would give an error, because "hi I like tacos" is not valid SQL. This:

```
1     SELECT * FROM 'users'; -- hi I like tacos
```

would not, because everything after -- is ignored.

An attacker can use these concepts to his advantage.

Imagine a "log-in" procedure in your code that goes like this:

```
1     userName = userInput("username")
2     passWord = userInput("password")
3
4     query = "SELECT * FROM 'users' WHERE username='" + userName + "' AND
password='" + passWord + "';"
5     mysql_execute(query)
```

The code just inserts the user's username input and password input into the SQL statement. Say I wanted to log in with username "george" and a password hash of "ef6dbe3". The statement would look something like this:

```
1     SELECT * FROM 'users' WHERE username='george' AND password='ef6dbe3';
```

If the username exists in the databases, but the password is WRONG, the SQL statement will not return a user row- there is no row that exists in which the username is 'george' and the password is some random, incorrect hash.

Think about this. What if an attacker were to enter this:

```
george';--
```

into the username field, and leave the password field blank? Go on, think about it.

The resulting statement is devious:

```
1     SELECT * FROM 'users' WHERE username='george';--' AND password='';
```

After the SQL comment is applied, that will evaluate to:

```
1     SELECT * FROM 'users' WHERE username='george';
```

Now you don't even need a working password to log in! If you can sneak it by the website, you can log into anyone's account. Worse, imagine this input:

```
'; DROP TABLE 'users';--
```

Which evaluates to:

```
1     SELECT * FROM 'users' WHERE username=''; DROP TABLE 'users';
```

Heh... there goes your entire table of user accounts.

Prevention of SQL injection attacks is easy. First, databases generally support individual database access accounts with different permissions. Create a user without the "DROP TABLE" permission, and at least your tables will be okay.

It's also possible to "sanitize" your input strings to safely be added into SQL without the possibility of code execution. That's not the best solution, but it's rather popular in basic websites.

As much as I'd love to pretend I'd like writing, I've hit the twenty-eighth page. Goodbye until the fourth marking period.