```
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 database, 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.