Introduction to Software Security
Chapter 3.8.1: SQL Injection Attacks

Loren Kohnfelder
loren.kohnfelder@gmail.com

Elisa Heymann
elisa@cs.wisc.edu

Barton P. Miller
bart@cs.wisc.edu

Revision 2.0, January 2022.

Objectives

- Understand how SQL injection attacks work.
- Learn how to recognize weaknesses related to SQL injections.
- Learn how to mitigate them, especially the use of *prepared statements*.

Background

SQL injections are the classic type of injection attack, still occurring year after year. If you are not familiar with the basic idea of an injection attack, then you should review Chapter 3.8: Introduction to Injection Attacks, before continuing with this section.

The Structured Query Language, invented well over forty years ago, is one of the longest lived software innovations still very much in widespread use today. The original design was created almost a generation before the World Wide Web and hence could not have anticipated the kind of potential abuse that leads to injection attacks. Today it is commonplace to use SQL in serving web pages where the public internet is a rich source of untrusted data — and the confluence of SQL statements with user submitted forms or other internet data is the perfect recipe to make these flaws rampant.

We assume readers have a basic familiarity with SQL. If not, there are many good introductions and tutorials available, for example, see:

- w3schools.com
- https://www.w3resource.com/sql/tutorials.php
- http://www.sqltutorial.org/

In general, relational databases provide SQL statement access as standard; there are many different subsets and variants of SQL but the principles of injection are the same.

Signs of vulnerability to beware include:

- Use of a database management system (DBMS) that is typically accessed via SQL.
- The commands being sent to the SQL interpreter are constructed by the program while it is executing. In other words, the commands are not constant strings.
- At least part of the data being used to construct the SQL query strings comes from user input. In other words, at least part of the input comes from the *attack surface*.

- The program does not correctly prevent the user input from changing the way that the programmer intended the SQL query to be interpreted.

We will use examples of SQL injections in Perl and Java, but the same principles apply across languages.

SQL injection Perl Example

An example of a SQL injection attack is the best way to understand the fundamental problem. Consider the following Perl statement that constructs a SQL query to look up a table entry for a given user. We assume that code has assigned `$dbh` with a database handle, so the `do` function performs the SQL query putting the results into `$sth`.

```
        $sth = $dbh->do("select * from t where u = '$u'");
```

Now, let's choose a value for the user name ($u) input that will cause the SQL interpreter to do something different than the programmer intended:

```
        $u = " '; drop table t --";
```

The query statement contains a bug that enables the vulnerability. A SQL `select` statement is constructed to perform the look-up using a statement template string and inserting the value of `$u` with string catenation with single quote characters on either side of the user supplied string.

The programmer of this code expected the user name, `$u`, to be an alphanumeric string. In this case, the query works fine, selecting the table row for the corresponding username entry. However, with the deviously chosen string for `$u`, the string catenation results in two SQL statements to be executed.

```
        select * from t where u = ' '; drop table t --'
```

The `select` statement now looks up rows with column `u` containing a single space character (presumably there are none as this would not be a valid username). Semicolon is a statement separator, so the following `drop` statement is executed next, deleting the table `t`. The double dash marks that remainder of the line as being a comment so the final single quote is simply ignored. Thus, by carefully constructing an unexpected username string, the attacker has managed to inject an entirely new SQL statement to be performed — in this case deleting the entire table in the database.

We see three basic techniques used in this attack:

1. Introducing a quote character into the input to change the meaning of a comparison.
2. Introducing a statement separator (semicolon) so that the attacker can introduce an additional SQL statement
3. Using the double dash comment to prevent extra characters at the end of the line from causing a syntax error and preventing the statement from being executed. In this case, it prevented the processing of the final single-quote character.

SQL injection mitigation strategies

There are several ways of fixing the problem: input validation, careful quoting and escaping, and, the best solution, using prepared statements instead of string manipulation to create dynamic SQL commands. **The**

best practice is to always use prepared statements, but to understand why that is we cover other approaches first. For simple cases in certain situations, these other methods can be made to work reliably, however there always exists the danger of underestimating the potential for attack — that is, that a clever attacker will think of something the programmer did not anticipate — and since prepared statements are quite easy to use there really is no need to ever take such a chance.

Input validation

We could add code to restrict the username (`$u`) to only contain alphanumeric characters as we assume valid username should be, so the attack string would fail to pass this test and never get executed. This works, but only if you can restrict the possible characters sufficiently to be safe without possibly losing functionality (such as if O'Brien is a valid user name). If we expand our names to include full international character sets, such as Unicode, this technique becomes even more difficult.

Quoting and escaping

Command languages such as SQL include syntax to include data within the body of the command, such as text strings which are usually enclosed in quotes. However, in the example above, this is subject to manipulation if the attacker gets to provide a string that uses a quote to prematurely close the quoted string, thereby allowing completely different commands to be injected masquerading within the malicious string.

The creators of SQL anticipated the need to include a quote character within a quoted string — but probably did not anticipate the threat of injection in those early days of computing — and provided that two consecutive quotes (to disambiguate it from the closing quote) represent one quote character within a quoted string. Using this feature, another way to block possible injection would be to transform strings containing quotes, replacing each single quote with two single quote characters which the SQL statement parser recognizes as representing one. Unfortunately, just handling quotes is not enough: it turns out there are other tricky strings that would get around this (e.g. involving the use of backslash escape character). With considerable effort and testing it is possible to get all this figured out, but the rules for SQL statement string literals are not simple and with multiple character codes it gets rather complicated and cannot be recommended.

Prepared statements

Fortunately, the work has already been done to develop a mechanism to handle these tricky strings when you use prepared statements, and this mechanism is quite easy to use. Prepared statements are SQL statement templates that have placeholders for dynamic values to be inserted in such a way that system ensures safe parsing, preventing statement injection.

```
     $sth = $dbh->do("select * from t where u = ?", $u);
```

Example 1: Perl code using a prepared statement

Even with the malicious attack string value, this will result in the following single SQL statement:

```
     select * from t where u = ' ''; drop table t --'
```

For clarity in the above statement, the string within the single quotes is underlined to show that this is just one SELECT statement with the entire attack string considered as a value for column `u`, rather than being two statements. This is exactly what is expected, and the potential attack is foiled.

SQL injection Java Examples

Consider another SQL example in Java, using the JDBC library: this example is a simplified version of real world code that we identified as a serious vulnerability during a security assessment.

The method `Login` accepts string username and password parameters, looks up the correct password in a database, and then is supposed to return true only if the correct password for the named authorized user account matches. To keep the example simple, this works with plaintext passwords stored in a database — a terrible design for secure password credential management — in order to focus on the SQL injection threat specifically.

```
boolean Login(String user, String pwd) {
  boolean loggedIn = false;
  conn = pool.getConnection( );
  stmt = conn.createStatement();
  rs = stmt.executeQuery("SELECT * FROM members"
    + "WHERE u='" + user
    + "' AND p='" + pwd + "'");
  if (rs.next())
    loggedIn = true;
  return loggedIn;
}
```

Example 2: Java code subject to SQL injection attack

When `Login` is invoked, it pessimistically sets the variable `loggedIn` to false, as is good practice with security, assuming the less empowered or safer case. The next few lines establish a connection to the database and create a SQL statement from a string constructed from the string values of the parameters `user` and `pwd`.

This example is different from the previous Perl code in that it uses the SQL statement to find a row in the table matching both the username and password, rather than look up only by the user name and then, presumably, have user code that compares the value in the password field (attribute) of the returned tuple.

For the `Login` function in Example 2, the normal use case might result in a SQL statement as follows:

```
SELECT * FROM members WHERE u='admin' AND p='secret'
```

If indeed the password for the admin account is “secret”, then that one row will be returned as the record set `rs`. The `if` statement invokes the `next()` method on the record set `rs` evaluating true if there is a next row — in this case, if there is at least one row returned — and setting `loggedIn` true. Finally, the variable `loggedIn` is returned as the verdict on accepting the given login credentials.

Now we consider how a clever attacker might take advantage of this code. Since the SQL statement is constructed from strings coming from the user, an attacker has an obvious opportunity for injection. Consider this call to `Login`:

```
Login(“admin”, “' OR 'x'='x”);
```

Of course the password that we provide is not the admin password, so let’s examine the resulting SQL statement carefully to see what will happen:

```
SELECT * FROM members WHERE u='admin' AND p=' ' OR 'x'='x'
```

SQL (as do most languages) uses operator precedence to evaluate expressions, so this is equivalent to:[1]

```
SELECT * FROM members WHERE ((u='admin' AND p=' ') OR 'x'='x')
```

Even though the user name and password are incorrect, the latter part of the OR clause is an identity, in other words it evaluates to true for every row of the table. Assuming any accounts exist at all in the table, the resulting record set will be nonempty and the `if` statement will succeed, returning true and allowing the login to proceed just as if it had provided a valid password for the admin account. Simply by knowing or guessing the username of any valid account, the attacker can log into it. By providing the admin account, the attacker might get the highest level of access to the system.

Note that in addition to allowing SQL injection, this code is rather sloppy in testing the record set to have *at least* one result when a tighter condition of there being *exactly* one would be better. More carefully written code that required exactly one result would have defeated this particular attack. For any database of users, we expect to see only one record per user. Nonetheless, fixing just this problem, but not eliminating the SQL injection vulnerability, would still leave this code vulnerable to attack with a different, more carefully designed set of parameters.

Using prepared statements, the fix is straightforward, replacing the *ad hoc* command string building.

```
boolean Login(String user, String pwd) {
     boolean loggedIn = false;
     conn = pool.getConnection( );
     PreparedStatement pstmt = conn.prepareStatement(
         "SELECT * FROM members WHERE u = ? AND p = ?");
     pstmt.setString(1, user);
     pstmt.setString(2, pwd);
     ResultSet rs = pstmt.executeQuery( );
     if (rs.next())
          loggedIn = true;
     return loggedIn;
}
```

[1] As a good programming practice in any languages, always feel free to add parentheses to remove doubt and make precedence explicit. There are just enough variations between languages that it is easy to make mistakes.

Example 3: Java code using prepared statement

Instead of using a string, `pstmt` is now a template for a prepared statement, where question marks are placeholders for values to be provided in code. The next two statements use the `setString` method to provide values where the first argument is the ordinal position of the placeholder being assigned (user name in the first, password in the second). Executing the constructed prepared statement does just what we expect, where the data values for the user name and password are never misinterpreted as SQL.

The equivalent SQL statement with data values included as part of the query (where backslash escaping handles the single quote characters in the provided password to preserve the integrity of the SQL command) would look like:

```
    SELECT * FROM members WHERE u='admin' AND p=' \' OR \'x\'=\'x'
```

Since the admin password is presumably different than the odd value provided with lots of quotes, the code is no longer subject to injection and does the right thing in the face of malicious inputs.

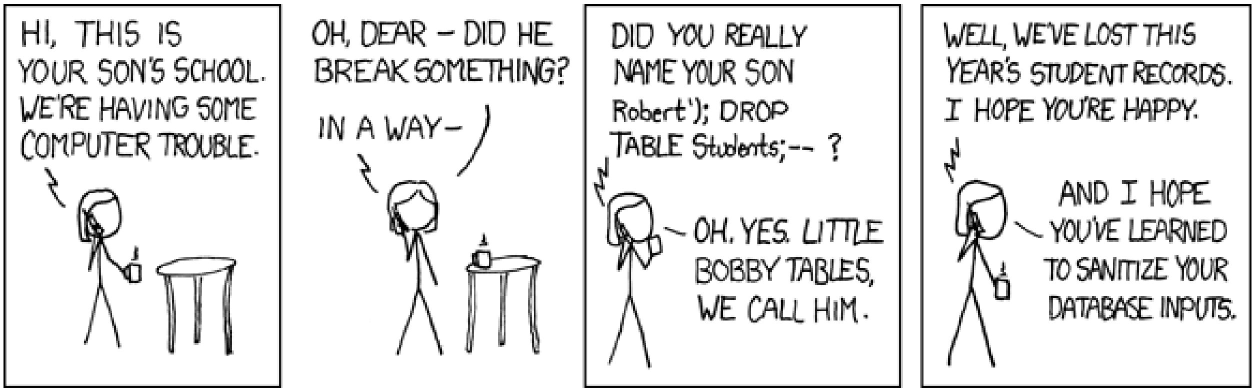
Summary

Injection attacks are always a concern when commands are dynamically synthesized using untrusted inputs. SQL injection is the classic most common case, but the same kind of flaw can arise with shell commands, XML, or constructed customized script.

For SQL statements, prepared statements are the best solution and, if used correctly, eliminate the problem of injections completely. Other types of injections tend not to have such simple answers. Strict validation of untrusted input can avoid injection if it is not too restrictive.

Metacharacters such as quotes and escape characters are typically both the means of attack and potentially also the mitigation, but writing *ad hoc* code to validate inputs or transform potentially dangerous string inputs into safe ones is both a lot of work and also risky to get exactly right.

With an appreciation of the threat of SQL injection, enjoy the classic XKCD comic about a very farsighted parent going to extreme lengths to teach the local school district IT staff a hard lesson in software security. Can you guess what they named their daughter? (Hover to reveal.)


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

https://xkcd.com/327/

Exercises

1. Modify Example 2 to require exactly one record in the result set (as described in the text). Design a different SQL injection attack on the modified code to successfully log in without knowing the actual password.
2. Create an example of code vulnerable to SQL injection and sets of inputs that exploit it.
3. Rewrite the code in Exercise 2 to use prepared statements and verify the fix.
4. Rewrite the code in Exercise 2 to fix the vulnerability without using prepared statements and try attacking it (or challenge a friend to break it).
5. This chapter explained how to use the **`PreparedStatement`** class to safely compose SQL statements that include untrusted user data. In the language of your choice, write and test code that constructs safe SQL statements using these **`PreparedStatement`**s.