Chapter 16
SQL Injection Attacks

Revision 3.1, December 2025.

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*.

16.1 Background

SQL injections are probably the most well-known type of injection attack and they keep occurring year after year. In this chapter, we will learn about the cause of these attacks and the surprisingly easy way to prevent them.

The Structured Query Language (SQL)[1] was introduced in the 1970s to apply the ideas of relation algebra and tuple calculus to databases. Thus, it was created on a firm theoretical foundation. It became an ANSI standard in 1986 and an ISO standard in 1987. Today, it is a foundation for building commerce and other types of services. The original design was created almost a generation before the World Wide Web and so could not have anticipated the kind of potential abuses today that lead to injection attacks. It is commonplace to use SQL in producing web pages. The confluence of SQL statements with user submitted forms or other types of remote requests is the perfect recipe to make SQL attacks extremely common.

In general, relational databases use some form of the SQL query language as their standard interface, though there are many different subsets and variants of SQL.

SQL injection attacks are possible when three conditions hold:

1. The commands being sent to a SQL interpreter are constructed by the program while it is executing. In other words, the commands are not constant strings.
2. The SQL query string is constructed data that originates from user input. In other words, at least part of the input comes from the attack surface.

[1] `https://www.w3schools.com/sql/`

3. The program does not correctly check or sanitize the user input to either reject the query or prevent it from changing the way that the programmer intended the query to be interpreted.

In this chapter, we will use examples from Perl and Java, though the same principles apply across all languages.

16.2 SQL injection Perl Example

We start with a simple example of a SQL injection attack 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'");
```

If this query is executed with `$u` set to “bart”, the resulting query will be:

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

Notice that the user name, which presumably came from user input such as from a web frame, resulted in a reasonable query. However, we can choose a value for the user name that will cause the SQL interpreter to do something different than the programmer intended:

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

The programmer of this code expected the user name to be a simple alphanumeric string. In this case though, 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 where the attributed `u` contains a single space character (presumably there are none as this would not be a valid username). The semicolon is a statement separator, so the following `drop` statement is executed next, deleting the entire table `t`. Presumably, this is an unexpected and unwelcome result.

The double dash marks (“`--`”) indicate that remainder of the line is 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 metacharacter, the quotation mark, 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.

The fundamental problem of introducing user values into a query in this way is that the user values are actually parsing as part of the query.

16.3 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. To better understand why prepared statements are important, we will first cover those other approaches.

For simple cases in certain situations, these other methods can be made to work, 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 so easy to use, there really is no need to ever take such a chance.

16.3.1 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 set of characters sufficiently to be safe without losing functionality. For example, there are common names, such as O'Brien, that contain punctuation characters. And really strange names, such as that given to a billionaire's son, "X Æ A-12". If we expand our names to include full international character sets, such as Unicode, this technique becomes even more difficult.

16.3.2 Quoting and Escaping

SQL can include text strings, which are usually enclosed in quotes. However, in the example above, the attacker can manipulate the meaning of the command if they can provide a string that uses a quote to prematurely close the quoted string.

Languages that process strings usually include an "escape" character to allow special characters, especially quote characters, to be part of the contents of the string. However, they did not anticipate that an attacker could use this feature to change the meaning of a command. With considerable effort and testing it is possible to figure all the details for strings, but the rules for SQL string literals are not simple so trying to understand these rules and detect dangerous cases is overly complicated and definitely not be recommended.

16.3.3 Prepared Statements

Fortunately, there is a solution in SQL that allows you to avoid all these tricky cases. This solution is called *prepared statements*, and is quite easy to use. Prepared statements are simply placeholders in SQL where a value from a variable can be inserted into the query. In Figure *1*, we see a query that has the same functionality as our example from Section 16.2 , except we notice two things. First, the string that will be processed by SQL is a constant that contains no user-provided data. Second, the string contains placeholder in the form of "?" characters. Each one of those placeholders references additional parameters that are passed to the SQL interpreter. In this case, we see an additional parameter of `$u`.

The key idea with prepared statements is that the user data (`$u` in this case) is never parsed by the SQL interpreter. It is simply used as the value for comparison with database entries when the query is being processed. So, there is no path from the user input to the query because the query contains no user input.

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

Figure 1: Perl Code using a Prepared Statement

Even with the malicious attack string value that we used earlier will have no affect. This query would simply look for u fields in the database that have the value "`'; drop table t --`". This would be a strange user name and unlikely to match any valid user name.

The best practice is to **always** use prepared statements. **Always**.

16.4 SQL Injection Java Examples

We will now look an example of a SQL injection attack on real system that we evaluated. The example shown in Figure 2 is written in Java using a JDBC library[2]. While we have slightly simplified the code for this example, it contains all the key elements of the real vulnerable code.

The `Login` method accepts user name and password string parameters, looks for a record in the database that matches the user name and password, and

[2] `https://en.wikipedia.org/wiki/Java_Database_Connectivity`

then is supposed to return true only if the there is such a record in the database. To keep the example simple, this works with plaintext passwords stored in a database – a terrible design for password credential management – to focus on the SQL injection threat specifically.

```
01 boolean Login(String user, String pwd) {
02   boolean loggedIn = false;
03   conn = pool.getConnection( );
04   stmt = conn.createStatement();
05   rs = stmt.executeQuery("SELECT * FROM members"
06     + "WHERE u='" + user + "' AND p='" + pwd + "'");
07   if (rs.next())
08     loggedIn = true;
09  return loggedIn;
10 }
```

Figure 2: Java Code that is Subject to a SQL Injection Attack

When `Login` is invoked, it sets the variable `loggedIn` to false on line 2, as is good practice with security operations. The default should be a denial of access and only after a success matching of credentials will the access be grant. (Recall the example in Figure 3 in Chapter 13 (Exceptions) where the programmer did the opposite and that resulted in allowing an exploit to occur.)

Lines 3 and 4 establish a connection to the database and create an empty SQL query. The main action happens on lines 5 and 6 where a query string constructed from the values passed as parameters `user` and `pwd`.

The normal and expect use of the `Login` method in Figure 2 would result in a SQL query that looks something like:

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

If the password for the admin account was "secret", then the one row in the database will be returned as the set of records `rs`. The `if` statement on line 7 calls the `next` method on the record set `rs`, returning true if there is a next row in the set. In other words, the call to `next` will return true if there is *at least one* record in the record set. This means that there was at least row in the database that matched the user name and password. If `next` returns true, then `loggedIn` is set to true and that value is returned, meaning that login should be allowed.

Now we consider how a clever attacker might take advantage of this code. This attack will, in fact, expose a few different problems in the code for `Login`. 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 us examine the resulting SQL statement to see what will happen:

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

In SQL (and in most languages[3]) AND has a higher precedence than OR, so gets evaluated first. For clarity we rewrite this query with parenthesis to explicitly specify the order of operations[4]:

```
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 a tautology, in other words it evaluates to true for every row of the table. The resulting record set will contain every row in the database table. So, the `if` statement on line 7 will succeed, returning true and allowing the login to proceed just as if it the method had been 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 will get the highest level of access to the system.

Note that in addition to allowing SQL injection, this code is written rather sloppily in that it tests that the record set has *at least* one result. It would be more precise and safe to get that it contained *exactly* one record. The more carefully written code 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, might still leave this code vulnerable to attack with a different, more carefully designed set of parameters.

Of course, using prepared statements and replacing the *ad hoc* command string building is simple and much safer.

Instead of using a string, `pstmt` is now a constant string where question marks are placeholders for values to be provided in code. The next two statements use the `setString` method to provide the values for the user name and password. As in the previous example, no user input is used in the construction of the SQL query so there can be no injection attack.

[3] Notable exceptions to this precedence rule are the bash and PowerShell shells, where AND (&&) and OR (||) have equal presence and evaluated left to right,

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

When this code is run with the attack query, the strange password will not match any in the database, so no rows in the table will be returned.

```
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;
}
```

Figure 3: Java Code with the SQL Injection Attack Prevent with Prepared Statements

16.5 Summary

Injection attacks are always a concern when commands are constructed using untrusted user inputs. This is a frustrating situation because in SQL, with prepared statements, we have an easy and strong solution. Prepared statements have been a part of SQL for more than 20 years, so it is difficult to understand why they are still so prevalent.

Do not let this happen to you!

Other types of injection attacks do not have such simple answers, as we will see in the following chapters.

Now, with your knowledge now of SQL injection attacks, you can enjoy the classic Bobby Droptables XKCD comic.

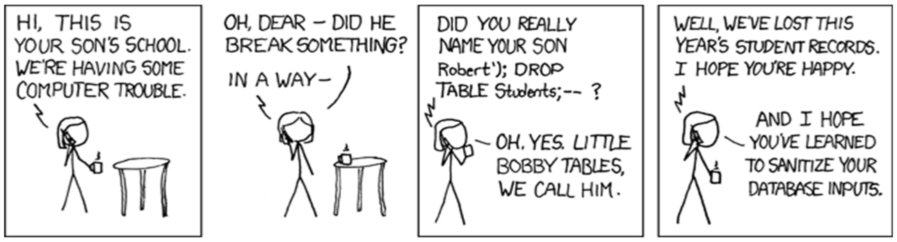
HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE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

Figure 4: Classic XKCD Bobby Droptables Comic Strip[5]

[5] https://xkcd.com/327/. Used by permission of the author.

16.6 Exercises

1. Modify the code in Figure 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 a SQL injection and show an example input set that can exploit it.
3. Rewrite the code in Exercise 2 to use prepared statements and verify the fix.
4. This chapter explains 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 `PreparedStatements`.