Chapter 11

SQL Injection Attack

In real-world web applications, data are usually stored in databases. To save data to or get data from a database, a web application need to construct a SQL statement, and send it to the database, which will execute the SQL statement, and return the results back to the web application. Usually, SQL statements contain the data provided by users; if a SQL statement is constructed inappropriately, users may be able to inject code into the SQL statement, causing the database to execute the code. This type of vulnerability is called SQL Injection, which is one of the most common mistakes in web applications. In this chapter, we discuss how SQL injection attacks work and how to defend against this type of attack.

Contents

11.1 A Brief Tutorial of SQL

To fully understand how SQL injection works, we need to learn a little bit about SQL (Structured Query Language), which is a special-purpose domain-specific language used in programming and is designed for managing data held in a relational database management system [Wikipedia, 2017v]. In this section, we give a brief tutorial on SQL. More comprehensive coverage of SQL can be found from Wikipedia and other online resources.

11.1.1 Log in to MySQL

We will use the MySQL database, which is an open-source relational database management system. We have already set up the MySQL server in our pre-built Ubuntu12.04 virtual machine. The server will run automatically when the operating system boots up. We can log in to the database system using the mysql program.

```
$ mysql -uroot -pseedubuntu
Welcome to the MySQL monitor.
...
mysql>
```

The options -u and -p specify the login name and password, respectively. In our pre-built virtual machine, the password for the root account is seedubuntu. Once the login is successful, we can see the mysql> prompt, from where we can type SQL commands.

11.1.2 Create a Database

Inside MySQL, we can create multiple databases. The SHOW DATABASES command can be used to list all the existing databases. Let us create a new database called dbtest. We can achieve that using the CREATE DATABASE command. SQL commands are not case sensitive, but we always capitalize commands, so they are clearly separated from non-commands in lowercase.

```
mysql> SHOW DATABASES;
......
mysql> CREATE DATABASE dbtest;
```

11.1.3 CREATE a Table

We have just created a database called dbtest, which is empty at this point. A relational database organizes its data using tables. A database can have multiple tables. Let us create a new table called employee, which is used to hold employee data. The CREATE TABLE statement is used to create a table. The following code creates a table called employee with seven attributes (i.e. columns). It should be noted that since there may be multiple databases in the system, we need to let the database system know which database we are going to use. That is achieved using the USE command. Once a table is created, we can use the DESCRIBE (or DESC in short) command to display the structure of the table.

```
mysql> USE dbtest
mysql> CREATE TABLE employee (
  ID       INT (6) NOT NULL AUTO_INCREMENT,
  Name     VARCHAR (30) NOT NULL,
  EID      VARCHAR (7) NOT NULL,
  Password VARCHAR (60),
  Salary   INT (10),
  SSN      VARCHAR (11),
  PRIMARY KEY (ID)
);
mysql> DESCRIBE employee;
+----------+-------------+------+-----+---------+----------------+
| Field    | Type        | Null | Key | Default | Extra          |
+----------+-------------+------+-----+---------+----------------+
| ID       | int(6)      | NO   | PRI | NULL    | auto_increment |
| Name     | varchar(30) | NO   |     | NULL    |                |
| EID      | varchar(30) | NO   |     | NULL    |                |
| Password | varchar(60) | YES  |     | NULL    |                |
| Salary   | int(10)     | YES  |     | NULL    |                |
| SSN      | varchar(11) | YES  |     | NULL    |                |
+----------+-------------+------+-----+---------+----------------+
```

Table columns are defined inside the parentheses after the table name. Each column definition starts with its name, followed by the data type. The number associated with the data type specifies the maximum length for the data in the column. Constraint can also be specified for each column. For example, `NOT NULL` is a constraint showing that the corresponding field cannot be NULL for any row.

Let us use the `ID` column as an example to explain the syntax. The data type is integer, and its value can have at most 6 digits. We set two constraint for `ID`. First, the value of `ID` cannot be NULL, because we plan to use it as the primary key of the table. Second, the value of `ID` will automatically increment every time we insert a new row; `Auto_Increment` allows a unique number to be generated when a new record is inserted into a table.

11.1.4 INSERT a Row

We can use the `INSERT INTO` statement to insert a new record into a table. In the following example, we insert a record into the `employee` table. We did not specify the value for the `ID` column, as it will be automatically set by the database.

```
mysql> INSERT INTO employee (Name, EID, Password, Salary, SSN)
       VALUES ('Ryan Smith', 'EID5000', 'paswd123', 80000,
               '555-55-5555');
```

11.1.5 The SELECT Statement

The `SELECT` statement is the most common operation on databases; it retrieves information from a database. The first statement in the following example asks the database for all its records, including all the columns, while the second statement only asks for the `Name`, `EID`, and `Salary` columns.

```
mysql> SELECT * FROM employee;
+----+---------+---------+----------+--------+-------------+
| ID | Name    | EID     | Password | Salary | SSN         |
+----+---------+---------+----------+--------+-------------+
|  1 | Alice   | EID5000 | paswd123 |  80000 | 555-55-5555 |
|  2 | Bob     | EID5001 | paswd123 |  80000 | 555-66-5555 |
|  3 | Charlie | EID5002 | paswd123 |  80000 | 555-77-5555 |
|  4 | David   | EID5003 | paswd123 |  80000 | 555-88-5555 |
+----+---------+---------+----------+--------+-------------+
mysql> SELECT Name, EID, Salary FROM employee;
+---------+---------+--------+
| Name    | EID     | Salary |
+---------+---------+--------+
| Alice   | EID5000 |  80000 |
| Bob     | EID5001 |  80000 |
| Charlie | EID5002 |  80000 |
| David   | EID5003 |  80000 |
+---------+---------+--------+
```

11.1.6 WHERE Clause

In practice, it is uncommon for a SQL query to retrieve all the records in a database, because a real-world database may easily contain thousands or millions of records. A typical query sets a condition so the query is only conducted on the records that satisfy the condition. WHERE clause is used to set conditions for several types of SQL statements, including SELECT, UPDATE, DELETE, etc. WHERE clause takes the following general form:

```
mysql> SQL Statement
       WHERE predicate;
```

The above SQL statement only affects the rows for which the predicate in the WHERE clause is True. Rows for which the predicate evaluates to False or Unknown (NULL) are not affected. The predicate is a logical expression; multiple predicates can be combined using the keywords AND and OR. In the following examples, the first query returns a record that has EID5001 in the EID field; the second query returns the records that satisfy either EID='EID5001' or Name='David'.

```
mysql> SELECT * FROM employee WHERE EID='EID5001';
+----+------+---------+----------+--------+-------------+
| ID | Name | EID     | Password | Salary | SSN         |
+----+------+---------+----------+--------+-------------+
|  2 | Bob  | EID5001 | paswd123 |  80000 | 555-66-5555 |
+----+------+---------+----------+--------+-------------+

mysql> SELECT * FROM employee WHERE EID='EID5001' OR Name='David';
+----+-------+---------+----------+--------+-------------+
| ID | Name  | EID     | Password | Salary | SSN         |
+----+-------+---------+----------+--------+-------------+
|  2 | Bob   | EID5001 | paswd123 |  80000 | 555-66-5555 |
|  4 | David | EID5003 | paswd123 |  80000 | 555-88-5555 |
+----+-------+---------+----------+--------+-------------+
```

If the condition is always `True`, then all the rows are affected by the SQL statement. For example, if we use `1=1` as the predicate in a `SELECT` statement, all the records will be returned. See the following example.

```
mysql> SELECT * FROM employee WHERE 1=1;
+----+---------+---------+----------+--------+-------------+
| ID | Name    | EID     | Password | Salary | SSN         |
+----+---------+---------+----------+--------+-------------+
|  1 | Alice   | EID5000 | paswd123 |  80000 | 555-55-5555 |
|  2 | Bob     | EID5001 | paswd123 |  80000 | 555-66-5555 |
|  3 | Charlie | EID5002 | paswd123 |  80000 | 555-77-5555 |
|  4 | David   | EID5003 | paswd123 |  80000 | 555-88-5555 |
+----+---------+---------+----------+--------+-------------+
```

This `1=1` predicate looks quite useless in real queries; it will become useful in the SQL injection attack. Therefore, it is important to understand the effect of such a "useless" predicate.

11.1.7 UPDATE SQL Statement

To modify an existing record, we can use the `UPDATE` statement. For example, we can use the following statement to set Bob's salary to 82,000.

```
mysql> UPDATE employee SET Salary=82000 WHERE Name='Bob';
mysql> SELECT * FROM employee WHERE Name='Bob';
+----+------+---------+----------+--------+-------------+
| ID | Name | EID     | Password | Salary | SSN         |
+----+------+---------+----------+--------+-------------+
|  2 | Bob  | EID5001 | paswd123 |  82000 | 555-66-5555 |
+----+------+---------+----------+--------+-------------+
```

11.1.8 Comments in SQL Statements

Comments can be placed in SQL statements. MySQL supports three comment styles:

- Text from the # character to the end of a line is treated as comment.
- Text from `--␣` to the end of a line is treated as comment. It should be noted that this comment style requires the second dash to be followed by at least one whitespace or control character (such as a space, tab, etc.).
- Similar to the C language, text between `/*` and `*/` is considered as comment. Unlike the previous two styles, this style allows comment to be inserted into the middle of a SQL statement, and comment can span multiple lines.

We show an example for each of the comment styles in the following. In a SQL injection attack, the first style (the # style) is the most convenient one to use.

```
mysql> SELECT * FROM employee;   # Comment to the end of line
mysql> SELECT * FROM employee;   -- Comment to the end of line
mysql> SELECT * FROM /* In-line comment */ employee;
```

11.2 Interacting with Database in Web Application

A typical web application consists of three major components: web browser, web application server, and database. Browser is on the client side; its primary function is to get content from the web server, present the content to the user, interact with the user, and get the user inputs. Web application servers are responsible for generating and delivering content to the browser; they usually rely on an independent database server for data management. Browsers communicate with web servers using the Hypertext Transfer Protocol (HTTP), while web servers interact with databases using database languages, such as SQL. Figure 11.1 illustrates the architecture of a typical web application.

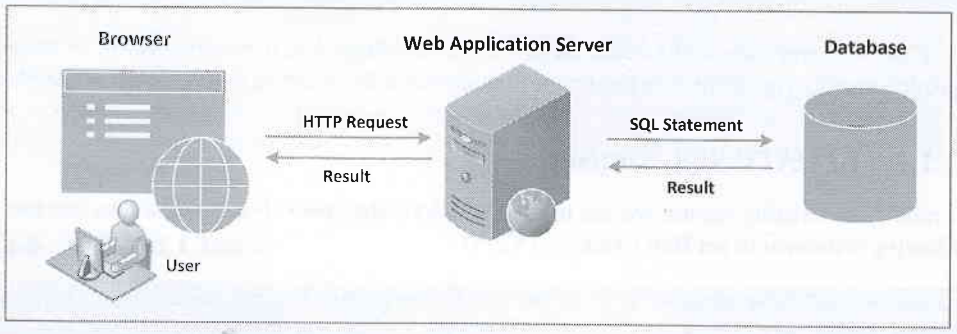
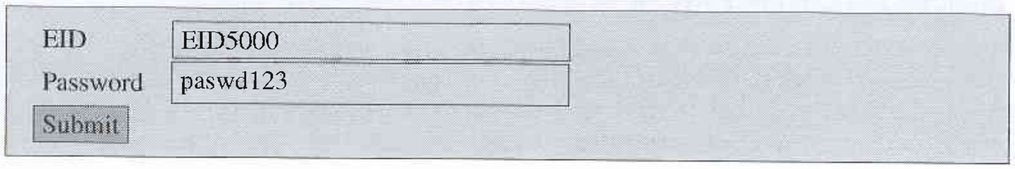


Figure 11.1: Web Architecture

SQL injection attacks can cause damages to the database, but from Figure 11.1, we can see that users do not interact with the database directly, so it seems that they pose no threat to the database. The culprit is the web application server, which provides a channel for users' data to reach the database. If the channel is not implemented properly, malicious users can attack the database via the channel. To understand how this channel works, we examine a sample web application program written in PHP, and see how such an attack surface is introduced.

11.2.1 Getting Data from User

As shown in Figure 11.1, browsers get inputs from users, and then communicate with the web application server using HTTP requests. User inputs are attached to HTTP requests. Depending on whether it is an GET or POST request, the ways how data are attached to HTTP requests are different. The following example shows a form where users can type their data; once the `submit` button is clicked, a corresponding HTTP request will be sent out, with the data attached.



EID	EID5000
Password	paswd123

The HTML source of the above form is shown in the following:

```
<form action="getdata.php" method="get">
  EID:      <input type="text" name="EID"><br>
  Password: <input type="text" name="Password"><br>
            <input type="submit" value="Submit">
</form>
```

When the user clicks the `Submit` button, an HTTP request will be sent out to the following URL:

```
http://www.example.com/getdata.php?EID=EID5000&Password=paswd123
```

The above HTTP request is a GET request, because the `method` field in the HTML code specifies the `get` type. In GET requests, parameters are attached after the question mark (?) in the URL. The above request passes two parameters: `EID` and `Password`. Each parameter is a name=value pair, with the name coming from the form entry's `name` attribute, and the value coming from whatever is typed into the form field. Parameters are separated by an ampersand (`&`) character. For the sake of simplicity, in the above example, we use the unsafe HTTP protocol, instead of the safe HTTPS protocol, to send the password. If we switch to HTTPS, the way how the parameters are sent is similar, except that the communication channel is encrypted.

Once the request reaches the target PHP script (e.g. `getdata.php` in the above example), the parameters inside the HTTP request will be saved to an array `$_GET` or `$_POST`, depending on the type of the HTTP request. The following example shows how a PHP script gets the user data from the `$_GET` array.

```
<?php
   $eid = $_GET['EID'];
   $pwd = $_GET['Password'];
   echo "EID: $eid --- Password: $pwd\n";
?>
```

11.2.2 Getting Data From Database

Web applications usually store their data in databases. After they get an input from the user, they often need to fetch additional data from the database, or store new information in the database. In the previous example, once a user provides his/her EID and password to the server-side script `getdata.php`, the script needs to send the user's data back, including the Name, salary, and SSN, as long as the user provides a correct password.

All users' data are actually stored in the database, so `getdata.php` needs to send a SQL query to the database in order to get the data. There are three main methods for PHP programs to interact with a MySQL database: (1) PHP's MySQL Extension, (2) PHP's MySQLi Extension, and (3) PHP Data Objects (PDO) [php.net, 2017a]. Among them, the `MySQLi` extension is the most commonly used. The extension allows PHP programs to access the functionality provided by MySQL 4.1 and above. We will only use this extension in our examples.

Connecting to MySQL Database. Before conducting queries on a database, a PHP program needs to connect to the database server first. We wrote the following `getDB()` function to make a connection to the database server.

```
function getDB() {
   $dbhost="localhost";
   $dbuser="root";
   $dbpass="seedubuntu";
   $dbname="dbtest";

   // Create a DB connection
   $conn = new mysqli($dbhost, $dbuser, $dbpass, $dbname);
   if ($conn->connect_error) {
         die("Connection failed: " . $conn->connect_error . "\n");
   }
   return $conn;
}
```

The code above uses `new mysqli(...)` to create a database connection. The four arguments include the hostname of the database server, the login name, the password, and the database name. Since in our case, the MySQL database is running on the same machine as the web application server, the name `localhost` is used. If the database runs on a separate machine, such as `db.example.com`, the machine's actual hostname needs to be used.

Constructing a SQL Query. We can now construct a SQL query to fetch user's data based on the provided `EID` and `Password`. A typical approach is to construct the query string first, and then use `mysqli::query()` to send the query string to the database for execution. The following code shows how a query string is constructed, executed, and how the query results are obtained.

```
/* getdata.php */
<?php
   $eid = $_GET['EID'];
   $pwd = $_GET['Password'];

   $conn = new mysqli("localhost", "root", "seedubuntu", "dbtest");
   $sql = "SELECT Name, Salary, SSN                         }  Constructing
           FROM employee                                    }  SQL statement
           WHERE eid= '$eid' and password='$pwd'";          }

   $result = $conn->query($sql);
   if ($result) {
      // Print out the result
      while ($row = $result->fetch_assoc()) {
        printf ("Name: %s -- Salary: %s -- SSN: %s\n",
                $row["Name"], $row["Salary"], $row['SSN']);
      }
      $result->free();
   }
   $conn->close();
?>
```

From the above process, we can see that whatever data are typed in the form, they will eventually become a part of the SQL string, which will be executed by the database. Therefore, although users do not directly interact with the database, there does exist a channel between the user and the database. The channel creates a new attack surface for the database, so if it is not protected properly, users may be able to launch attacks on the database through the channel. That is exactly what causes the SQL Injection vulnerability.

11.3 Launching SQL Injection Attacks

To understand what can go wrong, let us abstract away the details of complicated interactions among browser, web application, and database, so the whole process can be boiled down to the following: The web application creates a SQL statement template, and a user needs to fill in the blank inside the rectangle area. Whatever is provided by the user will become part of the SQL statement. The question is whether it is possible for a user to change the meaning of the SQL statement.

```
    SELECT Name, Salary, SSN
    FROM employee
    WHERE eid=' ▭ ' and password=' ▭ '
```

The intention of the web application developer is for a user to provide some data for the blank areas. However, what is going to happen if a user types some special character? Assume that a user types some random string (`"xyz"`) in the `password` entry, and types `"EID5002' #"` in the `eid` entry (not including the beginning and ending double quotation marks). The SQL statement will become the following:

```
    SELECT Name, Salary, SSN
    FROM employee
    WHERE eid= 'EID5002' #' and password='xyz'
```

Since everything from the # sign to the end of the line is considered as comment, the above SQL statement is equivalent to the following:

```
    SELECT Name, Salary, SSN
    FROM employee
    WHERE eid= 'EID5002'
```

By typing some special characters, such as apostrophe (`'`) and pound sign (`#`), we have successfully changed the meaning of the SQL statement. The above SQL query will now return the name, salary, and social security number of the employee whose EID is `EID5002`, even though the user does not know `EID5002`'s password. This is a security breach.

Let us push this a little bit further, and see whether we can get all the records from the database. Assume that we do not know all the EID's in the database. To achieve this goal, we need to create a predicate for the WHERE clause, so it is always true for all the records. We know that `1=1` is always true, so we type `"a' OR 1=1 #"` in the EID form entry, and the resulting SQL statement will be equivalent to the following:

```
    SELECT Name, Salary, SSN
    FROM employee
    WHERE eid= 'a' OR 1=1
```

The above SQL statement will return all the records in the database.

11.3.1 Attack Using cURL

In the previous section, we launched attacks using forms. Sometimes, it is more convenient to use a command-line tool to launch attacks, because it is easier to automate attacks without a graphic user interface. `cURL` is such a widely-used command-line tool for sending data over a number of network protocols, including HTTP and HTTPS. Using `cURL`, we can send out a form from a command line, instead of from a web page. See the following example.

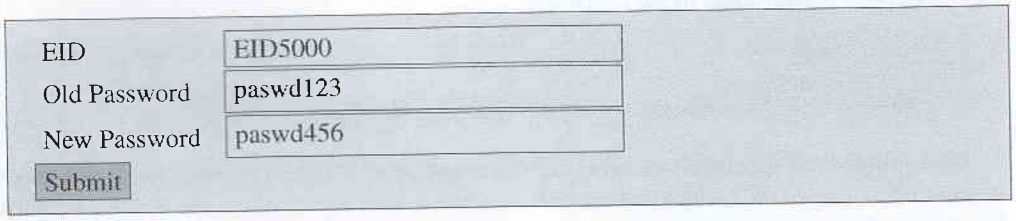
```
% curl 'www.example.com/getdata.php?EID=a' OR 1=1 #&Password='
```

However, the above command does not work. When an HTTP request is sent out, special characters in the attached data need to be encoded, or they may be mis-interpreted, because the URL syntax does use some special characters. In the above URL, we need to encode the apostrophe, whitespace, and the # sign. Their encodings are `%20` (for space), `%23` (for #), and `%27` (for apostrophe). The resulting `cURL` command and results are shown in the folowing:

```
% curl 'www.example.com/getdata.php?EID=a%27%20
                                    OR%201=1%20%23&Password='
Name: Alice -- Salary: 80000 -- SSN: 555-55-5555<br>
Name: Bob -- Salary: 82000 -- SSN: 555-66-5555<br>
Name: Charlie -- Salary: 80000 -- SSN: 555-77-5555<br>
Name: David -- Salary: 80000 -- SSN: 555-88-5555<br>
```

11.3.2 Modify Database

The attack described above shows how we can steal information from a database. We are not able to make changes to the database because the SQL statement affected is a `SELECT` query. If the statement is `UPDATE` or `INSERT INTO`, we will have a chance to change the database. The following is a form created for changing passwords. It asks users to fill in three pieces of information, EID, old password, and new password.



EID	EID5000
Old Password	paswd123
New Password	paswd456

When the `submit` button is clicked, an HTTP POST request will be sent to the following server-side script `changepasswd.php`, which uses an `UPDATE` statement to change the user's password.

```
/* changepasswd.php */
<?php
   $eid = $_POST['EID'];
   $oldpwd = $_POST['OldPassword'];
   $newpwd = $_POST['NewPassword'];

   $conn = new mysqli("localhost", "root", "seedubuntu", "dbtest");
   $sql = "UPDATE employee
           SET password='$newpwd'
```

```
            WHERE eid= '$eid' and password='$oldpwd'";

    $result = $conn->query($sql);
    $conn->close();
?>
```

Since user inputs are used to construct the SQL statement, there is a SQL injection vulnerability. We will see how to use this `UPDATE` SQL statement to change the database. Assuming Alice (EID5000) is not satisfied with the salary she gets. She would like to increase her own salary using the SQL injection vulnerability in the code above. She would type her own EID and password in the EID and "Old Password" boxes, respectively, but she would type the following in the "New Password" box:

New Password	paswd456', salary=100000 #

A single `UPDATE` statement can set multiple attribute of a matching record, if a list of attributes, separated by commas, is given to the `SET` command. The SQL statement in `changepasswd.php` is meant to set only one attribute, the password attribute, but by typing the above string in the `"New Password"` box, we can get the `UPDATE` statement to set one more attribute for us, the salary attribuate. Basically, we have turned the original SQL statement into the following one:

```
UPDATE employee
SET password='paswd456', salary=100000 #'
WHERE eid= 'EID5000' and password='paswd123'";
```

This SQL statement will set two attributes for the matching record, the password and salary fields. Therefore, although the intention of the PHP script is to change the password attribute, due to the SQL injection vulnerability, attackers can make changes to other attributes. In this case, the salary is modified.

Let us add some more fun to this. Alice does not like Bob, so she would like to reduce Bob's salary to 0, but she only knows Bob's EID (`EID5001`), not his password. She can put the following in the form. Readers can verify why this would set Bob's salary to 0.

EID	EID5001' #
Old Password	anything
New Password	paswd456', salary=0 #

11.3.3 Multiple SQL Statements

In the above attack, we have successfully changed the meaning of a SQL statement. Although we can cause damages, our damages are bounded because we cannot change everything in the existing SQL statement. It will be more damaging if we can cause the database to execute an arbitrary SQL statement. Let us try to append a new SQL statement `"DROP DATABASE dbtest"` to the existing SQL statement to delete the entire `dbtest` database. Here is what we can type in the EID box:

EID	a'; DROP DATABASE dbtest; #

The resulting SQL statement is equivalent to the following:

```
SELECT Name, Salary, SSN
FROM employee
WHERE eid= 'a'; DROP DATABASE dbtest;
```

In SQL, multiple statements, separated by semicolon (;), can be included in one statement string. Therefore, using a semicolon, we have successfully appended a new SQL statement of our choice to the existing SQL statement string. If the second SQL statement gets executed, the database `dbtest` will be deleted. We can also append a different statement, such as `INSERT INTO` or `UPDATE` statements, which can cause changes to the database.

Such an attack does not work against MySQL, because in PHP's `mysqli` extension, the `mysqli::query()` API does not allow multiple queries to run in the database server. This is due to the concern of SQL injection. Let us try the following PHP code:

```
/* testmulti_sql.php */
<?php
$mysqli = new mysqli("localhost", "root", "seedubuntu", "dbtest");
$res    = $mysqli->query("SELECT 1; DROP DATABASE dbtest");
if (!$res) {
  echo "Error executing query: (" .
        $mysqli->errno . ") " . $mysqli->error;
}
?>
```

The code above tries to execute two SQL statements using the `$mysqli->query()` API. When running the code, we get the following error message:

```
$ php testmulti_sql.php
Error executing query: (1064) You have an error in your SQL syntax;
   check the manual that corresponds to your MySQL server version
   for the right syntax to use near 'DROP DATABASE dbtest' at line 1
```

It should be noted that the MySQL database server does allow multiple SQL statements to be included in one statement string. If we do want to run multiple SQL statements, we can use `$mysqli->multi_query()`. For the sake of security, we should avoid using this API in our code, especially if the SQL statement string contains untrusted data.

11.4 The Fundamental Cause

Before we discuss the countermeasures against SQL injection attacks, we need to understand the fundamental causes of the vulnerability. We would like to see how various countermeasures address the fundamental causes.

Mixing data and code together is the cause of several types of vulnerabilities and attacks, including the cross-site scripting attack (see Chapter 10), the attack on the `system()` function (see Chapter 1), and the format string attack (see Chapter 6). We can now add the SQL injection vulnerability to that category. Figure 11.2 illustrates the common theme among these four different types of vulnerability.

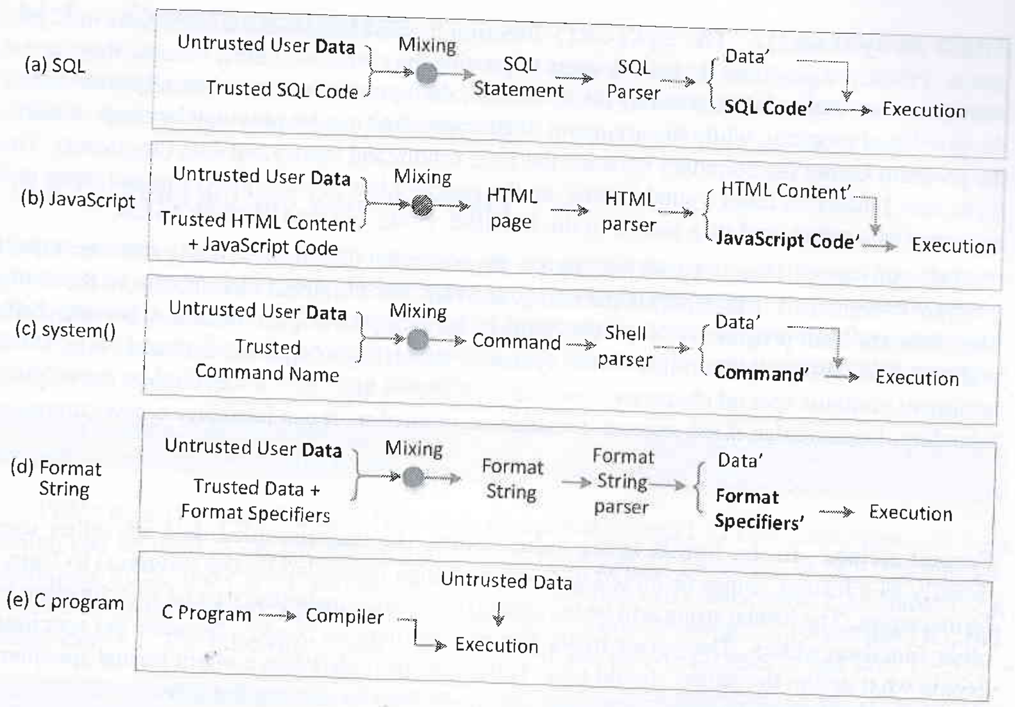


Figure 11.2: Mixing code with data

What they have in common is the following: First, they all mix two pieces of information together: one piece comes from users, which are untrusted, and the other piece is typically provided by the program, which is usually trusted. Before these two pieces of information are mixed, the developer knows the boundaries between them. After they are mixed, these boundaries will disappear. For example, in the SQL injection cases, when constructing a SQL statement, the programmer clearly knows where the user data should be placed inside the SQL statement. However, when the user data are merged into the SQL statement, if the data contain keywords or characters reserved for code, they will alter the original boundaries between code and data.

Second, after the two pieces of information are mixed, the result is passed to a parser. For SQL, it is database's SQL parser; for the case of cross-site scripting, it is the HTML parser. The parser needs to separate data and code, and so it can execute the code. If data contain keywords or special characters, they will be interpreted as code, even though they are part of the original data, because the parser does not know the original boundaries between code and data. That is how attackers can inject code into a vulnerable program via a data channel.

Cross-Site Scripting. In Cross-Site Scripting attacks, the data from the attackers contain JavaScript code, When a web application places the attackers' data into an HTML page, the boundary between the data and the rest of the HTML content (including code) will disappear. When the HTML page is sent to a victim, the parser of the victim's browser will interpret the JavaScript code inside the attacker's data as code, and thus execute the attacker's malicious code.

Attack on `system()`. The `system()` function is used to execute a command in C programs. Privileged programs do not ask users to provide the command name, because they do not want to run any command selected by users. Instead, the command name is typically provided by the privileged program, while the argument of the command can be provided by users. Clearly, the program knows the boundary between the code (command name) and data (argument). The `system()` function takes a single string, so the program has to place the command name and argument in a string, and then pass it to the function. Code and data are now mixed.

The `system()` function does not execute the command directly, it actually executes a shell program (`/bin/sh`), which parses the string, and runs the command identified from the string. Therefore, the shell program is the parser. Without knowing the original boundary between code and data, the shell program relies on the syntax of the string to separate data and code. If the argument contains special characters, such as `';'`, `'>'`, and `'&'`, it can change the original boundary. In particular, if the argument contains a semicolon, it can introduce a new command.

Format string. In the format string vulnerability, the user-provided data are either used directly as a format string, or mixed with other strings (provided by the program) to form a format string. The format string will be interpreted by a parser inside the `printf()` function (or other functions alike). The parser treats format specifiers as "code", because the specifiers decide what action the parser should take. If the user-provided data contain format specifiers, they can essentially introduce "code", and eventually lead to security breaches.

C programs. For comparison, let us look at C programs, and see how they handle data and code in general. The source code of a C program needs to be compiled into binary code first, before it can be executed. During the compilation time, there is no user data, so no user-provided data is mixed with the source code. When the program executes, it takes user data, but the data will not go through the C compiler again, so the data stay as data. Therefore, C programs clearly separates data and code. SQL and JavaScript programs are interpreted language, so they do not need to be compiled first. Therefore, during the runtime, we can dynamically create source code, and then give it to the interpreter for execution. That opens a door for data and code to be mixed together. That is why it is so much easier to launch code injection attacks against web applications than against C programs.

C programs can still suffer from code injection attacks. We have seen two cases above: attacks on `system()` and the format-string attacks. Neither case is caused by the C compiler; they are caused by additional parsers, the external shell parser for the first case, and the format-string parser for the second case.

Another type of code injection attack on C programs is the buffer-overflow attack, and this attack is also caused by the mixing of code and data. When a C program is run, its code and data are separated in the memory, so they are not mixed. However, some addresses for the code, such as the return address, are stored in the data region (stack); that is, they are mixed with data. Although the return address is not an instruction, it directly affects what instructions can be executed, so in essence, it is "code". Therefore, we have a similar situation, where code and data are mixed. Although the code in this case is not provided by users, it can be modified by users if there is a buffer overflow problem on the stack. By modifying the code (i.e., the return address), attackers can change the execution of a target program.

11.5 Countermeasures

There are three main approaches to protect against SQL injection attacks. Let us see how they address the fundamental cause of SQL injection.

11.5.1 Filtering and Encoding Data

Before we mix user-provided data with code, we can inspect the data, and filter out any character that may be interpreted as code. For example, the apostrophe character (`'`) is commonly used in SQL Injection attacks, so if we can get rid of it or encode it, we can prevent the parser from treating it as code. Encoding a special character tells the parser to treat the encoded character as data not as code. See the following example.

```
Before encoding:   aaa' OR 1=1 #
After encoding:    aaa\' OR 1=1 #
```

PHP's `mysqli` extension has a built-in method called `mysqli::real_escape_string`, which can be used to encode the characters that have special meanings in SQL statements, including `NULL` (ASCII 0), carriage return (`\r`), newline (`\n`), backspace (\b), table (`\t`), `Control-Z` (ASCII 26), backslash (\), apostrophe (`'`), double quote (`"`), percentage (%), and underscore (_). The following example shows how to use this API (Lines ① and ②).

```
/* getdata_encoding.php */
<?php
   $conn = new mysqli("localhost", "root", "seedubuntu", "dbtest");
   $eid = $mysqli->real_escape_string($_GET['EID']);            ①
   $pwd = $mysqli->real_escape_string($_GET['Password'];        ②
   $sql = "SELECT Name, Salary, SSN
           FROM employee
           WHERE eid= '$eid' and password='$pwd'";
?>
```

The filtering or escaping approach does not address the fundamental cause of the problem. Data and code are still mixed together. The approach does make the code more secure, but according to the existing studies, escaping special character can be bypassed by carefully constructing the injection [Dahse, 2010]. The approach is listed in this section for completeness, and we do not recommend readers to use this approach.

11.5.2 Prepared Statement

The best way to prevent SQL injection attacks is to separate code from data, so data can never become code. In the countermeasure against the attack on the `system()` function, we run commands using `execve()`, instead of `system()`, because `execve()` takes the command name and data separately, using separate arguments; it never treats the data argument as code. We can do the same for SQL statements by sending code and data in separate channels to the database server, so the database parser knows not to retrieve any code from the data channel.

We can achieve this objective using SQL's prepared statements [Wikipedia, 2017p]. In SQL databases, prepared statement is an optimization feature, which provides an improved performance if the same (or similar) SQL statement needs to be executed repeatedly. Every time a SQL statement is sent to a database for execution, the database needs to parse the statement, and generates binary code. If the SQL statement is the same (or similar), it is a waste of time to repeat the parsing and code generation.

Using prepared statements, we can send an SQL statement template to the database, with certain values left unspecified (they are called parameters). The database parses, compiles, and performs query optimization on the SQL statement template, and stores the result without executing it. Basically, the SQL statement is prepared. At a later time, we can bind values to the parameters in the prepared statement, and ask the database to executes the statement. We can bind different values to the parameters, and run the statement again and again. In different runs, only the data are different; the code part of the SQL statement is always the same, so the prepared statement, which is already compiled and optimized, can be reused.

Although prepared statements were not developed for security purposes, it is an ideal candidate for countermeasures against SQL injection attacks, because it allows us to separate code from data. Let us use an example to see how it can fundamentally solve the SQL injection problem. The goal of our example is to run an SQL statement using user-provided data. To do that, we construct an SQL statement by mixing the user-provided data (`$eid` and `$pwd`) with the SQL statement template. See the following:

```
$conn = new mysqli("localhost", "root", "seedubuntu", "dbtest");
$sql = "SELECT Name, Salary, SSN
        FROM employee
        WHERE eid= '$eid' and password='$pwd'";
$result = $conn->query($sql);
```

The above approach is vulnerable to SQL injection attacks, because code and data are mixed together. Let us use a prepared statement to make the code secure. PHP's `mysqli` extension has APIs for prepared statements.

```
$conn = new mysqli("localhost", "root", "seedubuntu", "dbtest");
$sql = "SELECT Name, Salary, SSN
        FROM employee
        WHERE eid= ? and password=?";                         ①

if ($stmt = $conn->prepare($sql)) {                           ②
   $stmt->bind_param("ss", $eid, $pwd);                       ③
   $stmt->execute();                                          ④

   $stmt->bind_result($name, $salary, $ssn);                  ⑤
   while ($stmt->fetch()) {                                   ⑥
      printf ("%s %s %s\n", $name, $salary, $ssn);
   }
}
```

Preparing SQL statement. Instead of sending a complete SQL statement, we first send an SQL statement template (Lines ① and ②) to the database, which will prepare the statement for future execution. The preparation includes parsing and compiling the template, conducting optimization, and storing the result. We have placed two question marks in the SQL template (Line ①), indicating that these places are placeholders for data. The template only consists of the code and data provided by the program itself; no untrusted data is included in the template.

Binding Data. When we are ready to run the SQL statement with user-provided data, we need to send the data to the database, which will bind them to those placeholders. This is done through the `mysqli::bind_param()` API (Line ③). Since we have two placeholders, we need to pass two data items, `$eid` and `$pwd`. The first argument of the API specifies the types of the data. The argument uses a string, with each character in the string specifying the data type of the corresponding data item. In our example, we have two `s` characters in the string, indicating that both data items are of string type. Other type characters include `i` for integer, `d` for double, and `b` for BLOB (Binary Large Object) [php.net, 2017b].

Execution and retrieving results. Once the data are bound, we can run the completed SQL statement using the `mysqli::execute()` API (Line ④). To get the query results, we can use the `mysqli::bind_result()` API (Line ⑤) to bind the columns in the result to variables, so when When `mysqli::stmt_fetch()` is called (Line ⑥), data for the bound columns are placed into the specified variables.

Why prepared statements can prevent SQL injection attacks. Using prepared statements, trusted code is sent via a code channel, while the untrusted user-provided data are sent via a data channel. Therefore, the database clearly knows the boundary between code and data. When it gets data from the data channel, it will not parse the data. Even though an attacker can hide code in data, the code will never be treated as code, so it will never be executed.

11.6 Summary

Web applications typically store their data in databases. When they need to access data from a database, they construct a SQL statement and send it to the database for execution. Normally, these SQL statements contain the data provided by untrusted users. Web applications need to ensure that no data from users can be treated as code, or the database may execute instructions from the users. Unfortunately, many web applications are not aware of such a risk, and they do not take extra efforts to prevent untrusted code from getting into their constructed SQL statements. As a result, these web applications may have SQL injection vulnerabilities. By exploiting the vulnerabilities, attackers can steal information from databases, modifying their records, or inserting new records.

There are two typical approaches to defeat SQL Injection attacks. One approach is to conduct data sanitization to ensure that user inputs do not contain any SQL code. A better approach is to clearly separate SQL code from data, so when we construct a SQL statement, we send the data and code separately to databases. This way, even if the user-provided data contains code, the code will only be treated as data, and will therefore have no damage to the databases. Prepared statements can be used to achieve this goal.