PART I

ATTACKING WEB 2.0

CHAPTER 1

COMMON INJECTION ATTACKS

Injection attacks were around long before Web 2.0 existed, and they are still amazingly common to find. This book would be incomplete without discussing some older common injection attacks, such as SQL injection and command injection, and newer injection issues, such as XPath injection.

HOW INJECTION ATTACKS WORK

Injection attacks are based on a single problem that persists in many technologies: namely, no strict separation exists between program *instructions* and user *data* (also referred to as user input). This problem allows for attackers to sneak program *instructions* into places where the developer expected only benign *data*. By sneaking in program instructions, the attacker can instruct the program to perform actions of the attacker's choosing.

To perform an injection attack, the attacker attempts to place data that is interpreted as instructions in common inputs. A successful attack requires three elements:

- Identifying the technology that the web application is running. Injection attacks are heavily dependent on the programming language or hardware possessing the problem. This can be accomplished with some reconnaissance or by simply trying all common injection attacks. To identify technologies, an attacker can look at web page footers, view error pages, view page source code, and use tools such as nessus, nmap, THC-amap, and others.
- Identifying all possible user inputs. Some user input is obvious, such as HTML forms. However, an attacker can interact with a web application in many ways. An attacker can manipulate hidden HTML form inputs, HTTP headers (such as cookies), and even backend Asynchronous JavaScript and XML (AJAX) requests that are not seen by end users. Essentially *all* data within every HTTP GET and POST should be considered *user input*. To help identify all possible user inputs to a web application, you can use a web proxy such as WebScarab, Paros, or Burp.
- Finding the user input that is susceptible to the attack. This may seem difficult, but web application error pages sometimes provide great insight into what user input is vulnerable.

The easiest way to explain injection attacks is through example. The following SQL injection example provides a solid overview of an injection attack, while the other examples simply focus on the problem with the specific language or hardware.

SQL Injection

Popularity:	*8*
Simplicity:	*8*
Impact:	*9*
Risk Rating:	***9***

Attackers use SQL injection to do anything from circumvent authentication to gain complete control of databases on a remote server.

SQL, the Structured Query Language, is the de facto standard for accessing databases. Most web applications today use an SQL database to store persistent data for the application. It is likely that any web application you are testing uses an SQL database in the backend. Like many languages, SQL syntax is a mixture of database instructions and user data. If a developer is not careful, the user data could be interpreted as instructions, and a remote user could perform arbitrary instructions on the database.

Consider, for example, a simple web application that requires user authentication. Assume that this application presents a login screen asking for a username and password. The user sends the username and password over some HTTP request, whereby the web application checks the username and password against a list of acceptable usernames and passwords. Such a list is usually a database table within an SQL database.

A developer can create this list using the following SQL statement:

```
CREATE TABLE user_table (
  id INTEGER PRIMARY KEY,
  username VARCHAR(32),
  password VARCHAR(41)
);
```

This SQL code creates a table with three columns. The first column stores an ID that will be used to reference an authenticated user in the database. The second column holds the `username`, which is arbitrarily assumed to be 32 characters at most. The third column holds the `password` column, which contains a hash of the user's password, because it is bad practice to store user passwords in their original form.

We will use the SQL function `PASSWORD()` to hash the password. In MySQL, the output of `PASSWORD()` is 41 characters.

Authenticating a user is as simple as comparing the user's input (username and password) with each row in the table. If a row matches both the username and password provided, then the user will be authenticated as being the user with the corresponding ID. Suppose that the user sent the username *lonelynerd15* and password *mypassword*. The user ID can be looked up:

```
SELECT id FROM user_table WHERE username='lonelynerd15' AND
password=PASSWORD('mypassword')
```

If the user was in the database table, this SQL command would return the ID associated with the user, implying that the user is authenticated. Otherwise, this SQL command would return nothing, implying that the user is not authenticated.

Automating the login seems simple enough. Consider the following Java snippet that receives the username and password from a user and authenticates the user via an SQL query:

```
String username = req.getParameter("username");
String password = req.getParameter("password");
```

```
String query = "SELECT id FROM user_table WHERE " +
    "username = '" + username + "' AND " +
    "password = PASSWORD('" + password + "')";

ResultSet rs = stmt.executeQuery(query);

int id = -1; // -1 implies that the user is unauthenticated.

while (rs.next()) {
      id = rs.getInt("id");
}
```

The first two lines grab the user input from the HTTP request. The next line constructs the SQL query. The query is executed, and the result is gathered in the `while()` loop. If a username and password pair match, the correct ID is returned. Otherwise, the `id` stays `-1`, which implies the user is not authenticated.

If the username and password pair match, then the user is authenticated. Otherwise, the user will not be authenticated, right?

Wrong! There is nothing stopping an attacker from injecting SQL statements in the `username` or `password` fields to change the SQL query.

Let's re-examine the SQL `query` string:

```
String query = "SELECT id FROM user_table WHERE " +
    "username = '" + username + "' AND " +
    "password = PASSWORD('" + password + "')";
```

The code expects the `username` and `password` strings to be data. However, an attacker can input any characters he or she pleases. Imagine if an attacker entered the username *'OR 1=1 --* and password *x*; then the `query` string would look like this:

```
SELECT id FROM user_table WHERE username = '' OR 1=1 -- ' AND password
= PASSWORD('x')
```

The double dash (`--`) tells the SQL parser that everything to the right is a comment, so the query string is equivalent to this:

```
SELECT id FROM user_table WHERE username = '' OR 1=1
```

The `SELECT` statement now acts much differently, because it will now return IDs where the `username` is a zero length string (`''`) or where 1=1; but 1=1 is always true! So this statement will return all the IDs from `user_table`.

In this case, the attacker placed SQL instructions (`'OR 1=1 --`) in the `username` field instead of data.

Choosing Appropriate SQL Injection Code

To inject SQL instructions successfully, the attacker must turn the developer's existing SQL instructions into a valid SQL statement. For instance, single quotes must be closed. Blindly doing so is a little difficult, and generally queries like these work:

- `' OR 1=1 --`
- `') OR 1=1 --`

Also, many web applications provide extensive error reporting and debugging information. For example, attempting `' OR 1=1 --` blindly in a web application often gives you an educational error message like this:

```
Error executing query: 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 'SELECT (title, body) FROM blog_table WHERE
cat='OR 1=1' at line 1
```

The particular error message shows the whole SQL statement. In this case, it appears that the SQL database was expecting an integer, not a string, so the injection string `OR 1=1 --`, without the proceeding apostrophe would work.

With most SQL databases, an attacker can place many SQL statements on a single line as long as the syntax is correct for each statement. For the following code, we showed that setting `username` to `' OR 1=1` and `password` to `x` returns that last user:

```
String query = "SELECT id FROM user_table WHERE " +
    "username = '" + username + "' AND " +
    "password = PASSWORD('" + password + "')";
```

However, the attacker could inject other queries. For example, setting the `username` to this,

```
' OR 1=1; DROP TABLE user_table; --
```

would change this query to this,

```
SELECT id FROM user_table WHERE username='' OR 1=1; DROP TABLE
user_table; -- ' AND password = PASSWORD('x');
```

which is equivalent to this:

```
SELECT id FROM user_table WHERE username='' OR 1=1; DROP TABLE
user_table;
```

This statement will perform the syntactically correct `SELECT` statement and erase the `user_table` with the SQL `DROP` command.