Basic SQL Injection
IN618 Security

Introduction

Online services are frequently connected to back-end data stores, often relational databases. Those databases frequently hold the valuable information in the service, so if we can find ways to exploint them we can carry out very successful attacks. In this lab we will introduce *SQL injection*, a very versatile and powerful attack method.

1 Exploit overview

A simple example of a vulnerable web page has been set up at `http://sql.foo.org.nz`. A user can supply a user name and password to log in, or an email address to request a password reset. This web site, like many others, uses an SQL database to store user account information. We can deduce a few things about this application just by looking at the forms: it stores user names, passwords, and email addresses. But we need to find out more before we can plan a really useful attack. If it turns out that we can sneak some SQL code into any of those input fields, we can use that to get information about the database and perhaps even make changes to it. To understand how this works, we need to have some idea of how the application interacts with its database.

Suppose we enter an email address in the password reset form. What is likely to happen? The script the processes the form submission probably checks the database to see if there is an account with the associated email address. It may use an SQL query like this one:

```
SELECT * FROM user_accounts WHERE email_addr = '<data entered into form>';
```

If that query string is naively constructed by inserting our input into the query string without any sort of input checking or careful handling, then we could input carefully writted SQL code and get it executed on the server. That is what we will try to do in this lab.

2 Test for the vulnerability

Let's use the password reset form to check for SQL injection vulnerability. Why that form?

- It's a simple form with only one field;
- We have a pretty good idea of how it might use an SQL query.

Try the following:

1. Enter your email address into the form and submit it. Since you are almost certianly not an authorized user for this web page, this shold show you how the form behaves when it doesn't find a matching address in the database.

2. We would like to know how it behaves when we put in a valid email address, but we don't know of any. We can however, make a good educated guess.

3. Now let's enter a simple bit of SQL code into the form. We'd like to enter somethng that makes the query syntactically incorrect. Then, if we are lucky the server will throw some kind of error. This will show that we are able to manipulate the query and so the web site is vulnerable. The simplest bit of SQL that works for this is a single quotation mark. Look at the example query above and see how it would be affected if we did this. Then enter a single quote in the form and submit it. Note what happens.

Note that the example query above is not the exact query that the web site code executes, but that doesn't matter. It's good enough to let us reason about how the system behaves.

Now, let's try a slightly more sophistcated query. In the password reset form, enter

```
x' AND 'a'='a
```

What is the reulting SQL when we drop that text into the query above? It turns out that this query always returns a result. Do you see why? It looks like we just sent a password reset email to somebody!

Now we know three things: What happens when we enter an email address that isn't in the user records, what happens when we enter a user's address, and what happens when we enter SQL code that causes an error. We also know that the web site is vulnerable to SQL injection.

3 Finding field names in the table

Next, we would like to use this exploit to learn about the structure of the user account table in the database. For example, we can see that there must be a column in that table that stores email addresses. Possible names for that column include `email_address` and `email`. If we try to access a column that does not exist, we will get an error. If not, we will either get a masseage telling us that a matching address can't be found, or a message we can use that to try various SQL statements and see what responses we get.

Recall that we have to make our injected SQL code fit into a query like the one above. So, to see if the field is `email_address` we would write a query like

```
SELECT * FROM user_accounts WHERE email_addr = 'x' OR email_address IS NULL; -'
```

The red text is the code that we enter into the form. The black text is the SQL that we assume is included in the application code. That first occurence of `email_addr`? Recall that we don't really know exactly what the query looks like, This is just an educated guess to help us think about the problem.

Now if the table has a column named `email_address` we will get a "no matching records" message. If it has no such column, we get an "error processing your request" message. Which do we get in this case? What if we modify the query to check for the field `email` instead of `email_address`?

Now try to find some other column names for things like the user name and password.

4 Finding the table name

At this point we suspect that there is a table that holds user account information and we know some of the column names in that table. It would be useful to know the table name as well. Our strategy is basically the same. If we submit a query that references a nonexistent table, we will get an error. Otherwise, we expect some sort of non-error response. A good query might be something like `SELECT COUNT(*) FROM possible_table_name`. We just need an appropriate way to insert that query into the application's query. We do this as follows (The red text is what we enter into the HTML form.):

```
SELECT * FROM user_accounts WHERE email_addr = 'x'; SELECT COUNT(*) FROM possible_table; -'
```

Basically, the "`x';`" terminates the application's query, telling it to seach for the email address "`x`". There's nothing special about the x, we really don't want to find a record anyway. After the first semicolon we can add our own query, the `SELECT COUNT...` one. We can do this repeatedly with various possible table names until we find one that doesn't produce an error. This is probably our user account table name. The `-` on the end are an SQL comment marker that cause the server to ignore whatever follows it. Try to find the name of the table yourself.

5 Can we write to the database?

At this point we know a lot about the remote database and we have been successful sending various queries to it. The question is, can we change the data in the database? We can certainly submit a query that woud do so, the only question is whether or not the server is configured in a way that lets the web application write to the database. Lets try it.

If we know that the table is named `users` and that it has columns `email`, `name`, and `password`, then the query to insert a record would look like

```
INSERT INTO USERS (email, name, password) VALUES ('tom@.example.com', 'tom', 'foo');
```

We just need to add that to a query we can use in the HTML form, like this:

```
SELECT * FROM user_accounts WHERE email_addr = 'x'; INSERT INTO USERS (email, name, password)
VALUES ('tom@.example.com', 'tom', 'foo'); -'
```

Again, the red text is what we need to insert into the HTML form. To see if it worked, you could try requestion a password reset for the email address `tom@example.com`.

6 Conclusions

Since relational databases actually hold most of the critical data in many applications, a carelessly written application that creates an SQL injection vulnerability can be very harmful. Such vulnerabilities are easily exploited and can lead to significant