SQLi Workshop

Eyvind Niklasson, een7

September 2016

1 What is SQL Injection?

SQL Injection is a vulnerability in an application (often web-application, but can be an API, local executable, etc.) whereby the end-user is able to send arbitrary commands to a back-end database server, or modify existing ones outside their intended parameters.

Such a vulnerability can be used for many purposes:

1. Bypassing authentication/short-circuiting application logic (i.e. login pages)
2. Extracting private information from the database, such as credit-cards, administrator credentials, other users' data, etc.
3. Reading or writing local files on the database server (as the database process)
4. Exploiting the database server software itself to achieve command execution
5. Defacement of the website by re-writing content in the database
6. Chaining with other exploits, such as injecting a persistent XSS

2 What is SQL?

SQL - Server Query Language, is a catch-all term that is used to refer to a large number of different query languages implemented in many popular database products, such as MySQL, Postgres, SQlite, etc. Most of these use very similar syntax, only differing on more technical and implementation-specific terminology.

Web-App - The most common deployment of an SQL server is as a backend database for a web application. Such a web-application can be written in any normal backend language (PHP, Python, Perl, etc.), and is often written in such a way that the web-application handles all request-related logic, such as reading GET, POST variables and essentially determining exactly what the user wants to see. It then opens a socket with the back-end SQL database server, which

is often on another box, and using SQL statements it "asks" this server for all the relevant data. For instance - if the user wants to see a certain product on an online shopping page, the web-app logic will issue a request to the database for possibly the name of the product, a picture of the product, the price, the number of available products, and so forth.

3 Where to get started looking for SQLi?

The most common vulnerability one looks for is a case where in some way the input of the user (GET variables, POST variables, etc.) ends up appearing within an SQL statement that's sent to the database server.

To get started, essentially on a certain page look for all possible "user inputs". Examples would be variables in the URL - such as

http://example.com/derp?variable1=value1&variable2=value2

If you are lucky, the corresponding SQL statement will look something like this

select name, title from some_table where id=value1

If the user input is not properly escaped or cast to an "int", any extra quotation marks will likely cause an error in the web-application/database. Such an error is not always readily apparent, but in the best case scenario the page will literally output "Database Error", like below!

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 " at line 1

Thus - step one is simply to add the following three types of quotation marks to various inputs to see if you can get an error!

"'`

4 I have SQLi - what now?

4.1 Bypassing Authentication

An often found case of SQLi is in various types of forms on websites, often login forms. In this case, the application logic is often something similar to the following query:

select username from users where username="user_inputted_username" and password=hash("user_inputted_password");

The application simply checks if this returns any rows, and if it does then it assumes the user exists! There can be some other variations of this query, such as extracting the password from the database and comparing it with the inputted password, and so forth. Playing with the SQLi should hopefully give you some sense of what the query might look like!

Once you have an idea of what it might look like - try and short circuit this logic! For instance if the application just checks if any rows are returned, maybe try ensuring that a row is returned with the administrator's username by modifying the query slightly!

4.2 Extracting Database Information

The most common next step is to try and extract as much information from the database as possible. This means we want to be able to return our own data into the page, so we can read it. In cases where the output of the database is in some way printed to the website, this is often doable. The key to doing this is the "UNION" statement. The MySQL documentation helpfully explains UNION as:

UNION is used to combine the result from multiple SELECT statements into a single result set.

This is very useful to us as it means we can simply tack on an extra "UNION SELECT X,Y,Z from T" to any query using our SQLi, and get the query to return additional information. To properly craft such a UNION SELECT statement, we need to make sure we are returning the same number of columns (X,Y,Z in this example), as there were in the original statement. A tried and tested approach to doing this is the following two steps:

1. Using your SQLi, insert "order by 1– -" to the end of the query. This orders the returned rows by the first row. Now try "order by 2– -", and so forth, until you reach an error. This will tell you exactly how many columns are being returned!

2. Now that you know the number of columns, try tacking on a "UNION SELECT 1,2,3– -" to the query (assuming there are 3 columns according to step 1). Somewhere on your page, a 1, 2 or 3 should show up! This is where you will be able to output data in the future!

Now that we have a consistent way of writing our own queries and seeing the output, we are interested in finding out what might be in the database! In other words, we are interested in knowing which tables, and which columns exist.

In most SQL implementations this is possible by querying a "meta-table" - namely a table that contains the names of all the tables and columns in the

database. For MySQL, this table is called "information_schema". For SQLite, this is "sqlite_master". However, for both cases it's best to try and look up some cheat-sheet on common commands specific to that implementation.

ADD CHEAT SHEET

4.3 Hack Steps

In summary:

1. Find all possible inputs on a given page. This can vary from URL-variables to POST variables (things you submit in a form, i.e. login form), to cookies, to HTTP headers. Try and reason which ones may or may not be used internally in an SQL query (i.e. if you are submitting a form, most likely all the values go into a database!, if you are selecting the "page" of results to view - this most likely goes into an SQL query, etc.)

2. Try putting some quotation marks - ', ", or ‘ into this input to see if any error is returned.

3. Once you think you have found an SQL injection, try doing a "UNION" query as described above to see if you can output your own results somewhere on the page. If it is a form or a login-form, try bypassing it by short-circuiting the query logic to always return a record.

4. Use the extracted information to log into an administrator console or just further your penetration in some way.

4.4 Hints

1. The first step is getting into the site itself, as it is now behind a login page. Could this login be using some SQL code in background? Look closely for any errors!

2. Be aware of url encoding. Before typing into a url, make sure that your payload is properly encoded.You can use an on-line tool like this one

3. Once you are logged in, look for any pages where you are providing your own input into a variable.

4. In both MySQL and SQLite, "– -" will force a query to end (it will make everything following it into a comment). In other words, if you are in the middle of a query and want to remove junk at the end, just inject a "– -" at the end of your statement.

5. Passwords are almost always stored as "hashes" - i.e. what you could call encrypted form. The most common encryption is MD5, but this is wildly insecure. A good initial step to "crack" a hash is to simply google the hash.