17. SQL Injection

17. SQL Injection

17.1. Code Injection

SQL injection is a special case of a more broad category of attacks called code injections.

As an example, consider a calculator website that accepts user input and calls `eval` in Python in the server backend to perform the calculation. For example, if a user types 2+3 into the website, the server will run eval('2+3') and return the result to the user.

If the web server is not careful about checking user input, an attacker could provide a malicious input like

2+3“); os.system(”rm -rf /

When the web server plugs this into the `eval` function, the result looks like

eval(“2+3"); os.system("rm .”)

If interpreted as code, this statement causes the web server to delete all its files!

The general idea behind these attacks is that a web server uses user input as part of the code it runs. If the input is not properly checked, an attacker could create a special input that causes unintended code to run on the server.

17.2. SQL Injection Example

Many modern web servers use SQL databases to store information such as user logins or uploaded files. These servers often allow users to interact with the database through HTTP requests.

For example, consider a website that stores a SQL table of course evaluations named `evals`:

`id`	`course`	`rating`
1	cs61a	4.5
2	cs61b	4.4
3	cs161	5.0

A user can make an HTTP GET request for a course rating through a URL:

http://www.berkeley.edu/evals?course=cs61a

To process this request, the server performs a SQL query to look up the rating corresponding to the course the user requested:

SELECT rating FROM evals WHERE course = ‘cs61a’

Just like the code injection example, if the server does not properly check user input, an attacker could create a special input that allows arbitrary SQL code to be run. Consider the following malicious input:

garbage’; SELECT password FROM passwords WHERE username = ’admin

When the web server plugs this into the SQL query, the resulting query looks like

SELECT rating FROM evals WHERE course = ‘garbage’; SELECT * FROM passwords WHERE username = ’admin’

If interpreted as code, this causes the query to return the password for the `admin` user!

17.3. SQL Injection Strategies

Writing a malicious input that creates a syntactically valid SQL query can be tricky. Let’s break down each part of the malicious input from the previous example:

- `garbage` is a garbage input to the intended query so that it doesn’t return anything.
- `'` closes the opening quote from the intended query. Without this closing quote, the rest of our query would be treated as a string, not SQL code.
- `;` ends the intended SQL query and lets us start a new SQL query.
- `SELECT password FROM passwords WHERE username = 'admin` is the malicious SQL query we want to execute. Note that we didn’t add a closing quote to `'admin`, because the intended SQL query will automatically add a closing quote at the end of our input.

Consider another vulnerable SQL query. This time, we have a `users` table that contains the `username` and `password` of every user.

When the web server receives a login request, it creates a SQL query by plugging in the username and password from the request. For example, if you make a login request with username `alice` and password `password123`, the resulting SQL query would be

SELECT username FROM users WHERE username = ‘alice’ AND password = ‘password123’

If the query returns more than 0 rows, the server registers a successful login.

Suppose we want to login to the server, but we don’t have an account, and we don’t know anyone’s username. How might we achieve this using SQL injection?

First, in the username field, we should add a dummy username and a quote to end the opening quote from the original query:

SELECT username FROM users WHERE username = ‘alice’’ AND password = ‘password123’

Next, we need to add some SQL syntax so that this query returns more than 0 rows (since we don’t know if `alice` is a valid username). One trick for forcing a SQL query to always return something is to add some logic that always evaluates to true, such as `OR 1=1`:

SELECT username FROM users WHERE username = ‘alice’ OR 1=1’ AND password = ‘________’

Next, we have to add some SQL so that the rest of the query doesn’t throw a syntax error. One way of doing this is to add a semicolon (ending the previous query) and write a dummy query that matches the remaining SQL:

SELECT username FROM users WHERE username = ‘alice’ OR 1=1; SELECT username FROM users WHERE username = ’alice’ AND password = ‘________’

The second query might not return anything, but the first query will return a nonzero number of entries, which lets us perform a login. The last step is to add some garbage as the password:

SELECT username FROM users WHERE username = ‘alice’ OR 1=1; SELECT username FROM users WHERE username = ’alice’ AND password = ‘garbage’

Thus, our malicious username and password should be

username = alice’ OR 1=1; SELECT username FROM users WHERE username = ’alice password = garbage

Another trick to make SQL injection easier is the `--` syntax, which starts a comment in SQL. This tells SQL to ignore the rest of the query as a comment.

In our previous example, we can instead start a comment to ignore parts of the query we don't want to execute:

SELECT username FROM users WHERE username = 'alice' OR 1=1–' AND password = 'garbage'

Thus, another malicious username and password is

username = alice' OR 1=1– password = garbage

Further reading: SQL Injection Attacks by Example

17.4. Defense: Escape Inputs

One way of defending against SQL injection is to escape any potential input that could be used in an attack. Escaping a character means that you tell SQL to treat this character as part of the string, not actual SQL syntax.

For example, the quote `"` is used to denote the end of a string in SQL. However, the escaped quote `\"` is treated as a literal quote character in SQL, and it does not cause the current string to end.

By properly replacing characters with their escaped version, malicious inputs such as the ones we've been creating will be treated as strings, and the SQL parser won't try to run them as actual SQL commands.

For example, in the previous exploit, if the server replaces all instances of the quote `"` and the dash `-` with escaped versions, the SQL parser will see

SELECT username FROM users WHERE username = 'alice' OR 1=1--' AND password = 'garbage'

The escaped quote won't cause the `username` string to end, and the escaped dashes won't cause a comment to be created. The parser will try to look up someone with a username `alice" OR 1=1--` and find nothing.

However, we have to be careful with escaping. If an attacker inputs a backslash followed by a quote `\"`, the escaper might "escape the escape" and give the input `\\"` to the SQL parser. The parser will treat the two backslashes `\\` as an escaped backslash, and the quote won't be escaped!

The key takeaway here is that building a good escaper can be tricky, and there are many edge cases to consider. There is almost no circumstance in which you should try to build an escaper yourself; secure SQL escapers exist in SQL libraries for almost every programming language. However, if you are running SQL statements with raw user input, escapers are often an ineffective solution, because you need to ensure that every call is properly escaped. A far more robust solution is to use parameterized SQL.

17.5. Defense: Parameterized SQL/Prepared Statements

A better defense against SQL injection is to use parameterized SQL or prepared statements. This type of SQL compiles the query first, and then plugs in user input after the query has already been interpreted by the SQL parser. Because the user input is added after the query is compiled and interpreted, there is no way for any attacker input to be treated as SQL code. Parameterized SQL prevents all SQL injection attacks, so it is the best defense against SQL injection!

In most SQL libraries, parameterized SQL and unsafe, non-paramaterized SQL are provided as two different API functions. You can ensure that you've eliminated *all* potential SQL vulnerabilities in your code by searching for every database query and replacing each API call with a call to the parameterized SQL API function.

The biggest problem with parameterized SQL is compatibility. SQL is a (mostly) generic language, so SQL written for MySQL can run on Postgres or commercial databases. Parameterized SQL requires support from the underlying database (since the processing itself happens on the database side), and there is no common standard for expressing parameterized SQL. Most SQL libraries will handle the translation for you, but switching to prepared statements may make it harder to switch between databases.

In practice, most modern SQL libraries support parameterized SQL and prepared statements. If the library you are using does not support parameterized SQL, it is probably best to switch to a different SQL library.

Further reading: OWASP Cheat Sheet on SQL Injection