Week 4, Lecture 2

COMP6[48]43

Hamish Cox

Mar 12th, 2025

Lecture 1 Recap

- SQL injection.
  - The basic idea
  - Getting information from other tables - let's do that again.

Mitigations

What ideas do we have?

- WAFs?
-
-

Mitigations

What ideas do we have?

- WAFs? Sure but we can do better.
-

-

Mitigations

What ideas do we have?

- WAFs? Sure but we can do better.
- Escaping - theoretically sound, but hard to do - best to let the library do it where possible.
-

Mitigations

What ideas do we have?

- WAFs? Sure but we can do better.
- Escaping - theoretically sound, but hard to do - best to let the library do it where possible.
- Parameterised Queries

An Example

Provide a query with placeholders.

Then provide the data as a separate argument.

```
results = cur.execute(
            "SELECT * FROM users WHERE username = %s",
            (username,)
          ).fetchone()
```

Tangent: A More General Mitigation Strategy

Tangent: A More General Mitigation Strategy

RTFM (Read The F*cking Manual)

Tangent: A More General Mitigation Strategy

RTFM (Read The F*cking Manual)

like srsly, do it

Examples

- Any application that accepts input must expect to handle bad data.

  The bad data might be accidental, such as out-of-range values or misformatted strings. The application can use server-side checks such as unique constraints and NOT NULL constraints, to keep the bad data from ever reaching the database. On the client side, use techniques such as exception handlers to report any problems and take corrective action.

  The bad data might also be deliberate, representing an "SQL injection" attack. For example, input values might contain quotation marks, semicolons, % and _ wildcard characters and other characters significant in SQL statements. Validate input values to make sure they have only the expected characters. Escape any special characters that could change the intended behavior when substituted into an SQL statement. Never concatenate a user input value into an SQL statement without doing validation and escaping first. Even when accepting input generated by some other program, expect that the other program could also have been compromised and be sending you incorrect or malicious data.

Examples

In order to avoid SQL Injection attacks, you should always escape any user provided data before using it inside a SQL query. You can do so using the `mysql.escape()`, `connection.escape()` or `pool.escape()` methods:

Examples

Parameterized query

If you are passing parameters to your queries you will want to avoid string concatenating parameters into the query text directly. This can (and often does) lead to sql injection vulnerabilities. node-postgres supports parameterized queries, passing your query text *unaltered* as well as your parameters to the PostgreSQL server where the parameters are safely substituted into the query with battle-tested parameter substitution code within the server itself.

Examples

Warning: Never, **never**, **NEVER** use Python string concatenation (+) or string parameters interpolation (%) to pass variables to a SQL query string. Not even at gunpoint.

Examples

> ⚠ Warning
>
> - Don't manually merge values to a query: hackers from a foreign country will break into your computer and steal not only your disks, but also your cds, leaving you only with the three most embarrassing records you ever bought. On cassette tapes.
> - If you use the `%` operator to merge values to a query, con artists will seduce your cat, who will run away taking your credit card and your sunglasses with them.
> - If you use `+` to merge a textual value to a string, bad guys in balaclava will find their way to your fridge, drink all your beer, and leave your toilet seat up and your toilet paper in the wrong orientation.
> - You don't want to manually merge values to a query: use the provided methods instead.

The Golden Rules of Web Application Security

The Golden Rules of Web Application Security

1. Never trust user input.
2.

The Golden Rules of Web Application Security

1. Never trust user input.
2. RTFM

The Golden Rules of Web Application Security

1. Never trust ~~user~~ input.
2. RTFM

Tangent: “Trust Boundaries” and Niche/Interesting Things

These places are where vulnerabilities can sneak in.

Some Advanced Techniques

Suppose we have an input that is looked up in a database and tells us whether it was successful or not, but we don’t actually get shown the result of the query afterwards. How can we extract data?

Some Advanced Techniques

Suppose we have an input that is looked up in a database and tells us whether it was successful or not, but we don’t actually get shown the result of the query afterwards. How can we extract data?

What if we don’t even get a success/fail and it just 500s without an error message?

Some Advanced Techniques

Suppose we have an input that is looked up in a database and tells us whether it was successful or not, but we don't actually get shown the result of the query afterwards. How can we extract data?

What if we don't even get a success/fail and it just 500s without an error message?

What if we don't even get 500s?

Moar Injection

Suppose we need to interface with another system.

Moar Injection

Suppose we need to interface with another system.

Suppose that system is another program on our server.

Moar Injection

Suppose we need to interface with another system.

Suppose that system is another program on our server.

How might we interact with it?

Let's use `ls` in Python

```
os.system("ls <directory>")
```

Let's use `ls` in Python

```
os.system(f"ls {request.args.get('directory', '.')}")
```

Let's use `ls` in Python

Stepping back for a moment:

```
ls -lah .
```

Commands, Arguments and Shells

What is this?

```c
int main(int argc, char **argv) {
  for (int i = 0; i < argc; i++){
    printf("%s\n", argv[i]);
  }

  return 0;
}
```

Commands, Arguments and Shells

Here's the Python equivalent:

```
import sys

print(sys.argv)
```

Commands, Arguments and Shells

What happens if we call one of the above two programs using `os.system`?

What is `system`?

Python's `os.system` calls out to C's stdlib:

> The **system**() library function behaves as if it used fork(2) to create a child process that executed the shell command specified in *command* using execl(3) as follows:
>
>   execl("/bin/sh", "sh", "-c", command, (char *) NULL);

Popen/exec

Ideally we want to tell the OS exactly what arguments we want to pass and bypass the shell parsing.

About the Midterm

Mix of short answer and challenges.

For challenges:

- Writeups are 4/5.
- The flag/solve is only 1/5.

Submitted on Moodle as a text file. There will be a template for this file so we can mark easily.

The exam will be available at https://midterm.quoccacorp.com.

Example writeup is available in the Midterm spec.