Chapter 31. Defending Against Injection

In Chapter 13, we discussed the risk that injection-style attacks bring against web applications. These attacks are still common (although they were more common in the past), typically as a result of improper attention on the part of the developer writing any type of automation involving a CLI and user-submitted data.

Injection attacks also cover a wide surface area. Injection can be used against CLIs or any other isolated interpreter running on the server (when it hits the OS level, it becomes command injection instead). As a result, when considering how we will defend against injection-style attacks, it is easier to break such defensive measures up into a few categories.

First off, we should evaluate defenses against SQL injection attacks—the most common and well-known form of injection. After investigating what we can do to protect against SQL injection, we can see which of those defenses will be applicable to other forms of injection attacks. Finally, we can evaluate a few generic methods of defense against injection that are not specific to any particular subset of injection-based attack.

Mitigating SQL Injection

SQL injection is the most common form of injection attack, and likewise one of the easiest to defend against. Since it is so widespread, potentially affecting nearly every complex web application (due to the prevalence of SQL databases), many mitigations and countermeasures have been developed against SQL injection.

Furthermore, because SQL injection attacks take place in the SQL interpreter, detecting such vulnerabilities can be quite simple. With proper detection and mitigation strategies in place, the odds of your web application being exposed to SQL injection attack are quite low.

Detecting SQL Injection

To prepare your codebase for defense against SQL injection attacks, first familiarize yourself with the form SQL injection takes and the locations in your codebase that would be most vulnerable. In most modern web applications, SQL operations would occur past the server-side routing level. This means we aren't too interested in anything on the client. For example, we have a web application code repository file structure that looks like this:

```
/api
  /routes
  /utils
/analytics
  /routes
/client
  /pages
  /scripts
  /media
```

We know we can skip searching the client, but we should consider the analytics route because even if it is built on OSS, it likely uses a database of some sort to store the analytics data. Remember that if data is persisting between devices and sessions, it is either stored in server-side memory, disk (logs), or in a database.

On the server, we should be aware that many applications make use of more than one database. This could mean that an application makes use of SQL server and MySQL, for example. So when searching the server, we need to make use of generic queries so that we can find SQL queries across multiple SQL language implementations.

Furthermore, some server software makes use of a DSL, which could potentially make SQL calls on our behalf, although these calls would not be structured similarly to a raw SQL call.

To properly analyze an existing codebase for potential SQL injection risks, we need to compile a list of all the preceding DSL and types of SQL and store it in one place.

If our application is a Node.js app and contains:

- SQL Server—via `NodeMSSQL` adapter (npm)
- MySQL—via `mysql` adapter (npm)

then we need to consider structuring searches in our codebase that can find SQL queries from both SQL implementations.

Fortunately, the module import system that ships with Node.js makes this easy when combined with the JavaScript language scope. If the SQL library is imported on a per-module basis, finding queries becomes as easy as searching for the import:

```
const sql = require('mssql')
// OR
const mysql = require('mysql');
```

On the other hand, if these libraries are declared globally, or inherited from a parent class, the work for finding queries becomes a bit more difficult.

Both of the two aforementioned SQL adapters for Node.js use a syntax that concludes with a call to `.query(x)`, but some adapters use a more literal syntax:

```
const sql = require('sql');

const getUserByUsername = function(username) {
  const q = new sql();
  q.select('*');
  q.from('users');
```

```
    q.where(`username = ${username}`);
    q.then((res) => {
      return `username is : ${res}`;
    });
  };
```

Prepared Statements

As mentioned earlier, SQL queries have been widespread in the past and are extremely dangerous. But they are also not very difficult to protect against in most cases.

One development that most SQL implementations have begun to support is *prepared statements*. Prepared statements reduce a significant amount of risk when using user-supplied data in a SQL query. Beyond this, prepared statements are very easy to learn and make debugging SQL queries much easier.

TIP

Prepared statements are often considered the "first line" of defense against injection. Prepared statements are easy to implement, well documented on the web, and highly effective at stopping injection attacks.

Prepared statements work by compiling the query first, with placeholder values for variables. These are known as *bind variables* but are often just referred to as placeholder variables. After compiling the query, the placeholders are replaced with the values provided by the developer. As a result of this two-step process, the intention of the query is set before any user-submitted data is considered.

In a traditional SQL query, the user-submitted data (variables) and the query itself are sent to the database together in the form of a string. This means that if the user data is manipulated, it could change the intention of the query.

With a prepared statement, because the intention is set in stone prior to the user-submitted data being presented to the SQL interpreter, the query itself cannot change. This means that a SELECT operation against users cannot be escaped and modified into a DELETE operation by any means. An additional query cannot occur after the SELECT operation if the user escapes the original query and begins a new one. Prepared statements eliminate most SQL injection risk and are supported by almost every major SQL database: MySQL, Oracle, PostgreSQL, Microsoft SQL Server, etc.

The only major trade-off between traditional SQL queries and prepared statements is that of performance. Rather than one trip to the database, the database is provided the prepared statement followed by the variables to inject after compilation and at runtime of the query. In most applications, this performance loss will be minimal.

Syntactically, prepared statements differ from database to database and adapter to adapter. In MySQL, prepared statements are quite simple:

```
PREPARE q FROM 'SELECT name, barCode from products WHERE price <= ?';
SET @price = 12;
EXECUTE q USING @price;
DEALLOCATE PREPARE q;
```

In this prepared statement, we are querying the MySQL database for products (we want name and barcode returned) that have a price less than `?`. First, we use the statement `PREPARE` to store our query under the name `q`. This query will be compiled and ready for use. Next, we set a variable `@price` to 12. This would be a good variable to have a user set if they were filtering against an ecommerce site, for example. Then we `EXCECUTE` the query providing `@price` to fill the *?* placeholder/bind variable. Finally, we use `DEALLOCATE` on `q` to remove it from memory so its namespace can be used for other things.

In this simple prepared statement, `q` is compiled prior to being executed with `@price`. Even if `@price` was set equal to `5; UPDATE users WHERE id = 123 SET balance = 10000`, the additional query would not fire because it would not be compiled by the database.

The much less secure version of this query would be:

```
'SELECT name, barcode from products WHERE price <= ' + price + ';'
```

As you can clearly see, the precompilation of prepared statements is an essential first step in mitigating SQL injection and should be used wherever possible in your web application.

Database-Specific Defenses

In addition to prepared statements that are widely adopted, each major SQL database offers its own functions for improving security. Oracle, MySQL, MS SQL Server, and Salesforce Object Query Language (SOQL) all offer methods for automatically escaping characters and character sets deemed risky for use in SQL queries. The method by which these sanitizations are decided is dependent on the particular database and engine being used.

Oracle (Java) offers an encoder that can be invoked with the following syntax:

```
ESAPI.encoder().encodeForSQL(new OracleCodec(), str);
```

Similarly, MySQL offers equivalent functionality. In MySQL, the following can be used to prevent the usage of improperly escaped strings:

```
SELECT QUOTE('test''case');
```

The `QUOTE` function in MySQL will escape backslashes, single quotes, or NULL, and return a properly single-quoted string. MySQL offers `mysql_real_escape_string()` as well, which escapes all of the preceding backslashes and single quotes, but also escapes double quotes, \n, and \r (linebreak).

Making use of database-specific string sanitizers for escaping risky character sets reduces the SQL injection risk by making it harder to write a

SQL literal versus a string. These should always be used if a query is being run that cannot be parameterized—though they should not be considered a comprehensive defense but instead a mitigation.

Generic Injection Defenses

In addition to being able to defend against SQL injection, you should also make sure your application is defended against other less common forms of injection. As we learned in Part II, injection attacks can occur against any type of command-line utility or interpreter.

We should be on the lookout for non-SQL injection targets and apply secure-by-default coding practices throughout our application logic to mitigate the risk of an unexpected injection vulnerability appearing.

Potential Injection Targets

In "Code Injection", we explored a scenario where video or image compression CLIs could be used as a potential injection target. But injection is not limited to CLIs such as FFMPEG. It extends across any type of script that takes text input and interprets the text in some type of interpreter or evaluates the text against some list of commands.

Typically, when on the lookout for injection, the following are high-risk targets:

- Task schedulers
- Compression/optimization libraries
- Remote backup scripts
- Databases
- Loggers
- Any call to the host OS
- Any interpreter or compiler

When first ranking components of your web application for potential injection risk, compare them with the preceding list of high-risk targets. Those are your starting points for investigation.

Dependencies can also be a risk because many dependencies bring in their own (sub) dependencies that can often fall into one of those categories.

Principle of Least Authority

The *principle of least authority* (often called *principle of least privilege*, which I believe to be a bit less succinct) is an important abstraction rule that should always be used when attempting to build secure web applications. The principle states that in any system, each member of the system should only have access to the information and resources required to accomplish their job (see Figure 31-1).

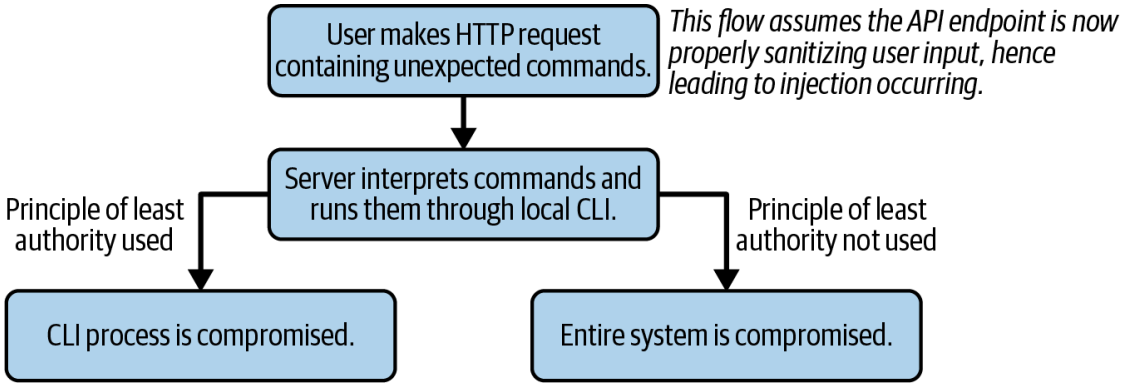

Figure 31-1. Using the principle of least authority when designing your web application, you can reduce the impact of any injection attack that is accidentally introduced

In the world of software, the principle can be applied as such: "each module in a software system should only have access to the data and functionality required for that module to operate correctly." It sounds simple in theory, but it is seldom applied in large-scale web applications where it should be. The principle actually becomes more important as an application scales in complexity, as interactions between modules in a complex application can bring unintended side effects.

Imagine you are building a CLI that integrates with your web application and automatically backs up user profile photos. This CLI is called either from the terminal (manual backups) or through an adapter written in the programming language that your web application is built in. If this CLI were to be built with the principle of least authority, then even if the CLI was compromised, the rest of the application would not be compromised. On the other hand, a CLI running as admin could expose an entire appli-

cation server in the case of a rogue injection attack being uncovered and exploited.

The principle of least authority may seem like a roadblock to developers—managing additional accounts, multiple keys, etc.—but proper implementation of this principle will limit the risk your application is exposed to in the case of a breach.

Allowlisting Commands

The biggest risk for injection is a functionality in a web application where the client (user) sends commands to a server to be executed. This is a bad architectural practice and should be avoided at all costs.

When user-chosen commands need to be executed on a server in any context that would allow them to create potential damage or alter the state of the application (in the case of misuse), additional steps are required. Instead of allowing user commands to be interpreted literally by the server, a well-defined allowlist of user-available commands should be created. This, in addition to a well-defined acceptable syntax for commands (order, frequency, params), should be used together, all stored in allowlist format rather than blocklist format.

Consider the following example:

```
<div class="options">
 <h2>Commands</h2>
 <input type="text" id="command-list"/>
 <button type="button" onclick="sendCommands()">Send Commands to Server</button>
</div>
```

```
const cli = require('../util/cli');

/*
 * Accepts commands from the client, runs them against the CLI.
 */
const postCommands = function(req, res) {
```

```
  cli.run(req.body.commands);
};
```

In this case, the client is capable of executing any commands against the server that are supported by the *cli* library. This means that the CLI execution environment and full scope are accessible to the end user simply by providing commands that are supported by the CLI, even if they are not intended for use by the developer.

In a more obscure case, perhaps the commands are all allowed by the developer, but the syntax, order, and frequency can be combined to create unintended functionality (injection) against the CLI on the server. A quick and dirty mitigation would be to only allowlist a few commands:

```
const cli = require('../util/cli');

const commands = [
 'print',
 'cut',
 'copy',
 'paste',
 'refresh'
];

/*
 * Accepts commands from the client, runs them against the CLI ONLY if
 * they appear in the allowlist array.
 */
const postCommands = function(req, res) {
  const userCommands = req.body.commands;
  userCommands.forEach((c) => {
    if (!commands.includes(c)) { return res.sendStatus(400); }
  });
  cli.run(req.body.commands);
};
```

This quick and dirty mitigation may not resolve issues involving the order or frequency of the commands, but it will prevent commands not intended for use by the client or end user from being invoked. A blocklist is not used because applications evolve over time. Blocklists are seen as a

security risk in the case of a new command being added that would provide the user with unwanted levels of functionality. When user input MUST be accepted and fed into a CLI, always opt for an allowlist approach over a blocklist approach.

Summary

Injection attacks are classically attributed to databases, in particular, SQL databases. But while databases are definitely vulnerable to injection attacks without properly written code and configuration, any CLI that an API endpoint (or dependency) interacts with could be a victim of injection.

Major SQL databases offer mitigations to prevent SQL injection, but SQL injection is still possible with shoddy application architecture and improperly written client-to-server code. Introducing the principle of least authority into your codebase will aid your application in the case of a breach by minimizing damage dealt to your organization and your application's infrastructure. An application architected in a security-first manner will never allow a client (user) to provide a query or command that will be executed on the server.

If user input needs to translate into server-side operations, the operations should be allowlisted so that only a subset of total functionality is available, and only functionality that has been vetted as secure by a responsible security review team.

By using those controls, an application will be much less likely to have injection-style vulnerabilities.