Intro to Command Injections

A Command Injection vulnerability is among the most critical types of vulnerabilities. It allows us to execute system commands directly on the back-end hosting server, which could lead to compromising the entire network. If a web application uses user-controlled input to execute a system command on the back-end server to retrieve and return specific output, we may be able to inject a malicious payload to subvert the intended command and execute our commands.

What are Injections

Injection vulnerabilities are considered the number 3 risk in OWASP's Top 10 Web App Risks, given their high impact and how common they are. Injection occurs when user-controlled input is misinterpreted as part of the web query or code being executed, which may lead to subverting the intended outcome of the query to a different outcome that is useful to the attacker.

There are many types of injections found in web applications, depending on the type of web query being executed. The following are some of the most common types of injections:

Injection	Description
OS Command Injection	Occurs when user input is directly used as part of an OS command.
Code Injection	Occurs when user input is directly within a function that evaluates code.
SQL Injections	Occurs when user input is directly used as part of an SQL query.
Cross-Site Scripting/HTML Injection	Occurs when exact user input is displayed on a web page.

There are many other types of injections other than the above, like `LDAP injection`, `NoSQL Injection`, `HTTP Header Injection`, `XPath Injection`, `IMAP Injection`, `ORM Injection`, and others. Whenever user input is used within a query without being properly sanitized, it may be possible to escape the boundaries of the user input string to the parent query and manipulate it to change its intended purpose. This is why as more web technologies are introduced to web applications, we will see new types of injections introduced to web applications.

OS Command Injections

When it comes to OS Command Injections, the user input we control must directly or indirectly go into (or somehow affect) a web query that executes system commands. All web programming languages have different functions that enable the developer to execute operating system commands directly on the back-end server whenever they need to. This may be used for various purposes, like installing plugins or executing certain applications.

PHP Example

For example, a web application written in `PHP` may use the `exec`, `system`, `shell_exec`, `passthru`, or `popen` functions to execute commands directly on the back-end server, each having a slightly different use case. The following code is an example of PHP code that is vulnerable to command injections:

Code: php

```
<?php
if (isset($_GET['filename'])) {
    system("touch /tmp/" . $_GET['filename'] . ".pdf");
}
?>
```

Perhaps a particular web application has a functionality that allows users to create a new `.pdf` document that gets created in the `/tmp` directory with a file name supplied by the user and may then be used by the web application for document processing purposes. However, as the user input from the `filename` parameter in the `GET` request is used directly with the `touch` command (without being sanitized or escaped first), the web application becomes vulnerable to OS command injection. This flaw can be exploited to execute arbitrary system commands on the back-end server.

NodeJS Example

This is not unique to `PHP` only, but can occur in any web development framework or language. For example, if a web application is developed in `NodeJS`, a developer may use `child_process.exec` or `child_process.spawn` for the same purpose. The following example performs a similar functionality to what we discussed above:

Code: javascript

```
app.get("/createfile", function(req, res){
    child_process.exec(`touch /tmp/${req.query.filename}.txt`);
})
```

The above code is also vulnerable to a command injection vulnerability, as it uses the `filename` parameter from the `GET` request as part of the command without sanitizing it first. Both `PHP` and `NodeJS` web applications can be exploited using the same command injection methods.

Likewise, other web development programming languages have similar functions used for the same purposes and, if vulnerable, can be exploited using the same command injection methods. Furthermore, Command Injection vulnerabilities are not unique to web applications but can also affect other binaries and thick clients if they pass unsanitized user input to a function that executes system commands, which can also be exploited with the same command injection methods.

The following section will discuss different methods of detecting and exploiting command injection vulnerabilities in web applications.


Powered by HACKTHEBOX