A Command Injection vulnerability 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.

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:

*<?php*

*if (isset($\_GET['filename'])) {*

*system("touch /tmp/" . $\_GET['filename'] . ".pdf");*

*} ?>*

The above PHP code 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**

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:

*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.

**Detection**

The process of detecting basic OS Command Injection vulnerabilities is the same process for exploiting such vulnerabilities. We attempt to append our command through various injection methods. If the command output changes from the intended usual result, we have successfully exploited the vulnerability.

**Identifying Filters**

A type of injection mitigation is utilizing blacklisted characters and words on the back-end to detect injection attempts and deny the request if any request contained them. Yet another layer on top of this is utilizing Web Application Firewalls (WAFs), which may have a broader scope and various methods of injection detection and prevent various other attacks like SQL injections or XSS attacks.

**Blacklisted Characters**

A web application may have a list of blacklisted characters, and if the command contains them, it would deny the request. The PHP code may look something like the following:

*$blacklist = ['&', '|', ';', ...SNIP...];*

*foreach ($blacklist as $character) {*

*if (strpos($\_POST['ip'], $character) !== false) {*

*echo "Invalid input"; } }*

**Bypassing Blacklisted Space Character**

1. Using tabs (%09) instead of spaces is a technique that may work, as both Linux and Windows accept commands with tabs between arguments, and they are executed the same.
2. Using the (IFS)LinuxEnvironmentVariablemayalsoworksinceitsdefaultvalueisaspaceandatab, which wouldw

{IFS}` where the spaces should be, the variable should be automatically replaced with a space, and our command should work. (127.0.0.1%0a${IFS}).

1. We can use the Bash Brace Expansion feature, which automatically adds spaces between arguments wrapped between braces. Eg – {ls,-la}

**Character Shifting**

There are other techniques to produce the required characters without using them, like shifting characters . For example, the following Linux command shifts the character we pass by 1 . So, all we have to do is find the character in the ASCII table that is just before our needed character (we can get it with man ascii ), then add it instead of [ in the below example. This way, the last printed character would be the one we need:

*man ascii # \ is on 92, before it is [ on 91*

*echo $(tr '!-}' '"-~'<<<[)*

*\*

**Command Injection Prevention**

**System Commands**

We should always avoid using functions that execute system commands, especially if we are using user input with them. Even when we are not directly inputting user input into these functions, a user may be able to indirectly influence them, which may eventually lead to a command injection vulnerability.

Instead of using system command execution functions, we should use built-in functions that perform the needed functionality, as back-end languages usually have secure implementations of these types of functionalities. For example, suppose we wanted to test whether a particular host is alive with PHP . In that case, we may use the fsockopen function instead, which should not be exploitable to execute arbitrary system commands.

If we needed to execute a system command, and no built-in function can be found to perform the same functionality, we should never directly use the user input with these functions but should always validate and sanitize the user input on the back-end. Furthermore, we should try to limit our use of these types of functions as much as possible and only use them when there's no built-in alternative to the functionality we require.

**Input Validation**

Whether using built-in functions or system command execution functions, we should always validate and then sanitize the user input. Input validation is done to ensure it matches the expected format for the input, such that the request is denied if it does not match.

**Input Sanitization**

The most critical part for preventing any injection vulnerability is input sanitization, which means removing any non-necessary special characters from the user input. Input sanitization is always performed after input validation. Even after we validated that the provided user input is in the proper format, we should still perform sanitization and remove any special characters not required for the specific format, as there are cases where input validation may fail (e.g., a bad regex).

In certain cases, we may want to allow all special characters (e.g., user comments), then we can use the filter\_var function, and use the escapeshellcmd filter to escape any special characters, so they cannot cause any injections. For NodeJS , we can simply use the escape(ip) function.

**Server Configuration**

We should make sure that our back-end server is securely configured to reduce the impact in the event that the webserver is compromised. Some of the configurations we may implement are:

1. Use the web server's built-in Web Application Firewall (e.g., in Apache mod\_security ), in addition to an external WAF (e.g. Cloudflare , Fortinet , Imperva ..)
2. Abide by the Principle of Least Privilege by running the web server as a low privileged user (e.g. www-data )
3. Prevent certain functions from being executed by the web server (e.g., in PHP disable\_functions=system)
4. Limit the scope accessible by the web application to its folder (e.g. in PHP open\_basedir = '/var/www/html' )
5. Reject double-encoded requests and non-ASCII characters in URLs
6. Avoid the use of sensitive/outdated libraries and modules (e.g. PHP CGI)