Command Injection Prevention

We should now have a solid understanding of how command injection vulnerabilities occur and how certain mitigations like character and command filters may be bypassed. This section will discuss methods we can use to prevent command injection vulnerabilities in our web applications and properly configure the webserver to prevent them.

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. In our example web application, we saw that there was an attempt at input validation on the front-end, but `input validation should be done both on the front-end and on the back-end`.

In `PHP`, like many other web development languages, there are built in filters for a variety of standard formats, like emails, URLs, and even IPs, which can be used with the `filter_var` function, as follows:

Code: php

```php
if (filter_var($_GET['ip'], FILTER_VALIDATE_IP)) {
    // call function
} else {
    // deny request
}
```

If we wanted to validate a different non-standard format, then we can use a Regular Expression `regex` with the `preg_match` function. The same can be achieved with `JavaScript` for both the front-end and back-end (i.e. `NodeJS`), as follows:

Code: javascript

```javascript
if(/^(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0
    // call function
}
else{
    // deny request
}
```

Just like `PHP`, with `NodeJS`, we can also use libraries to validate various standard formats, like is-ip for example, which we can install with `npm`, and then use the `isIp(ip)` function in our code. You can read the manuals of other languages, like .NET or Java, to find out how to validate user input on each respective language.

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 our example code, we saw that when we were dealing with character and command filters, it was blacklisting certain words and looking for them in the user input. Generally, this is not a good enough approach to preventing injections, and we should use built-in functions to remove any special characters. We can use `preg_replace` to remove any special characters from the user input, as follows:

Code: php

```php
$ip = preg_replace('/[^A-Za-z0-9.]/', '', $_GET['ip']);
```

As we can see, the above regex only allows alphanumerical characters (`A-Za-z0-9`) and allows a dot character (`.`) as required for IPs. Any other characters will be removed from the string. The same can be done with `JavaScript`, as follows:

Code: javascript

```javascript
var ip = ip.replace(/[^A-Za-z0-9.]/g, '');
```

We can also use the DOMPurify library for a `NodeJS` back-end, as follows:

Code: javascript

```
import DOMPurify from 'dompurify';
var ip = DOMPurify.sanitize(ip);
```

In certain cases, we may want to allow all special characters (e.g., user comments), then we can use the same `filter_var` function we used with input validation, 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. `However, as we have seen in this module, escaping special characters is usually not considered a secure practice, as it can often be bypassed through various techniques`.

For more on user input validation and sanitization to prevent command injections, you may refer to the Secure Coding 101: JavaScript module, which covers how to audit the source code of a web application to identify command injection vulnerabilities, and then works on properly patching these types of vulnerabilities.

Server Configuration

Finally, 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:

- 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`..)
- Abide by the Principle of Least Privilege (PoLP) by running the web server as a low privileged user (e.g. `www-data`)
- Prevent certain functions from being executed by the web server (e.g., in PHP `disable_functions=system,...`)
- Limit the scope accessible by the web application to its folder (e.g. in PHP `open_basedir = '/var/www/html'`)
- Reject double-encoded requests and non-ASCII characters in URLs
- Avoid the use of sensitive/outdated libraries and modules (e.g. PHP CGI)

In the end, even after all of these security mitigations and configurations, we have to perform the penetration testing techniques we learned in this module to see if any web application functionality may still be vulnerable to command injection. As some web applications have millions of lines of code, any single mistake in any line of code may be enough to introduce a vulnerability. So we must try to secure the web application by complementing secure coding best practices with thorough penetration testing.