How does Command Injection work?

Understanding Command Injection

Command injection vulnerabilities arise when a web application processes user-supplied data without proper sanitization. This allows attackers to execute arbitrary operating system commands through the application.

Practical Example: Consider a web application designed to copy a user's file to the /tmp directory using the following code:

```php
<?php

$file_name = $_POST['file_name']
$output = shell_exec('cp $file_name /tmp/');

?>
```

In a secure scenario, if a user uploads a file named letsdefend.txt, the application executes the command normally and copies it to the /tmp folder.

However, an attacker can exploit this by using a malicious filename like **letsdefend;ls;.txt.** In this case, the operating system sees the semicolon **(;)** as a separator that marks the **end of one command** and the start of another.

As a result, the system executes three separate commands instead of one:

1. **cp letsdefend:** The original copy command (which may fail due to incorrect syntax).
2. **ls:** The attacker's injected command to list all files in the directory.
3. **.txt:** An invalid command that the system tries to run but fails with an error.

```
└─$ cp letsdefend;ls;.txt
cp: missing destination file operand after 'letsdefend'
Try 'cp --help' for more information.
crowbar.log  Desktop    Downloads  Pictures  Templates  words.txt
crowbar.out  Documents  Music      Public    Videos
.txt: command not found
```

By using this method, the attacker successfully forces the web server to execute unauthorized commands!

When the malicious payload is processed, the operating system attempts to run the following sequence:

- **Command 1 (cp letsdefend):** This is the initial copy operation. Since the necessary destination parameters are missing due to the injection, this specific command will likely fail to execute correctly.
- **Command 2 (ls):** This is the core of the attack. It is a directory listing command that the attacker injected. Even if the web application does not display the output to the user, the operating system still executes it successfully in the background.
- **Command 3 (.txt):** The system attempts to run ".txt" as a standalone command. Since no such command exists in the operating system, it results in an error message.

As demonstrated, the attacker has successfully achieved Remote Code Execution (RCE) on the web server's operating system. This poses a severe threat.

How Attackers Exploit Command Injection

Attackers exploit this vulnerability by running unauthorized commands directly on the server's operating system. This puts the entire web application and all server data at high risk of being stolen or destroyed.

How to Prevent Command Injection

- **Sanitize All Data**: Never trust user input; always clean and validate every piece of data, including filenames.
- **Least Privilege**: Run the web application with the lowest possible permissions to limit what an attacker can do.
- **Use Containers**: Use technologies like **Docker** to isolate the application from the main system.

How to Detect Command Injection

- **Inspect Every Request**: Check all parts of a web request (parameters, headers, etc.) for suspicious data.pdf].
- **Watch for Terminal Keywords**: Look for common OS commands in user inputs, such as ls, dir, cat, or cp.
- **Identify Known Payloads**: Learn common attack patterns, especially those used to create **Reverse Shells**, to catch them quickly.

Reference: Educational content and examples provided by **Let's Defend** training modules.