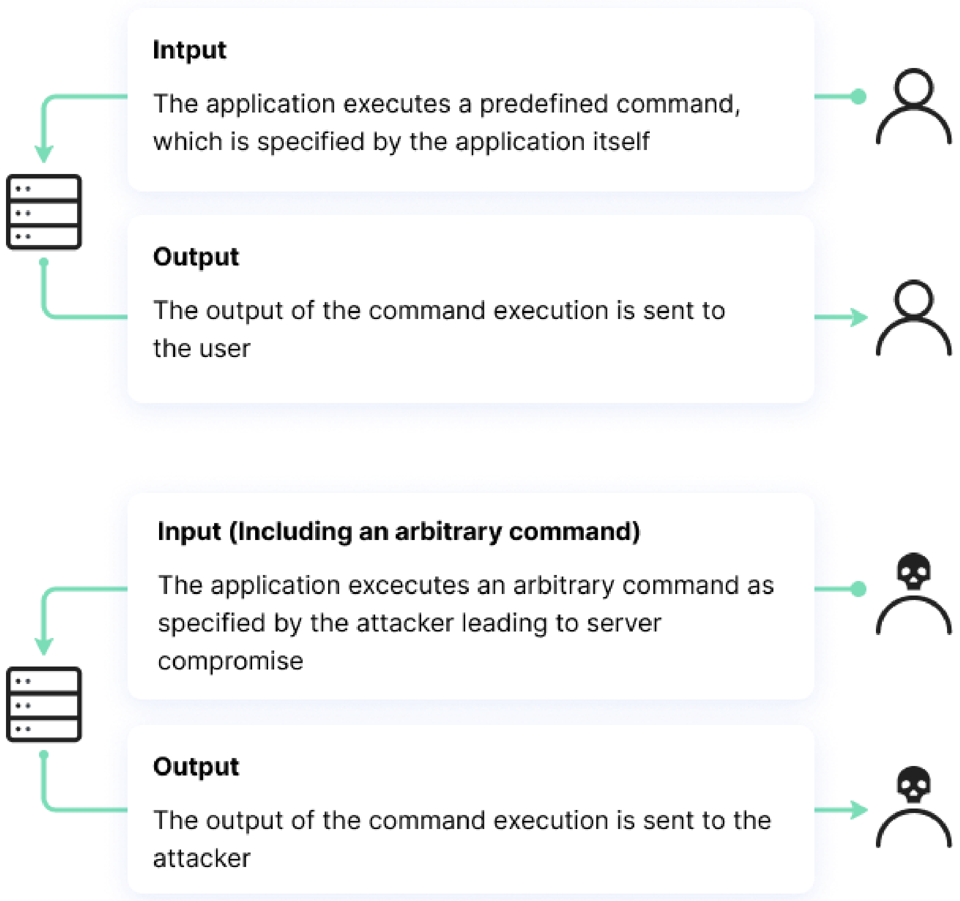
OS Command Injection

Command injection is a cyber-attack that involves executing arbitrary commands on a host operating system (OS). Typically, the threat actor injects the commands by exploiting an application vulnerability, such as insufficient input validation.


Intput
The application executes a predefined command, which is specified by the application itself
Output
The output of the command execution is sent to the user
Input (Including an arbitrary command)
The application excecutes an arbitrary command as specified by the attacker leading to server compromise
Output
The output of the command execution is sent to the attacker

How command injection works – arbitrary commands

For example, a threat actor can use insecure transmissions of user data, such as cookies and forms, to inject a command into the system shell on a web server. The attacker can then leverage the privileges of the vulnerable application to compromise the server.

Command injection takes various forms, including direct execution of shell commands, injecting malicious files into a server's runtime environment, and exploiting vulnerabilities in configuration files, such as XML external entities (XXE).

Command Injection Example:

The developer of the example PHP application wants the user to be able to see the output of the Windows ping command in the web application. The user needs to input the IP address and the application sends ICMP pings to that address. Unfortunately, the developer trusts the user too much and does not perform input validation. The IP address is passed using the GET method and then used in the command line.

```
<?php
$address = $_GET["address"];
$output = shell_exec("ping -n 3 $address");
echo "<pre>$output</pre>";
?>
```

The attacker abuses this script by manipulating the GET request with the following payload:

```
http://example.com/ping.php?address=8.8.8.8%26dir
```

The shell_exec function executes the following OS command: ping -n 3 8.8.8.8&dir. The & symbol in Windows separates OS commands. As a result, the vulnerable application executes an additional command (dir) and displays the command output (directory listing) on-screen:

```
Pinging 8.8.8.8 with 32 bytes of data:
Reply from 8.8.8.8: bytes=32 time=30ms TTL=56
Reply from 8.8.8.8: bytes=32 time=35ms TTL=56
Reply from 8.8.8.8: bytes=32 time=35ms TTL=56
Ping statistics for 8.8.8.8:
    Packets: Sent = 3, Received = 3, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 30ms, Maximum = 35ms, Average = 33ms
 Volume in drive C is OS
 Volume Serial Number is 1337-8055
 Directory of C:\Users\Noob\www
(...)
```

Code Injection vs. Command Injection:

Code Injection:

Code injection is a generic term for any type of attack that involves an injection of code interpreted/executed by an application. This type of attack takes advantage of mishandling of untrusted data inputs. It is made possible by a lack of proper input/output data validation.

A key limitation of code injection attacks is that they are confined to the application or system they target. If an attacker can inject PHP code into an application and execute it, malicious code will be limited by PHP functionality and permissions granted to PHP on the host machine.

Command Injection:

Command injection typically involves executing commands in a system shell or other parts of the environment. The attacker extends the default functionality of a vulnerable application, causing it to pass commands to the system shell, without needing to inject malicious code. In many cases, command injection gives the attacker greater control over the target system.

Command Injection Methods:

Here are some of the vulnerabilities that commonly lead to a command injection attack.

Arbitrary command injection:

Some applications may enable users to run arbitrary commands, and run these commands as is to the underlying host.

Arbitrary file uploads:

If an application allows users to upload files with arbitrary file extensions, these files could include malicious commands. On most web servers, placing such files in the webroot will result in command injection.

Insecure serialization:

Server-side code is typically used to deserialize user inputs. If deserialization is performed without proper verification, it can result in command injection.

Server-side template injection (SSTI):

Many web applications use server-side templates to generate dynamic HTML responses. This makes it possible for attackers to insert malicious server-side templates. SSTI occurs when user input is embedded in a template in an insecure manner, and code is executed remotely on the server.

XML external entity injection (XXE):

XXE occurs in applications that use a poorly-configured XML parser to parse user-controlled XML input. This vulnerability can cause exposure of sensitive data, server-side request forgery (SSRF), or denial of service attacks.

Command Injection Prevention:

Here are several practices you can implement in order to prevent command injections:

- Avoid system calls and user input—to prevent threat actors from inserting characters into the OS command.
- Set up input validation—to prevent attacks like XSS and SQL Injection.
- Create a white list—of possible inputs, to ensure the system accepts only pre-approved inputs.
- Use only secure APIs—when executing system commands such as execFile()
- Use execFile() securely—prevent users from gaining control over the name of the program. You should also map user input to command arguments in a way that ensures user input does not pass as-is into program execution.

Command Injection Prevention:

There are several methods to guarantee your application security and prevent arbitrary command execution via command injection. The simplest and safest one is never to use calls such as shell_exec in PHP to execute any host operating system commands. Instead, you should use the equivalent commands from the programming language. For example, if a developer wants to send mail using PHP on Linux/UNIX, they may be tempted to use the mail command available in the operating system. Instead, they should use the mail() function in PHP.

This approach may be difficult if there is no equivalent command in the programming language. For example, there is no direct way to send ICMP ping packets from PHP. In such cases, you need to use input sanitization before you pass the value to a shell command. As with all types of injections, the safest way is to use a whitelist. For example, in the ping.php script, you could check if the address variable is an IP address:

```
$address = filter_var($_GET["address"], FILTER_VALIDATE_IP);
```

We do not recommend using blacklists because attackers may find a way around them. However, if you absolutely must use a blacklist, you should filter or escape the following special characters:

```
Windows: ( ) < > & * ‘ | = ? ; [ ] ^ ~ ! . ” % @ / \ : + , `

Linux: { } ( ) < > & * ‘ | = ? ; [ ] $ – # ~ ! . ” % / \ : + , `
```

Reference:

- https://www.acunetix.com/blog/web-security-zone/os-command-injection/
- https://www.imperva.com/learn/application-security/command-injection/
- https://cybr.com/beginner-archives/os-command-injections-how-they-work-and-example-techniques/
- https://tvasherbrooke.ngontinh24.com/article/what-is-os-command-injection-and-how-to-prevent-it-web-security-academy