CWE-78 : OS Command Injection

This guide explains OS Command injection in more detail.

Contents:

- OS Command Injection (CWE-78)
- OS Command Injection (CWE-78) coverage by Kiuwan

OS Command Injection (CWE-78)

> **CWE-78** describes **OS Command Injection** as follows:
>
> “The software constructs all or part of an **OS command** using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended OS command when it is sent to a downstream component.”

OS Command injection is, therefore, an attack in which the goal is the execution of arbitrary commands on the host operating system.

These attacks are possible when an application passes unsafe user-supplied data (forms, cookies, HTTP headers, etc.) to a system shell, which is usually executed with the privileges of the vulnerable application.

The impact of command injection attacks ranges from loss of data confidentiality and integrity (such as accessing resources without proper privileges) to unauthorized remote access to the system that hosts the vulnerable application (being able to perform malicious actions such as delete files, add new users, etc.).

Unlike other injection attacks based on specific languages, command injection attacks can occur in any OS (Windows and Unix-based) and affect any programming language that might call OS commands (C /C++, Java, PHP, etc.).

The first remediation should go in the direction of using API calls instead of external commands (if possible) or to ensure that the application runs under a non-privileged account with rights for the intended commands.

Anyway, the main reason that an application is vulnerable to command injection attacks is due to incorrect or insufficient input data validation by the application. Therefore, **the sanitization of user input should always be done**.

In the case of a web app, the URL and form data needs to be sanitized for invalid characters. A **blacklist** of characters is an option, but it may be difficult to think of all of the characters to validate against. A better approach would be based on creating a **whitelist** containing only allowable characters or command list to validate the user input.

Let’s have a look at this very basic example. As you can see, user data is collected through program arguments and directly used to construct an OS command.

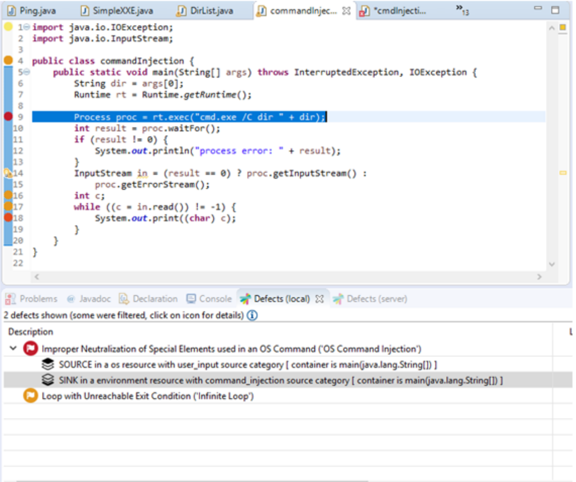
```
public class commandInjection {
       public static void main(String[] args) throws InterruptedException, 
IOException {
             String dir = args[0];
             Runtime rt = Runtime.getRuntime();

             Process proc = rt.exec("cmd.exe /C dir " + dir);
             int result = proc.waitFor();
             if (result != 0) {
                    System.out.println("process error: " + result);
             }
             InputStream in = (result == 0) ? proc.getInputStream() :
                    proc.getErrorStream();
             int c;
             while ((c = in.read()) != -1) {
                    System.out.print((char) c);
             }
       }
}
```

It’s easy to imagine the result of running this program with the next arguments:

```
"c:\tmp > dir.txt & type c:\Windows\system.ini"
```

In this example, the program will display the system.ini configuration file, but the most important thing is that the attacker gains full control on what to do in the attacked system. It’s an open door, a smart hacker will take full advantage of it, no doubt about it.

If you use Kiuwan for Developers, you will be automatically alerted of the vulnerability, with an indication of the sink and the source of the injection.



```
import java.io.IOException;
import java.io.InputStream;

public class commandInjection {
    public static void main(String[] args) throws InterruptedException, IOException {
        String dir = args[0];
        Runtime rt = Runtime.getRuntime();

        Process proc = rt.exec("cmd.exe /C dir " + dir);
        int result = proc.waitFor();
        if (result != 0) {
            System.out.println("process error: " + result);
        }
        InputStream in = (result == 0) ? proc.getInputStream() :
            proc.getErrorStream();
        int c;
        while ((c = in.read()) != -1) {
            System.out.print((char) c);
        }
    }
}
```

2 defects shown (some were filtered, click on icon for details)

Description

- Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
  - SOURCE in a os resource with user_input source category [container is main(java.lang.String[])]
  - SINK in a environment resource with command_injection source category [container is main(java.lang.String[])]
- Loop with Unreachable Exit Condition ('Infinite Loop')

As said above, a check of user data against a whitelist containing only allowed characters (or command list) will remediate the vulnerability.

```
if (Pattern.matches("[0-9A-Za-z@.]+", dir)) {
        Process proc = rt.exec("cmd.exe /C dir " + dir);
}
```

OS Command Injection (CWE-78) coverage by Kiuwan

In Kiuwan, you can search rules covering OS Command Injection (CWE-78) filtering by

- Vulnerability Type = **Injection**, and/or
- CWE tag = CWE:78

Kiuwan incorporates the following rules for OS Command Injection (CWE-78) for the following languages.

To obtain detailed information on functionality, coverage, parameterization, remediation, example codes, etc., follow the same steps as described in SQL Injection.

Language	Rule code
Abap	OPT.ABAP.SEC.CommandInjection
C	OPT.C.CERTC.ENV04
	OPT.C.CERTC.STR02
C#	OPT.CSHARP.CommandInjection
C++	OPT.CPP.CERTC.ENV04
	OPT.CPP.CERTC.STR02
Cobol	OPT.COBOL.SEC.OSCommandInjection
Java	OPT.JAVA.SEC_JAVA.CommandInjectionRule
Javascript	OPT.JAVASCRIPT.CommandInjection

Objective-C	OPT.OBJECTIVEC.DoNotUseSystem
PHP	OPT.PHP.CommandInjection
Python	OPT.PYTHON.SECURITY.CommandInjection
RPG IV	OPT.RPG4.SEC.OSCommandInjection
Swift	OPT.SWIFT.SECURITY.CommandInjection