IDS07-J. Sanitize untrusted data passed to the Runtime.exec() method

External programs are commonly invoked to perform a function required by the overall system. This practice is a form of reuse and might even be considered a crude form of component-based software engineering. Command and argument injection vulnerabilities occur when an application fails to sanitize untrusted input and uses it in the execution of external programs.

Every Java application has a single instance of class `Runtime` that allows the application to interface with the environment in which the application is running. The current runtime can be obtained from the `Runtime.getRuntime()` method. The semantics of `Runtime.exec()` are poorly defined, so it is best not to rely on its behavior any more than necessary, but typically it invokes the command directly without a shell. If you want a shell, you can use `/bin/sh -c` on POSIX or `cmd.exe` on Windows. The variants of `exec()` that take the command line as a single string split it using a `StringTokenizer`. On Windows, these tokens are concatenated back into a single argument string before being executed.

Consequently, command injection attacks cannot succeed unless a command interpreter is explicitly invoked. However, argument injection attacks can occur when arguments have spaces, double quotes, and so forth, or when they start with a `-` or `/` to indicate a switch.

Any string data that originates from outside the program's trust boundary must be sanitized before being executed as a command on the current platform.

Noncompliant Code Example (Windows)

This noncompliant code example provides a directory listing using the `dir` command. It is implemented using `Runtime.exec()` to invoke the Windows `di` command.

```
class DirList {
  public static void main(String[] args) throws Exception {
    String dir = System.getProperty("dir");
    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);
    }
  }
}
```

Because `Runtime.exec()` receives unsanitized data originating from the environment, this code is susceptible to a command injection attack.

An attacker can exploit this program using the following command:

```
java -Ddir='dummy & echo bad' Java
```

The command executed is actually two commands:

```
cmd.exe /C dir dummy & echo bad
```

which first attempts to list a nonexistent `dummy` folder and then prints `bad` to the console.

Noncompliant Code Example (POSIX)

This noncompliant code example provides the same functionality but uses the POSIX `ls` command. The only difference from the Windows version is the argument passed to `Runtime.exec()`.

```
class DirList {
  public static void main(String[] args) throws Exception {
    String dir = System.getProperty("dir");
    Runtime rt = Runtime.getRuntime();
    Process proc = rt.exec(new String[] {"sh", "-c", "ls " + 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);
    }
  }
}
```

The attacker can supply the same command shown in the previous noncompliant code example with similar effects. The command executed is actually

```
sh -c 'ls dummy & echo bad'
```

Compliant Solution (Sanitization)

This compliant solution sanitizes the untrusted user input by permitting only a small group of whitelisted characters in the argument that will be passed to `Runtime.exec()`; all other characters are excluded.

```
// ...
if (!Pattern.matches("[0-9A-Za-z@.]+", dir)) {
  // Handle error
}
// ...
```

Although it is a compliant solution, this sanitization approach rejects valid directories. Also, because the command interpreter invoked is system dependent, it is difficult to establish that this solution prevents command injections on every platform on which a Java program might run.

Compliant Solution (Restricted User Choice)

This compliant solution prevents command injection by passing only trusted strings to `Runtime.exec()`. The user has control over which string is used but cannot provide string data directly to `Runtime.exec()`.

```
// ...
String dir = null;

int number = Integer.parseInt(System.getProperty("dir")); // Only allow integer choices
switch (number) {
  case 1:
    dir = "data1";
    break; // Option 1
  case 2:
    dir = "data2";
    break; // Option 2
  default: // Invalid
    break;
}
if (dir == null) {
  // Handle error
}
```

This compliant solution hard codes the directories that may be listed.

This solution can quickly become unmanageable if you have many available directories. A more scalable solution is to read all the permitted directories from a properties file into a `java.util.Properties` object.

Compliant Solution (Avoid `Runtime.exec()`)

When the task performed by executing a system command can be accomplished by some other means, it is almost always advisable to do so. This compliant solution uses the `File.list()` method to provide a directory listing, eliminating the possibility of command or argument injection attacks.

```
import java.io.File;

class DirList {
  public static void main(String[] args) throws Exception {
    File dir = new File(System.getProperty("dir"));
    if (!dir.isDirectory()) {
      System.out.println("Not a directory");
    } else {
      for (String file : dir.list()) {
        System.out.println(file);
      }
    }
  }
}
```

Risk Assessment

Passing untrusted, unsanitized data to the `Runtime.exec()` method can result in command and argument injection attacks.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
IDS07-J	High	Probable	Medium	P12	L1

Automated Detection

Tool	Version	Checker	Description
The Checker Framework	2.1.3	**Tainting Checker**	Trust and security errors (see Chapter 8)
Coverity	7.5	**OS_CMD_INJECTION**	Implemented
Parasoft Jtest	2020.2	**PORT.EXEC**	Do not use 'Runtime.exec()'
SonarQube	6.7	**S2076**	OS commands should not be vulnerable to injection attacks

Related Vulnerabilities

CVE-2010-0886	Sun Java Web Start Plugin Command Line Argument Injection
CVE-2010-1826	Command injection in `updateSharingD`'s handling of Mach RPC messages
T-472	Mac OS X Java Command Injection Flaw in `updateSharingD` lets local users gain elevated privileges

Related Guidelines

SEI CERT C Coding Standard	ENV03-C. Sanitize the environment when invoking external programs ENV33-C. Do not call system()
SEI CERT C++ Coding Standard	ENV03-CPP. Sanitize the environment when invoking external programs VOID ENV02-CPP. Do not call system() if you do not need a command processor
SEI CERT Perl Coding Standard	IDS34-PL. Do not pass untrusted, unsanitized data to a command interpreter
ISO/IEC TR 24772:2013	Injection [RST]
MITRE CWE	CWE-78, Improper Neutralization of Special Elements Used in an OS Command ("OS Command Injection")

Android Implementation Details

`Runtime.exec()` can be called from Android apps to execute operating system commands.

Bibliography

[Chess 2007]	Chapter 5, "Handling Input," section "Command Injection"
[OWASP 2005]	A Guide to Building Secure Web Applications and Web Services
[Permissions 2008]	Permissions in the Java™ SE 6 Development Kit (JDK)
[Seacord 2015]	IDS07-J. Do not pass untrusted, unsanitized data to the Runtime.exec() method LiveLesson

---