# CWE Detail – CWE-114

## Description

Executing commands or loading libraries from an untrusted source or in an untrusted environment can cause an application to execute malicious commands (and payloads) on behalf of an attacker.

## Extended Description

Process control vulnerabilities take two forms: An attacker can change the command that the program executes: the attacker explicitly controls what the command is. An attacker can change the environment in which the command executes: the attacker implicitly controls what the command means. Process control vulnerabilities of the first type occur when either data enters the application from an untrusted source and the data is used as part of a string representing a command that is executed by the application. By executing the command, the application gives an attacker a privilege or capability that the attacker would not otherwise have.

## Threat-Mapped Scoring

Score: 1.8

Priority: P4 - Informational (Low)

## Related Attack Patterns (CAPEC)

* CAPEC-108
* CAPEC-640

## Attack TTPs

**•** T1505.005: Terminal Services DLL (Tactics: persistence)

**•** T1620: Reflective Code Loading (Tactics: defense-evasion)

**•** T1574.006: Dynamic Linker Hijacking (Tactics: persistence, privilege-escalation, defense-evasion)

**•** T1574.013: KernelCallbackTable (Tactics: persistence, privilege-escalation, defense-evasion)

## Modes of Introduction

**•** Implementation: REALIZATION: This weakness is caused during implementation of an architectural security tactic.

## Common Consequences

**•** Impact: Execute Unauthorized Code or Commands — Notes:

## Potential Mitigations

**•** Architecture and Design: Libraries that are loaded should be well understood and come from a trusted source. The application can execute code contained in the native libraries, which often contain calls that are susceptible to other security problems, such as buffer overflows or command injection. All native libraries should be validated to determine if the application requires the use of the library. It is very difficult to determine what these native libraries actually do, and the potential for malicious code is high. In addition, the potential for an inadvertent mistake in these native libraries is also high, as many are written in C or C++ and may be susceptible to buffer overflow or race condition problems. To help prevent buffer overflow attacks, validate all input to native calls for content and length. If the native library does not come from a trusted source, review the source code of the library. The library should be built from the reviewed source before using it. (Effectiveness: N/A)

## Applicable Platforms

**•** None (Class: Not Language-Specific, Prevalence: Undetermined)

## Demonstrative Examples

**•** The problem here is that System.loadLibrary() accepts a library name, not a path, for the library to be loaded. From the Java 1.4.2 API documentation this function behaves as follows [1]: A file containing native code is loaded from the local file system from a place where library files are conventionally obtained. The details of this process are implementation-dependent. The mapping from a library name to a specific filename is done in a system-specific manner. If an attacker is able to place a malicious copy of library.dll higher in the search order than file the application intends to load, then the application will load the malicious copy instead of the intended file. Because of the nature of the application, it runs with elevated privileges, which means the contents of the attacker's library.dll will now be run with elevated privileges, possibly giving them complete control of the system.

**•** The code in this example allows an attacker to load an arbitrary library, from which code will be executed with the elevated privilege of the application, by modifying a registry key to specify a different path containing a malicious version of INITLIB. Because the program does not validate the value read from the environment, if an attacker can control the value of APPHOME, they can fool the application into running malicious code.

**•** The problem is that the program does not specify an absolute path for liberty.dll. If an attacker is able to place a malicious library named liberty.dll higher in the search order than file the application intends to load, then the application will load the malicious copy instead of the intended file. Because of the nature of the application, it runs with elevated privileges, which means the contents of the attacker's liberty.dll will now be run with elevated privileges, possibly giving the attacker complete control of the system. The type of attack seen in this example is made possible because of the search order used by LoadLibrary() when an absolute path is not specified. If the current directory is searched before system directories, as was the case up until the most recent versions of Windows, then this type of attack becomes trivial if the attacker can execute the program locally. The search order is operating system version dependent, and is controlled on newer operating systems by the value of the registry key: HKLM\System\CurrentControlSet\Control\Session Manager\SafeDllSearchMode

## Notes

**•** Maintenance: CWE-114 is a Class, but it is listed a child of CWE-73 in view 1000. This suggests some abstraction problems that should be resolved in future versions.

**•** Maintenance: This entry seems more attack-oriented, or organized around common legitimate behaviors (process invocation) instead of the mistakes in those behaviors. There is likely too much overlap with other CWEs including CWE-73, CWE-426, CWE-427, or other weaknesses related to process invocation.