**Command Injection**

Executing commands from an untrusted source or in an untrusted environment can cause an application to execute malicious commands on behalf of an attacker.

**Explanation**

Command injection 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.

In this case we are primarily concerned with the first scenario, the possibility that an attacker may be able to control the command that is executed. Command injection vulnerabilities of this type occur when:

* Data enters the application from an untrusted source.
* The data is used as or 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.

**Example**

The following code from a system utility uses the system property APPHOME to determine the directory in which it is installed and then executes an initialization script based on a relative path from the specified directory.

...

String home = System.getProperty("APPHOME");

String cmd = home + INITCMD;

java.lang.Runtime.getRuntime().exec(cmd);

...

The code in Example 1 allows an attacker to execute arbitrary commands with the elevated privilege of the application by modifying the system property APPHOME to point to a different path containing a malicious version of INITCMD. Because the program does not validate the value read from the environment, if an attacker can control the value of the system property APPHOME, then they can fool the application into running malicious code and take control of the system.

**Example**

The following code is from an administrative web application designed to allow users to kick off a backup of an Oracle database using a batch-file wrapper around the rman utility and then run a cleanup.bat script to delete some temporary files. The script rmanDB.bat accepts a single command line parameter, which specifies the type of backup to perform. Because access to the database is restricted, the application runs the backup as a privileged user.

...

String btype = request.getParameter("backuptype");

String cmd = new String("cmd.exe /K

\"c:\\util\\rmanDB.bat "+btype+"&&c:\\util\\cleanup.bat\"")

System.Runtime.getRuntime().exec(cmd);

...

The problem here is that the program does not do any validation on the backuptype parameter read from the user. Typically the Runtime.exec() function will not execute multiple commands, but in this case the program first runs the cmd.exe shell in order to run multiple commands with a single call to Runtime.exec(). Once the shell is invoked, it will allow for the execution of multiple commands separated by two ampersands. If an attacker passes a string of the form "&& del c:\\dbms\\\*.\*", then the application will execute this command along with the others specified by the program. Because of the nature of the application, it runs with the privileges necessary to interact with the database, which means whatever command the attacker injects will run with those privileges as well.

**Example**

The following code is from a web application that allows users access to an interface through which they can update their password on the system. Part of the process for updating passwords in certain network environments is to run a make command in the /var/yp directory, the code for which is shown below.

...

System.Runtime.getRuntime().exec("make");

...

The problem here is that the program does not specify an absolute path for make and fails to clean its environment prior to executing the call to Runtime.exec(). If an attacker can modify the $PATH variable to point to a malicious binary called make and cause the program to be executed in their environment, then the malicious binary will be loaded instead of the one intended. Because of the nature of the application, it runs with the privileges necessary to perform system operations, which means the attacker's make will now be run with these privileges, possibly giving the attacker complete control of the system.

Some think that in the mobile world, classic vulnerabilities, such as command injection, do not make sense -- why would a user attack him or herself? However, keep in mind that the essence of mobile platforms is applications that are downloaded from various sources and run alongside each other on the same device. The likelihood of running a piece of malware next to a banking application is high, which necessitates expanding the attack surface of mobile applications to include inter-process communication.

**Example**

The following code reads commands to be executed from an Android intent.

...

String[] cmds = this.getIntent().getStringArrayExtra("commands");

Process p = Runtime.getRuntime().exec("su");

DataOutputStream os = new DataOutputStream(p.getOutputStream());

for (String cmd : cmds) {

os.writeBytes(cmd+"\n");

}

os.writeBytes("exit\n");

os.flush();

**Recommendations**

An attacker can indirectly control commands executed by a program by modifying the environment in which they are executed. The environment should not be trusted and precautions should be taken to prevent an attacker from using some manipulation of the environment to perform an attack. Whenever possible, commands should be controlled by the application and executed using an absolute path. In cases where the path is not known at compile time, such as for cross-platform applications, an absolute path should be constructed from trusted values during execution. Command values and paths read from configuration files or the environment should be sanity-checked against a set of invariants that define valid values.

Other checks can sometimes be performed to detect if these sources may have been tampered with. For example, if a configuration file is world-writable, the program might refuse to run. In cases where information about the binary to be executed is known in advance, the program may perform checks to verify the identity of the binary. If a binary should always be owned by a particular user or have a particular set of access permissions assigned to it, these properties can be verified programmatically before the binary is executed.

In the end it may be impossible for a program to fully protect itself from an imaginative attacker bent on controlling the commands it executes. You should strive to identify and protect against every conceivable manipulation of input values and the environment. The goal should be to shut down as many attack vectors as possible.

**References**

* http://www.hpenterprisesecurity.com/vulncat/en/vulncat/java/command\_injection.html