Command Injection

Description

Command injection is an attack in which the goal is execution of arbitrary commands on the host operating system via a vulnerable application. Command injection attacks are possible when an application passes unsafe user supplied data (forms, cookies, HTTP headers etc.) to a system shell. In this attack, the attacker-supplied operating system commands are usually executed with the privileges of the vulnerable application. Command injection attacks are possible largely due to insufficient input validation.

This attack differs from Code Injection, in that code injection allows the attacker to add their own code that is then executed by the application. In Command Injection, the attacker extends the default functionality of the application, which execute system commands, without the necessity of injecting code.

Examples

Example 1

The following code is a wrapper around the UNIX command *cat* which prints the contents of a file to standard output. It is also injectable:

```
#include <stdio.h>
#include <unistd.h>

int main(int argc, char **argv) {
 char cat[] = "cat ";
 char *command;
 size_t commandLength;

 commandLength = strlen(cat) + strlen(argv[1]) + 1;
 command = (char *) malloc(commandLength);
 strncpy(command, cat, commandLength);
 strncat(command, argv[1], (commandLength - strlen(cat)) );

 system(command);
 return (0);
}
```

Used normally, the output is simply the contents of the file requested:

```
$ ./catWrapper Story.txt
When last we left our heroes...
```

However, if we add a semicolon and another command to the end of this line, the command is executed by catWrapper with no complaint:

```
$ ./catWrapper "Story.txt; ls"
When last we left our heroes...
Story.txt               doubFree.c              nullpointer.c
unstosig.c              www*                    a.out*
format.c                strlen.c                useFree*
catWrapper*             misnull.c               strlength.c             useFree.c
commandinjection.c      nodefault.c             trunc.c                 writeWhatWhere.c
```

If catWrapper had been set to have a higher privilege level than the standard user, arbitrary commands could be executed with that higher privilege.

Example 2

The following simple program accepts a filename as a command line argument, and displays the contents of the file back to the user. The program is installed setuid root because it is intended for use as a learning tool to allow system administrators in-training to inspect privileged system files without giving them the ability to modify them or damage the system.

```
int main(char* argc, char** argv) {
 char cmd[CMD_MAX] = "/usr/bin/cat ";
 strcat(cmd, argv[1]);
 system(cmd);
}
```

Because the program runs with root privileges, the call to system() also executes with root privileges. If a user specifies a standard filename, the call works as expected. However, if an attacker passes a string of the form ";rm -rf /", then the call to system() fails to execute cat due to a lack of arguments and then plows on to recursively delete the contents of the root partition.

Example 3

The following code from a privileged program uses the environment variable $APPHOME to determine the application's installation directory, and then executes an initialization script in that directory.

```
...
char* home=getenv("APPHOME");
char* cmd=(char*)malloc(strlen(home)+strlen(INITCMD));
if (cmd) {
 strcpy(cmd,home);
 strcat(cmd,INITCMD);
 execl(cmd, NULL);
}
...
```

As in Example 2, the code in this example allows an attacker to execute arbitrary commands with the elevated privilege of the application. In this example, the attacker can modify the environment variable $APPHOME to specify a different path containing a malicious version of INITCMD. Because the program does not validate the value read from the environment, by controlling the environment variable, the attacker can fool the application into running malicious code.

The attacker is using the environment variable to control the command that the program invokes, so the effect of the environment is explicit in this example. We will now turn our attention to what can happen when the attacker changes the way the command is interpreted.

Example 4

The code below is from a web-based CGI utility that allows users to change their passwords. The password update process under NIS includes running *make* in the /var/yp directory. Note that since the program updates password records, it has been installed setuid root.

The program invokes make as follows:

```
system("cd /var/yp && make &> /dev/null");
```

Unlike the previous examples, the command in this example is hardcoded, so an attacker cannot control the argument passed to system(). However, since the program does not specify an absolute path for make, and does not scrub any environment variables prior to invoking the command, the attacker can modify their $PATH variable to point to a malicious binary named make and execute the CGI script from a shell prompt. And since the program has been installed setuid root, the attacker's version of make now runs with root privileges.

The environment plays a powerful role in the execution of system commands within programs. Functions like system() and exec() use the environment of the program that calls them, and therefore attackers have a potential opportunity to influence the behavior of these calls.

There are many sites that will tell you that Java's Runtime.exec is exactly the same as C's system function. This is not true. Both allow you to invoke a new program/process. However, C's system function passes its arguments to the shell (/bin/sh) to be parsed, whereas Runtime.exec tries to split the string into an array of words, then executes the first word in the array with the rest of the words as parameters. Runtime.exec does NOT try to invoke the shell at any point. The key difference is that much of the functionality provided by the shell that could be used for mischief (chaining commands using "&", "&&", "|", "||", etc, redirecting input and output) would simply end up as a parameter being passed to the first command, and likely causing a syntax error, or being thrown out as an invalid parameter.

Example 5

The following trivial code snippets are vulnerable to OS command injection on the Unix/Linux platform:

C:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv)
{
     char command[256];

     if(argc != 2) {
          printf("Error: Please enter a program to time!\n");
          return -1;
     }

     memset(&command, 0, sizeof(command));

     strcat(command, "time ./");
     strcat(command, argv[1]);

     system(command);
     return 0;
}
```

If this were a suid binary, consider the case when an attacker enters the following: `ls; cat /etc/shadow` . In the Unix environment, shell commands are separated by a semi-colon. We now can execute system commands at will!

Java:

There are many sites that will tell you that Java's `Runtime.exec` is exactly the same as C's system function. This is not true. Both allow you to invoke a new program/process. However, C's system

function passes its arguments to the shell (`/bin/sh`) to be parsed, whereas Runtime.exec tries to split the string into an array of words, then executes the first word in the array with the rest of the words as parameters. Runtime.exec does NOT try to invoke the shell at any point. The key difference is that much of the functionality provided by the shell that could be used for mischief (chaining commands using `&` , `&&` , `|` , `||` , etc, redirecting input and output) would simply end up as a parameter being passed to the first command, and likely causing a syntax error, or being thrown out as an invalid parameter.

Example 6

The following PHP code snippet is vulnerable to a command injection attack:

"); $file=$_GET['filename']; system("rm $file"); ?>
The following request and response is an example of a successful attack:

Request `http://127.0.0.1/delete.php?filename=bob.txt;id`

Response

```
Please specify the name of the file to delete

uid=33(www-data) gid=33(www-data) groups=33(www-data)
```

Sanitizing Input

```
Replace or Ban arguments with ";"
Other shell escapes available
Example:
-  &&
-  |
-  ...
```

Related Controls

Ideally, a developer should use existing API for their language. For example (Java): Rather than use `Runtime.exec()` to issue a 'mail' command, use the available Java API located at `javax.mail.*` .

If no such available API exists, the developer should scrub all input for malicious characters. Implementing a positive security model would be most efficient. Typically, it is much easier to define the legal characters than the illegal characters.

References

- CWE-77: Command Injection
- CWE-78: OS Command Injection

Category:Injection