Talos Vulnerability Report

TALOS-2022-1524

Blynk Blynk-Library BlynkConsole.h runCommand stack-based buffer overflow vulnerability

JUNE 15, 2022

CVE NUMBER

CVE-2022-29496

Summary

A stack-based buffer overflow vulnerability exists in the BlynkConsole.h runCommand functionality of Blynk -Library v1.0.1. A specially-crafted network request can lead to command execution. An attacker can send a network request to trigger this vulnerability.

Tested Versions

Blynk -Library v1.0.1

Product URLs

Blynk-Library - https://github.com/blynkkk/blynk-library

CVSSv3 Score

9.0 - CVSS:3.0/AV:N/AC:H/PR:N/UI:N/S:C/C:H/I:H/A:H

CWE

CWE-121 - Stack-based Buffer Overflow

Details

Blynk-Library is a small library for connecting more than 400 different embedded device models into a private or enterprise Blynk-Server instance. According to the git repository, it is the “most popular internet-of-things platform for connecting any hardware to the cloud.”

The Blynk-Library manages different types of commands. If some specific commands are issued to the device the library goes through a function called `runCommand` to parse the request and perform the operation, if the command is supported. The `runCommand` function:

```
ProcessResult runCommand(char* cmd) {
        char* argv[8] = { 0, };
[1]
        int argc = split_argv(cmd, argv);
[2]
        if (argc <= 0) {
        [...]
```

The first thing this function does is split the provided command into different arguments. At `[2]` the function `split_argv` is called. This function replaces each spaces with a null terminator; this will separate the command string into multiple strings. The pointers of each created string are placed in each position of the char pointer array at `[1]`.

The `split_argv` function does not know how long the `runCommand`'s `argv` buffer is, because the received command can have more than 7 spaces. This means that the `split_argv` function can overflow the stack buffer based on the received command. This can lead to overwriting the return address of the `runCommand` function with a value that points to the data of the `cmd` buffer, based on the architecture. This issue can lead to arbitrary command execution.

For example, the stack dump at `[2]` looks like:

```
$ x/20dwx $sp
0x20007e08:     0x20001eb8      0x2000067c      0x20007e18      0x00000000
0x20007e18:     0x00000000      0x00000000      0x00000000      0x00000000
0x20007e28:     0x00000000      0x00000000      0x00000000      0x00017c01
0x20007e38:     0x00000024      0x000131f1      0x00000024      0x00000018
0x20007e48:     0x20007e50      0x00006687      0x20007f04      0x20007f28
```

At 0x20007e14 starts the `argv` buffer. After 8 elements are parsed, by the `split_argv` function, the same portion of memory looks like this:

```
$ x/20dwx 0x20007e08
0x20007e08:     0x20001eb8      0x2000067c      0x20007e18      0x20001eb8
0x20007e18:     0x20001eba      0x20001ebc      0x20001ebe      0x20001ec0
0x20007e28:     0x20001ec2      0x20001ec4      0x20001ec6      0x00017c01
0x20007e38:     0x00000024      0x000131f1      0x00000024      0x00000018
0x20007e48:     0x20007e50      0x00006687      0x20007f04      0x20007f28
```

The value at 0x20007e4c, 0x00006687, is the return address of the function `runCommand`:

```
$ x/3i 0x00006687
0x6687 <BlynkWidgetWriteInternalPinDBG(BlynkReq&, BlynkParam const&)+110>:      adds
r3, r7, r6
0x6689 <BlynkWidgetWriteInternalPinDBG(BlynkReq&, BlynkParam const&)+112>:      movs
r0, r3
0x668b <BlynkWidgetWriteInternalPinDBG(BlynkReq&, BlynkParam const&)+114>:      bl
0x131d6 <arduino::String::~String()>
```

At some point one of the arguments parsed will overwrite that return address:

```
$x/20dwx 0x20007e08
0x20007e08:     0x20001eb8      0x2000067c      0x20007e18      0x20001eb8
0x20007e18:     0x20001eba      0x20001ebc      0x20001ebe      0x20001ec0
0x20007e28:     0x20001ec2      0x20001ec4      0x20001ec6      0x20001ec8
0x20007e38:     0x20001eca      0x20001ecc      0x20001ece      0x20001ed0
0x20007e48:     0x20001ed2      0x20001ed5      0x00000000      0x20007f28
```

And at 0x20001ed5 there are fully controllable data, so if the architecture is such that the parsed argument is located in a memory portion that is executable, that memory will eventually be executed.

Timeline

2022-06-08 - Vendor Disclosure
2022-06-15 - Public Release

CREDIT

Discovered by Francesco Benvenuto of Cisco Talos.