Force1 Discovery WiFi U818A FPV Drone Exploitation

CVE-2022-40918

Eric Kapitanski

What Is the Vulnerability?

There is a buffer overflow that when exploited allows remote code execution (RCE) as root user on the device.

Responsible Disclosure

Venders/manufacturers were notified approximately 90 days before public release, and had they requested additional time to patch the vulnerability (or indicated any desire to patch it), it would have been granted.

Acknowledgements:

The following exploit builds heavily upon the work of meekworth and tjhorner (links below). Special thanks to Boston Cybernetics Institute (BCI) who funded this effort.

- Boston Cybernetics Institute - https://www.bostoncyber.org/
- meekworth https://medium.com/@meekworth/exploiting-the-lw9621-drone-camera-module-773f00081368
- tjhorner https://www.npmjs.com/package/dronelib

How Does an Attacker Exploit This Vulnerability?

A Buffer overflow in the firmware “lewei_cam V1.0 @Jun 30 2016_09:22:37” of the Force1 Discovery WiFi U818A FPV Drone, allows attackers to gain remote code execution as a root user via two specifically crafted TCP packets over its unauthenticated and open WIFI network.

To successfully exploit the drone, an attacker connects to the drone’s open and unauthenticated network “udirc-FPV-5A20C2” and sends the specially crafted TCP packets. The packets will trigger a buffer overflow in a process (lewei_cam) run as root on the device allowing for remote code execution as the root user. There are no stack guards and the stack is executable, so an attacker simply needs to include the shell code they would like to run in the payload of the attack packet and jump to its location on the stack. However, due to multithreading and other issues, the location of the stack in the program’s memory does change between runs.

To address this issue of the non-static stack address, the exploit makes use of multiple vulnerabilities, one of which allows for arbitrary file downloads. The attacker is able to download the PID of the lewei_cam process by downloading “/proc/self/stats” and then is able to download the memory map of

the lewei_cam process by downloading "/proc/{PID}/maps" or "/proc/self/maps." From there, an examination of the file will show the location of the multiple stacks in memory of the process. The attacker can then use that information to determine which stack their payload should show up in, and the location in memory where the shellcode will be placed. The attacker then uses that information to determine the address in memory that the buffer overflow should execute to ensure remote code execution. The stack finding portion of the exploit is executed first, and the information from that is used to update the buffer overflow portion of the exploit.

In examining the above vulnerabilities, the drone firmware was extracted, the binary was run in an emulated version of the drone's hardware, and vulnerabilities were discovered and examined. The ultimate findings were then verified on the physical drone itself, and a POC was produced.

Exploitation

In order to exploit the drone, an attacker first connects to the drone's open and unauthenticated network "udirc-FPV-5A20C2." From there, an attacker simply run the provided proof of concept, which will add a new root user "shell-storm" to the drone. The attacker can then log in via telnet on port 23 as the "shell-storm" user using password "toor".

Detailed Description

The drone creates an open and unauthenticated Wi-Fi network "udirc-FPV-5A20C2," which a user connects to with a smartphone and controls the drone via an application run on their phone.

Figure 1: Drone open and unathenticated Wi-Fi network

Utilizing Wi-Fi sniffing I was able to observe interactions between the drone and the smartphone application used to control it. Although there are multiple ports open on the drone, in the interest of brevity, TCP communications on port 8060 will be the focus of this analysis.

In the figures below, you can see packet captures of communications between the smartphone app and the drone itself. The drone has IP "192.168.0.1" and the controlling app has the IP "192.168.0.21." Although the capture shows communications to this port begin with the null-terminated string "lewei_cmd," there are no checks on this in the code and the "lewei_cmd" string is not necessary. However, the number immediately following it at bytes 22-25 is important, as this indicates the command type (explained below).

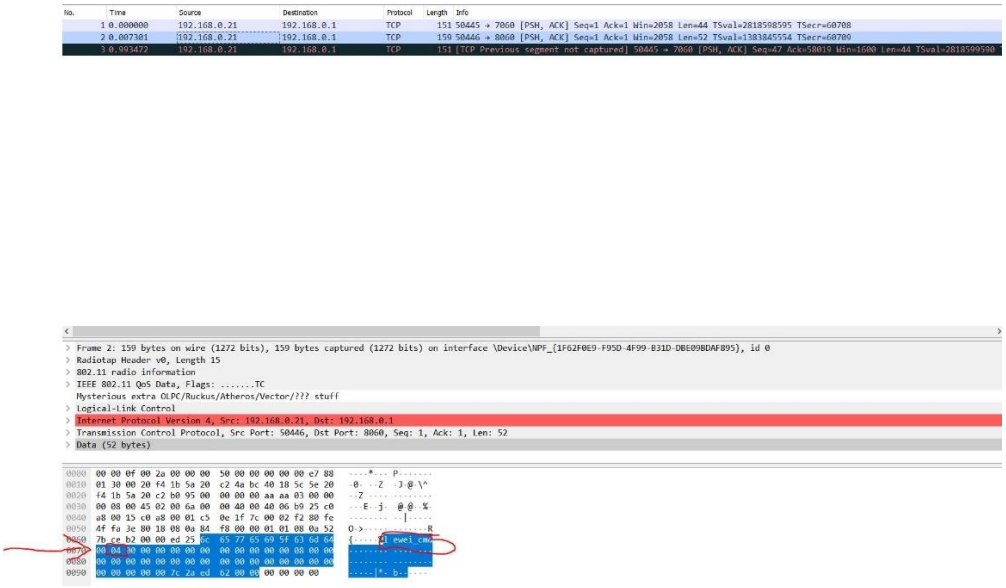

Figure 2: Port 8060 lewei_cmd 0x04

The code corresponding to these commands is presented below:

Inside the function lewei_tcp_cmd_thread you can observe a socket on port 8060 being opened.

```
Decompile: lewei_tcp_cmd_thread - (lewei_cam)
 4 void lewei_tcp_cmd_thread(void)
 5
 6 {
 7   uint local_1bc;
 8   uint auStack416 [57];
 9   timeval local_bc;
10   fd_set fStack180;
11   int local_34;
12   uint TCP_8060_socket;
13   uint TCP_8060;
14   uint local_28;
15   uint local_24;
16   ssize_t local_20;
17   void *local_1c;
18   uint local_18;
19   fd_set *local_14;
20   uint local_10;
21   fd_set *local_c;
22
23   TCP_8060_socket = do_init_socket(0x1f7c,3,5);
```

Figure 3: Opening TCP port 8060

lewei_tcp_cmd_thread then attempts to read 0x2e (46) bytes from the connection. If this succeeds (and exactly 46 bytes are read), lewei_cmd_execute is called as shown below. We will refer to these initial 46

bytes sent as payload1.

```
Decompile: lewei_tcp_cmd_thread - (lewei_cam)
57                  local_20 = read(auStack416[local_28 * 0x13],local_1c,0x2e);
58                  if (local_20 == 0x2e) {
59                    lewei_cmd_excute(auStack416[local_28 * 0x13],local_1c,0x2e);
60                  }
```

Figure 4: Initial read from port 8060

Inside lewei_cmd_execute, payload1 is processed. The variable used for switch cases is read from the bytes 22-25 in payload1, which for this exploit should be set to 4. Then, local_9a (the value used as the "size" argument to another call to net_recv), as shown below is read from bytes 10-13 in payload1.

```
Decompile: lewei_cmd_excute - (lewei_cam)
 96      case 4:
 97        local_c0 = 0;
 98        uStack188 = 0;
 99        local_80 = net_recv(param_1,&local_c0,WE_CONTROL_local_9a);
100        if (local_80 == WE_CONTROL_local_9a) {
```

Figure 5: Switch statement case 4, and additional read from port 8060

At this point, there will be another read to port 8060 of size local_9a. We will refer to this second read as payload2.

It is important to note that local_c0 is a single 4 byte variable, yet the address of this variable is where the data read from net_recv will be copied. This is where the buffer overflow occurs.

The value assigned to local_80 corresponds to the actual number of bytes read from payload2. In our case, we need the actual number of bytes read (local_80) to differ from the size argument to net_recv, which we control with payload1 (local_9a). We can accomplish this by setting a larger value for local_9a than we intend to send in payload2.

If local_9a (the size argument to net_recv) is not equal to local_80 (the number of bytes actually read), there will be a call made to free with local_7c as the argument. Local_7c is read from bytes 68-71 in payload2. We set these bytes to 0, which will cause the call to free to successfully return (although it won't do anything useful).

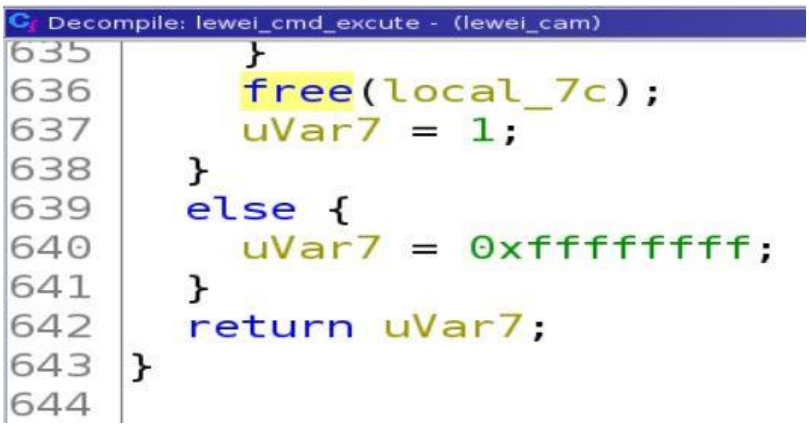

Figure 6: Call to free, and upon return, program counter control

At this point, our payload2 has completely overwritten the return address that will be popped off of the stack into the program counter after the call to free, and we now control program execution. As there is no ASLR, stack canary, or stack execution prevention, we can simply append our shellcode to payload2, and jump to it to gain remote code execution as the root user.

As previously mentioned, there are a few additional steps that must be taken to successfully exploit the drone. The main one is that the program appears to create threads in a non-deterministic manner. This causes issues for our exploit because the location of the stack associated with the thread that is reading from port 8060 changes. In order to address this issue, we utilize another exploit which allows for arbitrary file downloads from the done as discovered by tjhorner and meekworth (cited above). You can also refer to the provided proof of concept for further details. We utilize the arbitrary file download vulnerability to download /proc/self/maps, which details the addresses of the various thread stacks, which allow us to locate our shellcode in overall program memory (as it is at a fixed offset from the top of the thread stack).