Talos Vulnerability Report

TALOS-2021-1378

Anker Eufy Homebase 2 home_security CMD_DEVICE_GET_SERVER_LIST_REQUEST out-of-bounds write vulnerability

NOVEMBER 29, 2021

CVE NUMBER

CVE-2021-21950,CVE-2021-21951

SUMMARY

An out-of-bounds write vulnerability exists in the CMD_DEVICE_GET_SERVER_LIST_REQUEST functionality of the home_security binary of Anker Eufy Homebase 2 2.1.6.9h. A specially-crafted network packet can lead to code execution.

CONFIRMED VULNERABLE VERSIONS

The versions below were either tested or verified to be vulnerable by Talos or confirmed to be vulnerable by the vendor.

Anker Eufy Homebase 2 2.1.6.9h

PRODUCT URLS

Eufy Homebase 2 - https://us.eufylife.com/products/t88411d1

CVSSV3 SCORE

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

CWE

CWE-119 - Improper Restriction of Operations within the Bounds of a Memory Buffer

DETAILS

The Eufy Homebase 2 is the video storage and networking gateway that enables the functionality of the Eufy Smarthome ecosystem. All Eufy devices connect back to this device, and this device connects out to the cloud, while also providing assorted services to enhance other Eufy Smarthome devices.

The Eufy Homebase 2's `home_security` binary is a central cog in the device, spawning inordinate amount of pthreads immediately after executing, each with their own little task. For the purposes of this advisory, we care solely about the pthread in charge of a particular cloud connectivity occurring with IP address `18.224.66.194` on UDP port 8006. An example of such traffic is shown below:

```
// device -> cloud
0000   58 5a fe b9 0b 00 00 00 59 5e 42 61 01 00 00 00   XZ......Y^Ba....
0010   00 00 01 00 54 38 30 31 30 4e 31 32 33 34 35 36   ....T8010N123456
0020   37 48 39 3A 00                                    789A.
```

This particular packet is the `CMD_DEVICE_HEARTBEAT_CHECK`, and the server's response is seen below:

```
// cloud -> device response
0000   58 5a 32 b2 0b 00 1d 00 59 5e 42 61 01 00 01 00   XZ2.....Y^Ba....
0010   00 00 01 00 54 38 30 31 30 4e 31 32 33 34 35 36   ....T8010N123456
0020   38 48 39 3a 00 7b 22 64 65 76 69 63 65 5f 69 70   789a.{"device_ip
0030   22 3a 22 37 31 2e 31 36 32 2e 32 33 37 2e 33 34   ":"71.162.237.34
0040   22 7d                                             "}
```

While there is some interesting information already visible, reversing the protocol and viewing with a decoder is much more informative:

```
[>_>] ---Pushpkt---
Magic       : 0x5a58
CRC         : 0x1234
Opcode      : 0x000b (CMD_DEVICE_HEARTBEAT_CHECK)
Bodylen     : 0x0000
Time (unix) : 1632154786
msg_ver     : 0x0001
is_resp     : 0x00
idk_lol     : 0x00
idk_lol2    : 0x0000
non_zero    : 0x0001
Hub SN      : T8010N123456789a\x00

[<_<] response pkt:
[>_>] ---Pushpkt---
Magic       : 0x5a58
CRC         : 0x5678
Opcode      : 0x000b (CMD_DEVICE_HEARTBEAT_CHECK)
Bodylen     : 0x001d
Time (unix) : 1632154746
msg_ver     : 0x0001
is_resp     : 0x01
idk_lol     : 0x00
idk_lol2    : 0x0000
non_zero    : 0x0001
Hub SN      : T8010N123456789a\x00
Msgbody      : {"device_ip":"71.162.237.34"}
```

While this specific command doesn't particularly do much, there does exist a decent amount of other opcodes to interact with:

```
opcode_dict = {
    0xb   :  "CMD_DEVICE_HEARTBEAT_CHECK",
    0xc   :  "CMD_DEVICE_GET_SERVER_LIST_REQUEST", // [1]
    0xd   :  "CMD_DEVICE_GET_RSA_KEY_REQUEST",     // [2]
    0x22  :  "CMD_SERVER_GET_AES_KEY_INFO",
    0x3ea :  "zx_app_unbind_hub_by_server",
    0x3eb :  "zx_start_stream",
    0x3ec :  "zx_stream_delete",
    0x3f1 :  "zx_set_dev_storagetype_by_SN",
    0x40a :  "APP_CMD_HUB_REBOOT",
    0x410 :  "zx_unbind_dev_by_sn",
    0x464 :  "APP_CMD_GET_EXCEPTION_LOG",
    0x46d :  "CMD_GET_HUB_UPGRADE",
    0xbb8 :  "turn_on_facial_recognition?",
    0xfa0 :  "wifi_country_code_update",
    0xfa1 :  "wifi_channel_update",
    0x1388 : "CMD_SET_DEFINE_COMMAND_VALUE",
    0x1770 : "CMD_SET_DEFINE_COMMAND_STRING"
}
```

While some of these opcode names look tantalizing, only the opcodes less than 0x10 require no authentication, so we're limited to `CMD_DEVICE_GET_SERVER_LIST_REQUEST` [1] and `CMD_DEVICE_GET_RSA_KEY_REQUEST` [2]. For the purposes of this advisory, we only need one of these: `CMD_DEVICE_GET_SERVER_LIST_REQUEST`. Since the same vulnerable code pattern is found in two different functions, we'll discuss them separately.

CVE-2021-21950 - recv_server_device_response_msg_process

In function `recv_server_device_response_msg_process`, the `CMD_DEVICE_GET_SERVER_LIST_REQUEST` request is parsed as shown below:

```
05a1748  uint32_t recv_server_device_response_msg_process(struct dev_packet* devpkt, int32_t inp_msglen, struct sockaddr* dstaddr, int32_t
sockfd)

005a179c      uint32_t opcode = zx.d((zx.d(devpkt->opcode:1.b) << 8).w | zx.w(devpkt->opcode.b))
005a17c8      struct dev_packet_full resp_buf
005a17c8      memset(&resp_buf, 0, 0x425)
005a17d8      uint32_t scratch = opcode
005a17e0      struct aes_key_st scratchbuf

005a17e0      if (scratch == 0xc)
005a1acc          m_heart_timemout_nums = 0
005a1ad8          m_udp_server_connect = 1

005a1ae8          int32_t resp_time = 0
005a1b0c          memcpy(&resp_buf, devpkt, inp_msglen)
005a1b30          struct cJSON* jsonobj = cJSON_Parse(&resp_buf.msg)

005a1b48          if (jsonobj != 0)
005a1b6c              uint32_t udp_server_num = zx_Json_GetInt(jsonobj, "nums", 0)  // [3]
005a1ba4              resp_time = zx_Json_GetInt(jsonobj, "utc_time", 0)            // [4]
// [...]
005a1c84              s_udp_server_total_nums = udp_server_num
005a1e9c              for (int32_t ctr = 0; ctr s< udp_server_num; ctr = ctr + 1)
005a1cb8                  memset(&scratchbuf, 0, 0x80)
005a1cf8                  sprintf(&scratchbuf, 0x79ca14, 0x79ca1c, ctr + 1, var_798, var_794, var_790, var_78c, var_788)  {"%s%d"}
{"domain"}
005a1d20                  char* str_value = zx_Json_GetString(obj: jsonobj, string: &scratchbuf, output_ptr: nullptr)    // [5]
005a1d38                  if (str_value != 0 && strlen(str_value) u< 0x80)
005a1da0                      memset((ctr << 7) + 0x88287c, 0, 0x80)
005a1e00                      memcpy(0x88287c + (ctr << 7), str_value, strlen(str_value))
// [...]
005a1eb4              cJSON_Delete(cjson: jsonobj)
005a1ec0              int32_t var_768_4 = 0
005a1ed0              if (s_udp_server_total_nums s> 0)
005a1ed8                  update_udp_push_config_file()
005a1ee8          scratch = send_device_packet_by_command_id(opcode: 0xd)
```

Utilizing the cJSON library to pull out the `nums` field [3], the `utc_time` field [4], and also a list of domains from the json of the server's response [5], the `CMD_DEVICE_GET_SERVER_LIST_REQUEST` opcode then stores this server list information inside of the `zx_udp_push_config.ini` file. When valid, the file might look like so:

```
/~ cat /mnt/zx_udp_push_config.ini
[NET]
domain_total=3
current_index=2
app_server_domain=security-app.eufylife.com
domain1=p2p-vir-6.eufylife.com
domain2=p2p-vir-7.eufylife.com
domain3=mediaserver-usa3.eufylife.com
```

We now have enough context to discuss the vulnerability, so let us go back to a particular subset of the above `recv_server_device_response_msg_process` code:

```
005a1b48          if (jsonobj != 0)
005a1b6c              uint32_t udp_server_num = zx_Json_GetInt(jsonobj, "nums", 0)   // [6]
005a1ba4              resp_time = zx_Json_GetInt(jsonobj, "utc_time", 0)
// [...]
005a1c84              s_udp_server_total_nums = udp_server_num
005a1e9c              for (int32_t ctr = 0; ctr s< udp_server_num; ctr = ctr + 1)    // [7]
005a1cb8                  memset(&scratchbuf, 0, 0x80)
005a1cf8                  sprintf(&scratchbuf, 0x79ca14, 0x79ca1c, ctr + 1, var_798, var_794, var_790, var_78c, var_788)  {"%s%d"}
{"domain"}
005a1d20                  char* str_value = zx_Json_GetString(obj: jsonobj, string: &scratchbuf, output_ptr: nullptr)    // [8]
005a1d38                  if (str_value != 0 && strlen(str_value) u< 0x80)                // [9]
005a1da0                      memset((ctr << 7) + 0x88287c, 0, 0x80)                      // [10]
005a1e00                      memcpy(0x88287c + (ctr << 7), str_value, strlen(str_value))  // [11]
```

When pulling the `nums` field, which serves as the total number of UDP server domains, we must note the total lack of validation on this field anywhere thereafter [6]. Thus, the amount of iterations for the loop at [7] is entirely attacker-controlled, along with the `int32_t ctr` variable at [7] as well. If the search for `domain1`, `domain2`, … `domain%d` field from our packet JSON fails at [8], or if the string length of the value is greater than 0x80, we skip the branch at [9] but stay within our loop at [7]. The `ctr` variable keeps incrementing. With all this in mind we can reason that the lines at [10] and [11] can be hit with whatever value inside of `ctr` that an attacker chooses. While there is a total packet length limit of 0x425 bytes found much earlier in the codebase, there's no requirement for our JSON to have `domain1` and `domain2` and so on and so forth. We would quickly run out of bytes before we could write outside of the `char s_udp_server_list[0x80][8]` at 0x88287c. Thus, simply by inserting something like `"domain100000":aaaaaaaaaaaaaaa...` and `"nums":100001` into a `CMD_DEVICE_GET_SERVER_LIST_REQUEST`, one can write 0x80 bytes to `0x88287c + (100000 << 7)`, something applicable to any address in memory, resulting in a write-what-where and subsequent code execution.

Crash Information

```
Terminated with signal SIGSEGV, Segmentation fault.
#0  0x77226a84 in memset () from /lib/libc.so.0

[Current thread is 1 (LWP 5768)]

Backtrace stopped: frame did not save the PC
<(^.^)>#info reg
          zero       at       v0       v1       a0       a1       a2       a3
 R0   00000000 1100ff00 0216207c 0216207c 02162084 00000000 00000000 021620fc
           t0       t1       t2       t3       t4       t5       t6       t7
 R8   00000000 00000000 00000200 00000100 00000807 00000800 00000400 00000008
           s0       s1       s2       s3       s4       s5       s6       s7
 R16  6148b52a 7fe43468 00000036 771de280 7c6c0000 00000000 00000007 0000e436
           t8       t9       k0       k1       gp       sp       s8       ra
 R24  00000001 77226a30 00000000 00000000 0083bdb0 7c6fe498 00000000 005a1da8
           sr       lo       hi      bad    cause       pc
      0100ff13 00000000 00000002 0216207c 0080000c 77226a84
           fsr      fir
      00000000 00000000
<(^.^)>#x/4i $pc
=> 0x77226a84 <memset+84>:      sw      a1,-8(a0)
   0x77226a88 <memset+88>:      bne     a0,a3,0x77226a80 <memset+80>
   0x77226a8c <memset+92>:      sw      a1,-4(a0)
   0x77226a90 <memset+96>:      andi    t0,a2,0x4
<(^.^)>#bt
#0  0x77226a84 in memset () from /lib/libc.so.0
#1  0x005a1da8 in recv_server_device_response_msg_process (p_data=0x7c6ff8ec, data_len=175, server_sin=0x7c6ff8d8, socket_id=22) at
src/zx_push_interface.c:633
#2  0x005a6584 in process_msg (p_data=0x7c6ff8ec, data_len=175, server_sin=0x7c6ff8d8, socket_id=22) at src/zx_push_interface.c:1507
#3  0x005a108c in zx_push_recv_packet (socket_id=22) at src/zx_push_interface.c:462
#4  0x005a05b4 in init_udp_server_domain () at src/zx_push_interface.c:340
#5  0x005a1128 in zx_push_receiver_msg_process (argv=0x0) at src/zx_push_interface.c:473
#6  0x771c3264 in pthread_start_thread () from /lib/libpthread.so.0
#7  0x772007f8 in __thread_start () from /lib/libc.so.0

<(^.^)>#info reg a0
a0: 0x2162084

<(^.^)>#info proc map
Mapped address spaces:
        Start Addr   End Addr       Size     Offset objfile
          0x400000   0x7f8000   0x3f8000        0x0 /bin/home_security
          0x808000   0x837000    0x2f000   0x3f8000 /bin/home_security
```

CVE-2021-21951 - read_udp_push_config_file

The previously-mentioned bug is also found in a second function, `read_udp_push_config_file`, in the same code pattern. After reading the `CMD_DEVICE_GET_SERVER_LIST_REQUEST` in `recv_server_device_response_msg_process`, these values are then written into the `/mnt/zx_udp_push_config.ini` file. On reboot, `read_udp_push_config_file` is hit and we read the `/mnt/zx_udp_push_config.ini` config file, all the same vulnerability principles being applicable.

```
0059f2f4                 if (zx_file_readcfg(file: "/mnt/zx_udp_push_config.ini", char *section: 0x79c980, headername: "domain_total",
output: &var_110_0x80_size) != 0)  {"[NET]"}  // [1]
0059f348                     $v0_1 = 0xffffffff
0059f324                 else
0059f324                     s_udp_server_total_nums = atoi(&var_110_0x80_size)
0059f334                     if (s_udp_server_total_nums == 0)
0059f33c                         $v0_1 = 0xffffffff
0059f378                     else
0059f378                         memset(&var_110_0x80_size, 0, 0x80)
0059f3bc                         if (zx_file_readcfg(file: "/mnt/zx_udp_push_config.ini", char *section: 0x79c980, headername:
"current_index", output: &var_110_0x80_size) != 0)  {"[NET]"}
0059f400                             $v0_1 = 0xffffffff
0059f3ec                         else
0059f3ec                             s_current_udp_server_index = atoi(&var_110_0x80_size)

0059f6bc                             while (true)
0059f6bc                                 if (ctr s>= s_udp_server_total_nums) // [2]
0059f6bc                                     if (s_udp_server_total_nums u>= s_current_udp_server_index && sx.d(*
((s_current_udp_server_index << 7) + 0x88287c)) != 0)
0059f6e8                                         memset(0x8827f4, 0, 0x80)
0059f72c                                         strncpy(0x8827f4, 0x88287c + (s_current_udp_server_index << 7), 0x7f) /
// [...]
0059f7a8                                     $v0_1 = 0
0059f7a8                                     break
0059f430                                 memset(&var_110_0x80_size, 0, 0x80)
0059f460                                 void var_90
0059f460                                 memset(&var_90, 0, 0x80)
0059f4a0                                 sprintf(&var_90, 0x79ca14, 0x79ca1c, ctr + 1, var_138, var_134, var_130, var_12c, var_128)
{"%s%d"}  {"domain"}
0059f648                                 if (zx_file_readcfg(file: "/mnt/zx_udp_push_config.ini", char *section: 0x79c980, headername:
&var_90, output: &var_110_0x80_size) != 0)  {"[NET]"}
0059f648                                     $v0_1 = 0xffffffff
0059f64c                                     break
0059f524                                 memset((ctr << 7) + 0x88287c, 0, 0x80)
0059f554                                 if (strlen(&var_110_0x80_size) u< 0x80)
0059f5b4                                     strncpy((ctr << 7) + 0x88287c, &var_110_0x80_size, strlen(&var_110_0x80_size) - 1) //[3]
0059f5dc                                 var_138 = 0xb2
0059f5e4                                 var_134 = 0x28
0059f5f0                                 var_130 = 0x79ca24  {"%s:%s"}
0059f5f8                                 var_12c = &var_90
0059f5fc                                 var_128 = (ctr << 7) + 0x88287c
0059f624                                 dzlog(0x79c99c, 0x17, 0x79dd8c, 0x19, 0xb2, 0x28, 0x79ca24, var_12c, var_128)
{"src/zx_push_interface.c"}  {"%s:%s"}  {"read_udp_push_config_file"}
0059f63c                                 ctr = ctr + 1
```

At [1], we see the `domain_total` field being pulled out of the config file. This value serves as an upper-bound loop condition for the loop at [2]. As our `ctr` variable increments until it reaches `s_udp_server_total_nums`, the `domain%d` fields are again searched for. If a match is found, then we again write to `((ctr << 7) + 0x88287c`, an arbitrary value since we control the contents of the file. Thus, for example, with a `domain100000=aaaaaaaaaaaaa...` inside of the config file, an out-of-bounds write occurs, resulting in code execution.

TIMELINE

2021-09-30 - Vendor Disclosure

2021-11-22 - Vendor Patched

2021-11-29 - Public Release

CREDIT

Discovered by Lilith >_> of Cisco Talos.
