Talos Vulnerability Report

TALOS-2021-1382

Anker Eufy Homebase 2 home_security get_aes_key_info_by_packetid() authentication bypass vulnerability

NOVEMBER 29, 2021

CVE NUMBER

CVE-2021-21955

SUMMARY

An authentication bypass vulnerability exists in the get_aes_key_info_by_packetid() function of the home_security binary of Anker Eufy Homebase 2 2.1.6.9h. Generic network sniffing can lead to password recovery. An attacker can sniff network traffic to trigger this vulnerability.

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

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

CWE

CWE-334 - Small Space of Random Values

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 an 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",
    0xd   :  "CMD_DEVICE_GET_RSA_KEY_REQUEST", // [1]
    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"
}
```

This advisory deals with the fact that some of these opcodes require authentication, whilst others don't. Any opcodes greater that 0x10 (i.e. below [1]) require authentication. These authenticated commands are rather powerful and some of them can be exploited for further escalation, but this is a digression, for we now discuss exactly how this authentication occurs. To start, we first visit the `zx_push_receiver_msg_process` function, called on boot to initialize the device's cloud communication server:

```
005a10a0  int32_t zx_push_receiver_msg_process()
005a1114      dzlog(0x79c99c, 0x17, 0x79dcc4, 0x1c, 0x1d7, 0x28, 0x79cf78, getpid(), 0x83bdb0)  {"src/zx_push_interface.c"}  {"enter
PID=%d"}  {"zx_push_receiver_msg_process"}
005a1120      init_udp_server_domain()
005a112c      update_udp_push_config_file()
005a113c      s_aes_key = 0
005a1178      rand_str(&s_aes_key, rand_len: 0x10) // [2]
```

The only thing worth noting is that the `s_aes_key` static variable is initialized to a random secret on boot by the `rand_str` function:

```
004ec5a4  void* rand_str(void* arg1, int32_t rand_len)
004ec5c4      int32_t var_20 = 0
004ec5e4      void var_18
004ec5e4      void var_10
004ec5e4      gettimeofday(&var_18, &var_10)  // [3]
004ec5f8      int32_t var_14
004ec5f8      int32_t var_1c = var_14

004ec608      // seeded with usec
004ec614      srand(var_1c)        // [4]
004ec698      for (int32_t ctr = 0; ctr s< rand_len; ctr = ctr + 1) // [5]
004ec674          *(arg1 + ctr) = *((rand_r(&var_1c) & 0x3f) + 0x7895a4)  {"0123456789ABCDEFGHIJKLMNOPQRSTUV…"}  // [6]
004ec6b0      *(arg1 + rand_len) = 0
004ec6c4      return arg1
```

We can see the `tv.tv_usec` field of `gettimeofday` [3] being used to seed `srand` at [4], which is totally fine. At [5] we start looping and putting random characters into the output at [6]. The full keyspace looks as such:

```
>>> len("0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz~_")
64
```

Up until this point, the secret generation seems relatively normal, but let us now examine how this secret is utilized in the `process_msg` function, which handles validation and authentication:

```
005a28a0  int32_t aes_key_len = strlen(s_aes_key, pkttime)
005a28b4  if (aes_key_len != 0x10)
005a2d60      $v0_9 = dzlog(0x79c99c, 0x17, 0x79dc6c, 0xb, 0x33b, 0x28, 0x79d3f0)  {"src/zx_push_interface.c"}  {"process_msg
COMM_RANDOM_AES_ENCR…"}  {"process_msg"}
005a28c0  else
005a28c0      int32_t inp_key_str = 0
005a2924      if (get_aes_key_info_by_packetid(str: s_aes_key, packetid: devpkt_arg1->time, output: &inp_key_str) != 0) // [7]
005a2d0c          $v0_9 = dzlog(0x79c99c, 0x17, 0x79dc6c, 0xb, 0x337, 0x28, 0x79d39c)  {"src/zx_push_interface.c"}  {"process_msg
COMM_RANDOM_AES_ENCR…"}  {"process_msg"}
005a2948      else
005a2948          struct aes_key_st* preexpanded
005a2948          int_aes_decryptkey(inputkey: &inp_key_str, aeskey: &preexpanded) // [8]
005a2978          void some_output_s0x448_l0x401
005a2978          memset(&some_output_s0x448_l0x401, 0, 0x401)
005a2984          int32_t var_bb8_1 = 0
005a29b4          int32_t declen = aes_decrypt(key: &preexpanded, inp_enc: &devpkt_arg1->hub_sn, inplen: 0x10, decbytes:
&some_output_s0x448_l0x401) // [9]
005a29cc          int32_t sn_enc_strncmp
005a29cc          if (declen == 0x10)
005a29f0              sn_enc_strncmp = strncmp(g_hub_sn, &some_output_s0x448_l0x401, 0x10) // [10]
005a29cc              if (sn_enc_strncmp != 0)
005a2a58                  dzlog(0x79c99c, 0x17, 0x79dc6c, 0xb, 0x310, 0x28, 0x79d2f0, &some_output_s0x448_l0x401, 0x881c30)
{"src/zx_push_interface.c"}  {"process_msg COMM_RANDOM_AES_ENCR…"}  {"process_msg"}
005a2a68                  $v0_9 = send_device_packet_by_command_id(opcode: 0xd)
```

After getting through some basic validation of the packet length, packet time, etc, we get to the authentication. This process starts at [7], where the previously generated `s_aes_key` is taken, modified and then thrown into the `inp_key_str` stack variable at [7]. We'll skip the actual implementation of `get_aes_key_info_by_packetid` for a second and continue looking at this function. This new key is utilized as the secret for `AES_set_decrypt_key` inside `int_aes_decryptkey`[8], and then the decryption of our input packet's `hub_sn` field occurs at [9]. This decrypted string is then compared against the device's serial number at [10]. If it's a match, we've successfully authenticated. At first glance this might appear secure, but there are two very important facets of this code that we must examine further.

While the `hub_sn` eventually is encrypted for authenticated packets, as long as we can sniff the wire, this string flows periodically to the cloud servers unencrypted. Take for example the previously posted example push packet:

```
[>_>] ---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  // [11]
```

We can clearly see this `hub_sn` value [11] over the wire all the time, so this is a known value. Keeping this in mind, the second facet exists within the glossed-over `get_aes_key_info_by_packetid` function:

```
00551658  int32_t get_aes_key_info_by_packetid(char *inpstr, int32_t packetid, char* output)
00551680      void pktid_str
00551680      int32_t pktidlen
00551680      if (str != 0 && output != 0)
005516cc          strncpy(output, str, strlen(str))               // [12]
005516fc          sprintf(&pktid_str, 0x79071c, packetid) // "%d" // [13]
0055171c          pktidlen = strlen(&pktid_str)
```

To start and reiterate, the input secret is our randomly generated 0x10 length `s_aes_key`, and the `packetid` field is the `devpkt->time` field that we provide in our packet. At [12] the input secret is appropriately `strncpy`'ed to the output, and then at [13] we take the decimal format of the `devpkt->time`. For a quick example of what this would look like:

```
>>> str(time.time()).split(".")[0]
'1632411932'
```

It's important to note that the input `devpkt->time` field must be within 60 seconds of the unix time of the Eufy Homebase device, or else it is considered invalid and we never get to the authentication. Thus, since enough time has passed since epoch, the length of this string will always be 10 bytes. Also, to see the Eufy device's time, it suffices to sniff network traffic and pull it from one of the outbound packets. Continuing on within `get_aes_key_info_by_packetid`:

```
00551680      if (str != 0 && output != 0)
005516cc          strncpy(output, str, strlen(str))
005516fc          sprintf(&pktid_str, 0x79071c, packetid) // "%d" // [14]
0055171c          pktidlen = strlen(&pktid_str)
00551680      int32_t ret
00551680      if (str == 0 || (str != 0 && output == 0) || (str != 0 && output != 0 && pktidlen s>= 0x10))
0055178c          ret = 0xffffffff
00551680      if (str != 0 && output != 0 && pktidlen s< 0x10)
00551774          memcpy(&output[0x10 - pktidlen], &pktid_str, pktidlen) // [15]
00551780          ret = 0
00551798      return ret
```

After the `pktid_str` is generated at [14], it is then `memcpy`'ed on top of our `s_aes_key` copy at [15]. Since this occurs in an overlapping fashion and not in an appending fashion, the output secret will look something like `aC01_?` + `1632411932`, i.e. the first bytes of our randomly generated `s_aes_key` and then the unix time as a string. While the output secret is the same length as we had before, 0x10, this function reduces entropy of our AES encryption from 0x10 bytes to 0x6 bytes, and our keyspace goes from 64^16 (79228162514264337593543950336 combinations) to 64^6 (68719476736 combinations).

To summarize all this into a final vulnerability: Since we know what the decrypted authentication string is supposed to be (our device's `g_hub_sn`, `T8010N123456789a`), and we also know the last 10 bytes of the encryption key that's used for a given packet (the unix time), then it easily becomes possible to offline bruteforce the first six bytes of the AES secret that's actually used to encrypt the `g_hub_sn`. Once we know these first 6 bytes, it does not matter what the current unix time is, we can simply append the first six bytes of the bruteforced `s_aes_key` to the current unix time and gain privileges, resulting in an authentication bypass.

TIMELINE

2021-09-30 - Vendor Disclosure
2021-11-22 - Vendor Patched
2021-11-29 - Public Release

CREDIT

Discovered by Lilith >_> of Cisco Talos.

VULNERABILITY REPORTS

PREVIOUS REPORT
TALOS-2021-1381

NEXT REPORT
TALOS-2021-1379