Talos Vulnerability Report

TALOS-2021-1380

Anker Eufy Homebase 2 home_security process_msg() authentication bypass vulnerability

NOVEMBER 29, 2021

CVE NUMBER

CVE-2021-21953

SUMMARY

An authentication bypass vulnerability exists in the process_msg() function of the home_security binary of Anker Eufy Homebase 2 2.1.6.9h. A specially-crafted man-in-the-middle attack can lead to increased privileges.

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-300 - Channel Accessible by Non-Endpoint (‘Man-in-the-Middle’)

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",
    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, all of the opcodes greater than 0x10 require authentication of a sort. After initially syncing with the Eufy cloud servers and sharing a 0x10 byte long AES key, the two endpoints authenticate by taking this shared AES key and encrypting the device's serial number. To see what this chain of events would look like, here's an example of the key being loaded and then being used to call the 0x3eb opcode which requires authentication:

```
[>_>] ---Pushpkt Received from Device---
Magic       : 0x5a58
CRC         : 0x1234
Opcode      : 0x0016 (RSA_Encrypted_Auth_Key_Resp)
Bodylen     : 0x0080
Time (unix) : 1632324718
msg_ver     : 0x0001
is_resp     : 0x00
idk_lol     : 0x00
err_code    : 0x0000
non_zero    : 0x0000
Hub SN      : T8010N12345678\x00
Msgbody     : [...]
[o.o] got that rsa keyyyyyy
keybuf :
\x02i\x09\x10\xe7\xcbn\x1f\x12\xcb\xe1\x8a$\x0b\xda\xc4i\x16\x11\x89\xa8\x03\xbd;\xabC\xaa\x0a\x92\xe1c\xbeN_T\xe8BF\xb9m\xe3\x85u|;\xff\xed
P\xc2\x82?\xa8\xfdg\xfci\x01\xa5\x11x>\x86\xeb\xe0@c]&+C\xcc\
xfe\x91!\x04\xa56\x06\xa1#/b\xb3\x98\xea\x116\xb1}\x05\xbed_\x85\xd6\xe7\xed\xc6Q)\x07\x1fj\xdf\xd0lO\xa3\xb8\xa5\x00qpoGySfv5PFMuLiN
[^_^] Loaded key: b'poGySfv5PFMuLiN'

[<_<] sending back to device----
Magic       : 0x5a58
CRC         : 0x5678
Opcode      : 0x03eb (zx_start_stream)
Bodylen     : 0x001b
Time (unix) : 1632324718
msg_ver     : 0x0001
is_resp     : 0x00
idk_lol     : 0x00
err_code    : 0x0000
non_zero    : 0x0000
Hub SN (enc): \x04\x11\x64\x92\xc7\xf5\x21\x37\xff\x8e\x27\x5b\x79\x7d\xd5\x3a
Msgbody     : {"channel":0, "protocol":0}

[>_>] ---Pushpkt Received from Device---
Magic       : 0x5a58
CRC         : 0x9abc
Opcode      : 0x03eb (zx_start_stream)
Bodylen     : 0x001b
Time (unix) : 1632324718
msg_ver     : 0x0001
is_resp     : 0x01
idk_lol     : 0x00
err_code    : 0x0000
non_zero    : 0x0000
Hub SN      : [...]
Msgbody     : {"code":0,"msg":"success","data":{"url":""}}
[<_<] sending back to device----
```

While this might look like an okay authentication process, it's incredibly worth noting two facts. First, this occurs over UDP and is able to be Man-In-The-Middle'd (MITM). Second, the only thing encrypted in the entire packet is this `Hub_SN`. Thus, if there's an attacker sitting on the wire, they can simply edit the `Msgbody`, the `opcode`, the `Bodylen` and finally the CRC after the fact to be able to send arbitrary `Pushpkt` commands over this channel, resulting in an authentication bypass.

TIMELINE

2021-09-28 - Vendor Disclosure
2021-11-22 - Vendor Patched
2021-11-29 - Public Release

CREDIT

Discovered by Lilith >_> of Cisco Talos.

VULNERABILITY REPORTS

PREVIOUS REPORT TALOS-2021-1360

NEXT REPORT TALOS-2021-1381
