Talos Vulnerability Report

TALOS-2021-1379

Anker Eufy Homebase 2 home_security CMD_DEVICE_GET_RSA_KEY_REQUEST authentication bypass vulnerability

NOVEMBER 29, 2021

CVE NUMBER

CVE-2021-21952

SUMMARY

An authentication bypass vulnerability exists in the CMD_DEVICE_GET_RSA_KEY_REQUEST functionality of the home_security binary of Anker Eufy Homebase 2 2.1.6.9h. A specially-crafted set of network packets 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

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

CWE

CWE-288 - Authentication Bypass Using an Alternate Path or Channel

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", // [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_RSA_KEY_REQUEST`. Let's look at its handler function `recv_server_device_response_msg_process()`:

```
005a17ec      else if (opcode == 0xd)
005a1f1c          memcpy(&resp_buf, devpkt, inp_msglen)
005a1f70          dzlog(0x79c99c, 0x17, 0x79dc78, 0x27, 0x28e, 0x28, 0x79d174, zx.d(resp_buf.devpkt.datalen))  {"src/zx_push_interface.c"}
{"CMD_DEVICE_GET_RSA_KEY_REQUEST r…"}  {"recv_server_device_response_msg_…"}
005a1f7c          scratch = zx.d(resp_buf.devpkt.datalen)

005a1f84          if (scratch != 0)
005a1fc8              char var_114[0x101]
005a1fc8              memset(&var_114, 0, 0x101)
005a2000              scratch = get_aes_key_info_by_packetid(str: s_defaut_aes_key, packetid: resp_buf.devpkt.time, output: &aeskey)  // [1]
```

Starting out, we initially hit some setup code copying our packet, checking the length, etc., but most importantly is the function at [1], in which we find a parameter named `s_default_aes_key`. This `get_aes_key_info_by_packetid` function first copies the `key` parameter into the `output` parameter. It then takes the `time` field of our input packet, converts it to a decimal number via `sprintf(output, "%d", pkttime)`, and finally `memcpy`'s this decimal unix time on top of the end of the `key`. An example for clarity: Since the `s_default_aes_key` is the hardcoded string `&#%@!_eufy_anker`, if we send a packet with a `time` field of 0x0, we'll end up with the output `aeskey` being filled with `&#%@!_eufy_anke0`. Likewise if our packet's `time` was 0x10, we'd end up with `&#%@!_eufy_ank16` and so forth. Continuing on in `recv_server_device_response_msg_process()`

```
005a200c              if (scratch == 0)
005a2030                  int_aes_decryptkey(inputkey: &aeskey, aeskey: &scratchbuf) // [2]
005a2060                  memset(0x882ca4, 0, 0x101)  // s_rsa_public_key
005a2098                  aes_decrypt(key: &scratchbuf, inp_enc: &resp_buf.msg, inplen: 0x100, decbytes: &s_rsa_public_key) // [2]
```

Using this `&#%@!_eufy_anke0` buffer, we create a valid AES key inside [2] and then use this key to decrypt 0x100 bytes of the body of our input packet at [2], which is stored into `s_rsa_public_key`. Continuing on:

```
005a20a8                 uint32_t len = 0x25
005a20d0                 memset(&resp_buf, 0, 0x425)
005a20e0                 resp_buf.devpkt.magic_0x5a58 = 0x5a58
005a20e8                 resp_buf.devpkt.opcode = 0x16
005a20ec                 resp_buf.devpkt.datalen = 0
005a210c                 resp_buf.devpkt.time = time(tloc: nullptr)
005a2114                 resp_buf.devpkt.msg_version.b = 1
005a2118                 resp_buf.devpkt.is_resp = 0
005a211c                 resp_buf.devpkt.idk_6 = 0
005a213c                 strcpy(&resp_buf.devpkt.hub_sn, 0x881c30)
005a2154                 if (pst_rsa_public_key == 0)
005a21e0                     pst_rsa_public_key = rsa_get_pubKey(inpkey: &s_rsa_public_key) // [3]
005a2178                 else
005a2178                     RSA_free(pst_rsa_public_key)
005a21ac                     pst_rsa_public_key = rsa_get_pubKey(inpkey: &s_rsa_public_key) // [4]
```

The decrypted `s_rsa_public_key` bytes serve as an RSA key's modulus number, and the result is stored into `pst_rsa_public_key` at either [3] or [4], depending on if a `pst_rsa_public_key` already exists or not. To proceed:

```
005a2208                 void rsa_aes_enc
005a2208                 memset(&rsa_aes_enc, 0, 0x101)
005a223c                 int32_t len_of_key = rsa_encrypt(rsa: pst_rsa_public_key, from: &s_aes_key, to: &rsa_aes_enc) // [5]
005a2254                 if (len_of_key s< 0x81)
005a226c                     resp_buf.devpkt.datalen = len_of_key.w
005a2294                     memcpy(&resp_buf.msg, &rsa_aes_enc, len_of_key)
005a22b0                     len = 0x25 + len_of_key
005a22f0                 resp_buf.devpkt.crc = Crc16_DATAs(startaddr: &resp_buf.devpkt.opcode, len_to_crc: len.w - 4)
// [..]
005a2380                 scratch = sendto(sockfd: sockfd, buf: &resp_buf, len: len, flags: 0, dest_addr: dstaddr, addrlen: 0x10) // [6]
```

Our `pst_rsa_public_key` is then used to encrypt the contents of `s_aes_key`, a 0x10 bytes key of random bytes that are very important later on. Finally, this encrypted key is sent back to the server at [6]. While this all seems convoluted, let us recap to elucidate:

First, the server sends an RSA public key that is encrypted with a known and essentially static AES key. The device decrypts this RSA public key and then uses it to encrypt a secret set of bytes generated on boot (`s_aes_key`) and then sends it back to the server, presumably so that the server can decrypt the message with its corresponding RSA private key, and then both the server and device know `s_aes_key`. This `s_aes_key` is used as the authentication for all of the `home_security` opcodes > 0x10:

```
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"
}
```

An example of this entire process will look like so:

```
// The device's request for an RSA key:
[>_>] ---Pushpkt---
Magic       : 0x5a58
CRC         : 0x1234
Opcode      : 0x000d (CMD_DEVICE_GET_RSA_KEY_REQUEST)
Bodylen     : 0x0000
Time (unix) : 1632248245
msg_ver     : 0x0001
is_resp     : 0x00
idk_lol     : 0x00
idk_lol2    : 0x0000
non_zero    : 0x0001
Hub SN      : T8010N123456789a\x00

// Our response with the '"&#%@!_eufy_anke0" encrypted rsa pubkey
[>_>] ---Pushpkt---
Magic       : 0x5a58
CRC         : 0x2345
Opcode      : 0x000d (CMD_DEVICE_GET_RSA_KEY_REQUEST)
Bodylen     : 0x0100
Time (unix) : 0
msg_ver     : 0x0001
is_resp     : 0x01
idk_lol     : 0x00
idk_lol2    : 0x0000
non_zero    : 0x0001
Hub SN      : T8010N123456789a\x00
Msgbody     :
\x10$q+\xe7\xe2'9\x8b\x9d\xc5\x1bHD^J\x00\xac\xab\xd58\x8f9\xe0@\x00'Y\xd0\x13ox\xdb<<T\xa6\x01\xe2\x16Y\x10\x7fy\x84h\x97Jm37\xc86e\x80\x80
g\xf7:\xccf\xa65\xf8\xf9\x18\xa2.AZ\xe1\xa9\xcb\x1a\x8a1
\x97\xc9\x1c>\x88\xa3*\x16/\xa7\x0f=\xdby\xcaE\xcd\xa4=\x15\x9eC\xa2\\x01nH\xcc:k\xcf\xb5\xcfr\x1e\xbcg\xa6P<\xfdSS\xff\xf1\x84\x92\x14GC\xc
3\x0b\xa8\xde\xc8\xcaBHv\xd2\xee K*\xacaI\x85][\xa3\xd8lh\x9c8O\xbet{,
\x80\xa7.-
C\x11\x9bj\xd0x\x1d\xb2\x97\xd6\xb4\xa3\xce\x00g"\x8b\xfe7<\x16q\xd2\xbf_B)\xe4\x82D\xbf\xd3\x15\x04/\xea\xa44\x8c\xb4\xe3\xad\xb8}\xa9\xa7j
\xcc1\x82\x19\x9e\xf1\xbb\x80X\xe9\xb2\xb9\x15\x07\xb0\xbe\x
d5\xf0}\xa3akc\xa0,\xd7\xc6\x09\xe6\x81Lo\xdb\x1b\xeb\xca\xf9\x9e\xa7\xd1)\x9a\xf3\xfe\xd6G\xb5S

// their response with the AES_ECB encrypted via rsa pubkey
[>_>] ---Pushpkt---
Magic       : 0x5a58
CRC         : 0x3456
Opcode      : 0x0016 (????)
Bodylen     : 0x0080
Time (unix) : 1632248245
msg_ver     : 0x0001
is_resp     : 0x00
idk_lol     : 0x00
idk_lol2    : 0x0000
non_zero    : 0x0000
Hub SN      : T8010N123456789a\x00
Msgbody     :
,\xebq\xda\x05\xc7\x96\x87\xebb\x9c\xed[,K\xcb\x1e\x14\xeat\x0d\xf7kvR\xa1\xe2\xa9\xd1\x0d\xe4\x99O8^G\xb1\xf7>\x9d\xcb\x15\xc2mw\x13\x8c\xd
0\x80p]\x8d\xe4\x88\xbec\xe7\x92\xd2R\x1b>\xa5\xda9?\xb2
dT\xfa8}b\x04\xe3Jk\x80\x0e\x9eP&\xdd\x09\xf1b\xb4\xaa\xa8C\xe0\x86\xc4bu\xb3^\xd4\x90,*\xfd\xb9Z\xe4]\x1d=\x97\xa3\xccg/\x0a\xb3\xde\x8bTA\
x85Zv\xbd@\x1f\xefF}

// Using our RSA private key to decrypt the response, take the last 0x10 bytes as the s_aes_key: DCxj4M8wVZ6E78vp
[o.o] got that rsa keyyyyyy
keybuf : \x02t/\\xa0x\x1e\x11T\xe5\x14S-,-O\xed\xf41p\xa33?\\xa7\xb8\xbc\xa9\xe0\xc4C\xc6\xfc\x9e\xa8qG0\xf1\x06?
*@\xbe>\xaaW\x14\xcc\xd8U\xf0\xabE<\x90\x97\xd7\xde\x9dR\xc8n$\x80,\xf6\x9b\xd8\x88\xe8\xcf\x1e\x
98\xa0Y\xc5\xa0\xfb)\x9a\xfb\xf9\xa1<\xe0\x06\x84\xc6\xc7\xae\x9dQkh\xc40\xad\xf1\xf4\xbf\x9a\xf8\xe9\xa4\xa3\xb8"\x03\x12\x00DCxj4M8wVZ6E78
vp

// We can also verify the key by looking at process memory:
# dd if=/proc/6164/mem bs=1 skip=8924588 count=16
DCxj4M8wVZ6E78vp
16+0 records in
16+0 records out
```

Regardless of all these complex steps, and even though someone passively monitoring the wire could not know the `s_aes_key`, since all of this authentication and encryption stems from an initially known value (`&#%@!_eufy_anke0`), it follows that the entire chain of events is inherently unsecure. At any given time an attacker could send this `CMD_DEVICE_GET_RSA_KEY_REQUEST` packet (since it does not require authentication itself) and overwrite the `s_aes_key`, gaining access equal to that of the device's owner's phone app.

TIMELINE

2021-10-05 - Vendor Disclosure

2021-11-22 - Vendor Patched

2021-11-29 - Public Release

CREDIT

Discovered by Lilith >_> of Cisco Talos.