Talos Vulnerability Report

TALOS-2021-1369

Anker Eufy Homebase 2 pushMuxer processRtspInfo heap buffer overflow vulnerability

OCTOBER 11, 2021

CVE NUMBER

CVE-2021-21940

SUMMARY

A heap-based buffer overflow vulnerability exists in the pushMuxer processRtspInfo functionality of Anker Eufy Homebase 2 2.1.6.9h. A specially-crafted network packet can lead to a heap buffer overflow. An attacker can send a malicious packet 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

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

CWE

CWE-122 - Heap-based Buffer Overflow

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.

Just as videos captured by Eufycams can be stored in the Eufy Homebase 2 device's extra storage, so too can these videos be viewed after the fact via the RTSP server running on the Eufy Homebase 2, called `pushMuxer`. A relatively standard RTSP server, it supports the "OPTIONS", "DESCRIBE", "SETUP", "PLAY", "TEARDOWN" and "GET_PARAMETER" requests. For this advisory we don't care so much about the particular verbs, moreso how the server generically handles RTSP packets.

Before getting into the code we must note that the server is capable of handling 64 different RTSP sessions at a time. Each of these sessions corresponds to a size 0x9a8 `RTSPSession` structure in the heap. This `RTSPSession` structure contains useful members that determine: if the session is initialized; the file descriptor to read from; and a nested structure that I've named the `RecvPkt`. This nested `RecvPkt` contains the size 0x800 destination buffer of the session's RTSP packets, an offset pointer and the status of the packet, among other things.

With that all in mind, let's start with the `ProcessRtspInfo` function:

```
00428fe0  int32_t ProcessRtspInfo(int32_t param1, int32_t param2)
// [...]
00429090         int32_t loop_counter = 0
// [...]
004292f4         while (zx.d((loop_counter s< 0x40 ? 1 : 0).b) != 0)    // [1]
004290a8             int32_t loop_counter_m4 = loop_counter << 2
004290b4             int32_t loopcounter_m11 = (loop_counter_m4 << 2) - loop_counter_m4 - loop_counter
004290d8             struct RtspSession* RecvPktWrap = (((((loopcounter_m11 << 3) - loopcounter_m11) << 2) + loop_counter) << 3) + 0x91a230
// [2]
//
004290e4             uint32_t fd = RecvPktWrap->fd
004290f0             int32_t delete_session = 0
00429114             if (RecvPktWrap->init_flag != 0 && fd s> 0)        // [3]
00429150                 if (zx.d((*(param2 + (fd u>> 5 << 2)) s>> (fd & 0x1f)).b & 1) != 0)      // [4]
// [...]
004291b0                     void* recv_buffer_0x800
004291b0                     if (RecvPktWrap->recvpkt.pkt_read_status != 3)         //[5]
004291f0                         recv_buffer_0x800 = &RecvPktWrap->recvpkt.recv_buf //[6]
0042920c                         memset(recv_buffer_0x800, 0, 0x800)
004291d8                     else
004291d8                         recv_buffer_0x800 = &RecvPktWrap->recvpkt.recv_buf[RecvPktWrap->recvpkt.read_ptr]
00429238                     int32_t recv_ret = RecvData(fd: fd, buffer: recv_buffer_0x800, size: 0x800, buffer_offset: &buffer_offset)
//[7]
00429250                     if (recv_ret == 0)
00429268                         ProcessRecvData(recvpktwrap: RecvPktWrap, recv_buffer: recv_buffer_0x800, buffer_offset: buffer_offset) //
[8]
```

As mentioned before, there are 64 different possible RTSPSessions, and the loop at [1] iterates over all of them. The fancy math from [1] to [2] boils down to `0x91a230 + (0x9a8 * x)` whereby `x` is the current session number. This is where our `RtspSession RecvPktWrap` points, and it's important to note that these structures are consecutive in heap memory. At [3] and [4], the code just makes sure the session is initialized, and that the file descriptor is valid, before getting to the code we care about starting at [5]. If the `RtspSession->recvpkt.pkt_read_status` is not 3, then it's treated like a new RTSP request. The "recv" destination is the top of that 0x800 buffer [6], and the buffer is cleared immediately after. If the `pkt_read_status` is 3 however, then the read destination starts from `recvpkt.recv_buf[recvpkt.read_ptr]`, i.e. an offset within the buffer. As we can see at [7] however, the size of the read never actually changes and is a constant 0x800 bytes every time. If we can manage to get our `RtspSession->recvpkt.pkt_read_status` equal to 0x3, then we will easily overflow and read into the next `RtspSession`. To save time, there's not any real length checks within `RecvData` or `ProcessRecvData`, and we can get `recvpkt.pkt_read_status` to 0x3 by sending an RTSP packet with a valid RTSP verb, but without a "\r\n\r\n" within 0x800 bytes.

Let now us examine the memory of a code flow in which those two conditions are met, with a breakpoint after we have our `RtspSession *RecvPktwrap` pointer assigned to the current session:

```
[^.^]> x/20wx $v0 // our RtspSession.
0x91a230:       0x00000001      0x00000000      0x00000000      0x00000000 // [9]
0x91a240:       0x0000000c      0x00000000      0x004562f0      0x00000000 // [10]
0x91a250:       0x00000000      0x00000000      0x00000000      0x00000000
0x91a260:       0x00000000      0x00000000      0x00000000      0x00000000
0x91a270:       0x00000000      0x00000000      0x00000000      0x00000000
```

At [9], we see the `RecvPktWrap->init_flag` assigned to 1, and at [10] we see both the `recvPktWrap->fd` and a less important pointer to the global RTSP server context (`RtspSession->gRtspSvrCtx`). If we wait until after the `RecvData` call and look at `RtspSession->recvpkt->recv_buf`, we'll see our packet data starting at `&RtspSession+0xb8`:

```
[^~^]> x/20wx 0x91a230+0xb0
0x91a2d0:       0x00000000      0x00000000      0x4954504f      0x20534e4f
0x91a2e0:       0x00737472      0x70ffff00      0x312f2f3a      0x312e3239
0x91a2f0:       0x312e3836      0x3435322e      0x4c56623a      0x2e332f43
0x91a300:       0x2e393231      0x20353532

[^.^]> x/3s 0x91a2d8
0x91a2d8:       "OPTIONS rts"
0x91a2e4:       ""
0x91a2e5:       "\377\377p://192.168.1.254:bVLC/3.129.255"...
```

There's no processing done on the packet, only validation to make sure it contains a correct RTSP verb, as well as to check and see if there's a `\r\n\r\n` or not. Assuming there's no `\r\n\r\n`, the data sits in the buffer and the `RecvPktWrap->recvpkt.read_pointer` is set to 0x800. We then iterate over every other RTSP session before coming back to our current one, and on this second `ProcessRtspInfo` we can finally see the vulnerability in action:

```
Breakpoint 2, 0x76f08704 in recv () from /lib/libpthread.so.0

[x.x]> info reg a0 a1 a2
a0: 0xc
a1: 0x91aad8
a2: 0x800
```

For the arguments of `recv(fd, dstbuffer, len)`, we have a file descriptor of 0xC again and a read destination of `0x91aad8`, which is a problem considering that the second `RtspSession` object lives at `0x91abd8 // (0x91a230 + (x * 0x9a8))` . If we then look at the second `RtspSession` after this overflow has occurred, we can clearly see that the structure has been completely overflowed by attacker-controlled data:

```
[-.-]> x/40wx $v0
0x91abd8:       0x41414141      0x42424242      0x43434343      0x44444444
0x91abe8:       0x41414141      0x42424242      0x43434343      0x44444444
0x91abf8:       0x41414141      0x42424242      0x43434343      0x44444444
0x91ac08:       0x41414141      0x42424242      0x43434343      0x44444444
0x91ac18:       0x41414141      0x42424242      0x43434343      0x44444444
0x91ac28:       0x41414141      0x42424242      0x43434343      0x44444444
0x91ac38:       0x41414141      0x42424242      0x43434343      0x44444444
0x91ac48:       0x41414141      0x42424242      0x43434343      0x44444444
0x91ac58:       0x41414141      0x42424242      0x43434343      0x44444444
0x91ac68:       0x41414141      0x42424242      0x43434343      0x44444444
```

Turning this overflow into a write-what-where is a rather simple task that could be accomplished in a few different ways. The easiest would be to clear out the second `RtspSession` so that it's still considered inactive (i.e. the `RtspSession->init_flag` is set to 0x0), and then read another set of bytes on the first `RtspSession`. As long as the first 0x1000 bytes of the RTSP packet do not contain `\r\n\r\n`, a third pass over the first `RtspSession` will occur, and since the `RtspSession->read_ptr` member is below the read buffer and gets overflowed to an attacker-controlled value on the second pass, the third call to `RecvData` reads to an address completely under attacker control.

Crash Information

```
Program received signal SIGBUS, Bus error.
0x00429134 in ?? ()
[?.?] uhhhh...? Unhandled signal, gg. SIGBUS

[-.-]> info reg
        zero       at       v0       v1       a0       a1       a2       a3
 R0   00000000 1100ff00 8bbc29a4 7f8ffd78 00000001 0043ef4f 00000000 00000000
        t0       t1       t2       t3       t4       t5       t6       t7
 R8   00000000 61616161 61616161 61616161 61616161 873954f8 00000000 61616161
        s0       s1       s2       s3       s4       s5       s6       s7
 R16  7f8ffea0 00000000 00000003 76f2e280 7f801000 00000000 00000003 00000803
        t8       t9       k0       k1       gp       sp       s8       ra
 R24  00000000 76e638c0 00000000 00000000 0045ddf0 7f8ffc28 7f8ffc28 00429270
        status       lo       hi badvaddr    cause       pc
      0100ff13 00001381 000001be 8bbc29a4 00800010 00429134
          fcsr      fir      hi1      lo1      hi2      lo2      hi3      lo3
      00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
        dspctl  restart
      00000000 00000000

[x.x]> x/4i $pc
=> 0x429134:    lw      v1,0(v0)
   0x429138:    lw      v0,56(s8)
   0x42913c:    nop
   0x429140:    andi    v0,v0,0x1f
```

TIMELINE

2021-09-14 - Vendor Disclosure

2021-10-10 - Vendor Patched

2021-10-11 - Public Release

CREDIT

Discovered by Lilith >_> of Cisco Talos.