Talos Vulnerability Report

TALOS-2022-1440

Anker Eufy Homebase 2 mips_collector appsrv_server use-after-free vulnerability

JUNE 15, 2022

CVE NUMBER

CVE-2022-21806

SUMMARY

A use-after-free vulnerability exists in the mips_collector appsrv_server functionality of Anker Eufy Homebase 2 2.1.8.5h. A specially-crafted set of network packets can lead to remote code execution. The device is exposed to attacks from the network.

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 2.1.8.5h

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 `mips_collector` binary of the Eufy Homebase 2 manages a few different tasks, but today we are chiefly concerned about the `appsrv_server` that it binds onto TCP 0.0.0.0:5000. This server is in charge of dealing with a variety of different message types from the cloud, such as disassociating paired devices (cameras, doorbells, etc). The server receives messages in a particular format that will be referred as a `mt_msg`. This `mt_msg` protocol is as follows:

```
struct raw_mt_msg{
    uint8_t magic_byte;      // '\xfe', skipped
    uint16_t expected_len;   // can be 1 or 2 bytes
    uint8_t cmd0;
    uint8_t cmd1;
    uint8_t msgbuf[expected_len];
    uint32_t checksum;       // optional
}
```

The optional aspects of the `mt_msg` are both pre-determined by the hardcoded schema for the socket itself, so while these remain constant for the service, it could be subject to change. Even this is perhaps getting too far ahead. For the purposes of this vulnerability, repeatedly sending the same invalid packet (e.g. `bytearray(b'\xfe')`) can cause the crash to occur, and in fact the only real requirement seems to be that we're constantly opening and closing new connections, behaving in a manner very similar to TALOS-2021-1370. It thus behooves us to examine the server's `accept` and `close` codeflows. Starting with `appsrv_server_thread`, where the `accept` occurs:

```
int32_t  appsrv_server_thread(int32_t arg1, int32_t arg2)
// [...]
00429328                  accept_ret, $a3_4 = SOCKET_SERVER_accept(clisock:
&clisock_wrapper->inner.clisock, serversock: ssock_inner) // [1]
00429340                  if (accept_ret s< 0)
00429370                      LOG_bug_here(0x474fa0, 0x4753c0, 0x9b1, 0x475164)
{"appsrv.c"}  {"cannot accept!\n"}  {"appsrv_server_thread"}
00429370                      noreturn
00429384                  if (accept_ret == 0)
004293a8                      $a3_1 = LOG_printf(0, 1, 0x475174, $a3_4)  {"no
connection yet\n"}
004293b4                      continue
004293c4                  else
004293c4                      clisock_wrapper->running = 0
004293f8                      snprintf(&var_30, 0x1e, 0x475148, clisock_wrapper-
>connection_num, $v0_4)  {"connection-%d"}
00429430                      clisock_wrapper->connection_num_str = strdup(&var_30)
00429444                      if (clisock_wrapper->connection_num_str == 0)
00429444                          break
004294b0                      snprintf(&var_30, 0x1e, 0x475188, clisock_wrapper-
>connection_num)  {"thread-u2s-%d"}
004294e4                      struct thread_struct* $v0_31
004294e4                      $v0_31, $a3_1 = THREAD_create(iface_name: &var_30,
thread_flags: s2appsrv_thread, thread_func: clisock_wrapper, args4thread: nullptr)
// [2]
004294fc                      clisock_wrapper->__offset(0xbc).d = $v0_31
00429500                      clisock_wrapper = nullptr
00429500                      continue
00429538          LOG_printf(2, 0, 0x475198, $a3_1)  {"appsrv_server_thread:
dead!\n"}
```

In an effort to expedite your reading, I'll forgo getting into the specifics of the above structures, as there's quite a few. For our purposes, we only care about the lines at [1], where the `accept` clearly occurs, and [2], the subsequently created thread that handles the client connection. Before getting into the actual `s2appsrv_thread`, we need to examine the `THREAD_create` scaffolding that was built on top of normal `pthreads`, which starts with the `thread_struct` structure:

```
struct thread_struct __packed
{
    struct thread_struct* globalptr; // [3]
    char* iface_name;
    uint32_t is_destroyed;
    struct _THREAD* innerthread;
    void* start_func;
    struct clisock_wrapper* clisock_wrapper;
    void* thread_flags_cpy;
    uint32_t start_func_ret;
    uint8_t has_inner_thread;
    uint16_t idk2;
    uint8_t idk3;
    struct thread_struct* next; // [4]
};
```

In retrospect there were better names for it, but the `thread_struct` acts as a sort of linked-list wrapper for a newly created `pthread_t`. For all allocated `thread_structs`, the `struct thread_struct * globalptr` member [3] always points to `0x496c54`, which is four bytes into a global `thread_struct` that acts as the anchor or head of the linked list, which I call the `known_thread_list`. For the global `known_thread_list` structure, the `globalptr` member [3] instead points to the newest `thread_struct` that has been allocated. The `struct thread *next` member [4] is named for its purpose, linking a given `struct thread_struct` object to the previously allocated `thread_struct` in the list. With an idea of the datastructure, we can now examine the `THREAD_create` function:

```
00459fec  struct thread_struct* THREAD_create(char* iface_name, void* thread_flags,
void* thread_func, void* args4thread)
// [...]
0045a080      struct thread_struct* t_struct = calloc(1, 0x28)   // [5]
0045a08c      struct thread_struct* ret
0045a08c      if (t_struct == 0)
0045a0cc          LOG_printf(2, 0, 0x479940, free_const(ifacename_cpy))  {"no memory
for thread?\n"}
0045a0d8          ret = nullptr
0045a0f0      else
0045a0f0          t_struct->iface_name = ifacename_cpy
0045a100          t_struct->globalptr = 0x496c54
0045a110          t_struct->thread_flags_cpy = args4thread
0045a120          t_struct->clisock_wrapper = thread_func
0045a130          t_struct->start_func = thread_flags
0045a13c          t_struct->has_inner_thread = 0
0045a148          t_struct->is_destroyed.b = 0
0045a154          t_struct->start_func_ret = 0
0045a1e0          atomic_global_lock()                           // [6]
0045a1fc          t_struct->next = known_thread_list.globalptr
0045a20c          known_thread_list.globalptr = t_struct
0045a210          atomic_global_unlock()                         // [7]
0045a260          int32_t var_38
0045a260          t_struct->innerthread = _THREAD_create(thread_name: t_struct-
>iface_name, thread_func: port5000_recv_thread, t: var_38) // [8]
// [...]
0045a300          if (t_struct->innerthread == 0)
0045a314              THREAD_destroy(thread: t_struct)
0045a320              t_struct = nullptr
0045a324          ret = t_struct
0045a344      return ret
```

Our new `thread_struct` is allocated at [5] and initialized in all the code up until [6]. The `atomic_global_lock` function at [6] calls `_ATOMIC_global_lock()`, which is used in quite a few places. Most importantly, this mutex function appears in both `THREAD_create` and `THREAD_destroy`. Inside the critical section, our new `thread_struct` has its `next` pointer initialized to the next oldest `thread_struct`. Our global `known_thread_list` updates its pointer to our newest `thread_struct` before exiting the critical section at [7]. Before jumping into `_THREAD_create` at [8], let's peek at the `struct INNERTHREAD` object that will be returned:

```
struct INNERTHREAD __packed
{
    uint8_t* some_id;
    uint32_t pthread_t;
    uint32_t syscall_0x4222_ret;
    uint32_t pid;
    struct thread_struct* thread_wrapper;
    uint32_t init;
    void* somecb;
    void* global_threadlist;
};
```

And now for the `_THREAD_create` function:

```
0044cfb4  struct INNERTHREAD* _THREAD_create(char* thread_name, void* thread_func,
struct thread_struct* t) {
0044cff0      struct INNERTHREAD* newthread = calloc(1, 0x20)
0044d008      if (newthread == 0)
0044d018          _atomic_fatal(0x478578)  {"no memory for thread\n"}
0044d018          noreturn
0044d030      newthread->some_id = 0x4783f8
0044d040      newthread->thread_wrapper = t           // [9]
0044d050      newthread->somecb = thread_func
0044d054      _ATOMIC_global_lock()
0044d070      newthread->global_threadlist = global_clisock_list.inner.threadlist //
[10]
0044d080      global_clisock_list.inner.threadlist = newthread                    //
[11]
0044d08c      newthread->init = 0
0044d0b4      if (strcmp(0x478590, thread_name) == 0)  {"uart"}   // [12]
                              // [...]
0044d208      if (pthread_create(thread: &newthread->pthread_t, attr: nullptr,
start_routine: start_routine, arg: newthread) != 0) // [13]
0044d218          _atomic_fatal(0x478598)  {"cannot create thread\n"}
0044d218          noreturn
0044d234      uint32_t var_60 = newthread->pthread_t

0044d29c      if (pthread_detach(newthread->pthread_t) != 0) { //[...] }
0044d310      _ATOMIC_global_unlock()
0044d330      return newthread
```

Nothing particularly special, but it's worth noting that our new `INNERTHREAD`'s `struct thread_struct`[9] points back to our `struct thread_struct` from before, and it also has a singularly linked list of similar style to our `struct thread_struct`. The `INNERTHREAD` has its `global_threadlist` pointer assigned at [10] to a global list, and then the global list has its `threadlist` pointer assigned at [11]. While UART threads get some extra `pthread_attr` set, our network threads skip the branch at [12] and create a pthread with default attributes at [13].

But now that we've examined the beginning of these threads, we must follow them till their end. For this we go to the `int32_t s2appsrv_thread(struct clisock_wrapper* clisock)` function, i.e. the start of the pthread that is created above:

```
00428940  int32_t s2appsrv_thread(struct clisock_wrapper* clisock)

00428978      if (clisock == 0)
004289a8          LOG_bug_here(0x474fa0, 0x4753d8, 0x8e2, 0x474fac)  {"appsrv.c"}
{"pCONN is null?\n"}  {"s2appsrv_thread"}
004289a8          noreturn
004289e4      char iface_str[0x1e]
004289e4      int32_t $a1
004289e4      int32_t $a2
004289e4      int32_t $a3_1
004289e4      $a1, $a2, $a3_1 = snprintf(&iface_str, 0x1e, 0x474fbc, clisock-
>connection_num)  {"s2u-%d-iface"}
004289f8      clisock->inner.s2u_num_iface = &iface_str
00428a30      if (MT_MSG_interfaceCreate(iface: &clisock->inner, $a1, $a2, $a3_1) !=
0)                     // [14]
00428a60          LOG_bug_here(0x474fa0, 0x4753d8, 0x8f0, 0x474fcc)  {"appsrv.c"}
{"Cannot create socket interface?\n"}  {"s2appsrv_thread"}
00428a60          noreturn
00428b2c      while (zx.d(clisock->is_alive:1.b) == 0)
00428bbc          if (zx.d(clisock->inner.err) != 0)
00428be0              $a3_3 = LOG_printf(0x10, 0, 0x475044, $a3_3)
{"s2appsrv_thread:  socket interface is dead!\n"}
00428bf4              clisock->is_alive:1.b = 1
00428c30          else
00428c30              struct mt_msg* msg
00428c30              msg, $a3_3 = MT_MSG_LIST_remove(&clisock->inner.s2u_num_iface,
mt_msg_queue: &clisock->inner.mt_msg_queue, timeout: 0x3e8)  // [15]
00428c48              if (msg != 0)
// [...]
00428dc8                  s2appsrv_recv(clisock: clisock, mtmsg: msg)   // [16]
// [...]
00428e14                  $a3_3 = MT_MSG_free(mt_msg: msg)
```

At [14] our newly spawned `s2appsrv_thread` in fact creates another thread at [14], henceforth called the `mt_msg_rx_thread`, which actually calls `recv()` and processes `mt_msg` packets. These `mt_msg` packets get parsed and validated and then populated into a `mt_msg_queue` which is read by our `s2appsrv_thread` at [15]. Assuming a message exists, the actual packet commands will be run inside of [16]. I will refrain from talking too much more about this `mt_msg_rx_thread`, since all we really care about is that it tends to access memory and allocated memory and datastructures (like most threads). We must examine the much more important destruction of this `mt_msg_rx_thread`. Continuing in `s2appsrv_thread()`:

```
00428940  int32_t s2appsrv_thread(struct clisock_wrapper* clisock)
// [...]
00428e14                $a3_3 = MT_MSG_free(mt_msg: msg)
///[...]
00428e64      while (zx.d(clisock->is_alive.b) != 0)
00428e48          TIMER_sleep(0xa)
00428e6c      lock_app_mutex()
00428e80      struct clisock_wrapper* clisock_ptr = &global_clisock_list
00428ed0      while (clisock_ptr->is_alive != 0)
00428ea0          if (clisock_ptr->is_alive == clisock)
00428ea0              break
00428ebc          clisock_ptr = &clisock_ptr->is_alive->clisock_next
00428ef4      if (clisock_ptr->is_alive != 0)
00428f10          *clisock_ptr = clisock->clisock_next  // found our entry
00428f1c          clisock->clisock_next = nullptr
00428f20      unlock_app_mutex()
00428f48      MT_MSG_interfaceDestroy(clisock_iface: &clisock->inner)  // [17]
00428f74      SOCKET_destroy(clisock: clisock->inner.clisock)
00428f90      free(clisock)
00428fb4      return 0
```

Due to the complexity of the objects there's a lot of cleanup that needs to be done. Skipping past the global pointer cleanup, we go down to `MT_MSG_interfaceDestroy` at [17], since it's where our `mt_msg_rx_thread` is eventually destroyed:

```
0043bffc  struct clisock_interface* MT_MSG_interfaceDestroy(struct
clisock_interface* clisock_iface)
0043c01c      struct clisock_interface* $v0 = clisock_iface
0043c024      if ($v0 != 0)
0043c058          LOG_printf(0x20000, 0, 0x4768f4, clisock_iface->s2u_num_iface)
{"%s: Destroy interface\n"}
0043c06c          clisock_iface->err = 1
0043c080          if (clisock_iface->msg != 0)
0043c09c              MT_MSG_free(mt_msg: clisock_iface->msg)
0043c0b0              clisock_iface->msg = nullptr
0043c0c4          if (clisock_iface->clisock != 0)
0043c0ec              STREAM_close(clisock_iface->clisock)
0043c108              if (clisock_iface->sconfig != 0)
0043c128                  if (clisock_iface->sconfig->socktype != 0x63)
0043c184                      SOCKET_destroy(clisock: clisock_iface->clisock)
0043c150                  else
0043c150                      SOCKET_destroy(clisock: clisock_iface->clisock)
0043c198              clisock_iface->clisock = nullptr
0043c1ac          if (clisock_iface->mt_msg_rx_thread != 0)
0043c1d4              THREAD_destroy(thread: clisock_iface->mt_msg_rx_thread)  //
[18]
0043c1e8              clisock_iface->mt_msg_rx_thread = nullptr
0043c1fc          free_mt_msg_queue(mtarg2: &clisock_iface->mt_msg_queue)
0043c218          if (clisock_iface->mi_lock != 0)
0043c240              MUTEX_destroy(clisock_iface->mi_lock)
0043c254              clisock_iface->mi_lock = nullptr
0043c268          if (clisock_iface->srsp_semaphore != 0)
0043c290              SEMAPHORE_destroy(clisock_iface->srsp_semaphore)
0043c2a4              clisock_iface->srsp_semaphore = nullptr
0043c2b8          if (clisock_iface->frag_semaphore != 0)
0043c2e0              SEMAPHORE_destroy(clisock_iface->frag_semaphore)
0043c2f4              clisock_iface->frag_semaphore = nullptr
0043c308          if (clisock_iface->mi_tx_lock != 0)
0043c330              MUTEX_destroy(clisock_iface->mi_tx_lock)
0043c344              clisock_iface->mi_tx_lock = nullptr
0043c35c          MT_MSG_free(mt_msg: clisock_iface->pending_sreq)
0043c368          $v0 = clisock_iface
0043c370          $v0->pending_sreq = 0
0043c390      return $v0
```

Like I said, a lot of cleanup, but the only one we care about for now is [18], since the `THREAD_destroy` function is critical to our understanding of this bug:

```
0045a34c  struct thread_struct* THREAD_destroy(struct thread_struct* thread)

0045a370      struct thread_struct* isthread = check_if_thread(t: thread)
0045a380      struct thread_struct* $v0 = isthread
0045a388      if ($v0 != 0)
0045a3a0          if (zx.d(isthread->has_inner_thread) != 0)
0045a3b0              isthread->idk2.b = 1
0045a3d4              _THREAD_destroy(isthread->innerthread)      // [19]
0045a3e8              isthread->innerthread = nullptr
```

Just as `_THREAD_create` was nested within `THREAD_create`, so too are the destruction functions nested. Let's quickly jump into `_THREAD_destroy` before coming back here:

```
0044d420  int32_t _THREAD_destroy(struct INNERTHREAD* thread)
0044d454      void** var_10 = &global_clisock_list.inner.threadlist
0044d458      _ATOMIC_global_lock()
0044d4b0      while (*var_10 != 0)
0044d480          if (*var_10 == thread)
0044d480              break
0044d49c          var_10 = *var_10 + 0x1c
0044d4cc      if (var_10 != 0)
0044d4e8          *var_10 = thread->global_threadlist
0044d4f4      thread->global_threadlist = nullptr
0044d4f8      _ATOMIC_global_unlock()
0044d524      pthread_cancel(thread->pthread_t)  // [20]
0044d538      thread->init = 0
0044d564      memset(thread, 0, 0x20)
0044d59c      return free(thread)
```

After assorted locking and global structure cleanup, we hit `pthread_cancel` at [20], which schedules this `mt_msg_rx_thread` for destruction. After this, the thread is nulled out and freed, and we have no trace of the `pthread_t` object given to us by `pthread_create` anymore. This is important because there's no cross references to `pthread_join()` anywhere in this function or this binary. Why might we want to call `pthread_join()`? Let's ask the `pthread_cancel` manual:

```
... the return status of pthread_cancel() merely informs the caller whether the
cancellation request was successfully queued.
After a canceled thread has terminated, a join with that thread using
pthread_join(3) obtains PTHREAD_CANCELED as the thread's exit status.
(Joining with a thread is the only way to know that cancellation has completed.)
```

The last line there is clearly the most important, so let me reiterate: without `pthread_join()`, there's no guarantee that the `mt_msg_rx_thread` has actually canceled at a given point in time. While this might not be an issue under a normal traffic load, let's keep examining `THREAD_create` for issues that might cause processing delays and a subsequent extension of `mt_msg_rx_thread`'s lifespan after cancellation:

```
0045a34c  struct thread_struct* THREAD_destroy(struct thread_struct* thread)

0045a370      struct thread_struct* isthread = check_if_thread(t: thread)
0045a380      struct thread_struct* $v0 = isthread
0045a388      if ($v0 != 0)
0045a3a0          if (zx.d(isthread->has_inner_thread) != 0)
0045a3b0              isthread->idk2.b = 1
0045a3d4              _THREAD_destroy(isthread->innerthread)              // [19]
0045a3e8              isthread->innerthread = nullptr
0045a3ec          atomic_global_lock()                                    // [20]
0045a400          struct thread_struct* thread_iter = &known_thread_list   // [21]
0045a454          while (thread_iter->globalptr != 0)
0045a420              if (thread_iter->globalptr == isthread)
0045a420                  break
0045a43c              thread_iter = &thread_iter->globalptr->next
0045a478          if (thread_iter->globalptr == 0)
0045a4b8              char* var_20 = isthread->iface_name
0045a4c4              struct thread_struct* var_1c = isthread
0045a4f0              LOG_bug_here(0x479958, 0x479a2c, 0x159, 0x4799dc)
{"src/threads.c"}  {"Thread (%s) @ %p not found in list of known thread…"}
{"THREAD_destroy"}
0045a4f0              noreturn
0045a49c          *thread_iter = thread_iter->globalptr->next
0045a4fc          atomic_global_unlock()                                  // [22]
0045a518          if (isthread->iface_name != 0)
0045a540              free_const(isthread->iface_name)
0045a574          memset(isthread, 0, 0x28)
0045a590          $v0 = free(isthread)
0045a5b8      return $v0
```

At [20] we lock the global thread lock, and then at [21], we actually walk the entire linked list of threads, the size of which is at least `(2*x)` where `x` is the number of our connections. After these looped operations, we finally unlock at [22]. Let us compare the amount of operations for an addition to this thread list in `THREAD_create`:

```
0045a1e0          atomic_global_lock()
0045a1fc          t_struct->next = known_thread_list.globalptr
0045a20c          known_thread_list.globalptr = t_struct
0045a210          atomic_global_unlock()
```

Thus we see that the speed of thread creation is constant and small, whilst the speed of thread deletion is dependent on the number of threads we already have. So if we happen to constantly connect and disconnect with a network socket, there ends up being a build up of `mips_collector` threads in memory. The more threads there are, the less likely that a given thread is going to be able to execute. The most important fact we must keep in mind during all this is that `pthread_cancel` has already been called at [19] without a `pthread_join`. So we've got a bunch of threads all scheduled for destruction, and we're still actually freeing resources that these threads use. If we return back to `MT_MSG_interfaceDestroy`:

```
0043c1ac          if (clisock_iface->mt_msg_rx_thread != 0)
0043c1d4              THREAD_destroy(thread: clisock_iface->mt_msg_rx_thread)  //
[18]
0043c1e8              clisock_iface->mt_msg_rx_thread = nullptr
0043c1fc          free_mt_msg_queue(mtarg2: &clisock_iface->mt_msg_queue)
0043c218          if (clisock_iface->mi_lock != 0)
0043c240              MUTEX_destroy(clisock_iface->mi_lock)
0043c254              clisock_iface->mi_lock = nullptr
0043c268          if (clisock_iface->srsp_semaphore != 0)
0043c290              SEMAPHORE_destroy(clisock_iface->srsp_semaphore)
0043c2a4              clisock_iface->srsp_semaphore = nullptr
0043c2b8          if (clisock_iface->frag_semaphore != 0)
0043c2e0              SEMAPHORE_destroy(clisock_iface->frag_semaphore)
0043c2f4              clisock_iface->frag_semaphore = nullptr
0043c308          if (clisock_iface->mi_tx_lock != 0)
0043c330              MUTEX_destroy(clisock_iface->mi_tx_lock)
0043c344              clisock_iface->mi_tx_lock = nullptr
0043c35c          MT_MSG_free(mt_msg: clisock_iface->pending_sreq)
0043c368          $v0 = clisock_iface
0043c370          $v0->pending_sreq = 0
0043c390      return $v0
```

Since all of these resources are freed and are shared between threads (due to pthreads being used), and also because we have a bunch of threads scheduled to be terminated that are still running (since there's no `pthread_join()`), if there is enough build up of pthreads, we end up in a situation where our condemned `mt_msg_rx_threads` end up accessing freed resources in a variety of spots before they fully die:

```
[  293.572000] do_page_fault() #2: sending SIGSEGV to mips_collector(14899) for
invalid read access from
[  293.572000] 626e2038 (pc == 00454224, ra == 00454a7c)

[ 5510.340000] do_page_fault() #2: sending SIGSEGV to mips_collector(29100) for
invalid read access from
[ 5510.340000] 00000004 (pc == 77dd866c, ra == 77dd17c4)

[ 5543.732000] do_page_fault() #2: sending SIGSEGV to mips_collector(30069) for
invalid read access from
[ 5543.732000] 7cfffd84 (pc == 7767c66c, ra == 776757c4)

[ 5544.440000] ####Set_SignalUserPid_Proc,5897
[ 5462.384000] do_page_fault() #2: sending SIGSEGV to mips_collector(27384) for
invalid read access from
[ 5462.384000] 7d1ffd84 (pc == 7733466c, ra == 7732d7c4)

[ 5427.136000] do_page_fault() #2: sending SIGSEGV to mips_collector(26154) for
invalid read access from
[ 5427.136000] 626e2038 (pc == 00454224, ra == 00454a7c)

[ 5439.964000] do_page_fault() #2: sending SIGSEGV to mips_collector(26594) for
invalid read access from
[ 5439.964000] 7c5ffd84 (pc == 773d466c, ra == 773cd7c4)
```

Crash Information

```
[  666.160000]
[  666.160000] do_page_fault() #2: sending SIGSEGV to mips_collector(16359) for
invalid read access from
[  666.160000] 7ebffd84 (pc == 77af866c, ra == 77af17c4)


<(^.^)>#bt
#0  0x77af866c in strnlen () from /lib/libc.so.0
#1  0x77af17c4 in _vfprintf_internal () from /lib/libc.so.0
#2  0x77aee8b8 in vsnprintf () from /lib/libc.so.0
Backtrace stopped: frame did not save the PC
<(^.^)>#info reg
          zero       at       v0       v1       a0       a1       a2       a3
 R0   00000000 1100ff00 7ebffd84 7ebffd84 7ebffd84 ffffffff 80808080 fefefeff
            t0       t1       t2       t3       t4       t5       t6       t7
 R8   00000001 00000002 00000200 00000100 00000807 00000800 00000400 00000008
            s0       s1       s2       s3       s4       s5       s6       s7
 R16  00000000 7d1ffb38 77af6000 77b53a30 77af1210 77b53a48 00476760 00000000
            t8       t9       k0       k1       gp       sp       s8       ra
 R24  00000000 77af85c0 00000000 00000000 77b6f490 7d1ff988 7d1ffba0 77af17c4
            sr       lo       hi      bad    cause       pc
      0100ff13 000000a8 000000c0 7ebffd84 80800008 77af866c
           fsr      fir
      00000000 00000000

<(^.^)>#x/5i $pc-0x10
   0x77af865c <strnlen+156>:    addiu   v0,v1,3
   0x77af8660 <strnlen+160>:    addiu   v1,v1,4
   0x77af8664 <strnlen+164>:    move    v0,a1
   0x77af8668 <strnlen+168>:    sltu    t0,v1,a1
=> 0x77af866c <strnlen+172>:    bnezl   t0,0x77af861c <strnlen+92>
   0x77af8670 <strnlen+176>:    lw      v0,0(v1)
```

TIMELINE

2022-01-11 - Vendor Disclosure

2022-06-10 - Vendor Patch Release

2022-06-15 - Public Release

CREDIT

Discovered by Lilith >_> of Cisco Talos.

TALOS-2021-1438 TALOS-2022-1461