Talos Vulnerability Report

TALOS-2022-1463

TCL LinkHub Mesh Wifi GetValue buffer overflow vulnerability

AUGUST 1, 2022

CVE NUMBER

CVE-2022-24021,CVE-2022-24011,CVE-2022-24028,CVE-2022-24023,CVE-2022-24026,CVE-2022-24016,CVE-2022-24005,CVE-2022-24019,CVE-2022-24029,CVE-2022-24007,CVE-2022-24017,CVE-2022-24008,CVE-2022-24006,CVE-2022-24013,CVE-2022-24009,CVE-2022-24010,CVE-2022-24020,CVE-2022-24015,CVE-2022-24012,CVE-2022-24022,CVE-2022-24014,CVE-2022-24027,CVE-2022-24025,CVE-2022-24018,CVE-2022-24024

SUMMARY

A buffer overflow vulnerability exists in the GetValue functionality of TCL LinkHub Mesh Wi-Fi MS1G_00_01.00_14. A specially-crafted configuration value can lead to a buffer overflow. An attacker can modify a configuration value 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.

TCL LinkHub Mesh Wifi MS1G_00_01.00_14

PRODUCT URLS

LinkHub Mesh Wifi - https://www.tcl.com/us/en/products/connected-home/linkhub/linkhub-mesh-wifi-system-3-pack

CVSSV3 SCORE

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

CWE

CWE-120 - Buffer Copy without Checking Size of Input (‘Classic Buffer Overflow’)

DETAILS

The LinkHub Mesh Wi-Fi system is a node-based mesh system designed for Wi-Fi deployments across large homes. These nodes include most features standard in current Wi-Fi solutions and allow for easy expansion of the system by adding nodes. The mesh is managed solely by a phone application, and the routers have no web-based management console.

`SetValue` and `GetValue` are functions that the Linkhub heavily relies upon for getting configuration values to and from flash. These functions are wrappers for lower level functionality that passes messages over a socket file from various binaries across the system. Specifically, `GetValue` does not take into account the destination buffer size when copying data from the flash configuration, and as such can cause buffer overflows to occur at every instance of the function given control over the flash variable. This can be done using `cfm` the command line utility, or remotely using protobuf messages similar to those used by the TCL phone application. This vulnerability as per MITRE's definition of CVE should contain 1874 unique CVE's, but for the sake of brevity and usefulness, a CVE is only being issued for each binary that contains at least 1 call to `GetValue`.

One of the simplest examples of this vulnerability occurs in the same functionality as a previously reported vulnerability. Specifically the function responsible for changing the `sys.app.lang` variable from a protobuf message, `confctl_set_app_language`.

```
00416b4c  int32_t confctl_set_app_language(int32_t arg1, int32_t arg2, int32_t arg3)

00416b6c      arg_0 = arg1
00416b78      int32_t $a3
00416b78      arg_c = $a3
00416b80      int32_t $v0_1
00416b80      if (arg2 == 0) {
00416ba8          printf("[%s][%d][luminais] invalid param…",
"confctl_set_app_language", 0x114)
00416bb4          $v0_1 = 0xffffffff
00416bb4      } else {
00416bc0          int32_t var_224_1 = 0
00416bc4          int32_t var_228_1 = 0
00416be4          uint8_t var_21c[0x100]
00416be4          memset(&var_21c, 0, 0x100)
00416c0c          uint8_t var_11c[0x100]
00416c0c          memset(&var_11c, 0, 0x100)
00416c18          int32_t var_1c = 0
00416c1c          int32_t var_18_1 = 0
00416c20          int32_t var_14_1 = 0
00416c24          int32_t var_10_1 = 0
00416c38          unlink("/var/wan_detect_rst")
00416c60          struct AppLang* pkt = applang__unpack(0, arg3, arg2)
00416c74          if (pkt == 0) {
00416c9c              printf("[%s][%d][luminais] applang__unpa…",
"confctl_set_app_language", 0x123)
00416ca8              $v0_1 = 0xffffffff
00416ca8          } else {
00416cbc              if (pkt->lang != 0) {
00416ce0                  strcpy(&var_11c, pkt->lang)
[1]
00416d18                  var_224_1 = set_if_changed("sys.app.lang", &var_11c,
&var_21c)                  [2]
00416d00              }
00416d24              if (pkt->is_timestamp_present != 0 && is_rst_some_status() ==
0) {
00416d70                  sprintf(&var_11c, "%llu", pkt->timestamp.d, pkt-
>timestamp:4.d, 0x4ae4b0)
00416d94                  memset(&var_21c, 0, 0x100)
00416dcc                  var_228_1 = set_if_changed(0x47c4b8, &var_11c, &var_21c)
{"sys.cfg.stamp"}
00416db4              }
00416de0              if (var_224_1 != 0 || (var_224_1 == 0 && var_228_1 != 0)) {
00416df0                  CommitCfm()
00416e00                  if (var_224_1 != 0) {
00416e28                      sprintf(&var_1c, "op=%d", 0x12)
00416e48                      send_msg_to_netctrl(2, &var_1c)
00416e3c                  }
00416e00              }
00416e64              applang__free_unpacked(pkt, 0)
00416e78              set_extdns_by_lang()
[3]
00416e84              $v0_1 = 0
00416e84          }
00416e84      }
00416e98      return $v0_1
```

Here we see the obvious stack-based buffer overflow that occurs at [1]. If we provide a `lang` that is 0x11c bytes long, we will overwrite the return address. If we ignore that and go deeper into the function we see that at [2], `set_if_changed` is called, and then at [3] `set_extdns_by_lang` is called from `libcommonprod.so`. `set_if_changed` is a simple function that will compare the value currently in flash to the newly-provided value and then set the key to the new value if it is different, as seen below.

```
0000aeac  int32_t set_if_changed(char* key, char* newVal, char* oldVal)

0000aef4      int32_t $v0_3
0000aef4      if (key == 0 || (key != 0 && newVal == 0) || (key != 0 && newVal != 0
&& oldVal == 0)) {
0000aefc          $v0_3 = 0xffffffff
0000aefc      }
0000aef4      if (key != 0 && newVal != 0 && oldVal != 0) {
0000af18          GetValue(key, oldVal)  // Get the old flash value
0000af40          if (strcmp(newVal, oldVal) == 0) {  // If the values are the same,
return 0
0000af70              $v0_3 = 0
0000af70          } else {
0000af58              SetValue(key, newVal)  // If the strings are not the same, set
the key to the new value and return 1
0000af64              $v0_3 = 1
0000af64          }
0000af64      }
0000af84      return $v0_3
```

Using this function, we have direct control over the `sys.app.lang` flash variable remotely, and we can move on to the `set_extdns_by_lang` in assembly.

```
0000ec6c  int32_t set_extdns_by_lang()

0000ec6c  02001c3c…li      $gp, 0x20784
0000ec74  21e09903   addu    $gp, $gp, $t9
0000ec78  d0ffbd27   addiu   $sp, $sp, -0x30
0000ec7c  2c00bfaf   sw      $ra, 0x2c($sp) {__saved_$ra}
0000ec80  2800beaf   sw      $fp, 0x28($sp) {__saved_$fp}
0000ec84  21f0a003   move    $fp, $sp {var_30}
0000ec88  1000bcaf   sw      $gp, 0x10($sp) {var_20}  {0x2f3f0}
0000ec8c  1800c0af   sw      $zero, 0x18($fp) {var_18[0].d}  {0x0}
0000ec90  1c00c0af   sw      $zero, 0x1c($fp) {var_18[4].d}  {0x0}
0000ec94  2000c0af   sw      $zero, 0x20($fp) {var_18[8].d}  {0x0}
0000ec98  2400c0af   sw      $zero, 0x24($fp) {var_18[0xc].d}  {0x0}
0000ec9c  2480828f   lw      $v0, -0x7fdc($gp)  {data_27414}
0000eca0  98474424   addiu   $a0, $v0, 0x4798  {data_14798, "sys.app.lang"}
[4]
0000eca4  1800c227   addiu   $v0, $fp, 0x18 {var_18}
0000eca8  21284000   move    $a1, $v0 {var_18}
[5]
0000ecac  9481828f   lw      $v0, -0x7e6c($gp)  {GetValue}
0000ecb0  21c84000   move    $t9, $v0
0000ecb4  09f82003   jalr    $t9
0000ecb8  00000000   nop
...
```

We can see at [4] that we are loading `sys.app.lang` as the value to retrieve, and at [5] we are setting the destination buffer to a stack address `var_18`. This means, if we can write 0x18 bytes to this buffer, we can overwrite the return address of this function. Next we move on to the implementation of `GetValue`, which is extrodinarily simple. It is a wrapper around `cfms_mib_proc_handler` with a hardcoded third argument.

```
00035800  int32_t GetValue(char* arg1, char* arg2)

0003585c      return cfms_mib_proc_handle(arg1, arg2, 4)
```

The assembly version is also included for reference.

```
00035800  int32_t GetValue(char* arg1, char* arg2)

00035800  04001c3c…li      $gp, 0x40a10
00035808  21e09903   addu    $gp, $gp, $t9
0003580c  e0ffbd27   addiu   $sp, $sp, -0x20
00035810  1c00bfaf   sw      $ra, 0x1c($sp) {__saved_$ra}
00035814  1800beaf   sw      $fp, 0x18($sp) {__saved_$fp}
00035818  21f0a003   move    $fp, $sp {var_20}
0003581c  1000bcaf   sw      $gp, 0x10($sp) {var_10}  {data_76210}
00035820  2000c4af   sw      $a0, 0x20($fp) {arg_0}
00035824  2400c5af   sw      $a1, 0x24($fp) {arg_4}
00035828  2000c48f   lw      $a0, 0x20($fp) {arg_0}
0003582c  2400c58f   lw      $a1, 0x24($fp) {arg_4}
00035830  04000624   addiu   $a2, $zero, 4                              //hard coded
third argument
00035834  5480828f   lw      $v0, -0x7fac($gp)  {data_6e264}
00035838  ec4e4224   addiu   $v0, $v0, 0x4eec  {cfms_mib_proc_handle}
0003583c  21c84000   move    $t9, $v0  {cfms_mib_proc_handle}
00035840  09f82003   jalr    $t9  {cfms_mib_proc_handle}
00035844  00000000   nop
00035848  1000dc8f   lw      $gp, 0x10($fp) {var_10}  {data_76210}
0003584c  21e8c003   move    $sp, $fp
00035850  1c00bf8f   lw      $ra, 0x1c($sp) {__saved_$ra}
00035854  1800be8f   lw      $fp, 0x18($sp) {__saved_$fp}
00035858  2000bd27   addiu   $sp, $sp, 0x20
0003585c  0800e003   jr      $ra
00035860  00000000   nop
```

So moving on to the relevent code within `cfms_mib_proc_handle` seen below

```
00034eec  int32_t cfms_mib_proc_handle(char* arg1, char* arg2, int32_t messageType)

00034f38      char cfms_msg_buffer[0x7e0]
00034f38      memset(&cfms_msg_buffer, 0, 0x7e0)
00034f44      void* var_20 = nullptr
00034f48      struct UgwProcMsgHeader* var_1c = nullptr
00034f54      int32_t var_808 = 0xffffffff
00034f5c      cfms_msg_buffer[0].d = messageType
00034f68      int32_t $v0_10
00034f68      if (messageType u< 0x26) {
00034f74          switch (messageType) {
...
00034f98              case 4, 0xe, 0x12, 0x15, 0x1b, 0x23  // In this case, arg1 is
the keyInput to retrieve
00034f98                  int32_t $v0_9
00034f98                  if (arg1 != 0) {
00034fbc                      $v0_9 = strnlen(arg1, 0x200) u< 0x200 ? 1 : 0  //
Ensure that keyInput < 0x200 bytes long
00034fc0                      if ($v0_9 != 0) {
00034ff0                          strncpy(dest: &cfms_msg_buffer[4], src: arg1, n:
0x200)  // Build the cfms_msg by copying in the key we want to retrieve
00035170                          label_35170:
00035170                          int32_t $v0_22 = ugw_connect_server(2)
00035184                          if ($v0_22 s< 0) {
000351ac                              printf("func:%s, line:%d connect cfmd is…",
"cfms_mib_proc_handle", 0xd7)
000351b8                              $v0_10 = 0
000351b8                          } else {
000351c8                              int32_t var_18 = 2
000351cc                              int32_t var_14_1 = 0
000351e4                              ugw_set_socket_timeout($v0_22, &var_18)
00035220                              int32_t var_804_2
00035220                              if (cfms_encode_msg(&var_20, &cfms_msg_buffer)
== 1) {
00035228                                  var_804_2 = 0
00035228                              } else if (cfms_proc_send_msg($v0_22, var_20)
== 1) {
00035268                                  var_804_2 = 0
00035268                              } else {
0003528c                                  memset(&cfms_msg_buffer, 0, 0x7e0)
000352c0                                  if (ugw_proc_recv_msg($v0_22, OUT_buffer:
&var_1c) s<= 0) {
000352c8                                      var_804_2 = 0
000352ec                                      printf("func:%s, line:%d, recv msg is
fa…", "cfms_mib_proc_handle", 0xf1)
000352e0                                  } else if (cfms_decode_msg(var_1c,
&cfms_msg_buffer) == 1) {
00035338                                      var_804_2 = 0
0003535c                                      printf("func:%s, line:%d, decode ie
data…", "cfms_mib_proc_handle", 0xf9)
00035350                                  } else {
00035374                                      int32_t $v0_31 = cfms_msg_buffer[0].d
- 3                                                                    [6]
0003537c                                      if ($v0_31 u>= 0x24) {
000354b0                                          label_354b0:
000354b0                                          var_804_2 = 0
000354b0                                      } else {
00035384                                          switch ($v0_31) {
```

```
00035490                                                case 0, 0xc, 0x10, 0x17, 0x21
0003549c                                                    if
(strncmp(&cfms_msg_buffer[4], arg1, 0x200) == 0) {
0003549c                                                        goto label_354d0
0003549c                                                    }
000354a4                                                    var_804_2 = 0
0003539c                                                case 1, 3, 4, 5, 6, 7, 8, 9,
0xb, 0xd, 0xf, 0x12, 0x14, 0x16, 0x18, 0x1a, 0x1c, 0x1e, 0x20, 0x22
0003539c                                                    goto label_354b0
000353c0                                                case 2, 0x13, 0x19
[7]
000353c0                                                    int32_t $v0_35 =
strncmp(&cfms_msg_buffer[4], arg1, 0x200)
000353d8                                                    if ($v0_35 != 0 || ($v0_35
== 0 && sx.d(cfms_msg_buffer[0x204]) == 0)) {
000353e4                                                        *arg2 = 0
000353e8                                                        var_804_2 = 0
000353e8                                                    }
000353d8                                                    if ($v0_35 == 0 &&
sx.d(cfms_msg_buffer[0x204]) != 0) {
00035430                                                        strncpy(dest: arg2,
src: &cfms_msg_buffer[0x204], n: strlen(&cfms_msg_buffer[0x204]))           [8]
00035464
arg2[strnlen(&cfms_msg_buffer[0x204], 0x5dc)] = 0
000354d0                                                        label_354d0:
000354d0                                                        var_804_2 = 1
000354d0                                                    }
0003539c                                                case 0xa, 0xe, 0x11, 0x15,
0x1b, 0x1d, 0x1f, 0x23
0003539c                                                    goto label_354d0
0003539c                                            }
0003539c                                        }
0003539c                                    }
0003539c                                }
000354e0                                ugw_socket_shut_down($v0_22)
000354ec                                $v0_10 = var_804_2
000354ec                            }
000354ec                        }
```

We are only interested in case 4 since we know that it is hardcoded in `Getvalue`. `cfms_mib_proc_handle` is responsible for formatting the request properly, which includes making sure the key value is less than 0x200 bytes, connecting and sending via the proper socket, and then parsing the response. Since we need to know the value of `cfms_msg_buffer` to know which of the second switch cases to follow, we need to look at `cmfs_handle_socket` within `cfmd`.

```
00402cc4  int32_t cfms_handle_socket(int32_t arg1)

00402ce8      int32_t var_7f4 = 0
00402cec      int32_t var_7f0 = 0
00402d0c      int32_t var_7ec
00402d0c      memset(&var_7ec, 0, 0x7e0)
00402d18      int32_t var_7f8 = 0
00402d34      memset(&var_7ec, 0, 0x7e0)
00402d68      int32_t $v0_2
00402d68      if (ugw_proc_recv_msg(arg1, &var_7f4) s<= 0) {
00402d70          $v0_2 = 0
00402d70      } else if (cfms_decode_msg(var_7f4, &var_7ec) == 1) {
00402db4          $v0_2 = 0
00402db4      } else {
00402dc0          int32_t $v0_4 = var_7ec
00402dc8          void var_7e8
00402dc8          void var_5e8
00402dc8          switch ($v0_4) {
00402e50              case 2
00402e50                  SetCfmValue(&var_7e8, &var_5e8)
00402e60                  var_7ec = 3
00402e14              case 4
00402e14                  GetCfmValue(&var_7e8, &var_5e8, 0x5dc)
00402e24                  var_7ec = 5
[9]
...
    00403174          if ($v0_4 == 2 || $v0_4 == 4 || $v0_4 == 0xc || $v0_4 == 0xe
|| $v0_4 == 0x10 || $v0_4 == 0x12 || $v0_4 == 0x15 || $v0_4 == 0x17 || $v0_4 == 0x19
|| $v0_4 == 0x1b || $v0_4 == 0x1d || $v0_4 == 0x1f || $v0_4 == 0x21 || $v0_4 == 0x23
|| $v0_4 == 0x25) {
00403180              if (cfms_encode_msg(&var_7f0, &var_7ec) != 0) {
00403200                  $v0_2 = 0
00403200              } else if (cfms_proc_send_msg(arg1, var_7f0) != 1) {
004031f4                  $v0_2 = 1
004031f4              } else {
004031dc                  printf(0x40c2e8, 0x40c45c, 0x1c0)  {"func:%s, line:%d,
send msg is er…"}  {"cfms_handle_socket"}
004031e8                  $v0_2 = 0
004031e8              }
004031e8          }
00402dc8      }
00403214      return $v0_2
```

We know that the switch case is going to be hard coded to case 4, based on the `GetValue`, and thus we need to look at `GetCfmValue`. Included below are both the psuedocode and the assembly versions.

```
00409420  int32_t GetCfmValue(int32_t arg1, int32_t dst, int32_t max_str_length)

00409420  02001c3c…li      $gp, 0x1d660
00409428  21e09903   addu    $gp, $gp, $t9
0040942c  d8ffbd27   addiu   $sp, $sp, -0x28
00409430  2400bfaf   sw      $ra, 0x24($sp) {__saved_$ra}
00409434  2000beaf   sw      $fp, 0x20($sp) {__saved_$fp}
00409438  21f0a003   move    $fp, $sp {var_28}
0040943c  1000bcaf   sw      $gp, 0x10($sp) {var_18}  {0x426a80}
00409440  2800c4af   sw      $a0, 0x28($fp) {arg_0}
00409444  2c00c5af   sw      $a1, 0x2c($fp) {arg_4}
00409448  3000c6af   sw      $a2, 0x30($fp) {arg_8}
0040944c  1c00c0af   sw      $zero, 0x1c($fp) {var_c}  {0x0}
00409450  1800c0af   sw      $zero, 0x18($fp) {var_10}  {0x0}
00409454  1c00c227   addiu   $v0, $fp, 0x1c {var_c}
00409458  2800c48f   lw      $a0, 0x28($fp) {arg_0}
0040945c  21284000   move    $a1, $v0 {var_c}
00409460  2480828f   lw      $v0, -0x7fdc($gp)  {data_41eaa4}
00409464  fc874224   addiu   $v0, $v0, -0x7804  {find_in_hashtable}
00409468  21c84000   move    $t9, $v0  {find_in_hashtable}
0040946c  09f82003   jalr    $t9  {find_in_hashtable}
00409470  00000000   nop
00409474  1000dc8f   lw      $gp, 0x10($fp) {var_18}
00409478  1800c2af   sw      $v0, 0x18($fp) {var_10_1}
0040947c  1800c38f   lw      $v1, 0x18($fp) {var_10_1}
00409480  01000224   addiu   $v0, $zero, 1
00409484  0c006214   bne     $v1, $v0, 0x4094b8
00409488  00000000   nop

0040948c  1c00c28f   lw      $v0, 0x1c($fp) {var_c}
00409490  21184000   move    $v1, $v0
00409494  3000c28f   lw      $v0, 0x30($fp) {arg_8}
00409498  2c00c48f   lw      $a0, 0x2c($fp) {arg_4}
0040949c  21286000   move    $a1, $v1
004094a0  21304000   move    $a2, $v0
004094a4  5481828f   lw      $v0, -0x7eac($gp)  {strncpy}
004094a8  21c84000   move    $t9, $v0
004094ac  09f82003   jalr    $t9
004094b0  00000000   nop
004094b4  1000dc8f   lw      $gp, 0x10($fp) {var_18}  {0x426a80}

004094b8  1800c28f   lw      $v0, 0x18($fp) {var_10_1}
004094bc  21e8c003   move    $sp, $fp
004094c0  2400bf8f   lw      $ra, 0x24($sp) {__saved_$ra}
004094c4  2000be8f   lw      $fp, 0x20($sp) {__saved_$fp}
004094c8  2800bd27   addiu   $sp, $sp, 0x28
004094cc  0800e003   jr      $ra
004094d0  00000000   nop
```

Using the more straightfoward pseudocode below, we can see at [10] that we have an enforced `strncpy` to a max length of 0x5dc. `arg1` is loaded from the configuration value hashtable and copied into `dst`, which is a buffer used to format a new `cfms_msg_buffer` to be sent back to `cfms_mib_proc_handle` at [7]. At [9] we can see the message

type is hard coded to 5. Looking back at [7] we can use this information to calculate the switch case of 2, which is what we are interested in.

```
00409420  int32_t GetCfmValue(int32_t arg1, int32_t dst, int32_t max_str_length)

0040944c      int32_t var_c = 0
00409450      int32_t var_10 = 0
0040946c      int32_t $v0 = find_in_hashtable(arg1, &var_c)
00409484      if ($v0 == 1) {
004094ac          strncpy(dst, var_c, max_str_length)
[10]
004094ac      }
004094cc      return $v0
```

Once we are looking at case 2, we see that at [8] we have a `strncpy` with a max length of `strlen` of the return value. This return value has a max length of 0x5dc from being retrieved in `cfms_handle_socket`, but `GetValue` has no way of determining the size of the buffer being provided for the output. Returning to [5], the output buffer is only 0x10 bytes long, and if we can write 0x18 bytes, we can overwrite the return address. This can be done easily and allow for arbitrary code execution. This single example is present all over the firmware of the device and appears in 25 different binaries. One CVE has been issued for each binary, but this actually should be 1874 different CVEs because the GetValue prototype needs to be fixed to include the buffer size, or the buffers need to be changed such that all of them are at least size 0x5dc.

CVE-2022-24005 - ap_steer

ap_steer has 1 calls to GetValue

CVE-2022-24006 - arpbrocast

arpbrocast has 2 calls to GetValue

CVE-2022-24007 - cfm

cfm has 1 calls to GetValue

CVE-2022-24008 - confcli

confcli has 209 calls to GetValue

CVE-2022-24009 - confsrv

confsrv has 359 calls to GetValue

CVE-2022-24010 - cwmpd

cwmpd has 331 calls to GetValue

CVE-2022-24011 - device_list

device_list has 6 calls to GetValue

CVE-2022-24012 - fota

fota has 2 calls to GetValue

CVE-2022-24013 - gpio_ctrl

gpio_ctrl has 4 calls to GetValue

CVE-2022-24014 - logserver

logserver has 5 calls to GetValue

CVE-2022-24015 - log_upload

log_upload has 3 calls to GetValue

CVE-2022-24016 - mesh_status_check

mesh_status_check has 5 calls to GetValue

CVE-2022-24017 - miniupnpd

miniupnpd has 6 calls to GetValue

CVE-2022-24018 - multiWAN

multiWAN has 127 calls to GetValue

CVE-2022-24019 - netctrl

netctrl has 493 calls to GetValue

CVE-2022-24020 - network_check

network_check has 11 calls to GetValue

CVE-2022-24021 - online_process

online_process has 2 calls to GetValue

CVE-2022-24022 - pann

pann has 95 calls to GetValue

CVE-2022-24023 - pppd

pppd has 4 calls to GetValue

CVE-2022-24024 - rtk_ate

rtk_ate has 13 calls to GetValue

CVE-2022-24025 - sntp

sntp has 1 calls to GetValue

CVE-2022-24026 - telnet_ate_monitor

telnet_ate_monitor has 1 calls to GetValue

CVE-2022-24027 - libcommon.so

libcommon.so has 114 calls to GetValue

CVE-2022-24028 - libcommonprod.so

libcommonprod.so has 76 calls to GetValue

CVE-2022-24029 - rp-pppoe.so

rp-pppoe.so has 3 calls to GetValue

TIMELINE

2022-02-08 - Initial Vendor Contact

2022-02-09 - Vendor Disclosure

2022-08-01 - Public Release

CREDIT

Discovered by Carl Hurd of Cisco Talos.

VULNERABILITY REPORTS

PREVIOUS REPORT

TALOS-2022-1462

NEXT REPORT

TALOS-2022-1482