Talos Vulnerability Report

TALOS-2022-1558

Abode Systems, Inc. iota All-In-One Security Kit XCMD setAlexa OS command injection vulnerability

OCTOBER 20, 2022

CVE NUMBER

CVE-2022-33189

SUMMARY

An OS command injection vulnerability exists in the XCMD setAlexa functionality of Abode Systems, Inc. iota All-In-One Security Kit 6.9Z. A specially-crafted XCMD can lead to arbitrary command execution. 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 6.9Z

PRODUCT URLS

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

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-78 - Improper Neutralization of Special Elements used in an OS Command (‘OS Command Injection’)

DETAILS

The iota All-In-One Security Kit is a home security gateway containing an HD camera, infrared motion detection sensor, Ethernet, WiFi 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 `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 an XML payload structured in a specific format. 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 various child elements that contain information relevent only to that handler.

For example, a standard XCMD for `setAlexa` might appear as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<p>
  <mac v="B0:C5:CA:33:64:CD"/>
  <cmds>
    <cmd a="setAlexa">
      <regCode v="88729812-0943-4354-af62-2bc8214b6b64"/>
    </cmd>
  </cmds>
</p>
```

This XCMD is used to configure Alexa integration by modifying DNS Service Discovery configuration values and then communicating to another thread that the configuration has been modified and the other thread should react. The `setAlexa` XCMD expects only one element, `regCode`, which is ultimately stored as the `Alexa_RegCode` configuration parameter.

After updating the `Alexa_RegCode` value, the handler signals a separate thread, one responsible for DNS resolution, to spawn a new DNS Service Discovery process, located at `/bct/sbin/dns-sd`. The thread responsible for DNS resolution spawns this service in an exploitable way.

The decompilation of the `setAlexa` handler is included below for reference.

```
int __fastcall setAlexa(xml_t *xcmd, xstrbuf_t *response)
{
  char *regCode;
  const char *err_fmt_str;

  // [*] This vuln does not affect 6.9X because it did not have this file
  if ( file_exists("/bct/sbin/dns-sd", 0) )
  {

    // [1] Extract the value of the `regCode` tag
    regCode = get_xcmd_param(xcmd, "regCode");
    if ( regCode )
    {
      // [2] Update (and save) the new configuration value
      set_config_value("Alexa_RegCode", regCode);
      if ( config_save() >= 0 )
      {

        // [3] Inter-thread comm to submit changes to DNS thread
        append_action(&dns, &xmpp, 0, 0, DoLearnJoinNetwork, "alexa", 0);
        init_xml_response(response);
        return 0;
      }
      else
      {
        err_fmt_str = strtable_get("XCMD_ERR_FLASH_WRITE", 20);
        xml_construct_error_response(response, 0, 3, err_fmt_str, xcmd, response);
        return -44;
      }
    }
    else
    {
      err_fmt_str = strtable_get("XCMD_ERR_PARAM_EMPTY", 20);
      xml_construct_error_response(response, 0, 6, err_fmt_str, "regCode",
response);
      return -1;
    }
  }
  else
  {
    err_fmt_str = strtable_get("XCMD_ERR_ITEM_NOT_EXIST", 23);
    xml_construct_error_response(response, 0, 27, err_fmt_str, xcmd, response);
    return -3;
  }
}
```

It should be noted ([*] above) that this function only updates the configuration and triggers the 'DoLearnJoinNetwork' action if the `/bct/sbin/dns-sd` binary exists. In firmware version 6.9X this binary was not in place (at least on the test system) and therefore was not vulnerable. With the release of 6.9Z, the binary was added to the device, and this vulnerability is now exploitable.

Above, at `[1]` the supplied `regCode` value is extracted. If it exists, then at `[2]` the value is stored into the `/root/config/config.json` configuration file under the key `Alexa_RegCode`. Finally, at `[3]` the DNS thread is signaled with the action ‘DoLearnJoinNetwork’.

When signaled, the thread responsible for DNS resolution executes a function titled `dns_action_proc_action`, responsible for processing all actions that are placed on its queue. Without going into unnecessary detail, the structure passed around contains an enum detailing which action is being requested—in this case, DoLearnJoinNetwork. Within the `dns_action_proc_action` we can identify the code that executes when this type of action is received by the DNS thread.

```
pitem_t *dns_action_proc_action()
{
  pitem_t *action;
  actions ActionType;
  char *model_number;
  char normalized_mac[32];
  char mac[32];
  char Alexa_RegCode[192];
  char cmd[256];
  ...
  action = get_action_item(&dns, 0);
  ...
  if ( action ) {

    // [4] If action is 'DoLearnJoinNetwork'
    if ( action->ActionType == DoLearnJoinNetwork )
    {

      // [5] and if the first argument provided was "alexa"
      argv0 = action->strArgv[0];
      if ( argv0 && !strcmp("alexa", argv1) )
      {

       // [6] Kill the first process whose /proc/*/cmdline contains "_alexa._tcp"
        pid = find_proc_by_cmdline_args("_alexa._tcp");
        if ( pid > 0 )
          kill(pid, 9);

        memset(Alexa_RegCode, 0, sizeof(Alexa_RegCode));

        // [7] Load the Alexa_RegCode value from /root/config/config.json
        fetch_config_value("Alexa_RegCode", Alexa_RegCode, 191);

        //  [8] If Alexa_RegCode contains data, and if the binary exists
        if ( Alexa_RegCode[0] && file_exists("/bct/sbin/dns-sd", 0) )
        {
          memset(normalized_mac, 0, 0x20u);
          memset(mac, 0, 0x20u);
          memset(dest, 0, 0x100u);
          fetch_config_value("MAC", mac, 31);
          normalize_mac(mac, normalized_mac);
          model_number = get_model_number__();

          // [9] Construct a command using the attacker supplied Alexa_RegCode
          vsnprintf_nullterm(
            cmd,
            0xFFu,
            "%s -R %s-%s _amzn-alexa._tcp,_wwa . 55060 \"version=1\" \"dsn=%s\"
\"rid=%s\"&",
            "/bct/sbin/dns-sd",
            model_number,
            normalized_mac,
            normalized_mac,
            Alexa_RegCode);
          log(6, 14, "[Alexa DICE] %s", cmd);

          // [10] Execute the command
          popen_write(cmd);
```

```
            strtable_get("LOG_MSG_STARTING", 16);
            strtable_get("LOG_MSG_DNS", 11);
            write_log(7u, 14);
          }
        }
        ...
      }
      ...
    }
    ...
  }
```

At [4] we identify whether the current `action` being processed is a ‘DoLearnNetworkJoin’ event—the type of event we submitted at [3]. If so, then at [5] the first argument is compared against “alexa”, which is the argument hard-coded into the event from [3], so it will match. At [6], any existing process related to DNS-SD is killed. At [7] the malicious `Alexa_RegCode` stored at [2] is fetched. At [8], the system ensures that the binary it wishes to execute exists and that the configuration value exists. At [9] the `Alexa_RegCode` value is injected into the OS command. Finally, at[10] the command is executed.

Exploit Proof of Concept

Submitting the following XCMD:

```
<?xml version="1.0" encoding="UTF-8"?>
<p>
  <mac v="B0:C5:CA:00:00:00"/>
  <cmds>
    <cmd a="setAlexa">
      <regCode v="`sleep 11`"/>
    </cmd>
  </cmds>
</p>
```

will maliciously alter the `Alexa_RegCode` configuration value, resulting in the `dns_action_proc_action` function executing the following command:

```
/bct/sbin/dns-sd -R Z3-xxxxxxxxxxxx _amzn-alexa._tcp,_wwa . 55060 "version=1"
"dsn=b0c5ca000000" rid="`sleep 11`"
```

TIMELINE

2022-07-13 - Initial Vendor Contact

2022-07-14 - Vendor Disclosure

2022-10-20 - Public Release

CREDIT

Discovered by Matt Wiseman of Cisco Talos.

VULNERABILITY REPORTS PREVIOUS REPORT NEXT REPORT

TALOS-2022-1557 TALOS-2022-1559