Talos Vulnerability Report

TALOS-2022-1581

Abode Systems, Inc. iota All-In-One Security Kit XCMD testWifiAP format string injection vulnerabilities

OCTOBER 20, 2022

CVE NUMBER

CVE-2022-35876,CVE-2022-35875,CVE-2022-35877,CVE-2022-35874

SUMMARY

Four format string injection vulnerabilities exist in the XCMD testWifiAP functionality of Abode Systems, Inc. iota All-In-One Security Kit 6.9X and 6.9Z. Specially-crafted configuration values can lead to memory corruption, information disclosure and denial of service. An attacker can modify a configuration value and then execute an XCMD to trigger these vulnerabilities.

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

8.2 - CVSS:3.0/AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:L/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, 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 generates a significant volume of diagnostic logs, which it displays on its read-only physical UART console. These logs are formatted and put on the serial line inside of a function (located at offset `0xA3270`) which we refer to simply as `log`. The `log` function operates as a wrapper to `vsnprintf` and `puts` with the added functionality of prefixing supplied log messages with severity and task strings. The `log` function is variadic, and it crafts the final log message by passing the supplied `format` and variadic arguments to `vsnprintf`. If an attacker can inject content into the `format` parameter, they could potentially leak stack memory and write arbitrary memory.

```
/* Examples:
        log(6, 13, "Initialized SSL");  ->  [DBG!][NET ]Initialized SSL
        log(3, 13, "SSL init error: %d", error);  ->  [ERR!][NET ]SSL init error:
{error}
*/
void log(unsigned int severity, unsigned int task, const char *format, ...)
{
  char log_buffer[520];
  va_list var_args;

  va_start(var_args, format);
  if ( severity <= g_LOG_LEVEL && ((g_TASK_LOGGING_ENABLED_BITFIELD >> task) & 1) !=
0 )
  {
    // Prefix the message with the SEVERITY tag
    if ( severity >= MAX_SEVERITY )
      severity = MAX_SEVERITY;
    memcpy(log_buffer, g_SEVERITY_PREFIX[severity], 6u);

    // Prefix the message with the TASK tag
    if ( task >= MAX_TASK )
      task = MAX_TASK;
    memcpy(&log_buffer[6], g_TASK_PREFIX[task], 0xCu);

    // Populate remainder of message with format and var_args
    vsnprintf(&log_buffer[12], 499u, format, var_args);

    // Put crafted message on to UART
    puts(log_buffer);
  }
}
```

It is important to note that all output from format string injections stemming from misuse of the `log` function will only be available to a physically present attacker who has partially disassembled the device and connected to the UART console.

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.

The `testWifiAP` XCMD is used to validate the existing wireless network configuration, and it does not expect any parameters. The XCMD appears as follows.

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

The handler associated with `testWifiAP` is located at offset `0x104830` of the `/root/hpgw` binary included in version 6.9Z, and its decompilation is included here.

```
int __fastcall testWifiAP(xml_node_t *xcmd, xstrbuf_t *response)
{
  const char *err_str;
  wifi_config_t wifi_config;

  // [1] Copy the current wireless configuration into `wifi_config`
  fetch_wifi_config(&wifi_config);

  // [2] Pass the wireless configuration into the vulnerable function
  if ( do_test_wifiap(&wifi_config) )
  {
    err_str = strtable_get("XCMD_ERR_WIFI_AUTH", 18);
    xml_construct_error_response(response, 0, 82, err_str);
  }
  else
  {
    init_xml_response(response);
  }
  return 0;
}
```

The handler itself is very straightforward. At `[1]` it collects the current wireless configuration and passes it to a delegate function we've named `do_test_wifiap`. It is the `do_test_wifiap` that contains the vulnerability, but this function is only executed when the `testWifiAP` XCMD is received.

The function we refer to as `fetch_wifi_config` is located at offset `0x1C722C` of the `hpgw` binary included in firmware 6.9Z. Its very straightforward decompilation is included here.

```
void __fastcall fetch_wifi_config(wifi_config_t *config)
{
  fetch_config_value("WL_SSID", config->ssid, 191);
  fetch_config_value("WL_SSID_HEX", config->ssid_hex, 191);
  fetch_config_value("WL_AuthMode", config->auth_mode, 191);
  fetch_config_value("WL_WPAPSK", config->wpapsk, 191);
  fetch_config_value("WL_WPAPSK_HEX", config->wpapsk_hex, 191);
  fetch_config_value("WL_EncrypType", config->encryp_type, 191);
  fetch_config_value("WL_DefaultKeyID", config->default_key_id, 191);
  fetch_config_value("WL_Key", config->key, 191);
}
```

These Wi-Fi configuration values may be modified a few ways: by the user via mobile application or Abode web application; through the `setWifiAP` XCMD; and via either the `/action/wirelessPost` or `/action/configPost` endpoints of the device's local web interface. None of these mechanisms implement useful sanitization or sanity

checking on the values. For the purposes of the following vulnerabilities, we assume that the remote attacker has manipulated these parameters prior to triggering the vulnerable `testWifiAP` XCMD.

These configuration values are passed to the function we've named `do_test_wifiap`, which is located at offset `0x1c7d28`. The relevant portions of the decompilation are included below.

```
int __fastcall do_test_wifiap(wifi_config_t *config)
{
  int result;
  const char *static_command;
  int retry;
  int con_suc;
  const char *str_err;
  int wireless_enabled;
  char cmd_output[32];
  char command[256];

  wireless_enabled = 0;

  // [1] Ensure that wireless is enabled, otherwise exit early
  get_config_as_integer("WL_Enable", (int)&wireless_enabled);
  if ( !wireless_enabled )
    return -3;

  // [2] Ensure that one of `config->ssid` or `config->ssid_hex` is provided
  if ( !config->ssid[0] && !config->ssid_hex[0] )
  {
    log(7, 31, "No SSID!");
    return -2;
  }

  ...

  memset(command, 0, 0x80u);
  // [3] Identify whether the SSID is provided via the `config->ssid` or `config-
>ssid_hex` configuration value
  if ( config->ssid_hex[0] )
  {
    // [4] If via `config->ssid_hex`, inject directly into the command buffer
    log(7, 31, "with hex string");
    vsnprintf_nullterm(command, 0x7Fu, "driver/wpa_cli -i %s set_network 0 ssid %s",
"wlan0", config->ssid_hex);
  }
  else
  {
    // [5] If via `config->ssid`, inject directly into the command buffer
    log(7, 31, "with acii string");
    vsnprintf_nullterm(command, 0x7Fu, "driver/wpa_cli -i %s set_network 0 ssid
'\"%s\"'", "wlan0", config->ssid);
  }
  // [6] Call the log function with the format string injected with attacker-
controlled configuration values at [4] or [5]
  log(7, 1, command);

  popen_write(command);
  memset(command, 0, 0x80u);
  if ( strcmp(config->auth_mode, "WPA") && strcmp(config->auth_mode, "WPA2") )
  {

    // [7] If `config->auth_mode` is WPAPSK or WPA2PSK
    if ( !strcmp(config->auth_mode, "WPAPSK") || !strcmp(config->auth_mode,
"WPA2PSK") )
    {
      // [8] then inject `config->wpapsk` directly into the command buffer
```

```
      vsnprintf_nullterm(command, 0x7Fu, "driver/wpa_cli -i %s set_network 0 psk
'\"%s\"'", "wlan0", config->wpapsk);
      // [9] Call the log function with the format string injected with attacker-
controlled configuration values at [8]
      log(7, 1, command);
      p_command = command
    }

    // [10] Otherwise, if `config->auth_mode` is SHARED or WEP
    else if ( !strcmp(config->auth_mode, "SHARED") || !strcmp(config->auth_mode,
"WEP") )
    {
      log(7, 1, "driver/wpa_cli -i wlan0 set_network 0 key_mgmt NONE");
      popen_write("driver/wpa_cli -i wlan0 set_network 0 key_mgmt NONE");

      // [11] Construct a command buffer by injecting `config->default_key_id` and
`config->key`
      vsnprintf_nullterm(
        command,
        0x7Fu,
        "driver/wpa_cli -i %s set_network 0 wep_key%s '\"%s\"'",
        "wlan0",
        config->default_key_id,
        config->key);

      // [12] Call the log function with the format string injected with attacker-
controlled configuration values at [10]
      log(7, 1, command);

      popen_write(command);
      memset(command, 0, 0x80u);

      // [13] Then construct a second command buffer by injecting `config-
>default_key_id`
      vsnprintf_nullterm(
        command,
        0x7Fu,
        "driver/wpa_cli -i %s set_network 0 wep_tx_keyidx %s",
        "wlan0",
        config->default_key_id);

      // [14] Call the log function with the format string injected with attacker-
controlled configuration values at [12]
      log(7, 1, command);
      popen_write(command);

      if ( strcmp(config->auth_mode, "SHARED") )
        goto LABEL_19;
      log(7, 1, "driver/wpa_cli -i wlan0 set_network 0 auth_alg SHARED");
      p_command = "driver/wpa_cli -i wlan0 set_network 0 auth_alg SHARED";
    }
    else
    {
      log(7, 1, "driver/wpa_cli -i wlan0 set_network 0 key_mgmt NONE");
      p_command = "driver/wpa_cli -i wlan0 set_network 0 key_mgmt NONE";
    }
    popen_write(p_command);
  }
```

```
  ...
}
```

CVE-2022-35874 - `config->ssid/config->ssid_hex`

The first misuse of the `log` function occurs when logging the construction of an OS command meant to configure the device's Wi-Fi AP SSID. The SSID can be provided via either the `config->ssid` or `config->ssid_hex` configuration values, with `ssid_hex` taking priority if both are provided. Below is a partial decompilation of the `do_test_wifiap` function, with annotations.

```
...
memset(command, 0, 0x80u);
// [3] Identify whether the SSID is provided via the `config->ssid` or `config-
>ssid_hex` configuration value
if ( config->ssid_hex[0] )
{
    // [4] If via `config->ssid_hex`, inject directly into the command buffer
    log(7, 31, "with hex string");
    vsnprintf_nullterm(command, 0x7Fu, "driver/wpa_cli -i %s set_network 0 ssid %s",
"wlan0", config->ssid_hex);
}
else
{
    // [5] If via `config->ssid`, inject directly into the command buffer
    log(7, 31, "with acii string");
    vsnprintf_nullterm(command, 0x7Fu, "driver/wpa_cli -i %s set_network 0 ssid
'\"%s\"'", "wlan0", config);
}
// [6] Call the log function with the format string injected with attacker-
controlled configuration values at [4] or [5]
log(7, 1, command);
...
```

First, at `[3]`, it is determined whether to use the `ssid_hex` or `ssid` configuration value. In either case, the selected value will be used at `[4]`, or `[5]` to construct the command buffer. Finally, at `[6]` the injected command buffer is passed as the `format` parameter to the `log` function, resulting in attacker control of the format string.

Supplying a `config->ssid_hex` or `config->ssid` value of `%x.%x.%x.%x.%x.%x...` results in the following log message being generated:

```
[DBG ][WEB ]driver/wpa_cli -i wlan0 set_network 0 ssid
'"7148d871.7148d950.333a4143.252e51.0.33.76c46000.3a224e50.252e78"'
```

CVE-2022-35875 - `config->wpapsk`

This misuse of the `log` function occurs when logging the construction of an OS command meant to configure the WPAPSK of the wireless network. While the configuration values support the creation and fetching of a `config->ssid_hex` value, it is not used in this function. This vulnerability may only be triggered via the use of `config->ssid`. Below is the relevant portion of a decompilation of the `do_test_wifiap` function, with annotations.

```
...
if ( strcmp(config->auth_mode, "WPA") && strcmp(config->auth_mode, "WPA2") )
{
  // [7] If `config->auth_mode` is WPAPSK or WPA2PSK
  if ( !strcmp(config->auth_mode, "WPAPSK") || !strcmp(config->auth_mode, "WPA2PSK")
)
  {
    // [8] then inject `config->wpapsk` directly into the command buffer
    vsnprintf_nullterm(command, 0x7Fu, "driver/wpa_cli -i %s set_network 0 psk
'\"%s\"'", "wlan0", config->wpapsk);
    // [9] Call the log function with the format string injected with attacker-
controlled configuration values at [8]
    log(7, 1, command);
    p_command = command
  }
...
```

At `[7]`, it is first confirmed that the type of authentication required for the Wi-Fi AP is either WPA or WPA2. If so, then at `[8]` the `config->wpapsk` configuration value is used to construct the command buffer. Finally, at `[9]` the injected command buffer is passed as the `format` parameter to the `log` function, resulting in attacker control of the format string.

Supplying a `config->wpapsk` of `%x.%x.%x.%x.%x.%x...` results in the following log message being generated:

```
[DBG ][WEB ]driver/wpa_cli -i wlan0 set_network 0 psk
'"7148d870.7148db90.333a4143.252e51.0.33.76c46000.3a224e50.252e78"'
```

CVE-2022-35876 - `config->default_key_id` / `config->key`

This misuse of the `log` function occurs when logging the construction of an OS command meant to configure the WEP key management for the wireless network.The WEP key is provided via the `config->key` configuration value, and the key identifier is provided via the `config->default_key_id`. Below is the relevant portion of a decompilation of the `do_test_wifiap` function, with annotations.

```
...
// [10] Otherwise, if `config->auth_mode` is SHARED or WEP
else if ( !strcmp(config->auth_mode, "SHARED") || !strcmp(config->auth_mode, "WEP")
)
{
    log(7, 1, "driver/wpa_cli -i wlan0 set_network 0 key_mgmt NONE");
    popen_write("driver/wpa_cli -i wlan0 set_network 0 key_mgmt NONE");

    // [11] Construct a command buffer by injecting `config->default_key_id` and
`config->key`
    vsnprintf_nullterm(
      command,
      0x7Fu,
      "driver/wpa_cli -i %s set_network 0 wep_key%s '\"%s\"'",
      "wlan0",
      config->default_key_id,
      config->key);

    // [12] Call the log function with the format string injected with attacker-
controlled configuration values at [10]
    log(7, 1, command);
...
```

At `[10]`, it is first confirmed that the type of authentication required for the Wi-Fi AP is either WEP or SHARED. If so, then at `[11]` the `config->default_key_id` and `config->key` configuration values are used to construct the command buffer. Finally, at `[12]` the injected command buffer is passed as the `format` parameter to the `log` function, resulting in attacker control of the format string.

Supplying `%x.%x.%x.%x.%x.%x...` for both the `config->default_key_id` and `config->key` configuration values results in the following log message being generated:

```
[DBG ][WEB ]driver/wpa_cli -i wlan0 set_network 0
wep_key7148d88c.7148ddd0.7148de90.252e51.0.33.76c46000.3a224e50
'"252e78.7148df54.1.223a2243.220a2c22.5f534d4d.73736150.3a226477.a2c2222"'
```

CVE-2022-35877 - `config->default_key_id`

The final misuse of the `log` function within `testWifiAP` occurs almost immediately after the previous vulnerability, when logging the construction of an OS command meant to configure the WEP key management for the wireless network.The WEP key index is provided via the `config->default_key_id` configuration value. Below is the relevant portion of a decompilation of the `do_test_wifiap` function, with annotations.

```
// [13] Then construct a second command buffer by injecting `config->default_key_id`
vsnprintf_nullterm(
    command,
    0x7Fu,
    "driver/wpa_cli -i %s set_network 0 wep_tx_keyidx %s",
    "wlan0",
    config->default_key_id);

// [14] Call the log function with the format string injected with attacker-
controlled configuration values at [12]
log(7, 1, command);
popen_write(command);
```

At `[13]` the `config->default_key_id` is used to construct another OS command. At `[14]` the injected command buffer is passed as the `format` parameter to the `log` function, resulting in attacker control of the format string.

Supplying a `config->default_key_id` of `%x.%x.%x.%x.%x.%x...` results in the following log message being generated:

```
[DBG ][WEB ]driver/wpa_cli -i wlan0 set_network 0 wep_tx_keyidx
714ad874.714addd0.714ade90.252e51.0.33.76c66000.3a224e50
```

TIMELINE

2022-07-20 - Vendor Disclosure

2022-10-20 - Public Release

CREDIT

Discovered by Matt Wiseman of Cisco Talos.
