Talos Vulnerability Report

TALOS-2022-1583

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

OCTOBER 20, 2022

CVE NUMBER

CVE-2022-35881,CVE-2022-35880,CVE-2022-35878,CVE-2022-35879

SUMMARY

Four format string injection vulnerabilities exist in the UPnP logging functionality of Abode Systems, Inc. iota All-In-One Security Kit 6.9Z and 6.9X. A specially-crafted UPnP negotiation can lead to memory corruption, information disclosure, and denial of service. An attacker can host a malicious UPnP service 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

7.1 - CVSS:3.0/AV:A/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, 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 `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 crafts the final log message by passing the supplied `format` and variadic arguments to `vsnprintf`. If an attacker can inject content in to 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.

This log function is misused in three locations within the UPnP functionality of the Abode `iota`.

CVE-2022-35878 - DoEnumUPnPService - `ST/Location`

The first vulnerable instance of the `log` function occurs within the handler functionality associated with `DoEnumUPnPService` actions. The vulnerable call to `log` occurs at offset `0x1815A4` in the `hpgw` binary included in firmware version 6.9Z. This action can be generated by transmitting a `setIPCam` XCMD with an `action` tag containing the value `autoportfwd`. The `autoportfwd` functionality uses UPnP to set up port forwarding, where possible. The relevant portion of the `DoEnumUPnPService` handler function decompilation is included below, with annotations.

```
...
int tx_n = 0;
int rx_n = 0;
do
{
  while ( 1 )
  {
    if ( !rx_n )
    {
      // [1] Construct an M-SEARCH broadcast payload for URNs of interest (from
g_ST_LIST)
      tx_n = snprintf(
               m_search_payload,
               0x7FFu,
               "M-SEARCH * HTTP/1.1\r\n"
               "HOST: 239.255.255.250:1900\r\n"
               "ST: %s\r\n"
               "MAN: \"ssdp:discover\"\r\n"
               "MX: %u\r\n"
               "\r\n",
               g_ST_LIST[++urn_idx],
               timeout / 1000);
      memset(&req, 0, sizeof(req));
      req.ai_socktype = 2;
      if ( getaddrinfo("239.255.255.250", "1900", &req, &ai) )
      {
        log(3u, 0x11u, "upnp get addr info fail");
        v13 = strtable_get("LOG_MSG_FAIL", 12);
        v18 = strtable_get("LOG_MSG_UPNP", 12);
        v15 = (char *)"get addr info fail";
LABEL_16:
        log_to_file(7u, 17, v13, v18, v15, -1);
        close_fd(sock);
        return -3;
      }
      for ( i = ai; i; i = i->ai_next )
      {
        // [2] Transmit the M-SEARCH packet on all available interfaces
        tx_n = sendto(sock, m_search_payload, tx_n, 0, i->ai_addr, i->ai_addrlen);
        if ( tx_n < 0 )
          log(4u, 0x11u, "upnp sendto fail");
      }
      freeaddrinfo(ai);
      if ( tx_n < 0 )
      {
        log(3u, 0x11u, "upnp discovery null");
        v13 = strtable_get("LOG_MSG_FAIL", 12);
        v18 = strtable_get("LOG_MSG_UPNP", 12);
        v15 = (char *)"discovery null";
        goto LABEL_16;
      }
    }
    // [3] Listen for M-SEARCH replies
    rx_n = read_with_timeout(sock, m_search_payload, 0x800u, timeout);
    if ( rx_n < 0 )
    {
      log(3u, 0x11u, "upnp read error");
      v20 = strtable_get("LOG_MSG_FAIL", 12);
```

```
      v21 = strtable_get("LOG_MSG_UPNP", 12);
      log_to_file(7u, 17, v20, v21, "upnp read error", -1);
      close_fd(sock);
      return -4;
    }
    // [4] If a reply is received, break out of inner loop for parsing of response
    if ( rx_n )
      break;
    if ( !g_ST_LIST[urn_idx] )
      goto LABEL_30;
  }
  Location = 0;
  Location_len = 0;
  ST_buff = 0;
  ST_buff_len = 0;
  upnp_log("msearch", m_search_payload, rx_n, 2048);

  // [5] Parse the Location and ST fields out of any reply
  parse_msearch_reply(m_search_payload, rx_n, &Location, &Location_len, &ST_buff,
&ST_buff_len);
}
while ( !ST_buff || !Location );
// [6] Construct a log message using the attacker-controlled ST and Location values
snprintf(
  log_msg,
  0xFFu,
  "M-SEARCH Reply: ST: '%.*s' Location: '%.*s'",
  (int)ST_buff_len,
  ST_buff,
  Location_len,
  Location);

// [7] Pass the log message as the `format` parameter to the `log` function
log(6u, 0x11u, log_msg);
```

The function enters a series of loops, the first of which iterates over a list of STs: * urn:schemas-upnp-org:device:InternetGatewayDevice:1 * urn:schemas-upnp-org:service:WANIPConnection:1 * urn:schemas-upnp-org:service:WANPPPConnection:1 * upnp:rootdevice

At `[1]`, the function crafts an M-SEARCH payload using the currently selected ST. At `[2]`, the crafted payload will be broadcast on all possible interfaces. At `[3]`, the system listens for replies to the broadcasted payloads. If a reply is received, then at `[4]` the function will break out of the inner TX/RX loops. At `[5]`, the reply is passed into a parsing function responsible for extracting the (case-insensitive) `Location` and `ST` fields of the reply. At `[6]`, those attacker-supplied parameters are used to craft a log message. Finally, at `[7]`, the injected format string is passed to the vulnerable call to `log`

An attacker capable of listening for and replying to M-SEARCH packets can send a response with a maliciously-formatted `Location` or `ST` field, and that field will be used to craft a format string. Replying with `%x.%x.%x.%x.%x.%x.%x` as both the `ST` and `Location` field will result in the following log message being

generated:

```
[INFO][UPNP]M-SEARCH Reply: ST:
'7173e203.7173e3dc.14.7173e3c2.76f3af18.10000.7133b7a0' Location:
'5.7173e074.0.74acd540.7173e3c2.14.7173e3dc'
```

CVE-2022-35879 - DoUpdateUPnPbyService - `controlURL`

This vulnerable instance of the `log` function occurs within the handler logic associated with `DoUpdateUPnPbyService` actions. The vulnerable call to `log` occurs at offset `0x182038` in the `hpgw` binary included in firmware version 6.9Z. Similar to the previously-described UPnP vulnerability, the `DoUpdateUPnPbyService` action can be generated by transmitting a `setIPCam` XCMD with an `action` tag containing the value `autoportfwd`.

The `DoUpdateUPnPbyService` action operates on data received during the prior `DoEnumUPnPService` action execution. We discussed earlier what occurs when the device receives a reply to its M-SEARCH broadcast, but only in the context of exploiting the previous call to `log`. If instead we trace the execution that occurs when a response is not malicious, it will issue an HTTP request to the `Location` value indicated in the reply. After receiving a response to the HTTP request, it will parse the response to extract various fields and store them in a linked list, which we have referred to as `g_UPNP_LIST`.

Of particular interest is the `/root/device/serviceList/service/controlURL` tag. This tag is extracted and stored into the `g_UPNP_LIST` which will later be used in the following code.

```
...
upnp = g_UPNP_LIST;
advance_list(&upnp, offset);
snprintf(log_msg, 0xFFu, "upnp %u gw:'%s','%s','%s'", offset, &upnp->urn, &upnp-
>controlURL, &upnp->fullURL);
log(6u, 0x11u, log_msg);
...
```

The `controlURL` field is attacker-controlled, and what we've referred to as `fullURL` is crafted using the `controlURL` field, so any value supplied will be duplicated in the final result. An attacker capable of listening for and replying to M-SEARCH packets can send a response refrencing a malicious HTTP server that will respond to further UPnP HTTP queries with malicious data. Replying to the HTTP query with `%x.%x.%x.%x.%x.%x.%x` as the `/root/device/serviceList/service/controlURL` value will result in the following log message being written to the UART serial console:

```
[INFO][UPNP]upnp 0 gw:'urn:schemas-upnp-
org:service:WANCommonInterfaceConfig:1','7173d958.74ad3cb4.74ad3d34.74ad3db4.517c8e.
7173e1b8.45.45.1.45.74ad3ca8','http://10.1.1.207:8000/45.76b9cc78.f.7173dd30.262e72.
7173e050.7173e04c.76b6f9bc.10.0.0'
```

CVE-2022-35880 - DoUpdateUPnPbyService - `NewInternalClient`

This vulnerable instance of the `log` function occurs within the handler logic associated with `DoUpdateUPnPbyService` actions. The vulnerable call to `log` occurs at offset `0x182420` in the `hpgw` binary included in firmware version 6.9Z. This vulnerability occurs further in the UPnP port forward negotiation process that was partially described in the previous vulnerability. If instead we supply an appropriately formatted `controlURL`, the iota will continue the process of requesting a port forward.

This process includes requesting the deletion of any previous port forwarding configuration (`#DeletePortMapping`), followed by adding the newly requested port forwarding configuration (`#AddPortMapping`) and finally verifying the port mapping occured correctly (`#GetSpecificPortMappingEntry`). This vulnerability occurs within the final phase, the verification phase.

```
...
log(6u, 0x11u, "upnp verify");
v21 = strtable_get("LOG_MSG_LOG", 11);
v22 = strtable_get("LOG_MSG_UPNP", 12);
log_to_file(7u, 17, v21, v22, "upnp verify", -1);

// [1] Craft the GetSpecificPortMappingEntry SOAP request
snprintf(SOAPAction, 0x7Fu, "%s#GetSpecificPortMappingEntry", p_urn);
snprintf(
  SOAPBody,
  0x1FFu,
  "<u:GetSpecificPortMappingEntry xmlns:u=\"%s\"><NewRemoteHost></NewRemoteHost>
<NewExternalPort>%s</NewExternalPor"
  "t><NewProtocol>%s</NewProtocol></u:GetSpecificPortMappingEntry>",
  p_urn,
  new_external_port,
  new_protocol);

// [2] Send the SOAP request and receive the response
n = upnp_send_soap_command(p_ctl_url, SOAPAction, SOAPBody, SOAPResp);
v24 = strlen(SOAPResp);
upnp_log("verifyresp", SOAPResp, v24, 4096);
if ( n < 0 )
{
  log(3u, 0x11u, "upnp verify fail");
  v32 = strtable_get("LOG_MSG_FAIL", 12);
  v33 = strtable_get("LOG_MSG_UPNP", 12);
  v34 = v32;
  mismatch = -101;
  log_to_file(3u, 17, v34, v33, "upnp verify fail", -1);
}
else
{
  log(6u, 0x11u, "upnp verify ok");
  v25 = strtable_get("LOG_MSG_LOG", 11);
  v26 = strtable_get("LOG_MSG_UPNP", 12);
  log_to_file(7u, 17, v25, v26, "upnp verify ok", -1);

  // [3] Parse the `NewInternalClient` tag value out of the SOAP response
  upnp_parse(SOAPResp, "NewInternalClient", NewInternalClient, 31);

  // [4] Check whether the NewInternalClient value matches the expected value
  mismatch = strcmp(NewInternalClient, host);
  if ( mismatch )
  {
    // [5] If it does not match, craft a log message using the attacker-supplied
`NewInternalClient` value
    snprintf(log_msg, 0xFFu, "upnp verify mismatch:'%s'", NewInternalClient);

    // [6] Finally, pass the crafted message as the `format` parameter of the `log`
function
    log(3u, 0x11u, log_msg);
    ...
```

At [1], the system crafts a GetSpecificPortMappingEntry message. At [2], it transmits the message and waits for a response. If a response is received, then at [3] the system attempts to extract the NewInternalClient tag. At [4], the value is compared to the expected value, and we are specifically interested when they do not match. At [5], the non-matching NewInternalClient value is injected into a log message. Finally, at [6] the crafted log message is passed as the format parameter to the log function.

An attacker capable of listening for and replying to M-SEARCH packets can send a response refrencing a malicious HTTP server that will respond to further UPnP HTTP queries with malicious data. Replying to the various UPnP configuration requests with valid data until the server receives the GetSpecificPortMappingEntry request, then supplying a response where the NewInternalClient tag contains the value %x.%x.%x.%x.%x.%x.%x, will result in the following log message being written to the UART serial console:

```
[ERR!][UPNP]upnp verify
mismatch:'7173d8e6.2631a3.ffffffff.5763da.7173ec00.5763dc.45.45'
```

CVE-2022-35881 - DoUpdateUPnPbyService - errorCode / errorDescription

This vulnerable instance of the log function occurs within the handler logic associated with DoUpdateUPnPbyService actions. The vulnerable call to log occurs at offset 0x1824F8 in the hpgw binary included in firmware version 6.9Z. This vulnerability occurs when an error code is returned during the #AddPortMapping step of the UPnP port forward negotiation process that was partially described in the previous vulnerabilities. If the server responds with errorDescription and errorCode tags, the function will craft an error message detailing the failure to add the port mapping.

```
// [1] Craft the #AddPortMapping request
snprintf(SOAPAction, 0x7Fu, "%s#AddPortMapping", p_urn);
snprintf(
  SOAPBody,
  0x1FFu,
  "<u:AddPortMapping xmlns:u=\"%s\"><NewRemoteHost></NewRemoteHost>
<NewExternalPort>%s</NewExternalPort><NewProtocol>%s"
  "</NewProtocol><NewInternalPort>%s</NewInternalPort>
<NewInternalClient>%s</NewInternalClient><NewEnabled>1</NewEnable"
  "d><NewPortMappingDescription>1818upnp:%s</NewPortMappingDescription>
<NewLeaseDuration>0</NewLeaseDuration></u:AddPortMapping>",
  p_urn,
  new_external_port,
  new_protocol,
  internal_port,
  host,
  new_external_port);
// [2] Transmit the SOAP Command and expect a response
n = upnp_send_soap_command(p_ctl_url, SOAPAction, SOAPBody, SOAPResp);
v19 = strlen(SOAPBody);
upnp_log("addcmd", SOAPBody, v19, 512);
v20 = strlen(SOAPResp);
upnp_log("addresp", SOAPResp, v20, 4096);

if ( n > 0 )
{
  log(6u, 0x11u, "upnp add ok");
  arg = strtable_get("LOG_MSG_LOG", 11);
  v21 = strtable_get("LOG_MSG_UPNP", 12);
  log_to_file(7u, 17, arg, v21, "upnp add ok", -1);
  // [3] Attempt to extract an `errorCode` tag
  upnp_parse(SOAPResp, "errorCode", error_code, 31);
  if ( errorCode[0] )
  {
    // [4] Attempt to extract an `errorDescription` tag
    upnp_parse(SOAPResp, "errorDescription", err_desc, 63);

    // [5] Inject the `errorCode` and `errorDescription` values in to a log message
    snprintf_nullterm(
      log_msg,
      0xFFu,
      "upnp add fail: '%s','%s','%s', code='%s', msg='%s'",
      new_external_port,
      internal_port,
      host,
      error_code,
      err_desc);

    // [6] Call `log` passing the injected `log_msg` buffer as the `format`
parameter
    log(3u, 0x11u, log_msg);
    ...
  }
  ...
}
```

At [1], the system crafts a AddPortMapping message. At [2], it transmits the message and attempts to wait for a response. If a response is received, then at [3] the system attempts to extract the errorCode tag. If the response contained an errorCode tag, then at [4] it attempts to extract the errorDescription tag. At [5], the errorCode and errorDescription values are injected into a log message buffer. Finally, at [6] the crafted log message is passed as the format parameter to the log function.

An attacker capable of listening for and replying to M-SEARCH packets can send a response referencing a malicious HTTP server that will respond to further UPnP HTTP queries with malicious data. Replying to the various UPnP configuration requests with valid data until the server receives the AddPortMapping request, then supplying a response where the <errorCode> and <errorDescription> tags contain %x.%x.%x.%x.%x.%x.%x, will generate the following log message:

```
[ERR!][UPNP]upnp add fail: '0','0','10.1.1.201',
code='716fd903.7471a4d2.716fec00.716fd7d8.716fd7f8.7471a4d4.45.45', msg=''
```

Due to an irregularity in the parsing of the response that we were unable to diagnose, we could not coerce the device to inject the errorDescription tag, but the errorCode tag was confirmed during testing. Were that irregularity to be resolved, we feel confident the errorDescription tag would be equally vulnerable.

TIMELINE

2022-07-20 - Vendor Disclosure
2022-10-20 - Public Release

CREDIT

Discovered by Matt Wiseman of Cisco Talos.
