Talos Vulnerability Report

TALOS-2021-1420

Reolink RLC-410W cgiserver.cgi Login authentication bypass vulnerability

JANUARY 26, 2022

CVE NUMBER

CVE-2021-40404

Summary

An authentication bypass vulnerability exists in the cgiserver.cgi Login functionality of reolink RLC-410W v3.0.0.136_20121102. A specially-crafted HTTP request can lead to authentication bypass. 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

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

CWE

CWE-284 - Improper Access Control

Details

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

The RLC-410W enforces that the only API usable as a not-logged-in user is the `Login` API. It detects if the provided command is `Login` by checking the URL's `cmd` parameter. The `cgiserver.cgi` binary accepts a list of commands provided as an array of JSON objects in the body as the actual commands; this can lead to a bypass of the "please login first" check.

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. The 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>
        }
    },
]
```

The `parse_incoming_and_check_command` function parses the incoming request:

```
 int parse_incoming_and_check_command(cgi_request *req)

{
    [...]

    Json::Reader::Reader(json_reader);
    Json::Value::Value(&json_value,0);
    iVar1 = parse_request(req);
    if (iVar1 == 0) {
        if (((int)req->CONTENT_LENGTH < 1) || (req->is_commands_in_body == 0)) {
                        /* no body is present */
        [...]
        }
        std::basic_string<char,std::char_traits<char>,std::allocator<char>>::basic_string
                ((char *)post_data_as_basic_string,(allocator *)req->body_data);
        pbVar4 = post_data_as_basic_string;
        post_data_is_valid_json =
            Json::Reader::parse(json_reader,post_data_as_basic_string,&json_value,true);
        std::basic_string<char,std::char_traits<char>,std::allocator<char>>::~basic_string
                ((basic_string<char,std::char_traits<char>,std::allocator<char>> *)
                post_data_as_basic_string);
        if (post_data_is_valid_json == 0) {
        AVar3 = param error;
        }
        else {
        post_data_is_valid_json = Json::Value::isArray(&json_value,pbVar4);
        json_idx = 0;
        if (post_data_is_valid_json != 0) {
            for (; total_number_of_elements = Json::Value::size(&json_value),
                json_idx < total_number_of_elements; json_idx = json_idx + 1) {
            [... parse a JSON command object and insert it into the command list ...]                  [1]
            }
            goto LAB_0043ccbc;
        }
        AVar3 = protocol;
        }
        req->req_status = AVar3;
    }
    iVar1 = -1;
    LAB_0043ccbc:
    Json::Value::~Value(&json_value);
    Json::Reader::~Reader(json_reader);
    return iVar1;
}
```

At `[1]`, one at a time, the JSON commands are parsed and inserted into a command list. Then, if no username parameter is provided in the URL, the `associate_session_to_request` function is executed:

```
undefined4 associate_session_to_request(c_cgiserver_obj *cgi,cgi_request *req)

{
  dword dVar1;
  API_status_code AVar2;
  cgi_session *session_;
  int iVar3;
  cgi_session *session;
  session_node *session_node_cur;
  dword apStack56 [4];

  session_node_cur = (cgi->session_node).session_node_start;
  while( true ) {
    if (session_node_cur == (session_node *)&(cgi->session_node).session_node_end) {
      AVar2 = please login first;
      if (req->req_command_ID == Login) {                                                          [2]
        if (cgi->number_of_active_sessions < cgi->max_number_of_sessions) {
          session_ = (cgi_session *)operator.new(0xe8);
                    /* try { // try from 0043c5b4 to 0043c5bb has its CatchHandler @ 0043c644 */
          c_cgisession::c_cgisession(session_,cgi,cgi->next_session_ID);
          req->session_ID = session_->session_ID;
          iVar3 = c_cgisession::init(session_,req);
          if (-1 < iVar3) {
            cgi->next_session_ID = cgi->next_session_ID + 1;
            apStack56[2] = session_->session_ID;
            apStack56[3] = (dword)session_;
            std::

_Rb_tree<unsigned_int,std::pair<unsigned_int_const,c_cgisession*>,std::_Select1st<std::pair<unsigned_int_const,c_cgisession*>>,std::less<uns
igned_int>,std::allocator<std::pair<unsigned_int_const,c_cgisession*>>>
              ::_M_insert_unique((pair *)apStack56,&cgi->session_node,(dword)(apStack56 + 2));
            c_cgisession::cgi_req_proc(session_,req);                                              [3]
            return 0;
          }
          [...]
}
```

This function aims to bind the incoming request with an existing session or create a new one if the command is `Login`. At `[2]` the `cmd` parameter, provided in the URL, is checked against `Login`. If the command is `Login` a new session is created, and then the session and the request are passed as arguments, at `[3]`, to the `cgi_req_proc` function that then calls the proper requested APIs.

The `cgi_req_proc` function:

```
undefined4 __thiscall c_cgisession::cgi_req_proc(cgi_session *session,cgi_request *req)

{
  [...]
  for (cmd_node_cursor = req->cgi_cmd_node_base->cmd_node_start;
      cmd_node_cursor != (cgi_cmd_node *)&req->cgi_cmd_node_base->cmd_node_end;
      cmd_node_cursor =
           (cgi_cmd_node *)std::_Rb_tree_increment((_Rb_tree_node_base *)cmd_node_cursor)) {
    cgi_cmd = cmd_node_cursor->cgi_cmd;
    if (cgi_cmd->HTTP_status_code == OK) {
      if ((cgi_cmd->API_processing_status & 0xfffffffd) == 0) {
        cgi_cmd->API_processing_status = 1;
        command_struct = cgi_find_cmd_table(cgi_cmd->command_ID);
        if (command_struct != (command_struct *)0x0) {
          [... some log print ...]
          API_function = command_struct->API_function;
          API_result = (*API_function)(session,cgi_cmd);                                    [4]
          if ((API_result != 0) || (cgi_cmd->HTTP_status_code != OK)) {
            if (cgi_cmd->HTTP_status_code == OK) {
              cgi_cmd->HTTP_status_code = protocol;
            }
            [... some log print ...]
            cgi_cmd->API_processing_status = 3;
          }
        }
      }
    }
    [...]
  }
  return 0;
}
```

All the commands parsed at `[1]` are iterated in `cgi_req_proc`, and if the provided command name is valid, at `[4]`, the corresponding API function is executed.

The command list is populated at `[1]` regardless of the URL'`cmd` parameter value. In the specific case of the URL `cmd=Login`, if no username parameter is provided in the URL, the `cgi_req_proc` can be called with an arbitrary list of commands, which can be different from the `Login` one. This will lead to execution, for every command specified in the request body, of the actual API code.

For example, considering the URL `cmd` parameter equals to `Login`, it would be possible to send a body like the following:

```
[
    {
        "cmd": "Upgrade",
        "action": 0,
        "param": {}
    }
]
```

With the above command body, and the URL `cmd=Login`, it would be possible to reach the `Upgrade` API code.

Note that, the `session` struct contains a table with the permitted API. This table is populated after the `Login` API is executed with valid credentials. Because the process explained above exploits not going through the login process, there are no permissions for the session.

For instance, the relevant part of the `Upgrade` API:

```
undefined4 Upgrade(cgi_session *session,cgi_cmd *cmd)

{
    [...]

    if (cmd->parsing_status == NOT_HANDLED) {
        error_code = cgi_check_ability(cmd->command_ID,session,0);                          [5]
        if (error_code != NO error) {                                                       [6]
        [...]
        cmd->HTTP_status_code = error_code;
        cmd->associated_request->perform_reboot = 1;
        return 0xffffffff;
        }
        cmd->parsing_status = PARSE_OK;
    }
[...]
}
```

This code should not be reached for the not-logged-in users, but because of the problem explained above it is possible to reach the `Upgrade` code with an invalid `session`. At `[5]` the permission required is checked against the `sesssion` permissions. Because of the check at `[6]` it is not possible to complete the `Upgrade` API.

This vulnerability in combination with TALOS-2021-1421 leads to the reboot of the camera without authentication. This vulnerability in combination with TALOS-2021-1422 leads to the reboot of the camera without authentication. This vulnerability in combination with TALOS-2021-1425 leads to the execution of several APIs without authentication:

```
{'Login', 'HeartBeat', 'GetMdState', 'GetHddInfo', 'Unknown', 'Playback', 'UpgradePrepare', 'Format', 'SetMdAlarm', 'GetWifiSignal',
'GetAbility', 'GetMdAlarm', 'Logout'}
```

Timeline

2021-12-06 - Vendor Disclosure
2022-01-19 - Vendor Pathed

2022-01-26 - Public Release

CREDIT

Discovered by Francesco Benvenuto of Cisco Talos.

VULNERABILITY REPORTS

PREVIOUS REPORT TALOS-2021-1414

NEXT REPORT TALOS-2021-1422