Talos Vulnerability Report

TALOS-2021-1423

Reolink RLC-410W cgiserver.cgi session creation denial of service vulnerability

JANUARY 26, 2022

CVE NUMBER

CVE-2021-40406

Summary

A denial of service vulnerability exists in the cgiserver.cgi session creation functionality of reolink RLC-410W v3.0.0.136_20121102. A specially-crafted HTTP request can lead to prevent users from logging in. 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-400 - Uncontrolled Resource Consumption

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 maintains an in-memory session list. The `cgiserver.cgi` will keep the session list updated, removing the out of date sessions and the invalid ones.

A specially-crafted HTTP request can pollute the sessions list with a non-removable session, possibly filling the session list up and preventing the users from logging into the camera.

The `cgiserver.cgi` uses the `parse_incoming_and_check_command` function to parse the URL and body data. For many APIs their data can be specified either through the URL or trough the request body. For the `Login` API the body request looks like the following:

```
[
    {
        "cmd":    "Login",
        "action": "0",
        "param":{
          "User":{
              "userName":<username>,
              "password":<password>
          }
        }
    }
]
```

Since it is possible to provide a list of commands, the body request is composed of a JSON array of JSON objects. Each object represents a command with its parameters.

The function `parse_incoming_and_check_command`:

```
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]` a single command is parsed and inserted into a list of commands that will later be iterated on, and each command will be executed. Later on, in the case of the `Login` API, a session is created. If the `Login` data are not provided in the URI, the `associate_session_to_request` will be executed to create the session:

```
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) {
            if (cgi->number_of_active_sessions < cgi->max_number_of_sessions) {
            session_ = (cgi_session *)operator.new(0xe8);
            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);                                          [2]
                return 0;
            }
            session_->session_status = INVALID;
            AVar2 = login failed;
            }
            else {
            AVar2 = max session;
            }
        }
        req->req_status = AVar2;
        return 0xffffffff;
        }
        [...]
    }
    req->session_ID = session->session_ID;
    dVar1 = time((time_t *)0x0);
    session->last_time_checked = dVar1;
    session->lease_time = dVar1 + 0xe10;
    c_cgisession::cgi_req_proc(session,req);
    return 0;
}
```

At `[2]` is called the function that will iterate over the commands provided and execute them. Based on the `Login` API results, the `session_status` field of the sessions will be set, either to `VALID` or `INVALID`.

An unhandled logic case exists in the `Login` API execution flow. Indeed, if the provided URI `cmd` is `Login`, but then an empty array is provided as body, a session is created nevertheless. Because the list of commands is empty, the `Login` request will not be performed.

Since the session initialization does not set the `session_status` and assumes it will be set later on elaborating the provided command, the session will have an unknown state.

Following the `manage_existing_sessions` function, responsible for keeping the session list updated:

```
void manage_existing_sessions(c_cgiserver_obj *cgi)

{
    [...]

    current_time = time((time_t *)0x0);
    session_node_ptr = (cgi->session_node).session_node_start;
    while (psVar3 = (session_node *)&(cgi->session_node).session_node_end, session_node_ptr != psVar3)
    {
        session_data = session_node_ptr->session_data_ptr;
        session_status = session_data->session_status;
        if (session_status == VALID) { [3]
            [... Handle a VALID session ...]
        }
        if (session_status == INVALID) { [4]
            [... Handle an INVALID session ...]
        }
        else {
        get_next_node(&old_session_node,&session_node_ptr);
        }
    }
    [...]
}
```

The `manage_existing_sessions` function does only handle the `VALID` and `INVALID` states respectively at `[3]` and `[4]`. That means that the invalid created session will never be removed. This can fill up the session list, effectively blocking the login to the device.

Timeline

2021-12-06 - Vendor Disclosure
2022-01-19 - Vendor Patched
2022-01-26 - Public Release

CREDIT

Discovered by Francesco Benvenuto of Cisco Talos.
