Hacking Blues

Dionysus Blazakis <dion@trapbit.com> -- October 22, 2012

When each day brings a new collection of stories about the horrors of the cyber age, it's easy to forget why anyone would spend their professional lives in the security industry. Microsoft urged to sue exploit vendors! Huawei is coming -- hide your kids! The project I'm going to describe reminded me of the many tasks in this industry I enjoy. It also functioned as a soft target to run up the score a bit – after spending time fighting DEP, ASLR and process sandboxing, it's nice to pop a shell without the Rube Goldberg quality of a modern exploit every now and then (but that's fun too).

The Start

It started with a friend mentioning he was gifted a Blu-ray player equipped with ethernet, WiFi, and a JVM (which I later learned was part of the Blu-ray standard.) The player is a Magnavox MBP5120F/F7 – at the time of writing, refurbished devices are cheap to purchase and easy to find. Originally, I just wanted to examine the firmware, but the goal mutated into gaining code execution to pop-up a picture or change the splash screen – like Xzibit, I wanted to pimp my Bluray player. With a little digging, it was easy to find a copy of the latest firmware upgrade and a document describing the update process. The firmware upgrade was burned to as disc and placed in the player for upgrade. In a moment of innocence, we took a look at the firmware upgrade binary just in case it was unencrypted:

```
00000000  02 6d a0 d1 69 6e 64 65  7a b8 d8 b5 62 6c 65 00  |.m..indez...ble.|
00000010  7a 3c 49 5e 00 00 00 00  7a 3c 49 5e 00 00 00 00  |z<I^....z<I^....|
00000020  13 46 2c 3a 08 00 00 00  a6 06 83 f1 78 5f 74 61  |.F,:........x_ta|
00000030  7a 3c 49 5e 00 00 00 00  7a 3c 49 5e 00 00 00 00  |z<I^....z<I^....|
00000040  7a 3c 49 5e 00 00 00 00  76 28 4b df 84 01 00 00  |z<I^....v(K.....|
00000050  2f 54 90 d5 6d 61 69 6e  7a 28 59 de 6f 00 00 00  |/T..mainz(Y.o...|
00000060  7a 3c 49 5e 00 00 00 00  7a 3c 49 5e 00 00 00 00  |z<I^....z<I^....|
00000070  d6 28 4b df 84 01 00 00  96 df 84 30 14 b9 00 00  |.(K........0....|
```

Listing 1: First bytes of E5S02UD1015FA1.bin

I expected to find an encrypted file maybe with an unencrypted header. Oddly, this looks as if half the file is "encrypted." Just skimming the file, we saw many copies of the `0x5e493c7a` value. Since this value repeats (i.e. it isn't dependent on location in the stream) and seems to be at locations where the plaintext would be `0x00000000` (this is a hunch that turns out to be correct), my first thought was a simple XOR on every other integer. Unfortunately, this doesn't seem quite right either:

```
00000000  78 51 e9 8f 69 6e 64 65  00 84 91 eb 62 6c 65 00  |xQ..inde....ble.|
00000010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000020  69 7a 65 64 08 00 00 00  dc 3a ca af 78 5f 74 61  |ized.....:..x_ta|
00000030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000040  00 00 00 00 00 00 00 00  0c 14 02 81 84 01 00 00  |................|
00000050  55 68 d9 8b 6d 61 69 6e  00 14 10 80 6f 00 00 00  |Uh..main....o...|
00000060  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000070  ac 14 02 81 84 01 00 00  ec e3 cd 6e 14 b9 00 00  |...........n....|
```

Listing 2: Failed guess at the encoding

My “litmus” test here was the assumption that the value at offset `0x08` should be ‘`x_ta`’ so the string starting at offset `0x04` would be “`index_table`”. I goofed around a bit looking for another simple way of doing the encoding as a constant function and failing. At this point, I started to gather some more information about the mapping between the plaintext and the encoded bytes. I also took a quick frequency count of the encoded values – most were only a few bits away from the magic value I was assuming corresponds to the `0x00000000` plaintext value. Looking a bit more closely at the binary led me to this interesting section:

```
00000540  7a 3c 49 5e 00 00 00 00  91 2e 9a 4c 4f 50 54 49  |z<I^.......LOPTI|
00000550  d0 08 6c 71 4f 4d 45 5f  da 0b 53 97 54 45 52 00  |..lqOME_..S.TER.|
00000560  7a 3c 49 5e 00 00 00 00  0a 84 9a f3 64 69 73 61  |z<I^........disa|
00000570  7a 3c 49 5e 00 00 00 00  7a 3c 49 5e 00 00 00 00  |z<I^....z<I^....|
```

Listing 3: Evidence for the keyed encoding

Here, I noticed the string at offset `0x56C` should probably be “`disabled`” but the following value at offset `0x570` was an encoded zero. This observation led me to hypothesize the two 8-byte aligned integers are reordered or flipped after decoding the first:

```
x, y = read4u(), read4u()
write4u(y)
write4u(decode(x))
```

Listing 4: Attempt 1

Attempt 1 was disproven with the following bit of reasoning:

```
00000300  7a 3c 49 5e 00 00 00 00  91 2e 9a 51 4f 50 54 49  |z<I^.......QOPTI|
00000310  d2 8c 5b d7 53 42 00 00  7a 3c 49 5e 00 00 00 00  |..[.SB..z<I^....|
00000320  7a 3c 49 5e 00 00 00 00  16 06 f4 b2 65 6e 61 62  |z<I^........enab|
00000330  7a 3c 49 5e 00 00 00 00  7a 3c 49 5e 00 00 00 00  |z<I^....z<I^....|
*
00000360  7a 3c 49 5e 00 00 00 00  d3 97 26 56 42 44 50 5f  |z<I^......&VBDP_|
00000370  c2 19 1a b7 52 41 4c 00  7a 3c 49 5e 00 00 00 00  |....RAL.z<I^....|
00000380  7a 3c 49 5e 00 00 00 00  7a 2e 4b d7 33 35 00 00  |z<I^....z.K.35..|
00000390  7a 3c 49 5e 00 00 00 00  7a 3c 49 5e 00 00 00 00  |z<I^....z<I^....|
```

Listing 5: Two different encodings for 0x00000000

Here, by assuming the strings starting at `0x308` and `0x368` are both followed by multiple zero bytes, we have two different encodings of a `0x00000000` value: `0xd75b8cd2` and `0xb71a19c2`. To me, this implied the encoding was dependent on the `y` value also. As a leap of faith, I also assumed a bit more about the “decode” function:

```
def decode(x, y): return x ^ F(y)
x, y = read4u(), read4u()
write4u(y)
write4u(decode(x, y))
```

Listing 6: Attempt 2

Here, I'm using function `F()` to represent some unknown function dependent on a single input. Framing the decode function in this way avoids forcing `F()` to be invertible. In other words, both decode and encode functions will use the same function, `F()`. In an effort to convince myself this was a plausible structure for the algorithm, I validated it using some guessed plaintext:

```
000013d0  27 c7 98 92 74 68 72 65  ed 8d 08 8f 74 74 72 5f  |'...thre....ttr_|
000013e0  98 51 04 1c 00 70 74 68  b4 6a 0d cc 5f 73 65 6c  |.Q...pth.j.._sel|
000013f0  94 b7 1f cd 68 72 65 61  f1 dd 30 ee 74 72 5f 73  |....hrea..0.tr_s|
00001400  55 46 bb c6 68 65 64 70  7b 57 40 dc 79 00 6f 70  |UF..hedp{W@.y.op|
00001410  27 c7 98 92 74 68 72 65  f7 86 15 88 74 74 72 5f  |'...thre....ttr_|
00001420  00 c7 8d d0 63 68 65 64  15 0f 19 bb 6d 00 63 6c  |....ched....m.cl|
```

Listing 7: Part of a symbol table for verifying Attempt 2

```
>>> f_ttr__xor_init = 0x8f088ded # offset 0x13D8
>>> f_ttr__xor_sets = 0x881586f7 # offset 0x1418
>>> hex(f_ttr__xor_init ^ struct.unpack('<I', 'init')[0]) # should be F('ttr_')
'0xfb61e384L'
>>> hex(f_ttr__xor_sets ^ struct.unpack('<I', 'sets')[0]) # should be F('ttr_')
'0xfb61e384L'
```

Listing 8: Python shell showing evidence in support of Attempt 2

Starting at `0x13d0`, the string should read "`thread_attr_init`" (part of "`pthread_attr_init`",) and at `0x1410`, the string should read "`thread_attr_setschedparam`". This plaintext information is an educated guess using known libpthread function names. Using this information, we can construct two different applications of our algorithm that use the same `F(y)` value for XOR-ing with different `x` values. We can XOR out the plaintext to reveal the value of `F(y)`. The matching value for `F("ttr_")` is promising.

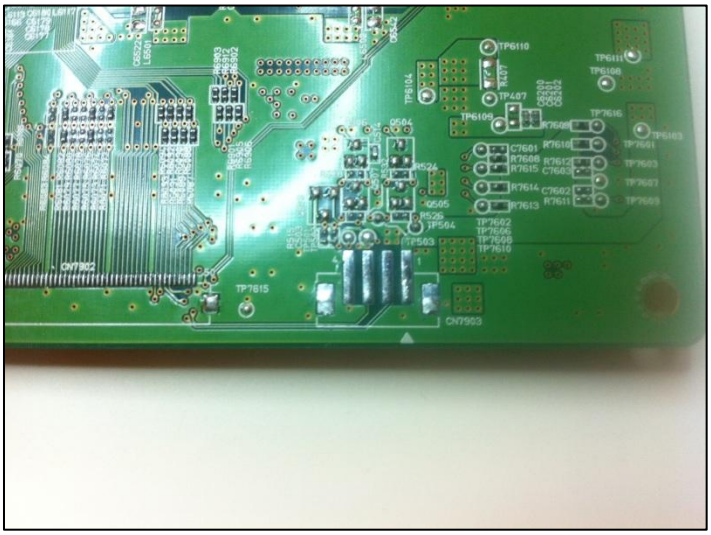
At this point in the project, I don't feel too bad admitting I hit a wall and failed miserably at figuring out how `F()` worked. A friend even found an entire font file in the firmware to reveal a large number of points on `F()`. I tried using a solver to determine a short set of operations (got up to 6) similar to the work documented by Denis Yurichev. I tried to evolve the function using a genetic programming. I started down the path of a differential analysis – looking at what changes occur in the output due to changes in the input – which would have produced results, but I got impatient and yanked the stupid NAND chip from the board. I think (now that I know how the algorithm works) that it would have been possible to deduce the rest of the algorithm using the known plaintext to mount a differential attack but that could be optimistic.

Before I dove into dumping the firmware from Flash, I also took some time to gather evidence via any debugging interfaces such as a serial port logging debug messages or a JTAG interface where I could dump the Flash without any further interfacing. The real goal is still to decrypt and unpack the firmware upgrades as a step towards running our own code on the device.

Debug Console?

Somewhere in the middle of the above head-wall relationship, I also disassembled the player to determine the main processor and to look for any UART or JTAG test points or headers. After opening

up the device, the main board has most of the components on the underside but there are two interesting features on the top of the board. The first is a set of 4 nice large pads right on the edge of the board. Some quick work with the multimeter shows a ground, Vcc and two data pins. Using the Saleae logic analyzer, I verified this theory – CN7903 exposes a serial port for debugging (and maybe manufacturing or field support). Pin 4 is GND, Pin 3 is Tx, Pin 2 is Rx, and Pin 1 is Vcc.



The other interesting feature is a marking on the PCB describing a "COM TOOL." This box marks the location of a 6-pin ribbon connector. Unfortunately, my probing didn't produce anything interesting on these pins by default.

Flipping the board over, we see the main processor is under a large heatsink/shield. With this removed and the thermal goo cleaned up, the part is a Panasonic MN2WS62. This seems to be in a family of feature rich "System LSI" (a term I'd never encountered) chips that package system peripherals, like AV encoder/decoders and security features, into a single package. The only member of this family on the Panasonic website at the time of writing is the MN2WS0260. Predictably, I was unable to find any sort of datasheet for this chip (or the MN2WS62 found in the player) but the marketing material on the webpage lists the processor core as a Panasonic MN103xx core (specifically, the AM34.)

Additionally, we can verify this by checking the `e_machine` field in the ELF header visible in the firmware:

```
000006e0  7a 3c 49 5e 00 00 00 00  cd 42 0e a6 7f 45 4c 46  |z<I^.....B...ELF|
000006f0  7a 3c 49 5e 00 00 00 00  7b 39 48 1f 02 00 59 00  |z<I^....{9H...Y.|
```

Listing 9: Firmware with the ELF header e_machine field in the clear

```
#define EM_MN10300      89      /* Panasonic/MEI MN10300, AM33 */
```

Listing 10: EM_MN10300 definition

There are a handful of unpopulated header pads. For the smaller sets of these, I looked for digital signals, but didn't find a second serial port. I didn't spend much time looking for a JTAG port because I assumed the fuse was blown and there are a huge number of testpoints on this board. Lacking a datasheet, I decided to punt on the JTAG hunt.

At this point, I had taken a few logs of the serial output during the boot sequence and searched the Internet looking for strings. For example, the bootloader appears to be called "SCBoot".

```
[SCBoot] SramBoot ver 1.01 End (08/07/30)
[SCBoot] Miniboot Ver 1.00 Start (08/07/30)
[SCBoot] Bank 1
[SCBoot] FE-FW Loading...(from afc00000)
[SCBoot] FE VERSION IS B35_001_000
[SCBoot] FE reset release.
```

Listing 11: Bootloader serial output

Shortly after this output, the Linux kernel starts. This also continues to hint (this architecture is described on the product page for Panasonic System LSI) at the dual system architecture (FE/BE or frontend/backend). Searching for some of the process names didn't dig up anything interesting for me on the Internet.

I also attempted entering input via the serial port. I got a simple prompt ("`> `") and could get errors about filling up the serial port buffer ("`S:Overflow`") but no amount of guessing could get me anything else.

Again, I reached the limit of what I could do via this blackbox approach. This was also about the time I reached the limit of my skills in decrypting the firmware upgrade binary. It was time to dump the firmware from Flash and hope it wasn't obfuscated in the NAND in some way.

In which every EE points and laughs

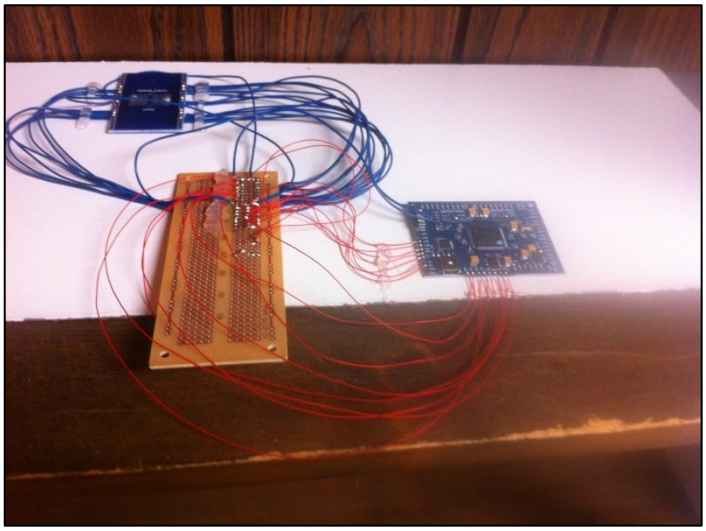
On the underside of the board, not far from the Panasonic chip, there was a TSOP-48 chip with a sticker and some crayon/china marker on it. Cleaning it up revealed a Hynix HY27UF081G NAND chip. This was the only significant storage I found on the board and must be where the main firmware resides. I started researching how to read this chip without having to de-solder it.

I found the ProgSkeet and 360-Clip. This was originally created to reprogram the Flash found in the Xbox 360 DVD player. This seems like the perfect solution. I ordered both the clip and ProgSkeet. When I received the clip, it didn't seat properly on the board (2 or 3 surface mount resistors are in the way). After carefully removing these components, I still couldn't get a good electrical connection using the

clip. In trying to get a good connection I mashed a corner of the leads coming off the Flash chip. This was mostly a failure.

After I repaired the hole in the wall and got back from a quick trip to the clinic to X-ray my hand, I read some blogs and watched a few YouTube videos on using the hot air rework station I have. I was ready. Using an X-Acto knife to apply upward force to one side of the TSOP, I started heating that side. I kept heating it. I heated it more. I slipped with the X-Acto, cut the PCB and lifted a few pads. I did slightly better on the other side, but ended up bending the already fragile leads on the smashed corner pretty bad. Another, uh, learning experience.

Finally, I went the route I should have gone originally and ordered Chip-Quik. If you've never seen this substance, it is alien magic. This alloy blends with solder and lowers the melting point so it will stay molten significantly longer. This makes de-soldering SMT components **much** easier. I also ordered a TSOP-48 breakout board and 2 more MBP5120 players (one for $2 off E-Bay with a broken Bluray tray). This time, the de-soldering effort was beautiful. Even more surprising, I was able to drag solder the chip to the breakout board without too much of a problem. Finally, I connected the breakout and the ProgSkeet. On my first try, I missed adding a pull-up resistor to the Ready/Busy pin, but with the resistor in place WinSkeet was able to successfully dump the chip.



Using WinSkeet, I was able to take a handful of dumps and with a little Python, I took a quorum of the dumps to remove most of the bit errors introduced by (I assume) my messy circuit. We'll see that there were still some persistent bit errors – I believe some of these were due to the Flash and would have been corrected by the ECC found in the OOB bytes. Maybe I'm way off. With a final image complete, the reversing effort can move back to software.

Digging In

A quick strings on the resulting flash image is glorious. Tons of kernel strings and, surprisingly to me, tons of ELF strings (e.g. symbol names and section names). One interesting string was a list of the partitions of the `/dev/fma` device and their sizes from what looks like the Linux kernel options.

```
root=/dev/fma5
 serial
 disable_eeprom
 fma=151552k
  fma1=40928k
  fma2=15872k:b
  fma3=256k
```

```
  fma4=3712k
  fma5=14336k
  fma6=10752k
  fma7=14080k
  fma8=43136k
  fma9=6656k
  fma10=3072k
  fma11=11008k
  fma12=1536k
  fma13=6656k
  fma14=0k
```

Listing 12: Linux boot arguments

A little playing around with a calculator and hex editor led me to see that if we drop the first one (fma1), the rest describe the partitions of the flash dump. With the dump broken into partitions, I dumped the root partition and retrieved `/etc/fstab` to determine the usage and format of the rest of the partitions. I knew the root filesystem was in fma5 from the boot args and that it was a cramfs both from "file" and from the header ("`Compressed ROMFS`" at offset 0x10 is a giveaway.)

```
none            /proc           proc    defaults        0 0
/dev/fma6       /usr            romfs   ro              0 0
/dev/fma7       /usr2           cramfs  ro              0 0
/dev/fma9       /work           jffs2   defaults        0 0
/dev/fma10       /si             jffs2  noauto          0 0
/dev/fma11      /nv             jffs2   noauto          0 0
/dev/fma12      /conf           jffs2   noauto          0 0
tmpfs           /tmp            tmpfs   defaults        0 0
tmpfs           /JARPreloadBuffer               tmpfs   size=4864k      0 0
tmpfs           /FontPreloadBuffer              tmpfs   size=4864k      0 0
tmpfs           /BDPreloadBuffer                tmpfs   size=4864k      0 0
none            /sys            sysfs   defaults        0 0
none            /proc/bus/usb   usbfs   defaults        0 0
```

Listing 13: /etc/fstab

Using this info I was able to dump each of the listed file systems. Due to some lingering bit errors in the dump, the program I used to dump the cramfs partitions would either abort the entire dump or the dump for that file. I patched cramfsck to zero the bytes of any block it failed to decompress and continue with the rest of the file. Still, it ran into one occasion where the error was in the field specifying where the next block was located – here, I still had to punt on the file. Luckily, it wasn't in anything critical to determining how the firmware upgrades are packed.

With the filesystems dumped, I wandered around taking in the general layout. One of the first things I looked at was the /etc/initrc file. This file is usually helpful to gain insight into the architecture of the system. In this case, it wasn't used directly:

```
# This rc file is only for DTV init process (not shell script)

NETWORK         = no
NFSCLIENT       = no
NFSSERVER       = no
```

```
GATEWAY         = no

HOSTNAME        = target-koma
HOSTADDR        = 10.73.181.28
NETMASK         = 255.255.252.0
DOMAINNAME      = tvrl.mei.co.jp
GATEWAYNAME     = 10.73.180.1

FSCK            = no
SWAPPER         = no
SWAPDEV         = /dev/fma2
SASH            = no
COREDUMPSIZE    = 0

INSMOD          = /lib/modules/internal-host-hcd.ko /lib/modules/8712u.ko

USERNAME        = m6030


TARGET          = /usr/target/startrc_rom
```

Listing 14: /etc/initrc

Luckily, it points us to `/usr/target/startrc_rom` which contains the usual shell script one expects to find in `initrc`.

```
#!/usr/bin/sash

more dd_version.txt

#ifconfig eth0 up

/sbin/mount /conf
automount --timeout 0 /tmp/jffs2 file /etc/auto.jffs2
source env_file
/usr/bin/date 081100002010

umask 000

mkdir /tmp/RPC
mkdir /tmp/dhcpc
mkdir /tmp/run
mkdir /work/rsvcron
mkdir /work/ib20

mkdir /nv/mail
mkdir /nv/mail/spool
mkdir /nv/repackmail
mkdir /nv/dialog
mkdir /nv/purchase
mkdir /nv/vudu
cp /usr/target/vudu_idfile /nv/vudu/idfile

cp /usr/target/debug_tbl /tmp/debug_tbl
chmod 0777 /tmp/debug_tbl
```

```
/usr/dtvrec/bin/ucodeDL TS ../../usr/dtvrec/data/ucode

/usr/dtvrec/bin/hdmi_tx &
/usr/dtvrec/bin/mocserver &

/usr/pie/bin/pied &
/usr/dtvrec/bin/sys_trs --exec-counts 1 --read-interval 0 --version --list-path
/usr/target/syssec.lst&
/usr/dtvrec/bin/mwinit -s&

cd /usr/dtvrec/bin

./avdec_main.out &
./dmgr_play.out &
./netmgr_play.out &
./nplay_manager.out &
exec userinterface_play.out
```

Listing 15: /usr/target/startrc_rom

This is great. It gave me a list of binaries to start looking for the firmware upgrade procedure. At this point, I loaded up IDA and dropped in a shared library I thought was interesting, libhttpdl.so.

Aside: Thank you, Hitmen

After dropping it into IDA 6.0, IDA Pro doesn't know the machine setting in the ELF header and doesn't have a processor module for the MN103/AM33. A quick search turns up the MN103 processor module by groepaz of the Hitmen. Writing this sort of module is both a pain and a large amount of work, so I'm very glad they made this release available. I updated this for the 6.0 SDK, added a bunch of instructions which are in the MN103E processors (using the extended registers and a `MOV xx, PC` instruction, among others,) and changed a few small things to make my life easier. I'd like to port this to Python, but I'd also like to sleep, so I probably won't. I will eventually put the updated source up somewhere, hopefully, but it is still missing most of the extended register instructions due to my laziness – I only added instruction I needed to understand certain functions. Unfortunately, IDA 6.2 is the first version with support for mn10300 ELF files but **not** support for the mn10300 processor (kind of a weird discrepancy, maybe the next release of IDA Pro will support the processor too). I've put up with quick and dirty scripting efforts to fix-up imports, but I've had a friend verify that newer IDA Pro versions will correctly fix up the imports using the latest IDA along with my modified processor module. Anyway, thanks again groepaz!

Crypto-ey

I started by searching the filesystem for binaries with "`http:`" and, among a few hits, `libhttpdl.so` sounded like a good place to start sniffing around. I cross compiled binutils for MN10300 to use for information collection prior to throwing everything into IDA. Using objdump, I took a look at the exports:

```
dion@slashem:~/projects/bluray/nand$ ~/local/bin/mn10300-elf-objdump -T
fma6/dtvrec/lib/libhttpdl.so | grep .text
3271a648 l    d  .text  00000000              .text
3271b9cc g    DF .text  0000026e  Base        fdl_sockopen
```

```
3271c1b3 g    DF .text  00000179  Base        fdl_downloadfile
3271ae8d g    DF .text  0000006d  Base        fdl_get_httpfile
3271b3d0 g    DF .text  0000005d  Base        fdl_inetaddr
3271b8e9 g    DF .text  00000026  Base        FDL_Stop
3271bc3a g    DF .text  0000013e  Base        fdl_sockgetc
3271bf19 g    DF .text  000000e0  Base        fdl_sockgets
3271ae01 g    DF .text  0000008c  Base        fdl_get_httpserver
3271b5f1 g    DF .text  000000d7  Base        FDL_GetList
3271af9a g    DF .text  000000ae  Base        fdl_get_connect_port
3271b6c8 g    DF .text  00000221  Base        FDL_GetBin
3271ab75 g    DF .text  00000054  Base        fdl_httpclose
3271bff9 g    DF .text  0000009a  Base        fdl_sockprintf
3271a895 g    DF .text  0000001b  Base        HTTP_DownLoadStop
3271abc9 g    DF .text  00000238  Base        fdl_httpopen
3271b048 g    DF .text  00000087  Base        fdl_reqhttp
3271a927 g    DF .text  000000c4  Base        HTTP_DownLoadVerupList
3271b42d g    DF .text  00000106  Base        fdl_vsprintf
3271bd78 g    DF .text  000001a1  Base        fdl_sockrecv
3271aefa g    DF .text  000000a0  Base        fdl_get_connect_host
3271aaaf g    DF .text  000000c6  Base        HTTP_DownLoadVerupBin_NoneHeader
3271b0cf g    DF .text  000000de  Base        fdl_strncasecmp
3271c629 g    DF .text  0000001c  Base        fdl_httpinterrupt
3271b533 g    DF .text  000000be  Base        fdl_sprintf
3271a8b0 g    DF .text  00000077  Base        HTTP_GetDLStatus
3271b96d g    DF .text  0000005f  Base        fdl_sockclose
3271c32c g    DF .text  000002fd  Base        fdl_httpget
3271b1ad g    DF .text  00000223  Base        fdl_getifconf
3271a9eb g    DF .text  000000c4  Base        HTTP_DownLoadVerupBin
3271c093 g    DF .text  00000120  Base        fdl_httpreq
```

Listing 16: libhttpdl.so exports

Judging by symbols like "`HTTP_DownLoadVerupList`", this certainly looks like a library used during the firmware upgrade process. A second way to verify that this is the library used in the firmware upgrade is through the headers used during the update. By setting the DNS to my workstation and running a minimal DNS server, I redirected the HTTP requests used during a network firmware upgrade to an HTTP server also running on my workstation:

```
GET http://j.go2service.net/funai/BDP/E5S02UD/E5S02UD.csv HTTP/1.0
User-Agent: funai-E5S02UD-1.013
Host: j.go2service.net
Content-Type: text/html; charset=UTF-8
X-fum-id: funai E5S02UD 1.013 0 192.168.1.22 00:e0:a9:09:ac:b7 0
X-fum-update: 01/Jan/1970 00:00:00 getVersionInformation success 0 0
```

Listing 17: HTTP request during a network firmware upgrade

Both the "`getVersionInformaton`" string and the "`X-fum-update`" string are found in the `libhttpdl.so` binary. Next, I searched for binaries importing functions from `libhttpdl.so` and the only hit is `/usr/dtvrec/bin/userinterface_play.out`. Now, the real reversing can begin.

```
.text:0807AFEA fw_parameter_state_dump:
.text:0807AFEA                 movm    [A3], (SP)
.text:0807AFEC                 mov     SP, A3
```

```
.text:0807AFED                  add     0xF0, SP
.text:0807AFF0                  mov     3, D0
.text:0807AFF2                  mov     aFwParameterLis, D1 ! "FW PARAMETER LIST\n"
.text:0807AFF8                  call    log_printf?, [], 0
...
.text:0807B0CD                  mov     (g_firmware_parameter_state), A0
.text:0807B0D3                  mov     (0x24,A0), D0
.text:0807B0D6                  mov     D0, (0xC,SP)
.text:0807B0D8                  mov     3, D0
.text:0807B0DA                  mov     aScramble_modeD, D1 ! "scramble_mode (%d)\n"
.text:0807B0E0                  call    log_printf?, [], 0
```

Listing 18: Offset of scramble_mode parameter

After loading userinterface_play.out into IDA, I started the analysis by examining the strings – despite being stripped of non-export symbols, the assertion and logging strings were left in the production binaries. Using the string, "`FW PARAMETER LIST`", I found a function that dumped a bunch of values from a global structure describing the firmware parameters. Using this global variable as a cross reference and the offset described as the scramble_mode (`0x24` as shown in the above listing), I was able to find the function responsible for setting this field. The function loops from 1 to 3, inclusive, calling a function with a buffer containing the first `0x20` bytes of the firmware and comparing the resulting buffer with "`index_table`". This function is the "scramble" function performing one of three different operations depending on the scramble_mode value passed in. The first mode is the identity – no transformation is used. The second mode appears to use AES-128-128 with a hardcoded key but I haven't found any firmware using this mode to verify. The third mode uses a homebrew algorithm slightly related to DES; this mode is what our firmware is encrypted with.

```
.text:0807BACC                  mov     (g_firmware_parameter_state), A2
.text:0807BAD2                  mov     (0xFFFFFFFC,A3), D0
.text:0807BAD5                  call    fw_deduce_scramble_mode, [], 0
.text:0807BADA                  mov     D0, (0x24,A2)
```

Listing 19: Setting scramble_mode parameter

Reversing the fw_scramble_3 function presents no additional difficulties; it is comprised of very little code, is self-contained, and has not been optimized (temporary stack variables are easy to follow). Below is a diagram of the final "`fw_scramble_3`" algorithm. There are 8 4-bit to 4-bit sboxes that have been taken from the DES standard (they used S4 and S5). The permutation table is P from the DES standard. The code suggests it was designed as a Feistel scheme of multiple rounds but it has been hard-coded to a single round:

```
.text:0808201E                  clr     D0 ! <-- round count - 1, a single round
.text:0808201F                  mov     D0, (0xFFFFFFFC,A3)
.text:08082022
.text:08082022 loc_8082022:                ! CODE XREF: fw_scramble_3_crypt+64j
.text:08082022                  mov     (0xFFFFFFFC,A3), D0
.text:08082025                  cmp     0, D0
.text:08082027                  bge     loc_808202B
.text:08082029                  bra     loc_8082067
.text:0808202B ! ------------------------------------------------------------------
...
```

```
.text:0808205D                  mov     (0xFFFFFFFC,A3), D0
.text:08082060                  add     0xFF, D0
.text:08082062                  mov     D0, (0xFFFFFFFC,A3)
.text:08082065                  bra     loc_8082022
.text:08082067 ! ------------------------------------------------------------------------
```

Listing 20: Restricting the round count to 1

This makes recovery of the key very easy. All operations are invertible. The only wrench in recovering the key via inverting a known plaintext encryption is due to a bug. I noticed while analyzing the encrypted firmware was the lack of a zero bit – all encrypted values had a zero in the least significant bit. The cause of this is an off-by-one bug in their permutation step resulting in the top bit lost and the bottom always zero. This means the top bit of the resulting Feistel function is unknown, but this is easily worked around by solving for both a one and zero for the top bit and trying both keys on the firmware.

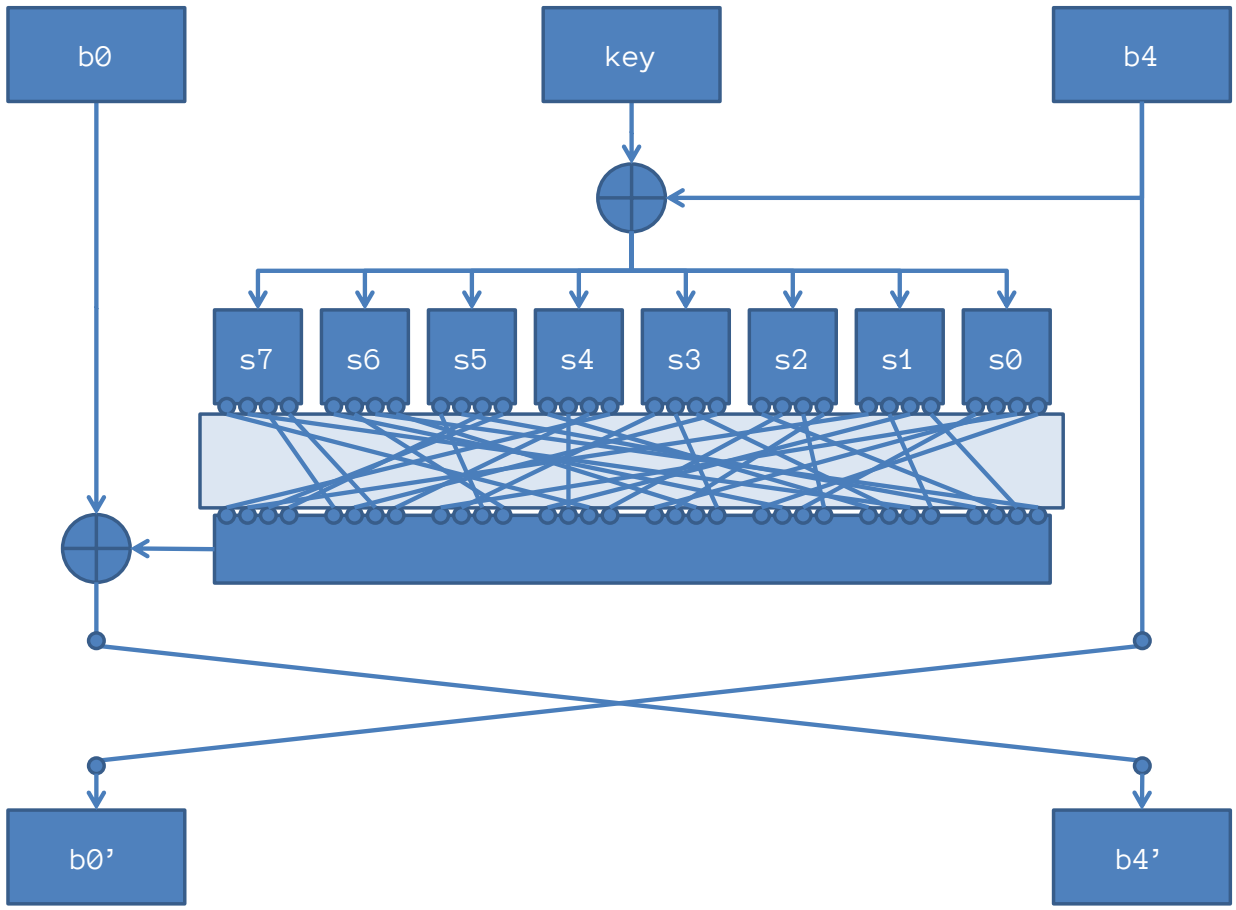


Figure 1: `fw_scramble_3_crypt (sub_8082001)`

I've coded this up in Python -- scramble, unscramble, and key recovery. The code is posted at https://github.com/justdionysus/mbp51xx-tools/tree/master/tools along with rest of the tools developed while doing this project. The key recovery was tested on the MBP5130F/F7 firmware which

uses the same algorithm with a different secret key. Next, with the firmware upgrade successfully decoded, I needed to figure out the format of the upgrade.

This is easy to blackbox reverse. The format follows a simple pattern with counts of entries in the index table where each entry contains a name, an offset, and a length. For verification, it is not difficult to find a few functions responsible to parse this index table just using string (e.g. various entry names like "`main_info`") and debug logging cross references. The `fw_extract` script performs this extraction. Each entry is extracted to a file with the name of the entry in the index table.

The entry we are most interested in is the "`nand_rom_cmpr`" entry containing the updated firmware for the NAND flash. This entry has a format that can, once again, be blackbox reversed. The format consists of a count of chunk and a list of chunks, their sizes, and whether or not they are present in the file. In other words, each entry describes a span in the Flash memory and, if this span has been updated, the updated chunk is present in the file. Otherwise, that span of memory is skipped during the NAND write. This parsing and update process is performed via the `nand_flash_write.out` binary. This extraction along with some intelligence, matching chunks up with partitions, is coded up in the `fw_nand_dump` script.

At this point, I was able to do the same filesystem extraction steps performed after the NAND extraction. For the firmware versions 1.019 and 1.015, the extracted partitions are complete and full filesystems were extracted – this time, with no chance of bit errors to destroy sections of the compressed files. At this point, I could have taken either of two paths to gain code execution: attempt a patched firmware update or find a vulnerability and exploit it remotely. Scared that I would brick another device by botching a firmware update, I took the second route.

LOL, 1988

Above, I mentioned that one of the first steps during a network firmware upgrade was the "getVersionInformation" HTTP request. The response is of the form:

```
E5S02UD 1.019   http://j.go2service.net/funai/BDP/E5S02UD/E5S02UD1019FA1.net35
```

Listing 21: E5S02UD1019FA1.csv

Looking at the cross references to the `HTTP_DownLoadVerupList` function, I saw that this file is downloaded and saved as `/tmp/fw_file_info.dat`. Using the filename as a cross reference, I found the "`fw_parse_file_info`" function (`sub_807B735`) responsible for parsing this info file and saving the entry with the latest version. This function and its sub functions contain multiple bugs. The most useful bug is in the "`csv_getcell`" function (`sub_80855C8`). This function takes a single argument, a pointer to a buffer, and reads from the downloaded file until a tab, carriage return, line feed, or the end of file is reached. It writes directly to the buffer with no further bounds checks. Most of the time, this function is passed a heap buffer allocated in the "`fw_parse_file_info`" function, but in the "`csv_setpos`" function (`sub_8085566`), a stack buffer is used instead. This is a textbook stack buffer overflow vulnerability and the stack is executable. Welcome to 1988. If the process dies via SEGV, the system will dump the register context prior to rebooting the system. Below is the output from a simple POC triggering the bug. A3 works as the frame pointer and is controlled along with the PC.

```
00162219[    SYS 203] [FWUP] Network Version CHECK START
http_server : j.go2service.net
http_file : /funai/BDP/E5S20UD/E5S20UD.csv
connect_host : j.go2service.net
connect_port : 80
00167051[    SYS 203] stat_data.st_size=2209
order is (/usr/bin/mv /tmp/model.csv /tmp/fw_file_info.dat)
SIGSEGV: userinterface_p(203)
PC:  43434343 EPSW: 00010f00
d0:  00000001   d1:  00000000   d2: 00000000   d3: 00000040
a0:  00000000   a1:  7400aa00   a2: 31118598   a3: 42424242
e0:  00000001   e1:  00000001   e2: 311189f8   e3: 00000001
e4:  3ffff79c   e5:  3f3ffc00   e6: 084f0760   e7: 084f0764
lar: 90166bbb   lir: f9d97af9  mdr: 08085510   sp: 3f3ff9b0
Process userinterface_p (pid: 203, stackpage=933de360)
```

Listing 22: Update parsing stack buffer PoC

From this point on, it was easy to write a simple first stage payload that downloads a shared object using the libhttpdl functions then with a dlopen/dlsym and we can easily drop in a second stage. The libc version hasn't changed in 4 different firmware versions that I have access to, so the payload seems robust, using dlopen and dlsym as hardcoded addresses and looking up the libhttpdl.so dynamically. The stack overflow doesn't destroy anything important on the stack beyond the frame pointer and the return address. Those I hardcoded for the continuation of execution, and are tied to the version of the firmware and the model. Both values are reported in the HTTP request and could be used to specialize the exploit. The existing firmware on the still functioning player was 1.013 but both the Flash dump and the firmware updates I could find were newer. This meant I had to write a payload to dump the stack to the debug console to calculate the correct return address (using the second return address and dumping the call instruction prior to that). All things considered this was a very easy bug to exploit reliably.

At this point, I had code execution. What next? I needed a way to compile a shared object for a MN10300 target. Then, I wanted to display something on screen.

Thanks, Redhat!

To get it all working, I was able to compile a mn10300-elf binutils and gcc. This was without a glibc, really, it was supposed to be just enough to compile glibc before compiling a full gcc, but it worked enough to compile a shared object that would dlopen/dlsym any functions it needed from the existing glibc binary from the firmware. Unfortunately, without a glibc/linux toolchain I was unable to build an executable (need the crt0 init/deinit code). I tried, in vain, to build a full toolchain using the glibc ports package. I was unable to find the right mix of versions of gcc/binutils/glibc/glibc-ports to get it all functioning. It seems the MN10300 support is stale. It's also possible I screwed something up along the way but I think the odds are about equal. Luckily, the Redhat team that contributed kernel patches to add MN10300 support to Linux posted a full toolchain (including both binaries and sources.) This is a project I worked on at night, a few days a week – I decided life was too short to fight with toolchain issues and used the binary package (despite having to build a 32-bit Linux VM.) It works pretty well.

Pop

Finally, I have a magic box that downloads an executable and runs it. I reversed the main process a bit more. I found references to osd_api_set_full_background_image_path and decided to try using this function to display a different background image already found on the system. I knew attempting this was a bit of a long shot – I didn't understand how it worked, I was just hoping for magic and that almost never works out. It didn't work. At this point I sat down and started reversing it in earnest.

When I think about reverse engineers, I always imagine two different groups. One group is the malware reversers and crackers. This group specializes in understanding obscure (but **important**) technical inconsistencies in the operating system and processor architecture. They need this information to understand the many different ways a piece of code may be protected and the techniques used against that protection to understand the underlying process. The second group of reverse engineers consists of vulnerability discovery engineers, exploit developers, or interoperability experts. This group is usually faced with a larger target but without any intentional defenses. The product may span multiple processes or even multiple devices. This group is concerned with gaining the overview quickly to reduce their workload and find a small piece to work on. At a high level, these two groups merge into one. That is, the best reverse engineers will be skilled in both de-obfuscated/anti-anti-debugging and large scale reversing. I've spent a lot more time trying to understand large systems. I'm still learning the tricks when analyzing a commercial content protection (unlike my business partner, Dan). In this case, it was time to drop back – instead of powering through and trying to get this one thing to work (drawing an image) I was now working to understand the overall system.

This system is larger than it first seems. The main binary spawns 8 different threads which operate as System V message queue servers. Each of these threads controls access to a logical subsystem and communications between the systems occur over messages and, occasionally, shared memory (for larger communications across additional processes). For example, one thread controls access to the debug console, one thread controls access to the front-end processor, one thread performs processing related to the user interface, and one thread performs the low-level graphics interactions. It turns out, I was calling the low-level OSD function across threads. A bit of poking and it's clear the thread stores some of the global variables required to access the graphics library in thread local storage. So, that's not going to work. I poked around in the message processing functions for the graphics thread and was unable to find a clear path to the function (for setting the background image) I wanted to call.

Next, I found the library responsible for interfacing with the graphics hardware – libdirectfb. Luckily, this is a commonly used open-source library for abstracting graphics hardware (usually) on embedded devices. I decided to create an executable to download a new image (using libcurl – already on the device) and display it using libdirectfb outside of the main process. My library payload would download the executable and the image to display using libcurl and then system() the executable. This plan went off without a hitch. At this point, I had spent over a year goofing around with the device sometimes for an hour a month, sometimes (for the last 2 weeks of the project) for 20 hours a week. I'd considered porting fceux (a popular NES emulator) to the Bluray but I'd rather find a new project to mess around with. See the video at http://youtu.be/1of3izOygHM.

Conclusion

This wasn't a terribly advanced device to attack. The commercial devices I'm paid to look at generally contain much more in the way of defense. I still learned things. I learned about the IDA Pro processor module interface. I learned about the MN10300. I learned about the current state of toolchain building for Linux (if it's modern, 2.6 kernel, things have gotten much better). I re-learned SYS V message queues and Linux shared memory. I learned a bit about the DirectFB library and interface. Finally, I learned some things about architecting a Bluray player.

It was fun but, most of all, I have a payload that ensures my wife will never watch another episode of *Frasier* on our TV again.