.braindump – RE and stuff

September 9, 2011

Reverse engineering an obfuscated firmware image E02 – analysis

Filed under: Uncategorized — Stefan @ 1:15 pm
Tags: arcadyan, arcor, ida, re

This part is again specific to firmware for the EasyBox 803 (and similar models by Arcadyan), but the techniques presented can easily applied to other firmware, even on different architectures.

Now that we've got a file with the actual instructions we need to load it into IDA. As the file (unlike a ELF/PE binary) does not come with all the information we need to properly load it into IDA, we have to gather some information manually.

The first challenge is to find the load address. This is the memory location where the binary would be located at, if it would actually be running on the device.

```
ROM:830007E4
ROM:830007E4 loc_830007E4:                          # CODE XREF: sub_830004D8+2A8↑j
ROM:830007E4                                        # sub_830004D8+2E0↑j ...
ROM:830007E4                 addiu   $a0, (aUnzippingFir_0 - 0x83000000)  # "\nUnzipping firmware at 0x%x ... "
ROM:830007E8                 lui     $a1, 0x8000
ROM:830007EC                 jal     printf
ROM:830007F0                 li      $a1, 0x80002000  # a1
ROM:830007F4                 move    $a0, $s1         # seg_loc
ROM:830007F8                 lui     $a1, 0x8000
ROM:830007FC                 jal     unzip_fw
ROM:83000800                 li      $a1, 0x80002000  # unpack_loc
ROM:83000804                 beqz    $v0, unzip_success
```

I've reverse engineered the bootloader so I have already this information. (see **unpack_loc** or the second argument for **printf**)
Note: even if you have the bootloader you will have to find the bootloader's load address -> chicken|egg. (btw: load address for the bootloader is `0x83000000`, at file offset `0x1000`!)

You can also find the address with trial and error, which basically works as follows:

- disassemble manually or with IDAPython magic
- look at **operands** of **li** (load immediate), **la** (load address) and **lui** (load upper immediate) instructions
- reload binary or relocate segment according to your observations
- your are done if you have xrefs right at the beginning of strings 🙂 (apparently parts of this process can be automatized: Reverse engineering the Airport Express Part 3)

In this case there is another option:
If you load the binary at `0x0` and make a function at `0x0` ('p') you will see a function which is responsible for CPU initialization. When looking further down you will see the following code:

```
nop
li      $a0, 0x808425E8
li      $a1, 0x83467160
subu    $a1, $a0
```

```
loc_E0:
sw      $zero, 0($a0)
sw      $zero, 4($a0)
addiu   $a1, -0x10
sw      $zero, 8($a0)
sw      $zero, 0xC($a0)
bgtz    $a1, loc_E0
addiu   $a0, 0x10
```

As you can see it is zeroing out the memory between `0x808425E8` and `0x83467160`. This indicates that it is setting up

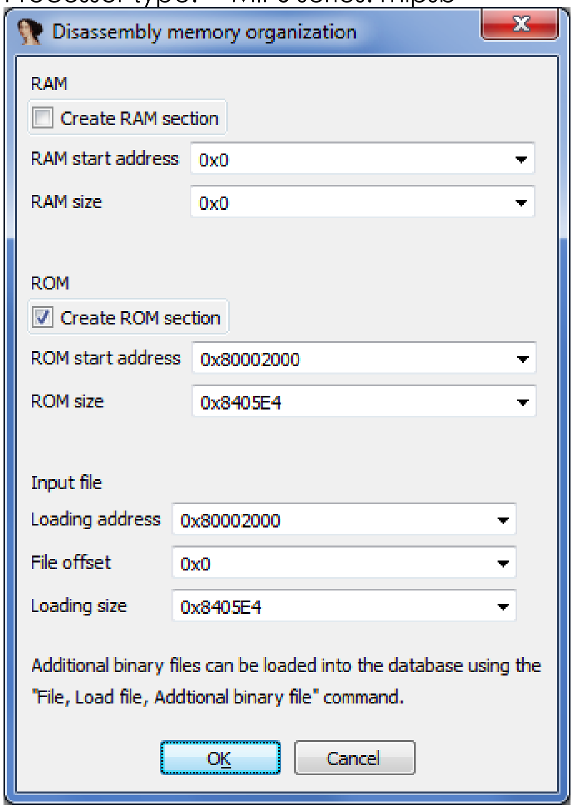
the .bss segment which usually comes right after the data segment.
If you would subtract the size of our input file (`0x008405E3` bytes) from `0x808425E8` you would also get `0x80002000` (`0x80002005` actually, but consider the rest alignment/padding).

Right after this code above we get another important information – the value of the $gp (global) pointer. This (static) pointer is frequently used for addressing data later on.

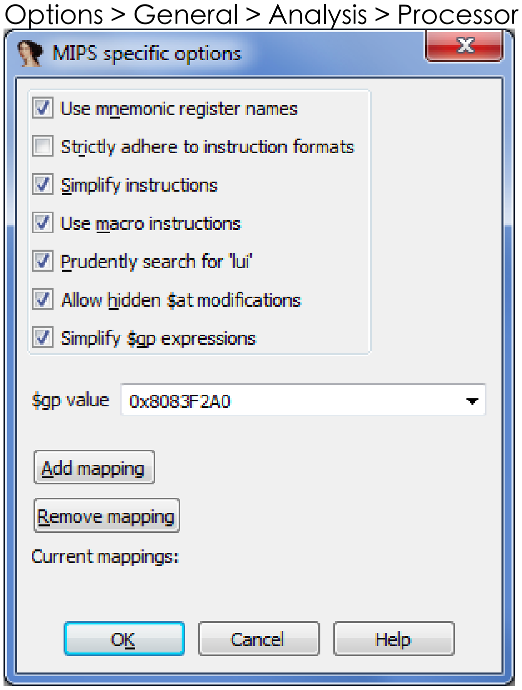
```
ROM:000000F8                 addiu   $a0, 0x10
ROM:000000FC                 li      $gp, 0x8083F2A0
```

Now it' s time to load out file into IDA properly.

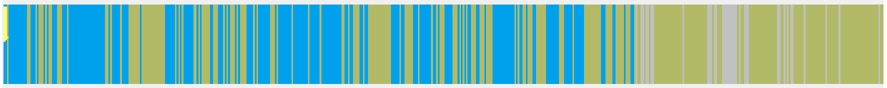
File > Load
Processor type: "MIPS series: mipsb"



Options > General > Analysis > Processor specific analysis options



Now start the auto-analysis by pressing 'p' or 'c' at `0x80002000`.
IDA does a reasonably good job at analyzing the file:



String references match nicely too:

```
ROM:805F0B3C aPingtestCgibuf:.ascii "[PingTest] *** cgiBuf is too small, %d\n"<0>
ROM:805F0B3C                                        # DATA XREF: sub_8000859C+4C↑o
ROM:805F0B64 aSystemOperatio:.ascii "System operation fail.\n"
ROM:805F0B64                                        # DATA XREF: sub_80008708+214↑o
ROM:805F0B64                 .ascii "\r"<0>
ROM:805F0B7D                 .byte    0
ROM:805F0B7E                 .byte    0
ROM:805F0B7F                 .byte    0
ROM:805F0B80 aDestinationHos:.ascii "Destination host unreachable.\n"
ROM:805F0B80                                        # DATA XREF: sub_80008708+228↑o
ROM:805F0B80                 .ascii "\r"<0>
ROM:805F0BA0 aReplyFromSByte:.ascii "Reply from %s: bytes=%u time=%ums\n"
ROM:805F0BA0                                        # DATA XREF: sub_80008708+50↑o
ROM:805F0BA0                 .ascii "\r"<0>
ROM:805F0BC4 aReplyFromSBy_0:.ascii "Reply from %s: bytes=%u time<10ms\n"
ROM:805F0BC4                                        # DATA XREF: sub_80008708:loc_80008774↑o
ROM:805F0BC4                 .ascii "\r"<0>
ROM:805F0BE8 aRequestTimedOu:.ascii "Request timed out.\n"  # DATA XREF: sub_80008708+F0↑o
ROM:805F0BE8                 .ascii "\r"<0>
ROM:805F0BFD                 .byte    0
ROM:805F0BFE                 .byte    0
ROM:805F0BFF                 .byte    0
ROM:805F0C00 aPingStatistics:.ascii "\n"             # DATA XREF: sub_80008708+100↑o
ROM:805F0C00                 .ascii "\rPing statistics for %s:\n"
```

We will help IDA to analyze the binary further by running this IDAPython script (end of CODE is at `0x805F0520`!):

```
def analyze_addiu_sp(curr_addr,end_addr):
        addiu = "27 BD" # 27 BD XX XX addiu  $sp, immediate
        n = 0
        if curr_addr < end_addr:
                print "mipsb addiu function search between: 0x%X and 0x%x" % (curr_addr,end_a

                while curr_addr < end_addr and curr_addr != BADADDR:
                    curr_addr = FindBinary(curr_addr, SEARCH_DOWN, addiu)

                    if GetFunctionAttr(curr_addr,FUNCATTR_START) == BADADDR and curr_addr !=
                        immediate = int(GetManyBytes(curr_addr+2, 2, False).encode('hex'),16)
                        #Jump(curr_addr) # useful for debugging, but has performance impact
                        if immediate & 0x8000: # check if most sigificant bit is set -> $sp -
                                if MakeFunction(curr_addr):
                                    n += 1
                                else:
                                    print 'MakeFunction(0x%x) failed - running 2nd time maybe
                    curr_addr += 1

                print "Created %d new functions\n" % n
                return n
        else:
                print "Invalid end address of CODE segment!"

curr_addr = ScreenEA() & 0xFFFFFFFC # makes sure start address is 4-byte aligned
end_addr = AskAddr(0, "Enter end address of CODE segment.")
analyze_addiu_sp(curr_addr,end_addr)
```

Note: If you get "MakeFunction(`0x80057600`) failed" -errors, wait for IDA to complete its analysis and run a 2nd time.

This script takes advantage of the fact that each function (which uses stack) has an addiu $sp -immediate instruction the beginning of its epilogue. (Of course this may not be the case with other compilers!)

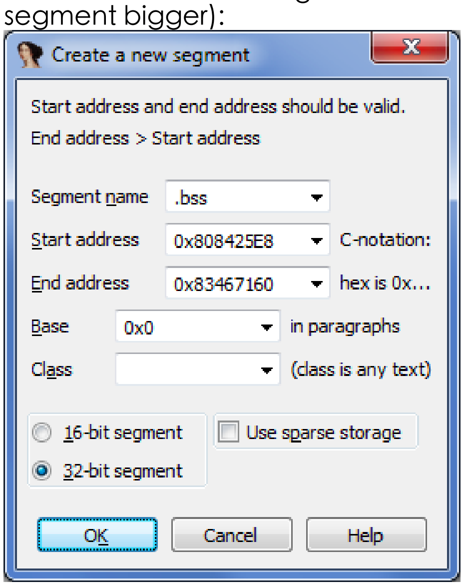
Now we will run a second script to convert the rest (=unexplored stuff) to functions (or at least code).

```
def analyze_unexplored(curr_addr,end_addr):
        if curr_addr < end_addr:
                print "unexplored function and code search between: 0x%X and 0x%x" % (curr_ad

```

```
                while curr_addr < end_addr and curr_addr != BADADDR:
                    curr_addr = FindUnexplored(curr_addr,SEARCH_DOWN)
                    #Jump(curr_addr) # useful for debugging, but has performance impact
                    if curr_addr != BADADDR and curr_addr < end_addr and curr_addr % 4 == 0:
                            MakeFunction(curr_addr)
                print 'done!'
        else:
                print "Invalid end address of CODE segment!"

curr_addr = ScreenEA() & 0xFFFFFFFC # makes sure start address is 4-byte aligned
end_addr = AskAddr(0, "Enter end address of CODE segment.")
analyze_unexplored(curr_addr,end_addr)
```

This script only works properly because there is not data inlined in the code segment. If that wouldn't be the case you would get false positives (eg. code that is actually data).

Let's create a new segment with the information we gathered earlier (alternatively we could just make the ROM segment bigger):



Note: .bss is apparently also used as stack (with $sp pointing to `0x83467160` initially) so the segment does not have to be this big to get all the references to data.
Note²: IDA fails to display all the xrefs to the new segment – Reanalyzing does the trick (Options > General > Analysis > Reanalyze program)

The first function that gets called from the entry function has a reference to the string "\nIn c_entry() function … \n".

```
ROM:800451D0                lui     $a0, 0x8060      # Load Upper Immediate
ROM:800451D4                jal     print            # Jump And Link
ROM:800451D8                la      $a0, aInC_entryFunct  # "\nIn c_entry() function
```

If you [Google](https://www.google.com) for this you will find logs posted by people who attached a serial cable to their Arcadyan devices and read what the device prints during boot.

Quote from http://comments.gmane.org/gmane.comp.embedded.openwrt.devel/4096:

```
In c_entry() function ...
install_exception
Co config = 80008483
[INIT] Interrupt ...
##### _ftext      = 0x80002000
##### _fdata      = 0x805BC0E0
##### __bss_start = 0x80663F44
##### end        = 0x81B8C09C
allocate_memory_after_end> len 687716, ptr 0x81b940a0
##### Backup Data from 0x805BC0E0 to 0x81B9409C~0x81C3BF00 len 687716
##### Backup Data completed
##### Backup Data verified
...
```

It would be nice to have this information for our device too, so let's find the function that prints this and then reconstruct its output:

```
ROM:80185C2C                lui     $a0, 0x8066
ROM:80185C30                li      $a1, 0x80002000
ROM:80185C38                jal     printf
ROM:80185C3C                la      $a0, a_ftext0xLp  # "##### _ftext      = 0x%lp\n"
ROM:80185C40                lui     $a0, 0x8066
ROM:80185C44                li      $a1, 0x807989C0
ROM:80185C4C                jal     printf
ROM:80185C50                la      $a0, a_fdata0xLp  # "##### _fdata      = 0x%lp\n"
ROM:80185C54                lui     $a0, 0x8066
ROM:80185C58                la      $a1, 0x808425E4
ROM:80185C5C                jal     printf
ROM:80185C60                la      $a0, a__bss_start0xL  # "##### __bss_start = 0x%lp\n"
ROM:80185C64                lui     $a0, 0x8066
ROM:80185C68                li      $a1, 0x83467160
ROM:80185C70                jal     printf
ROM:80185C74                la      $a0, aEnd0xLp    # "##### end         = 0x%lp\n"
ROM:80185C78                la      $v1, 0x808425E4
ROM:80185C7C                li      $v0, 0x807989C0
ROM:80185C84                subu    $s1, $v1, $v0
ROM:80185C88                move    $s3, $v0
ROM:80185C8C                li      $s2, 0x8346F160
ROM:80185C94                jal     sub_801853B0
ROM:80185C98                move    $a0, $s2
ROM:80185C9C                jal     sub_801853C4
ROM:80185CA0                move    $a0, $s1
ROM:80185CA4                addu    $s0, $s1, $s2
ROM:80185CA8                la      $a0, aBackupDataFrom  # "##### Backup Data from 0x%lp to 0x%lp~0"...
ROM:80185CB0                move    $a1, $s3
ROM:80185CB4                move    $a2, $s2
ROM:80185CB8                move    $a3, $s0
ROM:80185CBC                jal     printf
ROM:80185CC0                move    $t0, $s1
```

Output (if I’ m correct):

```
##### _ftext      = 0x80002000
##### _fdata      = 0x807989C0
##### __bss_start = 0x8083F2A0
##### __bss_end   = 0x83467160
allocate_memory_after_end> len %d, ptr 0x8346F160
##### Backup Data from 0x807989C0 to 0x8346F160~0x83515A40 len 682208
```

We can now add another segment (backup_data).

When we search for ‘all error operands’ a lot of entries will show up. From the IDA Pro Documentation:

> This commands searches for the ‘error’ operands. Usually, these operands are displayed with a red color.
> Below is the list of probable causes of error operands:
>
> ```
> - reference to an unexisting address
> - illegal offset base
> - unprintable character constant
> - invalid structure or enum reference
> - and so on...
> ```

We are only interested in the “reference to an unexisting address”-type.
The **lui** instructions before the error operands reveal where new segments should be added.

```
ROM:80184FB4                lui     $v0, 0xBE10
ROM:80184FB8                li      $v0, 0xBE100B10
ROM:80184FBC                lw      $v1, 0($v0)
ROM:80184FC0                li      $a0, 0xFFFFFFDF
ROM:80184FC4                and     $v1, $a0
ROM:80184FC8                sw      $v1, 0($v0)
```

This would tell us that we have to add a segment at `0xBE100000` (size at least `0xFFFF`)
Alternatively can search for text “lui “. If the (second) operand<<16 is not part of an existing segment, check if the register (first operand) is used for addressing data, if it is, add a new segment accordingly.

I think missing segments are (as always, I could be wrong) device memory and are not terribly interesting to me. For the sake of completeness I added them.

Name	Start	End	R	W	X	D	L	Align	Base	Type	Class	AD	ds	mips16
ROM	80002000	808425E8	?	?	?	.	.	byte	0000	public	CODE	32	0000	0000
.bss	808425E8	83467160	?	?	?	.	.	byte	0000	public		32	FFFFFFFF	0000
backup_data	83467160	83515A44	?	?	?	.	.	byte	0000	public		32	FFFFFFFF	0000
dev_foo_0	BC200000	BC20FFFF	?	?	?	.	.	byte	0000	public		32	FFFFFFFF	0000
dev_foo_1	BE100000	BE10FFFF	?	?	?	.	.	byte	0000	public		32	FFFFFFFF	0000
dev_foo_2	BE180000	BE18FFFF	?	?	?	.	.	byte	0000	public		32	FFFFFFFF	0000
dev_foo_3	BE190000	BE19FFFF	?	?	?	.	.	byte	0000	public		32	FFFFFFFF	0000
dev_foo_4	BE1B0000	BE1BFFFF	?	?	?	.	.	byte	0000	public		32	FFFFFFFF	0000
dev_foo_5	BF100000	BF10FFFF	?	?	?	.	.	byte	0000	public		32	FFFFFFFF	0000
dev_foo_6	BF200000	BF20FFFF	?	?	?	.	.	byte	0000	public		32	FFFFFFFF	0000
dev_foo_7	BF880000	BF88FFFF	?	?	?	.	.	byte	0000	public		32	FFFFFFFF	0000

Continued in E03: Reverse engineering an obfuscated firmware image – MIPS ASM (“maybe, sometime”)

Note: The IDAPython scripts are based on Craig Heffner’ s work over at /dev/ttyS0 – reccomended:Reverse Engineering VxWorks Firmware: WRT54Gv8
Note²: Make sure to comply with Vodafone’ s terms of use.

Share this:

Twitter Reddit

Like

Be the first to like this.

Comments (7)

7 Comments »

1. […] Continued in E02: Reverse engineering an obfuscated firmware image – analysis […]

   Pingback by Reverse engineering an obfuscated firmware image E01 – unpacking « .braindump – RE and stuff — September 9, 2011 @ 1:16 pm | Reply

2. “We will help IDA to analyze the binary further by running this IDAPython script (end of CODE is at 0x805F0520!):”

   How did you determine the end address of the CODE section? If I look at your ROM segment it’ s actually quite some bigger.

   Comment by justsome — September 25, 2011 @ 9:05 am | Reply

   - Hi justsome,

     The ROM segment contains executable code and data. You can find the boundary by just looking at the assembly: http://i.imgur.com/jBOOJ.png

     Cheers

     Comment by Stefan — September 25, 2011 @ 12:40 pm | Reply

- Thanks! That was actually common sense. Silly me 🙂

  Comment by justsome — September 26, 2011 @ [7:33 pm]

3. Hi,

   I am currently Reversing a Firmware which looks very similar to yours.

   I am having Problems with the $GP Value. Is it alway … as the BSS_Start Value?

   Also, how can I identify where the … mware? My firmware Is segmented in multiple Instruction sets, then a bunch of … ctions.

   Thank You,
   Andrew Borg

   Comment by Andrew Borg — De… | [Reply]

4. I can confirm the same method … Xperia Box V8).
   Now I need to read some disasse…
   It would be nice if I could diff it ag… are.

   Comment by [Rafael Vuijk] — Dec… | [Reply]

   - Hello Rafael,
     were you able to get in the … ge some setting which is not available to change for end user. So I an … figuration file.

     Thank you!
     Tom

     Comment by Tom — May 27, 2015 @ [6:54 am] | [Reply]

[RSS feed for comments on this post.] [TrackBack URI]

Leave a Reply

Enter your comment here...

- ## Recent Posts

  - [Wi-Fi Protected Setup PIN brute force vulnerability]
  - [UCSB iCTF smsgw Memory Corruption Exploit]
  - [A1/Telekom Austria PRG EAV4202N Default WPA Key Algorithm Weakness]
  - [Reverse engineering an obfuscated firmware image E02 – analysis]
  - [Reverse engineering an obfuscated firmware image E01 – unpacking]

- ## Archives

  - [December 2011]
  - [September 2011]

- Search
-
  - [RSS - Posts]
  - [RSS - Comments]
- Twitter: [(@sviehb)]