.braindump – RE and stuff

September 6, 2011

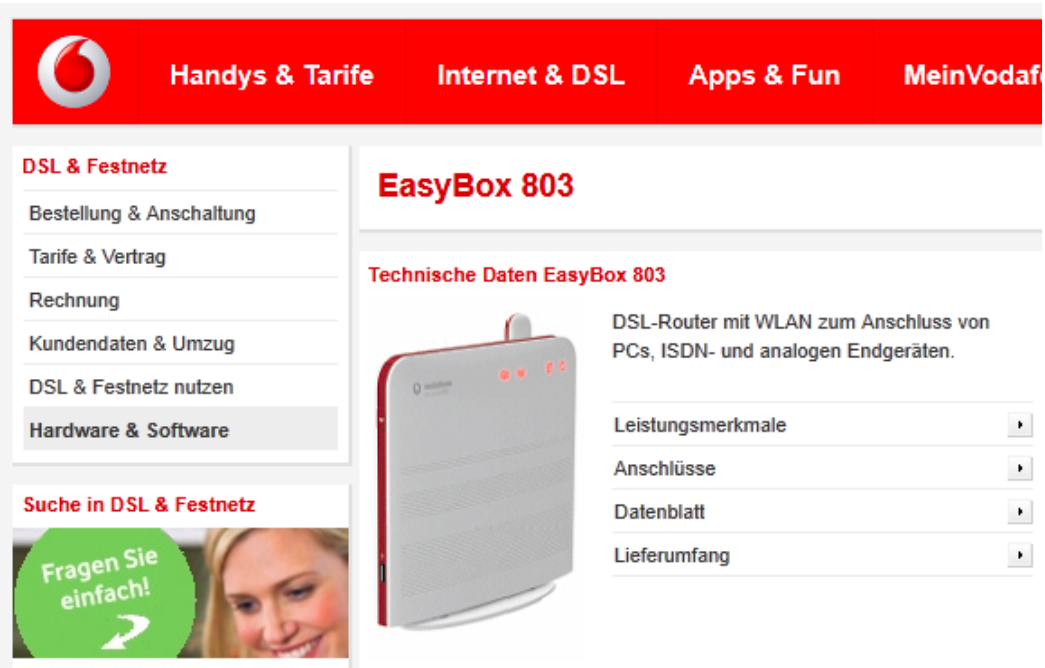
Reverse engineering an obfuscated firmware image E01 – unpacking

Filed under: Uncategorized — Stefan @ 10:38 am
Tags: arcadyan, arcor, fdd, mips, re

When reverse engineering Linux-based firmware images the following methodology usually works pretty well:

1. use Binwalk to identify different parts of a firmware image by their magic signatures
2. use dd to split the firmware image apart
3. unpack parts / mount/extract the filesystem(s)
4. find interesting config files/binaries
5. load ELF binaries into your favorite disassembler
6. start looking at beautiful MIPS/ARM/PPC ASM

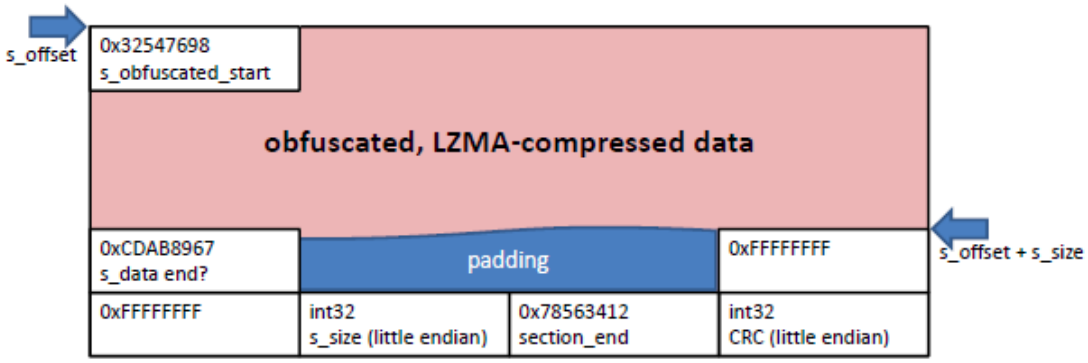
This approach unfortunately didn't work when I looked at firmware images for a broadband router called 'EasyBox 803' distributed by Vodafone Germany (formerly Arcor). Apart from two LZMA-packed segments containing information irrelevant for my research didn't find anything useful in the firmware image at first.
As I had to confirm a major vulnerability (default WPA keys based on ESSID/BSSID [1] [2]) I didn't give up at this point. But let's start right at the beginning …



I obtained a firmware update file for the EasyBox 803 from Vodafone's support page. A Google search reveals the following:

- the device is manufactured by Astoria Networks, which is the German subsidiary of the Taiwanese company Arcadyan
- there are tools available for unpacking Arcadyan firmware (SP700EX, arcadyan_dec)
- Arcadyan uses obfuscation (xor, swapping bits/bytes/blocks) to thwart analysis of their firmware files
- Arcadyan devices don't run Linux, instead they have their own proprietary OS
- MIPS big endian is their preferred architecture

I tried to unpack the firmware file with the tools I found, but although they can deobfuscate the firmware of other Arcadyan devices, they could not do the same for mine. Nevertheless the tools helped me in understanding the layout of my firmware image. It basically consists of several sections which are concatenated. From a high-level view a section looks like this:

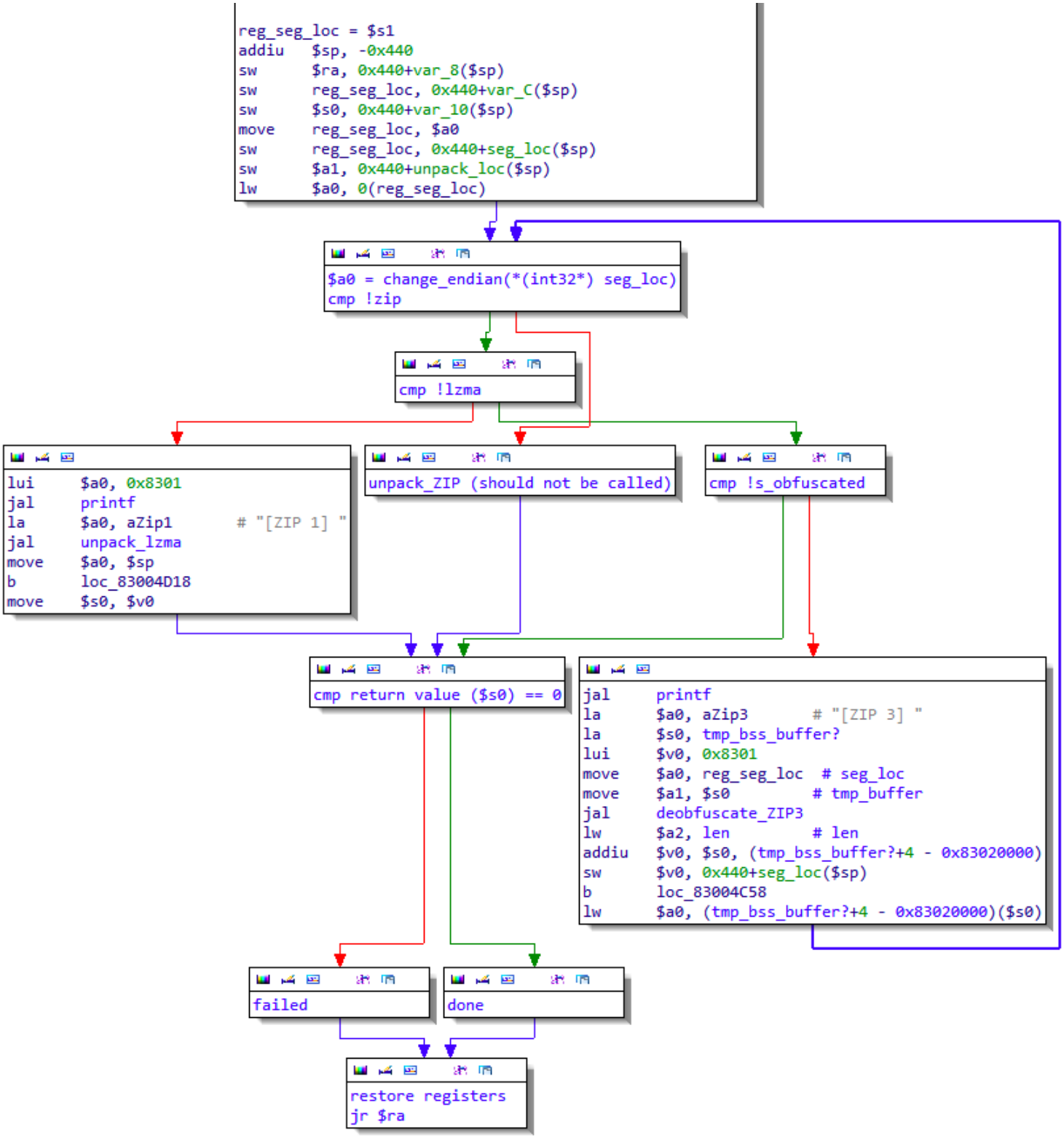



```
(relevant words marked with '||')
beginning of section:
00000000h:|32 54 76 98|11 AF 99 D3 AC FF EA 6C 43 62 39 C8 ; 2Tv˜.¯™Ó¬ÿêlCb9È
...
end of data:
001e83a0h: 0C D8 A3 4A|CD AB 89 67|EE 50 66 2C 53 00 15 93 ; .Ø£JÍ«‰gîPf,S..“
...
end of section:
001e83e0h: 0A 01 EF 8A 73 58 DE 85 00 00 00 00|FF FF FF FF|; ..ïŠsXÞ….....ÿÿÿÿ
001e83f0h:|FF FF FF FF|A4 83 1E 00|78 56 34 12|5E E2 53 5F|; ÿÿÿÿ¤ƒ..xV4.^âS_
beginning of next section:
001e8400h:|32 54 76 98|82 FF 4D 9D CF 6A 95 5E B0 5C 96 7F ; 2Tv˜,ÿMÏj•^°\—
...
```

After I miserably failed at recognizing the obfuscation method just by looking at the hexdump I had to move on. I suspected that the deobfuscation is handled by the bootloader itself, so that was the next thing I wanted to look at. Luckily Vodafone had to update the bootloader for Easybox 802 (predecessor to EasyBox 803) to enable some random functionality and kindly provided a copy, otherwise dumping the flash would have been necessary.

```
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
```

unzip_fw looks like this:



As the deobfuscation and LZMA unpacking is indeed handled by the bootloader, I reversed and reimplemented their fancy deobfuscation routine (deobfuscate_ZIP3):

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  #include<string.h>
4
5  //xor chars in str with xorchar
6  void xor(unsigned char* bytes, int len, char xorchar) {
7      int i;
8
9      for (i = 0; i < len; i++) {
```

```
10          bytes[i] = bytes[i] ^ xorchar;
11      }
12  }
13
14  //swap high and low bits in bytes in str
15  //0x12345678 -> 0x21436578
16  void hilobswap(unsigned char* bytes, int len) {
17      int i;
18
19      for (i = 0; i < len; i++) {
20          bytes[i] = (bytes[i] << 4) + (bytes[i] >> 4);
21      }
22  }
23
24  //swap byte[i] with byte[i+1]
25  //0x12345678 -> 0x34127856
26  void wswap(unsigned char* bytes, int len) {
27      int i;
28      unsigned char tmp;
29
30      for (i = 0; i < len; i += 2) {
31          tmp = bytes[i];
32          bytes[i] = bytes[i + 1];
33          bytes[i + 1] = tmp;
34      }
35  }
36
37  int main(int argc, char *argv[]) {
38      unsigned char* buffer;
39      unsigned char* tmpbuffer[0x400];
40      size_t insize;
41      FILE *infile, *outfile;
42
43      if (argc != 3) {
44          printf("usage: easybox_deobfuscate infile outfile.bin.lzma\n");
45          return -1;
46      }
47
48      //read obfuscated file
49      infile = fopen(argv[1], "rb");
50
51      if (infile == NULL) {
52          fputs("cant open infile", stderr);
53          return -1;
54      }
55
56      fseek(infile, 0, SEEK_END);
57      insize = ftell(infile);
58      rewind(infile);
59
60      buffer = (unsigned char*) malloc(insize);
61      if (buffer == NULL) {
62          fputs("memory error", stderr);
63          exit(2);
64      }
65
66      printf("read \t%i bytes\n", fread(buffer, 1, insize, infile));
67      fclose(infile);
68
69      printf("descrambling file ...\n");
70      //xor HITECH
71      xor(buffer + 0x404, 0x400, 0x48);
72      xor(buffer + 0x804, 0x400, 0x49);
73      xor(buffer + 0x4, 0x400, 0x54);
```

```
    xor(buffer + 0x404, 0x400, 0x45);
    xor(buffer + 0x804, 0x400, 0x43);
    xor(buffer + 0xC04, 0x400, 0x48);

    //swap 0x4 0x404
    memcpy(tmpbuffer, buffer + 0x4, 0x400);
    memcpy(buffer + 0x4, buffer + 0x404, 0x400);
    memcpy(buffer + 0x404, tmpbuffer, 0x400);

    //xor NET
    xor(buffer + 0x4, 0x400, 0x4E);
    xor(buffer + 0x404, 0x400, 0x45);
    xor(buffer + 0x804, 0x400, 0x54);

    //swap 0x4 0x804
    memcpy(tmpbuffer, buffer + 0x4, 0x400);
    memcpy(buffer + 0x4, buffer + 0x804, 0x400);
    memcpy(buffer + 0x804, tmpbuffer, 0x400);

    //xor BRN
    xor(buffer + 0x4, 0x400, 0x42);
    xor(buffer + 0x404, 0x400, 0x52);
    xor(buffer + 0x804, 0x400, 0x4E);

    //fix header #1
    memcpy(tmpbuffer, buffer + 0x4, 0x20);
    memcpy(buffer + 0x4, buffer + 0x68, 0x20);
    memcpy(buffer + 0x68, tmpbuffer, 0x20);

    //fix header #2
    hilobswap(buffer + 0x4, 0x20);
    wswap(buffer + 0x4, 0x20);

    //write deobfuscated file
    outfile = fopen(argv[2], "wb");

    if (outfile == NULL) {
        fputs("cant open outfile", stderr);
        return -1;
    }

    printf("wrote \t%i bytes\n", fwrite(buffer + 4, 1, insize - 4, outfile));
    fclose(outfile);

    printf("all done! - use lzma to unpack");

    return 0;
}
```

You can see that it would have been impossible to understand how the obfuscation works without looking at the actual assembly. Luckily this routine also works for EasyBox 803.

Let’s unpack first segment, which is the biggest one and therefore most likely to contain code.

```
>fdd if=dsl_803_752DPW_FW_30.05.211.bin of=dsl_803_s1_obfuscated count=0x1e83a4
count   : 0x1e83a4      1999780
skip    : 0x0   0
seek    : 0x0   0
1999780+0 records in
1999780+0 records out
1999780 bytes (2.00 MB) copied, 0.009540 s, 199.90 MB/s

>easybox_deobfuscate dsl_803_s1_obfuscated dsl_803_s1.bin.lzma
read    1999780 bytes
descrambling file ...
```

```
wrote   1999776 bytes
all done! - use lzma to unpack

>xz -d dsl_803_s1.bin.lzma

>l dsl_803_s1*
-rw-r--r--+ 1 stefan None 8.3M  6. Sep 11:27 dsl_803_s1.bin
-rw-r--r--+ 1 stefan None 2.0M  6. Sep 11:25 dsl_803_s1_obfuscated

dsl_803_s1.bin:
00000000h: 40 02 60 00 3C 01 00 40 00 41 10 24 40 82 60 00 ; @.`.<..@.A.$@‚`.
00000010h: 40 80 90 00 40 80 98 00 40 1A 60 00 24 1B FF FE ; @€.@€˜.@.`.$.ÿþ
00000020h: 03 5B D0 24 40 9A 60 00 40 80 68 00 40 80 48 00 ; .[Đ$@š`.@€h.@€H.
00000030h: 40 80 58 00 00 00 00 00 04 11 00 01 00 00 00 00 ; @€X.............
00000040h: 03 E0 E0 25 8F E9 00 00 03 89 E0 20 00 00 00 00 ; .àà%é...‰à ....
00000050h: 00 00 00 00 00 00 00 00 24 04 40 00 24 05 00 10 ; ........$.@.$...
00000060h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000070h: 3C 06 80 00 00 C4 38 21 00 E5 38 23 BC C1 00 00 ; <.€..Ä8!.å8#¼Á..
00000080h: 14 C7 FF FE 00 C5 30 21 00 00 00 00 00 00 00 00 ; .Çÿþ.Å0!........
00000090h: 00 00 00 00 00 00 00 00 24 04 40 00 24 05 00 10 ; ........$.@.$...
...
```

Now we can load the file into IDA. This sounds easier than it is, because the unpacked firmware segment is raw code (mipsb) and data without information about segmentation, like you would have when dealing with a PE or ELF binary.

Continued in E02: Reverse engineering an obfuscated firmware image – analysis

Note: Most of this research was conducted several months ago and my findings were probably not in this particular order. – I think it just makes more sense presenting it this way.
Note²: fdd is my silly Python implementation of dd. It takes HEX-offsets and has bs=1 by default.
Note³: Make sure to comply with Vodafone's terms of use.

Share this:

Twitter Reddit

Like

Be the first to like this.

Comments (23)

23 Comments »

1. Nice Work until here, but i like to ask something:
   What do you mean with this: "Note²: fdd is my silly Python implementation of dd. It takes HEX-offsets and has bs=1 by default (and works entirely different than dd)." and especially this part: "(and works entirely different than dd)."

For me it looks like the same as:
dd if=dsl_803_752DPW_FW_30.05.211.bin of=dsl_803_s1_obfuscated bs=1 count=1999780

Thank you
MasterQ

Comment by MasterQ — September 24, 2011 @ 4:47 am | Reply

- Hi MasterQ,
  By "works entirely different than dd" I mean that it does not make a syscall for each byte, instead it loads the whole binary at once.
  dd if=dsl_803_752DPW_FW_30.05.211.bin of=dsl_803_s1_obfuscated bs=1 count=1999780 would have the exactly same effect, but is a bit slower (depends on OS, of course).
  For clarity I removed the part in the brackets.

  Cheers

  Comment by Stefan — September 25, 2011 @ 12:28 pm | Reply

2. Could you help me to replace dsl firmware from annex B to annex A? I tried with sp700ex but it splits EB803 firmware in 7 parts and program itself can join back only 5. I think that dsl firmware for annex a is this http://mirror2.openwrt.org/sources/dsl_danube_firmware_adsl_a-02.04.04.00.00.01.tar.gz
   Thanks!

   Comment by Milan — November 27, 2011 @ 11:29 pm | Reply

3. hello, nice piece of work but i'm wondering something. did you use any other tool than your brain to reverse the MIPS function deobfuscate_ZIP3(). Did you use an automated mips decompiler?
   Regards,

   Comment by int0x13 — February

   - Thank you! The decompilati… 's not very hard once you've spent a few weeks staring a…
     As far as I know there is no w… …able.

     Cheers

     Comment by Stefan — Febr… …y

4. Could you post source for "obfu… …obfuscate data, "fool" original bootloader and upload uboot/o… …d for serial/UART access.
   Thx!

   Comment by angrybb — Februa…

5. Thank you! The decompilation wo…

   Comment by Nick — March 6, 2012 @ 8:16 pm | Reply

6. by any chance …. have you also been looking into the over all checksum algo of the firmware. the checksum is stored in the last section of the firmware (located where the first "0xffffffff" long in your high-level picture is located)

   additional just for the record: your s_data_end marker is just the (s_obfuscated_start) xor 0xffffffff and the padding is just the 0xff xored deobfuscated beginning of the firmware section.

   nevertheless the only part that is still missing is the algo for the checksum…. can you by any chance give me a hand….

   cheers
   /chefchen

   Comment by Chefchen — April 21, 2012 @ 11:50 am | Reply

7. […] to thank Stefan Viehböck for his blog .braindump in which he provides excellent information on how to descramble and unpack firmware images of Arcadyan […]

Pingback by Extract VoIP login data from o2 Box 4421 and o2 Box 6431 | hph's blog — February 13, 2013 @ 6:06 pm | Reply

8. I simply desired to thank you very much yet again.
I am not sure the things I could possibly have gone through without the
smart ideas shared by you concerning my theme. It absolutely was the traumatic problem
in my position, however , seeing a new professional manner you
solved it made me to jump for contentment. Now i am thankful for your help
and then have high hopes you recognize what a great job you're undertaking training many
others thru a blog. Most likely you've never got to know any of us.

   Comment by gratis sexfilme — July 31, 2013 @ 12:26 am | Reply

9. In this diet program system, the dieter is asked a set of 70 concerns, the answers to
which will decide the food habits, attitudes
and exercise habits of the individual.

   Comment by cellulite solutions — August 19, 2013 @ 4:06 am | Reply

10. Thanks everyone for coming out had chatting and sharing. Pass the word, I'd love to have some other authors comment and tell us what they do

    Comment by comoobtenerelcuil.com.ar — August 25, 2013 @ 11:57 pm | Reply

11. good post

    thank for your sharing 🙂

    Comment by nimal maula — November 26, 2013 @ 4:02 am | Reply

12. Hello,

    I'm more or less completely new to disassembling, but I managed to follow this two blog posts and reproduce the results. What I don't understand is how you determined 0x83000000 as ROM start address and 0x1000 as file offset for the bootloader file to be loaded in IDA in the first. Where did you get that from? From manual disassembly of the bootloader file provided by vodafone? If so, how?

    Comment by newbie — December 11, 2013 @ 11:01 am | Reply

13. Hi,

    Can you give some details about how you managed to reverse engineer the Infeneon Boot Loader? Details such as The Rom Load Address, and General Pointer.
    I am trying to replicate your Findings but I cannot get IDA to reference the strings properly, even after using the Python scripts mentioned in episode two.

    Thanks,
    Andrew Borg

    Comment by Andrew Borg — January 3, 2014 @ 11:52 am | Reply

14. […] firmware…' message is particularly interesting; a bit of Googling turned up this post on reversing Arcadyan firmware obfuscation, though it appears to be different from the obfuscation […]

    Pingback by Reversing the WRT120N's Firmware Obfuscation - /dev/ttyS0 — February 2, 2014 @ 4:06 am | Reply

15. […] 'Unzipping firmware…', быстрый гуглёж привел меня к посту про деобфускацию прошивок от Arcadyan, но тут, похоже, […]

    Pingback by [Перевод] Исследуем обфускацию прошивки Linksys WRT120N | Вести3.ру — Информационный журнал — February 19, 2014 @ 5:29 pm | Reply

16. […] 'Unzipping firmware…', быстрый гуглёж привел меня к посту

про деобфускацию прошивок от Arcadyan, но тут, похоже, […]

Pingback by [Перевод] Исследуем обфускацию прошивки Linksys WRT120N » CreativLabs — February 20, 2014 @ 11:53 am | Reply

17. […] 'Unzipping firmware…', быстрый гуглёж привел меня к посту про деобфускацию прошивок от Arcadyan, но тут, похоже, […]

    Pingback by Исследуем обфускацию прошивки Linksys WRT120N « Домик Миа — February 23, 2014 @ 6:10 pm | Reply

18. You post interesting posts here. Your blog deserves much more visitors.
    It can go viral if you give it initial boost,
    i know very useful tool that can help you, simply type in google: svetsern traffic tips

    Comment by Rhys — January 2, 2015 @ 6:49 pm | Reply

19. Now I am going to do my breakfast, when having my breakfast coming again to read other news.

    Comment by Mai — January 11, 2015 @ 2:49 am | Reply

20. hi sir i really need ur help please contact me @ ehab_jean2006@yahoo.com

    thanks in advance

    Comment by ehab — September 23, 2015 @ 8:06 am | Reply

21. thank you for sharing this article

    Comment by iswandi — October 2, 2015 @ 7:45 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)