Analyzing ELF Binaries with Malformed Headers Part 1 - Emulating Tiny Programs

JULIAN - SEPTEMBER 17, 2019

A simple but often effective method for complicating or preventing analysis of an ELF binary by many common tools (`gdb`, `readelf`, `pyelftools`, etc) is mangling, damaging or otherwise manipulating values in the ELF header such that the tool parsing the header does so incorrectly, perhaps even causing the tool to fail or crash. Common techniques include overlapping the ELF header with the program header table and writing non-standard values to ELF header fields that are not needed for composing the process image of the binary in memory. In addition to some programs designed for criminal purposes (e.g. the "mumblehard" family of malware programs), a few code-golf- and proof-of-concept-type programs have been created that employ these techniques. Examples of such programs include Brian Raiter's "teensy" files and @netspooky's "golfclub" programs. In this post, it will be demonstrated how emulation can be used to trace the execution of these types of binaries.

Overview

The following will be discussed:

- how header mangling works as an anti-analysis technique
- how to use the Unicorn Engine to analyze minimalist binaries with malformed headers

Tools:

- Capstone Engine
- Unicorn Engine
- Python3

Malformed ELF Headers

This technique has already been covered in depth elsewhere[1][2][3][4][5][6], so the discussion here will be brief. The main reason mangling the ELF header works to complicate analysis is that even though only a specific subset of the fields in the ELF header are read by the kernel when loading the program into memory, most ELF parsers do not parse the ELF header the way the kernel loader does and thus are prone to malfunction when reading unexpected or garbage values in these extraneous (from the perspective of loading) fields. The most typical examples of this are `gdb`, `objdump` and the rest of the `libbfd`-based binutils tools, which will not even read an object file unless its section information is present and intact.

The specially-crafted minimalist ELF programs - those that push the limits of the least number of bytes a file can consist of and still execute successfully - take advantage of the fact that not all ELF header fields are needed for loading and executing the program and can therefore be used to contain code or other non-standard values; as a case in point, the entry points of these programs often lie *inside* their ELF header. On the one hand, even though complicating analysis is not an explicit goal of their design, these programs serve to highlight the limitations of many common tools designed to work with the ELF format. On the other hand, since these minimalist binaries typically contain such little code, using fully-featured debuggers and other tools of this class for analysis would actually be overkill; one may have a good laugh about how NSA's Ghidra cannot properly load their tiny ELF file, but attempting use such a tool to analyze an extremely minimalist binary is akin to trying to shoot down a fruitfly with a railgun - heavyweight frameworks packaged with disassemblers plus debuggers and/or decompilers are unsuitable and unecessary for analyzing the runtime behavior of executables literally 45 or 62 bytes in size. If there are 10 bytes of code in a program, does it make sense to try to load it into a decompiler? Probably not. A simple script emulating the execution of these programs may be a more appropriate approach.

muppetlabs' tiny-i386: 45 bytes total, 7 bytes of code

This is the "Tiny" program from A Whirlwind Tutorial on Creating Really Teensy ELF Executables for Linux. This program, as well as the rest of the "Teensy" ELF files can be downloaded from the muppetlabs site.

The approach taken here to analyzing this file is as follows:

- attempting analysis with `readelf`, `gdb` and `r2`
- looking at the source code
- emulation

Using Standard Tools to Parse the file

It should be noted at the outset that the binary can be loaded and executed without any problems:

```
$ strace ./tiny-i386
execve("./tiny-i386", ["./tiny-i386"], 0x7ffc2a421f60 /* 52 vars */) = 0
strace: [ Process PID=30049 runs in 32 bit mode. ]
exit(42)                                = ?
+++ exited with 42 +++
```

However when `readelf` is used to try to read the program's ELF header, it fails:

```
$ readelf -h tiny-i386
readelf: Error: tiny-i386: Failed to read file header
```

`gdb` fails to recognize that it is indeed an ELF file:

```
$ gdb -q tiny-i386
GEF for linux ready, type `gef' to start, `gef config' to configure
80 commands loaded for GDB 8.1.0.20180409-git using Python engine 3.6
"home/reversing/tiny-i386": not in executable format: File format not recognized
gef➤  info file
gef➤  run
Starting program:
No executable file specified.
Use the "file" or "exec-file" command.
gef➤
```

In a pleasant surprise, we are able to debug and disassemble the code using r2:

```
$ r2 -d tiny-i386
Process with PID 29855 started...
```

```
= attach 29855 29855
bin.baddr 0x00010000
Using 0x10000
Warning: Cannot initialize program headers
Warning: Cannot initialize section headers
Warning: Cannot initialize strings table
Warning: Cannot initialize dynamic strings
Warning: Cannot initialize dynamic section
Warning: read (init_offset)
asm.bits 32
[0x00010020]> ds
[0x00010020]> ds
[0x00010020]> ds
[0x00010020]> ds
child exited with status 42

==> Process finished

Stepping failed!
Step failed
[0x00010020]> pd 10
            0x00010020      b32a           mov bl, 0x2a                ; ebx
            0x00010022      31c0           xor eax, eax
            0x00010024      40             inc eax
            ;-- eip:
            0x00010025      cd80           int 0x80                     <---------- exe
            0x00010027      003400         add byte [eax + eax], dh
            0x0001002a      2000           and byte [eax], al
            0x0001002c  ~   0100           add dword [eax], eax
            ;-- section_end.ehdr:
            0x0001002d      0000           add byte [eax], al
            0x0001002f      0000           add byte [eax], al
            0x00010031      0000           add byte [eax], al
[0x00010020]>
```

However, when radare2 is used to parse the binary, some of the field values look strange:

```
$ r2 -nn tiny-i386
[0x00000000]> pf.elf_header @ elf_header
     ident : 0x00000000 = .ELF.
      type : 0x00000010 = type (enum elf_type) = 0x2 ; ET_EXEC
   machine : 0x00000012 = machine (enum elf_machine) = 0x3 ; EM_386
   version : 0x00000014 = 0x00010020
     entry : 0x00000018 = 0x00010020
     phoff : 0x0000001c = 0x00000004
     shoff : 0x00000020 = 0xc0312ab3
```

```
     flags : 0x00000024 = 0x0080cd40
    ehsize : 0x00000028 = 0x0034
 phentsize : 0x0000002a = 0x0020
     phnum : 0x0000002c = 0xff01
 shentsize : 0x0000002e = 0xffff
     shnum : 0x00000030 = 0xffff
  shstrndx : 0x00000032 = 0xffff
[0x00000000]>
```

There do seem to be quite a few odd-looking values mixed together with ones that appear similar to what we are accustomed to seeing. What is happening here? Examining the source code will help explain some of this output.

A Look at the Source Code

```
; tiny.asm

BITS 32

              org     0x00010000

              db      0x7F, "ELF"             ; e_ident
              dd      1                                       ; p
              dd      0                                       ; p
              dd      $$                                      ; p
              dw      2                       ; e_type        ; p
              dw      3                       ; e_machine
              dd      _start                  ; e_version     ; p
              dd      _start                  ; e_entry       ; p
              dd      4                       ; e_phoff       ; p
_start:
              mov     bl, 42                  ; e_shoff       ; p
              xor     eax, eax
              inc     eax                     ; e_flags
              int     0x80
              db      0
              dw      0x34                    ; e_ehsize
              dw      0x20                    ; e_phentsize
              db      1                       ; e_phnum
                                              ; e_shentsize
                                              ; e_shnum
                                              ; e_shstrndx

filesize      equ     $ - $$
```

A few observations:

- the program header overlaps with the ELF header
- the entry point is inside the ELF header
  - The implication is that there is executable code inside the header
- the fields having to do with sections are empty
  - it is actually more precise to say that since the file is 45 bytes in size but the ELF header of a 32-bit binary should be 52 bytes in total, those fields are simply not there.

As it turns out, the subset of fields that must contain correct values in order to be loaded by the kernel consists of the following:

- The first 4 bytes of *e_ident* which includes:
  - EI_MAG0 - EI_MAG4: `0x7f`, `E`, `L`, `F`
- *e_type*
- *e_machine*
- *e_entry*
- *e_phoff*
- *e_phnum*

Summary from "A Whirlwind Tutorial on Creating Really Teensy ELF Executables for Linux" (bolding added):

> So: Here's what is and isn't essential in the ELF header. The **first four bytes** have to contain the magic number, or else Linux won't touch it. The other three bytes in the e_ident field are not checked, however, which means we have no less than twelve contiguous bytes we can set to anything at all. **e_type** has to be set to 2, to indicate an executable, and **e_machine** has to be 3, as just noted. e_version is, like the version number inside e_ident, completely ignored. (Which is sort of understandable, seeing as currently there's only one version of the ELF standard.) **e_entry** naturally has to be valid, since it points to the start of the program. And clearly, **e_phoff** needs to contain the correct offset of the program header table in the file, and **e_phnum** needs to contain the right number of entries in said table. e_flags, however, is documented as being currently unused for Intel, so it should be free for us to reuse. e_ehsize is supposed to be used to verify that the ELF header has the expected size, but Linux pays it no mind. e_phentsize is likewise for validating the size of the program header table entries. This one was unchecked in older kernels, but now it needs to be set correctly. Everything else in the ELF header is about the section header table, which doesn't come into play with executable files.

Emulation

Given that the program contains only 7 bytes of instructions and has a malformed header, emulation is a good alternative to heavyweight tools like radare2, IDA, Ghidra, etc. for analyzing/tracing/logging the runtime behavior of this kind of program. The program's code can be emulated via a small python script that utlizes the Unicorn Engine (at time of writing, the Qiling emulation framework is still in alpha and the code is not available). For our purposes right now, it does not matter that the ELF header is malformed, as the only information needed to emulate the binary is its architecture and the file offsets at which to begin and end emulation; this information can be retrieved from a hex dump of the binary without needing to parse the ELF header.

The approach to emulating the `tiny-i386` binary is as follows:

First, retrieve the start and end points for emulation from a hex dump. Then, when writing the script to emulate the program:

- read the file and map it to memory
- set up the stack
- initialize the emulation engine
- implement a hook that allows each executed instruction to be traced and logged to STDOUT
  - a Capstone disassembly engine object will be passed to this hook so that each instruction can be disassembled and its disassembly logged as well
- implement a hook that handles system calls

We know from the source code that the first instruction is `mov bl, 42` and the final instruction is `int 0x80`. We can find these easily in a dump of `tiny-i386`:

```
$ hexdump -C tiny-i386
00000000  7f 45 4c 46 01 00 00 00  00 00 00 00 00 00 01 00  |.ELF............|
```

```
00000010  02 00 03 00 20 00 01 00  20 00 01 00 04 00 00 00  |.... ... .......|
00000020  b3 2a 31 c0 40 cd 80 00  34 00 20 00 01           |.*1.@...4. ..|
0000002d
```

Narrowing down the output:

```
$ hexdump -C -s 0x20 -n 7 tiny-i386
00000020  b3 2a 31 c0 40 cd 80                              |.*1.@..|
00000027
```

There we have it: the offset at which to begin emulation is `0x20` and at which to end is `0x27`.

Since the architecture is already known to us, this is all the information required to emulate the program:

```
#!/usr/bin/python3

from unicorn import *
from unicorn.x86_const import *
from capstone import *
import struct


BASE = 0x100000
STACK_ADDR = 0x0
STACK_SIZE = 1024 * 1024

def read(name):
   with open(name, 'rb') as f:
      return f.read()

#https://github.com/unicorn-engine/unicorn/blob/master/bindings/python/shellcode.py
# callback for tracing instructions
def hook_code(uc, address, size, user_data):
    instruction = uc.mem_read(address, size)    # read this instruction code from mem
    md = user_data
    for i in md.disasm(instruction, address):
        print(">>> Tracing instruction at 0x%x, instruction size = 0x%x, disassembly:


# callback for tracing Linux interrupt
def hook_intr(uc, intno, user_data):
    # only handle syscall
    if intno != 0x80:
        print("got interrupt %x ???" %intno);
        uc.emu_stop()
```

```
        return

    eax = uc.reg_read(UC_X86_REG_EAX)
    eip = uc.reg_read(UC_X86_REG_EIP)

    print(">>> 0x%x: INTERRUPT: 0x%x, EAX = 0x%x" %(eip, intno, eax))

    uc.emu_stop()



def main():

    mu = Uc(UC_ARCH_X86, UC_MODE_32)    # initialize emulation engine class
    mu.mem_map(BASE, STACK_SIZE)        # allocate space at base address
    mu.mem_map(STACK_ADDR, STACK_SIZE)  # allocate space for stack

    mu.mem_write(BASE, read("./target_binaries/tiny-i386"))     # write file to memor
    mu.reg_write(UC_X86_REG_ESP, STACK_ADDR + STACK_SIZE - 1)   # initialize stack

    md = Cs(CS_ARCH_X86, CS_MODE_32)    # initialize disassembler engine class

    # add hooks
    mu.hook_add(UC_HOOK_CODE, hook_code, md)    # pass disassembler engine to hook
    mu.hook_add(UC_HOOK_INTR, hook_intr)

    mu.emu_start(BASE + 0x20, BASE + 0x27)

    print(">>> Emulation Complete.")

if __name__ == "__main__":
    main()
```

emulate_tiny-i386.py hosted with ❤ by **GitHub** **view raw**

When executed, we get a trace + disassembly:

```
$ ./emulate_tiny-i386.py
>>> Tracing instruction at 0x100020, instruction size = 0x2, disassembly:       mov
>>> Tracing instruction at 0x100022, instruction size = 0x2, disassembly:       xor
>>> Tracing instruction at 0x100024, instruction size = 0x1, disassembly:       inc
>>> Tracing instruction at 0x100025, instruction size = 0x2, disassembly:       int
>>> 0x100025: INTERRUPT: 0x80, EAX = 0x1
>>> Emulation Complete.
```

Looks good. No debugger needed.

netspooky's bye: 84 bytes total, 23 bytes of code

An advantage of emulation over debugging is that the emulated instructions (should) have no effect on the host system. Even if the program being emulated contains code that could potentially damage the system it runs on, its instructions are not actually being executed by the CPU, so emulation poses much less risk than debugging (unless there is some way to escape from the emulator, e.g. QEMU VM escape). This is useful for analyzing viruses, crimeware, etc. and in this particular case @Netspooky's `bye` binary, which executes the `reboot` syscall with the `LINUX_REBOOT_CMD_POWER_OFF` argument:

> On a desktop system, this binary will shut down your computer abruptly. There are some potential side effects from a shutdown like this, but personally I haven't experienced any issues with it. However, on a VPS, this specific syscall proves to be a bit of a problem. Since the virtual machine doesn't actually have any of it's own physical hardware (it's either virtualized or shared with the host), the power button on a VPS isn't really a thing. By executing a syscall the effectively "shuts off the power" to the operating system, this puts the VM in an unknown state. So far, whenever this is run on a VPS, it seemingly wipes out the entire instance.

Obviously it is advantageous to be able analyze the runtime behavior of such a program without having to actually load it into memory and execute since we do not want our machine to be shut down, and in a way that may potentially damage the system at that. The script used to analyze `tiny-i386` can be modified to support emulation of x86-64 code and of the `reboot` syscall. The same approach will be followed as before, with minor adjustments.

Before we begin, however, we can first try to read the file's ELF header with `readelf`, take a look at the source code, and then disassemble its code with Capstone to get a sense of what to expect from emulation.

Parsing the Header with readelf

```
$ readelf -h bye
ELF Header:
  Magic:   7f 45 4c 46 ba dc fe 21 43 be 69 19 12 28 eb 3c
  Class:                             <unknown: ba>
  Data:                              <unknown: dc>
  Version:                           254 <unknown: %lx>
  OS/ABI:                            <unknown: 21>
  ABI Version:                       67
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x4
  Start of program headers:          1 (bytes into file)
  Start of section headers:          28 (bytes into file)
  Flags:                             0x0
  Size of this header:               0 (bytes)
  Size of program headers:           0 (bytes)
  Number of program headers:         0
  Size of section headers:           0 (bytes)
  Number of section headers:         1
  Section header string table index: 0
readelf: Warning: possibly corrupt ELF header - it has a non-zero program header offset
```

The ELF header is clearly malformed. At least we can see the entry point is at offset `0x4`.

The Source Code

```
; 84 byte LINUX_REBOOT_CMD_POWER_OFF Binary Golf
BITS 64
  org 0x100000000
;---------------------+------+------------+------------------------------------------
; CODE LISTING        | OFFS | ASSEMBLY   | CODE COMMENT
;---------------------+------+------------+------------------------------------------
  db 0x7F, "ELF"      ; 0x0  |   7f454c46 | PROTIP: Can use magic as a constant ;)
_start:               ;------|------------|------------------------------------------
  mov edx, 0x4321fedc ; 0x04 | badcfe2143 | Moving magic values...
  mov esi, 0x28121969 ; 0x09 | be69191228 | into their respective places
  jmp short reeb      ; 0x0E |       eb3c | Short jump down to @x4c
  dw 2                ; 0x10 |       0200 |
  dw 0x3e             ; 0x12 |       3e00 |
  dd 1                ; 0x14 |   01000000 |
  dd _start - $$      ; 0x18 |   04000000 |
phdr:                 ;------|------------|------------------------------------------
  dd 1                ; 0x1C |   01000000 |
  dd phdr - $$        ; 0x20 |   1c000000 |
  dd 0                ; 0x24 |   00000000 |
  dd 0                ; 0x28 |   00000000 |
  dq $$               ; 0x2C |   00000000 |
                      ; 0x30 |   01000000 |
  dw 0x40             ; 0x34 |       4000 |
  dw 0x38             ; 0x36 |       3800 |
  dw 1                ; 0x38 |       0100 |
  dw 2                ; 0x3A |       0200 |
cya:                  ;------|------------|------------------------------------------
  mov al, 0xa9        ; 0x3C |       b0a9 | Load syscall
  syscall             ; 0x3E |       0f05 | Execute syscall
  dd 0                ; 0x40 |   00000000 | Filler, should try to keep as all 0's
  mov al, 0xa9        ; 0x44 |       b0a9 | Load syscall
  syscall             ; 0x46 |       0f05 | Execute syscall
  dd  0               ; 0x48 |   00000000 | Filler, should try to keep as all 0's
reeb:                 ;------|------------|------------------------------------------
  mov edi, 0xfee1dead ; 0x4C | bfaddee1fe | Load magic "LINUX_REBOOT_CMD_POWER_OFF"
  jmp short cya       ; 0x51 |       ebe9 | Short jmp back to e_shnum/p_filesz @0x3C
  nop                 ; 0x53 |         90 | Filler, could use this byte for code.
;---------------------+------+------------+------------------------------------------
; Note that we are overlaying the ELF Header with the program headers.
; You have 12 bytes minus your short jump from 0x4-0x10 to store code
; Then you have 8 bytes within the program headers at 0x4c for more
```

```
; code, plus e_shentsize and the lower bytes of p_filesz + p_memsz for
; storage and code if you stay within the bounds - still testing.
;
;      LINUX_REBOOT_CMD_POWER_OFF
;              (RB_POWER_OFF, 0x4321fedc; since Linux 2.1.30).  The message
;              "Power down." is printed, the system is stopped, and all power
;              is removed from the system, if possible.  If not preceded by a
;              sync(2), data will be lost.
; [ Compile ]
; nasm -f bin -o bye bye.nasm
;
; One Liner
; base64 -d <<< f0VMRrrc/iFDvmkZEijrPAIAPgABAAAABAAAAAEAAAAcAAAAAAAAAAAAAAAAAAAAAAQAAA
;
; Syscall reference: http://man7.org/linux/man-pages/man2/reboot.2.html
```

bye.asm hosted with ❤ by **GitHub** **view raw**

```
; [ Full breakdown ]
; --- Elf Header
; Offset  #  Value            Purpose
; 0-3     A  7f454c46         Magic number - 0x7F, then 'ELF' in ASCII
; 4       B  ba               1 = 32 bit, 2 = 64 bit
; 5       C  dc               1 = little endian, 2 = big endian
; 6       D  fe               ELF Version
; 7       E  21               OS ABI - usually 0 for System V
; 8-F     F  43be69191228eb3c Unused/padding
; 10-11   G  0200             1 = relocatable, 2 = executable, 3 = shared, 4 = core
; 12-13   H  3e00             Instruction set
; 14-17   I  01000000         ELF Version
; 18-1F   J  0400000001000000 Program entry position
; 20-27   K  1c00000000000000 Program header table position - This is actually in th
; 28-2f   L  0000000000000000 Section header table position (Don't have one here so
; 30-33   M  01000000         Flags - architecture dependent
; 34-35   N  4000             Header size
; 36-37   O  3800             Size of an entry in the program header table
; 38-39   P  0100             Number of entries in the program header table
; 3A-3B   Q  0200             Size of an entry in the section header table
; 3C-3D   R  b0a9             Number of entries in the section header table [holds m
; 3E-3F   S  0f05             Index in section header table with the section name [h
;
; --- Program Header
; OFFSET  #   Value           Purpose
; 1C-1F   PA  01000000        Type of segment
;                               0 = null - ignore the entry
```

```
;                                1 = load - clear p_memsz bytes at p_vaddr to 0, then
;                                2 = dynamic - requires dynamic linking
;                                3 = interp - contains a file path to an executable t
;                                4 = note section
; 20-23   PB  1c000000         Flags
;                                1 = executable
;                                2 = writable
;                                4 = readable
;                                In this case the flags are 1c which is 00011100
;                                The ABI only pays attention to the lowest three bits
; 24-2B   PC 0000000000000000   The offset in the file that the data for this segment
; 2C-33   PD 0000000001000000   Where you should start to put this segment in virtual
; 34-3B   PE 4000380001000200   Physical Address
; 3C-43   PF b0a90f0500000000   Size of the segment in the file (p_filesz) | NOTE: Ca
; 44-4B   PG b0a90f0500000000   Size of the segment in memory (p_memsz)    | are equa
; 4C-43   PH bfaddee1feebe990   The required alignment for this section (must be a po
;
; Breakdown of the hex dump according to the above data
;           A---------- B- C- D- E-  F----------------------
; 00000000  7f 45 4c 46 ba dc fe 21  43 be 69 19 12 28 eb 3c  |.ELF...!C.i..(.<|
;                                                PA---------
;           G---- H---- I----------  J----------------------
; 00000010  02 00 3e 00 01 00 00 00  04 00 00 00 01 00 00 00  |..>.............|
;           PB--------- PC---------------------- PD---------
;           K----------------------  L----------------------
; 00000020  1c 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
;           PD--------- PE---------------------- PF---------
;           M---------- N---- O----  P---- Q---- R---- S----
; 00000030  01 00 00 00 40 00 38 00  01 00 02 00 b0 a9 0f 05  |....@.8.........|
;           PF--------- PG---------------------- PH---------
; 00000040  00 00 00 00 b0 a9 0f 05  00 00 00 00 bf ad de e1  |................|
;           PH---------
; 00000050  fe eb e9 90                                       |....|
```

extendedcomment.asm hosted with ❤ by **GitHub** **view raw**

The bytes that the instructions are composed of are not contiguous - rather than consisting of a single stream of bytes, some code resides at the beginning and the end of the ELF header with data and `00` bytes in between; in addition to accounting for the output produced by `readelf` above, this may pose a challenge for correct disassembly.

Disassembly with Capstone and Radare2

According to the comments in the source code of the file, the last instruction is located at the last byte of the file - offset `0x53`. Using this information, a simple script to disassemble the code with Capstone can be written:

```python
#!/usr/bin/python3

from capstone import *

CODE = []
with open("./bye", "rb") as f:
    f.seek(4)
    CODE = f.read()

md = md = Cs(CS_ARCH_X86, CS_MODE_64)
for i in md.disasm(CODE, 0x1000):
    print("0x%x:\t%s\t%s" %(i.address, i.mnemonic, i.op_str))
```

disassemble_bye_1.py hosted with ❤ by **GitHub** **view raw**

This produces the following disassembly:

```
$ ./disassemble_bye.py
0x1000: mov     edx, 0x4321fedc
0x1005: mov     esi, 0x28121969
0x100a: jmp     0x1048
0x100c: add     al, byte ptr [rax]
0x100e: add     byte ptr ds:[rcx], al
0x1011: add     byte ptr [rax], al
0x1013: add     byte ptr [rax + rax], al
0x1016: add     byte ptr [rax], al
0x1018: add     dword ptr [rax], eax
0x101a: add     byte ptr [rax], al
0x101c: sbb     al, 0
0x101e: add     byte ptr [rax], al
0x1020: add     byte ptr [rax], al
0x1022: add     byte ptr [rax], al
0x1024: add     byte ptr [rax], al
0x1026: add     byte ptr [rax], al
0x1028: add     byte ptr [rax], al
0x102a: add     byte ptr [rax], al
0x102c: add     dword ptr [rax], eax
0x102e: add     byte ptr [rax], al
0x1030: add     byte ptr [rax], dil
0x1033: add     byte ptr [rcx], al
0x1035: add     byte ptr [rdx], al
0x1037: add     byte ptr [rax + 0x50fa9], dh
0x103d: add     byte ptr [rax], al
0x103f: add     byte ptr [rax + 0x50fa9], dh
0x1045: add     byte ptr [rax], al
0x1047: add     byte ptr [rdi - 0x11e2153], bh
```

```
0x104d: jmp     0x1038
0x104f: nop
```

This is clearly incorrect. What happened? As it turns out, Capstone is a *linear sweep*-based disassembler (as opposed to *recursive traversal*-based, like radare2)[7][8]. This means that beginning at the start address, it disassembles all bytes as code until the end address, ignoring flow-of-control. In the disassembly above, quite a bit of null bytes and data are being decoded as instructions. We can compensate for this manually somewhat by ignoring the bytes between the `jmp` at offset `0xa` and the `cya` label at offset `0x3c` (see the source code, lines 11 and 27 in particular):

```python
#!/usr/bin/python3

from capstone import *

buf_A = []
buf_B = []
with open("./bye", "rb") as f:
    f.seek(4)
    buf_A = f.read(12)
    f.seek(0x3c)
    buf_B = f.read()

CODE = buf_A + buf_B

md = md = Cs(CS_ARCH_X86, CS_MODE_64)
md.skipdata = True
for i in md.disasm(CODE, 0x1000):
    print("0x%x:\t%s\t%s" %(i.address, i.mnemonic, i.op_str))
```

disassemble_bye_2.py hosted with ❤ by **GitHub** **view raw**

The disassembly produced after these adjustments is less egregiously erroneous (but still not quite correct):

```
$ ./disassemble_bye_2.py
0x1000: mov     edx, 0x4321fedc
0x1005: mov     esi, 0x28121969
0x100a: jmp     0x1048                <------- jumps beyond the end of the buffer
0x100c: mov     al, 0xa9
0x100e: syscall
0x1010: add     byte ptr [rax], al    <------- error
0x1012: add     byte ptr [rax], al    <------- error
0x1014: mov     al, 0xa9
0x1016: syscall
0x1018: add     byte ptr [rax], al    <------- error
0x101a: add     byte ptr [rax], al    <------- error
0x101c: mov     edi, 0xfee1dead
0x1021: jmp     0x100c
```

```
0x1023: nop
```

At least it somewhat resembles the source code.

How does radare2 fare in disassembling this binary? Not well at all. In fact, it completely fails (maybe I am not using the correct flags?):

```
$ r2 bye
Warning: Cannot initialize program headers
Warning: Cannot initialize dynamic strings
Warning: Cannot initialize dynamic section
[0x00000004]> pd
            ;-- entry0:
            ;-- eip:
            0x00000004      ff             invalid
            0x00000005      ff             invalid
            0x00000006      ff             invalid
            0x00000007      ff             invalid
            0x00000008      ff             invalid
            0x00000009      ff             invalid
            0x0000000a      ff             invalid
            0x0000000b      ff             invalid
            0x0000000c      ff             invalid
                            .
                            .
                            .
            0x00000031      ff             invalid
            0x00000032      ff             invalid
            0x00000033      ff             invalid
            ;-- section_end.ehdr:
            0x00000034      ff             invalid
            0x00000035      ff             invalid
            0x00000036      ff             invalid
                            .
                            .
                            .
            0x00000040      ff             invalid
            0x00000041      ff             invalid
            0x00000042      ff             invalid
            0x00000043      ff             invalid
[0x00000004]>
```

Looks like disassembly is not particularly helpful here.

Emulation

Emulation seems to be the most reasonable option. The program responsible for handling emulation of `bye` includes code for handling x86-64 syscalls on lines 26 - 41, allowing us to see the arguments in the registers when the syscall is made:

```
1  #!/usr/bin/python3
2
3  from unicorn import *
4  from unicorn.x86_const import *
5  from capstone import *
6  import struct
7
8
9  BASE = 0x100000
10 STACK_ADDR = 0x0
11 STACK_SIZE = 1024 * 1024
12
13 def read(name):
14    with open(name, 'rb') as f:
15       return f.read()
16
17 #https://github.com/unicorn-engine/unicorn/blob/master/bindings/python/shellcode.py
18 # callback for tracing instructions
19 def hook_code(uc, address, size, user_data):
20     instruction = uc.mem_read(address, size)    # read this instruction code from mem
21     md = user_data
22     for i in md.disasm(instruction, address):
23         print(">>> Tracing instruction at 0x%x, instruction size = 0x%x, disassembly:
24
25
26 def hook_syscall64(mu, user_data):
27     rax = mu.reg_read(UC_X86_REG_RAX)
28     rdi = mu.reg_read(UC_X86_REG_RDI)
29     rsi = mu.reg_read(UC_X86_REG_RSI)
30     rdx = mu.reg_read(UC_X86_REG_RDX)
31
32     print(">>> got SYSCALL with RAX = %d" %(rax))
33
34     if rax == 0xa9:    #reboot
35         print(">>> SYSCALL:\treboot\n>>> ARGUMENTS:\tRDI = 0x%x\tRSI = 0x%x\tRDX = 0x
36         # see syscall table at https://blog.rchapman.org/posts/Linux_System_Call_Tabl
37     else:
38         rip = mu.reg_read(UC_X86_REG_RIP)
39         print(">>> Syscall Found at 0x%x: , RAX = 0x%x" %(rip, rax))
40
41     mu.emu_stop()
42
```

```python


def main():

    mu = Uc(UC_ARCH_X86, UC_MODE_64)     # initialize emulation engine class
    mu.mem_map(BASE, STACK_SIZE)         # allocate space at base address
    mu.mem_map(STACK_ADDR, STACK_SIZE)  # allocate space for stack

    mu.mem_write(BASE, read("./linux/bye"))     # write file to memory
    mu.reg_write(UC_X86_REG_RSP, STACK_ADDR + STACK_SIZE - 1)   # initialize stack

    md = Cs(CS_ARCH_X86, CS_MODE_64)     # initialize disassembler engine class

    # add hooks
    mu.hook_add(UC_HOOK_CODE, hook_code, md)     # pass disassembler engine to hook
    mu.hook_add(UC_HOOK_INSN, hook_syscall64, None, 1, 0, UC_X86_INS_SYSCALL) # hook

    mu.emu_start(BASE + 0x4, BASE + 0x53)

    print(">>> Emulation Complete.")

if __name__ == "__main__":
    main()
```

emulate_bye.py hosted with ❤ by **GitHub** **view raw**

Emulated execution trace:

```
$ ./emulate_bye.py
>>> Tracing instruction at 0x100004, instruction size = 0x5, disassembly:       mov
>>> Tracing instruction at 0x100009, instruction size = 0x5, disassembly:       mov
>>> Tracing instruction at 0x10000e, instruction size = 0x2, disassembly:       jmp
>>> Tracing instruction at 0x10004c, instruction size = 0x5, disassembly:       mov
>>> Tracing instruction at 0x100051, instruction size = 0x2, disassembly:       jmp
>>> Tracing instruction at 0x10003c, instruction size = 0x2, disassembly:       mov
>>> Tracing instruction at 0x10003e, instruction size = 0x2, disassembly:       syscall
>>> got SYSCALL with RAX = 169
>>> SYSCALL:    reboot
>>> ARGUMENTS:  RDI = 0xfee1dead        RSI = 0x28121969        RDX = 0x4321fedc
>>> Emulation Complete.
```

Very nice. Not only do we see the runtime behavior of the program without executing it, but we get essentially correct disassembly as well. According to the source code and the attempt at disassembly using Capstone, the `reboot` syscall is made twice, but obviously only the first one would ever be executed, meaning the instructions following the first `reboot` syscall are unreachable. Perhaps emulation is also useful for analysing obfuscated assembly code? ;)

Conclusion

As we can see, emulation via Unicorn is a very powerful methdod for analyzing programs that can't be properly parsed or disassembled with the ususal tools. However, the difficulty of writing the program that performs the emulation scales with the complexity of the program being emulated. An example of this is the necessity of implementing support manually for interrupts and syscalls. In the next post, somewhat larger programs with a greater range of functionality will be analyzed. Up to this point the start and end addresses of emulation have been manually retrieved from a dump of the target binary; a method of robustly parsing malformed ELF headers will also be explored so that the code start and end offsets can be retrieved in an automated fashion.

Links and References

1. ELF Crafting: Advanced Anti-analysis techniques for the Linux Platform
2. Striking Back GDB and IDA debuggers through malformed ELF executables
3. Screwing elf header for fun and profit
4. Modern Linux Malware Exposed
5. Understanding Linux Malware
6. Linux process execution and the useless ELF header fields
7. Disassembly of Executable Code Revisited - discusses linear sweep and recursive traversal disassembly algorithms
8. On Disassembling Obfuscated Assembly

Muppetlabs' Tiny Binaries:

- The Teensy Files

netspooky's Experiments:

- source code of "golfclub" binaries on github
- ELF Binary Mangling Part 1 — Concepts
- Elf Binary Mangling Pt. 2: Golfin'
- Elf Binary Mangling Part 3 — Weaponization

Unicorn Engine materials:

- Unicorn Engine tutorial
- Unicorn Engine Reference (Unofficial)
- sample_x86.py
- shellcode.py

1 Comment Binary Research

Login

Recommend Tweet Share Sort by Oldest

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Jim • 3 days ago

I'm not seeing the scripts/excerpts you mention, am I missing something or are those scripts available so I can recreate your results?

Reply • Share ›

Subscribe Add Disqus to your site Disqus' Privacy Policy DISQUS

---

content created by julian