Maze challenge

Thanh Dinh Ta[1]

Verimag

Abstract

Manually resolving a reverse engineering challenge is not a linear process, particularly in the case the challenge uses a execution model which is not familiar with the analyzer. We present below our process of analyzing a challenge which uses return-oriented programming technique.

1 Introduction

The challenge analyzed here is what I have to deal with in the job interview at TETRANE. It is a 32-bit `ELF` binary named `mecung`. I cannot resolve the challenge on the time of interview, I have been so nervous with the idea that I must be rejected if cannot finding out the correct key for the challenge. In the rest of this section I try to present several approaches that I have used in the interview, the failures and the lessons learned.

1.1 Behavior analysis

Running the challenge, we[1] know that it requests a serial as input argument; try with a random serial we get the result Listing 1, so no useful information obtained. A traditional approach

```
./mecung.elf sdfs
Oops, try again
```

Listing 1: Requesting a serial as input argument

in trying to get more information is to run the binary under some standard tracer, we try with `ltrace` and get the input as in Listing 2. We are lucky here, from the tracing result, we know that the binary requests the environment variable `SHOW_ME_THE_MECUNG`. We then set the

```
ltrace ./mecung.elf sfsdf
__libc_start_main([ "./mecung.elf", "sfsdf" ] <unfinished ...>
getenv("SHOW_ME_THE_MECUNG")
fwrite("\nOops, try again\n", 1, 17, 0xf7750e80
Oops, try again
)                                                                   = 17
exit(1 <no return ...>
+++ exited (status 1) +++
```

Listing 2: Running under `ltrace`

variable and trace the binary again, this time it shows a more interesting output:

[1]Here, "we" implies the person being interviewed, so that means indeed "I". The interviewers are experienced reversers, they are familiar already with the techniques of the challenge and obviously have not any problem as presented in the document.

```
set -x SHOW_ME_THE_MECUNG sfsdf
./mecung.elf sfsdf
#######
#*#   #
#   # #
##### #
#?    #
#######

Oops, try again
```

This is a maze, with a blink and colored star, but nothing more; we temporarily stop with this behavior analysis and come back to the binary.

1.2 Static analysis

The main function is found at `0x80488ef`, it calls a two-arguments function at `0x804896c` and another one-argument function at `0x804897b`. For the function at `0x804896c`, its first input is also the pointer to the buffer of the input serial, it parse also the serial character-by-character. For the function at `0x804897b`, it switches its argument with `esp`, so the purpose is to jump to the address stored at the argument.

We do not know where the maze and the string "`Oops, try again`" are printed out, then look into the `strings` of the binary, we jump to the address `0x8048e6f` which contains the string, and it is referenced from the instruction `push` at `0x80487e6`. The instruction is located in a function at `0x804868c`.

Looking into the function at `0x804868c`, it is direct to understand why the maze is printed only when the environment variable `SHOW_ME_THE_MECUNG`, and it is printed out by a function at `0x80485e5` (which is called by the function at `0x804868c`). We observe also the location where a string "`Well done!`" is printed out, this is obviously the location we have to reach (by giving a good serial)[1]. There are two questions must be answered here:

- where and when is the function at `0x804868c` called?, and
- what is its semantics and the relation between this function and the serial input argument?

As previously mentioned, there are two functions called by `main`: one at `0x80487fd` and the other at `0x8048990`. Looking into the function at `0x804896c`, we do not see any non-local control flow, by setting a break-point at the instruction at `0x8048971` we observe that when executing to this instruction, there are no output printed out yet: so the function at `0x804868c` must be called from somewhere in the function at `0x8048990`.

We observe also that the function calls two interesting functions, one at `0x804874f` and the other at `0x8048782`. We guess that the latter (i.e. one at `0x8048782`) is not important since it has no return value, and it seems to print the colored `star` only. The former is more complicated and more interesting, it is a 3-arguments functions and contains many checks, but we do not understand their semantics yet.

[1]By setting a break-point at the function, we know that this is the single function where the string "`Oops, try again`" (and maybe "`Well done!`" also) is printed out.

1.3 Dynamic analysis

Now we almost sure that the function at `0x804868c` is called in the function at `0x8048990`, moreover we know already that this function simply transforms its control flow to the address referenced by its single argument. By setting a break-point at the function, we start tracing it step-by-step, and start being confused: we observe multiple `ret`(s) which go after small 3-instructions code snippets.

We guess initially that this challenge is a kind of *virtual machine*, so its form is hard to be recognized by static analysis. We have implemented a `Pintool` [1] which can tracing while skipping several uninterested functions of a program (e.g. for the case of the challenge, they are `printf`, `exit`, `sleep`), and a tool which can construct the control-flow-graph of a program from traces[1].

There is a problem that the calls to `sleep` will slow down the tracing process, so we use the standard `LD_PRELOAD` trick to bypass this function. We implement an empty `sleep`, compile it as a shared library and set `LD_PRELOAD` to the compiled file (c.f. Listing 3). Start tracing the

```
unsigned int sleep(unsigned int nb_sec)
{
  return 0;
}
gcc -m32 -shared -fPIC -Wall sleep.c -o sleep.o
set -x LD_PRELOAD /home/bigraph/Documents/Research/sleep.o
```

Listing 3: `sleep.c`

binary with several random inputs, we obtained a partial CFG show in Figure 1; but now we are confused: where does this virtual machine come from?

1.4 Other behavior analysis

No answer given in Section 1.2 is solved, and even more questions are posed. We hopelessly observe the mazes shown by the challenge from different inputs, sometimes the star is displayed with different positions and locations, but nothing is clear.

1.5 Lessons learned

I cannot resolve the challenge on the time of interview, though Samuel (the interviewer) has given tons of suggestions. He knows that I am Vietnamese then he gave me the first hints.

Sam: Hey, does “me cung” mean something in Vietnamese?
Me: Well, that means “maze”.
Sam: This is a hint for you.

I do not understand what does it means, then I have tried some serials: `mecung`, `labyrinth`, `maze`, etc., but nothing works, obviously :(. He explained me how the function at `0x8048990` can execute 3-instructions code snippets by “unrolling” the stack organized by the function at `0x80487fd`. But I did not capture his suggestions, and could not complete the challenge :(.

[1]For a trace, we can only recover a partial CFG of the program, the CFG constructed from several traces gives still incomplete CFG but better that one constructed from a trace.

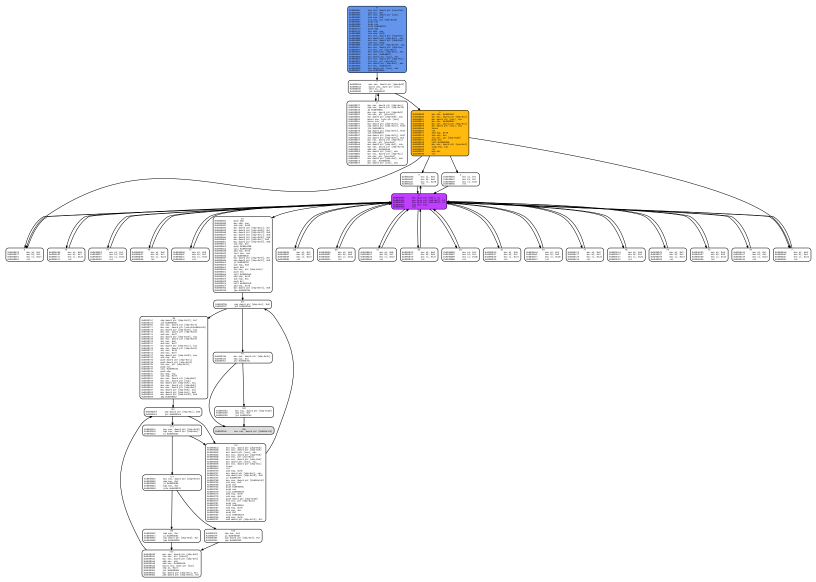

Figure 1: Partial CFG

References

[1] C.-K. Luk et al. "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation". In: PLDI. 2005.