Maze challenge

Thanh Dinh Ta[1]

Verimag

Abstract

Solving a reverse engineering challenge is like mathematical problem solving, it 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, but 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 get 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 and a weirdly question mark, 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). There are two questions must be answered here:

1. where and when is the function at `0x804868c` called?, and
2. what is its semantics and how does it relate with 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 cannot see any non-local control flow; moreover 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.

1.3 Dynamic analysis

Now we almost sure (c.f. Remark 1) 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, moreover the instructions of the block at `0x804899a` is repeatedly executed.

Remark 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.

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].

Remark 2. The CFG constructed from traces is partial, i.e. it is only a subgraph of the real CFG, but it is also a sound approximation of the real CFG of the program.

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: an empty `sleep` is implemented, compiled as a shared library and `LD_PRELOAD` is set 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: Fake `sleep`

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 Failures and lessons learned

When starting with challenge, I though that it must be resolved on interview time if not want to be rejected. I didn't focus on detail analyzing every machine instructions, I tried instead to figure out quickly its high level semantics though its behaviors; but this approach did not work in this case: the *signal-to-noise* of observing the differences between printed out mazes is so small that I can only recognize the difference but cannot figure out any useful idea.

Samuel, the interviewer, has given generously tons of suggestions. After I have discovered quickly the existence of `SHOW_ME_THE_MECUNG` but blocked (until the end of the interview) at

[1] The tools discussed above are obviously not implemented in the interview time: we have implemented them previously in our personal toolbox.

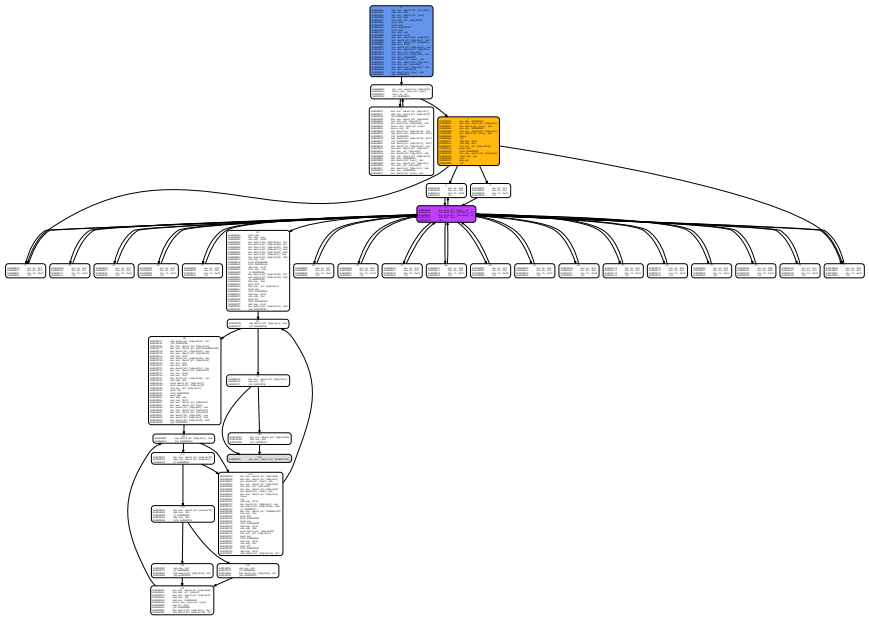

Figure 1: Partial CFG

understanding how the function at `0x804868c` relates with the input serial, he gave me the first hint (maybe he knows that I am Vietnamese):

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

I do not quite understand what does it means, then I have tried naively some serials: `mecung`, `labyrinth`, `maze`, etc., but nothing works, obviously.

Samuel noticed me the relation between the functions at `0x8048990` and at `0x80487fd`: they share indeed an argument. He then explained how the later can execute 3-instructions code snippets by “unrolling” the stack organized by the former; but I could not grasp his suggestions to tackle the challenge :(.

It should be better if we focus on the function at `0x80487fd` since the “stack unrolling” mechanism reveals the *execution model* of the program, understanding this model is one of the keys to solve the challenge.

2 Return-oriented programming

In this section, we present a complete analysis which reveals the execution model of the challenge, then allows us to get a correct serial input. The analysis is done after the interview.

2.1 Execution model

As suggested by Samuel, the “stack unrolling” mechanism of the function at `0x80487fd` is a kind of *object-oriented programming* [2] where each 3-instructions (and a followed `ret`) code

snippet is called a *gadget*, some of them can be observed in Figure 1. The execution model of the binary is realized by two function: one at `0x80487fd`, we named `execute_rop_table`; the other at `0x80487fd`, named `setup_rop_table`.

The function `execute_rop_table` transfers the control flow to the address contained in the first entry of a *rop table* passed through its single argument. This address points to a gadget which ends with a `ret` instruction, that means it transfers the control flow to an address (of the next gadget) contained in the next 4 bytes higher stack address. The semantics of `execute_rop_table` is presented by the `C++` function in Listing 4.

```
auto execute_rop_table(size_t* rop_table) -> void
{
  using rop_fun = void(*)();
  auto entry_rop = reinterpret_cast<rop_fun>(rop_table[0]);
  return entry_rop();
}
```

Listing 4: `setup_rop_table`

Remark 3. We note also that the rop table is located on the stack since the argument (i.e. the address of the rop table) passed into `execute_rop_table` is also a local parameter of `main`.

The *rop table* is also the second argument of `setup_rop_table`, looking into this function (c.f. Listing 5) we observe that the first two entries of the rop table are fixed. Indeed, the first is `0x8048997` (c.f. the `mov` instruction at `0x804881b`) which contains, beside the last `ret` instruction, only a `pop edi`. The second is a *buffer* at `0x804a1e0` (c.f. the `mov` instruction at `0x804882b`), this buffer is then moved into `edi` since the execution of the first gadget.

```
...
0x804881b       mov     edx, offset first_rop_entry
0x8048820       mov     [eax], edx
0x8048822       mov     eax, [ebp+current_rop]
0x8048825       lea     edx, [eax+4]
0x8048828       mov     [ebp+current_rop], edx
0x804882b       mov     edx, offset buffer
...
```

Listing 5: Setup first two instructions

Abstract machine In summary, the binary has implemented actually by an *abstract machine* whose opcodes are stored sequentially in the stack.

References

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

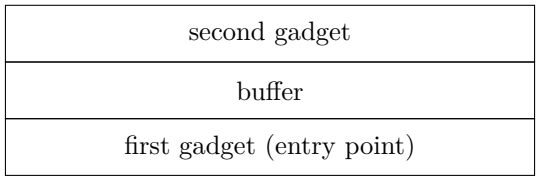

Figure 2: Execution model

[2] H. Shacham. "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)". In: *CCS*. 2007.