Maze challenge

Thanh Dinh Ta[1]

Verimag

Abstract

Solving 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 an analysis for 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
```

Listing 3: Showing the maze

This is a maze, with a blink and colored star and a weird 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

We are almost sure that the function at `0x804868c` is called only in the function at `0x8048990` (c.f. Remark 1), 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 get 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 location 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 4). Start tracing the

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

Listing 4: 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 yet, and even more questions are posed. We hopelessly observe the mazes shown by the challenge from different inputs, sometimes the star is displayed on different positions and with different colors, but nothing is clear.

1.5 Comments

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

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

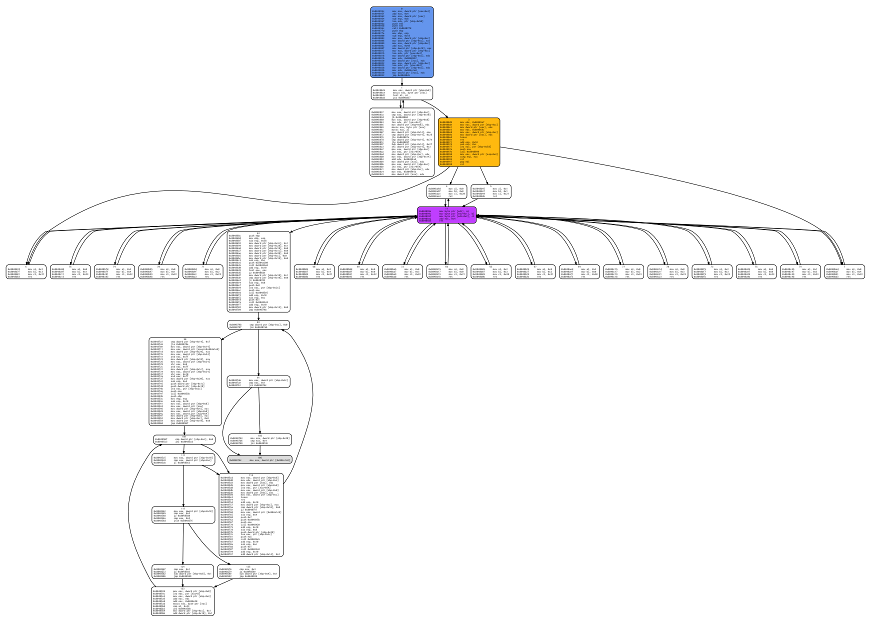

Figure 1: Partial CFG

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.

Samuele, 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 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 did not quite understand this hint, then I have tried naively some serials: `mecung`, `labyrinth`, `maze`, etc., but nothing worked, obviously. Samuele 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 :(.

In fact, it would 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 Samuele, the “stack unrolling” mechanism of the function at `0x80487fd` is a kind of *return-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`.

Rop table execution 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 5.

```
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 5: `execute_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`.

Rop table construction The *rop table* is also the second argument of `setup_rop_table`, looking into this function (c.f. Listing 6) 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 6: Setup first two instructions

Other entries are computed from characters of the serial, which is also the first argument. In short, each character is used to compute an offset with then be added with the base address

`0x0x80489ad`; for each input-dependent gadget added, the input-independent gadget at the address `0x804899a` is added also. The semantics of `setup_rop_table` is shown in Listing 8.

Rop gadgets In the function `setup_rop_table` (c.f. Listing 8), the address of an entry is computed from the corresponding character of the serial by:

$$address = (char - \texttt{0x27}) * 8 + \texttt{0x80489ad} \tag{1}$$

So we can find out the gadgets by disassembling addresses of form $8 * k +$ `0x80489ad`. We observe also that all gadgets have a single `ret` instruction at the end, this instruction transforms the control flow to the next gadget whose address is in the top of stack (i.e. `[esp]`); the stack's top is incremented then.

Remark 4. The control flow transformation between gadgets make the stack's top incremented each time 4 bytes, that gives the effect of "stack unrolling".

Abstract machine The challenge can be viewed as an "abstract program" executed by an *abstract machine*, the opcodes (i.e. addresses of gadgets) of the program are stored sequentially in the stack, they form the opcode table (c.f. Figure 2). The abstract machine executes the program by *fetching* an opcode, *decoding* it (i.e. jump to the address stored in the opcode), then *executing* it (i.e. execute instructions of the gadget).

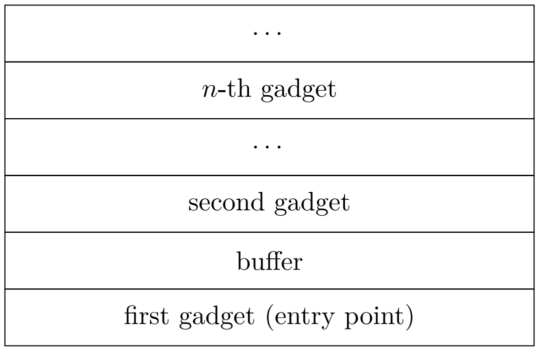

Figure 2: Execution model

The control flow of any program in this abstract machine is very simple: the entry point is always the first entry in the opcode table; the control flow is always transferred from one opcode into the next one in the opcode table, *conditional control flow does not exist.*

Remark 5. The "abstract program" is the opcode table, it is constructed from the input serial. This machine is not Turing complete, many algorithms cannot be implemented on it.

2.2 Cell moving and opcode semantics

As previously discussed (c.f. Section 1.2), the decision whether a serial is good or bad is realized in the function at `0x804868c`, we name it `check_maze`; this function is indeed a gadget which is put into the rop table by `setup_rop_table` (c.f. the `mov` instruction at `0x80488e3`). It is a function without argument since it is a gadget.

We observe that the function accesses to a global *buffer* at `0x804a1e0`, which is put into the second entry of the rop table (c.f. Listing 8). There is only one function called in `check_maze` which uses this buffer to change the local state of `check_maze`, this is the 3-arguments function at `0x804853b` (c.f. `call` instruction at `0x804874f`).

The function at `0x804853b` accesses to an interesting address `0x8048e20`, which contains the `ASCII` codes of the *maze*. It actually updates the *position* of a cell in the maze, depending on the number of *moving step* and the *moving direction*; they are passed respectively as the first, the second and the third argument of the function. Thus we name it `update_cell_pos`. In short, a cell will be moved several steps following the direction (up, down, left, right) until it infringes upon the wall or the number of moving step is reached; the function will return the state of the cell: whether it is free or it is in the wall. The semantics of `update_cell_pos` is shown in Listing 9.

Cell moving Once the semantics of `update_cell_pos` is revealed, the semantics of `check_maze` becomes also clear. It moves a cell from the position $(1, 1)$ (i.e. $row = 1$, $column = 1$) using the function `update_cell_pos`, the moving directions and steps are extracted from *buffer* at `0x804a1e0`. Concretely, for each 4-bytes entry of the buffer,

- the first represents the number of moving steps,
- the second represents the moving direction,
- the third represents the printing color of the cell,

and the last is not used. It stops when the cell infringes upon the wall or maximum 8 entries are parsed; and print out “`Well done!`” if the last position of the cell is $(4, 1)$ ($row = 4$, $column = 1$). The semantics of `check_maze` is shown in Listing 10.

Opcode semantics The output is good (i.e. “`Well done!`”) or bad (i.e. “`Oops, try again`”) depending on the data of the buffer at `0x804a1e0`. This buffer is filled by the execution of `opcodes` in the rop table of the abstract machine: each opcode will fill next 4 bytes in the buffer. Since we have known above how the data of the buffer is used, the semantics of each gadget (i.e. opcode semantics) is clear:

- the value moved to `al` represents the number of moving steps,
- the value moved to `bl` represents the moving direction,
- the value moved to `cl` represents the color of the cell.

Remark 6. Between opcodes whose semantics is discussed above, we do not count the opcode whose the corresponding gadget is at `0x804899a` (c.f. Figure 1), it is simply a data mover.

2.3 Solving the challenge

From the analysis in Section 2.1, the challenge has form of an “abstract machine” while the “abstract program” is the opcode table which is built from the input serial. In Section 2.2, we know that `check_maze` is an opcode of this abstract program, and it is indeed the *last executed opcode* since the calls to `exit` function. The last executed opcode consumes the buffer at `0x804a1e0` which is filled by previously executed opcodes

Algorithmically speaking, the good output is reached only by a moving process, from the position $(1, 1)$ to $(4, 1)$ in the maze, which satisfies the following constraints:

- the cell must not infringe upon the wall while moving, and
- the moving process must be realized by no more than 8 opcodes.

Looking into the form of the maze (c.f. Listing 3), the only way to satisfy these constraints is to use the following sequence of moves:

$$\downarrow, \rightarrow\rightarrow, \uparrow, \rightarrow\rightarrow, \downarrow\downarrow, \downarrow, \leftarrow\leftarrow, \leftarrow\leftarrow \tag{2}$$

where the direction of arrows show the moving direction, and the number of arrows represents the moving step. This sequence corresponds with the following sequence of gadget addresses:

$$\begin{gathered}\texttt{0x80489e5}, \texttt{0x8048a1d}, \texttt{0x80489dd}, \texttt{0x8048a1d}, \\ \texttt{0x8048a0d}, \texttt{0x80489e5}, \texttt{0x8048a15}, \texttt{0x8048a15}\end{gathered} \tag{3}$$

The characters of the serial is computed directly from Equation (1), we then get the serial:

$$\texttt{.5} - \texttt{53.44}$$

Executing the challenge with this serial, we observe a nice output which shows the moving process of the star to the question mark, and the string "`Well done!`" as shown in Listing 7.

```
set -x SHOW_ME_THE_MECUNG sfsdf
./mecung.elf .5-53.44
#######
# #   #
#   # #
##### #
#*    #
#######

Well done!
```

Listing 7: Good serial

Remark 7. The sequence 2 represents indeed the *algorithm* of the "abstract program" while the sequence 3 is the "abstract program" running on the abstract machine discussed in Section 2.1.

References

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

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

```cpp
static auto buffer = reinterpret_cast<size_t*>(0x804a1e0);
auto setup_rop_table (size_t* rop_table, char* serial) -> void
{
  auto rop_idx = rop_table;
  auto rop_bound = rop_table + 18;
  auto current_char = serial[0];

  *rop_idx = 0x8048997;
  rop_idx++;

  *rop_idx = reinterpret_cast<size_t>(buffer);
  rop_idx++;

  while (true) {
    if (*serial != 0) {
      if (rop_idx < rop_bound) {
        current_char = *serial;
        serial++;

        if (current_char <= '&' || current_char <= '~')
          throw std::logic_error("bad format: invalid char in serial\n");

        current_char -= 0x27;
        current_char <<= 3; // we can know the location of ROP functions

        *rop_idx = 0x80489ad + current_char;
        rop_idx++;

        *rop_idx = 0x804899a;
        rop_idx++;
      }
      else throw std::logic_error("bad format: serial is too long\n");
    }
    else {
      *rop_idx = 0x80489a7;
      *rop_idx = reinterpret_cast<size_t>(check_maze);
      break;
    }
  }

  return;
}
```

Listing 8: `setup_rop_table`

```
auto update_cell_pos (int* current_pos, int steps, int direction) -> int
{
  // read maze's config
  auto column = current_pos[0];
  auto row = current_pos[1];

  auto in_wall = 0;
  auto step_idx = 0;

  while ((in_wall == 0) && (step_idx < steps)) {
    if (direction == 2) ++row;
    else {
      if (direction > 2) {
        if (direction == 3) --column;
        else if (direction == 4) ++column;
      }
      else if (direction == 1) --row;
    }

    if (maze[column + row * 8] == '#') in_wall = 1;

    ++step_idx;
  }

  // update maze's config
  current_pos[0] = column;
  current_pos[1] = row;

  return in_wall;
}
```

Listing 9: `update_cell_pos`

```
auto check_maze () -> void
{
  auto column = 1, row = 1;

  auto move_steps = 0, move_direction = 0, color = 0;
  auto in_wall = 0; auto idx = 0;

  auto good_config = false;

  while (true) {
    if (in_wall != 0 || idx > 7) {
      if (column != 1 || row != 4) good_config = false;
      else good_config = true;  // column == 1 && row == 4
      break;
    }
    else {
      // first byte (for number of steps)
      move_steps = buffer[idx * 4] | 0xff;

      // second byte (for direction)
      move_direction = (buffer[idx + 4] >> 0x8) | 0xff;

      // third byte (for color)
      color = (buffer[idx + 4] >> 0x10) | 0xff;

      in_wall = update_cell_position(&column, move_steps, move_direction);

      idx++;
    }
  }

  if (good_config) printf("\nWell done!\n"); // column == 1 && row == 4
  else printf("\nOops, try again\n");

  return;
}
```

Listing 10: `check_maze`