GreHack 2015 - ReverseMe

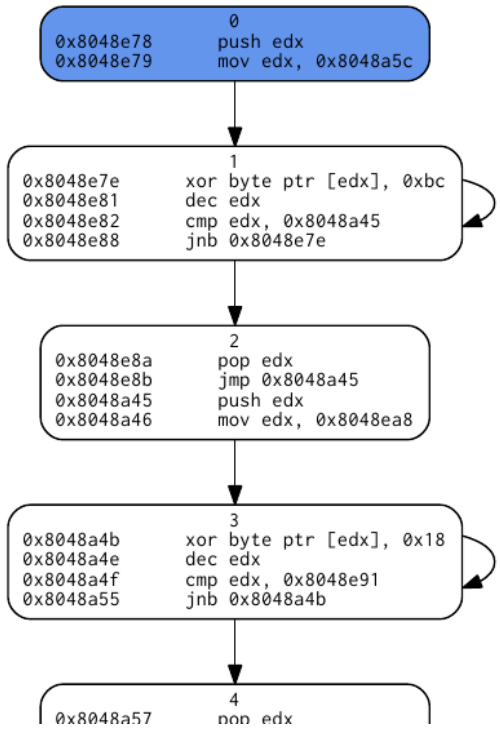
Thanh Dinh Ta[1]

Verimag

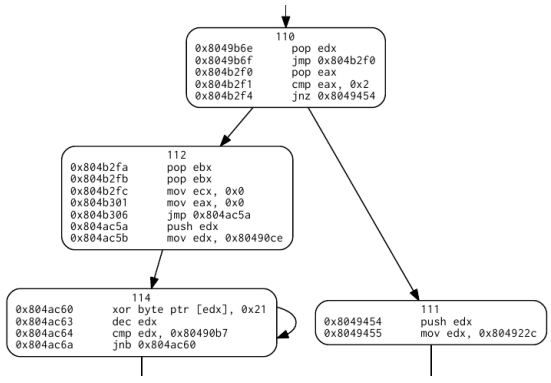
1 Reversing

The challenge is a 32 bit ELF binary, a quick behavior analysis shows that it asks for a single input: if the number of arguments is 0 or more than 1 then the program prints out the message: `Format`. Our goal would be to find out an input so that the program prints out some message different than: `Wrong!`.

Using IDA (evaluation version is enough) to examine the challenge, we quickly recognize that the crackme contains most of self-modifying codes: the next block of instructions is visible only by the execution of existing instructions; so disassembling does not give much information. Running it step by step, we get quickly bored since the interesting instructions seem to be hidden under several layers of `xor` decryption. Figure 1a shows only first 2 layer of decryption; in fact, as can be observed in Figure 1b, the first interesting instructions are visible only at the basic block 112!!!.



(a) First 2 `xor` decryption loops



(b) Instructions check number of input arguments

Figure 1: Initial analysis

A direct black-box attack (e.g. using `perf` to count the number of executed instructions

when the crackme consumes an input string[1]) does not seem to be working: if input is any printable character then the number of executed instructions is always 40681.

2 Differential analysis

We develop in Binsec project [1] a Pin-based [2] tracing tool with powerful functionalities, we use here only a simple one: tracing the list of executed instructions and some concrete information of each instruction: read/write values on registers and memory addresses.

Let's tracing the crackme with two different inputs, e.g. "a" and "Ab", observing the difference between two traces (both of them are of length 40681), *we understand immediately the trick*. Since the instructions used in `xor` decryption loops are identical in both traces, the only difference is at the execution of instructions handling the input. Figure 2 shows the first difference, we recognize immediately that the first byte of input string is xored with 0x41. There are 28 such a difference, next is a comparison to check whether the 29-th input character is 0 or not, so we can guess that the good input should have length 28; tracing the crackme with an input of length 28 confirms this guess (see Figure 3).

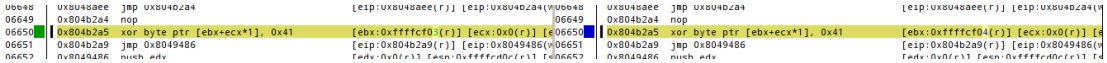


Figure 2: Difference reveals interesting instruction

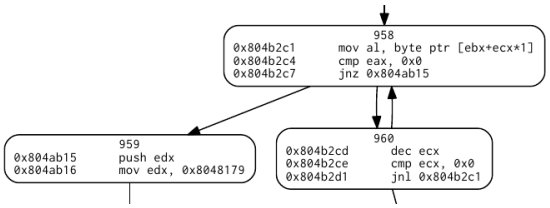


Figure 3: Check input length and value

We observe also in the loop in Figure 3 that, if the input length is 28 then each byte of xored input string is compared with 0. The `OCaml` program in Listing 1 calculates an input string based on 28 differences observed, it gives the key `Alph4t3stf0rc3mille!!!!!!!!!`, using this key as the input for crackme we get `Well done!`.

```
let () =
  let () = Printf.printf "reverseMe␣key:␣" in
  let key_chars = ['\x41'; '\x6c'; '\x70'; '\x68'; '\x34'; '\x74'; '\x33';
                   '\x73'; '\x74'; '\x66'; '\x30'; '\x72'; '\x63'; '\x33';
                   '\x6d'; '\x69'; '\x6c'; '\x6c'; '\x65'; '\x21'; '\x21';
                   '\x21'; '\x21'; '\x21'; '\x21'; '\x21'; '\x21'; '\x21']
  in
  List.iter (fun ch -> Printf.printf "%c" ch) key_chars
```

Listing 1: Calculate good input

[1]See the script of *jvoisin* at https://dustri.org/b/defeating-the-recons-movfuscator-crackme.html

3 Conclusion

This is very nice crackme, using the same principle, we think that the author can give more difficult challenges (for this case, we get the good input in about $30 - 40$ minutes). It may be interesting that our *concolic execution engine* [1] can solve this crackme almost immediately. Our approach is similar with one of our friend Aurélien of Airbus team, he has used directly `gdb` to get the trace, so this is much more cool ☺.

References

[1] BinSec. *Binary Code Analysis for Security*. 2015. URL: http://binsec.gforge.inria.fr/.

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