GreHack 2015 - Weak Up Panda

Thanh Dinh Ta[1]

Verimag

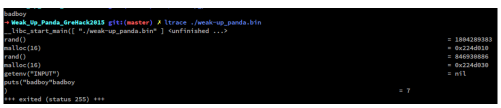
1 Reversing

The challenge is a 64 bit ELF binary, when executes, it prints `badboy` and quits. A quick analysis with `ltrace` reveals that it reads an environment variable named `INPUT` (cf. Figure 1a). So we try to set a value for this variable (our shell is `fish`[1]):

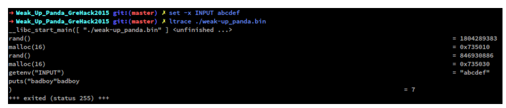
```
set -x INPUT abcdef
```

Listing 1: Setting `INPUT`

and execute the crackme again (cf. Figure 1b); still `badboy` though. There is no useful information yet, except that there are two `rand` calls, each followed by a `malloc`, maybe for storing the returned values of `rand`(s) (?); but we are not sure at this step ☻.



(a) INPUT is unset



(b) INPUT is set

Figure 1: Executing crackme with unset/set `INPUT`

We take a more detail analysis by drop this crackme into `IDA`, the `main` function can be quickly recognized from the familiar pattern: looking from the entry point, the address of `main` is passed as the first parameter for `libc_start_main`; here this address is `0x41806c`. At the first look, the structure of `main` seems simple, but it is not ☹. The nightmare commence from functions called by `main`: any of them has a *tail call* to another function, and this called function, in its turn, has also tail to another one, so on and so forth; we even cannot believe into our eyes when seeing the function call graph generated by `IDA`.

We try next to run the crackme step-by-step in IDA, but we give up soon since it is too long; moreover the semantics of most functions (in this calling chain) are quite simple; then they do not give useful information. This crackme maybe has intention to push the idea of *continuation passing style* to the extreme.

Automated analysis The manual analysis does not work well, so we try an automated approach using our Pintool developed in BinSec project [1]: this Pintool has capability of tracing only instructions executed in a range of addresses (here we set them as the first and the last address of `main`: `0x41806c` and `0x4181f6`), and tracing only internal instructions of the crackme (i.e. the instructions of external calls `rand`, `malloc`, etc will be not counted). The following sequence shows the numbers of executed instructions where lengths of `INPUT` is $1, 2, 3, 4, 5, 6, 7, 8$:

$$51886, 66187, 80511, \ldots, 137771, 168055$$

[1] http://fishshell.com/

and if the length of INPUT then the number instructions is always 168055. The first 7 numbers form an arithmetic progression with the difference 14301, but the difference between 7th to 8th is 30284. These observations suggest that the good INPUT should have the length 8 (or the crackme does consider only 8 first characters of INPUT).

2 Differential analysis

We now focus on values of INPUT with length 8, but observing that the execution trace of the crackme for this length is 168055, so manually examining such a number of instructions is not a good idea ☺. Moreover, by manually examining several functions with IDA, we observe that most of them do not compute anything, moreover many of them are identical!!!. One way to filter out unimportant instructions is to compare the execution traces result in from different values of INPUT. We choose, for example, "abcdefgh" and "HGFEDCBA", the differences between two traces disclosure how the crackme work.

Each character of the input is consumed by a loop which starts from `0x4181b5` (and backs to this address by a `leave` instruction). The prediction that the crackme considers only first 8 characters is confirmed by the following instructions (the order of current examined character is stored in `[rbp - 0x34]`):

```
0x4181b5  mov eax, dword ptr [rbp-0x34]
0x4181b8  and eax, 0xfffffff7
0x4181bb  mov edx, eax
0x4181bd  mov eax, dword ptr [rbp-0x34]
0x4181c0  xor eax, 0xfffffff7
0x4181c3  mov ecx, eax
0x4181c5  mov eax, dword ptr [rbp-0x34]
0x4181c8  sub eax, 0x8
0x4181cb  and eax, ecx
0x4181cd  or eax, edx
0x4181cf  test eax, eax
```

Listing 2: Checking if length is 8

The crackme uses unusual codes to check whether a character is NULL or not (cf. Listing 3; the character is stored in `[rax]`). We note that there are several irrelevant instructions between `0x418175` and `0x418183` (maybe for an obfuscation purpose ☺)

```
0x41816d  movzx eax, byte ptr [rax]
0x418170  movsx eax, al
0x418173  not eax
0x418175  mov ecx, eax
...
0x418183  movzx eax, byte ptr [rax]
0x418186  movsx eax, al
0x418189  sub eax, 0x1
0x41818c  and eax, ecx
0x41818e  test eax, eax
```

Listing 3: Checking if character is NULL

Checksum algorithm Each loop consume a character of INPUT, this character and an additional value are used to calculate a checksum; this checksum is used as the additional value for the next loop. In summary, the checksum is calculated by:

Listing 4: Calculating checksum

then is verified by (see also Listing 6):

Listing 5: Verifying checksum

```
0x41d74f  mov rdx, qword ptr [rax]
0x41d752  mov rax, 0x122d4d05a4299633
0x41d75c  lea rcx, ptr [rdx+rax*1]
0x41d760  mov rax, qword ptr [rbp-0x8]
0x41d764  mov rdx, qword ptr [rax]
0x41d767  mov rax, 0x122d4d05a4299633
0x41d771  add rax, rdx
0x41d774  sar rax, 0x3f
0x41d778  xor rcx, rax
0x41d77b  mov rax, qword ptr [rbp-0x8]
0x41d77f  mov rdx, qword ptr [rax]
0x41d782  mov rax, 0x122d4d05a4299633
0x41d78c  add rax, rdx
0x41d78f  shr rax, 0x3f
0x41d793  add rax, rcx
0x41d796  sub rax, 0x1
0x41d79a  shr rax, 0x3f
```

Listing 6: Instructions verifying checksum

Once the checksum calculation and verification are known, the calculation for good input is direct. We give a file `panda.smt2`[1] of SMT format, use `Z3` to check satisfiability and interpreted values, we obtain a value for `INPUT`: `g!r3h4ck`; that makes the crackme print: `goodboy`.

3 Conclusion

References

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

[1] https://github.com/tathanhdinh/write-ups/blob/master/Weak_Up_Panda_GreHack2015/panda.smt2