BZA Project: Binary Reversing

Andrej Zaujec

April 20, 2022

Contents

1 Assigment **1**

2 Bombs Landed **2**

3 Simple Keygen **6**

4 rop-obf **8**

1 Assigment

Assigment of this project is to reverse engineer the three chosen binaries and propose the write-up of solutions, which are either from hackthebox or the crackme. The difficulty of these binaries should be medium or easy. I did choose one binary from hackthebox and two others from crackme.

- Bombs Landed: had hard difficulty in time of reversing it and afterwards the difficulty was set to medium
- Simple Keygen: has easy diffuculty
- rop-obf: has medium diffuculty

Every of these challenges is ELF binary, I used cutter as the dissasembler, decompiler and debugger for every challenge. Cutter is extensive reversing tool that provides GUI features on-top of the radare2. Each of following chapters describes one of the challenges and way how I solved it.

2 Bombs Landed

The object of challenge is to find the password that prints 'you win', patching of binary was not forbidden. The most hardest challenge for me. First thing I did notice was the main function is splitted into two branches that are exclusive so whenever the program flow ended up in one branch it could not end up in the other. These two main branches can be seen on Figure.

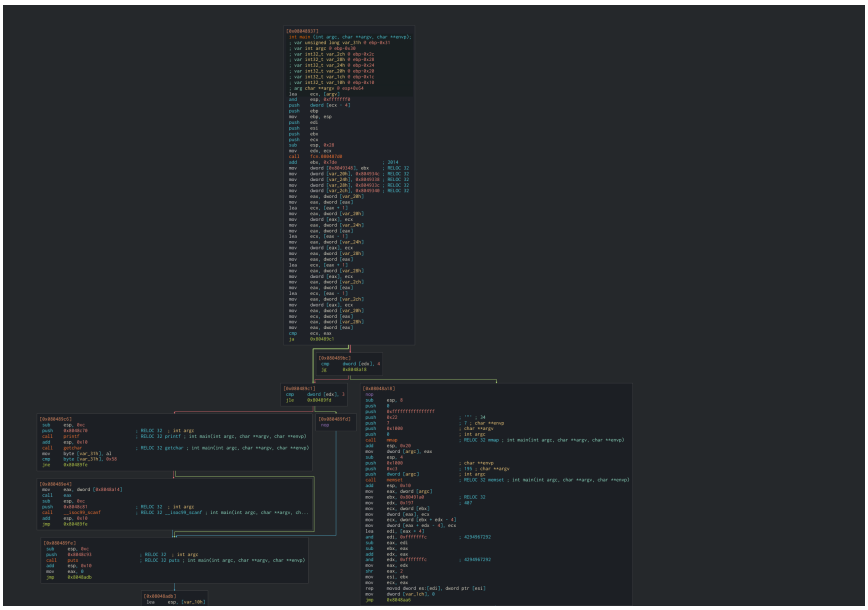

Figure 1: Two exclusive branches

The selection of the branch depends on the number of arguments passed to the program. If the count of arguments equals to three left branch will be selected. If the arguments count equals to 4 the righ branch is chosen. I did spent many time in the left branch as I thought it would be somehow important. The left branch can be seen on Figure. Firstly, there is prompting for input via the getchar call and then there is check for the first character of the given string. The check is for the 'X' letter, if the condiction succeed the program will call some address. But as much as I tried the calling address did not changed and at all and every attempts ended up as segmentation fault.

I decided to move to the right branch after quite time spent in left. The right branch from first look seemed to be doing some calculation and memory writting but nothing straightforward that could be seen without evaluation.

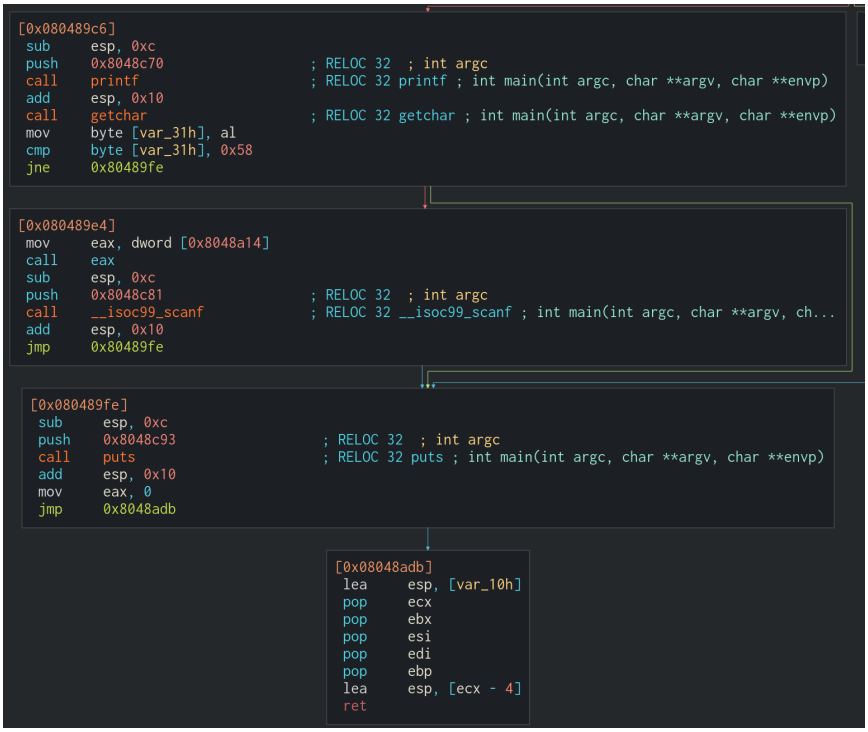

Figure 2: Left branch

The calculation and memory write of right branch is on the Figure. The most interesting was there was call to that newly written memory. Once I launched the debugger in cutter, hit breakpoint at the call and start stepping stepping arround, it was obvious the newly written data was dynamicly generated code. The decompiled function in cutter is on the Figure.

After thorough inspection, this newly generated code does prompt for input via scanf and then use strncmp to compare two strings. When I provided the exact same input as the strncmp was checking against it failed, which was weird. At this point, I did not known what to do with the failing strncmp as the inputs were same but it was returning non-zero code anyway. I stumbled around LD PRELOAD technique, on the internet that could help me with debugging strncmp. The aim of this technique is to replace any function with help of linker that is dynamically loaded from the shared libraries, typical example is the functions from libc library. As the binary is not statically compiled and was using shared libraries this was possible. The newly created strncmp looked like this and it yield the string that is compared against.

```
#define _GNU_SOURCE
#include <dlfcn.h>
#include <stdio.h>
#include <string.h>

 // Define an alternative name for strcmp()
 int (*orig_strncmp)(const char *str1, const char *str2, size_t n);

 int strncmp(const char *str1, const char *str2, size_t n) {
    // Backup the orginal call to strcmp() in orig_strcmp()
    // by initialazing the pointer of orig_strcmp().
    if(!orig_strncmp) orig_strncmp = dlsym(RTLD_NEXT, "strncmp");
    printf("You should try '%s'\n", str2);

    // return the proper result of strcmp()
    return orig_strncmp(str1,str2,n);
 }
```

This C code has to be compiled as shared library object and passed to the linker with help of env variable during running the binary, this is shown in below bash code.

```
#The m32 stands for 32-bit architecture
```

```
*(undefined4 *)(unaff_EBP + -0x14) = 0x6c333374;
*(undefined4 *)(unaff_EBP + -0x5c) = 0x647f6573;
*(undefined4 *)(unaff_EBP + -0x58) = 0x786f7c6f;
*(undefined4 *)(unaff_EBP + -0x54) = 0x6463656d;
*(undefined4 *)(unaff_EBP + -0x50) = 0x6c657e6d;
*(undefined4 *)(unaff_EBP + -0x4c) = 0x676e6463;
*(undefined2 *)(unaff_EBP + -0x48) = 0x6f;
*(undefined4 *)(unaff_EBP + -0x6d) = 0x61647a7d;
*(undefined4 *)(unaff_EBP + -0x69) = 0x75643460;
*(undefined4 *)(unaff_EBP + -0x65) = 0x7b636767;
*(undefined4 *)(unaff_EBP + -0x61) = 0x342e7066;
*(undefined *)(unaff_EBP + -0x5d) = 0;
*(undefined4 *)(unaff_EBP + -0xc) = 0;
while( true ) {
    iStack152 = unaff_EBP + -0x6d;
    uVar1 = strlen();
    if (uVar1 <= *(uint32_t *)(unaff_EBP + -0xc)) break;
    *(uint8_t *)(*(int32_t *)(unaff_EBP + -0xc) + unaff_EBP + -0x6d) =
         *(uint8_t *)(*(int32_t *)(unaff_EBP + -0xc) + unaff_EBP + -0x6d) ^ 0x14;
    *(int32_t *)(unaff_EBP + -0xc) = *(int32_t *)(unaff_EBP + -0xc) + 1;
}
*(undefined2 *)(unaff_EBP + -0x70) = 0x7325;
*(undefined *)(unaff_EBP + -0x6e) = 0;
iStack148 = unaff_EBP + -0x6d;
iStack152 = unaff_EBP + -0x70;
printf();
uStack144 = 0x11;
iStack148 = 0;
iStack152 = unaff_EBP + -0x6d;
memset();
iStack148 = unaff_EBP + -0x46;
iStack152 = unaff_EBP + -0x70;
__isoc99_scanf();
piVar3 = (int32_t *)auStack136;
if (*(int32_t *)(unaff_EBP + -0x14) != 0x6c333374) {
    iStack152 = 0;
    exit();
    piVar3 = &iStack152;
}
*(undefined4 *)((int32_t)piVar3 + -8) = 10;
*(int32_t *)((int32_t)piVar3 + -0xc) = unaff_EBP + -0x5c;
*(int32_t *)((int32_t)piVar3 + -0x10) = unaff_EBP + -0x46;
*(undefined4 *)((int32_t)piVar3 + -0x14) = 0xf7f0210e;
iVar2 = strncmp(*(char **)((int32_t)piVar3 + -0x10), *(char **)((int32_t)piVar3 + -0xc),
                *(unsigned long *)((int32_t)piVar3 + -8));
if (iVar2 == 0) {
    *(undefined4 *)(unaff_EBP + -0x79) = 0x34617b6d;
    *(undefined4 *)(unaff_EBP + -0x75) = 0x3a7a7d63;
    *(undefined *)(unaff_EBP + -0x71) = 0;
    *(undefined4 *)(unaff_EBP + -0x10) = 0;
    while( true ) {
        *(int32_t *)((int32_t)piVar3 + -0x10) = unaff_EBP + -0x79;
        *(undefined4 *)((int32_t)piVar3 + -0x14) = 0xf7f0215c;
        uVar1 = strlen();
        if (uVar1 <= *(uint32_t *)(unaff_EBP + -0x10)) break;
        *(uint8_t *)(*(int32_t *)(unaff_EBP + -0x10) + unaff_EBP + -0x79) =
             *(uint8_t *)(*(int32_t *)(unaff_EBP + -0x10) + unaff_EBP + -0x79) ^ 0x14;
        *(int32_t *)(unaff_EBP + -0x10) = *(int32_t *)(unaff_EBP + -0x10) + 1;
    }
```

Figure 3: Dynamically generated code

```
#Other flags serve for shared-library compilation
gcc -fPIC -shared -m32 preload_strcmp.c -o preload_strcmp.so

LD_PRELOAD=./preload_strcmp.so ./BombsLanded 1 2 3 4
```

To my surprise this technique did yield the correct password. This got me wonder why there is different string in strncmp during debugging. As I foudn out, the system strncmp is called but its called from the program modified strncmp but their both names are exactly same, so the author of challenge known that the user will not bother checking the system functions and looks only on the non-familliar ones. The program decompiled strncmp is at Figure. So now we can see that the program strncmp is calling system strncmp but in case the string has length 10 it will be xored with 10 firstly and then compared. That was the reason why my input, which was same as it was checked againts was not working.

3 Simple Keygen

Object of this challenge was to generate correct password. This challenge was simple and pretty much straightforward. The only custom function called by main was function named usage. The decompiled usage fuction is on Figure.

So according the usage function, the given password has to meet two conditions. First is the length should be equal to 16 or 0x10 in hexadecimal. The second condition is that each character on even index had to have the ordinal value lower by one from its neighbouring character that right on the right side of him. So for example, the string 'abcdefghijklmnop' is good password as it has sufficient length and every character pair is meeting the condition that left character is one lower than right character in ordinal value. The ordinal difference between characters in python is shown below.

```
ord('a') - ord('b') == -1
ord('c') - ord('d') == -1
ord('e') - ord('f') == -1
```

According the conditions, the characters can repeat so the 'abababababababab' string is valid as well.

```
undefined4 strncmp(char *s2, char *s1, unsigned long arg_10h)
{
    code *pcVar1;
    int32_t iVar2;
    uint32_t uVar3;
    int32_t iVar4;
    undefined4 uVar5;
    undefined4 uVar6;
    int32_t var_18h;
    char *var_14h;
    int var_10h;
    int32_t var_ch;

    pcVar1 = (code *)func_0xf7d2bbc0(0xffffffff, 0x8048ca2);
    if (arg_10h == 10) {
        iVar2 = func_0xf7d4b9e0(s1);
        iVar2 = func_0xf7d41d00(iVar2 + 1);
        var_ch = 0;
        while( true ) {
            uVar3 = func_0xf7d4b9e0(s1);
            if (uVar3 <= (uint32_t)var_ch) break;
            *(uint8_t *)(iVar2 + var_ch) = s1[var_ch] ^ 10;
            var_ch = var_ch + 1;
        }
        iVar4 = func_0xf7d4b9e0(s1);
        *(undefined *)(iVar2 + iVar4) = 0;
        uVar5 = func_0xf7d4b9e0(iVar2);
        uVar5 = (*pcVar1)(s2, iVar2, uVar5);
        uVar6 = func_0xf7d4b9e0(iVar2);
        func_0xf7e21ed0(iVar2, 0, uVar6);
        func_0xf7d41fd0(iVar2);
    } else {
        uVar5 = (*pcVar1)(s2, s1, arg_10h);
    }
    return uVar5;
}
```

Figure 4: Program custom strncmp

```
undefined8 usage(char *arg1)
{
    int64_t iVar1;
    undefined8 uVar2;
    uint64_t uVar3;
    int64_t iVar4;
    int32_t iStack52;
    char *var_8h;

    printf("%s [SERIAL]\n", arg1);
    iVar4 = 0xffffffff;
    exit();
    iVar1 = strlen(iVar4);
    if (iVar1 == 0x10) {
        for (iStack52 = 0; uVar3 = strlen(iVar4), (uint64_t)(int64_t)iStack52 < uVar3; iStack52 = iStack52 + 2) {
            if ((int32_t)*(char *)(iVar4 + iStack52) - (int32_t)*(char *)(iVar4 + (int64_t)iStack52 + 1) != -1) {
                return 0xffffffff;
            }
        }
        uVar2 = 0;
    } else {
        uVar2 = 0xffffffff;
    }
    return uVar2;
}
```

Figure 5: Usage function

4 rop-obf

The object of this challenge was to find another correct input and the program should then print '1' instead of '0'. There is no function that could be properly decompiled in whole binary. This is due to natur of used obfuscated technique. The example of binary code is on Figure.

As the code suggets, the name stand for return-orietended programming (rop) obfuscation (obf). This means that obfuscation is based on using the `ret` assembler instruction, which pops the value from stack and put it to the instruction pointer which shows the next executed instruction. Firstly, the whole program is pushed into the stack in the entry function as shown in Figure and first `ret` instruction is starting it. The only system used system function is for the reading `STDIN` but nothing with string comparison as strcmp where I could use the LD PRELOAD technique mentioned in BombsLanded. I did tried few random inputs and the program took always 6 inputs, so there will probable be six comparisons.

So the input comparison should be done with `cmp` instruction. With the help of radare2 and its reference finding, there are six `cmp` instructions and only 3 of these are used because there are referenced by some `push` instruction. The mentioned `cmp` instructions and references are on Figure. I put breakpoints on the referenced `cmp` instructions and as I was stepping through code, the input was firstly xored with some value and then compared with correct input. So I knew what should be the correct output after xor then let some CSP solver do the work. The simple code below shows the

```
entry0 ();
0x565f01e0      mov dword [0x565f2008], esp
0x565f01e6      push 0x565f0141
0x565f01eb      push 0x565f011f
0x565f01f0      push 1              ; 1 ; char **envp
0x565f01f2      push 0x565f010b     ; char **argv
0x565f01f7      push main           ; 0x565f013b ; int argc
0x565f01fc      push 0x565effcb
0x565f0201      push 0x565effea
0x565f0206      push 0xf7daccf0     ; 0x565ef000
                                    ; ebp
0x565f020b      push 0x565f010b
0x565f0210      push 0x565effea
0x565f0215      push 0x565efffc
0x565f021a      push 0x565f010b
0x565f021f      push 0x565effea
0x565f0224      push dword [stdout] ; 0x565f20ac
0x565f022a      push 0x565f010b
0x565f022f      push 0x565effcb
0x565f0234      push 0x565effea
0x565f0239      push 0xf7d933d0     ; 0x565ef000
                                    ; ebp
0x565f023e      push 0x565f010b
0x565f0243      push 0x565effea
0x565f0248      push 0x565efffc
0x565f024d      push 0x565f010b
0x565f0252      push 0x565effea
0x565f0257      push 0x565f00d0
0x565f025c      push 0x565effea
0x565f0261      push 0x565f00ea
0x565f0266      push 0x565f00f1
0x565f026b      push 0x565f012b
0x565f0270      push 0xc            ; 12
0x565f0272      push 0x565f010b
0x565f0277      push 0x565f00e3
0x565f027c      push 0x565f0135
0x565f0281      push 0x565f0122
0x565f0286      push 0              ; 0x565ef000
```

Figure 6: rop-obf obfuscation

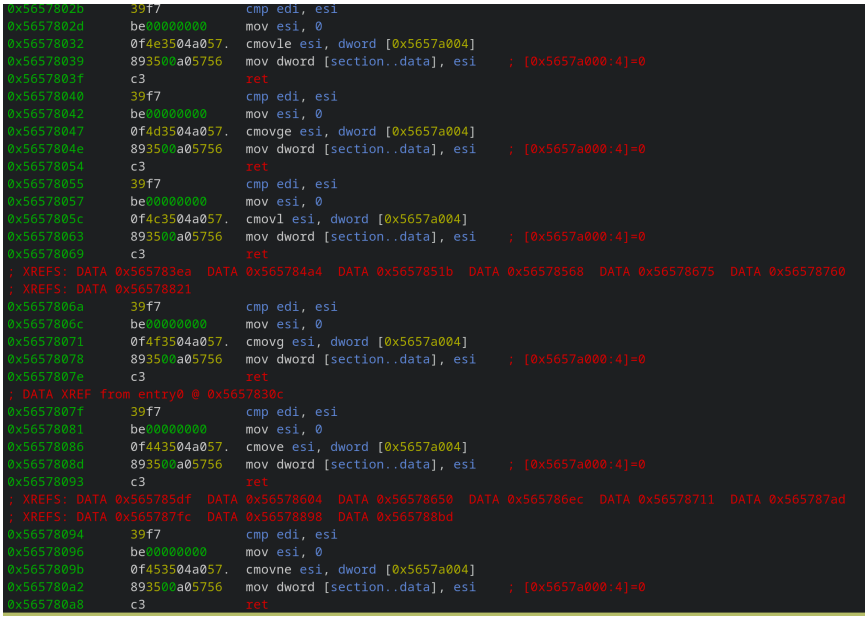

Figure 7: CMP instructions references

solution in python with help of z3-solver.

```
from z3 import *
s=Solver()
array=[BitVec("array%i"%i,32) for i in range(6)]

s.add(array[0]^0x83==0x87)
s.add(array[1]^0x36==0x3e)
s.add(array[2]^0x9d==0x92)
s.add(array[3]^0xcd==0xdd)
s.add(array[4]^0xec==0xfb)
s.add(array[5]^0xf6==0xdc)

if s.check()==sat:
    flag=s.model()
    for i in range(6):
        ans=((flag[array[i]]))
        print(ans)
```

And with help of script we have our solution as 4,8,15,16,23,42 which

prints ’1’.