The x86 police

Author:
Matteo Bordignon, 27/04/2024

Background

Software Reverse Engineering (SRE) 1-2 is the practice of understanding the design, functionality and how the software works in general, starting from the binary code (executable) and without having the source code.
SRE typically involves using different techniques such as decompilation, disassembly and, code analysis (static and dynamic 3-4). Reverse engineering can be employed for various purposes such as: understanding legacy software, finding bugs and vulnerabilities, replicating functionalities, modifying or adding features, ensuring interoperability with other systems, bypass verification, etc.
Usually participants to Reverse Engineering CTF are provided with an executable file. The goal is to understand the functionality of a given program such that you can identify deeper issues 5.

Vulnerability

The program takes in input a string then calls a function which checks if each character of the provided string xored with `0x42` (byte which is the character *B*) is equal to the corresponding character of the xored flag. If the provided input is the right flag the program will print *"Congrats, go submit that flag!"* else it will print *"Nope, try again!"*

The anti-reverse techniques employed to make this program harder to reverse engineer are:

- stripping: strip all symbols, function and variable names making the code difficult to read and understand;
- anti-disassemble 6 : uses specially crafted code or data in a program to cause disassembly analysis tools to produce an incorrect program listing;
- shared library pre-loading;
- flag obfuscation using xor operation.

Solution

To solve this challenge we'll make use of Ghidra which is a decompiler.

1. To find the flag, it is necessary to first identify the main function, to do that we can simply search for the function that is called by `_libc_start_main`. In our case the main function is `FUN_0010121d`. Once identified the main function we can start to try to understand how the program works:

```
1  undefined8 FUN_0010121d(void){
2    int iVar1;
3    long in_FS_OFFSET;
4    sigaction local_a8;
5    long local_10;
6
7    local_10 = *(long *)(in_FS_OFFSET + 0x28);
8    local_a8.__sigaction_handler.sa_handler = FUN_00101179;
9    local_a8.sa_flags = 4;
10   iVar1 = sigaction(4,&local_a8,(sigaction *)0x0);
11   if (iVar1 != -1) {
12     do {
13       invalidInstructionException();
14     } while( true );
15   }
16   if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
17                     /* WARNING: Subroutine does not return */
18     __stack_chk_fail();
19   }
20   return 1;
21 }
22
```

We can observe that the function `FUN_00101179` is the signal handler. Before checking what `FUN_00101179` does, let's observe the lines 11-15, since the decompiled code is not clear let us see the corresponding assembly code:

```
1          00101265 83 f8 ff        CMP        EAX,-0x1
2          00101268 75 07           JNZ        LAB_00101271
3          0010126a b8 01 00        MOV        EAX,0x1
4                   00 00
5          0010126f eb 65           JMP        LAB_001012d6
6                                LAB_00101271
   XREF[1]:     00101268(j)
7          00101271 0f 0b           UD2
8          00101273 48              ??         48h    H
9          00101274 8d              ??         8Dh
10         00101275 05              ??         05h
11         00101276 d2              ??         D2h
12         00101277 0d              ??         0Dh
13         00101278 00              ??         00h
14         00101279 00              ??         00h
15         0010127a 48              ??         48h    H
16         ...
17
```

It seems that Ghidra has not disassembled this code properly. We can force Ghidra to disassemble this snippet highlighting the code and then `right click > Disassemble 64-`

`bit x86` . The code which corresponds to the new disassembled code is much more clear:

```
void UndefinedFunction_00101273(void)

{
  printf("Gimme the flag to check: ");
  __isoc23_scanf(&DAT_00102066,&DAT_00104060);
  do {
    invalidInstructionException();
  } while( true );
}

```

We can observe that is asked to the user a flag to check and the input is assigned to the global variable `DAT_00104060` .

2. The flag check is handled with the function `sigaction` that is a system call used to change the action taken by a process on receipt of a specific signal [7]. We've seen before that the `sigaction` handler is the function `FUN_00101179` . Let's see what this function does:

```
void FUN_00101179(undefined8 param_1,undefined8 param_2,long param_3)

{
  byte *local_20;
  byte *local_18;

  DAT_001040a0 = 1;
  local_20 = &DAT_00104060;
  local_18 = &DAT_00102020;
  while( true ) {
    if (((*local_18 == 0) || (*local_18 == 10)) || (*local_20 == 0))
goto LAB_001011f8;
    if ((*local_20 ^ 0x42) != *local_18) break;
    local_20 = local_20 + 1;
    local_18 = local_18 + 1;
  }
  DAT_001040a0 = 0;
LAB_001011f8:
  *(long *)(param_3 + 0xa8) = *(long *)(param_3 + 0xa8) + 2;
  return;
}

```

It is likely that Ghidra has misinterpreted the highlighted lines, the corresponding assembly code looks much more like a for loop implementation than a while cycle, however we can observe that the variable `local_20` contains the user input and user_input xored with 42

must be equal to the content of the variable `local_18` , so we can deduce that inside `local_18` xored flag is stored.

3. Now we just have to retype the variable `local_18` . To do that follow this step: highlight the variable `> right click > retype variable > type "char *" > press enter` . Then to see the flag: highlight the variable content `Display Script Manager > search for "xor" > select the script > click on "run script" > type "42"` . Now we are able to see the flag. We used the following xor property: $a \oplus b = c \iff a = c \oplus b$

```
1  void FUN_00101179(undefined8 param_1,undefined8 param_2,long param_3)
2
3  {
4    byte *local_20;
5    char *local_18;
6
7    DAT_001040a0 = 1;
8    local_20 = &DAT_00104060;
9    local_18 = "UniTN{its_n0t_i11ega1_till_they_catch_y0u!}BGimme the flag
   to check: ";
10   while( true ) {
11     if (((*local_18 == '\0') || (*local_18 == '\n')) || (*local_20 ==
   0)) goto LAB_001011f8;
12     if ((*local_20 ^ 0x42) != *local_18) break;
13     local_20 = local_20 + 1;
14     local_18 = local_18 + 1;
15   }
16   DAT_001040a0 = 0;
17 LAB_001011f8:
18   *(long *)(param_3 + 0xa8) = *(long *)(param_3 + 0xa8) + 2;
19   return;
20 }
21
```

References

1. Software Reverse Engineering according to chat gpt
2. Reverse Engineering wikipedia
3. Static Code Analysis
4. Dynamic Analysis
5. What is a reverse engineering ctf?
6. Anti-disassamble
7. Sigaction man