The x86 Police

Luigi Dell'Eva, 21/04/2024.

Background

The challenge consists in analyzing an ELF binary file to find a way to retrieve the flag. When executed the file asks to enter the flag and then it prints if it is correct or not.

Reverse engineering, is a process through which one attempts to understand how a process, system, or piece of software accomplishes a task with very little insight into exactly how it does so. This process is usually done to perform security analysis, extract sensitive information, remodel obsolete objects or to understand how a software works. [1]

More specifically, **static reverse engineering** is a method of analyzing software without executing it. This approach involves examining the code and structure of a software, which can include analyzing the source code if available, or examining the binary executable using disassemblers or decompilers. [2]

Its impossible to completely prevent reverse engineering, since the processor must receive the instruction to execute, but it is possible to make it more difficult. Some techniques include **stripping** which consists in *strip* all the symbols such as function names, global variable names, etc (to strip shared object symbols we have to statically link the shared libraries) or **anti-disassembly** which manipulates software such as Ghidra or IDA, by inserting bytes that exploit bygs or limitation in their functionalities. These bytes will not affect the program execuion but can mislead the disassemblers and others techniques such as **packing**, **virtual machine obfuscator**, etc. [3]

Vulnerability

Analysing the provided binary file with Ghidra, we can see that in the `main` function the program assign to a `sigaction` struct to handle a function that I will call `check_flag`. After this there is an infinite loop with the `invalidInstructionException()`. Selecting this function, Ghidra hilights the `UD2` assembly instruction. This instruction as stated in [4] "generates an invalid opcode execution. This instruction is provided for software testing to explicitly generate an invalid opcode exception. The opcodes for this instruction are reserved for this purpose". When Ghidra runs into this instruction stops disassembling and marks the rest of the code as `??`, since it expect that the program will terminate after receiving a `SIGILL`.

However, proceeding into disassembling the code by ourself we can see that what was marked as `??` is actually code. Now, patching the two occurrences of `UD2` with `NOP` it is possible to see that Ghidra automatically disassembles the code replacing the infite loop.

The program works even if the `UD2` instruction is present, because it is built to handle the `SIGILL` signal that is generated by this instruction. When the sigaction catch the signal, it calls the `check_flag` function that checks if the flag is correct and then exits the program.

Solution

Looking into the fucntion `check_flag` we can see that the program check if the flag is correct by XORing each character of the input provided by the user with `0x42` and then comparing it with the flag hardcoded in the binary at the location `DAT_00102020` and `DAT_00102021` which are located into the .rodata section.

Once identified the flag location I can retrieve its byte values by using the *copy special* features of Ghidra copying as "Python byte string" and then use the following script to convert the byte string to the flag:

```python
enc_flag =
b'\x17\x2c\x2b\x16\x0c\x39\x2b\x36\x31\x1d\x2c\x72\x36\x1d\x2b\x73\x73\x27\x25\x23\x73\x1d\x36\x2b\x2e
\x2e\x1d\x36\x2a\x27\x3b\x1d\x21\x23\x36\x21\x2a\x1d\x3b\x72\x37\x63\x3f\x00'

flag = "".join([chr(x ^ 0x42) for x in enc_flag])

print(flag)
```

References

[1] https://en.wikipedia.org/wiki/Reverse_engineering

[2] https://cyberw1ng.medium.com/unveiling-the-hidden-an-introduction-to-the-fundamentals-of-reverse-engineering-karthikeyan-89b39ec4d0bb

[3] Ethical Hacking course slides

[4] x86 Instruction Set Reference: https://mudongliang.github.io/x86/html/file_module_x86_id_318.html