Background

The challenge is a reverse engineering exercise that focuses on retrieving the flag from an ELF binary file [5].

Binary reverse engineering [4] is the process of analyzing compiled software to understand how it functions without access to its source code. By examining the binary code, analysts can deduce the underlying algorithms, identify vulnerabilities, and repurpose or modify the software. This technique is essential for security assessments, malware analysis, and gathering insights into proprietary systems.

While complete prevention of reverse engineering is impossible, several techniques can complicate the process for attackers. One common approach is stripping, which removes symbols such as function names and global variable names, obscuring the code's intent [6].

Another method involves anti-disassembly techniques, where specific bytes are inserted to exploit limitations in disassemblers (e.g., Ghidra). This manipulation can mislead analysis tools without affecting the program's execution [1].

Additional methods include code packing, which compresses or encrypts the code to make analysis more difficult [2], and virtual machine obfuscation, where the code is transformed into a format that runs on a custom virtual machine [3].

Reversing

The binary uses several techniques to make reverse engineering more difficult:

Anti-Disassembly The UD2 instructions trigger SIGILL exceptions, causing disassemblers such as Ghidra to misinterpret the following bytes as data.

Signal Handling Obfuscation A custom SIGILL handler, established using a sigaction structure, redirects execution to the flag-checking routine. This method hides the actual control flow, making it harder to determine which address holds the user input and which the flag itself.

Stripped Symbols The removal of function names and variable identifiers requires more in-depth manual analysis, as no symbol information is provided to indicate the binary's structure.

Solution

The program when run outputs `"Gimme the flag to check:"`. After disassembling the binary with Ghidra, I searched for that string in but could not find it. Since it is a stripped binary, I could not directly jump to `main`, but finding it was possible via the `entry` function that calls `__libc_start_main`, containing the address of `main`.

In the `main` function, an `invalidInstructionException` function is called, that is obfuscated via UD2 instruction. Ghidra did not disassemble the code, but it can be forced to do so. It is in this disassembled function that the program reads the user input, and I found out that the user input is saved inside `DAT_00104060`.

Back in the `main` function, the function pointer at `FUN_00101179` is assigned to a `sigaction handler`. In that function, each byte of the user input XORED with `0x42` is compared with a stored variable, presumably the flag. By copying the bytes inside the variable with *Copy Special* → *Python Byte String*, the contents of the flag can be retrieved. Listing 1 outputs the flag.

```
enc_bytes = 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(a ^ 0x42) for a in enc_bytes)
print(flag)
```

Listing 1: Solution code

References

[1] ReasonLabs. Anti-disassembly. https://cyberpedia.reasonlabs.com/EN/anti-disassembly%20techniques.html.

[2] ReasonLabs. Code packing. https://cyberpedia.reasonlabs.com/EN/code%20packing.html.

[3] ReasonLabs. Virtual machine obfuscation. https://cyberpedia.reasonlabs.com/EN/virtual%20machine-based%20obfuscation.html.

[4] Wikipedia. Binary reverse engineering. https://en.wikipedia.org/wiki/Reverse_engineering#Binary_software.

[5] Wikipedia. ELF. https://en.wikipedia.org/wiki/Executable_and_Linkable_Format.

[6] Wikipedia. Stripping. https://en.wikipedia.org/wiki/Strip_(Unix).