Background

Reverse Engineering [1] is the process of analyzing a system to identify its components and their relationships, often with the goal of understanding how it works without having access to its source code or design documentation. In the context of binary executables, this typically involves inspecting compiled code to recover high-level logic, using tools like disassemblers, debuggers, and decompilers.

In real-world applications, reverse engineering has several legitimate uses, including malware analysis, vulnerability research, compatibility checks, and legacy software maintenance. However, it can also be misused for malicious purposes like software cracking or intellectual property theft, which is why many binaries attempt to hide or obfuscate their logic.

A particularly effective technique used both in malware and commercial software protection is self-modifying code [4]. Instead of executing a static sequence of instructions laid out in the file, a self-modifying program alters its own code segment at runtime by writing new bytes into memory, decrypting sections on the fly, or patching jump targets. This layer of indirection can:

- Tamper static analysis: Disassemblers see only the encrypted or uninitialized bytes, so the true logic remains concealed until execution.
- Detect tampering: If a watchpoint or breakpoint disrupts the timing or integrity of self-modification, the program can crash or take alternative (often misleading) paths.
- Implement polymorphism: Each run can decrypt or re-encrypt itself differently, making signature-based detection (e.g., by antivirus) far more difficult.

Another common anti-analysis measure is ptrace-based anti-tracing. On Unix-like systems, ptrace is the kernel utility that allows one process (a debugger) to observe and control another [3]. Programs can abuse ptrace to detect if they are being debugged by programs like gdb [2] or strace [5].

Reversing

The binary appears to implement a form of self-modifying code. A non-standard section named `mysec` contains code that plays a central role in the program's behavior, but it's initially obfuscated, since it is composed of useless or bad instructions.

During dynamic analysis with `gdb`, memory watchpoints were set on `mysec`, revealing that it was frequently modified at runtime, supporting the hypothesis of self-modifying behavior. The program also exhibited different crash signals (SIGILL or SIGSEGV) when incorrect inputs were provided (caused by the bad instructions that are executed if a wrong input is inserted), suggesting that the correctness of the input influences the execution path and potentially decrypts the code progressively. Moreover, a decryptor function that performs a xor over the `mysec` section is present in the code, and is invoked with the value 0x42, which appears to be part of an initialization routine. After decryption, different parts of the code in `mysec` become readable and contain logic that compares user input against hardcoded values.

Dynamic Solution

Since the challenge is called *Morphing Code Mystery*, it most likely had something to do with self-modifying code. Firstly, I checked the sections in the binary with `readelf -S ./bin` and saw `mysec`, which is not a default one, with offset `0x12a8`. After setting a breakpoint for memory changes at `mysec` with `watch*(int*)$mysec`, when executing the program the breakpoint was hit multiple times, confirming the suspicion of self-modifying code [1]. When inserting a string as input for the program,

[1] In order to trace the program I patched it with ghidra by removing the call to `ptrace`. With `LD_PRELOAD` it was hanging on gdb (still have to find out why).

it either went into a SIGILL or SIGSEGV exception, thus I created a gdb script to dump the contents of the disassembly of mysec to a file after each exception [Listing 1].

I got the values of the base address in memory with info proc mappings in gdb, and calculated the address of mysec with $base + 0x12a8. By executing gdb -x ./setup.gdb the script in Listsing 1 is used. Once inside the gdb shell, run the program that will ask for the flag. After inserting a string, the program will go into an exception, and the dumped disassembly will be found in last-char.txt.

By analyzing the assembly instructions, some cmp comparisons are made. In particular, in Listing 2, the cmp al,<hex> is the instruction that compares the input with some hardcoded hex value. If the values are the same, a call to a function is made, that is excatly the $decryptor function which performs xor operation on the whole mysec section.

To find the flag, simply run inside gdb with a string that contains the hex values found in each disassembly dump of the cmp al,<hex> instruction.

Static Solution

After disassembling the binary with ghidra, I found the main function via __libc_start_main. Here, a scanf gets the user input and passes it to a function, that resides in the mysec section. Moreover, by looking at all the functions found by ghidra (accessible via Window → Functions), the $decryptor function can be found, which performs a xor operation of the whole mysec section with a byte value given as input. Via the References in ghidra, I saw that this is called in an initializer function with the byte 0x42.

To solve the challenge, I selected the whole mysec section, xored it with 0x42 and analyzed the disassembly of the section. After this first xor, a call to the decryptor function is made some instructions after a comparison with the al register. In particular, after xoring with 0x42, the first comparison that is made is with the byte 0x55 which is the U letter, hinting at the first character of the flag.

I then iteratively xored the mysec section with the bytes found in cmp al,<hex>, disassembled the new xored section and searched for the new hex value, corresponding to a byte of the flag. This solution is also found in Listing 3.

```
file ./patched-bin

set $base = 0x0000555555554000
set $mysec = $base + 0x12a8
set $decrypt = 0x555555555189

# analyze the disassembly and get the next character
catch signal SIGSEGV SIGILL
commands
    set logging file last-char.txt
    set logging overwrite on
    set logging enabled on
    x/500i $mysec
    set logging enabled off
end
```

Listing 1: GDB setup

```
   ...
   0x5555555552b0:  mov    QWORD PTR [rbp-0x8],rdi
   0x5555555552b4:  mov    rax,QWORD PTR [rbp-0x8]
   0x5555555552b8:  movzx  eax,BYTE PTR [rax]
   0x5555555552bb:  cmp    al,0x55                 % comparison with the input
   0x5555555552bd:  je     0x5555555552c9          % jump below
   0x5555555552bf:  mov    eax,0x0
   0x5555555552c4:  jmp    0x5555555555fd
   0x5555555552c9:  mov    rax,QWORD PTR [rbp-0x8]
```

```
    0x5555555552cd:  movzx   eax,BYTE PTR [rax]
    0x5555555552d0:  movsx   eax,al
    0x5555555552d3:  mov     edi,eax
    0x5555555552d5:  call    0x555555555189          % call to $decryptor
    ...
```

Listing 2: Disassembly dump

```python
from pwn import disasm

def disassemble_and_xor(binary_string):
    modified_data = bytearray(binary_string)
    hex_values = []
    while True:
        instructions = list(disasm(binary_string).splitlines())
        cmp_found = False
        for i, line in enumerate(instructions):
            if 'cmp    al, 0x' in line: # instruction containing flag char
                if 'je ' in instructions[i + 1]:
                    hex_value = int(line.split("al,")[1].strip(), 16)
                    if hex_value == 0x3c: continue
                    hex_values.append(hex(hex_value))
                    hex_string = ''.join(h[2:] for h in hex_values)
                    print(bytes.fromhex(hex_string).decode('utf-8', errors='ignore'))
                    modified_data = [(a ^ hex_value) for a in modified_data]
                    cmp_found = True
                    break

        if not cmp_found:
            break

        binary_string = bytes(modified_data)

    return binary_string, hex_values


# bytes of mysec section copied as python string from ghidra
binary_string = open('./mysec.bin', 'rb').read()
initial_bytes = bytes((a ^ 0x42) for a in binary_string)

_, flag = disassemble_and_xor(initial_bytes)
hex_string = ''.join(h[2:] for h in flag)
res = bytes.fromhex(hex_string).decode('utf-8', errors='ignore')


# UniTN{m0rph1ng_x0r}
```

Listing 3: Solution code

References

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

[2] Wikipedia. GDB. https://en.wikipedia.org/wiki/GNU_Debugger.

[3] Wikipedia. Ptrace. https://en.wikipedia.org/wiki/Ptrace.

[4] Wikipedia. Self Modifying Code. https://en.wikipedia.org/wiki/Self-modifying_code#Use_as_camouflage.

[5] Wikipedia. strace. https://en.wikipedia.org/wiki/Strace.