Nicola Mores, 25/04/2024

Challenge: The x86 police

Background

This challenge is about Reverse Engineering, a technique in which an attacker can try to understand how a system or a program works, without having direct access to the high-level code it uses, but only to the ELF file it produces. In real life this technique can be used in order to produce malware, modify and crack softwares, by bypassing the licensing mechanisms.

Vulnerability

In this challenge we encounter some technique used to make things harder for the attacker: Obfuscation, a technique that hides function and variable names, changing them with random ones (for example, from main to FUN_0010121d). Also, in the instructions used by the program we can notice many UD2 that cause Gihdra to not decompile some of the following instructions. Due to this, we could say that some anti disassembly techniques are also applied to this program. Both of these techniques complicate our reasoning.

Solution

In order to complete this challenge we can try to use a decompiler, like Ghidra, in a way to reason about how the software works by looking at one of its possible decompiled code. Once we have disassembled and analyzed the program with Ghidra we can notice how there isn't a clear main function, due to the obfuscation applied to the file, so we first need to locate it manually. This can be achieved through the entry function, that usually contains the call `__libc_start_main` in which the first parameter is the main function (1). In the main we can observe the line with `iVar1 = sigaction(4,&local_a8,(sigaction *)0x0)` (2) which uses the sigaction struct defined by the variable `local_a8` in order to pass the control to a particular function whenever the process will receive a SIGILL signal (identified by the number 4). After that we can see the function call `invalidInstructionException()`, generated by the assembly instruction UD2 (3), that is used in order to raise an invalid opcode exception (SIGILL). Under normal conditions this should terminate the execution of the program, which is why the code following this line isn't automatically decompiled. However, in our case, we will handle this exception by using the previously mentioned sigaction function. Noticing this, we can try to decompile the unprocessed lines, revealing two functions used in order to ask and read for a flag to check and to give us feedback about its correctness, in particular by checking whether the variable `DAT_001040a0` is equal to 1 (correct) or 0 (wrong flag). Having identified where the program reads our input and where it tells us if the provided flag is correct, we only need to find where the actual check is done. We said that in order to provide us feedback, the program checks the value of a variable, so we might examine the code that actually modifies its value, and see how this is done in the same function used by the `sigaction` method. Exploring this new function we can see how we are working with two addresses (one of them being the one in which we save the input read from the user), comparing their content and repeating (after shifting both addresses of one position) the same operation until the two differ or until the content of one of them is equal to some specific values. Since we are XORing our input with 0x42 and comparing it with another value and noticing how the program modify the variable `DAT_001040a0`, we can imagine that this other value is the flag. We can give it a try by XORing it with 0x42 directly from Ghidra and thus revealing the actual flag of this challenge.

[1] __libc_start_main Documentation:
https://refspecs.linuxbase.org/LSB_3.1.1/LSB-Core-generic/LSB-Core-generic/baselib---libc-start-main-.html
[2] sigaction Documentation:
https://man7.org/linux/man-pages/man2/sigaction.2.html
[3] UD2 Documentation:
https://www.felixcloutier.com/x86/ud