Obfuscation

Jack Ford

Overview

- Obfuscation is the process of modifying a binary to slow down or prevent analysis
- Started by reading "Practical Malware Analysis" by Michael Sikorski and Andrew Honig
- Researched and downloaded obfuscated crackmes to practice analyzing obfuscation methods
- Most malware is written for windows, but I did all my analysis on linux binaries because, even though the book gave me a lot of windows knowledge, the amount of time it would've added just working on binaries with a new OS would've been detrimental
  - However, the concepts are the same
- The primary strategy used when analyzing binaries is to work backwards
- Useful operations to look for when looking at assembly are calls and jmps

Difficulty criteria

- Variety
  - Methods with multiple different ways to perform their obfuscation techniques have a higher degree of difficulty
    - e.g. there are a lot of packers and you can even make your own
- Rigor
  - How much effort it takes to perform the reverse engineering
- Prior knowledge required
- Uniqueness
  - The degree to which each method you encounter is something you haven't seen before

Types of obfuscation (easiest to hardest)

1. Encryption
2. Anti-disassembly
3. Self-modifying code
4. Packers
5. Anti-debug

Summary

- Anti-debug
  - Very difficult
    - Requires a great understanding of the operating system
    - Primarily ptraces used in linux, but I'm almost certain there are other methods
    - Can be combined with other methods to make this more difficult
- Packers
  - Simple in concept, but difficult in execution
    - Find the OEP (original entry point) and dump that code
- Self-modifying code
  - The one method I encountered was where the hexadecimal representation of the assembly code was stored in memory with the code and at execution, placed at the desired address
- Anti-disassembly
  - Simple
    - Just need to modify how the bytes are disassembled in ghidra (D and C hotkeys)
- Encryption
  - Simple
    - Most common method was xor, but the book also included base64 as a common method

Types of obfuscation (easiest to hardest)

1. Encryption
2. Anti-disassembly
3. Self-modifying code
4. Packers
5. Anti-debug

Exatlon_v1

- Followed a medium tutorial to understand how to manually unpack UPX
  - UPX (Ultimate Packer for eXecutables) is probably the most used packer for executables
- Started by opening the binary in ghidra
  - Quickly learned this is not helpful at all
- Transitioned to the test binary to try and learn from the tutorial, but not copy and paste the answers from it
  - You'll notice this becomes a trend
    - Helpful in determining what different code sections look like in binary (ptrace, WIFSTOPPED, etc.)

test

- Made a hello world program to understand how UPX works
- Learned many gdb commands that were useful in analyzing this binary
- To find the OEP, the tutorial hypothesized that the system call to munmap would be where the original binary is stored
  - Catching this syscall in gdb, I was able to find the magic bytes and exact replica of the assembly code for the main function
- The primary difficulty with this one once I got everything dumped was that the dumped version had an additional 0x1000 bytes of 0x00 in the middle that I had to use dd to remove with the help of ChatGPT
  - Only discovered these bytes by comparing the original binary to the dumped one
  - Other binary dumps did not have this issue
    - I would guess it happened to this one because of its small size
  - Was only able to find this with "readelf –d"
  - Along with the dynamic symbol table, other important aspects to linux binaries are the interpreter and the linked files (found using 'ldd')

Unpackme-upx (example)

- Analyzed the strace of the binary to evaluate where the unpacking might take place
- Opened the executable in gdb and set a catchpoint at the munmap system call
  - starti
  - catch syscall munmap
  - c (twice)
- Looked at the memory mapping of the process to try to determine which addresses contained the unpacked binary
  - info proc mapping
- Evaluated different addresses to see what is stored there
  - x/10w 0x400000
  - x/10i 0x400c60 or whatever
- Dumped the necessary addresses
  - dump binary memory dump1 0x400000 0x401000
- Cat the sections then chmod +x and it should be the same binary, but now it can be statically analyzed with ghidra and the flag can be found much easier

Packers

- DetectItEasy is a great tool for reading binaries that can tell you if a binary is packed by a known packer
- UPX seems to pack everything the same way
  - munmap before the tail jump
- Many different methods of packing
  - Another one covered in the book is where a pushad occurs to push all the registers on the stack and then a popad occurs just before the tail jump
    - To reverse engineer this, simply find the pushad, then set a breakpoint at the popad and the tail jump should be right there
- Readelf is useful in checking that you have dumped the correct sections for the original binary
  - Useful flags include h, d, and S

Types of obfuscation (easiest to hardest)

1. Encryption
2. Anti-disassembly
3. Self-modifying code
4. Packers
5. Anti-debug

cr4ckme

- Comments helped with this one
- The anti-debug trick is just a call to ptrace
  - ptrace is a linux system call that allows a process to trace another process or trace the execution of itself
  - Simply change the value returned by the function with the ptrace inside so that the code continues execution seamlessly
  - Besides this ptrace, the binary can pretty much be statically analyzed with ghidra to find the correct control flow to get you where the flag is revealed

Sh4ll4

- anti-debug technique using 2 ptrace calls
  - one is in the function called "function" and the other is before the password check
    - the first one changes the length of the actual password, making it impossible to find the actual password without getting past/around this ptrace call
    - the second one, I believe, is only for anti-debugging because it does not seem to occur when I run the program with strace
      - if the call to ptrace returns unfavorably, the code jumps into an infinite loop because it found a debugger attached
  - getting around both of these can reveal the original password in plain text following an ltrace, but that could require patching
  - essentially, the first one changes the length of the password so that it will always be wrong (19 instead of 20)
  - there is a stub later on in the code (looking in ghidra or ida) that appends an 'S' to the end of the string, which would make it go from 19 to 20
    - therefore, if you take the password that it checks against in an unpatched binary and add an 'S', it will be correct

Types of obfuscation (easiest to hardest)

1. Encryption
2. Anti-disassembly
3. Self-modifying code
4. Packers
5. Anti-debug

Nano (example)

- Noticed jz followed by jnz in ghidra which essentially means unconditional jump
  - This creates obfuscated assembly because ghidra disassembles code line by line instead of by following the control flow
- Rewrote bad instructions as data and then rewrote good instructions (usually one byte later) as assembly code
  - can use C and D hotkeys to do this
    - it's inverse in ida and ghidra (D = decode in ida, but data in ghidra)
- Patched these bytes in ghidra with NOP instructions to clarify the decompile view
- There is a function that calls ptrace instead of calling it from the main function
- The linux process completion status functions are not decompiled by ghidra
  - Needed to create a test file to compare a guess (WIFSTOPPED) to the actual
- Essentially, there is an instruction in the child process that sends a SIGSEGV signal that lets the parent function change %r12 (where the key is stored)
  - This allows the encryption with xor to produce a different result
- No further analysis is required because the new key can just be xor'd with the flag

Anti-debug

- Ptrace to check for a debugger (similar to isDebuggerPresent on windows)
- Ptrace to change the control flow of the running process
- Use of process forking and multiple calls to ptrace
  - Child process controls the actual execution of the process
  - Child process data gets modified by the parent process for further obfuscation
- Could be other OS methods to implement anti-debugging that I did not encounter and could be native to other operating systems than linux

Types of obfuscation (easiest to hardest)

1. Encryption
2. Anti-disassembly
3. Self-modifying code
4. Packers
5. Anti-debug

Facets (example)

- Was trying to analyze a different binary (crackme_v0.2.666) when I looked up a tutorial for self-modifying code and found this binary
- Ghidra is of minimal use because the instructions get placed at their proper addresses at runtime by the code itself
  - The binary sets a register (rdx) with rip to get the next instruction's address
  - It then uses mov instructions to set rdx+some_offset that correlates to the proper address where it wants to load the correct instruction with the correct instruction in hexadecimal
- Even after understanding this, the binary is still hard to analyze
  - There are two assembly instructions I have never seen before
    - cmpw $0x7, 0x4(%esp)
      - Compare word
    - cmove %cx,%bx
      - Conditional move if equal
  - The two possible return values are stored in cx and bx and for the desired return value to be given back to the main funciton, the input needs to be 7

STEEL CORGI

- Referenced the writeup to try to get started and also understand the program
  - Mentioned that the program is packed, so I try running it in DIE, but come up with nothing
- Analyzing it in ghidra, I find a normal looking start function, so I'm confused why it's called packed
- In an attempt to manually unpack the process, I try running it in gdb, but get a segfault
  - The program will not run in the current environment
- I believe I was able to find the OEP, but I don't know how to dump the process without gdb
- Ultimately, it seems that the program needs an environment variable to be set with an unpacking key by the APT group, which I do not have, so analyzing this malware does not seem feasible

Difficulties

- Finding binaries
- Deciding which methods of obfuscation to attempt to reverse engineer
- Working with linux in general (syscalls and dynamic linking)

Conclusions

- Self-modifying code is the most interesting technique that might be useful in creating stronger malware
- The uniqueness of packers is what makes them difficult
- Anti-debugging requires vast knowledge and understanding of the operating system
- Dynamic analysis is more useful than static analysis, but static analysis with ghidra is useful in understanding larger portions of code
- Some obfuscation methods prevent static analysis
  - Packers
  - Anti-disassembly
  - Encryption
- The tougher methods are those that prevent dynamic analysis because it usually means that static analysis will not help much, but dynamically analyzing the code as it changes or reading instructions that you don't know (because you don't have a great understanding of the operating system) can be very tiresome

Future plans

- Work on and develop de-obfuscation techniques for more packers
- Continue finding new methods for anti-debugging and self-modifying code
- Start working on windows binaries and actual live malware samples
  - Vxunderground and hybrid-analysis are great resources for finding malware