FinSpy VM Unpacking Tutorial Part 3: Devirtualization — Möbius Strip Reverse Engineering

By Rolf Rolles

Published: 2018-02-21 · Archived: 2026-04-06 01:29:35 UTC

1. Overview

This is the third and final part in my series on statically unpacking the FinSpy VM. After having deobfuscated the x86 implementation of FinSpy in part one and after having analyzed the VM and written a disassembler for the bytecode format for the particular sample in question in part two, we are now tasked with reconstructing x86 code corresponding to the pre-virtualized code. As before, the files are in the GitHub repository here. Of note:

- The IDB with the devirtualized code added, and mostly analyzed: here
- The devirtualization program: here
- FinSpy VM bytecode disassembly listings with various simplifications: here
- Binary files for the devirtualized code: here

Because the FinSpy VM is a weak and ineffective software protection, unpacking it is not very difficult. Over half of the VM instructions in the bytecode program for the sample we're analyzing already contain raw x86 machine code. It turns out that FinSpy only truly virtualizes a handful of x86 instruction types, and that simple pattern-matching is effective in reconstructing the original code. It's perhaps the easiest software protection I know of worthy of being called a "virtualization obfuscator" -- and that makes it good practice for the difficult ones.

As with part one, I am attempting to write this in a walk-through tutorial style. We start from first principles in inspecting the FinSpy VM bytecode disassembly listing, and show all of the steps (along with the code I wrote) along the way. Hopefully I've managed to capture the process of observation, evaluation of design considerations, actions taken including mis-steps, and above all, the iterative, trial-and-error nature of the affair.

2. Organizational Overview

This part ended up being longer and more difficult to write than expected. In fact, writing this document as a tutorial was considerably more difficult than doing the work in the first place. I hope the effort pays off in terms of educational value. Because I personally don't enjoy huge documents whose sections aren't well-segregated from one another, I have decided to split the devirtualization process, and also this document, into a number of "phases", each in its own individual blog entry. All entries are currently online and available for reading; links to them are immediately below. Each individual part links to the code and binary artifacts used in that phase.

Here are the contents of each phase in brief, along with links to the individual parts:

1. Part #3, Phase #1: analyzing and deobfuscating FinSpy VM bytecode programs. In part two, we analyzed the FinSpy VM and its instruction set, and ultimately wrote a disassembler for FinSpy VM programs. This

phase begins by reviewing the FinSpy VM instruction set. Next, we work from first principles in analyzing the FinSpy VM bytecode program. First we add a useful feature to our FinSpy VM disassembler. Next, we analyze a set of VM instructions -- group #2, in the parlance -- used by the VM bytecode program for obfuscation purposes. We determine several patterns, and write code to detect and simplify instances of those patterns within FinSpy VM bytecode programs. After one more small tweak to the disassembler, the FinSpy VM bytecode program is now ready to be devirtualized back into x86 machine code.

2. Part #3, Phase #2: initial devirtualization. The previous phase left the FinSpy VM bytecode program in a suitable form for devirtualization. This phase begins by discussing the theory behind devirtualizing the individual FinSpy VM instructions. Next, we write a tool to devirtualize FinSpy VM bytecode programs -- this tool takes as input the disassembly of a FinSpy VM program, and produces as output a blob of x86 machine code that no longer relies upon the VM. After producing output, we then inspect our initial results in devirtualization to determine a few deficiencies and remaining tasks before our devirtualization can be considered complete. This phase ends by fixing one of the issues revealed by inspecting the devirtualized output.
3. Part #3, Phase #3: devirtualizing function calls. The devirtualization generated by the previous stage still has a few deficiencies that need to be fixed before the output is fully usable. As it turns out, all of these deficiencies relate to how the FinSpy VM deals with function calls and function pointers. We discuss the source of the difficulties and determine what needs to be done to fix the issues. Then we discuss a few strategies that we might employ to solve the problems, including a somewhat sophisticated one involving x86 emulation. We settle for something less than that.
4. Part #3, Phase #4: second devirtualization. The previous phase collected information about virtualized functions in preparation for a second attempt at devirtualization. This phase incorporates that information into the devirtualization process, and then re-inspects the devirtualized code for defects. After fixing one more remaining major issue and two small ones, we now have our complete devirtualized blob of x86 machine code for the FinSpy VM program in our sample. We trick IDA into loading our devirtualized code, and now our devirtualization journey is complete.

After phase #4 was complete, I next completely analyzed the devirtualized FinSpy dropper binary. I may publish some techniques I saw that weren't widely documented, and I may publish a complete analysis of the dropper. However, that writing remains to be done, and is not part of this blog series on devirtualization.

Enjoy.
