Becoming a full-stack reverse-engineer

In only 3 short years

What this is

This is a roadmap to becoming capable of working at any level of abstraction in software. It won’t make you an expert reverse-engineer, breaker of things, kernel developer, compiler developer, graphics programmer, or anything else. It *will* give you the foundation to become any of those things, though.

Complete this and you’ll be in the top 1% of reverse-engineers and provide an easy pathway to the top 1% of security professionals.

What this isn’t

This isn’t a guide with a bunch of things to go reverse-engineer or otherwise break. In fact, almost none of this is actually directly reverse-engineering. That’s because reversing is 99.9% understanding the layers of abstraction and how they stack on top of each other, and 0.1% actually reading code.

This is not a direct path. It meanders and it runs off on tangents that seem random and unrelated to reversing. This is by design and I believe that it makes it better.

Caveats

- This is largely based on the trajectory I followed. It wasn't planned out, it wasn't necessarily the best, and it won't work for everyone. Take **everything** I say here with a grain of salt.
    - I'm doing my best here, but trust me: I don't know everything, even going on 20 years in.
- This schedule is an estimate at *best*. Some people might take a year for this, most probably will take at least 3, if not significantly more.
- Not everything here is necessary, by any means, and the suggestions I make are not going to be universally accepted by anyone in the industry, even myself. Skip around, skip things, dive deeper into some other things. It's your life!

Prerequisites

There are none!

Okay, that’s kind of a lie. If you already know C, x86 assembly, how to write compilers, how to write emulators, how JITs work, how kernels are constructed, and more, then this goes from 3 years to 20 minutes.

You really don’t **need** to know anything going in. If you get stuck or there’s something totally unknown to you, it’s time to learn. Anyone can go through this, it just will take you longer depending on how much you need to cram for each of the things I’m going to have you work on.

Year 1

Year 1: Read “Reversing”

Start by reading the book “Reversing” by Eldad Eilam. It’s not the perfect book, but it provides a great jumping-off point. What you’ll learn:

- Tools you should know about
- How to reverse-engineer code
- How to reverse-engineer file formats
- How copy protection works
- How decompilers work (kind of)

Year 1: Learn assembly

Learn assembly for at least one architecture; x86 is a great start. This isn't because it's simple, uniform and sane, or the best architecture; it's none of those things. But it will teach you everything you need to know, it's extremely widespread, and starting with some of the ugliest ASM will make you grateful for everything else.

- Write some C, compile it, disassemble it, then hand-decompile back to C
- Have your friend write and compile some C
    - Disassemble and decompile it
    - Have your friend check your work
- Repeat with more and more complex code; make sure to learn about floating point and vector code

Year 1: Reverse a game

Games provide a lot of great lessons to reverse-engineers and this is a good place to start out.

- Pick a 3d game, preferably late 90s to mid 00s, running on a custom engine
    - This is important because off-the-shelf engines typically have more-open formats and we don’t want that
- Reverse-engineer its data archive format
    - Write an unpacker
- Reverse-engineer its model format
    - Write a renderer
    - Bonus points: write it for OpenGL or WebGL and then port to Vulkan
        - Understanding 3d APIs is crazy valuable later

Year 1: Read “Compilers”

Read the Dragon book -- “Compilers” by Aho et al. This isn’t a great book, there are probably better out there, but I personally haven’t read them and can’t recommend any directly. This book does provide a good base level to work from, at least. What you’ll learn:

- How compilers work
- Lexing code
- Parsing code
- How type systems work
  - In a really simplistic, nearly useless way
- Generating and optimizing code
- Garbage collection

Year 1: Write a source-to-source compiler

Also known as “transpilers”, possibly the worst term in tech circles, these are a good way to get your feet wet with compiler development. This will teach you more than reading Compilers ever did.

- Write a source-to-source compiler for one high-level language to another
    - Good source language: Scheme, Forth, Python
    - Good target language: JavaScript, Python, Ruby
    - Consider making your own source language
        - It’s not that hard, it’s fun, and it teaches you a lot
    - Bonus points: make your compiler target multiple different languages
        - (Good luck)

Year 1: Write an assembler

Writing an assembler isn't actually that valuable an exercise, but it's an easy and short one that will eventually, one day, come to be useful to you. Take the few days/weeks and just do it.

- Write your own assembler to machine code or bytecode
    - **DO NOT** choose x86 for this
    - Pick MIPS, 32-bit ARM, or something like CIL
        - All the learning, 100x the effort

Year 2

Year 2: Write another compiler to assembly

This time, you're going to be compiling down to assembly. This is probably the single hardest thing on this entire list, and it's how we're going to start year 2, because it's just that valuable.

- Write a compiler from some language down to assembly for some target architecture
  - C is probably the best source language here
    - Really, you'll compile some subset of C; don't try building your own **entire** C compiler
  - Target doesn't matter too much, but do pick a real architecture and not a VM
    - You'll learn a lot more in this case

Year 2: Read “Reverse Compilation Techniques”

Understanding how decompilers really work is valuable in ways I can’t even explain. This sets you up to build decompilers and deobfuscators, reverse-engineer code, and generally break everything.

- Read “Reverse Compilation Techniques” by Cifuentes
    - This is the bible of decompilation, even as old as it is

Year 2: Write a bytecode decompiler

Both Android’s Dalvik and .NET’s CIL are easy to understand and work with. They also closely mirror the languages which compile to them, unlike most, making them optimal places to begin your decompiler journey.

- Write a decompiler for Dalvik or CIL
  - Start with goto-based flows
  - Move to reconstructing flow based on graph theory
  - Transform it to SSA form for optimization and cleanliness

Year 2: Write a machine code decompiler

Decompiling from machine code is drastically, drastically different than decompiling from bytecode. The starting languages -- C, C++, Rust, whatever -- are totally different from the target architecture in nearly every way. This means your decompiler needs to infer a lot more about what the program is trying to do. It also means a lot more you will learn in the process.

- Write a decompiler from machine code back to pseudo-C
    - ARM and MIPS are great options for this
    - x86 is a terrible option, and yet will teach you a lot and challenge you beyond belief
        - Half of me recommends against this; the other half wants to watch the world burn

Year 2: Read the OSDev wiki

The OSDev wiki is the absolute best resource on the web for all things kernels and operating systems. Whether it's details of target systems, how to write bootloaders, how basic graphics and interrupts work, or anything else -- it's there.

- Read the OSDev wiki
  - Then read it some more
  - And then a little bit more

Year 2: Write a toy kernel

This might be the thing that is most foreign to most people coming into this. It’s so different from everything else that you’ve done so far -- in this guide or in your career -- but you’ll learn a ton in the process. It’s also just fun.

- Write a toy kernel
  - Display some text on the screen, take input from the keyboard, and maybe display some basic graphics
  - Do it in C, targeting protected mode x86
    - This isn’t the time to choose Rust and targeting some obscure ARM SBC
    - This is the most well-documented path to getting something running quickly and easily

Year 2: Read the OSDev Wiki

- Go read the OSDev wiki some more
    - You know you need to

Year 2: Rewrite your toy kernel

Now that you've done it once, do it again but better. You know you made some horrible mistakes the first time around, and that you made it completely non-portable. Time to fix that.

- Rewrite your toy kernel
  - Now is the time where you could branch out and write it in a different language
  - Then port it to another target while sharing as much code as you can or want
    - Making a kernel -- even a toy kernel -- portable is a lot of work and teaches you a lot about proper design

Year 2: Write a microkernel

Your first kernel was almost certainly a monolithic mess, most likely without any separate address spaces, and probably didn't even contain processes. Time to go to the exact opposite end of the spectrum.

- Write a microkernel
  - Something inspired by L4 is a good place to start
  - If you want to be adventurous, try writing this in assembly or in a language like Rust
    - These are purposely tiny, so it's a pretty small bit of code whether you're super low-level or high-level

Year 3

Year 3: Write an interpreting emulator

Emulators for game systems allow you to really put everything you’ve learned so far into practice in a big way. Let’s start simple, though.

- Write an interpreting emulator
  - Pick a well-known platform for this -- NES, SNES, Gameboy, and Playstation are all good options
  - Your goal here should be clarity and simplicity of code, not performance

Year 3: Write a recompiling emulator

Now we can sprinkle in some compiler development. Recompilers are a fun and interesting topic, which astoundingly few people work on.

- Write a recompiling emulator
  - This could be based on your interpreting emulator from before, or something entirely different
  - I recommend sticking with well-documented platforms for this, however
    - You're going to have enough challenges already
  - One interesting idea is recompiling to JavaScript or .NET CIL
    - This allows you to get a lot of performance with minimal work on your JIT
    - I do recommend writing a JIT targeting like x86 or ARM, though
      - It's a compiler challenge more than anything else

Year 3: Write an emulator for a black box platform

There aren't many platforms without extensive documentation at this point, but they do still exist. The original Xbox, for instance, isn't well-documented. In fact, the older or newer something is, the less likely it is to be well-documented or have good existing emulators; most from the late 80s to mid 2000s are already done. One option you have is to pick up a kid's toy like a Vtech handheld and attempt to pull the code from that and emulate it; these are virtually untouched.

- Pick an unknown or semi-unknown platform and build an emulator
  - Read everything you can find about it
  - Begin writing an emulator
  - Read all the assembly you can find, for any unknown parts
  - Repeat until it works to a satisfactory level