An Introduction to Reverse Engineering

Jake Vossen

2019-03-19

Colorado School of Mines - oresec

Introduction

What is Software Reverse Engineering?

- IEEE defines it as “he process of analyzing a subject system to identify the system’s components and their interrelationships and to create representations of the system in another form or at a higher level of abstraction”
- Generally is taking a piece of compiled software and analyzing it, revealing information about the source code
- Often used in security research, but also have implication in game emulation and other areas of proprietary software
- Also used to analyze malware to create figure out how to get around ransomware and other attacks

Why Ghidra?

	Target OS: Linux	
IDAPROCL	IDA Pro Computer License [Linux]	1879 USD
IDAPROFL	IDA Pro Floating License [Linux]	2819 USD
IDASTACL	IDA Starter Computer License [Linux]	979 USD
IDASTAFL	IDA Starter Floating License [Linux]	1469 USD
HEXARM64FL	ARM64 Decompiler Floating License [Linux]	3944 USD
HEXARM64L	ARM64 Decompiler Fixed License [Linux]	2629 USD
HEXARMFL	ARM32 Decompiler Floating License [Linux]	3944 USD
HEXARML	ARM32 Decompiler Fixed License [Linux]	2629 USD
HEXPPCFL	PPC Decompiler Floating License [Linux]	3944 USD
HEXPPCL	PPC Decompiler Fixed License [Linux]	2629 USD
HEXX64FL	x64 Decompiler Floating License [Linux]	3944 USD
HEXX64L	x64 Decompiler Fixed License [Linux]	2629 USD
HEXX86FL	x86 Decompiler Floating License [Linux]	3944 USD
HEXX86L	x86 Decompiler Fixed License [Linux]	2629 USD
UPDHEXARM64FL	ARM64 Decompiler Floating Support Renewal [Linux]	1319 USD
UPDHEXARM64L	ARM64 Decompiler Fixed Support Renewal [Linux]	879 USD
UPDHEXARMFL	ARM32 Decompiler Floating Support Renewal [Linux]	1319 USD
UPDHEXARML	ARM32 Decompiler Fixed Support Renewal [Linux]	879 USD
UPDHEXPPCFL	PPC Decompiler Floating Support Renewal [Linux]	1319 USD
UPDHEXPPCL	PPC Decompiler Support Renewal [Linux]	879 USD
UPDHEXX64FL	x64 Decompiler Floating Support Renewal [Linux]	1319 USD

And if that wasn't enough...

	Shipping	
COURIER	Courier Shipping	75 USD

And Ghidra...

- Free and open source - Apache 2.0 Licensed or Public Domain (choice of contributor)
  `https://github.com/NationalSecurityAgency/ghidra`
- Has a lot better support for people working on teams then IDA
- Security professionals are saying it rivals the functionality of IDA

Compiler Theory (a 10,000 foot overview)

What are the Goals of a compiler?

- Take a programming language that is human-readable and writable, into something that the computer can run.
- Generally a program is considered compiled if it is in assembly code - assembly to machine code is done by a *assembler*, not a compiler.
- Three parts: parsing, type checking, and code generation. [1]
- Optionally, you can improve the efficiency as you do this

Wait, how do decompilers work anyways? Or even a compiler?

- Option A: Compile down into another language to be run. GNU yacc and bison are intermediate languages and parsers where you can define your new programming language and generate machine code from it.
- Option B (generally better): Compiler is written in the language itself.

How are compilers born for language X?

Problem: We want a compiler that can compile the compiler.

Solutions:

- Write a compiler for X in language Y. Now that we have an X compiler, rewrite the compiler in X itself
- Expand on an existing programming language and use it's compiler
- Write the compiler in X, hand compile it, now you have a compiler

So the very first compilers where written **by hand** in assembly.

Decompiler Theory (100,000 foot overview)

Why are decompilers are harder and more limited?

People do their best, but decompiling is even harder than compiling, which is already a really hard problem. Some of the things that make it harder is:

- Types are in the hidden, instead of being in plain text in the language.
- If something isn't formatted the way you want, you can't just throw and error like compiling.
- Output has to be human readable and clean, compared to assembly which
- Exception handling

Things like tail end recursion and stack/heap variable ranges are just too difficult sometimes, so decompiling is not perfect. Generally, it will result in "equivalent" code, not necessarily the the original code.

What are the steps decompilers take?

- Generate microcode: a step between machine code and assembly. This lets us do some optimization before we translate further
- Optimization: The microcode is verbose, so we need to clean it up a bit. This is done at a local (inside a function/subroutine) to a global level. This results in a more readable end result.
- Structural analysis: Look at our microcode and figure out where we are looping, jumping, etc to create a control flow graph
- Pseudocode generation: Generate the output from the optimization and structural analysis. After this is complete, we also have to optimize and clean it up, as it still may be too verbose.
- Type analysis: Black magic analysis of the pseudocode