Reverse Engineering a Bluetooth Application: Discovering the Secrets of a Makeup Printing Device

Independent Study Thesis

Presented in Partial Fulfillment of the Requirements for
the Degree Your degree in the
Department of Your Department at The College of Wooster

by
Natalie Pargas

The College of Wooster
2025

Advised by:

Your advisor (Department)

THE COLLEGE OF
WOOSTER

Abstract

Include a short summary of your thesis, including any pertinent results. This section is *not* optional, and the reader should be able to learn the meat of your thesis by reading this (short) section.

This work is dedicated to the future generations of Wooster students.

Acknowledgments

I would like to acknowledge Prof. Lowell Boone in the Physics Department for his suggestions and code.

Vita

Publications

Fields of Study Major field: Major

Minor field: Minor

Specialization: Area of IS research

Contents

Abstract iii

Dedication iv

Acknowledgments v

Vita vi

Contents vii

List of Figures x

List of Tables xi

List of Listings xii

CHAPTER PAGE

1 Introduction 1
- 1.1 What is Reverse Engineering? 1
  - 1.1.1 Purposes 2
    - 1.1.1.1 Education 2
    - 1.1.1.2 Legacy Systems 2
    - 1.1.1.3 Security 2
    - 1.1.1.4 Interoperability 2

2 Bluetooth 3
- 2.1 Architecture 4
  - 2.1.1 Pairing Based Network 4
  - 2.1.2 Piconet 4
  - 2.1.3 BR/EDR Piconet 5
- 2.2 Protocol Stack 5
- 2.3 Bluetooth Low Energy(BLE) 5
  - 2.3.1 Architecture 5
    - 2.3.1.1 Application 5
    - 2.3.1.2 Host 5
    - 2.3.1.3 Controller 5
  - 2.3.2 Layers 5
    - 2.3.2.1 Physical layer 5
    - 2.3.2.2 Link layer 5
      - 2.3.2.2.1 Bluetooth Address 5
    - 2.3.2.3 HCI Layer ? 6
  - 2.3.3 Peripherals and centrals 6
    - 2.3.3.1 Peripherals 6
    - 2.3.3.2 Centrals 6
    - 2.3.3.3 Smartphones in BLE 6
  - 2.3.4 Connection 6
    - 2.3.4.1 Connection event 6

2.3.4.2 Connection parameters ? . . . 6
2.3.4.3 Channel hopping . . . 6
2.3.4.4 White list and device filtering ? . . . 6
2.3.5 Services and Characteristics . . . 6
2.3.5.1 Attribute Protocol (ATT) . . . 6
2.3.5.1.1 UUID . . . 6
2.3.5.2 Generic Attributes Table (GATT) . . . 7
2.3.5.3 Services . . . 7
2.3.5.4 Characteristics . . . 7
2.3.5.5 Profiles . . . 7
2.4 BLE Hardware/Software tools . . . 7
2.4.1 Nordic Semiconductor . . . 7
2.4.1.1 NRF connect . . . 7
2.4.1.2 NRF UART . . . 7
2.4.1.3 NRF Logger . . . 7
2.4.2 Android BLE . . . 7
2.4.2.1 BLE Classes . . . 7
2.4.2.1.1 Bluetooth Adapter . . . 7
2.4.2.1.2 Bluetooth Gatt . . . 7
2.4.2.1.3 Bluetooth GATT Callback . . . 7
2.4.2.1.4 Bluetooth GATT service . . . 7
2.4.2.1.5 Bluetooth GATT characteristic . . . 7
3 Reverse Engineering 8
3.1 Analysis Methods and Tools . . . 8
3.1.1 Static Analysis . . . 8
3.1.1.1 Disassembler . . . 8
3.1.1.2 Decompiler . . . 8
3.1.1.3 Dumping Tools . . . 9
3.1.2 Dynamic Analysis . . . 10
3.1.2.1 Debugger . . . 10
3.1.2.1.1 User Mode Debuggers . . . 10
3.1.2.1.2 Kernel Mode Debuggers . . . 11
3.1.2.2 System Monitor . . . 11
3.1.2.3 Virtual Machine . . . 12
3.2 Compiled Code RE . . . 12
3.2.1 High level languages . . . 12
3.2.1.1 Control Flow . . . 13
3.2.2 Low level languages . . . 14
3.2.2.1 Registers . . . 14
3.2.2.2 The Stack . . . 15
3.2.2.3 Flags . . . 16
3.2.2.4 Functions . . . 16
3.3 Example Problems . . . 17
3.3.1 Reversing Hello World . . . 17
3.3.2 Crack Me . . . 18
3.4 Java RE . . . 23
3.4.1 JVM . . . 23
3.4.2 Reversing Bytecode . . . 23
3.4.3 Java Name Obfuscation . . . 23
3.4.3.1 Obfuscating names for .class file . . . 23
3.5 Mobile RE . . . 23
3.5.1 Hybrid Approach . . . 23
3.5.2 Symbolic Execution . . . 23

3.5.3 Generating UML Diagram . . . 23
3.5.4 Android . . . 23
3.5.4.1 Tools . . . 23
3.5.4.1.1 Dalvik VM Android . . . 23
3.5.4.1.2 APK Tools . . . 23
3.5.5 Apple? . . . 23
3.5.5.1 Obj C tools . . . 23
4 Methods 24
4.1 Environment Setup (components) . . . 26
4.1.1 Virtual Machine . . . 26
4.1.2 Tools . . . 26
4.1.2.1 JADX . . . 26
4.1.2.2 Ghidra . . . 26
4.1.2.3 APK Tools . . . 26
4.1.2.4 JBSE (symbolic Java VM) . . . 26
4.1.2.5 UML Diagram Creator (Ask Dr.Guarnera) . . . 26
4.1.2.6 Emulator ? . . . 26
4.2 Scripting . . . 26
4.3 App Interoperability . . . 26
References 27
Index 28

List of Figures

Figure		Page
3.1	Figure modified using Intel® 64 and IA-32 Architectures Software Developer's Manual and https://flint.cs.yale.edu/cs422/doc/pc-arch.html#register	15
3.2	Hello World Program	18
3.3	Hello World Disassembly	19
3.4	Crack Me Header	20
3.5	Crack Me Libraries	20
3.6	Crack Me Strings	21
3.7	Crack Me Ghidra Disassembly	21
3.8	Crack Me Code Section 1	21
3.9	Crack Me Code Section 2	21
3.10	Crack Me Code Executable	22

List of Tables

Table Page

List of Listings

Listing Page

CHAPTER 1

Introduction

1.1 What is Reverse Engineering?

Reverse Engineering is the process of disassembling something in order to understand how it works. It has been done long before it was used for technology. For example, the dissections that were a common high school experience. They were done to break down something as mystifying as life into clear components and functionalities. One of the best ways to fully understand how something works is to open it up.

Reverse engineering software can be done in many ways. The methods of analysis can be divided into two categories, static and dynamic analysis. Static analysis involves combing over a snapshot of the code in a single state. Dynamic analysis is done when the program is being executed and changing states. To start reverse engineering, usually a program's executable is passed to a disassembler which will break it down into the assembly level code. A programmer can use system monitoring tools to display information that was gathered by the operating system on the program and how it interacts with its environment. A debugger shows what happens in the CPU with the disassembled code one line at a time. Using these different tools, the programmer must use their intuition to guide the reversing process.

There are many reasons why someone might want to reverse engineer software. The only time that the workings of software is available to anyone is when that software is open source. Open source software is freely available source code that anyone can view, share, and modify. However, software developers sometimes opt out of publicly distributing the source code. There are a number of potential causes for this, including licensing and ownership issues, the need to preserve proprietary data, or other personal reasons.

Some of the most common reasons a person would choose to reverse engineer software is so that they can perform malware analysis, improve programming skills, recover lost source code, and implement interoperability between programs. My first introduction to reverse engineering was through a video where

the creator was reverse engineering Apple's Facetime for Mac so he could reinstate closed captioning for his hearing impaired mother. The list of uses is as extensive as programs out there.

Most software developers know reverse engineering in the context of malware analysis. A classic example is a Trojan virus. Malicious developers can hide malware in programs that are designed to look innocuous. Breaking apart these programs will unveil the malware hidden inside. Developers also often call on external libraries. With large projects, keeping track of dependencies can be hard. Hackers can hide entry points in these libraries so they can access unauthorized information. Reverse engineering is the only foolproof way to reveal exactly what is happening at each step of a program's execution.

Learning how something is being done can be great to improve skills. Reverse engineering requires extensive knowledge of both assembly code and how high level code structuring. Reverse engineering someone else's code is no simple feat and will cause the reverser to need to learn in depth the possible ways a program could be constructed. This project will be an endeavor in learning both forward engineering and reverse engineering.

People may reverse engineer their own program to recover lost source code. While this is no easy task, it is possible to use reverse engineering to recover source code. Many software developers know the pain of losing source code and, with an executable, reverse engineering offers a potential solution.

Interoperability is one of the reasons a person might choose to reverse engineer something. This purpose is also only able to be accomplished through reverse engineering. When working with external libraries or operating systems that have documentation on usage but no source code, oftentimes documentation doesn't cover all use cases. While a programmer could keep throwing things at the wall or try to contact the vendor, reverse engineering provides a surefire way of figuring things out.

1.1.1 Purposes

1.1.1.1 Education

1.1.1.2 Legacy Systems

1.1.1.3 Security

1.1.1.4 Interoperability

CHAPTER 2

Bluetooth

Bluetooth is an incredibly popular technology that has been expanding rapidly over the past decade. Bluetooth technology was designed to replace the need for short range wired connections. It can be used for pretty much any case that requires data transfer including music streaming, video streaming, medical devices that require constant updating, and more. It works similarly to WiFi, the major difference is that WiFi facilitates internet connection across multiple devices while bluetooth has more limited device connectivity and a lot more variability in its use cases [https://www.intel.com/content/www/us/en/products/docs/wireless/how-does-bluetooth-work.html]. Bluetooth also differs from WiFi in that it is designed for low latency applications that quickly send and receive small chunks of data.

There are also two main types of bluetooth. There is Bluetooth Basic Rate (BR) or Bluetooth Classic and Bluetooth Low Energy (BLE), the latter has gained much more popularity in modern applications. Bluetooth Basic Rate is the older version of the two and can support audio streaming and data transfer. It was used in versions 1, 2, and 3. It's mainly used for wireless audio streaming like headphones, speakers, and in-car entertainment systems. It also uses point-to-point device communication. Bluetooth Low Energy is supported in versions 4 and 5 of bluetooth. Bluetooth Low Energy has a more broad range of abilities, it can be utilized for audio streaming, data transfer, device networks, and location services. It is also more flexible in its communication topologies, supporting point-to-point, mesh, and broadcast. Bluetooth LE has also become a popular option for high accuracy location services. It uses relative device positioning using other devices to determine precise location. The architecture of Bluetooth has evolved and expanded as time has gone on.

2.1 Architecture

Bluetooth Basic Rate and Bluetooth Low Energy both have device discovery, connection establishment, and connection mechanisms [https://www.bluetooth.com/wp-content/uploads/Files/Specification/HTML/Core-54/out/en/architecture,-mixing,-and-conventions/architecture.html]. Basic Rate allows users to connect synchronously or asynchronously at data rates of 721.2 kb/s. Bluetooth Basic Rate introduced the option to enhance the connection with Bluetooth Enhanced Data Rate that ups the data rate to 2.1 Mb/s. Bluetooth Low Energy was created to support products that have lower current consumption requirements, lower costs for operation, and lower complexity than BR/EDR. BLE has also been designed for devices that have lower data rates. BLE has an optional 2 Mb/s physical layer data rate and offers isochronous data transfer.

2.1.1 Pairing Based Network

The core of bluetooth is creating a network where two (or slightly more) devices pair with each other. The devices being paired can be broken down into the Master and Slave(s). The term master and slave were also specific to classic bluetooth and they can be used interchangeably with the newer terms, Central and Peripheral. The master device coordinates the data being sent between it and any slaves. Coordinating that information includes running the functions necessary for time synchronization, sleep scheduling, and configuration of the channels being used through frequency hopping [next gen bluetooth]. Slave devices can communicate bi-directionally with the master device, sending or receiving data between the master depending on its request. Classic bluetooth mode has two network topology configurations, Piconet and Scatternet.

2.1.2 Piconet

A piconet is a type of ad hoc network topology, meaning it operates without preestablished infrastructure. Devices in a piconet communicate directly with one another without the need for a centralized device like a router or access point. The network is organized with one master device and one or more slave devices. A single master can support up to seven active slaves, while each slave can only connect to one master [next gen bluetooth]. The master device controls the timing and synchronization of all devices in the piconet. Slave devices cannot communicate directly with each other; when such communication is needed, the master serves as a relay, forwarding data between the slaves.

2.1.3 BR/EDR Piconet

All connections made using a BR/EDR controller are made within a piconet. The BR/EDR devices communicate through the same physical channel by synchronizing with a common clock and hopping sequence [bluetooth core spec6]. The piconet clock also known as the common clock is the same as from the central or master clock and the hopping sequence comes from the central's clock and the central's bluetooth device address.

Multiple independent piconets may exist near one another.

2.2 Protocol Stack

2.3 Bluetooth Low Energy(BLE)

2.3.1 Architecture

2.3.1.1 Application

2.3.1.2 Host

2.3.1.3 Controller

2.3.2 Layers

2.3.2.1 Physical layer

2.3.2.2 Link layer

2.3.2.2.1 Bluetooth Address

2.3.2.3 HCI Layer ?

2.3.3 Peripherals and centrals

2.3.3.1 Peripherals

2.3.3.2 Centrals

2.3.3.3 Smartphones in BLE

2.3.4 Connection

2.3.4.1 Connection event

2.3.4.2 Connection parameters ?

2.3.4.3 Channel hopping

2.3.4.4 White list and device filtering ?

2.3.5 Services and Characteristics

2.3.5.1 Attribute Protocol (ATT)

2.3.5.1.1 UUID

2.3.5.2 Generic Attributes Table (GATT)

2.3.5.3 Services

2.3.5.4 Characteristics

2.3.5.5 Profiles

2.4 BLE Hardware/Software tools

2.4.1 Nordic Semiconductor

2.4.1.1 NRF connect

2.4.1.2 NRF UART

2.4.1.3 NRF Logger

2.4.2 Android BLE

2.4.2.1 BLE Classes

2.4.2.1.1 Bluetooth Adapter

2.4.2.1.2 Bluetooth Gatt

2.4.2.1.3 Bluetooth GATT Callback

2.4.2.1.4 Bluetooth GATT service

2.4.2.1.5 Bluetooth GATT characteristic

CHAPTER 3

Reverse Engineering

3.1 Analysis Methods and Tools

3.1.1 Static Analysis

Static analysis is a practice of analyzing code without executing it. It can be used to help inform dynamic analysis. Static analysis tools can analyze at either the source code level with code that hasn't been compiled yet or machine language level with code that has. Static analysis tools can be used by reverse engineers to retrieve and comb over assembly level code. This is one of the most basic functions of any static analysis tool, but many of them offer a lot more functionality to help reverse engineers. Most static analysis tools are designed to help give the reverse engineer a starting off point with interpreting assembly level functions.

3.1.1.1 Disassembler

One of the most key static analysis tools for a reverse engineer is a disassembler. Disassemblers are applications that take in a program's executable file and generate a file with the assembly code. This isn't too difficult because assembly is just a text translation of the machine readable binary code. Unfortunately, that makes it a lot less clear than higher level programming languages. Disassembly is also unique to a person's processor, but there are disassemblers that can support more than one CPU architecture. Some examples of disassemblers are IDA, Binary Ninja, Hopper, Ollydbug, Radare2, and Ghidra.

3.1.1.2 Decompiler

Decompilers are similar but more complex than disassemblers. Instead of just producing the assembly language code, decompilers attempt to produce high-level readable code. They attempt to produce something that looks as close to the source code as possible by reversing the compilation process [**Reversing**]. The

front end of the decompiler decodes low-level assembly instructions and translates it into an intermediate representation specific to the decompiler. The intermediate representation is then iterated on to remove as many extraneous details as possible while preserving the important details. Lastly the back end takes the polished up intermediate representation and translates it again into a high level language.

While this may sound like a simple way to retrieve the source code, it is highly unlikely the high level language returned will be something that actually matches. Anyone who has ever run text through a translation website multiple times knows that each iteration causes a loss in fidelity and oftentimes the end result will be gibberish. Decompilers go a step further and have no immediate knowledge of things like function or variable names and structures. This does not make them useless, though. For example, a text translator will usually return your original result when given a simple phrase. Similarly a decompiler will most likely be able to pick up on more straightforward functions such as adding a + b. Decompilers offer hints and snippets into what the source code may look like which is valuable information for a reverse engineer. Most of the examples for disassemblers listed also count as decompilers.

3.1.1.3 Dumping Tools

Executable dumping is usually the first step in reverse engineering a program. Executable dumping helps inform a reverse engineer what a program does, how it interacts with external elements, and general knowledge of how a file is structured. Using dumping tools, a reverse engineer will start by figuring out what type of file they're dealing with. They will then start to unpack the format. Extracting strings will immediately offer insight into what language the source code is written in, what library modules it uses, what kinds of data it is dealing with, and what the goals of program functions may be. Other useful information we can retrieve is the layout of the file in memory and what the general logical structure of the file is. Once we know the type of file we also know which tools we can use. For example, some debuggers only work with certain high and low level languages. Some popular executable dumping programs are ones already built into your machine like DUMPBIN for windows or otool for mac.

Otool is identifies a file's mach header. On systems with a Mach kernel like macOS and iOS, the executable binaries they use will have a mach header. All headers have a magic number to identify them. T hin binaries only have that number while fat binaries with fat headers will have locations of the other executable's headers in them

3.1.2 Dynamic Analysis

Dynamic Analysis occurs when the program or section being analyzed is run during the analysis process. Because dynamic analysis requires the program to execute with unknown results, it is safest to do it in an enclosed or sandboxed environment to protect the rest of the system. Thus easily manipulated virtual machines are usually set up during the dynamic analysis process. Many tools can be part of the dynamic analysis process. Anything that needs to run and monitor some output is considered dynamic. Tools may run the full program, a portion of the program, or even just a single line. Tools that monitor the external environment while the program runs are also considered dynamic.

3.1.2.1 Debugger

Debuggers are a tool usually used for developers to locate and work through errors in their programs, but they're also vital tools in reverse engineering. Many debuggers can work through assembly language. While assembly language may be difficult for a human to parse through, it is the exact same logic that gets broken down and sent to a computer meaning that the computer can show what each assembly instruction is intended to do. Most debuggers software engineers use were actually designed from the ground up with the purpose of stepping through assembly code. The debugger can show the state of CPU registers along with a memory dump that shows what's in the stack. Debuggers usually contain disassemblers which as discussed is a vital reverse engineering tool. Many debuggers also contain both software and hardware breakpoints. Software breakpoints are instructions added into the code during runtime to pause and hand control over to the debugger. Hardware breakpoints are a CPU feature which allow the processor to pause execution when a certain memory address is reached and hand control over to the debugger. A reverse engineer can use insert breakpoints at data structures of interest and use the debugger to reveal what they are. Debuggers also offer a clear view of registers and memory to aid in a reverse engineer's ability to process low level information.

3.1.2.1.1 User Mode Debuggers Most debuggers that a programmer will use are user mode debuggers. They run on a system like any other application then seize control of the target program to debug **[Reversing]**. An advantage to them is that they are easier to set up and navigate than their kernel-mode counterparts. Usually it's fine to stay limited to user mode viewing of an application. It only creates troublesome limitations when the target application has kernel mode components such as device drivers. User mode debuggers are also not always sufficient when trying to debug a program before it reaches the main entry point. These kinds of programs are usually ones that have a lot of statically linked libraries in the executable. The final and most likely to be problematic issue is that user-mode debuggers can only view a single process in a

program. This can cause issues when the target application has processes that interact in unknown ways. The user may not know which process to zero in on that has the code of interest.

3.1.2.1.2 Kernel Mode Debuggers Kernel mode debuggers are different from user-mode debuggers because they capture the entire system and not just a single process. Instead of running atop the operating system, kernel mode debuggers sit alongside the system kernel and stay ready to capture the entire system's stats. Kernel mode debuggers can be more helpful to reverse engineers because they offer more clues as to what's going on with the system. A key tool for kernel-mode debuggers is that they allow the placement of low level code breakpoints. As a reverse engineer working with assembly code, being able to test different sets of assembly instructions that may be the code section a reverser is looking for is incredibly useful.

For example, picture a scenario where a reverse engineer is looking for the API responsible for handling moving windows? That will need to be managed by a windows manager in the kernel. The complexity is introduced when trying to identify which API moves a particular window. Since there are multiple APIs that can be used to move a window, pinpointing the one you are trying to focus on is a difficult task. This is where kernel mode debuggers come in handy. Low level breakpoints in the operating system responsible for shifting windows around will help you identify which API is getting called when a window is moved [**Reversing**]. From there the reverse engineer will be able to hone in on that API.

Unfortunately, kernel mode debuggers come with their own drawbacks. Setting them up can be challenging because they require access to the full system. They need to suspend the entire system while running so they can go line by line, which means the system can no longer have multiple threads going [**Reversing**]. They will also usually need to be set up inside a Virtual Machine which will be discussed in a later section [**PracticalRE**]. These are reasons why a reverser should exhaust their other options before jumping into using a kernel mode debugger. Still, they are powerful tools when the scenario calls for it.

3.1.2.2 System Monitor

System monitoring can be a crucial part of the reverse engineering process. In some cases, a reverse engineer can figure out what they need through system monitoring tools without ever looking at the decompiled code. System monitoring tools can capture what happens between the code and hardware using the intermediate channels of input/output [**Reversing**]. For local system monitoring, tools monitor things such as file operations (creating, deleting, moving, etc). For programs that communicate over networks, system monitoring might look like recording all TCP/UDP network traffic. System monitoring tools are commonly used in virtual machines.

3.1.2.3 Virtual Machine

Many programmers will encounter virtual machines (VM) at some point during their career. It's not uncommon for an application to interface exclusively with one type of operating system. One reason is because writing applications to work with different operating systems adds time and complexity that not all programmers can afford. Choosing to write in a high level language may make code more portable because there's more verbose built-in libraries or interpreters that deal with the specifics of the systems without the need for any programmer intervention. But high level languages often trade performance for convenience which is not a trade that a developer working with large amounts of data can necessarily afford to take. What happens when a developer has an application that runs on Windows while they have a Mac? This is where a virtual machine enters into the mix. Virtual machines are safe sandboxed environments that work as miniature computers with their own operating systems within a computer. They are constructed with an interpreter and bytecode. During compile time, select code snippets are compiled specific for the VM target architecture and then inserted into the program along with the interpreter. During run-time, the interpreter begins executing the bytecode. Virtual machines are costly to implement because they are so expansive which is why only the necessary code snippets get rendered [**MasteringRE**]. Because virtual machines work in their own isolated environment, they offer powerful protections against unknown or potentially malicious software. This makes them a useful tool for reverse engineers who deal with software that they do not know the contents of. The level of control over virtual machines also makes them a useful tool because there is ultimate knowledge of system conditions.

3.2 Compiled Code RE

It is important for a reverser to understand the different goals of high level and low level languages. Understanding these differences will help with interpretation of why code is structured a certain way.

3.2.1 High level languages

High level languages exist to take away the complexity of system specific programming. High level languages work so that a programmer can focus on specifying the clean structural logic without needing to dedicate time to figuring out system specific details. While writing in something like assembly can be very performative, it would be nearly impossible for a standalone human to write a modern application in assembly. Because high level languages favor simplicity over the flexibility to do exactly what is the most efficient for achieving a program's goals, many programmers don't even know what is going on at the assembly level. A reverse

engineer will be forced to pick apart what potentially roundabout ways a program's goals are being achieved. The main goal of a reverse engineer is to use what they know about what the program was trying to do, potential ways that can be accomplished in high level code, and how to read assembly code to make their best educated guess on what's happening where. For reverse engineers, the most important thing to know about a high level language is to what level does it abstract or conceal the underlying machine code [**Reversing**]. Languages like Python that have a built in interpreter will abstract it a lot. These programs may be full of extraneous machine code. On the other hand, languages like C are written much closer to the target processor and won't have nearly as much separation between the source and machine code.

3.2.1.1 Control Flow

Control flow is what makes code more user friendly. This manifests through statements like conditionals that give general instructions for what a program should do when. ' A processor has no knowledge what statements like 'if' or 'while' mean. Under the hood, these statements translate into verbose and daunting assembly code. This is because high level conditional statements are often broken down into operation sequences because it would otherwise overwhelm the processor [**Reversing**].

Other typical structural components of programs that aid in control flow are switch blocks and loops. Switch blocks, or $n - way$ conditionals, take in an input and have n number of code blocks that are potentially executed depending on the input value. Each block of code gets assigned at least one value prior to runtime. The compiler also generates code to receive the input value and search for the proper code block to execute. The values for the code blocks are usually stored in a lookup table that has pointers to each corresponding code block. Depending on the input value, the program will go through the process of searching the lookup table then jumping to the proper code block at runtime [**Reversing**]. Loops work to allow a program to repeatedly execute a certain code block multiple times. A loop has a counter to keep track of how many iterations it has performed. There is also a conditional statement that determines when the loop will stop. Loops and conditionals are inherently intertwined. The difference is that loops execute over and over until the condition is no longer met.

To understand control flow sequences, one must understand how low level control flow is implemented. This means that a reverse engineer needs to know the specific rules of each kind of low level architecture because low level control flow is individual to the platform and therefore the language.

3.2.2 Low level languages

Earlier we mentioned that complexity is introduced when working with low level system specific details. This is especially true when the goal is to translate high level logic into something that will be understood by your machine. Of course, there must be a way to do this because the CPU does it anytime a modern application is run. But it often involves keeping track of more details than a sole human is capable of. That is why a reverse engineer must develop the skill of parsing through assembly code and being able to create a kind of 'mental image' of how it relates to high level constructs. Data management is one of the things that a person's computer keeps track of that would be difficult for a human to do. To understand why this is, we can look at the data management of a relatively low level language, C, versus the assembly representation. Consider the code:

```
int divide(int a, int b) {
    int result;
    result = a // b;
    return result;
}
```

While this function may seem incredibly simple, there is no direct translation into a machine code representation. To execute this in a low level language, it would require first storing the machine state. Then memory would need to get allocated for result. Variables a and b would need to get loaded from memory into a register. Then a would need to be divided by b and the result would need to get stored in the register that got loaded in the beginning. The machine state from before would need to get reloaded. The pointer would need to return to the caller and bring back result [**Reversing**]. One line of high level code could result in any number of assembly instructions. Managing data is one of the biggest challenges of reverse engineering.

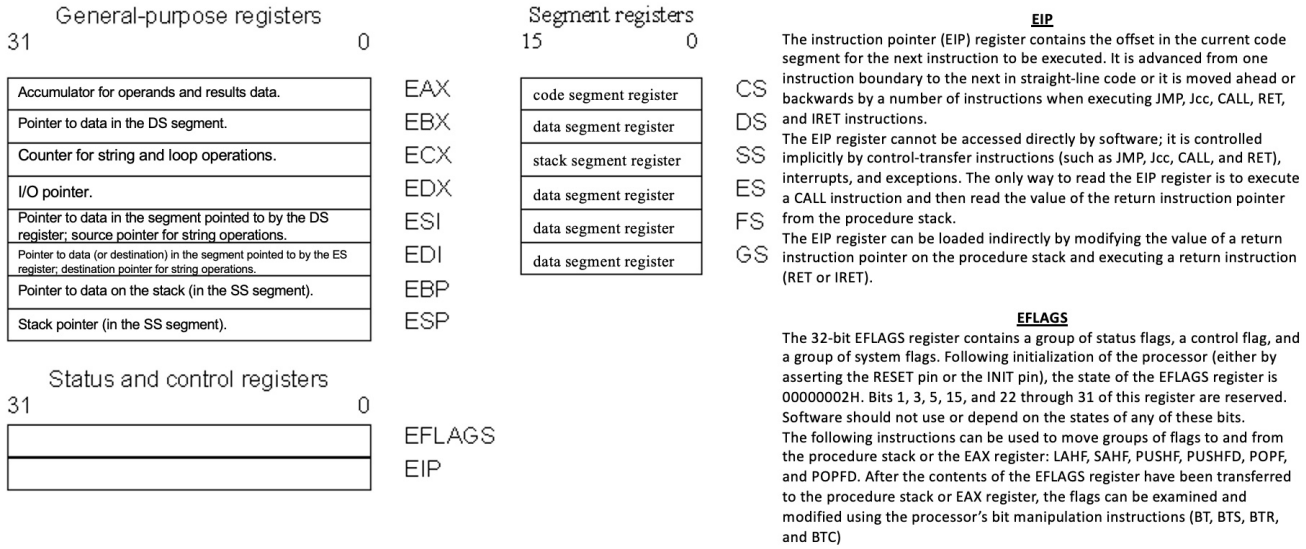
3.2.2.1 Registers

To keep from needing to constantly access RAM, microprocessors have a smaller internal memory that can be accessed with barely any performance cost [**Reversing**]. There are multiple types of internal memories that microprocessors keep and registers are one of them. Registers are little pieces of internal memory that live within the processor and are able to be accessed with an imperceptible cost to performance [**PracticalRE**]. The biggest problem with registers is that there are not very many of them. One of the most popular processors, the IA-32, only has eight 32-bit registers that can be used for anything. It does have more registers, but they all have highly specific use cases. Assembly language is written around utilizing registers because they're

so performant. They are not good for long term storage, though, that is when RAM becomes the better choice. One of the most important things to take away is that CPUs do not automatically manage this. Data management is outlined in the assembly code which is what reverse engineers will need to get comfortable sorting through.

One thing that reverse engineers will need to do is focus on figuring out what kind of values are getting loaded into a register. For example, it is easy to see when a register is only being used to grant instructions access to a specific value because that register will only show up when transferring value from memory to the instruction or vice versa. Another example is when a register shows up many times in one function. This is a good clue that one of the function's local variables is being stored in that register.

Figure 3.1: Figure modified using Intel® 64 and IA-32 Architectures Software Developer's Manual and https://flint.cs.yale.edu/cs422/doc/pc-arch.html#register



3.2.2.2 The Stack

The stack is one of two places that a value can be set aside. Registers are not managed by a processor and to use one you just need to load a value in it. Often there will not be registers available or there is a particular reason a variable will need to reside in RAM instead of in a register. That's when you'd put a variable on the stack instead [**Reversing**].

The stack is a short term storage space in memory that gets utilized by both the CPU and the program. It is the spot where short term information gets put when a register can not be used for whatever reason. Registers are for the shortest term data while the stack is for the second shortest. The stack lives in RAM like all other data and is just a carved out section for intermediate term data. Modern operating systems tend to

manage multiple stacks at the same time. Each of the concurrently running stacks is a representation of a program or thread [**MasteringRE**].

Stacks use Last In First Out data management. Items are pushed on the top of the stack and popped from the bottom of the stack. Memory in stacks is allocated from the top down where the first in address is allocated and used first while the stack grows backwards towards lower addresses [**Reversing**].

3.2.2.3 Flags

One of the registers you may have noticed from the previous figure is the EFLAGS register. This is a collection of special IA-32 registers that contain system and status flags. The system flags' job is to manage the different modes and states of the processor. The status flags are what a reverse engineer will typically be more interested in and are used by the processor to record its current logical state [**Reversing**]. They are often updated by various logical and integer instructions so they can record the outcome of these actions. There are also instructions whose operation conditions are dependent on the values for the status flags. This is what allows sequences of instructions to run different operations depending on what different input values may be, etc.

Flags in IA-32 are the crux of conditional code. Arithmetic instructions check operand conditions and then set processor flag conditions based on the resulting values. There are also sets of instructions designed to read the flags and do different operations based on the value. An example of a popular instruction set is Jcc or the Conditional Jump. It tests for predefined flag values and then jump depending on whether or not it matches with the specified conditional code [**intelManual**].

3.2.2.4 Functions

Instructions are the actual actions specified in assembly code. They are formatted with operation code (opcode) and one or two operands [**Reversing**] The opcode is what you are asking the computer to do such as MOV or JMP. The operand is the parameters that get passed to the opcode or the data being manipulated. Certain instructions will have no parameters. Data in assembly comes in three basic forms. There are register names, immediate data, and memory addresses. Register names are the names of the general purpose registers to be read from or written to like the EAX, EBX, etc. Immediate data is constant values that are embedded into the code and usually indicates there was a hard coded value in the source code. Memory addresses are the locations of operands stored in RAM. It can be hard coded and tell the processor where to read to and write from. It could also be a register with a value that will be used as a memory address. It is

also possible to combine a register with an arithmetic operation and a constant so that there is some base address and then an index offset [**Reversing**].

General purpose instructions are the ones that deal with program flow, logic, arithmetic, string operations, and basic data movement. They deal with data stored in memory at the general purpose registers, EFLAGS register, segment registers, and address information stored in memory [**Reversing**].

The MOV instruction shows up most frequently in most IA-32 instruction sets. This one deals with basic data movement. It takes in a destination operand and a source operand then moves the data from the source to the destination. The sources can be registers, immediate, or memory addresses. It is important to note that MOV cannot transfer data through memory, it can only take it out or put it in. The destination address can be a memory address (using a register or an immediate) or a register [**Reversing**].

3.3 Example Problems

3.3.1 Reversing Hello World

Before taking on the larger endeavor of reversing a program written in a declarative language (SwiftUI), it's important to start with the basics. Figure [insert figure here] shows a simple 'Hello World' program in C. To add some complexity, a few random variables and data structures were thrown in as Easter eggs. This is the program whose decompilation will be used as a jumping off point. Figure 3 is the decompilation of the program once it was put through Ghidra. The first thing to note is how with even a relatively low level language like C, the instructions in the code more than doubled. The disassembly being shown in the figure is also only a portion of Ghidra's output. The figure only shows the function section of the program output when imports, exports, labels, classes, etc take up the large majority of the program. One important part of reverse engineering is sorting through all of the information to find out what exactly is important to the reverser. Some reverse engineers refer to this process as finding the shape and edges of the data. With the important section identified, it is time to analyze the assembly code. One thing to keep in mind with the disassembled code is that all the values will be stored in hexadecimal. A good place to start is with the values we know will need to be stored. In the array there are 10 variables which all get set to 0, except for the eighth array member which is changed to equal 1. In our local_dope struct there are three variables equal to 'A', '0xffffffff', and 32000. The highlighted line in the figure shows 'A' being stored in the register w8. Shortly after, #0xffffff shows up without needing to be translated into hexadecimal. #0x7d000 is the hex for 32000. This surface level analysis offers a glimpse into the methodology of more complex reverse engineering. Lastly, the 'Hello_World!' gives us the final piece of information we need to know about what the program is.

Figure 3.2: Hello World Program

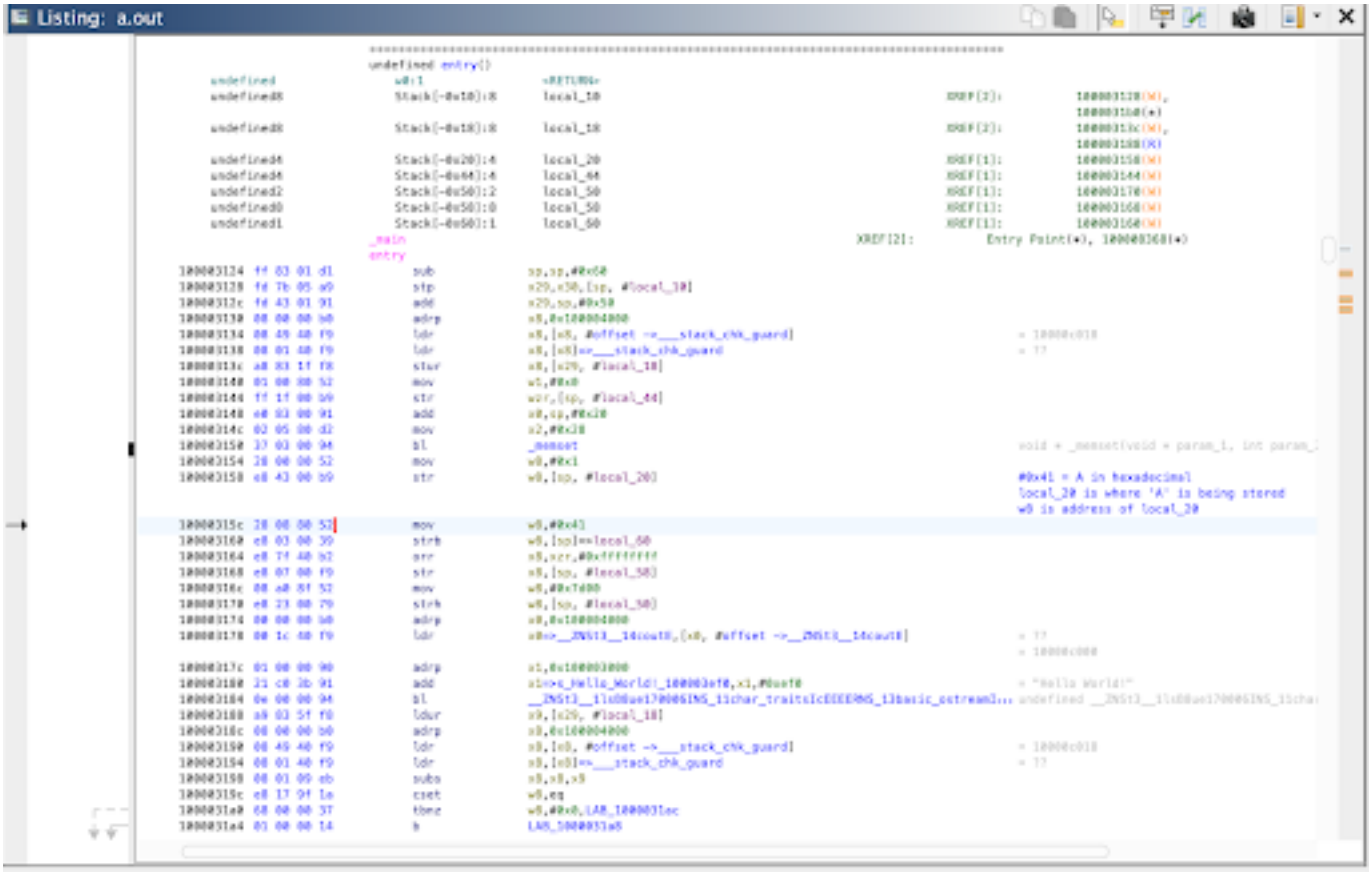
3.3.2 Crack Me

The next step in practicing reverse engineering is trying a Crack Me problem. Crack Me problems are toy problems designed by other software developers to help reverser's practice their skills. Usually there's an executable that opens up a little puzzle. The puzzles usually involve finding a password but can involve any number of things like decryption, key generation, etc. This section will cover the process of reverse engineering a simple crack me.

The first step I took in reversing this program was checking the header and libraries as shown in Figure 4 and 5.

Both of these steps were done with an executable dumping tool called 'otool'. The header information shows that the assembly language being used is x86. The library dump shows that the only library being linked is the system library. While these tools didn't offer that much information about this particular program, they are very useful in programs that have more dependencies and more specific assembly languages. These dumping tools ended up being vital later in the project because they offered insight into what libraries were being linked from the project's declarative language as well as what assembly language was being used with Apple's modified version of ARM64. Next, otool was used once again to list the strings. This is a vital part of reverse engineering and is useful to be able to refer back to during the process. Figure 6 shows the result of the string dump.

Figure 3.3: Hello World Disassembly



Examining this more closely, you can see that there's only a few strings that give a very clear idea of what the program is designed to do. There's a random string, the title of the program, a passphrase prompt, a string indicating the expected input, a congratulations message, and an incorrect guess message. With just this information, you could easily make an educated guess on what the password is. Most likely the passphrase will be the one random string that doesn't get given easily in the executable. But for the sake of gaining understanding of the process, it's good to go over the decompiled code. Figure 7 shows the decompiled code.

The first thing to do is find the entrance into the program. Ghidra already labeled the '_main' at the top but the strings and function calls reaffirm this as the right section to be looking over. Next, thinking back to the string dump, it's likely that the password will be near whatever function takes input. Prior knowledge of programming informs the reverser that after getting the user input, the input string will need to be compared to the right answer. Thankfully Ghidra was able to identify the function 'scanf'.

Inspecting the code section at figure 8, it shows that the variable local_88 contains the user input. This is because it loads a %99s into the RDI register to prepare for the input directly above where scanf is called. RSI

Figure 3.4: Crack Me Header

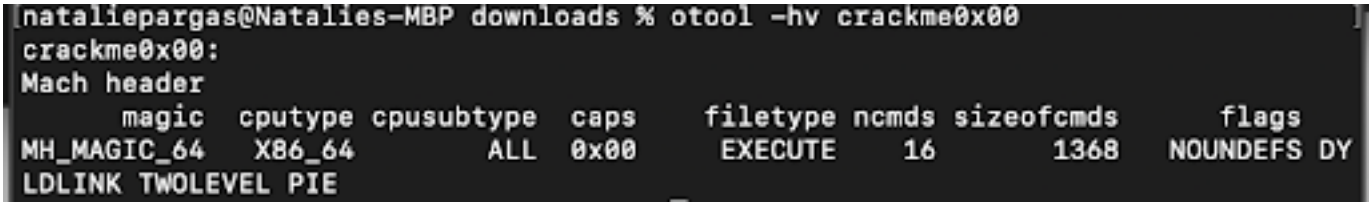
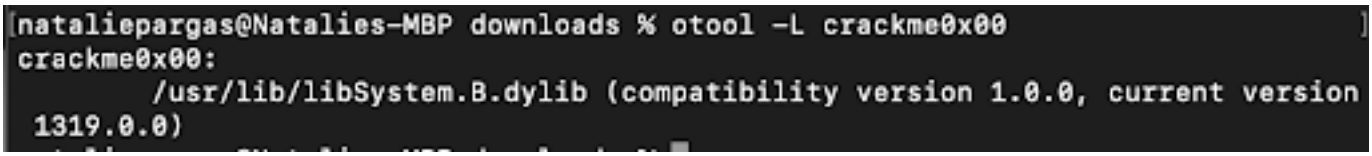


Figure 3.5: Crack Me Libraries



has local_1d loaded into it and that variable is immediately 'strcmp' compared to local_88. Using these clues, one could infer that local_1d contains the password. To verify, a reverser should always retrace and check.

The first instance of local_1d contains the register RAX 9. RAX has the qword ptr ' s_NoxIsTheBest_100003f26' moved into it right above. s_NoxIsTheBest_100003f25 only contains the string "NoxIsTheBest". That string is probably the password, but all that's left is to check. Figure 10 shows the result.

Figure 3.6: Crack Me Strings

```
[nataliepargas@Natalies-MBP downloads % strings crackme0x00
NoxIsTheBest
Crackme Level 0x00 (created by Nox)
Enter the passphrase:
%99s
Congrats on cracking the program!!
Hmmmm maybe try again.
```

Figure 3.7: Crack Me Ghidra Disassembly

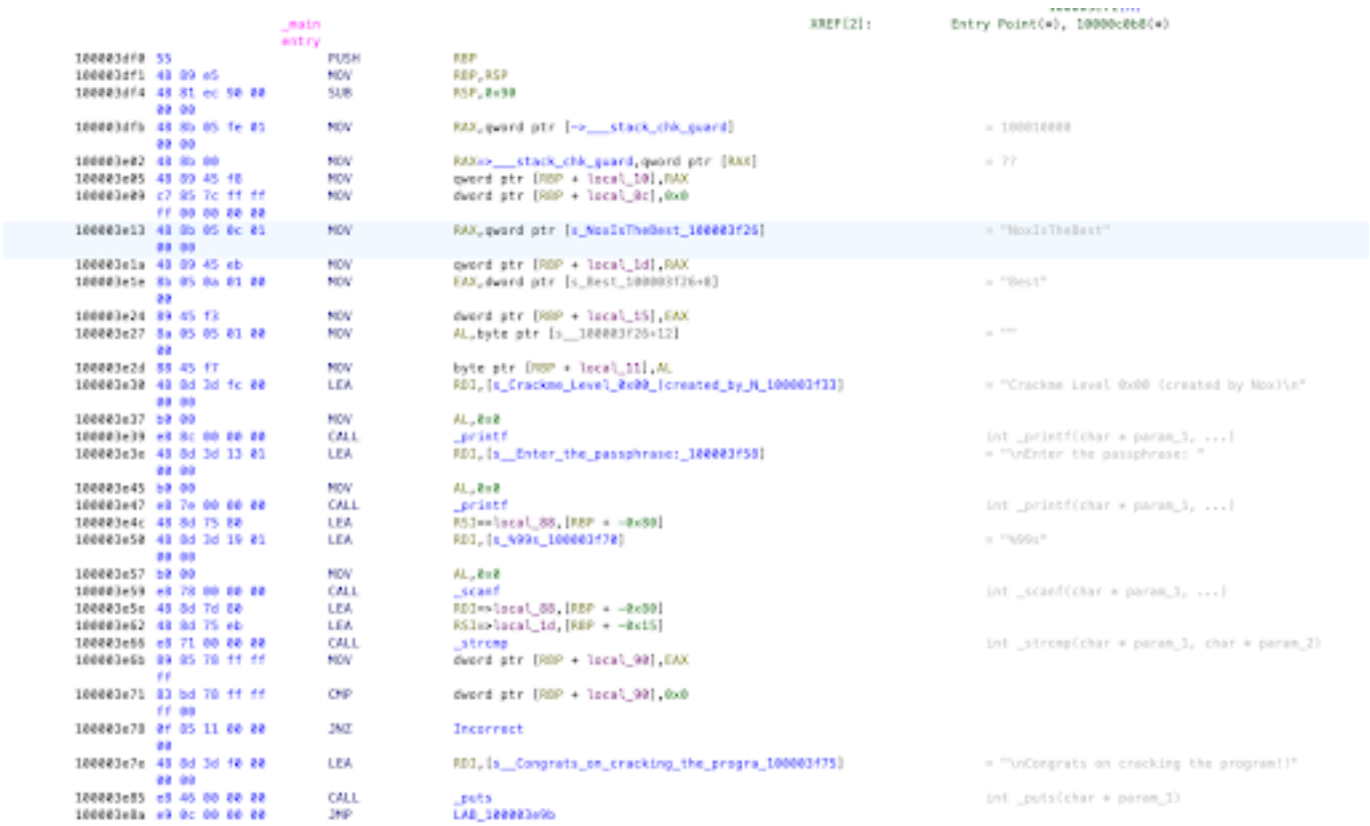


Figure 3.8: Crack Me Code Section 1

Figure 3.9: Crack Me Code Section 2

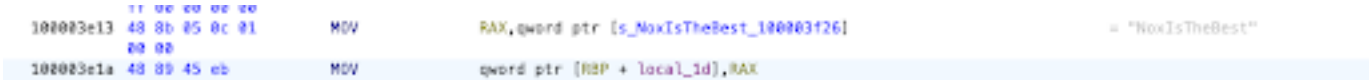
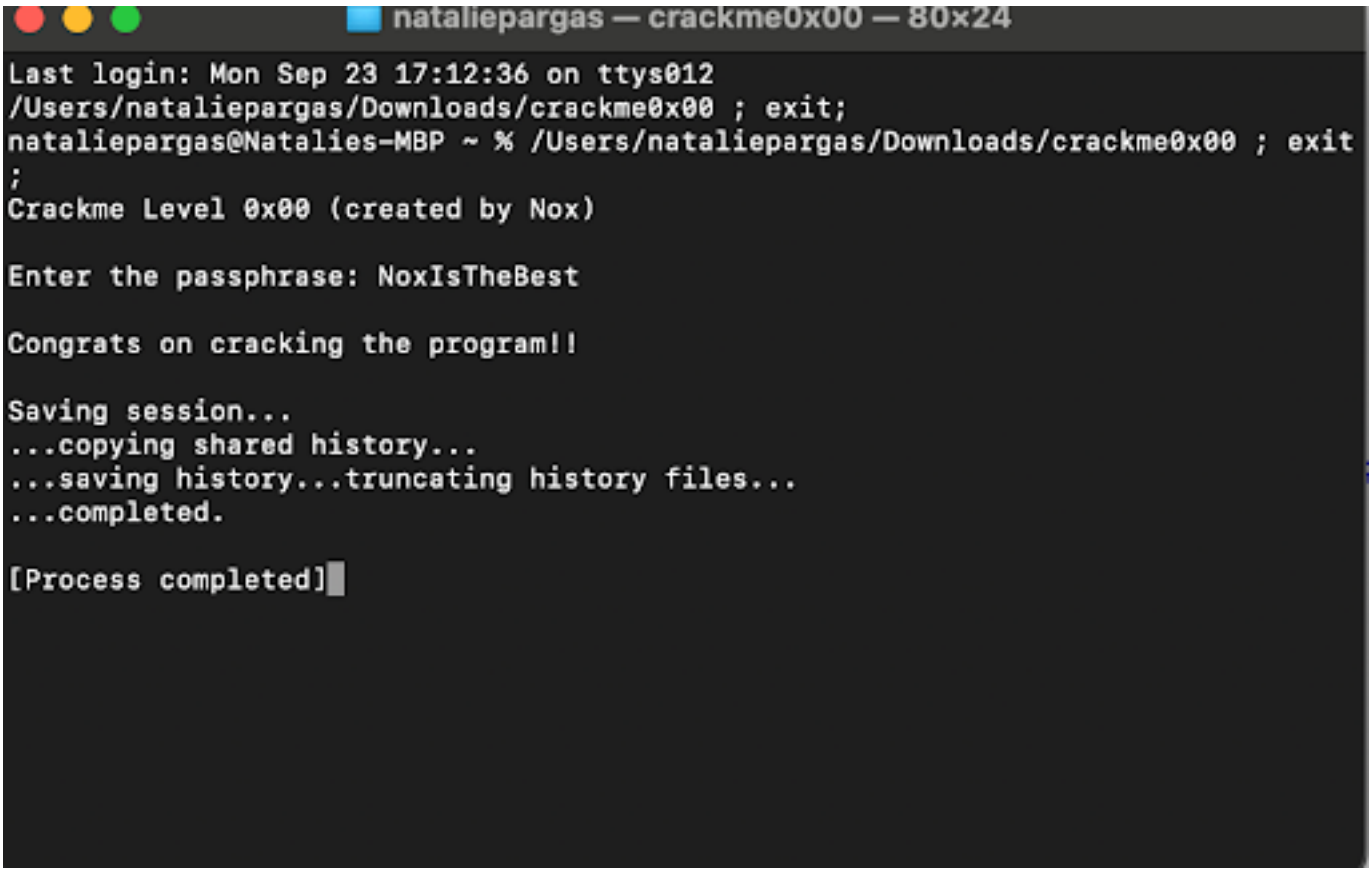


Figure 3.10: Crack Me Code Executable



The attempt was a success and the crack me was successfully reverse engineered!

3.4 Java RE

3.4.1 JVM

3.4.2 Reversing Bytecode

3.4.3 Java Name Obfuscation

3.4.3.1 Obfuscating names for .class file

3.5 Mobile RE

3.5.1 Hybrid Approach

3.5.2 Symbolic Execution

3.5.3 Generating UML Diagram

3.5.4 Android

3.5.4.1 Tools

3.5.4.1.1 Dalvik VM Android

3.5.4.1.2 APK Tools

3.5.5 Apple?

3.5.5.1 Obj C tools

CHAPTER 4

Methods

I would highly recommend that you use BibLaTeX for your bibliography, and that is what this class uses as of August 2022. BibLaTeX supports foreign languages and UTF-8 and provides several styles for formatting references (ACS, Chicago, APA, Turabian). For many people it also is easier to customize than BibTeX with natbib, should you need to customize the formatting of references. BibLaTeX also can make use of the Biber reference processing application which provides support for foreign languages and more sophisticated sorting options than BibTeX. You still process a special .bib file. The .bib file is where you enter your bibliographic information. Sample entries look something like

```
@article{feu02,
author=  {Thomas Feuerstack},
title=   {Introduction to pdf{\TeX{}}},
journal=  {TUGboat},
volume=  {23},
pages=  {329--334},
number=  {3/4},
url=   {http://www.tug.org/TUGboat/Articles/tb23-3-4/tb75feu.pdf},
year=   2002}
```

or

```
@book{mgbcr04,
author=  {Frank Mittelbach and Michel Goossens and
Johannes Braams and David Carlisle and Chris Rowley},
title=   {The \LaTeX\ Companion},
publisher=  {Addison Wesley Professional},
edition=  {2nd},
address=  {New York},
year=   2004}
```

For a Web site I would recommend the following

```
@misc{brei04,
author =   {Jon Breitenbucher},
title =   {{W}ooster related {L}a{T}e{X} files},
url =    {https://woolatex.spaces.wooster.edu},
howpublished= {World Wide Web},
year=   2021,
note =   {Accessed on 09/27/2021}}
```

You can make a reference by typing `\citet{mgbcr04}` to produce Mittelbach et al. [0]. Other forms for citation include `\citep{mgbcr04}` or `\citeauthor {mgbcr04}` to produce [0] or Mittelbach et al. respectively. You can consult Kopka and Daly [0] or Mittelbach et al. [0] to find out how to format entries in the .bib file and what options each reference type has.[1]

Indicies are also relatively easy to create. If I wanted to have Wooster show up in the index, I would enter `Wooster\index{Wooster}` in my source file. I could create a subentry for User Services by entering `User Services\index{Wooster!User Services}`. A subsubentry for Help Desk would be entered as `\index{Wooster!User Services!Help Desk}`.

To create the index, one needs to make sure to uncomment the `\makeindex` command in the `main.tex` file. One also needs to uncomment the makeidx entry in the `styles/packages.tex` file and then run the Makeindex program. Consult Kopka and Daly [0] or Mittelbach et al. [0] for further information.

[1] You could also use footnotes if your department called for that.

4.1 Environment Setup (components)

4.1.1 Virtual Machine

4.1.2 Tools

4.1.2.1 JADX

4.1.2.2 Ghidra

4.1.2.3 APK Tools

4.1.2.4 JBSE (symbolic Java VM)

4.1.2.5 UML Diagram Creator (Ask Dr.Guarnera)

4.1.2.6 Emulator ?

4.2 Scripting

4.3 App Interoperability

References

[0] Jon Breitenbucher. *Wooster related LaTeX files*. World Wide Web. Accessed on 09/27/2021. 2021. URL: http://woolatex.spaces.wooster.edu.

[0] Thomas Feuerstack. "Introduction to pdfTEX". In: *TUGboat* 23.3/4 (2002), pp. 329–334. URL: http://www.tug.org/TUGboat/Articles/tb23-3-4/tb75feu.pdf.

[0] Peter Flynn. "Formatting Information". In: *TUGboat* 23.2 (2003), pp. 115–237. URL: http://www.tug.org/tex-archive/info/beginlatex/beginlatex.letter.pdf.

[0] J. Garcia. *Personal Identification Apparatus*. Tech. rep. U.S. Patent and Trademark Office, 1986.

[0] George Grätzer. *Math into LATEX*. Boston: Birkhäuser, 1996. URL: http://www.tug.org/tex-archive/info/mil/mil.pdf.

[0] Stasinos Konstantopoulos. *BibTeX Entries*. World Wide Web. Accessed on 03/11/2004. 2004. URL: http://odur.let.rug.nl/~konstant/Ac/publications.html.

[0] Helmut Kopka and Patrick W. Daly. *Guide to LATEX*. 4th. New York: Pearson Education, 2003 (page 25).

[0] Frank Mittelbach et al. *The LATEX Companion*. 2nd. New York: Addison Wesley Professional, 2004 (page 25).

[0] Tobias Oetiker et al. "The Not So Short Introduction to LATEX2e". In: (2003). URL: http://www.ctan.org/tex-archive/info/lshort/english/lshort.pdf.

[0] Fred Simmonds. *Network Security: Data and Voice Communications*. McGraw-Hill, 1996.

[0] Simon Singh. *The Code Book*. New York: Anchor Books, 1999.

[0] Wade Trappe and Lawrence C. Washington. *Introduction to Cryptography with Coding Theory*. Upper Saddle River, NJ: Prentice Hall, 2002.

Index

', 30
\,, 28
\!, 32
\(, 27
\), 27
\,, 32
\:, 32
\;, 32
\[, 27
\], 27
^, 29, 31
_, 29, 31

amsmath, 32, 34
array, 33
arrow symbols, 30

blackboard bold, 29
bold symbols, 29, 37
braces, 31

commands
  Big, 32
  big, 32
  Bigg, 32
  bigg, 32
  binom, 31
  bmod, 31
  boldmath, 37
  boldsymbol, 37
  cdots, 32
  ddots, 32
  frac, 31
  genfrac, 31
  idotsint, 33
  iiiint, 33
  iiint, 33
  iint, 33
  int, 31
  label, 28
  ldots, 32
  left, 32
  mathbb, 29
  mathrm, 35
  notag, 34
  overbrace, 30
  overleftarrow, 30
  overline, 30
  overrightarrow, 30
  pmod, 31
  qquad, 28, 32
  quad, 28, 32
  ref, 28
  right, 32
  `right .`, 32, 33
  sqrt, 30
  sum, 31
  surd, 30
  textrm, 35
  underbrace, 30
  underline, 30
  vdots, 32
  vec, 30
  widehat, 30
  widetilde, 30

delimiters, 31
diagonal dots, 32
displaymath, 27

eqnarray, 34
equation, 27
equation systems, 34
exponent, 29
exscale, 32

formulae, 27
fraction, 31

Greek letters, 29

horizontal
  brace, 30
  dots, 32
  line, 30

integral operator, 31

long equations, 34

math font size, 35
math spacing, 32
mathcal, 43
mathematical
  accents, 30
  delimiter, 32
  functions, 30
mathematics, 27
mathscr, 43
modulo function, 31

packages
  alltt, 50
  amsbsy, 37

amsfonts, 29, 43, 50
amsmath, 33, 37, 50
amssymb, 29, 37, 43, 50
amsthm, 50
babel, 50
biblatex, 50
biblatex-chicago, 50
caption, 50
csquotes, 50
eso-pic, 50
eucal, 43, 50
eufrak, 43, 50
fancyhdr, 50
float, 50
floatflt, 50
fontenc, 50
fontspec, 50
geometry, 50
graphicx, 50
hyperref, 50
ifpdf, 50
ifthen, 50
ifxetex, 50
inputenc, 50
lettrine, 50
listings, 50
lmodern, 50
makeidx, 50
maple2e, 50
microtype, 50
pdftex, 50
polyglossia, 50
pxfonts, 50
setspace, 50
subfig, 50
textpos, 50
Ti*k*Z, 50
verbatim, 50
wrapfig, 50
xcolor, 50
xltxtra, 50
xunicode, 50
prime, 30

square root, 30
subscript, 29
sum operator, 31

three dots, 32

vectors, 30
vertical dots, 32

Wooster, 25
User Services, 25
Help Desk, 25
woosterthesis options
scottie, 48
wblack, 48
woostercopyright, 48