FIRST EDITION – CHAPTER 2 REV 1

Kevin Thomas

Forward

I remember a time before the days of the internet where computers were simple yet elegant and beautiful in their design, logic and functionality.

I was a teenager in the 1980’s when I got my first Commodore 64 for Christmas and the first thing I did was tear it out of the box and get it wired up to my console TV as the only thing I needed to see was that blinking console cursor on that blue background with the light blue border.

It was a blank slate. There were no libraries. There were no frameworks. If you wanted to develop something outside of the handful of games that you could get for it, you program it from scratch.

In addition to the C-64 there was a 300 baud modem with a 5.25” floppy disk which read DMBBS 4.8. I quickly read the small documentation that came with it and quickly took over the only phone line in the household.

I set up my BBS or bulletin board system, and called it THE ALLNIGHTER. I set it up and no one called obviously as no one knew it existed. I joined a local CUF group, computer user federation, where I met another DMBBS 4.8 user which helped me network my message boards to him.

At a given time of day my computer would call his and send my messages to his board and I would receive his messages from his board. It was computer networking before the internet and it was simply magic.

Over the next few months he taught me 6502 Assembler which was my first programming language that I ever learned. Every single instruction was given consideration of the hardware and a mastery over the computer was developed as we did literally everything from scratch on the bare metal of the hardware.

Today we live in an environment of large distributed systems where there are thousands of libraries and dozens of containers within pods in a large orchestrated Kubernetes cluster which defines an application.

Between the 1980’s and current, the birth of higher-level languages has made it possible to develop in a timely manner even on the most sophisticated distributed systems.

As we work within a series of large cloud ecosystems, there exists a programming language called Golang, or Go for short, which allows for easy software development to take advantage of multiple cores within a modern CPU in addition to out-of-the-box currency and ease of scale for enterprise-level network and product design.

With every great technology there arises threat actors that exploit such power.

Go can be compiled easily for multiple operating systems producing a single binary. The speed and power of Go makes it an easy choice for modern Malware Developers.

There are literally thousands of books and videos on how to reverse engineer traditional C binaries but little on Go as it is so relatively new.

The aim of this book is to teach basic Go and step-by-step reverse engineer each simple binary to understand what is going on under the hood.

We will develop within the Windows architecture (Intel x64 CISC) as most malware targets this platform by orders of magnitude.

Let’s begin...

Table Of Contents

Chapter 1: Hello Distributed System World

We begin our journey with developing a simple hello world program in Go on a Windows 64-bit OS.

We will then reverse engineer the binary in IDA Free.

Let’s first download Go for Windows.

https://go.dev/doc/install

Let’s download IDA Free.

https://hex-rays.com/ida-free/#download

Let’s download Visual Studio Code which we will use as our integrated development environment.

https://code.visualstudio.com/

Once installed, let’s add the Go extension within VS Code.

https://marketplace.visualstudio.com/items?itemName=golang.go

Let’s create a new project and get started by following the below steps.

New File
main.go

Now let’s populate our **main.go** file with the following.

```
package main

import "fmt"

func main() {
	fmt.Println("hello distributed system world")
}
```

Let’s open up the terminal by click CTRL+SHIFT+` and type the following.

```
go mod init main
go mod tidy
go build
```

Let’s run the binary!

```
.\main.exe
```

Output…

```
hello distributed system world
```

Congratulations! You just created your first hello world code in Go. Time for cake!

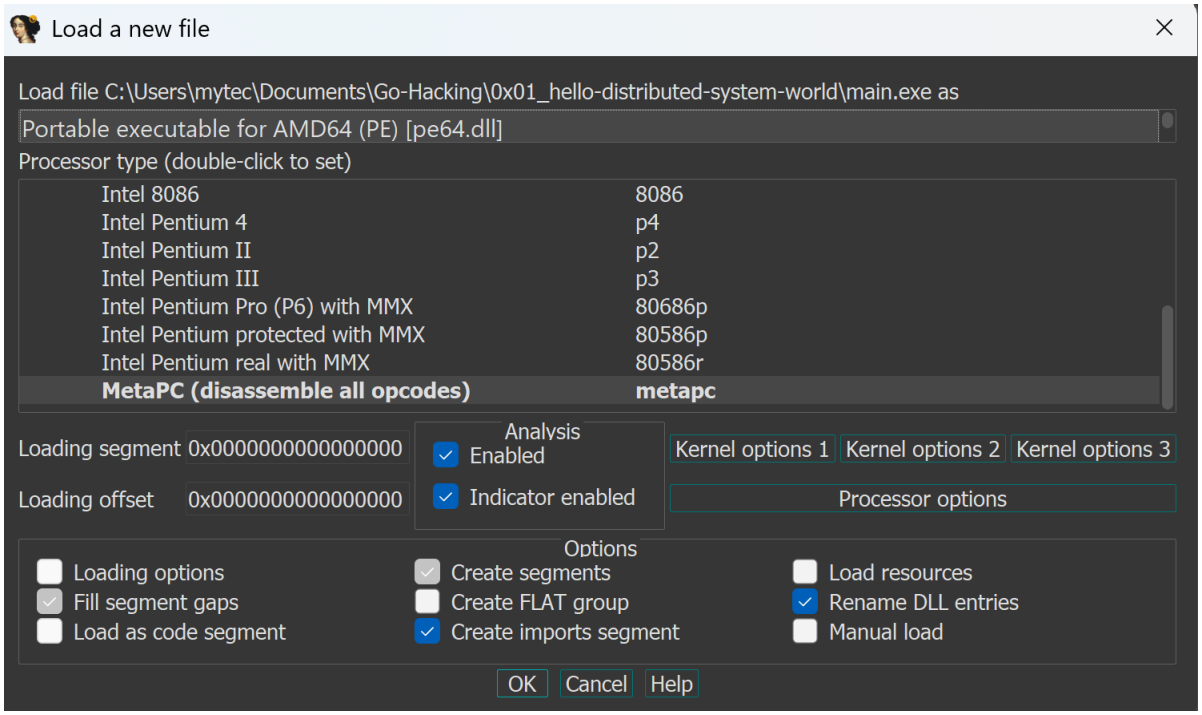
We simply created a hello world style example to get us started.

In our next lesson we will debug this in IDA Free!

Chapter 2: Debugging Hello Distributed System World

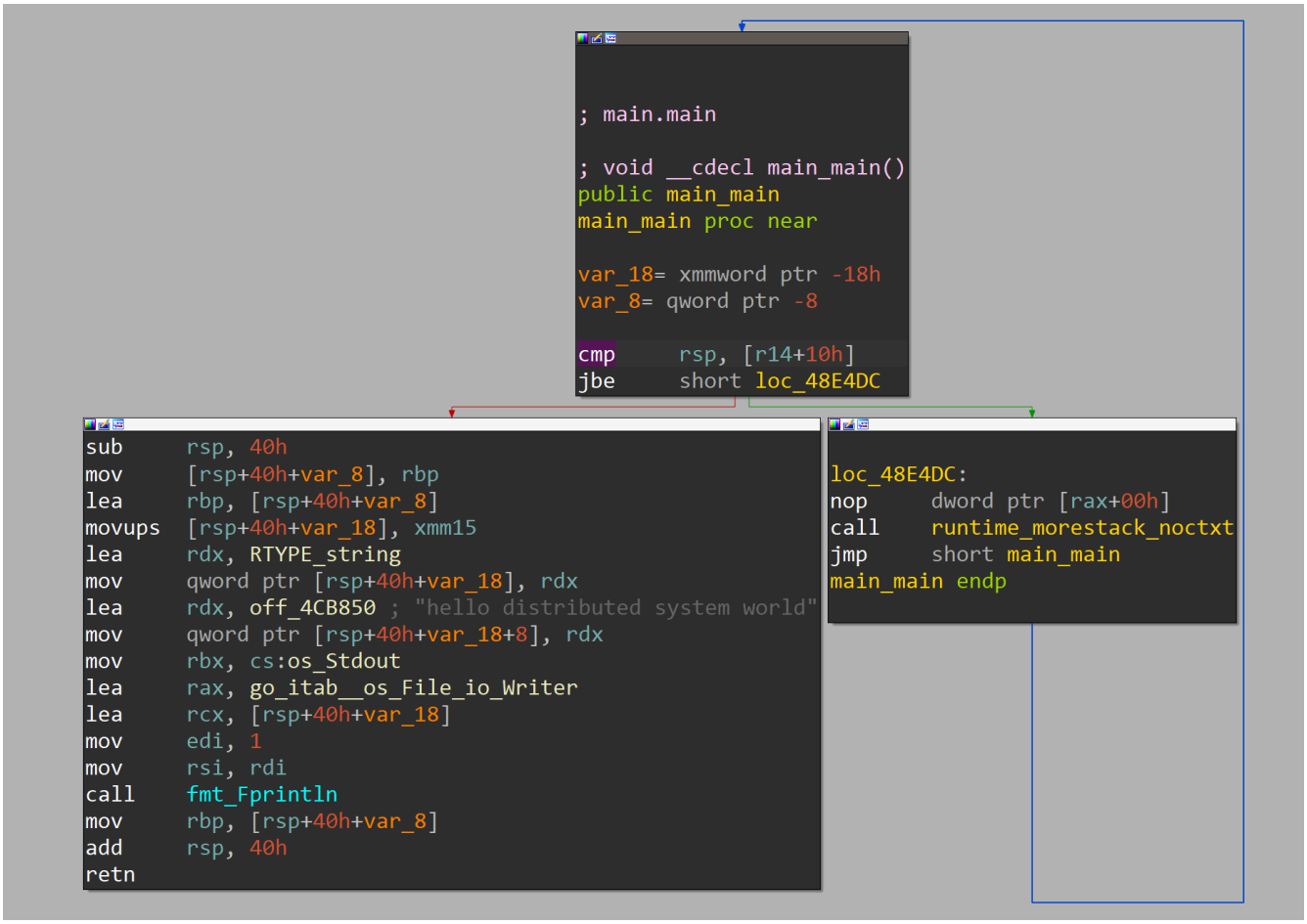
Let’s debug our app within IDA Free.

Open IDA Free and we see the load screen. We can keep all the defaults and simply click OK.



In Go at the assembler level we will need to search for the entry point of our app. This is the *main_main* function. You can use CTRL+F to search.

Now we can double-click on the *main_main* to launch the focus to this function and graph.



We can see in the bottom left box our *“hello distributed system world”* text.

If we double-click on *off_4CB850* it will take us to a new window where the string lives within the binary.

```
.rdata:00000000004CB850 off_4CB850      dq offset aHelloDistribut
.rdata:00000000004CB850                                         ; DATA XREF: main_main+26↑o
.rdata:00000000004CB850                                         ; "hello distributed system world"
.rdata:00000000004CB858                 db  1Eh
```

Here we see something very interesting. Unlike a C binary where the string is terminated by a null character, we see that there is the raw string in a large pool and a *1eh* value which represents the length of the string in hex.

If we double-click on the *“hello distributed system world”* text we will see the string pool within the binary.

```
.rdata:00000000004ADCFD aFreedeferWithD_0 db 'freedefer with d._panic != nil'
.rdata:00000000004ADCFD                                         ; DATA XREF: runtime_freedeferpanic+14↑o
.rdata:00000000004ADD1B aHelloDistribut db 'hello distributed system world'
.rdata:00000000004ADD1B                                         ; DATA XREF: .rdata:off_4CB850↓o
.rdata:00000000004ADD39 aInappropriateI db 'inappropriate ioctl for device'
.rdata:00000000004ADD39                                         ; DATA XREF: .data:0000000000541690↓o
.rdata:00000000004ADD57 aInvalidPointer db 'invalid pointer found on stack'
.rdata:00000000004ADD57                                         ; DATA XREF: runtime_adjustpointers+1BF↑o
.rdata:00000000004ADD75 aNotetsleepWait db 'notetsleep - waitm out of sync'
.rdata:00000000004ADD75                                         ; DATA XREF: runtime_notetsleep_internal:loc_40A906↑o
.rdata:00000000004ADD93 aProtocolWrongT db 'protocol wrong type for socket'
.rdata:00000000004ADD93                                         ; DATA XREF: .data:0000000000541740↓o
.rdata:00000000004ADDB1 aReflectElemOfI_0 db 'reflect: Elem of invalid type '
.rdata:00000000004ADDB1                                         ; DATA XREF: reflect__ptr_rtype_Elem+112↑o
.rdata:00000000004ADDCF aReflectLenOfNo db 'reflect: Len of non-array type'
.rdata:00000000004ADDCF                                         ; DATA XREF: .rdata:off_4CB840↓o
.rdata:00000000004ADDED aRunqputslowQue db 'runqputslow: queue is not full'
.rdata:00000000004ADDED                                         ; DATA XREF: runtime_runqputslow:loc_44353A↑o
.rdata:00000000004ADE0B aRuntimeBadGInC db 'runtime: bad g in cgocallback',0Ah
.rdata:00000000004ADE0B                                         ; DATA XREF: runtime_cgocallbackg+45↑o
.rdata:00000000004ADE29 aRuntimeBadPoin db 'runtime: bad pointer in frame '
.rdata:00000000004ADE29                                         ; DATA XREF: runtime_adjustpointers+15F↑o
.rdata:00000000004ADE47 aRuntimeFoundIn db 'runtime: found in object at *('
.rdata:00000000004ADE47                                         ; DATA XREF: runtime_badPointer+155↑o
.rdata:00000000004ADE65 aRuntimeImpossi_0 db 'runtime: impossible type kind '
.rdata:00000000004ADE65                                         ; DATA XREF: runtime_typesEqual+C25↑o
.rdata:00000000004ADE83 aSocketOperatio db 'socket operation on non-socket'
.rdata:00000000004ADE83                                         ; DATA XREF: .data:0000000000541670↓o
```

All of the strings are within this string pool which is a very different architectural design compared to other languages.

With this basic analysis we have a general idea of what is going on within this simple binary.

These lessons are designed to be short and digestible so that you can code and hack along.

In our next lesson we will learn how to hack this string and force the binary to print something else to the terminal of our choosing.

This will give us the first taste on hacking Go!