FIRST EDITION – 0.1 release

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.

In later chapters we will within a Raspberry Pi 64-bit ARM OS so that you can get a perspective of what hacking that architecture looks like in Golang at the binary level.

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!

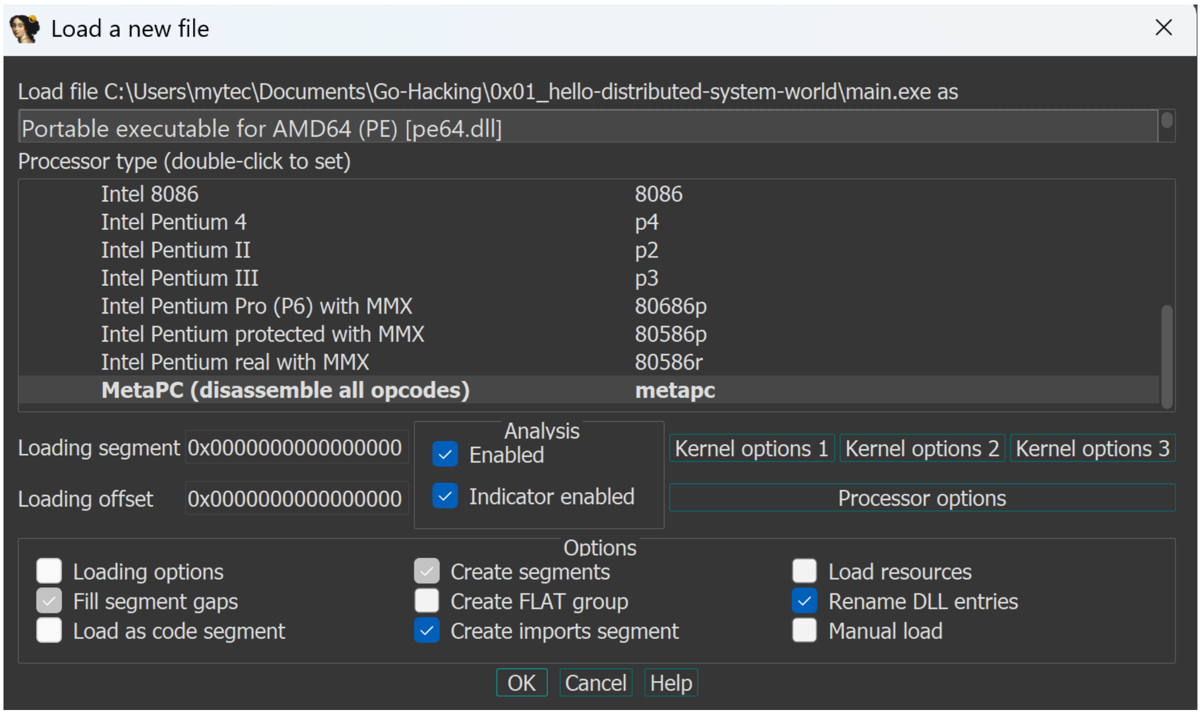
We simply created a hello world style example to get us started.

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

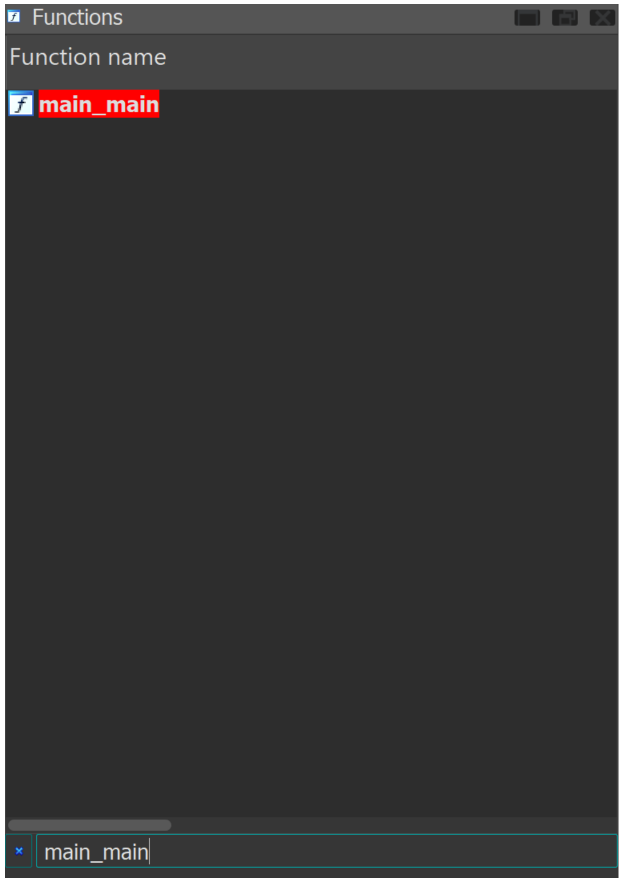
Chapter 2: Debugging Hello Distributed System World

Let's debug our app within IDA Free.

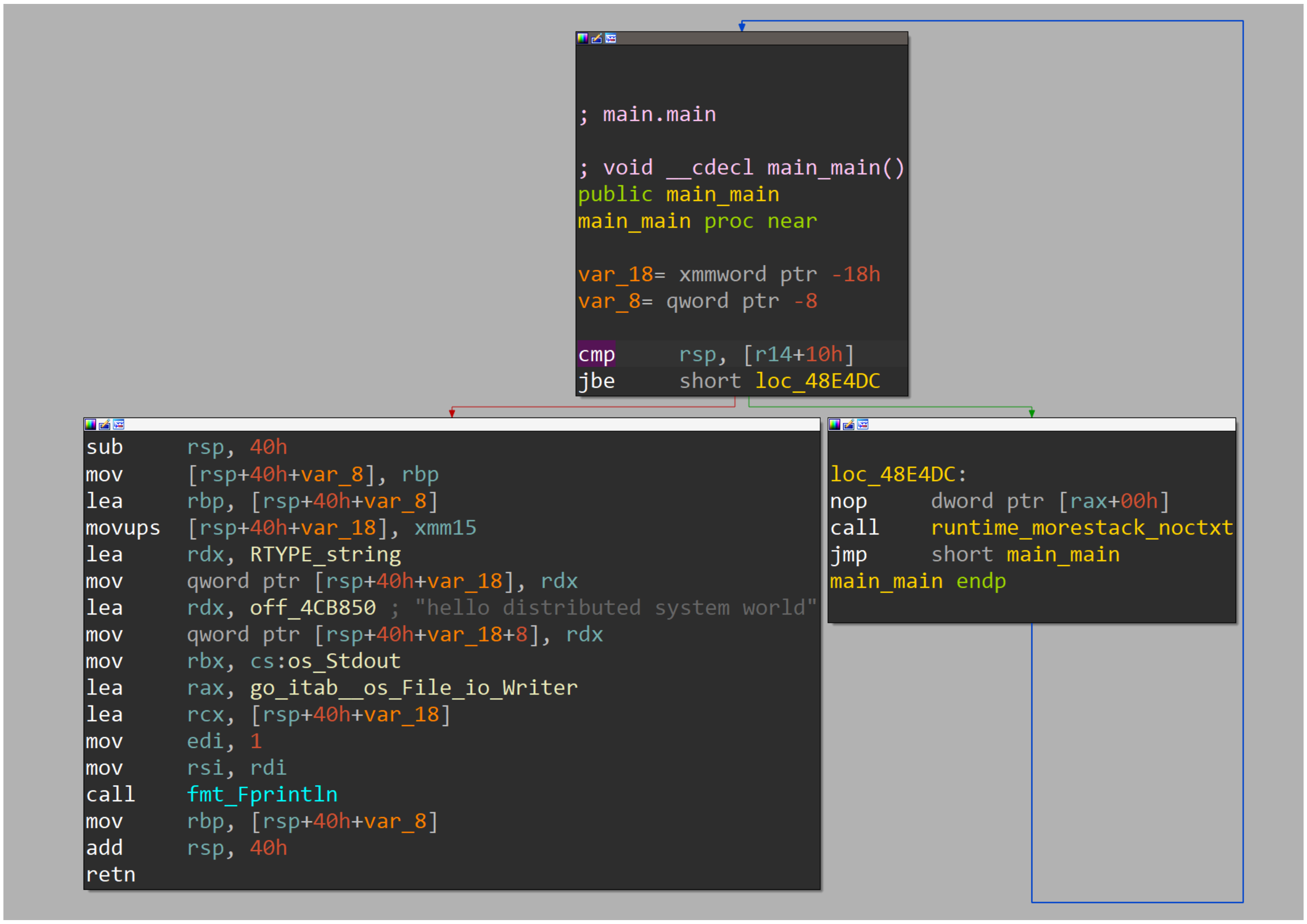
Open IDA Free and we see the load screen. We can keep all the defaults and simply click OK.



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.

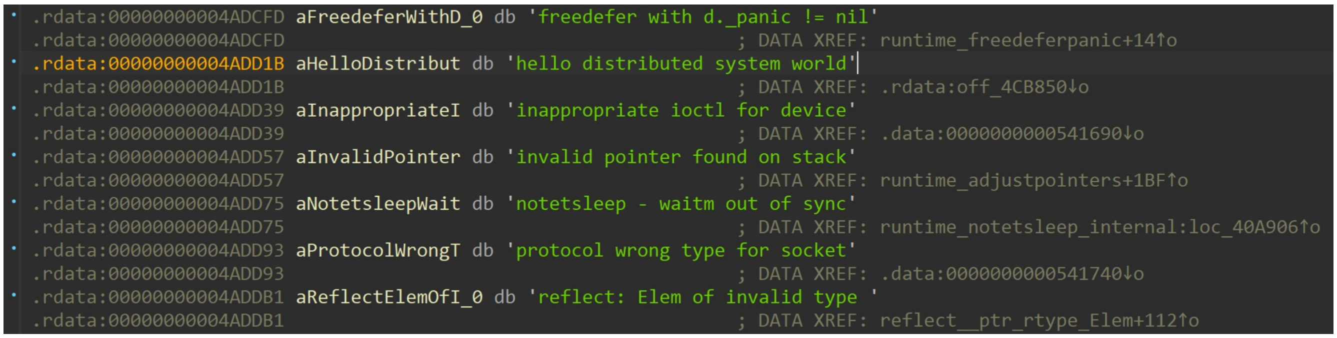
In our next lesson we will learn how to hack this string and force the binary to print something else to the terminal of our choosing.

This will give us the first taste on hacking Go!

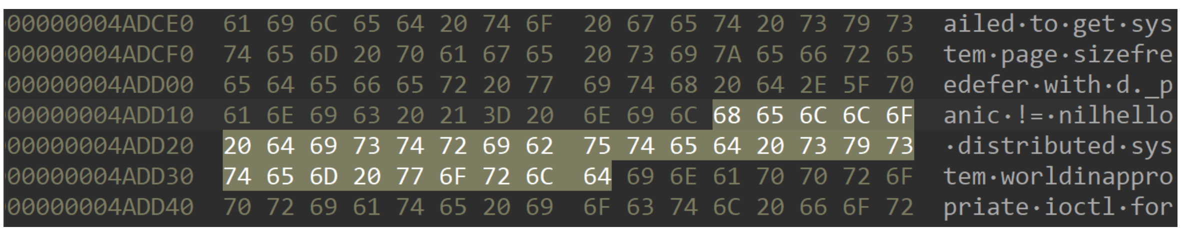
Chapter 3: Hacking Hello Distributed System World

Let's hack our app within IDA Free.

In our last lesson we saw our large string pool. Lets load up IDA and revisit that pool.



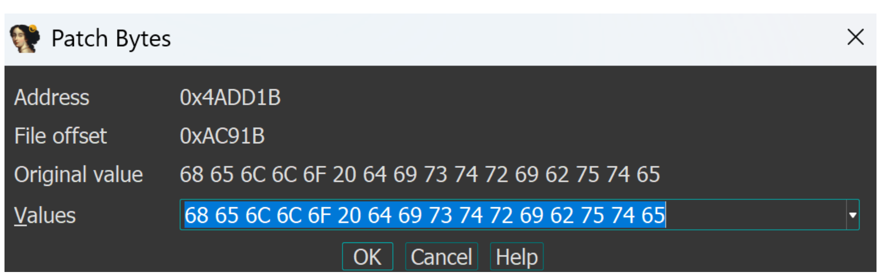
Let's select *Windows* then *Hex View-1*.



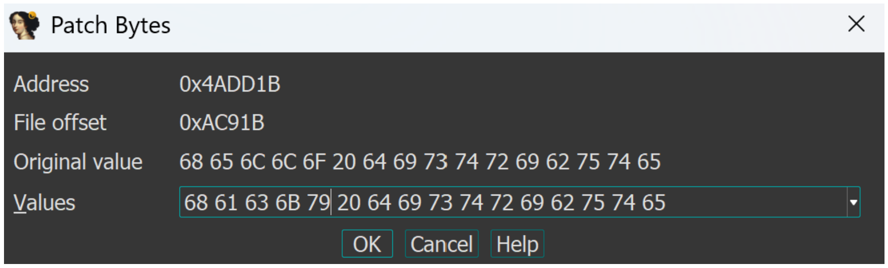
Here we see our string's hex values. It is literally as simple as this as this will be a very short and rewarding chapter.

Select *Edit, Patch program* then *Change byte…*

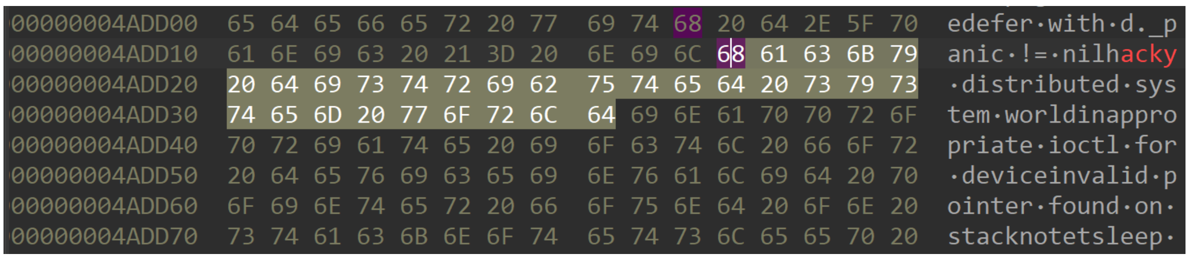
We can use the Ascii Table at https://www.asciitable.com/ to change our string from, *hello distributed system world* to *hacky distributed system world* by simply patching the bytes



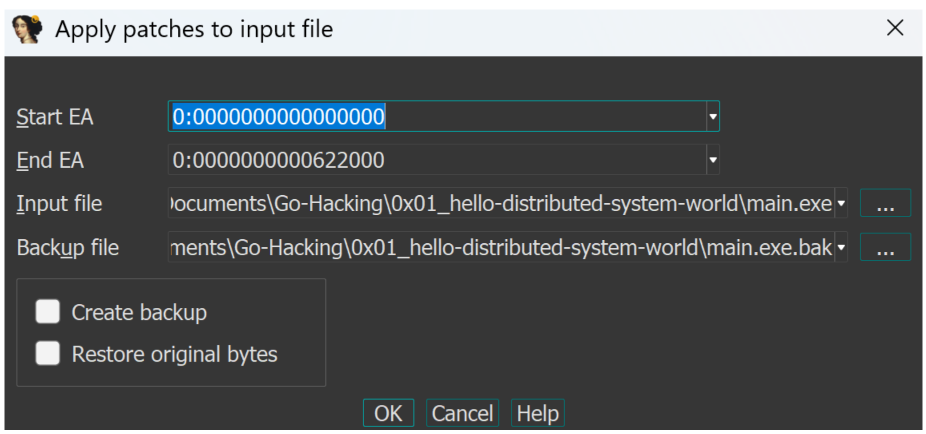
After we change hello to hacky we have the following.



Now we observe the following change.

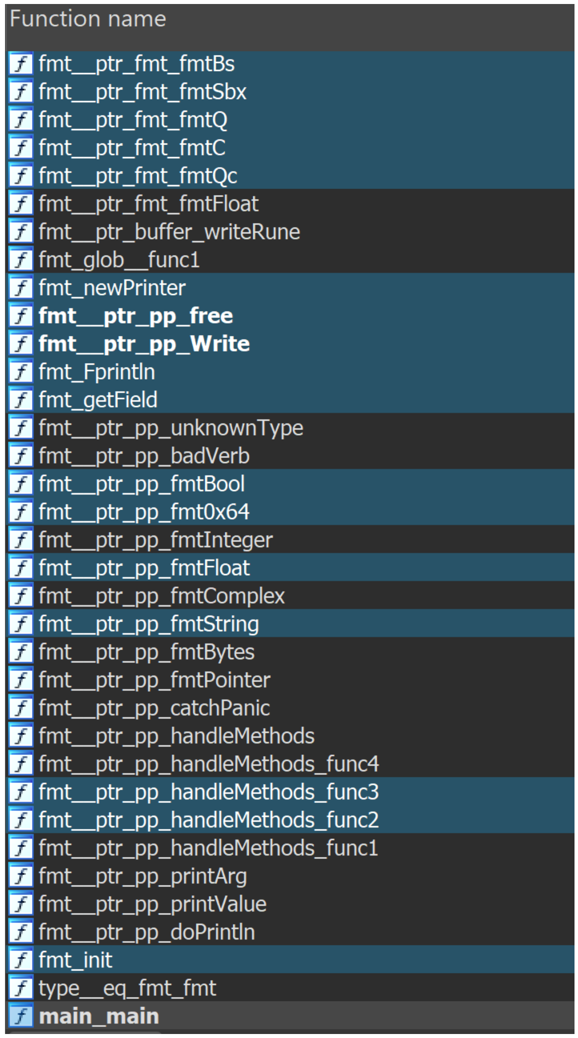


Let’s select *Edit, Patch program, Apply patches to input file…*



Now that we have successfully patched our program, let’s re-run it.

Let’s seek out *main_main* again in the function tree.



Here we can see our revised function.

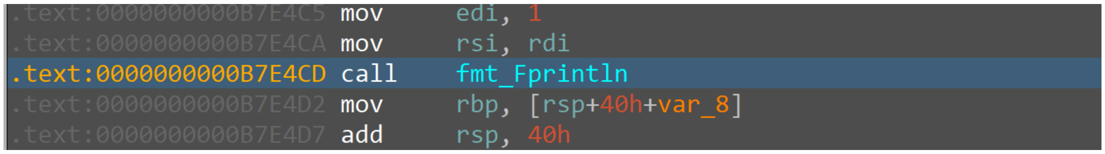
```
.text:000000000048E480                 cmp     rsp, [r14+10h]
.text:000000000048E484                 jbe     short loc_48E4DC
.text:000000000048E486                 sub     rsp, 40h
.text:000000000048E48A                 mov     [rsp+40h+var_8], rbp
.text:000000000048E48F                 lea     rbp, [rsp+40h+var_8]
.text:000000000048E494                 movups  [rsp+40h+var_18], xmm15
.text:000000000048E49A                 lea     rdx, RTYPE_string
.text:000000000048E4A1                 mov     qword ptr [rsp+40h+var_18], rdx
.text:000000000048E4A6                 lea     rdx, off_4CB850 ; "hacky distributed system world"
.text:000000000048E4AD                 mov     qword ptr [rsp+40h+var_18+8], rdx
.text:000000000048E4B2                 mov     rbx, cs:os_Stdout
.text:000000000048E4B9                 lea     rax, go_itab__os_File_io_Writer
.text:000000000048E4C0                 lea     rcx, [rsp+40h+var_18]
.text:000000000048E4C5                 mov     edi, 1
.text:000000000048E4CA                 mov     rsi, rdi
.text:000000000048E4CD                 call    fmt_Fprintln
.text:000000000048E4D2                 mov     rbp, [rsp+40h+var_8]
.text:000000000048E4D7                 add     rsp, 40h
.text:000000000048E4DB                 retn
.text:000000000048E4DC ; ---------------------------------------------------------------------------
.text:000000000048E4DC
.text:000000000048E4DC loc_48E4DC:                             ; CODE XREF: main_main+4↑j
.text:000000000048E4DC                 nop     dword ptr [rax+00h]
.text:000000000048E4E0                 call    runtime_morestack_noctxt
.text:000000000048E4E5                 jmp     short main_main
.text:000000000048E4E5 main_main       endp
```

Let’s set a breakpoint by pressing F2 on the call to *fmt_Fprintln*.

```
.text:000000000048E480                 cmp     rsp, [r14+10h]
.text:000000000048E484                 jbe     short loc_48E4DC
.text:000000000048E486                 sub     rsp, 40h
.text:000000000048E48A                 mov     [rsp+40h+var_8], rbp
.text:000000000048E48F                 lea     rbp, [rsp+40h+var_8]
.text:000000000048E494                 movups  [rsp+40h+var_18], xmm15
.text:000000000048E49A                 lea     rdx, RTYPE_string
.text:000000000048E4A1                 mov     qword ptr [rsp+40h+var_18], rdx
.text:000000000048E4A6                 lea     rdx, off_4CB850 ; "hacky distributed system world"
.text:000000000048E4AD                 mov     qword ptr [rsp+40h+var_18+8], rdx
.text:000000000048E4B2                 mov     rbx, cs:os_Stdout
.text:000000000048E4B9                 lea     rax, go_itab__os_File_io_Writer
.text:000000000048E4C0                 lea     rcx, [rsp+40h+var_18]
.text:000000000048E4C5                 mov     edi, 1
.text:000000000048E4CA                 mov     rsi, rdi
.text:000000000048E4CD                 call    fmt_Fprintln
.text:000000000048E4D2                 mov     rbp, [rsp+40h+var_8]
.text:000000000048E4D7                 add     rsp, 40h
.text:000000000048E4DB                 retn
.text:000000000048E4DC ; ---------------------------------------------------------------------------
.text:000000000048E4DC
.text:000000000048E4DC loc_48E4DC:                             ; CODE XREF: main_main+4↑j
.text:000000000048E4DC                 nop     dword ptr [rax+00h]
.text:000000000048E4E0                 call    runtime_morestack_noctxt
.text:000000000048E4E5                 jmp     short main_main
.text:000000000048E4E5 main_main       endp
```

Finally let’s debug!

We hit our breakpoint.



Let’s step over the call and watch what happens in the console window.

Success!

In our next lesson we will begin to understand primitive types in Go.

Chapter 4: Primitive Types

Golang has three basic types which are *bool*, *numeric* and *string*.

Once a variable is declared it is automatically populated with a null value.

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() {
	b := true
	i := 42
	f := 3.14
	s := "42"

	fmt.Println("bool: ", b)
	fmt.Println("int: ", i)
	fmt.Println("float32: ", f)
	fmt.Println("string: ", s)
}
```

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…

```
bool:  true
int:  42
float32:  3.14
string:  42
```

We can clearly see the respective values and how Golang handles them. In our next lesson we will debug this simple program.

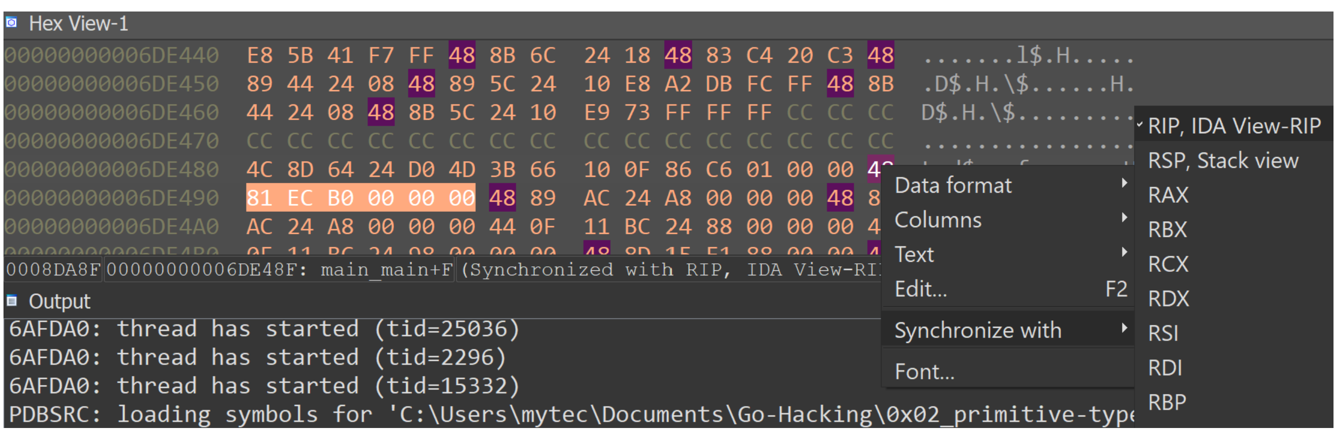
Chapter 5: Debugging Primitive Types

Let’s debug our app within IDA Free.

Let’s locate *main_main* and begin our analysis. In Chapter 2 we went step-by-step to accomplish this so please refer back if needed.

Let’s set a breakpoint on the *lea* instruction.

Before we get started I would sync the hex view with RIP as follows.



This way with each step we can see what is going on in the bin.

We step until the *lea* instruction highlighed below.

```
.text:00000000006DE48F sub     rsp, 0B0h
.text:00000000006DE496 mov     [rsp+0B0h+var_8], rbp
.text:00000000006DE49E lea     rbp, [rsp+0B0h+var_8]
.text:00000000006DE4A6 movups  [rsp+0B0h+var_28], xmm15
.text:00000000006DE4AF movups  [rsp+0B0h+var_18], xmm15
.text:00000000006DE4B8 lea     rdx, RTYPE_string
.text:00000000006DE4BF mov     qword ptr [rsp+0B0h+var_28], rdx
.text:00000000006DE4C7 lea     r8, off_71B8E8  ; "bool: "
.text:00000000006DE4CE mov     qword ptr [rsp+0B0h+var_28+8], r8
.text:00000000006DE4D6 lea     r8, RTYPE_bool
.text:00000000006DE4DD mov     qword ptr [rsp+0B0h+var_18], r8
```

Let’s double-click on the *off_71B8E8* and see what it contains.

```
.rdata:000000000071B8E8 off_71B8E8 dq offset aBool_2        ; DATA XREF: main_main+47↑o
.rdata:000000000071B8E8                                     ; "bool: "
.rdata:000000000071B8F0 db    6
```

We can see there is a string reference here which is, *“bool: “*, which should seem familiar from our last lesson. We also see the *RTYPE_string* which indicates our type for the *“bool: “* and *RTYPE_bool* for the *true* which we will see shortly is a *1*.

We also know how Golang handles string lengths. We can see the value of *6* which indicates the length of the string which as we have mentioned at length differs from other languages completely as there is not null terminated.

When we double-click on *aBool_2* we get taken to the string pool.

```
.rdata:00000000006F82D1 aYezidi db 'Yezidi'        ; DATA XREF: unicode_init+2F91↑o
.rdata:00000000006F82D7 aByte db '[]byte'          ; DATA XREF: fmt__ptr_pp_printArg+38E↑o
.rdata:00000000006F82DD aBool_2 db 'bool: '        ; DATA XREF: .rdata:off_71B8E8↓o
.rdata:00000000006F82E3 aChan_1 db 'chan<-'        ; DATA XREF: reflect_ChanDir_String:loc_6C2F8F↑o
.rdata:00000000006F82E9 aEfence db 'efence'        ; DATA XREF: .data:0000000000790668↓o
.rdata:00000000006F82EF aListen db 'listen'        ; DATA XREF: syscall_init+47A7↑o
.rdata:00000000006F82F5 aObject db 'object'        ; DATA XREF: runtime_badPointer+1AA↑o
.rdata:00000000006F82FB aPopcnt db 'popcnt'        ; DATA XREF: internal_cpu_doinit+202↑o
.rdata:00000000006F82FB                            ; internal_cpu_doinit+220↑o
.rdata:00000000006F8301 aRdtscp db 'rdtscp'        ; DATA XREF: internal_cpu_doinit+A7↑o
.rdata:00000000006F8307 aSelect db 'select'        ; DATA XREF: .data:00000000007904F0↓o
.rdata:00000000006F830D aSocket db 'socket'        ; DATA XREF: syscall_init+4959↑o
.rdata:00000000006F8313 aString_4 db 'string'      ; DATA XREF: .data:0000000000790260↓o
.rdata:00000000006F8313                            ; .data:0000000000790420↓o
.rdata:00000000006F8319 aStruct_1 db 'struct'      ; DATA XREF: .data:0000000000790270↓o
.rdata:00000000006F8319                            ; .data:0000000000790430↓o
.rdata:00000000006F831F aSweep_0 db 'sweep '       ; DATA XREF: runtime_markrootSpans+1E5↑o
.rdata:00000000006F8325 aSysmon db 'sysmon'        ; DATA XREF: .data:0000000000790810↓o
.rdata:00000000006F832B aTimers_0 db 'timers'      ; DATA XREF: .data:00000000007908D0↓o
.rdata:00000000006F8331 aUint16_1 db 'uint16'      ; DATA XREF: .data:0000000000790170↓o
```

We can see the strings are literally up against one another as this gives us deeper insight into Golang.

As mentioned we also drill down into the *true* or *1*.

```
.text:00000000006DE4D6 lea     r8, RTYPE_bool
.text:00000000006DE4DD mov     qword ptr [rsp+0B0h+var_18], r8
.text:00000000006DE4E5 lea     r8, unk_786988
```

Then…

```
.data:0000000000786988 unk_786988 db    1                    ; DATA XREF: main_main+65↑o
```

As we continue to press F7 and single-step we will see the calls to the Golang *Stdout* file descriptor and the *io.Writer* interface which allows you to write data to a wide variety of output streams and in our case stdout.

Finally we call *Fprintln* to print our string into the terminal.

```
.text:00000000006DE4F4 mov     rbx, cs:os_Stdout
.text:00000000006DE4FB lea     rax, go_itab__os_File_io_Writer
.text:00000000006DE502 lea     rcx, [rsp+0B0h+var_28]
.text:00000000006DE50A mov     edi, 2
.text:00000000006DE50F mov     rsi, rdi
.text:00000000006DE512 call    fmt_Fprintln
```

Our result so far…

```
C:\Users\mytec\Documents\G
bool:  true
```

We see the int and as well.

```
.text:00000000006DE523 lea     rdx, RTYPE_string
.text:00000000006DE52A mov     qword ptr [rsp+0B0h+var_48], rdx
.text:00000000006DE52F lea     r8, off_71B8F8   ; "int: "
.text:00000000006DE536 mov     qword ptr [rsp+0B0h+var_48+8], r8
.text:00000000006DE53B mov     eax, 2Ah ; '*'
.text:00000000006DE540 call    runtime_convT64
.text:00000000006DE545 lea     rdx, RTYPE_int
.text:00000000006DE54C mov     qword ptr [rsp+0B0h+var_38], rdx
.text:00000000006DE551 mov     qword ptr [rsp+0B0h+var_38+8], rax
.text:00000000006DE559 mov     rbx, cs:os_Stdout
.text:00000000006DE560 lea     rax, go_itab__os_File_io_Writer
.text:00000000006DE567 lea     rcx, [rsp+0B0h+var_48]
.text:00000000006DE56C mov     edi, 2
.text:00000000006DE571 mov     rsi, rdi
.text:00000000006DE574 call    fmt_Fprintln
```

We see here the literal value of *0x2a* is moved into *EAX* which is the lower half of *RAX* which of course is *42* decimal. We see a call to runtime_convT64 which if you step through it

```
.text:00000000006DE53B mov     eax, 2Ah ; '*'
.text:00000000006DE540 call    runtime_convT64
.text:00000000006DE545 lea     rdx, RTYPE_int
```

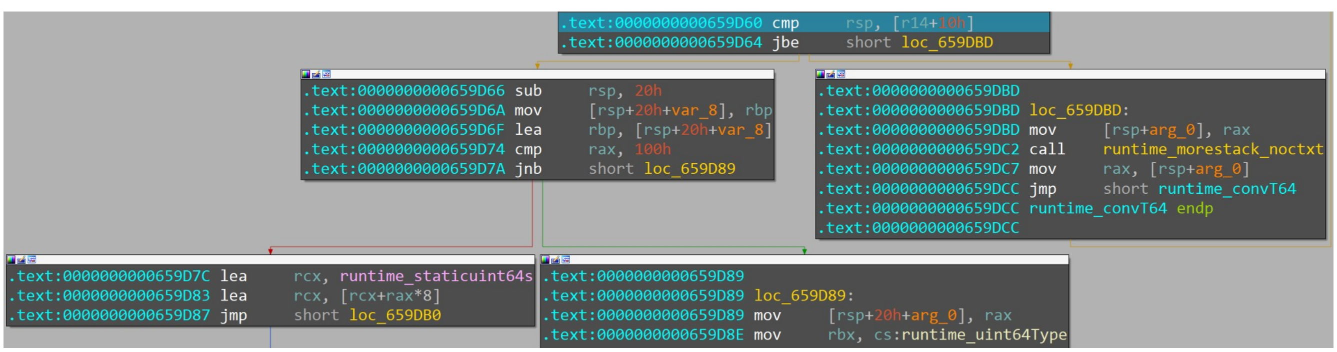
After calling Fprintln…

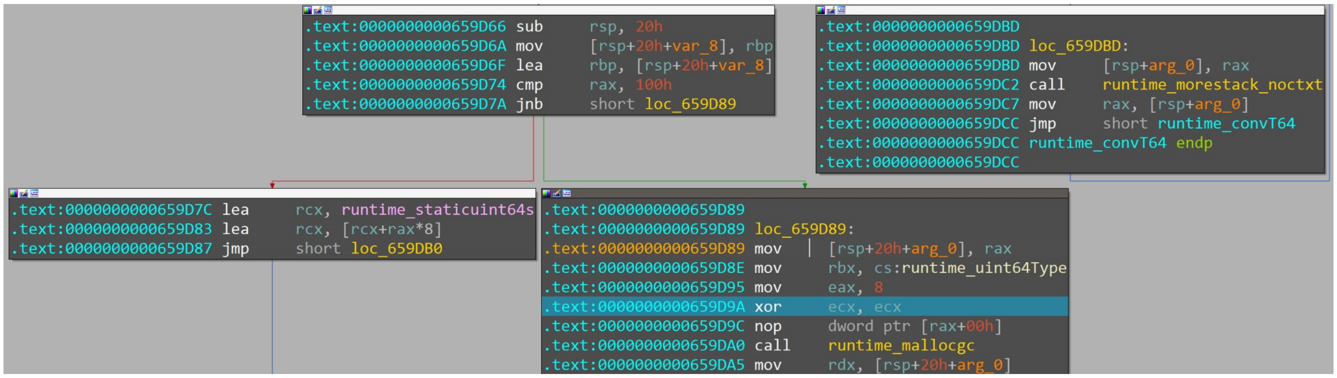
```
C:\Users\mytec\Documents\G
bool:  true
int:   42
```

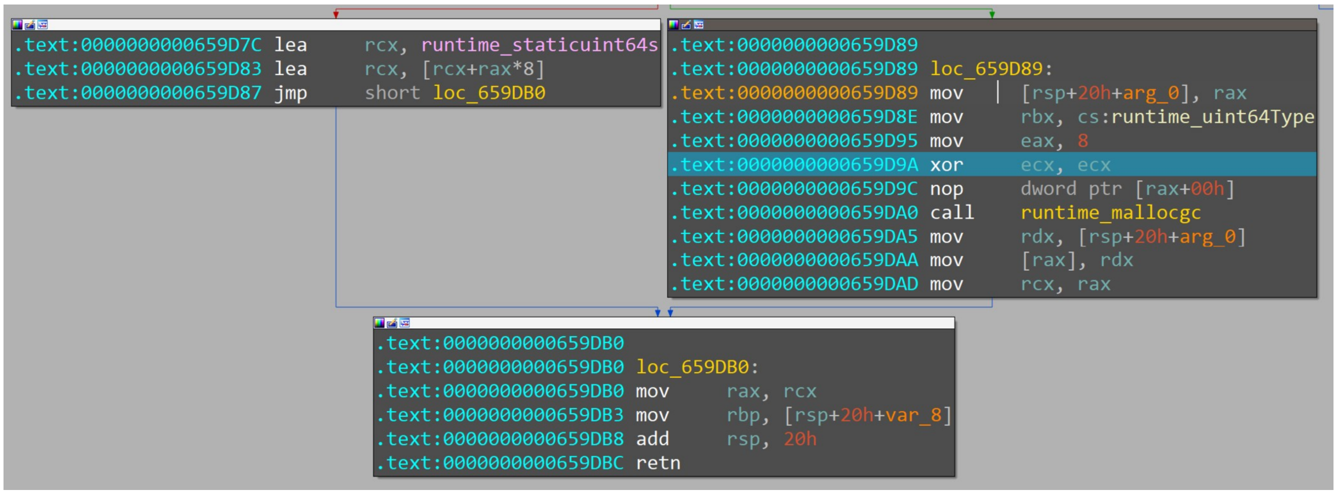
Regarding the float we see a very large number being put into *RAX*.

```
.text:00000000006DE585 lea     rdx, RTYPE_string
.text:00000000006DE58C mov     qword ptr [rsp+0B0h+var_68], rdx
.text:00000000006DE591 lea     r8, off_71B908  ; "float32: "
.text:00000000006DE598 mov     qword ptr [rsp+0B0h+var_68+8], r8
.text:00000000006DE59D mov     rax, 40091EB851EB851Fh
.text:00000000006DE5A7 call    runtime_convT64
.text:00000000006DE5AC lea     rdx, RTYPE_float64
.text:00000000006DE5B3 mov     qword ptr [rsp+0B0h+var_58], rdx
.text:00000000006DE5B8 mov     qword ptr [rsp+0B0h+var_58+8], rax
.text:00000000006DE5BD mov     rbx, cs:os_Stdout
.text:00000000006DE5C4 lea     rax, go_itab__os_File_io_Writer
.text:00000000006DE5CB lea     rcx, [rsp+0B0h+var_68]
.text:00000000006DE5D0 mov     edi, 2
.text:00000000006DE5D5 mov     rsi, rdi
.text:00000000006DE5D8 call    fmt_Fprintln
```

Digging into the call of *runtime_convT64*.

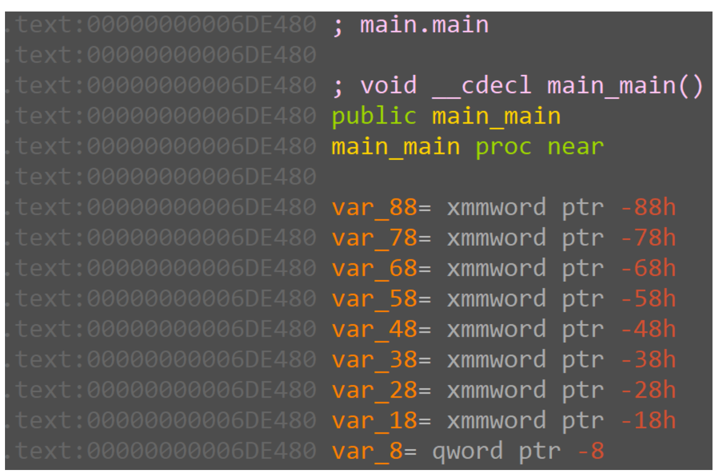






We see call to runtime_*morestack_noctxt* which allocates a new stack for a *goroutne* and a call to the garbage collector which is runtime_mallocgc.

As we continue we have to take a step back to the beginning of main_main where we see a number of xmmwords.



The *xmmword* pointer is a directive that is used to specify the size and type of a memory operand as it indicates the operand is a 128-bit value that is stored in the SSE register or memory.

The *xmmword* pointer is used with other instructions that operate on a floating-point values using the SSE2 SIMD (Single Instruction, Multiple Data) instructions.

In our case it does not do any math on it as it simply handles the conversion of our *3.14* into a printable format.

Keep in mind we used other *xmmword* pointers for our integers as well as well as other numbers however *var_58* and *var_68* is used for our float.

```
.text:00000000006DE5AC lea     rdx, RTYPE_float64
.text:00000000006DE5B3 mov     qword ptr [rsp+0B0h+var_58], rdx
.text:00000000006DE5B8 mov     qword ptr [rsp+0B0h+var_58+8], rax
.text:00000000006DE5BD mov     rbx, cs:os_Stdout
.text:00000000006DE5C4 lea     rax, go_itab__os_File_io_Writer
.text:00000000006DE5CB lea     rcx, [rsp+0B0h+var_68]
.text:00000000006DE5D0 mov     edi, 2
.text:00000000006DE5D5 mov     rsi, rdi
.text:00000000006DE5D8 call    fmt_Fprintln
```

The review of the following within *RDX*, *RAX* and *RCX* create our float.

```
RAX 000000000071BE18 .rdata:go_itab__os_File_io_Writer
RBX 000000C00000A018 debug055:000000C00000A018
RCX 000000C000117F10 debug061:000000C000117F10
RDX 00000000006E62A0 .rdata:RTYPE_float64
RSI 0000000000000002
RDI 0000000000000002
```

The *debug055* and *debug061* refers to the name of code or data at that address.

```
RBX debug055:000000C00000A018 db    0
    debug055:000000C00000A019 db  45h
    debug055:000000C00000A01A db    0
    debug055:000000C00000A01B db    0
    debug055:000000C00000A01C db 0C0h
    debug055:000000C00000A01D db    0
    debug055:000000C00000A01E db    0
    debug055:000000C00000A01F db    0
    debug055:000000C00000A020 db  80h
    debug055:000000C00000A021 db  47h
    debug055:000000C00000A022 db    0
    debug055:000000C00000A023 db    0
    debug055:000000C00000A024 db 0C0h
    debug055:000000C00000A025 db    0
    debug055:000000C00000A026 db    0
    debug055:000000C00000A027 db    0
    debug055:000000C00000A028 db 0A0h
    debug055:000000C00000A029 db 0C3h
    debug055:000000C00000A02A db  78h
```

The rest of the *db* values simply hold *0*.

```
RCX debug061:000000C000117F10 db 0A0h
    debug061:000000C000117F11 db  6Dh ; m
    debug061:000000C000117F12 db  6Eh ; n
    debug061:000000C000117F13 db    0
    debug061:000000C000117F14 db    0
    debug061:000000C000117F15 db    0
    debug061:000000C000117F16 db    0
    debug061:000000C000117F17 db    0
    debug061:000000C000117F18 db    8
    debug061:000000C000117F19 db 0B9h
    debug061:000000C000117F1A db  71h ; q
    debug061:000000C000117F1B db    0
    debug061:000000C000117F1C db    0
    debug061:000000C000117F1D db    0
    debug061:000000C000117F1E db    0
    debug061:000000C000117F1F db    0
    debug061:000000C000117F20 db 0A0h
    debug061:000000C000117F21 db  62h ; b
    debug061:000000C000117F22 db  6Eh ; n
```

We see a similar situation with *RCX*.

```
C:\Users\mytec\Documents\G
bool:  true
int:  42
float32:  3.14
```

We see that the values have been converted to a printable format with the help of Golang *Stdout* file descriptor and the *io.Writer* interface which allows you to write data to a wide variety of output streams as we mentioned.

Finally we see the same behavior with our string. We have done this before in our last debug so we do not have to cover this again but as an exercise please step through the Assembler.

```
.text:00000000006DE5E9 lea     rdx, RTYPE_string
.text:00000000006DE5F0 mov     qword ptr [rsp+0B0h+var_88], rdx
.text:00000000006DE5F5 lea     r8, off_71B918  ; "string: "
.text:00000000006DE5FC mov     qword ptr [rsp+0B0h+var_88+8], r8
.text:00000000006DE601 lea     rax, a42        ; "42"
.text:00000000006DE608 mov     ebx, 2
.text:00000000006DE60D call    runtime_convTstring
.text:00000000006DE612 lea     rdx, RTYPE_string
.text:00000000006DE619 mov     qword ptr [rsp+0B0h+var_78], rdx
.text:00000000006DE61E mov     qword ptr [rsp+0B0h+var_78+8], rax
.text:00000000006DE623 mov     rbx, cs:os_Stdout
.text:00000000006DE62A lea     rax, go_itab__os_File_io_Writer
.text:00000000006DE631 lea     rcx, [rsp+0B0h+var_88]
.text:00000000006DE636 mov     edi, 2
.text:00000000006DE63B mov     rsi, rdi
.text:00000000006DE63E xchg    ax, ax
.text:00000000006DE640 call    fmt_Fprintln
```

```
C:\Users\mytec\Documents\G
bool:  true
int:  42
float32:  3.14
string:  42
```

This now gives you a good handle of how Golang handles its implementation under the hood.

In our next chapter we will hack some of these values.

Chapter 6: Hacking Primitive Types

Let’s hack our app within IDA Free.

Lets load up IDA and put a breakpoint on our *bool* string.

```
lea     r8, off_96B8E8  ; "bool: "
```

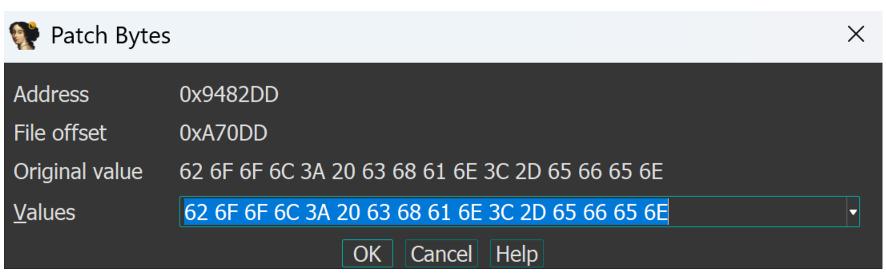
Double clicking on the offset we should see the following as we saw in the last lesson.

```
.rdata:000000000096B8E8 off_96B8E8 dq offset aBool_2          ; DATA XREF: main_main+47↑o
.rdata:000000000096B8E8                                       ; "bool: "
.rdata:000000000096B8F0 db    6
```

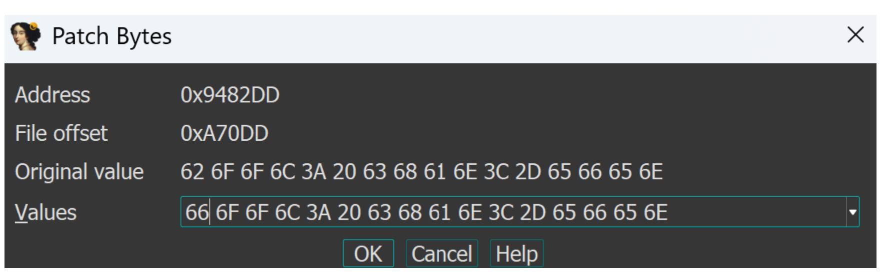
Lets double click on the label.

```
.rdata:00000000009482C5 aUtc11 db 'UTC-11'       ; DATA XREF: .rdata:000000000096D330↓o
.rdata:00000000009482CB aWancho db 'Wancho'      ; DATA XREF: unicode_init+2F11↑o
.rdata:00000000009482D1 aYezidi db 'Yezidi'      ; DATA XREF: unicode_init+2F91↑o
.rdata:00000000009482D7 aByte db '[]byte'        ; DATA XREF: fmt__ptr_pp_printArg+38E↑o
.rdata:00000000009482DD aBool_2 db 'bool: '      ; DATA XREF: .rdata:off_96B8E8↓o
.rdata:00000000009482E3 aChan_1 db 'chan<-'      ; DATA XREF: reflect_ChanDir_String:loc_912F8F↑o
.rdata:00000000009482E9 aEfence db 'efence'      ; DATA XREF: .data:00000000009E0668↓o
.rdata:00000000009482EF aListen db 'listen'      ; DATA XREF: syscall_init+47A7↑o
.rdata:00000000009482F5 aObject db 'object'      ; DATA XREF: runtime_badPointer+1AA↑o
.rdata:00000000009482FB aPopcnt db 'popcnt'      ; DATA XREF: internal_cpu_doinit+202↑o
.rdata:00000000009482FB                          ; internal_cpu_doinit+220↑o
```

We see our familiar string pool. Let’s select *Edit, Patch program, Apply patches to input file…*



0x62 we know is ‘b’ so lets change that to an ‘f’.



Let’s create another breakpoint and dig in again.

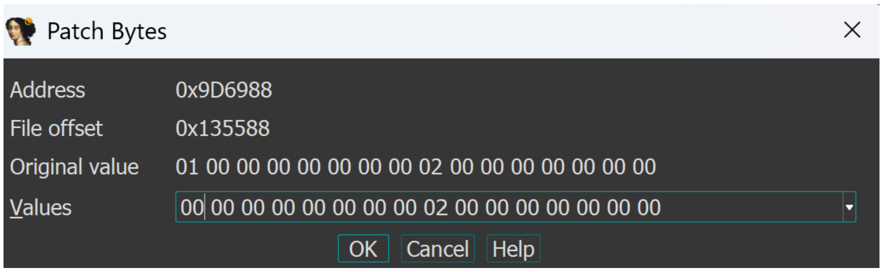
```
lea     r8, RTYPE_bool
mov     qword ptr [rsp+0B0h+var_18], r8
lea     r8, unk_9D6988
```

Here we can double click on the label.

```
.data:00000000009D6988 unk_9D6988 db    1                     ; DATA XREF: main_main+65↑o
.data:00000000009D6989 db    0
.data:00000000009D698A db    0
.data:00000000009D698B db    0
.data:00000000009D698C db    0
.data:00000000009D698D db    0
.data:00000000009D698E db    0
.data:00000000009D698F db    0
.data:00000000009D6990 db    2
.data:00000000009D6991 db    0
.data:00000000009D6992 db    0
.data:00000000009D6993 db    0
.data:00000000009D6994 db    0
.data:00000000009D6995 db    0
.data:00000000009D6996 db    0
.data:00000000009D6997 db    0
.data:00000000009D6998 db    3
.data:00000000009D6999 db    0
.data:00000000009D699A db    0
00135588 00000000009D6988: .data:unk_9D6988 (Synchronized with RIP, Hex View-1)

Hex View-1
00000000009D6930  00 00 02 00 DD A6 02 00  00 A7 02 00 34 B7 02 00  ....4.......
00000000009D6940  40 B7 02 00 1D B8 02 00  20 B8 02 00 A1 CE 02 00  @...............
00000000009D6950  B0 CE 02 00 E0 EB 02 00  00 F8 02 00 1D FA 02 00  ................
00000000009D6960  00 00 03 00 4A 13 03 00  00 01 0E 00 EF 01 0E 00  ....J...........
00000000009D6970  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000000009D6980  00 00 00 00 00 00 00 00  01 00 00 00 00 00 00 00  ................
00000000009D6990  02 00 00 00 00 00 00 00  03 00 00 00 00 00 00 00  ................
```

We can see that we were set to *true* so lets make that a *0* instead.



Let’s select *Edit, Patch program, Apply patches to input file…*

Now that we have successfully patched our program, let’s re-run it.

Ahh yes! You can follow the same technique to hack the rest of the program but this is all you need to get the job done.

In our next chapter we will explore control flow.

Chapter 7: Control Flow

Golang has three basic kinds of basic control flow which is if-else, for and switch-case. We will focus on the if-else as they will not be that different in the assembler.

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() {
	num := 42

	if num == 42 {
		fmt.Println(num, "the answer to life")
	} else {
		fmt.Println(num, "not the answer to life")
	}
}
```

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…

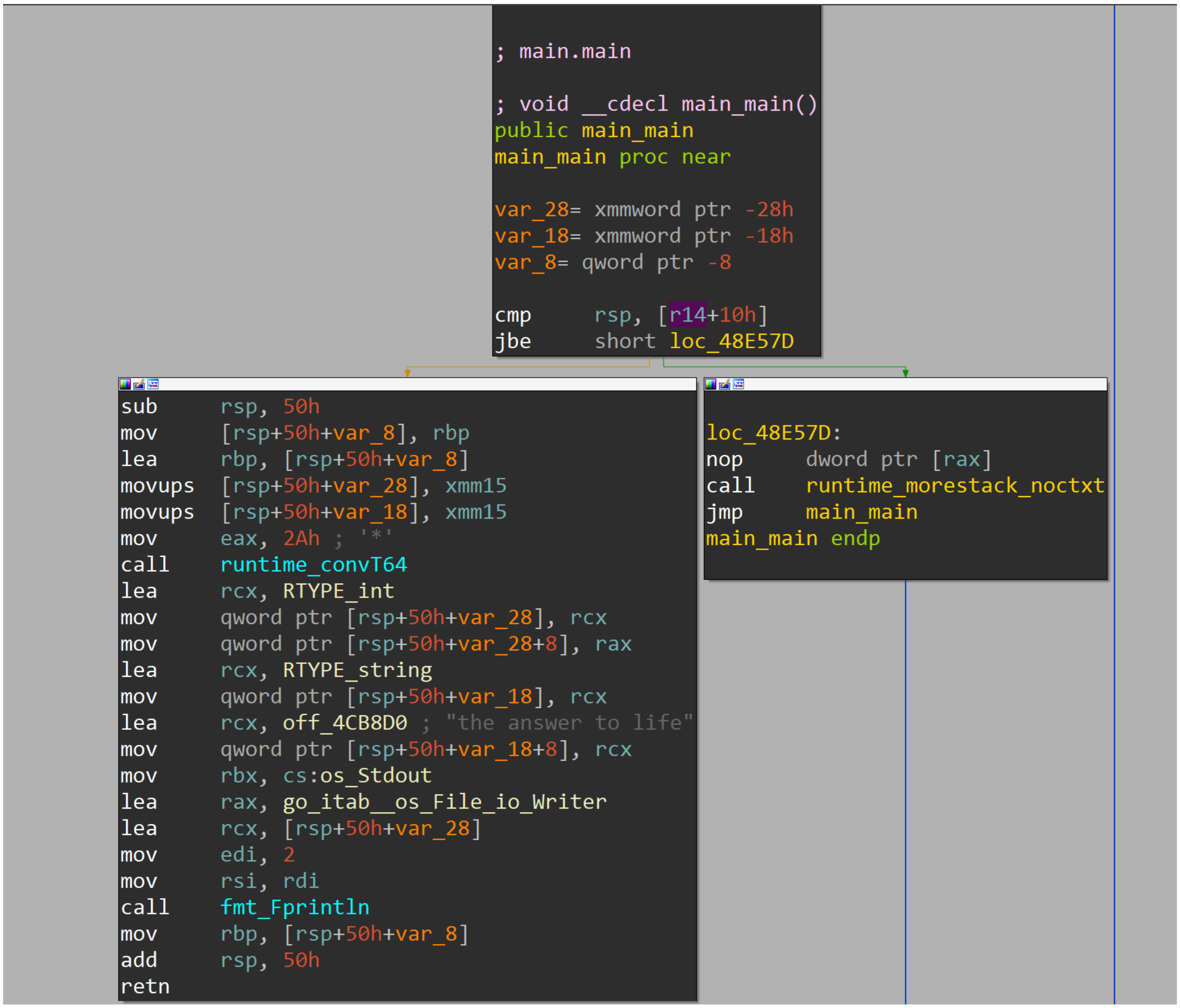
```
42 the answer to life
```

This trivial example demonstrates basic control flow in Go. In our next lesson we will debug this simple program.

Chapter 8: Debugging Control Flow

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. Let's load *main_main*.



So take a moment and look at this disassembly. What do you NOT see that was in our original source code?

If you said, "not the answer to life", you would be correct.

So what happened?

Here the compiler optimized away this else statement as there were no conditions where it would be used therefore we ONLY see. “the answer to life”.

I deliberately created this example to show that everything is not as it seems on the surface. We must be aware of compiler optimization as this will happen over and over in every language.

Let’s close this and mod our original source code to force an option to not optimize away.

Let’s close IDA Free and go back to VS Code.

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 (
	"bufio"
	"fmt"
	"os"
	"strconv"
)

func main() {
	scanner := bufio.NewScanner(os.Stdin)

	fmt.Print("Enter your favorite number: ")
	scanner.Scan()
	input := scanner.Text()

	num, err := strconv.Atoi(input)
	if err != nil {
		fmt.Println("Invalid input")
		return
	}

	if num == 42 {
		fmt.Println(num, "the answer to life")
	} else {
		fmt.Println(num, "not the answer to life")
	}
}
```

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…

```
Enter your favorite number: 42
42 the answer to life

Enter your favorite number: 66
66 not the answer to life

Enter your favorite number: ff
Invalid input
```

Here we see three independent runs of the code to show that if we enter *42* we get, *“the answer to life”* and if we enter in another valid integer we get, *“not the answer to life”* and finally if we enter in a non-integer we have proper error correction.

Let’s debug our new app within IDA Free.

Open IDA Free and we see the load screen. We can keep all the defaults and simply click OK. Let’s load *main_main*.

```
lea     r8, off_4CF588  ; "Enter your favorite number: "
mov     qword ptr [rsp+110h+var_D8+8], r8
mov     rbx, cs:os_Stdout
lea     rax, go_itab__os_File_io_Writer
lea     rcx, [rsp+110h+var_D8]
mov     edi, 1
mov     rsi, rdi
xchg    ax, ax
call    fmt_Fprint
lea     rax, [rsp+110h+var_88]
call    bufio__ptr_Scanner_Scan
```

We start off with our stdin input to obtain a response from a user.

If the user enters in something invalid, non-integer, we hit this block.

```
loc_491D75:
movups  [rsp+110h+var_E8], xmm15
lea     rdx, RTYPE_string
mov     qword ptr [rsp+110h+var_E8], rdx
lea     rdx, off_4CF598 ; "Invalid input"
mov     qword ptr [rsp+110h+var_E8+8], rdx
mov     rbx, cs:os_Stdout
lea     rax, go_itab__os_File_io_Writer
lea     rcx, [rsp+110h+var_E8]
mov     edi, 1
mov     rsi, rdi
call    fmt_Fprintln
mov     rbp, [rsp+110h+var_8]
add     rsp, 110h
retn
```

Otherwise we can see our two other choice blocks.

```
movups  [rsp+110h+var_A8], xmm15
movups  xmmword ptr [rsp+78h], xmm15
mov     eax, 2Ah ; '*'
call    runtime_convT64
lea     rcx, RTYPE_int
mov     qword ptr [rsp+110h+var_A8], rcx
mov     qword ptr [rsp+110h+var_A8+8], rax
lea     rcx, RTYPE_string
mov     [rsp+110h+var_98], rcx
lea     rcx, off_4CF5A8 ; "the answer to life"
mov     [rsp+110h+var_90], rcx
mov     rbx, cs:os_Stdout
lea     rax, go_itab__os_File_io_Writer
lea     rcx, [rsp+110h+var_A8]
mov     edi, 2
mov     rsi, rdi
nop     dword ptr [rax+rax+00h]
call    fmt_Fprintln
jmp     short loc_491D65
```

```
loc_491D07:
movups  [rsp+110h+var_C8], xmm15
movups  [rsp+110h+var_B8], xmm15
call    runtime_convT64
lea     rcx, RTYPE_int
mov     qword ptr [rsp+110h+var_C8], rcx
mov     qword ptr [rsp+110h+var_C8+8], rax
lea     rcx, RTYPE_string
mov     qword ptr [rsp+110h+var_B8], rcx
lea     rcx, off_4CF5B8 ; "not the answer to life"
mov     qword ptr [rsp+110h+var_B8+8], rcx
mov     rbx, cs:os_Stdout
lea     rax, go_itab__os_File_io_Writer
lea     rcx, [rsp+110h+var_C8]
mov     edi, 2
mov     rsi, rdi
nop     dword ptr [rax+00h]
call    fmt_Fprintln
```

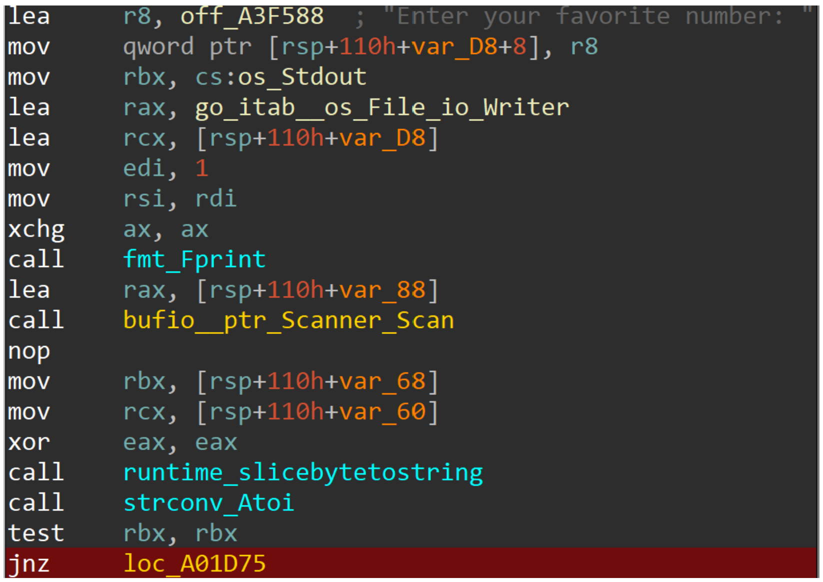
Here we see if they enter in decimal *42*. We see a *mov* into *eax, 2ah*. Hmm, is that *42*? Well yes it is the hex equivalent to *42* and therefore we get, “the answer to life” otherwise if another valid integer we get, “not the answer to life”.

In our next less we will hack this simple application!

Chapter 9: Hacking Control Flow

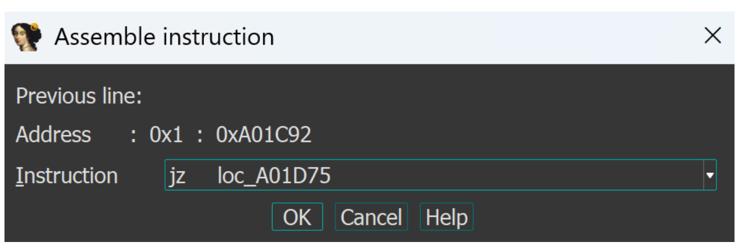
Let’s hack our app within IDA Free.

Lets load up IDA and put a breakpoint on our jump if not zero after the prompt to enter your favorite number.



Let’s patch the assembler to jump if zero so that we get our positive condition of, “the answer to life”.

Let’s select *Edit, Patch program, Assemble…*

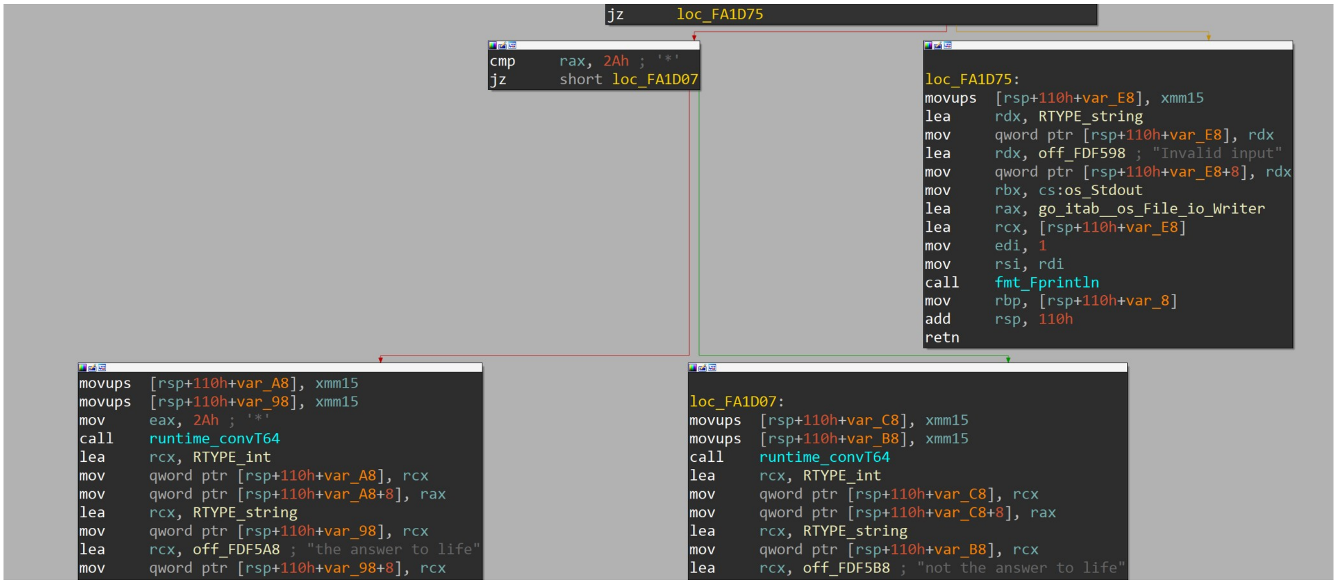


We simply changed the instruction to jump if zero.

We can remove the breakpoint and do the same thing to the *jnz* condition below otherwise if we left it at this point we would get, “not the answer to life”, however we would have still hacked the invalid input check successfully.

Now that both instructions are patched ensure you do the following.

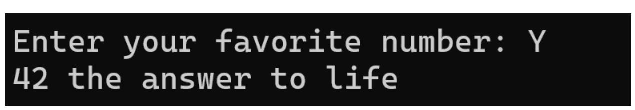
Let's select *Edit, Patch Program, Apply patches to input file…*



We can clearly see here at the top that both instructions have been patched and we can see how the outcome has been altered.

Now lets set a breakpoint on the return at the bottom and at this point it should be our only breakpoint.

Now run the debugger and enter in a *Y* which would be normally invalid and look at the result in the terminal.



Here we can clearly see how we hacked this operation to our liking.

In our next lesson we will cover Advanced Control Flow.

Chapter 10: Advanced Control Flow

Today we will focus on the switch-case control flow.

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() {

	i := 42
	switch i {
	case 42:
		fmt.Println("forty-two")
	case 1337:
		fmt.Println("thirteen thirty seven")
	case 3:
		fmt.Println("three")
	}
}
```

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…

```
forty-two
```

This example demonstrates switch-case control flow in Go. In our next lesson we will debug this simple program.

Chapter 11: Debugging Advanced Control Flow

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. Let's load *main_main*.

This will be a very simple lesson. I want you to take a moment and read the disassembled code. Do you notice anything?

Our original source code utilized a switch statement however the input was hardcoded. If you remember our original control flow lessons we had the same issue where the compiler optimized away the other options as they will never be reached.

I deliberately created this example as you have seen the flow before however wanted to show you what a switch case looked like at the machine level.

Put a breakpoint on the *jbe* within the first block.

You will see that we are comparing *rsp* with what is pointed to at *r14+10h*. We know under normal conditions this will flow to the left block.

So much about reverse engineering is understanding the flow even when compiler optimizations come into play. This is why these lessons are good to experiment with so when you face this in the wild you will have a better understanding of what is going on, bit-by-bit.

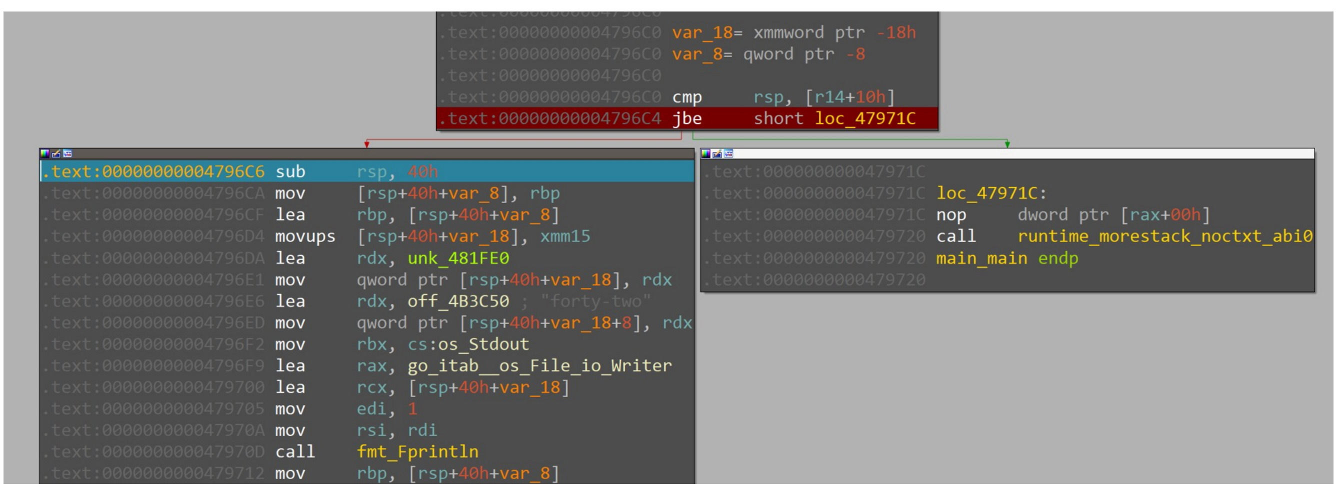
In our next lesson we will hack this to go into the right block.

Chapter 12: Hacking Advanced Control Flow

Let’s hack our app within IDA Free.

Lets load up IDA and put a breakpoint on our jump if below or equal prompt to enter *loc_47971C*.

When we press F7 we see we go into the left block.

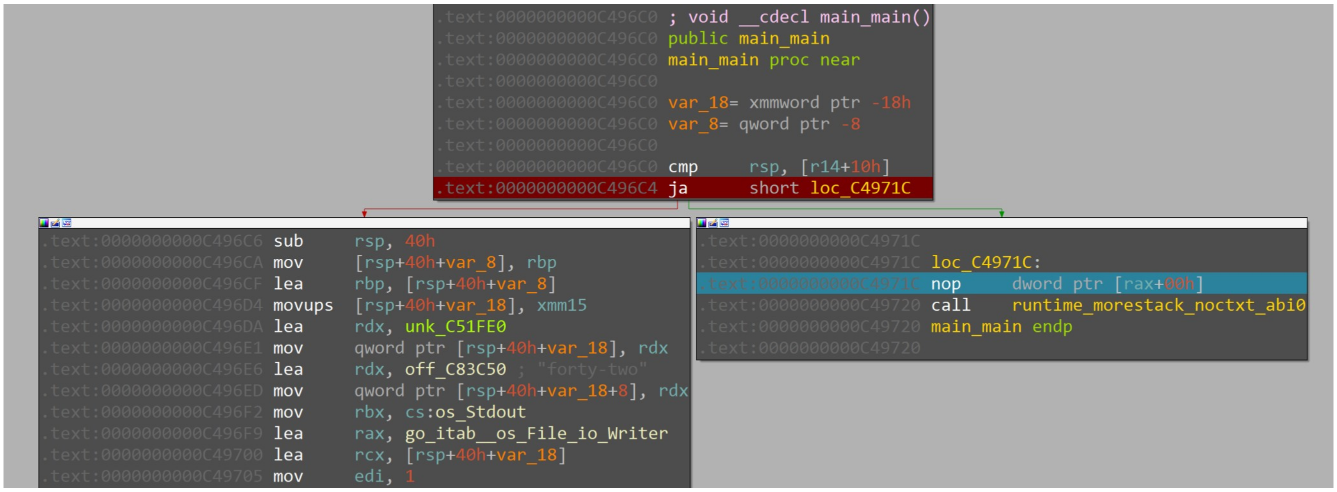


We step through and we know it will print out *forty-two* in our console.

Let’s hack that value to jump if not below or equal, patch and rerun.

Remember you need to patch and apply patches.

It changed to jump if above but that is ok. Let’s step and we can clearly see us moving into the right block.



When we run it through we see it terminate and our console remain empty.

These are simple hacks but taking the time to practice these will help you master the binary manipulation under the hood.

I hope these twelve chapters helped you to get a good handle on hacking with Golang!