Reversing GO binaries like a pro

GO binaries are weird, or at least, that is where this all started out. While delving into some Linux malware named Rex, I came to the realization that I might need to understand more than I wanted to. Just the prior week I had been reversing Linux Lady which was also written in GO, however it was not a stripped binary so it was pretty easy. Clearly the binary was rather large, many extra methods I didn't care about - though I really just didn't understand why. To be honest - I still haven't fully dug into the Golang code and have yet to really write much code in Go, so take this information at face value as some of it might be incorrect; this is just my experience while reversing some ELF Go binaries! If you don't want to read the whole page, or scroll to the bottom to get a link to the full repo, just go here.

To illistrate some of my examples I'm going to use an extremely simple 'Hello, World!' example and also reference the Rex malware. The code and a Make file are extremely simple;

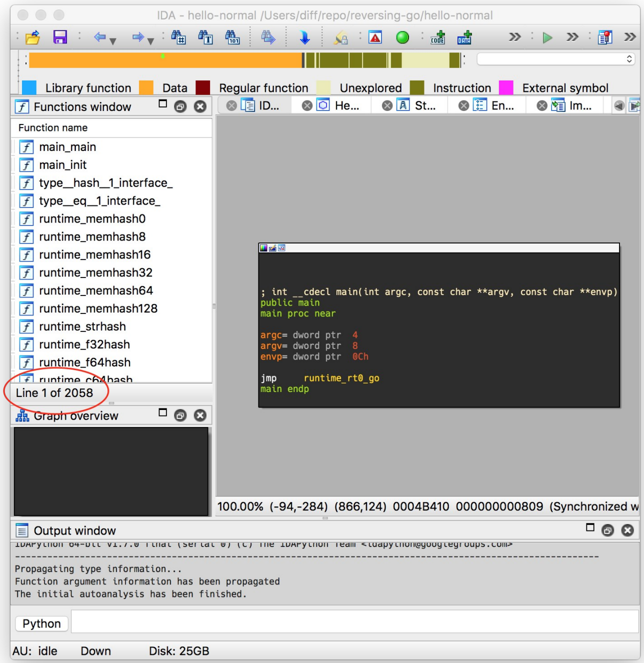
Hello.go

```go
package main
import "fmt"
func main() {
    fmt.Println("Hello, World!")
}
```

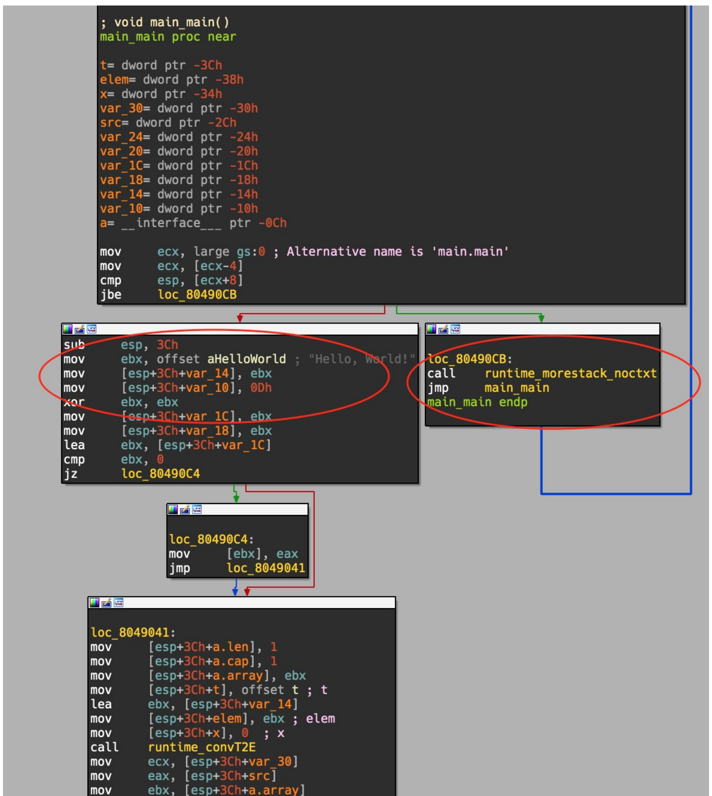
Makefile

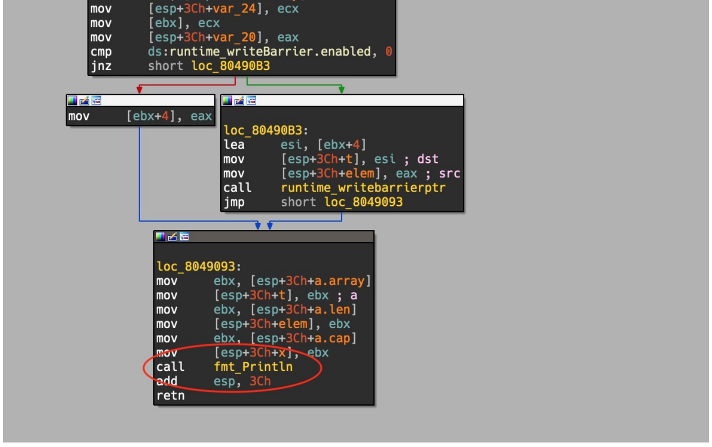
```makefile
all:
 GOOS=linux GOARCH=386 go build -o hello-stripped -ldflags "-s" hello.go
 GOOS=linux GOARCH=386 go build -o hello-normal hello.go
```

Since I'm working on an OSX machine, the above `GOOS` and `GOARCH` variables are explicitly needed to cross-compile this correctly. The first line also added the `ldflags` option to strip the binary. This way we can analyze the same executable both stripped and without being stripped. Copy these files, run `make` and then open up the files in your disassembler of choice, for this blog I'm going to use IDA Pro. If we open up the unstripped binary in IDA Pro we can notice a few quick things;



Well then - our 5 lines of code has turned into over 2058 functions. With all that overhead of what appears to be a runtime, we also have nothing interesting in the `main()` function. If we dig in a bit further we can see that the actual code we're interested in is inside of `main_main`;





This is, well, lots of code that I honestly don't want to look at. The string loading also looks a bit weird - though IDA seems to have done a good job identifying the necessary bits. We can easily see that the string load is actually a set of three `mov`s;

String load

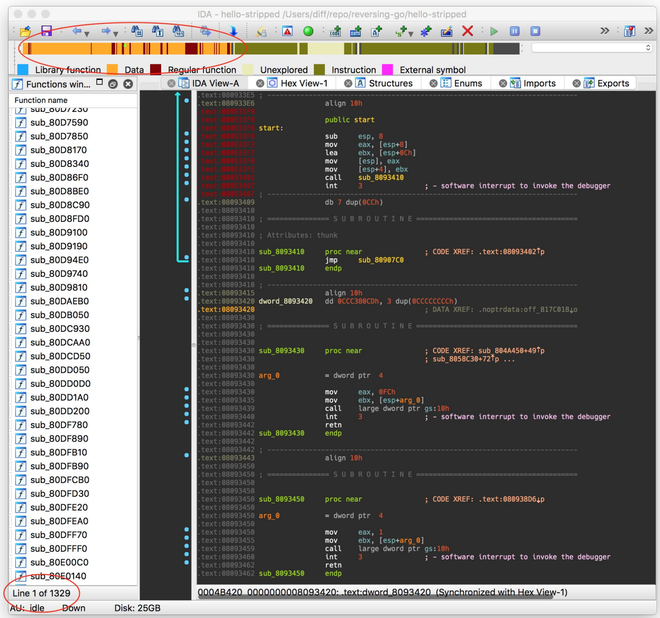
```
mov     ebx, offset aHelloWorld ; "Hello, World!"
mov     [esp+3Ch+var_14], ebx ; Shove string into location
mov     [esp+3Ch+var_10], 0Dh ; length of string
```

This isn't exactly revolutionary, though I can't off the top of my head say that I've seen something like this before. We're also taking note of it as this will come in handle later on. The other tidbit of code which caught my eye was the `runtime_morestack_context` call;

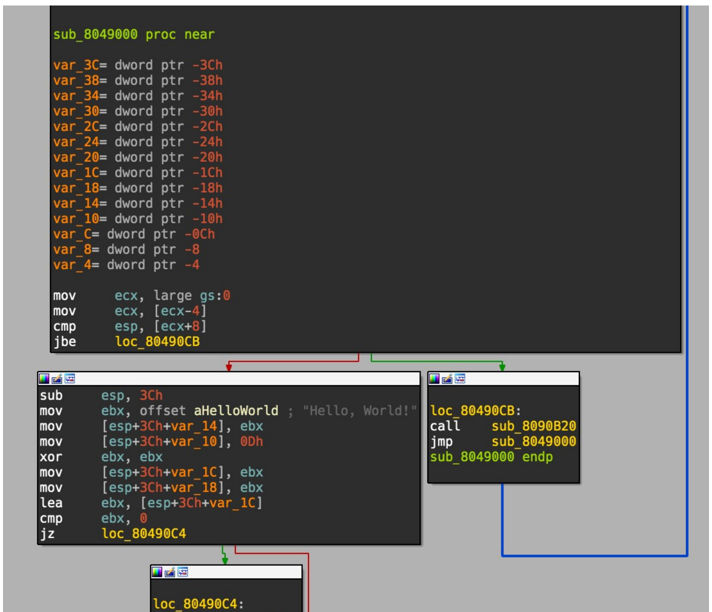
morestack_context

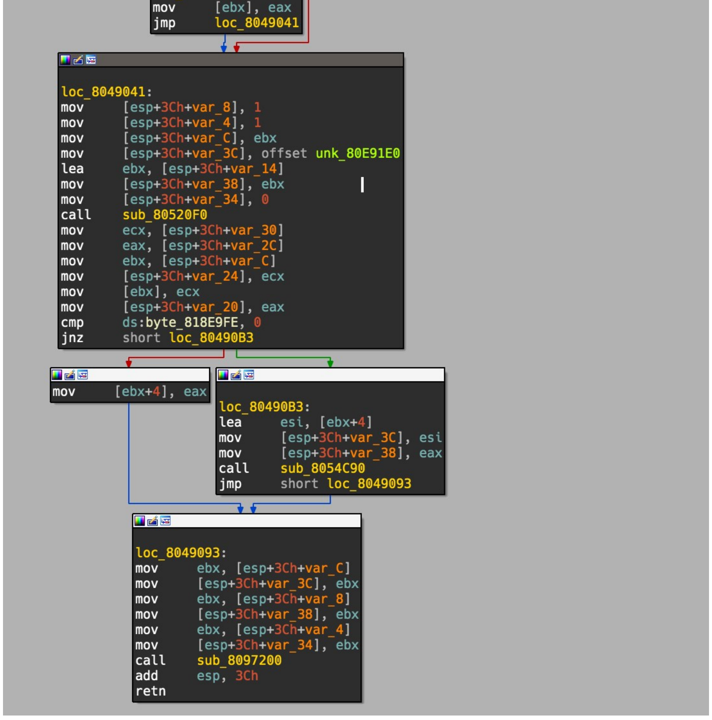
```
loc_80490CB:
call    runtime_morestack_noctxt
jmp     main_main
```

This style block of code appears to always be at the end of functions and it also seems to always loop back up to the top of the same function. This is verified by looking at the cross-references to this function. Ok, now that we know IDA Pro can handle unstripped binaries, lets load the same code but the stripped version this time.



Immediately we see some, well, lets just call them "differences". We have 1329 functions defined and now see some undefined code by looking at the navigator toolbar. Luckily IDA has still been able to find the string load we are looking for, however this function now seems much less friendly to deal with.





We now have no more function names, however - the function names appear to be retained in a specific section of the binary if we do a string search for `main.main` (which would be repesented at `main_main` in the previous screen shots due to how a `.` is interpreted by IDA);

.gopclntab

```
.gopclntab:0813E174                 db  6Dh ; m
.gopclntab:0813E175                 db  61h ; a
.gopclntab:0813E176                 db  69h ; i
.gopclntab:0813E177                 db  6Eh ; n
.gopclntab:0813E178                 db  2Eh ; .
.gopclntab:0813E179                 db  6Dh ; m
.gopclntab:0813E17A                 db  61h ; a
.gopclntab:0813E17B                 db  69h ; i
.gopclntab:0813E17C                 db  6Eh ; n
```

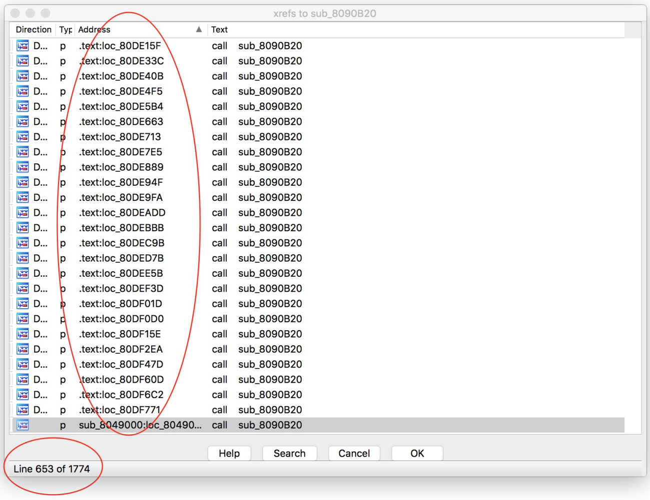
Alright, so it would appear that there is something left over here. After digging into some of the Google results into `gopclntab` and tweet about this - a friendly reverser George (Egor?) Zaytsev showed me his IDA Pro scripts for renaming function and adding type information. After skimming these it was pretty easy to figure out the format of this section so I threw together some functionally to replicate his script. The essential code is shown below, very simply put, we look into the segment `.gopclntab` and skip the first 8 bytes. We then create a pointer (`Qword` or `Dword` dependant on whether the binary is 64bit or not). The first set of data actually gives us the size of the `.gopclntab` table, so we know how far to go into this structure. Now we can start processing the rest of the data which appears to be the `function_offset` followed by the (function) `name_offset`). As we create pointers to these offsets and also tell IDA to create the strings, we just need to ensure we don't pass `MakeString` any bad characters so we use the `clean_function_name` function to strip out any badness.

renamer.py

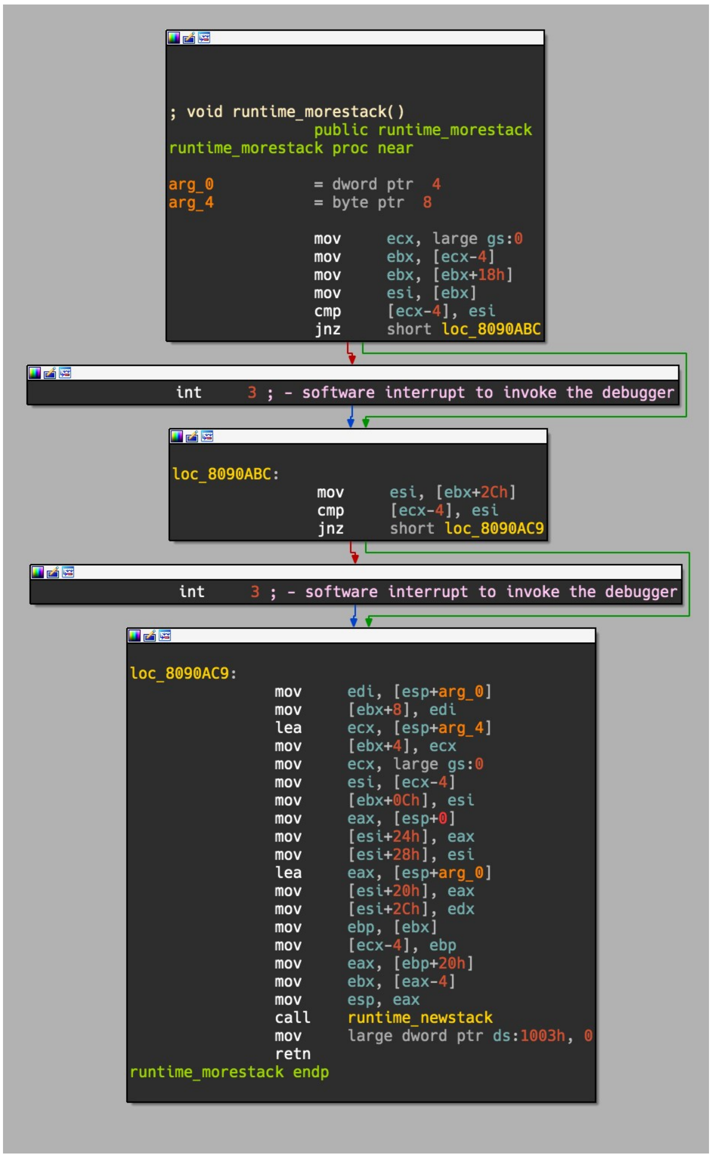
```python
def create_pointer(addr, force_size=None):
    if force_size is not 4 and (idaapi.get_inf_structure().is_64bit() or force_size is 8):
        MakeQword(addr)
        return Qword(addr), 8
    else:
        MakeDword(addr)
        return Dword(addr), 4

STRIP_CHARS = [ '(', ')', '[', ']', '{', '}', ' ', '"' ]
REPLACE_CHARS = ['.', '*', '-', ',', ';', ':', '/', '\xb7' ]
def clean_function_name(str):
    # Kill generic 'bad' characters
    str = filter(lambda x: x in string.printable, str)

    for c in STRIP_CHARS:
        str = str.replace(c, '')

    for c in REPLACE_CHARS:
        str = str.replace(c, '_')

    return str

def renamer_init():
    renamed = 0

    gopclntab = ida_segment.get_segm_by_name('.gopclntab')
    if gopclntab is not None:
        # Skip unimportant header and goto section size
        addr = gopclntab.startEA + 8
        size, addr_size = create_pointer(addr)
        addr += addr_size

        # Unsure if this end is correct
        early_end = addr + (size * addr_size * 2)
        while addr < early_end:
            func_offset, addr_size = create_pointer(addr)
            name_offset, addr_size = create_pointer(addr + addr_size)
            addr += addr_size * 2

            func_name_addr = Dword(name_offset + gopclntab.startEA + addr_size) + gopclntab.startEA
            func_name = GetString(func_name_addr)
            MakeStr(func_name_addr, func_name_addr + len(func_name))
            appended = clean_func_name = clean_function_name(func_name)
            debug('Going to remap function at 0x%x with %s - cleaned up as %s' % (func_offset, func_name,

            if ida_funcs.get_func_name(func_offset) is not None:
                if MakeName(func_offset, clean_func_name):
                    renamed += 1
                else:
                    error('clean_func_name error %s' % clean_func_name)

    return renamed
```

```
def main():
    renamed = renamer_init()
    info('Found and successfully renamed %d functions!' % renamed)
```

The above code won't actually run yet (don't worry full code available in this repo) but it is hopefully simple enough to read through and understand the process. However, this still doesn't solve the problem that IDA Pro doesn't know *all* the functions. So this is going to create pointers which aren't being referenced anywhere. We do know the beginning of functions now, however I ended up seeing (what I think is) an easier way to define all the functions in the application. We can define all the functions by utilizing `runtime_morestack_noctxt` function. Since every function utilizes this (basically, there is an edgecase it turns out), if we find this function and traverse backwards to the cross references to this function, then we will know where every function exists. So what, right? We already know where every function started from the segment we just parsed above, right? Ah, well - now we know the end of the function *and* the next instruction after the call to `runtime_morestack_noctxt` gives us a jump to the top of the function. This means we should quickly be able to give the bounds of the start and stop of a function, which is required by IDA, while seperating this from the parsing of the function names. If we open up the window for cross references to the function `runtime_morestack_noctxt` we see there are many more undefined sections calling into this. 1774 in total things reference this function, which is up from the 1329 functions IDA has already defined for us, this is highlighted by the image below;



After digging into mutliple binaries we can see the `runtime_morestack_noctext` will always call into `runtime_morestack` (with context). This is the edgecase I was referencing before, so between these two functions we should be able to see cross refereneces to ever other function used in the binary. Looking at the larger of the two functions, `runtime_more_stack` , of multiple binaries tends to have an interesting layout;



The part which stuck out to me was `mov large dword ptr ds:1003h, 0` - this appeared to be rather constant in all 64bit binaries I saw. So after cross compiling a few more I noticed that 32bit binaries used `mov qword ptr ds:1003h, 0`, so we will be hunting for this pattern to create a "hook" for traversing backwards on. Lucky for us, I haven't seen an instance where IDA Pro fails to define this specific function, we don't really need to spend much brain power mapping it out or defining it outselves. So, enough talk, lets write some code to find this function;

find_runtime_morestack.py

```
def create_runtime_ms():
    debug('Attempting to find runtime_morestack function for hooking on...')

    text_seg = ida_segment.get_segm_by_name('.text')
    # This code string appears to work for ELF32 and ELF64 AFAIK
    runtime_ms_end = ida_search.find_text(text_seg.startEA, 0, 0, "word ptr ds:1003h, 0", SEARCH_DOWN)
    runtime_ms = ida_funcs.get_func(runtime_ms_end)
    if idc.MakeNameEx(runtime_ms.startEA, "runtime_morecontext", SN_PUBLIC):
        debug('Successfully found runtime_morecontext')
    else:
        debug('Failed to rename function @ 0x%x to runtime_morestack' % runtime_ms.startEA)

    return runtime_ms
```

After finding the function, we can recursively traverse backwards through all the function calls, anything which is not inside an already defined function we can now define. This is because the structure always appears to be;

golang_undefined_function_example

```
.text:08089910                 ; Function start - however undefined currently accord
.text:08089910 loc_8089910:                             ; CODE XREF: .text:0808994B
.text:08089910                                          ; DATA XREF: sub_804B250+1A1
.text:08089910                 mov     ecx, large gs:0
.text:08089917                 mov     ecx, [ecx-4]
.text:0808991D                 cmp     esp, [ecx+8]
.text:08089920                 jbe     short loc_8089946
.text:08089922                 sub     esp, 4
.text:08089925                 mov     ebx, [edx+4]
.text:08089928                 mov     [esp], ebx
.text:0808992B                 cmp     dword ptr [esp], 0
.text:0808992F                 jz      short loc_808993E
.text:08089931
.text:08089931 loc_8089931:                             ; CODE XREF: .text:08089944
.text:08089931                 add     dword ptr [esp], 30h
.text:08089935                 call    sub_8052CB0
.text:0808993A                 add     esp, 4
.text:0808993D                 retn
.text:0808993E ; ---------------------------------------------------------------------------
.text:0808993E
.text:0808993E loc_808993E:                             ; CODE XREF: .text:0808992F
.text:0808993E                 mov     large ds:0, eax
.text:08089944                 jmp     short loc_8089931
.text:08089946 ; ---------------------------------------------------------------------------
.text:08089946
.text:08089946 loc_8089946:                             ; CODE XREF: .text:08089920
.text:08089946                 call    runtime_morestack ; "Bottom" of function, calls out to runtime_mor
.text:0808994B                 jmp     short loc_8089910 ; Jump back to the "top" of the function
```

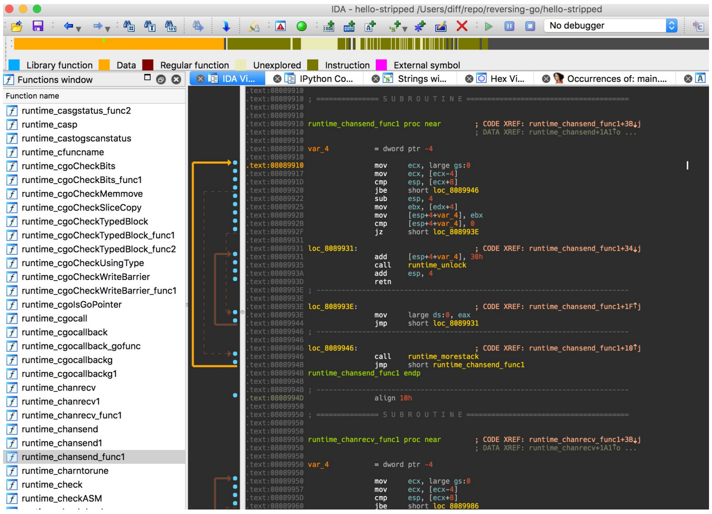
The above snippet is a random undefined function I pulled from the stripped example application we compiled already. Essentially by traversing backwards into every undefined function, we will land at something like line `0x0808994B` which is the `call runtime_morestack`. From here we will skip to the next instruction and ensure it is a jump above where we currently are, if this is true, we can likely assume this is the start of a function. In this example (and almost every test case I've run) this is true. Jumping to `0x08089910` is the start of the function, so now we have the two parameters required by `MakeFunction` function;

traverse_functions.py

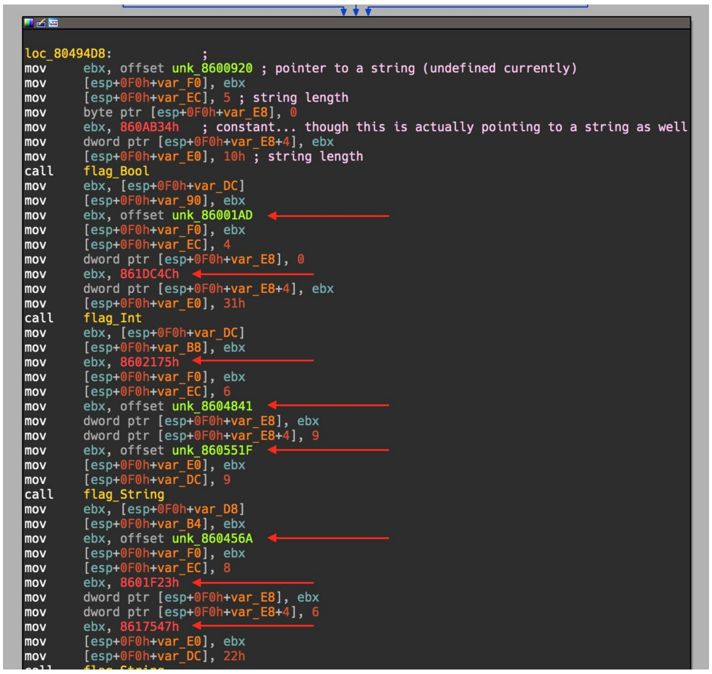
```
def is_simple_wrapper(addr):
    if GetMnem(addr) == 'xor' and GetOpnd(addr, 0) == 'edx' and  GetOpnd(addr, 1) == 'edx':
        addr = FindCode(addr, SEARCH_DOWN)
        if GetMnem(addr) == 'jmp' and GetOpnd(addr, 0) == 'runtime_morestack':
            return True

    return False

def create_runtime_ms():
    debug('Attempting to find runtime_morestack function for hooking on...')

    text_seg = ida_segment.get_segm_by_name('.text')
    # This code string appears to work for ELF32 and ELF64 AFAIK
    runtime_ms_end = ida_search.find_text(text_seg.startEA, 0, 0, "word ptr ds:1003h, 0", SEARCH_DOWN)
    runtime_ms = ida_funcs.get_func(runtime_ms_end)
    if idc.MakeNameEx(runtime_ms.startEA, "runtime_morestack", SN_PUBLIC):
        debug('Successfully found runtime_morestack')
    else:
        debug('Failed to rename function @ 0x%x to runtime_morestack' % runtime_ms.startEA)

    return runtime_ms

def traverse_xrefs(func):
    func_created = 0

    if func is None:
        return func_created

    # First
    func_xref = ida_xref.get_first_cref_to(func.startEA)
    # Attempt to go through crefs
    while func_xref != 0xffffffffffffffff:
        # See if there is a function already here
        if ida_funcs.get_func(func_xref) is None:
            # Ensure instruction bit looks like a jump
            func_end = FindCode(func_xref, SEARCH_DOWN)
            if GetMnem(func_end) == "jmp":
                # Ensure we're jumping back "up"
                func_start = GetOperandValue(func_end, 0)
                if func_start < func_xref:
                    if idc.MakeFunction(func_start, func_end):
                        func_created += 1
                    else:
                        # If this fails, we should add it to a list of failed functions
                        # Then create small "wrapper" functions and backtrack through the xrefs of this
                        error('Error trying to create a function @ 0x%x - 0x%x' %(func_start, func_end))
        else:
            xref_func = ida_funcs.get_func(func_xref)
            # Simple wrapper is often runtime_morestack_noctxt, sometimes it isn't though...
            if is_simple_wrapper(xref_func.startEA):
                debug('Stepping into a simple wrapper')
                func_created += traverse_xrefs(xref_func)
            if ida_funcs.get_func_name(xref_func.startEA) is not None and 'sub_' not in ida_funcs.get_fun
                debug('Function @0x%x already has a name of %s; skipping...' % (func_xref, ida_funcs.get_
            else:
                debug('Function @ 0x%x already has a name %s' % (xref_func.startEA, ida_funcs.get_func_na

        func_xref = ida_xref.get_next_cref_to(func.startEA, func_xref)

    return func_created

```

```
def find_func_by_name(name):
    text_seg = ida_segment.get_segm_by_name('.text')

    for addr in Functions(text_seg.startEA, text_seg.endEA):
        if name == ida_funcs.get_func_name(addr):
            return ida_funcs.get_func(addr)

    return None

def runtime_init():
    func_created = 0

    if find_func_by_name('runtime_morestack') is not None:
        func_created += traverse_xrefs(find_func_by_name('runtime_morestack'))
        func_created += traverse_xrefs(find_func_by_name('runtime_morestack_noctxt'))
    else:
        runtime_ms = create_runtime_ms()
        func_created = traverse_xrefs(runtime_ms)


    return func_created
```

That code bit is a bit lengthy, though hopefully the comments and concept is clear enough. It likely isn't necessary to explicitly traverse backwards recursively, however I wrote this prior to understanding that `runtime_morestack_noctxt` (the edgecase) is the only edgecase that I would encounter. This was being handled by the `is_simple_wrapper` function originally. Regardless, running this style of code ended up finding all the extra functions IDA Pro was missing. We can see below, that this creates a much cleaner and easier experience to reverse;



This can allow us to use something like Diaphora as well since we can specifically target functions with the same names, if we care too. I've personally found this is extremely useful for malware or other targets where you *really* don't care about any of the framework/runtime functions. You can quiet easily differentiate between custom code written for the binary, for example in the Linux malware "Rex" everything because with that name space! Now onto the last challenge that I wanted to solve while reversing the malware, string loading! I'm honestly not 100% sure how IDA detects most string loads, potentially through idioms of some sort? Or maybe because it can detect strings based on the `\00` character at the end of it? Regardless, Go seems to use a string table of some sort, without requiring null character. The appear to be in alpha-numeric order, group by string length size as well. This means we see them all there, but often don't come across them correctly asserted as strings, or we see them asserted as extremely large blobs of strings. The hello world example isn't good at illistrating this, so I'll pull open the `main.main` function of the Rex malware to show this;



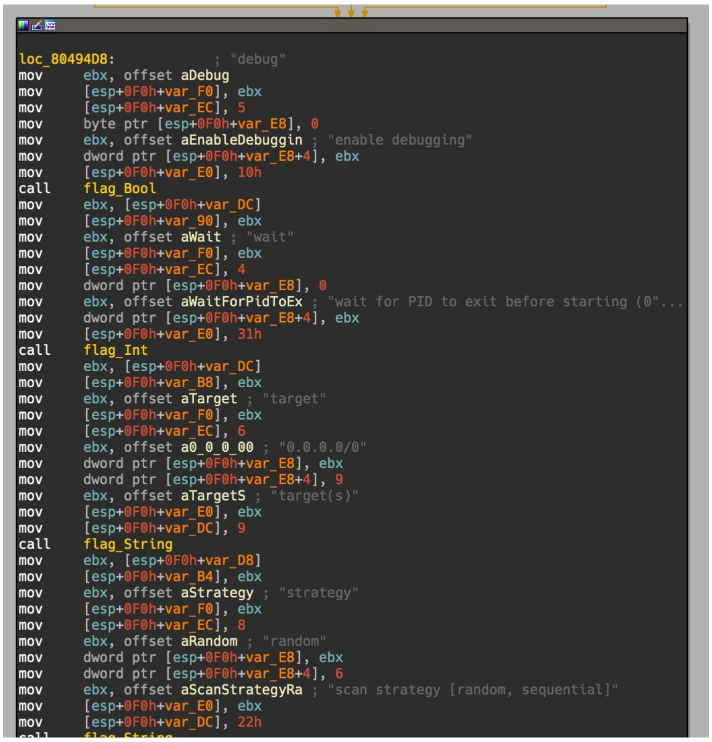
I didn't want to add comments to everything, so I only commented the first few lines then pointed arrows to where there should be pointers to a proper string. We can see a few different use cases and sometimes the destination registers seem to change. However there is definitely a pattern which forms that we can look for. Moving of a pointer into a register, that register is then used to push into a (d)word pointer, followed by a load of a lenght of the string. Cobbling together some python to hunt for the pattern we end with something like the pseudo code below;

string_hunting.py

```
# Currently it's normally ebx, but could in theory be anything - seen ebp
VALID_REGS = ['ebx', 'ebp']

# Currently it's normally esp, but could in theory be anything - seen eax
VALID_DEST = ['esp', 'eax', 'ecx', 'edx']

def is_string_load(addr):
    patterns = []
    # Check for first part
```

```
    if GetMnem(addr) == 'mov':
        # Could be unk_ or asc_, ignored ones could be loc_ or inside []
        if GetOpnd(addr, 0) in VALID_REGS and not ('[' in GetOpnd(addr, 1) or 'loc_' in GetOpnd(addr, 1))
            from_reg = GetOpnd(addr, 0)
            # Check for second part
            addr_2 = FindCode(addr, SEARCH_DOWN)
            try:
                dest_reg = GetOpnd(addr_2, 0)[GetOpnd(addr_2, 0).index('[') + 1:GetOpnd(addr_2, 0).index(
            except ValueError:
                return False
            if GetMnem(addr_2) == 'mov' and dest_reg in VALID_DEST and ('[%s' % dest_reg) in GetOpnd(addr
                # Check for last part, could be improved
                addr_3 = FindCode(addr_2, SEARCH_DOWN)
                if GetMnem(addr_3) == 'mov' and (('[%s+' % dest_reg) in GetOpnd(addr_3, 0) or GetOpnd(add
                    try:
                        dumb_int_test = GetOperandValue(addr_3, 1)
                        if dumb_int_test > 0 and dumb_int_test < sys.maxsize:
                            return True
                    except ValueError:
                        return False

def create_string(addr, string_len):
    debug('Found string load @ 0x%x with length of %d' % (addr, string_len))
    # This may be overly aggressive if we found the wrong area...
    if GetStringType(addr) is not None and GetString(addr) is not None and len(GetString(addr)) != string
        debug('It appears that there is already a string present @ 0x%x' % addr)
        MakeUnknown(addr, string_len, DOUNK_SIMPLE)

    if GetString(addr) is None and MakeStr(addr, addr + string_len):
        return True
    else:
        # If something is already partially analyzed (incorrectly) we need to MakeUnknown it
        MakeUnknown(addr, string_len, DOUNK_SIMPLE)
        if MakeStr(addr, addr + string_len):
            return True
        debug('Unable to make a string @ 0x%x with length of %d' % (addr, string_len))

    return False
```

The above code could likely be optimized, however it was working for me on the samples I needed. All that would be left is to create another function which hunts through all the defined code segments to look for string loads. Then we can use the pointer to the string and the string length to define a new string using the `MakeStr`. In the code I ended up using, you need to ensure that IDA Pro hasn't mistakenly already create the string, as it sometimes tries to, incorrectly. This seems to happen sometimes when a string in the table contains a null character. However, after using code above, this is what we are left with;



This is a much better piece of code to work with. After we throw together all these functions, we now have the golang_loader_assist.py module for IDA Pro. A word of warning though, I have only had time to test this on a few versions of IDA Pro for OSX, the majority of testing on 6.95. There is also very likely optimizations which should be made or at a bare minimum some reworking of the code. With all that said, I wanted to open source this so others could use this and hopefully contribute back. Also be aware that this script can be painfully slow depending on how large the `idb` file is, working on a OSX El Capitan (10.11.6) using a 2.2 GHz Intel Core i7 on IDA Pro 6.95 - the string discovery aspect itself can take a while. I've often found that running the different methods seperately can prevent IDA from locking up. Hopefully this blog and the code proves useful to someone though, enjoy!