

RedNaga Security

spicy security research

2016-09-21

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](#).

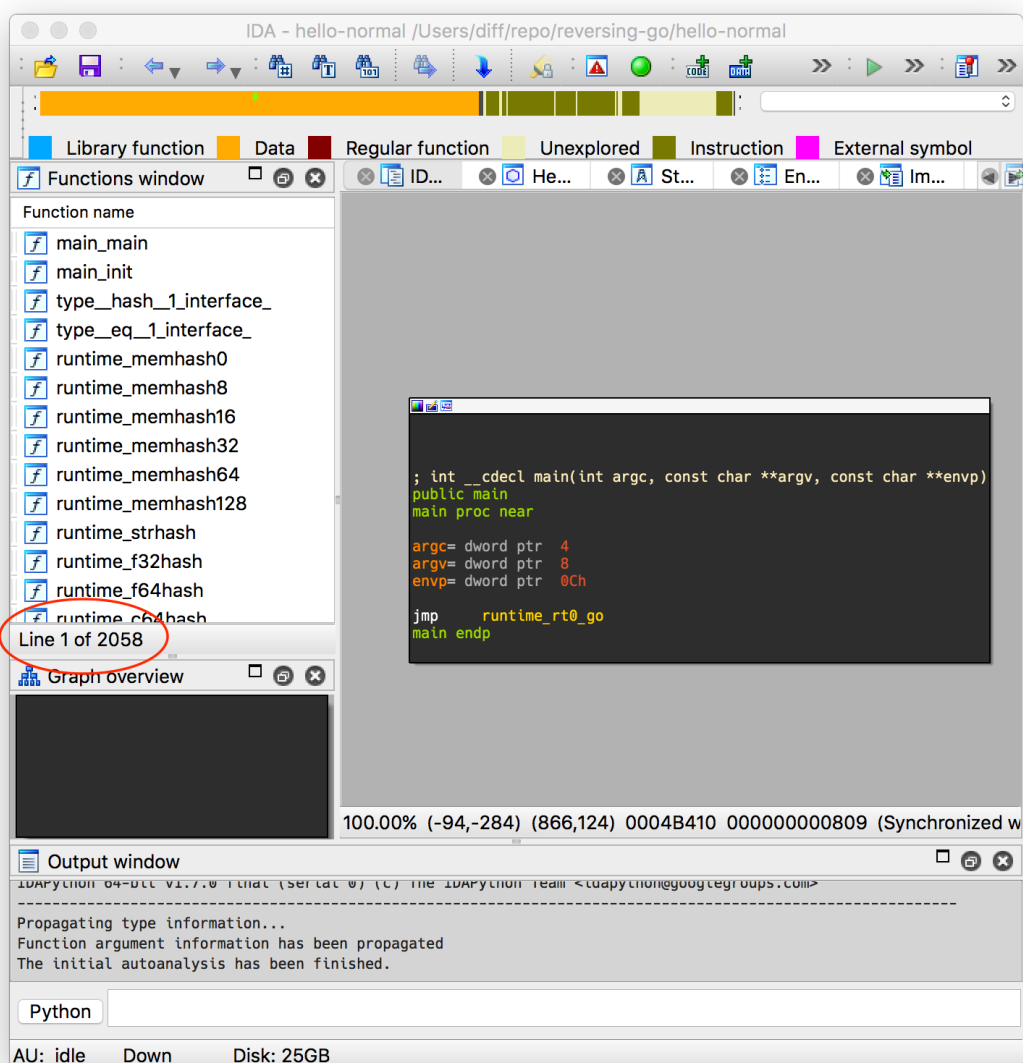
To illustrate some of my examples I'm going to use an extremely simple 'Hello, World!' example and also reference the Rex malware. The code and a Make file are extremely simple;

```
Hello.go
1 package main
2 import "fmt"
3 func main() {
4     fmt.Println("Hello, World!")
5 }
```

Makefile

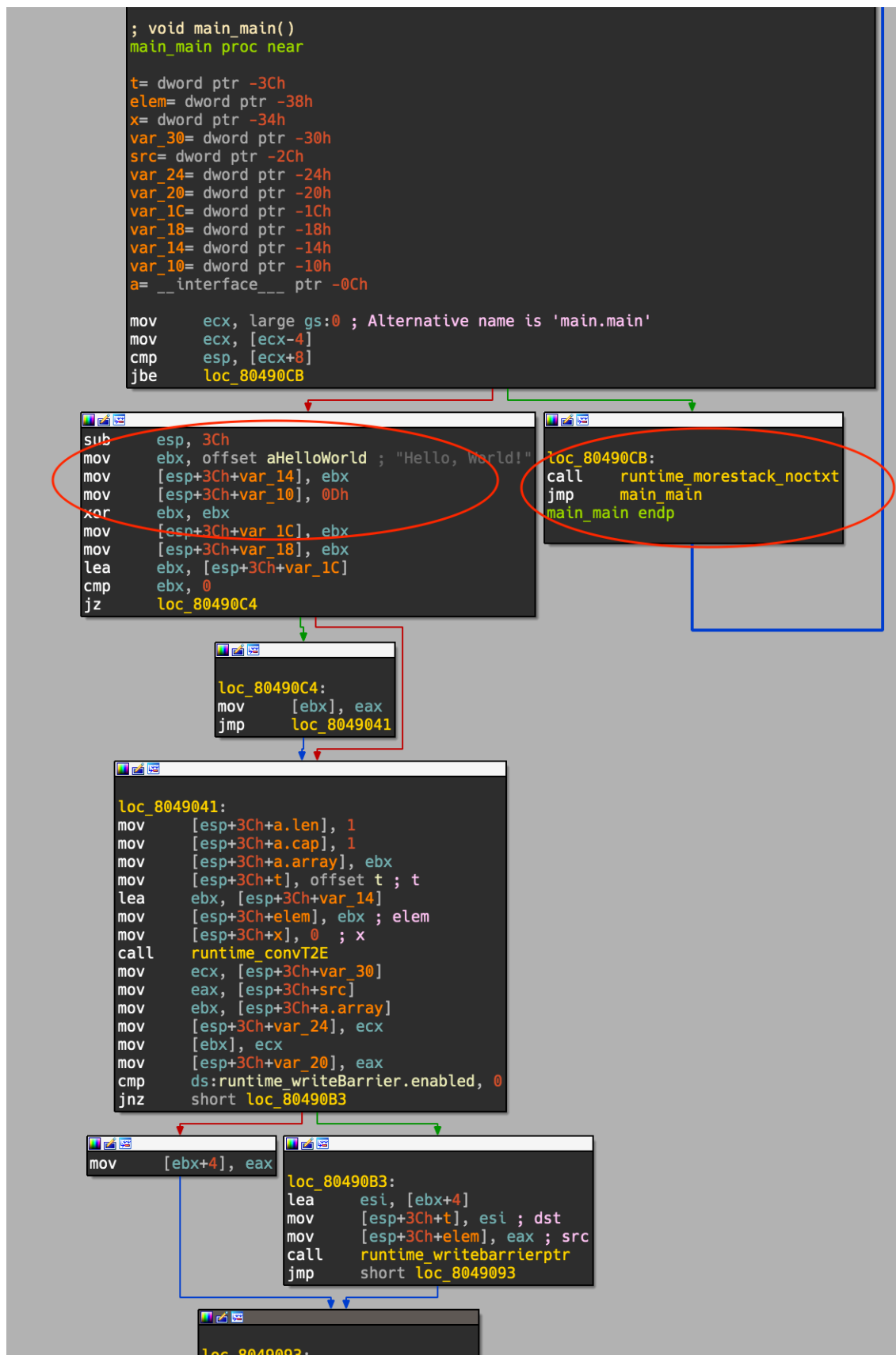
```
1 all:
2     GOOS=linux GOARCH=386 go build -o hello-stripped -ldflags '
3     GOOS=linux GOARCH=386 go build -o hello-normal hello.go
```

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`;



```

mov     ebx, [esp+3Ch+a.array]
mov     [esp+3Ch+t], ebx ; a
mov     ebx, [esp+3Ch+a.len]
mov     [esp+3Ch+elem], ebx
mov     ebx, [esp+3Ch+a.cap]
mov     [esp+3Ch+x], ebx
call    fmt_Printfln
add     esp, 3Ch
retn

```

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

```

1  mov     ebx, offset aHelloWorld ; "Hello, World!"
2  mov     [esp+3Ch+var_14], ebx ; Shove string into location
3  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;

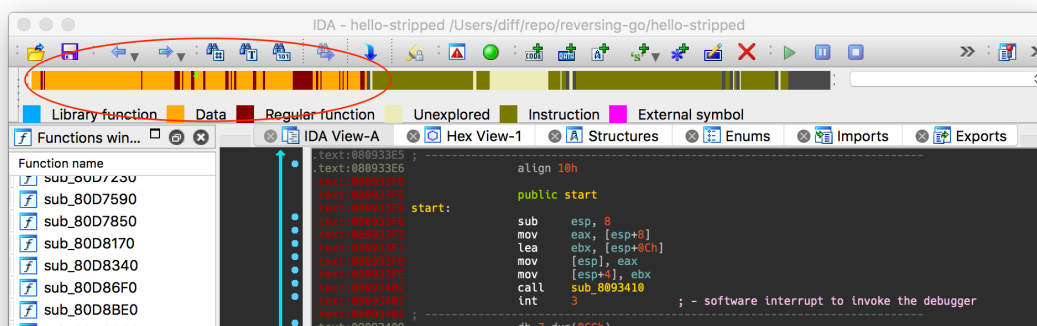
morestack_context

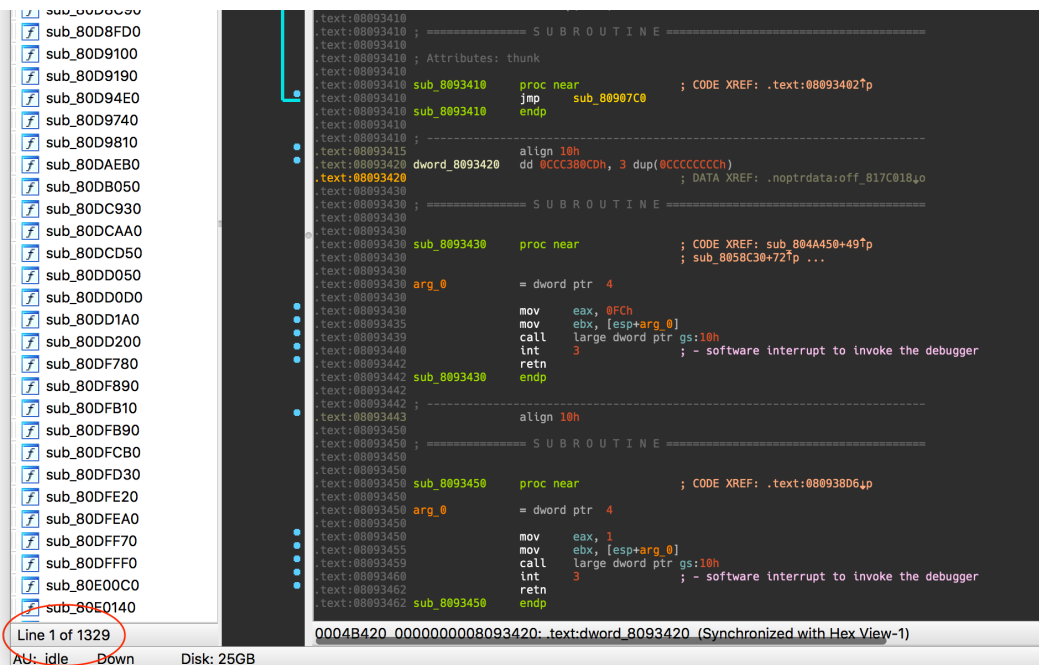
```

1  loc_80490CB:
2  call    runtime_morestack_noctxt
3  jmp     main_main

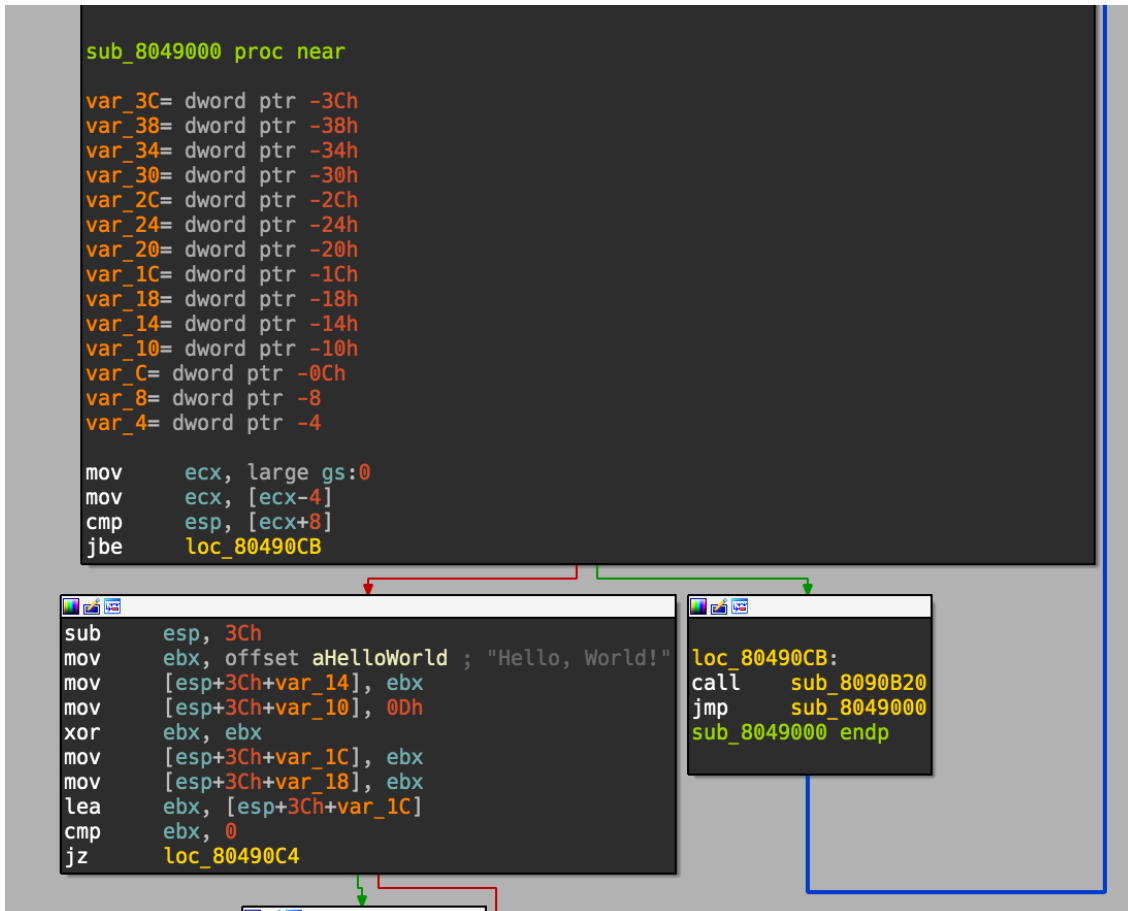
```

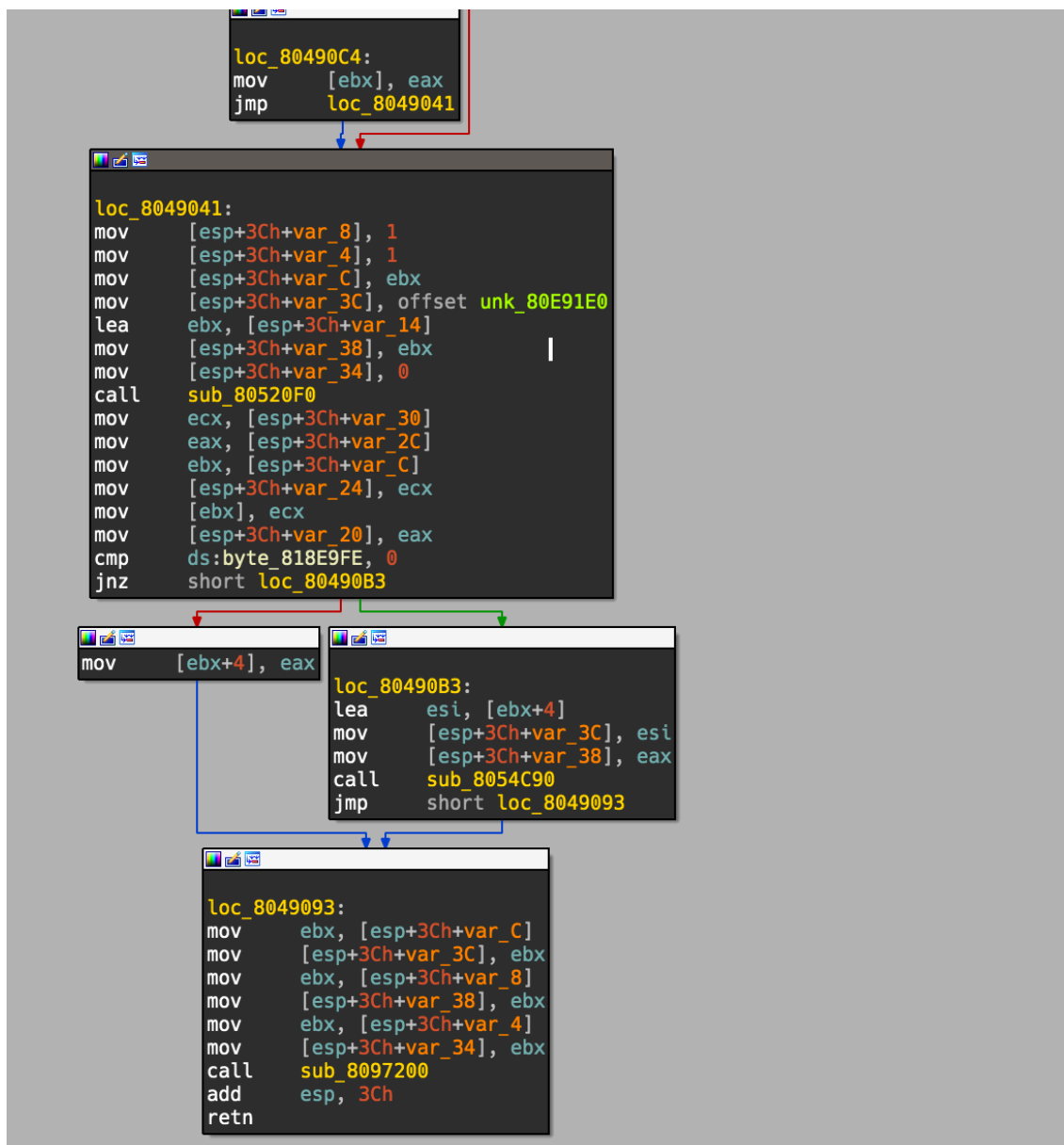
This style block of code appears to always be at the end of functions and it also seems to always loop back up to the top of the same function. This is verified by looking at the cross-references to this function. Ok, now that we know IDA Pro can handle unstripped binaries, lets load the same code but 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 represented at `main_main` in the previous screen shots due to how a `.` is interpreted by IDA);

`.gopclntab`

```

1  .gopclntab:0813E174          db  6Dh ; m
2  .gopclntab:0813E175          db  61h ; a
3  .gopclntab:0813E176          db  69h ; i
4  .gopclntab:0813E177          db  6Eh ; n
5  .gopclntab:0813E178          db  2Eh ; .
6  .gopclntab:0813E179          db  6Dh ; m
7  .gopclntab:0813E17A          db  61h ; a

```

```
8  .gopclntab:0813E17B          db  69h ; i
9  .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 functionality 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.

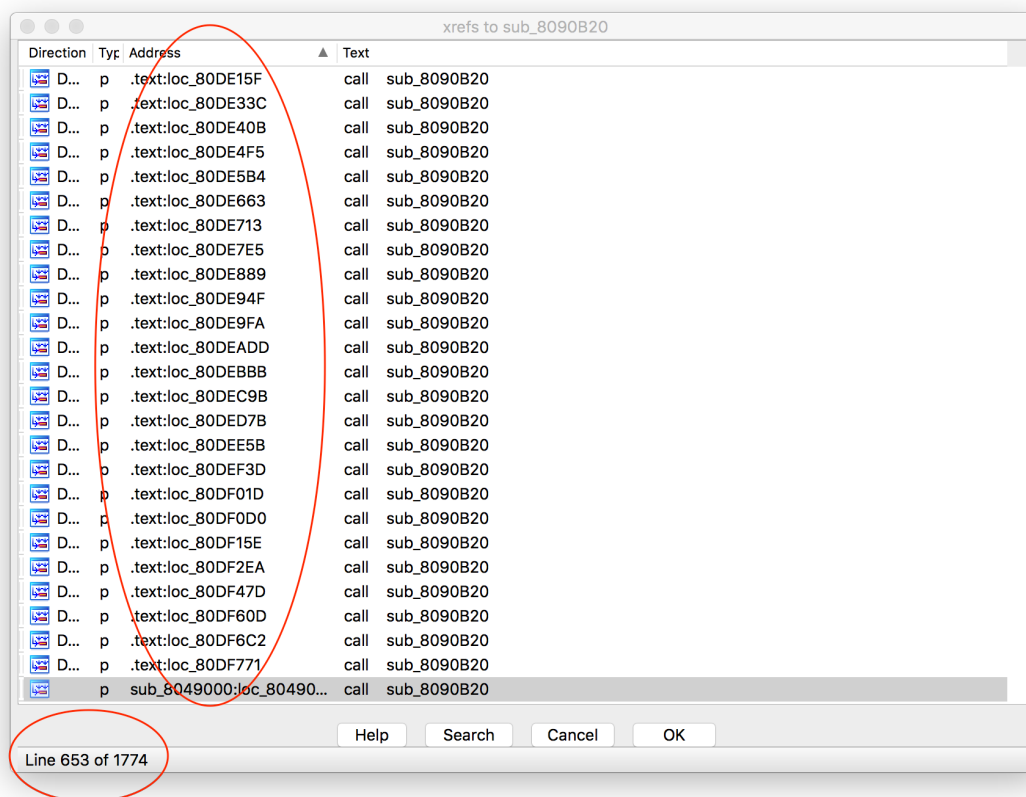
renamer.py

```
1  def create_pointer(addr, force_size=None):
2      if force_size is not 4 and (idaapi.get_inf_structure().is_64bit):
3          MakeQword(addr)
4          return Qword(addr), 8
5      else:
6          MakeDword(addr)
7          return Dword(addr), 4
8
9  STRIP_CHARS = [ '(', ')', '[', ']', '{', '}', ' ', '"' ]
10 REPLACE_CHARS = [ '.', '*', '-', ',', ';', ':', '/', '\\xb7' ]
11 def clean_function_name(str):
12     # Kill generic 'bad' characters
13     str = filter(lambda x: x in string.printable, str)
14
15     for c in STRIP_CHARS:
16         str = str.replace(c, '')
17
18     for c in REPLACE_CHARS:
19         str = str.replace(c, '_')
20
21     return str
22
```

```
23 def renamer_init():
24     renamed = 0
25
26     gopclntab = ida_segment.get_segm_by_name('.gopclntab')
27     if gopclntab is not None:
28         # Skip unimportant header and goto section size
29         addr = gopclntab.startEA + 8
30         size, addr_size = create_pointer(addr)
31         addr += addr_size
32
33         # Unsure if this end is correct
34         early_end = addr + (size * addr_size * 2)
35         while addr < early_end:
36             func_offset, addr_size = create_pointer(addr)
37             name_offset, addr_size = create_pointer(addr + addr_si
38             addr += addr_size * 2
39
40             func_name_addr = Dword(name_offset + gopclntab.startEA
41             func_name = GetString(func_name_addr)
42             MakeStr(func_name_addr, func_name_addr + len(func_name
43             appended = clean_func_name = clean_function_name(func_
44             debug('Going to remap function at 0x%x with %s - clean
45
46             if ida_funcs.get_func_name(func_offset) is not None:
47                 if MakeName(func_offset, clean_func_name):
48                     renamed += 1
49             else:
50                 error('clean_func_name error %s' % clean_func_
51
52     return renamed
53
54 def main():
55     renamed = renamer_init()
56     info('Found and successfully renamed %d functions!' % renamed)
```

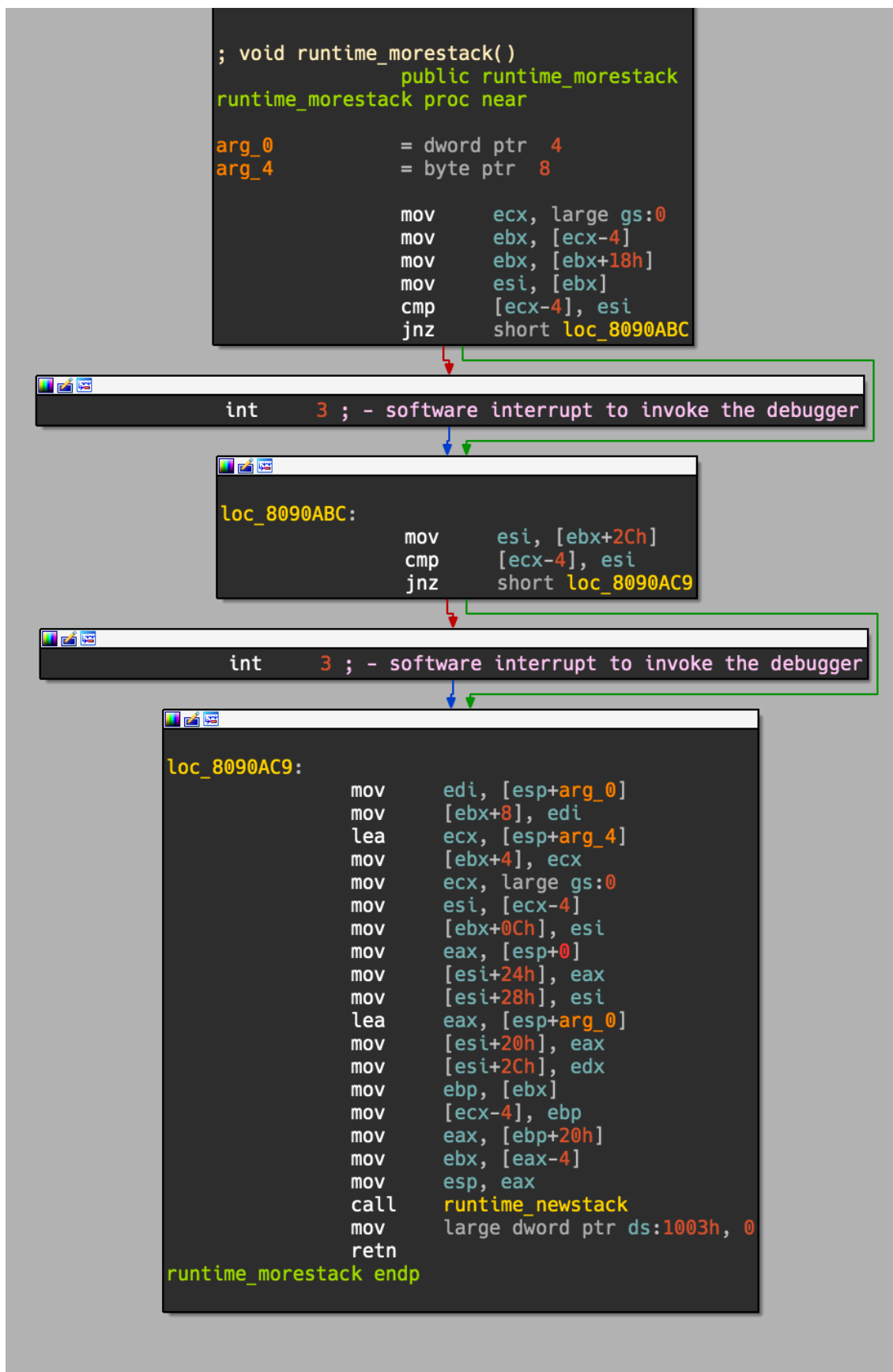
The above code won't actually run yet (don't worry full code available in [this repo](#)) but it is hopefully simple enough to read through and understand the process. However, this still doesn't solve the problem that IDA Pro doesn't know *all* the functions. So this is going to create pointers which 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 separating 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 multiple binaries we can see the `runtime_morestack_noctxt` 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 references to every 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 ourselves. So, enough talk, lets write some code to find this function;

```
find_runtime_morestack.py
```

```
1 def create_runtime_ms():
2     debug('Attempting to find runtime_morestack function for hooki
3
4     text_seg = ida_segment.get_segm_by_name('.text')
5     # This code string appears to work for ELF32 and ELF64 AFAIK
6     runtime_ms_end = ida_search.find_text(text_seg.startEA, 0, 0,
7     runtime_ms = ida_funcs.get_func(runtime_ms_end)
8     if idc.MakeNameEx(runtime_ms.startEA, "runtime_morecontext", S
9         debug('Successfully found runtime_morecontext')
10    else:
11        debug('Failed to rename function @ 0x%x to runtime_moresta
12
13    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
```

```
1 .text:08089910 ; Function
2 .text:08089910 loc_8089910: ; CODE XREF
3 .text:08089910 ; DATA XREF
4 .text:08089910 mov ecx, large gs:0
5 .text:08089917 mov ecx, [ecx-4]
6 .text:0808991D cmp esp, [ecx+8]
7 .text:08089920 jbe short loc_8089946
8 .text:08089922 sub esp, 4
9 .text:08089925 mov ebx, [edx+4]
10 .text:08089928 mov [esp], ebx
11 .text:0808992B cmp dword ptr [esp], 0
12 .text:0808992F jz short loc_808993E
13 .text:08089931
```

```

14  .text:08089931 loc_8089931:                                ; CODE XREF
15  .text:08089931          add     dword ptr [esp], 30h
16  .text:08089935          call   sub_8052CB0
17  .text:0808993A          add     esp, 4
18  .text:0808993D          retn
19  .text:0808993E ; -----
20  .text:0808993E
21  .text:0808993E loc_808993E:                                ; CODE XREF
22  .text:0808993E          mov     large ds:0, eax
23  .text:08089944          jmp     short loc_8089931
24  .text:08089946 ; -----
25  .text:08089946
26  .text:08089946 loc_8089946:                                ; CODE XREF
27  .text:08089946          call   runtime_morestack ; "Bottom
28  .text:0808994B          jmp     short loc_8089910 ; Jump ba

```

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

```

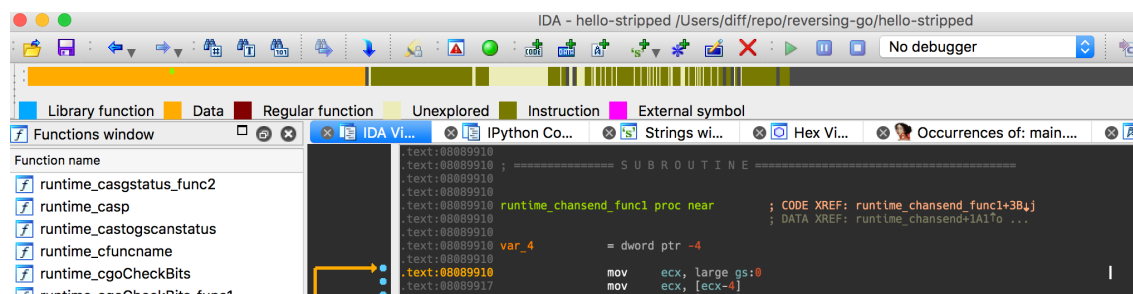
1  def is_simple_wrapper(addr):
2      if GetMnem(addr) == 'xor' and GetOpnd(addr, 0) == 'edx' and G
3          addr = FindCode(addr, SEARCH_DOWN)
4          if GetMnem(addr) == 'jmp' and GetOpnd(addr, 0) == 'runtime
5              return True
6
7      return False
8
9  def create_runtime_ms():
10     debug('Attempting to find runtime_morestack function for hooki
11
12     text_seg = ida_segment.get_segm_by_name('.text')
13     # This code string appears to work for ELF32 and ELF64 AFAIK
14     runtime_ms_end = ida_search.find_text(text_seg.startEA, 0, 0,

```

```
15     runtime_ms = ida_funcs.get_func(runtime_ms_end)
16     if idc.MakeNameEx(runtime_ms.startEA, "runtime_morestack", SN_
17         debug('Successfully found runtime_morestack')
18     else:
19         debug('Failed to rename function @ 0x%x to runtime_moresta
20
21     return runtime_ms
22
23 def traverse_xrefs(func):
24     func_created = 0
25
26     if func is None:
27         return func_created
28
29     # First
30     func_xref = ida_xref.get_first_cref_to(func.startEA)
31     # Attempt to go through crefs
32     while func_xref != 0xffffffffffffffff:
33         # See if there is a function already here
34         if ida_funcs.get_func(func_xref) is None:
35             # Ensure instruction bit looks like a jump
36             func_end = FindCode(func_xref, SEARCH_DOWN)
37             if GetMnem(func_end) == "jmp":
38                 # Ensure we're jumping back "up"
39                 func_start = GetOperandValue(func_end, 0)
40                 if func_start < func_xref:
41                     if idc.MakeFunction(func_start, func_end):
42                         func_created += 1
43                 else:
44                     # If this fails, we should add it to a lis
45                     # Then create small "wrapper" functions an
46                     error('Error trying to create a function @
47     else:
48         xref_func = ida_funcs.get_func(func_xref)
49         # Simple wrapper is often runtime_morestack_noctxt, so
50         if is_simple_wrapper(xref_func.startEA):
51             debug('Stepping into a simple wrapper')
52             func_created += traverse_xrefs(xref_func)
53         if ida_funcs.get_func_name(xref_func.startEA) is not N
54             debug('Function @0x%x already has a name of %s; sk
55     else:
56         debug('Function @ 0x%x already has a name %s' % (x
```

```
57
58     func_xref = ida_xref.get_next_cref_to(func.startEA, func_x
59
60     return func_created
61
62 def find_func_by_name(name):
63     text_seg = ida_segment.get_segm_by_name('.text')
64
65     for addr in Functions(text_seg.startEA, text_seg.endEA):
66         if name == ida_funcs.get_func_name(addr):
67             return ida_funcs.get_func(addr)
68
69     return None
70
71 def runtime_init():
72     func_created = 0
73
74     if find_func_by_name('runtime_morestack') is not None:
75         func_created += traverse_xrefs(find_func_by_name('runtime_
76         func_created += traverse_xrefs(find_func_by_name('runtime_
77     else:
78         runtime_ms = create_runtime_ms()
79         func_created = traverse_xrefs(runtime_ms)
80
81
82     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;



```

runtime_cgoCheckMemmove
runtime_cgoCheckSliceCopy
runtime_cgoCheckTypedBlock
runtime_cgoCheckTypedBlock_func1
runtime_cgoCheckTypedBlock_func2
runtime_cgoCheckUsingType
runtime_cgoCheckWriteBarrier
runtime_cgoCheckWriteBarrier_func1
runtime_cgolsGoPointer
runtime_cgocall
runtime_cgocallback
runtime_cgocallback_gofunc
runtime_cgocallbackg
runtime_cgocallbackg1
runtime_chanrecv
runtime_chanrecv1
runtime_chanrecv_func1
runtime_chansend
runtime_chansend1
runtime_chansend_func1
runtime_charntorune
runtime_check
runtime_checkASM

```

```

.text:00089920 jbe short loc_8089946
.text:00089922 sub esp, 4
.text:00089925 mov ebx, [edx+4]
.text:00089928 mov [esp+4+var_4], ebx
.text:0008992B cmp [esp+4+var_4], 8
.text:0008992F jz short loc_808993E
.text:00089931 loc_8089931: ; CODE XREF: runtime_chansend_func1+34j
.text:00089931 add [esp+4+var_4], 30h
.text:00089935 call runtime_unlock
.text:0008993A add esp, 4
.text:0008993D retch
;-----
.text:0008993E loc_808993E: ; CODE XREF: runtime_chansend_func1+1F7j
.text:0008993E mov large ds:0, eax
.text:00089944 jmp short loc_8089931
;-----
.text:00089946 loc_8089946: ; CODE XREF: runtime_chansend_func1+107j
.text:00089946 call runtime_morestack
.text:0008994B jmp short runtime_chansend_func1
runtime_chansend_func1 endp
;-----
.text:0008994B align 10h
.text:00089950 ; ===== SUBROUTINE =====
.text:00089950 runtime_chanrecv_func1 proc near ; CODE XREF: runtime_chanrecv_func1+3B4j
.text:00089950 ; DATA XREF: runtime_chanrecv+1A17o ...
.text:00089950 var_4 = dword ptr -4
.text:00089950 mov ecx, large ds:0
.text:00089957 mov ecx, [ecx-4]
.text:0008995D cmp esp, [ecx+8]
.text:00089960 jbe short loc_8089986

```

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 quite 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. They 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 illustrating this, so I'll pull open the `main.main` function of the Rex malware to show this;

```

loc_80494D8:
mov ebx, offset unk_8600920 ; pointer to a string (undefined currently)
mov [esp+0F0h+var_F0], ebx
mov [esp+0F0h+var_EC], 5 ; string length
mov byte ptr [esp+0F0h+var_E8], 0
mov ebx, 860AB34h ; constant... though this is actually pointing to a string as well
mov dword ptr [esp+0F0h+var_E8+4], ebx
mov [esp+0F0h+var_E0], 10h ; string length
call flag_Boot
mov ebx, [esp+0F0h+var_DC]
mov [esp+0F0h+var_90], ebx
mov ebx, offset unk_86001AD
mov [esp+0F0h+var_F0], ebx
mov [esp+0F0h+var_EC], 4
mov dword ptr [esp+0F0h+var_E8], 0
mov ebx, 861DC4Ch
mov dword ptr [esp+0F0h+var_E8+4], ebx
mov [esp+0F0h+var_E0], 31h
call flag_Int
mov ebx, [esp+0F0h+var_DC]
mov [esp+0F0h+var_B8], ebx
mov ebx, 8602175h
mov [esp+0F0h+var_F0], ebx
mov [esp+0F0h+var_EC], 6
mov ebx, offset unk_8604841

```

```

mov     ebx, offset unk_8604841
mov     dword ptr [esp+0F0h+var_E8], ebx
mov     dword ptr [esp+0F0h+var_E8+4], 9
mov     ebx, offset unk_860551F
mov     [esp+0F0h+var_E0], ebx
mov     [esp+0F0h+var_DC], 9
call    flag_String
mov     ebx, [esp+0F0h+var_D8]
mov     [esp+0F0h+var_B4], ebx
mov     ebx, offset unk_860456A
mov     [esp+0F0h+var_F0], ebx
mov     [esp+0F0h+var_EC], 8
mov     ebx, 8601F23h
mov     dword ptr [esp+0F0h+var_E8], ebx
mov     dword ptr [esp+0F0h+var_E8+4], 6
mov     ebx, 8617547h
mov     [esp+0F0h+var_E0], ebx
mov     [esp+0F0h+var_DC], 22h

```

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 length of the string. Cobbling together some python to hunt for the pattern we end with something like the pseudo code below;

string_hunting.py

```

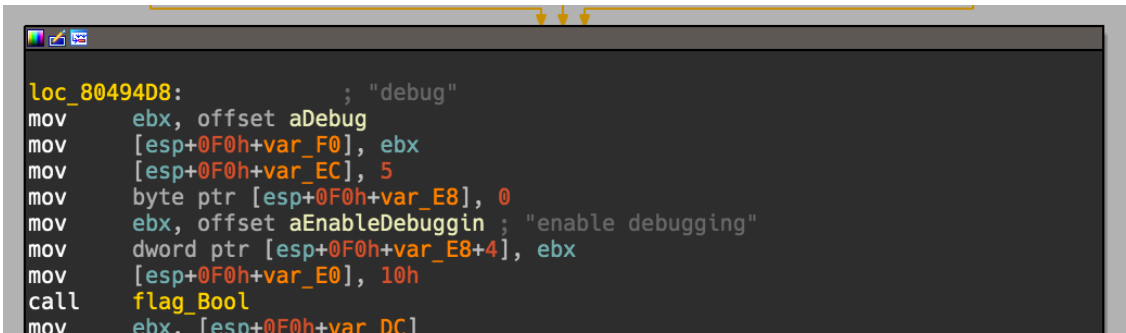
1  # Currently it's normally ebx, but could in theory be anything - s
2  VALID_REGS = ['ebx', 'ebp']
3
4  # Currently it's normally esp, but could in theory be anything - s
5  VALID_DEST = ['esp', 'eax', 'ecx', 'edx']
6
7  def is_string_load(addr):
8      patterns = []
9      # Check for first part
10     if GetMnem(addr) == 'mov':
11         # Could be unk_ or asc_, ignored ones could be loc_ or ins
12         if GetOpnd(addr, 0) in VALID_REGS and not ('[' in GetOpnd(
13             from_reg = GetOpnd(addr, 0)
14         # Check for second part
15         addr_2 = FindCode(addr, SEARCH_DOWN)
16         try:
17             dest_reg = GetOpnd(addr_2, 0)[GetOpnd(addr_2, 0).i
18         except ValueError:
19             return False
20         if GetMnem(addr_2) == 'mov' and dest_reg in VALID_DEST
21         # Check for last part, could be improved
22         addr_3 = FindCode(addr_2, SEARCH_DOWN)

```



```
23         if GetMnem(addr_3) == 'mov' and (('[%s+' % dest_re
24             try:
25                 dumb_int_test = GetOperandValue(addr_3, 1)
26                 if dumb_int_test > 0 and dumb_int_test < s
27                     return True
28             except ValueError:
29                 return False
30
31 def create_string(addr, string_len):
32     debug('Found string load @ 0x%x with length of %d' % (addr, st
33     # This may be overly aggressive if we found the wrong area...
34     if GetStringType(addr) is not None and GetString(addr) is not
35         debug('It appears that there is already a string present @
36         MakeUnknown(addr, string_len, DOUNK_SIMPLE)
37
38     if GetString(addr) is None and MakeStr(addr, addr + string_len
39         return True
40     else:
41         # If something is already partially analyzed (incorrectly)
42         MakeUnknown(addr, string_len, DOUNK_SIMPLE)
43         if MakeStr(addr, addr + string_len):
44             return True
45         debug('Unable to make a string @ 0x%x with length of %d' %
46
47     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;



```
loc_80494D8:                ; "debug"
mov     ebx, offset aDebug
mov     [esp+0F0h+var_F0], ebx
mov     [esp+0F0h+var_EC], 5
mov     byte ptr [esp+0F0h+var_E8], 0
mov     ebx, offset aEnableDebuggin ; "enable debugging"
mov     dword ptr [esp+0F0h+var_E8+4], ebx
mov     [esp+0F0h+var_E0], 10h
call    flag_Bool
mov     ebx, [esp+0F0h+var_DC]
```

```
mov [esp+0F0h+var_90], ebx
mov ebx, offset aWait ; "wait"
mov [esp+0F0h+var_F0], ebx
mov [esp+0F0h+var_EC], 4
mov dword ptr [esp+0F0h+var_E8], 0
mov ebx, offset aWaitForPidToEx ; "wait for PID to exit before starting (0"...
mov dword ptr [esp+0F0h+var_E8+4], ebx
mov [esp+0F0h+var_E0], 31h
call flag_Int
mov ebx, [esp+0F0h+var_DC]
mov [esp+0F0h+var_B8], ebx
mov ebx, offset aTarget ; "target"
mov [esp+0F0h+var_F0], ebx
mov [esp+0F0h+var_EC], 6
mov ebx, offset a0_0_0_00 ; "0.0.0.0/0"
mov dword ptr [esp+0F0h+var_E8], ebx
mov dword ptr [esp+0F0h+var_E8+4], 9
mov ebx, offset aTargetS ; "target(s)"
mov [esp+0F0h+var_E0], ebx
mov [esp+0F0h+var_DC], 9
call flag_String
mov ebx, [esp+0F0h+var_DB]
mov [esp+0F0h+var_B4], ebx
mov ebx, offset aStrategy ; "strategy"
mov [esp+0F0h+var_F0], ebx
mov [esp+0F0h+var_EC], 8
mov ebx, offset aRandom ; "random"
mov dword ptr [esp+0F0h+var_E8], ebx
mov dword ptr [esp+0F0h+var_E8+4], 6
mov ebx, offset aScanStrategyRa ; "scan strategy [random, sequential]"
mov [esp+0F0h+var_E0], ebx
mov [esp+0F0h+var_DC], 22h
call flag_String
```

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 separately can prevent IDA from locking up. Hopefully this blog and the code proves useful to someone though, enjoy!

#golang #ida pro #research

Comments Share

NEWER

HackingTeam back for your Androids, now extra insecure!

OLDER

Detecting Pirated and Malicious Android Apps with APKiD

5 Comments **RedNaga****1 Login** ▾ **Recommend** 2 **Tweet** **Share****Sort by Best** ▾

Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS (?)

Name **sapristi** • 10 months ago

i loved your article, although i don't really know any assembler. I'm currently looking for a safe way to distribute the source code of a server side app (which has a REST Api), which i'm currently developing in NodeJS but willing to move it to something more secure/stable like python or golang, all of them being scripting languages as far as I understand. Could you recommend me a good solution for this? I'd really like to program it in C++ but I honestly only know "a lot of javascript", "a fair amount of PHP" and a "tiny bit of python", though I'm open to learn something bigger.

^ | ▾ • Reply • Share >

**Caleb Fenton** Mod → sapristi • 10 months ago

Have you looked at using GitHub to distribute your source code? It's standard. There's also BitBucket, but GitHub is more popular. If you want to build a REST API server that will work fairly easily, I'd go with something like Flask (python), Django (also python), NodeJS, or maybe even Rails. Writing it in golang or C++ would certainly be educational but would be a lot more work and may be prone to errors.

^ | ▾ • Reply • Share >

**sapristi** → Caleb Fenton • 10 months ago

thanks, I meant to distribute the production package to install it on my clients' machines since some of them don't trust using other's cloud servers and i don't trust giving them the source code. I'm thinking of something like a Docker image but somehow making it non-understandable or non-readable by someone else, just launchable

^ | ▾ • Reply • Share >

RECENTS

[Hacking with dex-oracle for Android Malware Deobfuscation](#)

[Remote Kext Debugging \(No, really - it worked!\)](#)

[HackingTeam back for your Androids, now extra insecure!](#)

[Reversing GO binaries like a pro](#)

[Detecting Pirated and Malicious Android Apps with APKiD](#)

PAGES

[Team RedNaga](#)

TAGS

[android](#) (2)

[apkid](#) (2)

[deobfuscation](#) (1)

[dex-oracle](#) (1)

[gdb](#) (1)

[golang](#) (1)

[hackingteam](#) (1)

[ida pro](#) (2)

[kernel](#) (1)

[kext](#) (1)

[lldb](#) (1)

[macos](#) (1)

[malware](#) (1)

[research](#) (5)

[reverse engineering](#) (2)

[surveillance](#) (1)

[vmware](#) (1)

TAG CLOUD

[android](#) [apkid](#) [deobfuscation](#) [dex-oracle](#) [gdb](#) [golang](#) [hackingteam](#) [ida pro](#) [kernel](#) [kext](#) [lldb](#) [macos](#)
[malware](#) [research](#) [reverse engineering](#) [surveillance](#) [vmware](#)

ARCHIVES

[October 2017](#) (1)
[April 2017](#) (1)
[November 2016](#) (1)
[September 2016](#) (1)
[July 2016](#) (2)

TWITTER FEED

Tweets by @RedNagaSec

RedNaga Retweeted



Tim Strazzere
@timstrazz

Getting ready for @hoodsec later tonight, don't forget 8pm at The Layover #hoodsec -- maybe more Aussies will show up again @xntrik ? :D

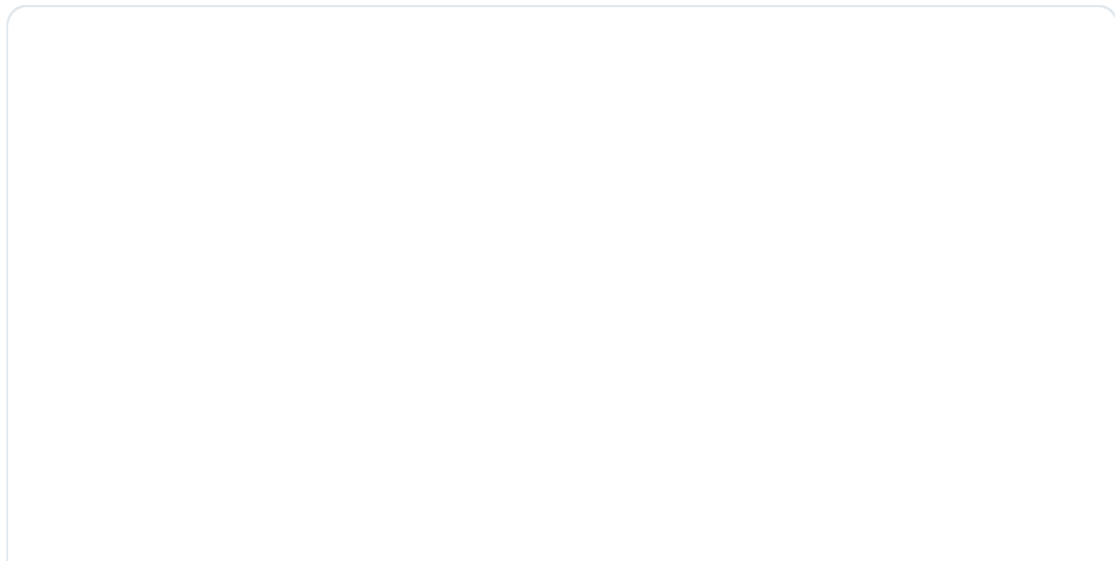
Sep 28, 2018

RedNaga Retweeted



HoodSec
@hoodsec

This week we will be having #hoodsec at The Layover - just around the corner from RadioBar oaklandlayover.com



THE LAYOVER

Home page of THE LAYOVER, from OAKLAND
oaklandlayover.com

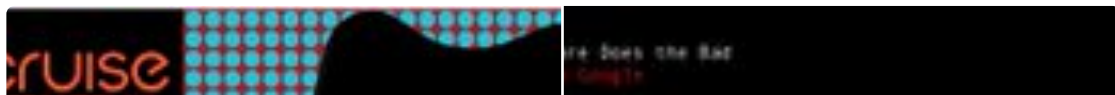
Jul 24, 2018

RedNaga Retweeted



Amanda Rousseau
@malwareunicorn

DeadDropSF is happening this evening! Thank you to @timstrazz for hosting the venue and for our speakers @maddiestone and @cooperq! We've maxed out the attendee list again. meetup.com/Dead-Drop-SF/



© 2017 RedNaga
Powered by [Hexo](#)