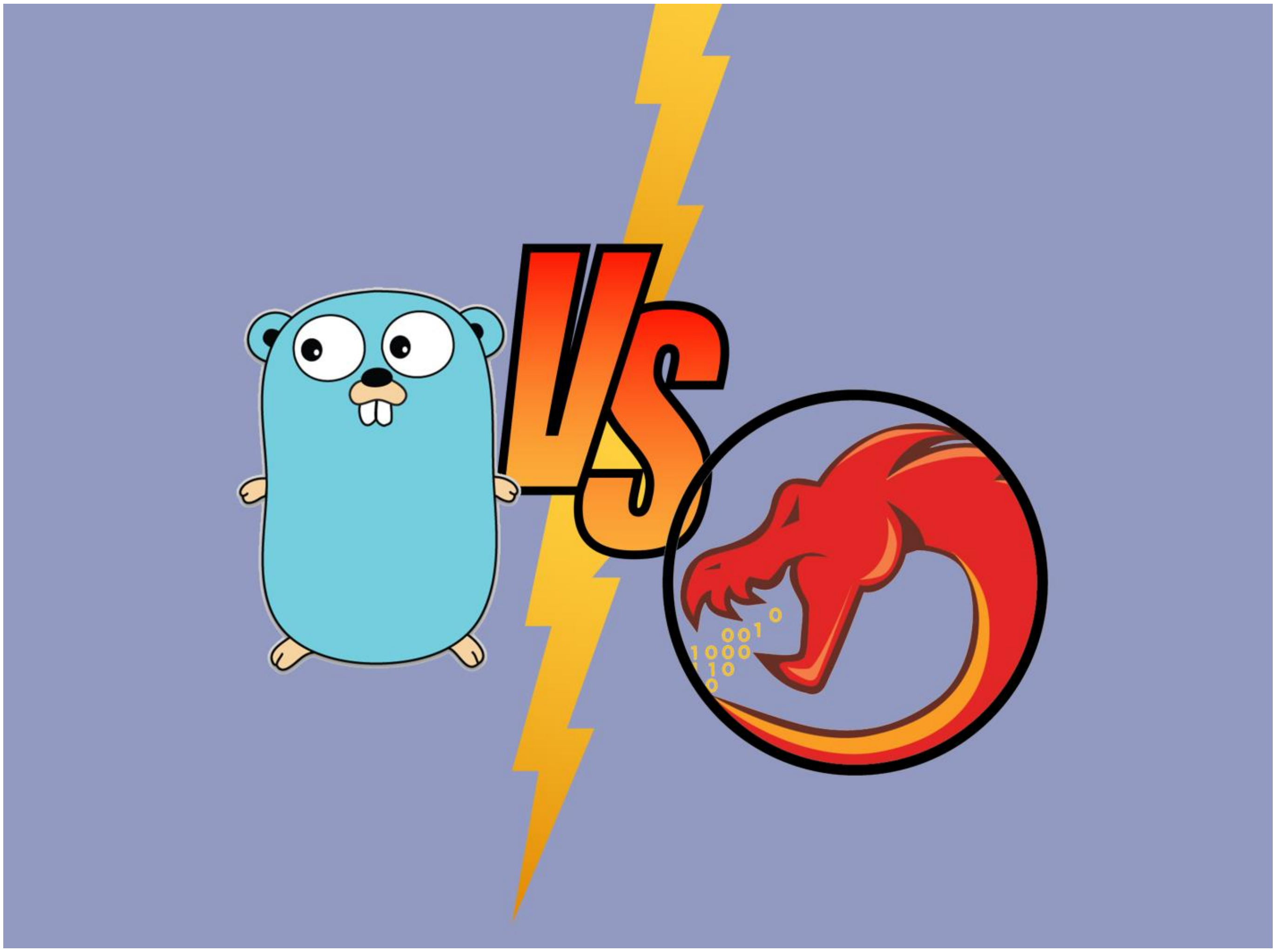
# Reverse engineering Golang binaries with Ghidra



CUJO AI

# Table of Contents

# Introduction

Go (also called Golang) is an open source programming language, designed by Google in 2007 and made available to the public in 2012. During the years it gained popularity among developers and as usually, it is not only used for good purposes, but attracted attention of malware developers as well. The fact that Go supports cross compiling to run binaries on various operating systems makes it a tempting choice for malware developers. The possibility to compile the same code for all major platforms (Windows, Linux, MacOS) makes the attackers life much easier, they don't have to develop and maintain different codebase for each target environment.

Some special features of the Go programming language give a hard time for reverse engineers when investigating Go binaries. Reverse engineering tools (e.g. disassemblers) can do a great job in analyzing binaries that are written in more popular languages (e.g. C, C++, .NET), but Go gives new challenges that makes the analysis more cumbersome.

Go binaries are usually statically linked, which means that all the necessary libraries are included in the compiled binary. This results in large binaries. On one hand this makes malware distribution more difficult for the attackers, but on the other hand some of the security products also have issues with handling such large files. It can help avoiding detection. The other advantage of statically linked binaries for the attackers is that the malware can run on the target systems without dependency issues.

As we see a continuous growth in the number of malware written in Go and we expect more families to emerge, we decided to dive deeper into the Go programming language and enhance our toolset to be more effective in investigating Go malware.

Below we will discuss two of the difficulties that reverse engineers have to face during Go binary analysis and we will show our solutions for those.

Ghidra is an open source reverse engineering tool developed by the National Security Agency, which we frequently use for static malware analysis. It is possible to create custom scripts and plugins for Ghidra to provide specific functionalities that are needed by researchers. We used this feature of Ghidra and created custom scripts to aid our Go binary analysis.

The topics discussed in this article were presented at the Hacktivity2020 online conference. The slides and other materials are available in our Github repository.

# Lost function names

The first issue is not specific to Go binaries, but stripped binaries in general. Compiled executable files can contain debug symbols which make debugging and analysis easier. When reverse engineering a program that was compiled with debugging information included, analysts can see not only memory addresses but also the names of the routines and variables. However, in order to reduce the size and make reverse engineering more difficult, malware authors usually compile the files without this information, creating so-called stripped binaries. In this case analysts cannot rely on the function names to help them finding their way around the code. For statically linked Go binaries, where all the necessary libraries are included, this can significantly slow down the analysis.

To illustrate this issue, we used simple “Hello Hacktivity” examples written in C[1] and Go[2] for comparison and compiled them to stripped binaries. Note the size difference of the two executables.


Ghidra function window lists all the defined functions within the binaries. In the non-stripped versions function names are nicely visible and provide a great help for reverse engineers.

Functions - 18 items

| Label | Location | Function Sign... | Function Size |
|---|---|---|---|
| _init | 00101000 | int _init(E... | 27 |
| __cxa_finalize | 00101040 | thunk undef... | 11 |
| puts | 00101050 | thunk int p... | 11 |
| _start | 00101060 | undefined _... | 47 |
| deregister_tm_clones | 00101090 | undefined d... | 34 |
| register_tm_clones | 001010c0 | undefined r... | 51 |
| __do_global_dtors_aux | 00101100 | undefined _... | 54 |
| frame_dummy | 00101140 | thunk undef... | 9 |
| main | 00101149 | undefined m... | 27 |
| __libc_csu_init | 00101170 | undefined _... | 101 |
| __libc_csu_fini | 001011e0 | undefined _... | 5 |
| _fini | 001011e8 | undefined _... | 13 |
| _ITM_deregisterTMCloneTable | 00105000 | thunk undef... | 1 |
| puts | 00105008 | thunk int p... | 1 |
| __libc_start_main | 00105010 | thunk undef... | 1 |
| __gmon_start__ | 00105018 | thunk undef... | 1 |
| _ITM_registerTMCloneTable | 00105020 | thunk undef... | 1 |
| __cxa_finalize | 00105028 | thunk undef... | 1 |

Filter:

Figure 1 - hello_c[3] function list

Functions - 1790 items

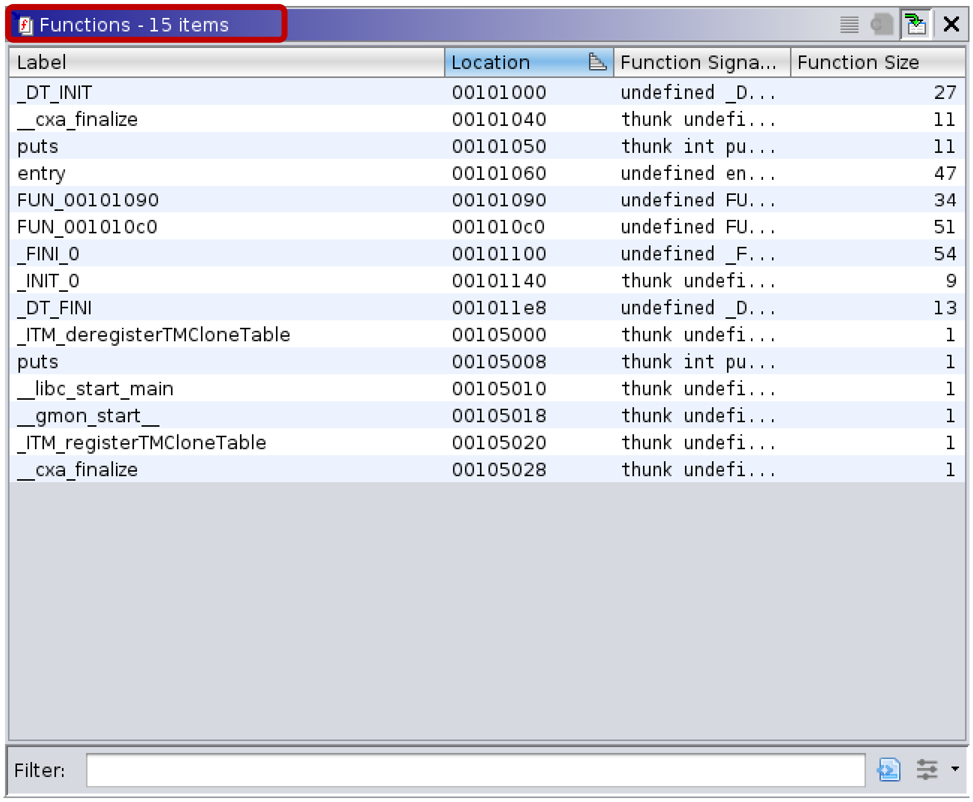
| Label | Location | Function Sign... | Function Size |
|---|---|---|---|
| internal/cpu.Initialize | 00401000 | undefined i... | 78 |
| internal/cpu.processOptions | 00401060 | undefined i... | 1877 |
| internal/cpu.indexByte | 004017c0 | undefined i... | 53 |
| internal/cpu.doinit | 00401800 | undefined i... | 1029 |
| internal/cpu.cpuid | 00401c20 | undefined i... | 27 |
| internal/cpu.xgetbv | 00401c40 | undefined i... | 17 |
| type..eq.internal/cpu.CacheLinePad | 00401c60 | undefined t... | 6 |
| type..eq.internal/cpu.option | 00401c80 | undefined t... | 165 |
| type..eq.[15]internal/cpu.option | 00401d40 | undefined t... | 139 |
| runtime/internal/sys.OnesCount64 | 00401de0 | undefined r... | 119 |
| runtime/internal/atomic.Cas64 | 00401e60 | undefined r... | 26 |
| runtime/internal/atomic.Casuintptr | 00401e80 | thunk undef... | 5 |
| runtime/internal/atomic.Storeuintptr | 00401ea0 | thunk undef... | 5 |
| runtime/internal/atomic.Store | 00401ec0 | undefined r... | 12 |
| runtime/internal/atomic.Store64 | 00401ee0 | undefined r... | 14 |
| internal/bytealg.init.0 | 00401f00 | undefined i... | 34 |
| cmpbody | 00401f40 | undefined c... | 569 |
| runtime.cmpstring | 00402180 | undefined r... | 30 |
| memeqbody | 004021a0 | undefined m... | 318 |
| runtime.memequal | 004022e0 | undefined r... | 36 |
| runtime.memequal_varlen | 00402320 | undefined r... | 35 |
| indexbytebody | 00402360 | undefined i... | 279 |
| internal/bytealg.IndexByteString | 00402480 | undefined i... | 24 |
| runtime.memhash128 | 004024a0 | undefined r... | 89 |

Filter:

Figure 2 - hello_go[5] function list

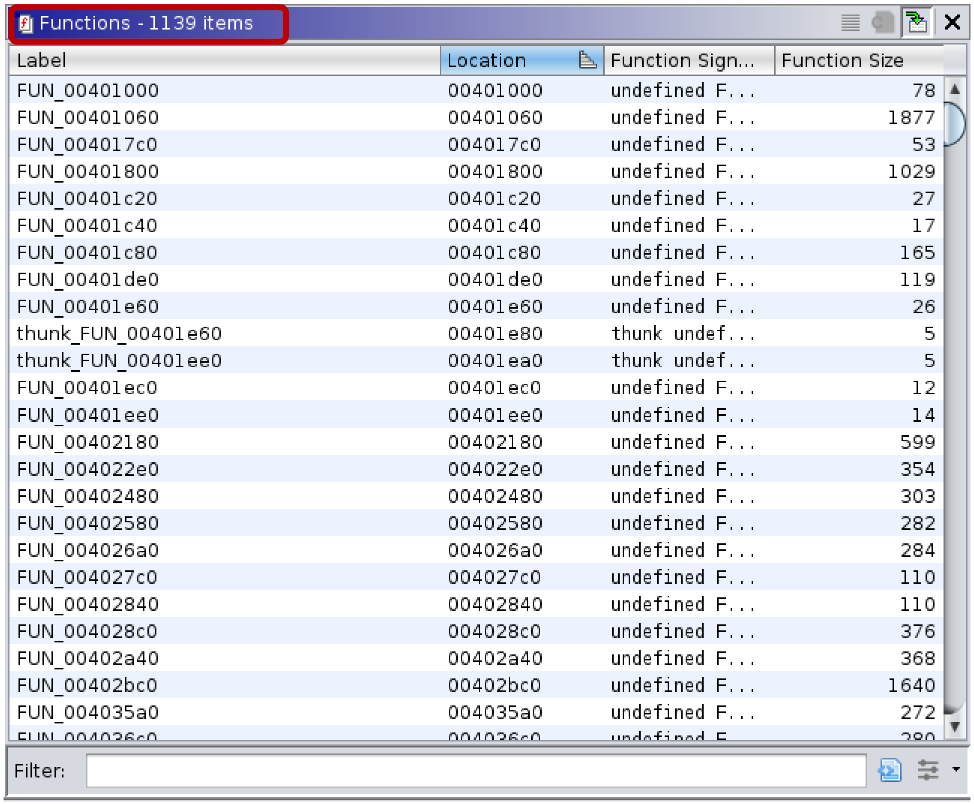
CUJO AI

For stripped binaries the function lists look the following:




| Label | Location | Function Signa... | Function Size |
|---|---|---|---|
| _DT_INIT | 00101000 | undefined _D... | 27 |
| __cxa_finalize | 00101040 | thunk undefi... | 11 |
| puts | 00101050 | thunk int pu... | 11 |
| entry | 00101060 | undefined en... | 47 |
| FUN_00101090 | 00101090 | undefined FU... | 34 |
| FUN_001010c0 | 001010c0 | undefined FU... | 51 |
| _FINI_0 | 00101100 | undefined _F... | 54 |
| _INIT_0 | 00101140 | thunk undefi... | 9 |
| _DT_FINI | 001011e8 | undefined _D... | 13 |
| _ITM_deregisterTMCloneTable | 00105000 | thunk undefi... | 1 |
| puts | 00105008 | thunk int pu... | 1 |
| __libc_start_main | 00105010 | thunk undefi... | 1 |
| __gmon_start__ | 00105018 | thunk undefi... | 1 |
| _ITM_registerTMCloneTable | 00105020 | thunk undefi... | 1 |
| __cxa_finalize | 00105028 | thunk undefi... | 1 |

Figure 3 - hello_c_strip[4] function list



Functions - 1139 items

| Label | Location | Function Sign... | Function Size |
|---|---|---|---|
| FUN_00401000 | 00401000 | undefined F... | 78 |
| FUN_00401060 | 00401060 | undefined F... | 1877 |
| FUN_004017c0 | 004017c0 | undefined F... | 53 |
| FUN_00401800 | 00401800 | undefined F... | 1029 |
| FUN_00401c20 | 00401c20 | undefined F... | 27 |
| FUN_00401c40 | 00401c40 | undefined F... | 17 |
| FUN_00401c80 | 00401c80 | undefined F... | 165 |
| FUN_00401de0 | 00401de0 | undefined F... | 119 |
| FUN_00401e60 | 00401e60 | undefined F... | 26 |
| thunk_FUN_00401e60 | 00401e80 | thunk undef... | 5 |
| thunk_FUN_00401ee0 | 00401ea0 | thunk undef... | 5 |
| FUN_00401ec0 | 00401ec0 | undefined F... | 12 |
| FUN_00401ee0 | 00401ee0 | undefined F... | 14 |
| FUN_00402180 | 00402180 | undefined F... | 599 |
| FUN_004022e0 | 004022e0 | undefined F... | 354 |
| FUN_00402480 | 00402480 | undefined F... | 303 |
| FUN_00402580 | 00402580 | undefined F... | 282 |
| FUN_004026a0 | 004026a0 | undefined F... | 284 |
| FUN_004027c0 | 004027c0 | undefined F... | 110 |
| FUN_00402840 | 00402840 | undefined F... | 110 |
| FUN_004028c0 | 004028c0 | undefined F... | 376 |
| FUN_00402a40 | 00402a40 | undefined F... | 368 |
| FUN_00402bc0 | 00402bc0 | undefined F... | 1640 |
| FUN_004035a0 | 004035a0 | undefined F... | 272 |

Filter:

Figure 4 - hello_go_strip[6] function list

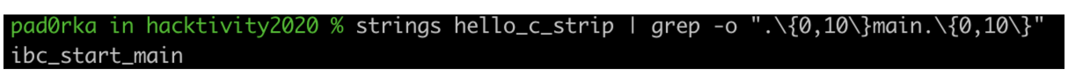
These examples nicely show that even a simple hello world Go binary is huge with more than a thousand functions and in the stripped version reverse engineers cannot rely on the function names to aid their analysis.

Note: Due to stripping not only the function names disappeared, but instead of 1790 defined functions only 1139 were recognized by Ghidra.

We were interested if there is any way to recover the function names within stripped binaries. First by a simple string search it can be checked if the function names are still available within the binaries. For the C example we were looking for the function name “main”, while in the Go example it is “main.main”.

```
pad0rka in hacktivity2020 % strings hello_c | grep -o ".\{0,10\}main.\{0,10\}"
ibc_start_main
ibc_start_main@@GLIBC_2.
main
```

Figure 5 - hello_c[3] strings – “main” was found



```
pad0rka in hacktivity2020 % strings hello_c_strip | grep -o ".\{0,10\}main.\{0,10\}"
ibc_start_main
```

Figure 6 - hello_c_strip[4] strings – “main” was not found

```
pad0rka in hacktivity2020 % strings hello_go | grep -o ".\{0,10\}main.\{0,10\}"
hasmain
edruntime.main not on m0
 p stateremaining pointe
 out of domainpanic whil
e space remainingreflect
routines (main called ru
runtime.main
runtime.main.func1
runtime.main.func2
main.main
main..inittask
runtime.main_init_done
runtime.mainStarted
runtime.mainPC
runtime.main
runtime.main.func1
runtime.main.func2
main.main
```

Figure 7 - hello_go[5] strings – “main.main” was found

```
pad0rka in hacktivity2020 % strings hello_go_strip | grep -o ".\{0,10\}main.\{0,10\}"
hasmain
edruntime.main not on m0
 p stateremaining pointe
 out of domainpanic whil
e space remainingreflect
routines (main called ru
runtime.main
runtime.main.func1
runtime.main.func2
main.main
```

Figure 8 - hello_go_strip[6] strings – “main.main” was found

While in the stripped C binary[4] the function name cannot be found with the strings utility, in the Go version[6] “main.main” is still available. This discovery gave us some hope that in stripped Go binaries function name recovery might be possible.
Loading the binary[6] to Ghidra and searching for the “main.main” string will show the exact location. As it can be seen on the image below the function name string is located within the .gopclntab section.

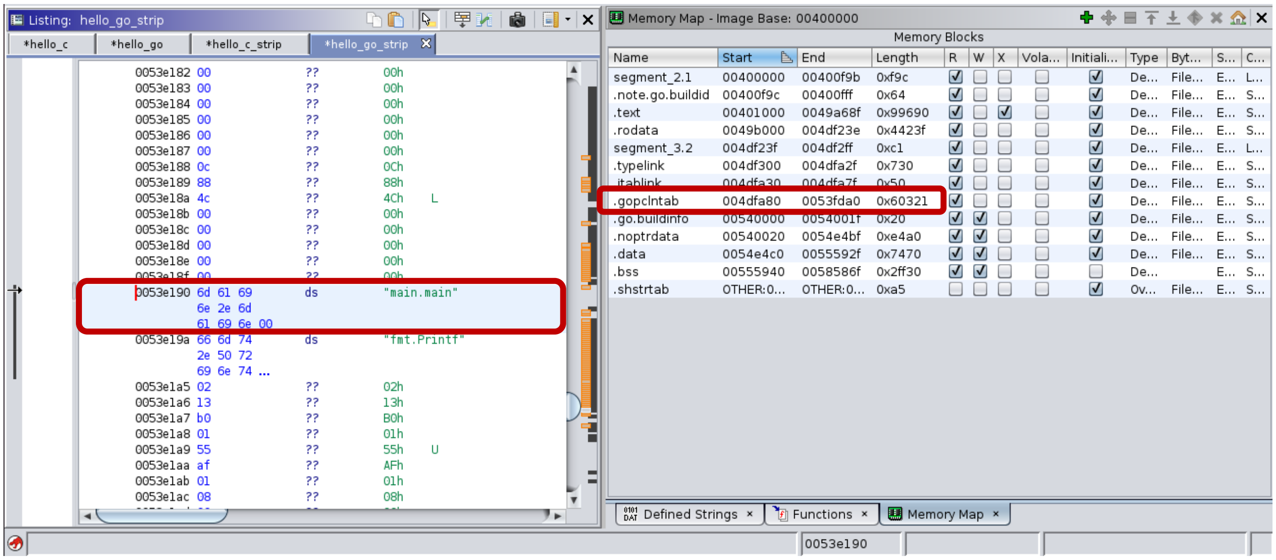
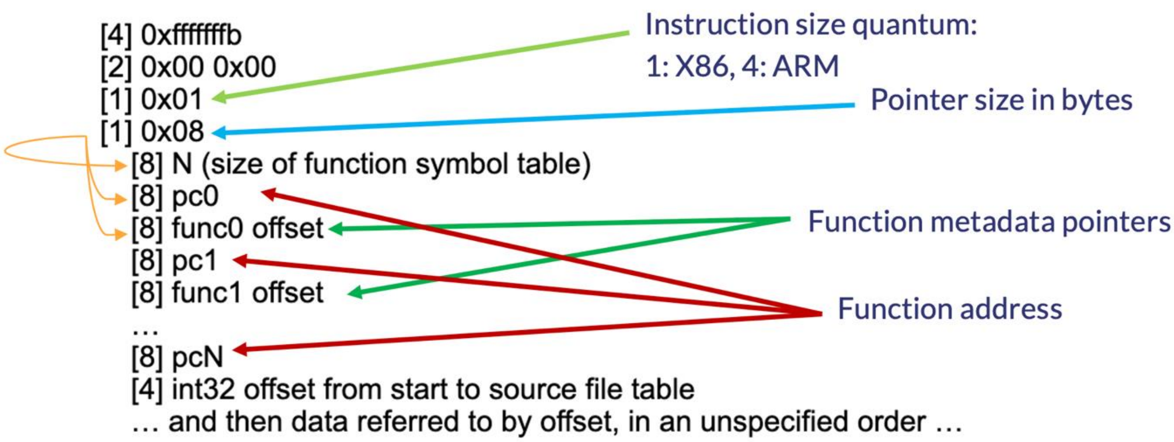


Figure 9 - hello_go_strip[6] main.main string in Ghidra

The pclntab structure is available since Go 1.2 and nicely documented. The structure starts with a magic value followed by information about the architecture. Then the function symbol table holds information about the functions within the binary. The address of the entry point of each function is followed by a function metadata table.



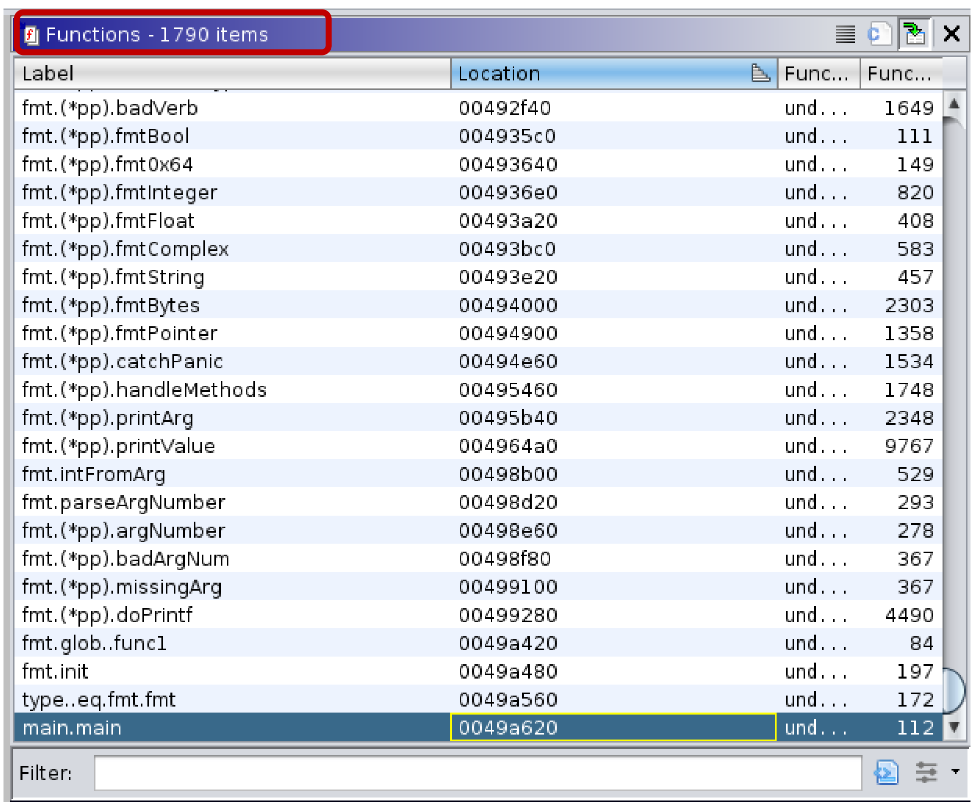

The function metadata table, among other important information, stores an offset to the function name.

```
struct       Func
{
        uintptr       entry;  // start pc
        int32 name;           // name (offset to C string)
        int32 args;           // size of arguments passed to function
        int32 frame;          // size of function frame, including saved caller PC
        int32         pcsp;               // pcsp table (offset to pcvalue table)
        int32         pcfile;          // pcfile table (offset to pcvalue table)
        int32         pcln;                // pcln table (offset to pcvalue table)
        int32         nfuncdata;          // number of entries in funcdata list
        int32         npcdata;           // number of entries in pcdata list
};
```

Using this information, it is possible to recover the function names. Our team created a script (go_func.py) for Ghidra to recover function names in stripped Go ELF files by executing the following steps:

- Locate pclntab structure
- Extract function addresses
- Find function name offsets

After executing our script not only the function names will be restored, but the previously unrecognized functions will be defined as well.

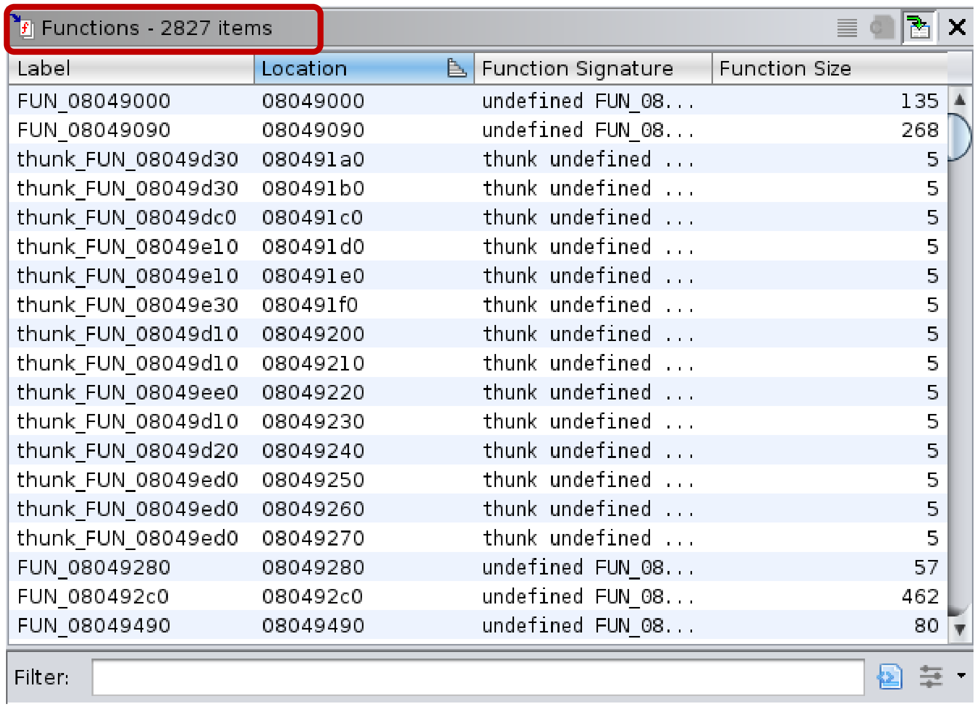

Functions - 1790 items

| Label | Location | Func... | Func... |
|---|---|---|---|
| fmt.(*pp).badVerb | 00492f40 | und... | 1649 |
| fmt.(*pp).fmtBool | 004935c0 | und... | 111 |
| fmt.(*pp).fmt0x64 | 00493640 | und... | 149 |
| fmt.(*pp).fmtInteger | 004936e0 | und... | 820 |
| fmt.(*pp).fmtFloat | 00493a20 | und... | 408 |
| fmt.(*pp).fmtComplex | 00493bc0 | und... | 583 |
| fmt.(*pp).fmtString | 00493e20 | und... | 457 |
| fmt.(*pp).fmtBytes | 00494000 | und... | 2303 |
| fmt.(*pp).fmtPointer | 00494900 | und... | 1358 |
| fmt.(*pp).catchPanic | 00494e60 | und... | 1534 |
| fmt.(*pp).handleMethods | 00495460 | und... | 1748 |
| fmt.(*pp).printArg | 00495b40 | und... | 2348 |
| fmt.(*pp).printValue | 004964a0 | und... | 9767 |
| fmt.intFromArg | 00498b00 | und... | 529 |
| fmt.parseArgNumber | 00498d20 | und... | 293 |
| fmt.(*pp).argNumber | 00498e60 | und... | 278 |
| fmt.(*pp).badArgNum | 00498f80 | und... | 367 |
| fmt.(*pp).missingArg | 00499100 | und... | 367 |
| fmt.(*pp).doPrintf | 00499280 | und... | 4490 |
| fmt.glob..func1 | 0049a420 | und... | 84 |
| fmt.init | 0049a480 | und... | 197 |
| type..eq.fmt.fmt | 0049a560 | und... | 172 |
| main.main | 0049a620 | und... | 112 |

Filter:

Figure 10 - hello_go_strip[6] function list after executing go_func.py

To see a real-world example let's look at an eCh0raix ransomware sample[9]:

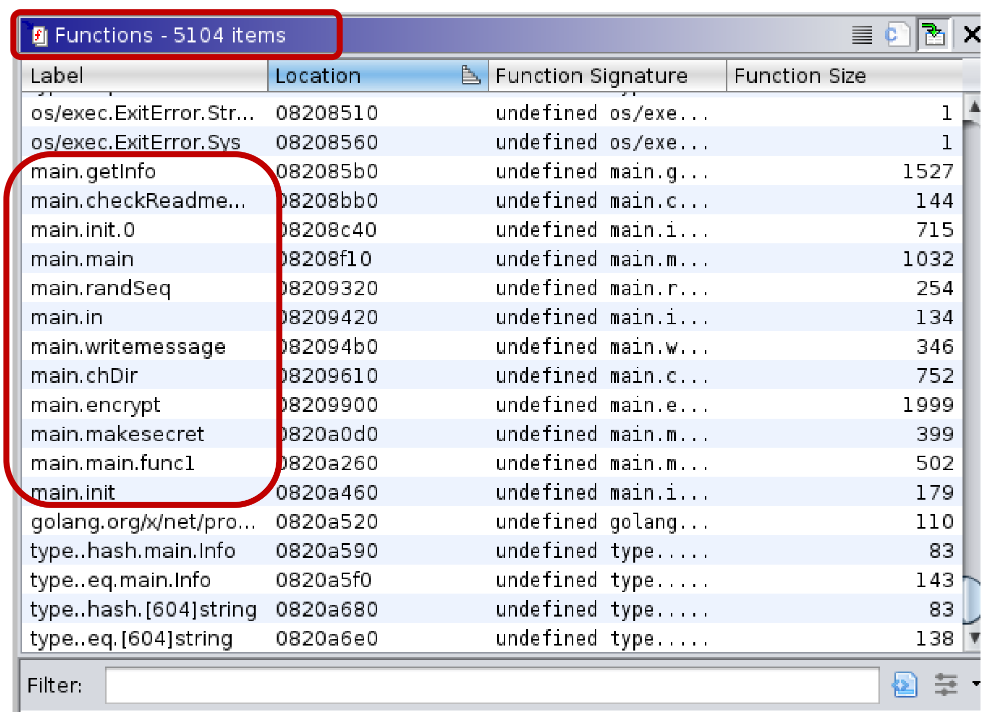


Functions - 2827 items

| Label | Location | Function Signature | Function Size |
| --- | --- | --- | --- |
| FUN_08049000 | 08049000 | undefined FUN_08... | 135 |
| FUN_08049090 | 08049090 | undefined FUN_08... | 268 |
| thunk_FUN_08049d30 | 080491a0 | thunk undefined ... | 5 |
| thunk_FUN_08049d30 | 080491b0 | thunk undefined ... | 5 |
| thunk_FUN_08049dc0 | 080491c0 | thunk undefined ... | 5 |
| thunk_FUN_08049e10 | 080491d0 | thunk undefined ... | 5 |
| thunk_FUN_08049e10 | 080491e0 | thunk undefined ... | 5 |
| thunk_FUN_08049e30 | 080491f0 | thunk undefined ... | 5 |
| thunk_FUN_08049d10 | 08049200 | thunk undefined ... | 5 |
| thunk_FUN_08049d10 | 08049210 | thunk undefined ... | 5 |
| thunk_FUN_08049ee0 | 08049220 | thunk undefined ... | 5 |
| thunk_FUN_08049d10 | 08049230 | thunk undefined ... | 5 |
| thunk_FUN_08049d20 | 08049240 | thunk undefined ... | 5 |
| thunk_FUN_08049ed0 | 08049250 | thunk undefined ... | 5 |
| thunk_FUN_08049ed0 | 08049260 | thunk undefined ... | 5 |
| thunk_FUN_08049ed0 | 08049270 | thunk undefined ... | 5 |
| FUN_08049280 | 08049280 | undefined FUN_08... | 57 |
| FUN_080492c0 | 080492c0 | undefined FUN_08... | 462 |
| FUN_08049490 | 08049490 | undefined FUN_08... | 80 |

Filter:

Figure 11 – eCh0raix[9] function list



Functions - 5104 items

| Label | Location | Function Signature | Function Size |
| --- | --- | --- | --- |
| os/exec.ExitError.Str... | 08208510 | undefined os/exe... | 1 |
| os/exec.ExitError.Sys | 08208560 | undefined os/exe... | 1 |
| main.getInfo | 082085b0 | undefined main.g... | 1527 |
| main.checkReadme... | 08208bb0 | undefined main.c... | 144 |
| main.init.0 | 08208c40 | undefined main.i... | 715 |
| main.main | 08208f10 | undefined main.m... | 1032 |
| main.randSeq | 08209320 | undefined main.r... | 254 |
| main.in | 08209420 | undefined main.i... | 134 |
| main.writemessage | 082094b0 | undefined main.w... | 346 |
| main.chDir | 08209610 | undefined main.c... | 752 |
| main.encrypt | 08209900 | undefined main.e... | 1999 |
| main.makesecret | 0820a0d0 | undefined main.m... | 399 |
| main.main.func1 | 0820a260 | undefined main.m... | 502 |
| main.init | 0820a460 | undefined main.i... | 179 |
| golang.org/x/net/pro... | 0820a520 | undefined golang... | 110 |
| type..hash.main.Info | 0820a590 | undefined type..... | 83 |
| type..eq.main.Info | 0820a5f0 | undefined type..... | 143 |
| type..hash.[604]string | 0820a680 | undefined type..... | 83 |
| type..eq.[604]string | 0820a6e0 | undefined type..... | 138 |

Filter:

Figure 12 - eCh0raix[9] function list after executing go_func.py

This example clearly shows how much help this simple function name recovery script can be during reverse engineering. Only by looking at the function names analysts can assume that they are dealing with a ransomware.

Note: In Windows Go binaries there is no specific section for the pclntab structure, rather researchers need explicitly search for the fields of this structure (e.g. magic value, possible field values). For MacOS _gopclntab section is available, similarly to .gopclntab in Linux binaries.

# Challenges

If a function name string is not defined by Ghidra, then the function name recovery script will fail to rename that specific function, since it cannot find the function name string at the given location. To overcome this issue our script always checks if a defined data type is located at the function name address and if not, then before renaming a function it tries to define a string data type at the given address.

In the below example the function name string “log.New” is not defined in an eCh0raix ransomware sample[9], so the corresponding function cannot be renamed without string creation first.

```
083aa0e4 6c          ??       6Ch    l
083aa0e5 6f          ??       6Fh    o
083aa0e6 67          ??       67h    g
083aa0e7 2e          ??       2Eh    .
083aa0e8 4e          ??       4Eh    N
083aa0e9 65          ??       65h    e
083aa0ea 77          ??       77h    w
083aa0eb 00          ??       00h
```

Figure 13 - eCh0raix[9] log.New function name undefined

```
                    **************************************************************
                    *                          FUNCTION                          *
                    **************************************************************
                    undefined FUN_08184fa0(undefined4 param_1, undefined4 pa…
     undefined         AL:1           <RETURN>
     undefined4        Stack[0x4]:4   param_1                          XREF[1]:     08184fc7(R)
     undefined4        Stack[0x8]:4   param_2                          XREF[2]:     08184fd8(R),
                                                                                    0818501d(R)
     undefined4        Stack[0xc]:4   param_3                          XREF[2]:     08184ff0(R),
                                                                                    0818500b(R)
     undefined4        Stack[0x10]:4  param_4                          XREF[1]:     08184fdf(R)
     undefined4        Stack[0x14]:4  param_5                          XREF[1]:     08184ff7(R)
     undefined4        Stack[0x18]:4  param_6                          XREF[1]:     08184ffe(W)
     undefined4        Stack[-0x4]:4  local_4                          XREF[1]:     08184fc3(R)
     undefined4        Stack[-0x8]:4  local_8                          XREF[1]:     08184fbb(*)
                    FUN_08184fa0                              XREF[2]:     0818502f(c),
                                                                           log.init:08186012(c)
08184fa0 65 8b 0d        MOV        ECX,dword ptr GS:[0x0]
         00 00 00 00
08184fa7 8b 89 fc        MOV        ECX,dword ptr [ECX + 0xfffffffc]
         ff ff ff
```

Figure 14 – eCh0raix[9] log.New function couldn't be renamed

The following lines in our script are responsible to solve this challenge:

```python
func_name = getDataAt(name_address)

#Try to define function name string.
if func_name is None:
    try:
        func_name = createAsciiString(name_address)
    except:
        print "ERROR: No name"
        continue
```

Figure 15 - go_func.py

# Unrecognized strings

The second issue that our scripts are solving is related to strings within Go binaries. Let's turn back to the "Hello Hacktivity" examples and take a look at the defined strings within Ghidra.
In the C binary[3] 70 strings are defined, among which "Hello, Hacktivity!" can be found. Meanwhile the Go binary[5] includes 6540 strings but searching for "hacktivity" gives no result. Having such a high number of strings already gives a hard time for reverse engineers to find the relevant ones, but in this case, the string that we would expect to find, is not even recognized by Ghidra.

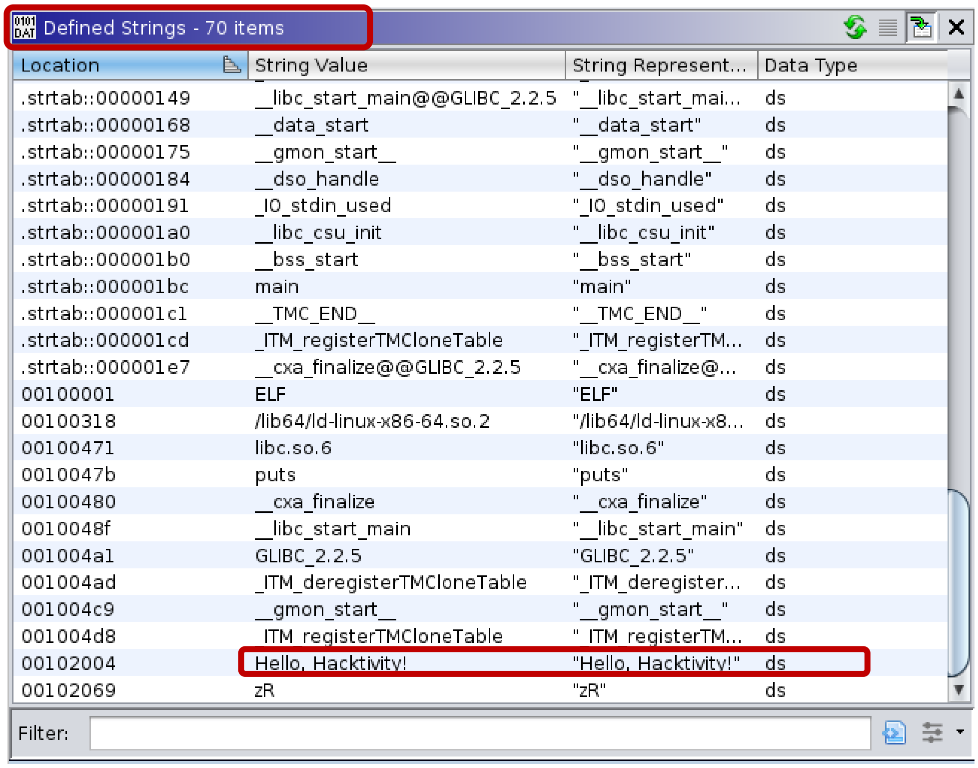



Figure 16 - hello_c[3] defined strings with “Hello, Hacktivity!”

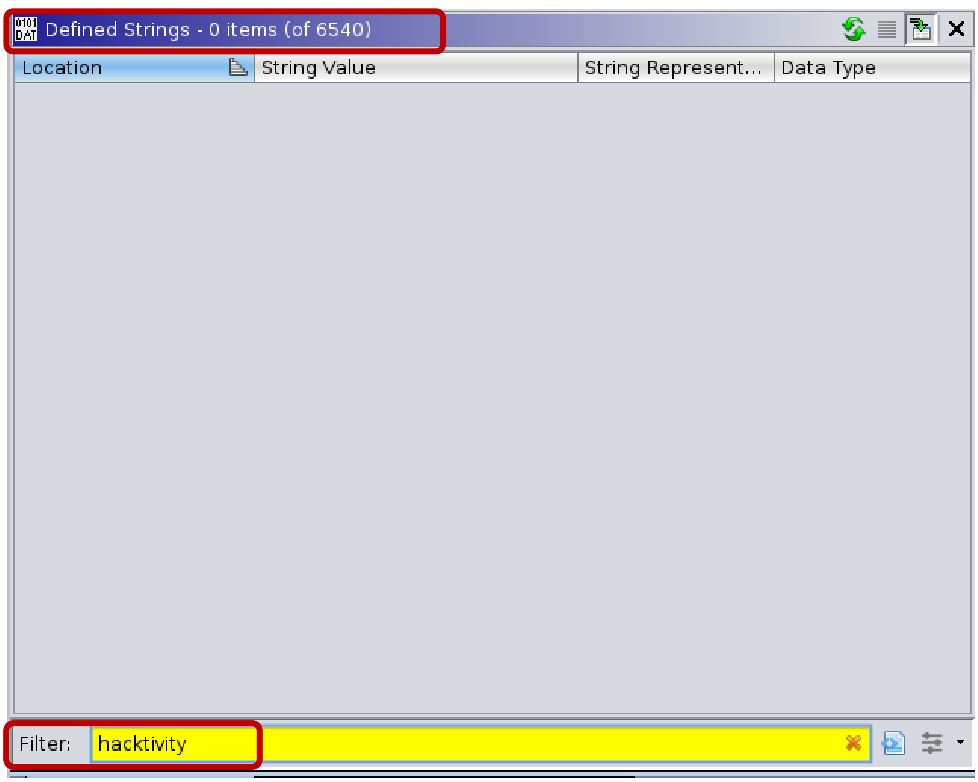



Figure 17 - hello_go[5] defined strings without "hacktivity"

To understand the problem here, the first step is to understand what a string in Go is. Unlike in C-like languages, where strings are sequence of characters terminated with a null character, in Go strings are sequences of bytes with a fixed length. Strings are Go specific structures, built up by a pointer to the location of the string and an integer, which is the length of the string.

```
type stringStruct struct {
        str unsafe.Pointer
        len int
}
```

These strings are stored within Go binaries as a large string blob, which consists of the concatenation of the strings without null character between them. So, while searching for “Hacktivity” using strings and grep gives the expected result in C, in case of Go a huge string blob is returned containing somewhere “hacktivity”.

```
pad0rka in hacktivity2020 % strings hello_c | grep Hacktivity
Hello, Hacktivity!
```

Figure 18 - hello_c[3] string search for "Hacktivity"

```
pad0rka in hacktivity2020 % strings hello_go | grep hacktivity
object is remotepacer: H_m_prev=reflect mismatchremote I/O errorruntime:  g:  g=runtime: addr = runtime:
 base = runtime: gp: gp=runtime: head = runtime: nelems=schedule: in cgosigaction failedtime: bad [0-9]*
workbuf is empty initialHeapLive= spinningthreads=, s.searchAddr = 0123456789ABCDEFX0123456789abcdefx119
2092895507812559604644775390625: missing method GC assist markingOld_North_ArabianOld_South_ArabianOther
_ID_ContinueSIGBUS: bus errorSIGCONT: continueSIGINT: interruptSentence_TerminalUnified_Ideographbad Tin
ySizeClassdebugPtrmask.lockentersyscallblockexec format errorfutexwakeup addr=g already scannedglobalAll
oc.mutexlocked m0 woke upmark - bad statusmarkBits overflowno data availablenotetsleepg on g0permission
deniedreflect.Value.Intreflect.Value.Lenreflect: New(nil)reflect: call of runtime/internal/runtime: leve
l = runtime: nameOff runtime: next_gc=runtime: pointer runtime: summary[runtime: textOff runtime: typeOf
f scanobject n == 0select (no cases)stack: frame={sp:swept cached spanthread exhaustionunknown caller pc
wait for GC cyclewrong medium type  but memory size  because dotdotdot to non-Go memory , locked to thre
ad298023223876953125Caucasian_AlbanianRFS specific errorRegional_IndicatorVariation_Selectorbad lfnode a
ddressbad manualFreeListconnection refusedfaketimeState.lockfile name too longforEachP: not donegarbage
collectionhello, hacktivity
```

Figure 19 - hello_go[5] string search for "hacktivity"

Since the definition of strings is different and as a result referencing them within the assembly code is also differ from the usual C-like solutions, Ghidra has a hard time to define the strings within Go binaries.

The string structure can be allocated in many different ways, it can be created statically or dynamically during runtime, it varies over architecture and even within one architecture multiple solutions are possible. Our team created two scripts to help Ghidra identifying strings.

# Dynamically allocated string structures

In the first case string structures are created runtime. A sequence of assembly instructions is responsible for setting up the structure before a string operation. Thanks to the different instruction sets it varies over architectures. In the next few paragraphs we will go through a couple of use cases and show the instruction sequences that our [script](script) (find_dynamic_strings.py) is looking for.

## x86

First let's start with the "Hello Hacktivity" example[5].

```
                    main.main                                      XREF[4]:     Entry Point(*),
                                                                                runtime.main:00434907…
                                                                                0049a68e(c),
                                                                                004c5b50(*)
0049a620 64 48 8b       MOV        RCX,qword ptr FS:[0xfffffff8]
         0c 25 f8
         ff ff ff
0049a629 48 3b 61 10    CMP        RSP,qword ptr [RCX + 0x10]
0049a62d 76 5a          JBE        LAB_0049a689
0049a62f 48 83 ec 58    SUB        RSP,0x58
0049a633 48 89 6c       MOV        qword ptr [RSP + local_8],RBP
         24 50
0049a638 48 8d 6c       LEA        RBP=>local_8,[RSP + 0x50]
         24 50
0049a63d 48 8b 05       MOV        RAX,qword ptr [os.Stdout]
         4c b3 0b 00
0049a644 48 8d 0d       LEA        RCX,[go.itab.*os.File,io.Writer]
         95 2e 04 00
0049a64b 48 89 0c 24    MOV        qword ptr [RSP]=>local_58,RCX=>go.itab.*os.File,io.Writer
0049a64f 48 89 44       MOV        qword ptr [RSP + local_50],RAX
         24 08                                                                          String location
0049a654 48 8d 05       LEA        RAX,[DAT_004bfc3d]
         e2 55 02 00
0049a65b 48 89 44       MOV        qword ptr [RSP + local_48],RAX=>DAT_004bfc3d
         24 10
0049a660 48 c7 44       MOV        qword ptr [RSP + local_40],0x12
         24 18 12
         00 00 00
0049a669 48 c7 44       MOV        qword ptr [RSP + local_38],0x0                               Length
         24 20 00
         00 00 00
0049a672 0f 57 c0       XORPS      XMM0,XMM0
0049a675 0f 11 44       MOVUPS     xmmword ptr [RSP + local_30],XMM0
         24 28
0049a67a e8 e1 82       CALL       fmt.Fprintf
```

Figure 20 - hello_go[5] dynamic allocation of string structure

```
                    DAT_004bfc3d
004bfc3d 68          ??          68h    h
004bfc3e 65          ??          65h    e
004bfc3f 6c          ??          6Ch    l
004bfc40 6c          ??          6Ch    l
004bfc41 6f          ??          6Fh    o
004bfc42 2c          ??          2Ch    ,
004bfc43 20          ??          20h
004bfc44 68          ??          68h    h
004bfc45 61          ??          61h    a
004bfc46 63          ??          63h    c
004bfc47 6b          ??          6Bh    k
004bfc48 74          ??          74h    t
004bfc49 69          ??          69h    i
004bfc4a 76          ??          76h    v
004bfc4b 69          ??          69h    i
004bfc4c 74          ??          74h    t
004bfc4d 79          ??          79h    y
004bfc4e 0a          ??          0Ah
```

Figure 21 - hello_go[5] undefined "hello, hacktivity" string

After executing the script, the code looks the following:

```
0049a654 48 8d 05       LEA        RAX,[s_hello,_hacktivity_004bfc3d]
         e2 55 02 00
0049a65b 48 89 44       MOV        qword ptr [RSP + local_48],RAX=>s_hello,_hacktivity_004bfc3d
         24 10
0049a660 48 c7 44       MOV        qword ptr [RSP + local_40],0x12
         24 18 12
         00 00 00
0049a669 48 c7 44       MOV        qword ptr [RSP + local_38],0x0
         24 20 00
         00 00 00
0049a672 0f 57 c0       XORPS      XMM0,XMM0
0049a675 0f 11 44       MOVUPS     xmmword ptr [RSP + local_30],XMM0
         24 28
0049a67a e8 e1 82       CALL       fmt.Fprintf
         ff ff
```

Figure 22 - hello_go[5] dynamic allocation of string structure after executing find_dynamic_strings.py

The string is defined:

```
                    s_hello,_hacktivity_004bfc3d              XREF[2]:     main.main:0049a654(*)...
                                                                           main.main:0049a65b(*)...
004bfc3d 68 65 6c        ds          "hello, hacktivity\n"
         6c 6f 2c
         20 68 61 ...
```

Figure 23 - hello_go[5] defined "hello hacktivity" string

And “hacktivity” can be found in the defined strings view in Ghidra:

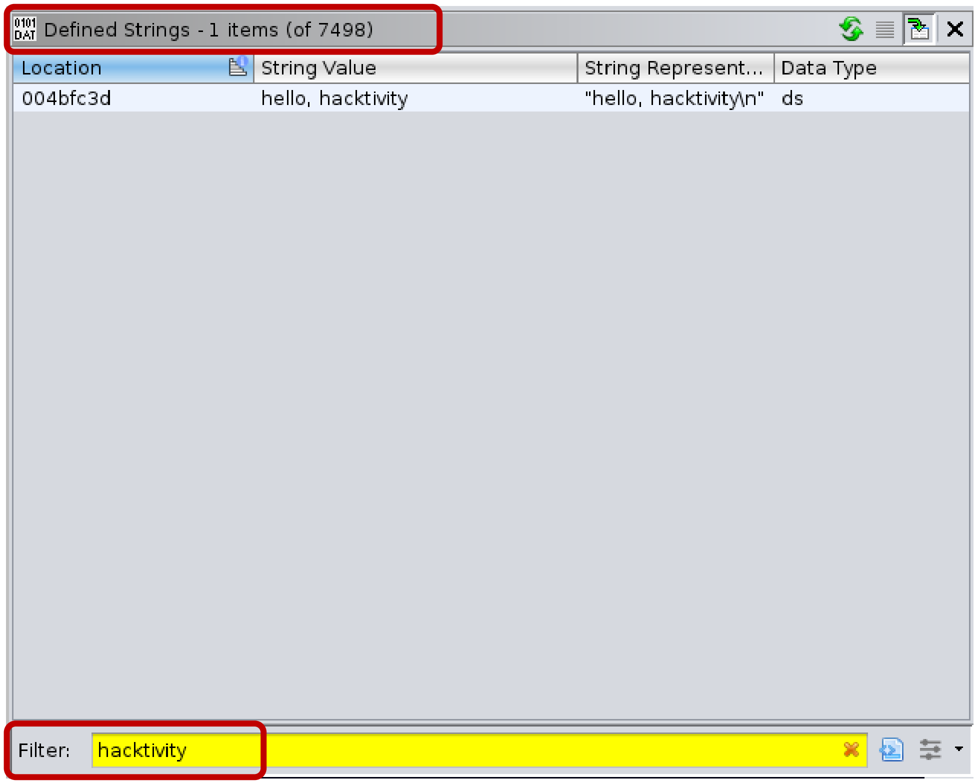



Figure 24 - hello_go[5] defined strings with "hacktivity"

The script is looking for the following instruction sequences in case of 32-bit and 64-bit x86 binaries:

```
#x86
#LEA REG, [STRING_ADDRESS]
#MOV [ESP + ..], REG
#MOV [ESP + ..], STRING_SIZE
```

```
08208bdc 8d 05 0e        LEA        EAX,[DAT_0827de0e]
         de 27 08
08208be2 89 44 24 0c     MOV        dword ptr [ESP + local_10],EAX=>DAT_0827de0e
08208be6 c7 44 24        MOV        dword ptr [ESP + local_c],0x17
         10 17 00
```

Figure 25 - eCh0raix[9] dynamic allocation of string structure

```
#x86_64
#LEA REG, [STRING_ADDRESS]
#MOV [RSP + ..], REG
#MOV [RSP + ..], STRING_SIZE
```

```
0049a654 48 8d 05      LEA       RAX,[DAT_004bfc3d]
         e2 55 02 00
0049a65b 48 89 44      MOV       qword ptr [RSP + local_48],RAX=>DAT_004bfc3d
         24 10
0049a660 48 c7 44      MOV       qword ptr [RSP + local_40],0x12
         24 18 12
```

Figure 26 - hello_go[5] dynamic allocation of string structure

## ARM

For the 32-bit ARM architecture an eCh0raix ransomware sample[10] will be used to illustrate the string recovery.

```
001e35bc 68 23 9f e5    ldr     r2,[PTR_DAT_001e392c]
001e35c0 10 20 8d e5    str     r2=>DAT_0025f560,[sp,#local_90]
001e35c4 44 20 a0 e3    mov     r2,#0x44
001e35c8 14 20 8d e5    str     r2,[sp,#local_8c]
001e35cc 18 00 8d e5    str     r0,[sp,#local_88]
001e35d0 1c 10 8d e5    str     r1,[sp,#local_84]
001e35d4 44 cc f9 eb    bl      runtime.concatstring3
```

Pointer to address containing the string location

String location

Length

Figure 27 - eCh0raix[10] dynamic allocation of string structure

```
                        PTR_DAT_001e392c                       XREF[1]:     main.main:001e35bc(R)
001e392c 60 f5 25 00    addr        DAT_0025f560
```

Figure 28 - eCh0raix[10] pointer to string address

```
                    DAT_0025f560                                          XREF[2]:     main.main:001e35c0(*),
                                                                                       001e392c(*)
0025f560 0d             ??         0Dh
0025f561 0a             ??         0Ah
0025f562 0d             ??         0Dh
0025f563 0a             ??         0Ah
0025f564 44             ??         44h    D
0025f565 6f             ??         6Fh    o
0025f566 20             ??         20h
0025f567 4e             ??         4Eh    N
0025f568 4f             ??         4Fh    O
0025f569 54             ??         54h    T
0025f56a 20             ??         20h
0025f56b 72             ??         72h    r
0025f56c 65             ??         65h    e
0025f56d 6d             ??         6Dh    m
0025f56e 6f             ??         6Fh    o
0025f56f 76             ??         76h    v
0025f570 65             ??         65h    e
0025f571 20             ??         20h
0025f572 74             ??         74h    t
0025f573 68             ??         68h    h
0025f574 69             ??         69h    i
0025f575 73             ??         73h    s
```

Figure 29 - eCh0raix[10] undefined string

After executing the script, the code looks the following:

```
001e35bc 68 23 9f e5    ldr      r2,[PTR_s__Do_NOT_remove_this_file_and_NOT_001e392c]
001e35c0 10 20 8d e5    str      r2=>s__Do_NOT_remove_this_file_and_NOT_0025f560,[sp,#local_90]
001e35c4 44 20 a0 e3    mov      r2,#0x44
001e35c8 14 20 8d e5    str      r2,[sp,#local_8c]
001e35cc 18 00 8d e5    str      r0,[sp,#local_88]
001e35d0 1c 10 8d e5    str      r1,[sp,#local_84]
001e35d4 44 cc f9 eb    bl       runtime.concatstring3
```

Figure 30 - eCh0raix[10] dynamic allocation of string structure after executing find_dynamic_strings.py

The pointer is renamed, and the string is defined:

```
                        PTR_s__Do_NOT_remove_this_file_and_NOT_001e392c  XREF[1]:     main.main:001e35bc(R)
001e392c 60 f5 25 00        addr        s__Do_NOT_remove_this_file_and_NOT_0025f560
```

Figure 31 - eCh0raix[10] pointer to string address after executing find_dynamic_strings.py

```
                     s__Do_NOT_remove_this_file_and_NOT_0025f560      XREF[2]:     main.main:001e35c0(*),
                                                                                   001e392c(*)
0025f560 0d 0a 0d          ds         "\r\n\r\nDo NOT remove this file and NOT remove last line in this file!\r\n"
         0a 44 6f
         20 4e 4f ...
```

Figure 32 – eCh0raix[10] defined string after executing find_dynamic_strings.py

The script is looking for the following instruction sequence in case of 32-bit ARM binaries:

```
#ARM, 32-bit
#LDR REG, [STRING_ADDRESS_POINTER]
#STR REG, [SP, ..]
#MOV REG, STRING_SIZE
#STR REG, [SP, ..]
```

For the 64-bit ARM architecture a Kaiji sample[12] will be used to illustrate the string recovery. Here two instruction sequences are used, that only differ in one instruction.

```
                         LAB_0020b59c                                  XREF[2]:     0020b814(j), 0020b988(j)
0020b59c 00 04 00 b0     adrp       x0,0x28c000
0020b5a0 00 c4 1c 91     add        x0,x0,#0x731
0020b5a4 e0 07 00 f9     str        x0=>DAT_0028c731,[sp, #local_68]
0020b5a8 e0 07 7e b2     orr        x0,xzr,#0xc                  String location
0020b5ac e0 0b 00 f9     str        x0,[sp, #local_60]
0020b5b0 e4 d3 ff 97     bl         ddos.PathExists              Length

0020b5b4 e0 63 40 39     ldrb       w0,[sp, #local_58]
0020b5b8 60 05 00 b5     cbnz       x0,LAB_0020b664

                         LAB_0020b5bc                                  XREF[2]:     0020b680(j), 0020b7f4(j)
0020b5bc 00 04 00 f0     adrp       x0,0x28e000
0020b5c0 00 84 28 91     add        x0,x0,#0xa21
0020b5c4 e0 07 00 f9     str        x0=>DAT_0028ea21,[sp, #local_68]
0020b5c8 80 02 80 d2     mov        x0,#0x14                     String location
0020b5cc e0 0b 00 f9     str        x0,[sp, #local_60]
0020b5d0 dc d3 ff 97     bl         ddos.PathExists              Length
0020b5d4 e0 63 40 39     ldrb       w0,[sp, #local_58]
0020b5d8 80 00 00 b5     cbnz       x0,LAB_0020b5e8
```

Figure 33 – Kaiji[12] dynamic allocation of string structure

After executing the script, the code looks the following:

```
                         LAB_0020b59c                                  XREF[2]:     0020b814(j), 0020b988(j)
0020b59c 00 04 00 b0     adrp       x0,0x28c000
0020b5a0 00 c4 1c 91     add        x0,x0,#0x731
0020b5a4 e0 07 00 f9     str        x0=>s_/etc/init.d/_0028c731,[sp, #local_68]
0020b5a8 e0 07 7e b2     orr        x0,xzr,#0xc
0020b5ac e0 0b 00 f9     str        x0,[sp, #local_60]
0020b5b0 e4 d3 ff 97     bl         ddos.PathExists

0020b5b4 e0 63 40 39     ldrb       w0,[sp, #local_58]
0020b5b8 60 05 00 b5     cbnz       x0,LAB_0020b664

                         LAB_0020b5bc                                  XREF[2]:     0020b680(j), 0020b7f4(j)
0020b5bc 00 04 00 f0     adrp       x0,0x28e000
0020b5c0 00 84 28 91     add        x0,x0,#0xa21
0020b5c4 e0 07 00 f9     str        x0=>s_/etc/systemd/system/_0028ea21,[sp, #local_68]
0020b5c8 80 02 80 d2     mov        x0,#0x14
0020b5cc e0 0b 00 f9     str        x0,[sp, #local_60]
0020b5d0 dc d3 ff 97     bl         ddos.PathExists
0020b5d4 e0 63 40 39     ldrb       w0,[sp, #local_58]
0020b5d8 80 00 00 b5     cbnz       x0,LAB_0020b5e8
```

Figure 34 – Kaiji[12] dynamic allocation of string structure after executing find_dynamic_strings.py

The strings are defined:

```
                      s_/etc/init.d/_0028c731                     XREF[1]:     main.runkshell:0020b5a4(*)
0028c731 2f 65 74         ds          "/etc/init.d/"
         63 2f 69
         6e 69 74 ...


                      s_/etc/systemd/system/_0028ea21             XREF[1]:     main.runkshell:0020b5c4(*)
0028ea21 2f 65 74         ds          "/etc/systemd/system/"
         63 2f 73
         79 73 74 ...
```

Figure 35 – Kaiji[12] defined strings after executing find_dynamic_strings.py

The script is looking for the following instruction sequences in case of 64-bit ARM binaries:

```
#ARM, 64-bit - version 1
#ADRP REG, [STRING_ADDRESS_START]
#ADD REG, REG, INT
#STR REG, [SP, ..]
#ORR REG, REG, STRING_SIZE
#STR REG, [SP, ..]

#ARM, 64-bit - version 2
#ADRP REG, [STRING_ADDRESS_START]
#ADD REG, REG, INT
#STR REG, [SP, ..]
#MOV REG, STRING_SIZE
#STR REG, [SP, ..]
```

As the above examples show, after executing the script dynamically allocated string structures can be recovered. It gives a great help to reverse engineers to read the assembly code or look for interesting strings within the defined string window in Ghidra.

## Challenges

The biggest drawback of this approach is that for each architecture and even for different solutions within the same architecture a new branch has to be added to the script. Also, it is very easy to evade these predefined instruction sets. In the example below in a Kaiji 64-bit ARM malware sample[12] the length of the string is moved to a register earlier than our script would expect, therefore this string will be missed.

Length

```
001fd734 21 01 80 d2     mov        param_2,#0x9
001fd738 e1 4b 00 f9     str        param_2,[sp, #local_c0]
001fd73c 62 04 00 d0     adrp       param_3,0x28b000
001fd740 42 fc 2f 91     add        param_3=>DAT_0028bbff,param_3,#0xbff
001fd744 e2 4f 00 f9     str        param_3=>DAT_0028bbff,[sp, #local_b8]
001fd748 e1 53 00 f9     str        param_2,[sp, #local_b0]
```

String location

Figure 36 – Kaiji[12] dynamic allocation of string structure in an unusual way

```
                 DAT_0028bbff                          XREF[6]:    ddos.sshgo:001fd740(*),
                                                                   ddos.sshgo:001fd744(*),
                                                                   ddos.sshgo:001fd788(*),
                                                                   ddos.sshgo:001fd7a4(*),
                                                                   ddos.sshgo:001fd7c0(*),
                                                                   ddos.sshgo:001fd7dc(*)
  0028bbff 6c          ??        6Ch    l
  0028bc00 69          ??        69h    i
  0028bc01 6e          ??        6Eh    n
  0028bc02 75          ??        75h    u
  0028bc03 78          ??        78h    x
  0028bc04 5f          ??        5Fh    _
  0028bc05 61          ??        61h    a
  0028bc06 72          ??        72h    r
  0028bc07 6d          ??        6Dh    m
```

Figure 37 – Kaiji[12] undefined string

# Statically allocated string structures

In the next case our script (find_static_strings.py) looks for string structures that are statically allocated, meaning the string pointer is followed by the string length within the data section of the code.

To illustrate this let's look at the x86 eCh0raix ransomware sample[9].

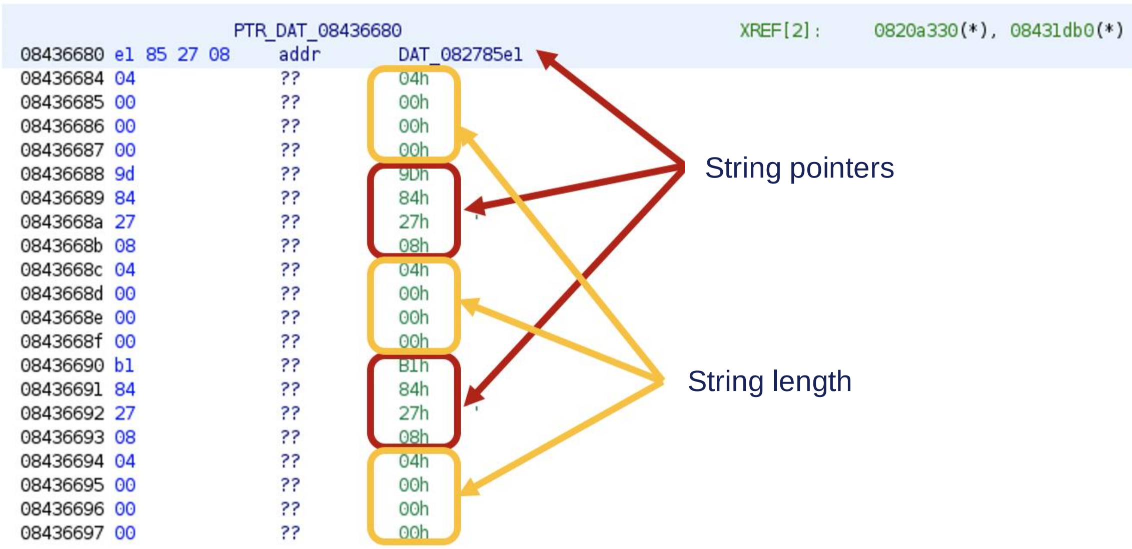



Figure 38 - eCh0raix[9] static allocation of string structures

On the image above string pointers are followed by string length values, however Ghidra couldn’t recognize the addresses, neither the integer data types, except for the first pointer, which is directly referenced from the code.

```
0820a30f 8b 44 24 20    MOV     EAX,dword ptr [ESP + 0x20]
0820a313 89 04 24       MOV     dword ptr [ESP],EAX
0820a316 8b 44 24 1c    MOV     EAX,dword ptr [ESP + 0x1c]
0820a31a 89 44 24 04    MOV     dword ptr [ESP + 0x4],EAX
0820a31e 8b 05 b0       MOV     EAX,dword ptr [PTR_PTR_DAT_08431db0]
         1d 43 08
0820a324 8b 0d b4       MOV     ECX,dword ptr [DAT_08431db4]
         1d 43 08
0820a32a 8b 15 b8       MOV     EDX,dword ptr [DAT_08431db8]
         1d 43 08
0820a330 89 44 24 08    MOV     dword ptr [ESP + 0x8],EAX=>PTR_DAT_08436680
0820a334 89 4c 24 0c    MOV     dword ptr [ESP + 0xc],ECX
0820a338 89 54 24 10    MOV     dword ptr [ESP + 0x10],EDX
0820a33c e8 df f0       CALL    FUN_08209420
         ff ff
```
Figure 39 – eCh0raix[9] pointer

Following the string addresses, the undefined strings can be found.

```
                             DAT_082785e1                                    XREF[1]:     08436680(*)
082785e1 2e              ??         2Eh    .
082785e2 64              ??         64h    d
082785e3 61              ??         61h    a
082785e4 74              ??         74h    t
082785e5 2e              ??         2Eh    .
082785e6 64              ??         64h    d
082785e7 62              ??         62h    b
082785e8 30              ??         30h    0
082785e9 2e              ??         2Eh    .
082785ea 64              ??         64h    d
082785eb 62              ??         62h    b
082785ec 61              ??         61h    a
082785ed 2e              ??         2Eh    .
082785ee 64              ??         64h    d
082785ef 62              ??         62h    b
082785f0 66              ??         66h    f
082785f1 2e              ??         2Eh    .
082785f2 64              ??         64h    d
082785f3 62              ??         62h    b
082785f4 6d              ??         6Dh    m
```

Figure 40 - eCh0raix[9] undefined strings

After executing the script, the string addresses will be defined, along with the string length values and the strings themselves.

```
                             PTR_s_.dat_08436680                             XREF[2]:     0820a330(*), 08431db0(*)
08436680 e1 85 27 08     addr       s_.dat_082785e1
08436684 04 00 00 00     int        4h
08436688 9d 84 27 08     addr       s_.lst_0827849d
0843668c 04 00 00 00     int        4h
08436690 b1 84 27 08     addr       s_.602_082784b1
08436694 04 00 00 00     int        4h
08436698 e5 82 27 08     addr       s_.7z_082782e5
0843669c 03 00 00 00     int        3h
084366a0 17 90 27 08     addr       s_.7-zip_08279017
084366a4 06 00 00 00     int        6h
084366a8 c1 84 27 08     addr       s_.abw_082784c1
084366ac 04 00 00 00     int        4h
084366b0 c5 84 27 08     addr       s_.act_082784c5
084366b4 04 00 00 00     int        4h
084366b8 11 8c 27 08     addr       s_.adoc_08278c11
084366bc 05 00 00 00     int        5h
084366c0 d9 84 27 08     addr       s_.aim_082784d9
084366c4 04 00 00 00     int        4h
084366c8 e1 84 27 08     addr       s_.ans_082784e1
084366cc 04 00 00 00     int        4h
```

String length

String pointers

Figure 41 - eCh0raix[9] static allocation of string structures after executing find_static_strings.py

```
                    s_.dat_082785e1                                    XREF[2]:     08436680(*), 084378b0(*)
082785e1 2e 64 61 74     ds          ".dat"

                    s_.db0_082785e5                                    XREF[1]:     08437248(*)
082785e5 2e 64 62 30     ds          ".db0"

                    s_.dba_082785e9                                    XREF[1]:     08437250(*)
082785e9 2e 64 62 61     ds          ".dba"

                    s_.dbf_082785ed                                    XREF[1]:     08437258(*)
082785ed 2e 64 62 66     ds          ".dbf"

                    s_.dbm_082785f1                                    XREF[1]:     08436ed8(*)
082785f1 2e 64 62 6d     ds          ".dbm"

                    s_.dbx_082785f5                                    XREF[1]:     08437260(*)
082785f5 2e 64 62 78     ds          ".dbx"

                    s_.dcr_082785f9                                    XREF[2]:     084369d8(*), 08437268(*)
082785f9 2e 64 63 72     ds          ".dcr"

                    s_.der_082785fd                                    XREF[2]:     08436d70(*), 08437270(*)
082785fd 2e 64 65 72     ds          ".der"
```
Figure 42 – eCh0raix[9] defined strings after executing find_static_strings.py

## Challenges

To eliminate false positives, we limit the string length, search only for printable characters and only in data sections of the binaries. Obviously, as a result of these limitations strings can be easily missed. If you use the script feel free to experiment with it, change the values and find the best settings for your analysis. The following lines in the code are responsible for the length and character set limitations:

```
#Look for strings with printable characters only to eliminate FPs.
def isPrintable(s, l):
    for i in range(l):
        if getByte(s) not in range(32,126):
            return False
        s = s.add(1)
    return True
```
Figure 43 - find_static_strings.py

```
length = getInt(length_address)
#Set the possible length to eliminate FPs.
if length not in range(1,100):
    continue
```
Figure 44 - find_static_strings.py

# Further challenges in string recovery

It can happen that Ghidra auto analysis falsely identify certain data types. When it happens, our script will fail to create the correct data at that specific location. To overcome this issue, first the incorrect data type has to be removed, then the new one can be created.
As an example, let's take a look at the eCh0riax ransomware[9] with statically allocated string structures.

```
                    PTR_DAT_08431980                              XREF[1]:     main.init.0:08208cec(R)
08431980 15 6f 28 08      addr          DAT_08286f15
                    DAT_08431984                                  XREF[1]:     main.init.0:08208cf2(R)
08431984 39 00 00 00      undefined4    00000039h
                    PTR_DAT_08431988                              XREF[1]:     main.getInfo:08208629(R)
08431988 bb c7 27 08      addr          DAT_0827c7bb
                    DAT_0843198c                                  XREF[1]:     main.getInfo:08208623(R)
0843198c 13 00 00 00      undefined4    00000013h
                    PTR_DAT_08431990                              XREF[1]:     net.readHosts:081448a0(R)
08431990 cc a0 27 08      addr          DAT_0827a0cc
                    DAT_08431994                                  XREF[1]:     net.readHosts:08144896(R)
08431994 0a 00 00 00      undefined4    0000000Ah

                              DAT_08286f15
                    08286f15 68          ??          68h     h
                    08286f16 74          ??          74h     t
                    08286f17 74          ??          74h     t
                    08286f18 70          ??          70h     p
                    08286f19 3a          ??          3Ah     :
                    08286f1a 2f          ??          2Fh     /
                    08286f1b 2f          ??          2Fh     /
                    08286f1c 73          ??          73h     s
                    08286f1d 67          ??          67h     g
                    08286f1e 33          ??          33h     3
                    08286f1f 64          ??          64h     d
                    08286f20 77          ??          77h     w
```

Figure 45 – eCh0raix[9] static allocation of string structures

Here the addresses are correctly identified, however the string length values, that supposed to be integer data types, are falsely defined as undefined4 values.

The following lines in our script are responsible for removing the incorrect data types:

```python
if getDataAt(length_address) is not None:
    data_type = getDataAt(length_address).getDataType()
    #Remove undefined data to be able to create int.
    #Keep an eye on other predefined data types.
    if data_type.getName() in ["undefined4", "undefined8"]:
        removeData(getDataAt(length_address))
```

Figure 46 - find_static_strings.py

After executing the script all the data types are correctly identified and the strings are defined.

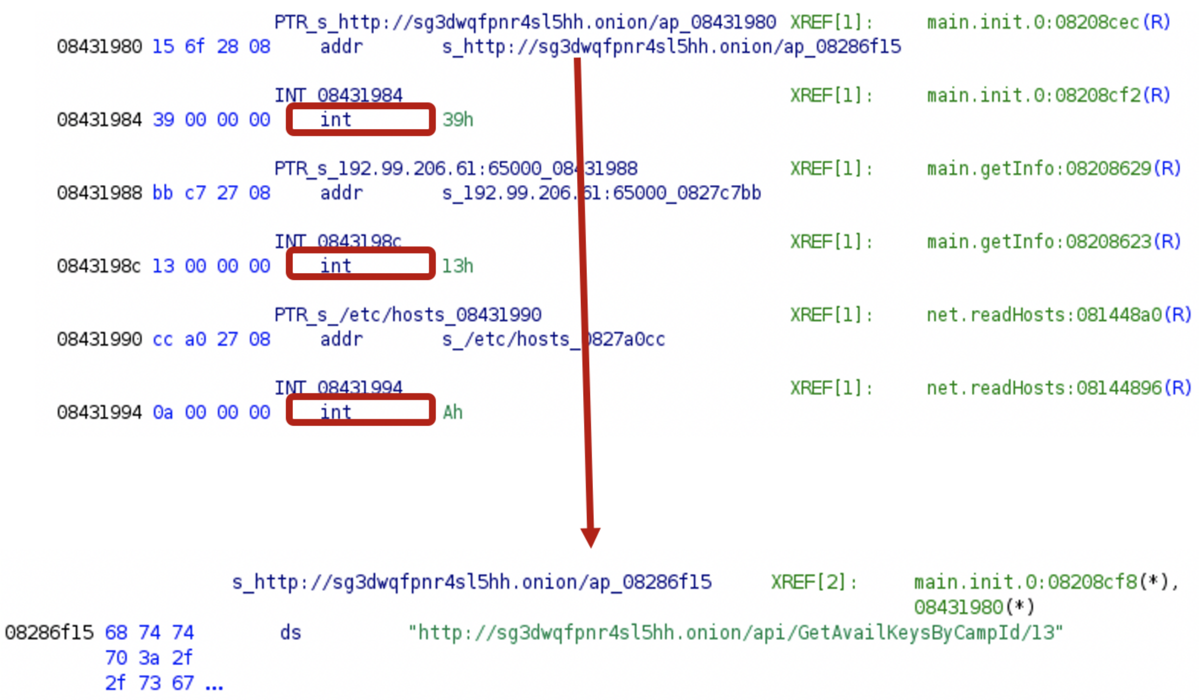



Figure 47 - eCh0raix[9] static allocation of string structures after executing find_static_strings.py

Another issue is coming from the fact that in Go binaries strings are stored concatenated, in a large string blob. In certain cases, Ghidra define these blobs as one string. These can be identified by the high number of offcut references. Offcut references are references to certain parts of the defined string, not the address where the string starts, rather somewhere inside the string.

The example below is from an ARM Kaiji sample[12].

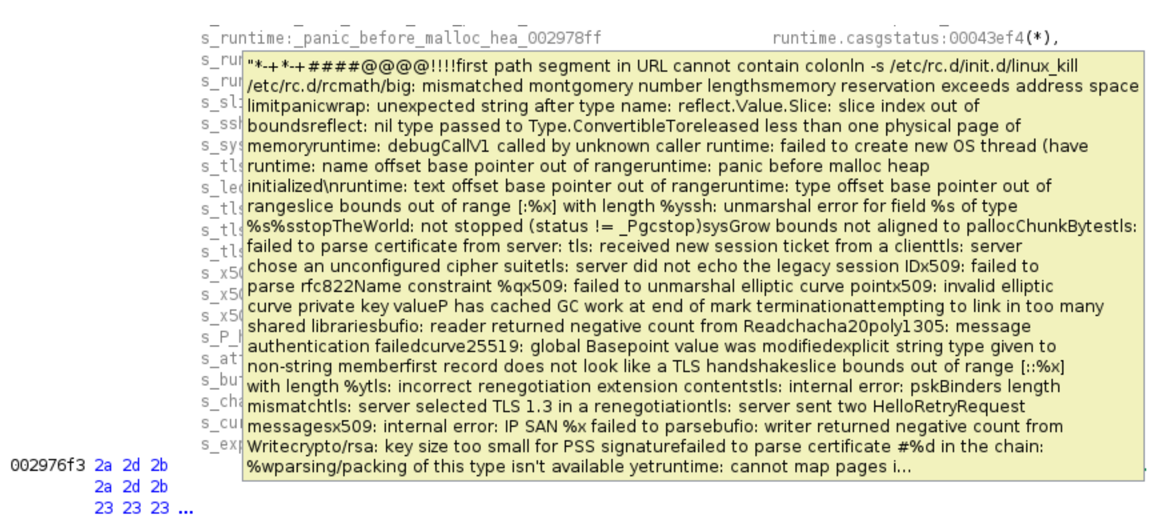



Figure 48 – Kaiji[12] falsely defined string in Ghidra

```
                       s_runtime:_panic_before_malloc_hea_002978ff            runtime.casgstatus:00043ef4(*),
                       s_runtime:_text_offset_base_pointe_0029792d            runtime.doInit:0004eefc(*),
                       s_runtime:_type_offset_base_pointe_0029795b            runtime.sigpanic:00055da4(*),
                       s_slice_bounds_out_of_range_[:%x]_w_00297989           runtime.sigpanic:00055de4(*),
                       s_ssh:_unmarshal_error_for_field_%_002979b7            runtime.sigpanic:00055f24(*),
                       s_sysGrow_bounds_not_aligned_to_pa_00297a13            runtime.sigpanic:00055f64(*),
                       s_tls:_failed_to_parse_certificate_00297a41            runtime.getStackMap:0005a7d4(*),
                       s_led_to_parse_certificate_from_se_00297a49            runtime.morestackc:0005a834(*),
                       s_tls:_received_new_session_ticket_00297a6f            runtime.resolveNameOff:00065b1c(...
                       s_tls:_server_chose_an_unconfigure_00297a9d
                       s_tls:_server_did_not_echo_the_leg_00297acb
                       s_x509:_failed_to_parse_rfc822Name_00297af9
                       s_x509:_failed_to_unmarshal_ellipt_00297b27
                       s_x509:_invalid_elliptic_curve_pri_00297b55
                       s_P_has_cached_GC_work_at_end_of_m_00297b83
                       s_attempting_to_link_in_too_many_s_00297bb2
                       s_bufio:_reader_returned_negative_c_00297be1
                       s_chacha20poly1305:_message_authen_00297c10
                       s_curve25519:_global_Basepoint_val_00297c3f
                       s_explicit_string_type_given_to_no_00297c6e
002976f3 2a 2d 2b        ds        "*-+*-+####@@@@!!!!first path segment in URL cannot contain colonln -s /etc/rc.d.
         2a 2d 2b
         23 23 23 ...
```

Figure 49 – Kaiji[12] offcut references of a falsely defined string

To find falsely defined strings, one can use the defined strings window of Ghidra and sort the strings by offcut reference count. Large strings with numerous offcut references can be undefined manually before executing the string recovery scripts, so the scripts can successfully create the correct string data types.

Defined Strings - 10814 items

| Location | String Value | Data Type | Byte Count | Offcut Reference Count |
|---|---|---|---|---|
| 0022073d | certificateAuthorities | ds | 23 | 1 |
| 00220ec1 | ReplaceAllLiteralString | ds | 24 | 1 |
| 00220ef5 | responseMessageReceived | ds | 24 | 1 |
| 00220f29 | verifyServerCertificate | ds | 24 | 1 |
| 00221561 | hashForClientCertificate | ds | 25 | 1 |
| 00221e1e | asn1:"explicit,tag:1" | ds | 22 | 1 |
| 00221e53 | handlePostHandshakeMessage | ds | 27 | 1 |
| 00222552 | secureRenegotiationSupported | ds | 30 | 1 |
| 00222ebd | asn1:"optional,tag:2" | ds | 23 | 1 |
| 00290069 | ckunpa | ds | 6 | 1 |
| 002903f7 | queuefinalizer during GC | ds | 24 | 1 |
| 00330cff | runtime.dropg | ds | 14 | 1 |
| 00460248 | -----END | ds | 12 | 1 |
| 00460258 | -----BEGIN | ds | 16 | 1 |
| 0029bb9c | 00010203040506070809101111... | ds | 969 | 2 |
| 002e9100 | expand 32-byte k | ds | 20 | 3 |
| 002e91a0 | expand 32-byte k | ds | 20 | 3 |
| 00293a08 | 3552713678800500929355621... | ds | 170 | 4 |
| 0028b3b3 | = is not mcount= minutes nallo... | ds | 225 | 23 |
| 002976f3 | *-+*-+####@@@@!!!!first pat... | ds | 4517 | 95 |

Figure 50 – Kaiji[12] defined strings

At last we will show an issue in Ghidra decompiler view. Once a string is successfully defined by either manually or by one of our scripts, it will be nicely visible in the listing view of Ghidra, giving a great help to reverse engineers when reading the assembly code. However, the decompiler view in Ghidra cannot handle fixed length strings correctly and regardless of the length of the string it will display everything until it finds a null character. Luckily this issue will be solved for the next release of Ghidra (9.2). This issue is illustrated below using the eCh0raix sample[9].

```
                     main.checkReadmeExists                        XREF[2]:     08208c3b(c),
                                                                                main.init.0:08208cda(c)
08208bb0 65 8b 0d        MOV        ECX,dword ptr GS:[0x0]
         00 00 00 00
08208bb7 8b 89 fc        MOV        ECX,dword ptr [ECX + 0xfffffffc]
         ff ff ff
08208bbd 3b 61 08        CMP        ESP,dword ptr [ECX + 0x8]
08208bc0 76 74           JBE        LAB_08208c36
08208bc2 83 ec 1c        SUB        ESP,0x1c
08208bc5 c7 04 24        MOV        dword ptr [ESP]=>local_1c,0x0
         00 00 00 00
08208bcc 8b 44 24 20     MOV        EAX,dword ptr [ESP + param_1]
08208bd0 89 44 24 04     MOV        dword ptr [ESP + local_18],EAX
08208bd4 8b 44 24 24     MOV        EAX,dword ptr [ESP + param_2]
08208bd8 89 44 24 08     MOV        dword ptr [ESP + local_14],EAX
08208bdc 8d 05 0e        LEA        EAX,[s_/README_FOR_DECRYPT.txt_0827de0e]
         de 27 08
08208be2 89 44 24 0c     MOV        dword ptr [ESP + local_10],EAX=>s_/README_FOR_DECRYPT.txt_0827de0e
08208be6 c7 44 24        MOV        dword ptr [ESP + local_c],0x17
         10 17 00
         00 00
08208bee e8 dd c1        CALL       runtime.concatstring2
```

Figure 51 - eCh0raix[9] defined string in listing view

[Decompile: main.checkReadmeExists] - (echoraix_test2)

```
  runtime.concatstring2
            (0,param_1,param_2,

             "/README_FOR_DECRYPT.txt/etc/apache2/mime.types/etc/pki/tls/cacert.pem23283064365386962
             890625<invalid reflect.Value>CPU time limit exceededLogical_Order_ExceptionMB during
             sweep; swept Noncharacter_Code_PointSIGIO: i/o now possibleSIGSYS: bad system
             callVariant Also Negotiatesacquirep: already in goasn1: structure error: bytes.Buffer:
             too largechan receive (nil chan)close of closed channelcommand not implementeddevice
             or resource busyfatal: morestack on g0\nflate: internal error: garbage collection
             scangcDrain phase incorrecthttp2: handler panickedhttp2: invalid trailershttp: request
             too largeinterrupted system callinvalid URI for requestinvalid m->lockedInt = json:
             cannot unmarshal left over markroot jobsmakechan: bad alignmentmalformed HTTP
             responsemissing port in addressmissing protocol schememissing type in runfinqmisuse of
             profBuf.writenanotime returning zeronet/http: abort Handlernetwork not implementedno
             application protocolno space left on devicenon-zero reserved fieldoperation not
             permittedoperation not supportedpanic during preemptoffprocresize: invalid
             argprofiling timer
             expiredreflect.Value.Interfacereflect.Value.NumMethodreflect.methodValueCallruntime:
             internal errorruntime: invalid type  runtime: netpoll failedruntime: s.allocCount=
             s.allocCount > s.nelemsschedule: holding lockssegment length too longskipping Question
             Classspan has no free stacksstack growth after forksyntax error in patterntext/css;
             charset=utf-8text/xml; charset=utf-8time: invalid duration too many pointers
             (>10)truncated tag or lengthunexpected address typeunexpected signal valueunknown
             error code 0x%xunlock of unlocked lockunpacking Question.Nameunpacking
             Question.Typeunsupported certificatevarint integer overflowwork.nwait >
             work.nproc%v.WithDeadline(%s
             [%s])/usr/share/lib/zoneinfo/116415321826934814453125582076609134674072265625Request
             Entity Too Largebad defer entry in panicbad defer size class: i=block index out of
             rangecan\'t scan our own stackconnection reset by peerdouble traceGCSweepStarterror
             decrypting messagefile size li..." /* TRUNCATED STRING LITERAL */
             ,0x17);
```

Figure 52 - eCh0raix[9] defined string in decompile view

# Future work

In this article we proposed solutions for two issues within Go binaries to help reverse engineers when they are using Ghidra to statically analyze malware written in Go. In the first topic we discussed how to recover function names in stripped Go binaries. Then we proposed multiple solutions for defining strings within Ghidra. The scripts that we created and files we used for the examples in this article are publicly available, the links can be found below.

There are even more possibilities to aid Go reverse engineering, the two topics that we discussed here are just the beginning. As a next step we are planning to dive deeper into Go function call conventions and type system.

In Go binaries arguments and return values are passed to functions using the stack, rather than registers. Currently Ghidra has a hard time to correctly detect these. Helping Ghidra to support Go's calling convention will help reverse engineers to understand the purpose of the analyzed functions.

The other interesting topic is types within Go binaries. Just like it was possible to extract function names from the investigated files, Go binaries also store information about the used types. Recovering these types can be a great help during reverse engineering. In the example below we recovered the main.Info structure in an eCh0raix ransomware sample[9]. This structure tells us what information the malware is expecting from the C2 server.

```
                    main.info_struct                         XREF[3]:     main.getInfo:082085fc(*),
                                                                          main.getInfo:08208602(*),
                                                                          08225100(*)
    0824bd20 10 00 00 00     ddw        10h
    0824bd24 0c 00 00 00     ddw        Ch
    0824bd28 15 e7 c0 27     ddw        27C0E715h
    0824bd2c 07              db         7h
    0824bd2d 04              db         4h
    0824bd2e 04              db         4h
    0824bd2f 19              db         19h
    0824bd30 28 c8 20 08     addr       PTR_PTR_type..hash.main.Info_0820c828
    0824bd34 fc a0 2b 08     addr       DAT_082ba0fc
    0824bd38 20 75 00 00     ddw        7520h
    0824bd3c e0 a0 01 00     ddw        1A0E0h
    0824bd40 00 00 00 00     ddw        0h
    0824bd44 60 bd 24 08     addr       PTR_rsapublickey_structfield_0824bd60
    0824bd48 02 00 00 00     ddw        2h
    0824bd4c 02 00 00 00     ddw        2h
    0824bd50 5c 0d 00 00     ddw        D5Ch
    0824bd54 00 00           dw         0h
    0824bd56 00 00           dw         0h
    0824bd58 28 00 00 00     ddw        28h
    0824bd5c 00 00 00 00     ddw        0h
```

Figure 53 - eCh0raix[9] main.info structure

```
                    PTR_rsapublickey_structfield_0824bd60          XREF[1]:     0824bd44(*)
0824bd60 60 aa 22 08     addr        rsapublickey_structfield
0824bd64 a0 a7 23 08     addr        string_type
0824bd68 00 00 00 00     ddw         0h
0824bd6c 18 cf 21 08     addr        readme_structfield
0824bd70 a0 a7 23 08     addr        string_type
0824bd74 10 00 00 00     ddw         10h
```

Figure 54 - eCh0raix[9] main.info fields

```
type main.Info struct{
        RsaPublicKey string
        Readme string
}
```

Figure 55 - eCh0raix[9] main.info structure

As these examples illustrated there are still a lot of interesting areas to discover within Go binaries from reverse engineering point of view. So, stay tuned for our next write-up.

# GitHub repository with scripts and additional materials

- https://github.com/getCUJO/ThreatIntel/tree/master/Scripts/Ghidra
- https://github.com/getCUJO/ThreatIntel/tree/master/Research_materials/Golang_reversing

# Files used during the research

| | File name | SHA-256 |
|---|---|---|
| [1] | hello.c | ab84ee5bcc6507d870fdbb6597bed13f858bbe322dc566522723fd8669a6d073 |
| [2] | hello.go | 2f6f6b83179a239c5ed63cccf5082d0336b9a86ed93dcf0e03634c8e1ba8389b |
| [3] | hello_c | efe3a095cea591fe9f36b6dd8f67bd8e043c92678f479582f61aabf5428e4fc4 |
| [4] | hello_c_strip | 95bca2d8795243af30c3c00922240d85385ee2c6e161d242ec37fa986b423726 |
| [5] | hello_go | 4d18f9824fe6c1ce28f93af6d12bdb290633905a34678009505d216bf744ecb3 |
| [6] | hello_go_strip | 45a338dfddf59b3fd229ddd5822bc44e0d4a036f570b7eaa8a32958222af2be2 |
| [7] | hello_go.exe | 5ab9ab9ca2abf03199516285b4fc81e2884342211bf0b88b7684f87e61538c4d |
| [8] | hello_go_strip.exe | ca487812de31a5b74b3e43f399cb58d6bd6d8c422a4009788f22ed4bd4fd936c |
| [9] | eCh0raix - x86 | 154dea7cace3d58c0ceccb5a3b8d7e0347674a0e76daffa9fa53578c036d9357 |
| [10] | eCh0raix - ARM | 3d7ebe73319a3435293838296fbb86c2e920fd0ccc9169285cc2c4d7fa3f120d |
| [11] | Kaiji - x86_64 | f4a64ab3ffc0b4a94fd07a55565f24915b7a1aaec58454df5e47d8f8a2eec22a |
| [12] | Kaiji - ARM | 3e68118ad46b9eb64063b259fca5f6682c5c2cb18fd9a4e7d97969226b2e6fb4 |

# References and further reading

- https://rednaga.io/2016/09/21/reversing_go_binaries_like_a_pro/
- https://2016.zeronights.ru/wp-content/uploads/2016/12/GO_Zaytsev.pdf
- https://carvesystems.com/news/reverse-engineering-go-binaries-using-radare-2-and-python/
- https://www.pnfsoftware.com/blog/analyzing-golang-executables/
- https://github.com/strazzere/golang_loader_assist/blob/master/Bsides-GO-Forth-And-Reverse.pdf
- https://github.com/radareorg/r2con2020/blob/master/day2/r2_Gophers-AnalysisOfGoBinariesWithRadare2.pdf

# Solutions by other researchers for various tools

IDA Pro

- https://github.com/sibears/IDAGolangHelper
- https://github.com/strazzere/golang_loader_assist

radare2 / Cutter

- https://github.com/f0rki/r2-go-helpers
- https://github.com/JacobPimental/r2-gohelper/blob/master/golang_helper.py
- https://github.com/CarveSystems/gostringsr2

Binary Ninja

- https://github.com/f0rki/bn-goloader

Ghidra

- https://github.com/felberj/gotools
- https://github.com/ghidraninja/ghidra_scripts/blob/master/golang_renamer.py