VB2021 localhost

7 - 8 October, 2021 / vblocalhost.com

REVERSE ENGINEERING GOLANG BINARIES WITH GHIDRA

Dorka Palotay & Albert Zsigovits

CUJO AI, Hungary

dorka.palotay@cujo.com
albert.zsigovits@cujo.com

INTRODUCTION

Go (also called Golang) is an open-source programming language that was designed by *Google* in 2007 and made available to the public in 2012. Over the years it has gained popularity among developers and, as usually happens, it has not only become popular with developers of legitimate software but has also attracted the attention of malware developers. 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* and *MacOS*) makes the attackers' lives much easier, as they don't have to develop and maintain a different codebase for each target environment.

Some special features of the Go programming language make investigating Go binaries difficult for reverse engineers. Reverse engineering tools (e.g. disassemblers) can do a great job in analysing binaries that are written in more popular languages (e.g. C, C++, .NET), but Go presents 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 the one hand this makes malware distribution more difficult for the attackers, but on the other hand some security products also have issues with handling such large files. 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 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.

In the first section of this paper we provide a list of the recently discovered malware families written in Go and briefly introduce a few of them.

In the next sections we will discuss two of the difficulties that reverse engineers face during Go binary analysis and we will show our solutions for those.

Ghidra [1] 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 plug-ins 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.

In our research we tested Go until version 1.15 and used Ghidra versions 9.1 and 9.2.3.

The slides and other materials accompanying this paper are available in our *GitHub* repository [2].

GO MALWARE FAMILIES

In this section, we will briefly look at some of the prominent Go malware families. Table 1 shows a list of the recently discovered malware families written in Go, some of which we introduce in the following sections.

FritzFrog P2P botnet

This piece of malware was discovered by *Guardicore* [27]. FritzFrog has been active since January 2020. With its decentralized nature, there is no single command-and-control server, which makes it very unique as a Peer-2-Peer (P2P) botnet. Its worm executable is completely written in Golang, and its P2P implementation is proprietary.

FritzFrog is also considered a highly advanced piece of malware due to its multi-threaded, modular and fileless nature, which is very rare in a Mirai- and Gafgyt-variant dominated world.

Once a victim is successfully breached, it starts running the UPX-packed malware, which immediately erases itself. The malware process runs under the names ifconfig and nginx, to minimize suspicion.

Its main targets were governmental offices, educational institutions, medical centres, banks and numerous telecom companies as it tried to infiltrate via brute-force through the SSH protocol.

Guardicore has also found FritzFrog to have some similarity to the Rakos botnet, as its function naming is similarly written, and its version numbers are very much alike. They have also developed a client program, which can send commands to the botnet by injecting its own node to participate in the P2P network.

The final goal of the malware is to deploy the malicious payload of a Monero cryptocurrency miner. FritzFrog has been observed with 20 different versions and variants since its inception.

HEH P2P botnet

Another botnet that made headlines as Go malware is the HEH botnet, discovered by *360 Netlab* [16].

HEH's initial vector of attack is the Telnet protocol, on port 23 or 2323, by brute-forcing its way through the login prompt. In the analysed variants, there were 171 usernames and 504 potential passwords stored in variables.

HEH uses a proprietary P2P protocol. HEH also has three clear distinct modules: a propagation module, an HTTP service module and a P2P module.

Family	**Reference**
Kaiji	Intezer - New Chinese Linux malware using Golang [3]
Zebrocy	A Zebrocy Go Downloader [4]
eCh0riax	Reverse Engineering Go Binaries with Ghidra - CUJO AI [5]
LiquorBot	Intezer on Twitter [6]
WellMess	Intezer on Twitter [7]
Smaug ransomware	Anomali Threat Research Releases First Public Analysis of Smaug Ransomware as a Service [8]
FritzFrog	FritzFrog: A New Generation Of Peer-To-Peer Botnets - Guardicore [9]
Godlike12	Holy water: ongoing targeted water-holing attack in Asia [10]
IRCFlu	muesli/ircflu [11]
IPStorm	The InterPlanetary Storm: New Malware in Wild Using InterPlanetary File System's (IPFS) p2p network [12]
Nephilim	Vitali Kremez on Twitter [13]
EKANS	EKANS Ransomware Targeting OT ICS Systems | FortiGuard Labs [14]
RobinHood	Vitali Kremez on Twitter [15]
HEH	https://blog.netlab.360.com/heh-an-iot-p2p-botnet/ [16]
Go Loader	TA416 Goes to Ground and Returns with a Golang PlugX Malware Loader | Proofpoint US [17]
GOSH	Intezer on Twitter [18]
Glupteba.Go	Glupteba malware hides in plain sight [19]
New RAT	There's a New a Golang-written RAT in Town [20]
BlackRota	https://blog.netlab.360.com/blackrota-an-obfuscated-backdoor-written-in-go-en/ [21]
Clipboard.Stealer	https://analyze.intezer.com/files/bd978ba0d723aea3106c6abc58cf71df5abe4d674d0d1fc38b37d4926d740738 [22]
CryptoStealer.Go	Analyzing a new stealer written in Golang - Malwarebytes Labs [23]
Sysrv-hello	Sysrv Botnet Expands and Gains Persistence | Official Juniper Networks Blogs [24]
Epsilon Red	A new ransomware enters the fray: Epsilon Red [25]
aicm	Intezer on Twitter [26]

Table 1: Go malware families.

According to *360 Netlab*, this botnet is not yet mature, as some of the more essential functions, like the attack module, have not been implemented yet, and there are flaws in the implementation of the P2P module too.

HEH starts with a shell script, which pulls down the malicious binaries to different types of architectures and, surprisingly, executes all of them on the target. The malicious binary then kills a series of service processes based on listening port numbers.

HEH also starts an HTTP server on TCP port 80, and an initial dummy content will be placed onto the server, which gets overwritten by the P2P module once data is transferred from another node.

Currently, the botnet can execute shell commands, update the Peer List and exchange data, but as the attack module is not yet finished, analysts expect that there will be several iterations of HEH versions.

Sysrv botnet

In April, researchers at *Juniper Threat Labs* [24] reported that they had discovered a surge of activity from the botnet Sysrv. Traces of Sysrv botnet activity date back to December 2020.

Previously, Sysrv had separate worm and miner executables, but more recently Sysrv combines the two in one malicious binary. We also know that Sysrv once used two mining pools but now focuses only on the miner pool, 'nanopool'.

Some developments have been observed in the loader script itself, which loads the malicious binary: the script now involves a procedure for adding an SSH key to the authorized_keys file on the target system to achieve persistence. Also, there is a *Linux* version of loader script, which is called ldr.sh, and a *Windows* one, called ldr.ps1.

The first variants of the malicious Sysrv payload exploited several different vulnerabilities, including the following:

- CVE-2020-16846 – Saltstack RCE
- CVE-2019-10758 – Mongo Express RCE
- CVE-2018-7600 – Drupal Ajax RCE
- CVE-2017-11610 – XML-RPC
- XXL-JOB Unauth RCE (without CVE)
- ThinkPHP RCE

Later versions of Sysrv started to include many other application-specific exploits, and we expect that they will keep incorporating more. These application-specific exploits are used to download and execute the first-stage loader script, ldr.sh or ldr.ps1.

Sysrv's goal is to spread further and deploy a Monero cryptocurrency miner on the infected systems.

Epsilon Red ransomware

Researchers at *Sophos* [25] discovered a Golang-based ransomware that was attacking a US-based business. The loader for the ransomware payload is a PowerShell script.

Analysts conclude that this new ransomware variant is quite a simple program, as it has no networking capabilities, and the encryption process is simple.

Epsilon Red will encrypt everything in its way, including all system files, possibly rendering the entire operating system unusable. Once the encryption is done, the ransomware appends the extension '.epsilonred' to all encrypted files. The ransomware spawns a new child process for every folder it encrypts, which results in an unnecessarily long list of running ransomware processes.

From a binary perspective, the malicious sample was compiled with MinGW, and packed with a modified version of the UPX packer. We have also observed that the sample contains code from the open-source project Godirwalk: this tool will scan the entire system storage and compile a list of directory paths, which is then used for the encryption.

Analysts have found that the ransomware note dropped by Epsilon has some similarity to the one left behind by the REvil ransomware.

We have made the following observations and predictions during the analysis of the aforementioned botnets:

- Ransomware that is written in Golang will become more common
- P2P botnets are still popular and introduce new concepts and modules
- Botnets still trying to deploy cryptocurrency miners as a final step

Due to these, we have decided to dive deep into the Go language to understand it better and to enhance our ability to tackle Go malware. In the next two sections we introduce two features of Go, the difficulties reverse engineers face during Go malware analysis thanks to those, and our solutions.

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, developers usually compile the files without this information, creating so-called stripped binaries. For malware authors another advantage of stripping binaries is that it makes reverse engineering more difficult. In this case analysts cannot rely on the function names to help them find 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 World' examples written in C(1) and Go(2) for comparison and compiled them to stripped binaries. Note the size difference between the two executables.

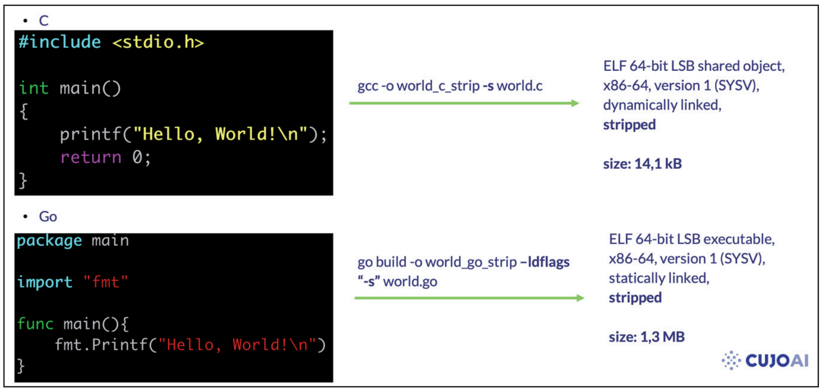
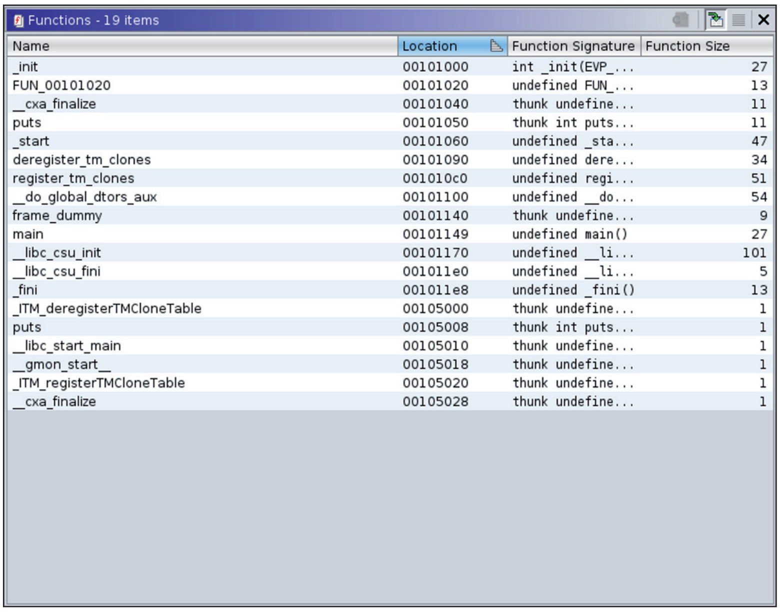


Figure 1: Hello World examples written in C[1] and Go[2].

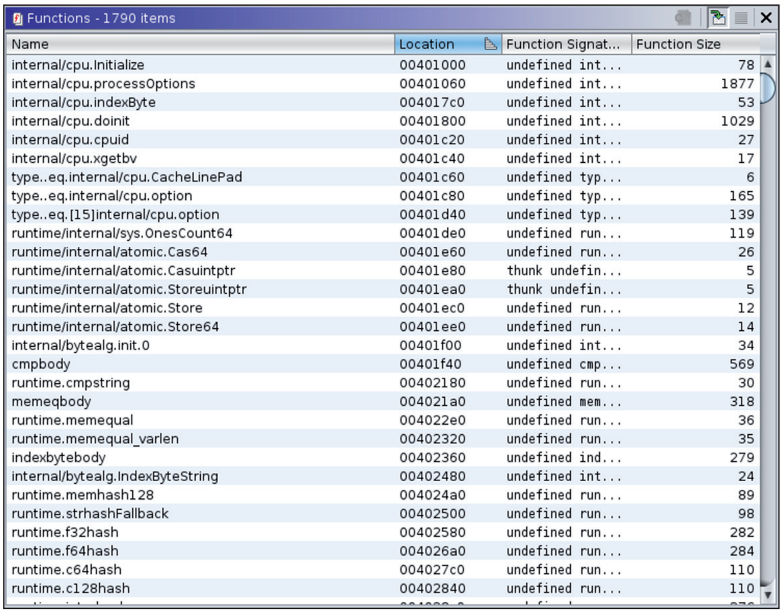
Ghidra's function window lists all the defined functions within the binaries. In the non-stripped versions, function names are nicely visible and provide a great help for reverse engineers.



Functions - 19 items

Name	Location	Function Signature	Function Size
_init	00101000	int _init(EVP_...	27
FUN_00101020	00101020	undefined FUN_...	13
__cxa_finalize	00101040	thunk undefine...	11
puts	00101050	thunk int puts...	11
_start	00101060	undefined _sta...	47
deregister_tm_clones	00101090	undefined dere...	34
register_tm_clones	001010c0	undefined regi...	51
__do_global_dtors_aux	00101100	undefined __do...	54
frame_dummy	00101140	thunk undefine...	9
main	00101149	undefined main()	27
__libc_csu_init	00101170	undefined __li...	101
__libc_csu_fini	001011e0	undefined __li...	5
_fini	001011e8	undefined _fini()	13
_ITM_deregisterTMCloneTable	00105000	thunk undefine...	1
puts	00105008	thunk int puts...	1
__libc_start_main	00105010	thunk undefine...	1
__gmon_start__	00105018	thunk undefine...	1
_ITM_registerTMCloneTable	00105020	thunk undefine...	1
__cxa_finalize	00105028	thunk undefine...	1

Figure 2: world_c[3] function list.



Functions - 1790 items

Name	Location	Function Signat...	Function Size
internal/cpu.Initialize	00401000	undefined int...	78
internal/cpu.processOptions	00401060	undefined int...	1877
internal/cpu.indexByte	004017c0	undefined int...	53
internal/cpu.doinit	00401800	undefined int...	1029
internal/cpu.cpuid	00401c20	undefined int...	27
internal/cpu.xgetbv	00401c40	undefined int...	17
type..eq.internal/cpu.CacheLinePad	00401c60	undefined typ...	6
type..eq.internal/cpu.option	00401c80	undefined typ...	165
type..eq.[15]internal/cpu.option	00401d40	undefined typ...	139
runtime/internal/sys.OnesCount64	00401de0	undefined run...	119
runtime/internal/atomic.Cas64	00401e60	undefined run...	26
runtime/internal/atomic.Casuintptr	00401e80	thunk undefin...	5
runtime/internal/atomic.Storeuintptr	00401ea0	thunk undefin...	5
runtime/internal/atomic.Store	00401ec0	undefined run...	12
runtime/internal/atomic.Store64	00401ee0	undefined run...	14
internal/bytealg.init.0	00401f00	undefined int...	34
cmpbody	00401f40	undefined cmp...	569
runtime.cmpstring	00402180	undefined run...	30
memeqbody	004021a0	undefined mem...	318
runtime.memequal	004022e0	undefined run...	36
runtime.memequal_varlen	00402320	undefined run...	35
indexbytebody	00402360	undefined ind...	279
internal/bytealg.IndexByteString	00402480	undefined int...	24
runtime.memhash128	004024a0	undefined run...	89
runtime.strhashFallback	00402500	undefined run...	98
runtime.f32hash	00402580	undefined run...	282
runtime.f64hash	004026a0	undefined run...	284
runtime.c64hash	004027c0	undefined run...	110
runtime.c128hash	00402840	undefined run...	110

Figure 3: world_go[5] function list.

For stripped binaries the function lists look the following:

Functions - 19 items

Name	Location	Function Signature	Function Size
_DT_INIT	00101000	undefined _DT_...	27
FUN_00101020	00101020	undefined FUN_...	13
__cxa_finalize	00101040	thunk undefine...	11
puts	00101050	thunk int puts...	11
entry	00101060	undefined entry()	47
FUN_00101090	00101090	undefined FUN_...	34
FUN_001010c0	001010c0	undefined FUN_...	51
_FINI_0	00101100	undefined _FIN...	54
_INIT_0	00101140	thunk undefine...	9
FUN_00101149	00101149	undefined FUN_...	27
FUN_00101170	00101170	undefined FUN_...	101
FUN_001011e0	001011e0	undefined FUN_...	5
_DT_FINI	001011e8	undefined _DT_...	13
_ITM_deregisterTMCloneTable	00105000	thunk undefine...	1
puts	00105008	thunk int puts...	1
__libc_start_main	00105010	thunk undefine...	1
__gmon_start__	00105018	thunk undefine...	1
_ITM_registerTMCloneTable	00105020	thunk undefine...	1
__cxa_finalize	00105028	thunk undefine...	1

Figure 4: world_c_strip[(4)] function list.

Functions - 1138 items

Name	Location	Function Signat...	Function Size
FUN_00401000	00401000	undefined FUN...	78
FUN_00401060	00401060	undefined FUN...	1877
FUN_004017c0	004017c0	undefined FUN...	53
FUN_00401800	00401800	undefined FUN...	1029
FUN_00401c20	00401c20	undefined FUN...	27
FUN_00401c40	00401c40	undefined FUN...	17
FUN_00401c80	00401c80	undefined FUN...	165
FUN_00401de0	00401de0	undefined FUN...	119
FUN_00401e60	00401e60	undefined FUN...	26
thunk_FUN_00401e60	00401e80	thunk undefin...	5
thunk_FUN_00401ee0	00401ea0	thunk undefin...	5
FUN_00401ec0	00401ec0	undefined FUN...	12
FUN_00401ee0	00401ee0	undefined FUN...	14
FUN_00402180	00402180	undefined FUN...	599
FUN_004022e0	004022e0	undefined FUN...	354
FUN_00402480	00402480	undefined FUN...	303
FUN_00402580	00402580	undefined FUN...	282
FUN_004026a0	004026a0	undefined FUN...	284
FUN_004027c0	004027c0	undefined FUN...	110
FUN_00402840	00402840	undefined FUN...	110
FUN_004028c0	004028c0	undefined FUN...	376
FUN_00402a40	00402a40	undefined FUN...	368
FUN_00402bc0	00402bc0	undefined FUN...	1640
FUN_004035a0	004035a0	undefined FUN...	272
FUN_004036c0	004036c0	undefined FUN...	280
FUN_004037e0	004037e0	undefined FUN...	198
FUN_004038c0	004038c0	undefined FUN...	119
FUN_00403940	00403940	undefined FUN...	72
FUN_004039a0	004039a0	undefined FUN...	338

Figure 5: world_go_strip[(6)] function list.

These examples show nicely 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: As a result of stripping, not only did the function names disappear, but instead of 1,790 defined functions only 1,138 were recognized by Ghidra.

We were interested to find out whether there is a way to recover the function names within stripped binaries. First, using a simple string search we can check 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'.

```
> strings world_c | grep -o ".\{0,10\}main.\{0,10\}"
ibc_start_main
ibc_start_main@@GLIBC_2.
main
```

Figure 6: world_c[(3)] strings – 'main' was found.

```
> strings world_c_strip | grep -o ".\{0,10\}main.\{0,10\}"
ibc_start_main
```

Figure 7: world_c_strip[4] strings – 'main' was not found.

```
> strings world_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 8: world_go[5] strings – 'main.main' was found.

```
> strings world_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 9: world_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 function name recovery might be possible in stripped Go binaries.

Loading the binary[6] to Ghidra and searching for the 'main.main' string will show the exact location. As can be seen in Figure 10, the function name string is located within the .gopclntab section.

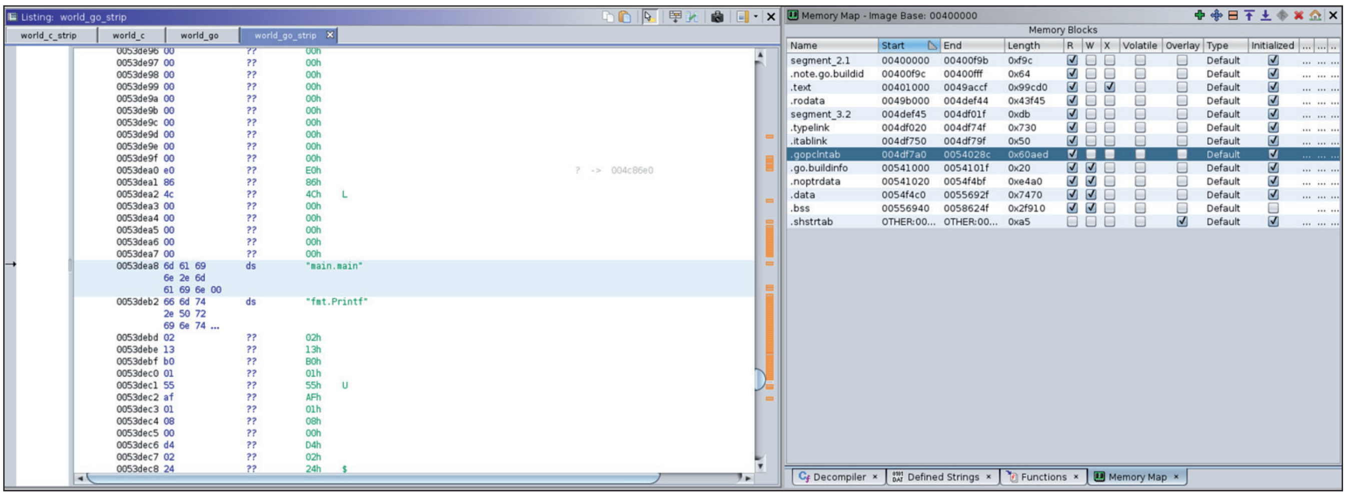


Figure 10: world_go_strip[6] main.main string in Ghidra.

The pclntab structure has been available since Go version 1.2 and is nicely documented [28]. 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.

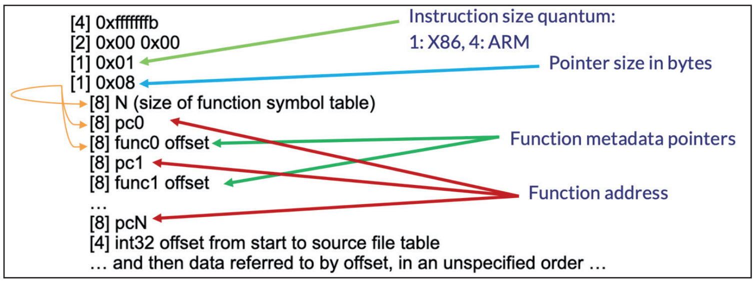


Figure 11: The pclntab structure.

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

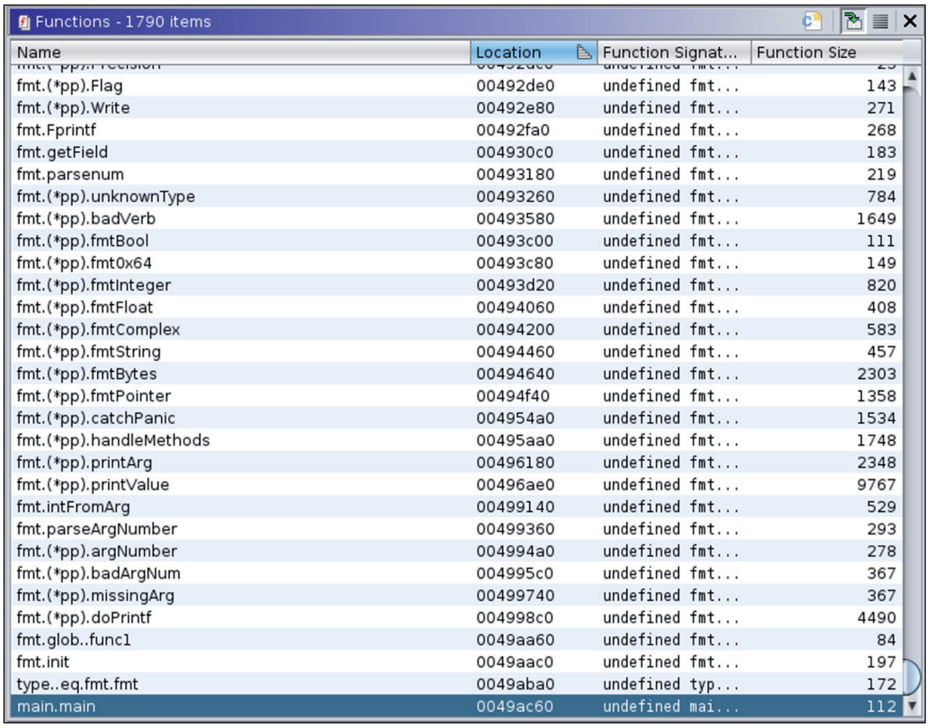
```
struct        Func
{
        uintptr        entry;  // start pc
        int32 name;          // name (offset to C string)
        int32 args;          // size of arguments passed to function
        int32 frame;         // size of function frame, including saved caller PC
        int32        pcsp;              // 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
};
```

Figure 12: Function metadata table.

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 will the function names be restored, but the previously unrecognized functions will be defined as well.



Functions - 1790 items

Name	Location	Function Signat...	Function Size
fmt.(*pp).Flag	00492de0	undefined fmt...	143
fmt.(*pp).Write	00492e80	undefined fmt...	271
fmt.Fprintf	00492fa0	undefined fmt...	268
fmt.getField	004930c0	undefined fmt...	183
fmt.parsenum	00493180	undefined fmt...	219
fmt.(*pp).unknownType	00493260	undefined fmt...	784
fmt.(*pp).badVerb	00493580	undefined fmt...	1649
fmt.(*pp).fmtBool	00493c00	undefined fmt...	111
fmt.(*pp).fmt0x64	00493c80	undefined fmt...	149
fmt.(*pp).fmtInteger	00493d20	undefined fmt...	820
fmt.(*pp).fmtFloat	00494060	undefined fmt...	408
fmt.(*pp).fmtComplex	00494200	undefined fmt...	583
fmt.(*pp).fmtString	00494460	undefined fmt...	457
fmt.(*pp).fmtBytes	00494640	undefined fmt...	2303
fmt.(*pp).fmtPointer	00494f40	undefined fmt...	1358
fmt.(*pp).catchPanic	004954a0	undefined fmt...	1534
fmt.(*pp).handleMethods	00495aa0	undefined fmt...	1748
fmt.(*pp).printArg	00496180	undefined fmt...	2348
fmt.(*pp).printValue	00496ae0	undefined fmt...	9767
fmt.intFromArg	00499140	undefined fmt...	529
fmt.parseArgNumber	00499360	undefined fmt...	293
fmt.(*pp).argNumber	004994a0	undefined fmt...	278
fmt.(*pp).badArgNum	004995c0	undefined fmt...	367
fmt.(*pp).missingArg	00499740	undefined fmt...	367
fmt.(*pp).doPrintf	004998c0	undefined fmt...	4490
fmt.glob..func1	0049aa60	undefined fmt...	84
fmt.init	0049aac0	undefined fmt...	197
type..eq.fmt.fmt	0049aba0	undefined typ...	172
main.main	0049ac60	undefined mai...	112

Figure 13: world_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)]:

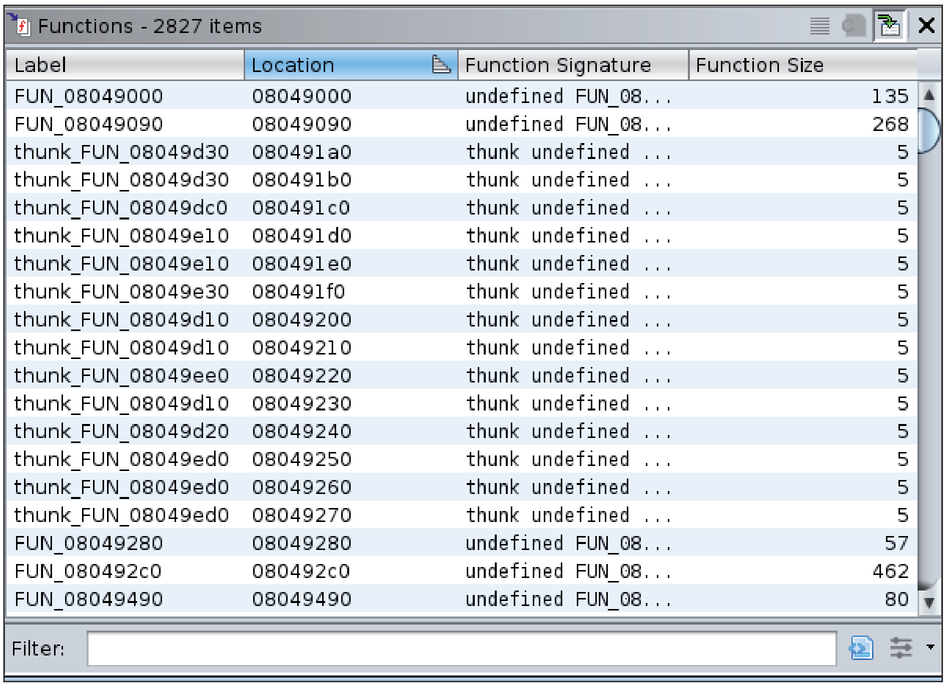


Figure 14: eCh0raix[(9)] function list.

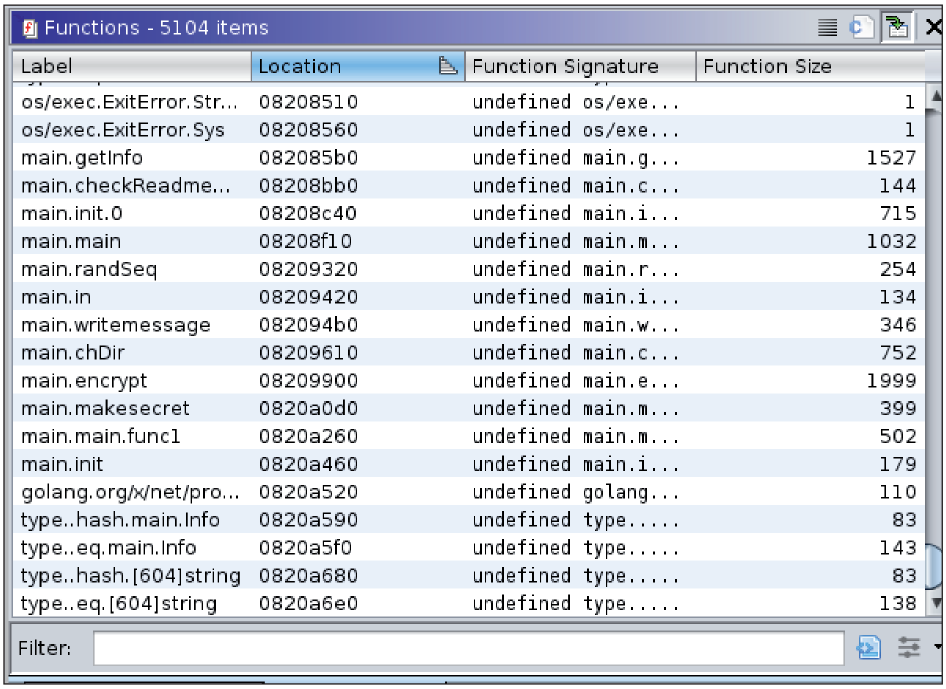


Figure 15: eCh0raix[(9)] function list after executing go_func.py.

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 can analysts 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 to search explicitly for the fields of this structure (e.g. magic value, possible field values). For *MacOS* the _gopclntab section is available, and similarly .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 it isn't, then before renaming a function it tries to define a string data type at the given address.

In the example shown in Figures 16 and 17 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 16: 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 17: eCh0raix[(9)] log.New function couldn't be renamed.

Figure 18 shows the lines in our script that are responsible for solving this challenge.

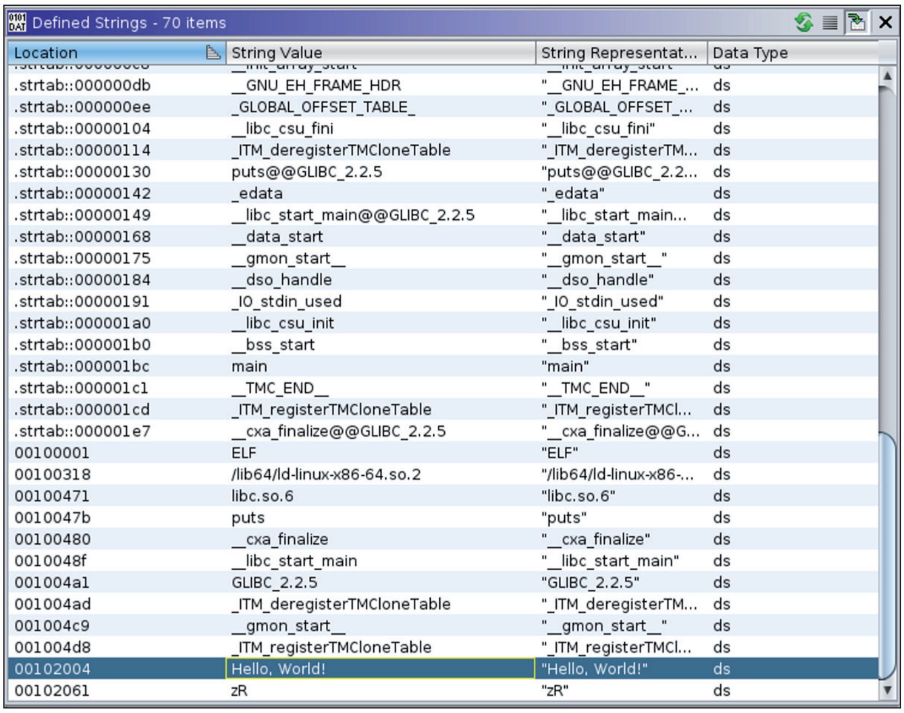
```
func_name = getDataAt(name_address)

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

Figure 18: go_func.py.

UNRECOGNIZED STRINGS

The second issue that our scripts help to solve is related to strings within Go binaries. Let's go back to the 'Hello World' examples and take a look at the defined strings within Ghidra.

In the C binary[(3)] 70 strings are defined, among which 'Hello, World!' can be found. Meanwhile, the Go binary[(5)] includes 6,544 strings but searching for 'Hello' gives no result. Having such a high number of strings already makes it hard 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.



Location	String Value	String Representat...	Data Type
.strtab::000000db	__GNU_EH_FRAME_HDR	"__GNU_EH_FRAME_...	ds
.strtab::000000ee	_GLOBAL_OFFSET_TABLE_	"_GLOBAL_OFFSET_...	ds
.strtab::00000104	__libc_csu_fini	"__libc_csu_fini"	ds
.strtab::00000114	_ITM_deregisterTMCloneTable	"_ITM_deregisterTM...	ds
.strtab::00000130	puts@@GLIBC_2.2.5	"puts@@GLIBC_2.2...	ds
.strtab::00000142	_edata	"_edata"	ds
.strtab::00000149	__libc_start_main@@GLIBC_2.2.5	"__libc_start_main...	ds
.strtab::00000168	__data_start	"__data_start"	ds
.strtab::00000175	__gmon_start__	"__gmon_start__"	ds
.strtab::00000184	__dso_handle	"__dso_handle"	ds
.strtab::00000191	_IO_stdin_used	"_IO_stdin_used"	ds
.strtab::000001a0	__libc_csu_init	"__libc_csu_init"	ds
.strtab::000001b0	__bss_start	"__bss_start"	ds
.strtab::000001bc	main	"main"	ds
.strtab::000001c1	__TMC_END__	"__TMC_END__"	ds
.strtab::000001cd	_ITM_registerTMCloneTable	"_ITM_registerTMCl...	ds
.strtab::000001e7	__cxa_finalize@@GLIBC_2.2.5	"__cxa_finalize@@G...	ds
00100001	ELF	"ELF"	ds
00100318	/lib64/ld-linux-x86-64.so.2	"/lib64/ld-linux-x86-...	ds
00100471	libc.so.6	"libc.so.6"	ds
0010047b	puts	"puts"	ds
00100480	__cxa_finalize	"__cxa_finalize"	ds
0010048f	__libc_start_main	"__libc_start_main"	ds
001004a1	GLIBC_2.2.5	"GLIBC_2.2.5"	ds
001004ad	_ITM_deregisterTMCloneTable	"_ITM_deregisterTM...	ds
001004c9	__gmon_start__	"__gmon_start__"	ds
001004d8	_ITM_registerTMCloneTable	"_ITM_registerTMCl...	ds
00102004	Hello, World!	"Hello, World!"	ds
00102061	zR	"zR"	ds

Figure 19: world_c[(3)] defined strings with 'Hello, World!'.

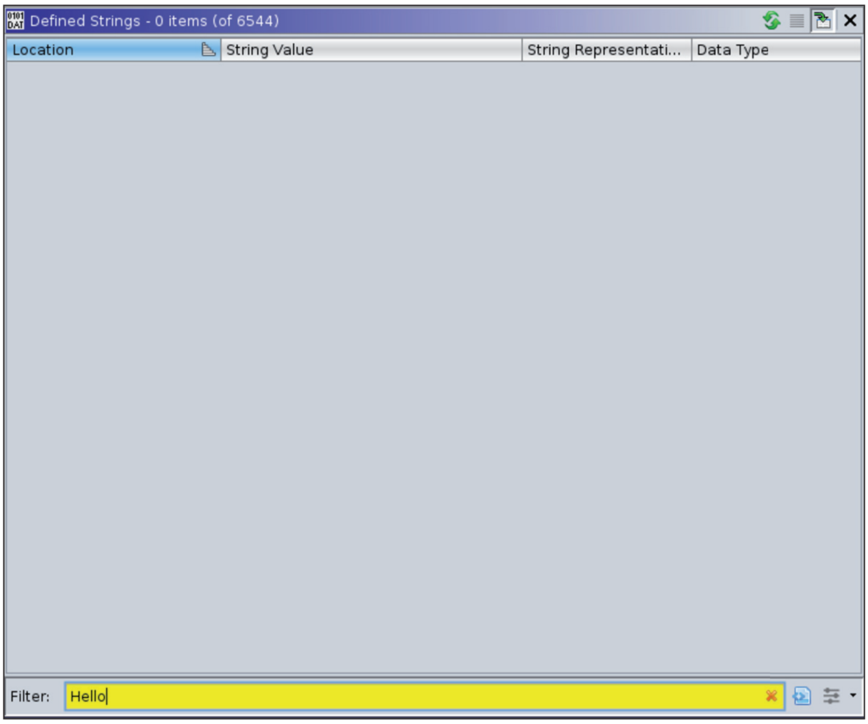


Figure 20: world_go[(5)] defined strings without 'Hello'.

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 sequences 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
}
```

Figure 21: A Go string.

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 ‘Hello’ using strings and grep gives the expected result in C, in the case of Go a huge string blob is returned containing somewhere ‘Hello’.

```
> strings world_c | grep Hello
Hello, World!
```

Figure 22: world_c(3) string search for ‘Hello’.

```
> strings world_go | grep Hello
entersyscallgcBitsArenasgcpacertracehost is downillegal seekinvalid slotlfstack.pushmadvdontneedmheapSpecialmspanSpecialnot pollableraceF
iniLockreleasep: m=runtime: gp=runtime: sp=short bufferspanSetSpinesweepWaiterstraceStringsuname failedwirep: p->m= != sweepgen  MB) work
ers= called from  failed with  flushedWork  heap_marked= idlethreads= is nil, not  nStackRoots= s.spanclass= span.base()= syscalltick= wo
rk.nproc=  work.nwait= , gp->status=, not pointer-byte block (3814697265625GC sweep waitGunjala_GondiHello, World!Masaram_GondiMende_Kika
kuiOld_HungarianSIGKILL: killSIGQUIT: quitbad flushGen bad map statedebugCall2048exchange fullfatal error: level 3 resetload64 failedmin
too largenil stackbaseout of memorysrmount errortimer expiredtraceStackTabtriggerRatio=value method xadd64 failedxchg64 failed}
```

Figure 23: world_go_println(13) string search for ‘Hello’.

Since the definition of strings is different, and as a result referencing them within the assembly code is also different from the usual C-like solutions, Ghidra has a hard time defining 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 the same architecture, multiple solutions are possible. Our team created two scripts to help Ghidra identify strings.

Dynamically allocated string structures

In the first case string structures are created at 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 across architectures. In the next few paragraphs we will go through a couple of use cases and show the instruction sequences that our script (find_dynamic_strings.py) [29] is looking for.

x86

First let’s start with the ‘Hello World’ example(5).

```
                    main.main                              XREF[4]:    Entry Point(*),
                                                                       runtime.main:00434ac7(c),
                                                                       0049acce(c), 004c5cb8(*)
0049ac60 64 48 8b      MOV     RCX,qword ptr FS:[0xfffffff8]
         0c 25 f8
         ff ff ff
0049ac69 48 3b 61 10   CMP     RSP,qword ptr [RCX + 0x10]
0049ac6d 76 5a         JBE     LAB_0049acc9
0049ac6f 48 83 ec 58   SUB     RSP,0x58
0049ac73 48 89 6c      MOV     qword ptr [RSP + local_8],RBP
         24 50
0049ac78 48 8d 6c      LEA     RBP=>local_8,[RSP + 0x50]
         24 50
0049ac7d 48 8b 05      MOV     RAX,qword ptr [os.Stdout]                    = ??
         0c bd 0b 00
0049ac84 48 8d 0d      LEA     RCX,[go.itab.*os.File,io.Writer]             =
         95 26 04 00
0049ac8b 48 89 0c 24   MOV     qword ptr [RSP]=>local_58,RCX=>go.itab.*os.File,i... =
0049ac8f 48 89 44      MOV     qword ptr [RSP + local_50],RAX
         24 08
0049ac94 48 8d 05      LEA     RAX,[DAT_004bf224]                           String location
         89 45 02 00
0049ac9b 48 89 44      MOV     qword ptr [RSP + local_48],RAX=>DAT_004bf224 = 48h
         24 10
0049aca0 48 c7 44      MOV     qword ptr [RSP + local_40],0xe
         24 18 0e
         00 00 00
0049aca9 48 c7 44      MOV     qword ptr [RSP + local_38],0x0               Length
         24 20 00
         00 00 00
0049acb2 0f 57 c0      XORPS   XMM0,XMM0
0049acb5 0f 11 44      MOVUPS  xmmword ptr [RSP + local_30[0]],XMM0
         24 28
0049acba e8 e1 82      CALL    fmt.Fprintf                                  undefined fmt.Fprintf(
         ff ff
```

Figure 24: world_go(5) dynamic allocation of string structure.

```
                    DAT_004bf224                              XREF[2]:     main.main:0049ac94(*),
                                                                           main.main:0049ac9b(*)
004bf224 48            ??          48h     H
004bf225 65            ??          65h     e
004bf226 6c            ??          6Ch     l
004bf227 6c            ??          6Ch     l
004bf228 6f            ??          6Fh     o
004bf229 2c            ??          2Ch     ,
004bf22a 20            ??          20h
004bf22b 57            ??          57h     W
004bf22c 6f            ??          6Fh     o
004bf22d 72            ??          72h     r
004bf22e 6c            ??          6Ch     l
004bf22f 64            ??          64h     d
004bf230 21            ??          21h     !
004bf231 0a            ??          0Ah
```

Figure 25: world_go(5) undefined 'Hello, World!' string.

Figure 26 shows how the code looks after executing the script.

```
                   main.main                                  XREF[4]:     Entry Point(*),
                                                                           runtime.main:00434ac7(c),
                                                                           0049acce(c), 004c5cb8(*)
0049ac60 64 48 8b        MOV       RCX,qword ptr FS:[0xfffffff8]
         0c 25 f8
         ff ff ff
0049ac69 48 3b 61 10     CMP       RSP,qword ptr [RCX + 0x10]
0049ac6d 76 5a           JBE       LAB_0049acc9
0049ac6f 48 83 ec 58     SUB       RSP,0x58
0049ac73 48 89 6c        MOV       qword ptr [RSP + local_8],RBP
         24 50
0049ac78 48 8d 6c        LEA       RBP=>local_8,[RSP + 0x50]
         24 50
0049ac7d 48 8b 05        MOV       RAX,qword ptr [os.Stdout]                        = ??
         0c bd 0b 00
0049ac84 48 8d 0d        LEA       RCX,[go.itab.*os.File,io.Writer]                 =
         95 26 04 00
0049ac8b 48 89 0c 24     MOV       qword ptr [RSP]=>local_58,RCX=>go.itab.*os.File,i... =
0049ac8f 48 89 44        MOV       qword ptr [RSP + local_50],RAX
         24 08
0049ac94 48 8d 05        LEA       RAX,[s_Hello,_World!_004bf224]                   = "Hello, World!\n"
         89 45 02 00
0049ac9b 48 89 44        MOV       qword ptr [RSP + local_48],RAX=>s_Hello,_World!_0... = "Hello, World!\n"
         24 10
0049aca0 48 c7 44        MOV       qword ptr [RSP + local_40],0xe
         24 18 0e
         00 00 00
0049aca9 48 c7 44        MOV       qword ptr [RSP + local_38],0x0
         24 20 00
         00 00 00
0049acb2 0f 57 c0        XORPS     XMM0,XMM0
0049acb5 0f 11 44        MOVUPS    xmmword ptr [RSP + local_30[0]],XMM0
         24 28
0049acba e8 e1 82        CALL      fmt.Fprintf                                      undefined fmt.Fprintf
         ff ff
```

Figure 26: world_go(5) dynamic allocation of string structure after executing find_dynamic_strings.py.

The string is defined as shown in Figure 27.

```
                    s_Hello,_World!_004bf224                  XREF[2]:     main.main:0049ac94(*),
                                                                           main.main:0049ac9b(*)
004bf224 48 65 6c        ds          "Hello, World!\n"
         6c 6f 2c
         20 57 6f ...
```

Figure 27: world_go(5) defined 'Hello, World!' string.

And 'Hello' can be found in the defined strings view in Ghidra, as shown in Figure 28.

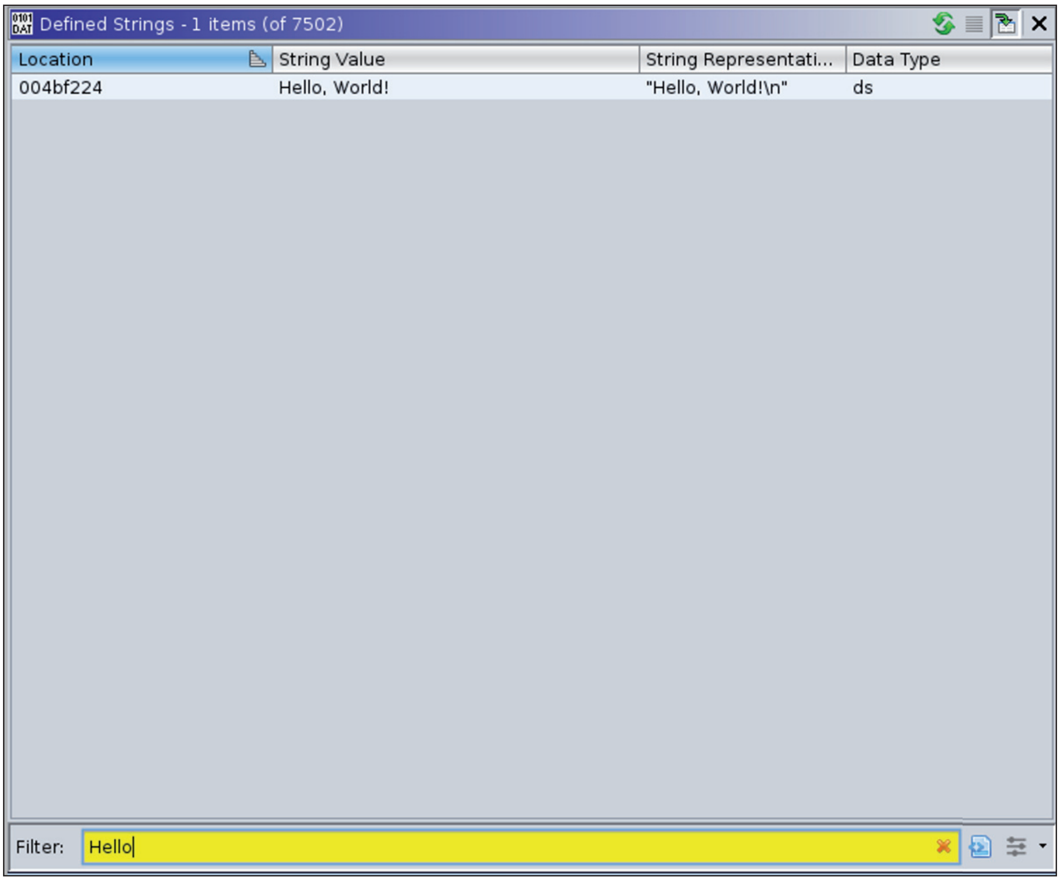


Figure 28: world_go[5] defined strings with 'Hello'.

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
```

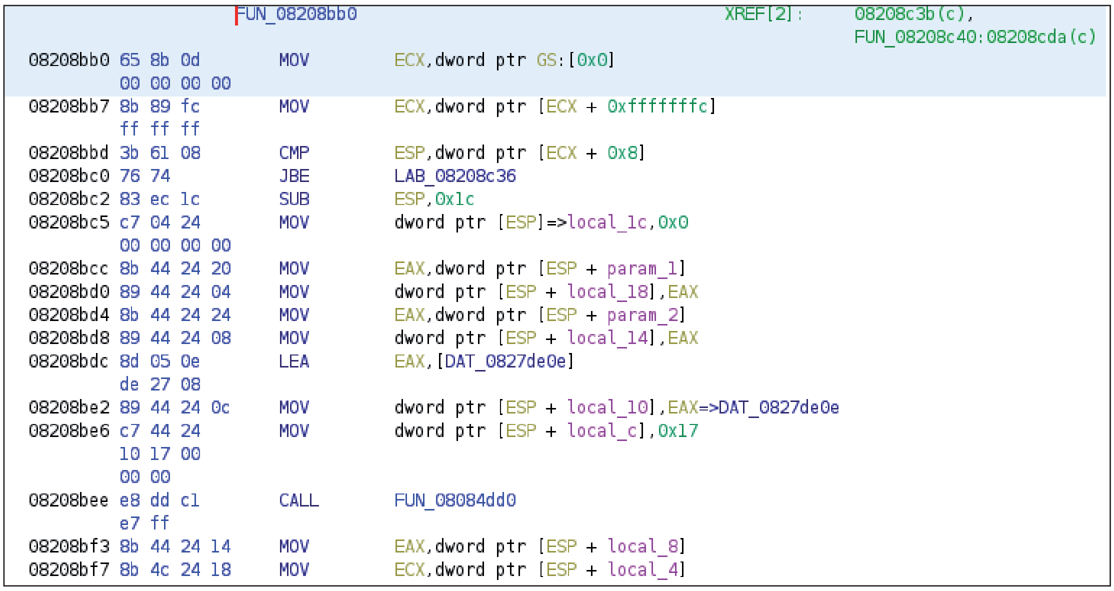


Figure 29: eCh0raix[9] dynamic allocation of string structure.

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

```
0049ac94 48 8d 05      LEA       RAX,[DAT_004bf224]                          = 48h     H
         89 45 02 00
0049ac9b 48 89 44      MOV       qword ptr [RSP + local_48],RAX=>DAT_004bf224  = 48h     H
         24 10
0049aca0 48 c7 44      MOV       qword ptr [RSP + local_40],0xe
         24 18 0e
         00 00 00
```

Figure 30: world_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 31: 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 32: 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 33: eCh0raix[10] undefined string.

Figure 34 shows how the code looks after executing the script.

```
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 34: 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 35: 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 36: 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, ..]
```

Figure 37: The instruction sequence the script looks for.

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
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=>DAT_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
```

String location

Length

String location

Length

Figure 38: Kaiji[12] dynamic allocation of string structure.

Figure 39 shows how the code looks after executing the script.

```
                     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 39: 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 40: 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, ..]
```

Figure 41: The instruction sequence the script looks for.

As the above examples show, after executing the script, dynamically allocated string structures can be recovered. This gives a great help to reverse engineers trying 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 shown in Figure 42, 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.

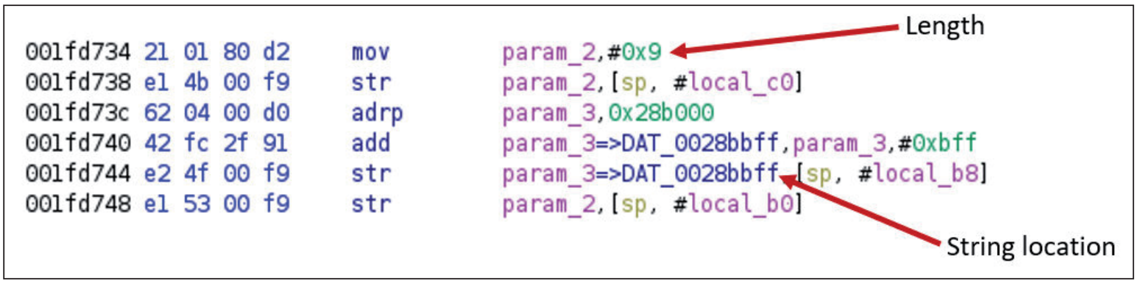


Figure 42: Kaiji(12) dynamic allocation of string structure in an unusual way.

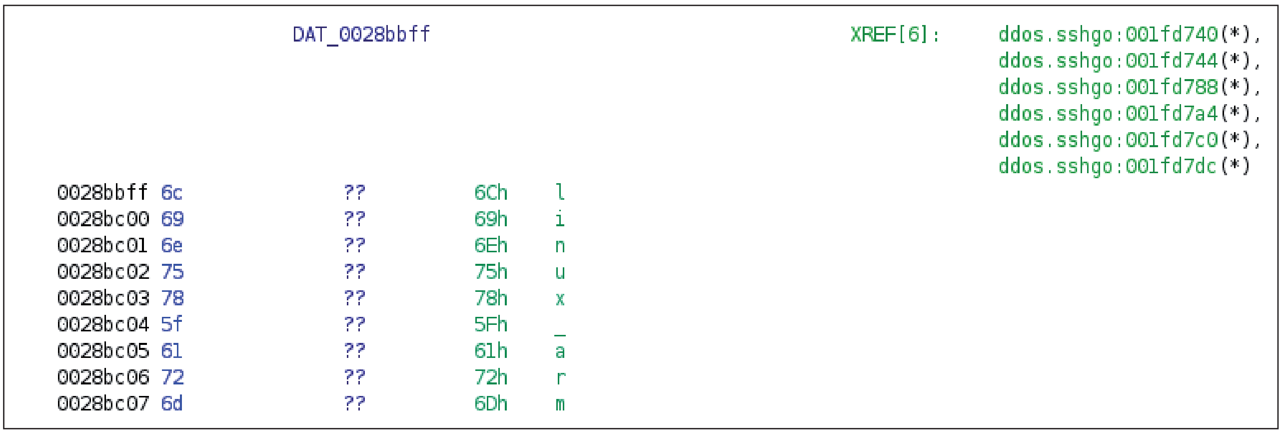


Figure 43: Kaiji(12) undefined string.

Statically allocated string structures

In the next case our script (find_static_strings.py) [30] 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).

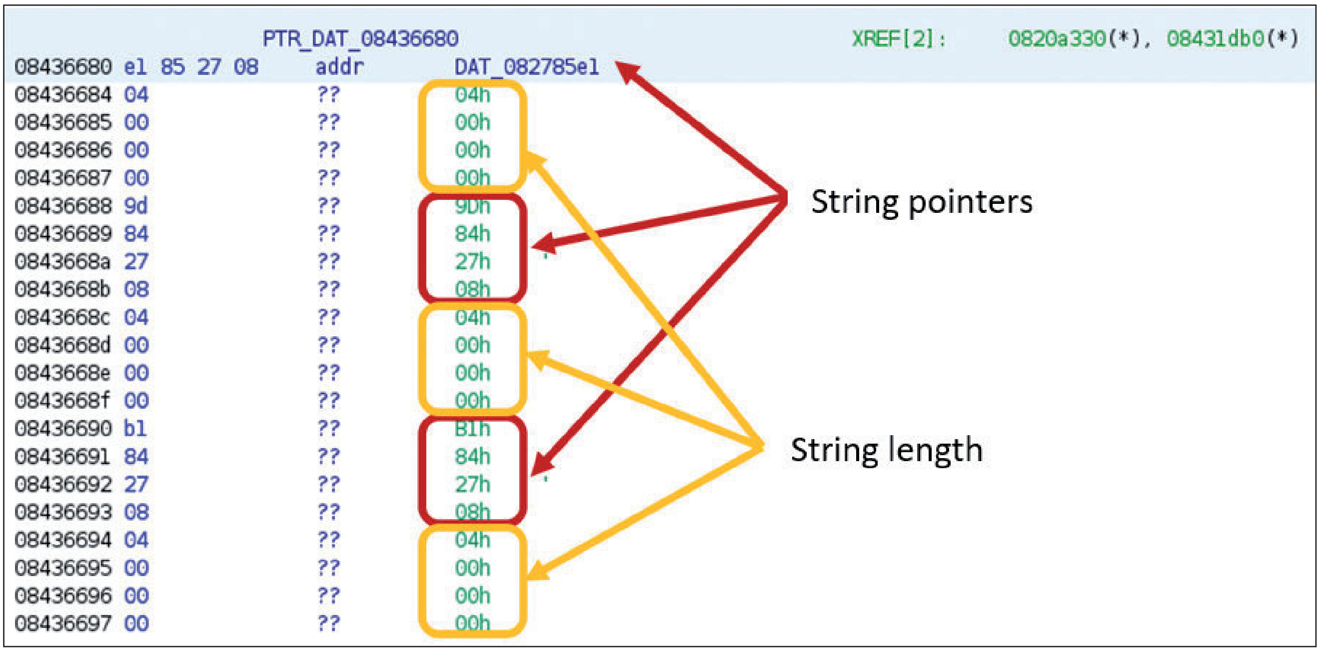


Figure 44: eCh0raix(9) static allocation of string structures.

In Figure 44 string pointers are followed by string length values, however Ghidra couldn't recognize the addresses or the integer data types, with the exception of 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 45: 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
082785f5 2e          ??         2Eh    .
082785f6 64          ??         64h    d
082785f7 62          ??         62h    b
082785f8 78          ??         78h    x
```

Figure 46: 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_.1st_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 47: 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 48: 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 easily be 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 49: find_static_strings.py.

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

Figure 50: find_static_strings.py.

Further challenges in string recovery

Ghidra auto analysis can falsely identify certain data types. When this 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. Figure 51 shows the static allocation of string structures.

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

Figure 52 shows the lines in our script that are responsible for removing the incorrect data types.

As shown in Figure 53, after executing the script all the data types are correctly identified and the strings are defined.

```
                    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 51: eCh0raix[9] static allocation of string structures.

```
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 52: find_static_strings.py.

```
                    PTR_s_http://sg3dwqfpnr4sl5hh.onion/ap_08431980  XREF[1]:    main.init.0:08208cec(R)
08431980 15 6f 28 08      addr        s_http://sg3dwqfpnr4sl5hh.onion/ap_08286f15

                    INT_08431984                                     XREF[1]:    main.init.0:08208cf2(R)
08431984 39 00 00 00      int         39h

                    PTR_s_192.99.206.61:65000_08431988               XREF[1]:    main.getInfo:08208629(R)
08431988 bb c7 27 08      addr        s_192.99.206.61:65000_0827c7bb

                    INT_0843198c                                     XREF[1]:    main.getInfo:08208623(R)
0843198c 13 00 00 00      int         13h

                    PTR_s_/etc/hosts_08431990                        XREF[1]:    net.readHosts:081448a0(R)
08431990 cc a0 27 08      addr        s_/etc/hosts_0827a0cc

                    INT_08431994                                     XREF[1]:    net.readHosts:08144896(R)
08431994 0a 00 00 00      int         Ah


                 s_http://sg3dwqfpnr4sl5hh.onion/ap_08286f15      XREF[2]:    main.init.0:08208cf8(*),
                                                                              08431980(*)
08286f15 68 74 74        ds          "http://sg3dwqfpnr4sl5hh.onion/api/GetAvailKeysByCampId/13"
         70 3a 2f
         2f 73 67 ...
```

Figure 53: eCh0raix[9] static allocation of string structures after executing find_static_strings.py.

Another issue comes 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 shown in Figures 54 and 55 is from an ARM Kaiji sample[12].

```
                     s_runtime:_panic_before_malloc_hea_002978ff              runtime.casgstatus:00043ef4(*),
"*-+*-+####@@@@!!!!first path segment in URL cannot contain colonln -s /etc/rc.d/init.d/linux_kill
/etc/rc.d/rcmath/big: mismatched montgomery number lengthsmemory reservation exceeds address space
limitpanicwrap: unexpected string after type name: reflect.Value.Slice: slice index out of
boundsreflect: nil type passed to Type.ConvertibleToreleased less than one physical page of
memoryruntime: debugCallV1 called by unknown caller runtime: failed to create new OS thread (have
runtime: name offset base pointer out of rangeruntime: panic before malloc heap
initialized\nruntime: text offset base pointer out of rangeruntime: type offset base pointer out of
rangeslice bounds out of range [:%x] with length %yssh: unmarshal error for field %s of type
%s%sstopTheWorld: not stopped (status != _Pgcstop)sysGrow bounds not aligned to pallocChunkBytestls:
failed to parse certificate from server: tls: received new session ticket from a clienttls: server
chose an unconfigured cipher suitetls: server did not echo the legacy session IDx509: failed to
parse rfc822Name constraint %qx509: failed to unmarshal elliptic curve pointx509: invalid elliptic
curve private key valueP has cached GC work at end of mark terminationattempting to link in too many
shared librariesbufio: reader returned negative count from Readchacha20poly1305: message
authentication failedcurve25519: global Basepoint value was modifiedexplicit string type given to
non-string memberfirst record does not look like a TLS handshakeslice bounds out of range [::%x]
with length %ytls: incorrect renegotiation extension contentstls: internal error: pskBinders length
mismatchtls: server selected TLS 1.3 in a renegotiationtls: server sent two HelloRetryRequest
messagesx509: internal error: IP SAN %x failed to parsebufio: writer returned negative count from
Writecrypto/rsa: key size too small for PSS signaturefailed to parse certificate #%d in the chain:
%wparsing/packing of this type isn't available yetruntime: cannot map pages i...
002976f3 2a 2d 2b
         2a 2d 2b
         23 23 23 ...
```

Figure 54: 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 55: 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. Figure 56 shows Kaiji's defined strings.

Finally, we will show an issue in versions of Ghidra decompiler view prior to version 9.2. Once a string is successfully defined, 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 earlier versions of Ghidra couldn't handle fixed length strings correctly and, regardless of the length of the string, it would display everything until it found a null character. Thankfully this issue was solved in Ghidra 9.2.

The issue is illustrated in Figures 57 and 58 using the eCh0raix sample[9].

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	000102030405060708091011...	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 56: Kaiji(12) defined strings.

```
                       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 57: 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 58: eCh0raix(9) defined string in decompile view in Ghidra 9.1.

FUTURE WORK

In this paper we proposed solutions for two issues within Go binaries to help reverse engineers when they are using Ghidra to statically analyse 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 paper 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 types system.

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

The other interesting topic is types within Go binaries. Just as 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 shown in Figures 59 – 61 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 59: 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 60: eCh0raix[9] main.info fields.

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

Figure 61: 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.

REFERENCES

[1] Ghidra. https://ghidra-sre.org/.

[2] CUJO AI Labs – Threat Intelligence Repository. https://github.com/getCUJO/ThreatIntel.

[3] Litvak, P. Kaiji: New Chinese Linux malware turning to Golang. Intezer. May 2020. https://www.intezer.com/blog/research/kaiji-new-chinese-linux-malware-turning-to-golang/.

[4] Securelist.A Zebrocy Go Downloader. January 2019. https://securelist.com/a-zebrocy-go-downloader/89419/.

[5] Palotay, D. Reverse Engineering Go Binaries with Ghidra. CUJO AI. October 2020. https://cujo.com/reverse-engineering-go-binaries-with-ghidra/.

[6] @IntezerLabs. https://twitter.com/IntezerLabs/status/1295698027517272064.

[7] @IntezerLabs. https://twitter.com/IntezerLabs/status/1295757824870539265.

[8] Anomali. Anomali Threat Research Releases First Public Analysis of Smaug Ransomware as a Service. August 2020. https://www.anomali.com/blog/anomali-threat-research-releases-first-public-analysis-of-smaug-ransomware-as-a-service.

[9] Harpaz, O. FritzFrog: A New Generation Of Peer-To-Peer Botnets. Guardicore. https://www.guardicore.com/2020/08/fritzfrog-p2p-botnet-infects-ssh-servers/.

[10] Kwiatkowski, I.; Aime, F.; Delcher, P. Holy water: ongoing targeted water-holing attack in Asia. Securelist. March 2020. https://securelist.com/holy-water-ongoing-targeted-water-holing-attack-in-asia/96311/.

[11] ircflu. https://github.com/muesli/ircflu.

[12] Anomali. The InterPlanetary Storm: New Malware in Wild Using InterPlanetary File System's (IPFS) p2p network. June 2019. https://www.anomali.com/blog/the-interplanetary-storm-new-malware-in-wild-using-interplanetary-file-systems-ipfs-p2p-network.

[13] @VK_Intel. https://twitter.com/VK_Intel/status/1281677718376120328.

[14] Hunter, B.; Gutierrez, F. EKANS Ransomware Targeting OT ICS Systems. Fortinet. July 2020. https://www.fortinet.com/blog/threat-research/ekans-ransomware-targeting-ot-ics-systems.

[15] @VK_Intel. https://twitter.com/VK_Intel/status/1233302763871854592.

[16] 360Netlab. HEH, a new IoT P2P Botnet going after weak telnet services. October 2020. https://blog.netlab.360.com/heh-a-new-iot-p2p-botnet-going-after-weak-telnet-services/.

[17] Proofpoint. TA416 Goes to Ground and Returns with a Golang PlugX Malware Loader. November 2020. https://www.proofpoint.com/us/blog/threat-insight/ta416-goes-ground-and-returns-golang-plugx-malware-loader.

[18] @IntezerLabs. https://twitter.com/IntezerLabs/status/1291355808811409408.

[19] Brandt, A. Glupteba malware hides in plain sight. Sophos News. June 2020. https://news.sophos.com/en-us/2020/06/24/glupteba-report/.

[20] Pripoae, S. There's a New a Golang-written RAT in Town. Bitdefender. October 2020. https://labs.bitdefender.com/2020/10/theres-a-new-a-golang-written-rat-in-town/.

[21] 360Netlab. Blackrota, a heavily obfuscated backdoor written in Go. November 2020. https://blog.netlab.360.com/blackrota-a-heavily-obfuscated-backdoor-written-in-go/.

[22] https://analyze.intezer.com/files/bd978ba0d723aea3106c6abc58cf71df5abe4d674d0d1fc38b37d4926d740738.

[23] hasherezade. Analyzing a new stealer written in Golang. Malwarebytes. January 2019. https://blog.malwarebytes.com/threat-analysis/2019/01/analyzing-new-stealer-written-golang/.

[24] Kimayong, P. Sysrv Botnet Expands and Gains Persistence. Juniper Networks. April 2021. https://blogs.juniper.net/en-us/threat-research/sysrv-botnet-expands-and-gains-persistence.

[25] Brandt, A. A new ransomware enters the fray: Epsilon Red. Sophos News. May 2021. https://news.sophos.com/en-us/2021/05/28/epsilonred/.

[26] @IntezerLabs. https://twitter.com/IntezerLabs/status/1401869234511175683.

[27] Harpaz, O. FritzFrog: A New Generation Of Peer-To-Peer Botnets. Guardicore. https://www.guardicore.com/labs/fritzfrog-a-new-generation-of-peer-to-peer-botnets/.

[28] Cox, R. Go 1.2 Runtime Symbol Information. July 2013. https://docs.google.com/document/d/1lyPIbmsYbXnpNj57a261hgOYVpNRcgydurVQIyZOz_o/pub.

[29] https://github.com/getCUJO/ThreatIntel/blob/master/Scripts/Ghidra/find_dynamic_strings.py.

[30] https://github.com/getCUJO/ThreatIntel/blob/master/Scripts/Ghidra/find_static_strings.py.

[31] RedNaga Security. Reversing GO binaries like a pro. September 2016. https://rednaga.io/2016/09/21/reversing_go_binaries_like_a_pro/.

[32] Zaytsev, G. Reversing Golang. Zeronights. https://2016.zeronights.ru/wp-content/uploads/2016/12/GO_Zaytsev.pdf.

[33] Wrightsell, J. Reverse engineering Go binaries using Radare 2 and Python. Carve Systems. August 2019. https://carvesystems.com/news/reverse-engineering-go-binaries-using-radare-2-and-python/.

[34] PNF Software. Analyzing Golang Executables. https://www.pnfsoftware.com/blog/analyzing-golang-executables/.

[35] https://github.com/strazzere/golang_loader_assist/blob/master/Bsides-GO-Forth-And-Reverse.pdf.

[36] https://github.com/radareorg/r2con2020/blob/master/day2/r2_Gophers-AnalysisOfGoBinariesWithRadare2.pdf.

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)	world.c	761301bb14ea3b678650fc1b6da768f009387ee726712e291d57e2d7985613d0
(2)	world.go	7cb3316a7b89eb996e8dbb0d0fb277136cd588cc54642f3b09aa84cd177cb3a2
(3)	world_c	76a5c4ef9277b97660f2c412e67ff2c3826e699913db86cd333e8f1d4fb5b8a3
(4)	world_c_strip	486a93362a6a8bc3b449fd6ba07656011c687ed31a19091c329a434bff4d75bb
(5)	world_go	d0d4781de4ffd5fbe18d59328eccd373a782eecdf55a2c5199b7dc6598cfb99e
(6)	world_go_strip	9b975bd9406a8b79a414195e184be0c82bb1593979577f0344c797f9bcd4ad0b
(7)	world_go.exe	9e36291f5fc67fdb9e5e17b636d34b39f2cc39f328916a9012a8f8d545e9d0c8
(8)	world_go_strip.exe	c5b66623942a0cea6df30541e92afe93172be7bb4dbdd42a1fa354e9edd79a1d
(9)	eCh0raix - x86	154dea7cace3d58c0ceccb5a3b8d7e0347674a0e76daffa9fa53578c036d9357
(10)	eCh0raix - ARM	3d7ebe73319a3435293838296fbb86c2e920fd0ccc9169285cc2c4d7fa3f120d
(11)	Kaiji - x86_64	f4a64ab3ffc0b4a94fd07a55565f24915b7a1aaec58454df5e47d8f8a2eec22a
(12)	Kaiji - ARM	3e68118ad46b9eb64063b259fca5f6682c5c2cb18fd9a4e7d97969226b2e6fb4
(13)	world_go_println	fa00f5ad2aa79a6245a28516bc285ae8c36f075d818787aadff6f3e850e2ec5c

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