Ready, Set, Go — Golang Internals and Symbol Recovery

By Mandiant

Published: 2022-02-28 · Archived: 2026-04-06 01:22:07 UTC

Written by: Stephen Eckels

Golang (Go) is a compiled language introduced by Google in 2009. The language, runtime, and tooling has evolved significantly since then. In recent years, Go features such as easy-to-use cross-compilation, self-contained executables, and excellent tooling have provided malware authors with a powerful new language to design cross-platform malware. Unfortunately for reverse engineers, the tooling to separate malware author code from Go runtime code has fallen behind.

Today, Mandiant is releasing a tool named GoReSym to parse Go symbol information and other embedded metadata. This blog post will discuss the internals of relevant structures and their evolution with each language version. It will also highlight challenges faced when analyzing packed, obfuscated, and stripped binaries.

Design Decisions

Go is a bit different from other languages in that it generates binaries that are fully self-contained. A system that executes a compiled Go binary doesn't require a runtime or additional dependencies to be installed. This contrasts with languages such as Java or .NET that require a user to install a runtime before binaries will execute correctly. With Go's approach, the compiler embeds runtime code for various language features (e.g., garbage collection, stack traces, type reflection) into each compiled program. This is a major reason why Go binaries are larger than an equivalent program written in a language such as C. In addition to the runtime code, the compiler also embeds metadata about the source code and its binary layout to support language features, the runtime, and debug tooling.

Some of this embedded information has been thoroughly documented, namely the pclntab, moduledata, and buildinfo structures. Each of these structures has seen major changes as Go has evolved. This evolution, combined with common obfuscator or packing tricks, can make type recovery trickier than expected. To effectively handle ever-changing runtime structures, GoReSym is based on the Go runtime source code to transparently handle all runtime versions. This makes supporting new Go versions trivial. We can also be more confident in edge cases since GoReSym uses the same parsers as the runtime.

Matching Recovered Symbols to Language Features

Go binaries without debug symbols, also referred to as stripped binaries, provide a unique challenge to reverse engineers. Without symbols, analyzing a binary can be extremely complex and time consuming. With symbols restored, a reverse engineer can begin to map disassembled code back to its original source. Figure 1 illustrates the importance of recovering symbols using the disassembly of two samples: one *without* symbols and another *with* symbols recovered using GoReSym.

BEFORE

```
lea     rax, stru_486460
xor     ebx, ebx
call    sub_404120
mov     [rsp+58h+var_20], rax
mov     rcx, [rsp+58h+var_30]
mov     [rsp+58h+var_28], rcx
lea     rax, unk_48D2A0
nop     dword ptr [rax]
call    sub_40BBC0
lea     rcx, sub_47F760
mov     [rax], rcx
mov     rcx, [rsp+58h+var_28]
mov     [rax+8], rcx
cmp     cs:dword_555FA0, 0
nop
jnz     short loc_47F6AD
mov     rcx, [rsp+58h+var_20]
mov     [rax+10h], rcx
jmp     short loc_47F6BB

                        ; CODE XREF: main_main+C0↑j
lea     rdi, [rax+10h]
mov     rcx, [rsp+58h+var_20]
call    sub_45ACA0

                        ; CODE XREF: main_main+CB↑j
mov     rbx, rax
xor     eax, eax
call    sub_43A5A0
mov     [rsp+58h+var_28], 0
mov     rax, [rsp+58h+var_20]
lea     rbx, [rsp+58h+var_28]
call    sub_405020
mov     rax, [rsp+58h+var_28]
```

AFTER

```
lea     rax, _chan_int
xor     ebx, ebx
call    runtime_makechan
mov     [rsp+58h+var_20], rax
mov     rcx, [rsp+58h+var_30]
mov     [rsp+58h+var_28], rcx
lea     rax, _chan_int_struct
nop     dword ptr [rax]
call    runtime_newobject
lea     rcx, main_main__dwrap__1
mov     [rax], rcx
mov     rcx, [rsp+58h+var_28]
mov     [rax+8], rcx
cmp     cs:runtime_writeBarrier, 0
nop
jnz     short loc_47F6AD
mov     rcx, [rsp+58h+var_20]
mov     [rax+10h], rcx
jmp     short loc_47F6BB

                        ; CODE XREF: main_main+C0↑j
lea     rdi, [rax+10h]
mov     rcx, [rsp+58h+var_20]
call    runtime_gcWriteBarrierCX

                        ; CODE XREF: main_main+CB↑j
mov     rbx, rax
xor     eax, eax
call    runtime_newproc
mov     [rsp+58h+var_28], 0
mov     rax, [rsp+58h+var_20]
lea     rbx, [rsp+58h+var_28]
call    runtime_chanrecv1
mov     rax, [rsp+58h+var_28]
```

Figure 1: Stripped Go binary vs. symbols recovered by GoReSym

Before we examine how GoReSymextracts this information, we'll use recovered symbols to illustrate core Go concepts such as channels, Go routines, deferred routines, and function returns. The examples in the sections that follow depict a Go binary that has been annotated using function names recovered by GoReSym. Table 1 summarizes how these concepts map directly to the runtime functions described in each section.

Concept	Keyword	Runtime Function Names
Goroutines	go	runtime.newproc
Channels	<- [recv channel] [send channel] <-	runtime.makechan runtime.chanrecv runtime.sendchan
Deferred routines	defer	runtime.deferproc runtime.deferprocStack runtime.deferreturn

Table 1: Go keyword-to-runtime function mappings

Go Routines and Channels

```
shl     rax, 1
mov     [rsp+20h+var_10], rax
mov     rax, rbx
lea     rbx, [rsp+20h+var_10]
call    runtime_chansend1
```

```go
1   package main
2
3   import (
4       "fmt"
5       "os"
6       "strconv"
7   )
8
9   func f(arg int, c chan int) {
10      c <- arg * 2 // send result via channel
11  }
12
13  func main() {
14      args := os.Args[1:]
15      i, err := strconv.Atoi(args[0])
16      if err == nil {
17          c := make(chan int)
18
19          go f(i, c)
20
21          result := <-c // block until received
22          fmt.Println(result)
23      }
24  }
25
```

```
lea     rax, _chan_int
xor     ebx, ebx
call    runtime_makechan
mov     [rsp+58h+var_20], rax
mov     rcx, [rsp+58h+var_30]
mov     [rsp+58h+var_28], rcx
lea     rax, _chan_int_struct
nop     dword ptr [rax]
call    runtime_newobject
lea     rcx, main_main__dwrap__1 ; go routine f
mov     [rax], rcx
mov     rcx, [rsp+58h+var_28]
mov     [rax+8], rcx
cmp     cs:runtime_writeBarrier, 0
nop
jnz     short loc_47F6AD
mov     rcx, [rsp+58h+var_20]
mov     [rax+10h], rcx
jmp     short loc_47F6BB

                        ; CODE XREF: main_main+C0↑j
lea     rdi, [rax+10h]
mov     rcx, [rsp+58h+var_20]
call    runtime_gcWriteBarrierCX

                        ; CODE XREF: main_main+CB↑j
mov     rbx, rax
xor     eax, eax
call    runtime_newproc ; spawn go routine
mov     [rsp+58h+var_28], 0
mov     rax, [rsp+58h+var_20]
lea     rbx, [rsp+58h+var_28]
call    runtime_chanrecv1
mov     rax, [rsp+58h+var_28]
```

Figure 2: Runtime functions implementing Go routines

The go keyword starts a new thread of execution that cooperatively interlaces execution with the rest of the application. This is not an operating system thread; instead, multiple Go routines take turns executing on one thread. Communication across Go routines is done via "channels" in a message passing fashion. When a channel is allocated, an interface type is passed to the runtime routine runtime.makechan, defining the type of data flowing across the channel. Data can be sent and received across a channel using the <- operator. Based on the direction, the routine runtime.chansend or runtime.chanrecv will be present in the disassembly. Channel logic is often adjacent to Go routine code, which begins execution by passing a function pointer to the runtime routine runtime.newproc.

Deferring Cleanup

If you're familiar with C++ destructors or finally blocks in C#, Go's defer keyword is similar. It allows Go programs to queue a routine for execution on function exit. This is commonly used to close handles or free resources. The runtime maintains a stack of functions to execute on scope exit in last-in, first-out (LIFO) order with each deferpushing to this stack. To add routines to the stack runtime.deferproc or its variant are called. To execute the routines on the stack the Go compiler places a call to runtime.deferreturn before the function exits. The source code and disassembly in Figure 3 illustrates this concept.

```
pointer_f        dq offset main_f
```

```go
package main

import (
    "fmt"
)

//go:noinline
func f() {
    fmt.Println("world")
}

func main() {
    defer f()
    fmt.Println("hello")
}
```

```
lea     rcx, pointer_f  ; deferred function pointer
mov     [rsp+0A0h+var_70], rcx
lea     rax, [rsp+0A0h+var_88]
nop
call    runtime_deferprocStack ; queues defer routine. runtime.deferproc does this too
test    eax, eax
jnz     short loc_47E3FC
jmp     short $+2
-------------------------------------------------------------

                        ; CODE XREF: main_main+49↑j
movups  [rsp+0A0h+var_30], xmm15
lea     rax, [rsp+0A0h+var_30]
mov     [rsp+0A0h+var_38], rax
test    [rax], al
lea     rdx, pointer_string
mov     qword ptr [rsp+0A0h+var_30], rdx
lea     rdx, hello_str
mov     qword ptr [rsp+0A0h+var_30+8], rdx
test    [rax], al
jmp     short $+2
-------------------------------------------------------------

                        ; CODE XREF: main_main+77↑j
mov     [rsp+0A0h+var_20], rax
mov     [rsp+0A0h+var_18], 1
mov     [rsp+0A0h+var_10], 1
mov     ebx, 1
mov     rcx, rbx
call    fmt_Println
nop

                        ; DATA XREF: .gopclntab:stru_50CE40↓o
call    runtime_deferreturn ; execute functions in defer stack
mov     rbp, [rsp+0A0h+var_8]
add     rsp, 0A0h
retn
```

Figure 3: Runtime functions implementing defer

Return Values with Errors

Most Go functions return both a value and an optional error. Unlike C, most Go functions return two values. Prior to Go 1.17, these values were passed on the stack. In recent versions, Go introduced a register-based ABI so both values are often stored in registers.

```go
package main

import (
    "fmt"
    "os"
    "strconv"
)

func f(arg int) (int, error) {
    if arg > 10 {
        return 0, fmt.Errorf("%d is too large", arg)
    }
    return arg * 2, nil
}

func main() {
    args := os.Args[1:]
    i, err := strconv.Atoi(args[0])
    if err == nil {
        result, err := f(i)
        if err == nil {
            fmt.Printf("Worked: %d", result)
        }
    }
}
```

```
call    strconv_Atoi
test    rbx, rbx
jnz     error_case
cmp     rax, 0Ah
```

Go 1.17. rbx = err, rax = i

```
                call    strconv_Atoi
                mov     rax, [rsp+78h+var_68]
                cmp     [rsp+78h+var_60], 0
                jz      short success_case

function_end:                           ; CODE XREF: main_main+A5↓j
                                        ; main_main+120↓j
                mov     rbp, [rsp+78h+var_8]
                add     rsp, 78h
                retn
; ---------------------------------------------------------------

success_case:                           ; CODE XREF: main_main+87↑j
                cmp     rax, 0Ah
```

Go 1.16. stack slot = err, rax = i

Figure 4: Stack vs. register ABI in Go return values

Returning a single value and an error is a common idiom but Go doesn't place a limit on the number of return values. It's possible to see functions returning tens of values.

Now that it's apparent why symbol recovery is significant, let's examine some important Go structures that GoReSym parses to extract symbol information, beginning with the pclntab structure.

PCLNTAB

The pclntab structure is short for “Program Counter Line Table”. The design for this table is documented on this page, but has evolved in more recent Go versions. The table is used to map a virtual memory address back to the nearest symbol name to generate stack traces with function and file names. The original specification states this information can be used for language features such as garbage collection, but this doesn’t appear to be true in modern runtime versions. For symbol recovery purposes, the pclntab is important because it stores function names, function start and end addresses, file names, and more.

Locating the pclntab works differently depending on the file format. For ELF and Mach-O files, the pclntab is located in a named section within the binary. ELF files contain a section named .gopclntab that stores the pclntab while Mach-O files use a section named __gopclntab. Locating the pclntab in PE files is more complex and begins with identifying a symbol table referred to in the Go source code as .symtab.

For PE files, the .symtab symbol table is pointed to by the FileHeader.PointerToSymbolTable field. A symbol in this table named runtime.pclntab contains the address of the pclntab. ELF and Mach-O files also have a .symtab symbol table that contains a runtime.pclntab symbol but do not rely on it to locate the pclntab. To locate the .symtab in ELF files, look for a section named .symtab of type SH_SYMTAB. In Mach-O files, the .symtab is referenced by an LC_SYMTAB load command. The following is a list of relevant symbols present in the .symtab:

- runtime.pclntab
- runtime.epclntab
- runtime.symtab
- runtime.esymtab
- pclntab
- epclntab
- symtab
- esymtab

Symbols without the runtime prefix are legacy symbols used instead of the runtime-prefixed ones. These won’t be referenced going forward but note that if the runtime-prefixed symbols are looked up and don’t exist, the legacy ones are usually attempted as a fallback. Symbols with an e prefix (e.g., epclntab) denote the end of a corresponding table.

The runtime.symtab symbol points to a second, Go-specific symbol table that is no longer filled with symbols as of Go 1.3. In ELF and Mach-O files, the runtime.symtab symbol points to a named section – .gosymtab and __gosymtab, respectively – that stores this legacy table. Despite no longer being filled with symbols, many tools expect the symbol and section pointed to by this symbol to exist. The Go runtime, without modification, will refuse to parse binaries without this legacy symbol table.

The graphical overview in Figure 5 illustrates how the pclntab, the .symtab, and this legacy symbol table are related.

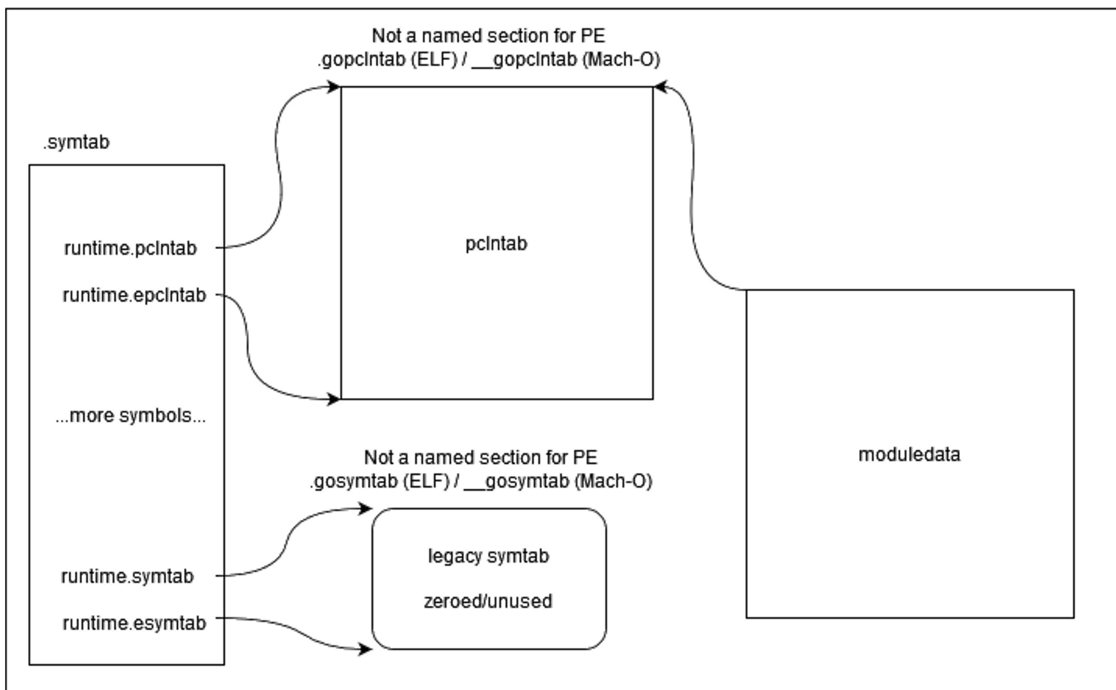

Figure 5: Layout of a Go binary's symbol tables

When a Go binary is stripped, the .symtab symbol table is zeroed out or not present. This removes the ability to find symbols such as runtime.pclntab. However, the data those symbols point to, such as the pclntab itself, is still present in the binary. For ELF and Mach-O files, the named sections (e.g., .gopclntab) are also still present and can be used to locate the pclntab . Therefore, manual location of the pclntab for both stripped and unstripped binaries can be performed in one of three ways, with earlier steps being preferred:

1. Locate the .symtab symbols and resolve runtime.pclntab
2. For ELF and Mach-O files, locate the .gopclntab or __gopclntab section
3. Scan the binary to locate the pclntab manually

The last option is the ultimate fallback mechanism for locating the table. Byte scanning is always required for stripped PE files and occasionally for ELF and Mach-O binaries with mutated section names. The byte scanning method involves searching for the pclntab header structure shown in Figure 6.

```
 type pcHeader struct {       // header of pclntab
     magic          uint32
     pad1, pad2     uint8   // 0,0
      ...
}
```

Figure 6: pclntab header structure

The start of the pclntab is always a version-specific magic number. This number dictates table parsing behavior for each layout format, which changes at a different versioning cadence than the rest of the runtime. Valid magic numbers are found on this GitHub page. As a result, we can perform a linear byte scan for these magic numbers to identify the pclntab.

Some packers such a UPX change section names and sizes. When scanning packed binaries, care must be taken to merge all section data prior to performing a scan. For some packed binaries, the pclntab spans two sections. If each section is scanned individually, only a partial pclntab may be discovered.

GoReSym includes the following changes to the Go runtime code to locate the pclntab:

- Modification of the file format parser to continue if the runtime.symtab symbol is missing. Since Go 1.3, this symbol, and the legacy table it points to, is not used but the runtime code validates it *always* exists, which is unnecessary.
- When symbols are located, validate they point to plausibly correct data
- If standard symbol or section-based location fails, perform byte scanning to locate the pclntab
- Return additional information about the pclntab such as its virtual address

Beyond storing important function metadata, the pclntab can also be used to locate a second metadata structure named moduledata.

MODULEDATA

The moduledata structure is an internal runtime table that stores information about the layout of the file as well as runtime information used to support core features such as garbage collection and reflection. Unlike the pclntab, this structure cannot be found via symbols. Its layout also changes much more frequently.

To locate the structure, we first must examine how it is defined. There are two definitions that depend on the Go runtime version. In modern versions, the pclntab is found via an offset in the pcHeaderstructure shown in Figure 7.

```
 type moduledata struct {
     pcHeader     *pcHeader
               …
}
type moduledata struct {
     pclntable    []byte
               …
}
```

Figure 7: Two variants of moduledata's structure

In older Go versions, the pclntab was directly embedded as a byte array or "slice" rather than referenced by an offset. The in-memory layout of both, however, is similar if we examine the format of a Go slice (Figure 8).

```
 type GoSlice struct {
  data *byte;
  len int;
  capacity int;
}
```

Figure 8: Go slice structure

Since a slice holds its data pointer as the first member, both versions of moduledata start as if they were defined like Figure 9.

```
type moduledata struct {
   pclntable *byte
   ...starts to differ...
}
```

Figure 9: Common moduledata memory layout

Therefore, to locate the moduledata table, we can do a linear byte scan for the virtual address of the pclntab (i.e., search for a structure that holds a pointer to this address as the first member). Alternatively, moduledata can also be found by disassembling the function runtime.moduledataverify, which holds a pointer to the moduledata and walks it.

Unlike the pclntab, the runtime does not include a generic moduledata parser that works across all Go versions. To overcome this, GoReSym includes parsing logic for each supported runtime version. Despite the moduledata layout changing frequently, field encodings such as strings are more stable. As a result, it’s easy to write utility routines that handle common encodings. This eases the transition to a new runtime version as only a few utility routines and structure layouts need changed.

Within the moduledata array there is a list named typelinks. It stores type information for types defined in a Go program and is used for reflection and garbage collection. The structure stored in this list is named rtype and varies between runtime versions; however, in each format the name of the type can be extracted. This can be useful for situations like channels where types are passed around. Without the type name it's not possible to know the type of data being passed over the channel.

Here’s an example *before* type recovery where the runtime_makechan function takes a pointer to an interface:

```
v5 = runtime_makechan((__int64)&unk_4A3D20, 0LL);
```

After type recovery we can name the interface:

```
v5 = runtime_makechan((__int64)&chan_bool, 0LL);
```

This indicates the channel has Boolean values passed through it. Examining the Go source code for this call confirms the type:

```
main_init_done = make(chan bool)
```

Each rtype can have additional fields depending on the type it represents. For channels, the extended structure resembles the following, where the base rtype is embedded as the first member (Figure 10).

```
 type chantype struct {
    typ  _type
    elem *_type
    dir  uintptr
}
```

Figure 10: Extended channel rtype structure

The dir field within this extended type structure allows us to recognize the direction of a channel (i.e., send, receive, or both). GoReSym handles all extended type structures and extracts additional relevant information. For types such as interface and struct, this extended information lists methods as well as fields and their offsets, which facilitates reconstruction of both internal and user-defined structures.

BUILDINFO

Starting in Go 1.18, additional metadata is provided in a table named buildinfo. This table is emitted by default but can be easily omitted with compiler flags. When present, the table can provide the following: compiler and linker flags, values of the environment variables GOOS and GOARCH, git information, and information about the package name of both the main and dependency packages. In previous Go versions, finding data such as the runtime version had to be done by linear scanning for string fragments like "go1.". This scan was required because the global value runtime.buildVersion is not referenced by any symbols, so locating it is difficult.

The buildinfo structure has a named section in ELF and Mach-O files but the runtime disregards this information during the location process. To identify this information, the runtime reads the first 64KB of a file, performs a linear scan for the magic string "\xff Go buildinf:", and parses the data immediately after. When this information is present, GoReSym uses it and only relies on byte scanning techniques when the GOOS and GOVERSION values are missing.

GoReSym Usage

GoReSym is a standalone application that can be executed on the command line or incorporated into a larger batch processing script. It doesn’t perform additional binary analysis outside of Go’s symbols. As a result, data extraction typically finishes in 1-5 seconds for even the most complex binaries.

By default, GoReSym prints concise output instead of all extracted information. Various flags can be used to alter GoReSym's behavior. The -t flag instructs GoReSym to emit type and interface lists, which are useful when reversing channels and other routines that accept types. In some cases, not all types present in the disassembly are listed in the typelinks array. Channel objects are a good example of this is. Figure 11 shows an unreferenced type being loaded into the rax register prior to the runtime_newobject call.

```
lea     rax, _chan_int
xor     ebx, ebx
call    runtime_makechan
mov     [rsp+58h+var_20], rax
mov     rcx, [rsp+58h+var_30]
mov     [rsp+58h+var_28], rcx
lea     rax, unk_48D2A0
nop     dword ptr [rax]
call    runtime_newobject
```

Figure 11: Unreferenced type 0x48D2A0

In these instances the -m flag and the virtual address of the type can be passed to GoReSym as shown in Figure 12. Note the -human flag is used to emit flat output rather than JSON.

```
 ./GoReSym -m 0x48D2A0 -human ../gotests/example

-TYPE STRUCTURES-
VA: 0x48d2a0
type struct {
    .F          uintptr
    .autotmp_11 int
    .autotmp_13 chan int Direction: (Both)
}
```

Figure 12: Manual rtype structure extraction

The type at the provided address is parsed by GoReSym and printed, allowing us to see it’s related to the allocation of the channel. Other useful flags include -p to print file paths present in a binary and the -d flag to print information that has been classified as part of a default package.

Finally, some obfuscators remove Go runtime version information. Because types are version-specific, this prevents GoReSym from parsing types. To overcome this, the -v flag can be provided to suggest a runtime version. GoReSym will attempt to parse types using structures from the suggested version. Trial and error can be used to guess the correct version. Also, the pclntab version can be used as a hint for where to start; however, recall the pclntab version differs from the runtime version.

Existing Tools

There are public and commercial tools that support some, or perhaps all, of the type parsing logic addressed in GoReSym. Prior works such as Redress and IDAGolangHelper are excellent and should be celebrated. The primary difference between GoReSym and existing tools is that GoReSym is based on the Go runtime source code. This helps counter the rapid pace of Go internal structure evolution. Additionally, because the runtime is already cross-architecture, GoReSym is too. All new parsing logic takes care to correctly support 32 and 64-bit big and little-endian architectures. Care was also taken to handle unpacked or partially corrupted samples where possible, something other tools may struggle with. Overall, GoReSym is designed to work in cases where other tools fail and offers a way to ease tool maintenance as Go evolves.

References and Credits

The original package classification ideas, but not code, behind redress is used in GoReSym and the recursive type parsing idea from the JEB blog post helped when implementing typelink enumeration. Thanks to these authors for making their work publicly available.

- Redress GitHub
- IDAGolangHelper GitHub
- Analyzing Golang Executables post on the JEB Decompiler Blog

Now go check out GoReSym!

Posted in

- Threat Intelligence
- Security & Identity

Source: https://www.mandiant.com/resources/blog/golang-internals-symbol-recovery