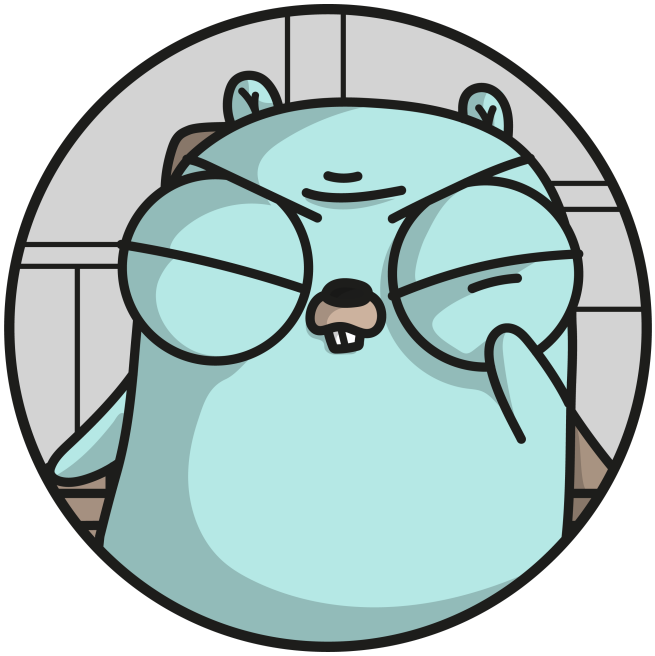
Let’s Go into the rabbit hole (part 2) — the challenges of dynamically hooking Golang programs

blog.quarkslab.com/lets-go-into-the-rabbit-hole-part-2-the-challenges-of-dynamically-hooking-golang-program.html

June 11, 2024

Golang is the most used programming language for developing cloud technologies. Tools such as *Kubernetes*, *Docker*, *Containerd* and *gVisor* are written in Go. Despite the fact that the code of these programs is open source, there is no way to analyze and extend their behaviour dynamically (for example through binary instrumentation) without recompiling their code. Is this due to the complex internals of the language? In this second blog post, we’ll showcase how to create runtime hooks for *Golang* programs using FFI (foreign function interfaces).

A Golang gopher doing a facepalm when looking into CGO's internals (by Ashley McNamara)



Introduction

Hooking, also known as a “detour”, is a mechanism for redirecting the execution flow of a program. Changing a program's execution flow can be done in two ways - statically, by modifying the source code of a program or its binary representation, and dynamically, by modifying the loaded program image within a running process. In these series of articles, we're going to talk about dynamic hooking or dynamic instrumentation of Go programs.

A lot of tools, libraries and literature can be found on the Internet on how to do runtime hooks for different programming languages such as C and C++. As of today, the most famous tool for dynamic instrumentation is probably Frida. On the other hand, hooking Go

code at runtime is not a straightforward process. It gets even more interesting when one tries to replace the execution flow of a Go program with the execution flow of another different Go program.

Why Hook Golang Programs During Runtime and What are the Challenges?

Nowadays, most modern cloud technologies are written in *Golang* (e.g. *Kubernetes*, *Docker*, *Containerd*, *runc*, *gVisor*, etc.). Most of these technologies have a big and complex architecture which is cumbersome to analyze statically. It could be great to have the means to analyze these tools dynamically alongside the static analysis. Sadly, at the time of writing, there isn't any solution for dynamic analysis without recompiling the source code of the programs. This could be a problem, because sometimes we can't modify the source code of these tools, and should interact directly with the process which is already executing the code. But why aren't there any tools in the wild which allow the insertion of some arbitrary logic inside a running Go program? We suppose that one of the problems could be that Gopl (*Golang* programming language) has a different ABI (Application binary interface) than the one used in C and C++ (hence, *Frida* does not work out of the box :( ). In addition, *Golang* incorporates a language-specific runtime which is responsible for complex procedures such as garbage collection and scheduling of goroutines. This runtime, and the way it is loaded inside the program, completely change the way we construct and insert hooks. Last but not least, initially the Gopl was intended to be self-contained — it was not designed to be extendible during runtime (e.g. loading shared libraries). Happily, this changed, but Go programs are still statically linked if they don't use the `net` or the `user` packages or they don't make use of `cgo`. However, with some assumptions and tweaking we were able to circumvent these problems.

In the previous blog post we showcased how we could dynamically instrument Golang code using x86 assembly and C. We also discussed some of the limitations of such approach:

- It is platform dependent and will not work on platforms such as Windows because of the platform's different ABI;
- It is architecture dependent and will not work for CPU architectures such as ARM or MIPS because of the used x86 assembly;
- All Go types have to be translated into C structures;
- It introduces global latency by slowing the garbage collection process and by starving waiting goroutines on the same processor.

In this blog post we'll demonstrate an approach which defeats some of the above limitations. Let's Go into the rabbit hole!

Note: The content below was produced on x86_64 Linux with go version go1.21.1

Foreign Function Interfaces in GO (CGO)

FFI (foreign function interface) is a common concept in modern programming languages. It allows programs written in one programming language to call procedures (or routines) written or compiled in other programming languages. In Go, this is implemented with the help of CGO which allows Go code to call routines written in C or C++ (or any language using the System V ABI). Under the hood, this is not a straightforward procedure and involves a lot of machinery. This is because of the language's particularities - the concurrent garbage collection, the goroutines' infinite resizable stacks and, of course, the custom ABI. To understand how FFI works in Go, let's look at the following example:

```
//example1.go
package main
/*
// The C code below will be compiled separately and
// linked to the Go code in this file

int AddGoToC(int a, int b) {
    return a+b;
}
*/
import "C" // <---- invokes CGO

func main() {
    a := C.int(4)
    b := C.int(2)
    res := C.AddGoToC(a, b)
    println("Result of AddGoToC: ", res)
}
```

Note: The commented C code is not just a code comment, it's going to be compiled alongside the Go code. Furthermore, the `import "C"` directive has to be just after the lines of commented C code (no blank lines in between).

```
# With the CC and CGO_CFLAGS environment variables we define what compiler to use
# for the C code and the flags which should be passed to it
# (for more info see the output of "go env" or do "go help env")
# To build Go files into executables, we need to first create a project
$ go mod init example1
$ CC="gcc" CGO_CFLAGS='-O2 -g' go build -o example1 example1.go && ./example1
Result of AddGoToC:  6
```

From the example above, we can see how one could call a C function from Go code. We can also do it the other way around - call Go code from C. Let's see our second example:

```
//example2.go
package main
/*
// Here we declare the signature of the C function which is going to be called from
the Go code
typedef long long ll;
ll AddCtoGo();
*/
import "C"

// The "export" statement will make the AddGo procedure callable
// from C/C++ code
//export AddGo
func AddGo(a, b int) int {
    println("Hello, I was called from C code!")
    return a + b
}

func main() {
    res := C.AddCtoGo()
    println("Result of calling AddGo through AddCtoGo: ", res)
}
```

In the above example, we see a call to a C function called `AddCtoGo` from the Go program's `main` function. This procedure is supposed to call back into the `AddGo` routine. But there is only the declaration of the function. As you'll see later, CGO is a massive code generator and when building the final executable, it will produce a bunch of C source files and their associated headers. If we define the `AddCtoGo` in the `example2.go` file, CGO will duplicate this definition inside two separate files which will result in a linker error because of a duplicated symbol definition. Hence, we have to define the C routine in a separate `.c` file :

```
// example2.c
// Go integers are 64 bits long while in C they're 32 bits long so we should use a
type with a correct size
typedef long long ll;

// The below function definition will be generated by CGO when building the
executable
// However, GCC (or other C compatible compiler) needs
// a signature declaration beforehand to be able to correctly allocate space for
function arguments
// arrange the function call ABI
extern ll AddGo(ll a, ll b);

// And finally we define our function
#include <stdio.h>
ll AddCtoGo() {
    puts("Hello, I was called from Go code");
    ll a = 4;
    ll b = 2;
    return AddGo(a, b);
}
```

By defining the C function in a separate file, the compilation process also changes:

```
$ go mod init example2
...
$ go build example2.go
# command-line-arguments
/usr/local/go/pkg/tool/linux_amd64/link: running gcc failed: exit status 1
/usr/bin/ld: /tmp/go-link-2187132663/000001.o: in function
`_cgo_90f27d3222f3_Cfunc_AddCtoGo':
/tmp/go-build/cgo-gcc-prolog:51: undefined reference to `AddCtoGo'
collect2: error: ld returned 1 exit status
```

One solution to this problem is to execute the `build` command in the following way:

```
$ go build -o example2  && ./example2
Hello, I was called from Go code
Hello, I was called from C code!
Result of calling AddGo through AddCtoGo: 6
```

Before understanding what is going on under the hood when using CGO, let's look at a 3rd and last example. With CGO we can call Go code through a C function without us defining it!

```
//example3.go
package main
/*
#include <stdio.h>
// We should first declare a C function for which arguments and return
// value types have a Golang equivalent.

extern long long int AddGo(long long int a, long long int b);
*/
import "C"

//export AddGo
func AddGo(a, b int64) int64 {
    println("I was called through a C interface!")
    return a + b
}

func main() {

    t := C.AddGo(4, 2)
    println("Result of calling a Go function through a generated C interface:", t)
}

$ go mod init example3
$ go build -o example3 example3.go && ./example3
I was called through a C interface!
Result of calling Go function through a generated C interface: 6
6
```

In the above example, the Go routine `AddGo` is called from the Go's `main` function by passing through an intermediate C interface. We also see that there is no explicit conversion from Go constants to C constants. But why did we need only a function declaration to call the Go procedure? To understand that, we have to dive a bit deeper into the functioning of CGO. This is also the basis of the dynamic instrumentation approach that we want to showcase in this blog post. Let's see what is going on under the hood when using FFI and CGO!

Note: From this moment on we're going to refer to the Go function AddGo as main.AddGo to avoid confusion with content generated by cgo

Let's CGO Into the Rabbit Hole

As we mentioned earlier, CGO is one massive code generator. In the Golang toolchain, there is also a tool called cgo. It processes Go files that contain the `import "C"` directive and then generates a bunch of C and Go code files. It then orchestrates their compilation using the C (like gcc or clang) and Go compilers. The purpose of generating this boilerplate

content is to manage the interlanguage calling conventions and to ensure that Go and C code can work together. Let's now compile the above 3rd example using the `build` frontend and look at what is actually going on:

```
# -x is going to output the actions performed by the build frontend
# -work is going to preserve the work directory used by golang to compile and link
the code into an executable
$ go build -x -work
############  BLOCK 1 ############
WORK=/tmp/go-build4112644966
mkdir -p $WORK/b001/
cd <redacted>/example3
TERM='dumb' CGO_LDFLAGS='"-O2" "-g"' /usr/local/go/pkg/tool/linux_amd64/cgo -objdir
$WORK/b001/ -importpath example3 -- -I $WORK/b001/ -O2 -g ./main.go
cd $WORK/b001
############  BLOCK 2 ############
TERM='dumb' gcc -I <redacted>/example3 -fPIC -m64 -pthread -Wl,--no-gc-sections -
fmessage-length=0 -ffile-prefix-map=$WORK/b001=/tmp/go-build -gno-record-gcc-
switches -I ./ -O2 -g -frandom-seed=rd0r8Mn0waQerYT5Yy-q -o ./_x001.o -c
_cgo_export.c
TERM='dumb' gcc -I <redacted>/example3 -fPIC -m64 -pthread -Wl,--no-gc-sections -
fmessage-length=0 -ffile-prefix-map=$WORK/b001=/tmp/go-build -gno-record-gcc-
switches -I ./ -O2 -g -frandom-seed=rd0r8Mn0waQerYT5Yy-q -o ./_x002.o -c main.cgo2.c
TERM='dumb' gcc -I <redacted>/example3 -fPIC -m64 -pthread -Wl,--no-gc-sections -
fmessage-length=0 -ffile-prefix-map=$WORK/b001=/tmp/go-build -gno-record-gcc-
switches -I ./ -O2 -g -frandom-seed=rd0r8Mn0waQerYT5Yy-q -o ./_cgo_main.o -c
_cgo_main.c
cd <redacted>/example3
TERM='dumb' gcc -I . -fPIC -m64 -pthread -Wl,--no-gc-sections -fmessage-length=0 -
ffile-prefix-map=$WORK/b001=/tmp/go-build -gno-record-gcc-switches -o
$WORK/b001/_cgo_.o $WORK/b001/_cgo_main.o $WORK/b001/_x001.o $WORK/b001/_x002.o -O2
-g
############  BLOCK 3 ############
TERM='dumb' /usr/local/go/pkg/tool/linux_amd64/cgo -dynpackage main -dynimport
$WORK/b001/_cgo_.o -dynout $WORK/b001/_cgo_import.go
############  BLOCK 4 ############
cat >/tmp/go-build4112644966/b001/importcfg << 'EOF' # internal
# import config
packagefile runtime/cgo=<redacted>/.cache/go-
build/49/495f77f39e0f47232d4a4fd954c94aa277964d83eaab5243606bad826fb61dfc-d
packagefile syscall=<redacted>/.cache/go-
build/2a/2a6cff20910af14df493cb9dfd49611c6fe453ff7815be78f4cac98b938d18d3-d
packagefile runtime=<redacted>/.cache/go-
build/ed/ede559fa50e219dabf49960d1bd8c73a79031dab0b0227484e5577c607602d01-d
EOF
/usr/local/go/pkg/tool/linux_amd64/compile -o $WORK/b001/_pkg_.a -trimpath
"$WORK/b001=>" -p main -lang=go1.21 -buildid rd0r8Mn0waQerYT5Yy-
q/rd0r8Mn0waQerYT5Yy-q -goversion go1.21.1 -c=4 -nolocalimports -importcfg
$WORK/b001/importcfg -pack $WORK/b001/_cgo_gotypes.go $WORK/b001/main.cgo1.go
$WORK/b001/_cgo_import.go

/usr/local/go/pkg/tool/linux_amd64/pack r $WORK/b001/_pkg_.a $WORK/b001/_x001.o
$WORK/b001/_x002.o # internal
/usr/local/go/pkg/tool/linux_amd64/buildid -w $WORK/b001/_pkg_.a # internal
############  BLOCK 5 ############
cp $WORK/b001/_pkg_.a <redacted>/.cache/go-
```

```
build/86/86684d8a7edb3314434284fc00e0e01ec58948ae5bebe2b398ef38f24e5e2677-d #
internal
cat >/tmp/go-build4112644966/b001/importcfg.link << 'EOF' # internal
packagefile example3=/tmp/go-build4112644966/b001/_pkg_.a
packagefile runtime/cgo=<redacted>/.cache/go-
build/49/495f77f39e0f47232d4a4fd954c94aa277964d83eaab5243606bad826fb61dfc-d
packagefile syscall=<redacted>/.cache/go-
build/2a/2a6cff20910af14df493cb9dfd49611c6fe453ff7815be78f4cac98b938d18d3-d
packagefile runtime=<redacted>/.cache/go-
build/ed/ede559fa50e219dabf49960d1bd8c73a79031dab0b0227484e5577c607602d01-d
packagefile runtime/internal/sys=<redacted>/.cache/go-
build/14/144f1e44fcc9e07dbe22009efae8b920b34004671f53bc44fe08c1200a3ecfe5-d
packagefile sync=<redacted>/.cache/go-
build/4e/4e42f270a1d9e284f04c4089b2574a424a5e9714ea906c7fd51ba7e63a8690b5-d
packagefile sync/atomic=<redacted>/.cache/go-
build/80/800c0d702a5f80d3aa8d308e77b228624efc4f07fc8c798158b86d24ff4fe3cd-d
packagefile errors=<redacted>/.cache/go-
build/b9/b9d803dfc325448bf4561bc01e9c523bddf09fa73a76f4dd12a298bc5fafe5ea-d
packagefile internal/bytealg=<redacted>/.cache/go-
build/c1/c1920a4806fb93ffb218f34f836052255c0523030f1633b3b8c63880ba7abdbf-d
packagefile internal/itoa=<redacted>/.cache/go-
build/f3/f333d53f5c8726457a0384659c73cd7e3a2b7e9a24ab681044d6115b83205609-d
packagefile internal/oserror=<redacted>/.cache/go-
build/1f/1f033cbe205f7369408b54d454cd94f9cb2cc48d9245fc6b498fee4f587a5701-d
packagefile internal/race=<redacted>/.cache/go-
build/59/5990c54bb02fd6fc6dbe9b98bebcf67925fc7aa8f2f82cee70dc6cd8a3e8cf4b-d
packagefile internal/abi=<redacted>/.cache/go-
build/4e/4eccdf6c319d9ecd751cd477a724658c96c9e8d88021e143e85136f6624ce64d-d
packagefile internal/coverage/rtcov=<redacted>/.cache/go-
build/0e/0e6ee4884ddf7f20fa61a3f672f33d5d0ca1bd8763f1a71d319cb0ddf2ae4695-d
packagefile internal/cpu=<redacted>/.cache/go-
build/4b/4b32895c30dff7d4163cbe8754ac0bacc43269d36aa70d2075c0212bd07df186-d
packagefile internal/goarch=<redacted>/.cache/go-
build/ad/ada09974db35ae787eccd6e8a73738ecb511ca92bc1d6939ce9e7ec6def407d3-d
packagefile internal/godebugs=<redacted>/.cache/go-
build/c9/c93ca8dd9fe79c89d1f10b077cb92f0083ea5c4b1163d93aece26666b56b51f9-d
packagefile internal/goexperiment=<redacted>/.cache/go-
build/08/0885babf0ba42d944048a6b25e0de58131d1b73fd329ccbc10aa29a81f4339de-d
packagefile internal/goos=<redacted>/.cache/go-
build/6f/6f05cebeae445234f466472b82b6b24e7271fd3ff18899fb6c71ad94e3b22200-d
packagefile runtime/internal/atomic=<redacted>/.cache/go-
build/c3/c39b63680e461bd008c040457579385f89df4b54bcc03dca4ca70d6cbcd79b2f-d
packagefile runtime/internal/math=<redacted>/.cache/go-
build/80/80bb31a4927a7bf44f2a3dbc06f9ba37367530e276220b6ebc41055a28a684b4-d
packagefile runtime/internal/syscall=<redacted>/.cache/go-
build/e1/e1678d9da9687c2f556caccc2124fff7de002e1c70bbe7d162768da4dceaaa44-d
packagefile internal/reflectlite=<redacted>/.cache/go-
build/35/3503f065aff2c2bb8b76336dde8b87fbe1cf2cf03633646ef1a02a595c3c4429-d
packagefile internal/unsafeheader=<redacted>/.cache/go-
build/6e/6eec1c69a7b7f0f4d8930730779a071eafae2b158dd4d5209dcc42ccd6efe727-d
modinfo
"0w\xaf\f\x92t\b\x02A\xe1\xc1\a\xe6\xd6\x18\xe6path\texample3\nmod\texample3\t(devel
```

```
)\t\nbuild\t-buildmode=exe\nbuild\t-
compiler=gc\nbuild\tCGO_ENABLED=1\nbuild\tCGO_CFLAGS=\nbuild\tCGO_CPPFLAGS=\nbuild\t
CGO_CXXFLAGS=\nbuild\tCGO_LDFLAGS=\nbuild\tGOARCH=amd64\nbuild\tGOOS=linux\nbuild\tG
OAMD64=v1\n\xf92C1\x86\x18 r\x00\x82B\x10A\x16\xd8\xf2"
EOF
mkdir -p $WORK/b001/exe/
cd .
/usr/local/go/pkg/tool/linux_amd64/link -o $WORK/b001/exe/a.out -importcfg
$WORK/b001/importcfg.link -buildmode=exe -
buildid=rfXXPRnKZejfvpJgCwuM/rd0r8Mn0waQerYT5Yy-
q/gSzN3qwcRBu3Ln61mkU2/rfXXPRnKZejfvpJgCwuM -extld=gcc $WORK/b001/_pkg_.a
/usr/local/go/pkg/tool/linux_amd64/buildid -w $WORK/b001/exe/a.out # internal
cp $WORK/b001/exe/a.out example3
```

For better comprehension, we've separated the most important phases into logical blocks. We're not going to dive deep into all of them because that is not the subject of this blog post. However, here is a high-level overview of what is happening:

1. **Block 1** -- The `cgo` tool will analyse the `main.go` file (or the whole module if there are more files) and generate a set of Go, C and object files inside the working directory. This is the above-mentioned boilerplate content.

2. **Block 2** -- Using the defined C compiler in `go env` (here `gcc`), compiles the C files into object files. The result of the compilation is called `_cgo_.o`.

3. **Block 3** -- The `cgo` tool will generate a _cgo_import.go file containing import and link directives for functionalities defined in `libc`. The produced binary will be dynamically linked to `libc` because some of the generated C code requires it (RIP statically linked binaries).

4. **Block 4** -- The generated Go files are compiled and packed inside an object archive (`_pkg_.a`) alongside the produced C object files by `gcc`. The generated `buildid` is used for compilation caching.

5. **Block 5** -- Everything is finally linked together (the generated C and Go object files stored in an archive as well as the necessary runtime functionalities) using first the Go linker and then the GNU system linker `ld` (called external linker). The latter is used through `gcc` and the result is a dynamically linked executable.

What is going on in the generation phase is quite interesting and, as mentioned earlier, it is essential in order to understand the approach we want to showcase. So let's take a look deeper.

How does CGO glue Go and C code?

The output of the command used in the first block is:

```
$ ls /tmp/go-build4112644966/b001
_cgo_export.c  _cgo_export.h  _cgo_flags  _cgo_gotypes.go  _cgo_main.c  _cgo_.o
main.cgo1.go  main.cgo2.c
```

Let's inspect the contents of the `main.cgo1.go` file:

```
// Code generated by cmd/cgo; DO NOT EDIT.


package main
/*
#include <stdio.h>
// We should first create a C function declaration which arguments and return
// value types have a Golang equivalent.

extern long long int AddGo(long long int a, long long int b);
*/
import _ "unsafe"

//export AddGo
func AddGo(a, b int64) int64 {
    println("I was called through a C interface!")
    return a + b
}

func main() {

    t := ( /*line :19:7*/_Cfunc_AddGo /*line :19:13*/)(4, 2)
    println("Result of calling a Go function through a generated C interface:", t)
    println("Result of calling a Go function through a generated C interface:", t)
}
```

We see the code from our 3rd example but there is a small change - the call to `C.AddGo(4, 2)` is replaced with a call to `_Cfunc_AddGo(4, 2)`. The definition of this function can be found in the `_cgo_gotypes.go` file:

```
//_cgo_gotypes.go
//go:cgo_unsafe_args
func _Cfunc_AddGo(p0 _Ctype_longlong, p1 _Ctype_longlong) (r1 _Ctype_longlong) {
    _cgo_runtime_cgocall(_cgo_819563c28add_Cfunc_AddGo,
uintptr(unsafe.Pointer(&p0)))
    if _Cgo_always_false {
        _Cgo_use(p0)
        _Cgo_use(p1)
    }
    return
}

...
type _Ctype_longlong int64
//go:linkname _cgo_runtime_cgocall runtime.cgocall
func _cgo_runtime_cgocall(unsafe.Pointer, uintptr) int32
//go:linkname _Cgo_always_false runtime.cgoAlwaysFalse
var _Cgo_always_false bool
//go:linkname _Cgo_use runtime.cgoUse
func _Cgo_use(interface{})
```

The file mainly contains type wrappers and function stubs which are going to be linked later by the internal Go linker to their implementations which are in the Go runtime.

The `_Cfunc_AddGo` function first calls into `_cgo_runtime_cgocall` which will be linked to `runtime.cgocall`. Below it, we see a call to _Cgo_use(`runtime.cgoUse`) taking as an argument each of the function arguments. If you look into the description of this function, it's an empty function! The purpose of these two calls to it and the conditional branch is related to the Go compiler's escape analysis and the argument liveliness information which will be emitted to the runtime by the compiler. Without going into too much detail, this will ensure that the Go arguments passed to the foreign code will not be garbage collected during its execution. This will, however, have some considerable limitations with regard to our approach (more on that in the next article).

The runtime.cgocall function is going to enter in a system call - in our example **it will announce that it will start executing foreign code**. This announcement will make the runtime create another M (OS thread) on which it is going to move all the current work P (processor) associated with M. After that it will turn off asynchronous preemption as the runtime should no longer have the possibility to send signals and to try to preempt the execution. `runtime.cgocall` will then call `runtime.asmcgocall` passing `_cgo_819563c28add_Cfunc_AddGo` and the frame as arguments. The assembly routine is going to switch, if necessary, the G (goroutine) stack to an M stack. In the previous article, we mentioned that G stacks were considerably smaller (2KB) than the OS thread stacks (8MB) on which regular C programs usually run. As the runtime can't predict the exact amount of stack space that the foreign code will use nor dynamically increase the size of the stack, it is required to switch back to the OS-allocated one. Finally,

`runtime.asmcgocall` calls into `runtime.asmcgocall_landingpad` **which is going to perform, as we call it, an ABI switch** and jump to the first argument of the call to `runtime.cgocall` - `_cgo_819563c28add_Cfunc_AddGo` passing as an argument the `uintptr(unsafe.Pointer(&p0))`. The ABI switch here simply means that rather than putting the address of `p0` in RAX (or on the stack) following the Go ABI, it will put it in RDI following the System V ABI.

```
//_cgo_gotypes.go
//go:cgo_import_static _cgo_819563c28add_Cfunc_AddGo
//go:linkname __cgofn__cgo_819563c28add_Cfunc_AddGo _cgo_819563c28add_Cfunc_AddGo
var __cgofn__cgo_819563c28add_Cfunc_AddGo byte
var _cgo_819563c28add_Cfunc_AddGo =
unsafe.Pointer(&__cgofn__cgo_819563c28add_Cfunc_AddGo)
```

The `_cgo_819563c28add_Cfunc_AddGo` variable holds the raw address of `__cgofn__cgo_819563c28add_Cfunc_AddGo`. By looking at the link directives (`go:linkname`), we see that this variable will actually be linked to a static symbol (`//go:cgo_import_static`) with name `_cgo_819563c28add_Cfunc_AddGo`. This symbol is associated with a C function definition which can be found in the `main.cgo2.c` file. Before inspecting this function, it's important to say a few words about the second argument of the function call - the address of `p0`. Further in these blog post series, we'll see that the Go compiler is going to construct a so-called **argument frame** onto the stack of the caller of this function (in our example that is `main`) and insert the two arguments contiguously in this frame. Space for storing any return values will also be allocated in it. A nice visual representation of the old calling convention can be found here. This is also known as ABI0 which is the old calling convention in Go (this is also indicated by the `go:cgo_unsafe_args` directive) which used to pass function arguments and store return values on the stack. This can be also confirmed by looking at how the linker annotated the function symbol in the binary:

```
$ go tool nm example3 | grep _Cfunc_AddGo
# Go symbols include the package name in which they are defined
# TEXT symbols only include the ABI version
45a900 t main._Cfunc_AddGo.abi0
...
```

Furthermore, we also see that the arguments of the function are typecasted to `_Ctype_longlong` which is a type wrapper around the Golang type `int64`.

Let's get back to the definition of the `_cgo_819563c28add_Cfunc_AddGo` function which is going to be called by `asmcgocall_landingpad`:

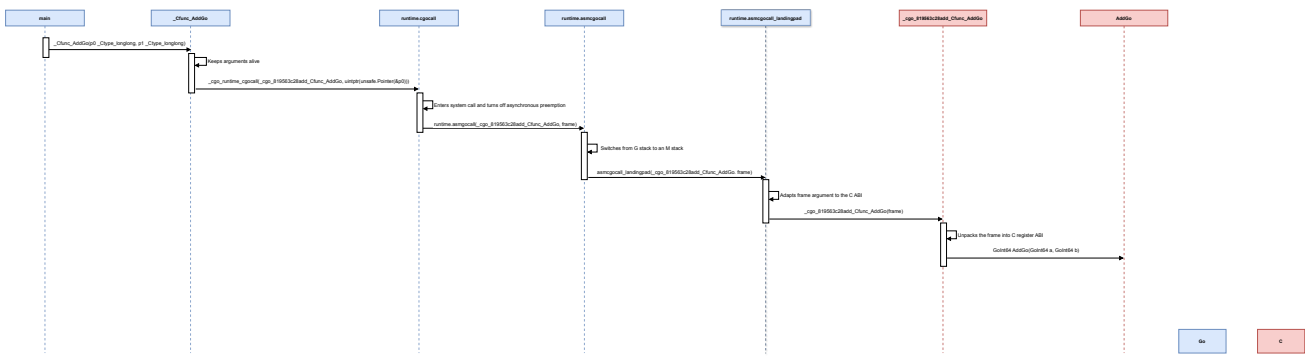
```
#include <stdio.h>
// We should first create a C function declaration which arguments and return
// value types have a Golang equivalent.

extern long long int AddGo(long long int a, long long int b);
#line 1 "cgo-generated-wrapper"

void
_cgo_819563c28add_Cfunc_AddGo(void *v)
{
    struct {
        long long int p0;
        long long int p1;
        long long int r;
    } __attribute__((__packed__, __gcc_struct__)) *_cgo_a = v;
    char *_cgo_stktop = _cgo_topofstack();
    __typeof__(_cgo_a->r) _cgo_r;
    _cgo_tsan_acquire();
    _cgo_r = AddGo(_cgo_a->p0, _cgo_a->p1);
    _cgo_tsan_release();
    _cgo_a = (void*)((char*)_cgo_a + (_cgo_topofstack() - _cgo_stktop));
    _cgo_a->r = _cgo_r;
    _cgo_msan_write(&_cgo_a->r, sizeof(_cgo_a->r));
}
```

By the time the execution flow arrives at the above function, we have exited the Go world and entered the C world - we're executing C code using the System V ABI on a regular OS thread without any runtime! The C code that we wrote in the `example3.go` file can also be seen here! The `_cgo_819563c28add_Cfunc_AddGo` function is going to make use of the above-mentioned argument frame by unpacking it into a structure. This cast actually reveals how the compiler arranged the contents of this frame. It will then do some housekeeping and it will call the `AddGo` function passing the two arguments extracted from the frame. After the control returns to this function, it will store the result back into the frame. The definition of the `AddGo` function is located in the `_cgo_export.c` file. This file contains generated C function wrappers for all Go procedures in a Go project which are annotated with the `export` directive.

But before going into it, here is a visual summary of what happened until now:



Procedure of calling a CGO-generated C interface from a Go function

Let's look into the `AddGo` function:

```
...
CGO_NO_SANITIZE_THREAD
GoInt64 AddGo(GoInt64 a, GoInt64 b)
{
    size_t _cgo_ctxt = _cgo_wait_runtime_init_done();
    typedef struct {
        GoInt64 p0;
        GoInt64 p1;
        GoInt64 r0;
    } __attribute__((__packed__, __gcc_struct__)) _cgo_argtype;
    static _cgo_argtype _cgo_zero;
    _cgo_argtype _cgo_a = _cgo_zero;
    _cgo_a.p0 = a;
    _cgo_a.p1 = b;
    _cgo_tsan_release();
    crosscall2(_cgoexp_819563c28add_AddGo, &_cgo_a, 24, _cgo_ctxt);
    _cgo_tsan_acquire();
    _cgo_release_context(_cgo_ctxt);
    return _cgo_a.r0;
}

extern void _cgoexp_819563c28add_AddGo(void *);
```

The `AddGo` function calls `_cgo_wait_runtime_init_done`. This function will start initialising a new OS-related Go runtime context similar to when a new Golang program is launched. You would probably ask why initialise a new runtime as there is already one from which we came here. Here I'm going to remind you the purpose of CGO and FFI - make Go code callable from non-Go code - code, for which we can't assume to have an existing runtime context. For example, one could compile the above code as a shared (or a static library) and load it into (or link it to) a C code. We're going to dive deeper into this in the next article, so stay tuned! ;)

Happily, the Golang developers handled the case where the C code calls back into the Go context which has triggered it (it's a bit of inception I know, but hang tight). Hence, they implemented a procedure for restoring the initial context before starting to execute the Go function (`main.AddGo`) hidden behind this context-switch adapter.

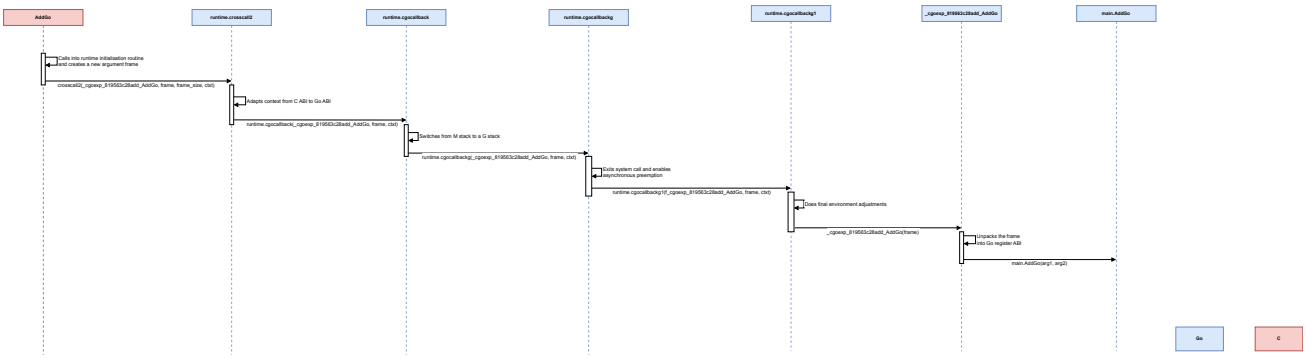
Now, let's get back to the `_cgoexp_819563c28add_AddGo` procedure. A structure type called `_cgo_argtype` is defined and a variable of this type is instantiated. Its fields are populated with the received function arguments. This structure really looks like the previously seen argument frame, doesn't it? It's also going to be used to transmit arguments and store the return values of the actual Go function that we want to call. A call to `runtime.crosscall2`

follows with arguments - `_cgoexp_819563c28add_AddGo`, the `_cgo_argtype _cgo_a`'s address, its size and a context variable. runtime.crosscall2 is an ABI adapter from System V (C) to Go (the reverse operation of `asmcgocall_landingpad`).

The `crosscall2` function then calls into the runtime.cgocallback routine passing as arguments `_cgoexp_819563c28add_AddGo`, `&_cgo_a`, and `_cgo_ctxt`. This routine is going to switch back the current M's stack to the old G's stack. Then a call to runtime.cgocallbackg follows which is going to request execution permission with respect to GOMAXPROCS by calling into runtime.exitsyscall. Finally, when G is allowed to run, asynchronous preemption is again turned on and the function cgocallbackg1 is called with the same arguments as `runtime.cgocallbackg`. The latter will make some final environment adjustments before calling into `_cgoexp_819563c28add_AddGo` passing the frame `&_cgo_a` as an argument.

```
//go:cgo_export_dynamic AddGo
//go:linkname _cgoexp_819563c28add_AddGo _cgoexp_819563c28add_AddGo
//go:cgo_export_static _cgoexp_819563c28add_AddGo
func _cgoexp_819563c28add_AddGo(a *struct {
        p0 int64
        p1 int64
        r0 int64
    }) {
    a.r0 = AddGo(a.p0, a.p1)
}
```

The above procedure is the final step before calling into the actual function defined in our 3rd example. The result (our Go function returns an `int`) is stored back inside the structure passed as an argument.



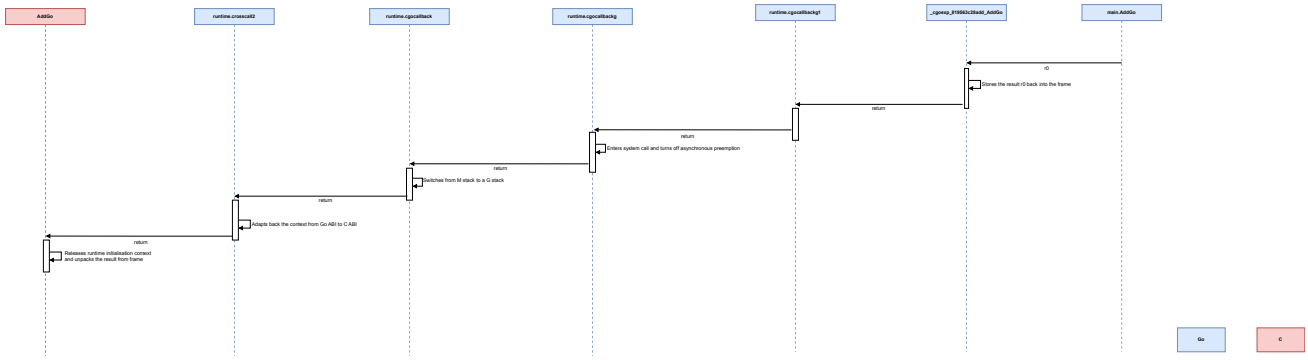
Procedure of calling Go function from a CGO-generated C interface

Now, let's roll back the process: the return value of `main.AddGo` is stored in `a.ro` and the `_cgoexp_819563c28add_AddGo` returns to `cgocallbackg1` which returns to `cgocallbackg`.

The latter will turn off asynchronous preemption again and enter a system call (for the same reason that we described earlier). Once P is moved and M is freed, the execution goes back to `cgocallback` which is going to switch back from G's stack to the previously used

OS thread stack (the one with which we arrived in the C world).

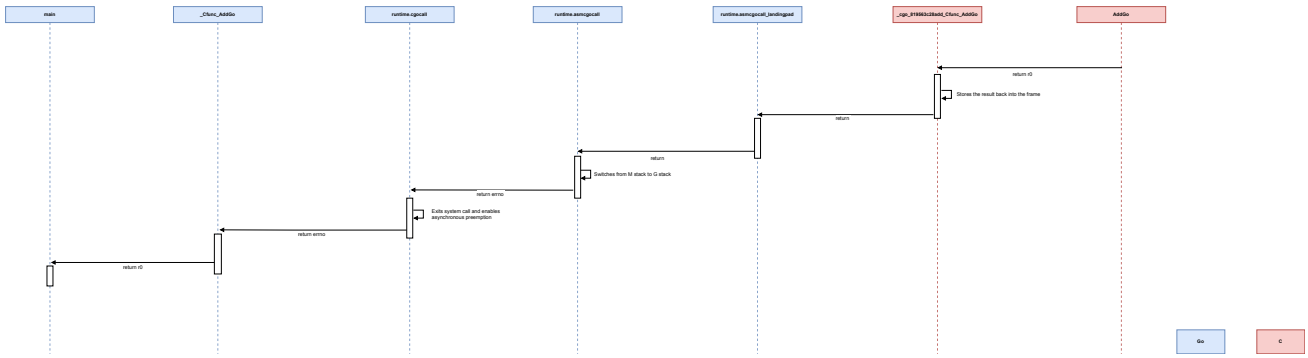
The execution continues back to `crosscall2` and then to `AddGo` which returns `_cgo_a.r0` using the C ABI.



Procedure of returning from a Go function to a CGO-generated C interface

The flow returns to `_cgo_819563c28add_Cfunc_AddGo` where the result is stored back into the frame (`_cgo_a->r`).

The execution goes back to `asmcgocall_landingpad` and then to `asmcgocall` which is going to restore the old G's stack (the one with which we arrived at this call site). Then, the execution returns to `cgocall` which will request to join a P and turn back on preemption. Finally, the flow returns to `_Cfunc_AddGo` and then to `main`.



Procedure of returning from a CGO-generated C interface back to a Go function

Conclusion

As you can see, the FFI process in Golang involves a lot of machinery and interfacing Go code with another language is a difficult and long procedure and this is not all of it... Furthermore, it has a lot of limitations and constraints. The goal of this series of articles is to showcase the research we did in Quarkslab on how to perform hooking of Go programs during runtime. To better and more easily understand our 2nd approach, we considered it

important to first describe how FFI and CGO work under the hood. In the next article, we are going to get our hands dirty and leverage CGO to reimplement the hooks described in the first article! Stay tuned! ;)

Resources

If you would like to learn more about our security audits and explore how we can help you, get in touch with us!