Let’s Go into the rabbit hole (part 1) — the challenges of dynamically hooking Golang programs

blog.quarkslab.com/lets-go-into-the-rabbit-hole-part-1-the-challenges-of-dynamically-hooking-golang-program.html

October 3, 2023

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 behavior dynamically without recompiling their code. Is this due to the complex internals of the language? In this blog post, we’ll look into the challenges of developing and inserting runtime hooks in *Golang* programs.

A Golang gopher

Introduction

Hooking, also known as a “detour”, is a mechanism for unconditionally redirecting the execution flow of a program. There is a lot of literature on the Internet on how this can be done for different programming languages such as C, C++. However, hooking Go code at runtime is not a straightforward process. It gets even more interesting when one tries to hook Go code with Go code which leads to a deep rabbit hole. In the end, it should be more natural to manipulate *Golang* data structures with *Golang*, right? In this series of blog posts, we’ll present a rather interesting strategy that we’ve developed at Quarkslab to achieve that. Before going into the rabbit hole, let’s first discuss why we got interested in implementing detours for Go programs and why it is more complicated than for other programs written in C or C++.

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

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. The way this runtime is placed inside the program alongside its functioning completely changes 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.

But before we demonstrate how we managed to do it, let's first see how we can hook *Golang* programs during runtime using C and pure assembly on *x86-64* CPU architecture. Let's start with a more formal explanation of what a hook is and why it is useful.

What Is Program Hooking and What Is It Useful for?

Hooking a program is the procedure of changing its default flow of execution, most of the time with the intent of either collecting information about the program's environment (e.g. inspect a function's arguments) or with the intent of changing its behavior (e.g. altering the arguments of a function).

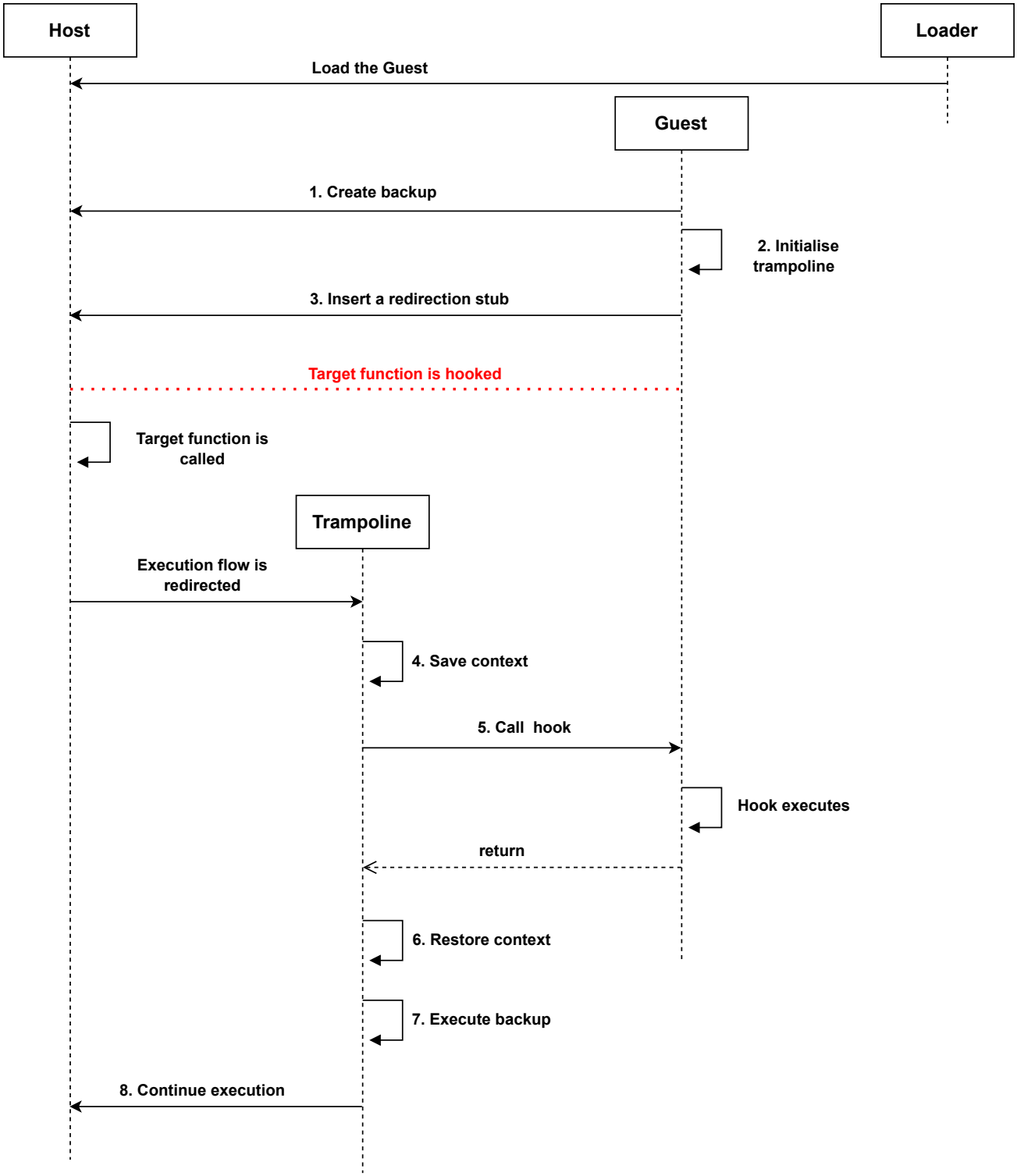
Detours are used for debugging, hot patching, metric collection but also for malware development, game cracks, etc. In general, there are two types of hooks:

- **Regular hooks** — hijack the original execution flow and replace it with an auxiliary logic.
- **Trampoline hooks** — hijack the original execution flow, execute an auxiliary logic and then execute the original flow.

Let's introduce the following notions which will be used in the entire series of blog posts:

- **Host** — the program whose execution flow is going to be hooked.
- **Guest** — an external piece of code to where the execution flow will be redirected.

Here is a simplified graphical representation of how a trampoline hook would work for a regular compiled program:



A scheme of a general program hooking process

The above schema illustrates a redirection of the execution flow of a function in the *Host* to another function in the *Guest*. The hooking happens during execution of the *Host*, hence all the above happens in the RAM, where its instructions are loaded (runtime hooking, right?). In the above schema, it is assumed that the *Guest* is loaded externally, by an auxiliary program that we call "loader", during the execution of the *Host*. You can see the *Guest* as an external object (shared libraries). This approach can be applied to hook any part of a function assuming that the function is not in-lined, or it can at least host a JUMP stub inside it. Let's clarify each phase having a numeric identifier in the above presented schema (the other phases are considered out of the scope of the article or straightforward to understand):

1. **Create backup** — this phase involves saving some instructions from the original function. The insertion of a redirection stub (in phase 2) would overwrite five or fourteen bytes of instructions depending on the size of the stub. To be able to execute the original instructions after the hook, one needs to save these bytes and execute them later. NB: The choice of which instructions to save is important. As these instructions are going to be stored in another segment, if they contain relative offsets, this could involve instruction patching. Another solution would be to overwrite instructions whose execution is independent of their position.

2. **Initialize trampoline** — in this phase the *Guest* initializes the trampoline segment (allocation, initialization of the addresses of the call stubs depending on where the Guest is loaded, insertion of the backup instructions, etc.).

3. **Insert a redirection stub** — in this phase the redirection stub (a conditional/unconditional assembly JUMP instruction) is inserted in the function body, overwriting the original instructions. When the execution flow gets to it, it will be redirected to an external segment containing the "trampoline" logic. This segment is not part of the *Host* so it's created and initialized by the *Guest* when it is loaded.

4. **Save context** — this phase is part of the trampoline segment where the execution flow ends after being redirected. Its objective is to preserve the execution context before calling into the hook function in the *Guest*. The hook could modify the CPU state when executing which could corrupt the program's execution in a future state (remember that the compiler hasn't predicted us interfering). In most programming languages, there are **caller-saved** (volatile registers, or call-clobbered) and **callee-saved** (non-volatile registers, or call-preserved) CPU registers. To not corrupt the program when the execution flow returns to its normal path we need to save the caller-saved registers so that our *Guest* can modify them freely. Additionally, in this phase we are also preparing for a function call which could require an ABI arrangement (adding, rearranging or removing function arguments) by the trampoline itself.

5. **Call hook** — a call instruction redirects the flow to the hook defined in the *Guest*'s `.text` segment.

6. **Restore context** — when the hook returns, in the trampoline section we restore and modify, if necessary (if the hook returns a result), the stored context (CPU registers).

7. **Execute backup** — the saved instructions are executed.

8. **Continue execution** — the flow is redirected to the first instruction after the redirection stub.

In the description above, technical and implementation details are intentionally excluded. Some steps can be simplified and further optimized, but take this as a general approach to create a trampoline hook. Despite this, hooking a Go program makes this procedure significantly more complex.

Hooking Go Using C and Pure Assembly

In this section we'll present a method for how one could create a runtime hook redirecting the execution flow from a Go function to a C function and discuss the limitations of this approach. Let's consider the following *Golang* program:

```
package main

import (
    "fmt"
    "os"
    "strings"
)
import "C"
// the "import C" statement is needed for the compiler to produce a binary
// which is going to load libc.so when launched. This is needed
// for the side-loading of the hook logic.
var SECRET string = "VALIDATEME"

func theGuessingGame(s string) bool {
    if s == SECRET {
        fmt.Println("Authorized")
        return true
    } else {
        fmt.Println("Unauthorised")
        return false
    }

}

func main() {
    var s string

    for {
        if _, err := fmt.Scanf("%s", &s); err != nil {
            panic(err)
        }
        s = strings.ToLower(s)
        if theGuessingGame(s) {
            os.Exit(0)
        }
    }
}
```

The above code receives a user-input string and compares it to a hard-coded value. The problem is that the user could never supply the right string as its input is lowercased and the hard-coded string is in uppercase. To get an “authorized” output we could do the following:

- Skip the call to `strings.ToLower` in `main` and jump directly to the call to `theGuessingGame`.
- When starting to execute `theGuessingGame`, jump directly to the `fmt.Println`code.
- Change the value of the string after it has been lowercased by calling into a hook which does the inverse operation (uppercasing). This can be done either directly after the call to `strings.ToLower` or at the beginning of the `theGuessingGame` function before the actual check is performed.

Even if the first two options are simpler, we'll take the third one as it's the subject of this article. We'll use a trampoline hook so that we can preserve the original execution flow and only change the argument of the function. There is one more interesting thing in the above snippet — the `import C` statement. This will instruct the compiler to add directives for the loader to load `libc` when the binary is loaded in memory. As we said earlier, by default Go binaries are statically linked and include an implementation of the standard library. This is needed for the side-loading the hook logic.

Representing a Golang String in C

If our *Host* program was using C-like strings then our routine in the *Guest* would have the following prototype `void toUpper(char *s);` (a Null terminated sequence of ASCII characters). However, strings in Go are represented differently. In *Golang* strings are seen as a UTF-8 sequence which could contain a Null byte in each and every position. Because of that, in Go, the actual sequence of characters is embedded into a structure alongside its length. The compiler definition of this structure (for Go version 1.20.3) is :

```
// src/internal/unsafeheader/unsafeheader.go:28

// String is the runtime representation of a string.
// It cannot be used safely or portably and its representation may
// change in a later release.
//
// Unlike reflect.StringHeader, its Data field is sufficient to guarantee the
// data it references will not be garbage collected.
type String struct {
    Data unsafe.Pointer
    Len  int
}
```

To define an equivalent structure in C we have to find a meaningful representation of each field:

- The `unsafe.Pointer` type in Go, in this case, can be seen as a `const char *` in C (in general it can be considered as a `void *`).
- The `int` type in Go is equivalent to `ptrdiff_t` (from `<stddef.h>`) in C (in general it can be considered as a `uint64_t`).

Combining the above, we can now represent a Go string in C using the following definition:

```
// hook.h
typedef struct GoString {
    char *p;
    ptrdiff_t n;
} GoString;
```

Now, we can define our `toUpper` routine in C. For the sake of simplicity, we'll assume that the actual byte data is the uppercase ASCII subset of the UTF-8 character set.

```
/*
Convert ASCII string (a-z) to uppercase  (A-Z).
We assume that the byte sequence of the string in str->p contains
only valid uppercase ASCII letters.
*/
void
toUpper(GoString str) {

    char * data = str.p;
    for (int i=0; i<str.n; i++){
        data[i] -= 32;
    }
}
```

Locating the Right Place to Insert the Hook

Now it's time to choose where to hijack the execution flow in the *Host* and redirect it to the *Guest*. We chose the `theGuessingGame` function so let's first compile the code with:

```
$ go build -o secret secret.go
```

We should ensure that the produced binary is dynamically linked and that `libc` is going to be loaded into it. Again — this is necessary for the side-loading of the *Guest*:

```
$ file secret && echo && ldd secret
secret: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=c7ba267d636a05fe5c7438b3cfba76116a26f878, for GNU/Linux 3.2.0, with
debug_info, not stripped

    linux-vdso.so.1 (0x00007fff8eabe000)
    libc.so.6 => /lib64/libc.so.6 (0x00007f11dab6c000)
    /lib64/ld-linux-x86-64.so.2 (0x00007f11dad54000)
```

Let's analyze the assembly code of `main` using GDB and see how `theGuessingGame` function is called (yeah we know that this can be done with *Ghidra*, *Ida* and Co. but we like the output of *GDB* with the peda extension):

```
...
0x0000000000493c15 <+341>:   call   0x48d3a0 <fmt.Fscanf> ; here we read from STDIN
and store the user string
0x0000000000493c1a <+346>:   test   rbx,rbx ; test for errors
0x0000000000493c1d <+349>:   jne    0x493c6a <main.main+426>
0x0000000000493c1f <+351>:   mov    rcx,QWORD PTR [rsp+0x38] ; load the string
structure in RCX
0x0000000000493c24 <+356>:   mov    rax,QWORD PTR [rcx] ; load the pointer to the
byte data in RAX (1st member of the structure)
0x0000000000493c27 <+359>:   mov    rbx,QWORD PTR [rcx+0x8] ; load the size of the
string in RBX (2nd member of the structure)
0x0000000000493c2b <+363>:   call   0x4934a0 <strings.ToLower> ; strings in Go are
immutable so lowercasing
one will create a new structure. RAX contains the pointer to the bytes of the new
string and RBX its size
0x0000000000493c30 <+368>:   mov    rdi,QWORD PTR [rsp+0x38] ; load the pointer to
the original string structure
0x0000000000493c35 <+373>:   mov    QWORD PTR [rdi+0x8],rbx  ; store the new size
# The instructions ensure that if the concurrent garbage collector is running, it's
up to him to update the pointer and update its view of the used heap dat
0x0000000000493c39 <+377>:   cmp    DWORD PTR [rip+0xea280],0x0        ; 0x57dec0
<runtime.writeBarrier>
0x0000000000493c40 <+384>:   jne    0x493c47 <main.main+391>
0x0000000000493c42 <+386>:   mov    QWORD PTR [rdi],rax
0x0000000000493c45 <+389>:   jmp    0x493c4c <main.main+396>
0x0000000000493c47 <+391>:   call   0x45f9c0 <runtime.gcWriteBarrier>
0x0000000000493c4c <+396>:   call   0x4939c0 <main.theGuessingGame> ; call into the
theGuessingGame where RAX holds a pointer to the sequence of UTF-8 data and RBX the
size of this data
...
```

Here we can see the *Golang* ABI. First argument goes in `RAX`, the second one in `RBX`, the third one in `RCX` and so on (more information can be found here).

Let's analyze the beginning of the `theGuessingGame` function before the comparison of the argument string and the hard-coded one takes place:

```
Dump of assembler code for function main.theGuessingGame:
    # The above 2 instructions ensures that the current goroutine stack has enough
place to accomodate the function execution
   0x00000000004939c0 <+0>:     cmp    rsp,QWORD PTR [r14+0x10]
   0x00000000004939c4 <+4>:     jbe    0x493a94 <main.theGuessingGame+212>
   0x00000000004939ca <+10>:    sub    rsp,0x50
   0x00000000004939ce <+14>:    mov    QWORD PTR [rsp+0x48],rbp
   0x00000000004939d3 <+19>:    lea    rbp,[rsp+0x48]
   0x00000000004939d8 <+24>:    mov    QWORD PTR [rsp+0x58], rax
   0x00000000004939dd <+29>:    mov    rdx,QWORD PTR [rip+0xa2f9c]        ; 0x536980
<main.SECRET>  - load into RDX the pointer to the data of the hardcoded string
indentified with main.SECRET+8
   0x00000000004939e4 <+36>:    cmp    QWORD PTR [rip+0xa2f9d],rbx        ; 0x536988
<main.SECRET+8> - compare the size of the parameter string with the size of the
hardcoded string's
   0x00000000004939eb <+43>:    jne    0x4939fc <main.theGuessingGame+60> ; if these
are not equal; no need to compare the actual data bytes
   0x00000000004939ed <+45>:    mov    rcx,rbx ; move the equal size of the two
strings in RCX
   0x00000000004939f0 <+48>:    mov    rbx,rdx ; move the pointer to the hardcoded
string in RBX
   0x00000000004939f3 <+51>:    call   0x4038e0 <runtime.memequal> ; the memory
regions are compared (RAX-> argument pointer to the string's data, RVX -> idem but
for the hardcoded one, rcx the number of bytes to be compared)
   0x00000000004939f8 <+56>:    test   al,al ; if al=0 then the strings are equal
```

We want to hijack the execution flow somewhere before `runtime.memequal`. We'll insert a JUMP stub of 14 bytes, so we should do a backup of at least fourteen instructions when inserting it:

```
push <last-four-bytes-of-destination-address>
move [rsp+4] <first-four-bytes-of-destination-address>
ret
```

We can replace the stack management routine (from `0x04939c0` up to `0x4939ce`) but this region contains a relative to the `RIP` instruction which would require an instructions patching. Another suitable spot is from `0x4939ca` up to `0x4939d8` which is a regular function prologue plus one additional instruction. Backing that up would not require any patching and the instructions can be executed as is even if they are placed at another location. Now it's time to load our hook.

Loading the Guest

To load the *Guest* containing the hook logic inside the *Host* we used a really common technique of side-loading shared libraries into running processes on *Linux* using the `ptrace` API. We're not going to go into detail of how this works as there are plenty of resources on the Internet. We used our own implementation written in Go. However, there are some important aspects which should be pointed out for the side-loading to work:

1. The C hook was compiled as a PIC (Position-Independent Code) using *gcc* with the option `-shared` which produces a shared object.
2. The library loading happens while the target program is running. It's done by attaching to the process using the ptrace API and then calling into `dlopen` which is part of the standard library (`libc`) loaded into the process. The argument of the `dlopen` function is the path to the compiled shared library which was previously written into the memory of the running program again using ptrace.
3. The loader process (the one loading the library into the target program) should be privileged or be owned by the same user as the target process and have the *CAP_SYS_PTRACE* capability.
4. The jump stub insertion logic was compiled as part of the shared library. The insertion is done when the shared library is loaded and its `__constructor__` function is called by the loader.

Inserting the JUMP Stub and Saving the Overwritten Instructions

The insertion of the redirection stub happens when the *Guest* is loaded. The beginning of the `theGuessingGame` function after loading the *Guest* is the following:

```
Dump of assembler code for function main.theGuessingGame:
   0x00000000004939c0 <+0>:     cmp    rsp,QWORD PTR [r14+0x10]
   0x00000000004939c4 <+4>:     jbe    0x493a94 <main.theGuessingGame+212>
   0x00000000004939ca <+10>:    push   0x4b9e4000 ; hohoho this is new
   0x00000000004939cf <+15>:    mov    DWORD PTR [rsp+0x4],0x7fc3 ; and this too
   0x00000000004939d7 <+23>:    ret
   0x00000000004939d8 <+24>:    mov    QWORD PTR [rsp+0x58],rax
```

We see our inserted stub leading to the address `0x7fc34b9e4000`. Let's inspect what is there:

```
0x7fc34b9e4000:      push   r9 ; r9 will be clobbered, so push it onto the stack
0x7fc34b9e4002:      movabs r9,0x7fc34b9e782d ; cloberring r9 with a function
address
0x7fc34b9e400c:      call   r9 ; calling the function;
0x7fc34b9e400f:      pop    r9 ; restore r9 from the stack
0x7fc34b9e4011:      sub    rsp,0x50 ; backup
0x7fc34b9e4015:      mov    QWORD PTR [rsp+0x48],rbp ; backup
0x7fc34b9e401a:      lea    rbp,[rsp+0x48] ; backup
0x7fc34b9e401f:      push   0x4939d8 ; the lower 4 bytes of the address of the next
instruction
0x7fc34b9e4024:      mov    DWORD PTR [rsp+0x4],0x0 ; the upper 4 bytes of the
address of the next instruction
0x7fc34b9e402c:      ret
```

We can see the trampoline section from our scheme. The last part (the execution of the overwritten instructions and the jump to the next instruction) is the same, but the first part is not. The trampoline calls into something located at `0x7fc34b9e782d`. But what's at this

address ? To answer that, let's first talk about the difference between the ABI of Go and C.

Hook Insertion — ABI Switch

Go and C have two different ABIs. Hence, if we want to call into a C function from Go we need to switch the ABI. As of the time of writing, Go uses a register-based ABI. We need to translate it to the C ABI (also known as System V). Here we have only two arguments—the pointer to the bytes of the string (in `RAX`) and its size (in `RBX`).

But how is the ABI of the `toUpper` function arranged in the *Guest*?

```
Dump of assembler code for function toUpper:
   0x00007fada9d711d9 <+0>:     push   rbp
   0x00007fada9d711da <+1>:     mov    rbp,rsp
   0x00007fada9d711dd <+4>:     mov    rax,rdi ; rdi contains the pointer to the
bytes of the Go string
   0x00007fada9d711e0 <+7>:     mov    rcx,rsi ; rsi contains the length of the the
Go string
   0x00007fada9d711e3 <+10>:    mov    rdx,rcx
   0x00007fada9d711e6 <+13>:    mov    QWORD PTR [rbp-0x20],rax ; save the pointer
to the Go string data
   0x00007fada9d711ea <+17>:    mov    QWORD PTR [rbp-0x18],rdx ; save the length of
the Go string data
   0x00007fada9d711ee <+21>:    mov    rax,QWORD PTR [rbp-0x20]
   0x00007fada9d711f2 <+25>:    mov    QWORD PTR [rbp-0x10],rax
   0x00007fada9d711f6 <+29>:    mov    DWORD PTR [rbp-0x4],0x0 ; the i varaible
   0x00007fada9d711fd <+36>:    jmp    0x7fada9d71227 <toUpper+78>
   0x00007fada9d711ff <+38>:    mov    eax,DWORD PTR [rbp-0x4] ; the beginning of
the loop modifying the string
   0x00007fada9d71202 <+41>:    movsxd rdx,eax
   0x00007fada9d71205 <+44>:    mov    rax,QWORD PTR [rbp-0x10]
   0x00007fada9d71209 <+48>:    add    rax,rdx
   0x00007fada9d7120c <+51>:    movzx  eax,BYTE PTR [rax]
   0x00007fada9d7120f <+54>:    lea    ecx,[rax-0x20]
   0x00007fada9d71212 <+57>:    mov    eax,DWORD PTR [rbp-0x4]
   0x00007fada9d71215 <+60>:    movsxd rdx,eax
   0x00007fada9d71218 <+63>:    mov    rax,QWORD PTR [rbp-0x10]
   0x00007fada9d7121c <+67>:    add    rax,rdx
   0x00007fada9d7121f <+70>:    mov    edx,ecx
   0x00007fada9d71221 <+72>:    mov    BYTE PTR [rax],dl
   0x00007fada9d71223 <+74>:    add    DWORD PTR [rbp-0x4],0x1
   0x00007fada9d71227 <+78>:    mov    eax,DWORD PTR [rbp-0x4]
   0x00007fada9d7122a <+81>:    movsxd rdx,eax
   0x00007fada9d7122d <+84>:    mov    rax,QWORD PTR [rbp-0x18] ; get the length of
the Go string
   0x00007fada9d71231 <+88>:    cmp    rdx,rax ; compare it with the i variable
   0x00007fada9d71234 <+91>:    jl     0x7fada9d711ff <toUpper+38> ; jump into the
loop
   0x00007fada9d71236 <+93>:    nop
   0x00007fada9d71237 <+94>:    nop
   0x00007fada9d71238 <+95>:    pop    rbp
   0x00007fada9d71239 <+96>:    ret
```

We can see that the pointer to the Go string data is expected to be in `RDI` while its size is in `RSI`. So the transition that we need to do is simple — `RAX->RDI` and `RBX—>RSI`. This should be done before calling into the C function and after inserting the JUMP stub. This logic can be either located on the heap or as part of the code segment of the shared library. Here is the simple assembly stub performing the ABI switch:

```
ABI_SWITCH:
    mov rdi, rax
    mov rsi, rbx

CALL_C_FUNC:
    mov r9, <address-of-toUpper>
    call r9

ABI_RESTORE:
    ; nothing to be done
```

We are almost there! However, in C there are the notions of callee and caller saved registers. In other words, we should save all register that C code would eventually clobber and restore them before the execution of the overwritten instructions in the trampoline section. In the *System V* ABI these are `RAX, RCX, RDX, RSI, RDI, R8, R9, R10, R11`, so we extend the above logic with the following:

```
SAVE_CTX:
    push rax
    push rcx
    push rdx
    push rdi
    push rsi
    push r8
    push r9
    push r10
    push r11

ABI_SWITCH:
    ...

CALL_C_FUNC:
    ...

ABI_RESTORE:

RESTORE_CTX:
    pop r11
    pop r10
    pop r9
    pop r8
    pop rsi
    pop rdi
    pop rdx
    pop rcx
    pop rax
```

Note: If the hook was returning a result the ABI restore logic should be adapted accordingly. Now if we jump to the `SAVE_CTX` segment, it should be fine? Well, not quite - we could run out of stack space!

Hook insertion — Stack Pivot

In the beginning of the `theGuessingGame` function, we had a prologue of four bytes:

```
0x00000000004939c0 <+0>:     cmp    rsp,QWORD PTR [r14+0x10] ; retrieves the
goroutine structure of the current thread
0x00000000004939c4 <+4>:     jbe    0x493a94 <main.theGuessingGame+212>
...
```

If we follow the jump, we end up here:

```
0x0000000000493a94 <+212>:   mov    QWORD PTR [rsp+0x8],rax ; save the first
argument on the stack
0x0000000000493a99 <+217>:   mov    QWORD PTR [rsp+0x10],rbx ; save the second
argument on the stack
0x0000000000493a9e <+222>:   xchg   ax,ax
0x0000000000493aa0 <+224>:   call   0x45d8c0 <runtime.morestack_noctxt> ; increase
the size of the stack and update its limit in the goroutine structure
0x0000000000493aa5 <+229>:   mov    rax,QWORD PTR [rsp+0x8] ; restore the first
argument
0x0000000000493aaa <+234>:   mov    rbx,QWORD PTR [rsp+0x10] ; restore the second
argument
0x0000000000493aaf <+239>:   jmp    0x4939c0 <main.theGuessingGame> ; continue the
execution from the beginning
```

In Go, the goroutines stacks are resizable and points to the heap. The system stack is used only by some components of the runtime. These instructions are actually verifying the size of the current goroutine stack (R14 contains a pointer to the goroutine structure and at offset `0x10` is located the stack limit called `stackguard`). If there is not enough stack space, the `runtime.morestack_noctxt` is called to increase the stack. In this function the runtime will allocate the correct amount of stack space based on the stack maps (description of the stack space of the current function used to allocate and free memory) inserted by the compiler. The goroutine stacks are small (2Kb). Theoretically, if we hijack the control flow before a stack resizing we could end up with not enough stack to store the registers and execute the hook code which will also use this stack (imagine if `toUpper` function allocated 3000 bytes on its stack). To solve that we could pivot the stack into a new RW region before calling into the C function (and before saving the registers) and restore the old stack afterwards. The allocation of the memory for the new stack is done when loading the *Guest*. Here is the stack pivoting logic:

```
STACK_PIVOT:
    ; save the current G stack in memory
    mov r9, <addr-to-store-g-stack>
    mov [r9], rsp
    ; load the new stack and pivot it (atomic swap)
    mov r9, <addr-new-stack>
    xchg r9, rsp

SAVE_CTX:
    ...
ABI_SWITCH:
    ...
CALL_C_FUNC:
    ...
ABI_RESTORE:

RESTORE_CTX:
    ...
STACK_PIVOT_REV:
    mov r9, <addr-stack-backup>
    xchg rsp, [r9]
    ret
```

For the ones paying close attention, there is a potential flow in this assembly logic. What if the target function in the *Host* was using stack space inferior to eight bytes (even if that's not the case really)? Don't forget that the compiler has not predicted our intervention into the execution flow! Hence, what if pushing R9 overflows the current stack? Don't worry—Go has us covered ;) As we said earlier, the check of the limit is done against a member of the goroutine struct called `stackguard` which can be seen is the bottom of the stack. However, this `stackguard` is not the real stack limit of the goroutine. The Go runtime will allow a certain number of bytes (a constant defined as `StackSmall=128`[bytes]) to protrude beyond this limit (also called spill zone). This tiny space can be used by functions with small or zero sized stack frames which don't need to resize their stacks or perform additional checks (it's also used for optimization). Examples of this kind of function (mostly written in assembly) can be found mostly in the runtime package (the ones annotated with `NOSPLIT`). So theoretically there should be enough space to push the R9 register.

Now everything looks fine, right? It is, and the hook logic will work. Now we know what's at the address 0x7fc34b9e782d in the trampoline section - the address of the STACK_PIVOT stub. However, there is another small problem that could theoretically arise, and we should be ready for it.

Hook Insertion — Mitigating Concurrency Issues

Our toy program is quite simple but, in general, Go programs tend to be highly concurrent. Hence, the above sequence of stubs introduce reentrancy problems — if two goroutines execute the same function and get both redirected, they could end up with inverse stacks!

This scenario would also break our assumptions on the safeties of the execution of C code as the second goroutine could end up using the small stack of the first one. This problem can be illustrated by adding the following code to the existing program:

```
...
s = strings.ToLower(s)
go theGuessingGame(s)
if theGuessingGame(s) {
    os.Exit(0)
}
```

It's not the most interesting example, but it should do the job for you to understand the problem.

To solve this we could use, for example, a simple semaphore introducing a busy wait. This is left as an exercise for the reader.

A working PoC of the above can be found here.

Limitations of the Above Approach

The above approach works on simple programs and sadly is quite architecture and platform dependent. Here are some of the limitations of this approach:

- On Windows the ABI is different, so the above code will not function.
- The used assembly snippets are for x86-64. For other architectures such as ARM or MIPS, the above will not work.
- All Go types and respective offsets have to be manually defined in the C header.
- The above approach introduces heavy concurrency issues.

Why implement runtime hooks written in Golang?

In the beginning we implemented our hooks in C using pure assembly (oh yes, we suffered a lot, but we made it). This was fine for small programs working with primitive data types. After that we started looking to apply our methodology to big projects such as *Docker* and *Containerd*, and then we realized that it was quite difficult and annoying. These programs were using complex data structures, some of which were only available in Go and not in C (e.g. channels, interfaces, slices, etc.). Hence, being able to manipulate these structures in C or assembly was a complex task. So we decided to facilitate our lives as much as possible and write the logic of our hooks in Go. In addition, we wanted to dive deep into the internals of the programming language and explore its true capabilities.

Conclusion

In this blogpost, we illustrated our first effort to define a hooking scheme for Go programs. We explored and understood quite interesting internals of the language while implementing a hook using C and assembly. However, we want to manipulate Go types with more ease. We also want something more universal, something that can be adapted to different platforms and CPU architectures. Happily (or not) the rabbit hole goes deeper ;)

Resources

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