
终端对抗
静态免杀
确保木马文件落地
流量免杀
消除和C2之间的通信流量特征
权限维持
机器重启后确保木马回连
EDR致盲
瘫痪EDR持续监测功能
强行关闭或卸载EDR
内存免杀
木马启动阶段的内存特征
木马运行-休眠周期的内存特征

contents|目录

CONTENTS

/01

回顾：攻防往事

Q：BYOVD技术已经成熟且武器化，且EDR检测能力逐渐完善，为什么还要研究内存检测逃逸相关的技术?

高压环境下的EDR致盲目的不再止步于
“一键卸载 安全卫士/电脑管家”

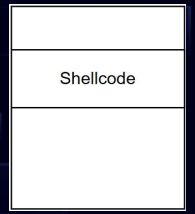
意义何在?

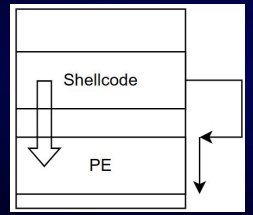
- 有以下几种场景是不能完全依赖BYOVD的，***我们的木马要被迫与完全体EDR共存。***
  - 加载驱动的黑白名单。
  - EDR的R3/R0组件中有类暗桩的存在。
  - 与EDR控制端通信的模块直接做在了R0里面，卸载驱动等同于直接切断通信。
  - 部分场景下，根本没有系统的高权限且无法提权。
  - 机器重启后，被致盲的EDR会恢复正常，权限维持的木马必须有一定存活能力。
  - .....

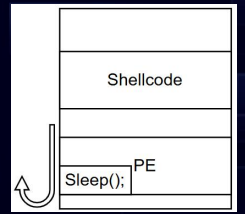
需要做什么

- 两个阶段
  - 木马Loader载入Shellcode并释放植入体的整个初始化过程需要逃逸EDR的检测；
  - 植入体线程在 “运行——休眠” 的过程中长期与EDR共存而不被检测到。

回顾：一个标准Loader的工作原理



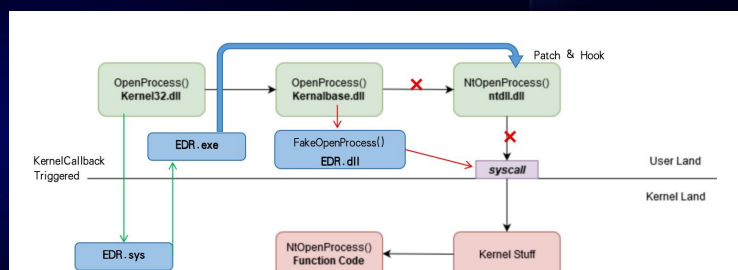




回顾：EDR检测原理

- R3：借助API HOOK拦截敏感函数调用，跟踪参数和返回值。

- R0：利用底层组件监控敏感调用。
  - **内核回调-Windows / 内核探针（Kprobes）-Linux**
  - 内核钩子
  - ETW
  - 硬件辅助
  - .....



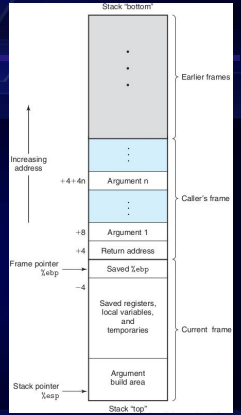
/02

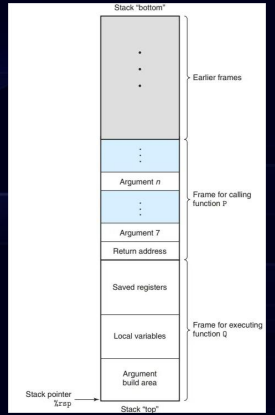
内存检测策略剖析

内存检测

- EDR的内存取证功能可以收集和分析内存中的数据，重建事件的时间线和上下文。包括分析内存中的残留数据、运行时状态和其他关键信息。

  - 关键词1 - 栈回溯：
    - 32位下可以很方便地找到每一个函数的返回地址和栈帧。
    - 64位默认开启了FPO优化，不保存RBP到栈上。
  - 关键词2 - “随机”内存扫描。
    - 标记的RWX内存页、高熵内存页
    - 由栈回溯发现的可疑地址或调用链触发
    - 线程休眠时期
    - .....



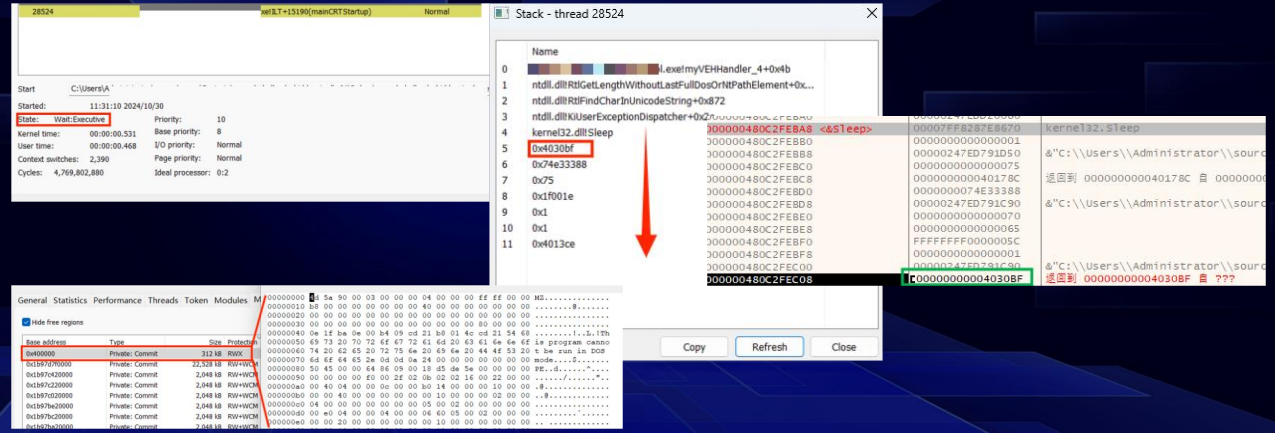



```
push    ebp
mov     ebp, esp
sub     esp, 38h
call    ds:GetTickCount
mov     ecx, 26AAh
```

```
sub     rsp, 68h
call    cs:__imp_GetTickCount
mov     ecx, 26AAh
xor     edx, edx
mov     r9d, 5Ch ; '\'
div     ecx
```

内存检测

一个x64下栈回溯检测的示例



内存检测

两类扫描策略偏好

策略① - 偏向精准检测

- - R3 Hook，初筛敏感API调用；
- - 检测SYSCALL；
- - 触发规则就进行栈回溯；
- - 重点扫描可疑地址对应的内存。

- 策略② - 偏向持续检测
- - 监控线程状态（包含线程的堆栈、运行状态等）；
- - 对私有内存页进行扫描（通常在线程休眠时）；
- - 搜索高熵区域、RWX等区域，重点标记；
- - 对重点标记区提升扫描频率或重点监控该区域的读写、访问行为，直到探测到植入体相关特征。

/03

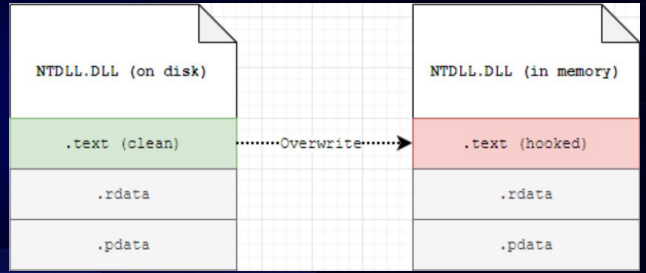
对抗方案

策略①-对抗方案

- 绕过R3 Hook检测
  - SYSCALL
  - Unhook
- 混淆栈回溯检测
  - 栈帧伪造

策略①-对抗方案

- Unhook
  - 方法1：将磁盘上“干净”的dll映射到当前进程中，读取.text节并覆盖被hook的dll的.text节。
  - 方法2：创建一个白名单进程，读取其未被hook的dll，覆盖当前进程中dll的.text节。
  - 方法3：没有白名单程序的情况下，在新进程启动加载完成dll时将其挂起，保留其中“干净”的dll快照，覆盖当前进程dll的.text节。
  - .....



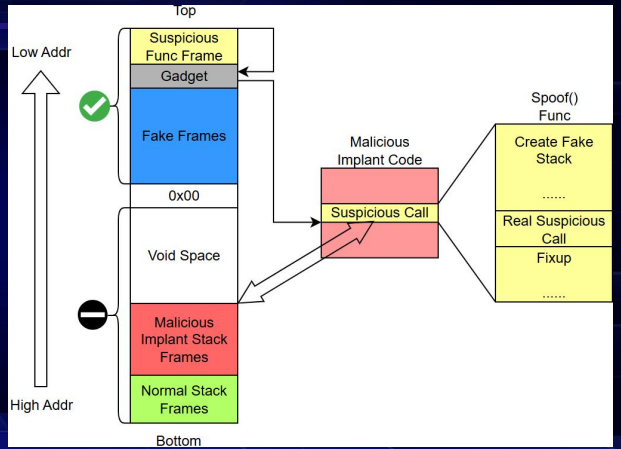
Unhook效果


Unhook_Test.exe - PID: 7528 - 模块: ntdll.dll - 线程: 主线程 5952 - x64dbg [管理员]
文件(F) 视图(V) 调试(D) 跟踪(N) 插件(P) 收藏夹(I) 选项(O) 帮助(H) Mar 8 2024 (TitanEngine)
CPU 日志 笔记 断点 内存布局 调用堆栈 SEH链 脚本
00007FF92543F955 CD 2E int 2E
00007FF92543F957 C3 ret
00007FF92543F960 E9 530D1EBC jmp 7FF8E16206B8
00007FF92543F965 CC int3
00007FF92543F966 CC int3
00007FF92543F967 CC int3
00007FF92543F968 F60425 0803FE7F 01 test byte ptr ds:[7FFE0308],1
00007FF92543F970 75 03 jne ntdll.7FF92543F975
00007FF92543F972 0F05 syscall
00007FF92543F974 C3 ret
00007FF92543F975 CD 2E int 2E
00007FF92543F977 C3 ret
00007FF92543F978 0F1F8400 00000000 nop dword ptr ds:[rax+rax],eax
00007FF92543F980 4C:8BD1 mov r10,rcx
00007FF92543F983 B8 19000000 mov eax,19
00007FF92543F988 F60425 0803FE7F 01 test byte ptr ds:[7FFE0308],1
00007FF92543F990 75 03 jne ntdll.7FF92543F995
00007FF92543F992 0F05 syscall

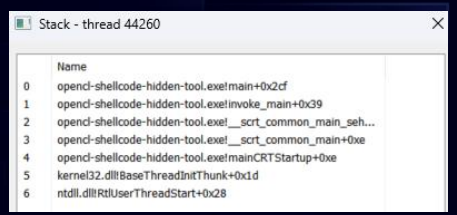

Unhook_Test.exe - PID: 7528 - 模块: ntdll.dll - 线程: 主线程 5952 - x64dbg [管理员]
文件(F) 视图(V) 调试(D) 跟踪(N) 插件(P) 收藏夹(I) 选项(O) 帮助(H) Mar 8 2024 (Tit
CPU 日志 笔记 断点 内存布局 调用堆栈 SEH链
00007FF92543F955 CD 2E int 2E
00007FF92543F957 C3 ret
00007FF92543F960 4C:8BD1 mov r10,rcx
00007FF92543F963 B8 18000000 mov eax,18
00007FF92543F968 F60425 0803FE7F 01 test byte ptr ds:[7FFE0308],1
00007FF92543F970 75 03 jne ntdll.7FF92543F975
00007FF92543F972 0F05 syscall
00007FF92543F974 C3 ret
00007FF92543F975 CD 2E int 2E
00007FF92543F977 C3 ret
00007FF92543F978 0F1F8400 00000000 nop dword ptr ds:[rax+rax],eax
00007FF92543F980 4C:8BD1 mov r10,rcx
00007FF92543F983 B8 19000000 mov eax,19
00007FF92543F988 F60425 0803FE7F 01 test byte ptr ds:[7FFE0308],1
00007FF92543F990 75 03 jne ntdll.7FF92543F995
00007FF92543F992 0F05 syscall

策略①-对抗方案

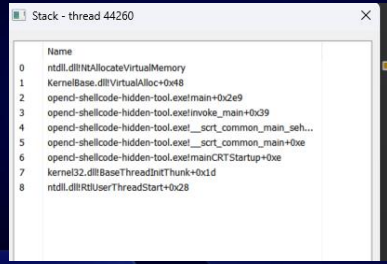
- 栈帧伪造
  - 1. 重写函数；
  - 2. 隐藏原始栈帧；
  - 3. 部署一个假栈；
  - 4. 部署一个Gadget Frame；
  - 5. SYSCALL；
  - 6. 利用Gadget修复栈帧并返回。

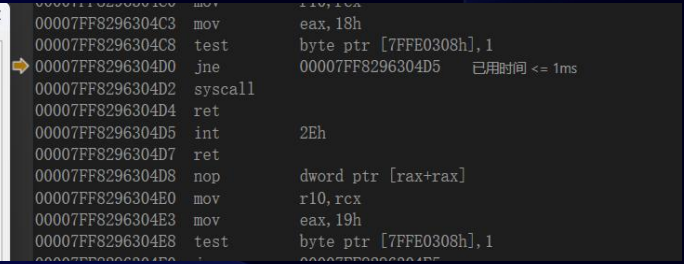


未伪造栈帧效果

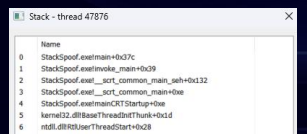


进入
NtAllocateVirtualMemory()

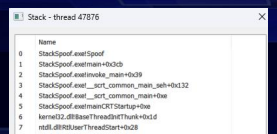




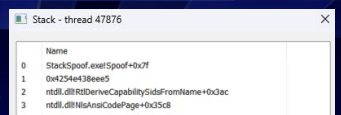
栈帧伪造效果



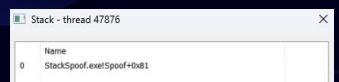
进入Spoof()



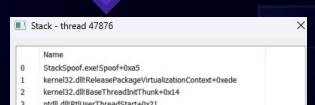
抬栈

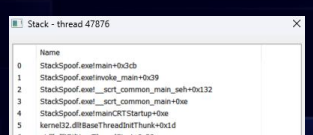


PUSH 0



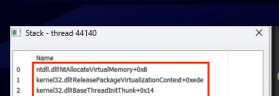
伪造假栈+Gadget

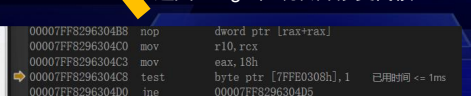




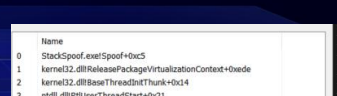
跳转回原位

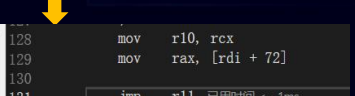
返回Gadget，跳转并修复栈帧



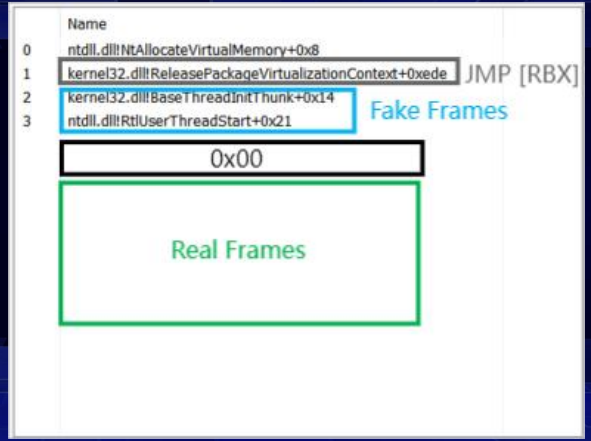


真实函数调用





栈帧伪造效果


Name
0
ntdll.dll!NtAllocateVirtualMemory+0x8
1
kernel32.dll!ReleasePackageVirtualizationContext+0xede
JMP [RBX]
2
kernel32.dll!BaseThreadInitThunk+0x14
3
ntdll.dll!RtlUserThreadStart+0x21
Fake Frames
0x00
Real Frames

策略②-对抗方案

- 防止反射加载器特征被扫描识别
- 降低探测到植入体休眠指令相关返回地址的概率
- 植入体“隐身”

- 参考案例-1：https://github.com/mgeeky/ThreadStackSpoofer/tree/master；其利用hook Sleep()截断栈帧，以对抗线程休眠期间的栈回溯探测。
  - Inline Hook ✕
- 参考案例-2：https://github.com/vxunderground/VXUG-Papers/blob/main/GpuMemoryAbuse.cpp；利用CUDA将内存转储至VRAM。
  - CUDA API，跨平台✕

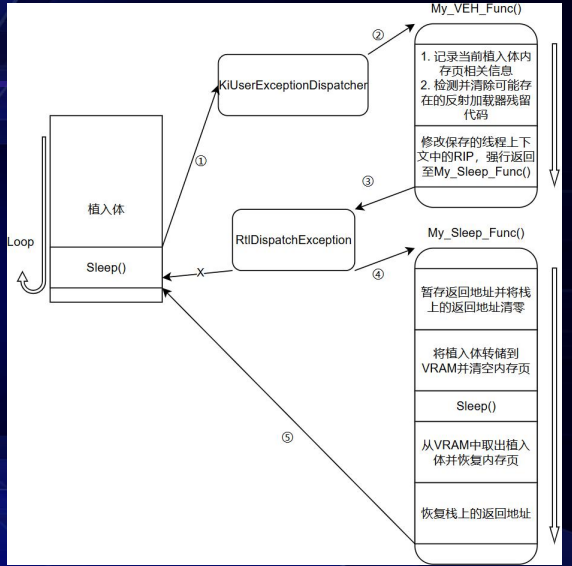
准备工作

- “无内存特征”Hook
  - 给Sleep()设置硬件断点，等待植入体执行后调用；
  - 自定义一个VEH处理函数并设置最高优先级，承接硬件断点；
  - 准备一个MySleep()方法，实现核心功能。

- VRAM内存转储
  - 对于没有安装任何GPU驱动程序的环境，如无显卡的服务器或未配置虚拟显卡的虚拟机，要安装CPU设备的专用OpenCL Runtime（msi安装包可使用命令一键安装，无需GUI操作）；
  - 在Loader中利用OpenCL的SDK创建一个全局对象，用来存储OpenCL缓冲区句柄和其他临时数据。

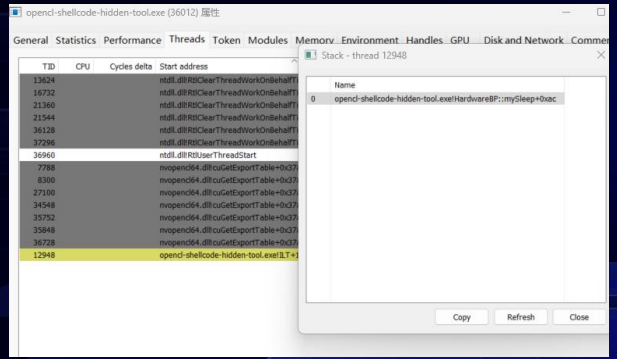
VRAM内存转储

- 工作原理
  - 0. 准备工作：创建全局OpenCL内存对象，并设置Sleep()的硬件断点，等待程序调用Sleep()；
  - 1. 在自定义VEH中记录当前植入体内存页相关信息并将反射加载器残留内存页释放；
  - 2. 修改VEH暂存的线程上下文结构体中的RIP，引导其返回到自定义Sleep()方法；
  - 3. 在自定义Sleep()方法中将可疑内存页写入OpenCL内存对象的缓冲区，通过OpenCL库写入VRAM；
  - 4. 在自定义Sleep()方法中关闭内存页X权限；
  - 5. 在自定义Sleep()方法中暂存返回地址并将真实返回地址覆盖为0x00，截断栈帧；
  - 6. 在自定义Sleep()方法中调用真正的Sleep()；
  - 7. 在自定义Sleep()方法返回前恢复内存页。
    - 访问OpenCL内存对象的缓冲区，取出之前转储的数据；
    - 重新开启内存页X权限；
    - 恢复暂存的返回地址。
  - 8. 正常返回到植入体代码的CALL Sleep()下一条指令位置，进入下一循环周期。

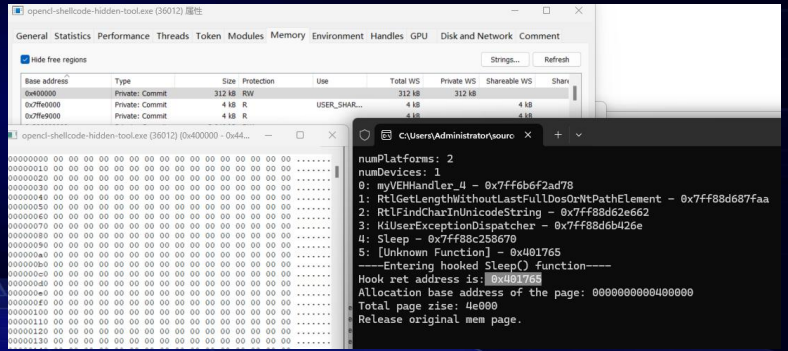


效果对比

休眠线程堆栈

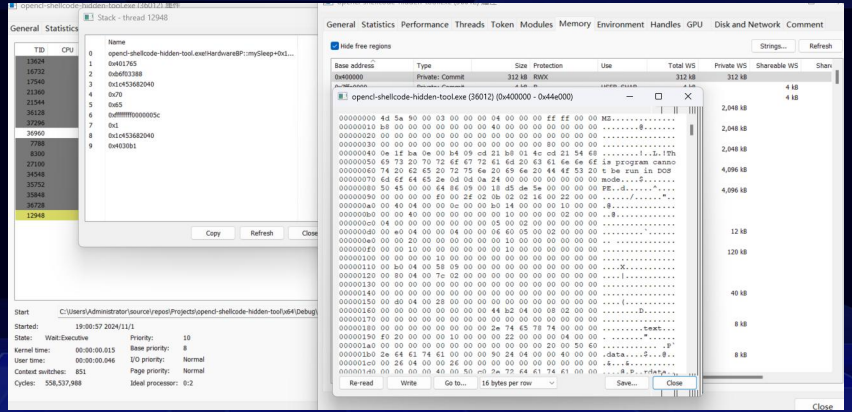


线程休眠时内存

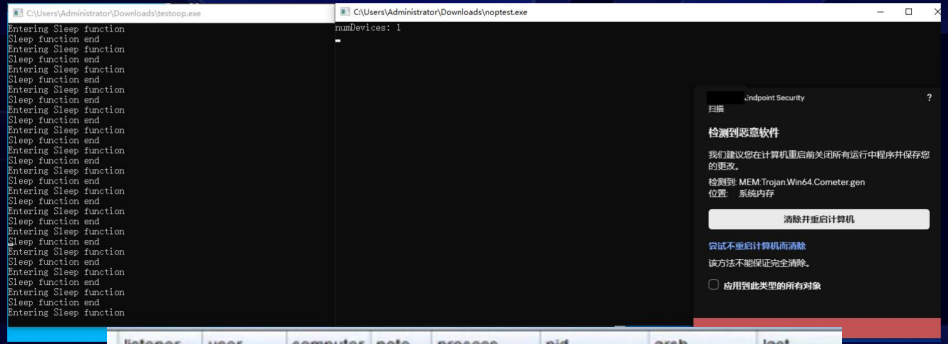


效果对比

未使用该方案的休眠堆栈和内存



效果对比



listener	user	computer	note	process	pid	arch	last
HTTPS	Adminis...	DESKT...		*testoop.exe	4100	x64	379ms
HTTPS	Adminis...	DESKT...		*noptest.exe	8124	x64	14h

THANKS