转自看雪——Hackshield内幕（thisIs）

原创

o330820350 于 2013-04-01 16:23:45 发布 1371 收藏

最近讨论游戏保护的帖子又见增多, 正好我以前逆过HS, 抠出点东西来跟大家分享下
今天也是我接触驱动, 逆向这一块整整一周年, 写个帖子聊表纪念

论坛上这方面帖子大多都是写的如何跳过驱动层的hook点来达到过保护的目的
再聊这些东西也没什么意思, 我就爆料一些HS内部的处理的细节吧

这里面有很多很好的安全编程方法值得我们借鉴和学习, 大家有时间的话可以拖到IDA中看一下,
比起写个驱动在内核跳来跳去, 学习到一些实用的编程技巧对自身的成长是大有益处的

写这篇帖子的时候我的电脑中并没有装带有HS保护的游戏,
所以下面讨论的内容全部根据以前逆向时留下的笔记和回忆来总结的,
所以具体的细节上可能与实际有所出入
这里的讨论的HS版本全部都是去年11月份左右的版本, 应该是5.5.xxx
而我最后一次玩某游戏好像版本有了小幅度的更新

HS也是通过一系列的内核HOOK来达到保护游戏目的的, 主要有以下这些<部分函数只是存在于驱动中, 实际并没有被使用>:

SSDT hook:
1.NtReadVirtualMemory
2.NtWriteVirtualMemory
3.NtOpenProcess
4.NtSuspendProcess
5.NtTerminateThread

引用到的结构如下:

```
// SSDT hook 函数相关结构
typedef struct _SSDTHookInfo{
  ULONG   SSDTNAME;   // 函数名称, 对应一个加密后的字串地址
  ULONG   SSDTINDEX;   // 函数索引
  ULONG   ProxyAddress;   // 代理函数地址
  ULONG   PreValue;   // hook前地址
  ULONG   PrevBeenHooked;  // 该函数是否被别人hook
  ULONG   someflag;   // 标志
}SSDTHookInfo;  //+0x18大小
```

SSDT Inline hook:

```
1  KeAttachProcess      //4
2  KiAttachProcess      //4
3  NtDeviceIoControlFile   //4
4  NtClose         //4
5  IopXxxControlFile     //4
6  NtOpenProcess      //4
7  NtReadProcessMemory     //4
8  NtWriteProcessMemory   //4
9  PsSuspendThread      //4
10  NtTerminateThread     //4
11  unknow
12  NtSetContextThread     //4
13  NtGetContextThread     //4
14  NtQueryPerformanceCounter  //2
15  NtGetContextThread_End    //4
```

引用到的结构如下:

```
// inline hook 函数相关的结构
```

```
typedef struct _SSDTInlineHookInfo{
  ULONG     FunctionName;     //函数名字串指针
  ULONG     FunctionRVA;     //+0X04   函数距离模块基址RVA
  ULONG     HookOffsetFromFunction; //+0X08   HOOK处距离函数头偏移
  ULONG     HookAddress;     //+0X0C   HOOK线性地址
  ULONG     JmpCodeAddress;     //+0X10   JmpCodeBuffer [0x18大小], 要跳转到的中间跳缓冲地址
  BOOLEAN     ProxySwitchOn;     //+0X14   过滤是否启用
  BOOLEAN     unknow;
  LARGE_INTEGER  OriginalData;     //+0X18   原始值
  ULONG     Flag;       //+0X20   一些标志
  ULONG     unknow;       //+0X24
}SSDTInlineHookInfo; //size = 0x28
```

Shadow SSDT Hook:

```
0  NtUserQueryWindow   0x0
1  NtUserBuildHwndList   0x0
2  NtUserFindWindowEx   0x0
3  NtUserGetForegroundWindow 0x0
4  NtUserWindowFromPoint  0x0
5  NtGdiBitBlt      0x20000
6  NtGdiStretchBlt     0x20000
7  NtGdiMaskBlt     0x20000
8  NtGdiPlgBlt      0x20000
9  NtGdiTransparentBlt   0x20000
10  NtUserSendInput     0x20
11  NtUserGetDC      0x20000
12  NtUserGetDCEx     0x20000
13  NtUserGetWindowDC   0x20000
```

引用到的结构与SSDT Hook一样

Shadow SSDT Inline Hook:

```
NtGdiGetPixel   0x20000
unknown      0x20 <未使用>
```

引用到的结构如下:

```
typedef struct _ShadowSSDTInlineHookInfo{
  ULONG   FunctionName:   // 函数名称字串指针
  ULONG   PreAddress;     // 函数源地址
  PVOID   JmpCodeAddress;   // 跳转指令的地址 这里的内存是分配的大小0x18 从这里跳到代理函数
  BYTE   IsHookInstalled;  // Hook是否已经安装
  BYTE   UnknowA;     // +0xD
  USHORT   Fill;      // 间隙填充
  UCHAR   OldValue[8];   // 原始值
  ULONG   Flag;      // 一些标志
  ULONG   UnknowB;     // +0x1C
}ShadowSSDTInlineHookInfo;
```

涉及到Hook的结构就只有上面这些, hs在驱动内部小心地安装或卸载Hook然后配合对各种名单的过滤来实现对游戏的保护
对于名单这一块由于不少操作相关的函数都被vm掉, 所以只逆出来了很少一部分, 主要有一下几个名单:

```
// 进程白名单
typedef struct _ProcessWhiteList{
  ListEntry ProcList;  // 白名单链
  ULONG   Flag;   // +0X08 标志<大多数情况下有4就代表存在>
  ULONG   UnknowA;  // +0X0C 这里是个枚举 0 1 2
  PEPROCESS  eprocess;  // +0x10 进程环境体
  PWCHAR   pProcName[20];  // +0x14 进程名

  ..
  UCHAR       // +56 是否打开过游戏进程标志
}ProcessWhiteList;
```

这个是个白名单的列表,列出了很多进程名称,如果有对应的存在 EPROCESS 域不是空的
这个结构数组是在初始化的时候就写定了, 全部内容如下:
进程白名单:

1. Name: patcher.exe
 Flag: 00080004
 Flag: 2
 +56 :

2. Name: WerFault.exe
 Flag: 00080004
 Flag: 0
 +56 :

3. Name: IAANTmon.exe
 Flag: 00080000
 Flag: 2
 +56 :

4. Name: avp.exe
 Flag: 00080000
 Flag: 2
 +56 :

5. Name: WmiApSrv.exe
 Flag: 00080000
 Flag: 2
 +56 :

6. Name: xsync.exe
 Flag: 00080000
 Flag: 2
 +56 :

7. Name: fssm32.exe
 Flag: 00080000
 Flag: 2
 +56 :

8. Name: LGDCORE.exe
 Flag: 00000020
 Flag: 2
 +56 :

9. Name: ACS.EXE
 Flag: 00080000
 Flag: 2
 +56 :

10.Name: ITPYE.EXE
 Flag: 00000020
 Flag: 2
 +56 :

11.Name: Joy2Key.exe
 Flag: 00000020
 Flag: 2
 +56 :

12.Name: JOYTOKEYHIDE.EXE
 Flag: 00000020
 Flag: 2
 +56 :

13.Name: JOYTOKEYKR.EXE
 Flag: 00000020
 Flag: 2
 +56 :

14.Name: JOYTOKEY.EXE
Flag: 00000020
Flag: 2
+56 :

15.Name: DWM.EXE
Flag: 00080000
Flag: 0
+56 :

16.Name: WMIPRVSE.EXE
Flag: 00080000
Flag: 2
+56 :

17.Name: DK2.EXE
Flag: 00080000
Flag: 2
+56 :

18.Name: CSTRIKE-ONLINE.EXE
Flag: 00080000
Flag: 2
+56 :

19.Name: RAGII.EXE
Flag: 00080000
Flag: 2
+56 :

20.Name: EKRN.EXE
Flag: 00080000
Flag: 2
+56 :

21.Name: GOM.EXE
Flag: 00080000
Flag: 2
+56 :

22.Name: GAMEMON.DES
Flag: 00080000
Flag: 2
+56 :

23.Name: VAIOCAMERACAPTUREUTILITY.EXE
Flag: 00080000
Flag: 2
+56 :

24.Name: IPOINT.EXE
Flag: 00000020
Flag: 2
+56 :

25.Name: NMCOSRV.EXE
Flag: 00080004
Flag: 2
+56 :

26.Name: DEKARON.EXE
Flag: 00080000
Flag: 2
+56 :

27.Name: AUDIODG.EXE
Flag: 00080000
Flag: 0

+56 :

28.Name: NGM.EXE
Flag: 00080004
Flag: 2
+56 :

29.Name: TASKMGR.EXE
Flag: 00080004
Flag: 0
+56 :

30.Name: HGSCRAPEDITORHELPER.EXE
Flag: 00080004
Flag: 2
+56 :

31.Name: SETPOINT.EXE
Flag: 00000020
Flag: 2
+56 :

32.Name: NMSERVICE.EXE
Flag: 00080004
Flag: 2
+56 :

33.Name: NSVCAPPFLT.EXE
Flag: 00080000
Flag: 2
+56 :

34.Name: UPSHIFTMSGR.EXE
Flag: 00080004
Flag: 2
+56 :

35.Name: NOD32KRN.EXE
Flag: 00080000
Flag: 2
+56 :

36.Name: IMJPCMNT.EXE
Flag: 00080000
Flag: 2
+56 :

37.Name: TCSERVER.EXE
Flag: 00080000
Flag: 2
+56 :

38.Name: SPOOLSV.EXE
Flag: 00080000
Flag: 0
+56 :

39.Name: IEXPLORE.EXE
Flag: 000A0024
Flag: 2
+56 :

40.Name: EXPLORER.EXE
Flag: 000A0024
Flag: 1
+56 :

41.Name: WINLOGON.EXE

Flag: 000A0024
Flag: 0
+56 :

42.Name: SERVICES.EXE
Flag: 00080024
Flag: 0
+56 :

43.Name: CSRSS.EXE
Flag: 000A0024
Flag: 0
+56 :

44.Name: LSASS.EXE
Flag: 00080024
Flag: 0
+56 :

45.Name: SVCHOST.EXE
Flag: 00080024
Flag: 0
+56 : 1

// 进程黑名单
typedef struct _ProcessBlackList{
ListEntry list; // 进程链
ULONG ProcessId; // +0x08
ULONG Flag; // +0x0c
}ProcessBlackList;
这里的标记:4应该是代表有效 0代表无效
进程销毁之后在这里会断链

// 句柄名单
句柄名单是有三个的, 应该都是使用了同一种结构
typedef struct _HandleTable{
ListEntry list;
ULONG ProcessId; // +0x0x8
Handle FileHandle; // +0x0C
}HandleTable;

句柄名单应该是两个白名单一个黑名单, 不过内部没有找到很多对于这三个名单的引用,
最直接的是在NtDeviceIoControlFile相关的Hook流程上用来动态添加名单,
顺便说一下NtDeviceIoControlFile中是对ARK进行过滤检测用的<或者说这是一部分功能>,

具体的做法是根据NtDeviceIoControlFile的文件句柄取得文件对象, 进而得到设备对象, 驱动对象,
然后根据驱动对象就得到目标驱动模块的DispatchDeviceIoControl接口, 然后通过一些算法在一个名单中进行匹配,
依此来进行过滤

// 驱动名单
这个结构与一个全局的结构数组相关
该全局数组一共0x19大小,保存了部分驱动名称,初始化的时候填写这部分内容
typedef struct _DriverTable{
ULONG Flag: // 标志 一般都是4
DRIVEROBJECT DriverObject; // 驱动对应的驱动对象地址 使用到的时候才会更新
ULONG Index; // 索引序号
PWCHAR DriverName[0x40]; // 驱动名称
}DriverTable; // size = 0x4C
这个名单在未被VM的代码中没有见到被引用

// 下面这个是最重要的内部结构, 不过很多域都没有看到被引用, 可能是被VM掉了, 遗憾
typedef struct _Important{
ListEntry list;

```
 FileObject   fileobj;   //+0x08
 EPROCESS    eprocess;    //+0x0C 游戏主进程
 ETHREAD     ethread;    //+0x10
 ULONG          //+0X14
 ULONG          //+0x18
 ULONG          //+0x1C
 ULONG          //+0x20 这里是一个标记  本进程是否还存在有效
 ULONG    flag;    //+0x24 这里是个标记
 ULONG          //+0x28
 ULONG          //+0x2C
 LARGE_INTEGER CpuCycleCount; //+0x30 cpu周期计数, 用作反调试
 ULONG    ProcId;    //+0x38 主进程ID

 HANDLE    hThreads[3F]; //+0X3C  从这里到 0x130都是线程ID 保存了一些游戏主进程的线程ID
 ..
 ULONG          //+0x130
 PWCHA          //+0x13C

 PIRP     Irp;    //+0x15C
 ..          //+0x160 貌似是个字串指针
 ..          //+x0164
 PIRP     Irp:    //+0x168
 ..          //+0x16C
 ..          //+0x170
 PIRP     Irp;    //+0x174
}Important;
```

还有几个与游戏线程相关的名单, 不过不怎么重要, 就不列出了

//**********

下面说一下HS的过滤, 以进程相关的操作为例 open read write, 过滤步骤如下:

```
 if (当前进程在黑名单中)
 {
  if (操作目标是要保护进程)
  {
   if (是游戏进程发起的操作)
   {
    从黑名单中移除自己;
    放行;
   }
   else
   {
    拒绝访问;
   }
  }
  else
  {
   放行
  }
 }
 else
 {
  if (操作目标是要保护进程)
  {
   if (是游戏进程发起的操作)
   {
    放行;
   }
   else
   {
    if (当前进程在白名单中)
    {
     放行
    }
```

```
    else
    {
     加入黑名单;
     拒绝访问;
    }
   }
  }
  else
  {
   放行
  }
 }
```

乍一看过滤方式够严密, 不过具体实施上却暴露出了一个大的漏洞,
因为在这一系列的操作中它都是进程ID来判断过滤的, 呵呵, 这样PID+1 +2 +3 就能简单地在三层绕过它了

当然这只针对NtOpenProcess可行, read write函数是传递的是进程句柄,
HS内部通过ZwQueryInformationProcess通过句柄得到进程ID来进行过滤

所以我们还是将自身进程添加到白名单中比较稳妥, 有了上面的白名单结构信息,
我们只需要随便跟踪hs的一个代理函数并找到名单的ListHead就OK了

另外一个重点是反调试这边了, 涉及到这方面的内核处理有:
KiAttachProcess inline Hook
NtSetContextThread inline hook
NtGetContextThread inline hook
PsSuspendThread inline hook <win7>
dpc例程

RING3层:
DbgBreakPoint 指令修改
DbgUiRemoteBreakin 指令修改

我所了解到的只有这些, 也可能还有别的地方, 内核inline Hook的几个函数大家都心知肚明, 不必多说
dpc例程主要用来获取CMOS中的时钟信息, CPU的时钟周期数, 然后进行一些运算, 应该是用来反内核调试用
关于dpc, 我的另一片帖子中有介绍: http://bbs.pediy.com/showthread.php?t=148135

另外RING3层的两个函数的修改也是很猥琐的, 我们在用OD附加到游戏上时会创建一个远程线程,
这个线程就是为了执行DbgBreakPoint而存在, 原来的DbgBreakPoint指令为 int 3 , retn
也就是执行一个中断指令以中断到调试器, 现在已经被修改为 retn, retn 直接返回了,
这样调试器就接收不到中断信号

另外一个DbgUiRemoteBreakin, 是执行DbgBreakPoint前的一个调用, 具体的作用我也没仔细研究,
不过看名称也是中断用的, hs会将这个函数头写入一个jmp, 直接跳转到LdrShutdownProcess,
LdrShutdownProcess将直接给每个已经加载的dll模块发消息, dll将被卸载
这样就导致了OD一加载, dll等模块就被卸载掉了

关于KiAttachProcess NtSetContextThread NtGetContextThread PsSuspendThread如何绕过,
这个是仁者见仁智者见智的了, 不过这几个代理函数中都有这样的指令序列:

最后的jz指令就跳转到原函数了, 呵呵, 这里就不言自明了吧

```
//*********************************************************************
```

hs内部的一套Hook框架很不错, 当初让我学到了很多东西, 受益匪浅, 在这里也跟大家分享下,
<这框架是我在hs框架基础上完善的>

```
// 用于索引
typedef enum _SSDTNameIndex{
 ssdtZwCreateFile,
 ssdtZwOpenFile,
 MAX_SSDT_HOOK_ITEM_COUNT
}SSDTNameIndex;

#define SERVICE_NAME_LEN  64
typedef struct _SSDTHookInfo{
```

```
  ULONG        ServiceIndex;        // ssdt索引
  hintSSDTNameIndex   Number;           // 序号
  PVOID        RealAddress;        // ssdt真实地址
  PVOID        PrevAddress;        // hook ssdt之前地址
  PVOID        ProxyAddress;        // 代理地址
  ULONG        ProxyBusyNow;        // 正忙标志
  BOOLEAN        IsHookInstalled;      // 是否安装了hook
  BOOLEAN        IsInUses;          // 是否启用这个hook
  WCHAR        ServiceName[SERVICE_NAME_LEN]; // ssdt名称
}SSDTHookInfo, *PSSDTHookInfo, *PSSDTHOOKINFO;

NTSTATUS CollectSSDTInfo()
{
  在这里初始化结构数组, 获取函数索引, 解析PE得到其真实地址等等
}


// 安装Hook
NTSTATUS SetupAllSSDTHook()
{
  ..
  for ( i = 0; i < MAX_SSDT_HOOK_ITEM_COUNT; i++ )   // 遍历结构数组
  {
    pEntry = (PSSDTHookInfo)&g_SSDTHookItems[i];
    if ( !pEntry ) continue;
    if ( !pEntry->IsInUses ) continue;        // 根据此标志启动这个函数Hook
    if ( pEntry->IsHookInstalled ) continue;     // is reinstall?
    if ( !pEntry->RealAddress ) continue;
    if ( pEntry->ServiceIndex > KeServiceDescriptorTable->ulNumberOfServices ) continue;

    ..进行Hook

    pEntry->PrevAddress = OldAddress;
    pEntru->IsHookInstalled = TRUE;
    ...
  }
}



// 代理例程
NTSTATUS Proxy_ZwOpenFile()
{
  ..
  pEntry = (PSSDTHookInfo)&g_SSDTHookItems[ssdtZwCreateFile];   // 取得对应项

  InterlockedIncrement(&pEntry->ProxyBusyNow);

  ..过滤

  InterlockedDecrement(&pEntry->ProxyBusyNow);
  return status;
}


// 恢复Hook
NTSTATUS RestoreAllSSDTHook()
{
  ..

  for ( i = 0; i < MAX_SSDT_HOOK_ITEM_COUNT; i++ )
  {
    pEntry = (PSSDTHookInfo)&g_SSDTHookItems[i];
    if ( !pEntry ) continue;
    if ( !pEntry->IsInUses ) continue;
    if ( !pEntry->IsHookInstalled ) continue;
    if ( !pEntry->RealAddress ) continue;
```

```
    ..取当前SSDT地址
    if ( CurrentAddress == pEntry->ProxyAddress )
    {
      // 没有被别人恢复时, 恢复Hook
      if ( MmIsAddressValid(pEntry->PrevAddress) )
      {
        ..使用Hook之前地址恢复
      }
      else
      {
        ..使用真实地址恢复
      }
    }
  }

  // 在这里等待所有代理函数都处理完毕 正常退出

  for ( i = 0; i < MAX_SSDT_HOOK_ITEM_COUNT; i++ )
  {
    pEntry = (PSSDTHookInfo)&g_SSDTHookItems[i];
    if ( !pEntry->IsInUses ) continue;

    if ( pEntry->ProxyBusyNow > 0 )
    {
      DueTime.QuadPart = 0;
      KeInitializeTimerEx(&timer, SynchronizationTimer);
      KeSetTimerEx(&timer, DueTime, 100, NULL);

      for ( j = 0; j < 20; j++ )
      {
        KeWaitForSingleObject(&timer, Executive, KernelMode, FALSE, NULL);
        if ( (LONG)pEntry->ProxyBusyNow <= 0 ) break;
      }
      KeCancelTimer(&timer);
    }

    pEntry->PrevAddress = NULL;
    pEntry->IsHookInstalled = FALSE;
    pEntry->ProxyBusyNow = 0;
  }
}
```

inline hook的框架总结起来比较麻烦了, 大家就去看一下IDA数据库自己总结下吧~~

hs.rar.