# TsFltMgr.sys分析报告

该驱动为qq管家函数过滤驱动，提供SSDT、SSSDT、进程和线程回调等过滤操作，导出接口给TsKsp.sys使用，2者共同做函数过滤操作，TsFltMgr提供设置函数过滤的框架，而实际拦截过程在TsKsp中。设备名[\\Device\\TsFltMgr](file:///\\Device\\TsFltMgr) ，符号名[\\DosDevices\\TsFltMgr](file:///\\DosDevices\\TsFltMgr) 。加密手段：Rabbit算法、MD5算法。通过InlineHook KifastCallEntry实现挂钩。

目录

[TsFltMgr.sys分析报告 1](#_Toc433297723)

[一、 驱动入口DriverEntry 2](#_Toc433297724)

[1.1 过滤模型 3](#_Toc433297725)

[1.2 检查当前系统是否为默认挂钩系统 3](#_Toc433297726)

[1.3 打开TsFltMgr日志记录 4](#_Toc433297727)

[1.4 控制信息 4](#_Toc433297728)

[1.5 全局表 4](#_Toc433297729)

[1.6 Proxy\*函数模型 19](#_Toc433297730)

[二、 驱动接口Interface 22](#_Toc433297731)

[2.1 DeviceExtension接口 22](#_Toc433297732)

[2.2 SetEvaluateTime 22](#_Toc433297733)

[2.3 SetDisablePrevFilter 23](#_Toc433297734)

[2.4 SetPostFilter 23](#_Toc433297735)

[2.5 ExecOriginFromPacket 23](#_Toc433297736)

[2.6 AddPrevFilter 23](#_Toc433297737)

[2.7 RemovePrevFilter 24](#_Toc433297738)

[2.8 GetCurrentHookInfo 25](#_Toc433297739)

[2.9 GetDProxyTable 26](#_Toc433297740)

[三、 基础库 27](#_Toc433297741)

[3.1 获取注册表键值 27](#_Toc433297742)

[3.2 通过进程名获取进程ID 29](#_Toc433297743)

[3.3 Rabbit加密算法 29](#_Toc433297744)

[四、 InlineHook KiFastCallEntry 35](#_Toc433297745)

[4.1 获取SSDT/SSSDT/Hook点 36](#_Toc433297746)

[4.2 从KiSystemService获取KiFastCallEntry 39](#_Toc433297747)

[4.3 获取SSSDT信息 39](#_Toc433297748)

[4.4 初始化InlineHook KiFastCallEntry跳转表 41](#_Toc433297749)

[4.5 获取系统服务号的2种方式 44](#_Toc433297750)

[4.6 InlineHook过程 45](#_Toc433297751)

[4.7 构造InlineHook跳转后的执行语句 48](#_Toc433297752)

[4.8 强制单核互斥执行指定Procedure 51](#_Toc433297753)

[4.9 进行SSDT hook 52](#_Toc433297754)

[4.10 对重要回调(进程回调、线程回调、映像加载回调)的挂钩 55](#_Toc433297755)

[4.11 Hook KeUserModeCallback 57](#_Toc433297756)

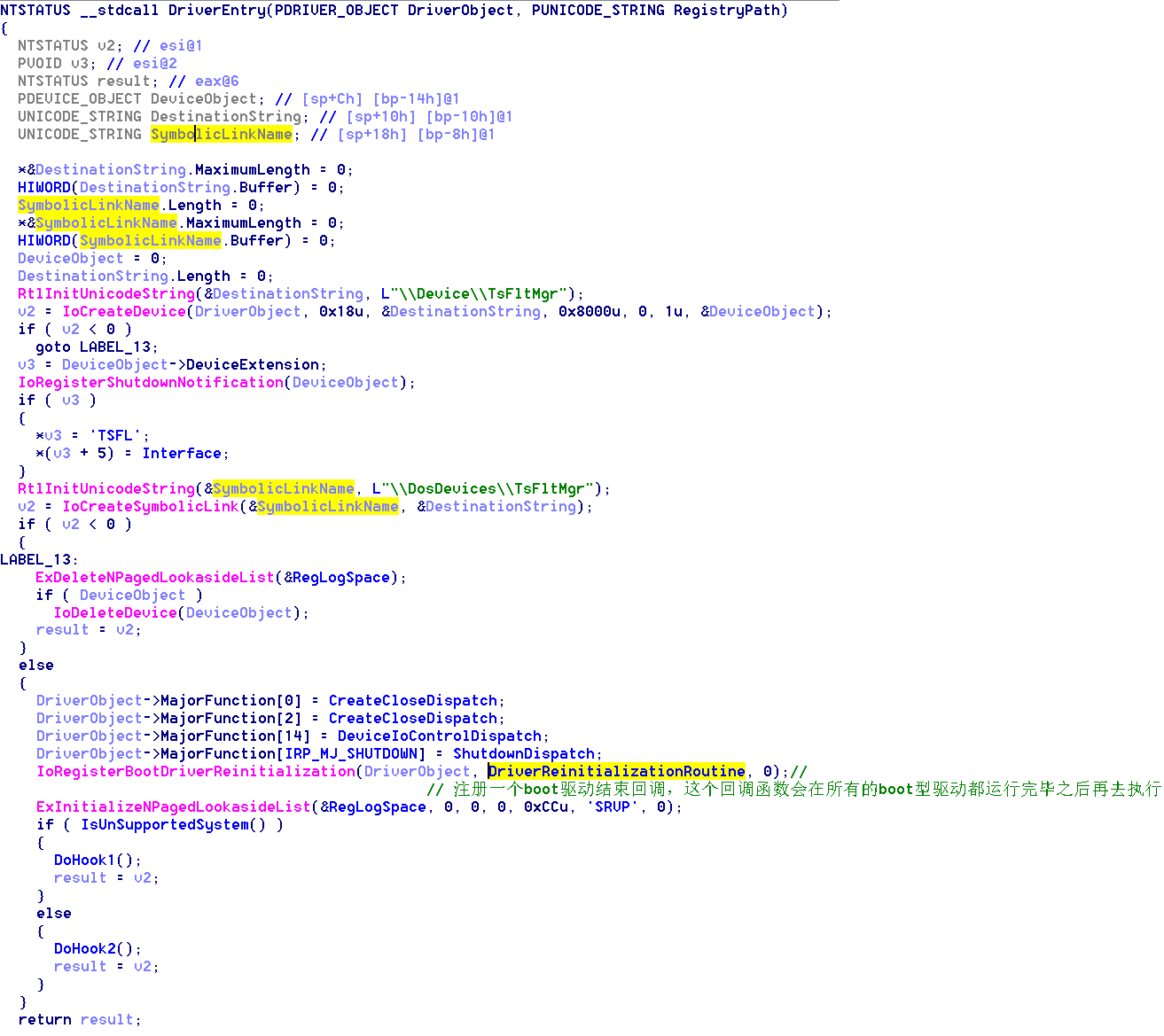
[4.12 交换内存 58](#_Toc433297757)

[4.13 获取函数Iat偏移 58](#_Toc433297758)

[4.14 另一种方式获取ShadowSSDT信息 59](#_Toc433297759)

## 驱动入口DriverEntry

* 创建[\\Device\\TSSysKit](file:///\\Device\\TSSysKit)设备和[\\DosDevices\\TSSysKit](file:///\\DosDevices\\TSSysKit)符号链接
* 设置DeviceExtension为通信接口（Interface函数指针）
* 分别注册IRP\_MJ\_CREATE、IRP\_MJ\_CLOSE、IRP\_MJ\_DEVICE\_CONTROL、IRP\_MJ\_SHUTDOWN(关机回调)派遣例程为，CreateCloseDispatch、DeviceIoControlDispatch、ShutdownDispatch
* 注册”Boot驱动加载结束”回调DriverReinitializationRoutine
* 为注册表日志记录分配资源RegLogSpace
* 检查当前系统是否为注册表version键指定的系统，如果在列表中则在挂钩KiFastCallEntry时需要做额外工作
* 设置注册表键IsBsod为1，用于检测该驱动是否引起蓝屏(正常关机置0)
* 获取系统BuildNumber
* 分配和设置”内核Api代理”结构
* 挂钩KiFastCallEntry
* 挂钩重要回调
* 启动注册表日志记录
* 挂钩KeUserModeCallback
* 记录当前配置



### 1.1 过滤模型

DProxyTable

NtCreateFile:

PrevFilterFunc[0]

PrevFilterFunc[1]

PrevFilterFunc[2]

...........

TsKsp.sys

HookPacket

HookPacket

TsFltMgr.sys

NtCreateFile

Filter

Proxy

NtCreateFile

HookPacket

PostSlotFunc

原始NtCreateFile

HookPacket

PostFilterFunc[0]

PostFilterFunc[1]

PostFilterFunc[2]

.........

* 1. Ntdll.NtCreateFile通过Sysenter调用进入nt.KiFastCallEntry
  2. 在nt.KiFastCallEntry 执行call ebx(原始为nt.NtCreateFile)前跳到TsFltMgr. InlineKiFastCallEntry
  3. 执行进入TsFltMgr.HookFilter，在这里通过ServiceMapTable表映射到对应Dproxy元素，将Dproxy->ProxyNtCreateFile设置到ebx，将其设置为ebx
  4. Nt.KiFastCallEntry执行call ebx，进入ProxyNtCreateFile
  5. 构造FilterPacket结构（用于承载参数、原始api和PostFilterFunc执行的所有过滤函数都用到），依次执行Dproxy->PrevFilterSlot的16个过滤函数（PrevFilter是Tsksp事先设置好的）
  6. 依次执行单个Tsksp.PrevFilter，进行真正的过滤或对packet. PostFilterSlot进行设置
  7. 返回TsFltMgr.ProxyNtCreateFile，执行nt.NtCreateFile
  8. 执行packet. PostFilterSlot的16个过滤函数(Tsksp)
  9. 返回nt.KiFastCallEntry

### 1.2 检查当前系统是否为默认挂钩系统

BOOLEAN IsUnSupportedSystem()

{

/\*注：\\Registry\\Machine\\SYSTEM\\CurrentControlSet\\Services\\TSKS version

存放没有预存 函数调用号ServiceIndex 的系统版本列表 格式：

BuildNumber1;BuildNumber2;...

对于这些版本在进行SSDT Hook时，会临时取得服务号

\*/

NTSTATUS status;

ULONG BuildNumber = 0,MajorVersion,MinorVersion;

const int BufSize = 1024;

ULONG Size,Type;

WCHAR BuildNumberStr[10] = {0};

BOOLEAN Match = FALSE;

UNICODE\_STRING UBuildNumber;

WCHAR\* Buffer = (WCHAR\*)ExAllocatePool(NonPagedPool,BufSize);

status = GetRegDataWithSizeAndType(L"\\Registry\\Machine\\SYSTEM\\CurrentControlSet\\Services\\TSKSP",L"version",

Buffer,BufSize,&Size,&Type);

if(NT\_SUCCESS(status) && Type == REG\_SZ && Size)

{

Buffer[510] = 0;

RtlInitUnicodeString(&UBuildNumber,BuildNumberStr);

PsGetVersion(&MajorVersion,&MinorVersion,&BuildNumber,NULL);

RtlIntegerToUnicodeString(BuildNumber,10,&UBuildNumber);

if(wcsstr((wchar\_t\*)Buffer,UBuildNumber.Buffer))

Match = TRUE;

}

ExFreePool(Buffer);

return Match;

}

### 1.3 打开TsFltMgr日志记录

在无保护情况下为[\\REGISTRY\\MACHINE\\SYSTEM\\CurrentControlSet\\Services\\TsFltMgr](file:///\\REGISTRY\\MACHINE\\SYSTEM\\CurrentControlSet\\Services\\TsFltMgr)添加TsDbgLog键，内容设置为目标文件路径（例如\??\C:\TsDbgLog.txt），如果不存在会自动创建文件，重启生效。内容示例：

[0x00000000] 2015.09.27 20:05:24.109 TS TsFltMgr DbgHelper

[0x00000001] 2015.09.27 20:06:13.750 [Sysnap DbgLog] Block--> TableIndex 0, Process spoolsv.exe[1800]

[0x00000002] 2015.09.27 20:10:35.156 [Sysnap DbgLog] Block--> TableIndex 4, Process regedit.exe[2296]

[0x00000003] 2015.09.27 20:13:46.500 [Sysnap DbgLog] Block--> TableIndex 4, Process regedit.exe[2296]

DriverReinitializationRoutine中做初始化，此时最后一个boot驱动初始化完毕

在执行KiFastCallEntry hook时再次尝试启动打印日志线程

ExecPrevSlotFunc中，如果存在过滤函数进行了放行和拦截，都会打印日志

### 1.4 控制信息

禁止hook

[\\REGISTRY\\MACHINE\\SYSTEM\\CurrentControlSet\\Services\\TsFltMgr](file:///\\REGISTRY\\MACHINE\\SYSTEM\\CurrentControlSet\\Services\\TsFltMgr) dws=1

[\\REGISTRY\\MACHINE\\SYSTEM\\CurrentControlSet\\services\\QQSysMon\\DWS](file:///\\REGISTRY\\MACHINE\\SYSTEM\\CurrentControlSet\\services\\QQSysMon\\DWS) dws!=0

强制SSDT hook

[\\REGISTRY\\MACHINE\\SYSTEM\\CurrentControlSet\\Services\\TsFltMgr](file:///\\REGISTRY\\MACHINE\\SYSTEM\\CurrentControlSet\\Services\\TsFltMgr) thm=1

关机回调

设置[\\REGISTRY\\MACHINE\\SYSTEM\\CurrentControlSet\\Services\\TsFltMgr](file:///\\REGISTRY\\MACHINE\\SYSTEM\\CurrentControlSet\\Services\\TsFltMgr) IsBsod=0，以便下次启动检测是否TsFltMgr引起蓝屏

### 1.5 全局表

系统Build号与全局表索引对应关系

BuildNumber:

Win2000

2195 1

WinXp

2600 2

WinServer2003

3790 3

WinVista

6000 4

6001 5

6002 6

Win7

7600 7

7601 8

Win8

8102 9

8250 10

8400 11

8432 12

8441 12

8520 13

Win8.1

9200 14

9600 15

Win10

9841 16

9860 17

9926 18

10041 19

10049 20

未知 0

enum

{

WIN2000=1,

WINXP,

WINXPSP3,

WINVISTA,

WINVISTASP1,

WINVISTASP2,

WIN7,

WIN7SP1,

WIN8\_8102,

WIN8\_8250,

WIN8\_8400,

WIN8\_8432,

WIN8\_8441=WIN8\_8432,

WIN8\_8520,

WIN81\_9200,

WIN81\_9600,

WIN10\_9841,

WIN10\_9860,

WIN10\_9926,

WIN10\_10041,

WIN10\_10049,

BUILDMAX,

};

enum

{

SSDT=0,

SSSDT=1,

END=2,

CALLBACK=3,

};

#define APINUMBER 105

struct SProxy

{

ULONG ServiceTableType;//0:SSDT 1:Shadow SSDT 2:结束符

PWCHAR ApiName;//函数名

ULONG ProxyFunc;//代理函数地址

ULONG ServiceIndex[BUILDMAX];

ULONG IndexInTable;//在全局表中的索引

};

struct DProxy

{

ULONG ServiceTableType;//0:SSDT 1:Shadow SSDT 2:结束符 3:回调函数

ULONG ServiceIndex;//服务号

PWCHAR ApiName;//函数名

ULONG TableIndex;//自定义序号

BOOLEAN IsInitialized;

ULONG PrevFilterRefCount;//引用计数

ULONG PostFilterRefCount;//引用计数

ULONG OriginFuncAddr;//原始函数地址

ULONG ProxyFuncAddr;//代理函数地址

PVOID Log;//用于记录日志

KEVENT Lock;

BOOLEAN DisablePrevFilter;//关闭Filter

ULONG UsedSlotCount;// 当前使用的Slot个数

FILTER\_SLOT PrevFilterSlot[16];//过滤函数结构

};

struct FILTER\_SLOT

{

ULONG Tag;

ULONG CallCount;

ULONG DeleteCount;

KTIMER Timer;

ULONG Filter;

};

struct FilterPacket

{

ULONG CurrentSlot;//当前Filter序号

ULONG ParamNumber;//参数个数

ULONG Params[12];//参数

ULONG TagSlot[16];//标志过滤函数用，也可用于传递修改参数

NTSTATUS Status;//执行结果

ULONG OriginFuncAddr;//原始函数

ULONG IndexInTable;//在DProxyTable中的索引

ULONG Access;//访问权限

ULONG PostFilterSlot[16];//过滤函数

ULONG UsedSlotCount;//当前使用的Slot个数

};

TsFltMgr有3张表与函数过滤相关:

静态Api代理表SProxy SProxyTable[APINUMBER+1] 用于初始化后面2个表

动态Api代理表DProxy\* DProxyTable[APINUMBER+1] 用于Proxy\*函数中进行实际过滤操作 方便用SProxy指定的序号配置

DProxy\* ServiceMapTable[2][1024] 用于InlineHook KiFastCallEntry改变ebx，映射ServiceIndex到Proxy\*函数。函数前1024个用于存储SSDT函数，后1024用于存储SSSDT函数

可以用简单的python命令自动获取到g\_ProxyApiTable内容

addr=0x25200

index=0

while index < 106:

if Dword(addr) == 0:

type="SSDT"

elif Dword(addr) == 1:

type="SSSDT"

else:

type="END"

ApiName=GetString(Dword(addr+4),-1,ASCSTR\_UNICODE)

ProxyFunc="Proxy"+ApiName

print "{\n\t%s,L\"%s\",%s,\n\t{" %(type,ApiName,ProxyFunc)

print "\t\t%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d" %(Dword(addr+12),Dword(addr+16),Dword(addr+20),Dword(addr+24),Dword(addr+28),Dword(addr+32),Dword(addr+36),Dword(addr+40),Dword(addr+44),Dword(addr+48),Dword(addr+52),Dword(addr+56),Dword(addr+60),Dword(addr+64),Dword(addr+68),Dword(addr+72),Dword(addr+76),Dword(addr+80),Dword(addr+84),Dword(addr+88),Dword(addr+92))

print "\t},%d\n}," %(Dword(addr+96))

addr=addr+100

index=index+1

struct SProxy SProxyTable[APINUMBER+1] =

{

{

SSDT,L"ZwCreateKey",ProxyZwCreateKey,

{

1023,35,41,43,64,64,64,70,70,347,351,351,350,350,350,354,355,355,356,359,359

},0

},

{

SSDT,L"ZwTerminateProcess",ProxyZwTerminateProcess,

{

1023,224,257,266,338,334,334,370,370,35,35,35,35,35,35,35,36,36,36,36,36

},1

},

{

SSDT,L"ZwSetInformationFile",ProxyZwSetInformationFile,

{

1023,194,224,233,305,301,301,329,329,78,79,79,78,78,78,81,82,82,82,82,82

},2

},

{

SSDT,L"ZwWriteFile",ProxyZwWriteFile,

{

1023,237,274,284,359,355,355,396,396,4,5,5,5,5,5,6,7,7,7,7,7

},3

},

{

SSDT,L"ZwSetValueKey",ProxyZwSetValueKey,

{

1023,215,247,256,328,324,324,358,358,48,48,48,48,48,48,49,50,50,50,50,50

},4

},

{

SSDT,L"ZwWriteVirtualMemory",ProxyZwWriteVirtualMemory,

{

1023,240,277,287,362,358,358,399,399,1,2,2,2,2,2,3,4,4,4,4,4

},5

},

{

SSDT,L"ZwCreateFile",ProxyZwCreateFile,

{

1023,32,37,39,60,60,60,66,66,351,356,356,355,355,355,360,361,361,362,365,365

},6

},

{

SSDT,L"ZwOpenProcess",ProxyZwOpenProcess,

{

1023,106,122,128,194,194,194,190,190,220,222,222,221,221,221,224,225,225,226,227,227

},7

},

{

SSDT,L"ZwDeleteKey",ProxyZwDeleteKey,

{

1023,53,63,66,123,123,123,103,103,310,314,314,313,313,313,317,318,318,319,321,321

},8

},

{

SSDT,L"ZwDeleteValueKey",ProxyZwDeleteValueKey,

{

1023,55,65,68,126,126,126,106,106,307,311,311,310,310,310,314,315,315,316,318,318

},9

},

{

SSDT,L"ZwRequestWaitReplyPort",ProxyZwRequestWaitReplyPort,

{

1023,176,200,208,275,276,276,299,299,108,110,110,109,109,109,112,113,113,114,114,114

},10

},

{

SSDT,L"ZwQueryValueKey",ProxyZwQueryValueKey,

{

1023,155,177,185,252,252,252,266,266,143,145,145,144,144,144,147,148,148,149,149,149

},11

},

{

SSDT,L"ZwEnumerateValueKey",ProxyZwEnumerateValueKey,

{

1023,61,73,77,136,136,136,119,119,292,296,296,295,295,295,299,300,300,301,303,303

},12

},

{

SSDT,L"ZwCreateThread",ProxyZwCreateThread,

{

1023,46,53,55,78,78,78,87,87,330,334,334,333,333,333,337,338,338,339,342,342

},13

},

{

SSDT,L"ZwDuplicateObject",ProxyZwDuplicateObject,

{

1023,58,68,71,129,129,129,111,111,300,304,304,303,303,303,307,308,308,309,311,311

},14

},

{

SSDT,L"ZwLoadDriver",ProxyZwLoadDriver,

{

1023,85,97,101,165,165,165,155,155,255,257,257,256,256,256,259,260,260,261,263,263

},15

},

{

SSDT,L"ZwDeviceIoControlFile",ProxyZwDeviceIoControlFile,

{

1023,56,66,69,127,127,127,107,107,304,308,308,307,307,307,311,312,312,313,315,315

},16

},

{

SSDT,L"ZwAlpcSendWaitReceivePort",ProxyZwAlpcSendWaitReceivePort,

{

1023,1023,1023,1023,38,38,38,39,39,381,386,386,385,385,385,390,391,391,393,396,396

},17

},

{

SSDT,L"ZwSetSystemInformation",ProxyZwSetSystemInformation,

{

1023,208,240,249,321,317,317,350,350,56,56,56,56,56,56,57,58,58,58,58,58

},18

},

{

SSDT,L"ZwDeleteFile",ProxyZwDeleteFile,

{

1023,52,62,65,122,122,122,102,102,311,315,315,314,314,314,318,319,319,320,322,322

},19

},

{

SSDT,L"ZwOpenSection",ProxyZwOpenSection,

{

1023,108,125,131,197,197,197,194,194,216,218,218,217,217,217,220,221,221,222,222,222

},20

},

{

SSDT,L"ZwCreateSection",ProxyZwCreateSection,

{

1023,43,50,52,75,75,75,84,84,333,337,337,336,336,336,340,341,341,342,345,345

},21

},

{

SSDT,L"ZwSuspendThread",ProxyZwSuspendThread,

{

1023,221,254,263,335,331,331,367,367,38,38,38,38,38,38,38,39,39,39,39,39

},22

},

{

SSDT,L"ZwTerminateThread",ProxyZwTerminateThread,

{

1023,225,258,267,339,335,335,371,371,34,34,34,34,34,34,34,35,35,35,35,35

},23

},

{

SSDT,L"ZwSystemDebugControl",ProxyZwSystemDebugControl,

{

1023,222,255,264,336,332,332,368,368,37,37,37,37,37,37,37,38,38,38,38,38

},24

},

{

SSDT,L"ZwProtectVirtualMemory",ProxyZwProtectVirtualMemory,

{

1023,1023,137,143,210,210,210,215,215,194,196,196,195,195,195,198,199,199,200,200,200

},38

},

{

SSDT,L"ZwCreateSymbolicLinkObject",ProxyZwCreateSymbolicLinkObject,

{

1023,45,52,54,77,77,77,86,86,331,335,335,334,334,334,338,339,339,340,343,343

},39

},

{

SSDT,L"ZwSetContextThread",ProxyZwSetContextThread,

{

1023,1023,213,221,293,289,289,316,316,91,92,92,91,91,91,94,95,95,95,95,95

},40

},

{

SSDT,L"ZwRenameKey",ProxyZwRenameKey,

{

1023,1023,192,200,267,267,267,290,290,117,119,119,118,118,118,121,122,122,123,123,123

},41

},

{

SSDT,L"ZwOpenThread",ProxyZwOpenThread,

{

1023,111,128,134,201,201,201,198,198,214,214,214,213,213,213,216,217,217,218,218,218

},42

},

{

SSDT,L"ZwGetNextThread",ProxyZwGetNextThread,

{

1023,1023,1023,1023,372,368,368,140,140,271,271,271,270,270,270,273,274,274,275,277,277

},43

},

{

SSDT,L"ZwCreateThreadEx",ProxyZwCreateThreadEx,

{

1023,1023,1023,1023,388,382,382,88,88,333,333,333,332,332,332,336,337,337,338,341,341

},44

},

{

SSDT,L"ZwRestoreKey",ProxyZwRestoreKey,

{

1023,1023,204,212,279,280,280,302,302,105,107,107,106,106,106,109,110,110,111,111,111

},55

},

{

SSDT,L"ZwReplaceKey",ProxyZwReplaceKey,

{

1023,1023,193,201,268,268,268,292,292,115,117,117,116,116,116,119,120,120,121,121,121

},56

},

{

SSDT,L"ZwGetNextProcess",ProxyZwGetNextProcess,

{

1023,1023,1023,1023,371,367,367,139,139,270,272,272,271,271,271,274,275,275,276,278,278

},45

},

{

SSDT,L"ZwUnmapViewOfSection",ProxyZwUnmapViewOfSection,

{

1023,231,267,277,352,348,348,385,385,19,19,19,19,19,19,19,20,20,20,20,20

},46

},

{

SSDT,L"ZwAssignProcessToJobObject",ProxyZwAssignProcessToJobObject,

{

1023,18,19,21,42,42,42,43,43,377,382,382,381,381,381,386,387,387,389,392,392

},47

},

{

SSDT,L"ZwAllocateVirtualMemory",ProxyZwAllocateVirtualMemory,

{

1023,16,17,18,18,18,18,19,19,403,407,407,406,406,406,411,412,412,415,418,418

},57

},

{

SSDT,L"ZwFreeVirtualMemory",ProxyZwFreeVirtualMemory,

{

1023,71,83,87,147,147,147,131,131,278,281,281,280,280,280,284,285,285,286,288,288

},58

},

{

SSSDT,L"NtUserFindWindowEx",ProxyNtUserFindWindowEx,

{

1023,368,378,377,391,391,391,396,396,455,457,458,459,460,459,460,462,466,466,466,467

},25

},

{

SSSDT,L"NtUserBuildHwndList",ProxyNtUserBuildHwndList,

{

1023,302,312,311,322,322,322,323,323,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023

},26

},

{

SSSDT,L"NtUserQueryWindow",ProxyNtUserQueryWindow,

{

1023,466,483,481,504,504,504,515,515,478,480,481,482,483,482,483,485,489,489,489,490

},27

},

{

SSSDT,L"NtUserGetForegroundWindow",ProxyNtUserGetForegroundWindow,

{

1023,393,404,403,418,418,418,423,423,426,428,429,430,430,429,430,431,435,435,435,435

},28

},

{

SSSDT,L"NtUserWindowFromPoint",ProxyNtUserWindowFromPoint,

{

1023,568,592,588,617,617,617,629,629,640,643,646,648,650,649,652,658,664,665,666,667

},29

},

{

SSSDT,L"NtUserSetParent",ProxyNtUserSetParent,

{

1023,510,529,526,550,550,550,560,560,582,585,587,589,591,590,593,595,601,602,603,604

},30

},

{

SSSDT,L"NtUserSetWindowLong",ProxyNtUserSetWindowLong,

{

1023,525,544,540,566,566,566,578,578,560,562,564,566,567,566,569,571,575,576,576,577

},31

},

{

SSSDT,L"NtUserMoveWindow",ProxyNtUserMoveWindow,

{

1023,449,465,464,484,484,484,495,495,499,501,502,503,504,503,505,507,511,511,511,512

},32

},

{

SSSDT,L"NtUserSetWindowPos",ProxyNtUserSetWindowPos,

{

1023,527,546,542,568,568,568,580,580,558,560,562,564,565,564,567,569,573,574,574,575

},33

},

{

SSSDT,L"NtUserSetWindowPlacement",ProxyNtUserSetWindowPlacement,

{

1023,526,545,541,567,567,567,579,579,559,561,563,565,566,565,568,570,574,575,575,576

},34

},

{

SSSDT,L"NtUserShowWindow",ProxyNtUserShowWindow,

{

1023,536,555,551,579,579,579,591,591,547,549,551,553,554,553,556,558,562,563,563,564

},35

},

{

SSSDT,L"NtUserShowWindowAsync",ProxyNtUserShowWindowAsync,

{

1023,537,556,552,580,580,580,592,592,546,548,550,552,553,552,555,557,561,562,562,563

},36

},

{

SSSDT,L"NtUserSendInput",ProxyNtUserSendInput,

{

1023,481,502,500,525,525,525,536,536,606,609,611,613,615,614,617,619,625,626,627,628

},37

},

{

SSSDT,L"NtUserSetWinEventHook",ProxyNtUserSetWinEventHook,

{

1023,533,552,548,576,576,576,588,588,550,552,554,556,557,556,559,561,565,566,566,567

},49

},

{

SSSDT,L"NtUserClipCursor",ProxyNtUserClipCursor,

{

1023,0,330,329,343,343,343,348,348,333,334,335,335,335,335,337,338,342,342,342,342

},48

},

{

SSSDT,L"NtUserSetWindowsHookEx",ProxyNtUserSetWindowsHookEx,

{

1023,530,549,545,573,573,573,585,585,553,555,557,559,560,559,562,564,568,569,569,570

},50

},

{

SSDT,L"ZwMakeTemporaryObject",ProxyZwMakeTemporaryObject,

{

1023,1023,105,110,174,174,174,164,164,246,248,248,247,247,247,250,251,251,252,254,254

},59

},

{

SSDT,L"ZwCreateUserProcess",ProxyZwCreateUserProcess,

{

1023,1023,1023,1023,1023,383,383,93,93,322,326,326,325,325,325,329,330,330,331,334,334

},60

},

{

SSSDT,L"NtUserMessageCall",ProxyNtUserMessageCall,

{

1023,444,460,459,479,479,479,490,490,504,506,507,508,509,508,510,512,516,516,516,517

},61

},

{

SSSDT,L"NtUserPostMessage",ProxyNtUserPostMessage,

{

1023,459,475,474,497,497,497,508,508,486,488,489,490,491,490,492,494,498,498,498,499

},62

},

{

SSSDT,L"NtUserPostThreadMessage",ProxyNtUserPostThreadMessage,

{

1023,460,476,475,498,498,498,509,509,485,487,488,489,490,489,491,493,497,497,497,498

},63

},

{

SSSDT,L"NtUserBuildHwndList\_WIN8",ProxyNtUserBuildHwndList\_WIN8,

{

1023,1023,1023,1023,1023,1023,1023,1023,1023,358,359,360,360,360,360,362,363,367,367,367,367

},64

},

{

SSDT,L"ZwFsControlFile",ProxyZwFsControlFile,

{

1023,1023,84,1023,150,150,150,134,134,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023

},65

},

{

SSSDT,L"NtUserSetImeInfoEx",ProxyNtUserSetImeInfoEx,

{

1023,1023,517,1023,1023,1023,1023,550,550,1023,1023,1023,1023,1023,600,603,605,611,612,613,1023

},66

},

{

SSDT,L"ZwCreateProcessEx",ProxyZwCreateProcessEx,

{

1023,1023,48,50,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023

},72

},

{

SSSDT,L"NtUserGetRawInputData",ProxyNtUserGetRawInputData,

{

1023,1023,428,1023,1023,1023,1023,448,448,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023

},67

},

{

SSSDT,L"NtUserGetRawInputBuffer",ProxyNtUserGetRawInputBuffer,

{

1023,1023,427,1023,1023,1023,1023,447,447,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023

},68

},

{

SSSDT,L"NtUserGetAsyncKeyState",ProxyNtUserGetAsyncKeyState,

{

1023,1023,383,1023,1023,1023,1023,402,402,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023

},69

},

{

SSSDT,L"NtUserGetKeyState",ProxyNtUserGetKeyState,

{

1023,1023,416,1023,1023,1023,1023,436,436,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023

},70

},

{

SSSDT,L"NtUserGetKeyboardState",ProxyNtUserGetKeyboardState,

{

1023,1023,414,1023,1023,1023,1023,434,434,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023

},71

},

{

SSDT,L"ZwQueueApcThread",ProxyZwQueueApcThread,

{

1023,1023,180,1023,1023,1023,1023,269,269,1023,1023,1023,1023,1023,139,142,143,143,144,144,144

},74

},

{

SSDT,L"ZwSetSecurityObject",ProxyZwSetSecurityObject,

{

1023,1023,237,1023,1023,1023,1023,347,347,1023,1023,1023,1023,1023,59,60,61,61,61,61,61

},75

},

{

SSDT,L"ZwOpenFile",ProxyZwOpenFile,

{

1023,1023,116,1023,1023,1023,1023,179,179,1023,1023,1023,1023,1023,232,235,236,236,237,238,238

},76

},

{

SSDT,L"ZwQueueApcThreadEx",ProxyZwQueueApcThreadEx,

{

1023,1023,1023,1023,1023,1023,1023,270,270,1023,1023,1023,1023,1023,138,141,142,142,143,143,143

},77

},

{

SSDT,L"ZwCreateMutant",ProxyZwCreateMutant,

{

1023,1023,43,45,67,67,67,74,74,1023,1023,1023,1023,1023,346,350,351,351,352,355,355

},78

},

{

SSDT,L"ZwQuerySystemInformation",ProxyZwQuerySystemInformation,

{

1023,1023,173,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023

},79

},

{

SSDT,L"ZwQueryIntervalProfile",ProxyZwQueryIntervalProfile,

{

1023,1023,158,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023

},80

},

{

SSDT,L"ZwSetInformationProcess",ProxyZwSetInformationProcess,

{

1023,1023,228,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023

},81

},

{

SSSDT,L"NtGdiAddFontMemResourceEx",ProxyNtGdiAddFontMemResourceEx,

{

1023,1023,4,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023

},82

},

{

SSDT,L"ZwReplyWaitReceivePortEx",ProxyZwReplyWaitReceivePortEx,

{

1023,1023,196,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023

},83

},

{

END,L"KeUserModeCallback",ProxyKeUserModeCallback,

{

1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023

},51

},

{

SSDT,L"ZwOpenKey",ProxyZwOpenKey,

{

1023,1023,119,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023

},84

},

{

SSDT,L"ZwMapViewOfSection",ProxyZwMapViewOfSection,

{

1023,1023,108,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023

},85

},

{

SSDT,L"ZwSetIntervalProfile",ProxyZwSetIntervalProfile,

{

1023,1023,231,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023

},86

},

{

SSSDT,L"NtGdiAddFontResourceW",ProxyNtGdiAddFontResourceW,

{

1023,1023,2,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023

},87

},

{

SSSDT,L"NtGdiAddRemoteFontToDC",ProxyNtGdiAddRemoteFontToDC,

{

1023,1023,3,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023

},88

},

{

SSDT,L"ZwQueryInformationProcess",ProxyZwQueryInformationProcess,

{

1023,1023,154,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023

},89

},

{

SSDT,L"ZwQueryInformationThread",ProxyZwQueryInformationThread,

{

1023,1023,155,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023

},90

},

{

SSDT,L"ZwCreateProfile",ProxyZwCreateProfile,

{

1023,1023,49,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023

},91

},

{

SSDT,L"ZwVdmControl",ProxyZwVdmControl,

{

1023,1023,268,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023

},92

},

{

SSDT,L"ZwCreateProcess",ProxyZwCreateProcess,

{

1023,1023,47,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023

},93

},

{

SSSDT,L"NtGdiAddEmbFontToDC",ProxyNtGdiAddEmbFontToDC,

{

1023,1023,214,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023

},94

},

{

SSDT,L"NtDebugActiveProcess",ProxyNtDebugActiveProcess,

{

1023,1023,57,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023

},95

},

{

SSDT,L"NtAlpcCreatePort",ProxyNtAlpcCreatePort,

{

1023,1023,1023,1023,1023,1023,1023,23,23,1023,1023,1023,1023,1023,401,406,407,407,410,413,413

},96

},

{

SSDT,L"NtCreatePort",ProxyNtCreatePort,

{

1023,1023,46,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023

},97

},

{

SSDT,L"ZwAdjustPrivilegesToken",ProxyZwAdjustPrivilegesToken,

{

1023,1023,11,1023,1023,1023,1023,12,12,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023

},98

},

{

SSDT,L"ZwConnectPort",ProxyZwConnectPort,

{

1023,1023,31,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023

},99

},

{

SSDT,L"ZwSecureConnectPort",ProxyZwSecureConnectPort,

{

1023,1023,210,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023

},100

},

{

SSDT,L"ZwQueryKey",ProxyZwQueryKey,

{

1023,1023,160,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023

},101

},

{

SSDT,L"ZwEnumerateKey",ProxyZwEnumerateKey,

{

1023,1023,71,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023

},102

},

{

SSDT,L"ZwClose",ProxyZwClose,

{

1023,1023,25,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023

},103

},

{

SSSDT,L"NtUserSystemParametersInfo",ProxyNtUserSystemParametersInfo,

{

1023,1023,559,1023,1023,1023,1023,559,595,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023

},104

},

{

END,NULL,NULL,

{

-1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,

},105

}

};

### 1.6 Proxy\*函数模型

NPAGED\_LOOKASIDE\_LIST FilterLookAside;

BOOLEAN g\_EvaluateTime;

//Proxy\*函数模型 3参数函数为例

NTSTATUS \_\_stdcall ProxyNtFunc(int param1,int param2,int param3)

{

NTSTATUS status = STATUS\_ACCESS\_DENIED;

ULONG Result;//自定义结果

ULONGLONG Time = 0;

DProxy\* proxydata = DProxyTable[ENtFunc];

FilterPacket\* packet = ExAllocateFromNPagedLookasideList(&FilterLookAside);

if(g\_EvaluateTime)

Time = KeQueryInterruptTime();

if(!packet)

{

if(!proxydata->OriginFuncAddr)

return status;

return proxydata->OriginFuncAddr(param1,param2,param3);

}

packet->Params[0] = param1;

packet->Params[1] = param2;

packet->Params[2] = param3;

packet->ParamNumber = 3;

packet->OriginFuncAddr = proxydata->OriginFuncAddr;

packet->IndexInTable = ENtFunc;

InterlockedIncrement(&proxydata->PrevFilterRefCount);

Result = ExecPrevFilter(packet,proxydata);//Prev过滤

InterlockedDecrement(&proxydata->PrevFilterRefCount);

if(Result == SYSMON\_UNHANDLED)

{

if(packet->OriginFuncAddr)

{

status = packet->OriginFuncAddr(param1,param2,param3);

packet->Status = status;

InterlockedIncrement(&proxydata->PostFilterRefCount);

Result = ExecPostFilter(packet);//Post过滤

InterlockedDecrement(&proxydata->PostFilterRefCount);

}

}

if(Result == SYSMON\_HANDLED)

{

status = packet->Status;

}

if(g\_EvaluateTime)

EvaluateTime(proxydata,Time);

ExDeleteNPagedLookasideList(packet);

return status;

}

ULONG ExecPrevFilter(FilterPacket\* packet,DProxy\* proxydata)

{

ULONG Result;

if(!proxydata || !packet || packet->DisablePrevFilter)

return SYSMON\_UNHANDLED;

for(int i=0;i<16;i++)

{

if(!proxydata->PrevFilterSlot[i].DeleteCount && proxydata->PrevFilterSlot[i].Filter && proxydata->SlotNum != 0)

{

InterlockedIncrement(&proxydata->PrevFilterSlot[i].CallCount);

packet->CurrentSlot = i;

Result = proxydata->PrevFilterSlot[i].Filter(packet);

InterlockedDecrement(&proxydata->PrevFilterSlot[i].CallCount);

if(packet->Access & 0x10)//如果权限被设置为放行

{

TsLogSprintfOutput(SysMonLogPt,"[Sysnap DbgLog] Modify--> TableIndex %d, Process %s[%d] ",

proxydata->TableIndex,PsGetProcessImageFileName(IoGetCurrentProcess()),PsGetCurrentProcessId());

}

switch(Result)

{

case SYSMON\_FORBID:

TsLogSprintfOutput(SysMonLogPt,"[Sysnap DbgLog] Block--> TableIndex %d, Process %s[%d] ",

proxydata->TableIndex,PsGetProcessImageFileName(IoGetCurrentProcess()),PsGetCurrentProcessId());

return SYSMON\_FORBID;

case SYSMON\_HANDLED:

return SYSMON\_HANDLED;

case SYSMON\_PASS:

case SYSMON\_PASS1:

return SYSMON\_UNHANDLED;

default:

break;

}

}

}

}

ULONG ExecPostFilter(FilterPacket\* packet)

{

ULONG Result;

if(!packet)

return SYSMON\_UNHANDLED;

FILTER\_SLOT\* CurrentSlot = DProxyTable[packet->IndexInTable]->PrevFilterSlot;

for(int i=0;i<16;i++)

{

if(!CurrentSlot [i].DeleteCount)

{

InterlockedIncrement(&CurrentSlot [i].CallCount);

packet->CurrentSlot = i;

Result = packet->PostFilterSlot(packet);

InterlockedDecrement(&CurrentSlot [i].CallCount);

}

switch(Result)

{

case SYSMON\_HANDLED:

return SYSMON\_HANDLED;

case SYSMON\_PASS:

case SYSMON\_PASS1:

case SYSMON\_FORBID:

return SYSMON\_UNHANDLED;

default:

break;

}

}

}

## 驱动接口Interface

### 2.1 DeviceExtension接口

DeviceObject->DeviceExtension结构：

+00h TAG=’TSFL’

+14h FARPROC Interface

FARPROC Interface(intindex)

NTSTATUS Interface(int index,FARPROC\* outfunc)

{//注意下面的函数都是自己实现的穿透函数

If(!outfunc)

Return STATUS\_UNSUCCESSFUL;

switch(index)

{

case 0:

\*outfunc = SetEvaluateTime;

Break;

case 1:

\*outfunc = DisablePrevFilter;

Break;

case 2:

\*outfunc = SetPostFilter;

Break;

case 3:

\*outfunc = ExecOriginFromPacket;

Break;

case 4:

\*outfunc = AddPrevFilter;

Break;

case 5:

\*outfunc = RemovePrevFilter;

Break;

case 6:

\*outfunc = GetCurrentHookInfo;

Break;

case 7:

\*outfunc = GetDProxyTable;

Break;

default:

\*outfunc = NULL;

Break;

}

Return STATUS\_SUCESS;

}

### 2.2 SetEvaluateTime

Void \_\_stdcall SetEvaluateTime(bool EvaluateTime)

{//设置计算过滤函数执行耗时

g\_EvaluateTime = EvaluateTime;

}

### 2.3 SetDisablePrevFilter

Void \_\_stdcall SetDisablePrevFilter(ULONG Index,BOOLEAN Disable)

{//设置是否执行PrevFilter(同时也是PostFilter)

If(Index < APINUMBER)

{

If(DProxyTable[Index])

DProxyTable[Index]-> DisablePrevFilter = Disable;

}

}

### 2.4 SetPostFilter

Void \_\_stdcall SetPostFilter(FilterPacket\* Packet,FARPROC Filter,ULONG Tag)

{//设置PostFilter函数

If(Packet)

{

Packet->TagSlot[Packet->CurrentSlot] = Tag;//用于修改参数或区分Filter

Packet->PostFilterSlot[Packet->CurrentSlot] = Filter;

Packet->SlotCount++;

}

}

### 2.5 ExecOriginFromPacket

NTSTATUS \_\_stdcall ExecOriginFromPacket(FilterPacket\* Packet)

{//执行Nt\*原始函数

If(!Packet || !Packet->OriginFuncAddr)

Return STATUS\_UNSUCCESSFUL;

\_asm

{

Mov eax, Packet->ParamNumber

Test eax,eax

Jbe tag1

Lea ecx,[eax-1]

Test ecx,ecx

Jl tag1

Lea edx,Packet->Params[ecx]//参数逐个压栈

Tag1:

Mov eax,[edx]

Push eax

Sub ecx,1

Sub edx,4

Test ecx,ecx

Jge tag2

Tag2:

Call Packet->OriginFuncAddr

}

}

### 2.6 AddPrevFilter

NTSTATUS \_\_stdcall AddPrevFilter(ULONG Index,FARPROC Filter,ULONG Tag,ULONG PreferSlot,PULONG OutSlot)

{

/\*

Index :Nt函数在全局表DProxyTable中的索引

Filter:要设置的过滤函数

Tag:标志过滤函数（可用于保存参数）

PreferSlot:优先选用的函数槽

OutSlot:实际选用的函数槽

\*/

if(!Filter || Index >= APINUMBER || PreferSlot >= 16)

return STATUS\_UNSUCCESSFUL;

if(!DProxyTable[Index] || !DProxyTable[Index]->IsInitialized)

return STATUS\_UNSUCCESSFUL;

if(DProxyTable[Index]->PrevFilterSlot[PreferSlot].Filter)

{//若函数槽已被占用

for(int i=0;i<16;i++)

{

if(!DProxyTable[Index]->PrevFilterSlot[i].Filter)

{

DProxyTable[Index]->PrevFilterSlot[i].Tag = Tag;

InterlockedExchange(&DProxyTable[Index]->PrevFilterSlot[i].Filter,Filter);

InterlockedIncrement(&DProxyTable[Index]->UsedSlotCount);

if(OutSlot)

\*OutSlot = i;

return STATUS\_SUCCESS;

}

}

}

else

{

DProxyTable[Index]->PrevFilterSlot[i].Tag = Tag;

InterlockedExchange(&DProxyTable[Index]->PrevFilterSlot[i].Filter,Filter);

InterlockedIncrement(&DProxyTable[Index]->UsedSlotCount);

if(OutSlot)

\*OutSlot = i;

return STATUS\_SUCCESS;

}

return STATUS\_UNSUCCESSFUL;

}

### 2.7 RemovePrevFilter

void WaitStop(FILTER\_SLOT\* CurrentSlot)

{

KeInitializeTimer(&CurrentSlot->Timer);

Li.QuadPart = 1000000;

while(CurrentSlot->CallCount)

{

KeSetTimer(&CurrentSlot->Timer,&Li,NULL);

KeWaitForSingleObject(&CurrentSlot->Timer,Executive,KernelMode,FALSE,NULL);

}

KeCancelTimer(&CurrentSlot->Timer);

}

NTSTATUS \_\_fastcall RemovePrevFilter(ULONG Index,FARPROC Filter)

{//从Index对应的函数过滤中查找删除Filter

if(!Filter || Index >= APINUMBER)

return STATUS\_UNSUCCESSFUL;

if(!DProxyTable[Index] || !DProxyTable[Index]->IsInitialized)

return STATUS\_UNSUCCESSFUL;

for(int i=0;i<16;i++)

{

if(DProxyTable[Index]->PrevFilterSlot[i].Filter == Filter)

{

LARGE\_INTEGER Li;

FILTER\_SLOT\* CurrentSlot = &DProxyTable[Index]->PrevFilterSlot[i];

CurrentSlot->DeleteCount = FALSE;

InterlockedIncrement(&CurrentSlot->DeleteCount);

WaitStop(CurrentSlot);

InterlockedExchange(&CurrentSlot->Filter,NULL);

InterlockedDecrement(&DProxyTable[Index]->UsedSlotCount);

WaitStop(CurrentSlot);

InterlockedDecrement(&CurrentSlot->DeleteCount);

return STATUS\_UNSUCCESSFUL;

}

}

return STATUS\_UNSUCCESSFUL;

}

### 2.8 GetCurrentHookInfo

NTSTATUS \_\_stdcall GetCurrentHookInfo(PULONG pLastErrorCode,PULONG pserHookType,PULONG pOsVer,PULONG pHookErrorIndex)

{

if(OsVer && !LastErrorCode && HookErrorIndex == -1)

{

if(pLastErrorCode)

\*pLastErrorCode = 0;//获取系统buildnumber

if(pserHookType)

\*pserHookType = serHookType;

if(pOsVer)

\*pOsVer = OsVer;

if(pHookErrorIndex)

\*pHookErrorIndex = HookErrorIndex;

}

return STATUS\_UNSUCCESSFUL;

}

LastErrorCode

0x01 ZwQuerySystemInformation获取失败

0x02 KeServiceDescriptorTable获取失败

0x04 KeAddSystemServiceTable获取失败

0x08 ShadowSSDT获取失败

0x10 MmUserProbeAddress获取失败

0x20 Int2E校验失败

InitState

0x01 ZwQuerySystemInformation获取成功

0x02 KeServiceDescriptorTable获取成功

0x04 ShadowSSDT获取成功

0x08 获取Inline Hook点成功

serHookType 对KiFastCallEntry做inline hook的类型

1. Nonhook HOOKTYPE\_NONE
2. Inlinehook HOOKTYPE\_INLINE
3. SSDTHook HOOKTYPE\_SSDT
4. 金山共存Inlinehook HOOKTYPE\_KSINLINE

OsVer

BuildNumber:

Win2000

2195 1

WinXp

2600 2

WinServer2003

3790 3

WinVista

6000 4

6001 5

6002 6

Win7

7600 7

7601 8

Win8

8102 9

8250 10

8400 11

8432 12

8441 12

8520 13

Win8.1

9200 14

9600 15

Win10

9841 16

9860 17

9926 18

10041 19

10049 20

?? 0

HookErrorIndex

-1

IsBsod

记录蓝屏

### 2.9 GetDProxyTable

Dproxy\*\* GetDProxyTable()

{

Return DProxyTable;

}

## 基础库

### 3.1 获取注册表键值

NTSTATUS GetRegDataWithType(PWCHAR RegPath,PWCHAR KeyName,PVOID OutData,ULONG BufSize,PULONG OutType)

{

NTSTATUS status;

HANDLE KeyHandle = NULL;

ULONG ResultLength = 0;

OBJECT\_ATTRIBUTES Oa;

UNICODE\_STRING URegPath,UKeyName;

PVOID ValueInfo;

const int BufSize = sizeof(PKEY\_VALUE\_PARTIAL\_INFORMATION) + sizeof(WCHAR[260]);

if(!RegPath || !KeyName)

return STATUS\_INVALID\_PARAMETER;

InitializeObjectAttributes(&Oa,&URegPath,OBJ\_CASE\_INSENSITIVE | OBJ\_KERNEL\_HANDLE,NULL,NULL);

status = ZwOpenKey(&KeyHandle,KEY\_EXECUTE,&Oa);

ValueInfo = ExAllocatePool(NonPagedPool,BufSize);

if(NT\_SUCCESS(status) && ValueInfo)

{

RtlZeroMemory(ValueInfo,BufSize);

RtlInitUnicodeString(&UKeyName,KeyName);

status = ZwQueryValueKey(KeyHandle,&UKeyName,KeyValuePartialInformation,ValueInfo,BufSize,&ResultLength);

if(NT\_SUCCESS(status))

{

PKEY\_VALUE\_PARTIAL\_INFORMATION Info = (PKEY\_VALUE\_PARTIAL\_INFORMATION)ValueInfo;

if(OutType)

\*OutType = Info->Type;

if(OutData)

RtlCopyMemory(OutData,Info->Data,BufSize<Info->DataLength?BufSize:Info->DataLength);

}

}

if(KeyHandle)

ZwClose(KeyHandle);

if(ValueInfo)

ExFreePool(ValueInfo);

return status;

}

NTSTATUS GetRegDataWithSize(PWCHAR RegPath,PWCHAR KeyName,PVOID OutData,ULONG BufSize,PULONG OutSize)

{

NTSTATUS status;

HANDLE KeyHandle = NULL;

ULONG ResultLength = 0;

OBJECT\_ATTRIBUTES Oa;

UNICODE\_STRING URegPath,UKeyName;

PVOID ValueInfo;

const int BufSize = sizeof(PKEY\_VALUE\_PARTIAL\_INFORMATION) + sizeof(WCHAR[260]);

if(!RegPath || !KeyName)

return STATUS\_INVALID\_PARAMETER;

InitializeObjectAttributes(&Oa,&URegPath,OBJ\_CASE\_INSENSITIVE | OBJ\_KERNEL\_HANDLE,NULL,NULL);

status = ZwOpenKey(&KeyHandle,KEY\_EXECUTE,&Oa);

ValueInfo = ExAllocatePool(NonPagedPool,BufSize);

if(NT\_SUCCESS(status) && ValueInfo)

{

RtlZeroMemory(ValueInfo,BufSize);

RtlInitUnicodeString(&UKeyName,KeyName);

status = ZwQueryValueKey(KeyHandle,&UKeyName,KeyValuePartialInformation,ValueInfo,BufSize,&ResultLength);

if(NT\_SUCCESS(status))

{

PKEY\_VALUE\_PARTIAL\_INFORMATION Info = (PKEY\_VALUE\_PARTIAL\_INFORMATION)ValueInfo;

if(OutSize)

\*OutSize = Info->DataLength;

if(OutData)

RtlCopyMemory(OutData,Info->Data,BufSize<Info->DataLength?BufSize:Info->DataLength);

}

}

if(KeyHandle)

ZwClose(KeyHandle);

if(ValueInfo)

ExFreePool(ValueInfo);

return status;

}

NTSTATUS GetRegDataWithSizeAndType(PWCHAR RegPath,PWCHAR KeyName,PVOID OutData,ULONG BufSize,PULONG OutSize,PULONG OutType)

{

NTSTATUS status;

HANDLE KeyHandle = NULL;

ULONG ResultLength = 0;

OBJECT\_ATTRIBUTES Oa;

UNICODE\_STRING URegPath,UKeyName;

PVOID ValueInfo;

const int BufSize = sizeof(PKEY\_VALUE\_PARTIAL\_INFORMATION) + sizeof(WCHAR[260]);

if(!RegPath || !KeyName)

return STATUS\_INVALID\_PARAMETER;

InitializeObjectAttributes(&Oa,&URegPath,OBJ\_CASE\_INSENSITIVE | OBJ\_KERNEL\_HANDLE,NULL,NULL);

status = ZwOpenKey(&KeyHandle,KEY\_EXECUTE,&Oa);

ValueInfo = ExAllocatePool(NonPagedPool,BufSize);

if(NT\_SUCCESS(status) && ValueInfo)

{

RtlZeroMemory(ValueInfo,BufSize);

RtlInitUnicodeString(&UKeyName,KeyName);

status = ZwQueryValueKey(KeyHandle,&UKeyName,KeyValuePartialInformation,ValueInfo,BufSize,&ResultLength);

if(NT\_SUCCESS(status))

{

PKEY\_VALUE\_PARTIAL\_INFORMATION Info = (PKEY\_VALUE\_PARTIAL\_INFORMATION)ValueInfo;

if(OutSize)

\*OutSize = Info->DataLength;

if(OutType)

\*OutType = Info->Type;

if(OutData)

RtlCopyMemory(OutData,Info->Data,BufSize<Info->DataLength?BufSize:Info->DataLength);

}

}

if(KeyHandle)

ZwClose(KeyHandle);

if(ValueInfo)

ExFreePool(ValueInfo);

return status;

}

### 3.2 通过进程名获取进程ID

HANDLE GetProcessIdByName(PWCHAR ProcessName)

{//根据进程名获取进程ID

HANDLE ProcessId = 0;

NTSTATUS status = STATUS\_SUCCESS;

SIZE\_T size = 512;

UNICODE\_STRING UProcessName;

LPVOID Buffer;

RtlInitUnicodeString(&UProcessName,ProcessName);

while(true)

{

Buffer = ExAllocatePool(PagedPool,size);

if(!Buffer)

return 0;

status = ZwQuerySystemInformation(SystemProcessesAndThreadsInformation,Buffer,size,NULL);

if(status != STATUS\_INFO\_LENGTH\_MISMATCH)

break;

ExFreePool(Buffer);

size \*= 2;

}

if(!NT\_SUCCESS(Buffer))

ExFreePool(Buffer);

PSYSTEM\_PROCESS\_INFORMATION ProcessInfo = (PSYSTEM\_PROCESS\_INFORMATION)Buffer;

while(ProcessInfo->NextEntryOffset)

{

ProcessInfo = (PSYSTEM\_PROCESS\_INFORMATION)((UCHAR\*)ProcessInfo + ProcessInfo->NextEntryOffset);

if(RtlEqualUnicodeString(&UProcessName,&ProcessInfo->ImageName,TRUE)

ProcessId = ProcessInfo->UniqueProcessId;

}

ExFreePool(Buffer);

return ProcessId;

}

### 3.3 Rabbit加密算法

Q管很多驱动中存在的用于传输时加密的可逆算法使用了Rabbit分组加密算法，可以分段加解密，加密过程和解密过程一致。根据驱动中加解密过程可以得到如下代码逻辑：

#define ld(x) ((x>>32)&0xFFFFFFFF)

#define hw(x) ((x>>16)&0xFFFF)

#define lw(x) (x&0xFFFF)

#define rotl(x,y) ((x<<y)|(x>>(32-y)))

struct Rabbit

{

unsigned int X[8];

unsigned int C[8];

unsigned int b;

};

bool Rabbit\_nextState(Rabbit\* rabbit);

bool Rabbit\_Init(Rabbit\* rabbit);

unsigned int Rabbit\_toInt(unsigned char\* key,int index);

bool Rabbit\_SetKey(unsigned char\* key,Rabbit\* rabbit);

bool Rabbit\_Init(Rabbit\* rabbit)

{

if(!rabbit)

return false;

for(int i=0;i<8;i++)

{

rabbit->C[i] = 0;

rabbit->X[i] = 0;

rabbit->b = 0;

}

return true;

}

unsigned int Rabbit\_toInt(unsigned char\* key,int index)

{

if(!key)

return 0;

return key[index+3]|(key[index+2]<<8)|(key[index+1]<<16)|(key[index+0]<<24);

}

bool Rabbit\_SetKey(unsigned char\* key,Rabbit\* rabbit)

{

if(!key || !rabbit)

return false;

unsigned int K0 = Rabbit\_toInt(key,0);

unsigned int K1 = Rabbit\_toInt(key,4);

unsigned int K2 = Rabbit\_toInt(key,8);

unsigned int K3 = Rabbit\_toInt(key,12);

rabbit->X[0] = K3;

rabbit->X[1] = (K0<<16)|(K1>>16);

rabbit->X[2] = K2;

rabbit->X[3] = (K3<<16)|(K0>>16);

rabbit->X[4] = K1;

rabbit->X[5] = (K2<<16)|(K3>>16);

rabbit->X[6] = K0;

rabbit->X[7] = (K1<<16)|(K2>>16);

rabbit->C[0] = (K1<<16)|(K1>>16);

rabbit->C[1] = ((K3&0xFFFF0000)|(K2&0xFFFF));

rabbit->C[2] = (K0<<16)|(K0>>16);

rabbit->C[3] = ((K2&0xFFFF0000)|(K1&0xFFFF));

rabbit->C[4] = (K3<<16)|(K3>>16);

rabbit->C[5] = ((K1&0xFFFF0000)|(K0&0xFFFF));

rabbit->C[6] = (K2<<16)|(K2>>16);

rabbit->C[7] = ((K0&0xFFFF0000)|(K3&0xFFFF));

rabbit->b = 0;

for(int i=0;i<4;i++)

{

Rabbit\_nextState(rabbit);

}

for(int i=0;i<8;i++)

{

rabbit->C[i] ^= rabbit->X[(i+4)&7];

}

return true;

}

unsigned int Rabbit\_u1(unsigned int n)

{

unsigned int r0 = ((lw(n) \* lw(n)) >> 17) + lw(n) \* hw(n);

unsigned int r1 = (r0 >> 15) + hw(n) \* hw(n);

unsigned int r2 = n \* n;

return r1 ^ r2;

// \_asm

// {

// mov esi,n

// movzx edx,si

// mov ecx,esi

// shr ecx,0x10

// push edi

// mov eax, edx

// imul eax, edx

// mov edi, ecx

// imul edi, edx

// mov edx, ecx

// imul edx, ecx

// mov ecx, esi

// imul ecx, esi

// shr eax, 11h

// add eax, edi

// shr eax, 0Fh

// add eax, edx

// pop edi

// xor eax, ecx

// }

}

bool Rabbit\_nextState(Rabbit\* rabbit)

{

unsigned int G[8]={0};

unsigned int c\_old[8];

if(!rabbit)

return false;

memcpy(c\_old,rabbit->C,sizeof(c\_old));

rabbit->C[0] += rabbit->b + 1295307597;

rabbit->C[1] += (rabbit->C[0] < c\_old[0]) - 749914925;

rabbit->C[2] += (rabbit->C[1] < c\_old[1]) + 886263092;

rabbit->C[3] += (rabbit->C[2] < c\_old[2]) + 1295307597;

rabbit->C[4] += (rabbit->C[3] < c\_old[3]) - 749914925;

rabbit->C[5] += (rabbit->C[4] < c\_old[4]) + 886263092;

rabbit->C[6] += (rabbit->C[5] < c\_old[5]) + 1295307597;

rabbit->C[7] += (rabbit->C[6] < c\_old[6]) - 749914925;

rabbit->b = rabbit->C[7] < c\_old[7];

for(int i=0;i<8;i++)

{

G[i] = Rabbit\_u1(rabbit->X[i]+rabbit->C[i]);

}

rabbit->X[0] = G[0] + rotl(G[7],16) + rotl(G[6],16);

rabbit->X[1] = G[1] + rotl(G[0],8) + G[7];

rabbit->X[2] = G[2] + rotl(G[1],16) + rotl(G[0],16);

rabbit->X[3] = G[3] + rotl(G[2],8) + G[1];

rabbit->X[4] = G[4] + rotl(G[3],16) + rotl(G[2],16);

rabbit->X[5] = G[5] + rotl(G[4],8) + G[3];

rabbit->X[6] = G[6] + rotl(G[5],16) + rotl(G[4],16);

rabbit->X[7] = G[7] + rotl(G[6],8) + G[5];

return true;

}

bool Rabbit\_toByte(unsigned int s,unsigned char\* d)

{

if(!d)

return false;

d[0] = (s>>24)&0xff;

d[1] = (s>>16)&0xff;

d[2] = (s>>8)&0xff;

d[3] = (s)&0xff;

return true;

}

bool Rabbit\_EncryptBuf(Rabbit\* rabbit,unsigned char\* src,int srclen,unsigned char\* dst,int dstlen)

{

unsigned int K[4] = {0};

unsigned char v[4] = {0};

if(!rabbit || !src || !dst || srclen>dstlen || srclen<=0)

return false;

for(int i=0;i<srclen;i+=16)

{

Rabbit\_nextState(rabbit);

K[0] = rabbit->X[0]^(rabbit->X[5]>>16)^(rabbit->X[3]<<16)^Rabbit\_toInt(src,i);

K[1] = rabbit->X[2]^(rabbit->X[5]<<16)^(rabbit->X[7]>>16)^Rabbit\_toInt(src,i+4);

K[2] = rabbit->X[4]^(rabbit->X[7]<<16)^(rabbit->X[1]>>16)^Rabbit\_toInt(src,i+8);

K[3] = rabbit->X[6]^(rabbit->X[3]>>16)^(rabbit->X[1]<<16)^Rabbit\_toInt(src,i+12);

for(int j=0;j<4;j++)

{

Rabbit\_toByte(K[j],v);

\*(unsigned int\*)(dst+i+4\*j) = \*(unsigned int\*)v;

}

}

return true;

}

下面是一个实例：

unsigned char encrypt[448] =

{//加密过的进程id

0x30,0xa2,0xf6,0xa4,0xdb,0x6c,0xe9,0xa2,0x4d,0xdd,0x66,0x18,0xd1,0x8e,0xa9,0x8a,

0x1c,0xa2,0xf6,0xa4,0xf3,0x6c,0xe9,0xa2,0x65,0xdd,0x66,0x18,0xc9,0x8e,0xa9,0x8a,

0x08,0xa2,0xf6,0xa4,0x0b,0x6c,0xe9,0xa2,0xb5,0xdd,0x66,0x18,0x09,0x8e,0xa9,0x8a,

0xc8,0xa2,0xf6,0xa4,0x43,0x6c,0xe9,0xa2,0xd5,0xdd,0x66,0x18,0x81,0x89,0xa9,0x8a,

0x4c,0xa5,0xf6,0xa4,0xbf,0x6b,0xe9,0xa2,0x51,0xda,0x66,0x18,0xf5,0x89,0xa9,0x8a,

0xc8,0xa5,0xf6,0xa4,0x43,0x6b,0xe9,0xa2,0xc5,0xda,0x66,0x18,0x6d,0x89,0xa9,0x8a,

0x7c,0xa4,0xf6,0xa4,0x17,0x6a,0xe9,0xa2,0x89,0xdb,0x66,0x18,0x1d,0x88,0xa9,0x8a,

0xc4,0xa4,0xf6,0xa4,0x5f,0x6a,0xe9,0xa2,0xf1,0xdb,0x66,0x18,0x55,0x88,0xa9,0x8a,

0x14,0xa7,0xf6,0xa4,0xe7,0x69,0xe9,0xa2,0x79,0xd8,0x66,0x18,0xdd,0x8b,0xa9,0x8a,

0x04,0xa7,0xf6,0xa4,0xf7,0x69,0xe9,0xa2,0x69,0xd8,0x66,0x18,0xcd,0x8b,0xa9,0x8a,

0xf8,0xa7,0xf6,0xa4,0x13,0x69,0xe9,0xa2,0x85,0xd8,0x66,0x18,0x29,0x8b,0xa9,0x8a,

0xe8,0xa7,0xf6,0xa4,0x2b,0x69,0xe9,0xa2,0xbd,0xd8,0x66,0x18,0x01,0x8b,0xa9,0x8a,

0x64,0xa6,0xf6,0xa4,0x97,0x68,0xe9,0xa2,0x39,0xd9,0x66,0x18,0x85,0x8a,0xa9,0x8a,

0xfc,0xa6,0xf6,0xa4,0x2b,0x68,0xe9,0xa2,0xbd,0xd9,0x66,0x18,0x09,0x8a,0xa9,0x8a,

0xc8,0xa6,0xf6,0xa4,0x43,0x68,0xe9,0xa2,0xd5,0xd9,0x66,0x18,0x5d,0x8a,0xa9,0x8a,

0x80,0xa6,0xf6,0xa4,0x17,0x6e,0xe9,0xa2,0xd1,0xdf,0x66,0x18,0x19,0x8c,0xa9,0x8a,

0x90,0xa0,0xf6,0xa4,0x5f,0x6e,0xe9,0xa2,0x61,0xdf,0x66,0x18,0x81,0x8e,0xa9,0x8a,

0xe0,0xa5,0xf6,0xa4,0x33,0x6b,0xe9,0xa2,0xa5,0xda,0x66,0x18,0xe1,0x88,0xa9,0x8a,

0x80,0xa5,0xf6,0xa4,0xe7,0x6a,0xe9,0xa2,0x6d,0xdb,0x66,0x18,0x15,0x88,0xa9,0x8a,

0xdc,0xa4,0xf6,0xa4,0x3f,0x6a,0xe9,0xa2,0x95,0xd8,0x66,0x18,0x7d,0x8b,0xa9,0x8a,

0xa0,0xa7,0xf6,0xa4,0x77,0x69,0xe9,0xa2,0xed,0xd8,0x66,0x18,0xe9,0x8b,0xa9,0x8a,

0x40,0xa7,0xf6,0xa4,0x8b,0x69,0xe9,0xa2,0xc9,0xdb,0x66,0x18,0x65,0x88,0xa9,0x8a,

0xa4,0xa4,0xf6,0xa4,0x03,0x68,0xe9,0xa2,0x69,0xd9,0x66,0x18,0xd5,0x8a,0xa9,0x8a,

0x78,0xa9,0xf6,0xa4,0x93,0x67,0xe9,0xa2,0x05,0xd6,0x66,0x18,0x91,0x85,0xa9,0x8a,

0x50,0xa9,0xf6,0xa4,0xab,0x67,0xe9,0xa2,0xdd,0xd6,0x66,0x18,0x05,0x84,0xa9,0x8a,

0x0c,0xab,0xf6,0xa4,0x6b,0x62,0xe9,0xa2,0xfd,0xd0,0x66,0x18,0xe9,0x83,0xa9,0x8a,

0xa8,0xaf,0xf6,0xa4,0xb7,0x63,0xe9,0xa2,0xad,0xce,0x66,0x18,0xa1,0x9c,0xa9,0x8a,

0x44,0xb2,0xf6,0xa4,0x7f,0x7c,0xe9,0xa2,0x15,0xca,0x66,0x18,0xb1,0x8d,0xa9,0x8a,

};

unsigned char decrypt[448]={0};//解密到目标内存

Rabbit rabbit;

Rabbit\_Init(&rabbit);//初始化

Rabbit\_SetKey((unsigned char\*)"0123456789ABCDEF",&rabbit);//加载密钥

Rabbit\_EncryptBuf(&rabbit,encrypt,448,decrypt,448);//加密/解密

/\*解密后的进程id

00000344 00000358 0000035c 00000360

00000368 00000370 00000374 00000378

0000037c 00000388 000003a4 000003b8

000003bc 000003c0 000003c4 00000430

00000438 0000043c 00000440 00000444

000004bc 000004c0 000004d4 000004dc

00000508 00000594 00000598 000005ac

000005b0 000005dc 000005e0 000005e4

00000660 00000664 00000668 0000066c

00000670 00000674 00000678 0000067c

0000068c 00000690 00000694 00000698

0000069c 000006a8 000006ac 000006b0

00000710 00000714 00000728 00000734

00000788 000007a8 000007ac 000007b8

000007bc 000007c0 000007c4 000007ec

000007f4 00000194 000001c0 000001a8

000001e4 000001dc 00000170 00000330

00000494 000004b0 000004b4 00000550

000004f4 00000564 0000057c 000005a4

000005a8 000005bc 00000684 000006cc

000006d4 000006f4 000006fc 00000658

00000634 00000608 000005d8 000005d4

000005d0 00000780 00000778 00000764

0000080c 00000810 00000814 00000820

00000824 00000828 000008cc 000009b4

00000a78 00000de8 00000eec 00000e58

00000edc 00000c34 000010bc 00001110

00001330 000013fc 00001404 00000000

\*/

## InlineHook KiFastCallEntry

* 1. 重置蓝屏死机计数 用于检测hook造成的蓝屏
  2. 获取SSDT/SSSDT/ hook点
  3. 分配跳转表ServiceMapTable
  4. 改写KiFastCallEntry使跳转表生效
  5. Hook重要回调
  6. 开启日志记录
  7. Hook KeUserModeCallback





### 4.1 获取SSDT/SSSDT/Hook点

ULONG InitState = 0;

ULONG LastErrorCode = 0;

UCHAR int2Ecode1[10] = {0x8B, 0xFC, 0x3B, 0x35, 0x00, 0x00, 0x00, 0x00, 0x0F, 0x83};

UCHAR int2Ecode2[22] = {0x8B, 0xFC, 0xF6, 0x45, 0x72, 0x02, 0x75, 0x06, 0xF6, 0x45, 0x6C,

0x01, 0x74, 0x0C, 0x3B, 0x35, 0x00, 0x00, 0x00, 0x00, 0x0F, 0x83};

ULONG GetHookPoint1()

{//检测系统原始KiFastCallEntry

\*(ULONG\*)(int2Ecode2 + 4) = MmUserProbeAddress;//构造cmp esi,dword ptr ds:[MmUserProbeAddress]

if(!MmUserProbeAddress)

return 0;

ULONG ptr = GetKiSystemServiceAddr();//KiFastCallEntry总在KiSystemService之后

if(ptr < NtosBase || ptr > NtosBase+NtosSize)

return 0;

for(int offset = 0;offset < 1024;offset++,ptr++)

{

/\*

mov edi,esp

cmp esi,dword ptr ds:[MmUserProbeAddress]

jae ??

\*/

if(RtlCompareMemory(ptr,int2Ecode1,sizeof(int2Ecode1)) == sizeof(int2Ecode1))

return ptr;

}

return 0;

}

ULONG GetHookPoint2()

{//检测是否被金山inline hook过得KiFastCallEntry

\*(ULONG\*)(int2Ecode2 + 16) = MmUserProbeAddress;//构造cmp esi,dword ptr ds:[MmUserProbeAddress]

if(!MmUserProbeAddress)

return 0;

ULONG ptr = GetKiSystemServiceAddr();

if(ptr < NtosBase || ptr > NtosBase+NtosSize)

return 0;

for(int offset = 0;offset < 1024;offset++,ptr++)

{

/\*

mov edi,esp

test byte ptr[ebp+72h],2

jne $1

test byte ptr[ebp+6Ch],1

je ??

$1:

cmp esi,dword ptr ds:[MmUserProbeAddress]

jae ??

\*/

if(RtlCompareMemory(ptr,int2Ecode2,sizeof(int2Ecode2)) == sizeof(int2Ecode2))

return ptr;

}

return 0;

}

ULONG InitState = 0,LastErrorCode = 0,serHookType = 0;

ULONG NtosBase,NtosSize,SSDTBase,SSDTLimit,ShadowSSDTBase,ShadowSSDTLimit,Win32kBase,Win32kSize;

ULONG dwcsrssId;

ULONG BuildIndex;

PKSERVICE\_TABLE\_DESCRIPTOR KeServiceDescriptorTable;

ULONG MmUserProbeAddress;

struct HOOKINFO

{

ULONG InlinePoint;

ULONG JmpBackPoint;

UCHAR OriginCode[8];

};

HOOKINFO HookInfo1={0},HookInfo2={0};

/\*

KiFastCallEntry+0xde:

mov ebx,dword ptr [edi+eax\*4]

jmp 81f9e3e0

nop => InlinePoint

nop

nop

jmp TsFltMgr+0x2300 (f8517300)

jae nt!KiSystemCallExit2+0x9f (8053e7ec) => JmpBackPoint

rep movs dword ptr es:[edi],dword ptr [esi]

call ebx

OriginCode[8];

8bfc mov edi,esp

3b35549a5580 cmp esi,dword ptr [nt!MmUserProbeAddress]

\*/

NTSTATUS PrepareHook()

{

RTL\_PROCESS\_MODULE\_INFORMATION Win32kInfo,NtosInfo;

NTSTATUS status = STATUS\_UNSUCCESSFUL;

ULONG ReturnLength = 0;

ULONG OutData;

LPVOID Buffer = NULL;

RtlZeroMemory(&Win32kInfo,sizeof(Win32kInfo));

RtlZeroMemory(&NtosInfo,sizeof(NtosInfo));

InitState = 0;

LastErrorCode = 0;

status = GetRegDataWithType(L"\\REGISTRY\\MACHINE\\SYSTEM\\CurrentControlSet\\Services\\TsFltMgr",

L"thm", &OutData, 4, 0)

if(NT\_SUCCESS(status) && OutData == 1)

serHookType = HOOKTYPE\_SSDT;

ZwQuerySystemInformation(SystemModuleInformation,&ReturnLength,0,&ReturnLength);

if(ReturnLength)

Buffer = ExAllocatePool(PagedPool,ReturnLength);

if(Buffer)

status = ZwQuerySystemInformation(SystemModuleInformation,Buffer,ReturnLength,NULL);

if(!NT\_SUCCESS(status))

{

ExFreePool(Buffer);

Buffer = NULL;

}

if(Buffer)

{

RtlCopyMemory(&NtosInfo,((PRTL\_PROCESS\_MODULES)Buffer)->Modules,sizeof(NtosInfo));

ExFreePool(Buffer);

NtosBase = NtosInfo.ImageBase;

NtosSize = NtosInfo.ImageSize;

ULONG FuncAddr = MiLocateExportName(NtosBase,"KeServiceDescriptorTable");

if(FuncAddr && MmIsAddressValid(FuncAddr))

{

PKSERVICE\_TABLE\_DESCRIPTOR KeServiceDescriptorTable = (PKSERVICE\_TABLE\_DESCRIPTOR)FuncAddr;

SSDTBase = KeServiceDescriptorTable[0].Base;

SSDTLimit = KeServiceDescriptorTable[0].Limit;

InitState |= 2;

dwcsrssId = GetProcessIdByName(L"csrss.exe");

if(GetProcessInfoByFileName("WIN32K.SYS",&Win32kInfo))

{

Win32kBase = Win32kInfo.ImageBase;

Win32kSize = Win32kInfo.ImageSize;

if(NT\_SUCCESS(GetShadowSSDTInfo()))

InitState |= 4;

}

UNICODE\_STRING UMmUserProbeAddress;

RtlInitUnicodeString(&UMmUserProbeAddress,L"MmUserProbeAddress");

ULONG MmUserProbeAddress = MmGetSystemRoutineAddress(&UMmUserProbeAddress);

if(MmUserProbeAddress && MmIsAddressValid(MmUserProbeAddress))

{

HookInfo1.InlinePoint = GetHookPoint1();

if(HookInfo1.InlinePoint)

HookInfo1.JmpBackPoint = HookInfo1.InlinePoint + 8;

else

{

HookInfo2.InlinePoint = GetHookPoint2();

if(HookInfo2.InlinePoint)

HookInfo2.JmpBackPoint = HookInfo2.InlinePoint + 6;

}

if(HookInfo1.InlinePoint == 0 && HookInfo2.InlinePoint == 0)

{

LastErrorCode |= 0x20;

return status;

}

InitState |= 8;

MmUserProbeAddress = \*(ULONG\*)MmUserProbeAddress;

status = STATUS\_SUCCESS;

}

else

{

LastErrorCode |= 0x10;

}

}

else

{

LastErrorCode |= 2;

}

}

else

{

LastErrorCode |= 1;

}

return status;

}

### 4.2 从KiSystemService获取KiFastCallEntry

struct IDTR

{

USHORT IDTLimit;

PKIDTENTRY IDTBase;

};

ULONG GetKiSystemServiceAddr()

{

IDTR idt = {0};

RTL\_PROCESS\_MODULE\_INFORMATION NtosInfo,Win32kInfo;

RtlZeroMemory(&NtosInfo,sizeof(NtosInfo));

\_\_sidt(&idt);

if(idt.IDTBase)

{

KIDTENTRY int2e = idt.IDTBase[0x2E];

if(MmIsAddressValid(int2e))

return MAKEULONG(int2e.ExtendedOffset,int2e.Offset);

}

return 0;

}

### 4.3 获取SSSDT信息

NTSTATUS GetShadowSSDTInfo()

{

NTSTATUS status = STATUS\_UNSUCCESSFUL;

PVOID pt = MiLocateExportName(NtosBase, "KeAddSystemServiceTable");

PMDL mdl = NULL;

if(pt && !MmIsAddressValid(pt))

{//页换出

mdl = MmCreateMdl(NULL,pt,0x1000);

if(mdl)

{

MmProbeAndLockPages(mdl,KernelMode,IoReadAccess);

}

}

if(!pt || !MmIsAddressValid(pt))

{

LastErrorCode |= 4;

goto END;

}

switch(BuildIndex)

{

case WIN2000:

case WINXP:

case WINXPSP3:

case WINVISTA:

case WINVISTASP1:

case WINVISTASP2:

case WIN7:

case WIN7SP1:

for(int i=0;i<256;i++)

{

if(MmIsAddressValid(pt) && \*(USHORT\*)pt == 0x888D)

{//8d88603f5580 lea ecx,nt!KeServiceDescriptorTableShadow

PKSERVICE\_TABLE\_DESCRIPTOR Table = \*(ULONG\*)(pt+2);

ShadowSSDTBase = Table[1].Base;

ShadowSSDTLimit = Table[1].Limit;

status = STATUS\_SUCCESS;

}

}

break;

case WIN8\_8102:

case WIN8\_8250:

case WIN81\_9200:

case WIN81\_9600:

for(int i=0;i<256;i++)

{

if(MmIsAddressValid(pt) && \*(USHORT\*)pt == 0xB983)

{//83b9603f558083 cmp dword ptr nt!KeServiceDescriptorTableShadow

PKSERVICE\_TABLE\_DESCRIPTOR Table = \*(ULONG\*)(pt+2);

if(KeServiceDescriptorTable == Table)

{

ShadowSSDTBase = Table[1].Base;

ShadowSSDTLimit = Table[1].Limit;

status = STATUS\_SUCCESS;

}

}

}

break;

case WIN10\_9841:

case WIN10\_9860:

case WIN10\_9926:

case WIN10\_10041:

case WIN10\_10049:

for(int i=0;i<256;i++)

{

if(MmIsAddressValid(pt) && \*(USHORT\*)pt == 0x3D83)

{//833d90d28d8100 cmp dword ptr [nt!KeServiceDescriptorTableShadow+0x10],0

PKSERVICE\_TABLE\_DESCRIPTOR Table = \*(ULONG\*)(pt+2) - 16;

if(KeServiceDescriptorTable == Table)

{

ShadowSSDTBase = Table[1].Base;

ShadowSSDTLimit = Table[1].Limit;

status = STATUS\_SUCCESS;

}

}

}

break;

}

if(!ShadowSSDTBase)

LastErrorCode |= 8;

END:

if(mdl)

{

MmUnlockPages(mdl);

IoFreeMdl(mdl);

}

}

### 4.4 初始化InlineHook KiFastCallEntry跳转表

BOOLEAN InitProxyTable()

{

ULONG BuildIndex = GetBuildNumberAsIndex();

for(int i = 0;i < APINUMBER;i++)

{

for(int j = 0;j < BUILDMAX;j++)

{

if(SProxyTable[i].ServiceIndex[j] == 0)

SProxyTable[i].ServiceIndex[j] = 1023;//置为无效

}

}

if(GetServiceIndex("ZwCreateKey") == -1)

{//不能获取到服务号

if(NtosBase)

{

for(int i = 0;i < APINUMBER;i++)

{

if(SProxyTable[i].ServiceTableType == END)

break;

else if(SProxyTable[i].ServiceTableType == SSDT)

{

if(SProxyTable[i].ApiName)

{

UNICODE\_STRING UApiName;

ANSI\_STRING AApiName;

RtlInitUnicodeString(&UApiName,SProxyTable[i].ApiName);

if(NT\_SUCCESS(RtlUnicodeStringToAnsiString(&AApiName,&UApiName,TRUE)))

{

GetServiceIndexFromNtos(AApiName.Buffer);

RtlFreeAnsiString(AApiName);

}

}

}

}

}

}

else

{//能获取到服务号 由于SProxyTable已存服务号，所以下面代码本身无效

for(int i = 0;i < APINUMBER;i++)

{

if(SProxyTable[i].ServiceTableType == END)

break;

if(SProxyTable[i].ServiceTableType == SSDT)

{

if(SProxyTable[i].ApiName)

{

UNICODE\_STRING UApiName;

ANSI\_STRING AApiName;

RtlInitUnicodeString(&UApiName,SProxyTable[i].ApiName);

if(NT\_SUCCESS(RtlUnicodeStringToAnsiString(&AApiName,&UApiName,TRUE)))

{

ULONG Index = GetServiceIndexFromNtdll(AApiName.Buffer);

if(Index != -1)

SProxyTable[i].ServiceIndex[BuildIndex] = Index;

RtlFreeAnsiString(AApiName);

}

}

}

}

}

return TRUE;

}

int AllocHookTable()

{//已知系统情况下初始化代理表，直接使用ServiceIndex

ULONG BuildIndex = GetBuildNumberAsIndex();

DProxy\* Proxydata = NULL;

ULONG Count = 0;//未成功布置的代理函数个数

if(BuildIndex >= 0 && BuildIndex < BUILDMAX)

{

InitProxyTable();

for(int i=0;i<APINUMBER;i++)

{

if(SProxyTable[i].ServiceTableType == END)

break;

if(SProxyTable[i].ServiceIndex[BuildIndex] != 1023)

{

if(SProxyTable[i].ApiName && SProxyTable[i].ProxyFunc && SProxyTable[i].IndexInTable < APINUMBER)

{

Proxydata = (DProxy\*)ExAllocatePool(NonPagedPool,sizeof(DProxy));

if(Proxydata)

{

Proxydata->IsInitialized = FALSE;

Proxydata->ApiName = SProxyTable[i].ApiName;

Proxydata->TableIndex = SProxyTable[i].IndexInTable;

Proxydata->ProxyFuncAddr = SProxyTable[i].ProxyFunc;

Proxydata->ServiceIndex = -1;

KeInitializeEvent(Proxydata->Lock,SynchronizationEvent,FALSE);

DProxyTable[SProxyTable[i].IndexInTable] = Proxydata;

ULONG ServiceType = SProxyTable[i].ServiceTableType;

ULONG ServiceIndex = SProxyTable[i].ServiceIndex[BuildIndex];

if(ServiceType >= 2 || ServiceIndex== -1 || ServiceIndex >= 1024)

{

Count++;

}

else

{

ServiceMapTable[ServiceType][ServiceIndex] = Proxydata;

Proxydata->ServiceTableType = ServiceType;

Proxydata->ServiceIndex = ServiceIndex;

}

}

}

}

}

return Count;

}

return APINUMBER;//返回未成功代理的个数

}

int AllocHookTableU()

{//未知系统情况下初始化代理表，临时从Ntos模块获取ServiceIndex

if(!NtosBase)

return 0;//返回成功布置代理的个数

for(int i=0;i<APINUMBER;i++)

{

if(SProxyTable[i].ServiceTableType == END)

break;

else if(SProxyTable[i].ServiceTableType == SSDT)

{

if(SProxyTable[i].ApiName)

{

PUNICODE\_STRING UApiName;

ANSI\_STRING AApiName;

ULONG Index = -1;

RtlInitUnicodeString(&UApiName,SProxyTable[i].ApiName);

if(NT\_SUCCESS(RtlUnicodeStringToAnsiString(&AApiName,&UApiName,TRUE)))

{

Index = GetServiceIndexFromNtos(AApiName.Buffer);

RtlFreeAnsiString(&AApiName);

}

if(Index != -1)

{

if(SProxyTable[i].ApiName && SProxyTable[i].ProxyFunc && SProxyTable[i].IndexInTable < APINUMBER)

{

Proxydata = (DProxy\*)ExAllocatePool(NonPagedPool,sizeof(DProxy));

if(Proxydata)

{

Proxydata->IsInitialized = FALSE;

Proxydata->ApiName = SProxyTable[i].ApiName;

Proxydata->TableIndex = SProxyTable[i].IndexInTable;

Proxydata->ProxyFuncAddr = SProxyTable[i].ProxyFunc;

Proxydata->ServiceIndex = -1;

KeInitializeEvent(Proxydata->Lock,SynchronizationEvent,FALSE);

DProxyTable[SProxyTable[i].IndexInTable] = Proxydata;

ULONG ServiceType = SProxyTable[i].ServiceTableType;

ULONG ServiceIndex = SProxyTable[i].ServiceIndex[BuildIndex];

if(ServiceType >= 2 || ServiceIndex== -1 || ServiceIndex >= 1024)

{

Count++;

ServiceMapTable[ServiceType][ServiceIndex] = Proxydata;

Proxydata->ServiceTableType = ServiceType;

Proxydata->ServiceIndex = ServiceIndex;

}

}

}

}

}

}

}

return Count;

}

### 4.5 获取系统服务号的2种方式

ULONG GetServiceIndexFromNtdll(PCHAR ApiName)

{

ULONG result = -1;

RTL\_PROCESS\_MODULE\_INFORMATION ProcessInfo;

ULONG Index;

RtlZeroMemory(&ProcessInfo,sizeof(ProcessInfo));

if(GetProcessInfoByFileName("ntdll.dll",&ProcessInfo) && ProcessInfo.ImageBase && ApiName)

{

Index = MiLocateExportName(ProcessInfo.ImageBase,ApiName);

if(Index)

{

result = \*(ULONG\*)(Index+1);

}

}

return result;

}

ULONG GetServiceIndexFromNtos(PCHAR ApiName)

{

ULONG result = -1;

ULONG Index = MiLocateExportName(ProcessInfo.ImageBase,ApiName);

if(Index)

{

result = \*(ULONG\*)(Index+1);

}

if(Index & 0xFFFFC000)

{

result = -1;

}

return result;

}

### 4.6 InlineHook过程

NTSTATUS DoInlineHook()

{

NTSTATUS status = STATUS\_UNSUCCESSFUL;

ULONG MajorVersion = 0,MinorVersion = 0,BuildNumber = 0;

PsGetVersion(&MajorVersion,&MinorVersion,&BuildNumber,NULL);

if(serHookType == HOOKTYPE\_SSDT || !(InitState & 8))//外部hook或没找到hook挂载点

return status;

if(HookInfo1.InlinePoint)

{//如果为默认方式hook

if(GetBuildNumberAsIndex() >= WINVISTA)

status = ModifyAndHook(HookInfo1.InlinePoint,InlineKiFastCallEntry1);

else

status = ModifyAndHook(HookInfo1.InlinePoint,InlineKiFastCallEntry2);

if(NT\_SUCCESS(status))

serHookType = HOOKTYPE\_INLINE;

}

else if(HookInfo2.InlinePoint)

{//如果为金山共存方式hook

if(GetBuildNumberAsIndex() >= WINVISTA)

status = ModifyAndHookK(HookInfo2.InlinePoint,InlineKiFastCallEntry3);

else

status = ModifyAndHookK(HookInfo2.InlinePoint,InlineKiFastCallEntry4);

if(NT\_SUCCESS(status))

serHookType = HOOKTYPE\_KSINLINE;

}

if(NT\_SUCCESS(status))

{

for(int i=0;i<APINUMBER;i++)

{

if(DProxyTable[i] != 0 && (DProxyTable[i]->ServiceTableType == SSDT || DProxyTable[i]->ServiceTableType == SSSDT))

DProxyTable[i]->IsInitialized = TRUE;

}

}

}

NTSTATUS DoInlineHookU()

{

NTSTATUS status = STATUS\_UNSUCCESSFUL;

ULONG MajorVersion = 0,MinorVersion = 0,BuildNumber = 0;

PsGetVersion(&MajorVersion,&MinorVersion,&BuildNumber,NULL);

if(serHookType == HOOKTYPE\_SSDT || !(InitState & 8))//采用外部hook或没找到hook挂载点

return status;

if(HookInfo1.InlinePoint)

{//如果为默认方式hook

if(GetBuildNumberAsIndex() >= WINVISTA)

status = ModifyAndHook(HookInfo1.InlinePoint,InlineKiFastCallEntry1);

else

status = ModifyAndHook(HookInfo1.InlinePoint,InlineKiFastCallEntry2);

if(NT\_SUCCESS(status))

serHookType = HOOKTYPE\_INLINE;

}

else if(HookInfo2.InlinePoint)

{//如果为金山共存方式hook

if(GetBuildNumberAsIndex() >= WINVISTA)

status = ModifyAndHookK(HookInfo2.InlinePoint,InlineKiFastCallEntry3);

else

status = ModifyAndHookK(HookInfo2.InlinePoint,InlineKiFastCallEntry4);

if(NT\_SUCCESS(status))

serHookType = HOOKTYPE\_KSINLINE;

}

if(NT\_SUCCESS(status))

{

for(int i=0;i<APINUMBER;i++)

{

if(DProxyTable[i] != 0 && (DProxyTable[i]->ServiceTableType == SSDT || DProxyTable[i]->ServiceTableType == SSSDT))

DProxyTable[i]->IsInitialized = TRUE;

}

}

}

NTSTATUS ModifyAndHook(ULONG InlinePoint,FARPROC JmpToFunc)

{//HOOKTYPE\_INLINE 更改KiFastCallEntry

NTSTATUS status;

PVOID MappedAddress = NULL;

if(!InlinePoint || !InlineHookFunc)

return STATUS\_UNSUCCESSFUL;

PMDL mdl = GetWritablePage(InlinePoint,16,&MappedAddress);

/\*

改写 为

mov edi,esp nop

cmp esp,dword ptr ds:[?] nop

nop

jmp ?

\*/

if(!mdl)

return STATUS\_UNSUCCESSFUL;

ULONG data[3] = {InlinePoint,JmpToFunc,MappedAddress};

status = MakeJmp(data,DoHook);

MmUnlockPages(mdl);

IoFreeMdl(mdl);

return status;

}

NTSTATUS ModifyAndHookK(ULONG InlinePoint,FARPROC JmpToFunc)

{//HOOKTYPE\_KSINLINE 更改KiFastCallEntry

NTSTATUS status;

PVOID MappedAddress = NULL;

if(!InlinePoint || !InlineHookFunc)

return STATUS\_UNSUCCESSFUL;

PMDL mdl = GetWritablePage(InlinePoint,16,&MappedAddress);

if(!mdl)

return STATUS\_UNSUCCESSFUL;

ULONG data[3] = {InlinePoint,JmpToFunc,MappedAddress};

status = MakeJmp(data,DoHookK);

MmUnlockPages(mdl);

IoFreeMdl(mdl);

return status;

}

NTSTATUS DoHook(ULONG\* data)

{//HOOKTYPE\_INLINE

ULONGLONG shellcode;

if(!data)

return STATUS\_UNSUCCESSFUL;

\*(ULONGLONG\*)(HookInfo1.OriginCode) = \*(ULONGLONG\*)data[0];//保存原始8字节

\*(ULONG\*)((UCHAR\*)&shellcode+4) = data[1] - data[0] - 8;//填写偏移 jmp offset

\*(ULONG\*)((UCHAR\*)&shellcode) = 0xE9909090;//写入新8字节

/\*

构造

nop

nop

nop

jmp ????

\*/

InterlockedCompareExchange64((LONGLONG\*)data[2],shellcode,\*(LONGLONG\*)data[2]);

return STATUS\_SUCCESS;

}

NTSTATUS DoHookK(ULONG\* data)

{//HOOKTYPE\_KSINLINE

ULONGLONG shellcode;

if(!data)

return STATUS\_UNSUCCESSFUL;

\*(ULONGLONG\*)(HookInfo1.OriginCode) = \*(ULONGLONG\*)data[0];//保存原始8字节

\*(USHORT\*)((UCHAR\*)&shellcode+6) = 0x0675;

\*(ULONG\*)((UCHAR\*)&shellcode+2) = data[1] - data[0] - 8;//填写偏移 jmp offset

\*(ULONG\*)((UCHAR\*)&shellcode) = 0xE990;//写入新8字节

InterlockedCompareExchange64((LONGLONG\*)data[2],shellcode,\*(LONGLONG\*)data[2]);

return STATUS\_SUCCESS;

}

PMDL GetWritablePage(PVOID VirtualAddress,ULONG Length,PVOID\* pMappedAddress)

{

PMDL mdl = IoAllocateMdl(VirtualAddress,Length,FALSE,TRUE,NULL);

if(mdl)

{

MmProbeAndLockPages(mdl,KernelMode,IoWriteAccess);

PVOID NewAddr = MmGetSystemAddressForMdlSafe (Mdl, NormalPagePriority);

if(!NewAddr)

{

MmUnlockPages(mdl);

IoFreeMdl(mdl);

mdl = NULL;

}

if(NewAddr)

\*pMappedAddress = NewAddr;

}

return mdl;

}

### 4.7 构造InlineHook跳转后的执行语句

ULONG Filter(ULONG ServiceIndex,ULONG OriginFunc,ULONG Base)

{

if(ServiceIndex >= 1024)

return OriginFunc;

ULONG TableType;

if(Base == SSDTBase && ServiceIndex <= SSDTLimit)

{

TableType = SSDT;

}

else if(KeServiceDescriptorTable && KeServiceDescriptorTable->Base == Base && ServiceIndex <= SSDTLimit)

{

TableType = SSDT;

}

else if(Base == ShadowSSDTBase && ServiceIndex <= ShadowSSDTLimit)

{

TableType = SSSDT;

}

else

{

return OriginFunc;

}

if(ServiceMapTable[TableType][ServiceIndex])

{

ULONG NewAddr = ServiceMapTable[TableType][ServiceIndex]->ProxyFuncAddr;

if(NewAddr)

{

ServiceMapTable[TableType][ServiceIndex]->OriginFuncAddr = OriginFunc;

return NewAddr;

}

}

return OriginFunc;

}

void \_\_declspec(naked) InlineKiFastCallEntry1()

{

/\* HOOKTYPE\_INLINE >=VISTA

输入

eax=ServiceIndex

edx=OriginFunc

edi=SSDTBase

输出

edx=FuncAddr 最终函数调用地址

\*/

\_asm

{

pushf;

pusha;

push edi;

push edx;

push eax;

call Filter;

mov [esp+0x14],eax;//改栈中的edx

popa;

popf;

mov edi,esp;

cmp esi,MmUserProbeAddress;

push dword ptr HookInfo1.JmpBackPoint;

retn;//跳回JmpBackPoint

}

}

void \_\_declspec(naked) InlineKiFastCallEntry2()

{

/\* HOOKTYPE\_INLINE <VISTA

输入

eax=ServiceIndex

edx=OriginFunc

edi=SSDTBase

输出

ebx=FuncAddr 最终函数调用地址

\*/

\_asm

{

pushf;

pusha;

push edi;

push edx;

push eax;

call Filter;

mov [esp+0x10],eax;//改栈中的ebx

popa;

popf;

mov edi,esp;

cmp esi,MmUserProbeAddress;

push dword ptr HookInfo1.JmpBackPoint;

retn;

}

}

void \_\_declspec(naked) InlineKiFastCallEntry3()

{

/\* HOOKTYPE\_KSINLINE <VISTA

输入

eax=ServiceIndex

edx=OriginFunc

edi=SSDTBase

输出

edx=FuncAddr 最终函数调用地址 (KiFastCallEntry后面mov ebx,edx)

\*/

\_asm

{

pushf;

pusha;

push edi;

push edx;

push eax;

call Filter;

mov [esp+0x14],eax;//改栈中的edx

popa;

popf;

mov edi,esp;

test byte ptr [ebp+0x72],2;

push dword ptr HookInfo2.JmpBackPoint;

retn;//跳回JmpBackPoint

}

}

void \_\_declspec(naked) InlineKiFastCallEntry4()

{

/\* HOOKTYPE\_KSINLINE <VISTA

输入

eax=ServiceIndex

edx=OriginFunc

edi=SSDTBase

输出

ebx=FuncAddr 最终函数调用地址

\*/

\_asm

{

pushf;

pusha;

push edi;

push edx;

push eax;

call Filter;

mov [esp+0x14],eax;//改栈中的edx

popa;

popf;

mov edi,esp;

test byte ptr [ebp+0x72],2;

push dword ptr HookInfo2.JmpBackPoint;

retn;//跳回JmpBackPoint

}

}

### 4.8 强制单核互斥执行指定Procedure

struct SpinData

{

KSPIN\_LOCK SpinLock;

ULONG SpinRefCount;

}g\_Spin = {0};

KDPC Dpc[32];

void DpcRoutine(PKDPC Dpc,PVOID DeferredContext,PVOID SystemArgument1,PVOID SystemArgument2)

{

KIRQL OldIrql;

SpinData\* spin = (SpinData\*)DeferredContext;zh

\_disable();

OldIrql = KeRaiseIrqlToDpcLevel();

InterlockedIncrement(spin->SpinRefCount);

KeAcquireSpinLockAtDpcLevel(spin->SpinLock);

KeReleaseSpinLockFromDpcLevel(spin->SpinLock);

KeLowerIrql(OldIrql);

\_enable();

}

NTSTATUS MakeJmp(ULONG\* data,FARPROC addr)

{

NTSTATUS status = STATUS\_UNSUCCESSFUL;

ULONG ulCurrentCpu = 0;

KIRQL OldIrql,NewIrql;

KAFFINITY CpuAffinity;

if(!addr)

return status;

CpuAffinity = KeQueryActiveProcessors();

for(int i = 0;i < 32;i++)

{

if((CpuAffinity >> i) & 1)

ulCurrentCpu++;

}

if(ulCurrentCpu == 1)//单核

{

\_disable();

OldIrql = KeRaiseIrqlToDpcLevel();

status = addr(data);

KeLowerIrql(OldIrql);

\_enable();

}

else//多核 将除当前cpu以外的cpu用自旋锁锁住

{

SpinData\* pSpinData = &g\_Spin;

ULONG ulNumberOfActiveCpu = 0;

KeInitializeSpinLock(&g\_Spin.SpinLock);

for(int i = 0;i < 32;i++)

{

KeInitializeDpc(&Dpc[i],DpcRoutine,&pSpinData);

}

pSpinData->SpinRefCount = 0;

\_disable();

NewIrql = KeAcquireSpinLock(&g\_Spin.SpinLock);

ulCurrentCpu = KeGetCurrentProcessorNumber();

for(int i = 0;i < 32;i++)

{

if((CpuAffinity >> i) & 1)

ulNumberOfActiveCpu++;

if(i != ulCurrentCpu)

{

KeSetTargetProcessorDpc(Dpc[i],i);

KeSetImportanceDpc(Dpc[i],HighImportance);

KeInsertQueueDpc(Dpc[i],NULL,NULL);

}

KeInitializeDpc(&Dpc[i],DpcRoutine,&pSpinData);

}

//此时只有一个核在工作

for(int i = 0;i < 16;i++)//尝试16次

{

for(int count = 1000000;count > 0;count--);//延时

if(g\_Spin.SpinRefCount == ulNumberOfActiveCpu - 1)//等待DpcRoutine全部执行到死锁

{

status = addr(data);

break;

}

}

KeReleaseSpinLock(&g\_Spin.SpinLock,NewIrql);//恢复多核运行

\_enable();

}

return status;

}

### 4.9 进行SSDT hook

在注册表Services\\TsFltMgr thm=1的情况下，或者Inline hook KiFastCallEntry失败的情况下，会进行SSDT表 hook

NTSTATUS BeginSSDTHook()

{

NTSTATUS status = STATUS\_UNSUCCESSFUL;

int result = -1;

PVOID MappedAddress = NULL;

PMDL mdl = NULL;

if(serHookType == HOOKTYPE\_INLINE || serHookType == HOOKTYPE\_KSINLINE)//已成功inline hook

return STATUS\_UNSUCCESSFUL;

serHookType = HOOKTYPE\_SSDT;

for(int i = 0;i <= APINUMBER;i++)

{

if(DProxyTable[i] && DProxyTable[i]->ServiceIndex != -1 && !DProxyTable[i]->IsInitialized)

{

if(DProxyTable[i]->ServiceTableType == SSDT)

{

int index = DProxyTable[i]->ServiceIndex;

if(index != 1023 && DProxyTable[i]->ProxyFunc && SSDTBase)

{

PVOID MappedAddress = NULL;

PMDL mdl = GetWritablePage(SSDTBase,4\*SSDTLimit,&MappedAddress);

if(mdl)

{

DProxyTable[i]->OriginFuncAddr = ((ULONG\*)SSDTBase)[index];

((ULONG\*)MappedAddress)[index] = DProxyTable[i]->ProxyFunc;

MmUnlockPages(mdl);

IoFreeMdl(mdl);

result = index;

}

}

}

else if(DProxyTable[i]->ServiceTableType == SSSDT)

{

int index = DProxyTable[i]->ServiceIndex;

if(index != 1023 && DProxyTable[i]->ProxyFunc && ShadowSSDTBase)

{

HANDLE CurProcessId = PsGetCurrentProcessId();

HANDLE SmssProcessId = GetProcessIdByName(L"smss.exe");

if(!dwcsrssId)//如果没有获取到csrss.exe的id

{

if(CurProcessId == SmssProcessId)//使用smss.exe的SSSDT

{

mdl = GetWritablePage(ShadowSSDTBase,4\*ShadowSSDTLimit,&MappedAddress);

if(mdl)

{

DProxyTable[i]->OriginFuncAddr = ((ULONG\*)ShadowSSDTBase)[index];

((ULONG\*)MappedAddress)[index] = DProxyTable[i]->ProxyFunc;

MmUnlockPages(mdl);

IoFreeMdl(mdl);

result = index;

}

}

}

else

{//附加到csrss.exe

PVOID attachobj = AttachDriverToProcess(dwcsrssId);

if(attachobj)

{

mdl = GetWritablePage(ShadowSSDTBase,4\*ShadowSSDTLimit,&MappedAddress);

if(mdl)

{

DProxyTable[i]->OriginFuncAddr = ((ULONG\*)ShadowSSDTBase)[index];

((ULONG\*)MappedAddress)[index] = DProxyTable[i]->ProxyFunc;

MmUnlockPages(mdl);

IoFreeMdl(mdl);

result = index;

}

DetachDriverFromProcess(attachobj);

}

}

}

}

if(result != -1)

DProxyTable[i]->IsInitialized = TRUE;

}

}

return STATUS\_SUCCESS;

}

struct AttachData

{

KAPC\_STATE ApcState;

PVOID Process;

};

PVOID AttachDriverToProcess(HANDLE ProcessId)

{

PVOID Process = NULL;

AttachData\* Buffer = NULL;

if(ProcessId == PsGetCurrentProcessId())

return 0xEEEEDDDD;//自身标识

if(ProcessId && NT\_SUCCESS(PsLookupProcessByProcessId(ProcessId,&Process)))

{

Buffer = (AttachData\*)ExAllocatePool(PagedPool,sizeof(AttachData));

if(!Buffer)

{

ObDereferenceObject(Process);

return NULL;

}

Buffer->Process = Process;

KeStackAttachProcess(Process,&Buffer->ApcState)

}

}

NTSTATUS DetachDriverFromProcess(PVOID Buffer)

{

if(Buffer && (ULONG)Buffer != 0xEEEEDDDD)

{

AttachData\* data = (AttachData\*)Buffer;

KeUnstackDetachProcess(&data->ApcState);

ObDereferenceObject(data->Process);

ExFreePool(Buffer);

}

}

### 4.10 对重要回调(进程回调、线程回调、映像加载回调)的挂钩

/\*

Sproxy Index：

普通api使用index 0-0x33 0x37-0x48 0x4A-0x69

52 CreateNotifyRoutine

53 ThreadNofify

54 ImageNotify

73 CreateNotifyRoutine2

\*/

enum

{

INDEX\_KECALLBACK = 51,

INDEX\_PROCESS = 52,

INDEX\_THREAD = 53,

INDEX\_IMAGE = 54,

INDEX\_PROCESSEX = 73,

};

NTSTATUS HookImportantNotify()

{

if(!DProxyTable[INDEX\_PROCESS])

{

DProxy\* Proxydata = (DProxy\*)ExAllocatePool(NonPagedPool,sizeof(DProxy));

if(Proxydata)

{

Proxydata->IsInitialized = FALSE;

Proxydata->ApiName = "ProcessNotify";

Proxydata->TableIndex = INDEX\_PROCESS;

Proxydata->ProxyFuncAddr = CreateNotifyRoutine1;

Proxydata->ServiceIndex = -1;

KeInitializeEvent(&Proxydata->Lock,SynchronizationEvent,FALSE);

DProxyTable[INDEX\_PROCESS] = Proxydata;

}

}

if(DProxyTable[INDEX\_PROCESS] && !DProxyTable[INDEX\_PROCESS]->IsInitialized)

{

DProxyTable[INDEX\_PROCESS]->ServiceTableType = CALLBACK;

if(NT\_SUCCESS(PsSetCreateProcessNotifyRoutine(CreateProcessNotify,FALSE)))

DProxyTable[INDEX\_PROCESS]->IsInitialized = TRUE;

}

if(!DProxyTable[INDEX\_PROCESSEX])

{

DProxy\* Proxydata = (DProxy\*)ExAllocatePool(NonPagedPool,sizeof(DProxy));

if(Proxydata)

{

Proxydata->IsInitialized = FALSE;

Proxydata->ApiName = "ProcessNotifyEx";

Proxydata->TableIndex = INDEX\_PROCESSEX;

Proxydata->ProxyFuncAddr = CreateNotifyRoutine1Ex;

Proxydata->ServiceIndex = -1;

KeInitializeEvent(&Proxydata->Lock,SynchronizationEvent,FALSE);

DProxyTable[INDEX\_PROCESSEX] = Proxydata;

}

}

if(DProxyTable[INDEX\_PROCESSEX] && !DProxyTable[INDEX\_PROCESSEX]->IsInitialized)

{

DProxyTable[INDEX\_PROCESSEX]->ServiceTableType = CALLBACK;

if(NT\_SUCCESS(PsSetCreateProcessNotifyRoutineEx(CreateProcessNotifyEx,FALSE)))

DProxyTable[INDEX\_PROCESSEX]->IsInitialized = TRUE;

}

if(!DProxyTable[INDEX\_THREAD])

{

DProxy\* Proxydata = (DProxy\*)ExAllocatePool(NonPagedPool,sizeof(DProxy));

if(Proxydata)

{

Proxydata->IsInitialized = FALSE;

Proxydata->ApiName = "ThreadNotify";

Proxydata->TableIndex = INDEX\_THREAD;

Proxydata->ProxyFuncAddr = CreateThreadNotify;

Proxydata->ServiceIndex = -1;

KeInitializeEvent(&Proxydata->Lock,SynchronizationEvent,FALSE);

DProxyTable[INDEX\_THREAD] = Proxydata;

}

}

if(DProxyTable[INDEX\_THREAD] && !DProxyTable[INDEX\_THREAD]->IsInitialized)

{

DProxyTable[INDEX\_THREAD]->ServiceTableType = CALLBACK;

if(NT\_SUCCESS(PsSetCreateThreadNotifyRoutine(CreateThreadNotify,FALSE)))

DProxyTable[INDEX\_THREAD]->IsInitialized = TRUE;

}

if(!DProxyTable[INDEX\_IMAGE])

{

DProxy\* Proxydata = (DProxy\*)ExAllocatePool(NonPagedPool,sizeof(DProxy));

if(Proxydata)

{

Proxydata->IsInitialized = FALSE;

Proxydata->ApiName = "ImageNotify";

Proxydata->TableIndex = INDEX\_IMAGE;

Proxydata->ProxyFuncAddr = LoadImageNotify;

Proxydata->ServiceIndex = -1;

KeInitializeEvent(&Proxydata->Lock,SynchronizationEvent,FALSE);

DProxyTable[INDEX\_IMAGE] = Proxydata;

}

}

if(DProxyTable[INDEX\_IMAGE] && !DProxyTable[INDEX\_IMAGE]->IsInitialized)

{

DProxyTable[INDEX\_IMAGE]->ServiceTableType = CALLBACK;

if(NT\_SUCCESS(PsSetLoadImageNotifyRoutine(LoadImageNotify,FALSE)))

DProxyTable[INDEX\_IMAGE]->IsInitialized = TRUE;

}

}

### 4.11 Hook KeUserModeCallback

NTSTATUS HookKeUserModeCallback()

{

NTSTATUS status = STATUS\_UNSUCCESSFUL;

if(!Win32kBase)

return status;

OBJECT\_ATTRIBUTES Oa;

IO\_STATUS\_BLOCK IoStatusBlock;

HANDLE SectionHandle = NULL;

ULONG ViewSize = 0;

HANDLE FileHandle = NULL;

PVOID BaseAddress = 0x5000000;

UNICODE\_STRING UKeUserModeCallback;

if(!DProxyTable[INDEX\_KECALLBACK])

{

DProxy\* Proxydata = (DProxy\*)ExAllocatePool(NonPagedPool,sizeof(DProxy));

if(Proxydata)

{

Proxydata->IsInitialized = FALSE;

Proxydata->ApiName = "ImageNotify";

Proxydata->TableIndex = INDEX\_KECALLBACK;

Proxydata->ProxyFuncAddr = ProxyKeUserModeCallback;

Proxydata->ServiceIndex = -1;

KeInitializeEvent(&Proxydata->Lock,SynchronizationEvent,FALSE);

DProxyTable[INDEX\_KECALLBACK] = Proxydata;

}

}

RtlInitUnicodeString(&UKeUserModeCallback,L"\\SystemRoot\\System32\\Win32k.sys");

InitializeObjectAttributes(&Oa,L"KeUserModeCallback",OBJ\_CASE\_INSENSITIVE|OBJ\_KERNEL\_HANDLE,NULL,NULL);

status = ZwOpenFile(&FileHandle,GENERIC\_READ,&Oa,&IoStatusBlock,FILE\_SHARE\_READ,FILE\_SYNCHRONOUS\_IO\_NONALERT);

if(NT\_SUCCESS(status))

{

Oa.ObjectName = NULL;

status = ZwCreateSection(&SectionHandle,SECTION\_MAP\_EXECUTE,&Oa,NULL,PAGE\_WRITECOPY,SEC\_IMAGE);

if(NT\_SUCCESS(status))

{

status = ZwMapViewOfSection(&SectionHandle,NtCurrentProcess(),&BaseAddress,0,0,NULL,&ViewSize,ViewUnmap,0,PAGE\_EXECUTE);

if(!NT\_SUCCESS(status))

{

BaseAddress = NULL;

ZwMapViewOfSection(&SectionHandle,NtCurrentProcess(),&BaseAddress,0,0,NULL,&ViewSize,ViewUnmap,0,PAGE\_EXECUTE);

}

}

}

if(BaseAddress)

{

if(!DProxyTable[INDEX\_KECALLBACK]->IsInitialized)

{

ULONG IatOffset = GetProcOffsetFromIat(BaseAddress);

DProxyTable[INDEX\_KECALLBACK]->OriginFuncAddr = ExchangeMem(IatOffset,Win32kBase,ProxyKeUserModeCallback);

if(DProxyTable[INDEX\_KECALLBACK]->OriginFuncAddr)

{

DProxyTable[INDEX\_KECALLBACK]->IsInitialized;

status = STATUS\_SUCCESS;

}

}

ZwUnmapViewOfSection(NtCurrentProcess(),BaseAddress);

}

return status;

}

### 4.12 交换内存

ULONG ExchangeMem(ULONG Offset,ULONG BaseAddress,ULONG NewValue)

{//替换基址BaseAddress 偏移Offset 处的ULONG数据，返回原始数据

HANDLE CurProcessId = PsGetCurrentProcessId();

HANDLE SmssProcessId = GetProcessIdByName(L"smss.exe");

ULONG OriginValue = 0;

PMDL mdl = NULL;

if(!dwcsrssId)//如果没有获取到csrss.exe的id

{

if(CurProcessId == SmssProcessId)//使用smss.exe的SSSDT

{

mdl = GetWritablePage(BaseAddress+Offset,16,&MappedAddress);

if(mdl)

{

OriginValue = InterlockedExchange(MappedAddress,NewValue);

MmUnlockPages(mdl);

IoFreeMdl(mdl);

}

}

}

else

{//附加到csrss.exe

PVOID attachobj = AttachDriverToProcess(dwcsrssId);

if(attachobj)

{

mdl = GetWritablePage(BaseAddress+Offset,4,&MappedAddress);

if(mdl)

{

OriginValue = InterlockedExchange(MappedAddress,NewValue);

MmUnlockPages(mdl);

IoFreeMdl(mdl);

}

DetachDriverFromProcess(attachobj);

}

}

return OriginValue;

}

### 4.13 获取函数Iat偏移

ULONG GetProcOffsetFromIat(PVOID ImageBase,PCHAR ModuleName,PCHAR ApiName)

{//ImageBase当前进程映像基址 ModuleName导入模块映像基址 ApiName导入函数名

ANSI\_STRING AApiName;

UNICODE\_STRING UApiname;

PIMAGE\_IMPORT\_DESCRIPTOR ImportModuleDirectory;

ULONG ObjProcAddr;

ULONG Offset = 0;

PCHAR ImportedName;

PIMAGE\_THUNK\_DATA FunctionNameList;

RtlInitAnsiString(&AApiName,ApiName);

if(!NT\_SUCCESS(RtlAnsiStringToUnicodeString(&UApiname,&AApiName,TRUE)))

return 0;

ObjProcAddr = MmGetSystemRoutineAddress(&UApiname);

RtlFreeUnicodeString(&UApiname);

ImportModuleDirectory = (PIMAGE\_IMPORT\_DESCRIPTOR)RtlImageDirectoryEntryToData(ImageBase,TRUE,IMAGE\_DIRECTORY\_ENTRY\_IMPORT);

if(!ImportModuleDirectory)

return 0;

for (;(ImportModuleDirectory->Name != 0) && (ImportModuleDirectory->FirstThunk != 0);ImportModuleDirectory++)

{

ImportedName = (PCHAR)ImageBase + ImportModuleDirectory->Name;

if(!stricmp(ModuleName,ImportedName))

{

FunctionNameList = (PIMAGE\_THUNK\_DATA)((UCHAR\*)ImageBase+ImportModuleDirectory->FirstThunk);

while(\*FunctionNameList != 0)

{

if((\*FunctionNameList) & 0x80000000)

{

if(FunctionNameList->u1.Function == ObjProcAddr)

return (UCHAR\*)FunctionNameList - (UCHAR\*)ImageBase;

}

else

{

IMAGE\_IMPORT\_BY\_NAME \*pe\_name = (IMAGE\_IMPORT\_BY\_NAME\*)((PCHAR)ImageBase + \*FunctionNameList);

if(pe\_name->Name[0] == 'K' && !stricmp(pe\_name->Name,"KeUserModeCallback"))

return (UCHAR\*)FunctionNameList - (UCHAR\*)ImageBase;

}

FunctionNameList++;

}

}

}

return 0;

}

### 4.14 另一种方式获取ShadowSSDT信息

当常规方式获取SSSDT失败时，会临时设置ZwSetSystemInformation的过滤函数捕获系统设置服务表（SYSTEM\_INFORMATION\_CLASS =SystemExtendServiceTableInformation），

void TryAnotherWayToGetShadowSSDT()

{

AddPrevFilter(EZwSetSystemInformation,ZwSetSystemInformationFilter1,0,0,NULL);

}

ULONG OldKeAddSystemServiceTable;

ULONG \_\_stdcall ZwSetSystemInformationFilter1(FilterPacket\* Packet)

{//改EAT以获取SSSDT

if(!ShadowSSDTBase && Packet->Params[0] == SystemExtendServiceTableInformation && !OldKeAddSystemServiceTable)

{

OldKeAddSystemServiceTable = ExchangeEat(NtosBase,NewKeAddSystemServiceTable);

if(OldKeAddSystemServiceTable)

{

Packet->TagSlot[Packet->CurrentSlot] = 0xABCDEEEE;//用于标记

Packet->PostFilterSlot[Packet->CurrentSlot] = ZwSetSystemInformationFilter2;

Packet->UsedSlotCount++;

}

}

return SYSMON\_PASSTONEXT;

}

ULONG \_\_stdcall ZwSetSystemInformationFilter2(FilterPacket\* Packet)

{//获取之后恢复EAT

if(Packet->TagSlot[Packet->CurrentSlot] == 0xABCDEEEE)//检查标记

{

if(serHookType != 1 && serHookType != 3)//是否SSDT型Hook

{

BeginSSDTHook();

}

}

return SYSMON\_PASSTONEXT;

}

BOOLEAN \_\_stdcall KeAddSystemServiceTable(PULONG\_PTR Base,PULONG Count,ULONG Limit,PUCHAR Number,ULONG Index)

{//获取SSSDT

RTL\_PROCESS\_MODULE\_INFORMATION ProcessInfo;

RtlZeroMemory(&ProcessInfo,sizeof(ProcessInfo));

if(!ShadowSSDTBase && GetProcessInfoByFileName("win32k.sys",&ProcessInfo) &&

Base >= ProcessInfo.ImageBase && Base <= ProcessInfo.ImageBase+ProcessInfo.ImageSize)

{

Win32kBase = ProcessInfo.ImageBase;

Win32kSize = ProcessInfo.ImageSize;

ShadowSSDTBase = Base;

ShadowSSDTLimit = Limit;

ExchangeEat(NtosBase,OldKeAddSystemServiceTable);

}

return OldKeAddSystemServiceTable(Base,Count,Limit,Number,Index);

}

ULONG ExchangeEat(ULONG ImageBase,ULONG NewFunc)

{

PVOID Address = MiLocateExportName(ImageBase,"KeAddSystemServiceTable");

if(Address)

{

return ExchangeMem(0,Address,NewFunc);

}

return 0;

}