[原创]Windows最全反调试知识汇总-附实现代码 优

小曾HanFeng

极客

2020-9-21 13:17

3699

今天浅谈一些Windows中的一系列反调试技术和大家分享

NTDLL：反调试中需要包含NtDll头文件

以上是包含的格式!

```
#include <iostream>
#include <stdio.h>
#include <Windows.h>
#include "crc32.h"
#include <vector>

#include "ntdll/ntdll.h"
#pragma comment (lib,"ntdll/ntdll_x86.lib")
```

PEB(进程环境块):
概述：在你进行打开这个程序或以调试方式打开这个程序、那么操作系统会对你这个进程的进程环境块的一些标志设置一系列的属性
调试器：如果你使用系统的调试方法、那么系统就会把相对应的标志位给设置上

1. PEB有四个标志位
2. 在fs : [0x30]这处地方就是PEB的指针 需要用*PEB指针进行接收

PEB中有四个标志位（检测是否被调试）：
PEB
{
BeingDebugger:1
NtGlobalFlag:0x70
HeapFlags:=2 offset (x32) 0x40!=2 则被调试状态 0x70!=0则被调试状态
ForceFlags:=0 offset (x32)0x44!=2 则被调试状态 0x74
}

```
mov eax,dword ptr fs:[0x30]
```

PEB的结构

```
typedef struct _PEB
{
    BOOLEAN InheritedAddressSpace;
    BOOLEAN ReadImageFileExecOptions;
    BOOLEAN BeingDebugged;
    union
    {
        BOOLEAN BitField;
        struct
        {
            BOOLEAN ImageUsesLargePages : 1;
            BOOLEAN IsProtectedProcess : 1;
            BOOLEAN IsImageDynamicallyRelocated : 1;
            BOOLEAN SkipPatchingUser32Forwarders : 1;
            BOOLEAN IsPackagedProcess : 1;
            BOOLEAN IsAppContainer : 1;
            BOOLEAN IsProtectedProcessLight : 1;
            BOOLEAN IsLongPathAwareProcess : 1;
        } s1;
    } u1;

    HANDLE Mutant;

```

```
    PVOID SubSystemData;
    PVOID ProcessHeap;
    PRTL_CRITICAL_SECTION FastPebLock;
    PVOID AtlThunkSListPtr;
    PVOID IFEOKey;
    union
    {
        ULONG CrossProcessFlags;
        struct
        {
            ULONG ProcessInJob : 1;
            ULONG ProcessInitializing : 1;
            ULONG ProcessUsingVEH : 1;
            ULONG ProcessUsingVCH : 1;
            ULONG ProcessUsingFTH : 1;
            ULONG ProcessPreviouslyThrottled : 1;
            ULONG ProcessCurrentlyThrottled : 1;
            ULONG ReservedBits0 : 25;
        } s2;
    } u2;
    union
    {
        PVOID KernelCallbackTable;
        PVOID UserSharedInfoPtr;
    } u3;
    ULONG SystemReserved[1];
    ULONG AtlThunkSListPtr32;
    PVOID ApiSetMap;
    ULONG TlsExpansionCounter;
    PVOID TlsBitmap;
    ULONG TlsBitmapBits[2];
    PVOID ReadOnlySharedMemoryBase;
    PVOID HotpatchInformation;
    PVOID *ReadOnlyStaticServerData;
    PVOID AnsiCodePageData;
    PVOID OemCodePageData;
    PVOID UnicodeCaseTableData;

    ULONG NumberOfProcessors;
    ULONG NtGlobalFlag;

    LARGE_INTEGER CriticalSectionTimeout;
    SIZE_T HeapSegmentReserve;
    SIZE_T HeapSegmentCommit;
    SIZE_T HeapDeCommitTotalFreeThreshold;
    SIZE_T HeapDeCommitFreeBlockThreshold;

    ULONG NumberOfHeaps;
    ULONG MaximumNumberOfHeaps;
    PVOID *ProcessHeaps;

    PVOID GdiSharedHandleTable;
    PVOID ProcessStarterHelper;
    ULONG GdiDCAttributeList;

    PRTL_CRITICAL_SECTION LoaderLock;

    ULONG OSMajorVersion;
    ULONG OSMinorVersion;
    USHORT OSBuildNumber;
    USHORT OSCSDVersion;
    ULONG OSPlatformId;
    ULONG ImageSubsystem;
    ULONG ImageSubsystemMajorVersion;
    ULONG ImageSubsystemMinorVersion;
    ULONG_PTR ActiveProcessAffinityMask;
    GDI_HANDLE_BUFFER GdiHandleBuffer;
    PVOID PostProcessInitRoutine;

    PVOID TlsExpansionBitmap;
    ULONG TlsExpansionBitmapBits[32];

    ULONG SessionId;

    ULARGE_INTEGER AppCompatFlags;
    ULARGE_INTEGER AppCompatFlagsUser;
    PVOID pShimData;
    PVOID AppCompatInfo;

    UNICODE_STRING CSDVersion;

    PVOID ActivationContextData;
    PVOID ProcessAssemblyStorageMap;
    PVOID SystemDefaultActivationContextData;
    PVOID SystemAssemblyStorageMap;

    SIZE_T MinimumStackCommit;

    PVOID *FlsCallback;
    LIST_ENTRY FlsListHead;
    PVOID FlsBitmap;
```

```
    PVOID WerRegistrationData;
    PVOID WerShipAssertPtr;
    PVOID pContextData;
    PVOID pImageHeaderHash;
    union
    {
        ULONG TracingFlags;
        struct
        {
            ULONG HeapTracingEnabled : 1;
            ULONG CritSecTracingEnabled : 1;
            ULONG LibLoaderTracingEnabled : 1;
            ULONG SpareTracingBits : 29;
        } s3;
    } u4;
    ULONGLONG CsrServerReadOnlySharedMemoryBase;
    PVOID TppWorkerpListLock;
    LIST_ENTRY TppWorkerpList;
    PVOID WaitOnAddressHashTable[128];
} PEB, *PPEB;
```


**如果PEB的BeingDebugger为True的话则说明被调试!

(NtGlobalFlag & 0x70) == True 则说明被调试状态**

IsDebuggerPresent:

这个API的原理就是获取TEB (fs:0x18)然后再获取PEB然后再获取PEB的BeingDebugger成员、如果为1则有调试

```
//WinApi IsDebuggerPresent 如果返回值为True则为调试状态 如果为False则为未调试状态
bool bRet = false;

bRet = IsDebuggerPresent();
```

WinApi：IsDebuggerPresent() 检测是被调试、如果为返回值为True则为调试状态、如果为False则为没有被调试

CheckRemoteDebuggerPresent :

概述：检查远程调试器

原理：原理用NtQueryInformationProcess检查ProcessDebugPort这个属性

```
WINBASEAPI
BOOL
WINAPI
CheckRemoteDebuggerPresent(
    _In_ HANDLE hProcess,
    _Out_ PBOOL pbDebuggerPresent
    );
```

参数一：HANDLE hProcess 句柄

参数二：PBOOL pbDebuggerPresent 指向是否被调试的指针、如果返回值为true则为调试状态、如果为false则为未调试状态

```
BOOL bRet = FALSE;
//参数一: HANDLE hProcess        句柄
//参数二: PBOOL pbDebuggerPresent    指向是否被调试的指针、如果返回值为true则为调试状态、如果为false则
CheckRemoteDebuggerPresent(NtCurrentProcess,
    &bRet
);

OUTPRINTF("CheckRemoteDebuggerPresent(检查远程调试器)", bRet);
```

NtQuerySystemInformation

这个原理就是查询SystemKernelDebuggerInformation的信息

这个API函数检测当前系统是否正在调试状态

这个函数可以判断当前这个系统是否被内核调试器给附加或者是给调试状态、比如双机调试

```
NTSYSCALLAPI
NTSTATUS
NTAPI
NtQuerySystemInformation(
    _In_ SYSTEM_INFORMATION_CLASS SystemInformationClass,
    _Out_opt_ PVOID SystemInformation,
    _In_ ULONG SystemInformationLength,
    _Out_opt_ PULONG ReturnLength
    );
```

参数一：SystemKernelDebuggerInformation

参数二：指向SYSTEM_KERNEL_DEBUGGER_INFORMATION结构体的指针

参数三：SYSTEM_KERNEL_DEBUGGER_INFORMATION结构体的长度

参数四：返回一个长度

返回值：应用层所有Nt函数成功都是返回0、和驱动层是反过来的、内核层成功返回1、所有可以用NT_SUCCESS这个宏来判断Nt函数是否被调用成功

```
NTSTATUS ntStatus = 0;
SYSTEM_KERNEL_DEBUGGER_INFORMATION pSystemKernelDebuggerInformation;
ULONG uRetLength = 0;

ntStatus =
NtQuerySystemInformation(SystemKernelDebuggerInformation,&pSystemKernelDebuggerInformation,si
th);
  //参数一：SystemKernelDebuggerInformation
 //参数二：指向SYSTEM_KERNEL_DEBUGGER_INFORMATION结构体的指针
 //参数三：SYSTEM_KERNEL_DEBUGGER_INFORMATION结构体的长度
 //参数四：返回一个长度
 //返回值：应用层、如果Nt函数成功一般都是返回0、和驱动层是反过来的、内核层成功返回1、所有可以用NT_SUCCESS
if (NT_SUCCESS(ntStatus))
{
    OUTPRINTF("NtQuerySystemInformation(检测内核调试器)", !pSystemKernelDebuggerInformation.Ker
    //KernelDebuggerEnabled：是否激活、一般为0
    //KernelDebuggerNotPresent ： 没有内核调试器附加则为 1、所以这里需要取反操作
}
```

typedef struct _SYSTEM_KERNEL_DEBUGGER_INFORMATION

{

BOOLEAN KernelDebuggerEnabled;

BOOLEAN KernelDebuggerNotPresent;

} SYSTEM_KERNEL_DEBUGGER_INFORMATION, *PSYSTEM_KERNEL_DEBUGGER_INFORMATION;

KernelDebuggerEnabled：是否激活、一般为0

KernelDebuggerNotPresent:没有内核调试器附加则为 True(1)

NtClose

其实就是一个CloseHandle、如果有调试器的情况下关闭一个无效的句柄则会触发一个异常、可以用VEH进行接收并处理

如果有调试器存在的话NtClose就会触发一个异常、则可以捕获这个异常
来判断是否被调试器调试状态

NtClose

参数一：*In* HANDLE Handle 句柄、这里需要用到ULongToHandle()进行转换

```
BOOL bRet = FALSE;
__try
{
    NtClose(ULongToHandle(0x1234));
}
__except (EXCEPTION_EXECUTE_HANDLER)//接受句柄异常
{
    bRet = TRUE;
}
OUTPRINTF("NtClose(句柄异常方式)", bRet);
```

NtQueryInformationProcess

这个函数是像系统查询环境块的属性、仅仅是查询的作用、一些调试信息保存在操作系统内核、所有用这个函数对它进行查询就行
这个是加密壳里用到最多的反调试函数、这个函数非常重要

函数的定义：

```
NTSYSCALLAPI
NTSTATUS
NTAPI
NtQueryInformationProcess(
    _In_ HANDLE ProcessHandle,
    _In_ PROCESSINFOCLASS ProcessInformationClass,
    _Out_ PVOID ProcessInformation,
    _In_ ULONG ProcessInformationLength,
    _Out_opt_ PULONG ReturnLength
```

参数一：当前窗口句柄
参数二：可以指定ProcessDebugPort、ProcessDebugObjectHandle、ProcessDebugFlags
参数三：接收进程信息的指针
参数四：指针的长度 比如sizeof(PVOID)
参数五：返回长度
返回值：成功返回0 失败返回1

```
PVOID pInfo;
ULONG uRetLength = 0;
NTSTATUS ntStatus = 0;
ntStatus =  NtQueryInformationProcess(NtCurrentProcess, ProcessDebugPort, &pInfo, sizeof(pInt
if (NT_SUCCESS(ntStatus))
{
    //如果ProcessInformation != NULL 、说明ProcessInformation不为0的情况下的情况下表示当前正在被调试
    OUTPRINTF("NtQueryInformationProcess::ProcessDebugPort(常见壳反调试)", pInfo != NULL);

}
```

注意一：
如果参数二等于ProcessDebugPort、ProcessDebugObjectHandle 则ProcessInformation != NULL (不为0)、的情况下的情况下表示当前正在被调试

```
ntStatus = NtQueryInformationProcess(NtCurrentProcess, ProcessDebugObjectHandle, &pInfo, sizeo

if (ntStatus!=0xC0000353)
{
    //如果是参数二等于ProcessDebugObjectHandle则Nt函数返回值是0xC0000353则当前没有被调试、否则即为调试
    OUTPRINTF("NtQueryInformationProcess::ProcessDebugObjectHandle(常见壳反调试)",true);
}
```

注意二：
如果是参数二等于ProcessDebugObjectHandle则Nt函数返回值是0xC0000353则当前没有被调试、否则即为调试状态

```
ntStatus = NtQueryInformationProcess(NtCurrentProcess, ProcessDebugFlags, &pInfo, sizeof(pInfo
if (NT_SUCCESS(ntStatus))
{
    //ProcessDebugFlags这种的话pInfo == NULL则为被调试状态
    OUTPRINTF("NtQueryInformationProcess::ProcessDebugFlags(常见壳反调试)", (DWORD)pInfo != 1);
}
```

注意三：
如果参数二等于ProcessDebugFlags标志、则ProcessInformation != 1则为被调试状态

NtSetInformationThread

如果成功调用这个API则会分离调试器、无论怎么下断点都断不下!
清除了DebuggerPort、调试器接收不了所有调试事件等
函数定义：

```
NTSYSCALLAPI
NTSTATUS
NTAPI
NtSetInformationThread(
    _In_ HANDLE ThreadHandle,
    _In_ THREADINFOCLASS ThreadInformationClass,
    _In_ PVOID ThreadInformation,
    _In_ ULONG ThreadInformationLength
    );
```

参数一：当前线程的句柄可以设置为NtCurrentThread
参数二：可以设置为ThreadHideFromDebugger标志
参数三：NULL
参数四：NULL

```
NTSTATUS ntStatus;

ntStatus = NtSetInformationThread(NtCurrentProcess, ThreadHideFromDebugger, NULL, NULL);

OUTPRINTF("ThreadHideFromDebugger(分离调试器) ",NT_SUCCESS(ntStatus));
```

NtDuplicateObject

相当于NtClose函数

这个反调试是SE壳商业版用的一个反调试SE壳商业版和普通班相比就多了一个

NtDuplicateObject函数

定义：可以利用句柄进行检测、如果当前为调试状态、利用这个函数复制完成之后然后关闭这个句柄、就会触发一个异常、但使用之前需要调用SetInformationObject进行设置

原理：NtDuplicateObject在内核中内核会检测是否有调试器、有调试器则发出一个异常

函数定义

```
NTSYSCALLAPI
NTSTATUS
NTAPI
NtDuplicateObject(
    _In_ HANDLE SourceProcessHandle,
    _In_ HANDLE SourceHandle,
    _In_opt_ HANDLE TargetProcessHandle,
    _Out_opt_ PHANDLE TargetHandle,
    _In_ ACCESS_MASK DesiredAccess,
    _In_ ULONG HandleAttributes,
    _In_ ULONG Options
    );
```

参数一：原进程的句柄

参数二：目标进程的句柄

参数三：复制到的进程句柄

参数四：用来返回真实的句柄

参数五：NULL

参数六：复制完成句柄操作、可关闭、可不关闭

Inherit=false //不能继承

ProtectFromClose//开启句柄保护

这个函数其实的意思就是当前的进程的句柄复制到当前的进程、然后最后一个属性对它进程关闭

NtQueryObejct

原理：就是查询系统的所有对象

方法：以类型的方式查找调试结构体标识残留

获取所有的长度、可以用它进行反调试操作

函数定义

```
NTSYSCALLAPI
NTSTATUS
NTAPI
NtQueryObject(
    _In_ HANDLE Handle,
    _In_ OBJECT_INFORMATION_CLASS ObjectInformationClass,
    _Out_opt_ PVOID ObjectInformation,
    _In_ ULONG ObjectInformationLength,
    _Out_opt_ PULONG ReturnLength
    );
```

参数一：可以指定当前的句柄 可以指定NtCurrentProcess

参数二：对象信息类结构体表示可以指定 ObjectTypesInformation

参数三：变量接收指针

参数四：变量长度

变量五：变量接收指针

```
typedef struct _OBJECT_TYPES_INFORMATION
{
    ULONG NumberOfTypes;
    OBJECT_TYPE_INFORMATION TypeInformation[1];
} OBJECT_TYPES_INFORMATION, *POBJECT_TYPES_INFORMATION;
```

因为这个结构体的TypeInformation成员是一个软成员、所以需要用内存搜索的方法搜出这个DebugObejct字符串、然后用因为这个字符串在BJECT_TYPE_INFORMATION 的最先所以需要减4 然后得到BJECT_TYPE_INFORMATION 最后一个成员 然后即可反推出这个结构指针

可以使用CONTAINING_RECORD(缓存区、结构体、结构体成员名字) 推出这个结构体的首指针

如果当前有调试器调试这个程序、则

ULONG TotalNumberOfObjects;

ULONG TotalNumberOfHandles;

不为0、否则等于0、可以判断这两个结构体判断当前的进程是否被调试、

注意：如果别的调试器的句柄打开没有关掉、或者是开了调试器有残留信息残留在操作系统、即时调试器没有附加和调试、这两个成员也会出现调试状态、解决办法是重启计算机！(不推荐使用这个反调试)

以下是代码

```
ULONG uRet;
NTSTATUS ntStatus;
PCHAR pBuf;
POBJECT_TYPE_INFORMATION pObjectTypes;
POBJECT_TYPE_INFORMATION pObj;
WCHAR wcszDebugObject[255] = {0};
wcscpy_s(wcszDebugObject, L"DebugObject");
//获取所有类型的长度
ntStatus = NtQueryObject(NtCurrentProcess, ObjectTypesInformation, &uRet, sizeof(uRet), &uRet
//申请的一个空间大小
pBuf = (PCHAR)calloc(1, uRet);
ntStatus = NtQueryObject(NtCurrentProcess, ObjectTypesInformation, pBuf, uRet, &uRet);
pObjectTypes = POBJECT_TYPE_INFORMATION(pBuf);
for (int i = 0; i < uRet - (wcslen(wcszDebugObject)  * sizeof(WCHAR)); i++)
{

    if (memcmp(wcszDebugObject, pBuf, wcslen(wcszDebugObject) * sizeof(WCHAR)) == 0)
    {
        break;
    }

    pBuf++;
}
pBuf -= sizeof(ULONG);
//因为pBuf得到的是POBJECT_TYPE_INFORMATION.DefaultNonPagedPoolCharge的成员指针、所以可以使用一个宏
pObj = CONTAINING_RECORD(pBuf, OBJECT_TYPE_INFORMATION, DefaultNonPagedPoolCharge);
//如果当前有调试器调试这个程序、则
//OBJECT_TYPE_INFORMATION 成员
//ULONG TotalNumberOfObjects;
//ULONG TotalNumberOfHandles;
//不为0、否则等于0、可以判断这两个结构体判断当前的进程是否被调试、

if (pObj->TotalNumberOfObjects || pObj->TotalNumberOfHandles)
{
    OUTPRINTF("pObj->TotalNumberOfObjects || pObj->TotalNumberOfHandles(以类型的方式查找调试结构

}
else
{
    OUTPRINTF("pObj->TotalNumberOfObjects || pObj->TotalNumberOfHandles(以类型的方式查找调试结构
}
//注意：如果别的调试器的句柄打开没有关掉、或者是开了调试器有残留信息残留在操作系统、
//     即时调试器没有附加和调试、这两个成员也会出现调试状态、解决办法是重启计算机！(不推荐使用这个反调试)
```

检测父进程：

NtQueryInformationProcess

参：ProcessBasicInformation

ProcessBasicInformation是用来检测当前的父进程、如果当前的父进程句柄不等于桌面的句柄即为当前为调试状态

NtQueryInformationProcess函数的第二个参数主要用来指定一个类型、

这里用ProcessBasicInformation指定父进程类型

父进程结构体：

```
typedef struct _PROCESS_BASIC_INFORMATION
{
    NTSTATUS ExitStatus;
    PPEB PebBaseAddress;
    ULONG_PTR AffinityMask;
    KPRIORITY BasePriority;
    HANDLE UniqueProcessId;
    HANDLE InheritedFromUniqueProcessId;
} PROCESS_BASIC_INFORMATION, *PPROCESS_BASIC_INFORMATION;
```

其中InheritedFromUniqueProcessId这个就是父进程的ID、如果父进程不等于桌面上打开、则说明你被其他进程所打开！即为调试状态、否则不为调试状态

```
PROCESS_BASIC_INFORMATION Basic;
ULONG uRet = 0;
NTSTATUS ntStatus = 0;
NtQueryInformationProcess(NtCurrentProcess, ProcessBasicInformation, &Basic, sizeof(Basic), &
if (NT_SUCCESS(ntStatus))
{
    //如果当前的进程的父进程不等于桌面进程的句柄、则为反调试
    if (Basic.InheritedFromUniqueProcessId != ULongToHandle(4376))
    {
        OUTPRINTF("Basic.InheritedFromUniqueProcessId(检测父进程是否为桌面)", TRUE);
    }
    else
    {
        OUTPRINTF("Basic.InheritedFromUniqueProcessId(检测父进程是否为桌面)", FALSE);
    }
}
```

枚举进程:

这个太简单了、我这里就不多说了、就是获取全部的进程信息挨个遍历、寻找相关的进程信息或者是进程标题

时间差检测:

RDTSC

GetTickCount

取启动时间 返回是从系统启动到现在所有毫秒数
//从破解者角度观察、因为破解者跟进一个循环肯定得需要时间进行理解、
//当他跳出这个循环再取一个时间、那么这个时间减去上一次的时间如果大于特定的毫秒数则为调试

```
DWORD t1, t2;
//从破解者角度观察、因为破解者跟进一个循环肯定得需要时间进行理解、
//当他跳出这个循环再取一个时间、那么这个时间减去上一次的时间如果大于特定的毫秒数则为调试
t1 =  GetTickCount();
for (size_t i = 0; i < 0x1000000; i++)
{
    __asm
    {
        nop;
        nop;
        nop;
        nop;
    }
}

t2 = GetTickCount();
//假设大于1000毫秒则被下断点
if (t2 - t1 > 1000 )
{

    OUTPRINTF("GetTickCount(时间差检测)", TRUE);
}
else {

    OUTPRINTF("GetTickCount(时间差检测)", FALSE);
}
```

如果在中间进行单步调试、则时间差一定大于1000毫秒、即为调试

禁止键盘输入

BlockInput

在函数头部加上这个禁止键盘输入的函数、然后在函数尾部恢复这个键盘输入、函数执行非常快、所以感受不到键盘有时候被禁止输入！所以这个方法有利于反单步调试(单步单步跟着键盘就失灵了笑死我)、这个可以与时间差反调试进行联合使用!

检测硬件断点:

可以获取当前线程的上下文、当前判断当前的调试寄存器DR0\DIR1\DR2\DIR3是否有值、如果这几个调试寄存器有值说明当前这个进程正在被调试

```
CONTEXT pContext = {0};
pContext.EFlags = CONTEXT_ALL;

if (GetThreadContext(NtCurrentThread, &pContext))
{
    if (pContext.Dr0 || pContext.Dr1 || pContext.Dr2 || pContext.Dr3)
    {
        OUTPRINTF("DR寄存器(检测硬件断点)", TRUE);
    }
}
```

检测硬件断点的地址:

异常方式检测硬件断点

反抗硬件断点调试

HOOK之后首先把DR寄存器全部清0 然后再调用VEH、所以别人用的是你的VEH

HOOK这个函数然后还原之前的硬件断点

```
PUCHAR dwEip = (PUCHAR)ExceptionInfo->ContextRecord->Eip;
//if (*dwEip == 0xCC)
if(ExceptionInfo->ExceptionRecord->ExceptionCode == 0xC0000005)//0xCC就是不可读
{


    if (ExceptionInfo->ContextRecord->Dr0 || ExceptionInfo->ContextRecord->Dr1 || ExceptionIn
>ContextRecord->Dr3)
    {
        OUTPRINTF("DR寄存器(检测到的硬件地址)", TRUE);



    }
    else
    {
        OUTPRINTF("DR寄存器(检测到的硬件地址)", FALSE);
    }


    ExceptionInfo->ContextRecord->Eip += 3;
    return EXCEPTION_CONTINUE_EXECUTION;

}
return EXCEPTION_CONTINUE_SEARCH;
```

自内存CRC:

对抗CRC:下内存硬件断点、然后一步一步跟踪、Nop掉CRC即可

自内存CRC需要很早时期先计算一遍内存CRC校验和!然后后续在根据这个CRC校验值再来判断

```
using namespace std;



typedef struct _CRC_HASHI
{
    PVOID m_pAddr;
    DWORD m_dwSize;
    DWORD m_dwHashVal;
}CRC_HASHI,*PCRC_HASHI;

vector<CRC_HASHI> g_crc_vtr;
```

反附加之前要先获取一次代码段的页面CRC校验和算出CRC、比如某些壳在链接的时候就已经算好了、这是最早的计算方式、越早越好
以下是代码、获取页面CRC

```
HMODULE ImageBase = 0;
ImageBase = GetModuleHandle(NULL);

PIMAGE_DOS_HEADER pDos = NULL;
PIMAGE_NT_HEADERS pNt = NULL;
PIMAGE_SECTION_HEADER pSection = NULL;
DWORD dwStartAdr, dwSize;

pDos = PIMAGE_DOS_HEADER((ULONG_PTR)ImageBase);
if (pDos->e_magic != IMAGE_DOS_SIGNATURE)
{
    return;
}
pNt = PIMAGE_NT_HEADERS((ULONG_PTR)ImageBase + pDos->e_lfanew);
if (pNt->Signature != IMAGE_NT_SIGNATURE)
{
    return;
}

pSection = IMAGE_FIRST_SECTION(pNt);
g_crc_vtr.clear();

for (size_t i = 0; i < pNt->FileHeader.NumberOfSections; i++)
{

    if (pSection->Characteristics & IMAGE_SCN_MEM_EXECUTE)
    {
        CRC_HASHI ctx;

        //这里计算CRC值
        dwStartAdr = pSection->VirtualAddress + (ULONG_PTR)ImageBase;
        dwSize = pSection->Misc.VirtualSize;
        ctx.m_pAddr = (PVOID)dwStartAdr;
        ctx.m_dwSize = dwSize;

        ctx.m_dwHashVal = crc32((const void*)ctx.m_pAddr, dwSize);


        g_crc_vtr.push_back(ctx);




    }

    pSection++;

}
```

循环校验CRC

```
PVOID pDbg =    GetProcAddress(GetModuleHandle(L"ntdll.dll"), "DbgBreakPoint");
byte bRet = 0xC3;

WriteProcessMemory(NtCurrentProcess, pDbg, &bRet, 1, 0);
```

反附加:

DbgBreakPoint

因为调试器在附加的时候会走DbgBreakPoint函数、所以HOOK这个函数就可以改变调试器运转流程、从而达到反附加!

```
PVOID pDbg =    GetProcAddress(GetModuleHandle(L"ntdll.dll"), "DbgBreakPoint");
byte bRet = 0xC3;

WriteProcessMemory(NtCurrentProcess, pDbg, &bRet, 1, 0);
```

对抗反调试方法：几乎是HOOK

好了本期的浅谈反调试技术已经到这里了、我是By小曾(Xzeng)、下期见!

62

5

¥

最新回复 (6)

niuzuoquan 5 2020-9-21 16:16 2 楼 0

感谢分享

极客

Caim Astraea 5 2020-9-21 22:15 3 楼 0

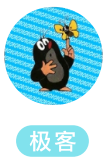
感谢分享

极客

橄榄绿o 0 2020-9-22 07:31 4 楼 0

感谢分享

临时



HlccFu 18 2020-9-22 09:01 5 楼 0

非常感谢楼主分享

极客

语过添情j 0 2020-9-29 09:41 6 楼 0

楼主能发个完整代码吗，感谢楼主

临时

黄小付 3 2020-9-29 16:11 7 楼 0

我想问一下，就是附加下断，断下之后程序就崩溃，是什么检测和原理，c3和硬件断点都是一样直接内存异常。

极客

tutuj

内容

回帖 表情 高级回复

返回