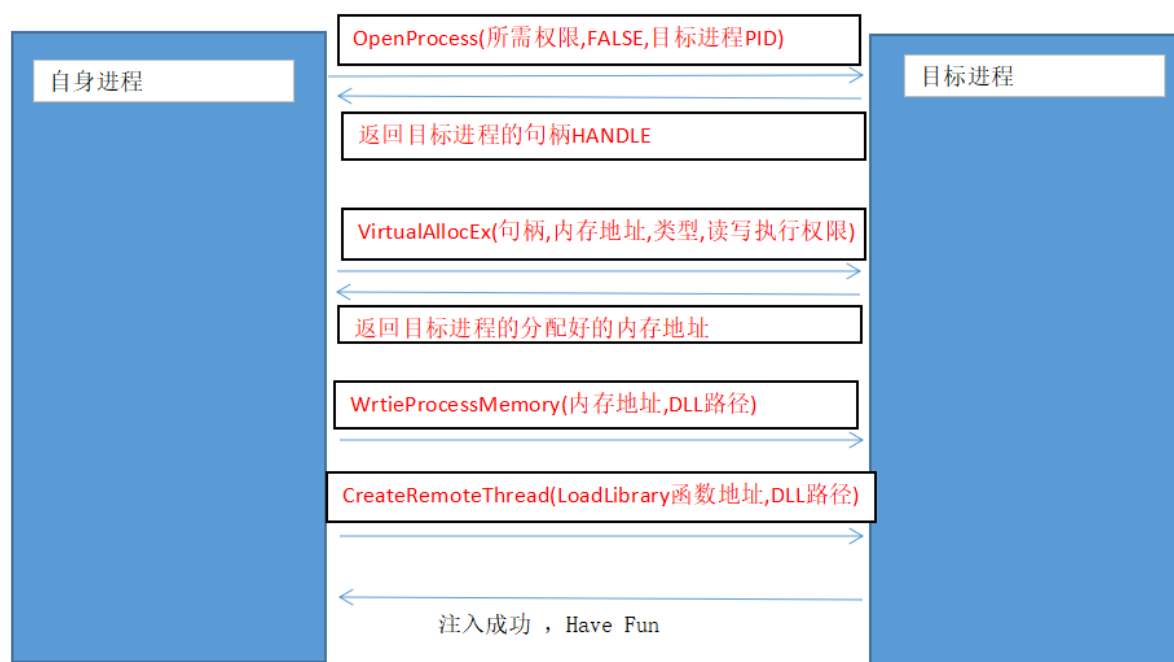


win10 1909 R3常规底层注入原理与实现（实战某FPS网游）

1.R3经典注入原理

原理：利用微软提供的公开API--CreateRemoteThread(), 在目标进程中创建一个线程，线程执行LoadLibrary函数，

目标进程则加载指定的DLL，并调用Dllmain。



注意：因为windows每个进程创建的时候都会映射kernel32.dll，ntdll.dll等系统dll，所以每个进程中这些DLL提供的导出函数的地址

都是相同的。

代码实现：

```
char a[] = "C:\\Users\\太上\\Desktop\\DLL注入\\信息框DLL.dll";
HANDLE Proc = OpenProcess(PROCESS_ALL_ACCESS, FALSE, 14752);
char* ProcMem = (char*)VirtualAllocEx(Proc, 0, sizeof(a), MEM_COMMIT, PAGE_READWRITE);
WriteProcessMemory(Proc, ProcMem, a, sizeof(a), 0);
HMODULE Kernel = GetModuleHandle(L"Kernel32.dll");
char* ModuleMem = (char*)GetProcAddress(Kernel, "LoadLibraryA");
CreateRemoteThread(Proc,
    0,
    0,
    (LPTHREAD_START_ROUTINE)ModuleMem,
    ProcMem,
    0,
    0);
```

2.对经典注入API进行逆向

1.对OpenProcess进行逆向。

在微软官网找函数原型：<https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-openprocess>

Filter by title

API structure

OpenProcess function

OpenProcessToken function

OpenThread function

OpenThreadToken function

PROCESS_INFORMATION structure

PROCESS_INFORMATION_CLASS enumeration

PROCESS_LEAP_SECOND_INFORMATION structure

PROCESS_MEMORY_EXHAUSTION_INFO structure

PROCESS_MEMORY_EXHAUSTION_TYPE enumeration

Download PDF

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	processthreadsapi.h (include Windows Server 2003, Windows Vista, Windows 7, Windows Server 2008 Windows Server 2008 R2, Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

在这个页面，往下拉，可以看到在 "Requirements" 里可以获得我们需要的信息。

在这里我们需要这样的几个信息：

- 1.第一个信息是 "Minimum supported client"，这个信息代表了OpenProcess这个API所支持的最低windows版本。
- 2.第二个信息是 "Header"，这个信息代表了 如果我们要在VS里调用这个函数，应该包含哪个头文件。注意，通常使用的windows API 都包含在 "windows.h" 这个头文件里。
- 3.第三个信息是最重要的，这里 "DLL" 代表OpenProcess这个函数是被哪个DLL动态链接库导出来的，或者说是这个函数的具体实现是在哪个DLL中。

综上所述，我们要研究OpenProcess的底层实现，必须对 Kernel32.dll 进行逆向分析。

既然要对DLL进行逆向分析，那么就有两种可以选择，用IDA静态分析或者用OD下断点调试。

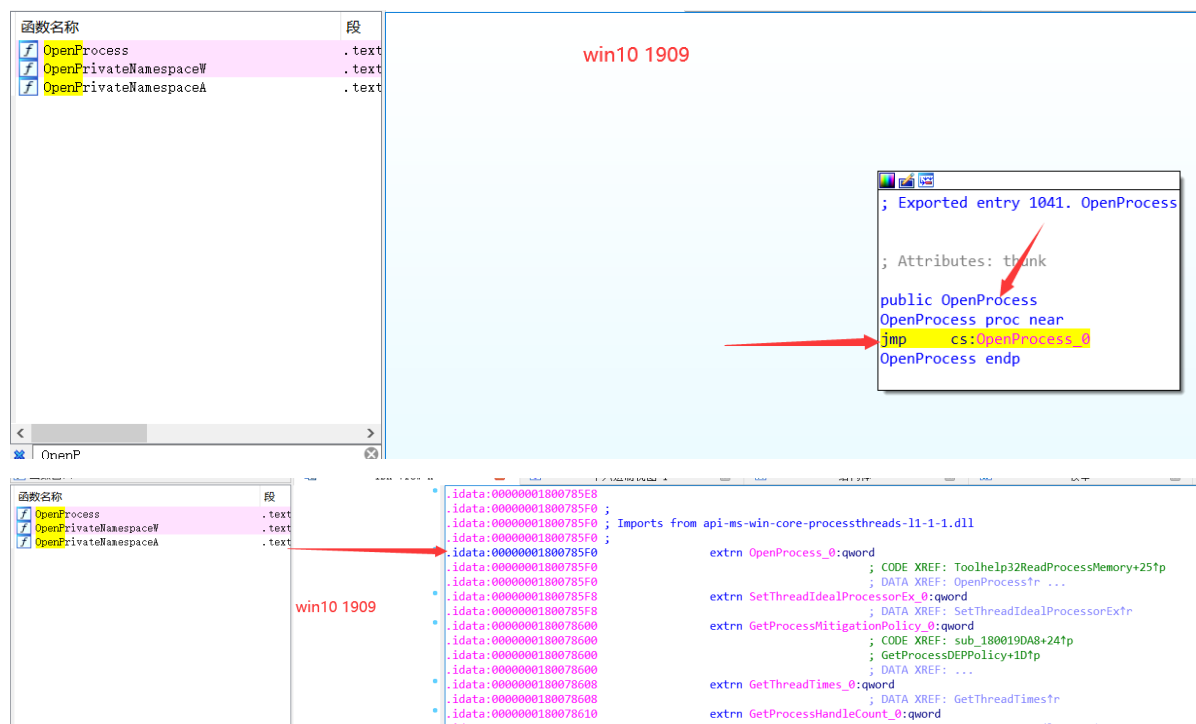
选择调试方法很简单，先用IDA反编译DLL，发现函数结构简单就不动调，函数结构复杂就动调。

第一步，找到 "kernel32.dll" 这个DLL文件，然后用 IDA 打开

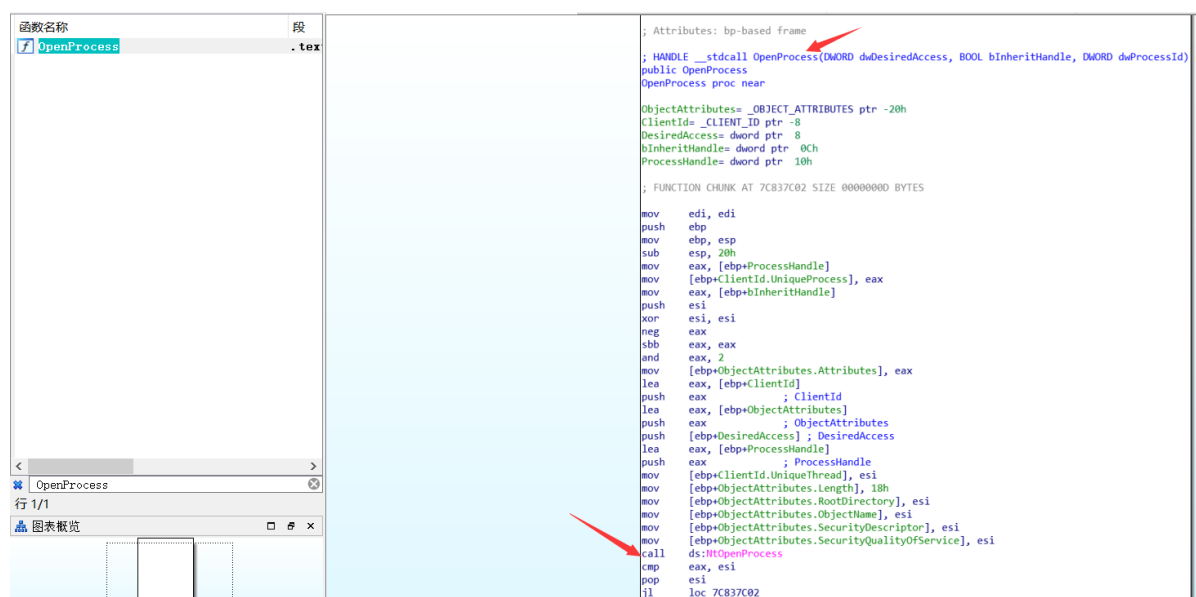
在 C:\Windows\System32目录下 找到这个DLL，复制一份到桌面，然后用IDA打开。但是，每个Windows系统都有不同版本的DLL文件。如何选择不同的DLL进行分析，这会降低我们的逆向难度。因为Windows的API，例如OpenProcess，它支持Windows XP 到Windows 10系列，这就意味着它的传参方式不会有任何变化，选择低版本windows的DLL，因为IDA符号的支持，会大大降低逆向的难度。

举例：

这是WIN10 1909的 Kernel32.dll 的 IDA 反编译图。



这是WIN7（或XP）的 Kernel32.dll 的 IDA 反编译图。



很明显，下面的IDA反编译容易理解，所以在文章后续，默认使用Win7(或XP)系统的DLL来分析。

在对OpenProcess函数分析中，可以发现：OpenProcess函数只是对传进来的参数进行简单的分析，然后把原来的参数整合到个新的结构中，这些新的结构又作为参数传到了"NtOpenProcess"函数中。这种操作意味着如果我们掌握了"NtOpenProcess"的参数构造，我们可以绕过"OpenProcess"，直接用我们构造的参数来调用"NtOpenProcess"。

提示："Nt"和"Zw"开头的函数都在"ntdll.dll"中，这个"ntdll"一般在 C:\Windows\System32 目录下。

接着我们在"CSDN"中看看能不能找到函数的定义，找到了就直接用，找不到再想别的办法。

<https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/ntddk/nf-ntddk-ntopenprocess>
发现在"CSDN"中找到了所需要的函数。

Filter by title

ry function

NtOpenProcess function

PCREATE_PROCESS_NOTIF

Y_ROUTINE callback

function

PCREATE_PROCESS_NOTIF

Y_ROUTINE_EX callback

function

PCREATE_THREAD_NOTIFY

_ROUTINE callback

function

PGET_LOCATION_STRING

Download PDF

NtOpenProcess function (ntddk.h)

04/30/2018 • 2 minutes to read

The ZwOpenProcess routine opens a handle to a process object and sets the access rights to this object.

Syntax

C++

Copy

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtOpenProcess(  
1 PHANDLE ProcessHandle,  
2 ACCESS_MASK DesiredAccess,  
3 POBJECT_ATTRIBUTES ObjectAttributes,  
4 PCLIENT_ID ClientId  
);
```

这个NtOpenProcess在CSDN上面显示有 4 个参数。和我们在IDA里看到的一致。

第 1 个参数是 "PHANDLE", "P"代表我们需要传进来一个地址, "HANDLE" 表示数据的类型, 在这里相当于传进来 &(HANDLE 变量名)。

第 2 个参数是 "ACCESS_MASK", "ACCESS_MASK"只是起的一个别名, 它的真正类型是 ULONG ,通过阅读,我们选择 "PROCESS_ALL_ACCESS", 意思是获得全部访问权限。

指定访问权限

10/17/2018 • 2分钟阅读 •

类中指定一组访问权限的位掩码类型。存取掩码一种访问控制入口。

句法

复制

```
typedef ULONG ACCESS_MASK;
```

以下标准特定的访问权限适用于所有类型的执行对象。

旗子	描述
删除	调用方可以删除对象。
访问控制	调用方可以读取文件的访问控制列表

第 3 个参数是 "POBJECT_ATTRIBUTES", 同第 1 个参数类似, 需要穿入&(OBJECT_ATTRIBUTES 变量名)

```
typedef struct _OBJECT_ATTRIBUTES {
    ULONG          Length;
    HANDLE         RootDirectory;
    PUNICODE_STRING ObjectName;
    ULONG          Attributes;
    PVOID          SecurityDescriptor;
    PVOID          SecurityQualityOfService;
} OBJECT_ATTRIBUTES;
```

注意

注意：当我们定义OBJECT_ATTRIBUTES 变量名时候，需要令 OBJECT_ATTRIBUTES 变量名={0}，并且使 变量名.Length=Sizeof(OBJECT_ATTRIBUTES)。

第 4 个参数是 "PCLIENT_ID"，同第 1 个参数类似，需要穿入&(CLIENT_ID 变量名)，但是在"MSDN"的描述里，我们并没有找到结构的定义。 这就需要想点别的办法来找到这个结构的定义。

ClientId

[in, optional] A pointer to a client ID that identifies the thread whose process is to be opened. In Windows Vista and later versions of Windows, this parameter must be a non-NULL pointer to a valid client ID. In Windows Server 2003, Windows XP, and Windows 2000, this parameter is optional and can be set to NULL if the **OBJECT_ATTRIBUTES** structure that *ObjectAttributes* points to specifies an object name. For more information, see the following Remarks section.

注意：往下拉网页，或者网上拉网页，我们可以看到"NtOpenProcess"函数是位于"ntddk.h"中，既然位于"ntddk.h"中，就在"ntddk.h"中找

CLIENT 结构的定义。（在这里需要安装 WDK 驱动开发环境）

```
5957 __kernel_entry NTSYSCALLAPI
5958 NTSTATUS
5959 NTAPI
5960 NtOpenProcess (
5961     _Out_ PHANDLE ProcessHandle,
5962     _In_  ACCESS_MASK DesiredAccess,
5963     _In_  POBJECT_ATTRIBUTES ObjectAttributes,
5964     _In_opt_ PCLIENT_ID ClientId
5965 );
```

```

7773
7774     typedef struct _CLIENT_ID {
7775         HANDLE UniqueProcess;
7776         HANDLE UniqueThread;
7777     } CLIENT_ID;
7778     typedef CLIENT_ID *PCLIENT_ID;

```

"HANDLE UniqueProcess" 是进程的PID,"HANDLE UniqueThread" 是线程的PID。

到此为止，已经分析完了整个NtOpenProcess函数，接下来就是调用它。

```

#include<windows.h>
#include <TlHelp32.h>
#include <stdio.h>
typedef struct _CLIENT_ID {
    HANDLE UniqueProcess;
    HANDLE UniqueThread;
} CLIENT_ID,*PCLIENT_ID;
typedef struct _OBJECT_ATTRIBUTES {
    ULONG Length;
    HANDLE RootDirectory;
    PUNICODE_STRING ObjectName;
    ULONG Attributes;
    PVOID SecurityDescriptor; // Points to type SECURITY_DESCRIPTOR
    PVOID SecurityQualityOfService; // Points to type
    SECURITY_QUALITY_OF_SERVICE
} OBJECT_ATTRIBUTES,* POBJECT_ATTRIBUTES;
typedef DWORD (WINAPI* PNTOpenProcess)(
    PHANDLE ProcessHandle,
    ACCESS_MASK DesiredAccess,
    POBJECT_ATTRIBUTES ObjectAttributes,
    PCLIENT_ID ClientId
);
HANDLE PID = 0;
HMODULE hm = GetModuleHandle("ntdll.dll");
int main()
{
    GetPIDByName("xxxx.exe"); //顾名思义，在压缩包里有函数具体实现。
    //获取目标进程句柄
    PNTOpenProcess NtOpenProcess = 0;
    NtOpenProcess = (PNTOpenProcess)GetProcAddress(hm, "NtOpenProcess");
    HANDLE Hprocess = 0;
    CLIENT_ID ClientId;
    ClientId.UniqueProcess = PID;
    ClientId.UniqueThread = 0;
    OBJECT_ATTRIBUTES OBJECTATTRIBUTES = {0};
    OBJECTATTRIBUTES.Length = sizeof(OBJECTATTRIBUTES);
    NtOpenProcess(&Hprocess,PROCESS_ALL_ACCESS,&OBJECTATTRIBUTES,&ClientId);

```

注意:

```
5957 __kernel_entry NTSYSCALLAPI
5958 NTSTATUS
5959 NTAPI
5960 NtOpenProcess (
5961     _Out_ PHANDLE ProcessHandle,
5962     _In_ ACCESS_MASK DesiredAccess,
5963     _In_ POBJECT_ATTRIBUTES ObjectAttributes,
5964     _In_opt_ PCLIENT_ID ClientId
5965 );
```

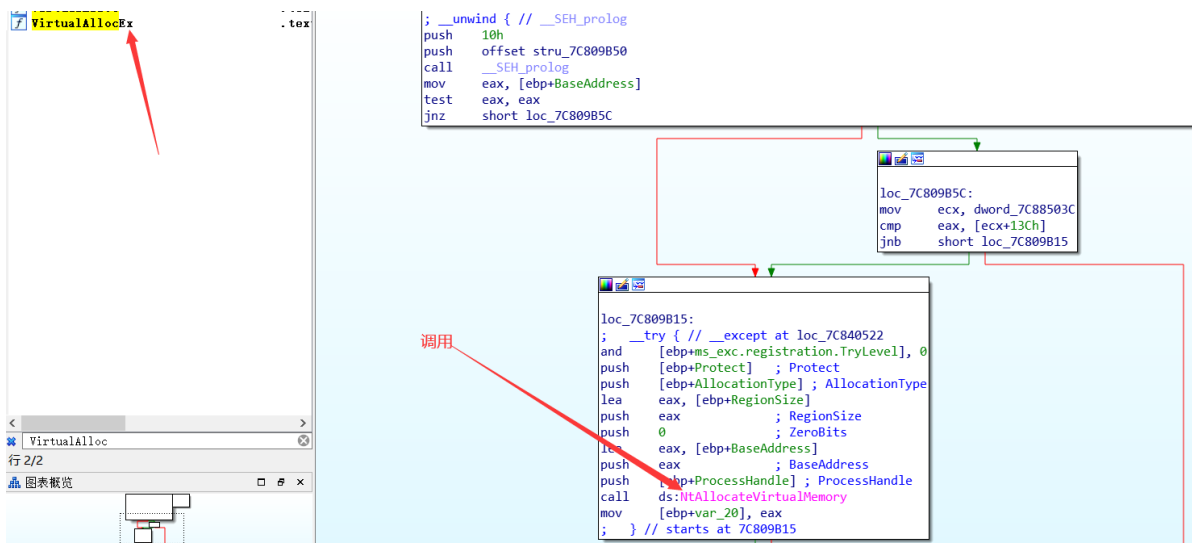
用DWORD替换

用WINAPI替换

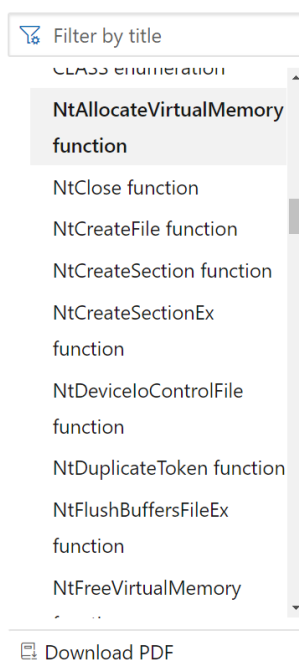
删掉

2.对VirtualAllocEx进行逆向。

同上，在IDA里反编译查看。



在CSDN里查找函数“NtAllocateVirtualMemory”。



NtAllocateVirtualMemory function (ntifs.h)

05/20/2020 • 6 minutes to read

The NtAllocateVirtualMemory routine reserves, commits, or both, a region of pages within the user-mode virtual address space of a specified process.

Syntax

```
C++
__kernel_entry NTSYSCALLAPI NTSTATUS NtAllocateVirtualMemory(
1 HANDLE ProcessHandle,
2 PVOID *BaseAddress,
3 ULONG_PTR ZeroBits,
4 PSIZE_T RegionSize,
5 ULONG AllocationType,
6 ULONG Protect
);
```

第 1 个参数是 "HANDLE"，在这里是NtOpenProcess函数的第 1 个参数。

第 2 个参数是 "PVOID"，这是一个空指针，NtAllocateVirtualMemory函数执行成功会返回一个内存地址。

第 3 个参数是 "ULONG_PTR"，填 0 (或者NULL)就可以。

第 4 个参数是 "PSIZE_T"，在这里是&(SIZE_T 变量名)，按0x1000对齐，不足0x1000补足0x1000。

第 5 个参数是 "ULONG AllocationType"，填"MEM_COMMIT| MEM_RESERVE"，在这里是给线性地址挂上物理页并且保留线性地址。一般都是填这个"MEM_COMMIT| MEM_RESERVE"。

第 6 个参数是 "ULONG Protect"，填"PAGE_EXECUTE_READWRITE"，这里是设置分配好的内存地址区域的属性，一般设置为可读可写可执行。

```
typedef DWORD (WINAPI* PNTAllocateVirtualMemory)(
    HANDLE ProcessHandle,
    PVOID* BaseAddress,
    ULONG_PTR ZeroBits,
    PSIZE_T RegionSize,
    ULONG AllocationType,
    ULONG Protect
);

//申请内存
PVOID a = 0;
SIZE_T RegionSize = 0x1000;
PNTAllocateVirtualMemory NtAllocateVirtualMemory=
(PNTAllocateVirtualMemory)GetProcAddress(hm, "NtAllocateVirtualMemory");
NtAllocateVirtualMemory(Hprocess,&a,0,&RegionSize,MEM_COMMIT|
MEM_RESERVE,PAGE_EXECUTE_READWRITE);
```

3.对WriteProcessMemory进行逆向。

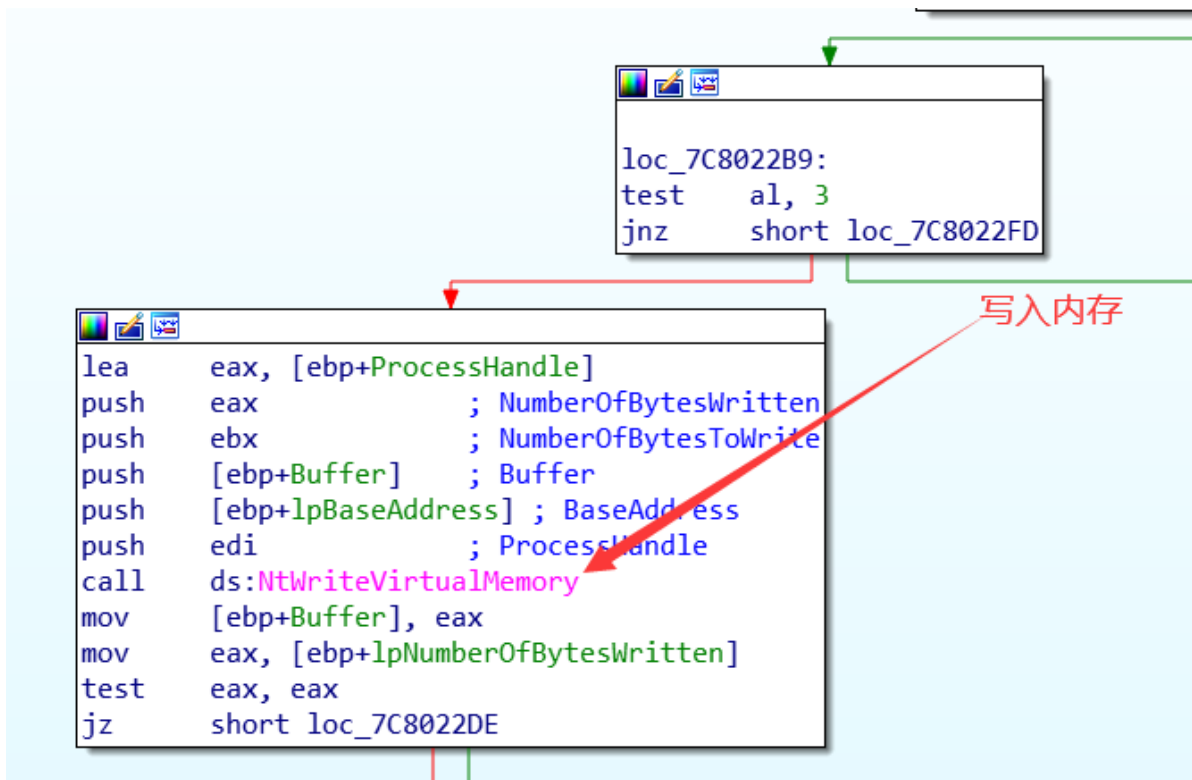
同上，在IDA里反编译查看。

```
; BOOL __stdcall WriteProcessMemory(HANDLE hProcess, LPVOID lpBaseAddress, LPVOID lpBuffer, DWORD nSize, LPDWORD lpNumberOfBytesWritten)
public WriteProcessMemory
WriteProcessMemory proc near
BaseAddress= dword ptr -8
NumberOfBytesToProtect= dword ptr -4
ProcessHandle= dword ptr 8
lpBaseAddress= dword ptr 0Ch
Buffer= dword ptr 10h
OldAccessProtection= dword ptr 14h
lpNumberOfBytesWritten= dword ptr 18h

mov     edi, edi
push    ebp
mov     ebp, esp
push    ecx
push    ecx
mov     eax, [ebp+lpBaseAddress]
push    ebx
mov     ebx, [ebp+OldAccessProtection]
push    esi
mov     esi, ds:NtProtectVirtualMemory
push    edi
mov     edi, [ebp+ProcessHandle]
mov     [ebp+BaseAddress], eax
lea     eax, [ebp+OldAccessProtection]
push    eax
push    ; OldAccessProtection
push    40h ; NewAccessProtection
lea     eax, [ebp+NumberOfBytesToProtect]
push    eax
push    ; NumberOfBytesToProtect
lea     eax, [ebp+BaseAddress]
push    eax
push    ; BaseAddress
push    edi ; ProcessHandle
mov     [ebp+NumberOfBytesToProtect], ebx
call    esi ; NtProtectVirtualMemory
cmp     eax, 0C000004Eh
```

有两个主要函数

先判断内存地址是否可写,判断成功可写就直接写入内存,不可写则就用这个函数更改内存属性为可写,然后再写入



我们发现 "WriteProcessMemory" 调用了两个主要的函数，一个是 "NtProtectVirtualMemory"，另一个是 "NtWriteVirtualMemory"。

注意：因为我们用 "NtAllocateVirtualMemory" 申请内存的时候，设置内存的属性为可读可写可执行。所以我们不需要调用 "NtProtectVirtualMemory"，直接调用 "NtWriteVirtualMemory" 即可。

很遗憾，在CSDN上，并没有找到"NtWriteVirtualMemory"的函数原型。

Microsoft | Docs 文档 Learn Q&A 代码示例

[登录](#)

筛选器

内容区域

- ☒ 全部 1
- ☐ Documentation 1
- ☐ Learn 0
- ☐ 参考 0

1 result for "NtWriteVirtualMemory"

We couldn't find any results matching "NtWriteVirtualMemory" in Desktop

Exploit Protection 参考 - Windows security

[/windows/security/threat-protection/microsoft-defender-atp/exploit-protection-reference](#)

[IProtectVirtualProtectEx](#) [NtProtectVirtualMemory](#) [HeapCreate](#) [RtlCreateHeap](#) [CreateProcessA](#) [CreateProcessW](#) [CreateProcessInternalA](#) [CreateProcessInternalW](#) [NtCreateUserProcess](#) [NtCreateProcess](#) [NtCreateProcessEx](#) [CreateRemoteThread](#) [CreateRemoteThreadEx](#) [NtCreateThreadEx](#) [WriteProcessMemory](#) [NtWriteVirtualMemory](#)

1

所以只能看"ntddk.h"中关于"NtWriteVirtualMemory"的定义。很遗憾，无论是"NtWriteVirtualMemory"或者"ZwWriteVirtualMemory"在ntddk.h中也没有定义。

```
NTSTATUS DriverEntry(PDRIVER_OBJECT pdriver, PUNICODE_STRING preg)
{
    NtWriteVirtualMemory();
    int NtWriteVirtualMemory()
```

这时候只能百度参考下别人的结构，和IDA对照着看。很明显，这种方法并不是那么稳妥，如果这种方法解决不了问题，就只能用OD动调，利用GetProcAddress函数找到NtWriteVirtualMemory函数的地址下断点，看传参了。

```
typedef DWORD(WINAPI* PNTWriteVirtualMemory)( //这是在网上找的NtWriteVirtualMemory  
函数改的函数指针，自测可用  
    IN HANDLE hProcess,  
    IN PVOID BaseAddress,  
    IN PVOID Buffer, //要写入数据的缓冲区的地址。  
    IN ULONG BytesToWrite, //要写入多少个字节  
    OUT PULONG BytesWritten //实际写入了多少个字节  
);
```

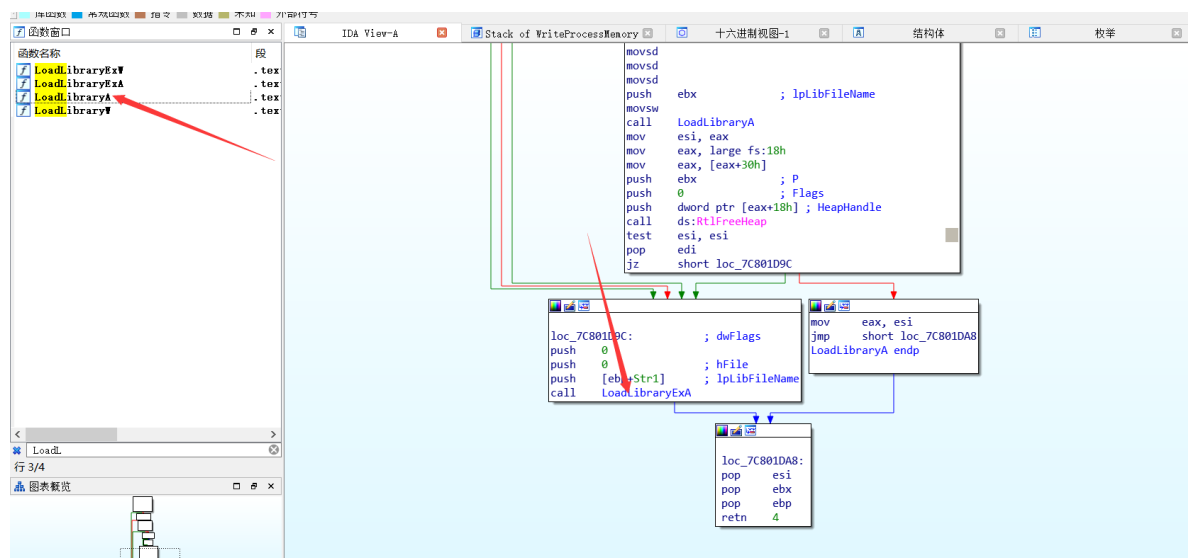
第 1 个参数填入 NtOpenProcess 的第一个参数。

第 2 个参数填入 NtAllocateVirtualMemory 的第二个参数。

4.对LdrLoadDll进行逆向分析(重点)

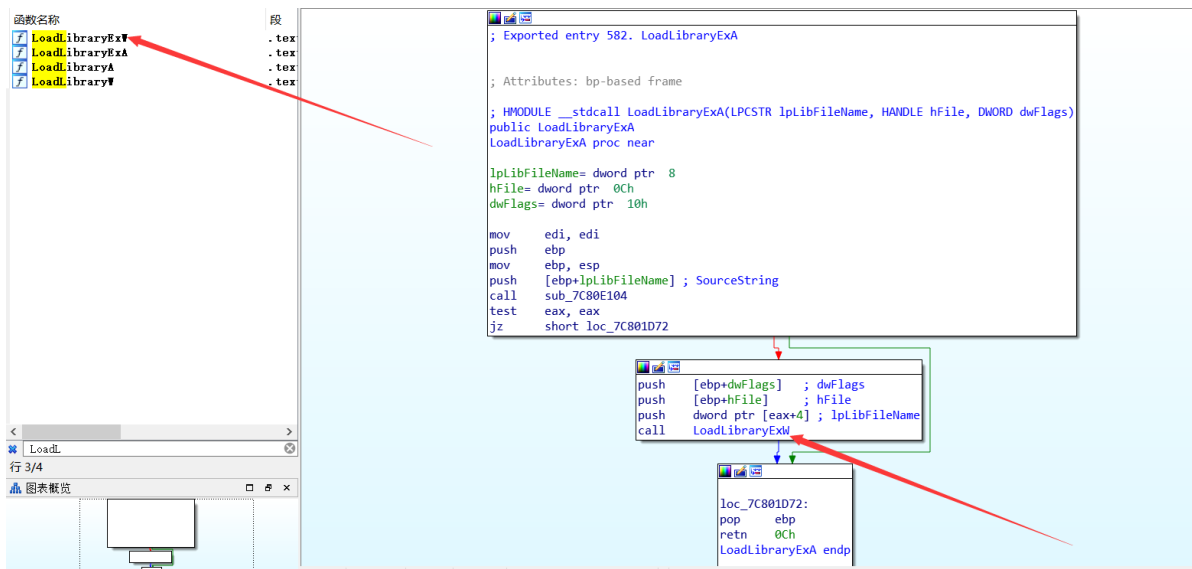
用IDA反编译后查看"LoadLibrary".

注意：查看LoadLibrary A的函数结构，便于分析。



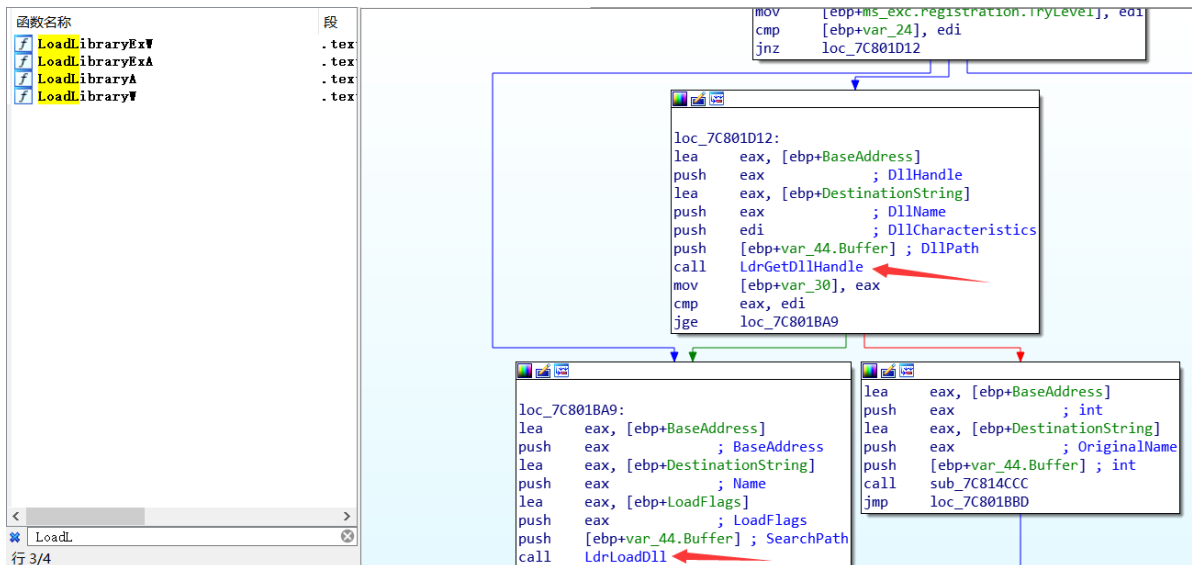
发现 LoadLibraryA 函数调用了 LoadLibraryExA。但是"LoadLibraryExA"并不是 ntdll里的函数，说明这个函数还不够底层。其次在 上面的截图里 我们可以发现 "LoadLibraryExA" 也是在 kernel32.dll 里。

进入"LoadLibraryExA"函数。



发现"LoadLibraryExA"调用了"LoadLibraryExW"函数，但是"LoadLibraryExW"函数同样不是底层函数。所以还要跟进"LoadLibraryExA"。

进入"LoadLibraryExW"函数。

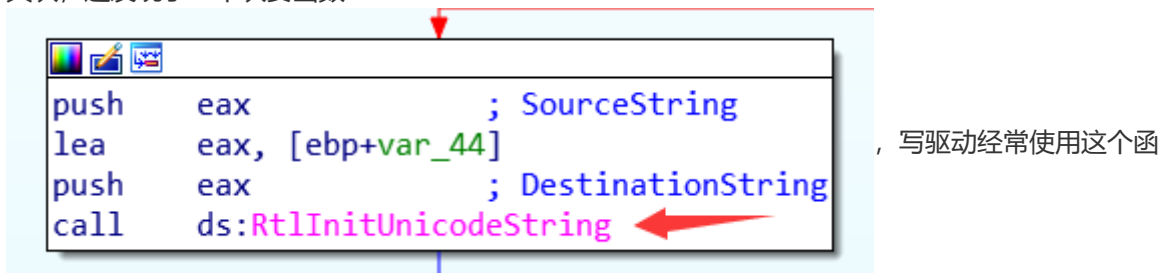


发现有两个主要的函数，一个是"LdrGetDllHandle"，另一个是"LdrLoadDLL"。顾名思义，我们可以从名字判断DLL函数的功能。

"LdrGetDllHandle" 应该是 GetModuleHandle函数的底层实现，"LdrLoadDLL" 应该是 LoadLibrary 的底层实现。

从IDA上看两个函数的参数，并没有实际的联系，经过测试，LdrLoadDLL 可以作为单独函数使用。

其次，还发现了一个次要函数



数，目的是初始化一个UNICODE格式的字符串。

LdrLoadDLL 在CSDN 和 ntddk.h 中 同样没有任何定义，下面的定义 我是从“某度”上搜索的。

```
typedef DWORD (WINAPI*PLdrLoadDll)(
    IN PWCHAR          PathToFile OPTIONAL,
    IN ULONG           Flags OPTIONAL,  //别人给的定义
    IN PUNICODE_STRING ModuleFileName,
    OUT PHANDLE        ModuleHandle);
```

经过测试，每次执行这个函数都会造成进程崩溃，猜测是函数传参的问题。

因为IDA不能看到具体传参情况，我们选择用OD动态调试。

要进行动态调试，首先要自己写一个程序来附加调试。

```
1  #include<stdio.h>
2  #include<windows.h>
3  int main()
4  {
5      const char* a;
6      a = "abc";
7      HMODULE b = 0;
8      system("pause");
9      b=LoadLibraryW(L"D:\\SYC_DLL.dll");
10
11
12      return 0;
13  }
```

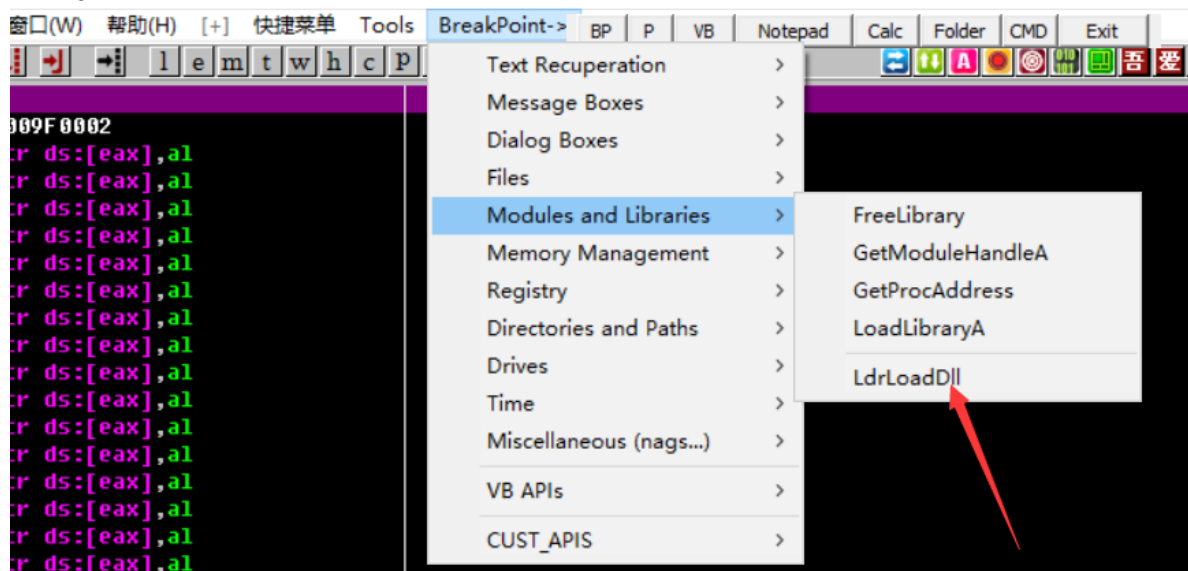
关键1 指向 `system("pause");`

关键2 指向 `b=LoadLibraryW(L"D:\\SYC_DLL.dll");`

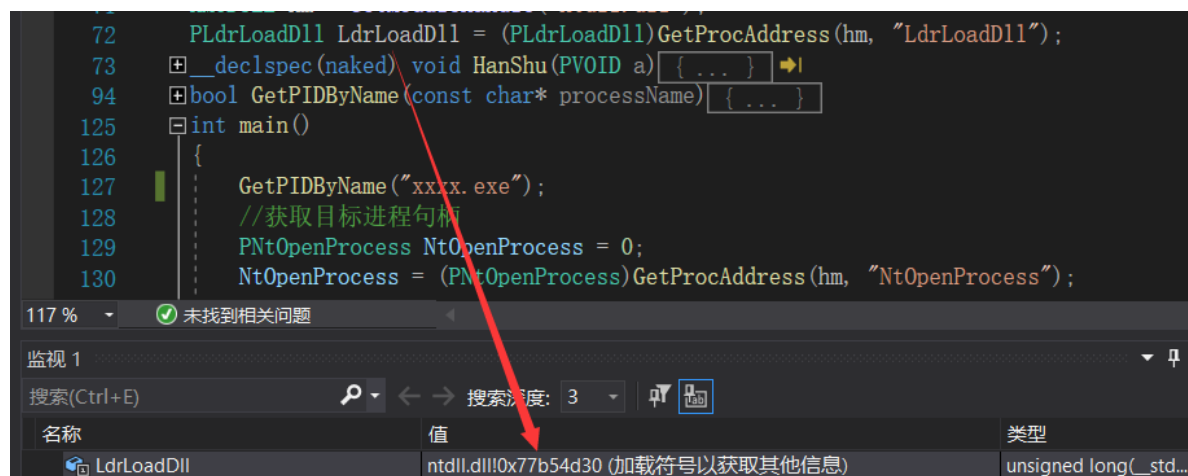
这个程序设计的巧妙在于：因为进程启动的时候，会映射系统DLL，这时候断下来的LdrLoadDll函数不是我们想要的，这时候我们先让程序跑完这段过程，然后用System("pause")暂停一下，再用OD附加断点LdrLoadDLL函数，这时候的参数就是我们想要的了。

提示：要断点LdrLoadDLL函数，可以用“吾爱破解OD”的自带插件。

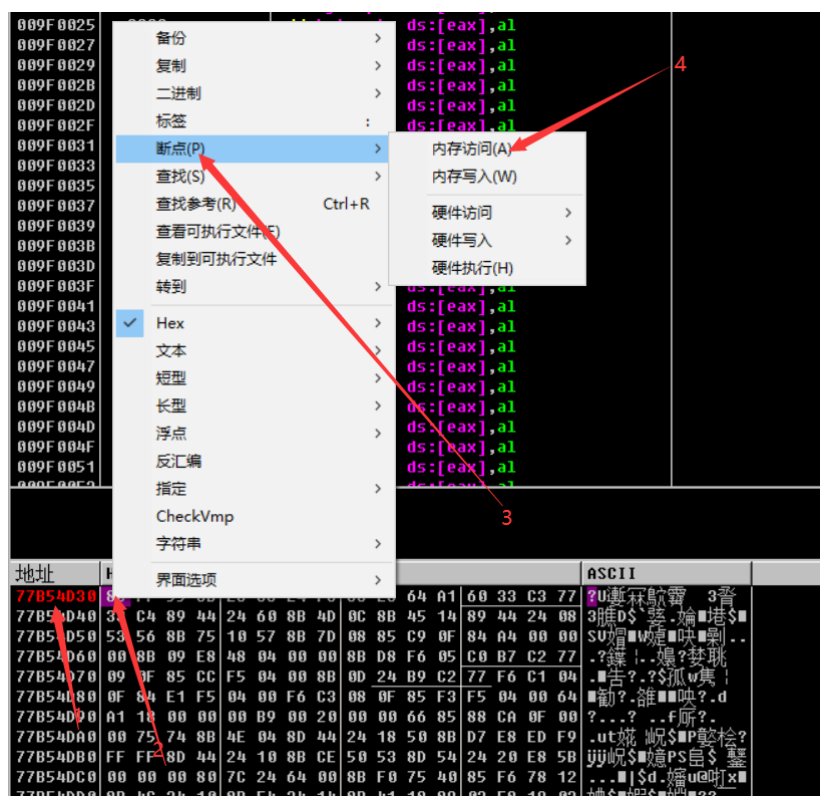
005118]



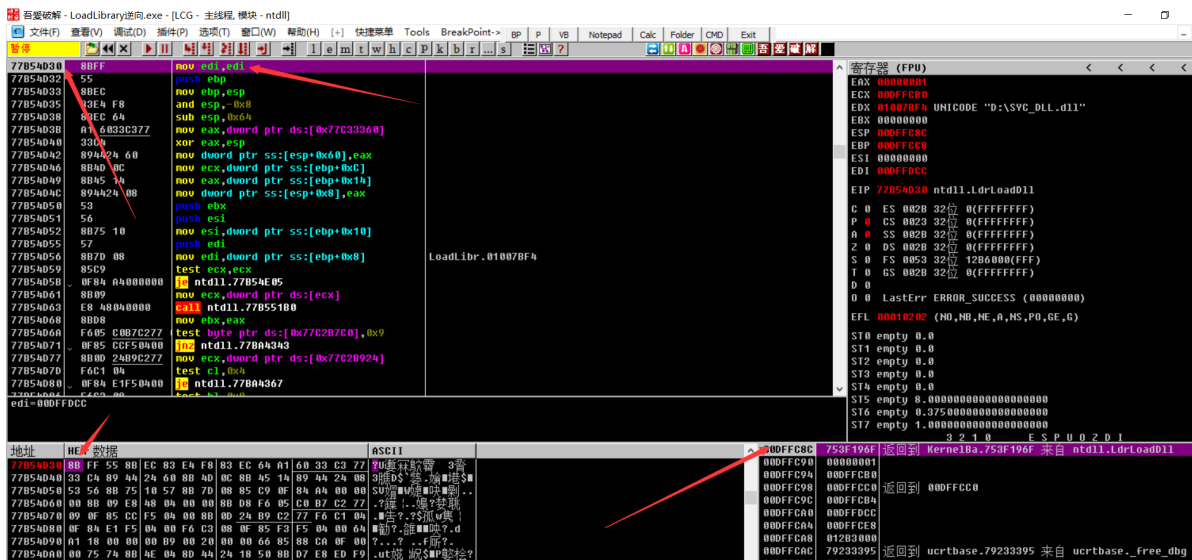
如果没有这个插件，就要用 `GetProcAddress(LoadLibrary("ntdll.dll"), "LdrLoadDll")` 来获取函数地址，在这里 `LoadLibrary` 不是必需的，可以用 `GetModuleHandle` 来代替。



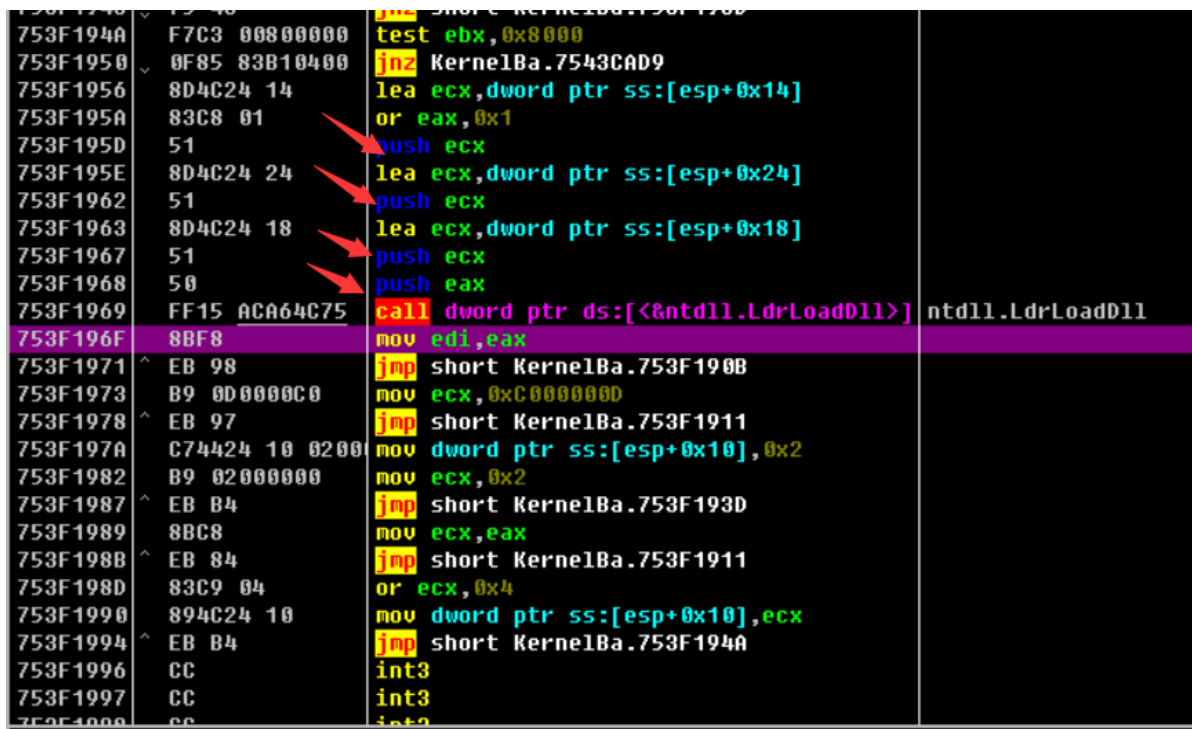
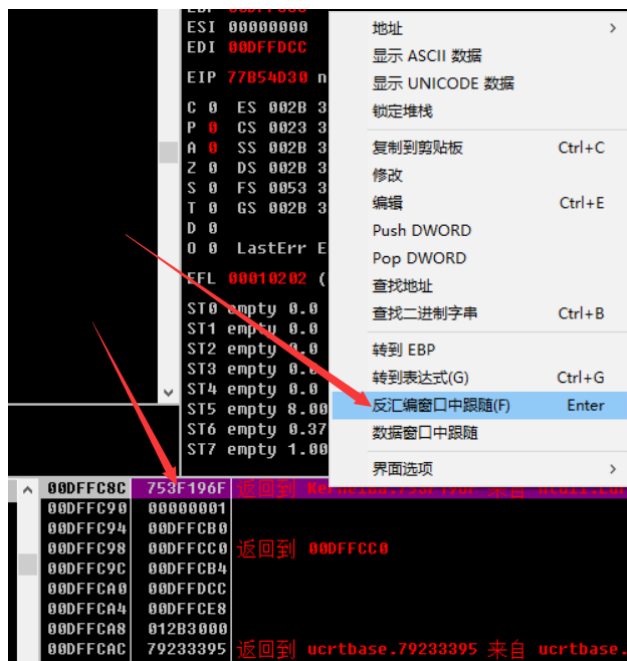
在我的系统上，`LdrLoadDLL`的函数地址是：0x77b54d30，因为系统启动后，每个进程的 `ntdll` 的导出函数的地址都一样。我们可以直接下内存访问断点



用这两种方法其中的一种下好断点后，然后在OD里把程序跑起来，返回到我们的程序窗口，按一下回车让程序也跑起来，这时候再会到OD界面，发现已经断下在关键位置。



我们断在了函数的内部，可以先回到 call 函数的位置，看看函数的传参。




```

753F1958 8308 01      or ecx,0x1
753F1959 51          push ecx
753F195E 804C24 24     lea ecx,duword ptr ss:[esp+0x24]
753F1962 51          push ecx
753F1963 804C24 18     lea ecx,duword ptr ss:[esp+0x18]
753F1967 51          push ecx
753F1968 50          push eax
753F1969 FF15 AC6AC75 call duword ptr ds:[<ntdll.LdrLoadDll>]
753F196F 8BF8        mov edi,eax
753F1971 EB 98        jmp short KernelBa.753F190B
753F1972 09 000000C0  mov ecx,ecx+000000C0
753F1978 EB 97        jmp short KernelBa.753F1911
753F197A C74424 10 0200 mov duword ptr ss:[esp+0x10],0x2
753F1982 B9 02000000  mov ecx,0x2
753F1987 EB 84        jmp short KernelBa.753F193D
753F1989 80E3        mov ecx,eax
753F198B EB 84        jmp short KernelBa.753F1911
753F198D 83C9 04      or ecx,0x4
753F199B 894C24 10     mov duword ptr ss:[esp+0x10],ecx
753F199E EB 84        jmp short KernelBa.753F194A
753F199F CC          int3
753F199F CC          int3

```

```

LDR 00000000 UNKCODE 0(<SYSCALL.dll>)
EBX 00000000
ESP 000FFC8C
EDP 000FFC8C
ESI 00000000
EDI 000FFC8C
EIP 77F54030 ntdll.LdrLoadDll

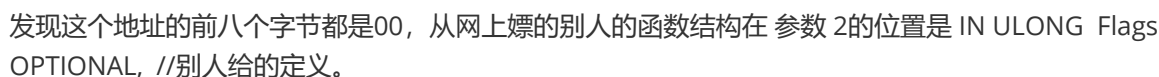
C 0 ES 0020 32 0 (FFFFFFFF)
P 0 GS 0023 32 0 (FFFFFFFF)
A 0 SS 0020 32 0 (FFFFFFFF)
Z 0 DS 0028 32 0 (FFFFFFFF)
S 0 FS 0052 32 0 (00000000)
I 0 GS 0028 32 0 (FFFFFFFF)
D 0
0 0
0 0 LastErr ERROR_SUCCESS (00000000)
EFL 00010202 (ND, NB, NE, A, NS, PD, GE, G)

ST0 empty 0.0
ST1 empty 0.0
ST2 empty 0.0
ST3 empty 0.0
ST4 empty 0.0
ST5 empty 0.000000000000000000000000
ST6 empty 0.375000000000000000000000
ST7 empty 1.000000000000000000000000
2.2.1.0 F S P U 0 2 0 1

```

地址	HEX 数据	ASCII
77B54030	80 FF 55 80 EC 83 EA F8 83 EC 64 A1 60 33 C3 27	00000000
77B54034	33 CA 89 44 24 60 88 04 0C 88 45 14 89 44 24 08	00000001
77B54038	53 56 80 75 15 57 80 70 08 95 C9 08 84 0A 00	00000000
77B5403C	00 8B 09 E8 04 00 00 00 08 08 F6 05 00 07 C2 77	00000000
77B54040	00 00 00 00 00 00 00 2A 09 02 77 F6 C1 04	00000000
77B54044	00 84 E1 F5 04 06 F6 C3 00 0C 85 F3 F5 04 04 00	00000000
77B54048	A1 18 00 00 00 00 00 00 00 00 00 66 85 88 CA 0F	00000000

参数1 不用管, 先看参数2:



发现一个错误，ULONG 应该是直接 push 值 才对，但是这里 push 0x00DFFCB0，是一个地址。我们对ULONG进行更正，应该是PULONG 或者 ULONG *

[illegible]

地址	HEX 数据	ASCII
01007BF4	44 00 3A 00 5C 00 53 00 59 00 43 00 5F 00 44 00	D.:. \.S.Y.C._.D.
01007C04	4C 00 4C 00 2E 00 64 00 6C 00 6C 00 00 00 00 00	L.L...d.l.l.....
01007C14	00 00 00 00 54 68 65 20 76 61 6C 75 65 20 6F 66The value of
01007C24	20 45 53 50 20 77 61 73 20 6E 6F 74 20 70 72 6F	ESP was not pro
01007C34	70 65 72 6C 79 20 73 61 76 65 64 20 61 63 72 6F	perly saved acro
01007C44	72 73 20 61 20 66 75 6F 62 74 60 6F 65 20 62 61	ss a function ca

再看参数4 (OUT PHANDLE ModuleHandle) :

3.实战某CXGO FPS网游。

首先是对上述的逆向分析的函数，进行开发。

```
#include<windows.h>
#include <TlHelp32.h>
#include <stdio.h>
typedef struct _UNICODE_STRING {
    USHORT Length;
    USHORT MaximumLength;
#ifdef MIDL_PASS
    [size_is(MaximumLength / 2), length_is((Length) / 2)] USHORT* Buffer;
#else // MIDL_PASS
    _Field_size_bytes_part_opt_(MaximumLength, Length) PWCH Buffer;
#endif // MIDL_PASS
} UNICODE_STRING;
typedef UNICODE_STRING* PUNICODE_STRING;
typedef struct _CLIENT_ID {
    HANDLE UniqueProcess;
    HANDLE UniqueThread;
} CLIENT_ID,*PCLIENT_ID;
typedef struct _OBJECT_ATTRIBUTES {
    ULONG Length;
    HANDLE RootDirectory;
    PUNICODE_STRING ObjectName;
    ULONG Attributes;
    PVOID SecurityDescriptor; // Points to type SECURITY_DESCRIPTOR
    PVOID SecurityQualityOfService; // Points to type
    SECURITY_QUALITY_OF_SERVICE
} OBJECT_ATTRIBUTES,* POBJECT_ATTRIBUTES;
typedef DWORD (WINAPI* PNTOpenProcess)(
    PHANDLE ProcessHandle,
    ACCESS_MASK DesiredAccess,
    POBJECT_ATTRIBUTES ObjectAttributes,
    PCLIENT_ID ClientId
);
typedef DWORD (WINAPI* PNTAllocateVirtualMemory)(
    HANDLE ProcessHandle,
    PVOID* BaseAddress,
    ULONG_PTR ZeroBits,
    PSIZE_T RegionSize,
    ULONG AllocationType,
    ULONG Protect
);
typedef DWORD(WINAPI* PNTWriteVirtualMemory)(
    IN HANDLE hProcess,
    IN PVOID BaseAddress,
    IN PVOID Buffer,
    IN ULONG BytesToWrite,
    OUT PULONG BytesWritten
);
typedef DWORD(WINAPI* PNTCreateThreadEx)(
    PHANDLE ThreadHandle,
    ACCESS_MASK DesiredAccess,
    LPVOID ObjectAttributes,
    HANDLE ProcessHandle,
    LPTHREAD_START_ROUTINE lpStartAddress,
```

```

    LPVOID lpParameter,
    ULONG CreateThreadFlags,
    SIZE_T ZeroBits,
    SIZE_T StackSize,
    SIZE_T MaximumStackSize,
    LPVOID pUnknow);
typedef void (WINAPI* PRTLInitUnicodeString)(
    PUNICODE_STRING DestinationString,
    PCWSTR SourceString
);

typedef DWORD (WINAPI*PLdrLoadDll)(
    IN PWCHAR PathToFile OPTIONAL,
    IN PULONG Flags OPTIONAL,
    IN PUNICODE_STRING ModuleFileName,
    OUT PHANDLE ModuleHandle);

HANDLE PID = 0;
HMODULE hm = GetModuleHandle("ntdll.dll");
PLdrLoadDll LdrLoadDll = (PLdrLoadDll)GetProcAddress(hm, "LdrLoadDll");
__declspec(naked) void HanShu(PVOID a)
{
    __asm
    {
        pushad;
        mov eax, [a];
        lea ebx, [eax + 0x10];
        push ebx ;
        lea ebx, [eax + 0x8];
        push ebx;
        lea ebx, [eax + 0x14];
        push ebx;
        mov ebx, 0;
        push ebx;
        lea edx, [eax+0x20];
        mov edx, [edx];
        call edx;
        popad;
        ret 0xc;
    }
}

bool GetPIDByName(const char* processName)
{
    // 创建系统快照
    HANDLE hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
    if (hSnapshot == INVALID_HANDLE_VALUE)
    {
        return false;
    }

    PROCESSENTRY32 ps;
    ZeroMemory(&ps, sizeof(PROCESSENTRY32));
    ps.dwSize = sizeof(PROCESSENTRY32);
    if (!Process32First(hSnapshot, &ps))
    {
        return false;
    }
    do

```

```

{
    if (lstrcmpi(ps.szExeFile, processName) == 0)
    {
        // 保存进程ID
        PID = (HANDLE)ps.th32ProcessID;

        CloseHandle(hSnapshot);

        return true;
    }
} while (Process32Next(hSnapshot, &ps));

return false;
}

int main()
{
    GetPIDByName("csgo.exe");
    //获取目标进程句柄
    PNTOpenProcess NtOpenProcess = 0;
    NtOpenProcess = (PNTOpenProcess)GetProcAddress(hm, "NtOpenProcess");
    HANDLE Hprocess = 0;
    CLIENT_ID ClientId;
    ClientId.UniqueProcess = PID;
    ClientId.UniqueThread = 0;
    OBJECT_ATTRIBUTES OBJECTATTRIBUTES = {0};
    OBJECTATTRIBUTES.Length = sizeof(OBJECTATTRIBUTES);
    NtOpenProcess(&Hprocess, PROCESS_ALL_ACCESS, &OBJECTATTRIBUTES, &ClientId);

    //申请内存
    PVOID a = 0;
    SIZE_T RegionSize = 0x1000;
    PNTAllocateVirtualMemory NtAllocateVirtualMemory =
(PNTAllocateVirtualMemory)GetProcAddress(hm, "NtAllocateVirtualMemory");
    NtAllocateVirtualMemory(Hprocess, &a, 0, &RegionSize, MEM_COMMIT |
MEM_RESERVE, PAGE_EXECUTE_READWRITE);

    //DWORD替换HANDLE
    //写入LdrLoad的参数
    typedef struct LdrLoadData
    {
        IN WCHAR          PathToFile = 0;
        IN ULONG          Flags = 0;
        IN UNICODE_STRING ModuleFileName;
        OUT DWORD         ModuleHandle = 0;
    }LdrLoadDataStruct;
    LdrLoadDataStruct LdrLoadData = { 0 };
    PRTLInitUnicodeString RtlInitUnicodeString =
(PRTLInitUnicodeString)GetProcAddress(hm, "RtlInitUnicodeString");
    PCWSTR DLL路径 = L"D:\\ONETAP.dll";
    RtlInitUnicodeString(&LdrLoadData.ModuleFileName, DLL路径);
    //修正DLL路径
    LdrLoadData.ModuleFileName.Buffer=(PWCHAR)((DWORD)a+0x100);

    PVOID BaseAddress = a;
    ULONG BytesToWrite=sizeof(LdrLoadData);
    ULONG BytesWritten = 0;

```

```

    PNTwriteVirtualMemory NtwriteVirtualMemory=
(PNTwriteVirtualMemory)GetProcAddress(hm, "NtwriteVirtualMemory");
    //内存刷0
    NtwriteVirtualMemory(Hprocess, a, 0, 0x1000, &BytesWritten);
    //写入结构体

NtwriteVirtualMemory(Hprocess,BaseAddress,&LdrLoadData,BytesToWrite,&BytesWritten);
    //重写DLL路径
    NtwriteVirtualMemory(Hprocess,PVOID((DWORD)a + 0x100),(PVOID)DLL路径, 0x50,
&BytesWritten);
    //写入函数(Shellcode
    NtwriteVirtualMemory(Hprocess, PVOID((DWORD)a + 0x200), HanShu, 0x200,
&BytesWritten);
    //写入函数地址
    DWORD LdrLoad函数地址 = (DWORD)LdrLoadDll;
    NtwriteVirtualMemory(Hprocess, PVOID((DWORD)a + 0x20), &LdrLoad函数地址, 0x4,
&BytesWritten);

    //执行写入的函数
    HANDLE ThreadHandle = 0;
    PNTCreateThreadEx NtCreateThreadEx=(PNTCreateThreadEx)GetProcAddress(hm,
"NtCreateThreadEx");
    NtCreateThreadEx(&ThreadHandle,PROCESS_ALL_ACCESS,0,Hprocess,
(LPTHREAD_START_ROUTINE)((DWORD)a + 0x200),a,0,0,0,0,NULL);

    CloseHandle(Hprocess);
    CloseHandle(ThreadHandle);
    return 0;
}

```

注意：还有一些细节并未提到。

```

declspec(naked) void HanShu(PVOID a)           //细节: declspec(naked)
{
    __asm
    {
        pushad;
        mov eax, [a]; //细节1。
        lea ebx, [eax + 0x10]; // lea的使用 取逻辑地址 的细节2。
        push ebx;
        lea ebx, [eax + 0x8];
        push ebx;
        lea ebx, [eax + 0x14];
        push ebx;
        mov ebx, 0;
        push ebx;
        lea edx, [eax+0x20]; // 重要细节3, 没有使用TEB和PEB, 直接利用windows特性写入
LdrLoadDLL函数地址
        mov edx, [edx];
        call edx;
        popad;
        ret 0xc; //注意平栈
    }
}

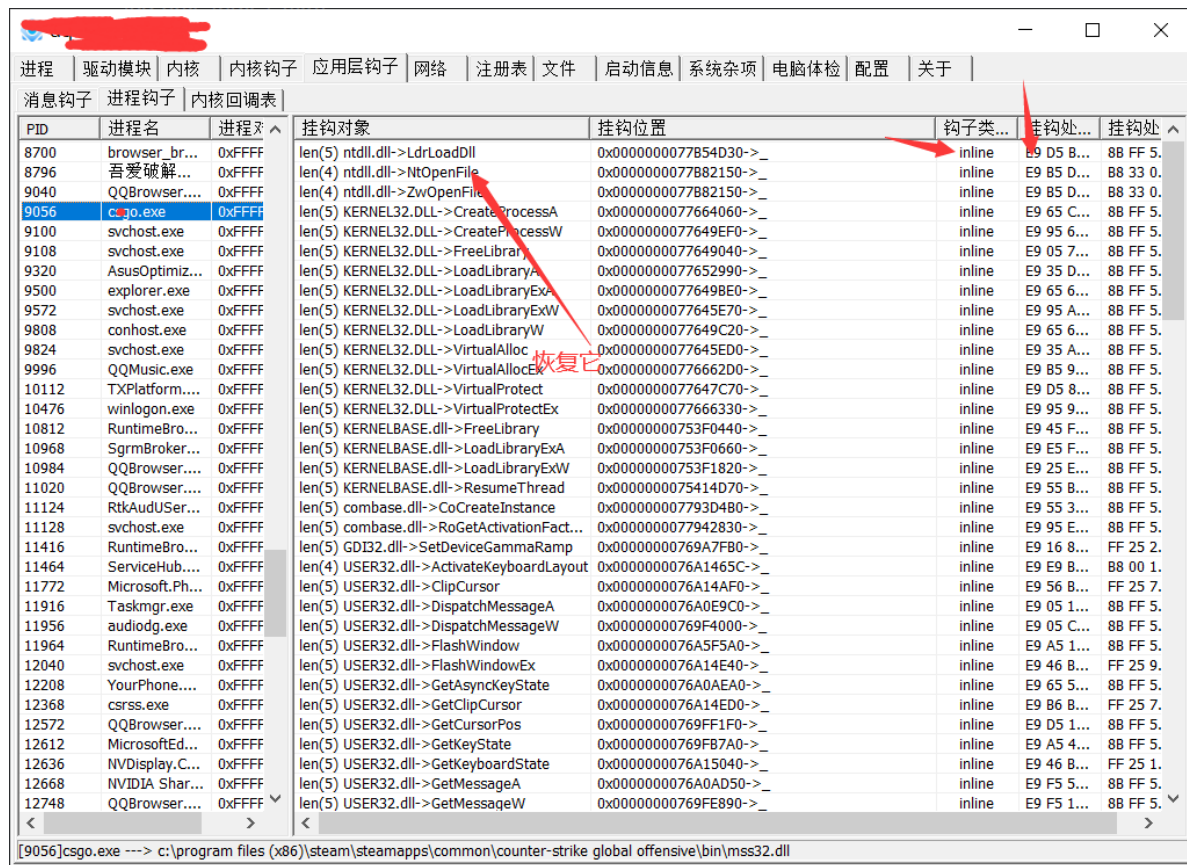
```

比较重要的一个是细节是：这个asm格式的Shellcode的远程调试方法，也是下内存断点访问。

4.实战暴露出的漏洞。

有人可能拿一个cpp保存了上述代码，编译跑起来后，发现对游戏并没有注入成功，但是对其他进程确实成功注入。

这是因为 这个FPS网游 有R3的进程保护。



PID	进程名	进程PPID	挂钩对象	挂钩位置	钩子类...	挂钩处...	挂钩处...
8700	browser_br...	0xFFFF	len(5) ntdll.dll->LdrLoadDll	0x0000000077854D30->	inline	E9 D5 B...	88 FF 5.
8796	吾爱破解...	0xFFFF	len(4) ntdll.dll->NtOpenFile	0x0000000077882150->	inline	E9 B5 D...	88 33 0.
9040	QQBrowser....	0xFFFF	len(4) ntdll.dll->ZwOpenFile	0x0000000077882150->	inline	E9 B5 D...	88 33 0.
9056	csq.exe	0xFFFF	len(5) KERNEL32.DLL->CreateProcessA	0x0000000077664060->	inline	E9 65 C...	88 FF 5.
9100	svchost.exe	0xFFFF	len(5) KERNEL32.DLL->CreateProcessW	0x0000000077649EF0->	inline	E9 95 6...	88 FF 5.
9108	svchost.exe	0xFFFF	len(5) KERNEL32.DLL->FreeLibrary	0x0000000077649040->	inline	E9 05 7...	88 FF 5.
9320	AsusOptimiz...	0xFFFF	len(5) KERNEL32.DLL->LoadLibraryExA	0x0000000077652990->	inline	E9 35 D...	88 FF 5.
9500	explorer.exe	0xFFFF	len(5) KERNEL32.DLL->LoadLibraryExW	0x00000000776498E0->	inline	E9 65 6...	88 FF 5.
9572	svchost.exe	0xFFFF	len(5) KERNEL32.DLL->LoadLibraryExW	0x0000000077645E70->	inline	E9 95 A...	88 FF 5.
9808	conhost.exe	0xFFFF	len(5) KERNEL32.DLL->LoadLibraryW	0x0000000077649C20->	inline	E9 65 6...	88 FF 5.
9824	svchost.exe	0xFFFF	len(5) KERNEL32.DLL->VirtualAlloc	0x0000000077645ED0->	inline	E9 35 A...	88 FF 5.
9996	QQMusic.exe	0xFFFF	len(5) KERNEL32.DLL->VirtualAllocEx	0x00000000776662D0->	inline	E9 B5 9...	88 FF 5.
10112	TXPlatform....	0xFFFF	len(5) KERNEL32.DLL->VirtualProtect	0x0000000077647C70->	inline	E9 05 8...	88 FF 5.
10476	winlogon.exe	0xFFFF	len(5) KERNEL32.DLL->VirtualProtectEx	0x0000000077666330->	inline	E9 95 9...	88 FF 5.
10812	RuntimeBro...	0xFFFF	len(5) KERNELBASE.dll->FreeLibrary	0x00000000753F0440->	inline	E9 45 F...	88 FF 5.
10968	SgrmBroker...	0xFFFF	len(5) KERNELBASE.dll->LoadLibraryExA	0x00000000753F0660->	inline	E9 E5 F...	88 FF 5.
10984	QQBrowser....	0xFFFF	len(5) KERNELBASE.dll->LoadLibraryExW	0x00000000753F1820->	inline	E9 25 E...	88 FF 5.
11020	QQBrowser....	0xFFFF	len(5) KERNELBASE.dll->ResumeThread	0x0000000075414D70->	inline	E9 55 8...	88 FF 5.
11124	RtkAudUser...	0xFFFF	len(5) combase.dll->CoCreateInstance	0x000000007793D480->	inline	E9 55 3...	88 FF 5.
11128	svchost.exe	0xFFFF	len(5) combase.dll->RoGetActivationFact...	0x0000000077942830->	inline	E9 95 E...	88 FF 5.
11416	RuntimeBro...	0xFFFF	len(5) GDI32.dll->SetDeviceGammaRamp	0x00000000769A7F80->	inline	E9 16 8...	FF 25 2.
11464	ServiceHub....	0xFFFF	len(4) USER32.dll->ActivateKeyboardLayout	0x0000000076A1465C->	inline	E9 E9 B...	88 00 1.
11772	Microsoft.Ph...	0xFFFF	len(5) USER32.dll->ClipCursor	0x0000000076A14AF0->	inline	E9 56 8...	FF 25 7.
11916	Taskmgr.exe	0xFFFF	len(5) USER32.dll->DispatchMessageA	0x0000000076A0E9C0->	inline	E9 05 1...	88 FF 5.
11956	audiodg.exe	0xFFFF	len(5) USER32.dll->DispatchMessageW	0x00000000769F4000->	inline	E9 05 C...	88 FF 5.
11964	RuntimeBro...	0xFFFF	len(5) USER32.dll->FlashWindow	0x0000000076A5F5A0->	inline	E9 A5 1...	88 FF 5.
12040	svchost.exe	0xFFFF	len(5) USER32.dll->FlashWindowEx	0x0000000076A14E40->	inline	E9 46 8...	FF 25 9.
12208	YourPhone....	0xFFFF	len(5) USER32.dll->GetAsyncKeyState	0x0000000076A0AEA0->	inline	E9 65 5...	88 FF 5.
12368	csrss.exe	0xFFFF	len(5) USER32.dll->GetClipCursor	0x0000000076A14ED0->	inline	E9 B6 8...	FF 25 7.
12572	QQBrowser....	0xFFFF	len(5) USER32.dll->GetCursorPos	0x00000000769FF1F0->	inline	E9 05 1...	88 FF 5.
12612	MicrosoftEd...	0xFFFF	len(5) USER32.dll->GetKeyState	0x00000000769FB7A0->	inline	E9 A5 4...	88 FF 5.
12636	NVDisplay.C...	0xFFFF	len(5) USER32.dll->GetKeyboardState	0x0000000076A15040->	inline	E9 46 8...	FF 25 1.
12668	NVIDIA Shar...	0xFFFF	len(5) USER32.dll->GetMessageA	0x0000000076A0AD50->	inline	E9 F5 5...	88 FF 5.
12748	QQBrowser....	0xFFFF	len(5) USER32.dll->GetMessageW	0x00000000769FE890->	inline	E9 F5 1...	88 FF 5.

代码实现：

```
//恢复钩子
ULONG 实际写入的大小 = 0;
char Bytes[5] = { 0 };
PVOID NtOpenFile = GetProcAddress(hm, "NtOpenFile");
memcpy(Bytes, NtOpenFile, 5);
WriteProcessMemory(Hprocess, NtOpenFile, Bytes, 0x5, &实际写入的大小);
//NtWriteVirtualMemory(Hprocess, (PVOID)0x07800000, Bytes, 0x5, &实际写入的大
小);
```

为什么用WriteProcessMemory，而不用NtWriteVirtualMemory？

这是因为

[csgo.exe]进程内存(3211)

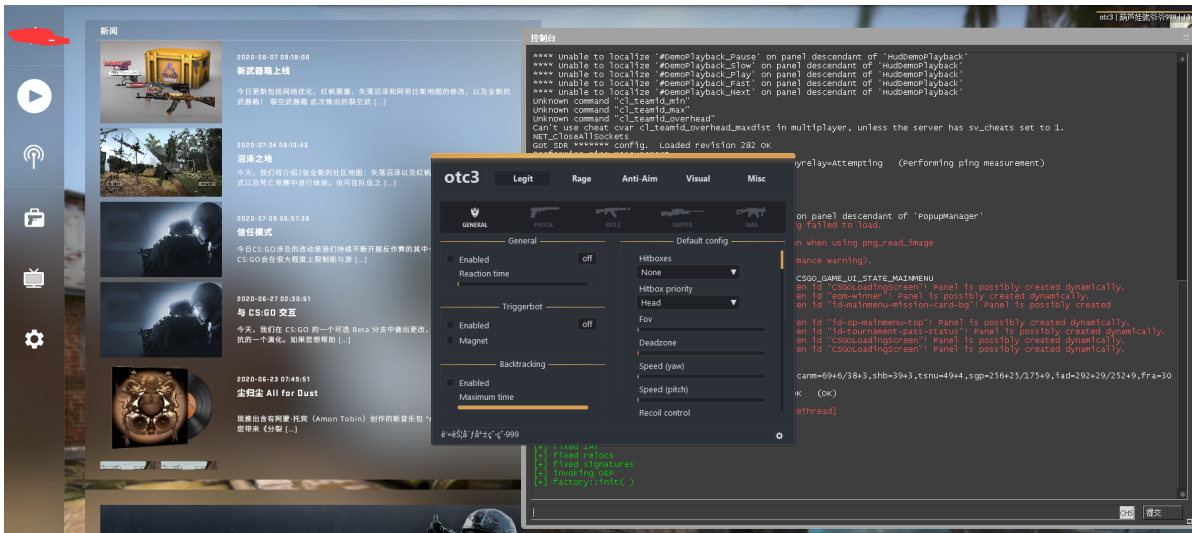
地址	大小	Protect	State	Type	模块名
0x0000000077B00000	0x000000...	Read	Commit	Image	wow64cpu
0x0000000077B01000	0x000000...	ReadExecute	Commit	Image	wow64cpu
0x0000000077B03000	0x000000...	Read	Commit	Image	wow64cpu
0x0000000077B04000	0x000000...	ReadWrite	Commit	Image	wow64cpu
0x0000000077B05000	0x000000...	Read	Commit	Image	wow64cpu
0x0000000077B06000	0x000000...	ReadExecute	Commit	Image	wow64cpu
0x0000000077B07000	0x000000...	Read	Commit	Image	wow64cpu
0x0000000077B09000	0x000000...	No Access	Free		
0x0000000077B10000	0x000000...	Read	Commit	Image	ntdll.dll
0x0000000077B11000	0x000000...	ReadExecute	Commit	Image	ntdll.dll
0x0000000077C2B000	0x000000...	ReadWrite	Commit	Image	ntdll.dll
0x0000000077C31000	0x000000...	Read	Commit	Image	ntdll.dll
0x0000000077CAA000	0x000000...	No Access	Free		
0x0000000077CB0000	0x000000...	ReadWriteExecute	Commit	Private	
0x0000000077CB1000	0x000000...	No Access	Free		

地址：0000000077B11000

大小：00000000011A000

Dump

没有写的权限。WriteProcessMemory 默认调用 NtProtectVirtualMemory 来更改写的权限，但是用 NtWriteVirtualMemory 需要自己手动实现 NtProtectVirtualMemory。



Have Fun!

By: 0x太上

By: 0x太上

By: 0x太上