

一次开发的意外逆向之旅


笔者最近从事 windows 内核开发的时候因为功能需要，所以需要 PspSetCreateProcessNotifyRoutine 回调函数数组进行遍历，于是笔者照往常思路在获取 PspCreateProcessNotifyRoutine 的时候发现了一些很有意思的事情，特此拿出来与诸君分享一下，第一次写文章难免有不足，希望诸位看客可以海涵，再次拜谢

一、首先我们先说说如何获取这个数组的思路，以 Windows7X64 为例，其他各版本类似，首先

PspCreateProcessNotifyRoutine 是一个 PVOID 的指针，它里面存放的是通过 PsSetCreateProcessNotifyRoutine 这个函数设置的各种回调函数。那我们怎么获取这个数组呢很简单：

- 1、我们先通过 MmGetSystemRoutineAddress 来获取 PsCreateProcessNotifyRoutine 这个函数的地址，通过这个地址我们不难发现在这个函数中调用了 PspSetCreateProcessNotifyRoutine 这个函数

```
3: kd> U PsSetCreateProcessNotifyRoutine
nt!PsSetCreateProcessNotifyRoutine:
fffff800`0431f3c0 4533c0 xor     r8d,r8d
fffff800`0431f3c3 e9e8fdffff jmp     nt!PspSetCreateProcessNotifyRoutine (fffff800`0431f1b0)
fffff800`0431f3c8 90      nop
fffff800`0431f3c9 90      nop
fffff800`0431f3ca 90      nop
fffff800`0431f3cb 90      nop
fffff800`0431f3cc 90      nop
fffff800`0431f3cd 90      nop
```




a)

- 2、紧接着在 PspSetCreateProcessNotifyRoutine 这个函数中我们会发现在 0x33 偏移位置不难发现有一次对 PspCreateProcessNotifyRoutine 的操作

```
nt!PspSetCreateProcessNotifyRoutine:
fffff800`0431f1b0 48895c2408 mov     qword ptr [rsp+8],rbx
fffff800`0431f1b5 48896c2410 mov     qword ptr [rsp+10h],rbp
fffff800`0431f1ba 4889742418 mov     qword ptr [rsp+18h],rsi
fffff800`0431f1bf 57      push    rdi
fffff800`0431f1c0 4154     push    r12
fffff800`0431f1c2 4155     push    r13
fffff800`0431f1c4 4156     push    r14
fffff800`0431f1c6 4157     push    r15
fffff800`0431f1c8 4883ec20 sub     rsp,20h
fffff800`0431f1cc 4533e4 xor     r12d,r12d
fffff800`0431f1cf 418ae8 mov     bpl,r8b
fffff800`0431fd2 4c8be9 mov     r13,rcx
fffff800`0431fd5 418d5c2401 lea     ebx,[r12+1]
fffff800`0431f1da 413ad4 cmp     dl,r12b
fffff800`0431f1dd 0f840e010000 je      nt!PspSetCreateProcessNotifyRoutine+0x141 (fffff800`0431f2f1)

nt!PspSetCreateProcessNotifyRoutine+0x33:
fffff800`0431f1e3 65488b3c2588010000 mov     rdi,qword ptr gs:[188h]
fffff800`0431f1ec 83c8ff or      eax,0FFFFFFFh
fffff800`0431f1ef 660187c4010000 add     word ptr [rdi+1C4h],ax
fffff800`0431f1f6 4c8d358395d6ff lea     r14,[nt!PspCreateProcessNotifyRoutine (fffff800`04088780)]

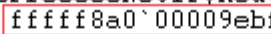
nt!PspSetCreateProcessNotifyRoutine+0x4d:
fffff800`0431f1fd 418bc4 mov     eax,r12d
fffff800`0431f200 4d8d3cc6 lea     r15,[r14+rax*8]
fffff800`0431f204 498bcf mov     rcx,r15
fffff800`0431f207 e8f4ecedff call    nt!ExReferenceCallBackBlock (fffff800`041fd00)
fffff800`0431f20c 33d2 xor     edx,edx
fffff800`0431f20e 488bf0 mov     rsi,rax
fffff800`0431f211 483bc2 cmp     rax,rdx
fffff800`0431f214 743d je      nt!PspSetCreateProcessNotifyRoutine+0xa3 (fffff800`0431f253)
```



a)

- 3、紧接着我们跟入这个函数地址就可以看到这是一张函数地址表，但是我们通过 uf 这些地址会发现这些地址是错误的，这是为什么呢？

```
3: kd> dp PspCreateProcessNotifyRoutine
fffff800`04088780 fffff8a0`00009ebf fffff8a0`0008878f
fffff800`04088790 fffff8a0`002eb0df fffff8a0`002b5bbf
fffff800`040887a0 fffff8a0`003de38f fffff8a0`00e5509f
fffff800`040887b0 fffff8a0`01468cef 00000000`00000000
fffff800`040887c0 00000000`00000000 00000000`00000000
fffff800`040887d0 00000000`00000000 00000000`00000000
fffff800`040887e0 00000000`00000000 00000000`00000000
fffff800`040887f0 00000000`00000000 00000000`00000000
```



a)

4、其实很简单，微软对这些函数都进行了加密，如果要取得真实的函数地址必须先对其进行解密，其实很简单只要取出来这些地址对其 &0xffffffff 就可以了

a)

```

3: kd> db 0xfffff8a0`00009eb8
fffff8a0`00009eb8 f0 ca ea 03 00 f8 ff ff-00 00 00 00 00 00 00 00 .....
fffff8a0`00009ec8 d4 a9 f0 14 fb 0f 3b 08-03 01 03 03 4f 62 44 69 ...p.....ObDi
fffff8a0`00009ed8 18 05 00 00 00 00 00-00 00 00 00 00 00 00 00 .....
fffff8a0`00009ee8 e0 31 d2 30 80 fa ff ff-ec 32 7e 9a 00 f8 ff ff .....1.0.....2"
fffff8a0`00009ef8 22 69 16 04 00 f8 ff ff-03 01 04 03 4f 62 4e 6d "i.....ObNm
fffff8a0`00009f08 70 7e e8 03 00 f8 ff ff-53 00 79 00 73 00 74 00 p.....S.y.s.t.
fffff8a0`00009f18 65 00 6d 00 45 00 72-00 72 00 6f 00 72 00 50 00 e.m.E.r.r.o.r.P.
fffff8a0`00009f28 6f 00 72 00 74 00 52-00 65 00 61 00 64 00 79 00 o.r.t.R.e.a.d.y.
3: kd> dp 0xfffff8a0`00009eb8
fffff8a0`00009eb8 fffff800`03eacaf0 00000000`00000000
fffff8a0`00009ec8 083b0fffb`1470a9d4 6944624f`03030103
fffff8a0`00009ed8 00000000`00000518 00000000`00000000
fffff8a0`00009ee8 fffff800`30d231e0 fffff800`9a7e32ce
fffff8a0`00009ef8 fffff800`04166922 6d4e624f`03040103
fffff8a0`00009f08 fffff800`03e87e70 00740073`00790053
fffff8a0`00009f18 00720045`006d0065 00500072`006f0072
fffff8a0`00009f28 00520074`0072006f 00790064`00610065

```

a)

The screenshot displays a Windows task manager window with the 'CPU' tab selected, showing a usage of 98%. In the background, there are two windows:

- A debugger window (likely Immunity Debugger) showing assembly code for a process named 'nviVCreateProcessCallback.exe'. The code includes instructions like 'mov', 'xor', 'push', 'pop', 'ret', and 'call'. A red arrow points to the instruction 'xor rax, rax'.
- A command prompt window showing the execution of the command 'taskmgr /v'. The output lists various system components and their status, such as '进程 [运行中] 内存 [可用]', '网络 [已连接]', etc.

a)

7、在拿到这张函数表之后对比模块的基地址和模块大小我们不难确定函数所属的模块归属谁所有，同样我们这时候根据模块名在 PsSetCreateProcessNotifyRoutine 的函数地址传入有目的地址和 True 之后，那么该模块的挂钩也自然而软取消掉了，除此之外也可以直接对其函数头部进行 Ret 0，不过可能出现一些问题所以不推荐这么做。

二、那么我遇到了什么问题呢，在 win7x86 版本上同样通过 MmGetSystemRoutineAddress 来获取 PsCreateProcessNotifyRoutine 这个函数地址的时候居然发现这个地址虽然是一个函数地址，但是这个地址居然是错的，因为我发现在 Windbg 中 uf 获取的 PsCreateProcessNotifyRoutine 和通过 MmGetSystemRoutineAddress 这个拿到的地址居然不一样！最开始我百思不得其解，很疑惑这是为什么，但是在冷静下来之后我开始慢慢分析这个过程，有趣的旅程就这么开始了！

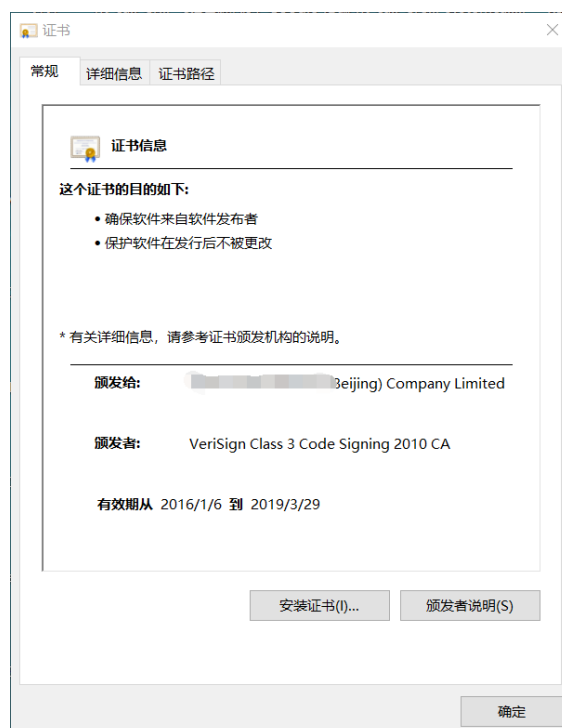
1、首先 windbg 的 u 一定是准确的，这个是根本，微软自家的调试器在自己函数有符号表的情况下都不知道在哪的话那么一首凉凉送给他不为过！而且确实 u 到的函数我进入看过也是正确的，于是我猜想问题一定在 MmGetSystemRoutineAddress 这个关键函数上

2、那么 MmGetSystemRoutineAddress 他是怎么做的呢，其实在翻阅了现有的资料之后我解开了我的疑惑，在 MmGetSystemRoutineAddress 的内部其实是解析了模块文件的 EAT 也就是我们俗称的导出表来获取函数调用者所需求的函数，通过对 EAT 的解析以及和模块基地址的运算结合 ImageLoad 的对齐方式，返回对应的函数位置，于是我们的思路就有了，因为是 X86 的操作系统，在没有 KPP 保护的情况下很有可能我的内核的 EAT 被一些三方软件挂了钩子，导致我获取函数不正确，于是在 windbg 中.reload 装载所有模块信息后，lm 一下所有模块地址也就出来了，对比看了一下各个模块基地址和模块大小也就大概确定是属于哪个模块，PCHunter 的内核挂钩也证明了我的猜想

三、模块分析

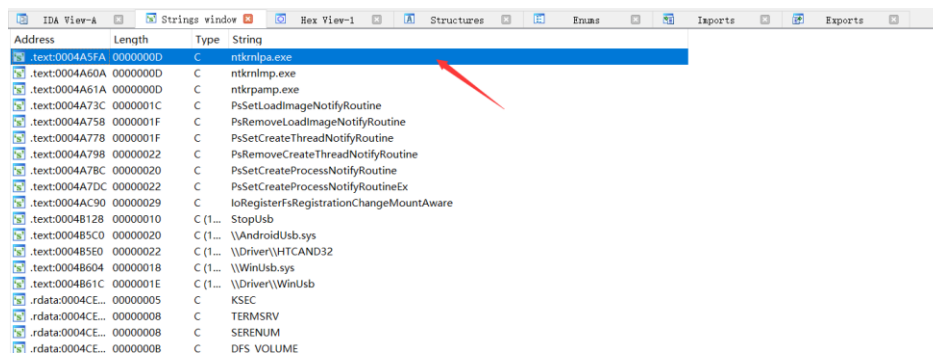
既然拿到了这个模块确认内核是被 EAT 挂钩那么不妨分析一下这个模块好了

1、通过数字签名我们不难看出这个模块是一个安全软件的模块，到这里困惑也就解开了，这里我们猜测一下是因为在 x86 系统上没有 kpp 保护，为了保护自身的钩子不被卸载或者为了监控别人的钩子，所以安全软件选择这个地点进行 EAT 挂钩用以保护自身的挂钩不被其他软件取消，这是一个很好的思路，那么到底是不是这样呢，我们接着分析



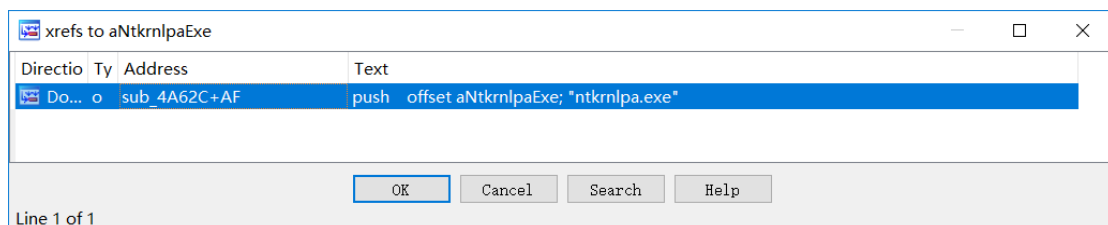
2、

- 3、既然我们已经知道被替换函数的地址了，也知道被 EAT 挂钩的名称，那么我们接下来从这两点开始进行逆向，首先我们先去找找字符串信息，根据模块名称



| Address | Length | Type | String |
|------------------|----------|---------|--|
| .text:0004A5FA | 0000000D | C | ntkrnlpa.exe |
| .text:0004A60A | 0000000D | C | ntkrnlmp.exe |
| .text:0004A61A | 0000000D | C | ntkrpamp.exe |
| .text:0004A73C | 0000001C | C | PsSetLoadImageNotifyRoutine |
| .text:0004A758 | 0000001F | C | PsRemoveLoadImageNotifyRoutine |
| .text:0004A778 | 0000001F | C | PsSetCreateThreadNotifyRoutine |
| .text:0004A798 | 00000022 | C | PsRemoveCreateThreadNotifyRoutine |
| .text:0004A7BC | 00000020 | C | PsSetCreateProcessNotifyRoutine |
| .text:0004A7DC | 00000022 | C | PsSetCreateProcessNotifyRoutineEx |
| .text:0004A79C | 00000029 | C | IoRegisterFsRegistrationChangeMountAware |
| .text:0004B128 | 00000010 | C (1... | StopUsb |
| .text:0004B5C0 | 00000020 | C (1... | \\AndroidUsb.sys |
| .text:0004B5E0 | 00000022 | C (1... | \\Driver\\HTCAND32 |
| .text:0004B604 | 00000018 | C (1... | \\WinUsb.sys |
| .text:0004B61C | 0000001E | C (1... | \\Driver\\WinUsb |
| .rdata:0004CE... | 00000005 | C | KSEC |
| .rdata:0004CE... | 00000008 | C | TERMSRV |
| .rdata:0004CE... | 00000008 | C | SERENUM |
| .rdata:0004CE... | 00000008 | C | DFS VOLUME |

- 4、
- 5、我们先根据字符串找到对这个字符串引用的地址，很明显只有这一处，我们跟进去，结合上下文看到了很关键的一个函数 ZwQuerySystemInformation，到这里其实有过内核开发经验的小伙伴们肯定已经猜到了这个函数就是在获取模块基地址，那么看他对于 v4 这个参数的引用应该就是需要返回的模块地址，那么我们对这个函数命名为 GetKeyModuleAddress,同时参数返回的就是模块大小

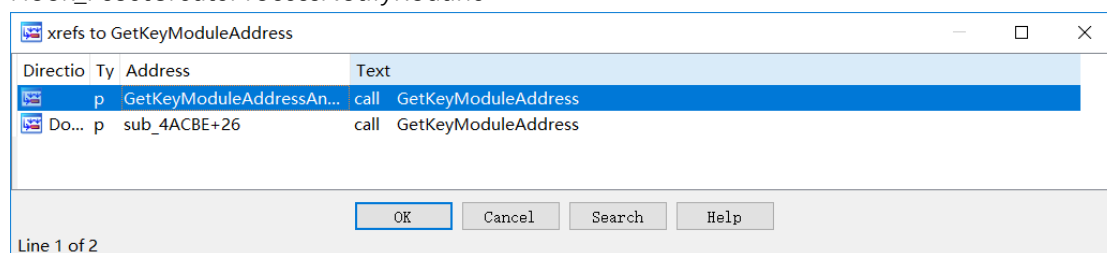


| Directio | Ty | Address | Text |
|----------|----|--------------|--|
| Do... | o | sub_4A62C+AF | push offset aNtkrnlpaExe; "ntkrnlpa.exe" |

- 6、
- ```
1
if (ZwQuerySystemInformation(SystemModuleInformation, v1, ResultLength, &ResultLength) < 0)
{
 ExFreePoolWithTag(v1, 0);
 return 0;
}
v3 = *v1;
v4 = (int)(v1 + 1);
v5 = v3;
v8 = 0;
if (v3 > 0)
{
 while (1)
 {
 v7 = (char *)*((unsigned __int16 *) (v4 + 26) + v4 + 28);
 if (!strcmp(v7, "ntoskrnl.exe")
 || !strcmp(v7, "ntkrnlpa.exe")
 || !strcmp(v7, "ntkrnlmp.exe")
 || !strcmp(v7, "ntkrpamp.exe"))
 {
 break;
 }
 v4 += 284;
 if (++v8 >= v5)
 goto LABEL_16;
 }
 RetModuleBase_v10 = *(_DWORD *) (v4 + 8);
 *ULModuleSize = *(_DWORD *) (v4 + 12);
}
ABEL_16:
ExFreePoolWithTag(P, 0);
}
return RetModuleBase_v10;
```

- 7、
- 8、紧接着我们对其 x 进行调用分析，可以看到有两处，我们跟入第一处，很幸运的找到了直接找到了需要的函数，在这个函数里面我们可以看到大量的关键函数的字符串而我们的 PsSetCreateProcessNotifyRoutine 也在其中，这个时候我们的第二条主线就排上用场了，我们可以看到下图中使用 PsSetCreateProcessNotifyRoutine 这个字符串的函数也引

用了 sub\_4A504 这个函数，而这个函数正是我们内核被挂钩 EAT 后跳过去的函数，所以我们猜测 sub\_49F84 这个函数应该是 GetProcAddressAndSetHook 这两个功能，于是我们对其命名 GetProcAddressAndSetHook，同时对 sub\_49F84 这个函数命名为 Hook\_PsSetCreateProcessNotifyRoutine



9、

```

 ulBaseSize_v2,
 (int)&DestinationString,
 (int)sub_4A480,
 &dword_52F64) >= 0)
 {
 RtlInitAnsiString(&DestinationString, "PsRemoveCreateThreadNotifyRoutine");
 GetProcAddressAndSetHook(
 v10,
 v9,
 _ModuleAddress_v1,
 ulBaseSize_v2,
 (int)&DestinationString,
 (int)sub_4A4CC,
 &dword_52F68);
 }
 RtlInitAnsiString(&DestinationString, "PsSetCreateProcessNotifyRoutine");
 _ModuleAddress = GetProcAddressAndSetHook(
 v12,
 v11,
 _ModuleAddress_v1,
 ulBaseSize_v2,
 (int)&DestinationString,
 (int)Hook_PsSetCreateProcessNotifyRoutine,
 &OldPsSetCreateProcessNotifyRoutine);
 if (*(&qword_536AC + 1) >= 6)
 {
 RtlInitAnsiString(&DestinationString, "PsSetCreateProcessNotifyRoutineEx");
 _ModuleAddress = GetProcAddressAndSetHook(
 v14,
 v13,
 _ModuleAddress_v1,
 ulBaseSize_v2,
 (int)&DestinationString,
 (int)sub_4A57A,
 &dword_52F70);
 }

```

10、

- 11、紧接着我们对 GetProcAddressAndSetHook (sub\_4A504) 的流程进行分析, 首先进入该函数后根据传入参数我们不难发现有一个函数名称还有一个 hook 的函数地址, 根据使用这两项的函数, 我们猜测 sub\_49DDE 这个函数的功能是获取函数地址, 在他调用的函数 sub\_49D60 可以明显看到存在 PE 特征 0x5A 0x4D 以及 0x45 0x50 这几个字节码, 所以这个函数肯定是根据函数名称获取函数地址无疑, 我们将其命名为 GetFunAddress (sub\_49D60), 其首参数为返回需求函数的地址

```
unsigned int __fastcall CheckPe(unsigned int ulBaseSize, unsigned int ModuleAddress, unsigned __int16 a3, _DWORD *a4)
{
 unsigned int v5; // eax
 unsigned int v6; // eax
 int v7; // esi

 if (ModuleAddress & 1)
 return 0;
 if (ulBaseSize < 0x1000)
 return 0;
 if (!ModuleAddress)
 return 0;
 if (ModuleAddress == -1)
 return 0;
 if (*((_WORD *)ModuleAddress) != 0x5A4D)
 return 0;
 v5 = *((_DWORD *)ModuleAddress + 60);
 if (v5 >= ulBaseSize - 24)
 return 0;
 v6 = ModuleAddress + v5;
 if (*((_DWORD *)v6) != 0x4550)
 return 0;
 if (v6 < ModuleAddress)
 return 0;
 if (v6 >= ModuleAddress + ulBaseSize - 248)
 return 0;
 if (*((_WORD *)v6 + 24) != 267)
 return 0;
 if ((unsigned int)a3 >= *((_DWORD *)v6 + 116))
 return 0;
 v7 = *((_DWORD *)v6 + 8 * a3 + 120);
}
```

12、

```
unsigned int __fastcall GetFunAddress(unsigned int ModuleAddress, int a2, unsigned int ulBaseSize, UNICODE_STRING *pustrFunName)
{
 unsigned int v4; // edi
 unsigned int v5; // eax
 _DWORD *v6; // ebx
 int v7; // eax
 unsigned int v8; // edx
 unsigned int v9; // ecx
 unsigned int v10; // ecx
 int v11; // esi
 _BYTE *v12; // eax
 int v13; // eax
 unsigned __int16 v14; // si
 unsigned int v15; // eax
 unsigned int *v16; // ecx
 unsigned int v17; // eax
 unsigned int v18; // edi
 unsigned int *result; // eax
 int v20; // [esp+24h] [ebp-30h]
 unsigned int v21; // [esp+28h] [ebp-2Ch]
 int v22; // [esp+2Ch] [ebp-28h]
 int v23; // [esp+30h] [ebp-24h]
 unsigned int v24; // [esp+34h] [ebp-20h]
 int v25; // [esp+38h] [ebp-1Ch]
 CPPEH_RECORD ms_exc; // [esp+3Ch] [ebp-18h]

 v4 = ModuleAddress;
 if (ModuleAddress < (unsigned int)PeSystemRangeStart || ModuleAddress + ulBaseSize < ModuleAddress)
 return 0;
 ms_exc.registration.TryLevel = 0;
 v5 = CheckPe(ulBaseSize, ModuleAddress, 0, &v20);
 v6 = (_DWORD *)v5;
 if (!v5)
 goto LABEL_32;
 if (v5 < v4)
 goto LABEL_32;
}
```

13、

- 14、紧接着跟入 sub\_49B6E 这个函数进行分析, 我们可以看到明显的内存页操作, 调用了 IoAllocateMdl, MmProbeAndLockPages, MmMapLockedPagesSpecifyCache 这几个关键函数, 这很明显是申请 MDL 对内存页进行锁定防止换页造成缺页异常等问题, 这一般是 hook 的必要操作, 所以我们对其命名为 LockPage

```
int __stdcall LockPage(PVOID VirtualAddress, ULONG Length, int a3, int a4)
{
 struct _MDL *v4; // eax
 struct _MDL *v5; // edi
 PVOID v7; // eax

 *((_DWORD *)a4) = 0;
 v4 = IoAllocateMdl(VirtualAddress, Length, 0, 0, 0);
 v5 = v4;
 if (!v4)
 return 0xC000009A;
 MmProbeAndLockPages(v4, 0, IoModifyAccess);
 v7 = MmMapLockedPagesSpecifyCache(v5, 0, MmCached, 0, 0, NormalPagePriority);
 *((_DWORD *)a3) = v7;
 if (!v7)
 ExRaiseStatus(-1073741670);
 *((_DWORD *)a4) = v5;
 return 0;
}
```

15、



- 16、 那么在完成页锁定之后，必然就是进行 hook 操作，这里采用了一个很好的方法并没有使用 memcpy 或者其他内存填充写入的方法，而是采用了原子操作，这种一箭双雕的方法，首先使用 \_InterlockedExchange 这原子操作交换函数可以很方便的解决了同步问题，其次在 \_InterlockedExchange 调用的时候返回值是上一次的状态，也很方便的保存了上一次的地址，以便于恢复，所以说是一种一箭双雕的方法，InterlockedExchange 的第二个参数也使用到了我们的传入地址，以及刚才 LockPage

```
signed int __fastcall GetProcAddressAndSetHook(int ecx0, int edx0, int ModuleAddress, int ulBaseSize, int pustrFunName, int Hook_Address, _DWORD *SaveAddress)
{
 unsigned int *v7; // eax
 signed __int32 v8; // eax
 PVOID v9; // esi
 struct _MDL *v10; // STQC_4
 PMDL MemoryDescriptorList; // [esp+4h] [ebp-8h]
 PVOID BaseAddress; // [esp+8h] [ebp-4h]

 BaseAddress = 0;
 MemoryDescriptorList = 0;
 v7 = GetFunAddress(ModuleAddress, edx0, ulBaseSize, (unsigned __int16 *)pustrFunName);
 if (!v7 || LockPage(v7, 4u, (int)&BaseAddress, (int)&MemoryDescriptorList) < 0)
 return 0xC0000001;
 v9 = BaseAddress;
 v8 = _InterlockedExchange((volatile signed __int32 *)BaseAddress, Hook_Address - ModuleAddress);
 v10 = MemoryDescriptorList;
 *SaveAddress = ModuleAddress + v8;
 UnlockPage(v9, v10);
 return 0;
}
```

- 17、
- 18、 在完成原子交换之后，GetProcAddressAndSetHook 的第五个参数被使用，这里可以看到使用结束之后，之前的地址被保存下来，所以可以论证这里是用于恢复使用的，而且结合外面的函数传入值来看这里是一个全局对象，而且这里这个值在 hook 的函数之中仍然需要去调用，所以也论证了这一点
- 19、 在之后紧接着调用了 sub\_4B340 这个函数在这个函数中就是一些基本的解除页面锁定的函数，我们将其命名为 UnlockPage
- 至此 Hook 的全套流程就已经分析完毕了，接下来我们来看一看 hook 掉的代理函数做了一些什么

### 三、代理函数分析

```
signed int __userpurgate Hook_PsSetCreateProcessNotifyRoutine@<eax>(int ebx0@<ebx>, int pFunAddress, bool bFlag)
{
 int CallAddress; // edi
 int bCallOldRet; // esi
 int savedregs; // [esp+8h] [ebp+0h]

 CallAddress = GetCallAddress((int)&savedregs);
 if (bFlag)
 // True为取消
 // False为设置
 //
 {
 bCallOldRet = calloldPsSetCreateProcessNotifyRoutine(pFunAddress, bFlag);
 if (bCallOldRet >= 0)
 LogAboutInformation(ebx0, 3, 1, pFunAddress, CallAddress, 1u);
 }
 else
 {
 bCallOldRet = calloldPsSetCreateProcessNotifyRoutine(pFunAddress, 0);
 if (LogAboutInformation(ebx0, 3, 0, pFunAddress, CallAddress, bCallOldRet >= 0))
 {
 calloldPsSetCreateProcessNotifyRoutine(pFunAddress, 1);
 return 0xC0000022;
 }
 }
 return bCallOldRet;
}
```

- 1、
- 2、 首先第一个函数 sub\_4A3F2 的操作非常奇怪，该函数作为替换函数应该是一个两参函数，但是很不幸 IDA 分析失败了，最开始因为经验欠缺我没有明白这个函数的意义，但随着之后的分析我茅塞顿开，这个函数是通过栈寄存器来获取调用地址的，因为在栈上有函数的调用地址，所以在之后的 LogAboutInformation 中会有使用

```
int __usercall sub_4A3F2@eax(int a1@ebp)
{
 return *(_DWORD*)(a1 + 4);
}
```

- 3、
- 4、根据 Hook\_PsSetCreateProcessNotifyRoutine 的第二个参数 True or False 来确定具体流程，无论是在取消设置还是在设置函数中都会调用 sub\_49CE0 这个函数，这个函数的唯一作用就是调用之前保存下来给全局变量的原始的 PsSetCreateProcessNotifyRoutine，我们将其命名为 CallRightOldPsSetCreaetProcess

```
int __stdcall __spoils<ecx> sub_49CE0(int pFunAddress, bool bFlag)
{
 int result; // eax

 if (OldPsSetCreaeProcessNotifyRoutine)
 result = OldPsSetCreaeProcessNotifyRoutine(pFunAddress, bFlag);
 else
 result = 0xC0000001;
 return result;
}
```

- 5、
- 6、紧接着会根据对于 PsSetCreateProcessNotifyRoutine 调用和失败会进入到 LogAboutInformation (sub\_4A2C8) 这个函数中，跟入该函数结合传入参数分析该函数的唯一意义就是获取设置的函数地址模块名称以及调用者的模块名称，对其进行格式化之后结合特定标志位上传到 r3 上

```
__userpurg LogAboutInformation@cal>(int a1@ecx, int SendType, bool bSetFlag, int FunAddress, int CallAddress, unsigned __int0 CallOldRetValue)
{
 int result; // a1
 _DWORD *v7; // eax
 void *v8; // esi
 PERESOURCE i; // edi
 int v10; // [esp+8h] [ebp-1ch]
 _UNICODE_STRING UnicodeString; // [esp+4h] [ebp-10h]
 int v12; // [esp+Ch] [ebp-8h]
 char v13; // [esp+13h] [ebp-1h]

 v13 = 0;
 v12 = 0;
 UnicodeString.Length = 0;
 UnicodeString.MaximumLength = 0;
 UnicodeString.Buffer = 0;
 if (!GetCurrentTid() || bSetFlag)
 return 0;
 if (GetModuleFromAddress(CallAddress, &UnicodeString) < 0
 && GetModuleFromAddress(FunAddress, &UnicodeString) < 0)
 return 0;
 if (!FreeUnicodeString(&UnicodeString))
 return 0;
 v7 = FormatSendMessage(0, 0, 59, 0, (int)&unk_509AC, (int)&UnicodeString);
 v8 = v7;
 if (v7)
 {
 v7[9] = CallOldRetValue;
 v7[10] = SendType;
 v7[6] = 0x800400;
 dword_55ADC(unk_55ADC, v7);
 for (i = NewIrl; i != (PERESOURCE)i->SystemResourcesList.Flink)
 {
 if (sub_1E6F4((int)i, 59))
 {
 }
 }
 }
}
```

- 7、
- 8、有意思的是在设置回调的代理函数 Hook\_PsSetCreateProcessNotifyRoutine 中在设置行为下是存在拦截操作的，拦截操作的行为依据来源于 LogAboutInformation 的返回值并且返回 0xC0000022，但是在 LogAboutInformation 的第三个参数为 0 的情况下 LogAboutInformation 直接返回 0，所以也就是说在该版本下拦截其实并不生效

```
signed int __userpurg Hook_PsSetCreateProcessNotifyRoutine@eax(int ebx0@ecx, int pFunAddress, bool bFlag)
{
 int CallAddress; // edi
 int bCallOldRet; // esi
 int savedregs; // [esp+8h] [ebp+0h]

 CallAddress = GetCallAddress((int)&savedregs);
 if (bFlag)
 // True为取消
 // False为设置
 //

 {
 bCallOldRet = CallOldPsSetCreateProcessNotifyRoutine(pFunAddress, bFlag);
 if (bCallOldRet >= 0)
 LogAboutInformation(ebx0, 3, 1, pFunAddress, CallAddress, 1u);
 }
 else
 {
 bCallOldRet = CallOldPsSetCreateProcessNotifyRoutine(pFunAddress, 0);
 if (LogAboutInformation(ebx0, 3, 0, pFunAddress, CallAddress, bCallOldRet >= 0))
 {
 CallOldPsSetCreateProcessNotifyRoutine(pFunAddress, 1);
 return 0xC0000022;
 }
 }
 return bCallOldRet;
}
```

- 9、



#### 四、最后谈谈恢复与绕过思路

- 1、通用思路，首先同样获取系统内核模块的相关函数地址，模拟 MmGetSystemRoutineAddress 的流程，但是我们这里需要解析文件并且解些 EAT 载入内存如果采用读文件的方式的话需要注意内存对齐问题
- 2、拿去到真正的函数地址的时候，如果需要恢复的话可以直接学习刚才那一套 mdl 锁+原子交换的理念，这样就可以解决了
- 3、但是这里其实并不提倡这种方法，因为在一些软件中会对于代码有 crc 校验等功能，如果强行解除 hook 的话很有可能导致 crc 校验失败导致不可预料的结果，所以直接可以将获取到的函数进行指针强转直接调用即可