映像回调可以拦截 RING3 和 RING0 的映像加载。某些游戏保护会用此来拦截黑名单中的驱动加载，比如 XUETR、WIN64AST 的驱动。同理，在反游戏保护的过程中，也可以拦截游戏驱动的加载。

跟进程/线程回调类似，映像回调也存储在数组里。这个数组的“符号名”是 PspLoadImageNotifyRoutine。我们可以在 PsSetLoadImageNotifyRoutine 中找到它：

```
lkd> uf PsSetLoadImageNotifyRoutine
nt!PsSetLoadImageNotifyRoutine:
fffff800`02e9fb60 48895c2408      mov     qword ptr [rsp+8],rbx
fffff800`02e9fb65 57              push    rdi
fffff800`02e9fb66 4883ec20        sub     rsp,20h
fffff800`02e9fb6a 33d2            xor     edx,edx
fffff800`02e9fb6c e8efaffeff      call    nt!ExAllocateCallBack (fffff800`02e8ab60)
fffff800`02e9fb71 488bf8          mov     rdi,rax
fffff800`02e9fb74 4885c0          test    rax,rax
fffff800`02e9fb77  7507                     jne         nt!PsSetLoadImageNotifyRoutine+0x20
(fffff800`02e9fb80)

nt!PsSetLoadImageNotifyRoutine+0x19:
fffff800`02e9fb79 b89a0000c0      mov     eax,0C000009Ah
fffff800`02e9fb7e  eb4a                     jmp         nt!PsSetLoadImageNotifyRoutine+0x6a
(fffff800`02e9fbca)

nt!PsSetLoadImageNotifyRoutine+0x20:
fffff800`02e9fb80 33db            xor     ebx,ebx

nt!PsSetLoadImageNotifyRoutine+0x22:
fffff800`02e9fb82   488d0d7799d9ff      lea             rcx,[nt!PspLoadImageNotifyRoutine
(fffff800`02c39500)]
fffff800`02e9fb89 4533c0          xor     r8d,r8d
fffff800`02e9fb8c 488bd7          mov     rdx,rdi
fffff800`02e9fb8f 488d0cd9        lea     rcx,[rcx+rbx*8]
fffff800`02e9fb93 e8b814f8ff      call    nt!ExCompareExchangeCallBack (fffff800`02e21050)
fffff800`02e9fb98 84c0            test    al,al
fffff800`02e9fb9a  7511                     jne         nt!PsSetLoadImageNotifyRoutine+0x4d
(fffff800`02e9fbad)
```

实现的代码如下：

```
ULONG64 FindPspLoadImageNotifyRoutine()
{
    ULONG64            i=0,pCheckArea=0;
    UNICODE_STRING     unstrFunc;
    RtlInitUnicodeString(&unstrFunc, L"PsSetLoadImageNotifyRoutine");
    pCheckArea = (ULONG64)MmGetSystemRoutineAddress (&unstrFunc);
```

```
    DbgPrint("PsSetLoadImageNotifyRoutine: %llx",pCheckArea);
    for(i=pCheckArea;i<pCheckArea+0xff;i++)
    {
        if(*(PUCHAR)i==0x48 && *(PUCHAR)(i+1)==0x8d && *(PUCHAR)(i+2)==0x0d)  //lea
rcx,xxxx
        {
            LONG OffsetAddr=0;
            memcpy(&OffsetAddr,(PUCHAR)(i+3),4);
            return OffsetAddr+7+i;
        }
    }
    return 0;
}
```

枚举的过程也跟枚举进程和线程回调类似，从数组中读取值之后，需要进行位运算“解密”才能得到地址：

```
void EnumLoadImageNotify()
{
    int i=0;
    BOOLEAN b;
    ULONG64  NotifyAddr=0,MagicPtr=0;
    ULONG64  PspLoadImageNotifyRoutine=FindPspLoadImageNotifyRoutine();
    DbgPrint("PspLoadImageNotifyRoutine: %llx",PspLoadImageNotifyRoutine);
    if(!PspLoadImageNotifyRoutine)
        return;
    for(i=0;i<8;i++)
    {
        MagicPtr=PspLoadImageNotifyRoutine+i*8;
        NotifyAddr=*(PULONG64)(MagicPtr);
        if(MmIsAddressValid((PVOID)NotifyAddr) && NotifyAddr!=0)
        {
            NotifyAddr=*(PULONG64)(NotifyAddr & 0xfffffffffffffff8);
            DbgPrint("[LoadImage]%llx",NotifyAddr);
        }
    }
}
```

删除回调的方法就是调用 PsRemoveLoadImageNotifyRoutine 实现，也可以通过写入机器码（RET）让回调直接返回。最后执行效果如下：

| Time | Debug Print |
|---|---|
| 0.00000000 | PsSetLoadImageNotifyRoutine: fffff80002e9fb60 |
| 0.00000286 | PspLoadImageNotifyRoutine: fffff80002c39500 |
| 0.00000468 | [LoadImage]fffff80002daecc0 |
| 0.00000629 | [LoadImage]fffff880082c7758 |

在干净的 WIN7X64 系统，可能没有 LoadImage 回调，为了体现枚举效果，可以在测试驱动前运行一下 WIN64AST。

但我想说明的是，用这三种回调（CreateProcess、CreateThread、LoadImage）来做监控其实并不怎么靠谱，因为系统里存在一个开关，叫做 PspNotifyEnableMask，如果它的值被设置为 0，那么所有的相关操作都不会经过回调。换句话说，如果 PspNotifyEnableMask 等于 0，那么所有的进程、线程、映像回调都会失效。不过这个变量并没有在导出函数中直接出现，所以找到它略难。