x86分页内存布局与CR3骚操作读写内存

原创 Frank MARS 已于 2024-07-04 00:44:09 修改 阅读量980 版权

文章标签： 硬件 c++ 驱动开发 windows

我们知道游戏外G的原理是通过读取游戏进程的内存，取出游戏规则中不允许展示给玩家的关键数据，达到玩家本人和对手信息差的目的。虽然，普通的用户态API是可以读其他进程的内存，比如ReadProcessMemory函数，但分分钟就能被anticheat检测到，本篇文章将带领各位实现内核态下通过切换到目标进程的CR3寄存器达到无痕读写内存的目的。

分页

首先需要了解一下内存的分页机制，至于分段机制，就不提了，系统都废掉不用了，何必多此一举。

硬盘一个扇区是512KB，也就是说只要读取，至少拿512KB的倍数，就算拿1KB，从磁盘取出来也是512KB，剩下的511KB就是浪费，但也不得不拿

目前物理内存的一个页普遍设计的是4KB，但这不是定死的。

分页功能操作相关寄存器和位

CPU上有五个寄存器，CR0~CR4，功能是控制用的，想要操作分页功能，找它们准没错。

位名称	在哪个寄存器的哪个位	含义
PG	CR0 (31)	分页功能的打开和关闭，0关闭1开启
PSE	CR4 (4)	页大小扩展，不想用4KB的页大小，想用更大的如2M，就得把这玩意打开
PAE	CR4 (5)	物理地址扩展，支持36位物理内存地址，即支持64G的物理内存

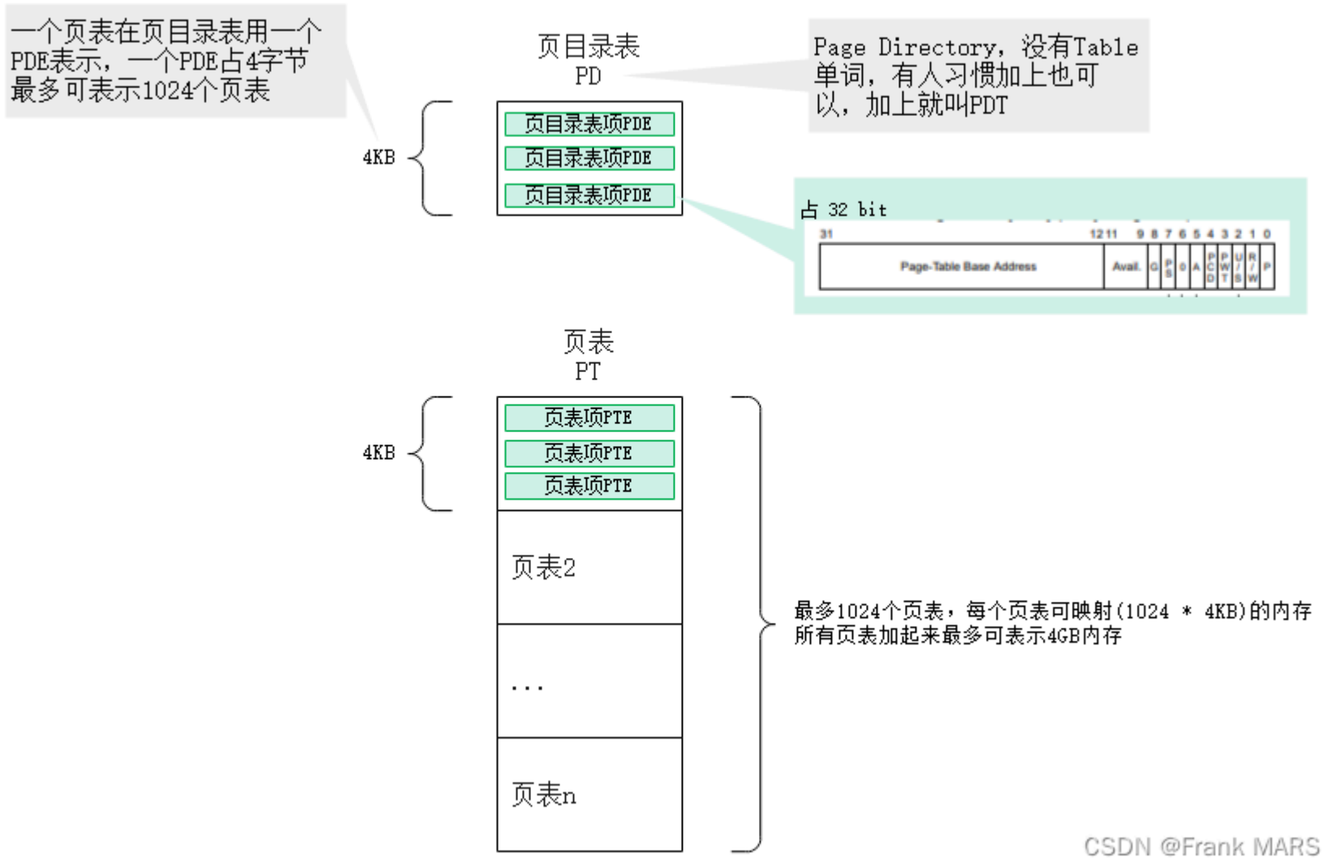
不同位的效果如下表

PG	PAE	PSE	PS	页大小	物理地址大小
0	N/A	N/A	N/A	N/A	分页功能关闭
1	0	0	N/A	4 KB	32 位
1	0	1	0	4 KB	32 位
1	0	1	1	4 MB	32 位
1	1	N/A	0	4 KB	36 位
1	1	N/A	1	2 MB	36 位

由表可知一个页的大小是固定的，不能说这个页4KB，到下个页就变成2MB了，这是不可能的，因此，在设计页表的时候，就不用再记录页大小了，大家都一样

神奇的两张表

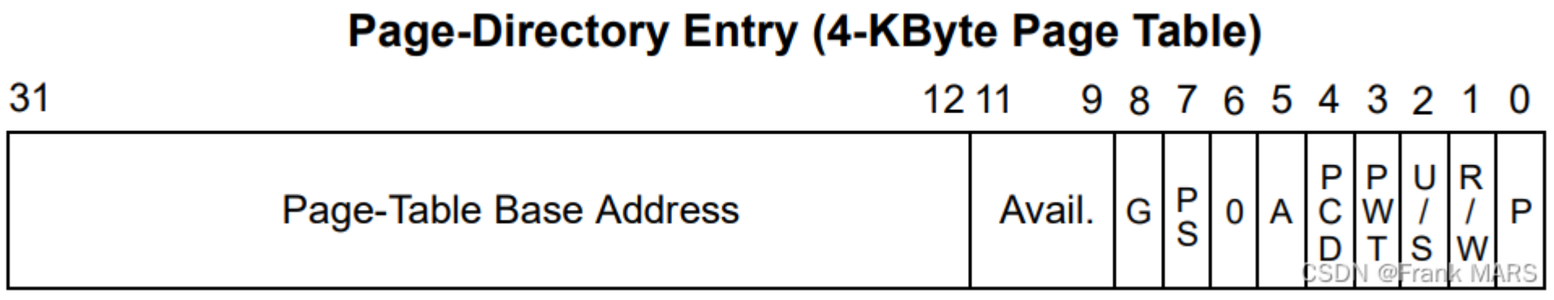
每个进程创建的时候都有这么一套表（一个页目录表 + n个页表），注意本篇文章不涉及开启PAE(Physical Address Extension)的情况，下同



其中，CR3寄存器永远指的是页目录表的地址，OS根据这就能找到啦，切进程的时候，把待切入进程的页目录表的地址送入CR3即可，这也是我们实现功能的关键，也就是说每个进程都有属于自己的CR3值，系统切进程的时候，就把当前进程环境的页目录表地址读取出来，写入CR3寄存器，这样的话，假如每个进程的虚拟地址都是123456，也不会打架，因为进程1有进程1的一套表，系统一查进程1的表，发现进程1的123456对应到物理内存条的地址是001。进程2有进程2的一套表，系统一查进程2的表，发现进程2的123456对应到物理内存条的地址是002。

PDE和PTE

页目录表和页表里面都放的是一项项的PDE或PTE，它们的结构如下



- P-TBA：页表的首地址
- Avail（3位）：intel不玩了，给OS玩的位，自己用
- G（Global）：intel刚开始也想和分段一样，搞个全局和局部的，有这想法，后来发现没意义了，就不管了，这位就废了
- PS：为0是4K的页，为1是4M的页
- 0：保留
- A：是否被访问过，方便交换功能用
- PCD：是否放缓存中，我们程序员不关心
- PWT：程序员不关心，不管
- U/S：特权级，为0表示Supervisor（Ring0），为1表示User（Ring3）
- R/W：内存权限，这里就给了一位，显然是不够表示那么多权限属性的，因此也是Intel设计的败笔，为此还导致了许多漏洞产生。我们来看看为什么是败笔：
  - 0：可读可执行 RE
  - 1：可读可写可执行 RWE

  发现没有，无论是0还是1，都有可执行的权限，因此可以随便执行数据区的代码，导致了缓冲区漏洞。

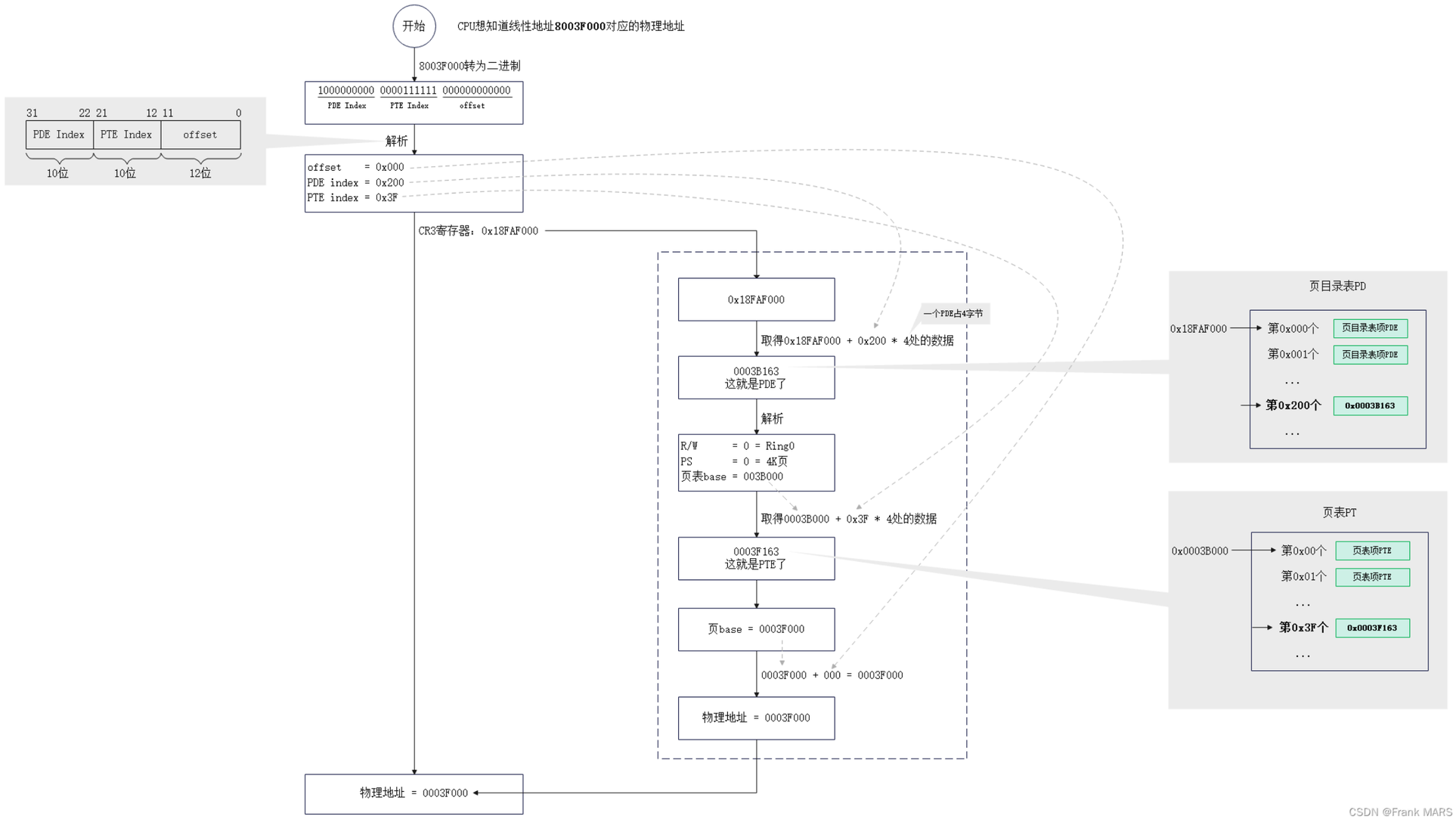
  高版本弥补了此漏洞，即Windows的DEP保护，它可以让一个页不能够执行代码，但DEP功能仅限64位系统使用

  回想一下，如果当时Intel能给两个bit到R/W，也就没这烂摊子了（笑）
- P：存在位。0无效，1有效，也是`MmIsAddressValid()`函数的原理，这函数查的就是这个bit

上面是PDE，其实PTE和PDE大差不差，就俩区别：

1. 第七位（PS位）改成了0（无效位）
2. 首地址是页的首地址

接下来，以一张图为例子，讲讲代码里面的8003F000是怎么通过一系列骚操作，变换成物理地址的0003F000的，请见下图



CR3读写

读

由图可知，CR3寄存器可以作为一个点位，我们只要在自己的进程环境下，将自己的CR3改成目标进程的CR3，改完后，操作的地址全在对方的内存下了，美滋滋。也就是说，**假如我是进程2，我把进程1的表先抢过来，当系统要查表时，我就把进程1的表瞒天过海塞给系统，这样，我自己代码的读写内存操作，实际上系统一查表，跑人家进程1对应的物理内存去执行了。**这样，你检测得到我吗，我在我自己进程里面读写内存，关你吊事，系统把表弄错了你找系统，跟我有毛关系。

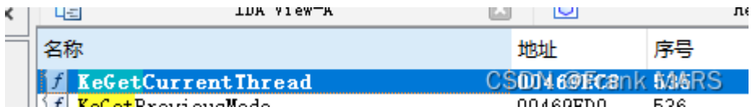
那么问题来了，对方进程的页目录表地址，也就是CR3保存在哪里呢？？

在微软的Windows下，通过 KPCR -> ETHREAD -> EPROCESS -> DirBase拿到页目录表地址，缺点是不同OS版本的偏移不一样，在我机子上确实没问题，到你机子上就崩了，因为这些结构体微软是没公开的，在MSDN上你是查不到的，因此可能变来变去的。暴力解法：每个OS版本都做一套

猥琐解法：读取系统API的二进制指令机器码，从而读取到当前系统的偏移值，这样就万能了。

笔者的测试环境是Windows XP SP3的32位系统，虽然微软未公开上述的结构体，但我们可以通过逆向分析系统内核代码来确定这些结构体的地址偏移。

比如ETHREAD结构，微软并未公开该结构体的文档，去官方文档是查不到的，我们可以通过逆向分析system32目录下的系统内核文件ntkrnlpa.exe来取得，在windows中有一个函数叫KeGetCurrentThread函数，可以用IDA查看到ntkrnlpa.exe确实有导出该函数



双击进入该函数体后，代码如下

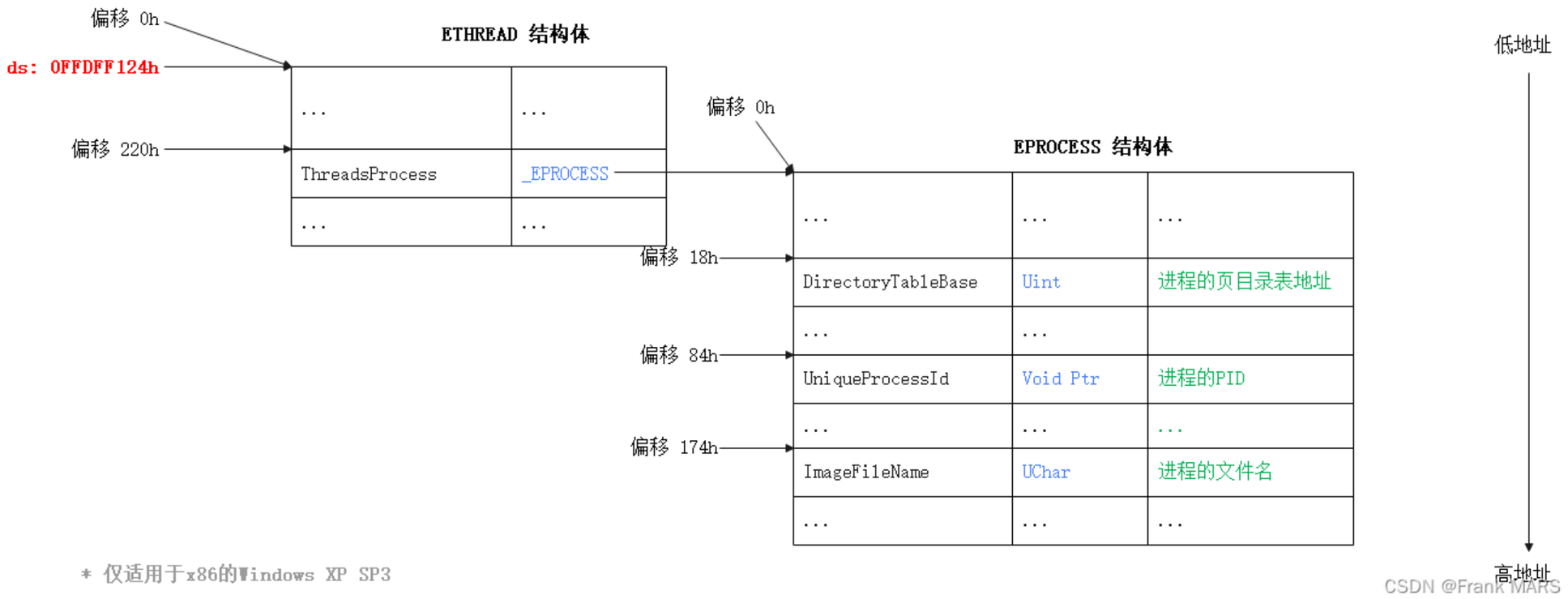
```
; PKTHREAD __stdcall KeGetCurrentThread()
public KeGetCurrentThread
KeGetCurrentThread proc near            ;
mov     eax, ds:0FFDFF124h
retn
```


很简单不是吗，看来ETHREAD结构体就存放在ds:0FFDFF124h处，请注意由于这里是未公开偏移，因此不同系统是不一样的，如在Windows 7的32位系统下，该值就变成了fs:124h处

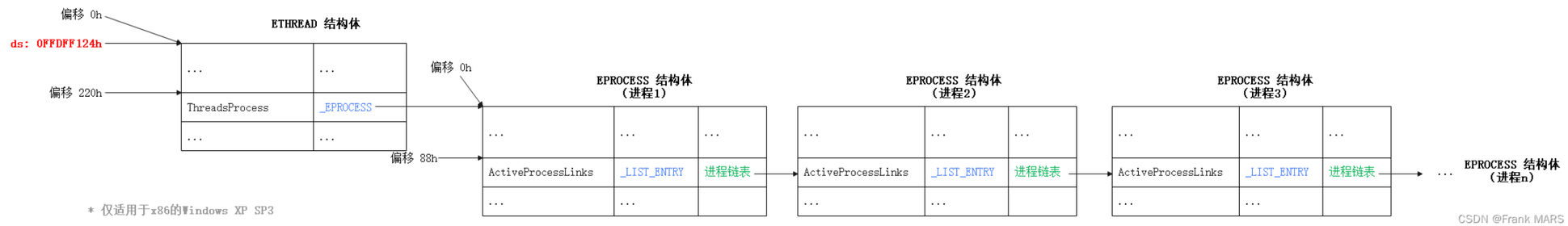
拿到结构体后，还需要进一步拿到页目录表地址，我们可以用windbg的dt _ethread命令查看当前系统下的ETHREAD结构，如下图

```
+0x208 LpcWaitingOnPort : Ptr32 Void
+0x20c ImpersonationInfo : Ptr32 _PS_IMPERSONATION_I
+0x210 IrpList          : _LIST_ENTRY
+0x218 TopLevelIrp      : Uint4B
+0x21c DeviceToVerify   : Ptr32 _DEVICE_OBJECT
+0x220 ThreadsProcess   : Ptr32 _EPROCESS
+0x224 StartAddress     : Ptr32 Void
+0x228 Win32StartAddress : Ptr32 Void
+0x228 LpcReceivedMessageId : Uint4B
+0x22c ThreadListEntry  : _LIST_ENTRY
```


可以发现在该结构体偏移的220h处，存放着EPROCESS结构体，然后我们再用dt _eprocess命令查看EPROCESS结构体的具体内容，其结构如下图



这样，我们就成功知道了我们心心念念的页目录表地址在哪里咯，接下来我们只要找到目标进程就可以了，那么问题又来了，我怎么样拿到对方进程的这些乱七八糟的结构体呢，想象系统就像监狱，每个进程就是监狱的一个牢房，我人在自己的牢房里面，想去别人的牢房里面整活，可以通过挖墙壁的方法，先挖一道墙，看隔壁是不是我想要的进程，发现隔壁牢房不是，就再挖隔壁的墙，去隔壁的隔壁……，如此往复，肯定就挖到目标进程的牢房了，好在操作系统真的也是这样设计“牢房”的，如下图所示



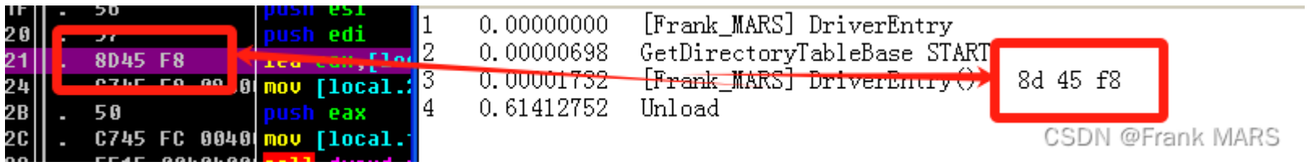
我们只要不断遍历这个链表，判断当前链表的元素是不是我们的目标进程即可。

```
PVOID GetDirectoryTableBase(ULONG pid) {
    DbgPrint("%s START\n", __FUNCTION__);
    PEPROCESS process = NULL;
    //汇编拿到当前进程的EPROCESS
    __asm {
        mov eax, ds: [0FFDFF124h]    //ETHREAD
        mov eax, [eax + 220h]  //EPROCESS
        mov process, eax
    }
```

```
    /*
     *   Windows 7
        __asm {
            mov eax,     fs: [124h]    //ETHREAD
            mov eax,     [eax + 50h]  //EPROCESS
            mov process, eax
        }*/

    PEPROCESS begin = process;
    __try {
        do
        {
            if (!MmIsAddressValid(process)) break;


            // Windows XP SP3
            ULONG uniqueProcessId = *(ULONG*)((char*)process + 0x084);
            PVOID directoryTableBase = *(PVOID*)((char*)process + 0x018);

            if (uniqueProcessId == pid) return directoryTableBase;

            process = (PEPROCESS)((CHAR*)*(PVOID*)((CHAR*)process + 0x088) - 0x088);

        } while (process != begin);

    }
    __except (EXCEPTION_EXECUTE_HANDLER) {
        DbgPrint("[Frank_MARS]  %s __except\n", __FUNCTION__);
    }

    return NULL;
}

bool ReadProcessMemoryByCR3(
    ULONG pid,            //目标进程的PID
    PVOID lpBaseAddress,  //欲读取的虚拟地址
    PVOID  lpBuffer,      //存放的缓冲区
    SIZE_T  nSize         //大小
    )
{
    //在内核中，HANDLE就是pid
    PVOID pDirBase = GetDirectoryTableBase(pid);
    PVOID pOldDirBase = NULL;
    bool  bRet = false;


    if (pDirBase == NULL) return false;
```

```
57  58      __asm {
59          mov eax, cr3
60          mov pOldDirBase, eax
61          mov eax, pDirBase
62          mov cr3, eax
63
64      }
65
66
67      //若缺页，则映射会失败，就有处理的余地了
68      PHYSICAL_ADDRESS pa = MmGetPhysicalAddress(lpBaseAddress);
69      if (pa.QuadPart != NULL)
70      {
71          PVOID pMapAddr = MmMapIoSpace(pa, nSize, MmNonCached);
72          if (pMapAddr != NULL)
73          {
74              RtlCopyMemory(lpBuffer, lpBaseAddress, nSize);
75              MmUnmapIoSpace(pMapAddr, nSize);
76              bRet = true;
77          }
78      }
79
80      //读取结束，把CR3给人家改回去
81      __asm {
82          mov eax, pOldDirBase
83          mov cr3, eax
84
85      }
86      return bRet;
87  }
88
89  NTSTATUS DriverEntry(IN PDRIVER_OBJECT DriverObject, IN PUNICODE_STRING RegisteryPathName)
90  {
91      UNREFERENCED_PARAMETER(RegisteryPathName);
92      UNREFERENCED_PARAMETER(DriverObject);
93      DbgPrint("[Frank_MARS] %s() START\n", __FUNCTION__);
94      DriverObject->DriverUnload = Unload;  //Unload函数实现略
95
96      UCHAR buf[3] = { 0 };
97      ReadProcessMemoryByCR3(148, (PVOID)0x00401F21, buf, sizeof buf);
98
99      DbgPrint("[Frank_MARS] %s() : %02x %02x %02x\n", __FUNCTION__, buf[0], buf[1], buf[2] );
100
101     return STATUS_SUCCESS;
102 }
```

上述代码中，读取了一个进程，这个进程的PID是148，欲读取数据的地址是0x00401F21，读取的长度是是三个字节，加载驱动后输出，读取成功了



写

读完了，接下来就是写了，就不展开讲了，很简单，同时写的代码也升级了一下，升级内容如下

- 取消__asm内联汇编，全部使用微软汇编函数，x64也能用啦
- 根据32/64位自动选择变量类型
- 采用KeStackAttachProcess函数代替了切换进程环境
- 原先的偏移现在均由内核API函数完成，兼容性好，理论上从Windows XP到Windows 11的系统不管32还是64位均可使用

代码如下

```
NTSTATUS WriteProcessMemory(
    ULONG pid,            //目标进程的PID
    PVOID lpBaseAddress,  //欲写入的虚拟地址
    PVOID  lpBuffer,      //欲写入数据存放的缓冲区
    SIZE_T  nSize         //欲写入数据的大小
)
{

    NTSTATUS  bRet = STATUS_UNSUCCESSFUL;

    PEPROCESS process;
    NTSTATUS status = PsLookupProcessByProcessId((HANDLE)pid, &process);
    if (!NT_SUCCESS(status)) return status;

    //切换到对方进程，切换后，后续所有的汇编代码全在对方的环境下执行
    KAPC_STATE apcState;
    KeStackAttachProcess(process, &apcState);
    PHYSICAL_ADDRESS pa = MmGetPhysicalAddress(lpBaseAddress);
    KIRQL  oldIrql;
    KeRaiseIrql(DISPATCH_LEVEL, &oldIrql);

#ifdef _M_X64
    __int64 oldCr0 = __readcr0();
#else
    unsigned long oldCr0 = __readcr0();
```

```
#endif
    __writecr0(oldCr0 & ~0x10000);


        //若缺页，则映射会失败，就有处理的余地了

    if (pa.QuadPart != NULL)
    {
        PVOID pMapAddr = MmMapIoSpace(pa, nSize, MmNonCached);
        if (pMapAddr != NULL)
        {
            RtlCopyMemory(pMapAddr, lpBuffer, nSize);
            MmUnmapIoSpace(pMapAddr, nSize);
            bRet = STATUS_SUCCESS;
        }


    }


    //写入结束，CR0给人家改回去

    __writecr0(oldCr0);


    KeLowerIrql(oldIrql);

    KeUnstackDetachProcess(&apcState);
    ObDereferenceObject(process);

    return bRet;
}
```

请注意，写的时候除了改CR3（已经用KeStackAttachProcess函数实现相同功能）外，还需要改CR0，因为CR0寄存器中有一位，叫WP位，标志了权限，将这一个bit设为0的话，直接无视内存的权限属性，随意、暴力、强制、无脑写入，爽不爽。

反正我是爽了。