漫谈兼容内核之十四：
Windows 的跨进程操作

毛德操

Jeffrey Richter 在他的“Advanced Windows”一书第 18 章“打破进程壁垒(Breaking Through Process Boundary Walls)”中讲述了一个有趣的实验，就是利用 OpenProcess()、CreateRemoteThread()、VirtualAllocEx()、WriteProcessMemory()等等 Win32 API 函数从一个进程向另一个进程的用户空间“注入(Inject)”一个 DLL。其过程大致如下：

- 给定目标进程的进程号 PID，通过 OpenProcess()“打开”这个进程，得到代表着目标进程“对象”的 Handle。OpenProcess()的基础是系统调用 NtOpenProcess()。
- 通过 VirtualAllocEx()在目标进程的用户空间分配一块内存。VirtualAllocEx()的基础是系统调用 NtAllocateVirtualMemory()。
- 通过 WriteProcessMemory()把一些代码和数据拷贝到目标进程用户空间中刚分配的那块内存中，这些代码的入口为 ThreadFunc()。WriteProcessMemory()的基础是系统调用 NtWriteVirtualMemory()。
- 通过 CreateRemoteThread()在目标进程中创建一个以 ThreadFunc()为入口的线程。CreateRemoteThread()的基础是系统调用 NtCreateThread()。
- 当目标进程中的线程 ThreadFunc()受调度运行时，就把预定的 DLL 装入目标进程的用户空间。
- 然后线程 ThreadFunc()退出运行而不复存在，但是所装入(并连接)的 DLL 却留在了目标进程的用户空间。

当然，这是个很有趣的实验，利用这个实验所揭示的特点也许可以开发出某些很好的应用。但是问题也随之而生：要是 ThreadFunc()是一段木马程序呢？比方说，要是这里的目标进程是网络浏览器，而 ThreadFunc()每当受调度运行时就把本地的某些信息发送给某个网站，然后睡眠一段时间，如此周而复始呢？显然，只要那个被 ThreadFunc()“附体”的网络浏览器进程还在运行，这段木马程序就可周期性地得到执行，而很难被察觉。

笔者在以前的漫谈中曾经讲过，Windows 与 Linux 的一个很明显、很重要的区别就是：在 Windows 中一个进程可以越俎代庖地替别的进程做好多事，其中就包括上面讲到的几项跨进程操作。我们在创建 Windows 进程、启动 PE 映像执行的过程中也看到过一些跨进程的操作，例如把可执行映像映射到子进程的用户空间、在子进程的映像中寻找函数入口、为子进程创建线程等等。除直接的跨进程操作外，还可以跨进程复制已打开对象的 Handle。而 Linux，则是不允许、或者说不提供此类跨进程操作的。而且，正是这方面的差异使得 Wine 的“核内差异核外补”策略难以有效实施。

相比之下，Linux 进程是“独立自主”的。当然，Linux 也有进程间通信，但那只是通信而已。在进程间通信的基础上，一个进程也可以应另一个进程的请求而在其自身的上下文中执行某些操作。但是那些操作都是预定的、预先就安排在这个进程的代码中的，所反映的是程序设计者的意志。从这个意义上说，除非程序中有错误(bug)，Linux 进程的行为是可预测的。而 Windows 进程则有可能发生不可预测的行为，因为别的进程居然可以把一段程序“注入”其空间并使之成为一个线程而得到执行。

可想而知，要是允许这样的跨进程操作不受限制地进行，对于系统的安全性是影响极大的，所以必定要有安全措施配套才行。

对于这么重要的问题，我们当然希望能了解 Windows 的跨进程操作和相应的安全措施

是怎么实现的。但是，遗憾的是：一方面是微软不向公众公开 Windows 内核的代码，另一方面是 ReactOS 尚未实现有关的安全措施。这样，我们现在能做的就只能是先通过 ReactOS 的代码了解跨进程操作的实现，而看不到有关安全措施的实现。所以下面我们只能在代码中看到“矛”的一面，而看不到“盾”的一面。

下面的代码仍取自 ReactOS 的 0.2.6 版本，不过这个版本的 ReactOS 尚未实现配套的安全措施，所以只能借此了解一下有关跨进程操作的实现。

先看 NtOpenProcess()的实现，因为所有的跨进程操作都是从这里开始的。

```
NTSTATUS STDCALL
NtOpenProcess(OUT PHANDLE              ProcessHandle,
              IN  ACCESS_MASK          DesiredAccess,
              IN  POBJECT_ATTRIBUTES   ObjectAttributes,
              IN  PCLIENT_ID           ClientId)
{
  . . . . . .

  if (ObjectAttributes != NULL && ObjectAttributes->ObjectName != NULL &&
      ObjectAttributes->ObjectName->Buffer != NULL)
  {
    NTSTATUS Status;
    PEPROCESS Process;

    Status = ObReferenceObjectByName(ObjectAttributes->ObjectName,
                     ObjectAttributes->Attributes, NULL, DesiredAccess,
                     PsProcessType, UserMode, NULL, (PVOID*)&Process);
    . . . . . .
    Status = ObCreateHandle(PsGetCurrentProcess(),Process, DesiredAccess,
                                                    FALSE, ProcessHandle);
    ObDereferenceObject(Process);
    return(Status);
  }
  else
  {
    PLIST_ENTRY current_entry;
    PEPROCESS current;
    NTSTATUS Status;

    ExAcquireFastMutex(&PspActiveProcessMutex);
    current_entry = PsActiveProcessHead.Flink;
    while (current_entry != &PsActiveProcessHead)
    {
      current = CONTAINING_RECORD(current_entry, EPROCESS,
                      ProcessListEntry);
      if (current->UniqueProcessId == ClientId->UniqueProcess)
```

```
            {
               if (current->Pcb.State == PROCESS_STATE_TERMINATED)
               {
                  Status = STATUS_PROCESS_IS_TERMINATING;
               }
             else
               {
                  Status = ObReferenceObjectByPointer(current,
                                                      DesiredAccess,
                                                      PsProcessType,
                                                      UserMode);
               }
             ExReleaseFastMutex(&PspActiveProcessMutex);
             if (NT_SUCCESS(Status))
               {
                  Status = ObCreateHandle(PsGetCurrentProcess(),
                                 current,
                                 DesiredAccess,
                                 FALSE,
                                 ProcessHandle);
                  ObDereferenceObject(current);
                  . . . . . .
               }
             return(Status);
              }
            current_entry = current_entry->Flink;
      }
      ExReleaseFastMutex(&PspActiveProcessMutex);
      DPRINT("NtOpenProcess() = STATUS_UNSUCCESSFUL\n");
      return(STATUS_UNSUCCESSFUL);
   }
   return(STATUS_UNSUCCESSFUL);
}
```

像别的对象一样，进程对象也可以有个对象名(注意进程的对象名与所执行的映像文件名是两码事)。同时，进程又有进程号。要打开一个进程对象时，既可以按对象名打开，也可以按进程号打开。如果是对象名打开，就把对象名填写在作为参数的 OBJECT_ATTRIBUTES 数据结构中，就是这里的参数 ObjectAttributes。如果是按进程号打开，则把进程号填写在也是作为参数的“客户标识”CLIENT_ID 数据结构中，就是这里的参数 ClientId。严格地说 CLIENT_ID 数据结构是进程 Handle 和线程 Handle 的组合，用来唯一地标识一个线程。之所以叫“客户”，可能是对服务进程 csrss 而言。但是 Handle 在本质上是数组下标，所以进程 Handle 其实也就是进程号。至于线程 Handle，则此刻不在关心之列，所以设置成 0 就可以了。注意 CLIENT_ID 中的进程 Handle 不同于打开一个进程以后所得到的 Handle，前者是全局的，内核中单独有个 Cid 对象表 PspCidTable；而后者是局部的，

作用于当前进程的打开对象表中。

看一下 ReactOS 的 Win32 API 函数 OpenProcess()的代码，就可以明白应该怎样使用 CLIENT_ID 于 NtOpenProcess()：

```
HANDLE STDCALL
OpenProcess(DWORD dwDesiredAccess, BOOL bInheritHandle, DWORD dwProcessId)
{
    . . . . . .
    ClientId.UniqueProcess = (HANDLE)dwProcessId;
    ClientId.UniqueThread = 0;
    . . . . . .
    errCode = NtOpenProcess(&ProcessHandle,…, &ClientId);
    . . . . . .
    return ProcessHandle;
}
```

可见，这里只是把进程号 dwProcessId 的类型转换成了 HANDLE，而数值并未改变。

NtOpenProcess()的另一个参数 DesiredAccess 则是以标志位的形式说明打开这进程对象的目的，例如：

```
#define PROCESS_TERMINATE               1
#define PROCESS_CREATE_THREAD           2
#define PROCESS_SET_SESSIONID           4
#define PROCESS_VM_OPERATION            8
#define PROCESS_VM_READ                 16
#define PROCESS_VM_WRITE                32
#define PROCESS_DUP_HANDLE              64
#define PROCESS_CREATE_PROCESS          128
#define PROCESS_SET_QUOTA               256
#define PROCESS_SET_INFORMATION         512
#define PROCESS_QUERY_INFORMATION       1024
```

这些标志位的作用与打开文件时所使用的可读、可写等等相似，一方面是为以后对这个已打开映像的访问设置一个范围，一方面是让内核可以针对所要求的操作实施权限检查。

如果打开成功，NtOpenProcess()就通过参数 ProcessHandle 返回已打开对象的 Handle。

所以，如果是按对象名打开，代码中就通过 ObReferenceObjectByName()在内核中比对寻找同名的对象，找到后就返回目标进程的进程控制块指针，然后在当前进程的打开对象表中加入一个表项，并返回其 Handle。按理说在这个过程中应该检查当前进程的权限，但在 ReactOS 的 0.2.6 版中尚未实现。

而如果是按进程号打开，那就要扫描(while 循环)当前的进程队列，找出进程号与 ClientId->UniqueProcess 相符的进程控制块，然后通过 ObReferenceObjectByPointer()找到相应的对象，再往下就都一样了。

打开进程是这样，打开线程也是差不多。

打开了目标进程以后，就可以对其实施跨进程操作了。允许跨进程操作的 Windows 系

统调用有很多，这些系统调用一般都以 ProcessHandle 为参数。跨线程的操作也可以间接地认为是跨进程操作，因为目标线程可以是别的进程中的线程。这样，直接或间接意义上的跨进程操作就数量不小了：

NtAllocateVirtualMemory()
NtFreeVirtualMemory()
NtQueryVirtualMemory()
NtLockVirtualMemory()
NtUnlockVirtualMemory()
NtReadVirtualMemory()
NtWriteVirtualMemory()
NtProtectVirtualMemory()
NtFlushVirtualMemory()
NtAllocateUserPhysicalPages()
NtFreeUserPhysicalPages()
NtMapUserPhysicalPages()
NtMapUserPhysicalPagesScatter()
NtGetWriteWatch()
NtResetWriteWatch()
NtMapViewOfSection()
NtUnmapViewOfSection()
NtCreateThread()
NtOpenThread()
NtTerminateThread()
NtQueryInformationThread()
NtSetInformationThread()
NtResumeThread()
NtGetContextThread()
NtSetContextThread()
NtQueueAPCThread()
NtAlertThread()
NtAlertResumeThread()
NtRegisterThreadTerminatePort()
NtImpersonateThread()
NtImpersonateAnonymousThread()
NtTerminateProcess()
NtQueryInformationProcess()
NtSetInformationProcess()
NtAssignProcessToJobObject()
NtOpenProcessToken()
NtOpenThreadToken()
NtCreateProfile()
NtDuplicateObject()

我们当然不可能在这里逐一考察所有这些系统调用，而只是顺着前面所说 Jeffrey Richter 的实验考察几个关键的系统调用。首先是在别的进程的用户空间分配一个内存区间，这是由 NtAllocateVirtualMemory()实现的。

```
NTSTATUS STDCALL
NtAllocateVirtualMemory(IN HANDLE ProcessHandle,
                        IN OUT PVOID* UBaseAddress,
                        IN ULONG ZeroBits,
                        IN OUT PULONG URegionSize,
                        IN ULONG AllocationType,
                        IN ULONG Protect)
{
    PEPROCESS Process;
    MEMORY_AREA* MemoryArea;
    . . . . . .

    /*
     * Check the validity of the parameters
     */
    if ((Protect & PAGE_FLAGS_VALID_FROM_USER_MODE) != Protect)
    {
        return(STATUS_INVALID_PAGE_PROTECTION);
    }
    if ((AllocationType & (MEM_COMMIT | MEM_RESERVE)) == 0)
    {
        return(STATUS_INVALID_PARAMETER);
    }

    PBaseAddress = *UBaseAddress;
    PRegionSize = *URegionSize;
    BoundaryAddressMultiple.QuadPart = 0;

    BaseAddress = (PVOID)PAGE_ROUND_DOWN(PBaseAddress);
    RegionSize = PAGE_ROUND_UP(PBaseAddress + PRegionSize) -
                    PAGE_ROUND_DOWN(PBaseAddress);

    Status = ObReferenceObjectByHandle(ProcessHandle,
                                       PROCESS_VM_OPERATION,
                                       NULL,
                                       UserMode,
                                       (PVOID*)(&Process),
                                       NULL);
    . . . . . .
```

```
Type = (AllocationType & MEM_COMMIT) ? MEM_COMMIT : MEM_RESERVE;
DPRINT("Type %x\n", Type);

AddressSpace = &Process->AddressSpace;
MmLockAddressSpace(AddressSpace);

if (PBaseAddress != 0)
{
    MemoryArea = MmLocateMemoryAreaByAddress(AddressSpace, BaseAddress);

    if (MemoryArea != NULL)
    {
        MemoryAreaLength = (ULONG_PTR)MemoryArea->EndingAddress -
                           (ULONG_PTR)MemoryArea->StartingAddress;
        if (MemoryArea->Type == MEMORY_AREA_VIRTUAL_MEMORY &&
            MemoryAreaLength >= RegionSize)
        {
            Status =
                MmAlterRegion(AddressSpace,
                              MemoryArea->StartingAddress,
                              &MemoryArea->Data.VirtualMemoryData.RegionListHead,
                              BaseAddress, RegionSize,
                              Type, Protect, MmModifyAttributes);
            MmUnlockAddressSpace(AddressSpace);
            ObDereferenceObject(Process);
            DPRINT("NtAllocateVirtualMemory() = %x\n",Status);
            return(Status);
        }
        else if (MemoryAreaLength >= RegionSize)
        {
            Status =
                MmAlterRegion(AddressSpace,
                              MemoryArea->StartingAddress,
                              &MemoryArea->Data.SectionData.RegionListHead,
                              BaseAddress, RegionSize,
                              Type, Protect, MmModifyAttributes);
            MmUnlockAddressSpace(AddressSpace);
            ObDereferenceObject(Process);
            DPRINT("NtAllocateVirtualMemory() = %x\n",Status);
            return(Status);
        }
        else
        {
            MmUnlockAddressSpace(AddressSpace);
```

```
                ObDereferenceObject(Process);
                return(STATUS_UNSUCCESSFUL);
            }
        }
    }

    Status = MmCreateMemoryArea(Process, AddressSpace,
                                MEMORY_AREA_VIRTUAL_MEMORY,
                                &BaseAddress, RegionSize, Protect,
                                &MemoryArea,
                                PBaseAddress != 0,
                                (AllocationType & MEM_TOP_DOWN) == MEM_TOP_DOWN,
                                BoundaryAddressMultiple);
    . . . . . .

    MemoryAreaLength = (ULONG_PTR)MemoryArea->EndingAddress -
                                    (ULONG_PTR)MemoryArea->StartingAddress;

    MmInitialiseRegion(&MemoryArea->Data.VirtualMemoryData.RegionListHead,
                               MemoryAreaLength, Type, Protect);

    if ((AllocationType & MEM_COMMIT) &&
            ((Protect & PAGE_READWRITE) || (Protect & PAGE_EXECUTE_READWRITE)))
    {
        MmReserveSwapPages(MemoryAreaLength);
    }

    *UBaseAddress = BaseAddress;
    *URegionSize = MemoryAreaLength;
    DPRINT("*UBaseAddress %x    *URegionSize %x\n", BaseAddress, RegionSize);

    MmUnlockAddressSpace(AddressSpace);
    ObDereferenceObject(Process);
    return(STATUS_SUCCESS);
}
```

先要着重说一下参数 AllocationType。这是一些标志位，主要有 MEM_RESERVE、MEM_COMMIT、MEM_RESERVE、MEM_RESET、以及 MEM_TOP_DOWN。调用者通过这些标志位说明调用 NtAllocateVirtualMemory()的意图。Windows 内核把虚存空间的分配与映射区分了开来：

标志位 MEM_RESERVE 表示要求“预订”、即分配一个虚拟地址区间。正如前一篇漫谈中所述，虚拟地址区间的分配只是“账面”上的操作，而并不涉及页面映射表的改变，所以并没有建立起有关页面的映射。要建立页面映射，就得为有关的虚存页面提供物理的存储、或者说后备。就 Windows 而言，这种物理的存储有两种形式。一种是物理的内存页面，另

一种是磁盘上的 Swap 文件。这样，一旦为一个虚存页面建立了映射，这个页面就要么体现为内存中的某个物理页面，要么体现为 Swap 文件中的某个页面(也是物理页面)，这两种形态之间的转换就是页面的换入/换出。从某种意义上说，映射的建立类似于所预订资源的兑现，为此就得投入相应的资源(Swap 文件页面或内存页面)作为代价，类似于“现金交割”，这就是标志位 MEM_COMMIT 所表示的意思。所以，虚存区间的分配实际上分成预订和交割两项操作，这两项操作既可以分两步走，也可以一步到位。如果是分两步走，就要先后调用 NtAllocateVirtualMemory()两次，第一次把 MEM_RESERVE 设成 1，第二次把 MEM_COMMIT 设成 1。也就是说：先预订，再交割。而若要一步到位，只调用 NtAllocateVirtualMemory()一次，那就把这两个标志位都设成 1。如果我们探讨这套方案的设计者的初衷，那么显然是要人们分两步走、甚至分多步走，目的是要减小 Swap 文件的大小。假定我们要分配一个 512MB 的虚存区间，如果要立即就建立映射，那么就要在 Swap 文件中提供 512MB 的空间，相当于一订货就把全部货款都付清了。但是，实际上往往并非所有这 512MB 的存储空间都是立即就要使用的，所以更好的办法是先预定，然后要用多少就交割多少，不用了就退掉，这样就可以少占 Swap 文件的空间、从而可以减小 Swap 文件的大小。Jeffrey Richter 的书中对此有比较详细的叙述。

但是，这当然不是唯一的方法，例如 Linux 就不采用这样的方法。在 Linux 中根本就不分甚么预订和交割，分配内存区间就是分配内存区间，也并不是在分配内存区间的时候就在 Swap 盘区上分配页面作为类似于“保证金”那样的后备，而是在真正需要的时候才动态分配 Swap 页面。这一方面可能是因为 Linux 基本上都是用一个磁盘或盘区作为 Swap 空间，不像 Windows 那样采用 Swap 文件而有文件大小的压力，另一方面结构上也比较简洁。不过这两种方法应该说是各有千秋，而并无绝对的好坏或高下。按说 ReactOS 在各个方面都在尽力模仿 Windows，但是在这方面却实际上采用了类似于 Linux 的方法，这一点下面就可以看到。

另一方面，在前一篇漫谈中我们看到的是映像文件的映射，而映像文件本身就起着相当于 Swap 文件的作用，而给定映像文件的大小本来就是固定的，所以不存在要设法减小其文件大小的问题。

明白了这些，下面就可以看 NtAllocateVirtualMemory()的代码了。

首先，程序中局部量 Type 的值来自 AllocationType，不是 MEM_COMMIT 就是 MEM_RESERVE，二者必居其一。不过 MEM_COMMIT 也可能蕴含了 MEM_RESERVE，因为两步可以并为一步走。

参数 UbaseAddress 表示对于起点地址的要求，为 0 表示任意。UbaseAddress 为 0 时参数 ZeroBits 表示要求所分配的起点地址前面有几位(二进制位)必须是 0，实际上就是要求所分配的区间大体上落在什么位置上。如果 UbaseAddress 非 0，这里的代码中就通过 MmLocateMemoryAreaByAddress()找一下，看这地址是否落在某个已分配的区间内。如果是的话(返回的 MemoryArea 指针非 0)，此时有三种可能：

- 这个区间在此前已经通过 NtAllocateVirtualMemory()预订或交割，因而其类型为 MEMORY_AREA_VIRTUAL_MEMORY，区间也够大。现在要做的是改变其一部或全部的状态，例如设置成 MEM_COMMIT、以及所要求的访问访问模式(例如可读写或可执行等等)。
- 这个区间是通过别的手段、而不是 NtAllocateVirtualMemory()分配的。例如 Section 的映射也会导致空间的分配，但是此时的类型不是 MEMORY_AREA_VIRTUAL_MEMORY(而是。。。。。。。。。)。只要区间够大，也允许通过 NtAllocateVirtualMemory()改变其一部或全部的状态。
- 这个区间不够大，因而失败返回。

在 ReactOS 的实现中，数据结构 MEMORY_AREA 代表着内存区间，在同一个内存区间中可以存在一个或多个 Region，以数据结构 MM_REGION 作为代表。我们既已称 Area 为区间，就只好称 Region 为“区段”了。之所以在一个区间中可以有多个区段，是因为它们的访问模式可能不同。例如可能要需要把一个区间的一部分设置成可执行，另一部分设置成只读，还有一部分设置成读写等等。此外，它们的状态也可能不同，例如在一次预订以后分好几次交割，因而可能有的区段状态为 MEM_RESERVE，而有的是 MEM_COMMIT。而所谓 Region，是指一块连续的，“均匀”的即具有相同模式和状态的虚存区段。所以前面有个参数名是 URegionSize，而不是 UAreaSize。此外，MEMORY_AREA 中的 Type 字段表示一个区间的性质和类型，例如 MEMORY_AREA_VIRTUAL_MEMORY，而 MM_REGION 中的 Type 字段则表示区段的状态，例如 MEM_COMMIT 或 MEM_RESERVE。

所以，如果找到了相应的区间，就通过 MmAlterRegion()改变目标区段的模式和状态。注意调用 MmAlterRegion() 时的最后一个参数是个函数指针，在这里指向 MmModifyAttributes()。如果 MmAlterRegion()发现所要求的空间可用，就会通过函数指针调用这个函数，其作用是对页面映射表作出相应的修改，以适应可能与前不同的访问模式，例如把只读改成读写。

读者也许会问：如果把一个区段的状态从 MEM_RESERVE 改成 MEM_COMMIT，这到底是否涉及 Swap 文件的页面分配呢？前面讲过，ReactOS 目前实际上采用的是类似于 Linux 的那种方法，所以只是改变了区段的状态，而并没有涉及 Swap 文件的页面分配，甚至没有涉及页面映射的建立。那这套机制怎么工作呢？当第一次访问某个页面时，CPU 会因为页面无映射而发生异常，而异常处理程序会根据引起异常的地址找到相应的区段并检查其状态，如果是 MEM_RESERVE 就作为出错，而若是 MEM_COMMIT 则为其分配物理页面和建立映射，并在 Swap 文件中也分配好后备页面。

如果并未指定起点地址，或者所指定的起点地址并不落在某个已分配的区间中，那就比较自由了，此时通过 MmCreateMemoryArea()分配一个地址区间并创建其 MEMORY_AREA 数据结构，再通过 MmInitialiseRegion()创建其第一个 MM_REGION 数据结构。

然后，如果参数 AllocationType 中的 MEM_COMMIT 标志位为 1，就通过 MmReserveSwapPages()记下一笔帐，以保留一定数量的 Swap 页面。不过 ReactOS 在这方面的程序还很粗糙，只能大致看出个意图。

由于本文的目的不在于存储管理，这里就不在这些问题上再深入下去了。

在目标进程的用户空间分配了一个虚存区间以后，就可以对其进行读写了。我们在这里特别感兴趣的是写入，因为 Jeffrey Richter 把一段程序拷贝到了另一个进程的用户空间。当然，由于这是在另一个进程的用户空间，不能像通常那样直接按地址指针随机写入，而需要通过另一个系统调用 NtWriteVirtualMemory()来进行成块的写入(拷贝)。

```
NTSTATUS STDCALL
NtWriteVirtualMemory(IN HANDLE ProcessHandle,
                     IN PVOID BaseAddress,
                     IN PVOID Buffer,
                     IN ULONG NumberOfBytesToWrite,
                     OUT PULONG NumberOfBytesWritten    OPTIONAL)
{
   NTSTATUS Status;
```

```
PMDL Mdl;
PVOID SystemAddress;
PEPROCESS Process;
ULONG OldProtection = 0;
PVOID ProtectBaseAddress;
ULONG ProtectNumberOfBytes;

. . . . . .

Status = ObReferenceObjectByHandle(ProcessHandle,
                                   PROCESS_VM_WRITE,
                                   NULL,
                                   UserMode,
                                   (PVOID*)(&Process),
                                   NULL);
. . . . . .

/* We have to make sure the target memory is writable.
 *
 * I am not sure if it is correct to do this in any case, but it has to be
 * done at least in some cases because you can use WriteProcessMemory to
 * write into the .text section of a module where memcpy() would crash.
 *    -blight (2005/01/09)
 */
ProtectBaseAddress = BaseAddress;
ProtectNumberOfBytes = NumberOfBytesToWrite;

/* Write memory */
if (Process == PsGetCurrentProcess())
{
    /*  目标进程就是本进程  */
    . . . . . .
    memcpy(BaseAddress, Buffer, NumberOfBytesToWrite);
}
else
{
    /* Create MDL describing the source buffer. */
    Mdl = MmCreateMdl(NULL, Buffer, NumberOfBytesToWrite);
    . . . . . .

    /* Make the target area writable. */
    Status = MiProtectVirtualMemory(Process,
                                    &ProtectBaseAddress,
                                    &ProtectNumberOfBytes,
```

```
                                        PAGE_READWRITE,
                                        &OldProtection);
        . . . . . .

        /* Map the MDL. */
        MmProbeAndLockPages(Mdl,
                            UserMode,
                            IoReadAccess);

        /* Copy memory from the mapped MDL into the target buffer. */
        KeAttachProcess(&Process->Pcb);

        SystemAddress = MmGetSystemAddressForMdl(Mdl);
        memcpy(BaseAddress, SystemAddress, NumberOfBytesToWrite);

        KeDetachProcess();

        /* Free the MDL. */
        if (Mdl->MappedSystemVa != NULL)
        {
            MmUnmapLockedPages(Mdl->MappedSystemVa, Mdl);
        }
        MmUnlockPages(Mdl);
        ExFreePool(Mdl);
    }

    /* Reset the protection of the target memory. */
    Status = MiProtectVirtualMemory(Process,
                                    &ProtectBaseAddress,
                                    &ProtectNumberOfBytes,
                                    OldProtection,
                                    &OldProtection);
    . . . . . .

    ObDereferenceObject(Process);

    if (NumberOfBytesWritten != NULL)
        MmCopyToCaller(NumberOfBytesWritten, &NumberOfBytesToWrite,
                                                    sizeof(ULONG));

    return(STATUS_SUCCESS);
}
```

首先当然要找到目标进程的进程控制块。如果目标进程即为当前进程，那就是同一用户

空间的拷贝，这当然很简单，调用一下 memcpy()就可以了。我们在这里只关心跨进程的拷贝。由于是跨进程的拷贝，这里有个如何处理源端数据的问题。显然，源端的数据是在当前进程的用户空间，而目标端是在另一个进程的用户空间，这不是简单的通过 memcpy()就可以完成的操作。怎么办呢？方法之一是分两步走，先在内核中分配一块足够大的缓冲区，从当前进程的用户空间把数据拷贝到这个缓冲区中，然后再从这个缓冲区拷贝到目标进程的用户空间。这样当然也是可以的，但是多了一次拷贝，降低了效率。另一个方法是先把源端数据所在的(物理)页面映射到内核里面、即系统空间。这样，同一个物理页面就有了两个映射，从而有了两个虚拟地址，一个在用户空间，另一个在系统空间。于是从其在系统空间的映像拷贝到目标进程的用户空间就行了，这样可以省去一次拷贝。Windows 在与设备驱动有关(包括文件操作)的系统调用中都提供了这样的手段，称为 MDL，这里就用上了。对于 MDL 将来我在谈到设备驱动框架时还会介绍，在这里读者只要知道有这么一回事就行了。

代码中先通过 MmCreateMdl()和 MmProbeAndLockPages()把源端的物理页面映射到内核里面，并加以验证。同时又通过 MiProtectVirtualMemory()把目标端页面的保护模式设置成 PAGE_READWRITE。这就为拷贝做好了准备。

接着就是所谓进程挂靠、即通过 KeAttachProcess()切换到目标进程的用户空间了。一旦切换到目标进程的用户空间，memcpy()就有了用武之地。然后又通过 KeDetachProcess()切换回原来的用户空间。

完成了拷贝以后，又通过 MiProtectVirtualMemory()恢复目标区间原有的页面保护。

最后的 MmCopyToCaller()只是把一个无符号长整数、即实际写入目标进程用户空间的长度复制到用户空间，作为系统调用的返回值。

现在，离开“阴谋”的实现只有一步之遥了，下一步就是在目标进程中为刚才拷贝过去的可执行代码创建一个线程，这是通过 NtCreateThread()实现的。

```
NTSTATUS STDCALL
NtCreateThread(OUT PHANDLE ThreadHandle,
               IN ACCESS_MASK DesiredAccess,
               IN POBJECT_ATTRIBUTES ObjectAttributes    OPTIONAL,
               IN HANDLE ProcessHandle,
               OUT PCLIENT_ID ClientId,
               IN PCONTEXT ThreadContext,
               IN PINITIAL_TEB InitialTeb,
               IN BOOLEAN CreateSuspended)
{
  . . . . . .
  PEPROCESS Process;
  PETHREAD Thread;
  . . . . . .

  Status = ObReferenceObjectByHandle(ProcessHandle, PROCESS_CREATE_THREAD,
                                     PsProcessType, PreviousMode, (PVOID*)&Process, NULL);
  . . . . . .
  Status = PsInitializeThread(Process, &Thread, ObjectAttributes, PreviousMode, FALSE);
  ObDereferenceObject(Process);
```

```
    . . . . . .

    /* create a client id handle */
    Status = PsCreateCidHandle(Thread, PsThreadType, &Thread->Cid.UniqueThread);
    . . . . . .
    Status = KiArchInitThreadWithContext(&Thread->Tcb, ThreadContext);
    . . . . . .
    Status = PsCreateTeb(ProcessHandle, &TebBase, Thread, InitialTeb);
    . . . . . .
    Thread->Tcb.Teb = TebBase;
    Thread->StartAddress = NULL;
    . . . . . .
    /*
     * Queue an APC to the thread that will execute the ntdll startup
     * routine.
     */
    LdrInitApc = ExAllocatePool(NonPagedPool, sizeof(KAPC));
    KeInitializeApc(LdrInitApc, &Thread->Tcb, OriginalApcEnvironment,
                    LdrInitApcKernelRoutine, LdrInitApcRundownRoutine,
                    LdrpGetSystemDllEntryPoint(), UserMode, NULL);
    KeInsertQueueApc(LdrInitApc, NULL, NULL, IO_NO_INCREMENT);

    /*
     * The thread is non-alertable, so the APC we added did not set UserApcPending to TRUE.
     * We must do this manually. Do NOT attempt to set the Thread to Alertable before the call,
     * doing so is a blatant and erronous hack.
     */
    Thread->Tcb.ApcState.UserApcPending = TRUE;
    Thread->Tcb.Alerted[KernelMode] = TRUE;

    oldIrql = KeAcquireDispatcherDatabaseLock ();
    PsUnblockThread(Thread, NULL, 0);
    KeReleaseDispatcherDatabaseLock(oldIrql);

    Status = ObInsertObject((PVOID)Thread, NULL, DesiredAccess, 0, NULL, &hThread);
    . . . . . .
    . . . . . .
    return Status;
}
```

参数 ThreadContext 指向一个 PCONTEXT 数据结构。这个数据结构因 CPU 而不同，对于 X86 是 CONTEXT_X86，其内容是要求新建线程开始运行时各个寄存器的初值。另一个参数 InitialTeb 指向一个“初始 TEB”，主要是给定了新建线程的堆栈位置。参数 ClientId 用

来返回一个“客户标识”CLIENT_ID，实质上是返回客户标识中的线程号。CreateSuspended 则表明是否要求新建线程一创建就被挂起。其余的参数就不言自明了。

首先当然还是找到目标进程的进程控制块。然后调用 PsInitializeThread()，这个函数虽然名为 InitializeThread，实际上却包括了创建线程、对线程的 ETHREAD 数据结构进行初始化、并将其挂入目标进程的线程队列等操作。注意对 ETHREAD 数据结构的初始化并不等同于对整个线程的初始化，因为 ETHREAD 并不代表着一个线程的全部，堆栈也是线程的一部分。

接着是 PsCreateCidHandle()。如前所述，一个 CID 是由两个 Handle 构成的，其一是进程控制块的 Handle，其二是线程控制块的 Handle。这里要做的就是为目标线程的控制块(ETHREAD)创建一个全局的线程 Handle，并把它填写在 Thread->Cid 中。

下面的 KiArchInitThreadWithContext()是个宏操作，因 CPU 的不同而定义为不同的函数，对于 X86 处理器定义为 Ke386InitThreadWithContext()。这个函数在目标线程的系统堆栈中伪造出一个中断现场，使得当目标进程被调度运行而返回用户空间时正好具有通过参数 ThreadContext 给定的上下文、即各寄存器的值。至于目标线程在用户空间的程序入口，则就是 ThreadContext 中寄存器 Eip 的值，这是必须在调用 NtCreateThread()之前设置好的。注意这与 APC 函数是两码事。

PsCreateTeb()根据参数 InitialTeb 在用户空间创建一个 TEB。TEB 的位置在用户空间顶部，PEB 的下面。由于一个进程可以有多个线程，在 PEB 下面实际上是一个 TEB 数组。每创建一个新的线程，就通过 ZwAllocateVirtualMemory()为其分配一个 TEB 页面，然后就通过 NtWriteVirtualMemory()填写其初始内容，内容主要来自 InitialTeb 和 Thread->Cid。

如果参数 CreateSuspended 非 0，表示新建线程应该一创建即被挂起，那么这里就是地方了，PsSuspendThread()将目标线程挂起。被挂起的线程将不会被调度运行。

再往下就是为新建线程准备并挂入 APC 函数了。这里通过 LdrpGetSystemDllEntryPoint()获取的还是指向 LdrInitializeThunk()的指针。我们知道，这个函数的主要功能是 DLL 的装入和动态连接，按理说只有目标进程中的第一个线程才需要执行这个函数。但是读者不妨回过去(漫谈十一)看一下__true_LdrInitializeThunk()的代码，DLL 的装入和动态连接只是在第一次进入这个函数时才执行，以后就跳过去了。而 LdrInitializeThunk()的“次要”功能，即对于 LdrpAttachThread()的调用，却是每一个线程都要执行的。特别是这里面还有对 TLS、即“线程本地存储(Thread Local Storage)”的初始化，所以每一个新建的线程都要到这个 APC 函数去转一下。

接着的 PsUnblockThread()又是关键。新建的线程至此还是被“阻塞(blocked)”的，其 ETHREAD 数据结构尚未被挂入调度队列。而 PsUnblockThread()的作用就是解除其阻塞并将其 ETHREAD 数据结构挂入调度队列。在这个操作的过程中当然不能允许发生调度，所以要用 KeAcquireDispatcherDatabaseLock ()和 KeReleaseDispatcherDatabaseLock 把这个过程保护起来。完成了这个过程以后，在发生调度的时候，新建的线程就有机会被调度运行了。

最后通过 ObInsertObject()创建一个 Handle 表项，并返回相应的 Handle，这就是目标线程的 Handle。

至于新建线程被调度运行时的流程，读者在以前就已经看到过的了。

显然，目前的 ReactOS 对这整个过程是“不设防”的，尚未实现理应与主流功能配套的安全措施，与 Windows 的代码应该还有很大的差距(有幸看到 Windows 源代码的人不妨重点考察一下有关的代码)。特别地，对于跨进程操作的安全性而言，需要有严密的“对象保护”机制。以后我们再来讨论这个问题。