ReadProcessMemory函数的分析

原创 zacklin 于 2012-04-06 16:29:39 发布 阅读量1.5k 收藏 1 点赞数 1
分类专栏: window系统内核编程 文章标签: buffer null crash list thread header

ReadProcessMemory 函数用于读取其他进程的数据。我们知道自远古时代结束后，user模式下的进程都有自己的地址空间，进程与进程间互不干扰，这叫私有财产神圣不可侵犯。但windows 里还真就提供了那么一个机制，让你可以合法的获取别人的私有财产，这就是ReadProcessMemory和WriteProcessMemory。为什么一个进程居然可以访问另一个进程的地址空间呢？因为独立的只是低2G的用户态空间，高2G的内核态空间是所有进程共享的。一段执行中的线程进入内核态后，它可以拿到别人的cr3寄存器，用该cr3替换自己的cr3便完成了地址空间的转换。理论说明完毕，下面来看实现细节：

```
BOOL
STDCALL
ReadProcessMemory (
        HANDLE  hProcess,
        LPCVOID lpBaseAddress,
        LPVOID  lpBuffer,
        DWORD   nSize,
        LPDWORD lpNumberOfBytesRead
        )
{

        NTSTATUS Status;

        Status = NtReadVirtualMemory( hProcess, (PVOID)lpBaseAddress,lpBuffer, nSize,
                (PULONG)lpNumberOfBytesRead
                );

        if (!NT_SUCCESS(Status))
        {
                SetLastErrorByStatus (Status);
                return FALSE;
        }
```

```
        return TRUE;
}
```

这是用户态ReadProcessMemory的实现，它只做了一件事那就是调用NtReadVirtualMemory。NtReadVirtualMemory函数位于ntdll中，属于所谓的桩函数，

作用就是把用户态的函数调用翻译成相应的系统调用，进入内核态。内核中一般有一个相同名字的处理函数，接收到该类型的系统调用后做实际的工作。系统调用

的细节按下不表，让我们来看NtReadVirtualMemory到底在做什么事情：

```
NTSTATUS STDCALL
NtReadVirtualMemory(IN HANDLE ProcessHandle,
                    IN PVOID BaseAddress,
                    OUT PVOID Buffer,
                    IN ULONG NumberOfBytesToRead,
                    OUT PULONG NumberOfBytesRead)
{
   NTSTATUS Status;
   PMDL Mdl;
   PVOID SystemAddress;
   PEPROCESS Process;

   DPRINT("NtReadVirtualMemory(ProcessHandle %x, BaseAddress %x, "
          "Buffer %x, NumberOfBytesToRead %d)\n",ProcessHandle,BaseAddress,
          Buffer,NumberOfBytesToRead);

   Status = ObReferenceObjectByHandle(ProcessHandle,
                                      PROCESS_VM_WRITE,
                                      NULL,
```

```
                                    UserMode,
                                    (PVOID*)(&Process),
                                    NULL);




    if (Status != STATUS_SUCCESS)
    {
       return(Status);
    }
```

ObReferenceObjectByHandle函数从代表目标进程的handle里获取EPROCESS类型的指针，存放在变量Process中。EPROCESS结构保存了能代表一个进程的

几乎所有关键数据，包括我们这里急需的cr3。

```
struct _EPROCESS
{
  /* Microkernel specific process state. */
  KPROCESS              Pcb;                          /* 000 */


。。。/*其他*/
```

```
typedef struct _KPROCESS
{
```

```
    /* So it's possible to wait for the process to terminate */
    DISPATCHER_HEADER     DispatcherHeader;             /* 000 */
    /*
     * Presumably a list of profile objects associated with this process,
     * currently unused.
     */
    LIST_ENTRY            ProfileListHead;              /* 010 */
    /*
     * We use the first member of this array to hold the physical address of
     * the page directory for this process.
     */
    PHYSICAL_ADDRESS      DirectoryTableBase;           /* 018 这是cr3*/


  。。。/*其他*/
```

接下来是从目标地址里创建一个MDL并将其锁定在主存里：

```
    Mdl = MmCreateMdl(NULL,
                      Buffer,
                      NumberOfBytesToRead);
    MmProbeAndLockPages(Mdl,
                        UserMode,
                        IoWriteAccess);
```

为什么要创建这个MDL？等会儿再说。

然后是最关键的一步，当前线程要当逃兵，叛逃至目标进程里了。。。

```
    KeAttachProcess(Process);
```

执行完KeAttachProcess后，当前线程就成了Process进程所属的线程了，悲剧啊。怎么着咱们就被策反了呢？细节我们等下再看，让我们完成主逻辑先。

```
    SystemAddress = MmGetSystemAddressForMdl(Mdl);
    memcpy(SystemAddress, BaseAddress, NumberOfBytesToRead);
```

```
    KeDetachProcess();
```

```
    if (Mdl->MappedSystemVa != NULL)
    {
       MmUnmapLockedPages(Mdl->MappedSystemVa, Mdl);
    }
    MmUnlockPages(Mdl);
    ExFreePool(Mdl);

    ObDereferenceObject(Process);

    *NumberOfBytesRead = NumberOfBytesToRead;
    return(STATUS_SUCCESS);
}
```

attach到目标进程里之后，我们又从之前生成好的MDL里获取一个虚拟地址映射，然后执行memcpy操作。这下为什么要创建MDL的秘密就清楚了，假如我们直接这样

写memcpy：

```
    memcpy(Buffer, BaseAddress, NumberOfBytesToRead);
```

看着好像没什么问题，其实问题很大。Buffer所代表的地址应该是前一个进程空间里的，但现在确实新进程空间里的，根本不是一回事。我们费劲拷贝

过去的数据，其实位于错误的内存里，等KeDetachProcess执行完切回原来的进程空间后，这些数据就全丢了，找都没地方找去。所以我们应该先从Buffer里

生成一个MDL，切换进程完成后再从该MDL里反生成一个Virtual Address，然后memcpy就可以正确的将数据拷贝到该去的地方了。

完成内存拷贝后，KeDetachProcess函数又将我们的线程从Process进程转回原来的进程，这下好，数据也偷到了，组织也回归了，原来这家伙是个间谍啊。。。

现在我们可以来看看KeAttachProcess函数到底做了什么事情了。核心行为很明确，那就是替换cr3，但是细节到底如何呢：

```
VOID STDCALL
KeAttachProcess (PEPROCESS Process)
{
   KIRQL oldlvl;
   PETHREAD CurrentThread;
   PULONG AttachedProcessPageDir;
   ULONG PageDir;

   DPRINT("KeAttachProcess(Process %x)\n",Process);

   CurrentThread = PsGetCurrentThread();
```

```
if (CurrentThread->OldProcess != NULL)
  {
     DbgPrint("Invalid attach (thread is already attached)\n");
     KEBUGCHECK(0);
  }

KeRaiseIrql(DISPATCH_LEVEL, &oldlvl);

KiSwapApcEnvironment(&CurrentThread->Tcb, &Process->Pcb);
```

这里我们把当前的IRQL提升到了DPC level，为的就是防止线程切换。然后调用KiSwapApcEnvironment把当前的apc队列也贴到目标进程里，按下不表。

```
   /* The stack of the current process may be located in a page which is
   not present in the page directory of the process we're attaching to.
   That would lead to a page fault when this function returns. However,
   since the processor can't call the page fault handler 'cause it can't
   push EIP on the stack, this will show up as a stack fault which will
   crash the entire system.
   To prevent this, make sure the page directory of the process we're
   attaching to is up-to-date. */

AttachedProcessPageDir = ExAllocatePageWithPhysPage(Process->Pcb.DirectoryTableBase);
MmUpdateStackPageDir(AttachedProcessPageDir, &CurrentThread->Tcb);
ExUnmapPage(AttachedProcessPageDir);
```

接下来如注释所说，Process->Pcb.DirectoryTableBase所代表的数据很有可能正在硬盘里的，物理如何也要保证它在内存里，因为函数返回时要做栈操作，

如果Process->Pcb.DirectoryTableBase在硬盘上，栈操作就会引起page fault，而处理page fault前又必须要push eip，悲剧就要发生了。同样的，

stack base 和 stack top这两哥们也一定得在内存里，MmUpdateStackPageDir做的就是这个事情。

```
    CurrentThread->OldProcess = PsGetCurrentProcess();
    CurrentThread->ThreadsProcess = Process;
    PageDir = Process->Pcb.DirectoryTableBase.u.LowPart;
    DPRINT("Switching process context to %x\n",PageDir);
    Ke386SetPageTableDirectory(PageDir);
    KeLowerIrql(oldlvl);
  }
```

最后做的事情就简单了，把当前线程的ThreadsProcess换成新的，再把当前的cr3换成Process->Pcb.DirectoryTableBase.u.LowPart。一番梳妆打扮后，

敌人就分不清咱的身份了。

至此为止，ReadProcessMemory函数分析完毕。个人觉得有几个细节是需要注意的：第一呢，lpBaseAddress和lpBuffer所在的进程空间是不同的。第二呢，

KeRaiseIrql和KeLowerIrql这两个函数一定要限制在进程空间切换的函数内，绝对不能把memcpy放在它们中间，因为KeRaiseIrql之后page fault就没法处理

了，而memcpy不产生page fault那是不可能的，想都不要想。