

我给出一个反汇编Win32 API函数CreateProcess的例子，来演示研究子系统的技术，同时演示Win32是如何与Windows NT的执行系统协同工作的。

从MSDN中得到函数原型：

```
BOOL CreateProcess(
    LPCTSTR lpApplicationName, // pointer to name of executable module
    LPTSTR lpCommandLine, // pointer to command line string
    LPSECURITY_ATTRIBUTES lpProcessAttributes, // process security attributes
    LPSECURITY_ATTRIBUTES lpThreadAttributes, // thread security attributes
    BOOL bInheritHandles, // handle inheritance flag
    DWORD dwCreationFlags, // creation flags
    LPVOID lpEnvironment, // pointer to new environment block
    LPCTSTR lpCurrentDirectory, // pointer to current directory name
    LPSTARTUPINFO lpStartupInfo, // pointer to STARTUPINFO
    LPPROCESS_INFORMATION lpProcessInformation // pointer to
PROCESS_INFORMATION
);
```

函数中所有的参数都没有详尽的描述。很快，在开始的几行中，建立了异常处理__except_handler3(堆栈中的结构体对应于Visual C的结构体)。然后根据dwCreationFlags进行相应有趣的处理。在任何情况下都会在dwCreationFlags里去掉标志CREATE_NO_WINDOW(对于我来说这是个谜)。之后检查不允许的标志组合DETACH_PROCESS | CREATE_NEW_CONSOLE。如果这些位被同时设置就会输出错误。从新进程优先级中选择一个优先级(除一个之外，清除所有dwCreationFlags中的优先级位)。如果要求是REAL_TIME优先级，但不能分给处理器，则设置为HIGH_PRIORITY。接下来是对参数lpApplicationName、lpCommandLine、lpEnvironment的繁琐处理。分析结果表明，函数CreateProcessW实际上在文档中已经写明。因此，我们考虑到命令行和应用程序名已不相同。DOS风格的形式为完整的路径。使用未公开的ntdll.dll中的函数：

```
NTSYSAPI
BOOLEAN
NTAPI
RtlDosPathNameToNtPathName_U (char* lpPath,
                               RTL_STRING *NtPath,
                               BOOLEAN AllocFlag,
                               RTL_STRING *Reserved);
```

结果会得到\\?\\:\样的路径。然后，填充公开了的OBJECT_ATTRIBUTES结构体，在ObjectName域中放入指向获得的路径的指针并调用未公开的函数：

```
NTSYSAPI
NTSTATUS
NTAPI
NtOpenFile (OUT DWORD* Handle, IN ACCESS_MASK DesiredAccess,
            OBJECT_ATTRIBUTES* ObjAttr, PIO_STATUS_BLOCK IoStatusBlock,
            DWORD ShareAccess, DWORD OpenOptions);
```

访问使用的是SYNCHRONIZE | FILE_EXECUTE。取得打开文件的句柄用于调用另外一个未公开的函数：

```
NTSYSAPI
NTSTATUS
NTAPI
NtCreateSection(
```

```

OUT PHANDLE SectionHandle,
IN ACCESS_MASK DesiredAccess,
IN POBJECT_ATTRIBUTES ObjectAttributes OPTIONAL,
IN PLARGE_INTEGER MaximumSize OPTIONAL,
IN ULONG Protect,
IN ULONG Attributes,
IN HANDLE FileHandle OPTIONAL
);

```

大多数未公开的系统函数都是由相应的公开的Win32 API调用的。API函数CreateFileMapping是对NtCreateSection的封装。实际上,即使系统直接调用这些函数,也没人会干扰(而且还节省开销)。有趣的是NtCreateSection的一个主要的、由API函数生成的参数:

```

DesiredAccess=(flProtectLow==PAGE_READWRITE)?STANDARD_RIGHTS_REQUIRED|7
:
STANDARD_RIGHTS_REQUIRED | 5;

```

DesiredAccess只可以取两个值。从CreateProcessW中调用的形式如下:

```

NtCreateSection ( &SectionHandle, STANDARD_RIGHTS_REQUIRED| 0x1F,
NULL, &qwMaximumSize,
PAGE_EXECUTE_READ, SEC_IMAGE, NtFileHandle );

```

这样就得到了映像,并将文件——映像源——关闭。这是用公开的NtClose函数进行的。来分析一下NtCreateSection返回后的代码。对错误处理这里就不进行了,否则会十分繁琐,要讨论大量的次要的函数。我们来研究没有发生错误而且映像是PE映像的情况。调用著名的未公开函数:

```

NTSYSAPI NTSTATUS NTAPI
NtQuerySection(
    IN HANDLE SectionHandle,
    IN SECTION_INFORMATION_CLASS SectionInformationClass,
    OUT PVOID SectionInformation,
    IN ULONG SectionInformationLength,
    OUT PULONG ReturnLength OPTIONAL
);

```

系统中有一些类似于NtQueryInformationXxxxx这样的函数(未公开)。要说是未公开的,在NTDDK.H中还是描述了一些函数的原型和调用这些函数用到的结构体信息。Matt Pietrek在其在Microsoft Systems期刊(MSDN中有)的文章中详细描述了NTDDK.H中的NtQueryInformationProcess的主要功能。遗憾的是,关于NtQuerySection函数的信息是不存在的。所有这样的函数都有实际上相同的原型并处理操作系统中的对象。NtQuerySection返回两类信息(SectionInformationClass可以为0或1)。对应于取0还是取1,结构体的大小为16或是58个字节。CreateProcessW调用的SectionInformation参数的信息类是1。

```

Struct SECTION_INFO_CLASS1 {
    DWORD EntryPoint;
    DWORD field_4;
    DWORD StackReserved;
    DWORD StackCommitted;
    DWORD SubSystem;
    DWORD ImageVersionMinor;
    DWORD ImageVersionMajor;
    DWORD unknown1;
    DWORD Characteristics;
};

```

```

DWORD Machine;
DWORD Unknown[4];
};

```

我们看到，这个信息是从PE映象的首部中取得的。在主要的域Characteristics中输出的是映象的类型(是否是可执行的)。然后检查机器类型，解析SubSystem域，检查映象版本。并最终调用未公开的函数：

```

NtCreateProcess(
    OUT PHANDLE ProcessHandle,
    IN ACCESS_MASK DesiredAccess,
    IN POBJECT_ATTRIBUTES ObjectAttributes,
    IN HANDLE ParentProcess, //-1
    IN BOOLEAN InheritHandles,
    IN HANDLE SectionHandle,
    IN HANDLE DebugPort OPTIONAL, // NULL
    IN HANDLE ExceptionPort OPTIONAL //NULL
);

```

由此建立了Windows NT的进程对象。关闭映象，因为已经不再需要了。接着设置对象属性，调用未公开函数NtSetInformationProcess

```

NTSYSAPI
NTSTATUS
NTAPI
NtSetInformationProcess(
    IN HANDLE ProcessHandle,
    PROCESSINFOCLASS ClassInfo,
    IN PVOID Information,
    IN ULONG Length,
);

```

在NTDDK.H中有枚举值_PROCESSINFOCLASS，这个值描述了信息类。调整信息类的值：ProcessDefaultHardErrorMode, ProcessBasePriority。对于这些类，信息结构体本身就是一个32位的DWORD。然后调用未公开的函数，Matt Pietrek在其文章中介绍过该函数：

```

NTSYSAPI
NTSTATUS
NTAPI
NtQueryInformationProcess(
    IN HANDLE ProcessHandle,
    IN PROCESSINFOCLASS ProcessInformationClass,
    OUT PVOID ProcessInformation,
    IN ULONG ProcessInformationLength,
    OUT PULONG ReturnLength OPTIONAL
);

```

我们取得的信息是ProcessBasicInfo，NTDDK.H文件中有对其的描述。

```

typedef struct _PROCESS_BASIC_INFORMATION {
    NTSTATUS ExitStatus;
    PPEB PebBaseAddress;
    KAFFINITY AffinityMask;
    KPRIORITY BasePriority;
    ULONG UniqueProcessId;
};

```

```

        ULONG InheritedFromUniqueProcessId;
    } PROCESS_BASIC_INFORMATION;

```

对于CreateProcessW来说，必需的信息是PEB的地址。因为在获得这项信息之后很快就调用内部函数_BasePushProcessParameters。从参数判断，其用途是调整仅由此进程产生的地址空间。接下来调用两个内部的复杂函数。先调用_BaseCreateStack。_BaseCreateStack分配并调整进程堆栈。第一，选出用于保留和提交(reservrd和committed)堆栈的值。而且，如果SizeReserved和SizeCommitted为0，则要从发出CreateProcess的进程的PE文件的首部中获取这些值。接着对这些值进行修整并在进程产生的地址空间中保留内存，对此用到未公开的函数NtAllocateVirtualMemory(对应于Win32 API函数VirtualAllocEx, VirtualAllocEx是对其非常简单的封装，而且这两个函数的参数完全相同)。之后，进行两个调用，用下面的伪码能更简洁的说明：

```

FreeReserved=SizeReserv-SizeCommitted;
ReservedAddr+=FreeReserved;
if(SizeReserved<=SizeCommitted) fl=0;
else {
    ReservedAddr-=Delta;
    SizeCommitted+=Delta;
    fl=1;
}
NtAllocateVirtualMemory(Han,&ReservedAddr,0,SizeCommitted,1000,4);

//[skipped]

NtProtectVirtualMemory
    (ProcHan,&ReservedAddr,Delta,PAGE_READWRITE|PAGE_GUARD,&OldProt);
    /* 对VirtualProtectEx的封装 */

```

可见，这里在保留区域中分配内存(在其末尾)。并且分配的内存大于Delta。这一部分(大小为Delta)的属性是PAGE_GUARD和PAGE_READWRITE。最后得到以下结构体：

```

***Stack***
-----?-ReservedAddr
|           |
|           |
| RESERVED   | <- SizeReserved - (SizeCommitted+Delta)
|           |
|-----| -CommittedAddr
| GUARD PAGE | <- Delta
|-----|
| READ_WRITE | <- SizeCommitted
|           |
L-----SS:ESP

```

这样，为堆栈分配了SizeCommitted字节。保留了SizeReserved。之后在堆栈之下的保留区分配的内存被转换为GUARD页(转换成这种页可以引发异常)。从源代码中可以看到，错误的Delta的大小可能会产生悲惨的后果。因为这可是个关键的信息——我们来看从哪里找出Delta的值：

```

.text:77F04B99      mov     eax, large fs:18h
.text:77F04B9F      mov     ecx, [eax+30h] ; PEB
.text:77F04BA2      mov     eax, [ecx+54h] ; READ ONLY DATA
                   ; ReadOnlyStaticServerData
.text:77F04BA5      mov     edx, [eax+4]

```

```
[skipped]
.text:77F04BB1      mov     esi, [edx+128h] ; Delta
```

在这一部分里, **EAX**寄存器指向用于所有进程的全局区域。这个区域只允许被读取。当然, 已给出的关于堆栈的更高层次的信息是众所周知的, 而这些信息的真实性在源代码中得到了证实。结果, 执行**BaseCreateStack**函数填充**StackInformation**结构体。

```
Typedef struct _StackInformation
{
    DWORD Reserved0;
    DWORD Reserved1;
    DWORD AddressOfTop;
    DWORD CommitAddress;
    DWORD ReservedAddress;
} StackInformation;
```

从这个结构体中得到信息本质上是个参数, 用来调用下面这个有趣的函数**BaseInitializeContext**:

```
BaseInitializeContext(PCONTEXT Context, // 0x200 bytes
PPEB Peb,
PVOID EntryPoint,
DWORD StackTop,
int Type // union (Process, Thread, Fiber)
);
```

这个函数的几个参数:**PEB**的地址, 堆栈的入口点和参数定义了要创建的上下文(纤程, 进程、线程)。函数填充**CONTEXT**结构体(**NTDDK.H**中有)的几个域。其中一个域很有意思, 在其中放置了起始点(**BaseFiberStart**、**BaseProcessStartThunk**、**BaseThreadStartThunk**中的一个)。这个点“分娩”出了线程, 产生的线程就在新的上下文中执行。实际上, 所有三个偏移处的代码都很简短——就是填充相应的堆栈映象并转到两个函数中的某一个。这两个函数分别是**_BaseProcessStart**和**_BaseThreadStart**。这两个函数很是相象, 我们只看**_BaseProcessStart**函数。

这个函数在链表中建立了第一个异常处理(见**TEB**)。当对内存进行了错误的访问时, 正是这个异常处理调用了那个有**OK**和**CANCEL**的对话框。这个处理程序会结束当前进程。但有时如果异常由错误的服务线程产生, 则只结束这个线程。

于是, 在**BaseInitializeContext**返回后, 就填充相应的结构体。并且这个结构体被用作未公开的**NtCreateThread**函数的参数。**NtCreateThread**的原型如下:

```
NTSYSAPI
NTSTATUS
NTAPI
NtCreateThread(
    OUT PHANDLE ThreadHandle,
    IN ACCESS_MASK DesiredAccess,
    IN POBJECT_ATTRIBUTES ObjectAttributes OPTIONAL,
    IN HANDLE ProcessHandle,
    OUT PCLIENT_ID ClientID,
    IN PCONTEXT Context, /* see _BaseInitializeContext */
    IN StackInformation* StackInfo, /* see _BaseCreateStack */
    IN BOOLEAN CreateSuspended /* == 1 */
);
```

终于, 在对PE映象的SubSystem主要域的数据进行处理之后, 通过LPC转到Win32服务。进程应该只在Win32子系统下创建。关于此原因的一些高层次信息可以在Halen Kaster的书中读到。

对于CreateProcess函数来说, 必须完成的任务就是启动线程(当然, 如果没有在参数dwCreationFlags中设置CREATE_SUSPEND标志)。线程的启动进行对NtResumeThread(对Win32的ResumeThread的封装)的调用。完成了! 现在剩下的还有释放内存和正确的退出。

到此对Win32子系统的CreateProcess函数的主要分析可以得出结论: 子系统通常与Windows NT的执行体系统协同工作, 子系统大多都使用未公开的函数, 子系统通过LPC与自己的服务器通讯, 许多Win32 API函数都是对Nt函数的封装。所有这些都是我们熟知的, 但我们需要用反汇编来证实。