

Windows NT主要是由C写成的, 所以总的来说进程本身的反汇编不是很复杂。通常对局部变量和参数的使用是通过地址和用EBP形成的stack frame来进行的。例如:

```
PAGE:801932D4  mov    eax, large fs:0
PAGE:801932DA  push   ebp
PAGE:801932DB  mov    ebp, esp ; 堆栈中的frame
PAGE:801932DD  push   0FFFFFFFh
PAGE:801932DF  push   offset $T13371
PAGE:801932E4  push   offset __except_handler3
PAGE:801932E9  push   eax
PAGE:801932EA  xor    eax, eax
PAGE:801932EC  mov    large fs:0, esp
PAGE:801932F3  sub    esp, 64h ; 局部变量在堆栈中的位置
PAGE:801932F6  mov    [ebp+Flag], al ; 使用局部变量
```

有时, 编译器会生成更为优化的代码, 直接使用ESP来引用堆栈。

```
.text:80124320  sub    esp, 8
.text:80124323  test   byte ptr ds:_NtGlobalFlag+2, 8
.text:8012432A  push   ebx
.text:8012432B  push   esi
.text:8012432C  push   edi
.text:8012432D  push   ebp
.text:8012432E  jnz    loc_FFFFF000_80124443
.text:80124334  mov    esi, [esp+18h+arg_0] ; 引用第一个参数
```

IDA PRO这个特别的反汇编器可以跟踪堆栈的创建, 所以建立了下面例子中的相应的常量。在调用函数时, 参数以相反的顺序放在堆栈中。调用函数自己要负责清理堆栈(编译器就是这样为C函数生成代码的)。

```
PAGE:80193288  push   [ebp+ExceptPort]
PAGE:8019328B  push   [ebp+DebugPort]
PAGE:8019328E  push   [ebp+SectionHandle]
PAGE:80193291  push   [ebp+bInheritHandle]
PAGE:80193294  push   edx ; ffffffff
PAGE:80193295  push   [ebp+ObjectAttributes] ;(参数3)
PAGE:80193298  push   [ebp+Access] ; (参数2)
PAGE:8019329B  push   ecx ; Handle (参数1)
PAGE:8019329C  call   _PspCreateProcess@32
; PspCreateProcess (Handle,Access,
; ObjectAttributes,-1,bInheritHandle,SectionHandle,DebugPort,ExceptPort);
PAGE:801932A1
PAGE:801932A1  NtCreateProcessExit: ; CODE XREF: _NtCreateProcess@32+71 j
PAGE:801932A1 ; _NtCreateProcess@32+80 j
PAGE:801932A1  mov    ecx, [ebp+var_10]
PAGE:801932A4  pop    edi
PAGE:801932A5  mov    large fs:0, ecx
PAGE:801932AC  pop    esi
PAGE:801932AD  pop    ebx
PAGE:801932AE  mov    esp, ebp
PAGE:801932B0  pop    ebp
PAGE:801932B1  retn   20h ; 清理堆栈(返回后ESP加上0x20)
```

为了提高速度, 有时在OS内核中会使用fastcall的函数, 参数通过堆栈传递。例如:

```
PAGE:8018CC9D  mov    ecx, [ebp+pObject] ; 第一个参数(下一个参数在edx中)
PAGE:8018CCA0  call   @ObfDereferenceObject@4 ; fastcall函数
```

在这个例子中使用了内核中内部的非导出函数。在下一个例子中, 我们使用HAL.DLL中的未公开的导出的fastcall函数:

```
.text:801335E2  mov    ecx, eax ; OldIrql
.text:801335E4  call   ds:__imp_@KfLowerIrql@4
```

fastcall函数在其名字中都有字符'f'。

Microsoft提供了符号信息, 这些符号信息可以用来调试程序。这些信息可以确定内部函数(非导出的)和全局变量的真实名称。这就简化了辨别函数和变量名称的工作, 而且除此之外, 通过后缀@N可以确定函数参数的数量。

调试信息是以NT 4.0的.DBG文件和Windows 2K的.PDB文件的形式提供的。SoftICE和IDA都通晓PDB和DBG文件(IDA使用插件来加载ntoskrnl.pdb)。

内核中的函数、变量和结构体的名称本身都能表达一些信息。前缀通常有两种意思, 描述函数的特征或是用于子系统的数据。例如:Mm - 子系统内存, Cc - 缓存, Ob - 对象管理器, Ps - 进程管理, Se - 内存管理器, Ke - 内核其它的结构体, Ex - 执行体系统。如果函数是初始化用的(或其可能会转入此类函数), 则在前缀的第一个字母后加一个字符'i'。例如Ki、Mi。使用Fastcall的函数在前缀后加'f'。系统调用使用前缀Nt。这些函数不是内核导出的函数, 它们的地址记录在service table里。调用服务要通过软中断0x2e。内核导出了Zw函数, 这些函数是对中断的封装。

```
.text:8011A49C _ZwCreateFile@44 proc near ; CODE XREF: _FsRtlpOpenDev@8+4D p
.text:8011A49C arg_0      = byte ptr 4
.text:8011A49C  mov    eax, 17h
.text:8011A4A1  lea    edx, [esp+arg_0]
.text:8011A4A5  int    2Eh ; 中断处理程序调用NtCreateFile
.text:8011A4A7  retn   2Ch
.text:8011A4A7 _ZwCreateFile@44 endp
```

我不知道字符Zw是什么意思(好像只有内核的设计者知道)。可能是Zero Wheel(或是零环)的意思, 因为Zw函数是从内核模式调用的(在DDK中描述了一些)。从用户模式下, 系统服务是通过NTDLL(实现于用户模式)调用的, NTDLL导出了NtXXX的封装函数ZwXXX。

内核中的名字主要要遵循一种描述规则。当然, 名字本身承载着意义。名字的内容通常是行为及其对象, 即对对象进行某种行为。如:ObCreateObject。

许多的操作系统函数都仅仅是对内核内部函数的封装。例如, NtCreateSection调用了MmCreateSection, 用的参数也都相同。现在, 如果统计一下的话, 许多Nt函数的原型都是Windows NT内核研究者所熟知的, 许多内部函数的原型也就可以不用通过逆向工程而获得。有了C语言函数的原型再学习其结构和思想就轻松多了。

理论上讲, 取得关于内核的信息的更简单的方法不是反汇编内核的映像, 而是其它的代码。例如, 使用WinDbg的kernel-mode extensions的代码。WinDbg的扩展中有额外的命令, 扩展的调试命令集。其中有明显使用内核内部结构的命令, 或是能减轻分析内核内部结构工作的命令。例如命令!ca、!tokenfields、!processfields等等。反汇编kdextx86.dll和kdex2x86.dll的代码可以得到某些结构的信息。

内核调试扩展是个.DLL。导出扩展的命令所用的名字与在WinDbg调试器中遇到的名字是相同的。例如, processfields。扩展的DLL导出了函数WinDbgExtensionDllInit, 这个函数是在

加载扩展后从WinDbg调试器中调用的。函数的原型如下：

```
VOID WinDbgExtensionDllInit(PWINDBG_EXTENSION_APIS lpExtensionApis,  
                           USHORT MajorVersion,  
                           USHORT MinorVersion)
```

第一个参数是指向在.DLL中使用的API的指针。WINDBG_EXTENSION_APIS结构体包含以下成员，这些成员定义了访问扩展函数集：

lpOutputRoutine	- 在控制台输出字符串
lpGetExpressionRoutine	- 计算表达式的值
lpGetSymbolRoutine	- 取得符号在内存中的地址
lpDisasmRoutine	- 反汇编内存
lpCheckControlCRoutine	- 检查是否按下CTRL-C (未实现)
lpReadProcessMemoryRoutine	- 读进程内存，带有对GPF的保护。
lpWriteProcessMemoryRoutine	- 写内存
lpGetThreadContextRoutine	- 取得进程寄存器的值
lpSetThreadContextRoutine	- 设置寄存器
lpIoctlRoutine	- 未实现
lpStackTraceRoutine	- 跟踪堆栈

这样.DLL导出了对应于扩展命令的函数，并可以用于与调试器的有限而熟悉的函数集的协同工作。进一步给出实现扩展命令的函数的原型。

```
#define DECLARE_API32(s) \
    CPPMOD VOID \
    s( \
        HANDLE hCurrentProcess, \
        HANDLE hCurrentThread, \
        ULONG dwCurrentPc, \
        ULONG dwProcessor, \
        PCSTR args \
    )
```

有趣的是参数args，它指向WinDbg中命令的字符串。借助反汇编可以取得足够的信息以研究扩展命令的工作逻辑。在首要的研究中可以选出直接操纵内核结构体和能辨别结构体成员的命令。例如，命令!ca的代码说明了内核结构体control area和segment。这个命令的逻辑并不复杂：辨别命令行，从内核内存中读取所要的结构体，输出域中的内容。

但是，扩展命令经常并不会列出内核结构体的所有内容。并且，从名字中并不总是能明确的推断出域的含义，但是反汇编这条命令可以简化对内核内部函数的分析工作。在任何情况下都会有机会对信息做对比，从各种各样的线索中取得信息。