بررسی معماری سیستم عامل ویندوز

(توابع API بومی ویندوز)

مریم نژادکمالی

nezhadkamali@cert.um.ac.ir

آزمایشگاه تخصصی آپا در زمینه امنیت فن‌آوری اطلاعات و ارتباطات

http://cert.um.ac.ir

cert@um.ac.ir

ویرایش اول – تیرماه ۱۳۹۳

شماره سند: APA_FUM_W_MAL_0120

چکیده

در این مقاله، واسط فراخوانی‌های ویندوز شرح داده می‌شود. ابتدا با نگاهی به برخی از ساختارهای مد هسته که باعث انجام فراخوانی‌های سیستمی شده‌اند، شروع می‌کنیم. سپس، نحوه‌ی شناسایی توابع API را با استفاده از ابزارهای اشکال‌زدا مانند kd.exe نشان می‌دهیم. بعد از آن به توضیح چگونگی یک فراخوانی می‌پردازیم و توضیح می‌دهیم که در صورتی که مستندی برای یک تابع API وجود نداشته باشد، چگونه اطلاعاتی از آن به دست آوریم. سرانجام، سیر اجرای یک تابع را از مد کاربر به مد هسته به صورت گام به گام بررسی می‌کنیم.

واژه‌های کلیدی

مد هسته، مد کاربر، جدول توصیف سرویس سیستمی، جدول توصیف وقفه، جدول بردار وقفه، توابع API، SSDT، IDT، IVT.

۱- مقدمه

اصلی‌ترین سرویسی که توسط سیستم عامل برای برنامه‌های مد کاربر ارائه می‌شود، مجموعه‌ای از روتین‌هایی است که از آن‌ها به عنوان API[1] یاد می‌شود. توابع API ویندوز در برخی از کتابخانه‌های DLL مانند advapi32.dll، kernel32.dll، user32.dll و wsock32.dll وجود دارند.

ویندوز برای بسیاری از کارهای خود و نیز برنامه‌های کاربردی به طور متعدد از این توابع استفاده می‌کنند، اما تعداد برنامه‌های کمی هستند که توابع بومی ویندوز را مورد استفاده قرار می‌دهند. توابع بومی به APIهایی اطلاق می‌گردد که اکثرا توسط خود سیستم عامل استفاده می‌شوند و یا برای مثال توسط روتین‌های موجود در kernel32 به منظور پیاده‌سازی APIهای ویندوز استفاده شده‌اند. اکثر فراخوانی توابع بومی توسط ntoskrnl.exe انجام شده و کاربران نیز برای فراخوانی این توابع می‌توانند از ntdll.dll استفاده کنند. در ادامه، به ساختار مد هسته نگاهی می‌اندازیم تا بتوانیم با نحوه‌ی فراخوانی‌های سیستمی آشنا شویم.

۲- جدول بردار وقفه یا جدول توصیف‌کننده‌ی وقفه

جدول بردار وقفه[2] (IVT) یا *جدول توصیف‌کننده‌ی وقفه*[3] (IDT) یک ساختمان داده‌ها برای پیاده‌سازی بردارهای وقفه در معماری x86 است. IDT توسط پردازنده‌ها به منظور پاسخ مناسب به وقفه‌ها و استثناها استفاده می‌شود. IDT می‌تواند توسط وقفه‌های سخت‌افزاری، وقفه‌های نرم‌افزاری و یا استثناهای پردازنده راه‌اندازی شود که به همه‌ی آن‌ها به صورت خلاصه وقفه گفته می‌شود.

در سیستم‌های عامل با *حالت واقعی*[4] مانند MS-DOS، اصلی‌ترین ساختمان داده‌های موجود در سطح سیستم، IVT بود که در واقع یک راه ورود برای رسیدن به هسته محسوب می‌شد. هر فراخوانی سیستمی در MS-DOS به وسیله‌ی یک وقفه‌ی نرم‌افزاری (به طور معمول با استفاده از دستور INT 0x21 و یک کد تابع که در داخل رجیستر

[1] Application Programming Interface
[2] Interupt Vector Table
[3] Interupt Descriptor Table
[4] Real mode

AH قرار می‌گرفت) ایجاد می‌شد. در سیستم عامل ویندوز این جدول با نام *جدول توزیع وقفه*[1] (IDT) تولد دوباره یافت و البته مقداری از زرق و برق‌هایش را از دست داد. البته این بدین معنا نیست که IDT به اندازه‌ی IVT مفید نیست، بلکه هنوز هم IDT یک راه خوب برای ورود به هسته محسوب می‌شود.

۲-۱- نگاهی دقیق‌تر به IDT

زمانی‌که ویندوز راه‌اندازی می‌شود، ترتیب پردازنده‌های در حال اجرا را بررسی کرده و متناسب با آن‌ها فراخوانی‌های سیستمی را تنظیم می‌کند. به طور خاص، پردازنده‌های قبل از Pentium II از دستور INT 0x21 به منظور فراخوانی‌های سیستمی استفاده می‌کردند. پردازنده‌های IA-32 استفاده از IDT را برای این وظیفه حذف کرده و به جای آن از دستور SYSENTER برای پریدن به فضای هسته استفاده می‌کنند. البته در ویندوزهای جدید هنوز هم از IDT برای پاسخ به وقفه‌های سخت‌افزاری و مدیریت استثناهای پردازنده استفاده می‌شود.

این را به خاطر داشته باشید که هر پردازنده یک رجیستر خاص به نام IDTR دارد که در آن آدرس شروع و طول IDT نگهداری می‌شود. بنابراین، هر یک از پردازنده‌ها IVT مخصوص به خود را دارد. در نتیجه، پردازنده‌های مختلف می‌توانند *روتین‌های سرویس وقفه*[2] (ISR) متفاوتی را فراخوانی کنند.

با توجه به خصوصیات اینتل، در هر IDT می‌تواند ۲۵۶ توصیف‌کننده وجود داشته باشد که هر کدام از آن‌ها ۸ بایتی هستند. می‌توان با استفاده از ابزار kd.exe به آدرس شروع و طول IDT پی برد. بدین منظور باید از IDT یک رونوشت گرفت.

```
kd> rM 0x100
gdtr=82430000   gdtl=03ff idtr=82430400   idtl=07ff tr=0028  ldtr=0000
```

با توجه به خروجی دستور می‌توان متوجه شد که IDT از آدرس 0x82430400 شروع می‌شود و دارای ۲۵۶ مدخل است. آدرس آخرین بایت IDT را می‌توان با جمع کردن IDTR (آدرس شروع) و IDTL (طول) به دست آورد.

اگر بخواهیم می‌توانیم از آدرس 0x82430400 تا 0x82430BFF یک رونوشت بگیریم و توصیف‌کننده را به طور دستی کدبرداری کنیم. اما یک راه حل ساده‌تر، استفاده از دستور !idt در kd.exe است. با استفاده از این دستور می‌توان

[1] Interupt Dispatch Table
[2] Interrupt Service Routines

از نام و آدرس روتین‌های ISR مطلع شد.

```
kd> !idt -a
00:      8188d6b0 nt!KiTrap00
01:      8188d830 nt!KiTrap01
02:      Task Selector = 0x0058
03:      8188dc84 nt!KiTrap03
...
```

خروجی بالا به صورت خلاصه آورده شده است. از ۲۵۴ مدخل، تقریبا یک چهارم آن‌ها با معنی هستند. اکثر مدخل‌ها (حدود ۲۰۰ مورد) شبیه ISR زیر هستند:

```
8188bf10 nt!KiUnexpectedInterrupt16
```

روتین‌های KiUnexpectedInterrupt به طور متوالی در حافظه قرار گرفته‌اند و همه‌ی آن‌ها در انتها به روتین KiEndUnexpectedRange ختم می‌شوند.

اگرچه سخت‌افزارهای جدید، ویندوز را مجبور می‌کنند تا در برابر دستور SYSENTER که یک پرش به فضای هسته را مهیا می‌کند تسلیم شود، اما همچنان وقفه‌ی 0x2E در IDT نیز این کار را انجام می‌دهد.

ISRای که وقفه‌ی 0x2E را مدیریت می‌کند، روتین KiSystemService نامیده می‌شود. این روتین در واقع یک *توزیع‌کننده‌ی سرویس‌های سیستمی*[1] است که از اطلاعاتی که برای آن فرستاده می‌شود، استفاده می‌کند تا آدرس تابع API بومی مورد نظر را پیدا کرده و آن را فراخوانی کند.

۳- فراخوانی‌های سیستمی از طریق وقفه

زمانی‌که دستور INT 0x2E برای فراخوانی توابع سیستمی استفاده می‌شود، *عدد سرویس سیستمی*[2] که با نام dispatch ID نیز شناخته می‌شود، در رجیستر EAX قرار می‌گیرد. این عدد، یک عدد منحصربه‌فرد بوه و نشان‌دهنده‌ی یکی از فراخوانی‌های سیستمی است. برای مثال، اگر به زمان ویندوز ۲۰۰۰ برگردیم، یعنی زمانی که فراخوانی‌ها به طور معمول مبتنی بر وقفه بودند، احضار روتین KiSystemService به صورت زیر بود:

[1] System Service Dispatcher
[2] System Service Number

```
ntdll!NtDeviceIoControlFile:
   move eax, 38h
   lea edx, [esp+4]
   int 2Eh
   ret 28h
```

این کد اسمبلی که یک کد مد کاربر است، برای فراخوانی تابع NtDeviceIoControlFile در ویندوز ۲۰۰۰ به کار می‌رود. این تابع در ntdll.dll قرار دارد که دروازه‌ی ورود به هسته‌ی سیستم عامل است. اولین چیزی که در این کد دیده می‌شود، بارگذاری عدد سرویس سیستمی معادل 0x38h در رجیستر EAX می‌باشد. بعد از آن، یک آدرس برای مقدار بالای پشته در رجیستر EDX ذخیره می‌شود و سرانجام وقفه‌ی 0x2E اجرا می‌گردد.

۴- دستور SYSENTER

امروزه اکثر ماشین‌ها برای رفتن از مد کاربر به مد هسته از دستور SYSENTER استفاده می‌کنند. برای این که پردازنده بداند که به کجا قرار است بپرد و مکان آن در کجای پشته قرار دارد، باید قبل از فراخوانی دستور SYSENTER سه رجیستر ۶۴ بیتی ماشین مقداردهی شوند. این رجیسترها که در جدول زیر قابل مشاهده هستند، می‌توانند با دستورهای RDMSR و WRMSR تغییر پیدا کنند.

Policy	Address	Description
IA32_SYSENTER_CS	0x174	Specifies kernel-mode code and stack segment selectors
IA32_SYSENTER_ESP	0x175	Specifies the location of the kernel-mode stack pointer
IA32_SYSENTER_EIP	0x176	Specifies the kernel-mode code's entry point

اگر از محتوای رجیسترهای IA32_SYSENTER_CS و IA32_SYSENTER_EIP با استفاده از دستور RDMSR یک رونوشت بگیریم، مشخص می‌شود که به یک مدخل از فضای هسته به نام KiFastCallEntry اشاره می‌کنند. اگر روتین KiSystemService را دی‌اسمبل کنیم، خواهیم دید که کنترل برنامه در انتها به روتین KiFastCallEntry منتقل خواهد شد. بنابراین، در نهایت هر دو فراخوانی سیستمی به یک مقصد هدایت می‌شوند.

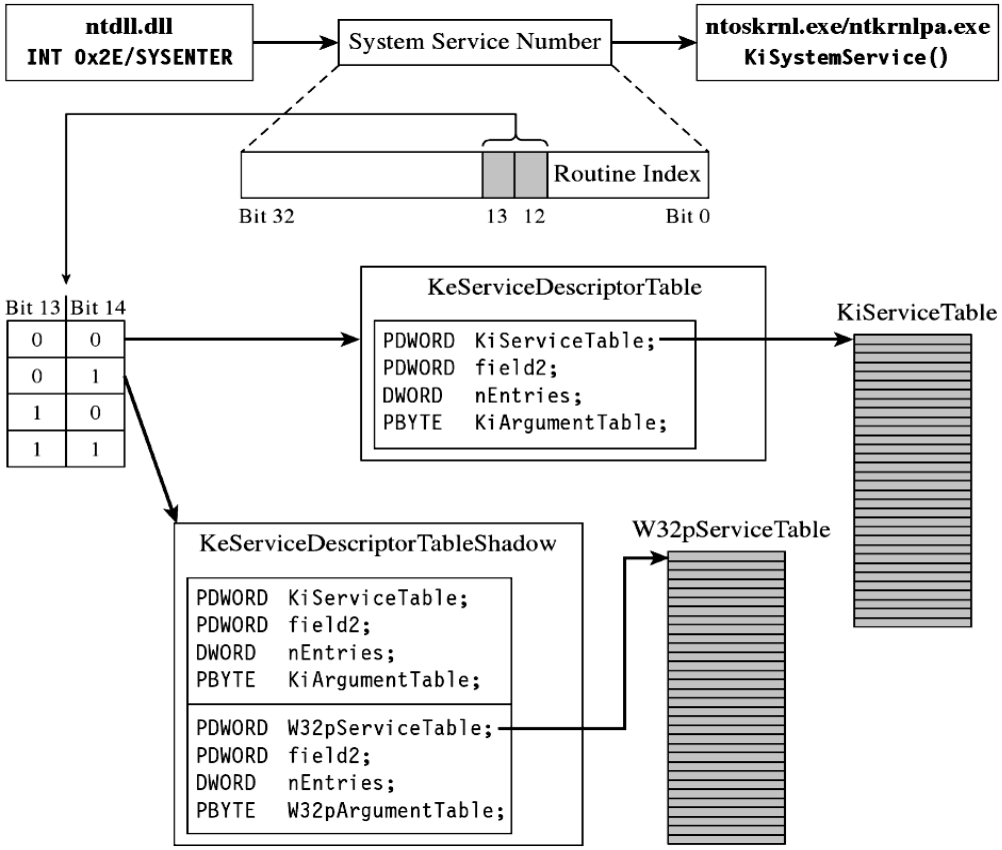
```
kd> x nt!KiSystemService
8264c24e  nt!KiSystemService = <no type information>
kd>u 8264c24e
nt!KiSystemService
...
8264c2cd  e9dd000000   jmp nt!KiFastCallEntry+0x8f<8264c3af>
```

در مورد دستور INT 0x2E نیز روند مشابه دستور SYSENTER است. یعنی قبل از فراخوانی باید عدد سرویس سیستمی در رجیستر EAX ذخیره شود. در مورد جزئیات اجرای این فرآیند، در ادامه به طور مختصر شرح خواهیم داد.

۵- جداول توزیع سرویس‌های سیستمی

صرف‌نظر از این که یک کد در مد کاربر با استفاده از دستور INT 0x2E اجرا می‌شود یا دستور SYSENTER، نتیجه‌ی نهایی یکسان خواهد بود. یعنی در هر دو مورد، در نهایت توزیع‌کننده‌ی سرویس‌های سیستمی هسته فراخوانی می‌شود. از عدد سرویس سیستمی برای پیدا کردن مدخل در یک *جدول جستجو*[1] استفاده می‌شود. عدد سرویس سیستمی یک عدد ۳۲ بیتی است که ۱۲ بیت اول آن نشان می‌دهد که در نهایت کدام سرویس سیستمی فراخوانی می‌شود. بیت‌های ۱۲ و ۱۳ آن یکی از چهار حالت ممکن برای *جداول توزیع سرویس‌های سیستمی*[2] (SSDT) را نشان می‌دهد. برای درک بیشتر موضوع، به شکل زیر توجه نمایید.

[1] Lookup table
[2] System Service Dispatch Tables



اگر چه چهار حالت ممکن برای جداول توزیع سرویس وجود دارد، اما به نظر می‌رسد که تنها دو جدول توزیع سرویس وجود دارد که نمادهای قابل مشاهده دارد. می‌توانیم این مطلب را با استفاده از دستور زیر در برنامه‌ی kd.exe مشاهده کنیم:

```
kd> dt nt!*descriptortable* -v
Enumerating symbols matching nt!*descriptortable*
Address   Size Symbol
81939900   000 nt!KeServiceDescriptorTableShadow (no type info)
819398c0   000 nt!KeServiceDescriptorTable (no type info)
```

فقط یکی از این دو نماد یعنی KeServieDescriptorTable توسط ntoskrnl.exe صادر می‌شود. بقیه‌ی جداول تنها در محدوده‌ی اجرایی[1] قابل مشاهده خواهند بود.

در صورتی که مقدار بیت‌های ۱۲ و ۱۳ برابر با 0x00 باشد (یعنی محدوده‌ی عدد سرویس سیستمی بین 0x0000 تا 0x0FFF است)، آن‌گاه از keServiceDescriptorTable استفاده خواهد شد. اما در صورتی که مقدار این دو بیت

[1] Executive

برابر با 0x01 باشد (یعنی محدوده‌ی عدد سرویس سیستمی در بین 0x1000 تا 0x1FFF است)، آن‌گاه از KeServiceDescriptorShadow استفاده خواهد شد. محدوده‌ی 0x2000-0x2FFF و 0x3000-0x3FFF به هیچ جدول توصیف‌کننده‌ی سرویسی نسبت داده نشده است.

این دو جدول توصیف‌کننده‌ی سرویس شامل زیرساختارهایی هستند که با نام *جداول سرویس‌های سیستم*[1] (SST) شناخته می‌شوند. یک SST یک جدول جستجوی آدرس است که در زبان C با چنین ساختاری تعریف می‌شود:

```
typedef struct _SYSTEM_SERVICE_TABLE
{
   PDWORD  serviceTable;  //array of function pointers
   PDWORD  field2; //not used in Windows free build
   DWORD   nEntries;  //number of function pointers in SSDT
   PBYTE   argumentTable; //array of byte counts
}SYSTEM_SERVICE_TABLE;
```

متغیر serviceTable یک اشاره‌گر به عنصر اول یک آرایه از آدرس‌های خطی است که هر آدرس یک نقطه‌ی ورودی به یک روتین از فضای هسته است. این آرایه با نام SSDT شناخته می‌شود. SSDT مشابه IVT در حالت واقعی است، با این تفاوت که SSDT یک ساختمان داده‌های مختص ویندوز است.

متغیر nEntries، تعداد عناصر موجود در SSDT را مشخص می‌کند. متغیر argumentTable، یک اشاره‌گر به عنصر اول آرایه‌ای از بایت‌هاست که هر بایت نماینده‌ی مقدار فضای اختصاص یافته برای پارامترهای تابعی است که روتین SSDT را فراخوانی کرده است. به این آرایه *جدول پارامترهای سرویس سیستم*[2] (SSPT) گفته می‌شود.

۱۶ بایت اول جدول KeServiceDescriptorTable یک ساختار SST است که SSDT را برای APIهای بومی ویندوز توصیف می‌کند. در واقع، این همان ساختمان داده‌های اصلی است که دنبال آن می‌گشتیم و شامل آدرس توابع API بومی هسته می‌باشد. جدول SSDT در سیستم‌های عامل قدیمی‌تر از ویندوز 7 شامل ۴۰۱ روتین بوده است (nEntries = 0x191).

```
kd> dds KeServiceDescriptorTable L4
819398c0  8187a890 nt!KiServiceTable
819398c4  00000000
819398c8  00000191
819398cc  8187aeb0 nt!KiArgumentTable
```

[1] System Service Tables
[2] System Service Parameter Table

۳۲ بایت اول ساختار KeServiceDescriptorTableShadow شامل دو ساختار SST است. SST اول، تنها یک کپی از KeServiceDescriptorTable است و SST دوم، SSDT را برای روتین‌های کاربر و GDI که توسط درایور مد هسته‌ی win32k.sys پیاده‌سازی شده‌اند، توصیف می‌کند. در واقع، تمام این‌ها توابعی هستند که گرافیک ویندوز را پشتیبانی می‌کنند.

۶- شناسایی توابع بومی ویندوز

اکنون ما می‌دانیم SSDT مربوط به توابع بومی ویندوز در کجا قرار دارد و چه مقدار حافظه می‌گیرد. در شکل زیر یک رونوشت خلاصه شده از آن گرفته‌ایم:

```
kd> dps KiServiceTable L191
8187a890  819c5891 nt!NtAcceptConnectPort
8187a894  818a5bff nt!NtAccessCheck
8187a898  819dd679 nt!NtAccessCheckAndAuditAlarm
...
```

یکی از نکات قابل توجه این است که تمامی توابع با پیشوند Nt شروع شده‌اند. به همین دلیل، اغلب به آن‌ها توابع Nt*( ) هم گفته می‌شود.

اما آیا کد مد کاربر به تمام ۴۰۱ روتین بومی دسترسی دارد؟ برای پاسخ به این سوال باید بدانیم که ntdll.dll چه تعداد تابع را صادر می‌کند. با استفاده از برنامه‌ی dumpbin.exe می‌توان متوجه شد که این فایل ۱۹۸۱ تابع را صادر می‌کند که تنها ۴۰۷ مورد از این توابع، توابع Nt*( ) هستند. توابعی در ntdll.dll وجود دارد که به طور کامل در فضای کاربر پیاده‌سازی شده‌اند. یکی از این توابع، تابع NtCurrentTeb() می‌باشد که دی‌اسمبل شده‌ی آن در شکل زیر نمایش داده شده است:

```
> uf ntdll!NtCurrentTeb
ntdll!NtCurrentTeb:
mov     eax,dword ptr fs:[00000018h]
ret
```

این کد نکته‌ی جالب توجهی را نمایش می‌دهد. از این کد می‌توان فهمید که بدون دانستن کد اسمبلی می‌توان به بلاک‌های اجرایی نخ‌ها در برنامه دسترسی داشت.

۶-۱- فراخوانی‌های Nt*( ) در برابر Zw*( )

اگر به رونوشت گرفته شده از ntdll.dll نگاهی بیندازیم، متوجه می‌شویم که به جز چند استثنا از هر تابع دو نسخه موجود است.

```
NtAcceptConnectPort    ZwAcceptConnectPort
NtAccessCheck  ZwAccessCheck
NtAccessCheckAndAuditAlarm ZwAccessCheckAndAuditAlarm
NtAccessCheckByType    ZwAccessCheckByType
......
```

یک تابع Nt*() با یک تابع Zw*() معادل است. برای مثال، تابع NtCreateToken() با تابع ZwCreateToken( ) جفت شده است. اما چرا این اتفاق افتاده است؟

در واقع، از دیدگاه مد کاربر هیچ تفاوتی بین این دو تابع وجود ندارد. هر دو در نهایت به یک کد ختم می‌شوند. برای مثال در فراخوانی توابع NtCreateProcess( ) و ZwCreateProcess( ) هر دو به یک کد مشابه ختم می‌شوند که با ابزار CDB.exe می‌توان آن را مشاهده کرد:

```
> x ntdll!NtCreateProcess
76e04ae0   ntdll!NtCreateProcess = <no type information>
> x ntdll!ZwCreateProcess
76e04ae0   ntdll!ZwCreateProcess = <no type information>
```

با توجه به شکل بالا می‌توان مشاهده کرد که در مد کاربر، هر دو کد در یک مکان حافظه مقیم هستند. اما این موضوع در مد هسته متفاوت است. برای بررسی در مد هسته ابتدا به سراغ بررسی پیاده‌سازی تابع NtReadFile() می‌رویم.

```
kd> u nt!NtReadFile
nt!NtReadFile:
81a04f31 6a4c            push    4Ch
81a04f33 68f0b08581      push    offset nt! ?? ::FNODOBFM::'string'+0x2060
81a04f38 e84303e5ff      call    nt!_SEH_prolog4 (81855280)
81a04f3d 33f6            xor     esi,esi
81a04f3f 8975dc          mov     dword ptr [ebp-24h],esi
81a04f42 8975d0          mov     dword ptr [ebp-30h],esi
81a04f45 8975ac          mov     dword ptr [ebp-54h],esi
81a04f48 8975b0          mov     dword ptr [ebp-50h],esi
...
```

اکنون به سراغ کد دی‌اسمبل شده‌ی تابع ZwReadFile() می‌رویم.

```
kd> u nt!ZwReadFile
nt!ZwReadFile:
81863400 b802010000      mov     eax,111h
81863405 8d542404        lea     edx,[esp+4]
81863409 9c              pushfd
8186340a 6a08            push    8
8186340c e89d130000      call    nt!KiSystemService (818647ae)
81863411 c22400          ret     24h
```

توجه کنید که از پیشوند !nt استفاده کرده‌ایم تا مطمئن شویم که از نمادهای موجود در حافظه‌ی تصویری ntoskrnl.exe استفاده کرده‌ایم. همان‌طور که می‌بینید، فراخوانی روتین ZwReadFile( ) در مد هسته مشابه فراخوانی روتین NtReadFile( ) نیست.

اگر به کد اسمبلی ZwReadFile( ) نگاهی بیندازیم، متوجه می‌شویم که این روتین مقدار عدد سرویس سیستمی متناظر با رویه را در رجیستر EAX بارگذاری می‌کند. سپس، EDX را به عنوان اشاره‌گر به پشته تنظیم کرده تا از طریق آن بتواند آرگومان‌ها را در فراخوانی‌های سیستمی کپی کند. سرانجام، روتین توزیع‌کننده‌ی سرویس سیستمی را فراخوانی می‌کند.

اما در مورد NtReadFile( ) مستقیما فراخوانی سیستمی را انجام می‌دهیم و آن را اجرا می‌کنیم. در واقع، در این حالت یک پرش از یک رویه‌ی مد هسته به رویه‌ی دیگری داریم. در حین این فرآیند، عملیات بررسی پارامتر و ارزیابی حق دسترسی بسیار کمی انجام می‌شود. اما در مورد ZwReadFile( ) چون از طریق KiSystemService( ) به سمت فراخوانی سیستمی می‌رویم، مد قبلی (یعنی مدی که دستور با آن فراخوانی شده است) به مد هسته تنظیم می‌شود. بنابراین، بررسی پارامتر و ارزیابی حق دسترسی با تنظیمات درست روی مد قبلی اجرا می‌شود. به عبارت دیگر، فراخوانی روتین Zw*( ) از مد هسته ترجیح داده می‌شود، چرا که تضمین می‌کند که اطلاعات رد و بدل شده از این کانال در حالت مناسبی هستند. مایکروسافت در توضیح روتین‌های NtXxx نوشته است: مجموعه‌ای از روتین‌های مورد استفاده توسط مؤلفه‌های مد کاربر سیستم عامل که برای ارتباط با مد هسته ایجاد شده‌اند. درایورها نباید این روتین‌ها را فراخوانی کنند. در واقع، درایورها برای انجام عملیات مشابه خود باید از روتین مشابه ZwXxx استفاده کنند.

۷- چرخه‌ی عمر یک فراخوانی سیستمی

تا این‌جا ما به قطعه‌های پازل به صورت مجزا نگاه کرده‌ایم. اکنون می‌خواهیم با دنبال کردن اجرای یک فراخوانی سیستمی، زمانی که یک برنامه‌ی مد کاربر یک روتین مد هسته را فراخوانی می‌کند، این قطعه‌ها را به هم بچسبانیم.

در این مثال می‌خواهیم بفهیم زمانی که کنترل یک برنامه به یک فراخوانی سیستمی پیاده‌سازی شده در ntoskrnl.exe می‌رود، چه اتفاقی می‌افتد. در واقع، در این مثال می‌خواهیم بدانیم با فراخوانی تابع بومی WriteFile() چه اتفاقی می‌افتد. امضای این تابع در مستندات SDK ویندوز به شرح زیر است:

```
BOOL WINAPI WriteFile
(
  __in         HANDLE hFile,
  __in         LPCVOID lpBuffer,
  __in         DWORD nNumberOfBytesToWrite,
  __out_opt    LPDWORD lpNumberOfBytesWritten,
  __inout_opt  LPOVERLAPPED lpOverlapped
);
```

برای این کار، پردازه‌ی Winlogon.exe را با استفاده از ابزار cdb.exe مورد بررسی قرار می‌دهیم. با استفاده از فایل دسته‌ای زیر می‌توانیم برای برنامه‌ی Winlogon.exe یک *نشست اشکال‌زدایی*[1] را آغاز کنیم.

```
set PATH=%PATH%;C:\Program Files\Debugging Tools for Windows
set DBG_OPTIONS=-v
set DBG_LOGFILE=-logo .\CdbgLogFile.txt
set DBG_SYMBOLS=-y SRV*C:\Symbols*http://msdl.microsoft.com/download/symbols
CDB.exe %DBG_LOGFILE% %DBG_SYMBOLS% .\winlogon.exe
```

اکنون با استفاده از دستور uf در ابزار kd.exe، کد اسمبلی روتین WriteFile() را مورد بررسی قرار می‌دهیم.

```
> uf WriteFile
kernel32!WriteFile+0x1f0:
7655dcfa ff75e4          push    dword ptr [ebp-1Ch]
7655dcfd e88ae80300      call    kernel32!BaseSetLastNTError (7659c58c)
7655dd02 33c0            xor     eax,eax
...
7655dd42 ff15f8115576    call    dword ptr [kernel32!_imp__NtWriteFile
                                 (765511f8)]
...
```

[1] Debugging session

اولین چیزی که در خروجی خلاصه شده‌ی بالا دیده می‌شود، این است که این تابع در kernel32.dll پیاده‌سازی شده است. همچنین، خط آخر نشان می‌دهد که این تابع، روتین موجود در آدرس 0x765511f8 که در یک جدول جستجو نگهداری می‌شود را فراخوانی می‌کند.

```
> dps 765511f8 L3
765511f8  77bb9278 ntdll!NtWriteFile
765511fc  77becc6d ntdll!ZwCancelIoFileEx
76551200  77b78908 ntdll!ZwReadFileScatter
```

بنابراین، کد WriteFile() که در kernel32.dll موجود است، سرانجام به فراخوانی یک تابع موجود در ntdll.dll ختم می‌شود.

```
> uf ntdll!NtWriteFile
ntdll!NtWriteFile:
77bb9278 b863010000      mov     eax,18Ch
77bb927d ba0003fe7f      mov     edx,offset SharedUserData!SystemCallStub
                                 (7ffe0300)
77bb9282 ff12            call    dword ptr [edx]
77bb9284 c22400          ret     24h
```

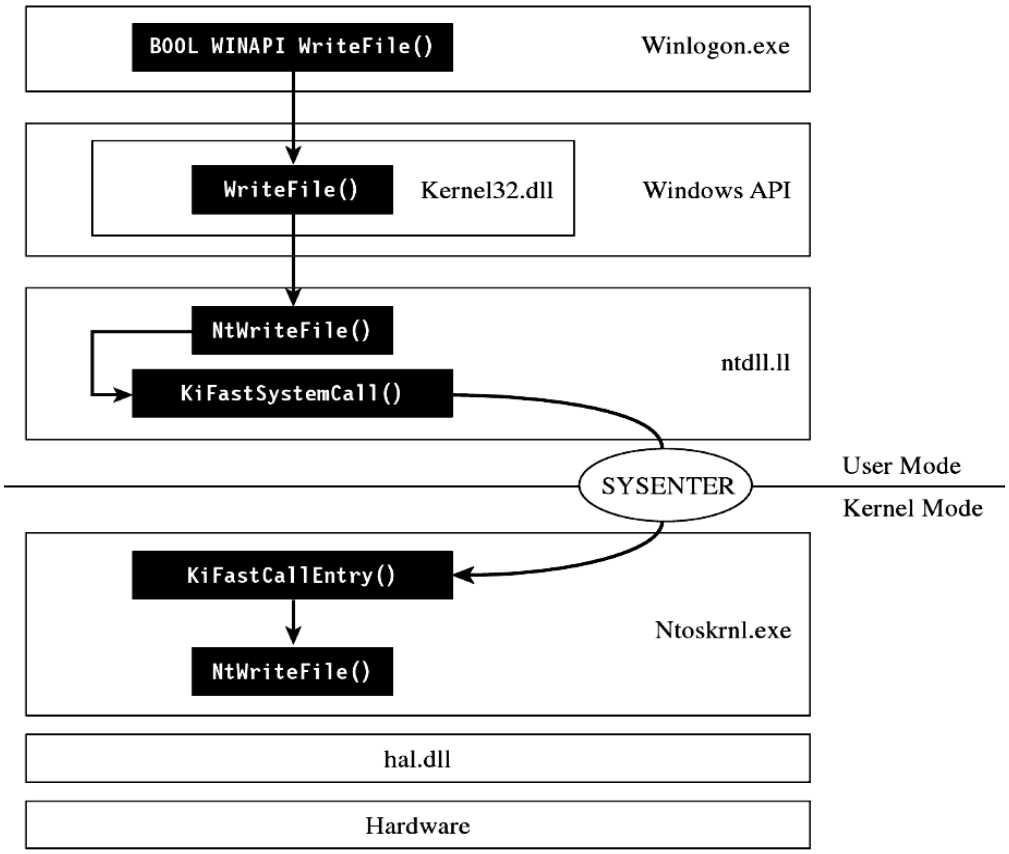
همان‌طور که می‌بینید، خروجی بالا پیاده‌سازی واقعی تابع API بومی NtWriteFile() نیست، بلکه فقط یک روتین کوچک در ntdll.dll است که شرایط استفاده از تابع NtWriteFile() واقعی موجود در ntoskrnl.exe را فراهم می‌کند. توجه کنید که مقدار عدد سرویس سیستمی (یعنی 0x18C) برای فراخوانی بومی NtWriteFile( ) در رجیستر EAX بارگذاری شده است.

```
> dps 7ffe0300
7ffe0300  77da0f30 ntdll!KiFastSystemCall
7ffe0304  77da0f34 ntdll!KiFastSystemCallRet
7ffe0308  00000000

> uf ntdll!KiFastSystemCall
ntdll!KiFastSystemCall:
77da0f30 8bd4            mov edx,esp
77da0f32 0f34            sysenter
77da0f34 c3              ret
```

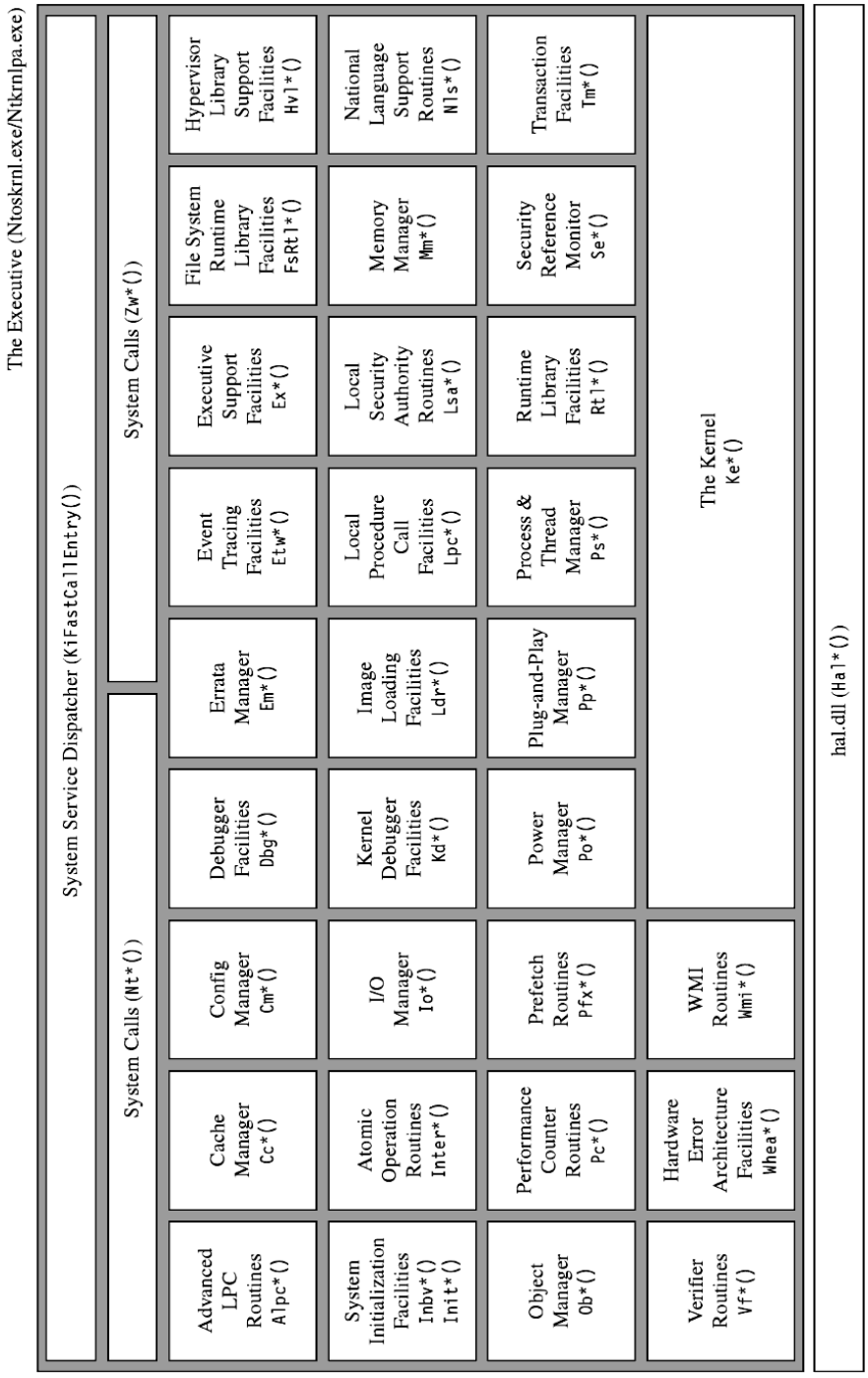
همان‌طور که قبلا گفتیم، دستور SYSENTER برنامه را وادار می‌کند تا کنترل را به روتین KiFastCallEntry() در ntoskrnl.exe ببرد که این کار منجر به فراخوانی رویه‌ی بومی NtWriteFile() می‌شود. شکل زیر این روند را به

طور کامل برای تابع WriteFile( ) نشان می‌دهد:



۸- سایر روتین‌های مد هسته

علاوه بر توابع API بومی (که شامل حدود ۴۰۰ فراخوانی سیستمی مختلف می‌شود)، هزاران روتین دیگر می‌توانند توسط مجری صادر شوند. Ntoskrnl.exe تعداد ۲۱۸۴ روتین را صادر می‌کند. بسیاری از این فراخوانی‌های سیستم می‌توانند به عنوان یک زیرگروه در یک زیرسیستم خاص قرار گیرند.



اجزایی که در شکل فوق نشان داده شده است، تمام زیرسیستم‌های اجرایی نیست. بخشی از آن‌ها به تنهایی گروهی از توابع مورد حمایت را نشان می‌دهد. در حقیقت، در این شکل تلاش بر این بوده که ترتیبی از زیرسیستم‌ها به نمایش گذاشته شود که نمایانگر نقش و عملکرد آنان باشد. همچنین، زیرسیستم‌های رسمی با عبارت Manager برچسب‌گذاری شده‌اند.

برای پیوستگی بین روتین‌های سیستمی، مایکروسافت تصمیم گرفت که یک مدل نام‌گذاری برای تمام توابع سیستمی (نه فقط توابع صادر شده از ntoskrnl.exe) ایجاد کند. بنابراین، یک قرارداد خاص برای نام‌گذاری شناسه‌ها در مایکروسافت اتخاذ گردید.

Prefix-Operation-Object –۱-۸

چند کاراکتر ابتدای نام‌ها یک پیشوند است که نشان می‌دهد هر کدام از روتین‌ها متعلق به کدام زیرسیستم است و چه عملکردی دارد. چند کاراکتر انتهایی هر نام (البته نه همیشه) نشان می‌دهد که چه نوع شیای قرار است دستکاری شود. کلمه‌ی بین پیشوند و شئ یک فعل است که نشان می‌دهد چه اتفاقی قرار است بی‌افتد. برای مثال، در فایل ntoskrnl.exe یک روتین با نام MmPageEntireDrive( ) وجود دارد که برای مدیریت حافظه پیاده‌سازی شده است. این روتین باعث می‌شود که تمام کدهای درایور و داده‌ها قابل صفحه‌بندی شوند. در جدول زیر فهرستی از پیشوندهای توابع و اجزای مد هسته مرتبط با آن‌ها آورده شده است.

Prefix	Kernel-Mode Component	Description
Alpc	Advanced LPC routines	Passes local messages between client and server software
Cc	Cache manager	Implements caching for all file system drivers
Cm	Configuration manager	Implements the Windows registry
Dbg	Debugging facilities	Implements break points, symbol loading, and debug output
Em	Errata manager	Offers a way to accommodate noncompliant hardware
Etw	Event tracing facilities	Helper routines for tracing events system-wide
Ex	Executive support facilities	Synchronization services and heap management
FsRtl	File system runtime library	Used by file system drivers and file system filter drivers
Hal	Hardware abstraction layer	Insulates the operating system and drivers from the hardware
Hvl	Hypervisor library routines	Kernel support for virtual machine operation
Invb	System initialization routines	Bootstrap video routines
Init	System initialization routines	Controls how the operating system starts up
Inter-locked	Atomic operation facilities	Implements thread-safe variable manipulation
Io	Input/output manager	Controls communication with kernel-mode drivers
Kd	Kernel debugger facilities	Reports on, and manipulates, the state of the kernel debugger
Ke	The kernel proper	Implements low-level thread scheduling and synchronization
Ki	Internal kernel routines	Routines that support interrupt handling and spin locks
Ldr	Image loading facilities	Support the loading of executables into memory
Lpc	Local procedure call facilities	An IPC mechanism for local software compo-nents
Lsa	Local security authentication	Manages user account rights
Mm	Memory manager	Implements the system's virtual address space
Nls	Native language support	Kernel support for multilingual environments
Nt	Native API calls	System call interface (the internal version)
Ob	Object manager	Implements an object model that covers all system resources
Pcw	Performance counter routines	Routines that allow performance data to be collected
Pf	Logical prefetcher	Optimizes load time for applications
Po	Power manager	Handles the creation and propagation of power events
Pp	Plug-and-play manager	Supports dynamically loading drivers for new hardware
Ps	Process and thread manager	Provides higher-level process/thread services
Rtl	Runtime library	General support routines for other kernel components
Se	Security reference monitor	Validates permissions at runtime when accessing objects
Tm	Transaction management	Support for the classic two-phase commit scheme
Vf	Verifier routines	Checks the integrity of kernel-mode code
Whea	Hardware error architecture	Defines a mechanism for reporting hardware errors
Wmi	Management instrumentation	Allows kernel-mode code to interact with the WMI service
Zw	Native call APIs	The safe (user-callable) versions of the Nt*() routines

۹- مستندسازی توابع API مد هسته

همان‌طور که قبلا نیز اشاره شد، مستندات توابع مد هسته ناقص هستند. بنابراین، اگر زمانی به یک تابع مد هسته برخورد کردید که آن را نمی‌شناختید، می‌توانید از منابع زیر استفاده کنید:

- مستندات رسمی
- مستندات غیر رسمی (مستنداتی غیر از مستندات مایکروسافت)
- فایل‌های سرآیند
- نمادهای اشکال‌زدایی
- دی‌اسمبل شده‌های خام

در یک سناریوی بهینه‌تر، تعریف این توابع را می‌توان در مستندات WDK[1] پیدا کرد. تعداد بی‌شماری از توابع مد هسته در فایل راهنمای مربوط به WDK در قسمت Driver Support Routines node می‌توان یافت. همچنین، سایت MSDN را به صورت برخط بر روی http://msdn.microsoft.com می‌توان مشاهده کرد.

۱۰- نتیجه‌گیری

در این مقاله، توابع API بومی سیستم عامل ویندوز که توسط کپی اجرایی هسته صادر می‌گردند، معرفی شدند. همچنین، نحوه‌ی فراخوانی این توابع در مد کاربر بیان گردید. به این منظور، ساختارهایی مانند IDT و SSDT که به ترتیب برای مدیریت وقفه‌ها و توابع API بومی ویندوز می‌باشند، تشریح گردیدند.

مراجع

[1] Bill Blunden : The Rootkit Arsenal Escape and Evasion in the Dark Corners of the System, Second Edition, 1969

[2] Interrupt descriptor table [Online]. Available : http://en.wikipedia.org/wiki/Interrupt_descriptor_table

[3] 4-Gigabyte Tuning[Online]. Available : http://msdn.microsoft.com/en-us/library/windows/desktop/bb613473(v=vs.85).aspx

[4] User Mode and Kernel Mode [Online]. Available :

[1] Windows Driver Kit

http://msdn.microsoft.com/en-us/library/windows/hardware/ff554836(v=vs.85).aspx

[5] Windows Programming: Kernel Mode vs User Mode [Online]. Available : http://en.wikibooks.org/wiki/Windows_Programming/User_Mode_vs_Kernel_Mode

[6] Native API . [Online]. Available : http://en.wikipedia.org/wiki/Native_API

[7] http://technet.microsoft.com/en-us/sysinternals/bb897447.aspx

[8] Aplication Programming Interface (API) [Online]. Available : http://en.wikipedia.org/wiki/Application_programming_interface

[9] Interupt Descriptor Table . [Online]. Available : http://en.wikipedia.org/wiki/Interrupt_descriptor_table