2007

اکسپلویت کردن WIN32 KERNEL

سیاوش صفایی

caserpent@gmail.com

Special Thanx For:
LUCIFER, IM4N , Immo2tal And All Other Friends!

CaC4
8/3/2007

مقدمه:

تقریبا 10 سال پیش Solaris Designer پیغامی را در لیست Bugtraq قرار داد که در آن کد exploit و جزییات مربوط به سرریز بافر از راه دور(Remote Buffer Overflow) را در مورد محصولی بنام Website v.1.1e برای ویندوز NT، به نمایش می گذاشت. احتمالا این اولین exploit سرریز بافر برای ویندوز بود که منتشر می شد.

هم اکنون 8 سال میگذرد و تمام روشها و تکنیکهای ممکن در مورد exploit کردن ویندوز، بصورت عمیق و با جزییات فراوان، مورد بررسی قرار می گیرند. اما یک موضوع که هنوز بصورت عمومی منتشر نشده است، exploit کردن از راه دور win32 kernel میباشد. البته تعدادی آسیب پذیری در رابطه با kernel منتشر گشته، اما هیچ کد exploit مربوط به آن بصورت عمومی در سایتها قرار نگرفته است.

با انتشار ویندوز XP SP2 و استفاده وسیع از دیوارهای آتشین شخصی، بسیاری از شرکت های نرم افزاری و امنیتی، خواستار ایجاد سیستم های ایمن می باشند.

در این مقاله، در مورد آسیب های بر پایه kernel و exploit کردن آنها از راه دور صحبت خواهد شد. مثال مورد استفاده، نقصی در دیوارهای آتشین شخصی شرکت Symantec میباشد. این نقص بدلیل مدیریت غیر صحیح پاسخ های DNS، بوجود آمده است. این نقص مدت زیادی است که برطرف شده است، اما در اینجا به منظور شرح تعداد خاصی از موانع مربوط به لایه های ارتباطات که سد راه exploit کردن دیوارهای آتشین host-based قرار می گیرند و باید برداشته شوند، مورد استفاده قرار می گیرد. دو مثال از shellcode مورد بررسی قرار می گیرد: مثال اول یک kernel-loader است که به ما اجازه اتصال و اجرای هر گونه کد دلخواه را می دهد. مثال دوم بطور کامل در سطح kernel عمل می کند.یک ثبت کننده کلید نصب می گردد و بافر کلیدهای ثبت شده از یک سیستم دور دریافت می شود. برای فهم این مقاله، آشنایی به زبان اسمبلی x86 و تجربه قبلی در زمینه exploit کردن win32 الزامی است.

Kernel در محیط کاربر

معماری x86 از چهار حلقه پشتیبانی می کند که سطوح انحصاری نیز نامیده می شوند. ویندوز NT از دو تا از این حلقه ها استفاده می کند، به این دلیل که سیستم عامل NT قابلیت اجرا بر روی معماری هایی که از همه چهار سطح پشتیبانی نمی کنند را، داشته باشد.

کدهای مربوط به کاربر، مانند برنامه های کاربردی و سرویس های سیستم در حلقه انحصاری 3 اجرا می شوند. فرآیندهای کاربر، فقط می توانند به 2 گیگابایت حافظه تخصیص داده شده به آنها، دسترسی داشته باشند.(نیمه بالایی 4 گیگابایت، هنگامی مورد دسترسی فرآیند قرار میگیرد که در حالت انحصاری اجرا شود)؛ کد مربوط به کاربر قابل صفحه بندی است و زمینه آن(Context) میتواند تغییر کند.

کد مربوط به kernel در سطح انحصاری 0 اجرا میشود. لایه انتزاعی سخت افزار(HAL) ، درایورها، ورودی/خروجی، مدیریت حافظه و رابط گرافیکی، همه مثال هایی از کد اجرایی در حلقه 0 است. کدی که در حلقه 0 اجرا میشود، از کلیه امتیازات سیستم بهره میبرد. دسترسی کامل به حافظه و قابلیت اجرای دستور العملهای انحصاری، همگی امکان پذیر میشود.

Native API

بواسطه طراحی، فرآیندهای کاربر نمی توانند بطور دلخواه سطوح انحصاری خود را تغییر دهند.

در بسیاری ز مواقع، یک کار مربوط به کاربر، نمی تواند بدون قدرت عملکردی از سطح kernel، به اتمام برسد.در اینجاست که Native API وارد عمل میشود.Native API مجموعه ای از توابع داخلی است که در حیطه مد kernel اجرا میشوند. Native API در واقع یک روش "امن" برای جهت فراخوانی سرویس های مد kernel از حیطه کاری کاربر می باشد.

یک برنامه در مد کاربر، می تواند توابع Native API را که از NTDLL.DLL صادر میگردند، فراخوانی کند.NTDLL.DLL تعداد زیادی از توابع را صادر می کند که به عنوان پوششی برای تابع kernel مربوط، عمل میکنند. اگر یکی از این توابع را disassembleکنید، یک خروجی مانند زیر نشان داده میشود:

Windows 2000:
mov eax, 0x0000002f
lea edx, [esp+04]
int 0x2e

هر تابع Native API که توسط NTDLL صادر میشود، به یک stub تجزیه میگردد که اجرا را به مد kernel انتقال میدهد. یک ثبات با یک شماره شاخص بارگذاری میشود که به جدول سرویسهای سیستم اشاره میکند و متعاقبا با آفست NTOSKRNL دسترسی پیدا میکند که نمایانگر تابع مورد نیاز است.

Windows XP:
mov eax, 0x0000002f
mov edx, 7ffe0300
call edx

At offset 0x7ffe0300:

```
mov edx, esp
sysenter
ret
```

در ویندوز xp ، عملیات تا حدی متفاوت است، در صورتی که کامپیوتر مورد بحث Pentium II یا بالاتر باشد. ویندوز xp برای تغییر مد کاری از (به) kernel از جفت دستورالعمل SYSENTER/SYSEXIT استفاده میکند. این موضوع گره ای در ایجاد shellcode بوجود می آورد که در مورد آن بیشتر صحبت خواهد شد. به منظور ایجاد shellcode در مد kernel می باست API در سطح کاربر را فراموش کرد و فقظ از توابع Native API موجود در kernel استفاده نمود.

جزییات صفحه آبی مرگ

فرض کنید که شما یک آسیب پذیری یافته اید. داده های packet خود را به سیستم مورد نظر(remote) می فرستید و با صفحه آبی(Blue Screen) مواجه میشوید.در این حالت، این موضوع نشانه خوبی است. اولیت قدم در exploit کردن یک آسیب پذیری بر پایه kernel، درک این مساله است که در پشت صحنه "صفحه آبی مرگ" چه میگذرد.

هر وقت که با BSOD مواجه میشوید، تابع KeBugCheckEx که Native میباشد، فراخوانی شده است.یک bugcheck به دو روش ممکن است ایجاد شود:

1 -توسط فرستنده خطای kernel
2 -KeBugCheckEx مستقیما بعد از بررسی خطا، فراخوانی شده است.

زنجیره رخدادهای صورت گرفته در جهت اداره خطای kernel به شرح زیر است:
وقتی که یک exception ایجاد میشود،kernel کنترل را توسط ورودی های توابع مختلف(KiTrapXX) در حیطه جدول توضیح وقفه(IDT) بدست می آورد. این توابع تشکیل دهنده Trap Handler سطح نخست هستند.

Trap Handler ممکن است خودش به exception رسیدگی کند، یک کنترل کننده exception پیدا کند و exception را به آن گذر دهد و یا اگر قابل اداره کردن نباشد، KeBugCheckEx را فراخوانی خواهد کرد.

در تمام حالات، ما باید Trap Frame را دریافت کنیم تا بفهمیم که Exception کجا و چرا رخ داده است.

Trap Frame مانند ساختار CONTEXT است.در این ساختار، می توانیم حالت تمام ثبات ها و مقدار اشاره گر دستور العمل(IP) را از آدرسی که Exception صورت گرفته است، پیدا کنیم. در صورت استفاده از برنامه SoftICE باید بطور دستی، پارامترهای پشته قبلی را بیابیم، اما Windbg عملکرد بهتری در زمینه شناخت ساختار ارائه میدهد.

اگر کامپیوتر شما به گونه ای تنظیم شده باشد که فایلهای ثبت حافظه را ذخیره کند، وقتی BSOD رخ میدهد، این فایل ثبت بطور پیش فرض در %SystemRoot%\MEMORY.DMP ذخیره میگردد. Windbg را اجرا کنید و گزینه "Open Crash Dump"را برای بارگذاری فایل ذخیره شده، انتخاب کنید.در مثال زیر، KeBugCheckEx مستقیما توسط Trap Handler فراخوانی شده است. پس از بارگذاری فایل ثبت، خروجی زیر نشان داده میشود:

```
*******************************************************************************
*                                                                             *
*                        Bugcheck Analysis                                    *
*                                                                             *
*******************************************************************************

Use !analyze -v to get detailed debugging information.

BugCheck D1, {41414141, 2, 0, 41414141}
Probably caused by : ntoskrnl.exe ( nt!KiTrap0E+2ad )
```

حال برای نمایش پشته از دستور **kv** استفاده میکنیم:

```
kd> !kv
ChildEBP RetAddr  Args to Child
80541980 804dce53 0000000a 41414141 00000002 nt!KeBugCheckEx+0x19 (FPO: [Non-
Fpo])
80541980 41414141 0000000a 41414141 00000002 nt!KiTrap0E+0x2ad (FPO: [0,0] TrapFrame @
8054199c)
WARNING: Frame IP not in any known module. Following frames may be wrong.
80541a0c 90909090 90909090 90909090 90909090 0x41414141
00000246 00000000 00000000 00000000 00000000 0x90909090
```

Windbg نشان میدهد که KeBugCheckEx توسط روتین تله (trap routine) KiTrap0E فراخوانی شده است و Trap Frame در آدرس 0x8054199C می باشد.

حال محتویات trap frame را بوسیله دستور "trap [address]" میابیم :

```
kd> .trap 8054199c
ErrCode = 00000000
eax=00000000 ebx=80dd3da8 ecx=00000000 edx=00000000 esi=f6dbfb6d edi=80e74c8b
eip=41414141 esp=80541a10 ebp=44444444 iopl=0         nv up ei pl zr na po nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00000246
41414141 ??              ???
```

اکنون میتوانیم وضعیت تمام ثبات ها را در زمانی که exception صورت گرفته است، مشاهده کنیم. همچنین میتوانیم نواحی حافظه را تا نقطه مشخصی نمایش دهیم. دقت کنید که مقدار ثبات EIP (اشاره گر دستور العمل) 0x414141 است(داده تعریف شده توسط کاربر). حالا می توانیم روند اجرا را به دلخواه خود تغییر دهیم.

در این مثال، داده ذخیره شده در ثبات ESP به شرح زیر است:

```
kd> d 80541a10
80541a10  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
80541a20  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
80541a30  90 90 90 2e 60 eb 03 5d-eb 05 e8 f8 ff ff ff 83  ....`..]........
80541a40  c5 1a 90 90 90 8b f5 8b-fe 33 c9 66 b9 d2 02 ac  .........3.f....
80541a50  3c 2e 74 03 34 80 aa e2-f6 0b b5 b8 70 5f 7f 2d  <.t.4.......p_.-
80541a60  19 2d c8 01 b8 cd da 10-80 f5 77 68 94 80 80 80  .-........wh....
80541a70  49 25 ac 2e 3f 5f e3 90-5f 87 78 52 d7 a3 fb 51  I%..?_.._.xR...Q
80541a80  1d e4 4b b7 d8 db 0b 7b-03 6b 9b b3 49 31 8a 12  ..K....{.k..I1..
```

حال میتوانیم از طریق جایگزینی 0x414141 آدرس دیگری که یکی از دستورات JMP ESP ، CALL ESP ، PUSH ESP را اجرا می کند، مسیر اجرا را تغییر دهیم.در ادامه میتوان از هر روش دیگری که برای exploit کردن آسیب پذیری های سرریزی بکار میرود، استفاده نمود.

اگر Bugcheck توسط فرستنده Exception منتشر شده بود، trap frame، سومین پارامتری می بود که به KiDispatchException گذر داده می شد. در آن صورت، شما می بایست آدرس سومین پارامتر با با دستور trap استفاده کنید. هنگام انتخاب یک آدرس جهت تغییر روند اجرا، باید از یک آدرس ایستا در حافظه(آدرسی که همیشه یکسان باشد) استفاده نمود.

مثالهایی از shellcode

اولین مثال از shellcode یک "kernel loader" میباشد که به شما اجازه اتصال هر گونه کد کاربر و اجرای امن آن را میدهد. جالب اینجاست که در این مورد شما کنترل کامل پردازنده را در دست میگیرید و در عین حال قابلیت بازگشت به مد کاربر را نیز دارید.

مثال دوم بطور خالص از kernel استفاده میکند. این مثال یک اداره کننده وقفه صفحه کلید را تنظیم و تمام کلیدهای فشرده شده را ضبط میکند. سپس shellcode ، اداره کننده ICMP در TCPIP.SYS را دستکاری میکند تا هنگام دریافت درخواست ICMP ECHO، بافر صفحه کلید را به یک کامپیوتر دور انتقال دهد. کد این مثال کوچک است و از تعداد کمی از توابع API استفاده می کند.

"Kernel Loader"

چند تکنیک به منظور گذر دادن کد از kernel به محیط کاربر و اجرای آن وجود دارد. برای مثال می توان مستقیما EIP مربوط به یکthread در حال اجرای کاربر را، به مکانی که کد شما در آن جا قرار دارد، تغییر داد. فرآیند در حال اجرا، پس از این کار، خودش را از بین میبرد.

یک روش دیگر، استفاده از توابع RtlCreateUserThread و RtlCreateUserProcess در چارچوب NTOSKRNL می باشد؛ این توابع برای ایجاد SMSS.EXE مورد استفاده قرار میگیرند که تنها فرآیند بدون parent است و به عنوان فرزند kernel شناخته میشود. اما دو مشکل وجود دارد: یکی اینکه این توابع صادر نمیشوند و دوم اینکه آنها در بخش INIT تابع NTOSKRNL قرار دارند، یعنی هنگام اجرا، این توابع از بین میروند.

متاسفانه در ویندوز xp یک مشکل بوجود می آید. ویندوز xp برای تغییر از(به) حلقه صفر، از راهکار منطقی تر SYSEXIT و SYSENTER بهره میبرد. اگر یک تابع صادر شده از NTDLL بطور مستقیم توسط kernel فراخوانی شود، بدون شک صفحه آبی در انتظار ماست. برای رفع این مشکل، باید قطعه کدی اضافی نوشته شود تا تابع NTOSKRNL مورد نیاز را از جدول سرویس سیستم، بیابد. در اینجا ما از APC ها برای اجرای تابع مورد نظر خود(shellcode) ، در محیط کاربر، استفاده می کنیم. این شیوه فقط از توابع مستقیما صادر شده از NTOSKRNL استفاده میکند.

با فراخوانی یک APC در یک thread اجرایی در مد کاربر و در "وضعیت انتظار قابل هشدار" ، تابع فراخوانی شده باید بلادرنگ اجرا شود و نیازی به ایجاد زمینه قبلی(context) نداریم. یک thread در "وضعیت انتظار قابل هشدار" ، هر گونه thread ی است که توابع SleepEx، WaitForSingleObjectEx، SignalObjectAndWai و MsgWaitForMultipleObjectsEx را با تنظیم پرچم bAlertable به مقدار TRUE ، فراخوانی کرده باشد. این شیوه از حداقل توابع API استفاده می کند و بسیار قابل اطمینان می باشد. تمام توابعی که ما از آنها استفاده می کنیم، از NTOSKRNL صادر میشوند. اولین قدم، بدست آوردن آدرس پایه NTOSKRNL بطور دستی میباشد.برای انجام این مرحله، از تکنیکی که بعنوان mid-delta از آن

یاد میکنند، استفاده می کنیم.بدین زریق که یک اشاره گر به فضای آدرس NTOSKRNL پیدا میکنیم و یکی یکی از آن کم میکنیم تا امضای اجرایی "MZ" را بیابیم. برای بدسا آوردن اشاره گر به فضای آدرس NTOSKRL ، اولین ورودی در جدول توضیح وقفه(IDT) را پیدا میکنیم زیرا که این ورودی همیشه به مکانی در چارچوب و در حواشی NTOSKRNL اشاره می کند.

قطعه کد زیر، ابتدا جهت بدست آوردن اشاره گر حافظه، به IDT دسترسی پیدا میکند و سپس آنقدر از آن کم میکند(یک واحد یک واحد) تا آدرس پایه را پیدا کند.

```
mov esi, dword ptr ds:[0ffdff038h] ;     get address of IDT
lodsd
cdq
lodsd ; get pointer into NTOSKRNL
@base_loop:
dec eax
cmp dword ptr [eax], 00905a4dh ;       check for MZ signature
jnz @base_loop
```

روش معمول برای دریافت آدرس پایه IDT ، استفاده از دستور SIDT است. اما چون IDT همچنین توسط اشاره گری در آدرس oxFFDFF038 نیز قابل دستیابی است، در کد بالا از این اشاره گر استفاده گردید.

ممکن است متوجه شده باشید که کد بالا، ورودی معتبری از IDT دریافت نکرد. ما فقط به کلمه بالایی ورودی احتیاج داریم، زیرا کلمه پایینی میتواند بطور امن از 0-0xFFFF اجرا شود و در محدوده حافظه NTOSKRNL باقی بماند.

در مرحله بعدی، یک جدول از مقادیر در هم سازی دو بایتی برای هر یک از توابع مورد نیاز ایجاد میکنیم.رشته تابع ها، مقدار زیادی فضا در محدوده shellcode ویندوز اشغال میکنند. بنابراین راهکار مبتنی بر درهم سازی منطقی به نظر میرسد.پس از آن، هر اشاره گر تابع در یک جدول ذخیره میشود و در مکان های مختلف shellcode از طریق EBX مورد دسترسی قرار میگیرد.

قدم بعدی پیاده سازی "GetProcAddress" است. این کد در جدول صادر شده از NTOSKRNL جستجو و آدرسهای توابع مربوط را دریافت میکند. تنها تفاوت در این پیاده سازی، تابع درهم سازی است که حاوی XOR/ROR هر بایت از لیست نام جدول صدور است. ما از مقادیر درهم سازی با اندازه یک word بجای Dword استفاده میکنیم تا اندازه shellcode مینیمم گردد.

هنگامی که آدرس توابع مورد نیاز خود را یافتیم، قدم بعدی اختصاص یک بلوک حافظه جدید برای ذخیره کردن بقیه shellcode میباشد. برای این منظور، تابع ExAllocatePool را با پارامتر NonPagedPool فراخوانی میکنیم. سپس shellcode مورد نظر را در بلوک غیر صفحه بندی شده، کپی کرده و با اجرای دستور العمل ساده JMP، به این ناحیه از حافظه دسترسی پیدا میکنیم.

هنگامی exploit کردن یک Driver ، باید دقت کنیم که هم اکنون در چه IRQL (سطح درخواست وقفه) به سر میبریم. سطح درخواست وقفه، سطح اولویت سخت افزاری است که یک روتین kernel در حال اجرا در آن است. بسیاری از توابع kernel برای اجرای موفقیت آمیز، به سطح PASSIVE(0) احتیاج دارند. اگر در سطح DISPATCH(2) باشیم که برای برنامه ریزی thread وDPC مورد استفاده قرار میگیرد، باید به سطح PASSIVE برگردیم. این امر از طریق فراخوانی تابع صادر شده از HAL با نام KeLowerIrql و گذر دادن پارامتر 0 به آن، امکان پذیر است.

هنگامی قصد اتصال به یک فرآیند در حیطه کاربر را داریم، به یک اشاره گر با ساختار EPROCESS نیاز داریم. هر فرآیند دارای یک ساختار متناظر EPROCESS میباشد. میتوان با dump کردن این ساختار در Windbg اطلاعات بیشتری در مورد این ساختار بدست آورد(dt NT!_EPROCESS) .توابعی که ما از آنها استفاده میکنیم، به آفست هایی احتیاج دارند که از EPROCESS بدست می آیند. اگر بتوان یک اشاره گر به هر گونه ساختار EPROCESS بدست آورد، آن وقت قادر خواهیم بود با جستجو در آن، اشاره گرهایی با ساختارهای تمام فرآیندهای فعال، بدست آوریم.

عموما از طریق فراخوانی PsGetCurrentProcess ، اشاره گری به ساختار EPROCESS اولیه، حاصل میشود. متاسفانه هنگام exploit کردن یک درایور دور، امکان فرود در فضای آدرس "Idle" وجود دارد. فرآیند "Idle" ساختار فرآیند معتبری را باز نمی گرداند. به جای آن، میتوان از PsLookupProcessByProcessId استفاده نمود و PID فرآیند سیستم را به عنوان پارامتر به آن گذر داد. در ویندوز xp این پارامتر 4 و در ویندوز 2000 ، 8 میباشد.

```
lea ebp, [edi-4]
push ebp
push 04
call dword ptr _pslookupprocessbyprocessid ;    Get System EPROCESS
mov eax, [ebp] ; Get EPROCESS pointer
```

حال که ساختار اولیه دریافت شده است، میتوان با ساختار هر یک از فرآیندهای فعال، دسترسی پیدا کرد. در اینجا کد مورد نظر خود را در فضای آدرس LSASS تزریق میکنیم. اما هر فرآیندی که به عنوان SYSTEM در حال اجرا باشد، هدف خوبی به نظر میرسد. برای دسترسی به LSASS ، حلقه ای ایجاد میکنیم و هر

ورودی را که توسط EPROCESS+ActiveProcessLinks به آن اشاره شده باشد، جستجو میکنیم و مقایسه ای بین آدرس ModuleName و LSASS انجام میدهیم.

```
mov cl, EP_ActiveProcessLinks ; offset to ActiveProcessLinks
add eax, ecx ; get address of EPROCESS+ActiveProcessLinks
@eproc_loop:
mov eax, [eax] ; get next EPROCESS struct
mov cl, EP_ModuleName
cmp dword ptr [eax+ecx], "sasl" ; is it LSASS?
jnz @eproc_loop
```

هنگامی که مکان فرآیند LSASS را پیدا کردیم، آفست ActiveProcessLinks را از آن کم میکنیم، در نتیجه اشاره گری به شروع ساختار EPROCESS برای LSASS بدست می آید.

قدم بعدی کپی کردن shellcode خود در فضای حافظه هدف میباشد. قبل از ویندوز xp sp2 میتوانستیمshellcode را در بلوک محیط فرآیند(PEB) ذخیره کنیم؛ چون PEB همیشه در آدرس 0x7ffdf000 قرار داشت.در نسخه ویندوز xp sp2 ، PEB هر بار در محلی تصادفی قرار میگیرد. در این شرایط میتوان Shellcode را در فضای حافظه ای که بنام SharedUserData رسمیت دارد، ذخیره کرد. در آدرس 0xffdf000 ، یک ناحیه ز حافظه قابل نوشتن وجود دارد که میتوانیم shellcode را در آن ذخیره کنیم. این آدرس، از محیط کاربر در 0x7ffe000 نگاشته و بصورت فقط خواندنی علامت گذلری شده است؛ این مکانهای نگاشته شده در همه پلتفرم ها یکسان هستند. از آنجایی که داده موجود در این مکان حافظه، بوسیله تمام فرآیندها قابل خواندن است، لازم نیست به فضای آدرس فرآیند هدف، تغییر مکان دهیم. بنابراین داده مورد نظر خود را در آدرس 0Xffdf000+0x800 از kernel مینویسیم و هنگام قرار دادن APC در صف، آدرس 0x7ffe000+0x800 را به آن گذر میدهیم.

```
call @get_eip2
@get_eip2:
pop esi
mov cx, shell code-$+1
add esi, ecx ; Get shell code address
mov cx, (shell code_end-shell code) ; Shell code size
mov dword ptr [edi], SMEM_ADDR ; 0xFFDF0000+0x800
push edi
mov edi, [edi] ; Copy shell code to SharedUserData
rep movsb
pop edi
```

حال باید یک thread بیابیم که نیازمندی های ما را در جهت اجرای تابع APC خود، برطرف کند. یک APC میتواند هم بعنوان مد کاربر و هم مد kernel، زمانبندی شود.در این حالت ما از APC در مد کاربر استفاده خواهیم کرد.

یک APC در مد کاربر، در صورتی فراخوانی خواهد شد که thread ی که به آن گذر داده میشود، در وضعیت انتظار قابل هشدار باشد. تکنیک پیدا کردن یک قابل هشدار باشد. تکنیک پیدا کردن یک thread قابل استفاده بدین صورت است:

ابتدا به اشاره گر ساختار ETHREAD فرایندددسترسی پیدا میکنیم و سپس با ایجاد یک حلقه و جستجوی هر یک از THREAD ها، THREAD مورد نظر خود را می یابیم.

```
mov edx, [edi+16] ; Pointer to EPROCESS
mov ecx, [edx+ET_ThreadListHead] ; Get ETHREAD pointer
@find_delay:
mov ecx, [ecx] ; Get next thread
cmp byte ptr [ecx-ET_ThreadState], 04h ; Thread in DelayExecution?
jnz @find_delay
```

قطعه کد بالا، ابتدا از طریق ThreadListHead در ساختار EPROCESS ، اشاره گری به ساختار LSASS ETHREAD دریافت میکند. سپس پرچم های وضعیت thread را بررسی کرده تا thread ی را که در وضعیت DelayExecution باشد، بیابیم. هنگامی که thread هدف ما پیدا شد، آدرس ابتدای ساختار KTHREAD را در ثبات EBP قرار میدهیم. در مرحله بعد، باید رویتین APC خود را مقدار دهی اولیه نمائیم.

```
push edx
push 01 ; push processor
push dword ptr [edi] ; push EIP of shell code (0x7ffe0000+0x800)
push edx ; push NULL
push offset KROUTINE ; push KERNEL routine
push edx ; push NULL
push ebp ; push KTHREAD
push esi ; push APC object
call dword ptr _keinitializeapc ; initialize APC
```

ما EIP مربوط به Shellcode در مدکاربر را(shellcode ذخیره شده در SharedUserData)، به عنوان پارامتر تابع KeInitializeApc در پشته قرار میدهیم. ما باید یک روتین kernel را نیز به آن گذر دهیم. این روتین تقریبا هیچ کاری برای ما انجام نمیدهد و فقط به یک دستور العملRET اشاره میکند. ساختار KTHREAD

آن thread ی که تابع APC ما را اجرا خواهد نیز، ضروری میباشد. شی APC ، در متغیری که توسط ثبات ESI به آن اشاره میگردد، قرار خواهد گرفت.

حالا تابع APC ، باید در صف APC مربوط به thread هدف، قرار گیرد.

```
push eax ; push 0
push dword ptr [edi+4] ; system arg
push dword ptr [edi+8] ; system arg
push esi ; APC object

call dword ptr _keinsertqueueapc
```

آخرین تابع مورد نیاز برای فرستادن APC ، تابع KeInsertQueueApc است.در کد بالا، EAX صفر است و هر دو آرگومان سیستم نیز به یک مکان NULL اشاره میکنند. ما همچنین شی APC بازگردانده شده از فراخوانی قبلی را، به عنوان پارامتر به KeInitializeApc گذر میدهیم. در آخر نیز به منظور جلوگیری از ایجاد BSOD توسط محتوای thread اصلی، باید thread را در وضعیت sleep قرار دهیم.

```
push offset LARGE_INT
push FALSE
push KernelMode
call dword ptr _kedelayexecutionthread
```

در اینجا تابع KeDelayExecutionThread به یک مقدار صحیح بزرگ به عنوان پارامتر(در این مثال 80000000:00000000) فراخوانی شده است.