# Create usermode thread from kernel land

By xSpy /vxjump.net
2014-07-30

**简单方法从内核创建用户态线程.**

在内核想要执行用户态的代码,通常的方式有.apc, usermodecallback 等.
但是都各有缺点.

APC.
1. apc 的分发必须不被禁用,
2. 目标进程必须有处于 alertable 的线程.
特别是后者这个条件,很多时候不一定有.
比如 explorer 进程有很多线程,通常能找到.
但是像记事本这种单线程程序,就找不到.

UserModeCallback.
必须在目标进程空间调用,不能是 attach.的.
必须加载过 User32 的.,这样才有 Kernelcallbacktable

在某些特定的时机,我们是有机会执行的.
比如在进程刚刚创建的时候,我们可以修改 OEP,修改 IAT 等加载我们的 dll

在第一个线程创建之后,我们可以插入 apc.这些条件都很好满足.

还有 WOW64 的兼容处理,在另一篇文章里说明.

但是如果任何时候.不限制调用的时机,比如在进程正常运行之后,这个时候,这些条件都不满足了.

虽然我们可以构造出场景,
比如,如果你是一个过滤驱动或者 hook 型的,那么总是有机会会切换到目标进程上空的,这个时候就有机会可以 UserModeCallback.

现在要说的就是没有这些限制的做法.可以在任意时机,任意进程空间在任意进程中执行代码.
那就是直接在内核态给一个用户态进程创建一个用户态的线程.

模拟用户态进程给自己创建一个非远程线程的基本流程.

1.创建线程初始的栈,分配和保留栈空间.设置栈保护页.实现栈的自动增长.

2 设置线程上下文,各个段寄存器和基本寄存器,设置 eip 指向 Kernel32!BaseThreadTrunk

```
    .text:7C810473
.text:7C810473                          ; ============== S U B R O U T I N E =======================================
.text:7C810473
```

```
.text:7C810473                         ; Attributes: bp-based frame
.text:7C810473
.text:7C810473                         ; int __stdcall BaseInitializeContext(PCONTEXT Context, PVOID Parameter, PVOID
StartAddress, PVOID StackAddress, ULONG ContextType)
.text:7C810473                         _BaseInitializeContext@20 proc near     ; CODE XREF:
CreateRemoteThread(x,x,x,x,x,x,x)+84↓p
.text:7C810473                                                                 ;
CreateProcessInternalW(x,x,x,x,x,x,x,x,x,x,x,x)+690↓p ...
.text:7C810473
.text:7C810473                         Context         = dword ptr  8
.text:7C810473                         Parameter       = dword ptr  0Ch
.text:7C810473                         StartAddress    = dword ptr  10h
.text:7C810473                         StackAddress    = dword ptr  14h
.text:7C810473                         ContextType     = dword ptr  18h
.text:7C810473
.text:7C810473                         ; FUNCTION CHUNK AT .text:7C81F10A SIZE 00000019 BYTES
.text:7C810473                         ; FUNCTION CHUNK AT .text:7C8316A2 SIZE 0000000F BYTES
.text:7C810473
.text:7C810473 8B FF                                   mov     edi, edi
.text:7C810475 55                                      push    ebp
.text:7C810476 8B EC                                   mov     ebp, esp
.text:7C810478 8B 45 08                                mov     eax, [ebp+Context]
.text:7C81047B 8B 4D 10                                mov     ecx, [ebp+StartAddress]
.text:7C81047E 83 A0 8C 00 00 00 00                    and     [eax+CONTEXT.SegGs], 0
.text:7C810485 83 7D 18 01                             cmp     [ebp+ContextType], 1
.text:7C810489 89 88 B0 00 00 00                       mov     [eax+CONTEXT._Eax], ecx
```

```
.text:7C81048F 8B 4D 0C                                  mov     ecx, [ebp+Parameter]
.text:7C810492 89 88 A4 00 00 00                         mov     [eax+CONTEXT._Ebx], ecx
.text:7C810498 6A 20                                     push    20h
.text:7C81049A 59                                        pop     ecx
.text:7C81049B 89 88 94 00 00 00                         mov     [eax+CONTEXT.SegEs], ecx
.text:7C8104A1 89 88 98 00 00 00                         mov     [eax+CONTEXT.SegDs], ecx
.text:7C8104A7 89 88 C8 00 00 00                         mov     [eax+CONTEXT.SegSs], ecx
.text:7C8104AD 8B 4D 14                                  mov     ecx, [ebp+StackAddress]
.text:7C8104B0 C7 80 90 00 00 00 38 00+                  mov     [eax+CONTEXT.SegFs], 38h
.text:7C8104BA C7 80 BC 00 00 00 18 00+                  mov     [eax+CONTEXT.SegCs], 18h
.text:7C8104C4 C7 80 C0 00 00 00 00 30+                  mov     [eax+CONTEXT.EFlags], 3000h
.text:7C8104CE 89 88 C4 00 00 00                         mov     [eax+CONTEXT._Esp], ecx
.text:7C8104D4 0F 85 30 EC 00 00                         jnz     loc_7C81F10A
.text:7C8104DA C7 80 B8 00 00 00 29 07+                  mov     [eax+CONTEXT._Eip], offset _BaseThreadStartThunk@8 ;
BaseThreadStartThunk(x,x)
.text:7C8104E4
.text:7C8104E4                           loc_7C8104E4:                           ; CODE XREF:
BaseInitializeContext(x,x,x,x,x)+ECAB↓j
.text:7C8104E4                                                                   ;
BaseInitializeContext(x,x,x,x,x)+21239↓j
.text:7C8104E4 83 C1 FC                                  add     ecx, 0FFFFFFFCh
.text:7C8104E7 C7 00 07 00 01 00                         mov     [eax+CONTEXT.ContextFlags], 10007h
.text:7C8104ED 89 88 C4 00 00 00                         mov     [eax+CONTEXT._Esp], ecx
.text:7C8104F3 5D                                        pop     ebp
.text:7C8104F4 C2 14 00                                  retn    14h
.text:7C8104F4                           _BaseInitializeContext@20 endp
```

```
.text:7C8104F4
.text:7C8104F4                          ; ---------------------------------------------------------------------------
```

3 对于 vista 以后,还得分配 TEB 的 ActiveContextStackPointer.要不然执行某些用户态的 API 的时候,那些 API 没有检查 TEB 的 ActiveContextStackPointer 是否为 NULL 就从中取值,造成崩溃.
windows 的 CreateThread 也做了这些事.

```
.text:0DCEBD8A 23 4D 10                 and     ecx, [ebp+dwStackSize]
.text:0DCEBD8D 51                       push    ecx             ; MaximumStackSize
.text:0DCEBD8E F7 D8                    neg     eax
.text:0DCEBD90 1B C0                    sbb     eax, eax
.text:0DCEBD92 23 45 10                 and     eax, [ebp+dwStackSize]
.text:0DCEBD95 50                       push    eax             ; StackSize
.text:0DCEBD96 53                       push    ebx             ; ZeroBits
.text:0DCEBD97 56                       push    esi             ; CreateThreadFlags
.text:0DCEBD98 FF B5 B8 FD FF FF        push    [ebp+StartContext] ; StartContext
.text:0DCEBD9E FF B5 D0 FD FF FF        push    [ebp+StartRoutine] ; StartRoutine
.text:0DCEBDA4 FF B5 CC FD FF FF        push    [ebp+ProcessHandle] ; ProcessHandle
.text:0DCEBDAA FF B5 BC FD FF FF        push    [ebp+ObjectAttributes] ; ObjectAttributes
```

```
.text:0DCEBDB0 68 FF FF 1F 00                push    1FFFFFh         ; DesiredAccess
.text:0DCEBDB5 8D 85 E4 FD FF FF             lea     eax, [ebp+hThread]
.text:0DCEBDBB 50                            push    eax             ; ThreadHandle
.text:0DCEBDBC FF 15 74 13 CE 0D             call    ds:__imp__NtCreateThreadEx@44 ;
NtCreateThreadEx(x, x, x, x, x, x, x, x, x, x, x)
.text:0DCEBDC2 89 85 E8 FD FF FF             mov     [ebp+var_218], eax
.text:0DCEBDC8 3B C3                         cmp     eax, ebx
.text:0DCEBDCA 0F 8C A5 F8 01 00             jl      loc_DD0B675
.text:0DCEBDD0 89 5D FC                      mov     [ebp+ms_exc.disabled], ebx
.text:0DCEBDD3 64 A1 18 00 00 00             mov     eax, large fs:18h
.text:0DCEBDD9 8B 8D C0 FD FF FF             mov     ecx, [ebp+var_240]
.text:0DCEBDDF 3B 48 20                      cmp     ecx, [eax+20h]
.text:0DCEBDE2 75 73                         jnz     short loc_DCEBE57
.text:0DCEBDE4 8D 85 E0 FD FF FF             lea     eax, [ebp+var_220]
.text:0DCEBDEA 50                            push    eax
.text:0DCEBDEB FF 15 70 13 CE 0D             call    ds:__imp__RtlAllocateActivationContextStack@4 ;
RtlAllocateActivationContextStack(x)
.text:0DCEBDF1 89 85 E8 FD FF FF             mov     [ebp+var_218], eax
.text:0DCEBDF7 3B C3                         cmp     eax, ebx
.text:0DCEBDF9 0F 8C B2 F8 01 00             jl      loc_DD0B6B1
.text:0DCEBDFF 8B 85 E0 FD FF FF             mov     eax, [ebp+var_220]
.text:0DCEBE05 8B 8D D4 FD FF FF             mov     ecx, [ebp+var_22C]
.text:0DCEBE0B 89 81 A8 01 00 00             mov     [ecx+1A8h], eax
.text:0DCEBE11 53                            push    ebx
.text:0DCEBE12 6A 08                         push    8
.text:0DCEBE14 8D 85 D8 FD FF FF             lea     eax, [ebp+var_228]
```

```
.text:0DCEBE1A 50                                      push    eax
.text:0DCEBE1B 56                                      push    esi
```

4 获取当前进程的 BaseObject 目录,可以是默认的

5 ZwCreateThread 创建线程对象了.挂起的

6 最重要的一点了.通知 csrss 进程,有新线程创建了.

```
.text:0DCEBE65 8B 85 E4 FD FF FF                       mov     eax, [ebp+hThread]
.text:0DCEBE6B 89 85 18 FE FF FF                       mov     [ebp+var_1E8], eax
.text:0DCEBE71 8B 85 C0 FD FF FF                       mov     eax, [ebp+var_240]
.text:0DCEBE77 89 85 1C FE FF FF                       mov     [ebp+var_1E4], eax
.text:0DCEBE7D 8B 85 C4 FD FF FF                       mov     eax, [ebp+var_23C]
.text:0DCEBE83 89 85 20 FE FF FF                       mov     [ebp+var_1E0], eax
.text:0DCEBE89 6A 0C                                   push    0Ch
.text:0DCEBE8B 68 01 00 01 00                          push    10001h
.text:0DCEBE90 53                                      push    ebx
.text:0DCEBE91 8D 85 F0 FD FF FF                       lea     eax, [ebp+var_210]
.text:0DCEBE97 50                                      push    eax
.text:0DCEBE98 FF 15 F0 11 CE 0D                       call    ds:__imp__CsrClientCallServer@16 ;
CsrClientCallServer(x,x,x,x)
```

```
.text:0DCEBE9E 8B 85 10 FE FF FF                        mov     eax, [ebp+var_1F0]
.text:0DCEBEA4
.text:0DCEBEA4                          loc_DCEBEA4:                            ; CODE XREF:
GetDiskFreeSpaceExA(x,x,x,x)+2FB9↓j
.text:0DCEBEA4 89 85 E8 FD FF FF                        mov     [ebp+var_218], eax
```

7 恢复线程的执行.

```
.text:0DCEBEC8
.text:0DCEBEC8                          loc_DCEBEC8:                            ; CODE XREF:
CreateRemoteThreadEx(x,x,x,x,x,x,x,x)+22A↑j
.text:0DCEBEC8 F6 45 1C 04                              test    byte ptr [ebp+dwCreationFlags], 4
.text:0DCEBECC 75 13                                    jnz     short loc_DCEBEE1
.text:0DCEBECE 8D 85 AC FD FF FF                        lea     eax, [ebp+var_254]
.text:0DCEBED4 50                                       push    eax
.text:0DCEBED5 FF B5 E4 FD FF FF                        push    [ebp+hThread]
.text:0DCEBEDB FF 15 3C 13 CE 0D                        call    ds:__imp__NtResumeThread@8 ; NtResumeThread(x,x)
.text:0DCEBEE1
.text:0DCEBEE1                          loc_DCEBEE1:                            ; CODE XREF:
CreateRemoteThreadEx(x,x,x,x,x,x,x,x)+238↑j
.text:0DCEBEE1                                                                  ; GetDiskFreeSpaceExA(x,x,x,x)+2F6E↓
j ...
.text:0DCEBEE1 C7 45 FC FE FF FF FF                     mov     [ebp+ms_exc.disabled], 0FFFFFFFEh
.text:0DCEBEE8 E8 34 00 00 00                           call    sub_DCEBF21
.text:0DCEBEED 8B 85 E4 FD FF FF                        mov     eax, [ebp+hThread]
.text:0DCEBEF3
```

```
.text:0DCEBEF3                          loc_DCEBEF3:                            ; CODE XREF:
GetDiskFreeSpaceExA(x,x,x,x)+2F27↓j
.text:0DCEBEF3 E8 A1 AD FF FF                       call    __SEH_epilog4_GS
.text:0DCEBEF8 C2 20 00                             retn    20h
.text:0DCEBEF8                          _CreateRemoteThreadEx@32 endp
.text:0DCEBEF8
.text:0DCEBEF8                          ; ---------------------------------------------------------------------------
```

对于 windows 的 CreateThread 还有一些其他的操作,比如判断是否是 csrss 进程自己在创建线程.
vista 以后对于远程的线程,还有 session 的检查等.

听起来很麻烦的一件事情,其实我们可以简化问题.
在我的实现里,不考虑 csrss 自己给自己创建线程的情况,
实际上我们创建的都是普通的线程,非远程的,

很多同学尝试过模拟这个过程,大部分都在第 6 步卡住了,.这一步比较麻烦.

每个用户态进程在创建的时候,都会连接 \\Windows\\ApiPort ,
但是发现,如果我们在内核直接连接 csrss 的这个 port,是连不上的.需要 patch.

其实可以不用 patch.,直接切换到 csrss 空间,自己来操作 CsrProcessTable 等内置数据结构,但是不同意.

我用到的办法比较简单.
因为目标进程已经连接过了.这个句柄还是有符号的, CsrPortHandle.

既然从内核连接不上,我们可以在系统句柄表里去搜索这个句柄,

搜索所有的 LpcPort 或 AlpcPort 类型的句柄,
判断是否是我们需要的进程,
然后判断他们的 ConnectionPort 是否是\\Windows\\ApiPort.
找到句柄之后,duplicate 到当前进程,

就可以 ZwRequestWaitReplyPort 或 ZwAlpcSendWaitReceivePort 通知 csrss 了.

关于 Kernel32!BaseThreadTrunk 我并没有直接把 eip 指向这个地方,这个函数没有导出.

```
.text:7C810729
.text:7C810729                         ; ============== S U B R O U T I N E =======================================
.text:7C810729
.text:7C810729                         ; Attributes: noreturn
.text:7C810729
.text:7C810729                         ; int __stdcall BaseThreadStartThunk(int, int)
.text:7C810729                         _BaseThreadStartThunk@8 proc near       ; DATA XREF:
BaseInitializeContext(x,x,x,x,x)+67↑o
.text:7C810729
.text:7C810729                         arg_0           = dword ptr  4
.text:7C810729                         arg_4           = dword ptr  8
.text:7C810729
.text:7C810729 33 ED                                   xor     ebp, ebp
.text:7C81072B 53                                      push    ebx             ; Param
```

```
.text:7C81072C 50                                push    eax             ; StartAddress
.text:7C81072D 6A 00                             push    0
.text:7C81072F E9 BE AF FF FF                    jmp     _BaseThreadStart@8 ; BaseThreadStart(x,x)
.text:7C81072F                          _BaseThreadStartThunk@8 endp
.text:7C81072F
.text:7C81072F                          ; ---------------------------------------------------------------------------


.text:7C80B6F2
.text:7C80B6F2                          ; =============== S U B R O U T I N E =======================================
.text:7C80B6F2
.text:7C80B6F2                          ; Attributes: noreturn bp-based frame
.text:7C80B6F2
.text:7C80B6F2                          ; int __stdcall BaseThreadStart(int StartAddress, int ThreadParam)
.text:7C80B6F2                          _BaseThreadStart@8 proc near            ; CODE XREF:
BaseThreadStartThunk(x,x)+6↓j
.text:7C80B6F2                                                                  ; BaseFiberStart()+12↓p
.text:7C80B6F2
.text:7C80B6F2                          Teb             = dword ptr -20h
.text:7C80B6F2                          ms_exc          = CPPEH_RECORD ptr -18h
.text:7C80B6F2                          StartAddress    = dword ptr  8
.text:7C80B6F2                          ThreadParam     = dword ptr  0Ch
.text:7C80B6F2
.text:7C80B6F2 6A 10                             push    10h
.text:7C80B6F4 68 30 B7 80 7C                    push    offset stru_7C80B730
.text:7C80B6F9 E8 D8 6D FF FF                    call    __SEH_prolog
```

```
.text:7C80B6FE 83 65 FC 00                          and     [ebp+ms_exc.disabled], 0
.text:7C80B702 64 A1 18 00 00 00                    mov     eax, large fs:18h
.text:7C80B708 89 45 E0                             mov     [ebp+Teb], eax
.text:7C80B70B 81 78 10 00 1E 00 00                 cmp     dword ptr [eax+10h], 1E00h
.text:7C80B712 75 0F                                jnz     short loc_7C80B723
.text:7C80B714 80 3D 08 50 88 7C 00                 cmp     _BaseRunningInServerProcess, 0
.text:7C80B71B 75 06                                jnz     short loc_7C80B723
.text:7C80B71D FF 15 F8 12 80 7C                    call    ds:__imp__CsrNewThread@0 ; CsrNewThread()
.text:7C80B723
.text:7C80B723                      loc_7C80B723:                           ; CODE XREF: BaseThreadStart(x,x)+20
↑j
.text:7C80B723                                                              ; BaseThreadStart(x,x)+29↑j
.text:7C80B723 FF 75 0C                             push    [ebp+ThreadParam]
.text:7C80B726 FF 55 08                             call    [ebp+StartAddress]
.text:7C80B729 50                                   push    eax             ; dwExitCode
.text:7C80B72A
.text:7C80B72A                      loc_7C80B72A:                           ; CODE XREF: .text:7C83AB3B↓j
.text:7C80B72A E8 C9 09 00 00                       call    _ExitThread@4   ; ExitThread(x)
.text:7C80B72A                      _BaseThreadStart@8 endp
.text:7C80B72A
.text:7C80B72A                      ; ---------------------------------------------------------------------------
```

而且我还需要分配 ActiveContextStackPointer,.
所以新线程的 eip 实际上是指向一段 stub,
在 stub 里分配 ActiveContextStackPointer,然后模拟的 call 线程的起始地址,

然后调用 RtlExitUserThread,确保在 StartAddress ret 的时候,可以自行退出.
就像系统做的那样.

```
mov edi,API_RtlExitUserThread
test edi,edi
je _DirectRet

;调用用户提供的线程函数地址
mov eax,var_StartAddress
mov ebx,var_ThreadParam

push ebx    ;线程的参数
call eax    ;线程的起始地址

;是的用户线程函数返回时,我们可以让线程退出
push eax
call API_RtlExitUserThread
```

流程说完了.现在我们已经在内核模拟一个用户态线程给自己创建了一个线程.
非远程的,支持 WOW64.

没有那么多限制条件执行用户态代码之后,可以做的事情就只局限于你的想象力了.
给目标进程注入一个 dll 简直是一个小意思了.

如果有事先执行的机会,就可以伪造各个杀毒软件或者系统进程的身份了.

附一些代码,因为依赖比较多,只贴关键的说明问题.

```
//创建用户栈
NTSTATUS _BaseCreateStack(IN HANDLE hProcess,OUT INITIAL_TEB* pInitialTeb)
{
    NTSTATUS Status = STATUS_UNSUCCESSFUL;
    SYSTEM_BASIC_INFORMATION* pSysBasicInfo = NULL;
    ULONG_PTR ulSize = 0;

    ULONG_PTR StackReserve = 0;
    ULONG_PTR StackCommit = 0;
    ULONG_PTR Stack = 0;
    BOOLEAN UseGuard = FALSE;
    ULONG_PTR GuardPageSize = 0;
    ULONG Dummy = 0;

    LPFN_ZwProtectVirtualMemory  fnZwProtectVirtualMemory = NULL;

    do
    {
        if ( (NULL == hProcess) || (NULL == pInitialTeb) )
```

```
{
    xDebugA(("[-] 参数不正确! \n"));
    break;
}

fnZwProtectVirtualMemory = INIT_ZW_API(ZwProtectVirtualMemory);
if (NULL == fnZwProtectVirtualMemory)
{
    xDebugA(("[-] 获取 fnZwProtectVirtualMemory 失败! \n"));
    break;
}

pSysBasicInfo = (SYSTEM_BASIC_INFORMATION*)xAlloc(sizeof(SYSTEM_BASIC_INFORMATION));
if (NULL == pSysBasicInfo)
{
    xDebugA(("[-]内存分配失败! \n"));
    break;
}

//获取内存信息
Status = ZwQuerySystemInformation(SystemBasicInformation,
    pSysBasicInfo,
    sizeof(SYSTEM_BASIC_INFORMATION),
    &ulSize
    );
```

```
if (!NT_SUCCESS(Status))
{
	xDebugA(("[-] 获取系统基本信息失败! %s \n" , Status2Str(Status) ));
	break;
}

//系统默认的栈信息
StackReserve = SIZE_MB * 1;
StackCommit = SIZE_KB * 64;;

//栈提交大小是否大于栈保存的大小
if (StackCommit >= StackReserve)
{
	//增大保存的大小,1MB对齐
	StackReserve = ROUND_UP(StackCommit,1024 * 1024);
}

//对齐到页面大小
StackReserve = ROUND_UP(StackReserve,pSysBasicInfo->AllocationGranularity);
StackCommit = ROUND_UP(StackCommit,pSysBasicInfo->PageSize);

//为栈分配保留的内存
Status = ZwAllocateVirtualMemory(hProcess,
	(PVOID*)&Stack,
	0,
	&StackReserve,
```

```
        MEM_RESERVE,
        PAGE_READWRITE
        );

if (!NT_SUCCESS(Status))
{
        xDebugA(("[-]为栈保留内存  失败!    %s \n " ,Status2Str(Status) ));
        break;
}

//初始化TEB
pInitialTeb->PreviousStackBase = NULL;
pInitialTeb->PreviousStackLimit = NULL;
pInitialTeb->AllocatedStackBase = (PVOID)Stack;
pInitialTeb->StackBase = (PVOID)(Stack + StackReserve);

//更新栈的位置
Stack += StackReserve - StackCommit;

//判断是否需要栈保护页来实现栈的自动增长
if (StackReserve > StackCommit)
{
        //空出一页作为保护页
        Stack -= pSysBasicInfo->PageSize;
        StackCommit += pSysBasicInfo->PageSize;
        UseGuard = TRUE;
```

```
}


//真正的分配栈内存
Status = ZwAllocateVirtualMemory(hProcess,
      (PVOID*)&Stack,
      0,
      &StackCommit,
      MEM_COMMIT,
      PAGE_READWRITE
      );

if (!NT_SUCCESS(Status))
{
      xDebugA(("[-]为栈分配内存  失败!    %s \n " ,Status2Str(Status) ));
      break;
}

//栈的限制大小
pInitialTeb->StackLimit = (PVOID)Stack;

//创建保护页
if (UseGuard)
{
      /* Attempt maximum space possible */
      GuardPageSize = pSysBasicInfo->PageSize;
```

```
            Status = CALL_API(ZwProtectVirtualMemory)(hProcess,
                (PVOID*)&Stack,
                &GuardPageSize,
                PAGE_GUARD | PAGE_READWRITE,
                &Dummy
                );

            if ( !NT_SUCCESS(Status))
            {
                xDebugA(("[-]为栈创建保护页  失败!    %s \n " ,Status2Str(Status) ));
                break;
            }

            /* Update the Stack Limit keeping in mind the Guard Page */
            pInitialTeb->StackLimit = (PVOID)((ULONG_PTR)pInitialTeb->StackLimit + GuardPageSize);
        }

    } while (FALSE);

    xFree(pSysBasicInfo);

    return Status;
}

NTSTATUS _BaseInitializeContext
```

```
(
    IN PEPROCESS Process,
    IN BOOLEAN bWow64,
    IN PCONTEXT Context,
    IN PVOID Parameter,
    IN PVOID StartAddress,
    IN PVOID StackAddress
)
{
    ULONG ulThunkSize = 0;
    PVOID pLocalThunk = NULL;          //内核的Thunk数据
    PVOID pRemoteThunk = NULL;  //用户态的Thunk地址
    NTSTATUS Status = STATUS_UNSUCCESSFUL;
    ULONG ulInfoSize = 0;


    do
    {

#ifdef _WIN64
        if (bWow64)
        {
            pLocalThunk = _BaseThreadStartThunk_x86;
            ulThunkSize = sizeof(_BaseThreadStartThunk_x86);
        }
        else
```

```
        {
            pLocalThunk = _BaseThreadStartThunk_x64;
            ulThunkSize = sizeof(_BaseThreadStartThunk_x64);
        }

#else
        pLocalThunk = _BaseThreadStartThunk_x86;
        ulThunkSize = sizeof(_BaseThreadStartThunk_x86);
#endif

        Status = kVirtualAllocEx(Process,
            &pRemoteThunk,
            ulThunkSize,
            MEM_COMMIT|MEM_RESERVE,
            PAGE_EXECUTE_READWRITE
            );
        if (!NT_SUCCESS(Status))
        {
            xDebugA(("[-] 分配 thunk内存失败! \n"));
            break;
        }

        Status = kVirtualWrite(Process,
            pRemoteThunk,
            pLocalThunk,
            ulThunkSize,
```

```
            &ulInfoSize
            );
        if (!NT_SUCCESS(Status))
        {
            xDebugA(("[-] 写入 thunk到ring3失败! \n"));
            break;
        }


#ifdef _WIN64

        /* Setup the Initial Win32 Thread Context */
        Context->Rax = (ULONG_PTR)StartAddress;
        Context->Rbx = (ULONG_PTR)Parameter;
        Context->Rsp = (ULONG_PTR)StackAddress;
        /* The other registers are undefined */

        /* Setup the Segments */
        Context->SegGs = 0x0028 | 0x0003;
        Context->SegEs = 0x0028 | 0x0003;
        Context->SegDs = 0x0028 | 0x0003;
        Context->SegCs = 0x0030 | 0x0003;
        Context->SegSs = 0x0028 | 0x0003;
        Context->SegFs = 0x0050 | 0x0003;

        /* Set the EFLAGS */
```

```
        Context->EFlags = 0x3000; /* IOPL 3 */

        /* Set the Context Flags */
        Context->ContextFlags = CONTEXT_FULL;

        /* Give it some room for the Parameter */
        Context->Rsp -= sizeof(PVOID);

        Context->Rip = (ULONG_PTR)pRemoteThunk;
#else

        /* Setup the Initial Win32 Thread Context */
        Context->Eax = (ULONG)StartAddress;
        Context->Ebx = (ULONG)Parameter;
        Context->Esp = (ULONG)StackAddress;
        /* The other registers are undefined */

        /* Setup the Segments */
        Context->SegFs = 0x38;
        Context->SegEs = 0x20;
        Context->SegDs = 0x20;
        Context->SegCs = 0x18;
        Context->SegSs = 0x20;
        Context->SegGs = 0;

        /* Set the EFLAGS */
```

```
            Context->EFlags = 0x3000; /* IOPL 3 */

            /* Set the Context Flags */
            Context->ContextFlags = CONTEXT_FULL;

            /* Give it some room for the Parameter */
            Context->Esp -= sizeof(PVOID);

            Context->Eip = (ULONG)pRemoteThunk;
#endif

            xDebugA(("[*] ThreadTrunk = 0x%p \n" , pRemoteThunk ));

            Status = STATUS_SUCCESS;

        } while (FALSE);


        if (!NT_SUCCESS(Status))
        {
            if (NULL != pRemoteThunk)
            {
                kVirtualFree(Process,pRemoteThunk);
                pRemoteThunk = NULL;
            }
        }
```

```
    return Status;
}

//创建用户态线程
NTSTATUS kCreateUserModeThread(
    IN PEPROCESS Process,
    IN BOOLEAN bCreateSuspended,
    IN void* pStartAddress,   //用户态地址
    IN PVOID pParameter,          //用户态地址
     IN OUT HANDLE* phThreadHandle,
     IN OUT HANDLE* phThreadId
)
{
    HANDLE hProcess = NULL;
    OBJECT_ATTRIBUTES ObjectAttributes = {0};
    NTSTATUS Status = STATUS_UNSUCCESSFUL;
    CONTEXT ThreadContext = {0};
    HANDLE hThread = NULL;
    CLIENT_ID ClientId = {0};
    INITIAL_TEB UserStack = {0};

    CSR_API_MSG ApiMessage = {0};
     BASE_CREATE_THREAD* pCreateThreadRequest = &ApiMessage.Data.CreateThread;

    ULONG SuspendCount = 0;
```

```
SIZE_T Dummy = 0;
BOOLEAN bWow64 = FALSE;

do
{
    if (    (NULL == Process) || (NULL == pStartAddress) )
    {
        xDebugA(("[-]  参数不正确! \n"));
        break;
    }

    if (NULL == INIT_ZW_API(ZwCreateThread) )
    {
        xDebugA(("[-]  获取  ZwCreateThread  地址失败! \n"));
        break;
    }

    if (NULL == INIT_ZW_API(ZwResumeThread) )
    {
        xDebugA(("[-]  获取  ZwResumeThread  地址失败! \n"));
        break;
    }

    if (NULL == INIT_ZW_API(ZwTerminateThread) )
    {
        xDebugA(("[-]  获取  ZwTerminateThread  地址失败! \n"));
```

```
        }


#ifdef _WIN64
        kIsWow64Process(Process,&bWow64);
#endif


        Status = ObOpenObjectByPointer(Process,
            OBJ_KERNEL_HANDLE,
            NULL,
            PROCESS_ALL_ACCESS,
            NULL,
            KernelMode,
            &hProcess
            );

        if (!NT_SUCCESS(Status))
        {
            xDebugA(("[-]  打开进程失败! %s \n" , Status2Str(Status) ));
            break;
        }

        //创建一个用户态的栈
        Status = _BaseCreateStack(hProcess,&UserStack);
        if (!NT_SUCCESS(Status))
```

```
{
	xDebugA(("[-] Creat UserMode Stack faild %s \n" , Status2Str(Status)    ));
	break;
}

//初始化新线程的上下文
_BaseInitializeContext(
	Process,
	bWow64,
	&ThreadContext,
	pParameter,
	pStartAddress,
	UserStack.StackBase
	);

InitializeObjectAttributes(
	&ObjectAttributes,
	NULL,
	OBJ_KERNEL_HANDLE,
	NULL,
	NULL
);

ClientId.UniqueProcess = PsGetProcessId(Process);

//创建线程,
```

```
Status = CALL_API(ZwCreateThread)(
	&hThread,
	THREAD_ALL_ACCESS,
	&ObjectAttributes,
	hProcess,
	&ClientId,
	&ThreadContext,
	&UserStack,
	TRUE   //挂起
);

if (!NT_SUCCESS(Status))
{
	xDebugA(("[-]  创建线程失败  %s \n" , Status2Str(Status) ));
	break;
}

pCreateThreadRequest->ClientId.UniqueProcess = ClientId.UniqueProcess;
pCreateThreadRequest->ClientId.UniqueThread = ClientId.UniqueThread;
pCreateThreadRequest->hThread = hThread;

//通知csrss,非常重要,
//这个操作涉及到搜索句柄,耗时  80 ~ 230 ms不等.
Status = kInformCsrss(Process,
	(CSR_API_MSG*)&ApiMessage,
	CSR_CREATE_API_NUMBER( BASESRV_SERVERDLL_INDEX,BasepCreateThread),
```

```
            sizeof(BASE_CREATE_THREAD)
            );
        if (!NT_SUCCESS(Status))
        {
            xDebugA(("[-] 通知csrss 失败 %s \n" , Status2Str(Status) ));
            break;
        }

        //恢复线程的执行.
        if (!bCreateSuspended)
        {
            CALL_API(ZwResumeThread)(hThread,&SuspendCount);
        }

        xDebugA(("[+] 创建线程成功, Pid: %d Tid: %d hThread: 0x%p , 起始地址: 0x%p \n" ,
                ClientId.UniqueProcess,
                ClientId.UniqueThread,
                hThread,
                pStartAddress
            ));

        Status = STATUS_SUCCESS;
    } while (FALSE);

    if (!NT_SUCCESS(Status))
    {
```

```
    if (NULL != hProcess)
    {
        NtFreeVirtualMemory(hProcess,
            &UserStack.AllocatedStackBase,
            &Dummy,
            MEM_RELEASE
        );
    }

    if (NULL != hThread)
    {
        if (NULL != INIT_API(ZwTerminateThread))
        {
            CALL_API(ZwTerminateThread)(hThread,Status);
        }

        ZwClose(hThread);
        hThread = NULL;
    }
}

if (NULL != hProcess)
{
    ZwClose(hProcess);
    hProcess = NULL;
}
```

```
    if (NULL != phThreadHandle)
    {
        *phThreadHandle = hThread;
    }
    else
    {
        if (NULL != hThread)
        {
            ZwClose(hThread);
            hThread = NULL;
        }
    }

    if (NULL != phThreadId)
    {
        *phThreadId = ClientId.UniqueThread;
    }

    return Status;
}


//使用指定进程通知Csrss,
NTSTATUS kInformCsrss(
    IN PEPROCESS Process,
```

```
    IN OUT CSR_API_MSG* pCsrMsg,
    IN CSR_API_NUMBER ApiNumber,
    IN ULONG ArgLength
)
{
    NTSTATUS Status = STATUS_INVALID_PARAMETER;
    ULONG FixArgLength = 0;
    HANDLE hCsrPortHandle = NULL;

    do
    {
        if ( (NULL == Process) || (NULL == pCsrMsg)    )
        {
            xDebugA(("[-]  参数不正确! \n"));
            break;
        }

        if (kGetOSVer() >= OS_VISTA)
        {
            if (NULL == INIT_API(ZwAlpcSendWaitReceivePort))
            {
                xDebugA(("[-]  获取  ZwAlpcSendWaitReceivePort  地址失败! \n"));
                break;
            }
        }
```

```
//获取进程的CsrPostHandle,会自动dup到当前进程
Status = kGetProcessCsrPortHandle(Process,&hCsrPortHandle);
if ( (!NT_SUCCESS(Status)) || (NULL == hCsrPortHandle)    )
{
    xDebugA(("获取Port Handle  失败  %s \n" , Status2Str(Status) ));
    Status = STATUS_UNSUCCESSFUL;
    break;
}

FixArgLength = ArgLength;
if ( (LONG)ArgLength < 0 )
{
    FixArgLength = (ULONG)(-(LONG)ArgLength);
    pCsrMsg->PortMessage.u2.s2.Type = 0;
}
else
{
    pCsrMsg->PortMessage.u2.ZeroInit = 0;
}

FixArgLength |= (FixArgLength << 16);

if (kGetOSVer() < OS_VISTA)
{
    FixArgLength +=     0x2C0010;
}
```

```
else
{
    #ifdef _WIN64
        FixArgLength += 0x400018;

    #else
        FixArgLength += 0x280010;
    #endif

}

pCsrMsg->PortMessage.u1.Length = FixArgLength;

pCsrMsg->CaptureData = NULL;

pCsrMsg->ApiNumber = ApiNumber;

if (kGetOSVer() < OS_VISTA)
{
    Status = ZwRequestWaitReplyPort(hCsrPortHandle,
        (PORT_MESSAGE*)pCsrMsg,
        (PORT_MESSAGE*)pCsrMsg
    );
}
else
{
```

```
            //这个在我的虚拟机上慢的时候竟然需要300ms ! WTF
            Status = CALL_API(ZwAlpcSendWaitReceivePort)(
                hCsrPortHandle,
                0,
                (PORT_MESSAGE*)pCsrMsg,
                NULL,
                (PORT_MESSAGE*)pCsrMsg,
                NULL,
                NULL,
                NULL
            );
        }

    } while (FALSE);

    if (NULL != hCsrPortHandle)
    {
        ZwClose(hCsrPortHandle);
        hCsrPortHandle = NULL;
    }

    return Status;
}
```

```
typedef struct _CSR_API_MSG
{
    PORT_MESSAGE PortMessage;  //    /*0x00*/
    union
    {
        BASE_API_CONNECTINFO ConnectionInfo;      //Base Api    /*0x18*/

        struct
        {
            CSR_CAPTURE_HEADER* CaptureData;          /*0x18*/
            ULONG       ApiNumber;            /*0x1C*/
            NTSTATUS Status;   //ReturnValue;         /*0x20*/
            ULONG       Reserved;                     /*0x24*/
            union
            {

                BASE_CREATE_THREAD    CreateThread;       //apiNumber = 0       /*0x28*/
                BASE_CREATE_PROCESS CreateProcess;        //apiNumber = 0

                //这里只是为了占位
                ULONG_PTR ApiMessageData[39];
            }Data;
        };
    };
}CSR_API_MSG;
```

```
//创建线程
typedef struct    _BASE_CREATE_THREAD
{
      HANDLE hThread;
      CLIENT_ID ClientId;
}BASE_CREATE_THREAD;
```

全文代码在下列系统测试通过.

xp/2003 32
win7/8/8.1 32/64

xSpy@binvul.com
xSpy@vxjump.net
2014.07.29