Windows Security Internals

--- Bhavesh/Skynet/Larry

==========

The Windows Kernel

==========

Powershell doesn't allow locally created scripts which aint signed

" Set-ExecutionPolicy -Scope CurrentUser -ExecutionPolicy RemoteSigned –Force "

Install module :

" Install-Module NtObjectManager -Scope CurrentUser –Force "

Update-Module NtObjectManager

Import-Module NtObjectManager

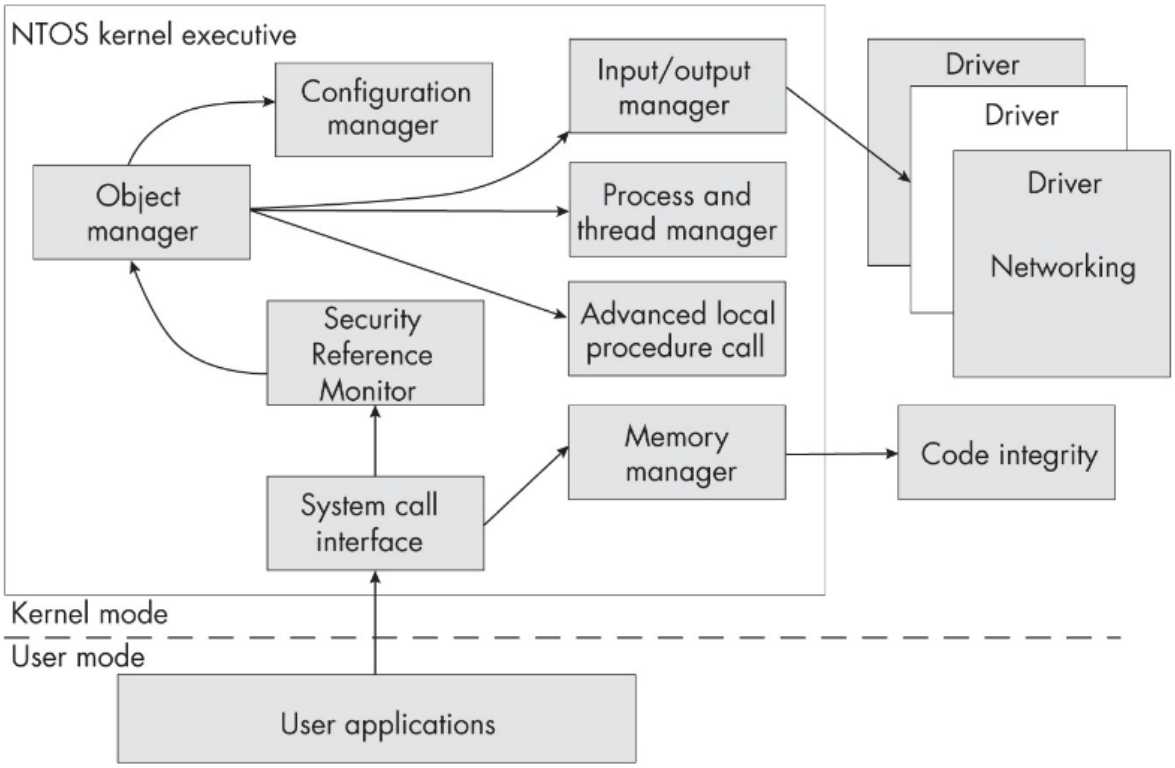


Figure 2-1: The Windows kernel executive modules

Kernel is called the heart of OS

User Mode Components:

User Applications

- Programs that run outside the kernel (Word, Chrome, games, etc.)
- Cannot directly access hardware or kernel functions
- Must request services through the System Call Interface

Kernel Mode Components:

System Call Interface

- **Purpose**: Gateway between user mode and kernel mode
- **Function**: Validates and routes application requests to appropriate kernel modules
- **Security**: Checks permissions before allowing access to kernel functions

Memory Manager

- **Purpose**: Controls all system memory
- **Function**:
    - Virtual memory management (paging)
    - Memory allocation/deallocation
    - Memory protection between processes
    - Cache management

Code Integrity

- **Purpose**: Prevents unauthorized code execution
- **Function**:
    - Validates digital signatures of drivers and system files
    - Prevents kernel-mode rootkits
    - Ensures only trusted code runs in kernel mode

Security Reference Monitor (SRM)

- **Purpose**: Central security enforcement
- **Function**:
    - Validates all access to objects
    - Enforces security policies
    - Checks access tokens against ACLs (Access Control Lists)
    - Audits security events

Object Manager

- **Purpose**: Unified resource management system
- **Function**:
    - Creates, manages, and deletes kernel objects (files, processes, threads, etc.)
    - Provides namespace for all system resources
    - Implements reference counting and garbage collection

Configuration Manager

- **Purpose**: Manages system configuration database
- **Function**:

- Handles the Windows Registry
- Stores and retrieves system/application settings
- Manages registry hives and keys

Input/Output Manager

- **Purpose**: Coordinates all input/output operations
- **Function**:
  - Manages device drivers
  - Handles I/O requests from applications
  - Provides caching and buffering
  - Routes I/O to appropriate drivers

Process and Thread Manager

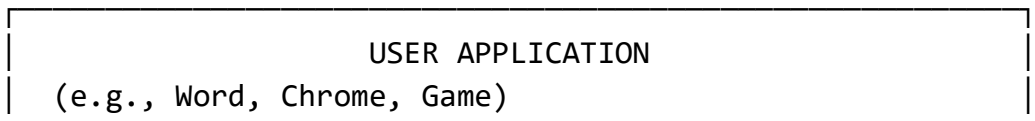
- **Purpose**: Manages program execution units
- **Function**:
  - Creates, schedules, and terminates processes/threads
  - Manages CPU time allocation
  - Handles process priorities and states

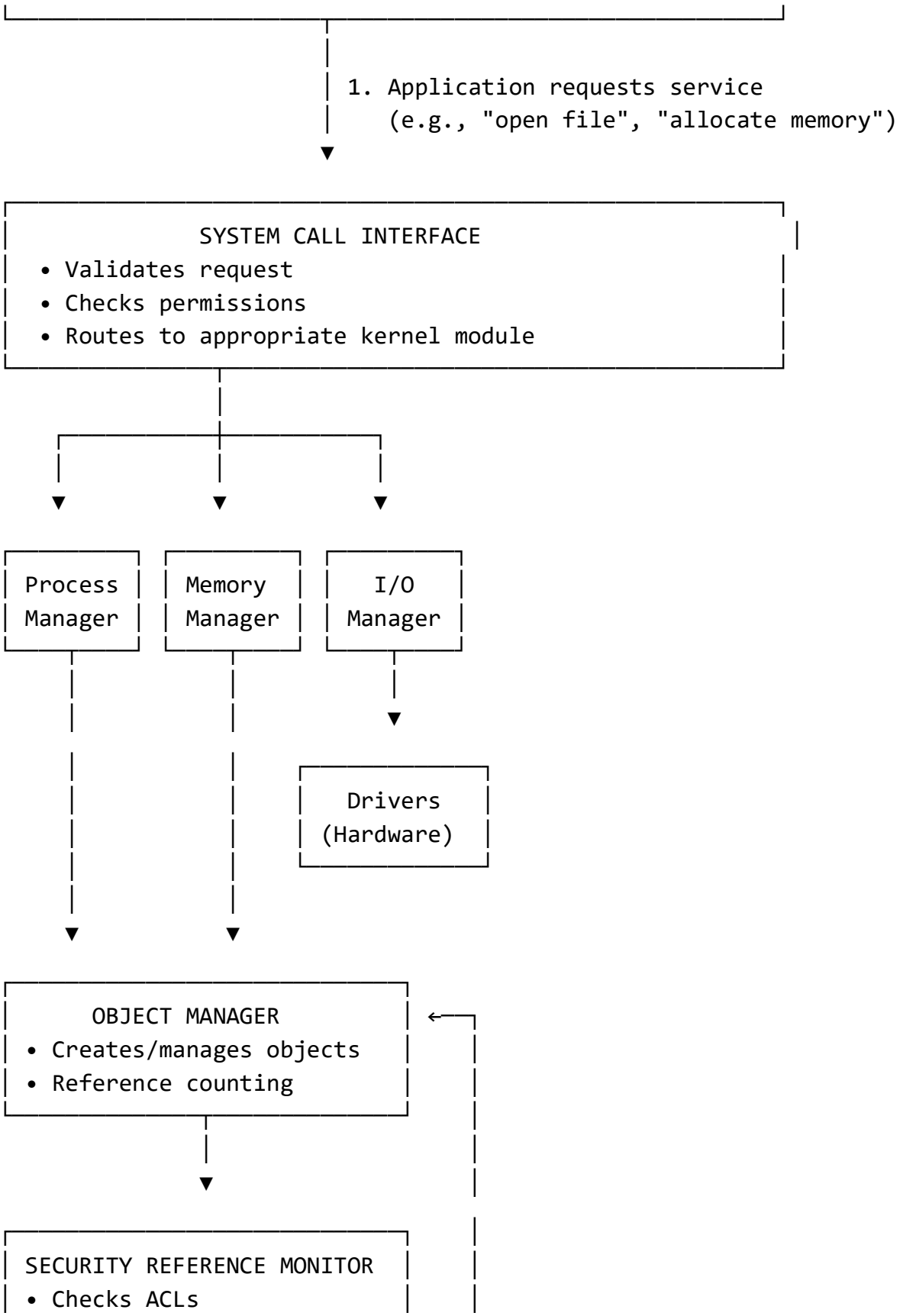
Advanced Local Procedure Call (ALPC)

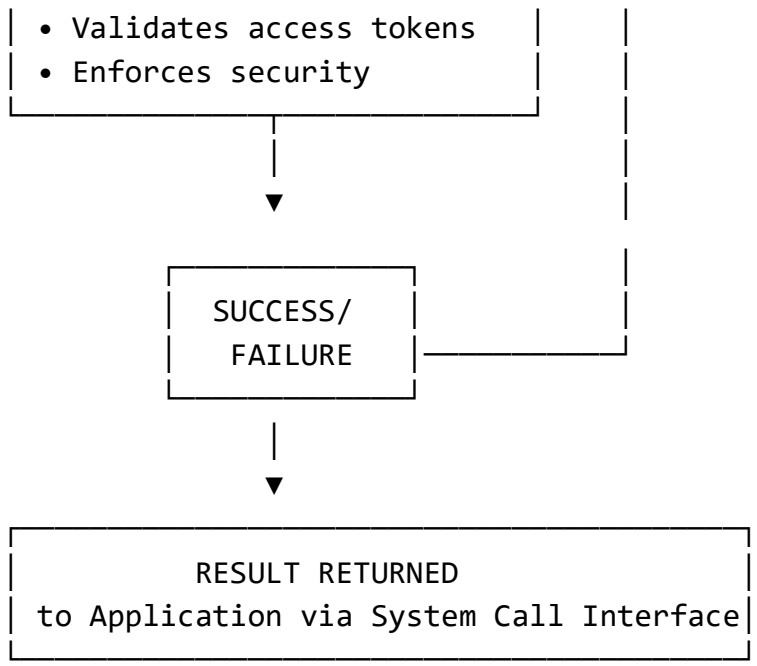
- **Purpose**: High-speed inter-process communication
- **Function**:
  - Enables communication between processes
  - Used by client-server applications
  - More efficient than traditional IPC methods

Drivers (multiple)

- **Purpose**: Hardware and software device interfaces
- **Function**:
  - Translate I/O requests to hardware commands
  - Provide abstraction between OS and hardware
  - Handle specific devices (graphics, storage, USB, etc.)




1. Application requests service
(e.g., "open file", "allocate memory")
SYSTEM CALL INTERFACE
• Validates request
• Checks permissions
• Routes to appropriate kernel module
Process Manager
Memory Manager
I/O Manager
Drivers (Hardware)
OBJECT MANAGER
• Creates/manages objects
• Reference counting
SECURITY REFERENCE MONITOR
• Checks ACLs



API Prefix-to-Subsystem Mapping :-

Nt or Zw : System call interface

Se : Security Reference Monitor

Ob : Object manager

Ps : Process and thread manager

Cm : Configuration manager

Mm : Memory manager

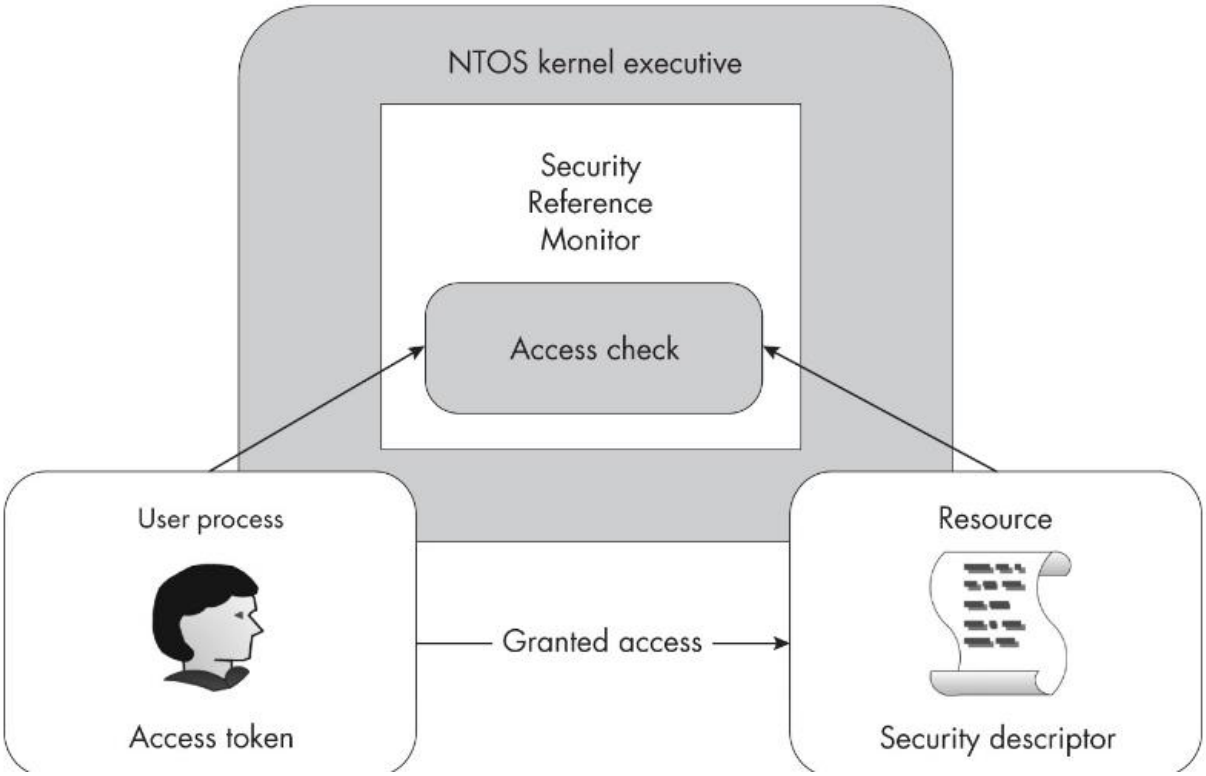
Io : Input/output manager

Ci : Code integrity

==========

The Security Reference Monitor : -

It implements the security mechanisms that restrict which users can access different resources . Without the SRM, you wouldn't be able to prevent other users from accessing your files



Every process running on the system is assigned an access token when it's created.

This access token is managed by the SRM and defines the identity of the user associated with that process.

The SRM can then perform an operation called an access check.

This operation queries a resource's security descriptor, compares it to the process's access token, and either calculates the level of granted access or indicates that access is denied to the caller.

The SRM is also responsible for generating audit events whenever a user accesses a resource

Auditing is disabled by default due to the volume of events it can produce, so an administrator must enable it first.

These audit events can be used to identify malicious behavior on a system as well as to diagnose security misconfigurations.

The SRM expects users and groups to be represented as binary structures called security identifiers (SIDs).

However, passing around raw binary SIDs isn't very convenient for users, who normally refer to users and groups by meaningful names (for example, the user bob or the Users group)

These names need to be converted to SIDs before the SRM can use them.

The task of name–SID conversion is handled by the Local Security Authority Subsystem (LSASS), which

runs inside a privileged process independent from any logged-in users.

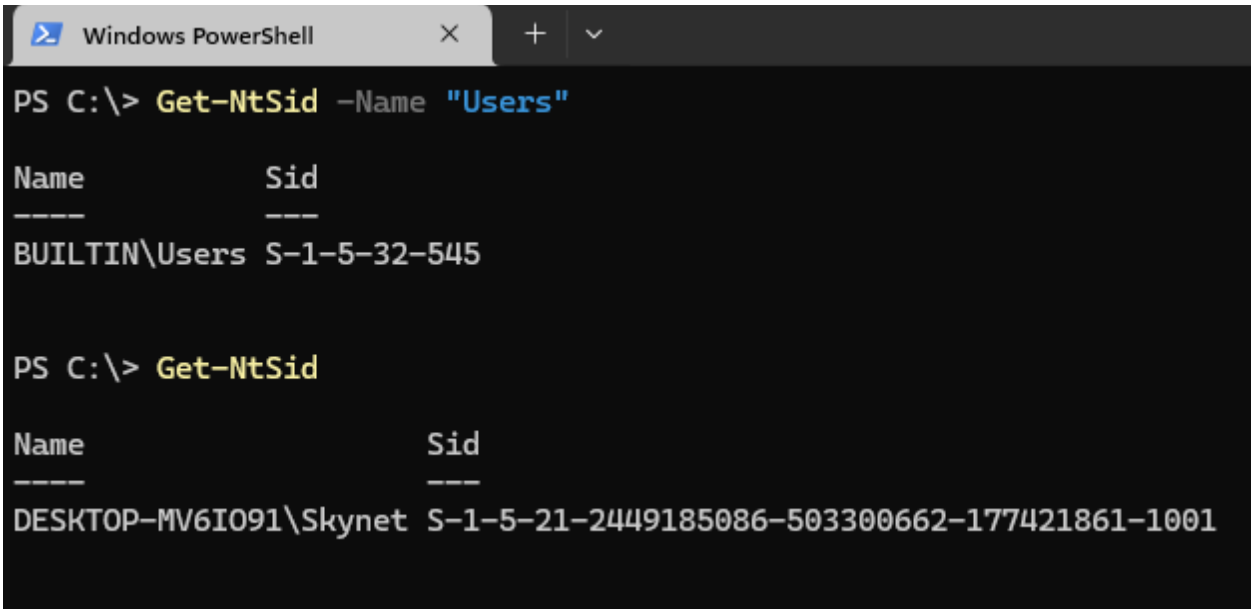
It's infeasible to represent every possible SID as a name, so Microsoft defines the Security Descriptor Definition Language (SDDL) format to represent a SID as a string.

SDDL can represent the entire security descriptor of a resource, but for now we'll just use it to represent the SID

we use PowerShell to look up the Users group name using the Get-NtSid command;

this should retrieve the SDDL string for the SID.

PS> Get-NtSid -Name "Users"



We pass the name of the Users group to Get-NtSid, which returns the fully qualified name, with the local domain BUILTIN attached

The BUILTIN\Users SID is always the same between different Windows systems.

The output also contains the SID in SDDL format, which can be broken down into the following dash-separated parts:

The S character prefix. This indicates that what follows is an SDDL SID.

The version of the SID structure in decimal. This has a fixed value of 1.

The security authority. Authority 5 indicates the built-in NT authority.

Two relative identifiers (RIDs), in decimal. The RIDs (here, 32 and 545)represent the NT authority group.

We can also use Get-NtSid to perform the reverse operation, converting an SDDL SID back to a name

==========

The Object Manager :-

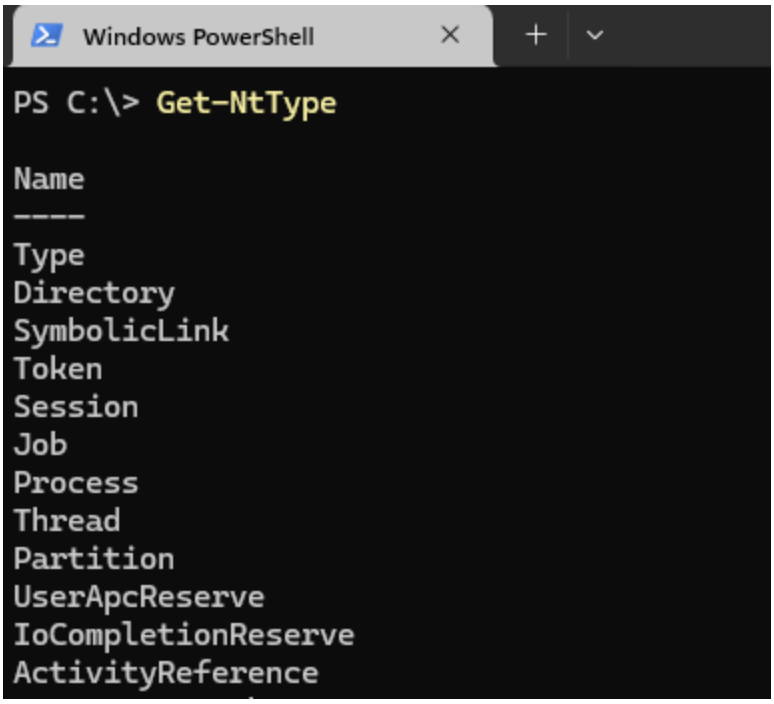
On Windows, everything is an object, meaning that every file, process, and thread is represented in kernel memory as an object structure

for security, each of these objects canhave an assigned security descriptor, which restricts which users can access the object and determines the type of access they have (for example, read or write)

The object manager is the component of the kernel responsible for managing these resource objects, their memory allocations, and their lifetimes.

Object Types :-

The kernel maintains a list of all the types of objects it supports. This is necessary, as each object type has different supported operations and security properties



The Object Manager Namespace :-

As a user of Windows, you typically see your filesystem drives in Explorer.

But underneath the user interface is a whole additional filesystem just for kernel

Objects.

Access to this filesystem, referred to as the object manager namespace (OMNS).

The OMNS is built out of Directory objects.

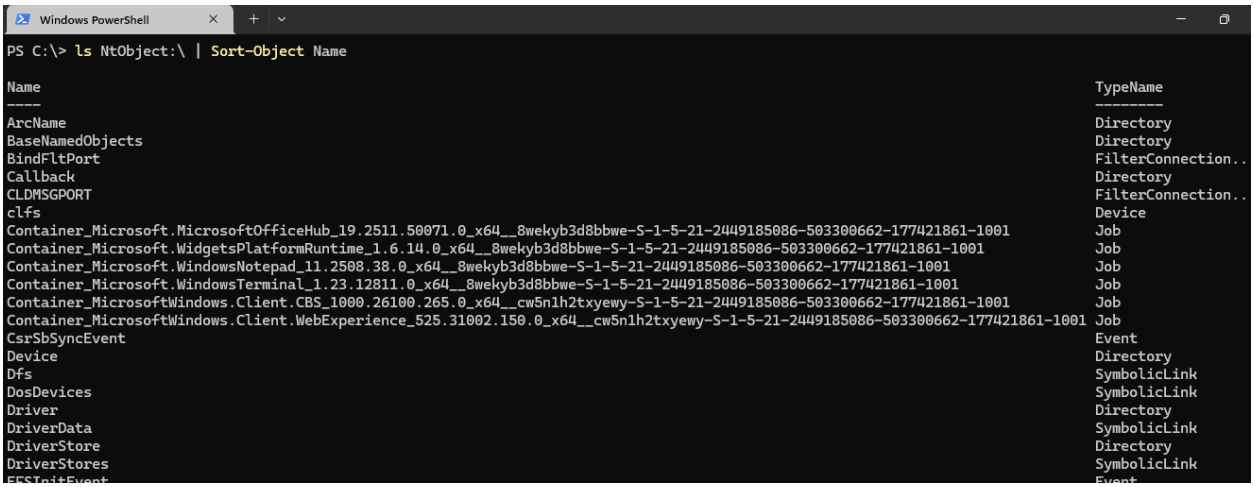
The objects act as if they were in a filesystem, so each directory contains other objects, which you can consider to be files

However, they are distinct from the file directories you're used to.

Each directory is configured with a security descriptor that determines which users can list its contents and which users can create new subdirectories and objects inside it

We can enumerate the OMNS by using a drive provider

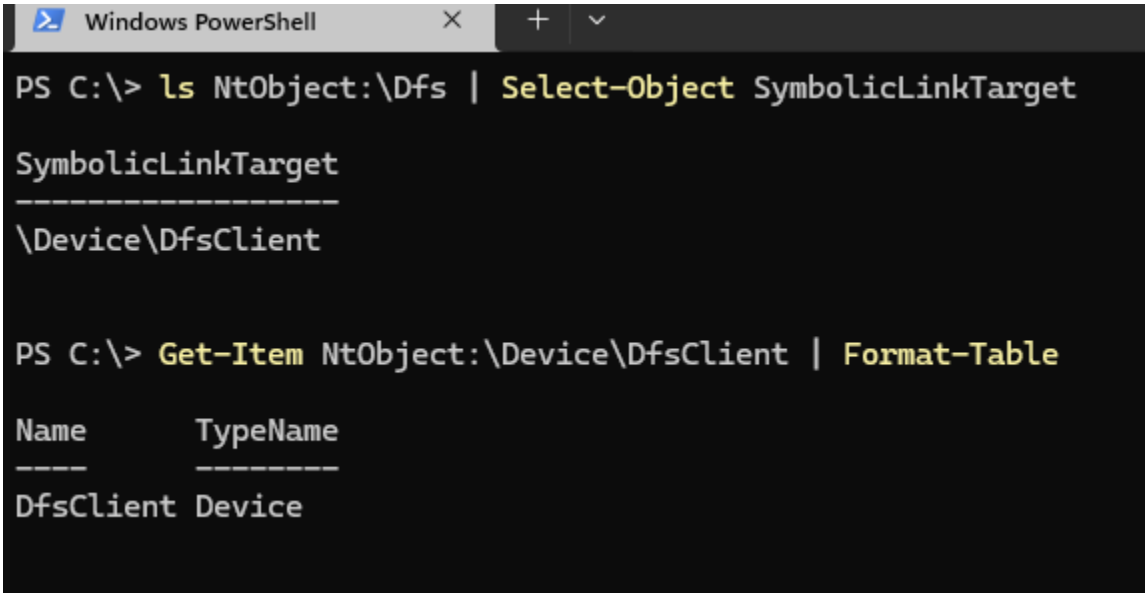
We can enumerate the OMNS by using a drive provider : ls NtObject:\ | Sort-Object Name



We can also see another important type, SymbolicLink.

You can use symbolic links to redirect one OMNS path to another

A SymbolicLink object contains a SymbolicLinkTarget property, which itself contains the target that the link should open.



Well-Known Object Directories and Descriptions :

Path : Description

\BaseNamedObjects : Global directory for user objects

\Device Directory : containing devices such as mounted filesystems

\GLOBAL?? : Global directory for symbolic links, including drive mappings

\KnownDlls : Directory containing special, known DLL mappings

\ObjectTypes : Directory containing named object types

\Sessions : Directory for separate console sessions

\Windows : Directory for objects related to the Window Manager

\RPC Control : Directory for remote procedure call endpoints

==========

System Calls :-

How can we access the named objects in the OMNS from a user-mode application?

If we're in a user-mode application, we need the kernel to access the objects, and we can call kernel-mode code in a user-mode application using the system call interface

Most system calls perform some operation on a specific type of kernel object exposed by the object manager

The name of a system call follows a common pattern. It starts with either Nt or Zw

Common system-call verbs that perform an operation on a kernel object include :

Create : Creates a new object. Maps to New-Nt<Type> PowerShell commands.

Open : Opens an existing object. Maps to Get-Nt<Type> PowerShell commands.

QueryInformation : Queries object information and properties.

SetInformation : Sets object information and properties.

the NtCreateMutant system call creates a new Mutant object :

NTSTATUS NtCreateMutant(

HANDLE* FileHandle,

ACCESS_MASK DesiredAccess,

OBJECT_ATTRIBUTES* ObjectAttributes,

BOOLEAN InitialOwner

);

The first parameter for the system call is an outbound pointer to a HANDLE.

Common in many system calls, this parameter is used to retrieve an opened handle to the object (in this case, a Mutant) when the function succeeds.

We use handles along with other system calls to access properties and perform operations.

In the case of our Mutant object, the handle allows us to acquire and release the lock to synchronize threads

Next is DesiredAccess, which represents the operations the caller wants to be able to perform on the Mutant using the handle.

The access granted depends on the results of the SRM's access check

Object Attribute Flags and Description :

PowerShell : name Description

Inherit : Marks the handle as inheritable.

Permanent : Marks the handle as permanent.

Exclusive : Marks the handle as exclusive if creating a new object. Only the same process can open a handle to the object.

CaseInsensitive : Looks up the object name in a case-insensitive manner.

OpenIf : If using a Create call, opens a handle to an existing object if available.

OpenLink : Opens the object if it's a link to another object; otherwise, follows the link. This is used only by the configuration manager.

KernelHandle : Opens the handle as a kernel handle when used in kernel mode. This prevents user-mode applications from accessing the handle directly.

ForceAccessCheck : When used in kernel mode, ensures all access checks are performed, even if calling the Zw version of the system call.

IgnoreImpersonatedDeviceMap : Disables the device map when impersonating.

DontReparse : Indicates not to follow any path that contains a symbolic link

---

NTSTATUS Codes :-

All system calls return a 32-bit NTSTATUS code. This status code is composed of

multiple components packed into the 32 bits,

NT Status Severity Codes :

Severity name	Value
STATUS_SEVERITY_SUCCESS	0
STATUS_SEVERITY_INFORMATIONAL	1
STATUS_SEVERITY_WARNING	2
STATUS_SEVERITY_ERROR	3

The most significant two bits (31 and 30) indicate the severity of the status code

CC, is the customer code. This is a single-bit flag that indicates whether the status code is defined by Microsoft (a value of 0) or defined by a third party (a value of 1).

R bit, a reserved bit that must be set to 0.

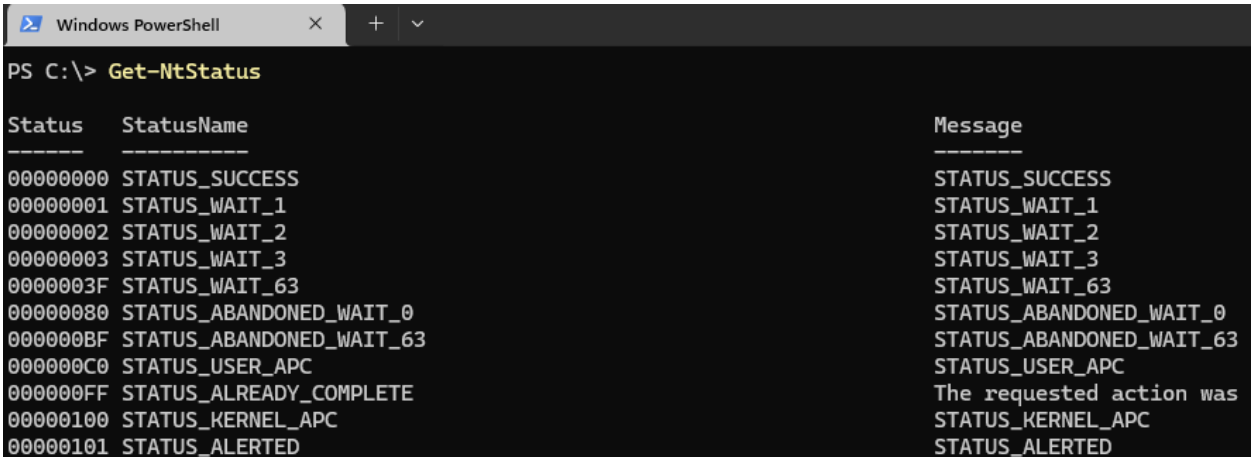
The next 12 bits indicate the facility—that is, the component or subsystem associated with the status code. Microsoft has predefined around 50 facilities for its own purposes

Common Status Facility Values :

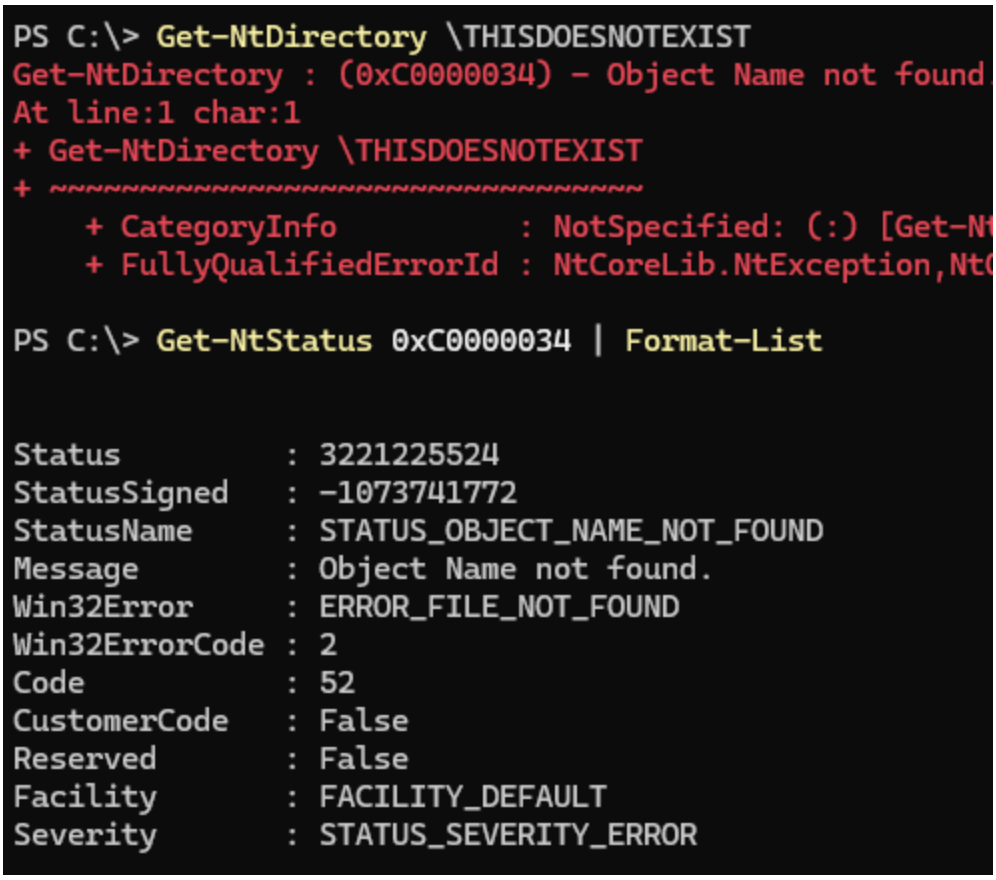
Facility name	Value	Description
FACILITY_DEFAULT	0	The default used for common status codes
FACILITY_DEBUGGER	1	Used for codes associated with the debugger
FACILITY_NTWIN32	7	Used for codes that originated from the Win32 APIs

The final component, the status code, is a 16-bit number chosen to be unique for the facility. It's up to the implementer to define what each number means

The PowerShell module contains a list of known status codes, which we can query using the Get-NtStatus command with no parameters :



When we call a system call via a PowerShell command, its status code is surfaced through a .NET exception. For example, if we try to open a Directory object that doesn't exist, we'll see the exception :



==========--

Object Handles :-

The object manager deals with pointers to kernel memory

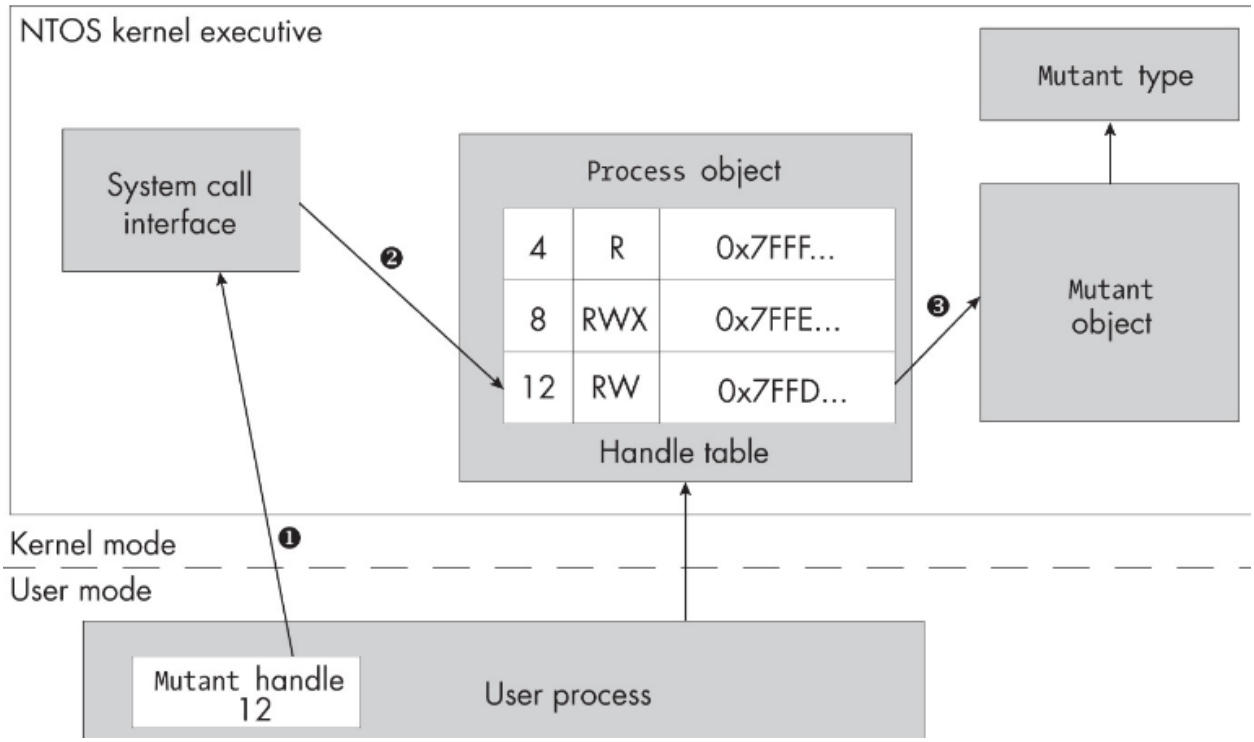
A user-mode application cannot directly read or write to kernel memory, so how can it access an object? It does this using the handle returned by a system call

Each running process has an associated handle table containing three pieces of information:

The handle's numeric identifier

The granted access to the handle; for example, read or write

The pointer to the object structure in kernel memory



When a user process wants to use a handle, it must first pass the handle's value to the system call ,

The system call implementation then calls a kernel API to convert the handle to a kernel pointer by referencing the handle's numeric value in the process's handle table

To determine whether to grant the access, the conversion API considers the type of access that the user has requested for the system call's operation, as well as the type of object being accessed

If the requested access doesn't match the granted access recorded in the handle table entry, the API will return STATUS_ACCESS_DENIED and the conversion operation will fail. Likewise, if the object types don't match , the API will return STATUS_OBJECT_TYPE_MISMATCH

The access check ensures that the user can't perform an operation on a handle to which they don't have access

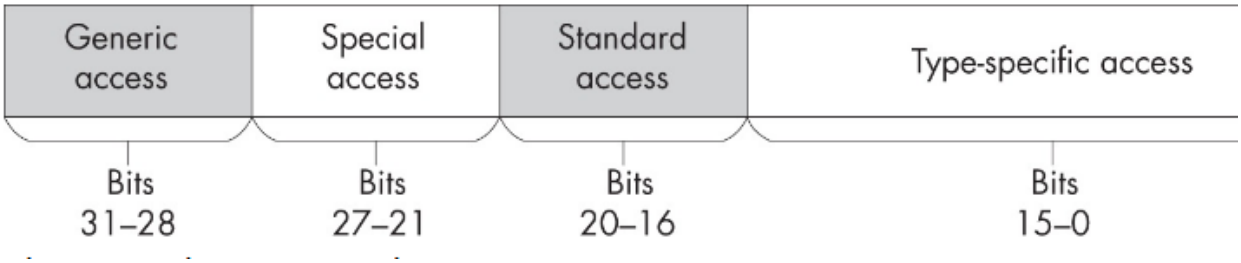
The type check ensures the user hasn't passed an unrelated kernel object type, which might result in type confusion in the kernel, causing security issues such as memory

Corruption

Access Masks :-

The granted access value in the handle table is a 32-bit bitfield called an access Mask

An access mask has four components :



The most important one is the 16-bit type-specific access component, which defines the operations that are allowed on a particular kernel object type

the standard access component of the access mask defines operations that can apply to any object type. These operations include:

Delete : Removes the object;

ReadControl : Reads the security descriptor information for the object

WriteDac : Writes the security descriptor's discretionary access control (DAC) to the object

WriteOwner : Writes the owner information to the object

Synchronize : Waits on the object

Before this are the reserved and special access bits. Most of these bits are

reserved, but they include two access values:

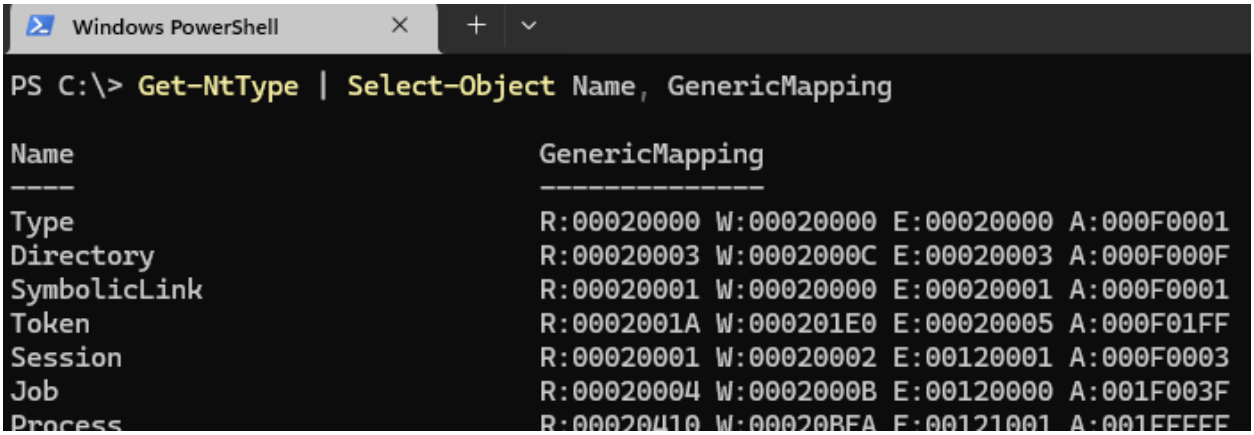
AccessSystemSecurity : Reads or writes audit information on the object

MaximumAllowed : Requests the maximum access to an object when performing an access check

the four high-order bits of the access mask (the generic access component) are used only when requesting access to a kernel object using the system call's DesiredAccess parameter. There are four broad categories of access: GenericRead, GenericWrite, GenericExecute, and GenericAll.

This means you'll never receive access to a handle with GenericRead; instead, you'll be granted access to the specific access mask that represents read operations for that type.

To facilitate the conversion, each type contains a generic mapping table, which maps the four generic categories to type-specific access.

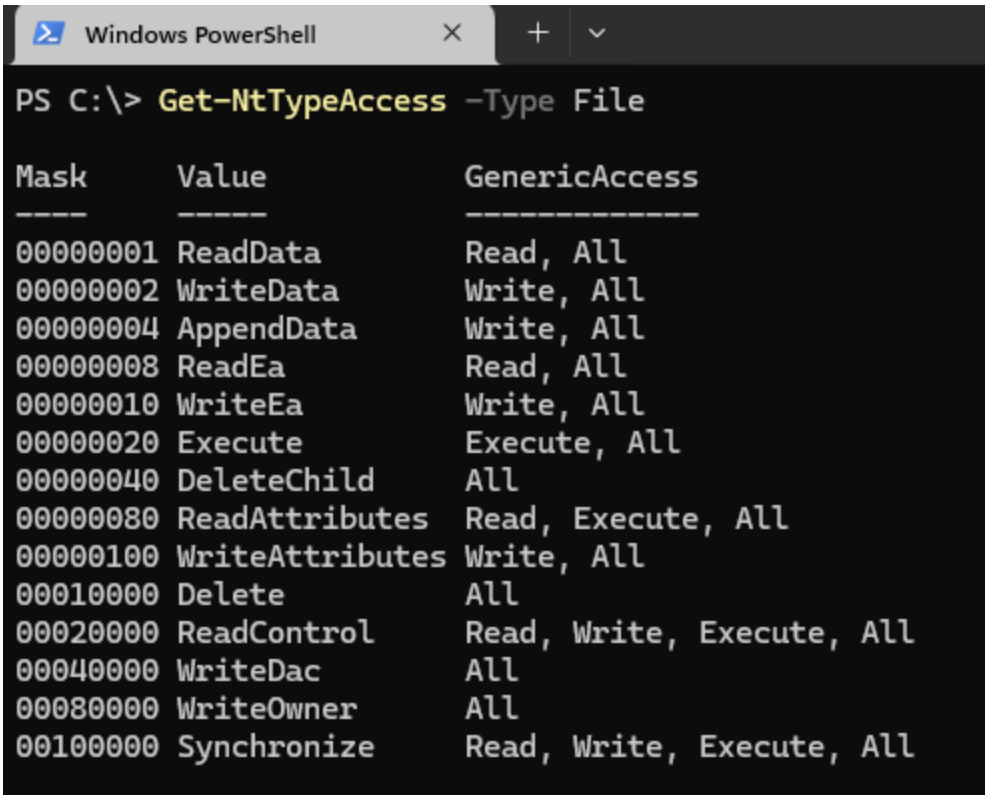


The type data doesn't provide names for each specific access mask.

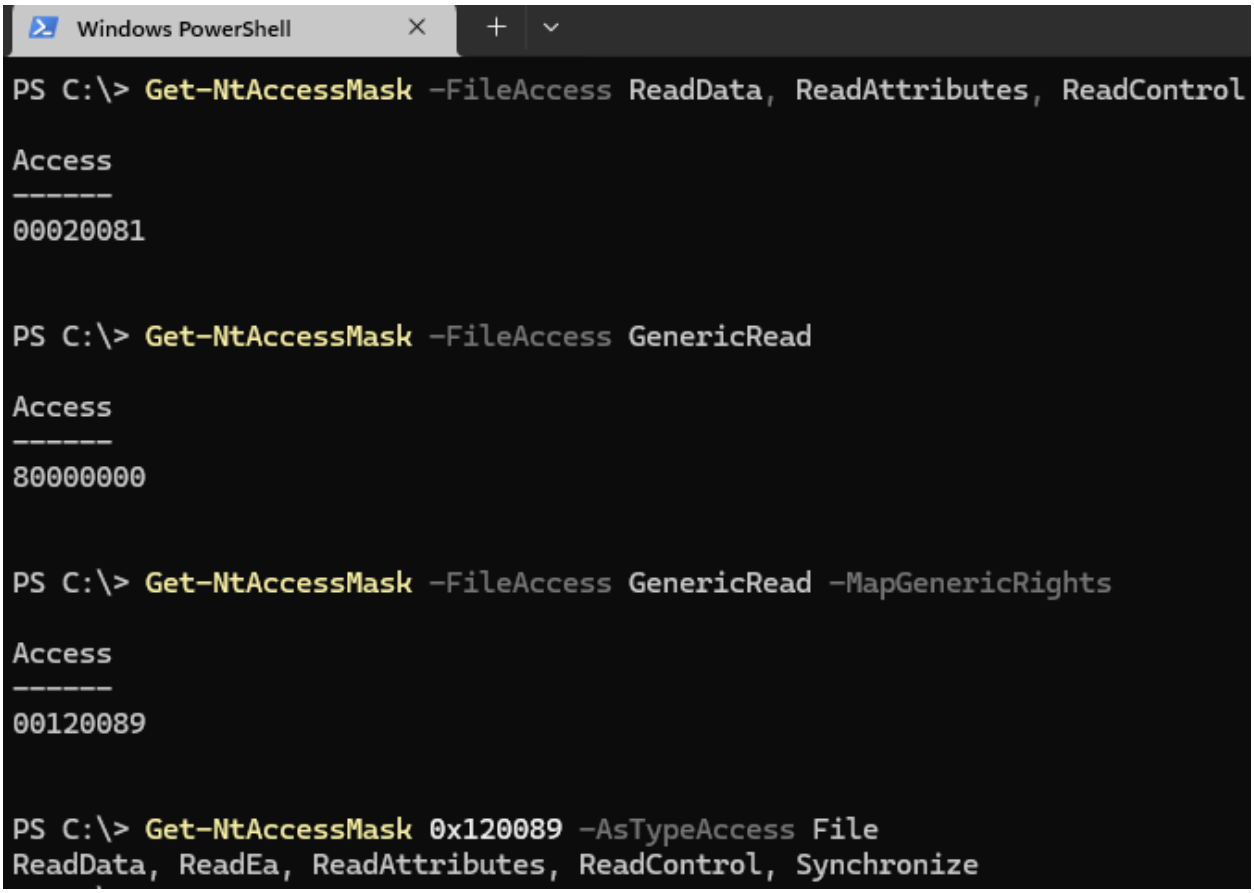
However, for all common types, the PowerShell module provides an enumerated

type that represents the type-specific access. We can access this type through the

Get-NtTypeAccess command

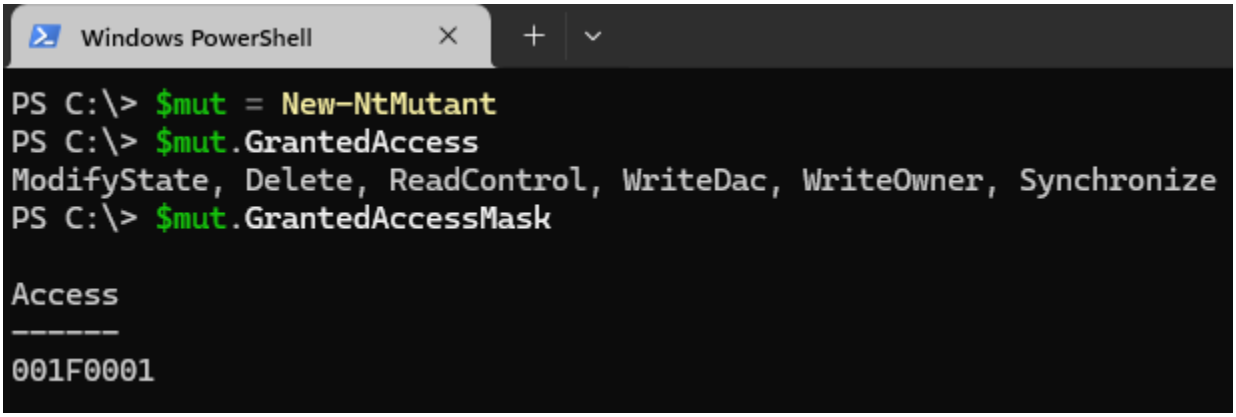


convert between a numeric access mask and specific object types :-



you can query an object handle's granted access

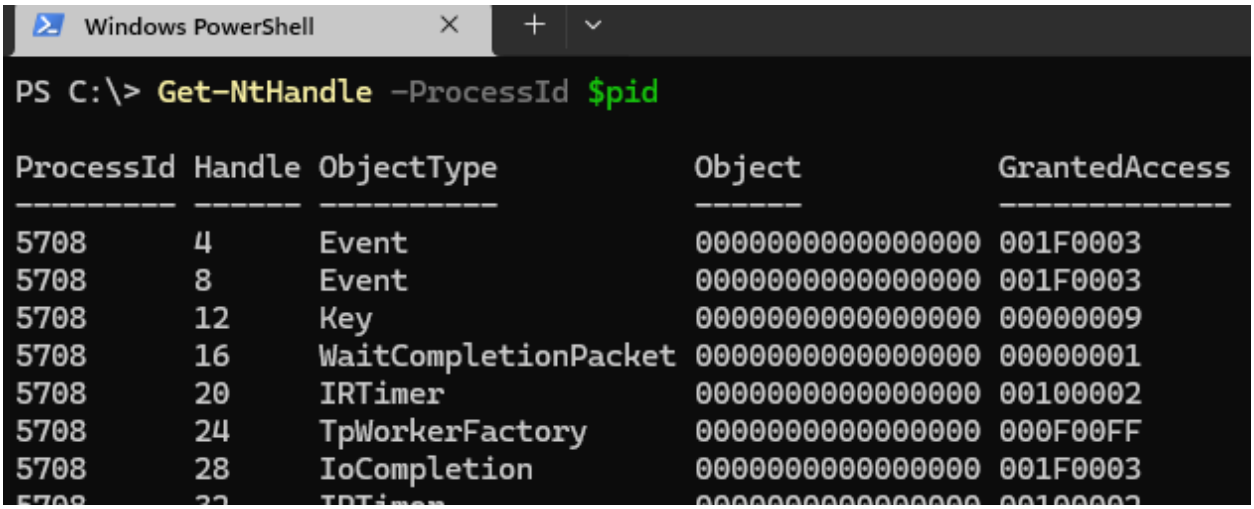
mask through the PowerShell object's GrantedAccess property :



The kernel provides a facility to dump all handle table entries on the system

through the NtQuerySystemInformation system call. We can access the handle

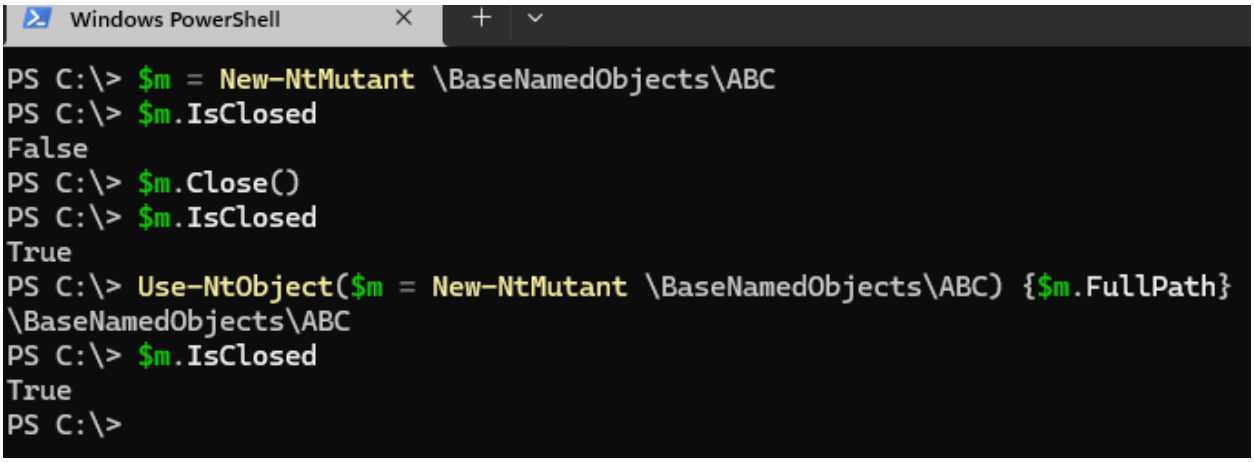
table from PowerShell using the Get-NtHandle command



Once an application has finished with a handle, it can be closed using the NtClose API.

If you've received a PowerShell object from a Get or New call, then you can call the Close method on the object to close the handle.

You can also close an object handle automatically in PowerShell by using the Use-NtObject command to invoke a script block that closes the handle once it finishes executing



You should get into the habit of manually closing handles, though; otherwise, you might have to wait a long time for the resources to be released, as the garbage collector could run at any time.

Handle Duplication :-

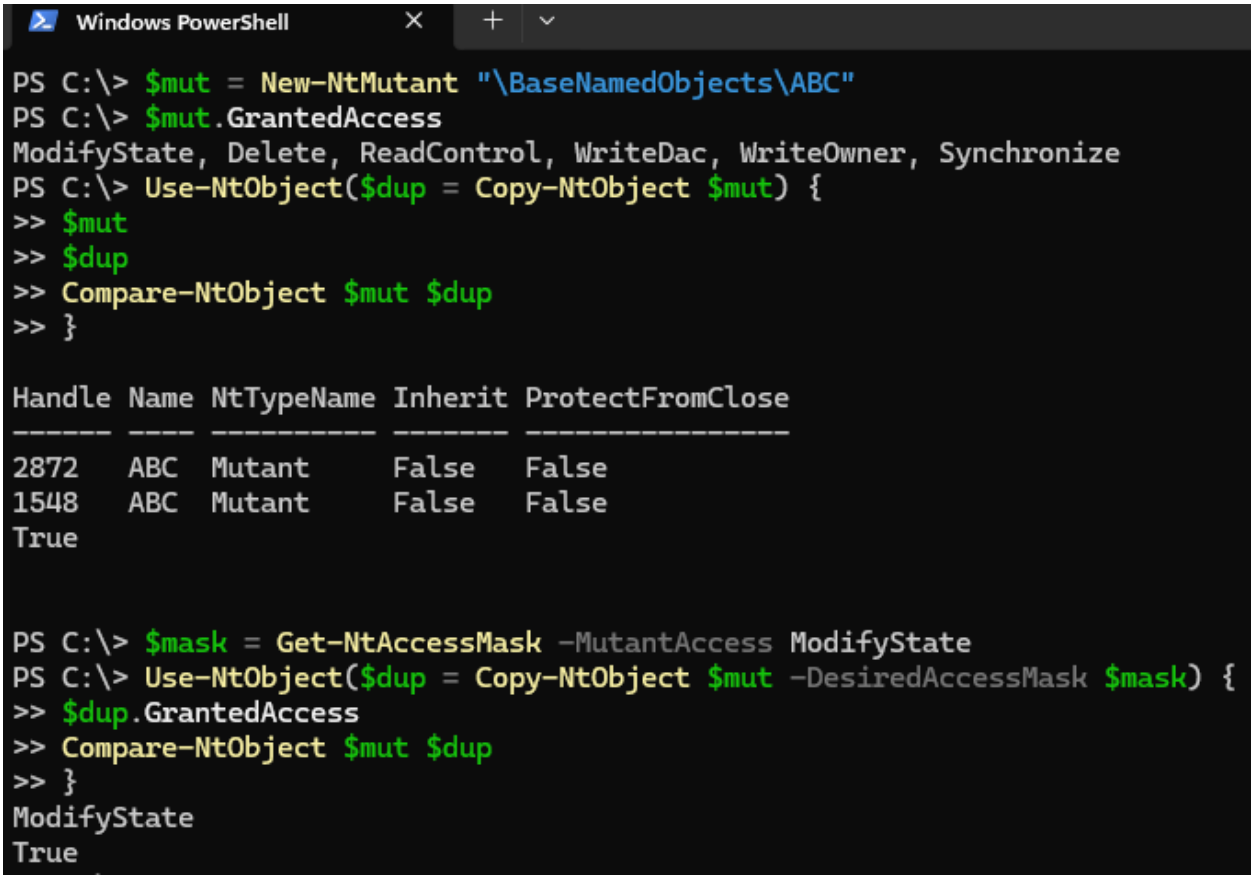
You can duplicate handles using the NtDuplicateObject system call

The primary reason you might want to do this is to allow a process to take an additional reference to a kernel object

The kernel object won't be destroyed until all handles to it are closed, so creating a new handle maintains the kernel object.

Handle duplication can additionally be used to transfer handles between processes if the source and destination process handles have DupHandle access.

You can also use handle duplication to reduce the access rights on a handle



Line-by-Line Explanation:

Line 1: Create a Mutex

```
PS C:\> $mut = New-NtMutant "\BaseNamedObjects\ABC"
```

- **`$mut`** = Variable storing the mutex object
- **`New-NtMutant`** = Creates a new mutex kernel object
- **`"\BaseNamedObjects\ABC"`** = Name in Windows Object Manager namespace

- **** = Root of object namespace
- **BaseNamedObjects** = Directory for named objects visible to all sessions
- **ABC** = Name of your mutex

What happens in kernel:

1. Object Manager creates a mutex object
2. Stores it at `\BaseNamedObjects\ABC`
3. Returns a handle to your process
4. Handle value stored in `$mut`

Line 2: Check Access Rights

```
PS C:\> $mut.GrantedAccess
ModifyState, Delete, ReadControl, WriteDac, WriteOwner, Synchronize
```

- **GrantedAccess** = What permissions your handle has
- **ModifyState** = Can release/acquire the mutex
- **Delete** = Can delete the mutex object
- **ReadControl** = Can read security descriptor
- **WriteDac** = Can modify security descriptor (change permissions)
- **WriteOwner** = Can change object owner
- **Synchronize** = Can wait on the mutex (thread synchronization)

Default permissions = Full control minus dangerous rights

Lines 3-8: Duplicate and Compare

```
PS C:\> Use-NtObject($dup = Copy-NtObject $mut) {
>> $mut
>> $dup
>> Compare-NtObject $mut $dup
>> }

Handle Name NtTypeName Inherit ProtectFromClose
------ ---- ---------- ------- ----------------
```

```
2872   ABC  Mutant     False   False
1548   ABC  Mutant     False   False
True
```

Breakdown:

```
Use-NtObject($dup = Copy-NtObject $mut) { ... }
```

- **`Copy-NtObject $mut`** = Duplicates the handle
- **`$dup =`** = Stores duplicate in new variable
- **`Use-NtObject(...) { ... }`** = Ensures $dup is automatically closed after block

Inside the block:

```
$mut  # Output first mutex properties
```

- **Handle 2872** = Original handle value (unique in your process)
- **Name ABC** = Same named object
- **NtTypeName Mutant** = Type is mutex
- **Inherit False** = Child processes won't inherit this handle
- **ProtectFromClose False** = Handle can be closed

```
$dup  # Output duplicate properties
```

- **Handle 1548** = DIFFERENT handle value!
- Same name, type, but different handle number

```
Compare-NtObject $mut $dup
```

- Returns **True** = Both handles point to SAME kernel object
- Different handles, same underlying mutex

Lines 9-10: Create Access Mask

```
PS C:\> $mask = Get-NtAccessMask -MutantAccess ModifyState
```

- **`$mask`** = Variable storing access rights
- **`Get-NtAccessMask`** = Creates bitmask of permissions
- **`-MutantAccess ModifyState`** = Only request "ModifyState" permission
- **Result**: Bitmask where only ModifyState bit is set (0x00000001)

Why? = To create a restricted duplicate handle

Lines 11-15: Restricted Duplicate

```
PS C:\> Use-NtObject($dup = Copy-NtObject $mut -DesiredAccessMask
$mask) {
>> $dup.GrantedAccess
>> Compare-NtObject $mut $dup
>> }
ModifyState
True
```

Key change:

```
-DesiredAccessMask $mask
```

- Creates duplicate with ONLY `ModifyState` permission
- Original handle had 6 permissions, duplicate has 1

Output:

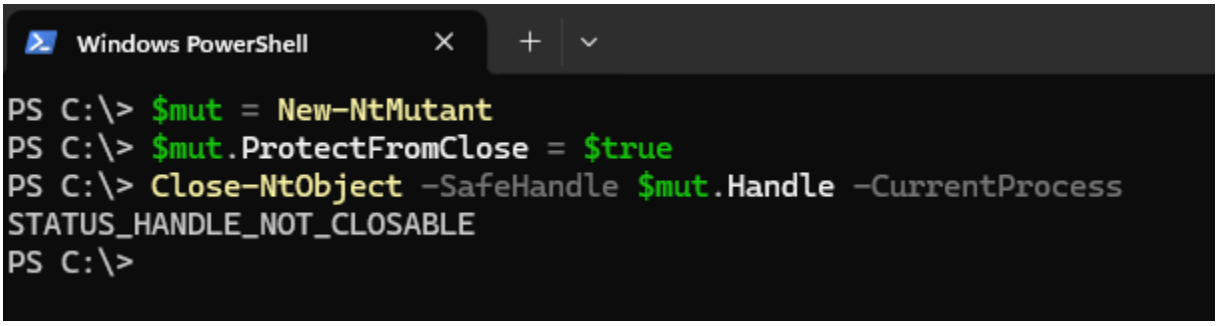
```
$dup.GrantedAccess  # Shows only "ModifyState"
```

- Proof: Duplicate is restricted
- Can only release/acquire mutex
- CANNOT delete, change security, etc.

```
Compare-NtObject $mut $dup  # Still "True"
```

- Still same kernel object
- Just different access rights on the handle

Setting ProtectFromClose protects the handle from being closed



==========

Query and Set Information System Calls :-

A kernel object typically stores information about its state

To allow us to retrievethis information, the kernel could have implemented a specific “get process creation time” system call.

However, due to the volume of information stored for the various types of objects, this approach would quickly become unworkable.

Instead, the kernel implements generic Query and Set information system calls whose parameters follow a common pattern for all kernel object types.

NTSTATUS NtQueryInformationProcess(

HANDLE Handle,

PROCESS_INFORMATION_CLASS InformationClass,

PVOID Information,

ULONG InformationLength,

PULONG ReturnLength

)

All Query information system calls take an object handle as the first parameter

The second parameter, InformationClass, describes the type of process information to query

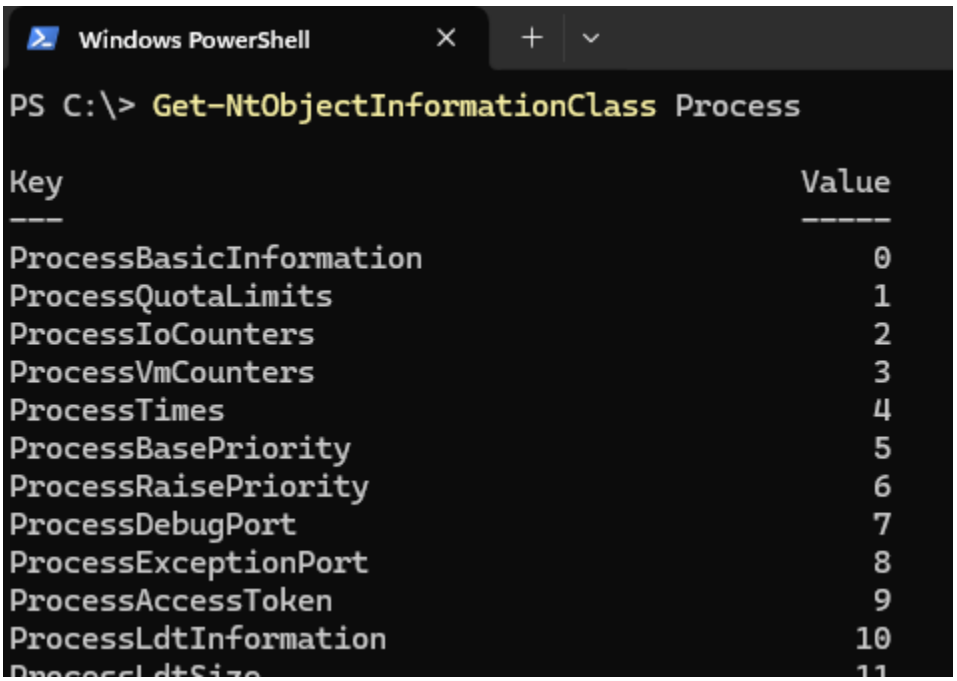
The information class is an enumerated value; the SDK specifies the names of the information classes

For every information class, we need to specify an opaque buffer to receive the queried information, as well as the length of the buffer

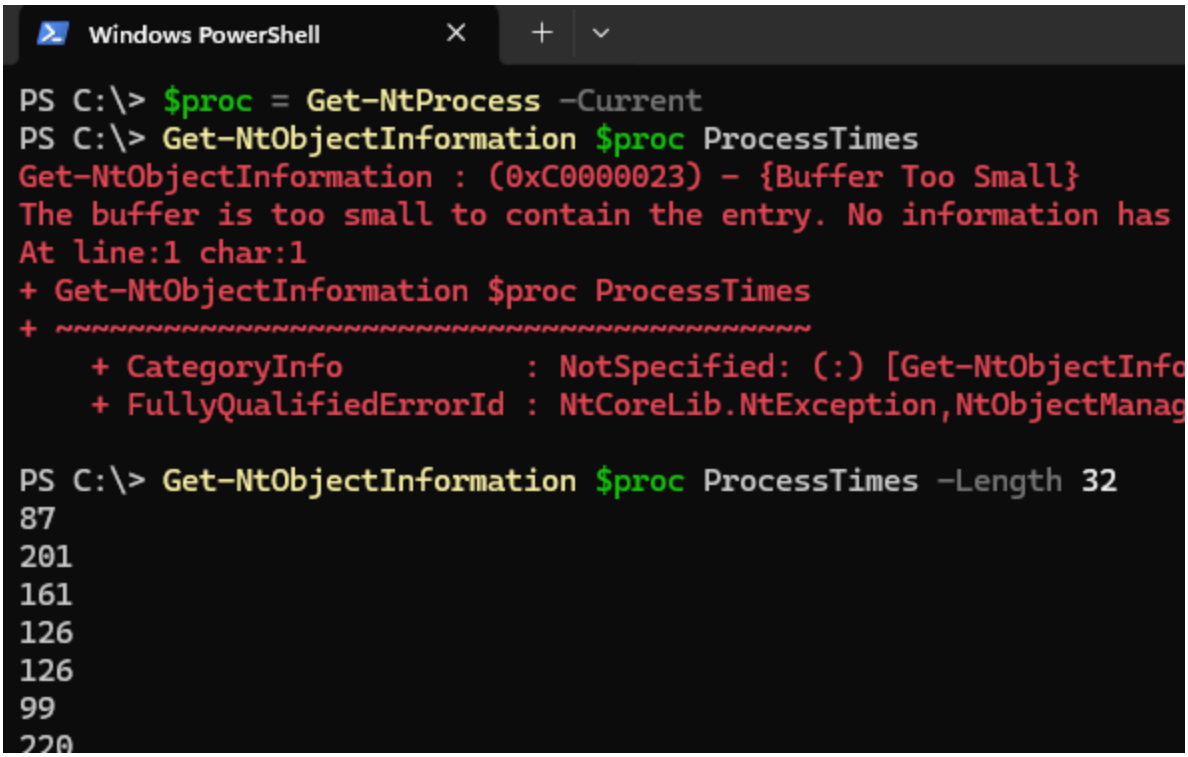
The system call also returns a length value, which serves two purposes: it indicates how much of the buffer was populated if the system call was successful, and if the system call failed, it indicates how big the buffer needs to be

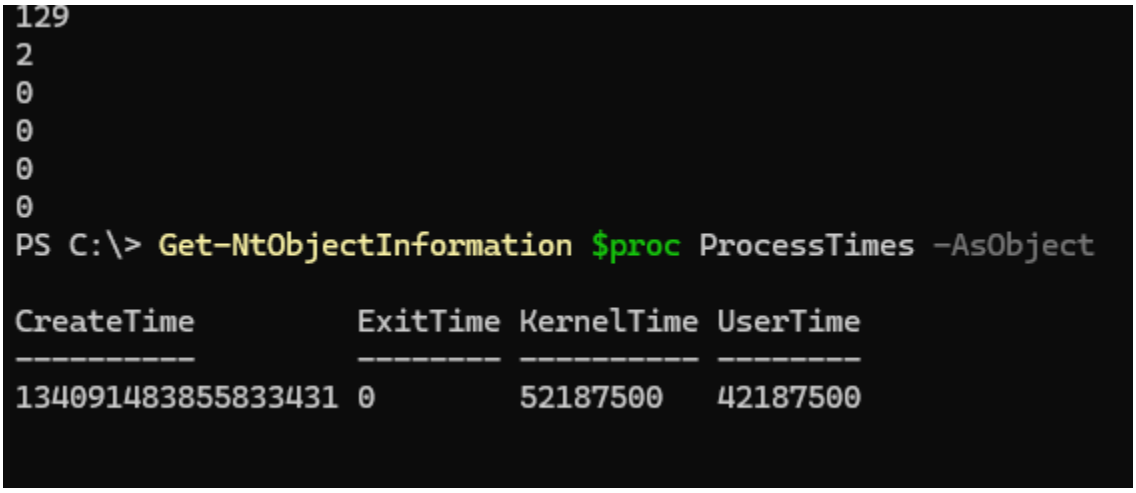
In the PowerShell module, you can query a type's information class names

using the Get-NtObjectInformationClass command,



To call the Query information system call, use Get-NtObjectInformation, specifying an open object handle and the information class. To call SetInformation, use Set-NtObjectInformation





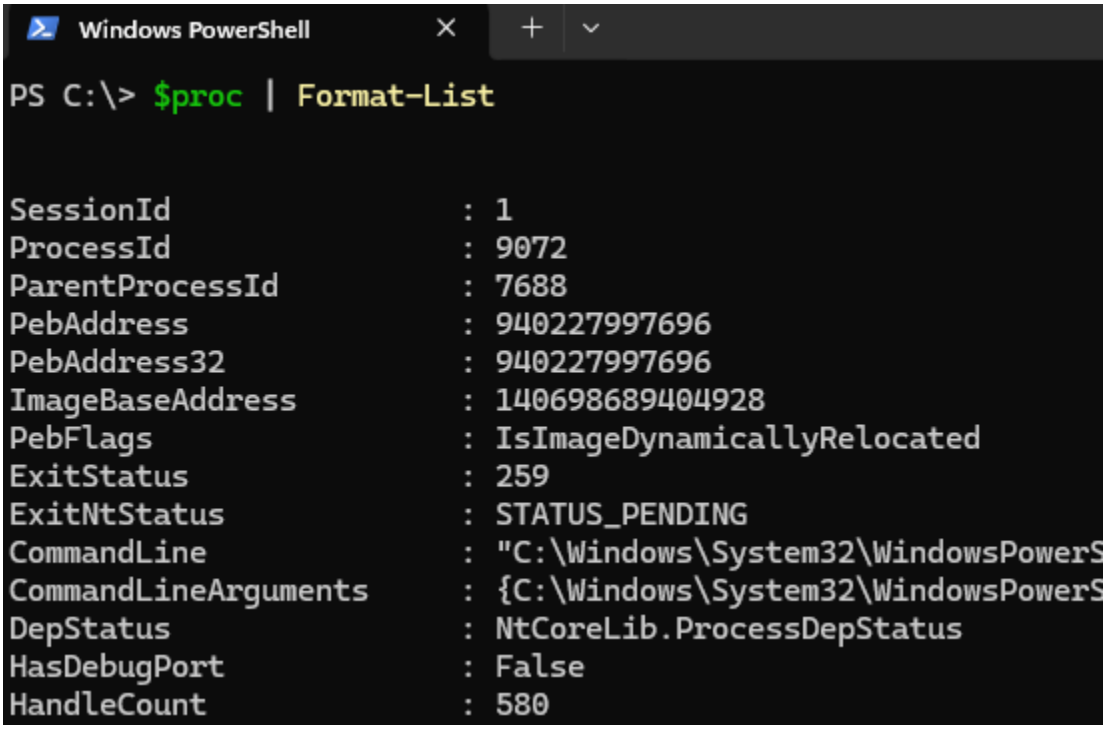
The Process type doesn't set the return length for the ProcessTimes information class, so if you don't specify any length, the operation generates a STATUS_BUFFER_TOO_SMALL error

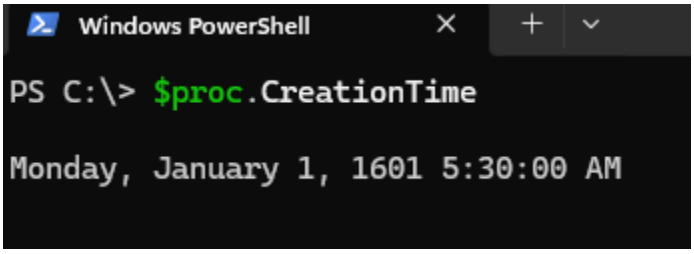
However, through inspection or brute force, you can discover that the length of the data is 32 bytes. Specifying this value using the Length parameter allows the query to succeed and return the data as an array of bytes

For many information classes, the Get-NtObjectInformation command knows the size and structure of the query data. If you specify the AsObject parameter , you can get a preformatted object rather than an array of bytes.

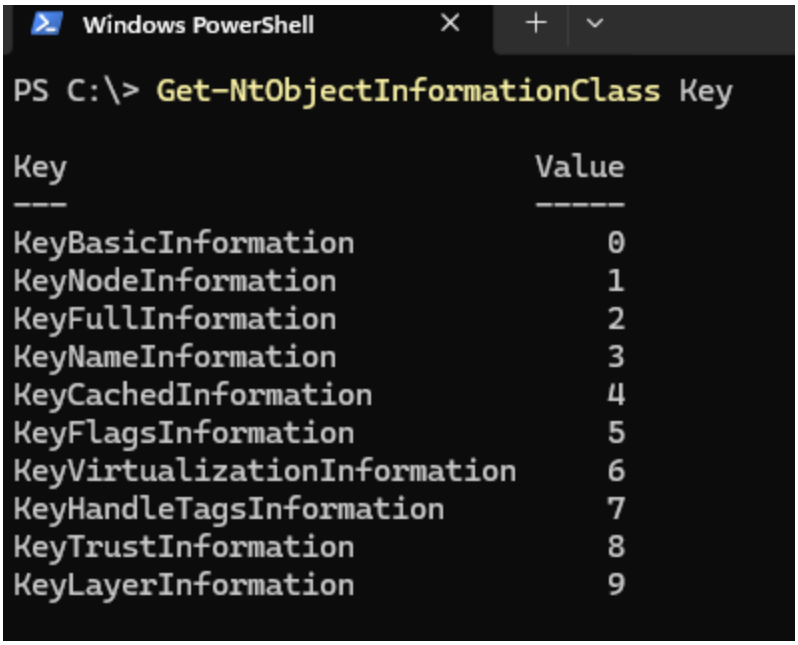
You can easily inspect properties by accessing them on the object or using

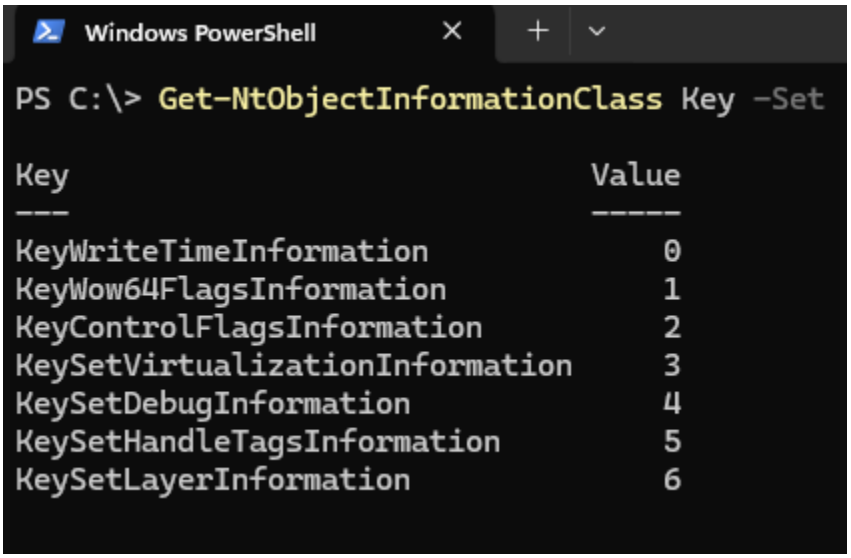
the Format-List command :





The QueryInformation and SetInformation classes for a type typically have the same enumerated values. The kernel can restrict the information class's enumerated values to one type of operation, returning the STATUS_INVALID _INFO_CLASS status code if it's not a valid value





==========

The Input/Output Manager :-

The input/output (I/O) manager provides access to I/O devices through device Drivers.

The primary purpose of these drivers is to implement a filesystem , The I/O manager supports other kinds of drivers, for devices such as keyboards and video cards, but these other drivers are really just filesystem drivers in disguise.

You can manually load a new driver through the NtLoadDriver system call or do so automatically using the Plug and Play (PnP) manager.

You can list the contents of this directory only if you're an administrator. Fortunately, as a

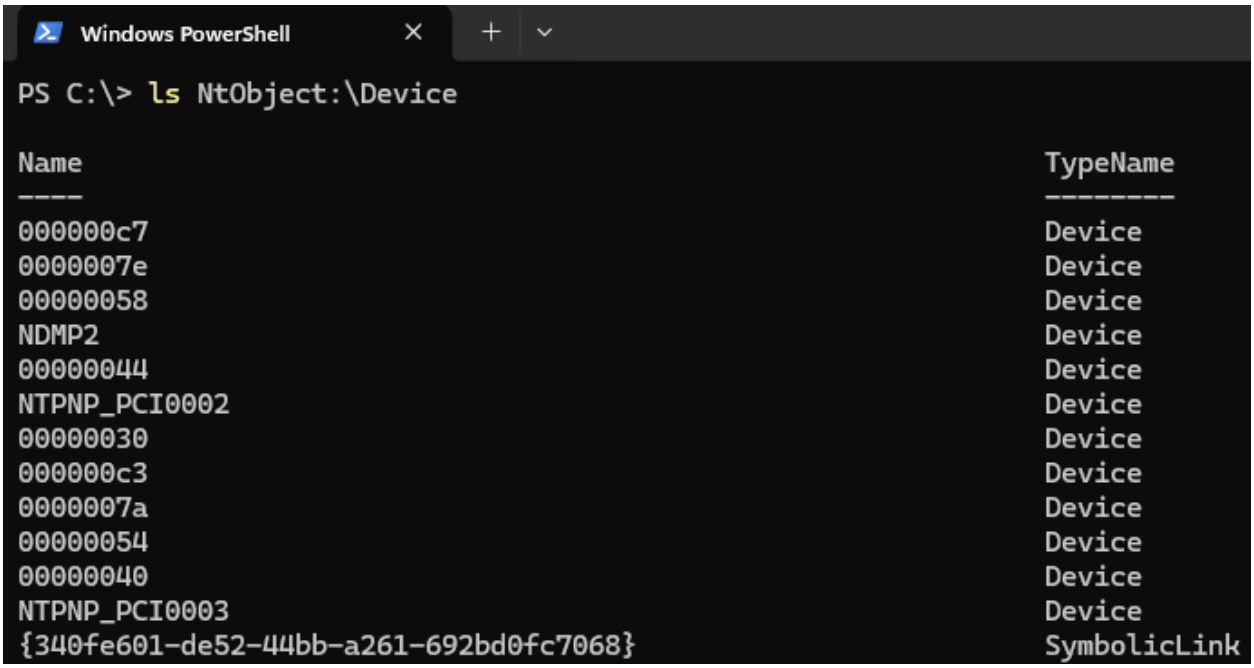
normal user, you don't need to access anything in the Driver directory

Instead, you can interact with the driver through a Device object, normally created in the

Device directory.

Drivers are responsible for creating new Device objects using the IoCreateDevice API. A driver can have more than one Device object associated with it; it may also have zero associated Device objects if it doesn't require user interaction.

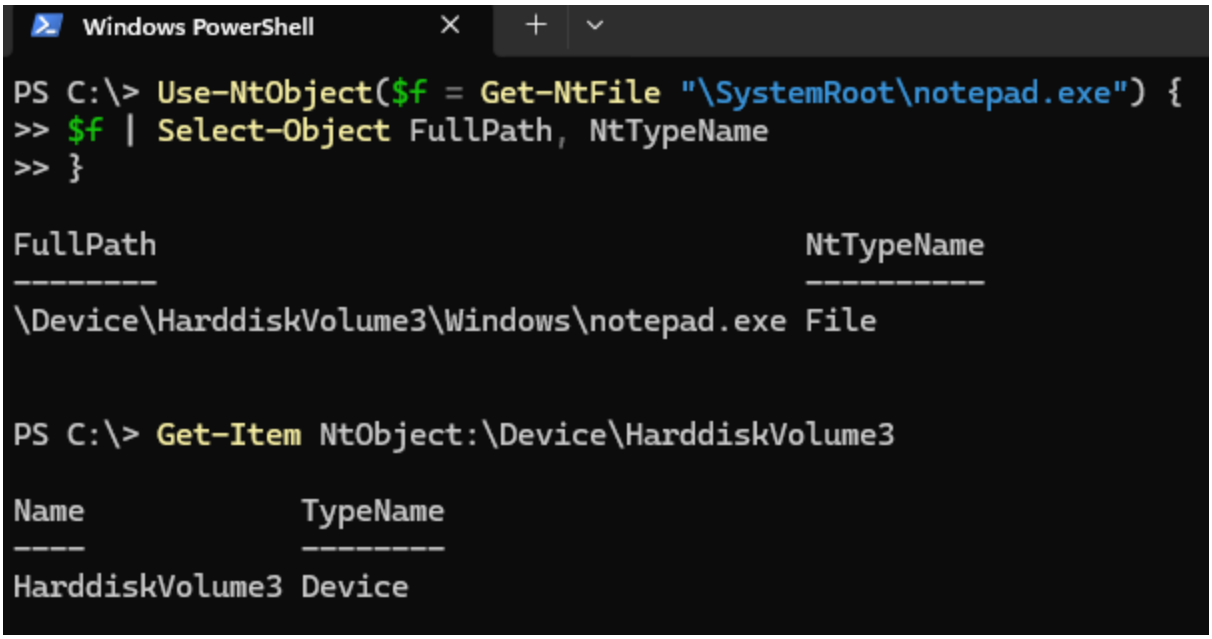
we can list the contents of the Device directory as a normal user through the OMNS



However, if you go looking for a system call with Device in the name, you'll come up empty. That's because we don't interact with the I/O manager usingdedicated system calls; rather, we use File object system calls such as NtCreateFile

We can access these system calls through New-NtFile and Get-

NtFile, which create and open files, :



The SystemRoot symbolic link points to the Windows directory on the system drive.

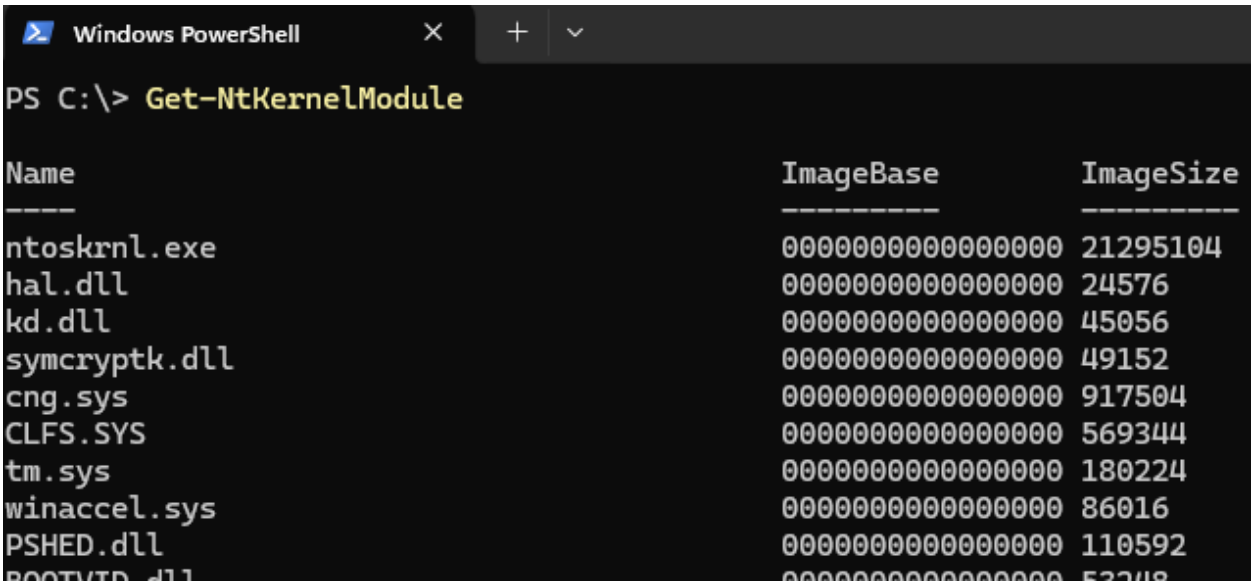
As the SystemRoot symbolic link is part of the OMNS, the OMNS initially handles file access ,

With an open handle, we can select the full path to the file and the type name

Once the object manager finds the Device object, it hands off responsibility for the rest of the path to the I/O manager, which calls an appropriate method inside the kernel driver.

We can list the drivers loaded into the kernel using the Get-NtKernelModule

Command :



Windows does not implement core network protocols like TCP/IP using built-in system calls

Instead, Windows has an I/O manager driver, the Ancillary Function Driver (AFD), which provides access to networking services for an application. You

don't need to deal with the driver directly; Win32 provides a BSD sockets-style API, called WinSock, to handle access to it

In addition to the standard internet protocol suite, such as TCP/IP, AFD also implements other network socket types, such as Unix sockets and bespoke Hyper-V sockets for communication with virtual machines.

----==========

The Process and Thread Manager :-

All user-mode code lives in the context of a process, each of which has one or more threads that control the execution of the code

Processes and threads are both securable resources, if you could access a process, you

could modify its code and execute it in the context of a different user identity.

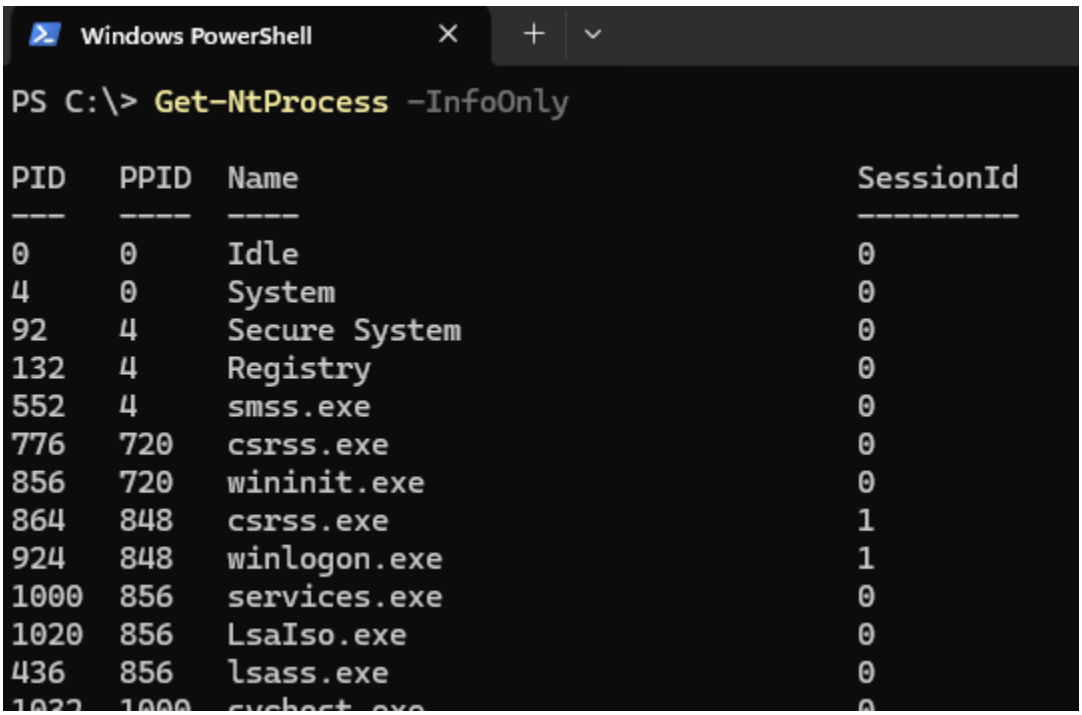
So, unlike most other kernel objects, you can't open a process or thread by name. Instead, you must open them via a unique, numeric process ID (PID) or thread ID (TID)

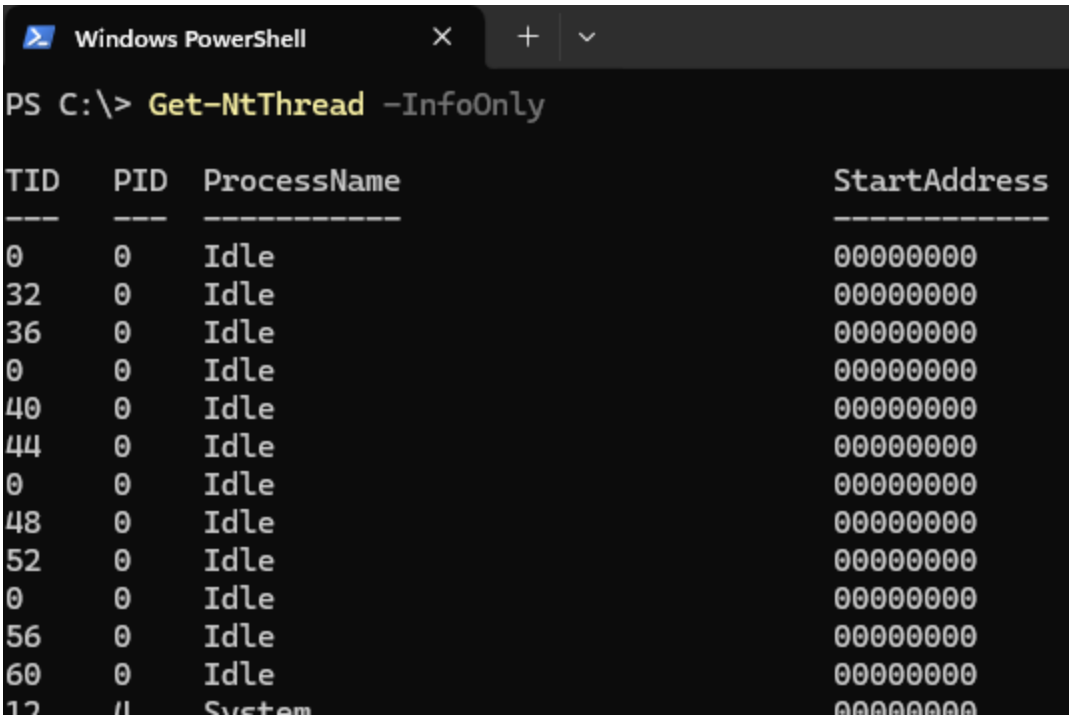
the NtQuerySystemInformation system call provides the SystemProcessInformation information class, which lets us enumerate processes and threads without having access to the Process object.

We can access the list of processes and threads by using the Get-NtProcess and Get-NtThread commands and passing them the InfoOnly parameter

We can also use the built-in Get-Process command to produce a similar output.

Each of the returned objects has a Threads property that we can query for the thread information.

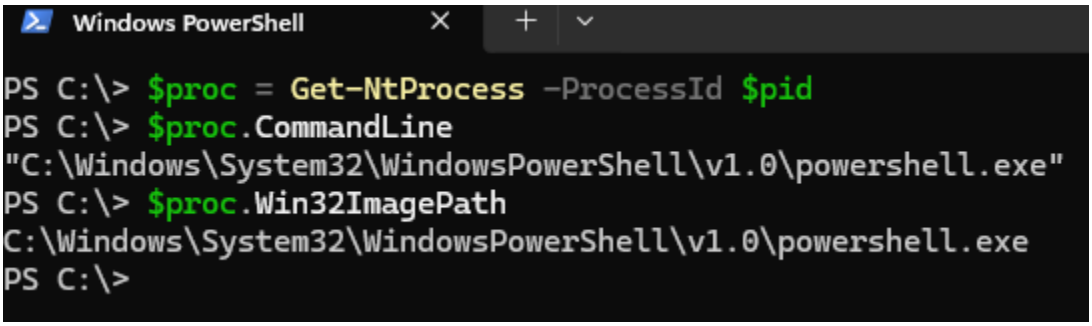




The first two processes listed in the output are special. The first is the Idle process, with PID 0. This process contains threads that execute when the operating system is idle, hence its name . It's not a process you'll need to deal with regularly.

The System process, with PID 4, is important because it runs entirely in kernel mode. When the kernel or a driver needs to execute a background thread, the thread is associated with the System process.

To open a process or thread, we can pass Get-NtProcess or Get-NtThread the PID or TID we want to open. The command will return a Process or Thread object that we can then interact with



When you open a Process or Thread object using its ID, you'll receive a handle. For convenience, the kernel also supports two pseudo handles that refer to the current process and the current thread

The current process's pseudo handle is the value -1 converted to a handle, and for the current thread, it's -2.

==========

The Memory Manager :-

Every process has its own virtual memory address space for a developer to use as they see fit

A 32-bit process can access up to 2GB of virtual memory address space (4GB on 64-bit Windows), while a 64-bit process can access up to 128TB

The kernel's memory manager subsystem controls the allocation of this address space

The virtual memory space is shared by memory allocations, and it stores each process's running state as well as its executable code.

Each memory allocation can have a range of protection states, such as ReadOnly or ReadWrite, ExecuteReadWrite. which must be set according to the memory's purpose.

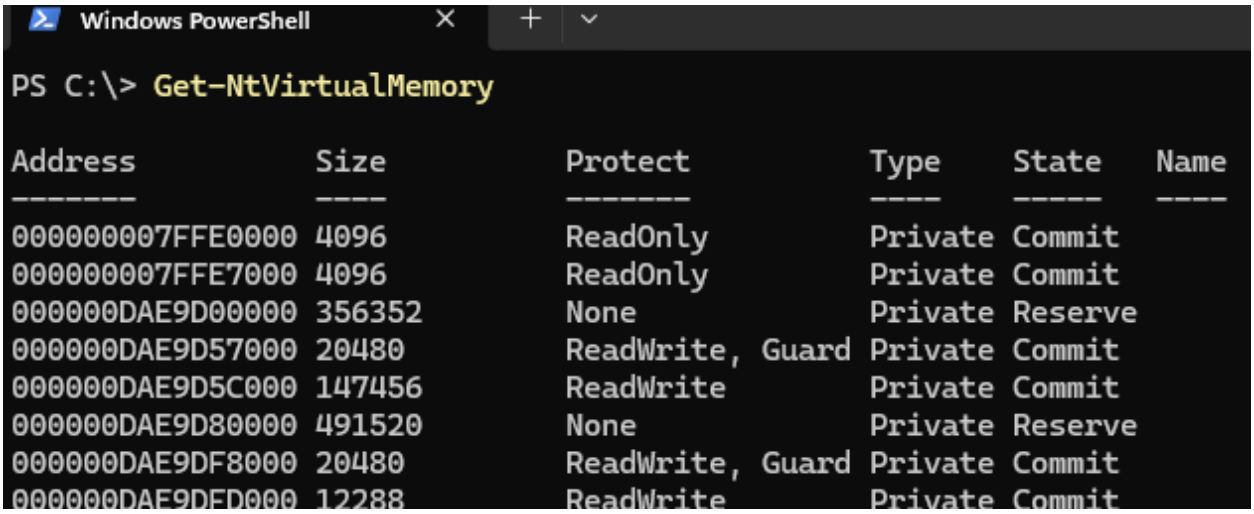
You can query all memory status information for a process by calling NtQueryVirtualMemory, if you have the QueryLimitedInformation access right on the process handle.

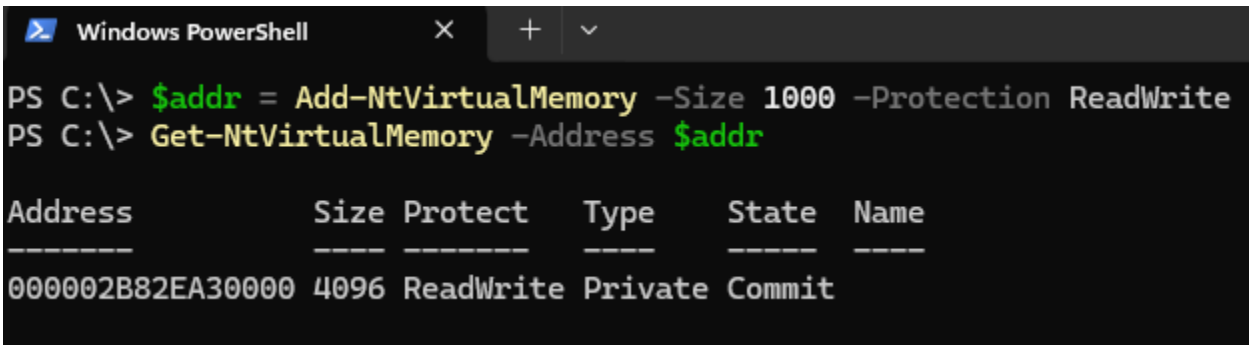
However, reading and writing the memory data requires the VmRead and VmWrite access rights, respectively, and a call to NtReadVirtualMemory and NtWriteVirtualMemory

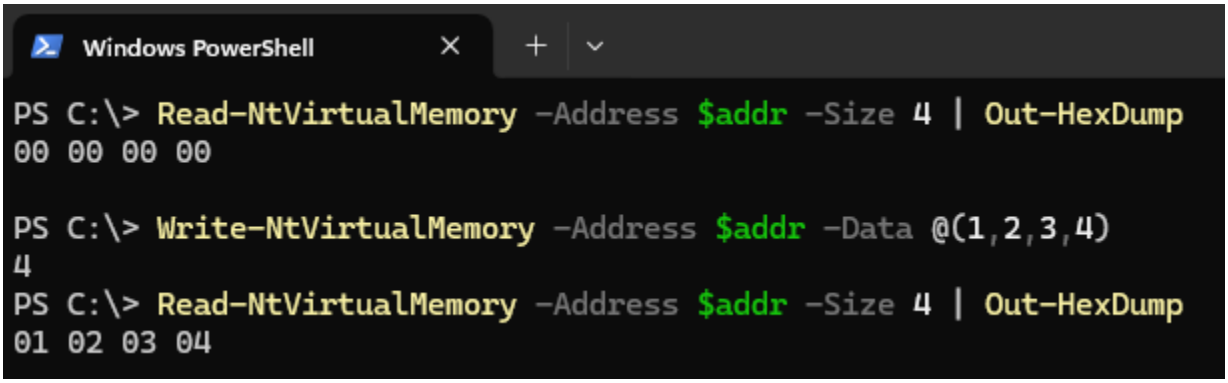
It's possible to allocate new memory and free memory in a process using NtAllocateVirtualMemory and NtFreeVirtualMemory, which both require the VmOperation access right

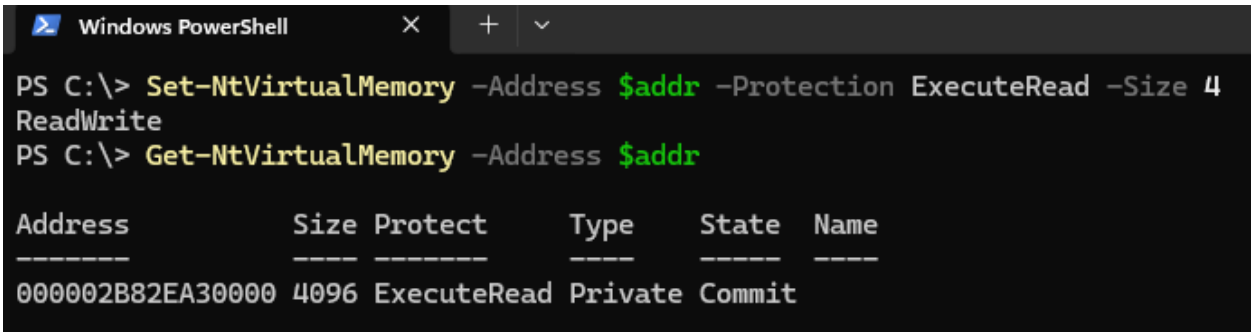
you can change the protection on memory using NtProtectVirtualMemory, which also requires VmOperation access.

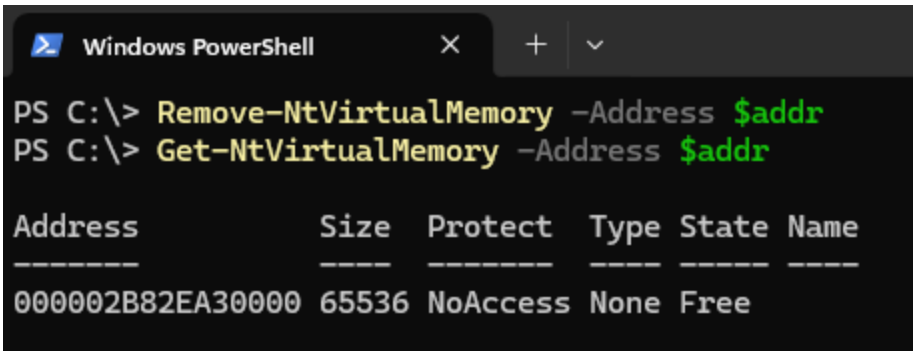
NtVirtualMemory Commands :-


Windows PowerShell
PS C:\> Get-NtVirtualMemory
Address Size Protect Type State Name
000000007FFE0000 4096 ReadOnly Private Commit
000000007FFE7000 4096 ReadOnly Private Commit
000000DAE9D00000 356352 None Private Reserve
000000DAE9D57000 20480 ReadWrite, Guard Private Commit
000000DAE9D5C000 147456 ReadWrite Private Commit
000000DAE9D80000 491520 None Private Reserve
000000DAE9DF8000 20480 ReadWrite, Guard Private Commit


Windows PowerShell
PS C:\> $addr = Add-NtVirtualMemory -Size 1000 -Protection ReadWrite
PS C:\> Get-NtVirtualMemory -Address $addr
Address Size Protect Type State Name
000002B82EA30000 4096 ReadWrite Private Commit


Windows PowerShell
PS C:\> Read-NtVirtualMemory -Address $addr -Size 4 | Out-HexDump
00 00 00 00
PS C:\> Write-NtVirtualMemory -Address $addr -Data @(1,2,3,4)
4
PS C:\> Read-NtVirtualMemory -Address $addr -Size 4 | Out-HexDump
01 02 03 04


Windows PowerShell
PS C:\> Set-NtVirtualMemory -Address $addr -Protection ExecuteRead -Size 4
ReadWrite
PS C:\> Get-NtVirtualMemory -Address $addr
Address Size Protect Type State Name
000002B82EA30000 4096 ExecuteRead Private Commit



First we use Get-NtVirtualMemory to list all the memory regions being used by the current process . It includes the address of the memory region, its size, its protection, and its state.

There are three possible state values :-

Commit : Indicates that the virtual memory region is allocated and available for use.

Reserve : Indicates that the virtual memory region has been allocated but there is currently no backing memory. Using a reserved memory region will cause a crash.

Free : Indicates that the virtual memory region is unused. Using a free memory region will cause a crash.

The Reserve state allows you to reserve virtual memory regions for later use so that nothing else can allocate memory within that range of memory addresses

You can later convert the Reserve state to Commit by re-calling NtAllocateVirtualMemory.

The Free state indicates regions freely available for allocation

Next, we allocate a 1,000-byte read/write region and capture the address in a variable

Passing the address to Get-NtVirtualMemory allows us to query only that specific virtual memory region

You might notice that although we requested a 1,000-byte region, the size of the region returned is 4,096 bytes. This is because all virtual memory allocations on Windows have a minimum allocation size; on the system I'm using, the minimum is 4,096 bytes

It's therefore not possible to allocate a smaller region ,

For this reason, these system calls are not particularly useful for general program allocations; rather, they're primitives on which "heap" memory managers are built, such as malloc from the C library.

Next we read and write to the memory region we just allocated. First we use Read-NtVirtualMemory to read out 4 bytes of the memory region and find that the bytes are all zeros

Next, we write the bytes 1, 2, 3, and 4 to the memory region using Write-NtVirtualMemory

We read the bytes to confirmthat the write operation succeeded

With the memory allocated, we can change the protection using Set - NtVirtualMemory. In this case, we make the allocated memory executable by specifying the protection as ExecuteRead

Querying the current state of the memory region using the Get-NtVirtualMemory command shows that the protection has changed from ReadWrite to ExecuteRead

Also notice that although we requested to change the protection of only 4 bytes, the entire 4,096- byte region is now executable. This is again due to the minimum memory allocation size

Finally, we free the memory using Remove-NtVirtualMemory and verify that the memory is now in the Free state

Section Objects :-

Another way of allocating virtual memory is through Section objects. A Section object is a kernel type that implements memory-mapped files

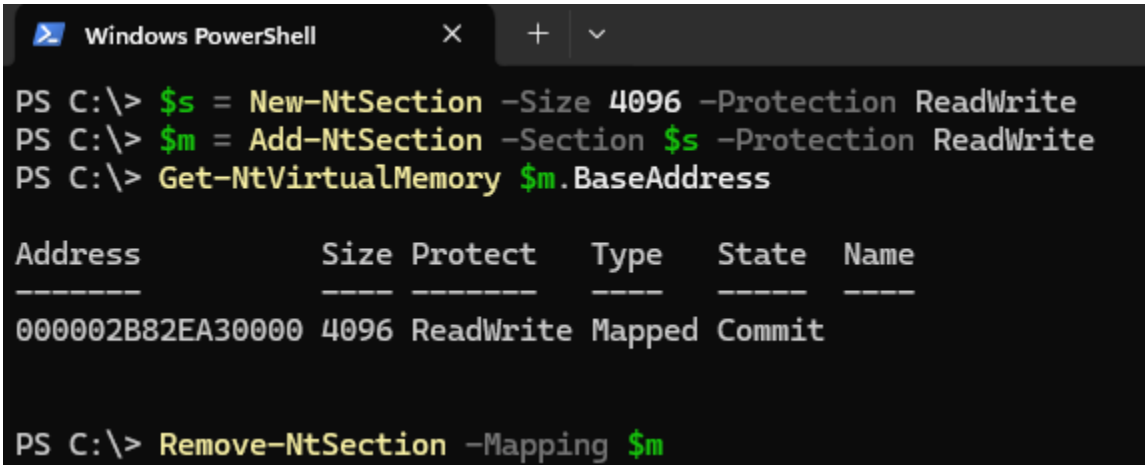
We can use Section objects for two related purposes:

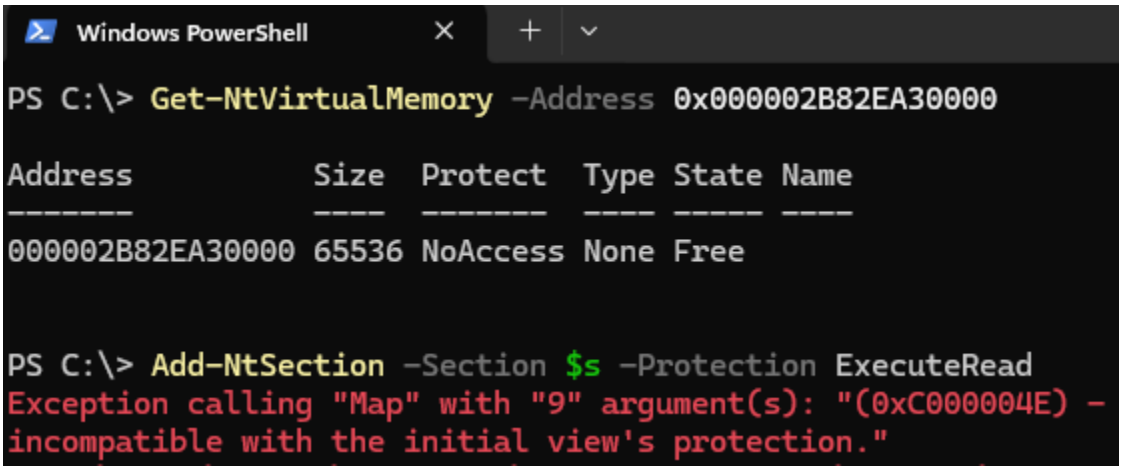
Reading or writing a file as if it were all read into memory

Sharing memory between processes so that the modification in one process is reflected in the other

We can create a Section object via the NtCreateSection system call or the New-NtSection PowerShell command. We must specify the size of the mapping, the protection for the memory, and an optional file handle; in return, we get a handle to the section.

However, creating a section doesn't automatically allow us to access the memory; we first need to map it into the virtual memory address space using NtMapViewOfSection or Add-NtSection





we create a Section object with a size of 4,096 bytes and protection of ReadWrite. We don't specify a File parameter, which means it's anonymous and not backed by any file.

If we gave the Section object an OMNS path, the anonymous memory it represents could be shared with other processes

We then map the section into memory using Add-NtSection, specifying the protection we want for the memory, and query the mapped address to verify that the operation succeeded

Note that the Type is set to Mapped. When we're done with the mapping, we call Remove-NtSection to unmap the section and then verify that it's now free

Finally, we demonstrate that we can't map a section with different protection than that granted when we created the Section object . When we try to map the section with read and execute permissions, which aren't compatible, we see an exception

The protection you're allowed to use to map a Section object into memory depends on two things :

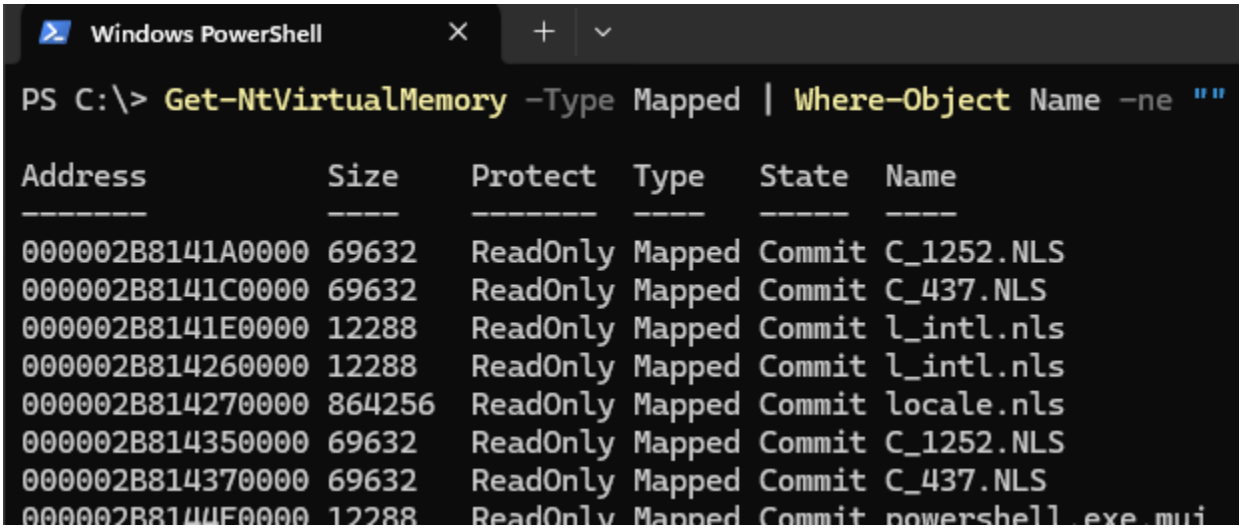
The first is the protection specified when the Section object was created.

The second dependency is the access granted to the section handle you're mapping.

It's possible to map a section into another process by specifying a process handle to Add-NtSection. We don't need to specify the process to Remove- NtSection, as the mapping object knows what process it was mapped in

In the memory information output, the Name column would be populated by the name of the backing file, if it exists.

The section we created was anonymous, so we don't see anything in the Name column, but we can perform a query to find mapped sections that are backed by files using the command shown :

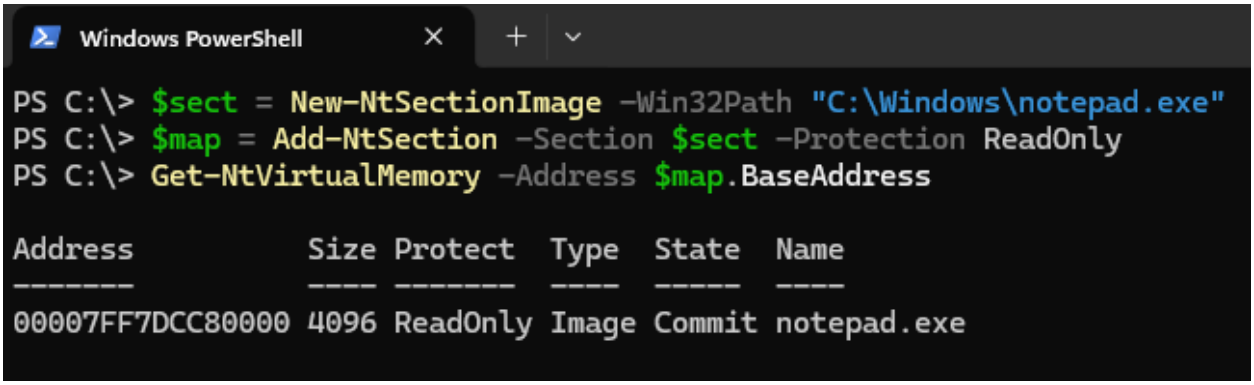


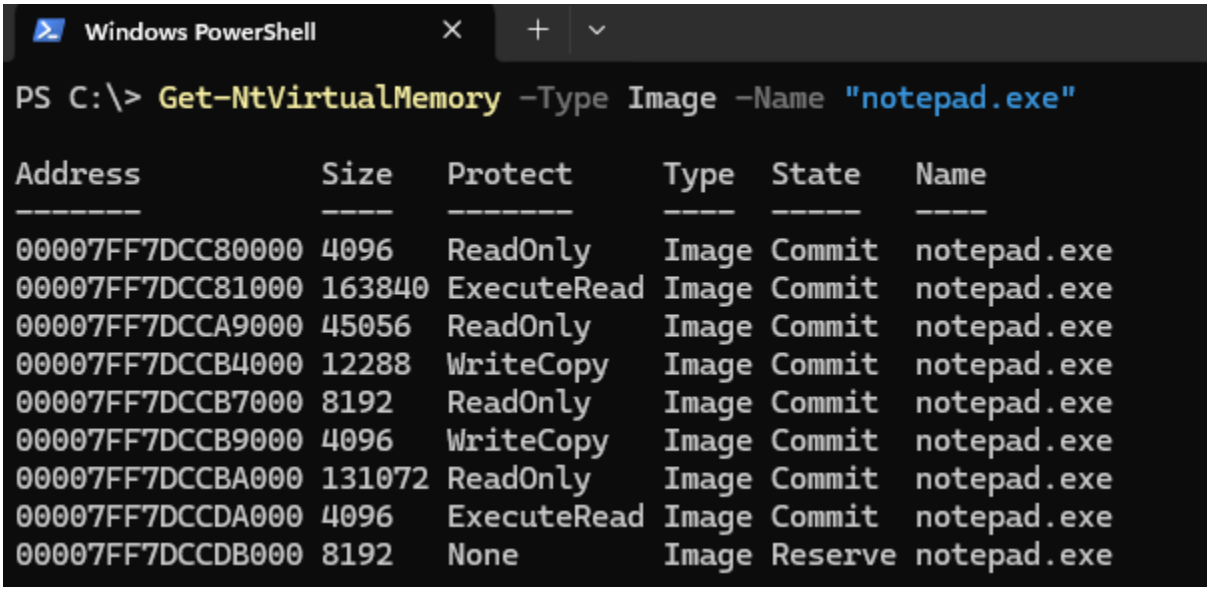
In addition to the Anonymous and Mapped types, there is a third section type, the Image type

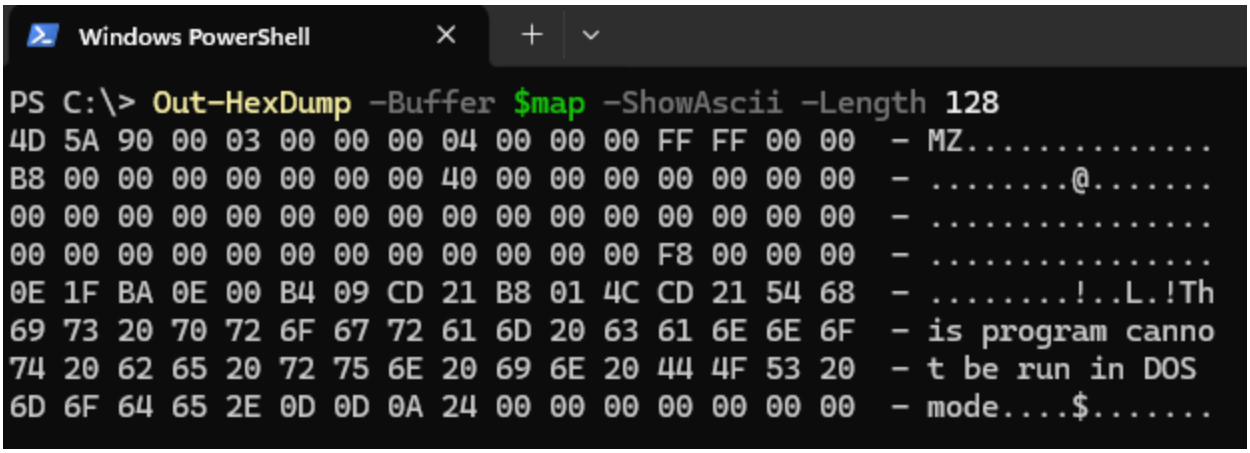
When provided with a file handle to a Windows executable, the kernel will automatically parse the format and generate multiple subsections that represent the various components of the executable

To create a mapped image from a file, we need only Execute access on the file handle; the file doesn't need to be readable for us.

Windows uses image sections extensively to simplify the mapping of executables into memory. We can specify an image section by passing the Image flag when creating the Section object or by using the New-NtSectionImage command :







As you can see, we don't need to specify ExecuteRead or ExecuteReadWrite protection when mapping the image section

Any protection, including ReadOnly, will work ,

When we get the memory information for a map-based address, we see that there is no executable memory there and that the allocation is only 4,096 bytes , which seems far too small for notepad.exe

This is because the section is made up of multiple smaller mapped regions. If we filter out the memory information for the mapped name ❸, we can see the executable memory. Using the Out-HexDump command, we can print the contents of the mapped file buffer

==========

Code Integrity :-

One important security task is ensuring that the code running on your computer is the same code the manufacturer intended you to run

If a malicious user has modified operating system files, you might encounter security issues such as the leaking of private data.

This code integrity subsystem verifies and restricts what files can execute in the kernel, and optionally in user mode, by checking the code's integrity.

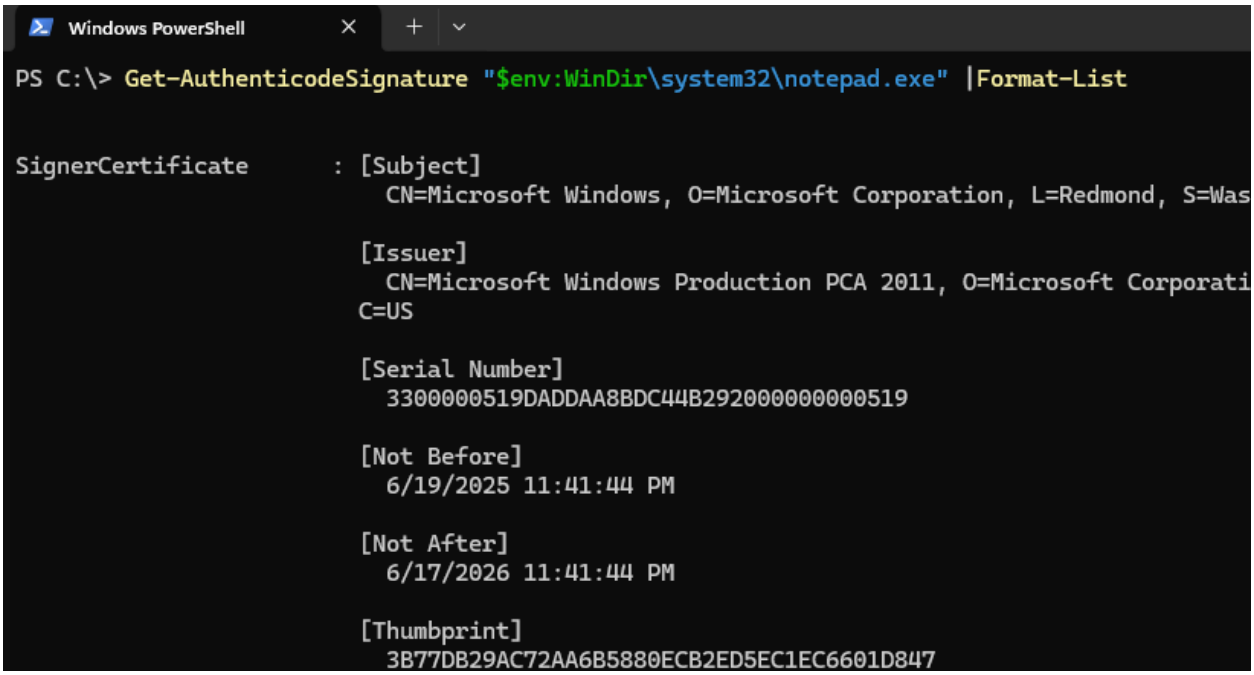
The memory manager can consult with the code integrity subsystem when it loads an image file if it needs to check whether the executable is correctly signed.

Almost every executable on a default Windows installation is signed using a mechanism called Authenticode.

This mechanism allows a cryptographic signature to be embedded in the executable file or collected inside a catalog file. The code integrity subsystem can read this signature, verify that it's valid, and make trust decisions based on

We can use the Get-AuthenticodeSignature command to query the signing

status of an executable, :



Here, we query the signing status of the notepad.exe executable file, formatting the command's output as a list.

The output starts with information about the signer's X.509 certificate. Here, I've shown only the subject name, which clearly indicates that this file is signed by Microsoft.

Next is the status of the signature; in this case, the status indicates that the file is valid and that the signature has been verified. It's possible to have a signed file whose signature is invalid;

The SignatureType property shows that this signature was based on a catalog file rather than being embedded in the file. We can also see that this file is an operating system binary, as determined by information embedded in the signature

Each driver file must have a signature that derives its trust from a Microsoft-issued key. If the signature is invalid or doesn't derive from a Microsoft-issued key, then the kernel can block loading of the driver to preserve system integrity.

==========

Advanced Local Procedure Call :-

The advanced local procedure call (ALPC) subsystem implements local, cross- process communication

To use ALPC, you must first create a server ALPC port using the NtCreateAlpcPort system call and specify a name for it inside the OMNS.

A client can then use this name by calling the NtConnectAlpcPort system call to connect to the server port

At a basic level, the ALPC port allows the secure transmission of discrete messages between a server and a client. ALPC provides the underlying transport for local remote procedure call APIs implemented in Windows.

==========

The Configuration Manager :-

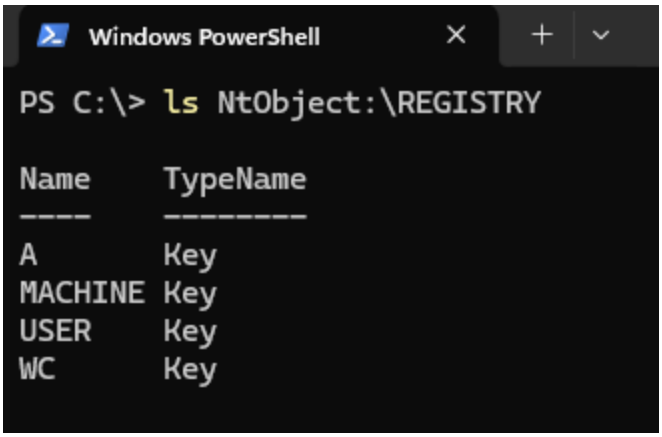
The configuration manager, known more commonly as the registry, is an important component for configuring the operating system

It stores a variety of configuration information, ranging from the system-critical list of available I/O manager device drivers to the (less critical) last position on the screen of your text editor's window

You can think of the registry as a filesystem in which keys are like folders and values are like files.

You can access it through the OMNS, although you must use registry-specific system calls

The root of the registry is the OMNS path REGISTRY. You can list the registry in PowerShell using the NtObject drive :-



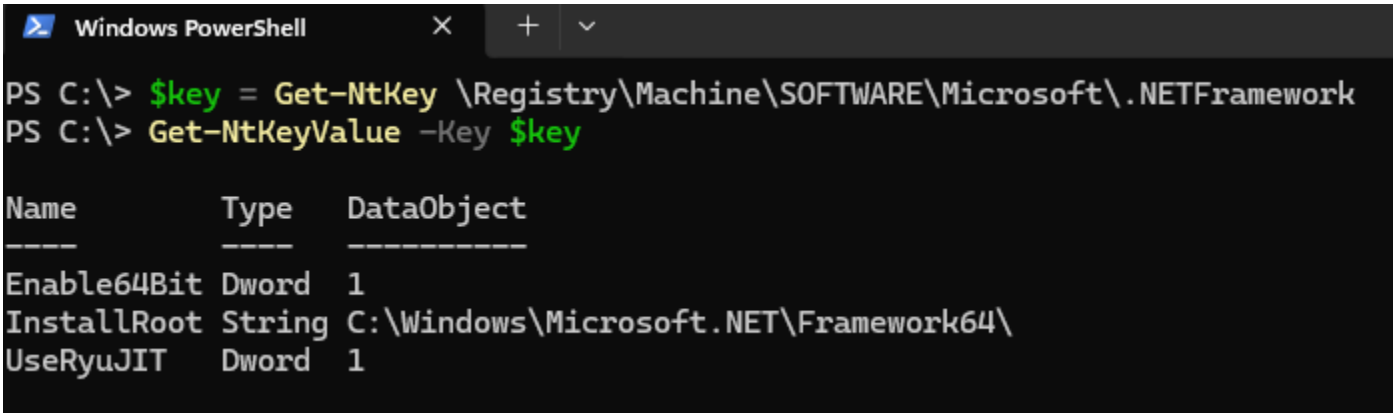
You can replace NtObject:\REGISTRY in Listing 2-35 with NtKey:\ to make accessing the registry simple

The kernel pre-creates the four keys shown here when it initializes. Each of the keys is a special attachment point at which you can attach a registry hive

A hive is a hierarchy of Key objects underneath a single root key. An administrator can load new hives from a file and attach them to these preexisting keys

Note that PowerShell already comes with a drive provider that you can use to access the registry. However, this drive provider exposes only the Win32 view of the registry, which hides the internal details about the registry from view.

You can interact with the registry directly, using the Get-NtKey and New-NtKey commands to open and create Key objects, respectively. You can also use Get-NtKeyValue and Set-NtKeyValue to get and set key values. To remove keys or values, use Remove-NtKey or Remove-NtKeyValue.



We open a Key object using the Get-NtKey command. We can then query the values stored in the Key object using the Get-NtKeyValue command. Each entry in the output shows the name of the value, the type of data stored, and a string representation of the data

==========

==========