VOLUME 1

Windows Security Internals with PowerShell

James Forshaw

NO STARCH PRESS EARLY ACCESS PROGRAM: FEEDBACK WELCOME!

The Early Access program lets you read significant portions of an upcoming book while it's still in the editing and production phases, so you may come across errors or other issues you want to comment on. But while we sincerely appreciate your feedback during a book's EA phase, please use your best discretion when deciding what to report.

At the EA stage, we're most interested in feedback related to content—general comments to the writer, technical errors, versioning concerns, or other high-level issues and observations. As these titles are still in draft form, we already know there may be typos, grammatical mistakes, missing images or captions, layout issues, and instances of placeholder text. No need to report these—they will all be corrected later, during the copyediting, proofreading, and typesetting processes.

If you encounter any errors ("errata") you'd like to report, please fill out **this Google form** so we can review your comments.

WINDOWS SECURITY INTERNALS WITH POWERSHELL

James Forshaw

Early Access edition, 08/02/23

ISBN 13: 978-1-7185-0198-0 (print)
ISBN 13: 978-1-7185-0199-7 (ebook)

Publisher: William Pollock
Managing Editor: Jill Franklin
Production Manager: Sabrina Plomitallo-González
Developmental Editor: Frances Saux

CONTENTS

The chapters in **red** are included in this Early Access PDF.

1

SETTING UP A POWERSHELL TESTING ENVIRONMENT

In this chapter, you'll configure PowerShell so you can work through the code examples presented in the rest of the book. Then, we'll walk through a very quick overview of the PowerShell language, including its types, variables, and expressions. We'll also cover how to execute its commands, how to get help, and how to export data for later use.

Choosing a PowerShell Version

The most important tool you'll need to use this book effectively is PowerShell, which has been installed in the Windows operating system by default since Windows 7. However, there are many different versions of this tool. The version installed by default on currently supported versions of Windows is 5.1, which is suitable for our purposes, even though Microsoft no longer fully supports it. The most recent version, PowerShell 7, is now open source.

All of the code presented in this book will run in both PowerShell 5.1 and the latest open source version, so it doesn't matter which you choose. If you want to use the open source version of PowerShell, visit the project's linked GitHub page at *https://github.com/PowerShell/PowerShell* to find installation instructions for your version of Windows.

Configuring PowerShell

The first thing we need to do in PowerShell is set the *script execution policy*, which determines what types of scripts PowerShell can execute. For Windows clients running PowerShell 5.1, the default is `Restricted`, which blocks all scripts from running unless they are signed with a trusted certificate. As the scripts in this book are unsigned, we'll change the execution policy to `RemoteSigned`. This execution policy allows us to run unsigned PowerShell scripts if they're created locally but will not allow us to execute unsigned scripts downloaded in a web browser or attached to emails. Run the following command to set the execution policy:

```
PS> Set-ExecutionPolicy -Scope CurrentUser -ExecutionPolicy RemoteSigned -
Force
```

The command changes the execution policy for the current user only, not the entire system. If you want to change it for all users, you'll need to start PowerShell as an administrator, then re-run the command, removing the `Scope` parameter.

If you're using the open source version of PowerShell or version 5.1 on Windows Server, then the default script execution policy is `RemoteSigned` and you do not need to change anything.

Now that we can run unsigned scripts, we can install the PowerShell module we'll be using for this book. A PowerShell *module* is a package of scripts and .NET binaries that export PowerShell commands. Every installation of PowerShell comes pre-installed with several modules for tasks ranging from configuring your applications to setting up Windows Update. You can install a module manually by copying files, but the easiest approach is to use the PowerShell Gallery (*https://www.powershellgallery.com*), an online repository of modules.

To install a module from the PowerShell Gallery, we use PowerShell's `Install-Module` command. For this book, we need to install the `NtObjectManager` module using the following command:

```
PS> Install-Module NtObjectManager -Scope CurrentUser -Force
```

Make sure to say yes if the installer asks you any questions (after you've read and understood the question, of course). If you have the module installed already, you can check that you have the latest version by using the `Update-Module` command:

```
PS> Update-Module NtObjectManager
```

Once it's installed, you can load the module using the `Import-Module` command:

```
PS> Import-Module NtObjectManager
```

If you see any errors after importing the module, double-check that you've correctly set the execution policy; that's the most common reason for the module not loading correctly. As a final test, let's run a command that comes with the module to ensure it's working. Execute the command in Listing 1-1 and verify that the output matches what you see in the PowerShell console. We'll describe the purpose of this command in a later chapter.

```
PS> New-NtSecurityDescriptor
```

```
Owner DACL ACE Count SACL ACE Count Integrity Level
----- -------------- -------------- ---------------
NONE  NONE           NONE           NONE
```

Listing 1-1 Testing that the `NtObjectManager` module is working

If everything is working, you can move on to the rest of the book. However, if you need a quick refresher on the PowerShell language, keep reading.

An Overview of the PowerShell Language

This book can't teach you how to use PowerShell from scratch. However, this section touches on various language features that will ensure that you can use this book most effectively.

Understanding Types, Variables, and Expressions

PowerShell supports many different types, from basic integers and strings to complex objects. Table 1-1 shows common built-in types, along with the underlying .NET runtime type and some simple examples.

Table 1-1 Common Basic PowerShell Types with .NET Types and Examples

Type	.NET type	Examples
int	`System.Int32`	`142`, `0x8E`, `0216`
long	`System.Int64`	`142L`, `0x8EL`, `0216L`
string	`System.String`	`"Hello"`, `'World!'`
double	`System.Double`	`1.0`, `1e10`
bool	`System.Boolean`	`$true`, `$false`
array	`System.Object[]`	`@(1, "ABC", $true)`
hashtable	`System.Collections.Hashtable`	`@{A=1; B="ABC"}`

To perform calculations on basic types, we can use well-known operators such as `+`, `-`, `*`, and `/`. These operators can be overloaded; for example, `+` is used for addition as well as for concatenating strings and arrays. Table 1-2 shows a list of common operators, with simple examples and their results. You can test the examples yourself to check the output of the operator.

Table 1-2 Common Operators with Examples

Operator	Name	Examples	Result
+	Addition/concatenation	1 + 2, "Hello" + "World!"	3, "HelloWorld!"
-	Subtraction	2 – 1	1
*	Multiplication	2 * 4	8
/	Division	8 / 4	2
%	Modulus	6 % 4	2
[]	Index	@(3, 2, 1, 0)[1]	2
-f	String formatter	"0x{0:X} {1}" -f 42, 123	"0x2A 123"
-band	Bitwise AND	0x1FF -band 0xFF	255
-bor	Bitwise OR	0x100 -bor 0x20	288
-bxor	Bitwise XOR	0xCC -bxor 0xDD	17
-bnot	Bitwise NOT	-bnot 0xEE	-239
-and	Boolean AND	$true -and $false	$false
-or	Boolean OR	$true -or $false	$true
-not	Boolean NOT	-not $true	$false
-eq	Equals	"Hello" -eq "Hello"	$true
-ne	Not equals	"Hello" -ne "Hello"	$false
-lt	Less than	4 -lt 10	$true
-gt	Greater than	4 -gt 10	$false

You can assign values to variables using the assignment operator, =. A variable has an alphanumeric name prefixed with the $ character. For example, you can capture an array and use the indexing operator to lookup a value, as in Listing 1-2.

```
PS> $var = 3, 2, 1, 0
PS> $var[1]
2
```

Listing 1-2 Capturing an array in a variable and indexing it via the variable name

You can enumerate all variables using the `Get-Variable` command. There are also some pre-defined variables we'll use in the rest of this book. These variables are:

`$null`
Represents the NULL value, which indicates the absence of a value in comparisons

`$pwd`
Contains the current working directory

`$pid`
Contains the process ID of the shell

`$env`

Accesses the process environment (for example, `$env:WinDir` to get the *Windows* directory)

In Table 1-1, you might have noticed that there were two string examples, one with a double quote and one with a single quote. Is there any difference between the two? The double quoted string can perform *string interpolation*, in which you specify a variable name to insert into the string. Listing 1-3 shows examples of string interpolation.

```
PS> $var = 42
PS> "The magic number is $var"
The magic number is 42
PS> 'It is not $var'
It is not $var
```

Listing 1-3 Examples of string interpolation

We define a variable with the value `42` to insert into a string and then create a double quoted string with the variable name inside of it. We can see that the result is the string with the variable name replaced by its value formatted as string. If you want more control over the formatting, you should use the format operator defined in Table 1-2.

To demonstrate the different behavior of a single quoted string, we define one with the variable name inline. We can observe that the variable name is copied verbatim and is not replaced by the value.

Another difference is that the double quoted string can contain character escapes that are ignored in single quoted strings. These escapes use a similar syntax to those of the C programming language, but instead of a backslash character `\`, PowerShell uses the backtick `` ` ``. This is because Windows uses the backslash as a path separator, and writing out filepaths would be very annoying if you had to escape every backslash. Table 1-3 gives a list of character escapes you can use in PowerShell.

Table 1-3 String Character Escapes

Character escape	Name
`0	NUL character, with a value of zero
`a	Bell
`b	Backspace
`n	Line feed
`r	Carriage return
`t	Horizontal tab
`v	Vertical tab
``	Backtick character

You might notice that Table 1-3 has a NUL character. As PowerShell uses the .NET string type, it can contain embedded NUL characters. Unlike the C language, adding the NUL will not terminate the string prematurely.

As all values are .NET types, you can invoke methods and access properties on an object. For example, the following calls the `ToCharArray` method on a string to convert it to an array of single characters:

```
PS > "Hello".ToCharArray()
H
e
l
l
o
```

We can use PowerShell to construct almost any .NET type. The simplest way to construct a type is to cast a value to that type by specifying the .NET type in brackets. When casting, PowerShell will try to find a suitable constructor for the type to invoke. For example, the following command will convert a string to a `System.Guid` object; PowerShell will find a constructor that accepts a string and call it:

```
PS> [System.Guid]"6c0a3a17-4459-4339-a3b6-1cdb1b3e8973"
```

You can also call a constructor explicitly by calling the `new` method on the type. The previous example can be rewritten as follows:

```
PS> [System.Guid]::new("6c0a3a17-4459-4339-a3b6-1cdb1b3e8973")
```

This syntax can also be used to invoke static methods on the type. For example, the following calls the `NewGuid` static method to create a new random GUID:

```
PS> [System.Guid]::NewGuid()
```

You can also create new objects using the `New-Object` command:

```
PS> New-Object -TypeName Guid -ArgumentList "6c0a3a17-4459-4339-a3b6-
1cdb1b3e8973"
```

This example is equivalent to the call to the static `new` function:

Executing Commands

Almost all commands in PowerShell are named using a common pattern: a verb and a noun separated by a dash. For example, consider the command `Get-Item`: the `Get` verb implies retrieving an existing resource, while `Item` is the type of resource to return.

Each command can accept a list of parameters that controls the behavior of the command. For example, the `Get-Item` command accepts a `Path` parameter, which indicates the existing resource to retrieve, as shown below:

```
PS> Get-Item -Path "C:\Windows"
```

The `Path` parameter is also a positional parameter. This means that you can omit the name of the parameter, and PowerShell will do its best to select the best match. For example, the previous command can also be written as the following:

```
PS> Get-Item "C:\Windows"
```

If a parameter takes a string value, and the string does not contain any special characters or whitespace, then you do not need to use quotes around the string. For example, the `Get-Item` command would also work with the following:

```
PS> Get-Item C:\Windows
```

The output of a single command is zero or more values, and could be basic or complex object types. You can pass the output of one command to another as input using a *pipeline*, which is represented by a vertical bar character, `|`. We'll see examples of using a pipeline when we discuss filtering, grouping, and sorting.

You can capture the result of an entire command or pipeline into a variable, then interact with the results. For example, the following captures the result of the `Get-Item` command and queries for the `FullName` property:

```
PS> $var = Get-Item -Path "C:\Windows"
PS> $var.FullName
C:\Windows
```

If you don't want to capture the result in a variable, you can enclose the command in parentheses and directly access its properties and methods, as shown below:

```
PS> (Get-Item -Path "C:\Windows").FullName
C:\Windows
```

The length of a command line is effectively infinite. However, you'll want to try to split up long lines to make the command more readable. The shell will automatically split a link on the pipe character. If you need to split a long line with no pipes, you can use the backtick character, then start a new line. The backtick must be the last character on the line; otherwise, an error will occur when the script is parsed.

Discovering Commands and Getting Help

A default installation of PowerShell has hundreds of commands to choose from. This means that finding a command to perform a specific task can be difficult, and even if you find a command, it might not be clear how to use it. To help, you can use two built-in commands, `Get-Command` and `Get-Help`.

The `Get-Command` command can be used to enumerate all the commands available to you. In its simplest form, you can execute it without any parameters and it will print all commands from all modules. However, it's probably more useful to filter on a specific word you're interested in. For example, Listing 1-4 will list only the commands with the word `SecurityDescriptor` in their names.

```
PS> Get-Command -Name *SecurityDescriptor*
CommandType     Name                                     Version    Source
-----------     ----                                     -------    ------
Function        Add-NtSecurityDescriptorControl          1.1.28
NtObjectManager
```

```
Function        Add-NtSecurityDescriptorDaclAce      1.1.28
NtObjectManager
Function        Clear-NtSecurityDescriptorDacl       1.1.28
NtObjectManager
Function        Clear-NtSecurityDescriptorSacl       1.1.28
NtObjectManager
--snip--
```

Listing 1-4 Using `Get-Command` to enumerate commands with the word `SecurityDescriptor` in them

This command uses *wildcard syntax* to list only commands whose names include the specified word. Wildcard syntax uses a `*` character to represent any character or series of characters. Here, we've put the `*` on either side of `SecurityDescriptor` to indicate that any text can come before or after it.

You can also list the commands available in a module. For example, Listing 1-5 will list only the commands that are both exported by the `NtObjectManager` module and begin with the verb `Start`.

```
PS> Get-Command -Module NtObjectManager -Name Start-*
CommandType     Name                                Version    Source
-----------     ----                                -------    ------
Function        Start-AccessibleScheduledTask       1.1.28     NtObjectManager
Function        Start-NtFileOplock                  1.1.28     NtObjectManager
Function        Start-Win32ChildProcess             1.1.28     NtObjectManager
Cmdlet          Start-NtDebugWait                   1.1.28     NtObjectManager
Cmdlet          Start-NtWait                        1.1.28     NtObjectManager
```

Listing 1-5 Using `Get-Command` to enumerate `Start` commands in the `NtObjectManager` module

Once you've found a command that looks promising, you can use the `Get-Help` command to inspect its parameters and get some usage examples. Let's take the `Start-NtWait` command from Listing 1-5 and pass it to `Get-Help` in Listing 1-6.

```
PS> Get-Help Start-NtWait
NAME
  1 Start-NtWait
SYNOPSIS
  2 Wait on one or more NT objects to become signaled.
SYNTAX
  3 Start-NtWait [-Object] <NtObject[]> [-Alertable <SwitchParameter>]
    [-Hour <int>] [-MilliSecond <long>]
    [-Minute <int>] [-Second <int>] [-WaitAll <SwitchParameter>]
```

```
    [<CommonParameters>]

    Start-NtWait [-Object] <NtObject[]> [-Alertable <SwitchParameter>]
    [-Infinite <SwitchParameter>] [-WaitAll <SwitchParameter>]
    [<CommonParameters>]
4 DESCRIPTION
    This cmdlet allows you to issue a wait on one or more NT
    objects until they become signaled.
--snip--
```

Listing 1-6 Displaying help for the `Start-NtWait` command

By default, `Get-Help` outputs the name of the command 1, a short synopsis 2, the syntax of the command 3, and a more in-depth description 4. In the command syntax section, you can see its multiple possible modes of operation, in this case either specifying a time broken up in hours, minutes, seconds, and milliseconds or specifying `Infinite` to wait indefinitely.

When any part of the syntax is shown in square brackets, `[]`, that means it's optional. For example, the only required parameter is `Object`, which takes an array of `NtObject` values. Even the name of this parameter is optional, as `-Object` is in brackets. You can get some more detail about a parameter by using the `Parameter` command. Listing 1-7 shows the details for the `Object` parameter.

```
PS> Get-Help Start-NtWait -Parameter Object
-Object <NtObject[]>
    Specify a list of objects to wait on.

    Required?                    true
    Position?                    0
    Default value
    Accept pipeline input?       true (ByValue)
    Accept wildcard characters?  False
```

Listing 1-7 Querying the `Object` parameter details

Specify a wildcard for the parameter name to select a group of similar parameter names. For example, if you specify `Obj*`, then you'll get information about any parameters that start with the `Obj` prefix. If you want usage examples, then use the `Examples` parameter, as shown in Listing 1-8.

```
PS> Get-Help Start-NtWait -Examples
--snip--
```

```
    ----------  EXAMPLE 1  ----------
1 $ev = Get-NtEvent \BaseNamedObjects\ABC
  Start-NtWait $ev -Second 10

2 Get an event and wait for 10 seconds for it to be signaled.
--snip--
```

Listing 1-8 Showing examples for `Start-NtWait`

Each example should show a one- or two-line snippet of PowerShell script 1 and a description of the example 2. You can also see the full help output for the command by specifying the `Full` parameter. To view this output in a GUI, use the `ShowWindow` parameter. For example, try running this command:

```
PS> Get-Help Start-NtWait -ShowWindow
```

You should see the dialog shown in Figure 1-1.

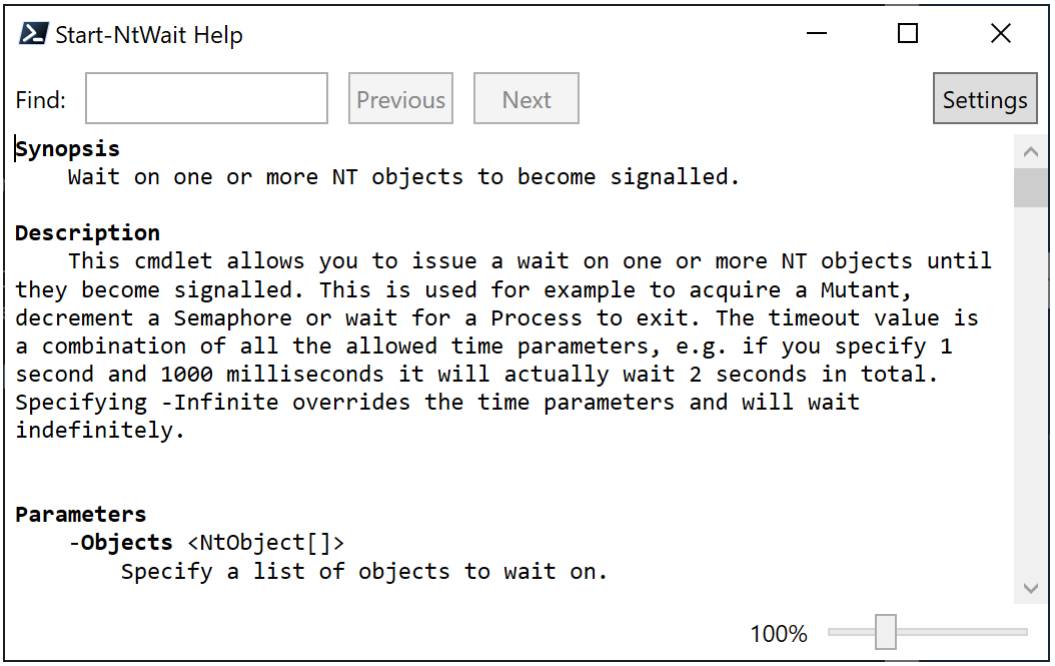

Figure 1-1 A dialog to view help information produced by using the `ShowWindow` parameter for `Get-Help`

One final topic to mention about commands is that you can setup *aliases*, alternative names for the commands. For example,

you can use an alias to make commands shorter to type. PowerShell comes with many aliases pre-defined, and you can display these using the `Get-Alias` command. You can also define your own using the `New-Alias` command. For example, we can set the `Start-NtWait` command to have the alias `swt` by doing the following:

```
PS> New-Alias -Name swt -Value Start-NtWait
```

We'll avoid using aliases unnecessarily through this book, as it can make the scripts more confusing if you don't know what the alias represents.

Defining Functions

As with all programming languages, it pays to reduce complexity in PowerShell. One way of reducing complexity is to combine common code into a function. Once a function is defined, the PowerShell script can call the function rather than needing to repeat the same code in multiple places. The basic function syntax in PowerShell is simple; Listing 1-9 shows an example.

```
PS> function Get-NameValue {
>>  param(
>>    [string]$Name = "",
>>    $Value
>>  )
>>  return "We've got $Name with value $Value"
>> }

PS> Get-NameValue -Name "Hello" -Value "World"
We've got Hello with value World

PS> Get-NameValue "Goodbye" 12345
We've got Goodbye with value 12345
```

Listing 1-9 Defining a simple PowerShell function called `Get-NameValue`

The syntax for defining a function starts with the keyword `function` followed by the name of the function you want to define. While it's not required to use the standard PowerShell naming convention, it pays to do so, as it makes it clear to the user what your function does.

The function then defines some named parameters. This definition follows the normal variable syntax: using a name prefixed with `$`, as you can see in Listing 1-9. You can specify a type in brackets; in this case, `$Name` is a string type. However, you don't need to specify a type; here, the `$Value` parameter can take any value from the caller. You also don't need to specify named parameters. If no `param` block is specified, then any passed arguments are placed in the `$args` array. The first parameter is located at `$args[0]`, the second at `$args[1]`, and so on.

The body of this function takes the parameters and builds a string using string interpolation. The function returns the string using the `return` keyword, which also immediately finishes the function. You can omit the `return` keyword in this case, as PowerShell will return any values uncaptured in variables.

After defining the function, we invoke it. You can specify the parameter names explicitly. However, if the call is unambiguous, then specifying the parameter names is not required. In Listing 1-9, we show both approaches.

If you want to run a small block of code without defining a function, you can create a script block. A *script block* is one or more statements enclosed in braces, `{}`. This block can be assigned to a variable and executed when needed using the `Invoke-Command` command or the `&` operator (Listing 1-10).

```
PS> $script = { Write-Output "Hello" }
PS> & $script
Hello
```

Listing 1-10 Creating a script block and executing it

Here, we create a script block, assign it to a variable, and execute it.

Displaying and Manipulating Objects

If you execute a command and do not capture the results in a variable, the results are passed to the PowerShell console. The console will use a formatter to display the result, either in a table

or in list format. The format is chosen automatically depending on the type of objects in the result. It's also possible to specify custom formatters. For example, if you use the built-in `Get-Process` command, PowerShell uses a custom formatter to display the entries as a table, as shown in Listing 1-11.

```
PS> Get-Process
Handles  NPM(K)    PM(K)      WS(K)     CPU(s)      Id  SI ProcessName
-------  ------    -----      -----     ------      --  -- -----------
    476      27    25896      32044       2.97    3352   1 ApplicationFrameHost
    623      18    25096      18524     529.95   19424   0 audiodg
    170       8     6680       5296       0.08    5192   1 bash
    557      31    23888        332       0.59   10784   1 Calculator
--snip--
```

Listing 1-11 Outputting the process list as a table

If you want to reduce the number of columns in the output, you can use the `Select-Object` command to select only the properties you need. For example, Listing 1-12 selects the `Id` and `ProcessName` properties.

```
PS> Get-Process | Select-Object Id, ProcessName
   Id ProcessName
   -- -----------
3352  ApplicationFrameHost
19424 audiodg
 5192 bash
10784 Calculator
--snip--
```

Listing 1-12 Selecting only `Id` and `ProcessName` properties

You can change the default behavior of the output by using the `Format-Table` or `Format-List` commands, which will force table or list formatting, respectively. For example, Listing 1-13 shows how to use the `Format-List` command to change the output to a list.

```
PS> Get-Process | Format-List
Id      : 3352
Handles : 476
CPU     : 2.96875
SI      : 1
Name    : ApplicationFrameHost
--snip--
```

Listing 1-13 Using `Format-List` to show processes in a list view

To find the names of the available properties, you can use the `Get-Member` command on one of the process objects. For example, Listing 1-14 lists the properties for the process object.

```
PS> Get-Process | Get-Member -Type Property
   TypeName: System.Diagnostics.Process
Name                MemberType Definition
----                ---------- ----------
BasePriority        Property   int BasePriority {get;}
Container           Property   System.ComponentModel.IContainer Container
{get;}
EnableRaisingEvents Property   bool EnableRaisingEvents {get;set;}
ExitCode            Property   int ExitCode {get;}
ExitTime            Property   datetime ExitTime {get;}
--snip--
```

Listing 1-14 Listing all properties of the `Process` object

You might notice that there are other properties not included in the output. To display them, you need to override the custom formatting. The simplest way to do that is to use `Select-Object` to select the properties or explicitly specify them on `Format-Table` or `Format-List`. You can even use `*` to show all properties, as in Listing 1-15.

```
PS> Get-Process | Format-List *
Name                       : ApplicationFrameHost
Id                         : 3352
PriorityClass              : Normal
FileVersion                : 10.0.18362.1 (WinBuild.160101.0800)
HandleCount                : 476
WorkingSet                 : 32968704
PagedMemorySize            : 26517504
--snip--
```

Listing 1-15 Showing all the properties of the `Process` object in a list

Many objects also have methods you can call to modify the object or perform some action. Listing 1-16 shows how you can use `Get-Member` to query for methods.

```
PS> Get-Process | Get-Member -Type Method
   TypeName: System.Diagnostics.Process

Name                      MemberType Definition
----                      ---------- ----------
BeginErrorReadLine        Method     void BeginErrorReadLine()
BeginOutputReadLine       Method     void BeginOutputReadLine()
CancelErrorRead           Method     void CancelErrorRead()
```

```
CancelOutputRead          Method     void CancelOutputRead()
Close                     Method     void Close()
--snip--
```

Listing 1-16 Displaying the methods on a `Process` object

If the output from a command is too long, you can page the output so that it will wait for you to press a key to continue. You add paging by piping the output to the `Out-Host` command, specifying the `Paging` parameter, or by using the `more` command. Listing 1-17 shows an example.

```
PS> Get-Process | Out-Host -Paging
Handles  NPM(K)    PM(K)      WS(K)     CPU(s)      Id  SI ProcessName
-------  ------    -----      -----     ------      --  -- -----------
    476      27    25896      32044       2.97    3352   1 ApplicationFrameHost
    623      18    25096      18524     529.95   19424   0 audiodg
    170       8     6680       5296       0.08    5192   1 bash
    557      31    23888        332       0.59   10784   1 Calculator
<SPACE> next page; <CR> next line; Q quit
```

Listing 1-17 Paging output using `Out-Host`

You can directly write to the console window using the `Write-Host` command in your own scripts. This is appealing, as you can change the colors of the output to suit your tastes. This also has the advantage of not inserting objects into the pipeline by default. However, that also means that, by default, you can't redirect the output to a file or into a pipeline, as shown below:

```
PS> $output = Write-Host "Hello"
Hello
```

However, you can redirect the host output by redirecting its stream to the standard output stream using the following command:

```
PS> $output = Write-Host "Hello" 6>&1
PS> $output
Hello
```

PowerShell also supports a basic GUI to display tables of objects. To access it, use the `Out-GridView` command. Note that the custom formatting will still restrict what columns PowerShell displays. If you want to view other columns, use `Select-Object` in the pipeline to select out the properties. The following example displays all properties in the Grid View:

```
PS> Get-Process | Select-Object * | Out-GridView
```

Running this command should show a dialog like Figure 1-2.

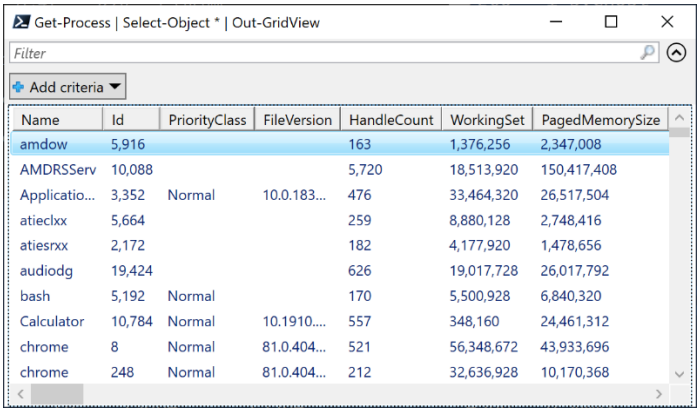

Figure 1-2 Showing process objects in a grid view

You can filter and manipulate the data in the Grid View GUI. Try playing around with the controls. You can also specify the `PassThru` parameter to `Out-GridView`. This parameter causes the command to wait for you to click the OK button on the GUI. Any rows in the view that were selected when you clicked OK will be written to the command pipeline.

Filtering, Ordering, and Grouping Objects

A traditional shell passes raw text between commands; PowerShell passes objects. Passing objects lets you access individual properties of the objects and trivially filter the pipeline. You can even order and group the objects easily.

You can filter objects using the `Where-Object` command, which has the aliases `Where` and `?`. The simplest filter is to check for the value of a parameter, as shown in Listing 1-18, where we filter the output from the built-in `Get-Process` command to find the `explorer` process.

```
PS> Get-Process | Where-Object ProcessName -EQ "explorer"
Handles  NPM(K)    PM(K)      WS(K)     CPU(s)      Id  SI ProcessName
-------  ------    -----      -----     ------      --  -- -----------
   2792     130   118152     158144     624.83    6584   1 explorer
```

Listing 1-18 Filtering a list of processes using `Where-Object`

In Listing 1-18, we pass through only `Process` objects where the `ProcessName` equals (`-EQ`) `"explorer"`. There are numerous operators you can use for filtering, some of which are shown in the Table 1-4.

Table 1-4 Some Common Operators for `Where-Object`

Operator	Example	Description
`-EQ`	`ProcessName -EQ "explorer"`	Equal to the value
`-ME`	`ProcessName -NE "explorer"`	Not equal to the value
`-Match`	`ProcessName -Match "ex.*"`	Matches string against a regular expression
`-NotMatch`	`ProcessName -NotMatch "ex.*"`	Inverse of the `-Match` operator
`-Like`	`ProcessName -Like "ex*"`	Matches string against a wildcard
`-NotLike`	`ProcessName -NotLike "ex*"`	Inverse of the `-Like` operator
`-GT`	`ProcessName -GT "ex"`	Greater than comparison
`-LT`	`ProcessName -LT "ex”`	Less than comparison

You can investigate all of the supported operators by using `Get-Help` on the `Where-Object` command. If the condition to filter on is more complex than a simple comparison, you can use a script block. The script block should return `True` to keep the object in the pipeline or `False` to filter it. For example, Listing 1-18 could also be written as the following:

```
PS> Get-Process | Where-Object { $_.ProcessName -eq "explorer" }
```

The `$_` variable passed to the script block represents the current object in the pipeline. By using a script block, you can use the entire language in your filtering, including calling functions.

To order objects, use the `Sort-Object` command. If the objects can be ordered, as in the case of strings or numbers, then you just need to pipe the objects into the command. Otherwise, you'll need to specify a property to sort on. For example, you can sort the process list by its handle count, represented by the `Handles` property, as shown in Listing 1-19.

```
PS> Get-Process | Sort-Object Handles
Handles  NPM(K)    PM(K)      WS(K)     CPU(s)      Id  SI ProcessName
-------  ------    -----      -----     ------      --  -- -----------
      0       0       60          8                 0   0 Idle
     32       9     4436       6396               1032   1 fontdrvhost
     53       3     1148       1080                496   0 smss
     59       5      804       1764                908   0 LsaIso
```

```
--snip--
```

Listing 1-19 Sorting processes by the number of handles

To sort in descending order instead of ascending order, use the `Descending` parameter, as shown in Listing 1-20.

```
PS> Get-Process | Sort-Object Handles -Descending
Handles  NPM(K)    PM(K)      WS(K)     CPU(s)      Id  SI ProcessName
-------  ------    -----      -----     ------      --  -- -----------
   5143       0      244      15916                4   0 System
   2837     130   116844     156356     634.72   6584   1 explorer
   1461      21    11484      16384               1116   0 svchost
   1397      52    55448       2180      12.80  12452   1 Microsoft.Photos
```

Listing 1-20 Sorting processes by the number of handles in descending order

It's also possible to filter out duplicate entries at this stage by specifying the `Unique` parameter to `Sort-Object`.

Finally, we can group objects based on a property name using the `Group-Object` command. Listing 1-21 shows that this command returns a list of objects, each with `Count`, `Name` and `Group` properties.

```
PS> Get-Process | Group-Object ProcessName
Count Name                      Group
----- ----                      -----
    1 ApplicationFrameHost      {System.Diagnostics.Process
(ApplicationFrameHost)}
    1 Calculator                {System.Diagnostics.Process (Calculator)}
   11 conhost                   {System.Diagnostics.Process (conhost)...}
--snip--
```

Listing 1-21 Grouping `Process` objects by `ProcessName`

Alternatively, you could use all of these commands together in one pipeline, as shown in Listing 1-22.

```
PS> Get-Process | Group-Object ProcessName |
>>     Where-Object Count -GT 10 | Sort-Object Count
Count Name                      Group
----- ----                      -----
   11 conhost                   {System.Diagnostics.Process (conhost),...}
   83 svchost                   {System.Diagnostics.Process (svchost),...}
```

Listing 1-22 Combining `Where-Object`, `Group-Object` and `Sort-Object`

This listing combines the `Where-Object`, `Group-Object`, and `Sort-Object` commands.

Exporting Data

Once you've got the perfect set of objects you want to inspect, you might want to persist that information to a file on disk. PowerShell provides numerous options for this, and I'll discuss only a few of them. The first option is to output the objects as text to a file using `Out-File`. This command captures the formatted text output and writes it to a file. You can use the `Get-Content` to read the file back in again, as shown in Listing 1-23.

```
PS> Get-Process | Out-File processes.txt
PS> Get-Content processes.txt
Handles  NPM(K)    PM(K)      WS(K)     CPU(s)      Id  SI ProcessName
-------  ------    -----      -----     ------      --  -- -----------
    476      27    25896      32044       2.97    3352   1 ApplicationFrameHost
    623      18    25096      18524     529.95   19424   0 audiodg
    170       8     6680       5296       0.08    5192   1 bash
    557      31    23888        332       0.59   10784   1 Calculator
--snip--
```

Listing 1-23 Writing content to a text file and reading it back in again

You can also use the greater-than operator to send the output to a file, as in other shells. This is shown below:

```
PS> Get-Process > processes.txt
```

If you want a more structured format, you can use `Export-Csv` to convert the object to a comma-separated value (CSV) table format. You could then import this file into a spreadsheet program to analyze offline. The example in Listing 1-24 selects some properties of the `Process` object and exports them to the CSV file *processes.csv*.

```
PS> Get-Process | Select-Object Id, ProcessName |
>>     Export-Csv processes.csv -NoTypeInformation
PS> Get-Content processes.csv
"Id","ProcessName"
"3352","ApplicationFrameHost"
"19424","audiodg"
"5192","bash"
"10784","Calculator"
--snip--
```

Listing 1-24 Exporting objects to a CSV

It's possible to reimport the CSV using the `Import-Csv` command. However, if you expect to export the data and then reimport it later, you'll probably prefer the CLI XML format. This format includes the object structure and types of the original object, which allows you to reconstruct it when you import the data. Listing 1-25 shows how you can use the `Export-CliXml` and `Import-CliXml` commands to export objects in this format and then reimport them.

```
PS> Get-Process | Select-Object Id, ProcessName | Export-Clixml processes.xml
PS> Get-Content processes.xml
<Objs Version="1.1.0.1"
xmlns="http://schemas.microsoft.com/powershell/2004/04">
  <Obj RefId="0">
    <TNRef RefId="0" />
    <MS>
      <I32 N="Id">3352</I32>
      <S N="ProcessName">ApplicationFrameHost</S>
    </MS>
  </Obj>
--snip--
</Objs>
PS> $ps = Import-Clixml processes.xml
PS> $ps[0]
  Id ProcessName
  -- -----------
3352 ApplicationFrameHost
```

Listing 1-25 Exporting and importing CLI XML files

This concludes our discussion about using the PowerShell language. If you're a little rusty, I recommend picking up a good book on the topic, such as *PowerShell for Sysadmins* by Adam Bertram.

Wrapping Up

This chapter provided a short overview of setting up your PowerShell environment so that you can run the code examples included throughout this book. We discussed configuring PowerShell to run scripts and installing necessary external PowerShell modules.

We then provided a bit of background on the PowerShell language. This included the basics of PowerShell syntax, as well as discovering commands using `Get-Command`, getting help using `Get-Help`, and displaying, filtering, grouping, and exporting PowerShell objects.

With the basics of PowerShell out of the way, we can start to dive into the inner workings of the Windows operating system. In the next chapter, we'll discuss the Windows kernel and how you can interact with it using PowerShell.

2

THE WINDOWS KERNEL

Windows is a secure, multi-user operating system. However, it's also one of the most challenging modern operating systems to understand in detail. Before we can delve into the intricacies of its security, we'll provide you with an overview of the operating system's structure. We'll also take this opportunity to understand how to use the PowerShell modules that will form the core of this book.

We'll consider the two parts of the running operating system: the kernel and the user-mode applications. The kernel makes the security decisions that determine what a user can do on the system. However, most of the applications you use on a Windows

machine run in user mode. This chapter will focus on the kernel; the next chapter will focus on user-mode applications.

In the following sections, we'll define the various subsystems that make up the Windows kernel. For each subsystem, we'll explain its purpose and how it's used. We'll begin with the object manager, where we'll also detail system calls, which allow a user-mode application to access kernel objects. We'll then discuss the input/output (I/O) manager, how applications are created through the process and thread manager, and how memory is represented with the memory manager. Throughout, we'll outline how you can inspect the behavior of these subsystems using PowerShell.

The Windows Kernel Executive

The *Windows NTOS kernel executive*, or *kernel* for short, is the heart of Windows. It provides all the operating system's privileged functionality, as well as interfaces through which the user applications can communicate with the hardware. The kernel is split into multiple subsystems, each with a dedicated purpose. Figure 2-1 shows a diagram of the components in which we'll be most the interested in this book.

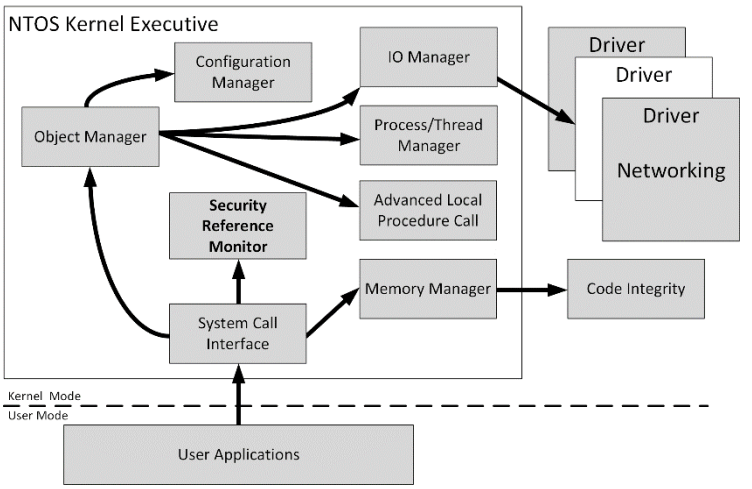

Figure 2-1 The Windows kernel executive modules

Each subsystem in the kernel executive exposes APIs for other subsystems to call. If you are looking at kernel code, you can quickly determine what subsystem each API belongs to using its two-character prefix. The prefixes for the subsystems in Figure 2-1 are shown in Table 2-1.

Table 2-1 API Prefixes to Subsystem

Prefix	Subsystem	Example
`Nt` or `Zw`	System call interface	`NtOpenFile`/`ZwOpenFile`
`Se`	Security reference monitor	`SeAccessCheck`
`Ob`	Object manager	`ObReferenceObjectByHandle`
`Ps`	Process and thread manager	`PsGetCurrentProcess`
`Cm`	Configuration manager	`CmRegisterCallback`
`Mm`	Memory manager	`MmMapIoSpace`
`Io`	Input/output manager	`IoCreateFile`
`Ci`	Code integrity	`CiValidateFileObject`

We'll detail these subsystems in the sections that follow.

The Security Reference Monitor

For the purposes of this book, the *Security Reference Monitor (SRM)* is the most important subsystem in the kernel. It implements the security mechanisms that restrict which users can access what resources. Without the SRM, you wouldn't be able to prevent other users from accessing your files. Figure 2-2 shows the SRM and its related system components.

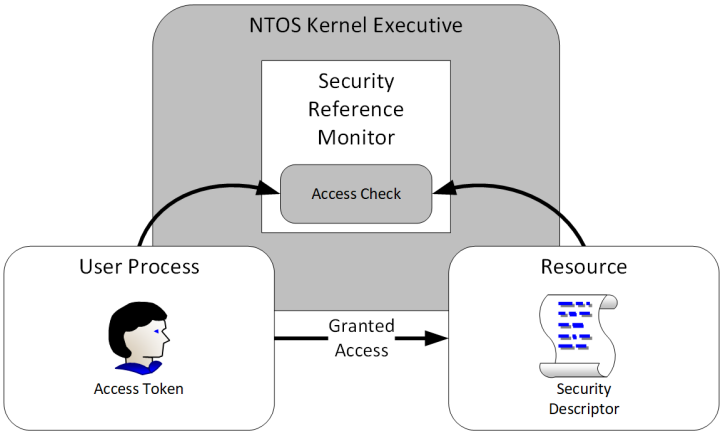

Figure 2-2 Components of the security reference monitor

The SRM defines the identity of a user by assigning an *access token* to every process running on the system. Using an access token, the SRM can then perform an operation called an *access check*. This operation queries a resource's security descriptor, compares it to the current access token, and either calculates the level of granted access or indicates that access is denied to the caller. In essence, the SRM determines the level of access a process has to a resource.

The SRM is also responsible for auditing *events*, which an administrator can configure to generate whenever a user accesses a resource. This auditing information 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). This name needs to be converted to a SID before the SRM can use it. 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 an SID as a string. SDDL can represent the entire security descriptor of a resource; for now, we'll just use it to represent the SID. In Listing 2-1, 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"
Name                 Sid
----                 ---
BUILTIN\Users        S-1-5-32-545
```

Listing 2-1 Querying for the *Users* group using `Get-NtSid`

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 system. The output also contains the SID in SDDL format, which can be broken down as follows:

- The `S` character prefix. This indicates that what follows is an SDDL SID.
- The version of the SID structure in decimal. This has the fixed value `1`.
- The security authority. In this example, it's authority `5`, which indicates the built-in NT authority.
- Two relative identifiers (RID) `32` and `545`, in decimal. These represent the NT authority group.

We can also use `Get-NtSid` to perform the reverse operation: converting an SDDL SID back to a name (Listing 2-2).

```
PS> Get-NtSid -Sddl "S-1-5-32-545"
Name          Sid
----          ---
BUILTIN\Users S-1-5-32-545
```

Listing 2-2 Parsing an SDDL SID

I'll describe the SRM and its functions in much greater depth in Chapters 4 through 9, and we'll revisit the SID structure in Chapter 6, when we discuss security descriptors. For now, remember that SIDs represent users and groups and that we can represent them as strings in SDDL form. Let's move on to another of the core Windows kernel executive subsystems, the object manager.

The Object Manager

On Unix-like operating systems, everything is a file. On Windows, everything is an object, meaning that every file, process, and thread is represented in kernel memory as an object structure. Importantly for security, these objects can have 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. In this section, we'll first discuss the types of objects the object manager supports. Then, we'll show how kernel objects can be opened through a naming convention and called by the kernel using a system call. Finally, we'll detail how to use a handle to access the object once the system call has finished.

Object Types

The kernel maintains a list of all the types of objects it supports. This is necessary, as each object type has different operations and security properties. Listing 2-3 shows how to use the `Get-NtType` command to list all supported types in PowerShell.

```
PS> Get-NtType
Name
----
Type
Directory
SymbolicLink
```

```
Token
Job
Process
Thread
--snip--
```

Listing 2-3 Executing `Get-NtType`

I've truncated the list of types; the machine I'm using supports 67 of them. However, we can already see some interesting type names. The first entry in the generated list is `Type`; even the list of kernel types is built from objects. Other interesting types are `Process` and `Thread`, which, perhaps unsurprisingly, represent the kernel object for a process and thread, respectively. We'll describe other object types in more detail later in this chapter.

Each type entry returns additional useful information, and we'll come back to some of it soon. (If you want to start now, you can display all properties of a type by passing it to the `Format-List` command.) The next question is how to access each of these types. To answer it, we'll need to talk about the object manager namespace.

The Object Manager Namespace

As a user of Windows, you typically see just 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)*, isn't very well documented or exposed to most developers, which makes it even more interesting.

The OMNS is built out of `Directory` object types. The object directories act as if they were in a filesystem: each directory contains other objects, which you can consider to be files. However, they are separate 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 sub-directories and objects. You can specify the full path to an object with a backslash-separated string. We can enumerate the OMNS in using a drive provider that is part of this book's PowerShell module. As shown in Listing 2-4, this exposes the OMNS as if it's a filesystem by listing the `NtObject` drive.

```
PS> ls NtObject:\ | Sort-Object Name
Name                 TypeName
----                 --------
ArcName              Directory
BaseNamedObjects     Directory
BindFltPort          FilterConnectionPort
Callback             Directory
CLDMSGPORT           FilterConnectionPort
clfs                 Device
CsrSbSyncEvent       Event
Device               Directory
Dfs                  SymbolicLink
DosDevices           SymbolicLink
--snip--
```

Listing 2-4 Listing the root OMNS directory

Listing 2-4 shows a short snippet of the root OMNS directory. By default, this output includes the name of each object and its type. We can see a few `Directory` objects; you can list them if you have permission to do so. 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. For example, Listing 2-5 shows the target for a symbolic link in the root of the OMNS:

```
PS> ls NtObject:\Dfs | Select-Object SymbolicLinkTarget
SymbolicLinkTarget
------------------
\Device\DfsClient

PS> Get-Item NtObject:\Device\DfsClient | Format-Table
Name      TypeName
----      --------
DfsClient Device
```

Listing 2-5 Showing the target of a symbolic link

We list the *\Dfs* OMNS path, then extract the `SymbolicLinkTarget` property to get the real target. Next,

we check the target path, *\Device\DfsClient*, to show it's a `Device` type, which is what the symbolic link can be used to access. Windows pre-configures several important object directories, shown in Table 2-2.

Table 2-2 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

The first directory in Table 2-2, *BaseNamedObjects* (BNO) is important in the context of the object manager. It's a directory that allows any user to create named kernel objects. This single directory allows the sharing of resources between different users on the local system. Note that you don't have to create objects in the BNO directory; it's only a convention.

We'll describe the other object directories in more detail later in this chapter. For now, you can list them in PowerShell by prefixing the path with `NtObject:` as I've shown in Listing 2-5.

System Calls

How can we access the named objects in the OMNS from a user-mode application? Well, if we're in a user-mode application, then we need the kernel to access the object, 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. For example, the `NtCreateMutant` system call creates a `Mutant` object, a mutual exclusion primitive used for locking and thread synchronization.

The name of a system call follows a common pattern. It starts with either `Nt` or `Zw`. For user-mode callers, the two prefixes are

equivalent; however, if the system call is invoked by code executing in the kernel, the `Zw` changes the security checking process. We'll come back to the implications of the `Zw` prefix in Chapter 7, when we talk about access modes.

After the prefix comes the operation's verb: `Create`, in the case of `NtCreateMutant`. The rest of the name relates to the kernel object type the system call operates on. 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.

Certain system calls perform type-specific operations. For example, `NtQueryDirectoryFile` is used to query the entries in a `File` object directory. Let's look at the C-language prototype for the `NtCreateMutant` system call to understand what parameters need to be passed to a typical call. As shown in Listing 2-6, the `NtCreateMutant` system call creates a new `Mutant` object.

```
NTSTATUS NtCreateMutant(
    HANDLE* FileHandle,
    ACCESS_MASK DesiredAccess,
    OBJECT_ATTRIBUTES* ObjectAttributes,
    BOOLEAN InitialOwner
);
```

Listing 2-6 The C Prototype for `NtCreateMutant`

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. For example, we could request access that allows us to wait for the `Mutant` to be unlocked. If we didn't request that access, any application that tried to wait on the `Mutant` would immediately fail. The access granted depends on the results of the SRM's access check. We'll discuss handles and `DesiredAccess` in more detail in the next section.

Third is the `ObjectAttributes` parameter, which defines the attributes for the object to open or create. The `OBJECT_ATTRIBUTES` structure is defined as shown in Listing 2-7.

```
struct OBJECT_ATTRIBUTES {
  ULONG           Length;
  HANDLE          RootDirectory;
  UNICODE_STRING* ObjectName;
  ULONG           Attributes;
  PVOID           SecurityDescriptor;
  PVOID           SecurityQualityOfService;
}
```

Listing 2-7 The `OBJECT_ATTRIBUTES` structure

This C language structure starts with `Length`, which represents the length of the structure. Specifying the structure length at the start is a common C style idiom to ensure that the correct structure has been passed to the system call.

Next come `RootDirectory` and `ObjectName`. These are taken together, as they indicate how the system call should look up the resource being accessed. The `RootDirectory` is a handle to an opened kernel object to use as the base for looking up the object. The `ObjectName` field is a pointer to a

`UNICODE_STRING` structure. This is a counted string, defined in Listing 2-8 as a C language structure.

```
struct UNICODE_STRING {
  USHORT Length;
  USHORT MaximumLength;
  WCHAR* Buffer;
};
```

Listing 2-8 The `UNICODE_STRING` structure

The structure references the string data through `Buffer`, which is a pointer to an array of 16-bit Unicode characters. The string is represented in UCS-2 encoding; Windows predates many of the changes to Unicode, such as UTF-16 or UTF-8.

The `UNICODE_STRING` structure also contains two length fields, `Length` and `MaximumLength`. The first length field represents the total valid length of the string in bytes (not in Unicode characters) pointed to by `Buffer`. If you're coming from a C programming background, this length does not include any `NUL` terminating character. In fact, a `NUL` character is permitted in object names.

The second length field represents the maximum length of the string data in bytes pointed to by `Buffer`. Because the structure has two separate lengths, it's possible to allocate an empty string with a large maximum length and a zero valid length, then update the string value using the `Buffer` pointer. Note that the lengths are stored as `USHORT` values, which are unsigned 16-bit integers. Coupled with the length-representing bytes, this means a string can be at most 32,767 characters long.

To specify the name of an object, you could either, for example, set `ObjectName` to an absolute path of *\BaseNamedObjects\ABC*, or set `RootDirectory` to a `Directory` object for *\BaseNamedObjects* and then pass `ABC` as the `ObjectName`. These two actions would open the same object.

Return to Listing 2-7. After the `ObjectName` comes `Attributes`, which is a set of flags to modify the object name

lookup process or change the returned handle's properties. Table 2-3 shows the valid values for the `Attributes` field.

Table 2-3 Object Attribute Flags and Descriptions

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. 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. We will discuss this value in more detail in Chapter 5.
`DontReparse`	Indicates not to follow any path that contains a symbolic link.

The final two fields in the `OBJECT_ATTRIBUTES` structure allow the caller to specify the *Security Quality of Service (SQoS)* and security descriptor for the object. We'll come back to SQoS in Chapter 4 and the security descriptor in Chapter 5.

Next in the `NtCreateMutant` system call in Listing 2-6, the `InitialOwner` Boolean parameter is specific to the type. In this case, it represents whether the created `Mutant` is owned by the caller or not. Many other system calls, especially for files, have more complex parameters, which we'll discuss in more detail later in the book.

NTSTATUS Codes

All system calls return a 32-bit `NTSTATUS` code. This status code is composed of multiple components packed into the 32-bits, as shown in Figure 2-3.

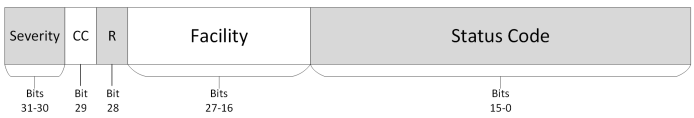

Figure 2-3 The NT status code structure

The most significant two bits (31 and 30) indicate the severity of the status code. Table 2-4 shows the available values.

Table 2-4 NT Status Severity Codes

Severity name	Value
STATUS_SEVERITY_SUCCESS	0
STATUS_SEVERITY_INFORMATIONAL	1
STATUS_SEVERITY_WARNING	2
STATUS_SEVERITY_ERROR	3

If the severity is a warning or an error, then bit 31 of the status code will be set to 1. If the status code is treated as a signed 32-bit integer, this bit represents a negative value. It's common coding practice to assume that, if the status code is negative, then the code represents an error, and if it is positive, it represents a success. As we can see from the table, this assumption isn't completely true, as the negative status code could also be a warning, but it works well enough in practice.

The next component in Figure 2-3, 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). Third parties are not obliged to follow this specification, so don't treat it as fact.

Following the customer code is R, which for an NT status code is a reserved bit that must be set to 0.

After R comes Facility, which indicates the component or subsystem associated with the status code. Microsoft has pre-defined around 50 facilities for its own purposes. Third parties should define their own facility and combine it with the customer code to distinguish themselves from Microsoft. Table 2-5 shows a few commonly encountered facilities:

Table 2-5 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, `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 (Listing 2-9).

```
PS> Get-NtStatus
Status     StatusName                    Message
------     ----------                    -------
00000000   STATUS_SUCCESS                STATUS_SUCCESS
00000001   STATUS_WAIT_1                 STATUS_WAIT_1
00000080   STATUS_ABANDONED_WAIT_0       STATUS_ABANDONED_WAIT_0
000000C0   STATUS_USER_APC               STATUS_USER_APC
000000FF   STATUS_ALREADY_COMPLETE       The requested action was completed
by...
00000100   STATUS_KERNEL_APC             STATUS_KERNEL_APC
00000101   STATUS_ALERTED                STATUS_ALERTED
00000102   STATUS_TIMEOUT                STATUS_TIMEOUT
00000103   STATUS_PENDING                The operation that was requested is
p...
--snip--
```

Listing 2-9 Example output from `Get-NtStatus`

Notice how, in Listing 2-9, some status values, such as `STATUS_PENDING`, have a human readable message. This message isn't embedded in the PowerShell module; instead, it's stored inside a Windows library and can be extracted at runtime.

When we call a system call via a PowerShell command, its status code is surfaced through a .NET exception. For example, if we try an open a `Directory` object that doesn't exist, we'll see the exception shown in Listing 2-10 displayed in the console.

```
PS> Get-NtDirectory \THISDOESNOTEXIST
Get-NtDirectory : (0xC0000034) - Object Name not found.
--snip--

PS> Get-NtStatus 0xC0000034 | Format-List
Status          : 3221225524
StatusSigned    : -1073741772
StatusName      : STATUS_OBJECT_NAME_NOT_FOUND
Message         : Object Name not found.
Win32Error      : ERROR_FILE_NOT_FOUND
Win32ErrorCode : 2
Code            : 52
CustomerCode    : False
Reserved        : False
```

```
Facility        : FACILITY_DEFAULT
Severity        : STATUS_SEVERITY_ERROR
```

Listing 2-10 An `NTSTATUS` exception generated when trying to open a nonexistent directory

In Listing 2-10, we use `Get-NtDirectory` to open the nonexistent path *\THISDOESNOTEXIST*. This generates the `NTSTATUS 0xC0000034` exception, shown here along with the decoded message 1. If you want more information about the status code, you can pass it to `Get-NtStatus` and format it as a list to view all of its properties, including its `Facility` and `Severity`. The NT status code is an unsigned integer value; however, it's common to also see it printed as a signed value incorrectly 2.

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? The application can use the handle returned by a system call, as discussed in the previous section. Each running process has an associated *handle table* containing three pieces of information:

- A handle's numeric identifier.
- The granted access to the handle, for example read or write.
- The pointer to the object structure in kernel memory.

Before the kernel can use a handle, the system call implementation must look up the kernel object pointer from the handle table using a kernel API such as `ObReferenceObjectByHandle`. By providing this handle indirectly, a kernel component can return the handle number to the user-mode application without exposing the kernel object directly. Figure 2-4 shows the handle lookup process.

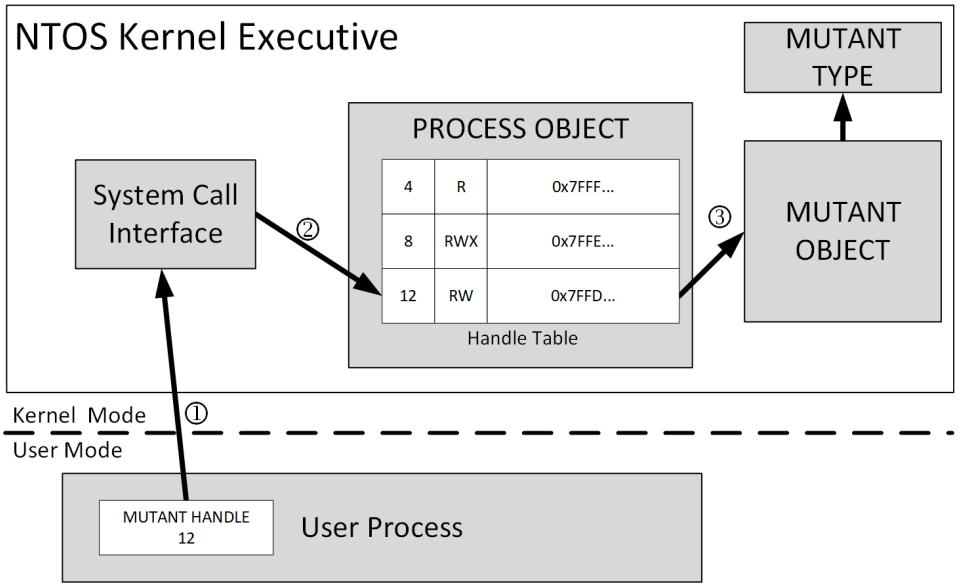

Figure 2-4 The handle table lookup process

In Figure 2-4, the user process is trying to perform some operation on a `Mutant` object. When a user process wants to use a handle, it must first pass the handle's value to the system call we defined in the previous section 1. 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 2.

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 3, the API will return `STATUS_OBJECT_TYPE_MISMATCH`.

These two checks are crucial for security. The access check ensures that the user can't perform an operation on a handle to which they don't have access (for example, writing to a file for which they have only read access). The type check ensures the

user hasn't passed an unrelated kernel object type, which might result in a type confusion in the kernel, causing security issues such as memory corruption. If the conversion succeeds, the system call now has a kernel pointer to the object, which it can use to perform the user's requested operation.

Access Masks

The granted access value in the handle table is 32-bit bitfield called an *access mask*. This is the same bitfield used for the `DesiredAccess` parameter specified in the system call. We'll discuss how the `DesiredAccess` and the access check process determines the granted access in more detail in Chapter 7. An access mask is composed of four bitfields, as shown in Figure 2-5.

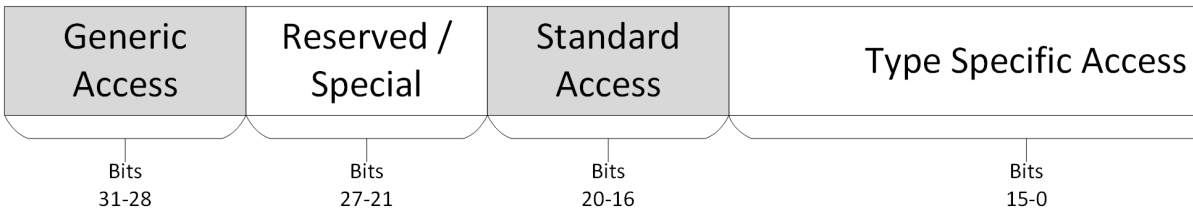

Figure 2-5 The structure of access mask bitfields

The most important component of the access mask is the 16-bit *type specific access*, a set of bits that grant operations defined for particular kernel object types. For example, a `File` object might have a separate bit to specify reading (`ReadData`) and writing to the file (`WriteData`). Alternatively, a synchronization `Event` might have only `Signal` event access.

The *standard access* component of the access mask grants operations that can apply to any object type. These operations include the following:

`Delete`
Removes the object, for example by deleting it from disk or from the registry

`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; for example, waits for a process to exit or a mutant to be unlocked

We'll cover the security-related access in more detail in Chapter 4 and 5.

Next, the access mask contains *reserved* and *special access* bits. Most of these bits are reserved, but they include two access values:

`AccessSystemSecurity`
Reads or writes auditing information on the object

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

We'll cover `AccessSystemSecurity` in Chapter 8 and `MaximumAllowed` in Chapter 7.

The final component of the access mask is *generic access*. These access bits allow an application to request access to a kernel object using the system call's `DesiredAccess` parameter. There are four broad categories of access: `GenericRead`, `GenericWrite`, `GenericExecute`, and `GenericAll`.

When you request one of these generic access rights, the SRM will first convert the access into the corresponding type-specific access. 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 the type. To facilitate the conversion each type contains a *generic mapping table*, which maps `GenericRead`, `GenericWrite`, `GenericExecute` and `GenericAll` to type-specific access. We can display the mapping table using `Get-NtType`, as shown in Listing 2-11.

```
PS> Get-NtType | Select-Object Name, GenericMapping
Name                            GenericMapping
----                            --------------
Type                            R:00020000 W:00020000 E:00020000 A:000F0001
Directory                       R:00020003 W:0002000C E:00020003 A:000F000F
SymbolicLink                    R:00020001 W:00020000 E:00020001 A:000F0001
Token                           R:0002001A W:000201E0 E:00020005 A:000F01FF
--snip--
```

Listing 2-11 Displaying the generic mapping table for object types

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. Listing 2-12 shows an example for the `File` type.

```
PS> Get-NtTypeAccess -Type File
Mask       Value                GenericAccess
----       -----                -------------
00000001   ReadData             Read, All
00000002   WriteData            Write, All
00000004   AppendData           Write, All
00000008   ReadEa               Read, All
00000010   WriteEa              Write, All
00000020   Execute              Execute, All
00000040   DeleteChild          All
00000080   ReadAttributes       Read, Execute, All
00000100   WriteAttributes      Write, All
00010000   Delete               All
00020000   ReadControl          Read, Write, Execute, All
00040000   WriteDac             All
00080000   WriteOwner           All
00100000   Synchronize          Read, Write, Execute, All
```

Listing 2-12 Displaying the access mask for the `File` object type

The output of the `Get-NtTypeAccess` command shows the access mask value, the name of the access as known to the

PowerShell module, and the generic access from which it will be mapped. Note how some access types are granted only to `All`; this means that even if you requested generic read, write, and execute access, you wouldn't be granted access to those rights.

SOFTWARE DEVELOPMENT KIT NAMES

To improve usability, the PowerShell module has modified the original names of the access rights found in the Windows software development kit (SDK). You can view the equivalent SDK names using the `SDKName` property with the `Get-NtTypeAccess` command:

```
PS> Get-NtTypeAccess -Type File | Select SDKName, Value
SDKName                          Value
-------                          -----
FILE_READ_DATA                  ReadData
FILE_WRITE_DATA                WriteData
FILE_APPEND_DATA              AppendData
--snip--
```

These name mappings are useful for porting native code to PowerShell.

You can convert between a numeric access mask and specific object types using the `Get-NtAccessMask` command, as shown in Listing 2-13.

```
PS> Get-NtAccessMask -FileAccess ReadData, ReadAttributes, ReadControl
Access
------
00020081

PS> Get-NtAccessMask -FileAccess GenericRead
Access
------
80000000

PS> Get-NtAccessMask -FileAccess GenericRead -MapGenericRights
Access
------
00120089

PS> Get-NtAccessMask 0x120089 -AsTypeAccess File
ReadData, ReadEa, ReadAttributes, ReadControl, Synchronize
```

Listing 2-13 Converting access masks using `Get-NtAccessMask`

In Listing 2-13, we first request the access mask from a set of `File` access names and receive the numeric access mask in hexadecimal. Next, we get the access mask for the `GenericRead` access; as you can see, the value returned is just

the numeric value of `GenericRead`. Next, we request the access mask for `GenericRead` but specify that we want to map the generic access to a specific access by using the `MapGenericRights` parameter. As we've specified the access for the `File` type, this command uses the `File` type's generic mapping to convert to the specific access mask. We can then convert a raw access mask back to a type access using the `AsTypeAccess` parameter and specify the kernel type to use.

You can query an object handle's granted access mask through the PowerShell object's `GrantedAccess` property. This returns the enumerated type format for the access mask. To retrieve the numeric value, use the `GrantedAccessMask` property, shown in Listing 2-14.

```
PS> $mut = New-NtMutant
PS> $mut.GrantedAccess
QueryState, Delete, ReadControl, WriteDac, WriteOwner, Synchronize

PS> $mut.GrantedAccessMask
Access
------
001F0001
```

Listing 2-14 Displaying the numeric value of the access mask using `GrantedAccessMask`

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 (Listing 2-15).

```
PS> Get-NtHandle -ProcessId $pid
ProcessId  Handle     ObjectType           Object               GrantedAccess
---------  ------     ----------           ------               -------------
22460      4          Process              FFFF800224F02080     001FFFFF
22460      8          Thread               FFFF800224F1A140     001FFFFF
22460      12         SymbolicLink         FFFF9184AC639FC0     000F0001
22460      16         Mutant               FFFF800224F26510     001F0001
--snip--
```

Listing 2-15 Displaying the handle table for the current process using `Get-NtHandle`

Each handle entry in Listing 2-15 contains the type of the object, the address of the kernel object in kernel memory, and the granted access mask.

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` and `New` call, then you can call the `Close` method on the object to close the handle. You can also close an object automatically in PowerShell by using the `Use-NtObject` command to invoke a script block and that closes once finishes.

If you do not close handles manually, they will be closed automatically by the .NET *garbage collector* if the handle object is not referenced (for example, held in a PowerShell variable). You should get into the habit of manually closing handles, as the garbage collector could run at any time, and you might have to wait a long time for the resources to be released. Listing 2-16 provides an example of manually closing objects.

```
PS> Use-NtObject($m = New-NtMutant \BaseNamedObjects\ABC) {
        $m.FullPath
}
\BaseNamedObjects\ABC
PS> $m.IsClosed
True

PS> $m = New-NtMutant \BaseNamedObjects\ABC
PS> $m.IsClosed
False
PS> $m.Close()
PS> $m.IsClosed
True
```

Listing 2-16 Closing an object handle

If the kernel object structure is no longer referenced, either through a handle or by a kernel component, then this object will also be destroyed. Once an object is destroyed, all of its allocated memory is cleaned up, and if it exists, its name in the OMNS is removed. However, `File` and `Key` objects have permanent names; to remove them, you must explicitly delete them.

PERMANENT OBJECTS

It is possible to get the kernel to mark an object as permanent, preventing the object from being destroyed when all handles close and allowing its name to remain in the OMNS. To make an object permanent, you need to either specify the `Permanent` attribute flag when creating the object or use the system call `NtMakePermanentObject`, which is mapped to the `MakePermanent` call on any handle

object returned by the `Get` or `New` commands. You need a special privilege, `SeCreatePermanentPrivilege`, to do this; we'll discuss privileges in Chapter 4.

The reverse operation, `NtMakeTemporaryObject` (or the `MakeTemporary` method in PowerShell), removes the permanent setting and allows an object to be destroyed. The destruction won't happen until all handles to the object have closed. This operation doesn't require any special privilege, but it does require `Delete` access on the object to succeed.

Handle Duplication

We 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. For example, when you pass a file handle to a new process, you could grant only read access right, preventing the new process from writing to the object. However, you should rely on reducing the handle's granted access for security; if the process with the handle has access to the resource, they can just reopen it to get write access.

Listing 2-17 shows some examples of using the `Copy-NtObject` command, which wraps `NtDuplicateObject`, to perform some duplication in the same process. We'll come back to process duplication and security checks in Chapter 6.

```
1 PS> $mut = New-NtMutant "\BaseNamedObjects\ABC"
  PS> $mut.GrantedAccess
  QueryState, Delete, ReadControl, WriteDac, WriteOwner, Synchronize

2 PS> Use-NtObject($dup = Copy-NtObject $mut) {
      $mut
      $dup
      Compare-NtObject $mut $dup
  }
  Handle Name NtTypeName Inherit ProtectFromClose
  ------ ---- ---------- ------- ----------------
  1616   ABC  Mutant     False   False
  2212   ABC  Mutant     False   False
  True
```

```
3 PS> $mask = Get-NtAccessMask -MutantAccess QueryState
  PS> Use-NtObject($dup = Copy-NtObject $mut -DesiredAccessMask $mask) {
      $dup.GrantedAccess
      Compare-NtObject $mut $dup
  }
4 QueryState
  True
```

Listing 2-17 Using `Copy-NtObject` to duplicate handles

We create a new `Mutant` object to test handle duplication. We then extract the current granted access, which shows six access rights 1. For the first duplication, we'll keep the same granted access 2. You can see in the first column of the output that the handles are different. We call `Compare-NtObject` to determine whether the two handles are the same underlying kernel object, which returns `True`. Next, we get an access mask for `Mutant QueryState` access 3 and duplicate the handle requesting that access. We can see in the output the granted access is now only `QueryState` 4. However, the `Compare-NtObject` return still indicates the handles refer to the same object.

Also relevant to handle duplication are the handle attributes `Inherit` and `ProtectFromClose`. Setting `Inherit` allows a new process to inherit the handle when it's created. This allows you to pass handles to a new process to perform tasks such as redirecting console output text to a file. We'll cover handle inheritance later in this chapter.

On the other hand, `ProtectFromClose` does what is says: it protects the handle from being closed. You can set the `ProtectFromClose` attribute by setting the `ProtectFromClose` property. Listing 2-18 shows an example of `ProtectFromClose`.

```
PS> $mut = New-NtMutant
PS> $mut.ProtectFromClose = $true
PS> Close-NtObject -SafeHandle $mut.Handle -CurrentProcess
STATUS_HANDLE_NOT_CLOSABLE
```

Listing 2-18 Testing the `ProtectFromClose` handle attribute

Any attempt to close the handle will fail with a `STATUS_HANDLE_NOT_CLOSEABLE` status code, and the handle will stay open.

Query and Set Information System Calls

A kernel object typically stores information about its state. For example, a `Process` object stores the command line it was created with. To allow us to retrieve or set this information, the kernel could have implemented a specific "get process command line" system call; however, due to the volume of stored information, this approach would quickly become unworkable.

For that reason, the kernel implements generic `Query` and `Set` information system calls whose parameters follow a common pattern for all kernel object types. Listing 2-19 shows the `Query` information system call's pattern.

```
NTSTATUS NtQueryInformationProcess(
   HANDLE                   Handle,
   PROCESS_INFORMATION_CLASS InformationClass,
   PVOID                    Information,
   ULONG                    InformationLength,
   PULONG                   ReturnLength)
```

Listing 2-19 An example `Query` information system call for the `Process` type

Here, we've used the `Process` type as an example, but the system call follows the same pattern for all types; just replace `Process` with the name of the kernel type.

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, which we can extract and implement in PowerShell. Querying certain kinds of information might require special privileges or administrator access.

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

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 a success, and if the system call failed, it indicates how big the buffer needs to be with `STATUS_INFO_LENGTH_MISMATCH` or `STATUS_BUFFER_TOO_SMALL`.

Unfortunately, you can't rely on this automatic sizing behavior. Some information classes and types will return data only if the requested length exactly matches the size of the data to return from the kernel. This makes it difficult to query data without knowing its format in advance. Even the SDK rarely documents the exact sizes required.

The `Set` information call is almost the same as that for `Query`, except the buffer is now an input to the system call rather than an output. Also, since we no longer need a return length parameter, this parameter has been removed, as shown in Listing 2-20.

```
NTSTATUS NtSetInformationProcess(
   HANDLE                  Handle,
   PROCESS_INFORMATION_CLASS InformationClass,
   PVOID                   Information,
   ULONG                   InformationLength)
```

Listing 2-20 An example `Set` information system call for the `Process` type

In the PowerShell module, you can query a type's information class names using the `Get-NtObjectInformationClass` command, shown in Listing 2-21. Bear in mind that some information class names might be missing from the list, as Microsoft doesn't always document them.

```
PS> Get-NtObjectInformationClass Process
Key                                          Value
---                                          -----
ProcessBasicInformation                          0
ProcessQuotaLimits                               1
ProcessIoCounters                                2
ProcessVmCounters                                3
ProcessTimes                                     4
--snip--
```

Listing 2-21 Listing the information classes for the `Process` type

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`. Listing 2-22 shows an example of how to use `Get-NtObjectInformation`.

```
PS> $proc = Get-NtProcess
1 PS> Get-NtObjectInformation $proc ProcessTimes
Get-NtObjectInformation : (0xC0000023) - {Buffer Too Small}
The buffer is too small to contain the entry. No information has been written
to the buffer.
--snip--

2 PS> Get-NtObjectInformation $proc ProcessTimes -Length 32
43
231
39
138
--snip--

3 PS> Get-NtObjectInformation $proc ProcessTimes -AsObject
CreateTime         ExitTime KernelTime UserTime
----------         -------- ---------- --------
132480295787554603 0        35937500   85312500
```

Listing 2-22 Querying a `Process` object for basic information

The `Process` type doesn't set the return length for the `ProcessTimes` information class, so you if you don't specify any length, the operation generates the `STATUS_BUFFER_TOO_SMALL` error 1. 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 2 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 3.

Also, the handle object for many information classes already exposes properties and methods to set or query values. The value will be decoded into a usage format; for example, in Listing 2-15,

the times are in an internal format. The `CreationTime` property on the object will take this internal format and convert it to a human-readable date and time.

You can easily inspect properties by accessing them on the object or using the `Format-List` command. For example, Listing 2-23 lists all the properties on a `Process` object, then queries for the formatted `CreationTime`.

```
PS> $proc | Format-List
SessionId                : 2
ProcessId                : 5484
ParentProcessId          : 8108
PebAddress               : 46725963776
--snip--

PS> $proc.CreationTime
Saturday, October 24, 17:12:58
```

Listing 2-23 Querying a handle object for properties and inspecting the `CreationTime`

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. For some types, such as registry keys, the information class differs between querying and setting, as you can see in Listing 2-24.

```
PS> Get-NtObjectInformationClass Key
Key                          Value
---                          -----
KeyBasicInformation              0
--snip--

PS> Get-NtObjectInformationClass Key -Set
Key                            Value
---                            -----
KeyWriteTimeInformation            0
--snip--
```

Listing 2-24 Inspecting the `QueryInformation` and `SetInformation` classes for the `Key` type

Calling `Get-NtObjectInformationClass` with just the type name returns the `QueryInformation` class. If you specify the type name and the `Set` parameter, you get the `SetInformation` class. Notice the how the two entries shown

have different names and therefore represent different information.

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. For example, when you open a document on your computer, the file is made available through a filesystem driver. 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.

We can manually load a new driver through the `NtLoadDriver` system call or do so automatically using the Plug and Play (PnP) manager. For every driver, the I/O manager creates an entry in the *\Driver* directory. 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 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 (Listing 2-25).

```
PS> ls NtObject:\Device
Name                                           TypeName
----                                           --------
_HID00000034                                   Device
DBUtil_2_3                                     Device
000000c7                                       Device
000000b3                                       Device
UMDFCtrlDev-0f8ff736-55d7-11ea-b5d8-2...       Device
0000006a                                       Device
--snip--
```

Listing 2-25 Displaying the `Device` objects

In the output, we can see that the objects' type names are all `Device`. 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 using dedicated 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, respectively, as shown in Listing 2-26.

```
PS> Use-NtObject($f = Get-NtFile "\SystemRoot\notepad.exe") {
    $f | Select-Object FullPath, NtTypeName
}
FullPath                                    NtTypeName
--------                                    ----------
1 \Device\HarddiskVolume3\Windows\notepad.exe File

PS> Get-Item NtObject:\Device\HarddiskVolume3
Name                                        TypeName
----                                        --------
HarddiskVolume3                             Device
```

Listing 2-26 Opening a device object and displaying its volume path

In this example, we open *notepad.exe* from the *\Windows* folder. 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.

Looking at the result, we can see that the full path starts with *\Device\HarddiskVolume3*, followed by *Windows\notepad.exe* 1. If we try to display the device, we find it's of type `Device`. 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 (Listing 2-27).

```
PS> Get-NtKernelModule
Name                 ImageBase        ImageSize
----                 ---------        ---------
ntoskrnl.exe         FFFFF8053BEAA000 11231232
hal.dll              FFFFF8053BE07000 667648
```

```
kd.dll                  FFFFF8053B42E000 45056
msrpc.sys               FFFFF8053B48E000 393216
ksecdd.sys              FFFFF8053B45E000 172032
--snip--
```

Listing 2-27 Enumerating all loaded kernel drivers

Unlike other operating systems such as Linux, 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.

That's all we'll say for now about the I/O manager. We'll discuss kernel drivers and their security in more detail in Chapter 16. For now, let's turn to another important subsystem.

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. This makes sense: 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)*.

Because processes and threads don't have names and are securable resource, it can be difficult to list the ones running on a system. You could brute-force the ID space, but that would take a while. Fortunately, 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 using the built-in `Get-NtProcess` and `Get-NtThread` commands and passing them the `-InfoOnly` parameter (Listing 2-28). You can also use the built-in `Get-Process` command to produce a similar output. Each of the returned objects has a `Threads` property that you can query for the thread information.

```
PS> Get-NtProcess -InfoOnly
PID  PPID Name           SessionId
---  ---- ----           ---------
0    0    Idle           0
4    0    System         0
128  4    Secure System  0
192  4    Registry       0
812  4    smss.exe       0
920  892  csrss.exe      0
--snip--

PS> Get-NtThread -InfoOnly
TID PID ProcessName StartAddress
--- --- ----------- ------------
0   0   Idle        FFFFF8004C9CAFD0
0   0   Idle        FFFFF8004C9CAFD0
--snip--
```

Listing 2-28 Displaying processes and threads without high privilege

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, PID 4, is important, as 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 processes and threads, 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 we can then interact with. For example, Listing 2-29 shows how to query the command line and executable path of the current process.

```
PS> $proc = Get-NtProcess -ProcessId $pid
```

```
PS> $proc.CommandLine
"C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe"
PS> $proc.Win32ImagePath
C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
```

Listing 2-29 Opening the current process by its process ID

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 pseudo handle is the value `-1` converted to a handle, and for the current thread, it's `-2`. You can access these pseudo handles by passing the `-Current` parameter instead of an ID to the `Get-NtProcess` and `Get-NtThread` commands.

Note that the security of a process and its threads is independent. If you know the ID of a thread, it's possible to access the thread handle inside a process even if you can't access the process itself.

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 which controls the allocation of this address space.

You're unlikely to have 128TB of physical memory in your computer, but the memory manager has ways of making it look like you have more physical memory than you do. For example, it can use a dedicated file on your filesystem, called a *pagefile*, to temporarily store memory when it's not currently needed. As your filesystem's available storage space is much larger than physical memory, this can provide the appearance of a large amount of memory.

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`, which must be set according to the memory's purpose. For example, for code to be executed, the memory must have an `Execute` protection.

You can query all memory status information for a process if you have the `QueryLimitedInformation` access right on the process handle and call `NtQueryVirtualMemory`. However, reading or 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. Finally, you can change the executable protection on memory using `NtProtectVirtualMemory`, which also requires `VmOperation` access.

The NtVirtualMemory Commands

PowerShell wraps these system calls using the `Get-`, `Add-`, `Read-`, `Write-`, `Remove-`, and `Set-NtVirtualMemory` commands. Note that these commands all accept an optional `Process` parameter that lets you access memory in a different process from the current one. Listing 2-30 shows the commands in action.

```
PS> Get-NtVirtualMemory
Address          Size          Protect          Type    State   Name
-------          ----          -------          ----    -----   ----
000000007FFE0000 4096          ReadOnly         Private Commit
000000007FFEF000 4096          ReadOnly         Private Commit
000000E706390000 241664        None             Private Reserve
000000E7063CB000 12288         ReadWrite, Guard Private Commit
000000E7063CE000 8192          ReadWrite        Private Commit
--snip--

PS> $addr = Add-NtVirtualMemory -Size 1000 -Protection ReadWrite
PS> Get-NtVirtualMemory -Address $addr
```

```
Address           Size Protect   Type    State  Name
-------           ---- -------   ----    -----  ----
000002624A440000 4096 ReadWrite Private Commit

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

PS> Set-NtVirtualMemory -Address $addr -Protection ExecuteRead -Size 4
ReadWrite
PS> Get-NtVirtualMemory -Address $addr
Address           Size Protect     Type    State  Name
-------           ---- -------     ----    -----  ----
000002624A440000 4096 ExecuteRead Private Commit

PS> Remove-NtVirtualMemory -Address $addr
PS> Get-NtVirtualMemory -Address $addr
Address           Size   Protect  Type State Name
-------           ----   -------  ---- ----- ----
000002624A440000 196608 NoAccess None Free
```

Listing 2-30 Performing various memory operations on a process

Here, we perform several operations. First, we use `Get-NtVirtualMemory` to list all the memory regions being used by the current process 1. The returned list will be large, but the excerpt shown here should give you a rough idea of how the information is presented. 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 is unused. Using a free memory region will cause a crash.

What's the difference between `Reserve` and `Free`, if using either memory region would 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`. `Free` memory is just that; regions freely available for allocation. We'll cover what the `Type` and `Name` columns indicate later in this section.

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, but 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 us `Read-NtVirtualMemory` to read out four 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 confirm that the write operation succeeded; the two values should match, as shown in the output.

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`. If we query the current state of the memory region using the `Get-NtVirtualMemory` command, we find that the protection has changed from `ReadWrite` to `ExecuteRead`. Also notice that although we requested to change the protection of only four 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. Memory allocated using `NtAllocateVirtualMemory` is considered private, as indicated by the value of the `Type` property shown in Listing 2-30.

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 of 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 the 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`. Listing 2-21 provides an example in which we create an anonymous section and map it into memory.

```
PS> $s = New-NtSection -Size 4096 -Protection ReadWrite
PS> $m = Add-NtSection -Section $s -Protection ReadWrite
PS> Get-NtVirtualMemory $m.BaseAddress ③
Address           Size Protect   Type   State  Name
-------           ---- -------   ----   -----  ----
000001C3DD0E0000 4096 ReadWrite Mapped Commit

PS> Remove-NtSection -Mapping $m ④
PS> Get-NtVirtualMemory -Address 0x1C3DD0E0000
Address           Size Protect   Type   State  Name
-------           ---- -------   ----   -----  ----
000001C3DD0E0000 4096 NoAccess  None   Free
```

```
PS> Add-NtSection -Section $s -Protection ExecuteRead ⑤
Exception calling "Map" with "9" argument(s):
  "(0xC000004E) - A view to a section specifies a protection which is
   incompatible with the initial view's protection."
```

Listing 2-31 Creating a section and mapping it into memory

We first need to create the `Section` object with a size of 4,096 bytes and protection of `ReadWrite`. We didn'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, other processes could be open it, and the handle could be duplicated to another process.

We then map the `Section` into memory using `Add-NtSection`, specifying the protection we want for the memory. We 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 can call `Remove-NtSection` to un-map the section and then verify that it's now free.

Finally, we demonstrate that we can't map a section with different protection than granted when we created the `Section` object. When we try to map the section with read/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. For example, if the section was created with `ReadOnly` protection, you can never map it to be writeable.

The second dependency is the access granted to the `Section` handle you're mapping. If you want to map the section as readable, then the handle must have `MapRead` access. To map it to be writeable, you need both `MapRead` and `MapWrite`. (And, of course, having just `MapWrite` access isn't sufficient to map the section as writeable if the original `Section` was not specified with a writeable protection.)

It's possible to map a section into another process by specifying a process handle to `Add-NtSection`. We do not

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 Listing 2-32.

```
PS> Get-NtVirtualMemory -Type Mapped | Where-Object Name -ne ""
Address           Size    Protect  Type   State  Name
-------           ----    -------  ----   -----  ----
000001760DB90000 815104  ReadOnly Mapped Commit locale.nls
000001760DC60000 12288   ReadOnly Mapped Commit powershell.exe.mui
000001760DEE0000 20480   ReadOnly Mapped Commit winnlsres.dll
000001760F720000 3371008 ReadOnly Mapped Commit SortDefault.nls
--snip--
```

Listing 2-32 Listing mapped files with names

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, you 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 using the `New-NtSectionImage` command, as shown in Listing 2-33.

```
PS> $sect = New-NtSectionImage -Win32Path "c:\windows\notepad.exe"
1 PS> $map = Add-NtSection -Section $sect -Protection ReadOnly
2 PS> Get-NtVirtualMemory -Address $map.BaseAddress
Address           Size Protect  Type  State  Name
-------           ---- -------  ----  -----  ----
00007FF667150000 4096 ReadOnly Image Commit notepad.exe

3 PS> Get-NtVirtualMemory -Type Image -Name "notepad.exe"
Address           Size   Protect     Type  State  Name
-------           ----   -------     ----  -----  ----
00007FF667150000 4096   ReadOnly    Image Commit notepad.exe
```

```
00007FF667151000 135168 ExecuteRead Image Commit notepad.exe
00007FF667172000 36864  ReadOnly    Image Commit notepad.exe
00007FF66717B000 12288  WriteCopy   Image Commit notepad.exe
00007FF66717E000 4096   ReadOnly    Image Commit notepad.exe
00007FF66717F000 4096   WriteCopy   Image Commit notepad.exe
00007FF667180000 8192   ReadOnly    Image Commit notepad.exe

4 PS> Out-HexDump -Buffer $map -ShowAscii -Length 128
   4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00  - MZ..............
   B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  - ........@.......
   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  - ................
   00 00 00 00 00 00 00 00 00 00 00 00 F8 00 00 00  - ................
   0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68  - ........!..L.!Th
   69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F  - is program canno
   74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20  - t be run in DOS
   6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00  - mode....$.......
```

Listing 2-33 Mapping *notepad.exe* and viewing the loaded image

As you can see, we don't need to specify an executable protection when mapping the image section. Any protection, including `ReadOnly`, will work 1. 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 2. This seems far too small for *notepad.exe* but occurs because the section is made up of multiple smaller mapped regions. We can filter out the memory information for the mapped name, as shown at 3. Using the `Out-HexDump` command, we can print the contents of the mapped file buffer 4.

We'll revisit the topic of memory management when we talk about memory protections and mitigations in later chapters.

Code Integrity

One important security task is ensuring that the code running on your computer is the same code that 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.

Microsoft considers the integrity of code running on Windows to be so important there is an entire subsystem to deal with it.

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 image files if it needs to check whether the executable is correctly signed.

Almost every executable on a Windows installation, if provided by Microsoft, 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 component can read this signature, verify that it's valid, and make trust decisions based on it.

We can query the signing status of an executable using the `Get-AuthenticodeSignature` command, as shown in Listing 2-34.

```
PS> Get-AuthenticodeSignature "$env:WinDir\system32\notepad.exe" | Format-List
SignerCertificate      : [Subject]
     CN=Microsoft Windows, O=Microsoft Corporation, L=Redmond, S=Washington,
C=US
--snip--
Status                 : Valid
StatusMessage          : Signature verified.
Path                   : C:\WINDOWS\system32\notepad.exe
SignatureType          : Catalog
IsOSBinary             : True
```

Listing 2-34 Displaying the Authenticode signature for a kernel driver

We query the signing status of the *notepad.exe* executable file, formatting its 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. This can occur when the certificate has been revoked. In that case, the status is likely to show an error, such as `NotSigned`.

The next 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.

The most common trust decision that code integrity makes is to check whether a kernel driver can load. 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 the Microsoft issued key, then the kernel can reject the loading of the driver to preserve system integrity.

Advanced Local Procedure Call

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

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.

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 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, as shown in Listing 2-35.

```
PS> ls NtObject:\REGISTRY
Name    TypeName
----    --------
A       Key
MACHINE Key
USER    Key
WC      Key
```

Listing 2-35 Enumerating the registry root key

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

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 pre-existing 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. We'll cover the Win32 view of the registry separately in Chapter 3.

We can interact with the registry directly using the `Get-NtKey` and `New-NtKey` commands to open or create key objects. You can also use `Get-NtKeyValue` and `Set-NtKeyValue` to get or set key values. To remove keys or values, use `Remove-NtKey` or `Remove-NtKeyValue`. Listing 2-36 show these commands in action.

```
PS> $key = Get-NtKey \Registry\Machine\SOFTWARE\Microsoft\.NETFramework
PS> Get-NtKeyValue -Key $key
Name                    Type   DataObject
----                    ----   ----------
Enable64Bit             Dword  1
InstallRoot             String C:\Windows\Microsoft.NET\Framework64\
UseRyuJIT               Dword  1
DbgManagedDebugger      String "C:\Windows\system32\vsjitdebugger.exe" ...
```

```
DbgJITDebugLaunchSetting Dword  16
```

Listing 2-36 Opening a registry key and querying its values

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.

Worked Examples

Using PowerShell, you can easily to change this book's example scripts to do many different things. To encourage experimentation, each chapter wraps up with worked examples repurposing the various commands you've learned.

In these examples, I'll also highlight times where I've discovered security vulnerabilities using this tooling. This should give you a clear indication of what to look for in Microsoft or third-party applications if you're a security researcher; likewise, for developers, it will help you avoid certain pitfalls.

Finding Open Handles by Name

The objects returned by the `Get-NtHandle` command have additional properties that allow you to query the name and the security descriptor of the object. These properties are not shown by default, as they're expensive to lookup; doing so requires first opening the process containing the handle for `DupHandle` access, duplicating the handle back to the caller PowerShell instance, and finally querying the property.

If performance doesn't matter to you, then you can use the code in Listing 2-37 to find all open files matching a specific filename.

```
PS> $hs = Get-NtHandle -ObjectType File | Where-Object Name -Match Windows
PS> $hs | Select-Object ProcessId, Handle, Name
ProcessId Handle Name
--------- ------ ----
```

```
     3140     64 \Device\HarddiskVolume3\Windows\System32
     3140   1628 \Device\HarddiskVolume3\Windows\System32\en-
US\KernelBase.dll.mui
     3428     72 \Device\HarddiskVolume3\Windows\System3
     3428    304 \Device\HarddiskVolume3\Windows\System32\en-
US\svchost.exe.mui
     3428    840 \Device\HarddiskVolume3\Windows\System32\en-
US\crypt32.dll.mui
     3428   1604 \Device\HarddiskVolume3\Windows\System32\en-
US\winnlsres.dll.mui
--snip--
```

Listing 2-37 Finding `File` object handles, which match a specific name

This script queries for all `File` object handles and filters them to only the ones with the string `Windows` in the `Name` property, which represents the file path. Once the `Name` property has been queried, it's cached so you can then display it to the console with a custom selection.

Note that, because it duplicates the handle from the process, this script can only show handles in processes the caller can open. To get the best results, run it as an administrator user who can open the maximum number of processes.

Finding Shared Objects

When you query the list of handles using the `Get-NtHandle` command, you also get the address of the object in kernel memory. When you open the same kernel object, you'll get different handles, but they will still point to the same kernel object address.

You can use the object address to find processes that share handles. This can be interesting for security in cases when an object is shared between two processes at different privileges. The low-privileged process might be able to modify the properties of the object to bypass security checks in the higher privileged process, enabling it to gain additional privileges.

In fact, I used this technique to find the security issue CVE-2019-0943 in Windows. At the root of the issue was a privileged process, the Windows Font Cache, which shared `Section`

handles with a low-privileged process. The low-privileged process could map the shared `Section` to be writeable and modify contents that the privileged process assumed couldn't be modified. This resulted in the low privileged process being able to modify arbitrary memory in the privileged process, resulting in privileged code execution.

Listing 2-38 gives an example of finding writeable `Section` objects shared between two processes.

```
1 PS> $ss = Get-NtHandle -ObjectType Section -GroupByAddress
  | Where-Object ShareCount -eq 2
  PS> $mask = Get-NtAccessMask -SectionAccess MapWrite
2 PS> $ss = $ss | Where-Object { Test-NtAccessMask $_.AccessIntersection $mask }
  PS> foreach($s in $ss) {
3     $count = ($s.ProcessIds | Where-Object {
              Test-NtProcess -ProcessId $_ -Access DupHandle
      }).Count
      if ($count -eq 1) {
          $s.Handles | Select ProcessId, ProcessName, Handle
      }
  }
  ProcessId ProcessName       Handle
  --------- -----------       ------
       9100 Chrome.exe          4400
       4072 audiodg.exe         2560
```

Listing 2-38 Finding shared `Section` handles

We first get the handles, specifying the `GroupByAddress` parameter 1. Instead of returning a list of handles, it returns a list of groups organized based on the kernel object address.

You can also group handles using the built-in `Group-Object` command; however, the groups returned by `GroupByAddress` have additional properties, including `ShareCount`, which indicates the number of unique processes an object is shared with. We filter to include only handles that are shared between two processes.

Next, we want to find `Section` objects that can be mapped as writeable. We first check that all the handles have `MapWrite` access. As mentioned earlier, the `Section` object protection must be also be writeable for us to be able to map it as writeable, but checking for `MapWrite` access is the simple proxy, as oddly,

you can't query for the original protection that was assigned when the `Section` was created. We use the `AccessIntersection` property, which contains the granted access rights shared between all the handles [2].

Now that we have potential candidates for shared sections, we need to work out which meet the criteria: that we can access only one of the processes containing the `Section` handle. We're make another assumption: if we can open only one of the two processes that share the handle for `DupHandle` access, then we've got a `Section` shared between a privileged and a low-privileged process [3]. After all, if you had `DupHandle` to both processes, you could already compromise the processes by stealing all their handles or duplicating their process handle, and if you couldn't get `DupHandle` to either process, then you couldn't get access to the `Section` handle at all.

The result shown in Listing 2-38 is a shared section between Chrome and the Audio Device Graph process. The shared section is used to play audio from the browser, and it's probably not a security issue. However, if you run the script on your own system, you might find shared sections that are.

Note that once the `Section` object is mapped into memory, the handle is no longer required. Therefore, you might miss some shared sections that were mapped when the original handle closed. It's also highly likely you'll get false positives, such as `Section` objects that are intentionally writeable by everyone. The goal here is to find a potential attack surface on Windows. You must then go and inspect the handles to see if sharing them has introduced a security issue.

Modifying a Mapped Section

If you find an interesting `Section` object to modify, you can map it into memory using `Add-NtSection`. But how do you modify the mapped memory? The simplest approach from the command line is to use the `Write-NtVirtualMemory` command, which supports passing a mapped section and an array

of bytes to write. Listing 2-39 demonstrates this technique by assuming you have a handle of interest in the `$handle` variable.

```
1 PS> $sect = $handle.GetObject()
  PS> $map = Add-NtSection -Section $sect -Protection ReadWrite
2 PS> $random = Get-RandomByte -Size $map.Length
  PS> Write-NtVirtualMemory -Mapping $map -Data $random
  4096

  ③ PS> Out-HexDump -Buffer $map -Length 16 -ShowAddress -ShowHeader
                    00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
  ------------------------------------------------------------------
  000001811C860000: DF 24 04 E1 AB 2A E1 76 EB 19 00 8D 79 28 9C BA
```

Listing 2-39 Mapping and modifying a `Section` object

We can call the `GetObject` method on the handle to duplicate it into the current process and return a `Section` object 1. For this to succeed, the process in which you're running this command must be able to access the process with the handle. We then map the handle as read-write into the current process's memory.

We can now create a random array of bytes up to the size of the mapped section and write them to the memory region using `Write-NtVirtualMemory` 2. This is a quick and dirty fuzzer for the shared memory. The hope is that by modifying the memory, the privileged process will mishandle the contents of the memory region. If the privileged process were to crash, we should investigate it to determine whether we could control the crash using a more targeted modification of the shared memory.

Of course, you can display the memory using `Out-HexDump` 3. One of the useful features of this command over the built-in `Format-Hex` is that it'll print the address in memory based on the mapped file, whereas `Format-Hex` just prints an offset starting at `0`.

You can also create a GUI hex-editor using the `Show-NtSection` command and passing it a section object to edit. As the section can be mapped into any process, writing it in the GUID hex-editor will also modify all other mappings of that section. Here is the command to display the hex-editor:

```
PS> Show-NtSection -Section $sect
```

Figure 2-6 shows an example of the editor generated by running the previous command.

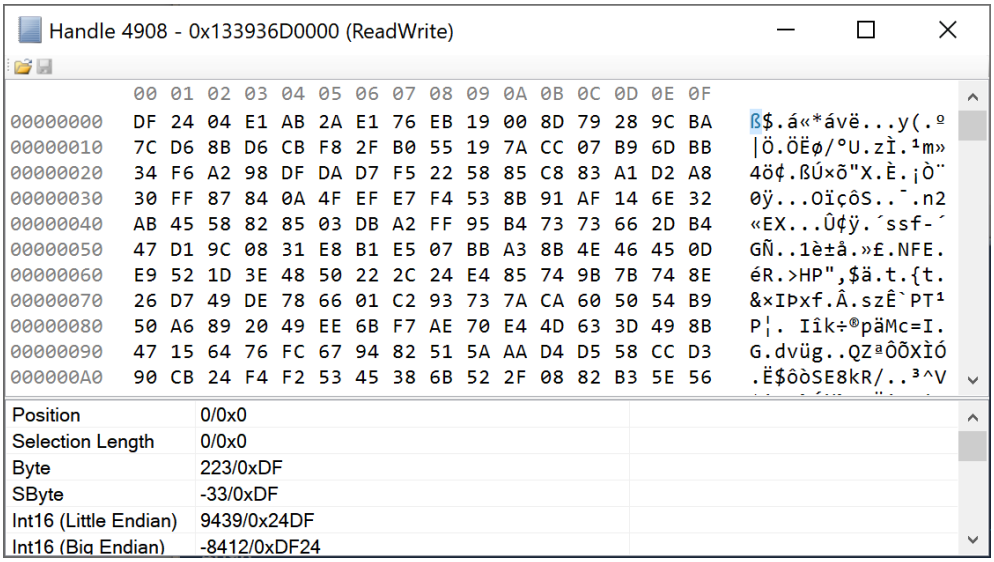

Figure 2-6 The section editor GUI

The GUI shown in Figure 2-6 maps the section into memory and then displays it in a hex editor form. If the section is writeable, you can modify the contents of the memory through the editor.

Finding Writeable and Executable Memory

In the chapter's final worked example, we'll find memory that is both writeable and executable. In Windows, for a process to execute instructions, the memory must be marked as executable. However, it's also possible to map the memory as both writeable and executable. Malware sometimes uses this combination of permissions to inject shell code into a process and run malicious code using the host process's identify.

Finding executable and writeable memory, as we do in Listing 2-40, might indicate that something malicious is going on, although in most cases, this memory will be benign. For example,

the .NET runtime creates writeable and executable memory to perform just-in-time (JIT) compilation of the .NET byte code into native instructions.

```
PS> $proc = Get-NtProcess -ProcessId $pid -Access QueryLimitedInformation
PS> Get-NtVirtualMemory -Process $proc | Where-Object {
      $_.Protect -band "ExecuteReadWrite"
}
Address           Size   Protect          Type    State  Name
-------           ----   -------          ----    -----  ----
0000018176450000 4096   ExecuteReadWrite Private Commit
0000018176490000 8192   ExecuteReadWrite Private Commit
0000018176F60000 61440  ExecuteReadWrite Private Commit
--snip--
PS> $proc.Close()
```

Listing 2-40 Finding executable and writeable memory in a process.

We start by opening a process for `QueryLimitedInformation` access, which is all we need to enumerate the virtual memory regions. Here, we're opening the current PowerShell process; as PowerShell is .NET, we know it will have some writeable and executable memory regions, but the process you open can be anything you want to check.

We then enumerate all the memory regions using `Get-NtVirtualMemory` and filter on the `ExecuteReadWrite` protection type. We need to use a bitwise AND operation as there are additional flags that can be added to the protection, such as `Guard`, which creates a guard page that prevents doing a direct equality check.

Wrapping Up

This chapter provided a tour through the Windows kernel and its internals. The kernel consists of many separate subsystems, such as the security reference monitor, the object manager, the configuration manager (or registry), the I/O manager, and the process and thread manager.

You learned about how the object manager manages kernel resources and types, how to access kernel resources through system calls, and how handles are allocated with specific access

rights. You also accessed object manager resources through the `NtObject` drive provider as well as through individual commands.

I then discussed the basics of process and thread creation and demonstrated the use of commands such as `Get-NtProcess` to query for process information on the system. I explained how to inspect the virtual memory of a process, as well as some of the individual memory types.

A user doesn't directly interact with the kernel; instead, user-mode applications power the user experience. In the next chapter, we'll discuss the user-mode components in more detail.

3

USER-MODE APPLICATIONS

In the previous chapter, we discussed the Windows kernel. But a user doesn't typically interact directly with the kernel. Instead, they interact with user-facing applications, such as word processors and file managers. This chapter will detail how these user-mode applications are created and how they interact with the kernel to provide services to the user.

We'll start by discussing the Win32 application programming interfaces (APIs) designed for user-mode application development and how they relate to the design of the Windows operating system. Then, we'll cover the structure of the Windows user interface and how you can inspect it programmatically.

Multiple users of a Windows system can all access a user interface at the same time; we'll cover how console sessions can isolate one user's interface and application resources from those of other users on the same system.

To understand how user-mode applications function, it's also important to understand how the provided APIs interface with the underlying kernel system call interface. We'll look at this, along with the conversion process that filepaths must undergo to become compatible with the kernel. We'll then consider how Win32 handles process and thread creation and describe some important system processes.

Win32 and the User-Mode Windows APIs

Most of the code that runs on Windows does not directly interact with system calls. This is an artifact of the *Windows NT* operating system's original design. Microsoft initially developed Windows NT as an updated version of IBM's OS/2 operating system, intending it to have multiple subsystems that implemented different APIs. At various times, it supported POSIX, OS/2, and the Win32 APIs.

Eventually, Microsoft's relationship with IBM went sour, and Microsoft took the API set it had developed for Windows 95, *Win32*, then built a subsystem to implement it. The largely unloved OS/2 subsystem was removed in Windows 2000, while POSIX survived until Windows 8.1. By Windows 10, Win32 was the only remaining subsystem (though Microsoft subsequently implemented Linux compatibility layers, such as Windows Subsystem for Linux, that don't use the old subsystem extension points).

To allow for these multiple APIs, the Windows kernel implements a generic set of system calls. It's the responsibility of each subsystem's specific libraries and services to convert their APIs to the low-level system call interface. Figure 3-1 shows an overview of the Win32 subsystem API libraries.

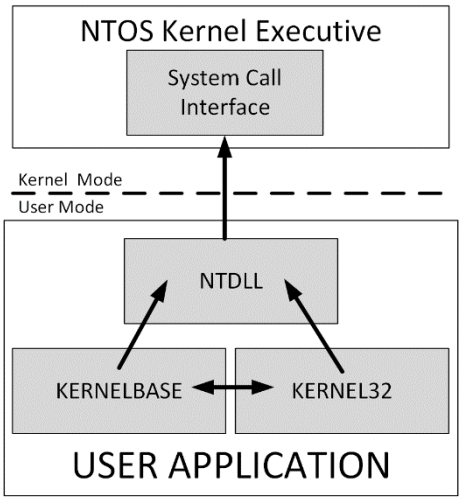

Figure 3-1 The Win32 API modules

As you can see, the core of the Win32 APIs is implemented in the *KERNEL32* and *KERNELBASE* libraries. These libraries then call methods in the system-provided *NT Layer library (NTDLL)*, which implements system call dispatches, as well as a copy of the kernel runtime library APIs.

Most user-mode applications do not directly contain the implementation of the Windows APIs. Instead, NTDLL includes the *dynamic link library (DLL)* loader, which loads new libraries on demand. The loading process is mostly opaque to the developer: when building a program, you will link against a set of libraries, and the compiler and toolchain will automatically add an import table to your executable file to reflect your dependencies. The DLL loader will then inspect the import table, automatically load any dependent libraries, and resolve the imports. You can also specify exported functions from your application so that other code can rely on your APIs.

Loading a New Library

It's possible to access exported functions manually at runtime without needing an import table entry. You can load a new library using the `LoadLibrary` Win32 API, which is exposed to PowerShell using the `Import-Win32Module` command. To find the memory address of a function exported by a DLL, use the Win32 API `GetProcAddress`, exposed with the PowerShell `Get-Win32ModuleExport` command (Listing 3-1).

```
1 PS> $lib = Import-Win32Module -Path "kernel32.dll"
  PS> $lib
  Name          ImageBase        EntryPoint
```

```
----          ---------         ----------
2 KERNEL32.DLL 00007FFA088A0000 00007FFA088B7C70

3 PS> Get-Win32ModuleExport -Module $lib
Ordinal Name                            Address
------- ----                            -------
1       AcquireSRWLockExclusive         NTDLL.RtlAcquireSRWLockExclusive
2       AcquireSRWLockShared            NTDLL.RtlAcquireSRWLockShared
3       ActivateActCtx                  0x7FFA088BE640
4       ActivateActCtxWorker            0x7FFA088BA950
--snip--

4 PS> "{0:X}" -f (Get-Win32ModuleExport -Module $lib -ProcAddress
"AllocConsole")
7FFA088C27C0
```

Listing 3-1 Exports for the *KERNEL32* library

In Listing 3-1, we use PowerShell to load the *KERNEL32* library and enumerate the exported and imported APIs. To do this, we first need to ensure it's loaded into memory 1 using `Import-Win32Module`. The *KERNEL32* library is always loaded, so this command will just return the existing loaded address; for other libraries, however, the load will cause the DLL to be mapped into memory and initialized.

WARNING The `Import-Win32Module` command will load a DLL into memory and potentially execute code. In this example, this is acceptable, as *NTDLL* is part of the trusted system libraries. However, do not use the command on an untrusted DLL, especially if you're analyzing malware, as it might result in malicious code execution. To be safe, always perform malware analysis on a segregated system dedicated to that purpose.

Once it's loaded into memory, we can display some properties for the library 2. These include the name of the library, as well as the loaded memory address and the address of the `EntryPoint`. A DLL can optionally define a function, `DllMain`, to run when the DLL is loaded. The `EntryPoint` address is the first instruction in memory to execute when the DLL is loaded.

Next, we can dump all exported functions from the DLL 3. In this case, we see three pieces of information: `Ordinal`, `Name`, and `Address`. The `Ordinal` is a small number that uniquely identifies the exported function in the DLL. It's possible to import an API by its ordinal number, which means there is no need to export a name; you'll see certain names missing from export tables in DLLs whenever Microsoft doesn't want to officially support the function as a public API.

The `Name` is just the name of the exported function. It doesn't need to match what the function was called in the original source code, although typically it does. Finally, `Address` is the address in memory of the function's first instruction. You'll notice that the first two exports have a string instead of an address. This is a case of *export forwarding*; it allows a DLL to export a function by name and have the loader automatically redirect it to another DLL. In this case, `AcquireSRWLockExclusive` is implemented as `RtlAcquireSRWLockExclusive` in NTDLL. We can also use `Get-Win32ModuleExport` to lookup a single exported function using the `GetProcAddress` API 4.

Viewing Imported APIs

In a similar fashion, we can view the APIs that an executable has imported from other DLLs using the `Get-Win32ModuleImport` command, as shown in Listing 3-2.

```
PS> Get-Win32ModuleImport -Path "kernel32.dll"
DllName                                          FunctionCount DelayLoaded
-------                                          ------------- -----------
api-ms-win-core-rtlsupport-l1-1-0.dll            13            False
ntdll.dll                                        378           False
KERNELBASE.dll                                   90            False
api-ms-win-core-processthreads-l1-1-0.dll        39            False
--snip--

PS> Get-Win32ModuleImport -Path "kernel32.dll" -DllName "ntdll.dll" |
Where-Object Name -Match "^Nt"
Name                            Address
----                            -------
NtEnumerateKey                  7FFA090BC6F0
NtTerminateProcess              7FFA090BC630
NtMapUserPhysicalPagesScatter   7FFA090BC110
NtMapViewOfSection              7FFA090BC5B0
--snip--
```

Listing 3-2 Enumerating imports for the *KERNEL32* library

Listing 3-2 starts by calling `Get-Win32ModuleImport` and specifying the *KERNEL32 DLL* as the path. When you specify a path, the command will call `Import-Win32Module` for you and display all imports, each of which includes the name of the DLL to load and the number of functions imported. The final column indicates whether the DLL is *delay loaded*. This is a performance optimization; it allows a DLL to be loaded at the point when one of its exported functions is used. This delay avoids loading all DLLs into memory during initialization if they are rarely accessed.

Next, we dump the imported functions for a DLL. As the executable can import code from multiple libraries, we specify the one we want using the `DllName` property. We then filter to all imported functions starting with the `Nt` prefix; this allows us to see exactly what system calls *KERNEL32* imports from *NTDLL*.

API SETS

You might notice something odd in the list of imported DLL names in Listing 3-2. If you search your filesystem for the *api-ms-win-core-rtlsupport-l1-1-0.dll* file, you won't find it. This is because the DLL name refers to an API set name. *API sets* were introduced in Windows 7 to modularize the system libraries, and they abstract from the name of the set to the DLL that exports the API.

API sets allow an executable to run on multiple different versions of Windows, such as a client, a server, or an embedded version, and change its functionality at runtime based on what libraries are available. When the DLL loader encounters one of these API set names, it consults a table loaded into every process, sourced from the file *apisetschema.dll*, that maps the name to the real DLL:

```
PS> Get-NtApiSet api-ms-win-core-rtlsupport-l1-1-0.dll
Name                                HostModule Flags
----                                ---------- -----
api-ms-win-core-rtlsupport-l1-1-1 ntdll.dll  Sealed

PS> Get-Win32ModuleImport -Path "kernel32.dll" -ResolveApiSet
DllName                           FunctionCount DelayLoaded
-------                           ------------- -----------
ntdll.dll                         392           False
KERNELBASE.dll                    867           False
ext-ms-win-oobe-query-l1-1-0.dll 1             True
RPCRT4.dll                        10            True
```

You can query the details for an API set by using the `Get-NtApiSet` command and specifying the name of the API set 1. We can see that in this case the API set resolves to the *NTDLL* library. You can also specify the `ResolveApiSet` parameter to the `Get-Win32ModuleImport` command to group the imports based on the real DLLs.

If you compare the output in Listing 3-2 to that of the same command shown here, you'll notice that the resolved imports is much shorter, and that the core libraries have gained additional function imports. Also notice the unresolved API set name, *ext-ms-win-oobe-query-l1-1-0.dll*. Any API set with the prefix `api` should always be present, whereas one with the prefix `ext` might not be. In this case, the API set is not present, and trying to call the imported function will fail. However, because the function is marked as delay loaded, an executable can check whether the API set is available before calling the function by using the `IsApiSetImplemented` Win32 API.

Searching for DLLs

When loading a DLL, the loader creates an image section object from the executable file and maps it into memory. The kernel is responsible for mapping the executable memory; however, user-mode code still needs to parse the import and export tables.

Let's say you pass the string *ABC.DLL* to the `LoadLibrary` API. How does the API know where to find that DLL? If the file hasn't been specified as an absolute path, the API implements a path-searching algorithm. The algorithm, as originally implemented in Windows NT 3.1, searches for files in the following order:

1. The same directory as the current process's executable file
2. The current working directory
3. The Windows *SYSTEM32* directory
4. The *WINDOWS* directory
5. Each semicolon-separated location in the `PATH` environment variable

The problem with this load order is that it can lead to a privileged process loading a DLL from an insecure location. For example, if a privileged process changed its current working directory using the `SetCurrentDirectory` API to a location a less-privileged user could write to, the DLL would be loaded from that location before any DLL from the *SYSTEM32* directory. This attack is called *DLL hijacking*, and it's a persistent problem on Windows.

Vista changed the default load order to the following, which is safer:

1. The same directory as the current process's executable file
2. The Windows *SYSTEM32* directory
3. The *WINDOWS* directory
4. The current working directory
5. Each semicolon-separated location in the `PATH` environment variable

Here, we no longer load from the current working directory before the *SYSTEM32* or *WINDOWS* folders. However, if an attacker could write to the executable's directory, a DLL hijack could still take place. Therefore, if an executable run as a privileged process, only administrators should be able to modify its directory to prevent a DLL hijack from occurring. If an application tries to load a DLL that doesn't exist, it could still potentially load code from the current working directory or the `PATH`.

THE DLL FILE EXTENSION

A separate loading quirk involves the handling of file extensions in a DLL's filename. If no extension is specified, the DLL loader will automatically add a *.DLL* extension. If any extension is specified, the filename is treated as-is. Finally, if the extension consists of a single period (for example

LIB.), the loader removes the period and tries to load the file *LIB*.

This file extension behavior can introduce mismatches between the DLL an application is trying to load and its filename. For example, an application might check that the file *LIB* is valid (that is, correctly cryptographically signed); however, the DLL loader would then load *LIB.DLL*, which was not checked. This can result in security vulnerabilities if you can trick a privileged application into loading the wrong DLL into memory, as the entry point will execute in the privileged context.

While the DLL loader will normally turn to the disk to retrieve a library, some libraries are used so often that it makes sense to pre-initialize them. This improves performance and prevents the DLLs from being hijacked. Two obvious examples are *KERNEL32* and *NTDLL*.

Before any user applications start on Windows, the system configures a `KnownDlls` OMNS directory containing a list of pre-loaded image sections. A `KnownDlls` section object's name is just the filename of the library. The DLL loader can check `KnownDlls` first before going to the disk. We can list the object directory using the `NtObject` drive, as shown in Listing 3-4.

```
PS> ls NtObject:\KnownDlls
Name                                    TypeName
----                                    --------
kernel32.dll                            Section
kernel.appcore.dll                      Section
windows.storage.dll                     Section
ucrtbase.dll                            Section
MSCTF.dll                               Section
--snip--
```

Listing 3-3 Listing the contents of the `KnownDlls` object directory

We've covered the basics of the Win32 subsystem and how it uses libraries to implement the APIs that a user-mode application can use to interface with the operating system. We'll come back to the Win32 APIs later, but first, we must discuss the Window's user interface, which is inextricably linked to how the Win32 subsystem functions.

The Win32 GUI

The name "Windows" refers to the structure of the operating system's GUI. This GUI consists of one or more windows that the user can interact with using controls such as buttons and text input. Since Windows 1.0, the GUI has been the most important feature of the operating system, so it should come as no surprise that its model is complex. The implementation of the GUI is split between the kernel and user mode, as shown in Figure 3-2.

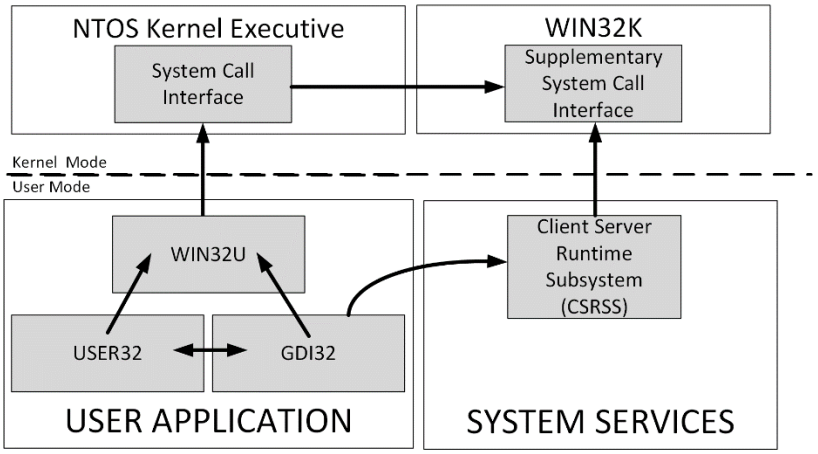

Figure 3-2 The Win32 GUI modules

You might notice that the left-hand side of Figure 3-2 looks a lot like Figure 3-1, which showed the modules for the normal Win32 APIs. Instead of *NTDLL*, however, there is *WIN32U*, which implements system call stubs for the kernel to call. Two libraries call *WIN32U*: *USER32* and *GDI32*. *USER32* implements the window UI elements and generally manages the GUI, whereas *GDI32* implements drawing primitives, like fonts and shapes.

One big difference between Figure 3-2 and Figure 3-1 is that the GUI is not actually implemented inside the main NTOS kernel executive. Instead, its system calls are implemented in the *WIN32K* driver, which interfaces with the object manager, the kernel, and the display drivers to handle user interactions and display the results. The *WIN32K* driver also implements a system call table that is separate from the kernel's.

NOTE In versions of Windows prior to 10, the system call dispatch code in *WIN32U* was embedded directly inside the user-mode DLLs. This made it hard for an application to directly call *WIN32K* system calls without writing assembly language.

The GUI APIs also interact with a special privileged process: the *Client Server Runtime Subsystem (CSRSS)*. The CSRSS process is responsible for handling certain privileged operations for lower-privileged clients, such as configuring per-user drive mappings, process management, and error handling. Prior to Windows NT 4, CSRSS was much more important: almost all the GUI code was implemented in user-mode. But, for the hardware available in the early 1990s, running everything in user-mode wasn't very performant. To

improve performance, the GUI code was split between the *WIN32K* kernel driver and some vestigial components left inside CSRSS.

GUI Kernel Resources

The GUI is made up of four types of kernel resources:

Window Stations
Objects that represent the connection to the screen and the user interface, such as the keyboard and mouse

Windows
GUI elements for interacting with the user, accepting input, and displaying a result

Desktops
Objects that represent the visible desktop and act as a host for windows

Drawing Resources
Bitmaps, fonts, or anything else that needs to be displayed to the user

While the Win32 kernel and user components handle the windows, the window stations and desktops are accessible through the object manager. There is a kernel object type for window stations and desktops, as shown in Listing 3-5.

```
PS> Get-NtType WindowStation,Desktop
Name
----
WindowStation
Desktop
```

Listing 3-4 Showing the `WindowStation` and `Desktop` type objects

A window station is assigned to a process either at process startup or using the `NtUserSetProcessWindowStation` API. Desktops are assigned on a per-thread basis using `NtUserSetThreadDesktop`. We can query the names of the window stations and desktops with the commands in Listing 3-5.

```
1 PS> Get-NtWindowStationName
  WinSta0
  Service-0x0-b17580b$

2 PS> Get-NtWindowStationName -Current
  WinSta0

  3 PS> Get-NtDesktopName
  Default
```

```
WinLogon
4 PS> Get-NtDesktopName -Current
Default
```

Listing 3-5 Displaying all window stations, the current window station, and the desktops

We start by querying the names of all available window stations 1. In this example, there are two: the default `WinSta0` windows station and `Service-0x0-b17580b$`, which another process has created. The ability to create separate window stations allows a process to isolate its GUI interactions from other processes running at the same time. However, `WinSta0` is special, as it is the only object connected to the user's console.

Next, we check what our current window station name is by using the `-Current` parameter 2. We can see we're on `WinSta0`.

We can then query for the names of the desktops on our current window station 3. We see only two desktops: `Default` and `WinLogon`. The `WinLogon` desktop will be visible only if you run the `Get-NtDesktopName` command as an administrator, as it's used solely to display the login screen, which a normal user application shouldn't be able to access. `Desktop` objects must be opened relative to a window station path; there isn't a specific object directory for desktops. Therefore, the name of the desktop reflects the name of the window station object.

Finally, we can check the name of the current thread's desktop 4. The desktop we're attached to is shown as `Default`, as that's the only desktop available to normal user applications. We can enumerate the windows created in a desktop using `Get-NtDesktop` and `Get-NtWindow` (Listing 3-6).

```
PS> $desktop = Get-NtDesktop -Current
PS> Get-NtWindow -Desktop $desktop
Handle   ProcessId ThreadId ClassName
------   --------- -------- ---------
66104    11864     12848    GDI+ Hook Window Class
65922    23860     18536    ForegroundStaging
65864    23860     24400    ForegroundStaging
65740    23860     20836    tooltips_class32
--snip--
```

Listing 3-6 Enumerating windows for the current desktop

As you can see, each window has a few properties. First is its *handle*, which is unique to the desktop. This is not the same type of handle we discussed in the last chapter for kernel objects; instead, it's a value allocated by the WIN32 subsystem.

To function, a window receives *messages* from the system. For example, when you click a mouse button on a window, the system will send a message to notify the window of the click and what mouse button was pressed. The window can then handle the message and change its behavior accordingly. You can also manually send messages to a window using the `SendMessage` or `PostMessage` APIs.

A windows message consists of a numeric identifier—such as 0x10, which represents the message `WM_CLOSE` to close a window—and two additional parameters. The meaning of the two parameters depends on the message. For example, if the message is `WM_CLOSE`, then neither parameter is used; for other messages, they might represent pointers to strings or integer values.

Messages can be sent or posted. The difference between sending and posting a message is that sending waits for the window to handle message and return a value, while posting just sends the message to the window and returns immediately.

In Listing 3-6, the `ThreadId` identifies the thread that created the windows using an API such as `CreateWindowEx`. A window has what's called *thread affinity*, which means that only the creating thread can manipulate the state of the window and its handle messages. However, any thread can send messages to the window. To handle messages, the creating thread must run a *message loop*, which calls the `GetMessage` API to receive the next available message and then dispatches it to the Window's message handler callback function using the `DispatchMessage` API. When an application is not running the loop, you might see Windows applications hanging, as without the loop, the GUI cannot be updated.

The final column in Listing 3-6 is the `ClassName`. This is the name of a *window class*, which acts as a template for a new window. When `CreateWindowEx` is called, the `ClassName` is specified and the window is initialized with default values from the template, such as the style of the border or a default size. It's common for an application to register its own classes to handle unique windows. Alternatively, it can use system-defined classes for things like buttons and other common controls.

Window Messages

Let's see a simple example in which we send a windows message to a window. Before running the code in Listing 3-7, make sure there is at least one copy of Notepad running.

```
1 PS> $top = Get-NtWindow | Where-Object ClassName -eq "Notepad" |
  Select-Object -First 1
2 PS> $edit = Get-NtWindow -Children -Parent $top |
```

```
   Where-Object ClassName -eq "Edit"
   PS> $top, $edit | Out-Host
   Handle  ProcessId ThreadId ClassName
   ------  --------- -------- ---------
3  1509748 16128     20152    Notepad
   917770  16128     20152    Edit

4  PS> while($true) {
       $len = Send-NtWindowMessage -Window $edit -Message 0xE -Wait
       Start-Sleep -Seconds 1
       Write-Host "Length: $len"
   }
   Length: 0
   Length: 5
   --snip--
```

Listing 3-7 Sending the `WM_GETTEXTLENGTH` message to the *notepad* edit window

Listing 3-7 starts by finding a window with the `ClassName` of `Notepad` 1. This is a window class that Notepad creates when it starts up, and it's used as the template for Notepad's main window. The *notepad.exe* executable is fundamentally a wrapper around the system *edit control*, which is created as a child window of the main window; we can get it by using the `Children` parameter, passing the main window handle and retrieving the `Edit` class 2.

If there's a copy of *notepad.exe* running on the current desktop, you should see the output 3, although the handle and ID values will be different. In a loop, we send the `WM_GETTEXTLENGTH` message (which is message number 0xE) to the edit window 4. The edit window will handle the message, returning the number of characters we enter in the edit window. While the loop is running, try typing some more characters into notepad's window; you should see the length change. You could also use `WM_GETTEXT` message to query or `WM_SETTEXT` to set the text contents of the edit box, but that's perhaps more advanced than necessary here.

There is much more to explore in the windowing system, but those details are outside the scope of this book. I recommend Charles Petzold's seminal work on the topic, *Programming Windows (5th Edition)* if you want to know more about the development of Win32 applications. Next, we need to describe how multiple users can use their own user interfaces on the same system through the creation of console sessions.

Console Sessions

The first version of Windows NT allowed multiple users to be authenticated at the same time and each run processes. However, before the introduction of *Remote Desktop Services (RDS)*, it wasn't possible for different

interactive desktops to run multiple user accounts concurrently on the same machine. All authenticated users needed to share a single physical console. Windows NT 4 introduced multiple-console support as an optional, server-only feature before it became standard in Windows XP.

RDS is a service on Windows workstations and servers that allows you to remotely connect to the GUI and interact with the system. It's used for remote administrators but also to provide shared hosting for multiple users on the same network-connected system. Moreover, its functionality has been repurposed to support a mechanism that can switch between users on the same system without having to log users out.

To prepare for a new user login to Windows, the session manager service creates a new session on the console. This session is used to organize a user's window station and desktop objects so that they're separate from those belonging to any other user authenticated at the same time. The kernel creates a `Session` object to keep track of resources, and a named reference to the object is stored in the `KernelObjects` OMNS directory. However, the `Session` object is usually only exposed to the user as an integer. There's no randomness to the integer; it's just incremented as each new *console session* is created.

The session manager starts several processes in this new session before any user logs in. These include a dedicated copy of CSRSS and the `WinLogon` process, which display the *credentials* user interface and handle the authentication of the new user. We'll dig into the authentication process more in Chapter 10.

The console session that a process belongs to is assigned when the process starts. (Technically, the console session is specified in the access token, but that's a topic for Chapter 4.) We can observe the processes running in each session by running some PowerShell commands, as shown in Listing 3-8.

```
PS> Get-NtProcess -InfoOnly | Group-Object SessionId
Count Name        Group
----- ----        -----
  156 0           {, System, Secure System, Registry...}
    1 1           {csrss.exe}
    1 2           {csrss.exe}
  113 3           {csrss.exe, winlogon.exe, fontdrvhost.exe, dwm.exe...}
```

Listing 3-8 Displaying a process's console sessions using `Get-NtProcess`

Windows has only one physical console, which is connected to the keyboard, mouse, and monitor. However, it's possible to create a new remote desktop over the network by using a client that communicates using the remote desktop protocol (RDP).

It's also possible to switch the user logged on to the physical console; this is used for Windows' *fast user switching* feature. When the physical console switches to a new user, the previous user is still logged on and running in the background, but you cannot interact with that user's desktop.

Each console session has its own special kernel memory region. Having duplicated resources ensures that the console sessions are separated; this acts as a security boundary. Session number 0 is special, in that it's only for privileged services and system management. It's normally not possible to use a GUI with processes running in this session.

SHATTER ATTACKS

Prior to Windows Vista, both services and the physical console ran in session 0. As any process was able to send window messages to any other process in the same session, this introduces a security weakness called a *shatter attack*.

A shatter attack occurs when a normal user can send a window message to a more privileged application in the same session to elevate privileges. For example, the `WM_TIMER` message could accept an arbitrary function pointer that the more privileged application would call when it received the message. A normal user could send this message with a carefully chosen function pointer and get arbitrary code execution in the context of the privileged application.

Windows Vista mitigates shatter attacks with two related security features that are still present in the latest versions of Windows. The first is *Session 0 Isolation*, which was moved the physical console from session 0 so that a normal user application could not send messages to services.

The second feature is *User Interface Privilege Isolation (UIPI)*. UIPI prevents lower-privileged processes from interacting with windows at higher privileges. Therefore, even if a service created a window on the user's desktop, the system would reject any messages sent by the user to a privileged service.

Another important feature associate with the console sessions is the separation of named objects. In the previous chapter, we discussed the BNO directory, a global location for named objects. However, if multiple users can logged into the system at the same time, you could easily get name conflicts. Windows solves the name-conflict problem by creating a per-console session directory for the BNO under the directory *\Sessions*\<N>*\BaseNamedObjects*, where <N> is the console session ID. The *\Sessions* directory also contains a directory for the window stations, under *\Sessions*\<N>*\Windows*, which ensures that window resources, too, are separated. You can list the BNO of the current console session with the *NtObjectSession* drive, as shown in Listing 3-10.

```
PS> ls NtObjectSession:\ | Group-Object TypeName
Count Name                      Group
----- ----                      -----
  246 Semaphore                 {SM0:10876:304:WilStaging_02_p0h...}
  263 Mutant                    {SM0:18960:120:WilError_02,...}
  164 Section                   {fd8HWNDInterface:3092e,...}
```

```
  159 Event                     {BrushTransitionsCom... }
    4 SymbolicLink              {AppContainerNamedObjects, Local, Session,
Global}
    1 ALPC Port                 {SIPC_{2819B8FF-EB1C-4652-80F0-7AB4EFA88BE4}}
    2 Job                       {WinlogonAccess, ProcessJobTracker1980}
    1 Directory                 {Restricted}
```

Listing 3-9 Enumerating the contents of a session's BNO directory and grouping items by `TypeName`

There is no per-console session BNO for session 0; it uses the global BNO directory.

THE ORIGINS OF REMOTE DESKTOP SERVICES

The RDS feature didn't originate at Microsoft. Rather, a company called Citrix developed the technology for Windows and licensed it to Microsoft for use in NT 4. The technology was originally called Terminal Services, so it's common to sometime see it referred to using that name. To this day, it's possible to buy a Citrix version of RDS that uses a different network protocol, Independent Computing Architecture (ICA), instead of Microsoft's RDP.

Comparing Win32 APIs and System Calls

As a case study, let's compare a Win32 API and its equivalent system call. Not all system calls are directly exposed through Win32, and in some cases, the Win32 API reduces the functionality of exposed system calls. However, it's worth describing common differences between a system call and its Win32 API equivalent.

We'll pick the `CreateMutexEx` API, as it's the Win32 version of `NtCreateMutant` system call we described in the last chapter. The API has the C prototype shown in Listing 3-10.

```
HANDLE CreateMutexEx(
    SECURITY_ATTRIBUTES* lpMutexAttributes,
    const WCHAR*         lpName,
    DWORD                dwFlags,
    DWORD                dwDesiredAccess
);
```

Listing 3-10 The prototype for the `CreateMutexEx` Win32 API

Compare it to the `NtCreateMutant` prototype, shown in Listing 3-11:

```
NTSTATUS NtCreateMutant(
  HANDLE*              MutantHandle,
  ACCESS_MASK          DesiredAccess,
  OBJECT_ATTRIBUTES*   ObjectAttributes,
  BOOLEAN              InitialOwner
```

```
);
```

Listing 3-11 The prototype for the NT system call `NtCreateMutant`

The first difference between the prototypes is that the Win32 API returns a handle to the kernel object, while the system call returns an `NTSTATUS` code (and receives the handle instead via a pointer as the first parameter).

You might wonder: how do errors get propagated back to an API's caller, if not via an `NTSTATUS` code? In this respect, the Win32 APIs are not always consistent. If the API returns a handle, then it's common to return a value of `NULL`. However, some APIs, such as the file APIs, return the value `-1` instead. If a handle it not returned, it's common to return a Boolean value, with `TRUE` indicating success and `FALSE` indicating an error.

We can retrieve the error status for the current thread using the `GetLastError` API, which returns an error code. Unlike the `NTSTATUS` code, this error code doesn't have any structure; it's just a number. The `CreateMutexEx` API converts the `NTSTATUS` code to the error using the `RtlNtStatusToDosError` API from *NTDLL*, which also sets the current thread's last error value so it can be queried. We can look up error codes in PowerShell using `Get-Win32Error`, as shown in Listing 3-12.

```
PS> Get-Win32Error 5
ErrorCode Name                 Message
--------- ----                 -------
        5 ERROR_ACCESS_DENIED Access is denied.
```

Listing 3-12 Looking up Win32 error code 5

The second big change between the system call and Win32 API is that the API does not take the `OBJECT_ATTRIBUTES` structure. Instead, it splits the attributes between two parameters: `lpName`, used to specify the object's name, and `lpMutexAttributes`, which is a pointer to a `SECURITY_ATTRIBUTES` structure.

The `lpName` parameter is a 16-bit character-size Unicode NUL-terminated string. Even though the object manager uses the counted `UNICODE_STRING`, the Win32 API uses a C-style terminated string. This means that while the NUL character is a valid character for an object name, it's impossible to specify using the Win32 API.

Another difference is that the name is not a full path to the OMNS location for the object; instead, it's relative to the current session's BNO. This means that if the name is *ABC*, then the final path used is *\Sessions*<N>*\BaseNamedObjects\ABC*, where <N> is the console session ID. If you want to create an object in the global BNO, you can prefix the name

with *Global* (for example, *Global\ABC*). This works because *Global* is a symbolic link to `\BaseNamedObjects`, which is automatically created along with the per-session BNO directory. If you want to simulate this behavior using the `Get` and `Set` PowerShell commands, pass them the `-Win32Path` option, as shown in Listing 3-13.

```
PS> $m = New-NtMutant ABC -Win32Path
PS> $m.FullPath
\Sessions\2\BaseNamedObjects\ABC
```

Listing 3-13 Create a new `Mutant` with `-Win32 Path`

Listing 3-14 shows the `SECURITY_ATTRIBUTES` structure.

```
struct SECURITY_ATTRIBUTES {
  DWORD  nLength;
  VOID*  lpSecurityDescriptor;
  BOOL   bInheritHandle;
};
```

Listing 3-14 The `SECURITY_ATTRIBUTES` structure

The `SECURITY_ATTRIBUTES` structure allows you to specify the security descriptor of the new object, as well as whether the handle should be inheritable. The `CreateMutexEx` Win32 API exposes no other options from `OBJECT_ATTRIBUTES`.

On to the final two parameters in Listing 3-10: `dwDesiredAccess` directly maps to `DesiredAccess`, and the native `InitialOwner` parameter is specified through `dwFlags` with the `CREATE_MUTEX_INITIAL_OWNER` flag. If you try to look up the `CreateMutexEx` export in *KERNEL32*, you might get a surprise (Listing 3-15).

```
PS> Get-Win32ModuleExport "kernel32.dll" -ProcAddress CreateMutexEx
Exception calling "GetProcAddress" with "2" argument(s):
"(0x8007007F) - The specified procedure could not be found."
```

Listing 3-15 Getting `CreateMutexEx` from *KERNEL32*

If you run the command in Listing 3-15, you'll notice you don't receive the address; instead, you get an exception. Did we pick the wrong library? Let's try to find the API by dumping all exports and filtering them by name, as shown in Listing 3-16.

```
PS> Get-Win32ModuleExport "kernel32.dll" | Where-Object Name -Match
CreateMutexEx

Ordinal Name           Address
------- ----           -------
```

```
217     CreateMutexExA 0x7FFA088C1EB0
218     CreateMutexExW 0x7FFA088C1EC0
```

Listing 3-16 Find the `CreateMutexEx` API by listing all exports

As you can see, the `CreateMutexEx` API is there not once, but twice. Each function has a suffix, either `A` or `W`. This is because Windows 95 (where most of the APIs were initially created) didn't natively support Unicode strings, so the APIs used single-character strings in the current text encoding. With the introduction of Windows NT, the kernel became 100 percent Unicode, but it provided two APIs for a single function to enable older Windows 95 applications.

One API, with the `A` suffix, accepts single-character strings, or *ANSI strings*. These APIs convert their strings into Unicode strings to pass to the kernel, and convert them back again if a string needs to be returned. Applications built for Windows NT, on the other hand, can use the API with the `W` suffix, or *wide string*, which doesn't need to do any string conversions. Which API you get when you build a native application depends on your build configuration and is a topic for a completely different book.

Win32 Registry Paths

In Chapter 2, you learned the basics of how to access the registry by using native system calls to retrieve them from the OMNS. The Win32 APIs used to access the registry, such as `RegCreateKeyEx`, do not expose this native path. Instead, you can access registry keys relative to pre-defined root keys. You'll be familiar with these keys if you're ever used the Windows *regedit* application, shown in Figure 3-3.

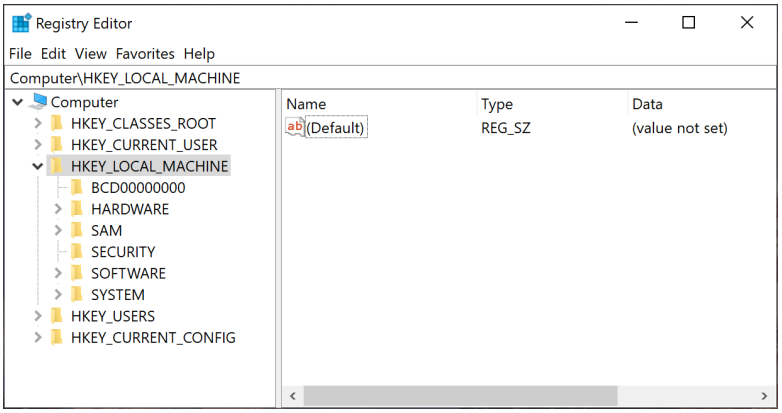

Figure 3-3 The main view of *regedit* utility

The handle values displayed in Figure 3-3 are shown in Table 3-1, along with their corresponding OMNS paths.

Table 3-1 Pre-Defined Registry Handles and Their Native Equivalent

Pre-defined handle name	OMNS path
HKEY_LOCAL_MACHINE	*\REGISTRY\MACHINE*
HKEY_CURRENT_CONFIG	*\REGISTRY\MACHINE\SYSTEM\CurrentControlSet\Hardware Profiles\Current*
HKEY_USERS	*\REGISTRY\USER*
HKEY_CURRENT_USER	*\REGISTRY\USER\<SDDL SID>*
HKEY_CLASSES_ROOT	Merged view of *\REGISTRY\MACHINE\SOFTWARE\Classes* and *\REGISTRY\USER\<SDDL SID>_Classes*

The first three pre-defined handles, `HKEY_LOCAL_MACHINE`, `HKEY_USERS`, and `HKEY_CURRENT_CONFIG`, are not particularly special; they directly map to a single OMNS registry key path. The next handle, `HKEY_CURRENT_USER`, is more interesting; it maps to a hive loaded for the currently authenticated user. The name of the hive's key is the SDDL string of the user's SID.

The final key, `HKEY_CLASSES_ROOT`, which stores information such as file extension mappings, is a merged view of a user's classes hive and the machine's hive. The user's hive takes precedence over the machine hive, allowing the user to change their file extensions without needing an administrator.

Opening Keys

When using the `Get-NtKey` and `New-NtKey` commands, you can specify a Win32 path by using the `Win32Path` parameter (Listing 3-17).

```
PS> Use-NtObject($key = Get-NtKey \REGISTRY\MACHINE\SOFTWARE) {
      $key.Win32Path
}
HKEY_LOCAL_MACHINE\SOFTWARE

PS> Use-NtObject($key = Get-NtKey -Win32Path "HKCU\SOFTWARE") {
      $key.FullPath
}
\REGISTRY\USER\S-1-5-21-818064985-378290696-2985406761-1002\SOFTWARE
```

Listing 3-17 Interacting with the registry using Win32 paths

We start by opening a `Key` object using the `Get-NtKey` command. We use the OMNS path to open the key, then convert the path to its Win32 version using the `Win32Path` property. In this case, we see that

\REGISTRY\MACHINE\SOFTWARE is mapped to *HKEY_LOCAL_MACHINE\SOFTWARE*.

We then do the reverse and open a key using a Win32 name by specifying the `Win32Path` parameter and printing its native OMNS path. Here, we use the current user's hive. Notice we're using a shortened form of the pre-defined key name: `HKCU`, instead of `HKEY_CURRENT_USER`. All of the other pre-defined keys have similar shortened forms. For example, `HKLM` refers to `HKEY_LOCAL_MACHINE`.

In the output, you can see the SDDL SID string, which represents the current user. As we've demonstrated, using the Win32 path to access the current user's hive is much simpler than looking up the current user's SID and opening it with the OMNS path.

Listing the Registry's Contents

In the previous chapter, we showed how to list the registry's contents using the *NtObject* or *NtKey* drive provider paths. For the Win32 registry, you have a few additional options. To simplify accessing the current user's hive, you can use `NtKeyUser`. For example, you can list the current user's software key with the following.

```
PS> ls NtKeyUser:\SOFTWARE
```

PowerShell also comes with built-in drives, `HKLM` and `HKCU`, for the local machine and current user hives, respectively. For example, the equivalent to previous command is the following:

```
PS> ls HKCU:\SOFTWARE
```

Why would you use one of these drive providers over another? Well, the PowerShell module's drive provider allows you to view the entire registry. It also uses the native APIs, which use counted strings, and supports the use of NUL characters in the names of the registry keys and values, while the Win32 APIs uses NUL-terminated C-style strings, which cannot handle embedded NUL characters. Therefore, if a NUL is embedded into a name, it's impossible for the built-in provider to access that key or value. Listing 3-18 demonstrates this.

```
1 PS> $key = New-NtKey -Win32Path "HKCU\ABC`0XYZ"
2 PS> Get-Item "NtKeyUser:\ABC`0XYZ"
  Name    TypeName
  ----    --------
  ABC XYZ Key

3 PS> Get-Item "HKCU:\ABC`0XYZ"
  Get-Item : Cannot find path 'HKCU:\ABC XYZ' because it does not exist.
```

```
PS> Remove-NtKey $key
PS> $key.Close()
```

Listing 3-18 Adding and accessing a registry key with a NUL character

We start by creating a new key with a NUL character in the name, indicated by the *`0* escape 1. If you access this path via the *NtKeyUser* drive, you can successfully retrieve the key 2. However, doing the same with the built-in drive provider doesn't work; it can't find the registry key 3.

This behavior of the Win32 APIs can lead to security issues. For example, it's possible for malicious code to hide registry keys and values from any software that uses the Win32 APIs. This can prevent the malicious code from being detected. We'll see how to detect the use of this hiding technique in "Listing the Registry's Contents" on page XX.

It's also possible to get a mismatch if some software uses the native system calls and other software uses the Win32 APIs. For example, if some code checks the *ABC`0XYZ* path to ensure it has been correctly set up, then hands this to another application, which uses the path with the Win32 APIs, the new application will instead access the unrelated *ABC* key, which hasn't been checked. This could lead to information disclosure issues if the contents of *ABC* were returned to the caller.

The built-in registry provider does have an advantage: it can be used without the installation of an external module. It also allows you to create new keys and add values, which the module's provider does not allow you to do.

DOS Device Paths

Another big difference between the Win32 APIs and the native system calls is how they handle filepaths. In the previous chapter, you saw that we can access a mounted filesystem using a *\Device\VolumeName* path. However, we can't specify this native path using the Win32 APIs. Instead, we'll use well-known paths, such as *C:\Windows*, that have drive letters. Because the drive letter paths are a vestige of MS-DOS, we call them *DOS device paths*.

Of course, the Win32 API needs to pass the system call a native path for the system call to work correctly. The NTDLL API `RtlDosPathNameToNtPathName` handles this conversion process. This API takes a DOS device path and returns the fully converted native path. The simplest conversion occurs when the caller has supplied a full drive path: for example, *C:\Windows*. In these cases, the conversion process merely prefixes

the path with the pre-defined path component *\??* to get the result *\??\C:\Windows*.

The *\??* Path, also called the *DOS device map prefix*, indicates that the object manager should use a two-step lookup process to find the drive letter. The object manager will first check a per-user DOS device map directory, in the path *\Sessions\0\DosDevices*<AUTHID>. Because the object manager checks a per-user location first, the user can create their own drive mappings. The AUTHID component is related to the authentication session of the caller's `Token`, which I'll described in Chapter 4, but for now, it's enough to know that its value is unique for each user. Note that the use of *0* for the console session ID is not a typo: all DOS device mappings are placed in a single location, regardless of which console session the user's logged in to.

If the drive letter is not found in the per-user location, the object manager will check a global directory, *\GLOBAL??*. If it's not found there, then the file lookup fails. The drive letter is an object manager symbolic link that points to the mounted volume device. We can see this in action by using the `Get-NtSymbolicLink` command to open the drive letters and display their properties (Listing 3-19).

```
PS> Use-NtObject($cdrive = Get-NtSymbolicLink "\??\C:") {
  $cdrive | Select-Object FullPath, Target
}
FullPath     Target
--------     ------
1 \GLOBAL??\C: \Device\HarddiskVolume3

2 PS> Add-DosDevice Z: C:\Windows

PS> Use-NtObject($zdrive = Get-NtSymbolicLink "\??\Z:") {
  $zdrive | Select-Object FullPath, Target
}
FullPath                                      Target
--------                                      ------
\Sessions\0\DosDevices\00000000-011b224b\Z:   \??\C:\windows

3 PS> Remove-DosDevice Z:
```

Listing 3-19 Displaying the symbolic links for the *C:* and *Z:* drives

We open the *C:* drive symbolic link and display its `FullPath` and `Target` properties. The full path is in the *\GLOBAL??* directory, and the target is the volume path 1. We then create a new *Z:* drive using the `Add-DosDevice` command, pointing the drive to the Windows directory 2. Note, that the *Z:* drive is accessible in any user application, not just in PowerShell. By displaying the *Z:* drive properties, we can see that it's in the per-user DOS device map, and that the target is the native path to the *Windows* directory. The

shows that the target of a drive letter doesn't have to point directly to a volume if it gets there eventually (in this case, after following the *C:* drive symbolic link). Finally, for completeness, we remove the *Z:* drive with `Remove-DosDevice` 3.

Path Types

Table 3-2 shows several different path types that the Win32 APIs supports, along with an example native path after conversion.

Table 3-2 Win32 Path Types

DOS Path	Native Path	Description
some\path	*\??\C:\ABC\some\path*	Relative path to current directory
C:\some\path	*\??\C:\some\path*	Absolute path
C:some\path	*\??\C:\ABC\some\path*	Drive relative path
\some\path	*\??\C:\some\path*	Rooted to current drive.
\\.\C:\some\..\path	*\??\C:\path*	Device path, canonicalized
\\?\C:\some\..\path	*\??\C:\some\..\path*	Device path, non-canonicalized
\??\C:\some\path	*\??\C:\some\path*	Device path, non-canonicalized
\\server\share\path	*\??\UNC\server\share\path*	UNC path to share on server

Due to the way DOS paths are specified, multiple DOS paths might represent the same native path. To ensure the final native path is correct, the DOS path must go through a *canonicalization* process to convert these different representations into the same canonical form.

One simple operation undertaken in canonicalization is the handling of path separators. For native paths, there is only one path separator, the backslash (`\`) character. If you use a forward slash (`/`), the object manager will treat it as just another filename character. However, DOS paths support both forward slashes and backslashes as path separators. The canonicalization process takes care of this by ensuring all forward slashes are converted to back slashes. Therefore, *C:\Windows* and *C:/Windows* are equivalent.

Another canonicalization operation is the resolving of parent directory references. When writing a DOS path, you might specify a filename with one dot (.) or two dots (..), each of which has a special meaning. A single dot refers to the current directory, and the canonicalization process will remove it from the path. A double dot refers to the parent, so the parent directory will be removed. Therefore, the path *C:\ABC\.\XYZ* will get converted to *C:\ABC\XYZ*, and *C:\ABC\..\XYZ* will get converted to *C:\XYZ*. As with the forward slash, the native APIs do not know about these special filenames, and will assume that they're the names of the file to look up.

NOTE Most other operating systems, such as Linux, handle this canonicalization process in the kernel. However, due to the subsystem model. Windows must do the path canonicalization in user mode, inside the subsystem-specific library. This is to support any differences in behavior in OS/2 or POSIX environments.

If the DOS path is prefixed with \\.\ or \?\, then the path is not canonicalized and is instead used verbatim, including any parent directory references or forward slashes. In some cases, the \?\ prefix can confuse the Win32 APIs with a current drive-rooted path, resulting in the opening of a path such as \?*C*:\?*Path*. It's unclear why Microsoft added this DOS path type, considering its potential for confusion.

You can manually convert a Win32 path to a native path using the `Get-NtFilePath` command. You can also check the path type using the `Get-NtFilePathType` command. Listing 3-20 shows some examples of using the `Get-NtFilePath` and `Get-NtFilePathType`:

```
PS> Set-Location $env:SystemRoot
PS C:\Windows> Get-NtFilePathType "."
Relative
PS C:\Windows> Get-NtFilePath "."
\??\C:\Windows
PS C:\Windows> Get-NtFilePath "..\"
\??\C:\

PS C:\Windows> Get-NtFilePathType "C:ABC"
DriveRelative
PS C:\Windows> Get-NtFilePath "C:ABC"
\??\C:\Windows\ABC

PS C:\Windows> Get-NtFilePathType "\\?\c:\abc/..\xyz"
LocalDevice
PS C:\Windows> Get-NtFilePath "\\?\c:\abc/..\xyz"
\??\c:\abc/..\xyz
```

Listing 3-20 Examples of Win32 file path conversion

If you're using the `Get-NtFile` or `New-NtFile` commands, you can use the `Win32Path` property to treat the path as a Win32 path and automatically convert it.

Maximum Path Lengths

The maximum filename supported by Windows is limited by the maximum number of characters in a `UNICODE_STRING` (namely, 32,767 characters). However, Win32 APIs have a stricter requirement. By default, passing a path longer than the value of `MAX_PATH`, defined as 260 characters,

will fail. This behavior is implemented inside the *NTDLL* API `RtlDosPathNameToNtPathName` when converting the path from Win32 to native format, as shown in Listing 3-21.

```
PS> $path = "C:\$('A'*256)"
PS> $path.Length
259
PS> Get-NtFilePath -Path $path
\??\C:\AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA...

PS> $path += "A"
PS> $path.Length
260
PS> Get-NtFilePath -Path $path
Get-NtFilePath : "(0xC0000106) - A specified name string is too long..."

PS> $path = "\\?\" + $path
PS> $path.Length
264
PS> Get-NtFilePath -Path $path
\??\C:\AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA...
```

Listing 3-21 Testing the `Win32 MAX_PATH` path limit

We call the `RtlDosPathNameToNtPathName` API via the `Get-NtFilePath` command. The first path we create is 259 characters long, which we can successfully convert to a native path. We then add one more character to the path, making the path 260 characters long; now, it fails with the error `STATUS_NAME_TOO_LONG`. If `MAX_PATH` is 260, shouldn't 259-character-long path succeed? Unfortunately, no: the APIs include the NUL-terminating character as part of the path's length, so the maximum path length is really only 259 characters.

Listing 3-21 also shows a way of bypassing this limitation. By adding the device prefix `\\?\` to the path, the conversion succeeds even though the length of the path is now 264 characters. This is because the prefix is replaced with the DOS device prefix `\??\`, and the remaining path is left verbatim. While this technique works, note that it also disables useful features, such as path canonicalization. In current versions of Windows, there is a way of opting into long filenames, shown in Listing 3-22.

```
PS> $path = "HKLM\SYSTEM\CurrentControlSet\Control\FileSystem"
PS> Get-NtKeyValue -Win32Path $path -Name "LongPathsEnabled"
Name             Type  DataObject
----             ----  ----------
LongPathsEnabled Dword 1

PS> (Get-Process -Id $pid).Path | Get-ExecutableManifest |
Select-Object LongPathAware
LongPathAware
```

```
-------------
        1 True

2 PS> $path = "C:\$('A'*300)"
  PS> $path.Length
  303
  PS> Get-NtFilePath -Path $path
  \??\C:\AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA...
```

Listing 3-22 Checking and testing long, path-aware applications

The first thing we do in Listing 3-22 is verify that the `LongPathsEnabled` registry value is set to the value `1`. The value must be set to `1` before the process starts, as it will be read only once during process initialization. However, just enabling the long path feature isn't sufficient, the process's executable file must opt-in by specifying a manifest property. We can query this property using the `Get-ExecutableManifest` command and selecting `LongPathAware`. Fortunately, PowerShell has this manifest option enabled 1. We can now convert much larger paths successfully, as shown with a 303-character path 2.

Are long paths a security issue? It's common for security issues to be introduced in places where there is an interface boundary. In this case, the fact that a filesystem can support exceptionally long paths could lead to the incorrect assumption that a filepath can never be longer than 260 characters. A possible issue might occur when an application queries the full path to a file and then copies that path into a memory buffer with a fixed size of 260 characters. If the length of the filepath is not first checked, this operation could result in the corruption of memory after the buffer, which might allow an attacker to gain control of the application's execution.

Process Creation

Processes are the main way to execute user-mode components and isolate them for security purposes, so it's important that we explore how to create them in detail. In the previous chapter, I mentioned that you can create a process using the `NtCreateUserProcess` system call. However, most processes won't be created directly using this system call; rather, they'll be created with the Win32 `CreateProcess` API.

The system call isn't often used directly, because `CreateProcess` needs to interact with other user-mode components, especially CSRSS, to correctly initialize the process for use on the user's desktop. We won't discuss process and thread creation in great detail in this book. In this section, we'll

give a quick overview of the `New-Win32Process` PowerShell command, which calls `CreateProcess` under the hood.

Command Line Parsing

The simplest way to create a new process is to specify a command line string representing the executable to run. The `CreateProcess` API will then parse the command line to find the executable file to pass to the kernel.

To test this command line parsing, let's create a new process using the `New-Win32Process` PowerShell command, which executes `CreateProcess` under the hood. We could use a built-in command such as `Start-Process` to do this, but `New-Win32Process` is useful because it exposes the full set of the `CreateProcess` APIs functionality. You can start a process using the following command:

```
PS> $proc = New-Win32Process -CommandLine "notepad test.txt"
```

We provide a command line containing the name of the executable to run, Notepad, and the name of a file to open, *test.txt*. This string doesn't necessarily need to provide a full path to the executable, The `New-Win32Process` command will parse the command line to try and distinguish the name of the initial executable image file from the file to open. That's not as simple of a process as it sounds.

The first thing `New-Win32Process` will do is parse the command line using an algorithm that splits on whitespace, unless that whitespace is enclosed in double quotes. In this case, it will parse the command line into two strings, `notepad` and `test.txt`. The command then takes the first string and tries to find a matching process; however, there's a slight complication: there is no *notepad* executable file, only *notepad.exe*. Though it's not required, Windows executables commonly have an *.exe* extension, so the search algorithm will automatically append this extension if one doesn't already exist.

The command will then search the following locations for the executable, much like the DLL path searching we previously discussed. Note that the executable search path is the same as the unsafe DLL search path:

1. The same directory as the current process's executable file
2. The current working directory
3. The Windows *SYSTEM32* directory
4. The *Windows* directory
5. Each semicolon-separated location in the `PATH` environment variable

If `New-Win32Process` can't find *notepad.exe*, it will next try to find the file *notepad test.txt*, in case that's what you meant. As the filename has an extension already, it won't replace it with *.exe*. If `New-Win32Process` can't find the file, it returns an error. Note that if we passed `notepad` surrounded by double quotes, as in `"notepad" test.txt`, then `New-Win32Process` would search for *notepad.exe* only and never fall back to trying all combinations of the name with the whitespace.

This command line parsing behavior has two security implications. Firstly, if the process is being created by a more privileged process and a less privileged user can write a file to a location earlier in the path search list, then the process could be hijacked. For example, if the privileged process's current directory is set to a directory the user can write to, then any process it creates that is not an executable directly would be searched first.

The second security implication is that the path-searching algorithm changes if the first value contains a path separator. In this case, instead of using the path-searching rules, `New-Win32Process` splits the path by whitespace and then tries each component as if it's a path, searching for the name either with the *.exe* extension or without. Let's look at an example. If we specify a command line of `C:\Program Files\abc.exe`, then the following paths would be searched for the executable file:

- *C:\Program*
- *C:\Program.exe*
- *C:\Program Files\abc.exe*
- *C:\Program Files\abc.exe.exe*

If the user could write the file *C:\Program* or *C:\Program.exe*, then they could hijack execution. Fortunately, on a default installation of Windows, a normal user can't write files to the root of system drive; however, configuration changes sometimes allow this writing. Also, the executable path might be on a different drive that does allow writing to the root.

To avoid both security implications, the caller can specify the executable's full path name by setting the `ApplicationName` property when calling `New-Win32Process`:

```
PS> $proc = New-Win32Process -CommandLine "notepad test.txt"
-ApplicationName "c:\windows\notepad.exe"
```

If we specify the path, the command will pass it verbatim to the new process.

Shell APIs

If you double-click a non-executable filetype, such as a text document, in Explorer, it will helpfully start an editor for you. However, if you try to run a document with `New-Win32Process`, you'll get the error shown below:

```
PS> New-Win32Process -CommandLine "document.txt"
Exception calling "CreateProcess": "%1 is not a valid Win32 application"
```

This error indicates that the text file is not a valid Win32 application.

The reason Explorer can start the editor is that it doesn't use the underlying `CreateProcess` API directly; instead, it uses a shell API. The main shell API used to start the editor for a file is `ShellExecuteEx`, implemented in the *SHELL32* library. This API, and its simpler sibling `ShellExecute`, are much too complex to cover into detail here. Instead, we'll give just a brief overview of them.

For our purposes, we need to specify three parameters to `ShellExecuteEx`:

- The path to the file to execute
- The verb to use on the file
- Any additional arguments

The first thing `ShellExecute` does is look up the handler for the extension of the file to execute. For example, if the file is *test.txt*, then it needs to lookup the handler for the *.txt* extension. The handlers are registered in the registry under the `HKEY_CLASSES_ROOT` key, which, as we saw in Chapter 2, is a merged view of parts of the machine software and the user's registry hive. In Listing 3-23, we query the handler:

```
PS> $base_key = "NtKey:\MACHINE\SOFTWARE\Classes"
1 PS> Get-Item "$base_key\.txt" | Select-Object -ExpandProperty Values
Name           Type   DataObject
----           ----   ----------
Content Type   String text/plain
PerceivedType  String text
             2 String txtfile

3 PS> Get-ChildItem "$base_key\txtfile\Shell" | Format-Table
Name    TypeName
----    --------
open    Key
print   Key
printto Key

4 PS> Get-Item "$base_key\txtfile\Shell\open\Command" |
```

```
Select-Object -ExpandProperty Values | Format-Table
Name Type         DataObject
---- ----         ----------
   5 ExpandString %SystemRoot%\system32\NOTEPAD.EXE %1
```

Listing 3-23 Querying the shell handler for *.txt* files

We start by querying the machine class's key for the *.txt* extension 1. Although we could have checked for a user-specific key, checking the machine class's key ensures that we inspect the system default. The *.txt* registry key doesn't directly contain the handler. Instead, the default value, represented by an empty name, refers to another key, in this case the `txtfile` 2. We then list the subkeys of `txtfile` and find three keys: *open*, *print*, and *printto* 3. We can pass these verbs by name to `ShellExecute`.

Each of these verb keys can have a subkey, called *Command*, that contains a command line to execute 4. We can see that the default for a *.txt* file is to open Notepad 5; the `%1` is replaced with the path to the file being executed. (The command could also contain `%*`, which includes any additional arguments passed to `ShellExecute`.) The `CreateProcess` API can now start the executable and handle the file.

There are many different standard verbs you can pass to `ShellExecute`. Table 3-3 shows a list of common ones you'll encounter.

Table 3-3 Common Shell Verbs and Descriptions

Verb	Description
`Open`	Open the file; this is typically the default.
`Edit`	Edit the file.
`Print`	Print the file.
`Printto`	Print to a specified printer.
`Explore`	Explore a directory; this is used to open a directory in an Explorer window.
`Runas`	Open the file as an Administrator. Typically, defined for executables only.
`runasuser`	Open the file as another user. Typically, defined for executables only.

You might find it odd that there is both an `open` and an `edit` verb. If you opened a *.txt* file, for example, the file would open in Notepad, and you'd be able to edit it. But the distinction is useful for files such as batch files, where the `open` verb would execute the file and `edit` would open it in a text editor.

To use `ShellExecute` from PowerShell, you can run the `Start-Process` command. By default, `ShellExecute` will use the *open* verb, but you can specify your own verb using the `Verb` parameter. Below, we print a *.txt* file as an administrator using the `print` verb:

```
PS> Start-Process "test.txt" -Verb "print"
```

Verb configurations can also to improve security. For example, PowerShell scripts with a *.ps1* extension have the `open` verb registered. However, clicking a script will open the script file in Notepad rather than executing the script. Therefore, if you double-click the script file in Explorer, it won't execute. Instead, you must right-click the file and explicitly choose *Run with PowerShell*.

The full details of the shell APIs are out of scope for this book; as you might expect, the full picture is not quite as simple as I've shown here.

System Processes

Throughout this chapter, I've alluded to various processes, such as LSASS, that run with high privileges. This is because, even when no user is logged in to the operating system, the system still needs to perform tasks like waiting for authentication, managing hardware, and communicating over the network.

The kernel could perform some of these tasks. However, writing kernel code is more difficult for a number of reasons: the kernel doesn't have as wide a range of APIs available; it's resource constrained, especially in terms of memory; and any coding mistake could result in the system crashing or being exposed to a security vulnerability.

To avoid these challenges, Windows runs a variety of processes outside of kernel mode, with a high-privilege level, to provide important facilities. We'll go through some of these special processes in this section.

The Session Manager

The *session manager (SMSS)* is the first user-mode process started by the kernel after boot. It's responsible for setting up the working environment for subsequent processes. Some of its responsibilities include:

- Loading known DLLs and creating the section objects
- Starting subsystem processes such as CSRSS
- Initializing base DOS devices such as serial ports
- Running automatic disk-integrity checks

The Windows Logon Process

The *windows logon* process is responsible for setting up a new console session, as well as displaying the Logon user interface (primarily through the *LogonUI* application). It's also responsible for starting the *user-mode font driver (UMFD)* process, which renders fonts to the screen, and starting the *desktop window manager (DWM)* process, which performs desktop compositing operations to allow for fancy, transparent windows and modern GUI touches.

The Local Security Authority Subsystem

We've already mentioned LSASS in the context of the SRM. However, it's worth stressing its important role in authentication. Without LSASS, a user would not be able to log on to the system. We'll cover LSASS's roles and responsibilities in much more detail in Chapter 10.

The Service Control Manager

The *service control manager (SCM)* is responsible for starting most privileged system processes on Windows. It manages these processes, referred to as *services*, and can start and stop them as needed. For example, the SCM could start a service based on certain conditions, such as a network becoming available.

Each service is a securable resource with fine-grained controls determining which users can manipulate its state. By default, only an administrator can manipulate a service. I'll note some of the most important services running on any Windows system:

Remote Procedure Call Subsystem (RPCSS)
The RPCSS service manages the registration of remote procedure call endpoints, exposing the registration to local clients as well as over the network. This service is essential to a running system; in fact, if this process crashes, it will force Windows to reboot.

DCOM Server Process Launcher
The DCOM Server Process Launcher is a counterpart to RPCSS (and used to be part of the same service). It's used to start Component Object Model (COM) server processes on behalf of local or remote client.

Task Scheduler
Being able to schedule an action to run at a specific time and date is a useful feature of an operating system. For example, perhaps you want to ensure that

you delete unused files on a specific schedule. You could set up an action with the task scheduler service to run a cleanup tool on that schedule.

Windows Installer
This service can be used to install new programs and features. By running as a privileged service, it permits installation and modification in normally protected locations on the filesystem.

Windows Update
Having a fully up-to-date operating system is crucial to the security of your Windows system. When Microsoft releases new security fixes, they should be installed as soon as possible. To avoid requiring the user to check for updates, this service runs in the background, waking up periodically to check the internet for new patches.

Application Information
This service provides a mechanism for a user to switch between an administrator and non-administrator user on the same desktop. This feature is usually referred to as *user account control (UAC)*. You can start an administrator process by using the `runas` verb with the shell APIs. We'll cover how UAC works under the hood in the next chapter.

We can query the status of all services controlled by the SCM using various tools. PowerShell has the built-in `Get-Service` command; however, the modules used in this book provide a more comprehensive command, `Get-Win32Service`, which can inspect the configured security of a service as well as additional properties not exposed using the default command. Listing 3-24 shows how to query for all current services.

```
PS> Get-Win32Service
Name                Status   ProcessId
----                ------   ---------
AarSvc              Stopped  0
AESMService         Running  7440
AJRouter            Stopped  0
ALG                 Stopped  0
AppIDSvc            Stopped  0
Appinfo             Running  8460
--snip--
```

Listing 3-24 Displaying all services using `Get-Win32Service`

The output shows the name of the service, its status, either `Stopped` or `Running`, and if it's running, the process ID of the service process. If you list the service's properties using `Format-List`, you'll also be able to see additional information, such as a full description of the service.

Worked Examples

Let's walk through some worked examples to practice using the various commands covered in this chapter for security research or systems analysis.

Finding Executables That Import Specific APIs

We saw earlier how to use the `Get-Win32ModuleImport` command to extract an executable file's imported APIs. One use for this command that I find especially helpful when I'm trying to track down security issues is identifying all the executables that use a particular API, such as `CreateProcess`, and then using this list to reduce the files I need to reverse engineer. You can perform such a search with the basic PowerShell script shown in Listing 3-25.

```
PS> $imps = ls "$env:WinDir\*.exe" | ForEach-Object {
     Get-Win32ModuleImport -Path $_.FullName
}
PS> $imps | Where-Object Names -Contains "CreateProcessW" |
Select-Object ModulePath
ModulePath
----------
C:\WINDOWS\explorer.exe
C:\WINDOWS\unins000.exe
```

Listing 3-25 Finding executables that import `CreateProcess`

We start by enumerating all *.exe* files in the Windows directory. For every executable file, we call the `Get-Win32ModuleImport` command. This will load the module and parse its imports. The process of loading each executable and parsing its imports can be time consuming, so it's best to capture the results into a variable, as we do here.

Next, we select out only the imports that contain the `CreateProcessW` API. The `Names` property is a list containing the imported names for a single DLL. To get the resulting list of executable files that import a specific API, we can select the `ModulePath` property, which contains the original loaded path name.

You can use the same technique to enumerate DLL files or drivers and quickly discover targets for reverse engineering.

Finding Hidden Registry Keys or Values

Earlier in this chapter, we mentioned that one of the big advantages of using the native system calls over the Win32 APIs to interact with the registry

is that you can access keys and values with NUL characters in the name. It would be useful to be able to find these keys and values so you can try to detect software on your system that is actively trying to hide registry keys or values from the user. Some malware families, such as Kovter and Poweliks, use this technique of hiding keys. Let's start by finding keys with NUL characters in the name (Listing 3-27).

```
PS> $key = New-NtKey -Win32Path "HKCU\SOFTWARE\`0HIDDENKEY"
PS> ls NtKeyUser:\SOFTWARE -Recurse | Where-Object Name -Match "`0"
Name                TypeName
----                --------
SOFTWARE\ HIDDENKEY Key

PS> Remove-NtKey $key
PS> $key.Close()
```

Listing 3-26 Finding hidden registry keys

We first create a key in the current user's hive with a NUL character in it. If you try to find this key using the built-in registry provider, it will fail. We do a recursive listing of the current user's hive and select any keys that have a NUL character in the name. In the output, you can see that the hidden key was discovered.

To find hidden values, you can query the list of values on a key by enumerating its `Values` property. Each value contains the name of the key and the data value (Listing 3-27).

```
1 PS> $key = New-NtKey -Win32Path "HKCU\SOFTWARE\ABC"
  PS> Set-NtKeyValue -Key $key -Name "`0HIDDEN" -String "HELLO"


2 PS> function Select-HiddenValue {
      [CmdletBinding()]
      param(
          [parameter(ValueFromPipeline)]
          $Key
      )

      Process {
        3 foreach($val in $Key.Values) {
            if ($val.Name -match "`0") {
                [PSCustomObject]@{
                    RelativePath = $Key.RelativePath
                    Name = $val.Name
                    Value = $val.DataObject
                }
            }
        }
      }
  }
```

```
4 PS> ls -Recurse NtKeyUser:\SOFTWARE | Select-HiddenValue | Format-Table
  RelativePath Name    Value
  ------------ ----    -----
  SOFTWARE\ABC  HIDDEN HELLO

  PS> Remove-NtKey $key
  PS> $key.Close()
```

Listing 3-27 Finding hidden registry values

We start by creating a normal key, then adding a value with a NUL character in the name 1. We then define a function, `Select-HiddenValue` 2, that will check keys in the pipeline and select any value with a NUL character in the name, returning a custom object to the pipeline 3.

We then recursively enumerate the current user's hive and filter the keys through the `Select-HiddenValue` function 4. You can see in the output that we discovered the hidden value.

Wrapping Up

This chapter provided a quick tour through the Windows user-mode components. We started with a dive into Win32 APIs and the loading of DLLs. Understanding this topic is important, as it reveals how user-mode applications communicate with the kernel and implement common features.

Next, I provided an overview of Win32 GUI, including a description of the separate system call table used for WIN32K, which is the kernel-mode component of the WIN32 subsystem. I introduced the window station and desktop object types and outlined the purpose of the console session, as well as how it corresponds to the desktop you see as a user.

I then returned to the topic of Win32 APIs by detailing the differences and similarities between a Win32 API, in this case `CreateMutexEx`, and the underlying system call, `NtCreateMutant`. This discussion should have given you a better understanding of how the Win32 APIs interact with the rest of the operating system. I also introduced the differences between DOS device paths and native paths as understood by a system call, a topic that is important for understanding how user-mode applications interact with the filesystem.

I concluded with a description of several topics related to Win32 processes and threads: the APIs used to create processes directly or through the shell, as well as an overview of well-known system processes. In later chapters, we'll

revisit many of these topics in more depth. In the next three chapters, we'll focus on how Windows implements security through the SRM.

4

SECURITY ACCESS TOKENS

The *security access token*, or *token* for short, is at the heart of Windows security. The SRM uses tokens to represent identities, such as user accounts, and then grant or deny them access to resources. Windows represents tokens with `Token` kernel objects, which contain, at a minimum, the specific identity they represent, any security groups the identity belongs to, and the special privileges the identity has been granted.

Like other kernel objects, tokens support `Query` and `Set` information system calls, which allow the user to inspect the properties of a token and set certain properties. Though less commonly used, some Win32 APIs also expose these `Set` and

`Query` system calls: for example, `GetTokenInformation` and `SetTokenInformation`.

Let's start with an overview of the two main types of tokens you'll encounter when analyzing a Windows system's security. We'll then detail many of the important properties a token contains. You'll need to understand these before we can discuss access checking in Chapter 7.

Primary Tokens

Every process has an assigned token that describes its identity for any resource-access operation. When the SRM performs an access check, it will query the process's token and use it to determine what kind of access to grant. When a token is used for a process, it's called a *primary token*.

You can open a process's token using the `NtOpenProcessToken` system call, which will return a handle that you can use to query token information. Because the `Token` object is a securable resource, the caller needs to pass an access check to get the handle. Note you also need a handle to the process with `QueryLimitedInformation` access to be able to query the token.

When opening a `Token` object, you can request the following access rights:

`AssignPrimary`
Assigns the `Token` object as a primary token

`Duplicate`
Duplicates the `Token` object

`Impersonate`
Impersonates the `Token` object

`Query`
Queries the properties of the `Token` object, such as its groups and privileges

`QuerySource`
Queries the source of the `Token` object

`AdjustPrivileges`
Adjusts a `Token` object's privilege list

`AdjustGroups`
Adjust a `Token` object's group list

`AdjustDefault`
Adjusts properties of a `Token` object not covered by the other access rights

`AdjustSessionId`
Adjusts the `Token` object's session ID

We can see a list of accessible processes and their tokens by running the PowerShell command `Show-NtToken -All`. This should open the Token Viewer application, as in Figure 4-1.

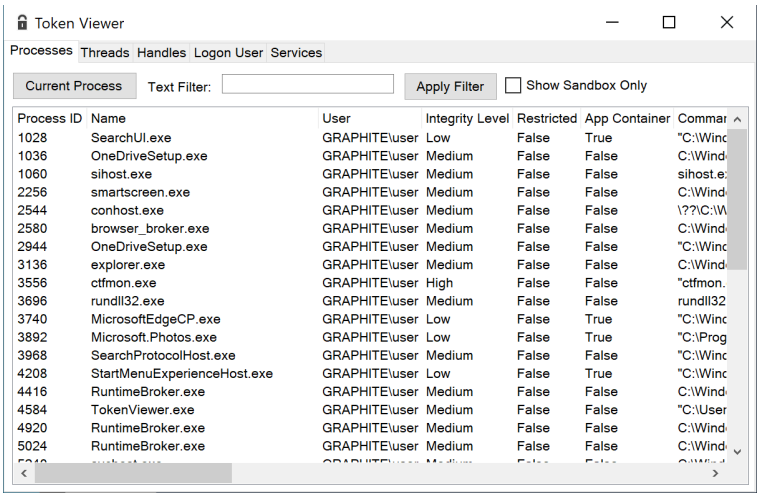

Figure 4-1 The Token Viewer, which lists all accessible processes and their tokens

The list view provides only a simple overview of the available tokens. If you want to see more information, double-click one of the process entries to bring up a detailed view of the token, as shown in Figure 4-2.

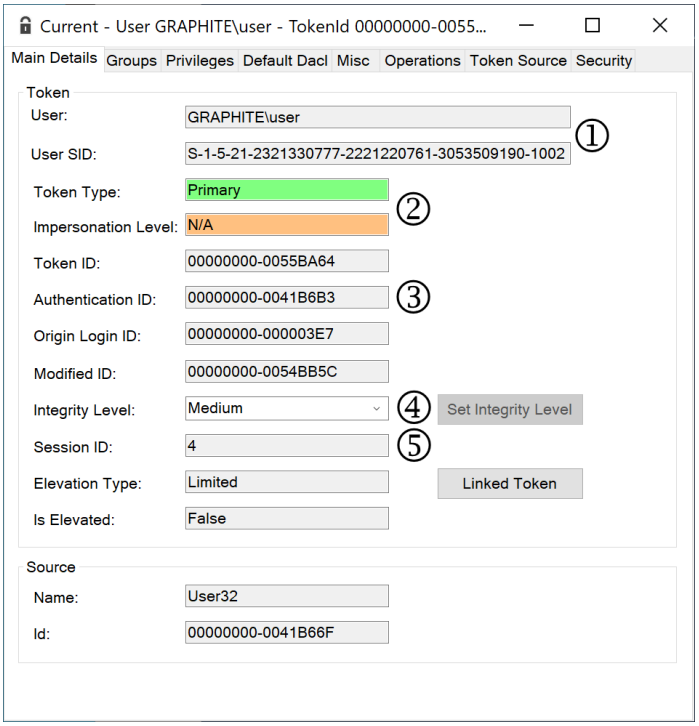

Figure 4-2 The detailed view for a process's `Token` object

Let's highlight a few important pieces of information in the token view shown in Figure 4-2. At the top 1 is the user's name and SID. The `Token` object stores only the SID. However, the view will display the name if it's available.

Next is an indication of the type of token 2. As we're inspecting a primary token, the `Token Type` is set to `Primary`. The impersonation level is used only for impersonation tokens, which we'll discuss in the next section. It's not needed for primary tokens, so it's set to N/A.

In the middle of the dialog is a list of four 64-bit integer identifiers 3:

Token Identifier
A unique value assigned when the `Token` object was created.

Authentication Identifier

A value that indicates the logon session the token belongs to.

Origin Login Identifier
The authentication identifier of the parent logon session.

Modified Identifier
A unique value that is updated when certain token values are modified.

The LSASS creates a *logon session* when a user authenticates to a Windows machine. The logon session tracks authentication-related resources for a user; for example, it stores a copy of the user's credentials so that they can be reused. During the logon session creation process, the SRM generates a unique authentication identifier value used to reference the correct logon session. Therefore, for a given logon session, all user tokens will have the same authentication identifier. If a user authenticates twice to the same machine, the SRM will generate different authentication identifiers.

The *origin login identifier* indicates who created the token's logon session. If you authenticate a different user on your desktop (by calling the `LogonUser` API with a username and password, for example), then the origin login identifier will serve as the calling token's authentication identifier. Notice that this field in Figure 4-2 shows the value 00000000-000003E7, one of four fixed authentication identifiers defined by the SRM. In this case, it indicates the *SYSTEM* logon session. Table 4-1 shows these fixed values, along with the SID for the user account associated with the session:

Table 4-1 Authentication Identifiers and User SIDs for Fixed Logon Sessions

Authentication identifier	User SID	Logon session username
00000000-000003E4	S-1-5-20	NT AUTHORITY\NETWORK SERVICE
00000000-000003E5	S-1-5-19	NT AUTHORITY\LOCAL SERVICE

00000000-000003E6	S-1-5-7	NT AUTHORITY\ANONYMOUS LOGON
00000000-000003E7	S-1-5-18	NT AUTHORITY\SYSTEM

After the identifiers in Figure 4-2 is the *integrity level* of the token 4. Windows Vista first added the integrity level to implement a simple *mandatory access control* mechanism, whereby systemwide policies enforce access to resources, rather than allowing an individual resource to specify its access. We'll discuss integrity levels later in this chapter in "Token Groups" on page XX.

The final highlighted value in Figure 4-2 is the session ID 5, a number assigned to the console session the process is attached to. Even though the console session is a property of the process, the value is specified in the process's token.

LOCALLY UNIQUE IDENTIFIERS (LUIDS)

I mentioned that a token's identifiers are 64-bit integers. Technically, they're really *Locally Unique Identifier (LUID)* structures. LUID structures contain two 32-bit values put together, turning them into one 64-bit integer. LUIDs are a common system type, and the SRM uses them often when it needs a unique value. For example, they're used to uniquely identify privilege values.

You can allocate your own LUID by calling the `NtAllocateLocallyUniqueId` system call or the `Get-NtLocallyUniqueId` PowerShell command. When you use a system call, Windows ensures it has a central authority for generating the next unique ID; reusing a value might be catastrophic. For instance, if an LUID is reused for a token's authentication ID, it might overlap with one of the system's, defined in Table 4-1. The reuse could trick the system into thinking a more privileged user is accessing a resource, resulting in privilege escalation.

The Token Viewer GUI is great if you want to manually inspect a token's information. For programmatic access, we can open a `Token` object in PowerShell using the `Get-NtToken` command. Use the following to get the current process's token:

```
PS> $token = Get-NtToken
```

If you want to open the token for a specific process, you can use the following command, replacing *<PID>* with the process ID of the target process:

```
PS> $token = Get-NtToken -ProcessId <PID>
```

The result of the `Get-NtToken` command is a `Token` object whose properties you can query. For example, you can display the token's user, as shown in Listing 4-1.

```
PS> $token.User
Name                           Attributes
----                           ----------
GRAPHITE\user                  None
```

Listing 4-1 Displaying the user via a `Token` object's properties

Use the `Format-NtToken` command to output basic information to the console, as shown in Listing 4-2.

```
PS> Format-NtToken $token -All
USER INFORMATION
----------------
Name                           Attributes
----                           ----------
GRAPHITE\user                  None

GROUP SID INFORMATION
-----------------
Name                           Attributes
----                           ----------
GRAPHITE\None                  Mandatory, EnabledByDefault
Everyone                       Mandatory, EnabledByDefault
--snip--
```

Listing 4-2 Displaying properties of a token using `Format-NtToken`

You can pass the opened `Token` object to `Show-NtToken` to display the same GUI shown in Figure 4-2.

Impersonation Tokens

The other type of token you'll encounter is the *impersonation token*. Impersonation tokens are most important for system services, as they allow a process with one identity to temporarily impersonate another identity for the purposes of an access check. For example, a service might need to open a file belonging to another user while performing some operation. By allowing that service to impersonate the calling user, the system grants the service access to the file, even if the service couldn't open the file directly.

Impersonation tokens are assigned to threads, not processes. This means that only the code running in that thread will take on the impersonated identity. There are three ways an impersonation token can be assigned to a thread:

- By explicitly granting a `Token` object `Impersonate` access and a `Thread` object `SetThreadToken` access
- By explicitly granting a `Thread` object `DirectImpersonation` access
- Implicitly, by impersonating a remote procedure call (RPC) request

You're most likely to encounter implicit token assignment, as it's the most common case for system services, which expose RPC mechanisms. For example, if a service creates a named pipe server, it can impersonate clients that connect to the pipe using the `ImpersonateNamedPipe` API. When a call is made on the named pipe, the kernel captures an *impersonation context* based on the calling thread and process. This impersonation context is used to assign an impersonation token to the thread that calls `ImpersonateNamedPipe`. The impersonation context can be based on either an existing impersonation token on the thread or a copy of the process's primary token.

Security Quality of Service

What if you don't want to give the service the ability to impersonate your identity? The SRM supports a feature called *Security Quality of Service (SQoS)* that enables you to control this. When you open a named pipe using the filesystem APIs, you can pass a `SECURITY_QUALITY_OF_SERVICE` structure in the `SecurityQualityOfService` field of the `OBJECT_ATTRIBUTES` structure. The SqoS structure contains three configuration values: the impersonation level, the context tracking mode, and the effective token mode.

The *impersonation level* in the SqoS is the most important field for controlling what the server can do with your identity. It defines the level of access granted to the service when it

implicitly impersonates the caller. The level can be one of four values, in ascending order of privilege:

Anonymous is the lowest level. It prevents the service opening the `Token` object and querying the user's identity. Only a limited set of services would function if the caller specified this level.

Identification allows the service to open the `Token` object and query the user's identity, groups, and privileges. However, the thread cannot open any secured resources while impersonating the user.

Impersonation allows the service to fully exercise the user's identity on the local system. The service can open local resources secured by the user and manipulate them. It can also access remote resources for the user if the user has locally authenticated to the system. However, if the user authenticated over a network connection, such as SMB, then the service can't use the `Token` object to access remote resources.

Delegation is the highest level, enabling the service to open all local and remote resources as if they were the user. To access a remote resource from network-authenticated users, however, it's not enough to have this impersonation level. The Windows domain must also be configured to allow it. We'll discuss this impersonation level more in Chapter 14 on Kerberos authentication.

You can specify the impersonation level in the SqoS either when calling a service or when creating a copy of an existing token. To restrict what a service can do, specify the Identification or Anonymous levels. As a result, the service won't be able to access any resources, although at Identification level, the server will still be able to access the token and perform operations on the caller's behalf.

Let's run a test using the `Invoke-NtToken` PowerShell command. In Listing 4-3, we impersonate a token at a specified level and execute a script that opens a secured resource. We

specify the impersonation level using the `ImpersonationLevel` property.

```
PS> $token = Get-NtToken
PS> Invoke-NtToken $token {
   Get-NtDirectory -Path "\"
} -ImpersonationLevel Impersonation
Name                           NtTypeName
----                           ----------
                               Directory
PS> Invoke-NtToken $token {
   Get-NtDirectory -Path "\"
} -ImpersonationLevel Identification
Get-NtDirectory : (0xC00000A5) - A specified impersonation level is invalid.
--snip--
```

Listing 4-3 Impersonating a token at different levels and opening a secured resource

The first command we execute gets a handle to the current process's primary token. We then call `Invoke-NtToken` to impersonate the token at the Impersonation level and run a script that calls `Get-NtDirectory` to open the root OMNS directory. The open operation succeeds, and we print the directory object to the console.

We repeat the operation, now at the Identification level, and receive the error `0xC00000A5`, or `STATUS_BAD_IMPERSONATION_LEVEL`. Note that the open operation doesn't return an "access denied" error because the SRM doesn't get far enough to check whether the impersonated user can access the resource. Now you'll know the reason for this error if you see it when developing an application or using the system.

ANONYMOUS USERS

The Anonymous impersonation level is not the same as the anonymous logon user referenced in Table 4-1. It's possible to run with an anonymous user identity and be granted access to a resource by an access check, whereas an Anonymous-level token cannot pass any access check, regardless of how the resource's security is configured.

The kernel implements the `NtImpersonateAnonymousToken` system call, which will impersonate the anonymous user on a specified thread. You can also access the anonymous user token using `Get-NtToken`:

```
PS> Get-NtToken -Anonymous | Format-NtToken
NT AUTHORITY\ANONYMOUS LOGON
```

The other two fields in the SqoS are used less frequently, but they're still important. The *context tracking mode* determines whether to statically capture the user's identity when a connection is made to the service. If the identity is not statically captured, then if the caller impersonates another user before calling the service, the new impersonated identity will become available to the service, not to the process identity. Note that the impersonated identity can be passed to the service only if it's at the Impersonation or Delegation levels. If the impersonated token is at the Identification or Anonymous levels, the SRM generates a security error and rejects the impersonation operation.

Effective token mode changes the token passed to the server in a different way. It's possible to disable groups and privileges before making a call, and if effective token mode is disabled, the server can re-enable those groups and privileges and use them. However, if effective token mode is enabled, the SRM will strip out the groups and privileges so that the server can't re-enable them or use them.

By default, if no SqoS structure is specified when opening the IPC channel, the caller's level is Impersonation with static tracking and a non-effective token. If an impersonation context is captured and the caller is already impersonating, then the impersonation level of the thread token must be greater or equal to the Impersonation level; otherwise, the capture will fail. This is enforced even if the SqoS requests the Identification level. This is an important security feature; it prevents a caller at Identification level and below from calling over an RPC channel and pretending to be another user.

NOTE I've described how SqoS is specified at the native system call level, as the `SECURITY_QUALITY_OF_SERVICE` structure is not exposed through the Win32 APIs directly. Instead, it's usually specified using additional flags; for example, `CreateFile` exposes SqoS by specifying the `SECURITY_SQOS_PRESENT` flag.

Explicitly Impersonating a Token

There are two ways to impersonate a token explicitly. If you have an impersonation `Token` object handle with `Impersonate` access, you can assign it to a thread using the `NtSetInformationThread` system call and the `ThreadImpersonationToken` information class.

If, instead, you have a thread you want to impersonate with `DirectImpersonation` access, you can use the other mechanism. With the handle to a source thread, you can call the `NtImpersonateThread` system call and assign an impersonation token to another thread. Using `NtImpersonateThread` is a mix between explicit and implicit impersonation. The kernel will capture an impersonation context as if the source thread has called over a named pipe. You can even specify the SqoS structure to the system call.

You might be thinking that impersonation surely opens up a giant security backdoor. If I set up my own named pipe and convince a privileged process to connect to me, and the caller doesn't set SqoS to limit access, couldn't I gain elevated privileges? We'll come back to how this is prevented later in this chapter in "Token Assignment" on page XX.

Converting Between Token Types

You can convert between the two token types using duplication. When you duplicate a token, the kernel creates a new `Token` object and makes a deep copy of all the object's properties. While the token is duplicating, you can change its type.

This duplication operation differs from the handle duplication we discussed in Chapter 3, as duplicating a handle to a token would merely create a new handle pointing to the same `Token` object. To duplicate the actual `Token` object, you need to have `Duplicate` access rights on the handle.

You can then use either the `NtDuplicateToken` system call or the `Copy-NtToken` PowerShell command to duplicate the token. For example, to create an impersonation token at the Delegation level based on an existing token, use the script in Listing 4-4.

```
PS> $imp_token = Copy-NtToken -Token $token -ImpersonationLevel Delegation
PS> $imp_token.ImpersonationLevel
Delegation
PS> $imp_token.TokenType
Impersonation
```

Listing 4-4 Duplicating a token to create an impersonation token using `Copy-NtToken`

We can convert the impersonation token back to a primary token using `Copy-NtToken` again, as shown in Listing 4-5.

```
PS> $pri_token = Copy-NtToken -Token $imp_token -Primary
PS> $pri_token.TokenType
Primary
PS> $pri_token.ImpersonationLevel
Delegation
```

Listing 4-5 Converting an impersonation token to a primary token

Note something interesting in the output: the new primary token has the same impersonation level as the original token. This is because the SRM considers only the `TokenType` property; if the token is a primary token, the impersonation level is ignored.

Seeing as we can convert an impersonation token back to a primary token, you might be wondering: could we convert an Identification-level or Anonymous-level token back to a primary token, create a new process, and bypass the SqoS settings? Let's try it in Listing 4-6.

```
PS> $imp_token = Copy-NtToken -Token $token -ImpersonationLevel Identification
PS> $pri_token = Copy-NtToken -Token $imp_token -Primary
Exception: "(0xC00000A5) - A specified impersonation level is invalid.
```

Listing 4-6 Duplicating an Identification-level token back to a primary token

Listing 4-6 shows that we can't duplicate an Identification-level token back to a primary token, as the second line causes an exception. The operation would break a security guarantee of the

SRM (specifically, that the SQoS allows the caller to control how its identity is used).

A final note: if you're opening a token using `Get-NtToken`, you can perform the duplication operation in one step by specifying the `-Duplicate` command.

Pseudo Token Handles

To access a token, you must open a handle to the `Token` object, then remember to close the handle after use. Windows 10 introduced three *pseudo handles* that allow you to query token information without opening a full handle to a kernel object. Here are those three handles, with their handle values parentheses:

Primary (`-4`)
The primary token for the current process

Impersonation (`-5`)
The impersonation token for the current thread; fails if the thread is not impersonating

Effective (`-6`)
The impersonation token, if it is impersonating; otherwise, the primary token

Unlike the current process/thread pseudo handles, you can't duplicate these token handles; you can use them for certain limited uses only, such as querying information or performing access checks. The `Get-NtToken` command can return these handles if you specify the `Pseudo` parameter, as shown in Listing 4-7.

```
PS> Invoke-NtToken -Anonymous {Get-NtToken -Pseudo -Primary | Get-NtTokenSid}
Name                           Sid
----                           ---
GRAPHITE\user                  S-1-4-21-2318445812-3516008893-216915059-1002

PS> Invoke-NtToken -Anonymous {Get-NtToken -Pseudo -Impersonation | Get-
NtTokenSid}
Name                           Sid
----                           ---
```

```
2 NT AUTHORITY\ANONYMOUS LOGON S-1-4-7

  PS> Invoke-NtToken -Anonymous {Get-NtToken -Pseudo -Effective | Get-
  NtTokenSid}
  Name                           Sid
  ----                           ---
3 NT AUTHORITY\ANONYMOUS LOGON S-1-4-7

  PS> Invoke-NtToken -Anonymous {Get-NtToken -Pseudo -Effective} | Get-
  NtTokenSid
  Name                           Sid
  ----                           ---
4 GRAPHITE\user                  S-1-4-21-2318445812-3516008893-216915059-1002
```

Listing 4-7 Querying pseudo tokens

We've queried the three types of pseudo tokens while impersonating the anonymous user. The first command queries the primary token and extracts its user SID 1. The next command queries the impersonation token, which returns the anonymous user SID 2. We then query the effective token, and as we're impersonating the anonymous user, and return the anonymous user SID 3. Finally, we query the effective token again, but this time, wait until after the script block has executed to extract the user SID. This operation returns the primary token's user SID 4, demonstrating that the pseudo token is context sensitive.

Token Groups

If administrators had to secure every resource for each possible user, identity security would become too unwieldy to manage. *Groups* allow users to share a broader security identity. Most of the access control operations on Windows grant access to groups rather than individual users.

From the SRM's perspective, a group is just another SID that could potentially define access to a resource. We can display the groups in the PowerShell console using the `Get-NtTokenGroup` command, as shown in Listing 4-8.

```
PS> Get-NtTokenGroup $token
Name                                Attributes
----                                ----------
GRAPHITE\None                       Mandatory, EnabledByDefault, Enabled
```

```
Everyone                                Mandatory, EnabledByDefault, Enabled
BUILTIN\Users                           Mandatory, EnabledByDefault, Enabled
BUILTIN\Performance Log Users           Mandatory, EnabledByDefault, Enabled
NT AUTHORITY\INTERACTIVE                Mandatory, EnabledByDefault, Enabled
--snip--
```

Listing 4-8 Querying the current token's groups

You can also use `Get-NtTokenGroup` to filter for specific attribute flags by specifying the `Attributes` parameter. Table 4-2 shows the possible attribute flags you can pass the command.

Table 4-2 Group Attributes, in SDK and PowerShell Format

SDK attribute name	PowerShell attribute name
`SE_GROUP_ENABLED`	`Enabled`
`SE_GROUP_MANDATORY`	`Mandatory`
`SE_GROUP_ENABLED_BY_DEFAULT`	`EnabledByDefault`
`SE_GROUP_LOGON_ID`	`LogonId`
`SE_GROUP_OWNER`	`Owner`
`SE_GROUP_USE_FOR_DENY_ONLY`	`UseForDenyOnly`
`SE_GROUP_INTEGRITY`	`Integrity`
`SE_GROUP_INTEGRITY_ENABLED`	`IntegrityEnabled`
`SE_GROUP_RESOURCE`	`Resource`

The following sections describe what each of these flags means.

Enabled, EnabledByDefault, and Mandatory

The most important flag is `Enabled`. When it's set, the SRM considers the group during the access-check process; otherwise, it will ignore the group. Any group with the `EnabledByDefault` attribute set is automatically enabled.

It's possible to disable a group using the `NtAdjustGroupsToken` system call if you have `AdjustGroups` access on the token handle; the `Set-NtTokenGroup` PowerShell command exposes this system call. However, you can't disable groups that have the `Mandatory` flag set.

While all groups in a normal user's token have the `Mandatory` flag, certain system tokens have non-mandatory groups. If a group is disabled when you pass an impersonation

token over RPC and the effective token flag is set in SQoS, the impersonation token will delete the group.

LogonId

The `LogonId` flag identifies any SID that is granted to all tokens on the same desktop. For example, if you run a process as a different user using the `runas` utility, the new process's token will have the same logon SID as the caller, even though it's a different identity. This behavior allows the SRM to grant access to session-specific resources, such as the session object directory. The SID is always in the format *`S-1-4-4-X-Y`*, where X and Y are the two 32-bit values of a LUID that was allocated when the authentication session was created. We'll come back to the logon SID and where it applies in the next chapter.

Owner

All securable resources on the system belong to either a group SID or a user SID. Tokens have an `Owner` property that contains a SID to use as the default owner when creating a resource. The SRM allows only a specific set of the users' SIDs to be specified in the `Owner` property: either the user's SID, or any group SID that is marked with the `Owner` flag.

You can get or set the token's current `Owner` property using the `Get-NtTokenSid` command. For example, Listing 4-9 gets the owner SID from the current token, then tries to set the owner.

```
PS> Get-NtTokenSid $token -Owner
Name          Sid
----          ---
GRAPHITE\user S-1-4-21-818064984-378290696-2985406761-1002

PS> Set-NtTokenSid -Owner -Sid "S-1-2-3-4"
Exception setting "Owner": "(0xC000005A) - Indicates a particular Security ID
may not be assigned as the owner of an object.
```

Listing 4-9 Getting and setting the token's owner SID

In the second command we run, we try to set the `Owner` property to the SID `S-1-2-3-4`. As this SID isn't our current user SID or in our list of groups, it fails with an exception.

UseForDenyOnly

The SRM's access check either allows or denies access to a SID. But when a SID is disabled, it will no longer participate in allow or deny checks, which can result in incorrect access checking.

Let's give a simple example. Imagine there are two groups, *Employee* and *Remote Access*. A user creates a document that they want all employees to read except for those remotely accessing the system, as the content of the document is sensitive, and the user doesn't want it to leak. The document is configured to grant all members of the *Employee* group access but to deny access to users in the *Remote Access* group.

Now imagine that a user belonging to both those groups could disable a group when accessing a resource; they could simply disable *Remote Access* to be granted access to the document based on the *Employee* group, trivially circumventing the access restrictions.

For this reason, a user will rarely be allowed to disable groups. However, in certain cases, such as sandboxing, you'll want to be able to disable a group so that it can't be used to access a resource. The `UseForDenyOnly` flag solves the problem. When a SID is marked with this flag, it won't be considered when checking for allow access but will still be considered in deny-access checks. A user can mark their own groups as `UseForDenyOnly` by filtering their token and using it to create a new process. We'll discuss token filtering later in this chapter, when we describe restricted tokens in "Sandbox Tokens" on page XX.

Integrity and IntegrityEnabled

The `Integrity` and `IntegrityEnabled` attribute flags indicate that a SID represents the token's integrity level and is enabled. Group SIDs marked with the `Integrity` attribute flag store this integrity level as a 32-bit number in their final RID. The integrity SID is issued by the label security authority, which has the value 16. The RID can be any arbitrary value; however, there are seven pre-defined levels in the SDK, as shown in Table 4-3. Only the first six are in common use and accessible from a user process:

Table 4-3 Pre-Defined Integrity Level Values

Integrity Level	SDK Name	PowerShell Name
0	`SECURITY_MANDATORY_UNTRUSTED_RID`	`Untrusted`
4096	`SECURITY_MANDATORY_LOW_RID`	`Low`
8192	`SECURITY_MANDATORY_MEDIUM_RID`	`Medium`
8448	`SECURITY_MANDATORY_MEDIUM_PLUS_RID`	`MediumPlus`
12288	`SECURITY_MANDATORY_HIGH_RID`	`High`
16384	`SECURITY_MANDATORY_SYSTEM_RID`	`System`
20480	`SECURITY_MANDATORY_PROTECTED_PROCESS_RID`	`ProtectedProcess`

The default level for a user is `Medium`. Administrators are usually assigned `High`, and services are assigned `System`. We can query a token's integrity level SID using `Get-NtTokenSid`, as shown in Listing 4-11.

```
PS> Get-NtTokenSid $token -Integrity
Name                                    Sid
----                                    ---
Mandatory Label\Medium Mandatory Level  S-1-16-8192
```

Listing 4-10 Getting the token's integrity level SID

We can also set a new token integrity level if it's less than or equal to the current value. It's possible to increase it, but this requires special privileges and having `SeTcbPrivilege` enabled. While you can set the entire SID, it's usually more convenient to set just the value. For example, the script in Listing 4-11 will set the current token's integrity level to the `Low` level.

```
PS> Set-NtTokenIntegrityLevel Low -Token $token
PS> Get-NtTokenSid $token -Integrity
Name                                    Sid
```

```
----                                    ---
Mandatory Label\Low Mandatory Level     S-1-16-4096
```

Listing 4-11 Setting token integrity level to `Low`

If you run the script, you might find that you start to get errors in your PowerShell console due to blocked file access. We'll discuss why file access is blocked when we cover mandatory integrity control in Chapter 7.

Resource

The final attribute flag deserves only a passing mention. The `Resource` attribute indicates that the group SID is a *domain local SID*. We'll come back to this SID type in Chapter 10.

Device Groups

A token can also have a separate list of *device groups*. These group SIDs are added when a user authenticates to a server over a network in an enterprise environment, as shown in Listing 4-12.

```
PS> Get-NtTokenGroup -Device
Name                                Attributes
----                                ----------
BUILTIN\Users                       Mandatory, EnabledByDefault, Enabled
AD\CLIENT1$                         Mandatory, EnabledByDefault, Enabled
AD\Domain Computers                 Mandatory, EnabledByDefault, Enabled
NT AUTHORITY\Claims Value           Mandatory, EnabledByDefault, Enabled
--snip--
```

Listing 4-12 Displaying device groups using `Get-NtTokenGroup`

You can query the groups on the token by using `Get-NtTokenGroup` and passing the `Device` parameter.

Privileges

Groups allow system administrators to control a user's access to specific resources. *Privileges*, in contrast, are granted to a user to allow them to short-circuit certain security checks for all types of resources, such as by bypassing an access check. A privilege

can also apply to certain privileged actions, like changing the system's clock. You can view a token's privileges in the console using `Get-NtTokenPrivilege` (Listing 4-13).

```
PS> Get-NtTokenPrivilege $token
Name                             Luid                Enabled
----                             ----                 -------
SeShutdownPrivilege              00000000-00000013    False
SeChangeNotifyPrivilege          00000000-00000017    True
SeUndockPrivilege                00000000-00000019    False
SeIncreaseWorkingSetPrivilege    00000000-00000021    False
SeTimeZonePrivilege              00000000-00000022    False
```

Listing 4-13 Listing token privileges using `Get-NtTokenPrivilege`

The output is split into three columns. The first column is the privilege's common name. As with SIDs, the SRM does not use this name directly; instead, it uses the privilege's LUID value, which we can see in the second column. The last column indicates whether the privilege is currently enabled. Privileges can be in an enabled or disabled state.

Any check for a privilege should make sure that the privilege is enabled and not just present. In certain circumstances, such as sandboxing, a token might have a privilege listed, but the sandbox restrictions might prevent it from being marked as enabled. The `Enabled` flag is really a set of attribute flags, like the attributes for the group SIDs. We can view these attributes by formatting the privileges as a list (Listing 4-14).

```
PS> Get-NtTokenPrivilege $token -Privileges SeChangeNotifyPrivilege | Format-
List
Name        : SeChangeNotifyPrivilege
Luid        : 00000000-00000017
Attributes  : EnabledByDefault, Enabled
Enabled     : True
DisplayName : Bypass traverse checking
```

Listing 4-14 Display all properties of the `SeChangeNotifyPrivilege` privilege

In the output, we can now see the attributes, which include both `Enabled` and `EnabledByDefault`. The `EnabledByDefault` attribute specifies whether the default state of the privilege is to be enabled. We also now see an

additional `DisplayName` property, used to provide additional information to a user.

To modify the state of a token's privileges, you need `AdjustPrivileges` access on the token handle; then, you can use the `NtAdjustPrivilegesToken` system call to adjust the attributes and enable or disable a privilege. The `Enable-NtTokenPrivilege` and `Disable-NtTokenPrivilege` PowerShell commands expose this system call, as shown in Listing 4-15.

```
PS> Enable-NtTokenPrivilege SeTimeZonePrivilege -Token $token -PassThru
Name                           Luid                Enabled
----                           ----                -------
SeTimeZonePrivilege            00000000-00000022   True

PS> Disable-NtTokenPrivilege SeTimeZonePrivilege -Token $token -PassThru
Name                           Luid                Enabled
----                           ----                -------
SeTimeZonePrivilege            00000000-00000022   False
```

Listing 4-15 Enabling and disabling the `SeTimeZonePrivilege`

Using the `NtAdjustPrivilegesToken` API, it's also possible to remove a privilege entirely by specifying the `Remove` attribute, which you can accomplish with the `Remove-NtTokenPrivilege` PowerShell command. Removing a privilege ensures that the token can never use it again. If you only disable the privilege, then it could be re-enabled inadvertently. Listing 4-16 shows how to remove a privilege.

```
PS> Get-NtTokenPrivilege $token -Privileges SeTimeZonePrivilege
Name                           Luid                Enabled
----                           ----                -------
SeTimeZonePrivilege            00000000-00000022   False

PS> Remove-NtTokenPrivilege SeTimeZonePrivilege -Token $token
PS> Get-NtTokenPrivilege $token -Privileges SeTimeZonePrivilege
WARNING: Couldn't get privilege SeTimeZonePrivilege
```

Listing 4-16 Removing a privilege from a token

To check privileges, a user application can call the `NtPrivilegeCheck` system call, while kernel code can call the `SePrivilegeCheck` API. You might be wondering whether we couldn't just manually test whether a privilege is

enabled rather than using a dedicated system call. In this instance, yes; however, it's always worth using system facilities where possible in case you make a mistake in your implementation or haven't considered some edge case. The `Test-NtTokenPrivilege` PowerShell command wraps the system call, as shown in Listing 4-17.

```
PS> Set-NtTokenPrivilege SeChangeNotifyPrivilege
PS> Set-NtTokenPrivilege SeTimeZonePrivilege -Disable

PS> Test-NtTokenPrivilege SeChangeNotifyPrivilege
True

PS> Test-NtTokenPrivilege SeTimeZonePrivilege, SeChangeNotifyPrivilege -All
False

PS> Test-NtTokenPrivilege SeTimeZonePrivilege, SeChangeNotifyPrivilege
-All -PassResult
EnabledPrivileges           AllPrivilegesHeld
-----------------           -----------------
{SeChangeNotifyPrivilege}   False
```

Listing 4-17 Performing privilege checks

Listing 4-17 shows some example privilege checks using `Test-NtTokenPrivilege`. We start by enabling `SeChangeNotifyPrivilege` and disabling `SeTimeZonePrivilege` . These are common privileges granted to all users, but you might need to change the example if your token doesn't have them. We then test for just `SeChangeNotifyPrivilege`; it's enabled, so it returns `True`. Next, we check for both `SeTimeZonePrivilege` and `SeChangeNotifyPrivilege`; we can see that we don't have all the privileges, so it returns `False`. Finally, we run the same command but specify the `-PassResult` option to return the full check result. We can see in the `EnabledPrivileges` column that only `SeChangeNotifyPrivilege` is enabled.

The following are some of the privileges available on the system:

`SeChangeNotifyPrivilege`
This privilege name is misleading. It allows a user to receive notifications of changes to the filesystem or registry, but it's also

used to bypass traversal checking. We'll discuss traversal checking in Chapter 8.

`SeAssignPrimaryTokenPrivilege` **and** `SeImpersonatePrivilege`
These privileges allow the user to bypass the assigning primary token and impersonation checks, respectively. Unlike most privileges on this list, these must be enabled on the current process's primary token, not on an impersonation token.

`SeBackupPrivilege` **and** `SeRestorePrivilege`
These privileges allow the user to bypass the access check when opening specific resources, like files or registry keys. This lets the user back up and restore resources without needing to be granted access to them explicitly. These privileges have also been repurposed for other users: for example, the restore privilege allows a user to load arbitrary registry hives.

`SeSecurityPrivilege` **and** `SeAuditPrivilege`
The first of these privileges allows a user to be granted the `AccessSystemSecurity` access right on a resource. This allows the user to modify the resource's auditing configuration. The `SeAuditPrivilege` privilege allows a user to generate arbitrary object audit messages from a user application. We'll discuss auditing in Chapter 5 and 6.

`SeCreateTokenPrivilege`
This privilege should be given to only a very select group of users, as it grants the ability to craft arbitrary tokens using the `NtCreateToken` system call.

`SeDebugPrivilege`

The name of this privilege implies that it's necessary for debugging processes. However, that's not really the case, as it's possible to debug a process without it. The privilege does allow the user to bypass any access check when opening a process or thread object.

`SeTcbPrivilege`
The name of this privilege comes from *trusted computing base*

(TCB), a term used to refer to the privileged core of the Windows operating system, including the kernel. The privilege is a catch-all for privileged operations not covered by a more specific privilege. For example, it allows users to bypass the check for increasing the integrity level of a token (up to the limit of the `System` level), but also to specify a fallback exception handler for a process, two operations that have little in common.

`SeLoadDriverPrivilege`
We can load a new kernel driver through the `NtLoadDriver` system call, although it's more common to use the SCM. This privilege is required to successfully execute that system call. Note that having this privilege doesn't allow you circumvent kernel driver checks such as code signing.

`SeTakeOwnershipPrivilege` **and** `SeRelabelPrivilege`
These privileges have the same immediate effect: they allow a user to be granted `WriteOwner` access to a resource, even if the normal access control wouldn't allow it. `SeTakeOwnershipPrivilege` allows a user to take ownership of a resource, as having `WriteOwner` is necessary for that purpose. `SeRelabelPrivilege` bypasses checks on the mandatory label of a resource; normally, you can only set a label to be equal or lower than the caller's integrity level. Setting the mandatory label also requires `WriteOwner` access on a handle, as we'll see in Chapter 6.

We'll show specific examples of these privileges' uses in later chapters, when we discuss security descriptors and access checks. For now, let's turn to ways of restricting access through sandboxing.

Sandbox Tokens

In our connected world, we must process a lot of untrusted data. Attackers might craft data for malicious purposes, such as to exploit a security vulnerability in a web browser or a document reader. To counter this threat, Windows provides a method of

restricting the resources a user can access by placing any processes of theirs that handle untrusted data into a sandbox. If the process is compromised, the attacker will have only a limited view of the system won't be able to access the user's sensitive information. Windows implements sandboxes through three special token types: restricted tokens, write-restricted tokens, and lowbox tokens.

Restricted Tokens

The *restricted token* type is the oldest sandbox token in Windows. It was introduced as a feature in Windows 2000 but not used widely as a sandbox until the introduction of the Google Chrome web browser. Other browsers, such as Firefox, have since replicated Chrome's sandbox implementation, as have document readers such as Adobe Reader.

You can create a restricted token using the `NtFilterToken` system call or the `CreateRestrictedToken` Win32 API, which let you specify a list of restricted SIDs to limit the resources the token will be permitted to access. The SIDs do not have to already be available in the token. For example, Chrome's most restrictive sandbox specifies the NULL SID (`S-1-0-0`) as the only restricted SID. The NULL SID is never granted to a token as a normal group.

Any access check must allow both the normal list of groups as well as the list of restricted SIDs; otherwise, the user will be denied access, as we'll discuss in detail in Chapter 7. The `NtFilterToken` system call can also mark normal groups with the `UseForDenyOnly` attribute flag and delete privileges. We can combine the ability to filter a token with restricted SIDs or use it on its own, to create a lesser-privileged token without more comprehensive sandboxing.

It's easy to build a restricted token that can't access any resources. Such a restriction produces a good sandbox but also makes it impossible to use the token as a process's primary token, as the process won't be able to start. This puts a serious limitation on how effective a sandbox using restricted tokens can be. Let's

create a restricted token and extract the results with the script in Listing 4-18.

```
PS> $token = Get-NtToken -Filtered -RestrictedSids RC -SidsToDisable WD
-Flags DisableMaxPrivileges

PS> Get-NtTokenGroup $token -Attributes UseForDenyOnly
Name                           Attributes
----                           ----------
Everyone                       UseForDenyOnly

PS> Get-NtTokenGroup $token -Restricted
Name                           Attributes
----                           ----------
NT AUTHORITY\RESTRICTED        Mandatory, EnabledByDefault, Enabled

PS> Get-NtTokenPrivilege $token
Name                           Luid                Enabled
----                           ----                -------
SeChangeNotifyPrivilege        00000000-00000017   True

PS> $token.Restricted
True
```

Listing 4-18 Creating a restricted token and displaying groups and privileges

We start by creating a restricted token using the `Get-NtToken` command. We specify one restricted SIDs: `RC`, which maps to a special `NT AUTHORITY\RESTRICTED` SID that is commonly configured for system resources to permit read access. We also specify that we want to convert the `Everyone` group (`WD`) to `UseForDenyOnly`. Finally, we specify a flag to disable the maximum number of privileges.

Next, we display the properties of the token, starting with all normal groups, using the `UseForDenyOnly` attribute. The output shows that only the `Everyone` group has the flag set. We then display the restricted SIDs list, which shows the `NT AUTHORITY\RESTRICTED` SID. Finally, we display the privileges.

Note that even though we've asked to disable the maximum privileges, the `SeChangeNotifyPrivilege` is still there. This privilege is not deleted, as it can become very difficult to access resources without it. If you really want to get rid of it, you

can specify it explicitly to `NtFilterToken` or delete it after the token has been created.

Finally, we query the token property that indicates whether it's a restricted token.

INTERNET EXPLORER PROTECTED MODE

The first sandboxed web browser on Windows was Internet Explorer 7, introduced in Windows Vista. Internet Explorer 7 used the ability to lower the integrity level of a process's token to restrict the resources the browser could write to. Windows 8 ultimately replaced this simple sandbox, called *protected mode*, with a new type of token, the *lowbox* token, which we'll describe in "AppContainer and Lowbox Tokens" on page XX. The lowbox token provided greater isolation (called *enhanced protected mode*). It's interesting to note that Microsoft didn't use restricted tokens even though they had been available since Windows 2000.

Write-Restricted Tokens

A *write-restricted token* prevents write access to a resource but allows read and execute access. You can create a write-restricted token by passing the `WRITE_RESTRICTED` flag to `NtFilterToken`.

Windows XP SP2 introduced this token type to harden system services. It is much easier to use as a sandbox than restricted tokens, as you don't need to worry about the token not being able to read critical resources such as DLLs. However, it creates a less useful sandbox. For example, if you can read files for a user, you might be able to steal their private information, such as passwords stored by a web browser, without needing to escape the sandbox.

For completeness, let's create a write-restricted token and view its properties (Listing 4-19).

```
PS> $token = Get-NtToken -Filtered -RestrictedSids WR -Flags WriteRestricted
PS> Get-NtTokenGroup $token -Restricted
Name                           Attributes
----                           ----------
NT AUTHORITY\WRITE RESTRICTED  Mandatory, EnabledByDefault, Enabled

PS> $token.Restricted
True

PS> $token.WriteRestricted
```

```
True
```

Listing 4-19 Creating a write-restricted token

We start by creating a write-restricted token using the `Get-NtToken` command. We specify one restricted SID, `WR`, which maps to a special `NT AUTHORITY\WRITE RESTRICTED` SID that is equivalent to `NT AUTHORITY\RESTRICTED` but assigned to write access on specific system resources. We also specify the `WriteRestricted` flag to make a write-restricted token rather than a normal restricted token.

Next, we display the token's properties. In the list of restricted SIDs, we see `NT AUTHORITY\WRITE RESTRICTED`. If we display the `Restricted` property, we find that the token is considered restricted. However, it's also marked as `WriteRestricted`.

AppContainer and Lowbox Tokens

Windows 8 introduced the *AppContainer* sandbox to protect a new Windows application model. AppContainer implements its security using a *lowbox token*. You can create a lowbox token from an existing token with the `NtCreateLowBoxToken` system call. There is no direct equivalent Win32 API for this system call, but you can create an AppContainer process using the `CreateProcess` API. We won't go into more detail here on how to create a process using this API; instead, we'll focus only on the lowbox token.

When creating a lowbox token, you need to specify a package SID and a list of capability SIDs. Both SID types are issued by the *package authority* (which has the value of 15) You can distinguish between package SIDs and capability SIDS by checking their first RID, which should be `2` and `3`, respectively. The package SID works like the user's SID in the normal token, whereas the capability SIDs act like restricted SIDs. We'll leave the actual details of how these affect an access check to Chapter 7.

Capability SIDs modify the access check process but can mean something in isolation. For example, there's capabilities to allow network access that are handled specially by the Windows Firewall even through that's not directly related to access checking. There are two types of capability SIDs:

Legacy. A small set of predefined SIDs introduced in Windows 8.

Named
The SID's RIDs, derived from a textual name

Appendix A contains a more comprehensive list of named capability SIDs. Table 4-4 shows the legacy capabilities.

Table 4-4 Legacy Capability SIDs

Capability name	SID
Your Internet connection	S-1-15-3-1
Your Internet connection, including incoming connections from the Internet	S-1-15-3-2
Your home or work networks	S-1-15-3-3
Your pictures library	S-1-15-3-4
Your videos library	S-1-15-3-5
Your music library	S-1-15-3-6
Your documents library	S-1-15-3-7
Your Windows credentials	S-1-15-3-8
Software and hardware certificates or a smart card	S-1-15-3-9
Removable storage	S-1-15-3-10
Your Appointments	S-1-15-3-11
Your Contacts	S-1-15-3-12
Internet Explorer	S-1-15-3-4096

We can use `Get-NtSid` to query for package and capability SIDs (Listing 4-20).

```
PS> Get-NtSid -PackageName 'my_package' -ToSddl
1 S-1-15-2-4047469452-4024960472-3786564613-914846661-3775852572-3870680127-
  2256146868

2 PS> Get-NtSid -PackageName 'my_package' -RestrictedPackageName "CHILD" -ToSddl
  S-1-15-2-4047469452-4024960472-3786564613-914846661-3775852572-3870680127-
  2256146868-951732652-158068026-753518596-3921317197
```

```
PS> Get-NtSid -KnownSid CapabilityInternetClient -ToSddl
S-1-15-3-1

PS> Get-NtSid -CapabilityName registryRead -ToSddl
S-1-15-3-1024-1065365936-1281604716-3511738428-1654721687-432734479-
3232135806-4053264122-3456934681

PS> Get-NtSid -CapabilityName registryRead -CapabilityGroup  -ToSddl
S-1-5-32-1065365936-1281604716-3511738428-1654721687-432734479-3232135806-
4053264122-3456934681
```

Listing 4-20 Creating package and capability SIDs

We create two package SIDs and two capability SIDs. We generate the first package SID by specifying its name to `Get-NtSid` and receive the resulting SID 1. This package SID is derived from the lowercase form of the name hashed with the SHA256 digest algorithm. The 256-bit digest is broken up into seven 32-bit chunks that act as the RIDs. The final 32-bit value of the digest is discarded.

Windows also supports a restricted package SID. This restricted package SID is designed to allow a package to create new secure child packages that can't interact with each other. The classic Edge web browser used this feature to separate internet- and intranet-facing children so that if one was compromised, it couldn't access data in the other. To create the child, you use the original package family name plus a child identifier 2. The created SID extends the original package SID with another four RIDs, as you can see in the output.

The first capability SID 3 is a legacy capability for internet access. Note that the resulting SDDL SID has one additional RID value (`1`). The second SID is derived from a name 4, in this case the name `registryRead`, which is used to allow read access to a group of system registry keys. Like the package SID, the named capability RIDs are generated from the SHA256 hash of the lowercase name. To differentiate between legacy and named capability SIDs, the second RID is set to `1024` followed by the SHA256 hash. You can generate your own capability SIDs using this method, although there's not much you can do with the capability unless some resource is configured to use it.

Windows also supports a *capability group*, a group SID that can be added to the normal list of groups 5. A capability group sets the first RID to `32` and the rest of the RIDs to the same SHA256 hash that was derived from the capability name. Now that we've got the SIDs, we can create a lowbox token using `Get-NtToken` (Listing 4-21).

```
1 PS> $token = Get-NtToken -LowBox -PackageSid 'my_package'
  -CapabilitySid "registryRead", "S-1-15-3-1"
2 PS> Get-NtTokenGroup $token -Capabilities | Select-Object Name
  Name
  ----
  NAMED CAPABILITIES\Registry Read
  APPLICATION PACKAGE AUTHORITY\Your Internet connection

3 PS> $package_sid = Get-NtTokenSid $token -Package -ToSddl
  PS> $package_sid
  S-1-15-2-4047469452-4024960472-3786564613-914846661-3775852572-3870680127-
  2256146868

4 PS> Get-NtTokenIntegrityLevel $token
  Low

  PS> $token.Close()
```

Listing 4-21 Creating an AppContainer token and listing its properties

Let's go through each of these steps. First, we call `Get-NtToken`, passing it the package name (though the SID as SDDL would also work). We also pass it the list of capabilities to assign to the lowbox token 1. We can then query for the list of capabilities 2. Notice that the names of the two capability SIDs are different: the SID derived from a name is prefixed with the `NAMED CAPABILITIES`. There's no way of converting a named capability SID back to the name it was derived from; the PowerShell module must generate the name based on a large list of known capabilities. The second SID is a legacy SID, so LSASS can resolve it back to a name.

Next, we query the package SID 3. As the package SID is derived from a name using SHA256, it's not possible to resolve back to the package name. Again, the PowerShell module has a list of names that it can use to work out what the original name was.

An lowbox token is always set to the `Low` integrity level [4]. In fact, if a privileged user changes the integrity level to `Medium` or above, all lowbox properties, such as package SIDs and capability SIDs, are removed, and the token reverts non-sandbox token.

We've covered making a user less privileged by converting their token into a sandbox token. We'll now go to the other side, and look at what makes a user privileged enough to administrate the Windows system.

What Makes an Administrator User?

If you come from a Unix background, you'll know user ID 0 as the administrator account, or *root*. As root, you can access any resource and configure the system however you'd like. When you install Windows, the first account you configure will also be an administrator. However, unlike root, the account won't have special SID that the system treats differently. So, what makes an administrator account on Windows?

The basic answer is that Windows is configured to give certain groups and privileges special access. Administrator access is inherently discretionary, meaning it's possible to be an administrator but still be locked out of resources; there is no real equivalent of a root account (although the *SYSTEM* user comes close).

Administrators generally have three characteristics. First, when you configure a user to be an administrator, you typically add them to the `BUILTIN\Administrators` group, then configure Windows to allow access to the group when performing an access check. For example, the system folders, such as *C:\Windows*, are configured to allow the group to create new files and directories.

Second, administrators are granted access to "God" privileges, which effectively circumvent the system's security controls. For example, `SeDebugPrivilege` allows a user to get full access to any other process or thread on the system, no

matter what security it has been assigned. With access to a process, it's possible inject code into it and run as a different user, even if that other user is an administrator.

Third, administrators run at an integrity level above `Medium`. An administrator runs at the `High` level, whereas system services run at the `System` level. By increasing the administrator's integrity level, we make it harder to accidentally leave administrator resources, especially processes and threads, accessible to non-administrators. A common misconfiguration is granting weak access control to a resource; however, if the resource is also marked with an integrity level above `Medium`, then non-administrator users won't be able to write to the resource.

A quick way to check whether a token is an administrator is to the check the `Elevated` property on the `Token` object. This property indicates whether the token has certain groups and available privileges found in a fixed list in the kernel. Listing 4-23 shows an example for a non-administrator.

```
PS> $token = Get-NtToken
PS> $token.Elevated
False
```

Listing 4-22 The elevated property for a non-administrator

If the token has one of the following "God" privileges, it's automatically considered elevated.

- `SeCreateTokenPrivilege`
- `SeTcbPrivilege`
- `SeTakeOwnershipPrivilege`
- `SeLoadDriverPrivilege`
- `SeBackupPrivilege`
- `SeRestorePrivilege`
- `SeDebugPrivilege`

- `SeImpersonatePrivilege`
- `SeRelabelPrivilege`
- `SeDelegateSessionUserImpersonatePrivilege`

The privilege doesn't have to be enabled, just available in the token.

For elevated groups, the kernel doesn't have a fixed list of SIDs; instead, it inspects only the last RID of the SID. If the RID is set to one of the following values, then the SID is considered elevated: 114, 498, 512, 516, 517, 518, 519, 520, 521, 544, 547, 548, 549, 550, 551, 553, 554, 556, and 569. For example, the `BUILTIN\Administrators` group is `S-1-4-32-544`. As 544 is in this list, the SID is considered elevated. (Note that the SID `S-1-1-2-3-4-544` would also be considered elevated, even though there is nothing special about it.)

HIGH INTEGRITY LEVEL DOESN'T EQUAL ADMINISTRATOR

It's a common misconception that if a token has a `High` integrity level, it's an administrator token. However, the `Elevated` property doesn't check a token's integrity level, just its privileges and groups. The `BUILTIN\Administrators` group would still function with a lower integrity level, allowing access to resources such as the Windows filesystem directory. The only restriction is that "God" privileges can't be enabled if the integrity level is less than `High`.

It is also possible for a non-administrator to run with a `High` integrity level, as in the case of UI access processes, which sometimes run at a `High` integrity level but are not granted any special privileges or groups to make them an administrator.

User Account Control

I mentioned that when you install a new copy of Windows, the first user you create is always an administrator. It's important to configure the user in this way; otherwise, it would be impossible to modify the system and install new software.

However, prior to Windows Vista, this default behavior was a massive security liability, because average consumers would install the default account and likely never change it. This meant that most people used a full administrator account for everyday activities like surfing the web. If a malicious attacker were able to

exploit a security issue in the user's browser, the attacker would get full control over the Windows machine. In the days prior to widespread sandboxing, this threat proved serious.

In Vista, Microsoft changed this default behavior by introducing *User Account Control (UAC)* and the split-token administrator. In this model the default user remains an administrator; however, by default, all programs run with a token whose administrator groups and privileges have been removed. When a user needs to perform an administrative task, the system elevates a process to a full administrator and shows a prompt, like the one in Figure 4-3, requesting the user's confirmation before continuing.

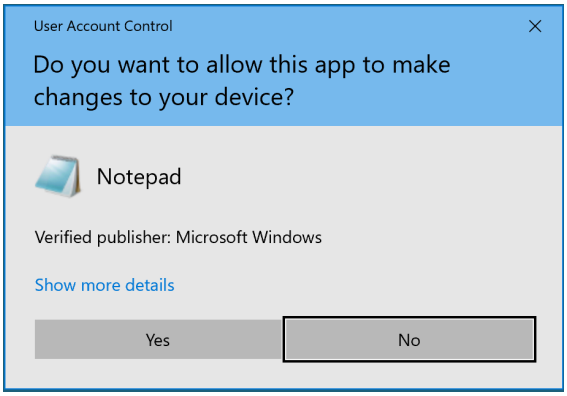

Figure 4-3 The User Account Control consent dialog for privilege elevation

To make using Windows easier for users, you can configure a program to force this elevation when it's started. A program's elevation property is stored in a manifest XML file embedded in the executable image. Run the example in Listing 4-23 to get the manifest information for all the executables in the *SYSTEM32* directory.

```
PS> ls c:\windows\system32\*.exe | Get-ExecutableManifest
Name                            UiAccess   AutoElevate     ExecutionLevel
----                            --------   -----------     --------------
aitstatic.exe                   False      False           asInvoker
alg.exe                         False      False           asInvoker
appidcertstorecheck.exe         False      False           asInvoker
appidpolicyconverter.exe        False      False           asInvoker
ApplicationFrameHost.exe        False      False           asInvoker
```

```
appverif.exe                    False      False          highestAvailable
--snip--
```

Listing 4-23 Querying executable manifest information

If it's a special, Microsoft-approved program, the manifest can specify whether the program should be automatically, and silently, elevated, as indicated by the `AutoElevate` column. The manifest also indicates whether the process can run with UI access, a topic we'll discuss later in "User Account Control" on page XX. There are three possible values for the `ExecutionLevel` column:

`asInvoker`
Run the process as the user who created it. This is the default setting.

`highestAvailable`
If the user is a split-token administrator, then force elevation to the administrator token. If not, then run as the user who created the process.

`requireAdministrator`
Whether the user is a split-token administrator or not, elevation will be forced. If the user's not an administrator, they'll be prompted for a password for an administrator account.

When something creates an executable with an elevated execution level, the application information service calls the RPC method `RAiLaunchAdminProcess`. This method checks the manifest and starts the elevation process, including showing the consent dialog. It's also possible to manually elevate any application using the `ShellExecute` API and requesting the `runas` operation. PowerShell exposes this behavior using the `Start-Process` command, as shown below:

```
PS> Start-Process notepad -Verb runas
```

When you run this command, you should see the UAC prompt. If you click the yes button on the prompt, *notepad.exe* should run as an administrator on the desktop.

Linked Tokens and Elevation Type

When an administrator authenticates to the desktop, the system tracks two tokens for the user:

`Limited`
The unelevated token used for most running processes

`Full`
The full administrator token, used only after elevation

The name *split-token administrator* comes from these two tokens, as the user's granted access is split between the limited and full tokens.

Where does the system keep track of the two tokens? The `Token` object has a field used to link the tokens together, and we can query it as a normal user with `NtQueryInformationToken` and the `TokenLinkedToken` information class. Querying the linked token for a `Limited` token returns the `Full` token, and vice-versa. In Listing 4-24.

```
1 PS> Use-NtObject($token = Get-NtToken -Linked) {
          Format-NtToken $token -Group -Privilege -Integrity -Information
      }
      GROUP SID INFORMATION
      -----------------
      Name                           Attributes
      ----                           ----------
2     BUILTIN\Administrators         Mandatory, EnabledByDefault, Enabled, Owner
      --snip--

      PRIVILEGE INFORMATION
      ---------------------
      Name                           Luid              Enabled
      ----                           ----              -------
      SeIncreaseQuotaPrivilege       00000000-00000005 False
3     SeSecurityPrivilege            00000000-00000008 False
      SeTakeOwnershipPrivilege       00000000-00000009 False
      --snip--

      INTEGRITY LEVEL
      ---------------
4     High

      TOKEN INFORMATION
```

```
   -----------------
5  Type           : Impersonation
   Imp Level      : Identification
   Auth ID        : 00000000-0009361F
6  Elevated       : True
7  Elevation Type: Full
   Flags          : NotLow
```

Listing 4-24 Displaying properties of the linked token

We first access the linked token by passing the `Linked` parameter to `Get-NtToken` 1. We then format the token to display its groups, privileges, integrity level, and token information. In the groups, we can see the `BUILTIN\Administrators` group enabled 2. In the privileges, we find some "God" privileges, such as `SeSecurityPrivilege` 3. The combination of the groups and privileges confirm this is an administrator token.

The integrity level of the token is set to `High` 4, which, as we discussed earlier, prevents the token from accidentally leaving sensitive resources accessible to the non-administrator user. In the token information, one interesting result is an impersonation token at Identification level 5. To get a token that can create a new process, the caller needs `SeTcbPrivilege`, which means only system services, such as the application information service, can get the token. Finally, we can see that the token is marked as elevated 6, and the that the token elevation type indicates this is the full token 7. Let's compare it with the limited token in Listing 4-25.

```
1  PS> Use-NtObject($token = Get-NtToken) {
       Format-NtToken $token -Group -Privilege -Integrity -Information
   }
   GROUP SID INFORMATION
   -----------------
   Name                                                        Attributes
   ----                                                        ----------
2  BUILTIN\Administrators                                      UseForDenyOnly
   --snip--

   PRIVILEGE INFORMATION
   ---------------------
   Name                          Luid              Enabled
   ----                          ----              -------
3  SeShutdownPrivilege           00000000-00000013 False
```

```
   SeChangeNotifyPrivilege       00000000-00000017 True
   SeUndockPrivilege             00000000-00000019 False
   SeIncreaseWorkingSetPrivilege 00000000-00000021 False
   SeTimeZonePrivilege           00000000-00000022 False

   INTEGRITY LEVEL
   ---------------
4  Medium

   TOKEN INFORMATION
   -----------------
   Type           : Primary
   Auth ID        : 00000000-0009369B
5  Elevated       : False
6  Elevation Type: Limited
7  Flags          : VirtualizeAllowed, IsFiltered, NotLow
```

Listing 4-25 Displaying properties of the limited token

We get a handle to the current token and format it with the same formatting we used in Listing 4-24 1. We can see that the `BUILTIN\Administrators` group has been converted to a `UseForDenyOnly` group 2. Any other group that would match the elevated RID check would be converted in the same way.

Next, the output shows only five privileges 3. These are the only five privileges that the limited token can have. Even if a privilege isn't considered a "God" privilege such as the `SeIncreaseQuotaPrivilege` we saw in Listing 4-24, it will be removed from the limited token. The integrity level of the token is set to `Medium`, from `High` in the full token 4. In the token information, the token is not elevated 5, and the elevation type indicates that this is the limited token 6.

Finally, the flags contain the value `IsFiltered` 7. This flag indicates the token has been filtered using the `NtFilterToken` system call. This is because, to create the limited token, LSASS will first create a new full token so that its authentication ID has a unique value. (If you compare the `Auth ID` values in Listing 4-24 and 4-25, you'll notice they're indeed different.) This allows the SRM to consider the two tokens to be in separate logon sessions.

LSASS then passes the token to `NtFilterToken` with the `LuaToken` parameter flag to convert any elevated group to `UseForDenyOnly` and delete all privileges other than the five permitted ones. `NtFilterToken` does not drop the integrity level from `High` to `Medium`—that must be done separately. Then, LSASS calls `NtSetInformationToken` to link the two tokens together using the `TokenLinkedToken` information class.

There is a third type of elevation, *default*, used for any token not associated with a split-token administrator:

```
PS> Use-NtObject($token = Get-NtToken -Anonymous) { $token.ElevationType }
Default
```

In this example, the anonymous user's token is not a split-token administrator, so it has the default token elevation type.

UI Access

One of the other security features introduced in Windows Vista is *User Interface Privilege Isolation (UIPI)*, which prevents a lower-privileged process from programmatically interacting with the user interface of a more privileged process. This is enforced using integrity levels, and it's another reason UAC administrators run at a `High` integrity level.

But UIPI presents a problem for applications that are designed to interact with the user interface, such as screen readers and touch keyboards. To get around this limitation without granting the process too much privilege, a token can set a UI access flag. Whether a process is granted UI access depends on the `uiAccess` setting in the executable's manifest file.

This UI access flag signals to the desktop environment that it should disable the UIPI checks. In Listing 4-26, we query for this flag in a suitable process, the On-Screen Keyboard (OSK).

```
PS> $process = Start-Process "osk.exe" -PassThru
PS> $token = Get-NtToken -ProcessId $process.Id
PS> $token.UIAccess
True
```

Listing 4-26 Querying the UI access flag in the On-Screen Keyboard primary token

We start the OSK and open its `Token` object to query the UI access flag. To set this flag, the caller needs the `SeTcbPrivilege` privilege. The only way to create a UI access process as a normal user is to use the UAC service. Therefore, any UI access process needs to be started with `ShellExecute`, which is why we used `Start-Process` in Listing 4-26. This all happens behind the scenes when you create the UI access application.

Virtualization

Another problem introduced in Vista because of UAC is the question of how to handle legacy applications, which expect to be able to write to administrator-only locations such as the *Windows* folder or the Local Machine Registry Hive. Vista implemented a special workaround: if a virtualization flag is enabled on the primary token, it will silently redirect writes from these locations to a per-user store. This made it seem to the process as if it had successfully added resources to secure locations.

By default, the virtualization flag is enabled on legacy applications automatically. However, you can specify it manually by setting a property on the primary token. Run the commands in Listing 4-27 in a non-administrator shell.

```
1 PS> $file = New-NtFile -Win32Path c:\windows\hello.txt -Access GenericWrite
  New-NtFile : (0xC0000022) - {Access Denied}
  A process has requested access to an object, but has not been granted those
  access rights.

  PS> $token = Get-NtToken
2 PS> $token.VirtualizationEnabled = $true
3 PS> $file = New-NtFile -Win32Path c:\windows\hello.txt -Access GenericWrite
4 PS> $file.Win32PathName
  C:\Users\user\AppData\Local\VirtualStore\Windows\hello.txt
```

Listing 4-27 Enabling virtualization on the `Token` object and creating a file in *C:\Windows*

In Listing 4-27, we first try to create a writeable file, *c:\windows\hello.txt* 1. This operation fails with an access denied exception. We then get the current primary token and set the

`VirtualizationEnabled` property to `True` 2. When we repeat the file creation, it now succeeds 3. If we query the location of the file, we find it's under the user's directory in a virtual store 4. Only normal, unprivileged tokens can enable virtualization; system service or administrators tokens have virtualization disabled. You can learn whether virtualization is permitted by querying the `VirtualizationAllowed` property on the `Token`.

Security Attributes

A token's *security attributes* are a list of name-value pairs that provide arbitrary data. There are three types of security attributes associated with a token: *local*, *user claims*, and *device claims*. Each security attribute can have one or more values, which must all be of the same type. Table 4-5 shows the valid types for a security attribute.

Table 4-5 Security Attribute Types

Type name	Description
`Int64`	Signed 64-bit integer
`UInt64`	Unsigned 64-bit integer
`String`	A Unicode string
`Fqbn`	A fully qualified binary name. Contains a version number and a Unicode string.
`Sid`	A SID
`Boolean`	A true or false value, stored as an Int64, with 0 being false and 1 being true
`OctetString`	An arbitrary array of bytes

A set of flags can be assigned to the security attribute to change aspects of its behavior, such as whether new tokens can inherit it. Table 4-6 shows the defined flags.

Table 4-6 Security Attribute Flags

Flag name	Description
`NonInheritable`	The security attribute can't be inherited by a child process token
`CaseSensitive`	If the security attribute contains a string value, the comparison should be case-sensitive
`UseForDenyOnly`	The security attribute is used only when checking for denied access

DisabledByDefault	The security attribute is disabled by default
Disabled	The security attribute is disabled
Mandatory	The security attribute is mandatory
Unique	The security attribute should be unique on the local system
InheritOnce	The security attribute can be inherited once by a child, then should be set NonInheritable

Almost every process token has the TSA://ProcUnique security attribute. This security attribute contains a unique LUID allocated during process creation. We can display its value for the effective token using Show-NtTokenEffective (Listing 4-28).

```
PS> Show-NtTokenEffective -SecurityAttributes
SECURITY ATTRIBUTES
-------------------
Name              Flags                  ValueType Values
----              -----                  --------- ------
TSA://ProcUnique NonInheritable, Unique UInt64    {133, 1592482}
```

Listing 4-28 Querying the security attributes for the current process

From the output, we can see that the name of the attribute is TSA://ProcUnique. It has two UInt64 values, which form a LUID when combined. Finally, it has two flags: NonInheritable, which means the security attribute won't be passed to new process tokens, and Unique, which means the kernel shouldn't try to merge the security attribute with any other attribute on the system with the same name.

To set local security attributes, the caller needs the SeTcbPrivilege privilege before calling NtSetInformationToken. User and device claims must be set during token creation, which we discuss in the next section.

Creating Tokens

Typically, LSASS creates tokens when a user authenticates to the computer. However, it can also create tokens for users who don't exist, such as virtual accounts used for services. These tokens might be interactive, for use in a console session, or they could be network tokens for use over the local network. A locally

authenticated user can create another user's token by calling a Win32 API such as `LogonUser`, which calls into LSASS to perform the token creation.

We won't discuss LSASS at length until Chapter 10. However, it's worth understanding how LSASS creates tokens. To do so, LSASS calls the `NtCreateToken` system call. As I mentioned earlier, this system call requires the `SeCreateTokenPrivilege` privilege, which is granted to a limited number of processes. This privilege is about as privileged as it gets, as you can use it to create arbitrary tokens with any group or user SID and access any resource on the local machine.

While you won't often have to call `NtCreateToken` from PowerShell, you can do so through the `New-NtToken` command so long as you have `SeCreateTokenPrivilege` enabled. The `NtCreateToken` system call takes the following parameters:

Token Type
Either primary or impersonation

Authentication ID
The LUID authentication ID; can be set to any value you'd like

Expiration Time
Allows the token to expire after a set period

User
The user SID

Groups
The list of group SIDs

Privileges
The list of privileges

Owner
The owner SID

Primary Group

The primary group SID

Source
The source information name

In addition, Windows 8 introduced new features to the system call, which you can access though the `NtCreateTokenEx` system call:

Device Group
A list of additional SIDs for the device

Device Claim Attributes
A list of security attributes to define device claims

User Claim Attributes
A list of security attributes to define user claims

Mandatory Policy
A set of flags that indicate the token's mandatory integrity policy

Anything not in these two lists can be configured only by calling `NtSetInformationToken` after the new token has been created. Depending on what token property is being set, you might need a different privilege, such as `SeTcbPrivilege`. Let's demonstrate how to create a new token using the script in Listing 4-29, which you must run as an administrator.

```
PS> Set-NtTokenPrivilege SeDebugPrivilege
PS> $imp = Use-NtObject($p = Get-NtProcess -Name lsass.exe) {
    Get-NtToken -Process $p -Duplicate
}
PS> Set-NtTokenPrivilege SeCreateTokenPrivilege -Token $imp
PS> $token = Invoke-NtToken $imp {
     New-NtToken -User "S-1-0-0" -Group "S-1-1-0"
}
PS> Format-NtToken $token -User -Group
USER INFORMATION
----------------
Name     Sid
----     ---
NULL SID S-1-0-0

GROUP SID INFORMATION
-----------------
```

```
   Name                                 Attributes
   ----                                 ----------
5  Everyone                             Mandatory, EnabledByDefault, Enabled
   Mandatory Label\System Mandatory Level Integrity, IntegrityEnabled
```

Listing 4-29 Creating a new token

A normal administrator does not have `SeCreateTokenPrivilege` by default. Therefore, we'll need to borrow a token from another process that does. In most cases, the easiest process to borrow from is LSASS. We open the LSASS process and its token, duplicating it to an impersonation token 1. Next, we ensure that `SeCreateTokenPrivilege` is enabled on the token 2. We can then impersonate the token and call `New-NtToken`, passing it a SID for the user and a single group 3. Finally, we can print out the details for the new token, including its user SID set 4 and group set 5. The `New-NtToken` command also adds a default system integrity level SID that you can see in the group list.

Token Assignment

If a normal user account could assign arbitrary primary or impersonation tokens, it could elevate its privileges to access the resources of other users. This is especially problematic when it comes to impersonation, as another user account need only open a named pipe to inadvertently allow the server to get an impersonation token.

For that reason, the SRM imposes limits on what a normal user can do without the privileges `SeAssignPrimaryTokenPrivilege` or `SeImpersonationPrivilege`. Let's go into the criteria necessary to assign a token for a normal user.

Assigning a Primary Token

A new process can be assigned a primary token in one of three ways:

- It can inherit the token from the parent process.
- The token can be assigned during process creation (for example, using the `CreateProcessAsUser` API).
- The token can be set after process creation using `NtSetInformationProcess`, before the process starts.

Inheriting the token from the parent is by far the most common means of token assignment. For example, when you start an application from the Windows Start Menu, the new process will inherit the token from the Explorer process.

If a process does not inherit a token from its parent, the process will be passed the token as a handle that must have the `AssignPrimary` access right. If the access to the `Token` object is granted, the SRM imposes further criteria on the token to prevent the assignment of a more privileged token (unless the caller's primary token has `SeAssignPrimaryTokenPrivilege` enabled).

The kernel function `SeIsTokenAssignableToProcess` imposes the token criteria. First, it checks that the assigned token must have an integrity level less than or equal to that of the current process's primary token. If that criterion is met, it then checks whether the token meets either of the criteria shown in Figure 4-4: that the token is either a child of the caller's primary token or a sibling of the primary token.

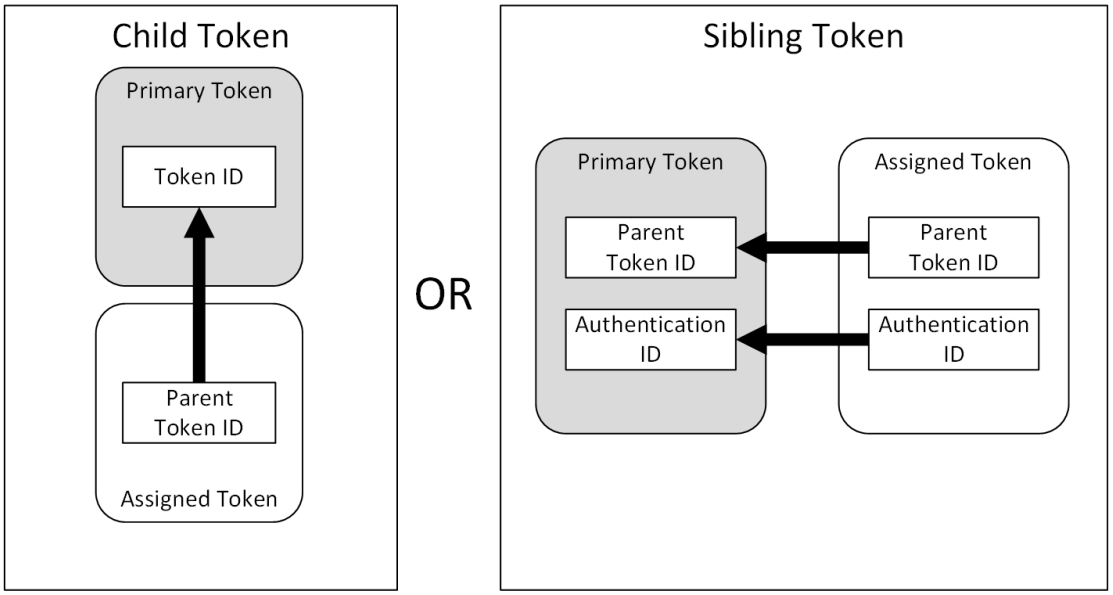

Figure 4-4 The `SeIsTokenAssignableToProcess` primary token assignment criteria

Let's first cover the case of a child token. A user process can create a new token based on an existing one. When this occurs, the `ParentTokenId` property in the new token's kernel object is set to the ID of the parent token. If the new token's `ParentTokenId` matches the current primary token's ID value, then the assignment is granted. Restricted tokens are examples of child tokens; when you create a restricted token using `NtFilterToken`, the new token's parent token ID is set to the ID of the original token.

A *sibling token* is a token created as part of the same authentication session as the existing token. To test this criterion, the function compares the parent token ID and the authentication ID of the two tokens. If both fields are equal, then the token can be assigned. This check also tests whether the authentication sessions are special sibling sessions set by the kernel (a rare configuration). Common examples of a sibling token include tokens duplicated from the current process token and lowbox tokens.

Note that the function doesn't checks the user that the token represents, and if the token matches one of the criteria, it's possible to assign it to a new process. If the criteria don't match, then the `STATUS_PRIVILEGE_NOT_HELD` error will be returned during token assignment.

How does the `runas` utility creates a new process as a normal user with these restrictions? It uses the `CreateProcessWithLogon` API, which authenticates a user and starts the process from a system service that has the required privileges to bypass these checks.

If we try to assign a process token, we'll see how easily the operation can fail, even when we're assigning tokens for the same user. Run the code in Listing 4-30 as a non-administrator user.

```
PS> $token = Get-NtToken -Filtered -Flags DisableMaxPrivileges
1 PS> Use-NtObject($proc = New-Win32Process notepad -Token $token) {
    $proc | Out-Host
  }
  Process            : notepad.exe
  Thread             : thread:11236 - process:9572
  Pid                : 9572
  Tid                : 11236
  TerminateOnDispose : False
  ExitStatus         : 259
  ExitNtStatus       : STATUS_PENDING

2 PS> $token = Get-NtToken -Filtered -Flags DisableMaxPrivileges -Token $token
  PS> $proc = New-Win32Process notepad -Token $token
3 Exception calling "CreateProcess" with "1" argument(s): "A required privilege
  is not held by the client"
```

Listing 4-30 Creating a process using restricted tokens

We create two restricted tokens and use them to create an instance of Notepad. In the first attempt, we create the token based on the current primary token 1. The parent token ID field in the new token will be set to the primary token's ID, and when we use the token during process creation, the operation succeeds.

In the second attempt, we create another token 2, but base it on the one we created previously. Creating a process with this token fails with a privilege error 3. This is because the second token's parent token ID is set to the ID of the crafted token, not

the primary token. As the token doesn't meet either the child or sibling criteria, it will fail during assignment.

You can set the token after creating the process by using the `NtSetInformationProcess` system call or `ProcessAccessToken`, which PowerShell exposes with the `Set-NtToken` command, used in Listing 4-31.

```
PS> $proc = Get-NtProcess -Current
PS> $token = Get-NtToken -Duplicate -TokenType Primary
PS> Set-NtToken -Process $proc -Token $token
Set-NtToken : (0xC00000BB) - The request is not supported.
```

Listing 4-31 Setting an access token after a process has started

As you can see, this operation does not circumvent any of the assignment checks we've discussed. Once the process's initial thread starts executing, the option to set the primary token is disabled, so when we try to set the token on a started process, we get the `STATUS_UNSUPPORTED` error.

Assigning an Impersonation Token

As with primary tokens, the SRM requires that an assigned impersonation token meet a specific set of criteria; otherwise, it will reject the assignment of the token to a thread. Interestingly, the criteria are not the same as those for the assignment of primary tokens. This can lead to situations in which it's possible to assign an impersonation token but not a primary token, and vice versa.

If the token is specified explicitly, then the handle must have the `Impersonate` access right. If the impersonation happens implicitly, then the kernel is already maintaining the token, and it requires no specific access right.

The `SeTokenCanImpersonate` function in the kernel handles the check for the impersonation criteria. As shown in Figure 4-5, this check significantly more complex than that for assigning primary tokens.

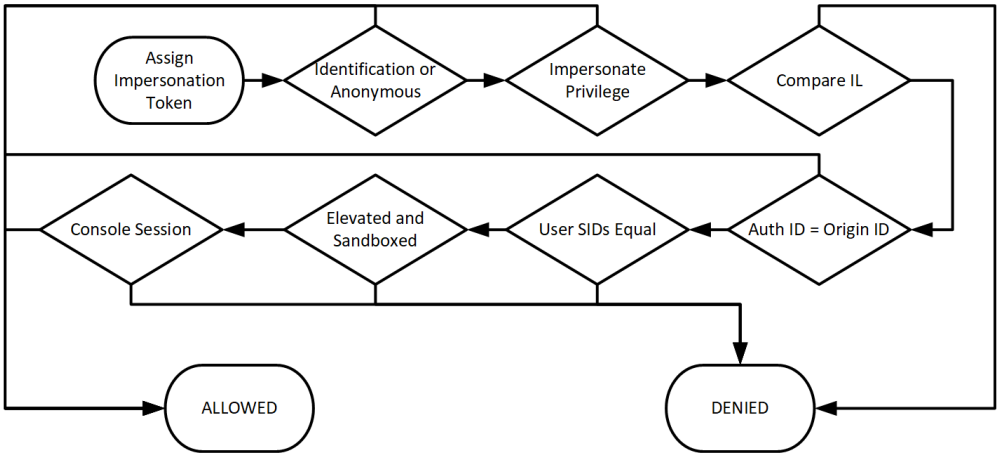

Figure 4-5 The `SeTokenCanImpersonate` impersonation token checks

Let's walk through each check and describe what it considers on both the impersonation and the primary token. Note that, because it's possible to assign an impersonation token to a thread in another process (if you have an appropriate handle to that thread), the primary token being checked is the one assigned to the process that encapsulates the thread, and not the primary token of the calling thread:

Checks for an Identification or Anonymous impersonation level. The first check tests whether the impersonation token has an impersonation level of Identification or Anonymous. If so, assigning it to the thread isn't a security risk, and the SRM immediately allows the assignment. This check also checks the token's authentication ID to see whether it is the `ANONYMOUS LOGON` token.

Checks for the impersonate privilege. The next check determines whether the primary token has `SeImpersonatePrivilege` enabled. If so, the SRM again immediately allows the assignment.

Compares integrity levels. The SRM compares integrity level of the primary token to that of the impersonation token. If the primary token's integrity level is less than that of the impersonation token, the assignment is denied. If it's greater or equal, the checks continue.

Checks that the authentication ID equals the origin ID. If the origin logon ID of the impersonation token equals the authentication ID of the primary token, the SRM allows the assignment. Otherwise, it continues making checks.

Note that there is an interesting consequence of this check. As discussed earlier in this chapter, the origin logon ID of normal user tokens is set to the authentication identifier of the *SYSTEM* user. This is because the authenticating process runs as the *SYSTEM* user. As a consequence, the *SYSTEM* user can impersonate any other token on the system if it meets the integrity level requirement, even if `SeImpersonatePrivilege` is not available.

Checks that user SIDs are equal. If the primary token's user SID does not equal the impersonation token's user SID, the SRM denies the assignment. Otherwise, it continues making checks. This criterion allows a user to impersonate their own user account but blocks them from impersonating another user unless they have the other user's credentials. When authenticating the other user, LSASS returns an impersonation token with the origin logon ID set to the caller's authentication ID, so the token will pass the previous check, and the user SIDs will never be compared.

Checks for the elevated flag. This check ensures that the caller can't impersonate a more privileged token for the same user. If the impersonation token has the `Elevated` flag set but the primary token does not, the impersonation will be denied. This check wasn't always in place; versions of Windows prior to 10 did not perform this check, so it was possible to impersonate a UAC administrator token if you first reduced the integrity level.

Checks for sandboxing. This check ensures that the caller can't impersonate a less-sandboxed token. To impersonate a lowbox token, the new token must either match the package SID or be a restricted package SID of the primary token; otherwise, impersonation will be denied. No check is made on the list of capabilities. For a restricted token, it's enough that the new token is also a restricted token, even if the list of

restricted SID list is different. The same applies to write-restricted tokens. The SRM has various hardening mechanisms to make it difficult to get ahold of a more privileged sandbox token.

Checks the console session. The final check is whether the console session is session 0 or not. This prevents a user from impersonating a token in session 0, which can grant elevated privileges, such as being able to create global `Section` objects.

You might assume that, if the function denies the assignment, it would return a `STATUS_PRIVILEGE_NOT_HELD` error, but that is not the case: instead, the SRM duplicates the impersonation token as an Identification-level token and assigns it. This means that even if the impersonation assignment fails, the thread can still inspect the properties of the token.

We can check whether you can impersonate a token using the `Test-NtTokenImpersonation` PowerShell command. This command impersonates the token and then reopens it from the thread. It then compares the impersonation level of the original token and the reopened token and returns a Boolean result. In Listing 4-32, we run through a simple example that would fall foul of the integrity level check.

```
PS> $token = Get-NtToken -Duplicate
PS> Test-NtTokenImpersonation $token
True

PS> Set-NtTokenIntegrityLevel -IntegrityLevel Low
PS> Test-NtTokenImpersonation $token
False

PS> Test-NtTokenImpersonation $token -ImpersonationLevel Identification
True
```

Listing 4-32 Checking token impersonation

These checks are quite simple. First, we get a duplicate of the current process token and pass it to `Test-NtTokenImpersonation`. The result is `True`, indicating that we could impersonate the token at Impersonation level. For the next check, we lower the integrity level of the current process's

primary token to `Low` and run the test again. (It's best not to run this script in a PowerShell process you care about, as you won't be able to restore the original integrity level.) This check now returns `False`, as it's no longer possible to impersonate the token at the Impersonation level. Finally, we check if we can impersonate the token at the Identification level, which returns `True`.

Worked Examples

Let's walk through some worked examples so you can see how to use the various commands presented in this chapter for security research or systems analysis.

Finding UI Access Processes

It's sometimes useful to enumerate all the processes you can access and check the properties of their primary token. This could help you find processes running as specific users or with certain properties. For example, you could identify processes with the UI access flag set. Earlier in this chapter, we discussed how to check the UI access flag in isolation. In Listing 4-33, we'll perform the check for all processes we can access.

```
PS> $ps = Get-NtProcess -Access QueryLimitedInformation -FilterScript  {
    Use-NtObject($token = Get-NtToken -Process $_ -Access Query) {
    $token.UIAccess
  }
}
PS> $ps
Handle Name       NtTypeName Inherit ProtectFromClose
------ ----       ---------- ------- ----------------
3120   ctfmon.exe Process    False   False
3740   TabTip.exe Process    False   False

PS> $ps.Close()
```

Listing 4-33 Finding processes with UI access

We start by calling the `Get-NtProcess` command to open all processes with `QueryLimitedInformation` access. We

also provide a filter script. If the script returns `True`, the command will return the process; otherwise, it will close the handle to the process.

In the script, we open the process's token for `Query` access and return the `UIAccess` property. The result filters the process list to only processes running with UI access tokens. We display the processes we've found.

Finding Tokens Handles to Impersonate

There are several official ways of getting access to a token to impersonate, such as using an RPC call or opening the process's primary token. However, another approach is to find existing handles to `Token` objects that you can duplicate and use for impersonation.

This technique can be useful if you're running as a user with `SeImpersonatePrivilege` but not as an administrator, as in the case of a service account such as *LOCAL SERVICE*. It could also be used to evaluate the security of a sandbox, to make sure the sandbox can't open and impersonate a more privileged token.

Thirdly, you could use the technique to access another user's resources by waiting for them to connect to the Windows machine, such as over the network. If you grab their token, you can reuse their identity without needing to know their password. Listing 4-34 shows a simple implementation of the idea.

```
function Get-ImpersonationTokens {
 1 $hs = Get-NtHandle -ObjectType Token
    foreach($h in $hs) {
        try {
          2 Use-NtObject($token = Copy-NtObject -Handle $h) {
              3 if (Test-NtTokenImpersonation -Token $token) {
                    Copy-NtObject -Object $token
                }
            }
        } catch {
        }
    }
}
4 PS> $tokens = Get-ImpersonationTokens
5 PS> $tokens | Where-Object Elevated
```

Listing 4-34 Finding elevated `Token` handles to impersonate

We get a list of all handles of type `Token` using the `Get-NtHandle` command 1. Then for each handle, we try to duplicate the handle to the current process using the `Copy-NtObject` command 2. If this succeeds, we test whether we can successfully impersonate the token; if so, we make another copy of the token, so it doesn't get closed 3.

Running the `Get-ImpersonationTokens` function returns all accessible token handles that can be impersonated 4. With these token objects, we can query for properties of interest. For example, we can check whether the token is elevated or not 5, which might indicate that we could use the token to gain additional privileged groups through impersonation.

Removing Administrator Privileges

One thing you might want to do while running a program as an administrator is temporarily drop your privileges so that you can perform some operation without risking damaging the computer, such as accidentally deleting system files. You can use the same approach that UAC uses to create a filtered, lower-privileged token, and use it to perform the operation. Run the code in Listing 4-35 as an administrator.

```
PS> $token = Get-NtToken -Filtered -Flags LuaToken
PS> Set-NtTokenIntegrityLevel Medium -Token $token
w PS> $token.Elevated
False
PS> "Admin" > "$env:windir\admin.txt"
PS> Invoke-NtToken $token { "User" > "$env:windir\user.txt" }
out-file : Access to the path 'C:\WINDOWS\user.txt' is denied.
PS> $token.Close()
```

Listing 4-35 Removing administrator privileges

We start by filtering the current token and specifying the `LuaToken` flag. This flag removes all administrator groups and the "God" privileges described in this chapter. The `LuaToken` flag does not lower the integrity level of the token, so we must set it to `Medium` manually. We can verify the token is no longer

consider an administrator by checked that the `Elevated` property is `False`.

We can now write a file to an administrator-only location, such as the Windows directory. If we first run it under the current process token, it will succeed. However, if we try to perform the same file write while impersonating the token, it will fail with an access denied error. You could also use the token with the `New-Win32Process` PowerShell command to start a new process with the lower-privileged token.

Wrapping Up

In this chapter, we first discussed the two main types of tokens, primary and impersonation. Primary tokens are associated with a process. Impersonation tokens are associated with a thread and allow a process to temporarily impersonate a different user.

For both types of tokens, we discussed their important properties, such as groups, privileges, and integrity levels, and how those properties affect the security identity that the token exposes. We then discussed the two types of sandbox tokens, restricted and lowbox, which applications such as web browsers and document readers use to limit the damage of a potential remote code execution exploit.

Next, we considered how tokens are used to represent administrator privilege, including how Windows implements User Account Control and split-token administrators for normal desktop users. This includes discussing the specifics of what the operating system considers to be an administrator or elevated token.

Finally, we discussed the steps involved in assigning tokens to processes and threads. We defined the specific criteria that need to be met for a normal user to assign a token and how the checks for primary tokens and impersonation tokens differ.

5

SECURITY DESCRIPTORS

In the last chapter, we discussed the security access token, which describes the user's identity to the SRM. In this chapter, you'll learn how security descriptors define a resource's security. A *security descriptor* does several things. It specifies the owner of a resource, allowing the SRM to grant specific rights to users who are accessing their own data. It also contains the *discretionary access control (DAC)* and *mandatory access control (MAC)*, which grant or deny access to users and groups. Finally, it can contain entries that

generate auditing events. Almost every kernel resource has a security descriptor, and user-mode applications can implement their own access control through security descriptors without needing to create a kernel resource.

Understanding the structure of security descriptors is crucial to understanding the security of Windows, as they're used to secure every kernel object and many user-mode components, such as services. You'll even find security descriptors used across network boundaries to secure remote resources. While developing a Windows application or researching Windows security, you'll inevitably have to inspect or create a security descriptor, so having a clear understanding of what a security descriptor contains will save you a lot of time. Let's start this chapter by describing the structure of a security descriptor in more detail.

The Structure of a Security Descriptor

Windows stores security descriptors as binary structures on disk or in memory. While you'll rarely have to manually parse that structure, it's worth understanding what it contains. A security descriptor consists of the following seven components:

- The revision
- Optional resource manager flags
- Control flags
- An optional owner SID
- An optional group SID
- An optional discretionary access control list (DACL)
- An optional system access control list (SACL)

The first component of any security descriptor is the *revision*, which indicates the version of the security descriptor's binary

format. There is only one version, so the revision is always set to the value *1*. Next is an optional set of flags for use by a resource manager. You'll almost certainly never need to use these flags, so I won't document them further.

The following sections of a security descriptor are the *control flags*. The control flags have three uses. First, they define which optional components of the security descriptor are valid. Second, they define how the security descriptors and components were created. Finally, they define how to process the security descriptor when applying it to an object. Table 5-1 shows the list of valid flags and their descriptions.

Table 5-1 Valid Control Flags and Their Descriptions

Name	Value	Description
`OwnerDefaulted`	0x0001	The owner SID was assigned through a default method.
`GroupDefaulted`	0x0002	The group SID was assigned through a default method.
`DaclPresent`	0x0004	A DACL is present in the security descriptor.
`DaclDefaulted`	0x0008	The DACL was assigned through a default method.
`SaclPresent`	0x0010	The SACL is present in the security descriptor.
`SaclDefaulted`	0x0020	The SACL was assigned through a default method.
`DaclUntrusted`	0x0040	When combined with `ServerSecurity`, the DACL is untrusted.
`ServerSecurity`	0x0080	The DACL is replaced with a server ACL (more on the use of this in Chapter 6).
`DaclAutoInheritReq`	0x0100	DACL auto-inheritance for child objects is requested.
`SaclAutoInheritReq`	0x0200	SACL auto-inheritance for child objects is requested.
`DaclAutoInherited`	0x0400	The DACL supports auto-inheritance.
`SaclAutoInherited`	0x0800	The SACL supports auto-inheritance.
`DaclProtected`	0x1000	The DACL is protected from inheritance.
`SaclProtected`	0x2000	The SACL is protected from inheritance.
`RmControlValid`	0x4000	The resource manager flags are valid.
`SelfRelative`	0x8000	The security descriptor is in a relative format.

In the next chapter, we'll cover many of the terms in Table 5-1, such as inheritance, in more detail.

After the control flags comes the owner SID, which represents the owner of the resource. This is typically the user's SID; however, it can also be assigned to a group, such as the administrators SID. Being the owner of a resource grants you certain privileges, including the ability to modify the resource's security descriptor. By allowing the owner to always modify their

resource's security descriptor, the system prevents a user from locking themselves out of their own resources.

The group SID is like the owner SID, but it's rarely used. It exists is primarily to ensure POSIX compatibility in the days when Windows still had a POSIX subsystem, and takes no part in access control for Windows applications.

The most important part of the security descriptor is the discretionary access control list (DACL). The DACL contains a list of *access control entries (ACEs)*, which define what access an SID is given. It's considered *discretionary* because the user or system administrator can choose the level of access granted. There are many different types of ACEs, and we discuss these in "Access Control Lists and Access Control Entries" on page XX. For now, the basic information in each ACE includes the following:

- The SID of the user or group to which the ACE applies
- The type of ACE (for example, *access allowed* or *access denied*)
- The access mask to which the SID will be allowed or denied access

The final component of the security descriptor is the SACL, which stores auditing rules. Like the DACL, it contains a list of ACEs, but rather than determining access based on whether a defined SID matches the current user's, it determines the rules for generating audit events when the resource is accessed. Since Windows Vista, the SACL has also been the preferred location in which to store additional, non-auditing ACEs, such as the resource's mandatory label.

Two final elements to point out in the DACL and SACL are the `DaclPresent` and `SaclPresent` control flags. These flags indicate that the DACL and SACL, respectively, are present in the security descriptor. Using flags rather than just the presence of an ACL allows for the setting of a *NULL ACL*, where the present flag is set but no value has been specified for the ACL

field in the security descriptor. A NULL ACL indicates that no security for that ACL has been defined, and causes the SRM to effectively ignore it. This is distinct from an empty ACL, where the present flag is set and a value for the ACL is specified but the ACL contains no ACEs.

The Structure of a SID

Until now, we've talked about SIDs as opaque binary values or strings of numbers. Let's gain a better understanding of what a SID contains. The diagram in Figure 5-1 shows a SID as it's stored in memory.

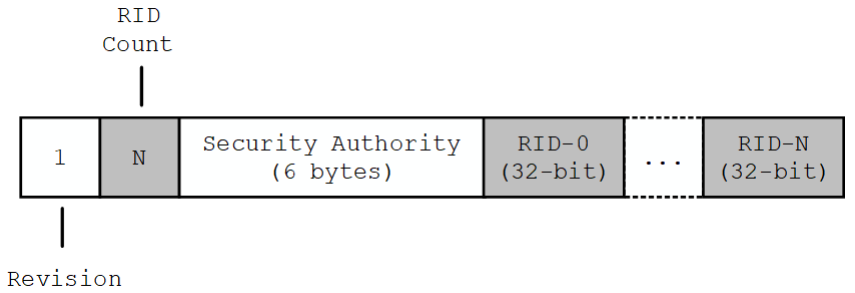

Figure 5-1 The security identifier (SID) structure in memory

There are four components to a binary SID:

The Revision
A value that is always set to 1, as there is no other defined version number

The Relative Identifier (RID) Count
The number of relative identifiers in the SID

The Security Authority
A value representing the party that issued the SID

Relative Identifiers (RIDs)
Zero or more 32-bit numbers that represent the user or group

The security authority can be any value, but Windows has pre-defined some commonly used ones. All well-known authorities start with five zero bytes followed by a value from Table 5-2.

Table 5-2 Well-Known Authorities and Their Values

Name	Final Value	Example name
Null	0	NULL SID
World	1	Everyone
Local	2	CONSOLE LOGON
Creator	3	CREATOR OWNER
Nt	5	BUILTIN\Users
Package	15	APPLICATION PACKAGE AUTHORITY\Your Internet connection
MandatoryLabel	16	Mandatory Label\Medium Mandatory Level
ScopedPolicyId	17	N/A
ProcessTrust	19	TRUST LEVEL\ProtectedLight-Windows

After the security authority come the relative identifiers (RIDs). An SID can contain one or more RIDs. This list of RIDs is usually split into two parts: the domain RIDs, followed by the user RIDs.

Let's walk through how the SID is constructed for a well-known group, *BUILTIN\Users*. When writing the group, we separate the domain component from the group name with a backslash. In this case, the domain is *BUILTIN*. This is a pre-defined domain, represented by a single RID, `32`. Listing 5-1 constructs the domain SID for the *BUILTIN* domain using the `Get-NtSid` PowerShell command.

```
PS> $domain_sid = Get-NtSid -SecurityAuthority Nt -RelativeIdentifier 32
PS> Get-NtSidName $domain_sid
Domain  Name    Source  NameUse Sddl
------  ----    ------  ------- ----
BUILTIN BUILTIN Account Domain  S-1-5-32
```

Listing 5-1 A query for the *BUILTIN* domain SID

The *BUILTIN* domain's SID is a member of the `Nt` security authority. We specify this security authority using the `SecurityAuthority` parameter and specify the single RID using the `RelativeIdentifier` parameter.

We then pass the SID to the `Get-NtSidName` command to retrieve the account name for the SID. The output contains a retrieved domain name and the name of SID. In this case, those values are the same; this is just a quirk of the *BUILTIN* domain's registration.

The next column indicates the location from which the name was retrieved. In this example, the source `Account` indicates that the name was retrieved from *LSASS*. If the source were `WellKnown`, this would indicate that PowerShell knew the name ahead of time and didn't need to query *LSASS*. The fourth column, `NameUse`, indicates the SID's type. In this case, it's `Domain`, which we might have expected. The final column is the SID in its SDDL format.

Any RIDs specified for SIDs following the domain SID identify a particular user or group. For the *Users* group, we use a single RID with the value `545`; Windows pre-defines this number. Listing 5-2 creates a new SID by adding an additional `545` RID to the base domain's SID.

```
PS> $user_sid = Get-NtSid -BaseSid $domain_sid -RelativeIdentifier 545
PS> Get-NtSidName $user_sid
Domain  Name  Source  NameUse Sddl
------  ----  ------  ------- ----
BUILTIN Users Account Alias   S-1-5-32-545

PS> $user_sid.Name
BUILTIN\Users
```

Listing 5-2 Constructing an SID from security authority and RIDs

In the output, we receive the *Users* group inside of the *BUILTIN* domain. The `NameUse` in this case is set to `Alias`. This indicates that the SID represents a local, built-in group, as distinct from `Group`, which represents a user-defined group. When we print the `Name` property on the SID, it outputs the fully qualified name, with the domain and the name separated by a backslash.

You can find lists of known SIDs on the Microsoft Developer Network (MSDN) and other websites. However, Microsoft sometimes add SIDs without documenting them. Therefore, I

encourage you to test multiple security authority and RID values to see what other users and groups you can find. Merely checking for different SIDs won't cause any damage. For example, try replacing the user RID in Listing 5-2 with `544`. This new SID represents the *BUILTIN\Administrators*, as shown in Listing 5-3.

```
PS> Get-NtSid -BaseSid $domain_sid -RelativeIdentifier 32, 544
Name                        Sid
----                        ---
BUILTIN\Administrators      S-1-5-32-544
```

Listing 5-3 Querying the administrators group SID using `Get-NtSid`

Remembering the RIDs for a specific SID can be tricky, and you might not recall the exact name to look up with *LSASS*. Therefore, `Get-NtSid` implements a mode that can create a SID from a known set of SIDs. For example, to create the *administrators* group, you can use the command shown in Listing 5-4.

```
PS> Get-NtSid -KnownSid BuiltinAdministrators
Name                        Sid
----                        ---
BUILTIN\Administrators      S-1-5-32-544
```

Listing 5-4 Querying the known *administrators* group SID using `Get-NtSid`

You'll find SIDs used throughout the Windows operating system. It's crucial that you understand how they're structured, as this will allow you to quickly assess what a SID might represent. For example, if you identify a SID with the `Nt` security authority and its first RID is `32`, you can be sure it's representing a built-in user or group. Knowing the structure also allows you to identify and extract SIDs from crash dumps or memory in cases when better tooling isn't available.

Absolute and Relative Security Descriptors

The kernel supports two security descriptor formats: absolute and relative. Each format has its advantages and disadvantages, and we'll consider both formats so you can understand what those are.

The formats start with the same three values: the revision, the resource manager flags, and the control flags. The `SelfRelative` flag in the control flags determines which format to use, as shown in Figure 5-2.

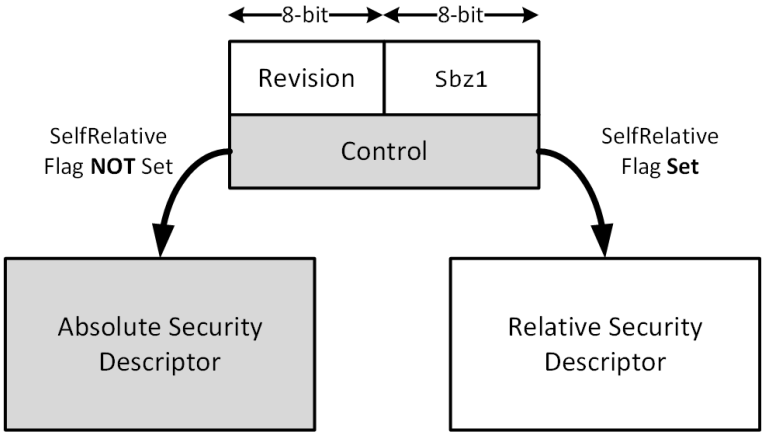

Figure 5-2 Selecting the security descriptor format based on the `SelfRelative` control flag

The total size of the security descriptor's header is 32-bits, split between two 8-bit values, `Revision` and `Sbz1`, and the 16-bit `Control` flags. The security descriptor's resource manager flags are stored in `Sbz1`, and are only valid if the `RmControlValid` control flag is set, although the value will be present in either case. The rest of the security descriptor is then stored immediately after the header.

The simplest format, the absolute security descriptor, is used when the `SelfRelative` flag is not set. After the common header, the absolute format defines four pointers to reference in memory: the owner SID, the group SID, the DACL, and the SACL, in that order, as shown in Figure 5-3.

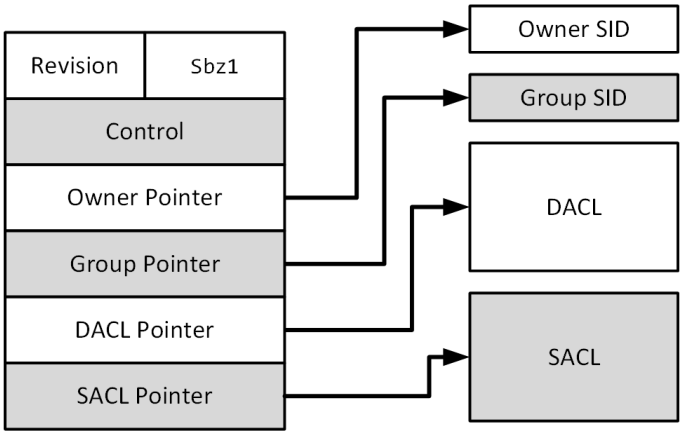

Figure 5-3 The structure of the absolute security descriptor

Each pointer references an absolute memory address at which the data is stored. The size of the pointer therefore depends on whether the application is 32- or 64-bit. It's also possible to specify a NULL value for the pointer to indicate that the value is not present. The owner and group SID values are stored using the binary format we defined in "The Structure of a SID" on page XX.

When the `SelfRelative` flag is set, the security descriptor instead follows the relative format. Instead of referencing its values using absolute memory addresses, relative security descriptor instead stores these locations as positive offsets relative to the start of its header. Figure 5-4 shows an example of a relative security descriptor.

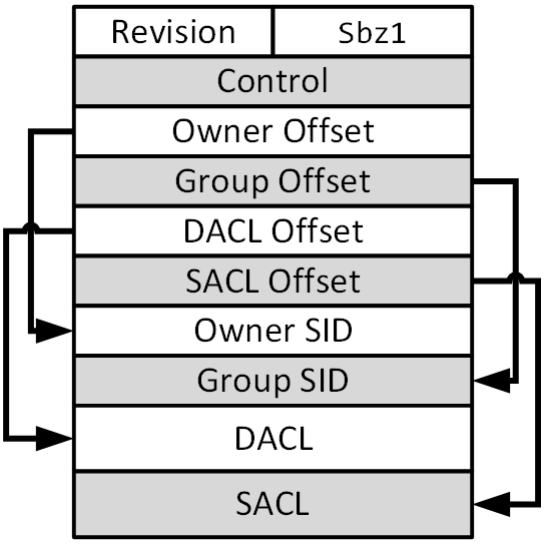

Figure 5-4 An example of a relative security descriptor

These values are stored in contiguous memory. The ACL format is already a relative format and therefore doesn't require any special handling when used in a relative security descriptor. (The next section will specify the details of an ACL's structure.) Each offset is always 32 bits long, regardless of the system's bit size. If an offset is set to 0, the value doesn't exist, as in the case of NULL for an absolute security descriptor.

The main advantage of an absolute security descriptor is that you can easily update its individual components. For example, to replace the `Owner` SID, you'd allocate a new SID in memory and assign its memory address to the `Owner` pointer. In comparison, modifying a relative security descriptor in the same way might require adjusting its allocated memory if the new `Owner` SID structure is larger than the old one.

On the other hand, the big advantage of the relative security descriptor is that it can be built in a single contiguous block of memory. This allows you to serialize the security descriptor to a persistent format, such as to a file or a registry key. When you're trying to determine the security of a resource, you might need to extract its security descriptor from memory or a persistent store.

By understanding the two formats, you can determine how to read the security descriptor into something you can view or manipulate.

Most APIs and system calls accept either security descriptor format, determining how to handle a security descriptor automatically by checking the value of the `SelfRelative` flag. However, you'll find some exceptions in which an API takes only one format or another; in that case, if you pass the API a security descriptor in the wrong format, you'll typically receive an error such as `STATUS_INVALID_SECURITY_DESCR`. Security descriptors returned from an API will almost always be in relative format due to the simplicity of their memory management. The system provides the APIs `RtlAbsoluteToSelfRelativeSD` and `RtlSelfRelativeToAbsoluteSD` to convert between the two formats if needed.

The PowerShell module handles all security descriptors using a `SecurityDescriptor` object, regardless of format. This object is written in .NET and converts to a relative or absolute security descriptor only when it's required to interact with native code. You can determine whether a `SecurityDescriptor` object was generated from a relative security descriptor by inspecting the `SelfRelative` property.

Access Control Lists and Access Control Entries

The DACL and SACL make up most of data in a security descriptor. While these elements have different purposes, they share the same basic structure. In this section, we cover how they're arranged in memory, leaving the details of how they contribute to the access-checking process to Chapter 6.

The Header

All ACLs consists of an ACL header followed by a list of zero or more ACEs in one contiguous block of memory. Figure 5-5 shows this top-level format.

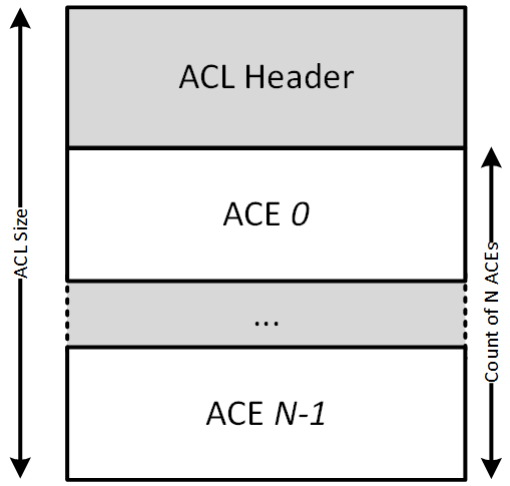

Figure 5-5 A top-level overview of the ACL structure

The ACL header contains a revision, the total size of the ACL in bytes, and the number of ACE entries that follow the header. Figure 5-6 shows the header structure.

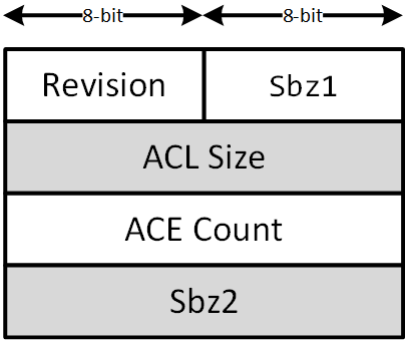

Figure 5-6 The ACL header

The ACL header also contains two reserved fields, `Sbz1` and `Sbz2`, which should always be zero. They serve no purpose in modern versions of Windows and are there in case the ACL structure needs to be extended. Currently, the revision field can have one of three values; these determine the ACL's valid ACEs. If the ACL uses an ACE that the revision doesn't support, the ACL won't be considered valid. Windows supports the following revisions:

Revision 2
The lowest currently supported ACL revision. Supports all the basic ACE types, such as `Allowed` and `Denied`.

Revision 3
Adds support for compound ACEs.

Revision 4
Adds support for object ACEs.

The ACE List

Following the ACL header is the list of ACEs, which determines what access the SID has. Each ACE has a variable length but always starts with a header that contains the type of ACE, additional flags, and the ACE's total size. The header is followed by data specific to the ACE type. Figure 5-7 shows this structure.

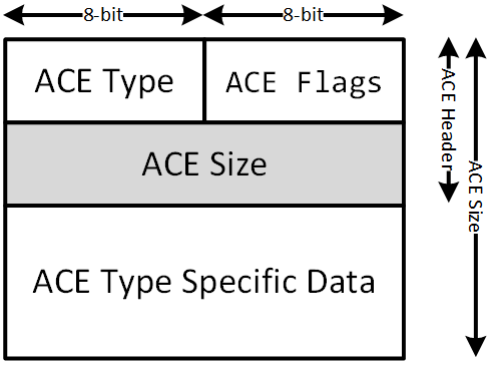

Figure 5-7 The ACE header

The ACE header is common to all ACE types. This allows an application to safely access the header when processing an ACL. The ACE type value can then be used to determine the exact format of the ACE's type-specific data. If the application doesn't understand the particular ACE type, it can use the size field to skip the ACE entirely. We'll discuss how types affect access checking in Chapter 7. Windows supports the ACE types shown

in Table 5-3, which included the ACL locations for which they're valid.

Table 5-3 Names of ACE Types, Their Minimum ACL Revisions, and Their Locations

ACE Type	Value	ACL	Description
`Allowed`	0x0	DACL	Grants access to a resource
`Denied`	0x1	DACL	Denies access to a resource
`Audit`	0x2	SACL	Audits access to a resource
`Alarm`	0x3	SACL	Alarms upon access to a resource; unused
`AllowedCompound`	0x4	DACL	Grants access to a resource during impersonation
`AllowedObject`	0x5	DACL	Grants access to a resource with an object type
`DeniedObject`	0x6	DACL	Denies access to a resource with an object type
`AuditObject`	0x7	SACL	Audits access to a resource with an object type
`AlarmObject`	0x8	SACL	Alarms with an object type; unused
`AllowedCallback`	0x9	DACL	Grants access to a resource with a callback
`DeniedCallback`	0xA	DACL	Denies access to a resource with a callback
`AllowedCallbackObject`	0xB	DACL	Grants access with a callback and an object type
`DeniedCallbackObject`	0xC	DACL	Denies access with a callback and an object type
`AuditCallback`	0xD	SACL	Audits access with a callback
`AlarmCallback`	0xE	SACL	Alarms access with a callback; unused
`AuditCallbackObject`	0xF	SACL	Audits access with a callback and an object type
`AlarmCallbackObject`	0x10	SACL	Alarms access with a callback and an object type; unused
`MandatoryLabel`	0x11	SACL	Specifies the mandatory label/integrity level
`ResourceAttribute`	0x12	SACL	Specifies attributes for the resource
`ScopedPolicyId`	0x13	SACL	Specifies a central access policy ID for the resource
`ProcessTrustLabel`	0x14	SACL	Specifies a process trust label to limit resource access
`AccessFilter`	0x15	SACL	Specifies an access filter for the resource

While Windows officially supports all of these ACE types, the kernel does not use the alarm types. User applications can specify their own ACE types, but various APIs in user and kernel mode check for valid types and will generate an error if the ACE type isn't known.

An ACE's type-specific data falls primary into one of three formats: normal ACEs, such as allow and deny; compound ACEs; and object ACEs. A *normal ACE* contains the following fields after the header, with the field's size is indicated in parentheses:

Access Mask (32-bit)
The access mask to be granted or denied based on the ACE type

SID (Variable)
The SID, in the binary format described earlier in this chapter

System services use the *compound ACE* during impersonation, and the only valid type for this ACE is `AllowedCompound`. The ACE can grant access to both the impersonated caller and the service user at the same time. Even though the latest version of Windows still supports compound ACEs, they're effectively undocumented and presumably deprecated. I've included them in this book for completeness. Their format is as follows:

Access Mask (32-bit)
The access mask to be granted

Compound ACE Type (16-bit)
Set to 1, which means the ACE is used for impersonation

Reserved (16-bit)
Always 0

Server SID (Variable)
The server SID in binary format; matches the service user

SID (Variable)
The SID in a binary format; matches the impersonated user

Microsoft introduced the *object ACE* format to support access control for Active Directory Services. Active Directory uses a 128-bit *globally unique ID (GUID)* to represent a Directory Services object type; the object ACE determines access for specific types of objects, such as computers or users. For example, using a single security descriptor, a directory could

grant a SID the access needed to create a computer object but not access to a user. The object ACE format is as follows:

Access Mask (32-bit)
The access mask to be granted or denied based on the ACE type

Flags (32-bit)
Indicate which of the following GUIDs are present

Object type (16-bytes)
The object type GUID; present only if the flag in bit 0 is set

Inherited object type (16-bytes)
The inherited object GUID; present only if the flag in bit 1 is set

SID (Variable)
The SID in a binary format

ACEs can be larger than their types' defined structure, and may use additional space to stored unstructured data. Most commonly, they use this unstructured data for the callback ACE types, such as `AllowedCallback`, which define a conditional expression that determines whether the ACE should be active during an access check. We can inspect the data that would be generated from a conditional expression using the `ConvertFrom-NtAceCondition` PowerShell command, shown in Listing 5-5.

```
PS> ConvertFrom-NtAceCondition 'WIN://TokenId == "XYZ"' | Out-HexDump -ShowAll
          00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  - 0123456789ABCDEF
-------------------------------------------------------------------------------
00000000: 61 72 74 78 F8 1A 00 00 00 57 00 49 00 4E 00 3A  - artx.....W.I.N.:
00000010: 00 2F 00 2F 00 54 00 6F 00 6B 00 65 00 6E 00 49  - ././.T.o.k.e.n.I
00000020: 00 64 00 10 06 00 00 00 58 00 59 00 5A 00 80 00  - .d......X.Y.Z...
```

Listing 5-5 Parsing a conditional expression and displaying binary data

We refer to these ACEs as *callback ACEs* because, prior to Windows 8, an application needed to call the `AuthzAccessCheck` API to handle them. The API accepted a callback function that would be invoked to determine whether to include a callback ACE in the access check. Since Windows 8, the kernel access check has built-in support for conditional ACEs

in the format shown in Listing 5-5, although a user application is free to specify their own format and handle these ACEs manually.

The primary use of the ACE flags is to specify inheritance rules for the ACE. The inheritance flags take up only the lower five bits leaving the top three bits for ACE-specific flags. Table 5-4 shows the defined ACE flags.

Table 5-4 ACE flags with values and descriptions

ACE Flag	Value	Description
`ObjectInherit`	0x1	The ACE can be inherited by an object.
`ContainerInherit`	0x2	The ACE can be inherited by a container.
`NoPropagateInherit`	0x4	The ACE's inheritance flags are not propagated to children.
`InheritOnly`	0x8	The ACE is used only for inheritance, and not for access checks.
`Inherited`	0x10	The ACE was inherited from a parent container.
`Critical`	0x20	The ACE is critical and can't be removed. Applies only to allow ACEs.
`SuccessfulAccess`	0x40	An audit event should be generated for a successful access.
`FailedAccess`	0x80	An audit event should be generated for a failed access.
`TrustProtected`	0x40	When used with an `AccessFilter` ACE, this flag prevents modification.

Constructing and Viewing Security Descriptors

Now that we've described the structure of a security descriptor, let's construct and manipulate them using PowerShell. By far the most common reason to construct and manipulate security descriptors is to view its contents so you can understand the access is applied to a resource. Another important use case is if you need to construct a security descriptor to lock down a resource. The PowerShell module used in this book aims to make constructing and viewing security descriptors as simple as possible.

Creating a New Security Descriptor

To create a new security descriptor, you can use the `New-NtSecurityDescriptor` command. By default, it creates a new `SecurityDescriptor` object with no owner, group,

DACL, or SACL set. You can use the command's parameters to add these parts of the security descriptor, as shown in Listing 5-6.

```
PS> $world = Get-NtSid -KnownSid World
PS> $sd = New-NtSecurityDescriptor -Owner $world -Group $world -Type File
PS> $sd | Format-Table
Owner    DACL ACE Count SACL ACE Count Integrity Level
-----    -------------- -------------- ---------------
Everyone NONE           NONE           NONE
```

Listing 5-6 Creating a new security descriptor with a specified owner

We first get a SID object for the *World* group to use as the `Owner` and `Group` fields. When calling `New-NtSecurityDescriptor` to create a new security descriptor, we use this SID object to specify its `Owner` and `Group`. We also specify the name of the kernel object type this security descriptor will be associated with; this step makes some of the later commands easier to use. In this case, we'll assume it's a `File` object's security descriptor.

We then format the security descriptor as a table. In the output, we can see the `Owner` field is set to `Everyone`. The `Group` value isn't printed by default, as it's not as important. Neither the DACL not the SACL are currently present in the security descriptor, and there is no integrity level specified.

To add some ACEs, we can use the `Add-NtSecurityDescriptorAce` command. For normal ACEs, we need to specify the ACE type, the SID, and the access mask. Optionally you can also specify the ACE flags. The script in Listing 5-7 adds some ACEs to our new security descriptor.

```
1 PS> $user = Get-NtSid
2 PS> Add-NtSecurityDescriptorAce $sd -Sid $user -Access WriteData, ReadData
  PS> Add-NtSecurityDescriptorAce $sd -KnownSid Anonymous -Access GenericAll
  -Type Denied
  PS> Add-NtSecurityDescriptorAce $sd -Name "Everyone" -Access ReadData

3 PS> Add-NtSecurityDescriptorAce $sd -KnownSid World -Access Delete
  -Type Audit -Flags FailedAccess
4 PS> Set-NtSecurityDescriptorIntegrityLevel $sd Low

5 PS> Set-NtSecurityDescriptorControl $sd DaclAutoInherited, SaclProtected

6 PS> $sd | Format-Table
```

```
Owner    DACL ACE Count SACL ACE Count Integrity Level
-----    -------------- -------------- ---------------
Everyone 3              2              Low

PS> Get-NtSecurityDescriptorControl $sd
DaclPresent, SaclPresent, DaclAutoInherited, SaclProtected

PS> Get-NtSecurityDescriptorDacl $sd | Format-Table
Type    User                          Flags Mask
----    ----                          ----- ----
Allowed GRAPHITE\user                 None  00000003
Denied  NT AUTHORITY\ANONYMOUS LOGON  None  10000000
Allowed Everyone                      None  00000001

PS> Get-NtSecurityDescriptorSacl $sd | Format-Table
Type           User                                Flags        Mask
----           ----                                -----        ----
Audit          Everyone                            FailedAccess 00010000
MandatoryLabel Mandatory Label\Low Mandatory Level None         00000001
```

Listing 5-7 Adding ACEs to the new security descriptor

We start by getting the SID of the current user using `Get-NtSid` 1. We use this SID to add a new `Allowed` ACE 2 to the DACL. We also add a `Denied` ACE for the anonymous user by specifying the `Type` parameter. Finally, we add another `Allowed` ACE for the *Everyone* group. We then modify the SACL to add an audit ACE 3 and set the mandatory label to a low integrity level 4. To finish creating the security descriptor, we set the `DaclAutoInherited` and `SaclProtected` control flags.

We can now print details about the security descriptor we've just created. When displaying the security descriptor itself, we can see that the DACL now contains three ACEs, the SACL `2`, and an integrity level of `Low` 6. We also display the control flags 7, and the list of ACEs in the DACL 8 and SACL 9.

Ordering the ACEs

Because of how access checking works, there is a canonical ordering to the ACEs in an ACL. All `Denied` ACEs should come before `Allowed` ACEs; otherwise, the system might grant access to a resource based on which ACEs come first. The SRM doesn't enforce this canonical ordering; it trusts that any

application has correctly ordered the ACEs before passing them for an access check. A canonical ACE orders its ACEs according to the following rules:

1. All `Denied`-type ACEs must come before `Allowed` types.
2. The `Allowed` ACEs must come before `Allowed` object ACEs.
3. The `Denied` ACEs must come before `Denied` object ACEs.
4. All non-inherited ACEs must come before ACEs with the `Inherited` flag set.

In Listing 5-7, we added a `Denied` ACE to the DACL after we added an `Allowed` ACE, failing the second order rule. We can ensure the DACL is canonicalized by using the `Edit-NtSecurity` command with the `CanonicalizeDacl` parameter. You can also test whether it's already canonical by using the `Test-NtSecurityDescriptor` PowerShell command with the `DaclCanonical` parameter, as in Listing 5-8.

```
PS> Test-NtSecurityDescriptor $sd -DaclCanonical
False
PS> Edit-NtSecurityDescriptor $sd -CanonicalizeDacl
PS> Test-NtSecurityDescriptor $sd -DaclCanonical
True
PS> Get-NtSecurityDescriptorDacl $sd | Format-Table
Type    User                          Flags Mask
----    ----                          ----- ----
Denied  NT AUTHORITY\ANONYMOUS LOGON None  10000000
Allowed GRAPHITE\user                 None  00000003
Allowed Everyone                      None  00000001
```

Listing 5-8 Canonicalizing the DACL

If you compare the list of ACEs in Listing 5-8 with the list in Listing 5-7, you'll notice that the `Denied` ACE has been moved from the middle to the start of the ACL. This ensures that it will be processed before any `Allowed` ACEs.

Formatting Security Descriptors

We can print the values in the security descriptor manually, though the `Format-Table` command, but this is time consuming. Another problem with manual formatting is that the access masks won't be decoded, so instead of `ReadData`, for example, you'll see `00000001`. It would be nice to have a simple way of printing out the details of a security descriptor and formatting them based on the object type. That's what `Format-NtSecurityDescriptor` is for. You can pass it a security descriptor, and the command will print it to the console. Listing 5-9 provides an example.

```
PS> Format-NtSecurityDescriptor $sd -ShowAll
Type: File
Control: DaclPresent, SaclPresent

<Owner>
 - Name  : Everyone
 - Sid   : S-1-1-0

<Group>
 - Name  : Everyone
 - Sid   : S-1-1-0

<DACL> (Auto Inherited)
- Type  : Denied
 - Name  : NT AUTHORITY\ANONYMOUS LOGON
 - SID   : S-1-5-7
 - Mask  : 0x10000000
 - Access: GenericAll
 - Flags : None

 - Type  : Allowed
 - Name  : GRAPHITE\user
 - SID   : S-1-5-21-2318445812-3516008893-216915059-1002
 - Mask  : 0x00000003
 - Access: ReadData|WriteData
 - Flags : None

 - Type  : Allowed
 - Name  : Everyone
 - SID   : S-1-1-0
 - Mask  : 0x00000001
 - Access: ReadData
 - Flags : None

<SACL> (Protected)
 - Type  : Audit
```

```
  - Name  : Everyone
  - SID   : S-1-1-0
  - Mask  : 0x00010000
  - Access: Delete
  - Flags : FailedAccess

<Mandatory Label>
  - Type  : MandatoryLabel
  - Name  : Mandatory Label\Low Mandatory Level
  - SID   : S-1-16-4096
  - Mask  : 0x00000001
  - Policy: NoWriteUp
  - Flags : None
```

Listing 5-9 Displaying the security descriptor using `Format-NtSecurityDescriptor`

First, we call `Format-NtSecurityDescriptor`, passing it the opened security descriptor we generated `1`. We also pass the `ShowAll` parameter to ensure we display the entire contents of the security descriptor; by default, `Format-NtSecurityDescriptor` won't output the SACL or less common ACEs, such as `ResourceAttribute`. Note that the output kernel object type matches the `File` type we specified when creating the security descriptor in Listing 5-6. Specifying the kernel object type allows the formatter to print the decoded access mask for the type rather than a generic hex value.

The next line in the output shows the current control flags. These control flags are calculated on the fly based on the current state of the security descriptor; later, we'll discuss how to change these flags to change the security descriptor's behavior. Next come the owner and group SIDs, respectively.

The DACL takes up the majority of the output. Any DACL-specific flags appear next to the header; in this case, these indicate that we set the `DaclAutoInherited` flag. Next, the output prints each of the ACEs in the ACL in order, starting with the type of ACE. Because the command knows the object type, it prints the decoded access for the type, as well as the original access mask in hexadecimal.

Next is the SACL, which shows our single audit ACE as well as the `SaclProtected` flag. The final component shown is the

mandatory label. The access mask for a mandatory label is the mandatory policy, and it's decoded differently from the rest of the ACEs that use the type-specific access rights. The mandatory policy can be set to one or more of the bit flags shown in Table 5-5.

Table 5-5 Mandatory Policy Values

Name	Value	Description
NoWriteUp	0x00000001	A lower-integrity-level caller can't write to this resource.
NoReadUp	0x00000002	A lower-integrity-level caller can't read this resource.
NoExecuteUp	0x00000004	A lower-integrity-level caller can't execute this resource.

By default, `Format-NtSecurityDescriptor` can be a bit verbose. To shorten its output, specify the `Summary` parameter, which will remove as much data as possible while keeping the important information (Listing 5-10).

```
PS> Format-NtSecurityDescriptor $sd -ShowAll -Summary
<Owner> : Everyone
<Group> : Everyone
<DACL>
<DACL> (Auto Inherited)
NT AUTHORITY\ANONYMOUS LOGON: (Denied)(None)(GenericAll)
GRAPHITE\user: (Allowed)(None)(ReadData|WriteData)
Everyone: (Allowed)(None)(ReadData)
<SACL> (Protected)
Everyone: (Audit)(FailedAccess)(Delete)
<Mandatory Label>
Mandatory Label\Low Mandatory Level: (MandatoryLabel)(None)(NoWriteUp)
```

Listing 5-10 Displaying the security descriptor in summary format

I mentioned in Chapter 2 that the PowerShell module used in this book uses simple names for most common flags, for ease of use. However, you could display the full SDK names if you prefer (for example, to compare the output with native code). To display SDK names, use the `SDKName` property, as shown in Listing 5-11.

```
PS> Format-NtSecurityDescriptor $sd -SDKName -SecurityInformation Dacl
Type: File
Control:
SE_DACL_PRESENT|SE_SACL_PRESENT|SE_DACL_AUTO_INHERITED|SE_SACL_PROTECTED
<DACL> (Auto Inherited)
 - Type  : ACCESS_DENIED_ACE_TYPE
 - Name  : NT AUTHORITY\ANONYMOUS LOGON
```

```
  - SID   : S-1-5-7
  - Mask  : 0x10000000
  - Access: GENERIC_ALL
  - Flags : NONE

  - Type  : ACCESS_ALLOWED_ACE_TYPE
  - Name  : GRAPHITE\user
  - SID   : S-1-5-21-2318445812-3516008893-216915059-1002
  - Mask  : 0x00000003
  - Access: FILE_READ_DATA|FILE_WRITE_DATA
  - Flags : NONE

  - Type  : ACCESS_ALLOWED_ACE_TYPE
  - Name  : Everyone
  - SID   : S-1-1-0
  - Mask  : 0x00000001
  - Access: FILE_READ_DATA
  - Flags : NONE
```

Listing 5-11 Formatting a security descriptor with SDK names

One quirk of `File` objects: their access mask has two naming conventions, one for files and one for directories. You can request that `Format-NtSecurityDescriptor` print the directory version of the access mask by using the `Container` parameter, or more generally, by setting the `Container` property of the security descriptor object to `True`. Listing 5-12 shows the impact of setting the `Container` parameter on the output.

```
PS> Format-NtSecurityDescriptor $sd -ShowAll -Summary -Container
<Owner> : Everyone
<Group> : Everyone
<DACL>
NT AUTHORITY\ANONYMOUS LOGON: (Denied)(None)(GenericAll)
GRAPHITE\user: (Allowed)(None)(ListDirectory|AddFile)
Everyone: (Allowed)(None)(ListDirectory)
--snip--
```

Listing 5-12 Formatting the security descriptor as a container

Note how the output line at 1 has changed from `ReadData|WriteData` to `ListDirectory|AddFile` when we've formatted it as a container. The `File` type is the only object type with this behavior in Windows. This is important to security, as you could easily misinterpret `File` access rights if you formatted the security descriptor for a directory as a file, and vice-versa.

If a GUI is more your thing, you can start a viewer using the following `Show-NtSecurityDescriptor` command:

```
PS> Show-NtSecurityDescriptor $sd
```

Running the command should open the dialog shown in Figure 5-8.

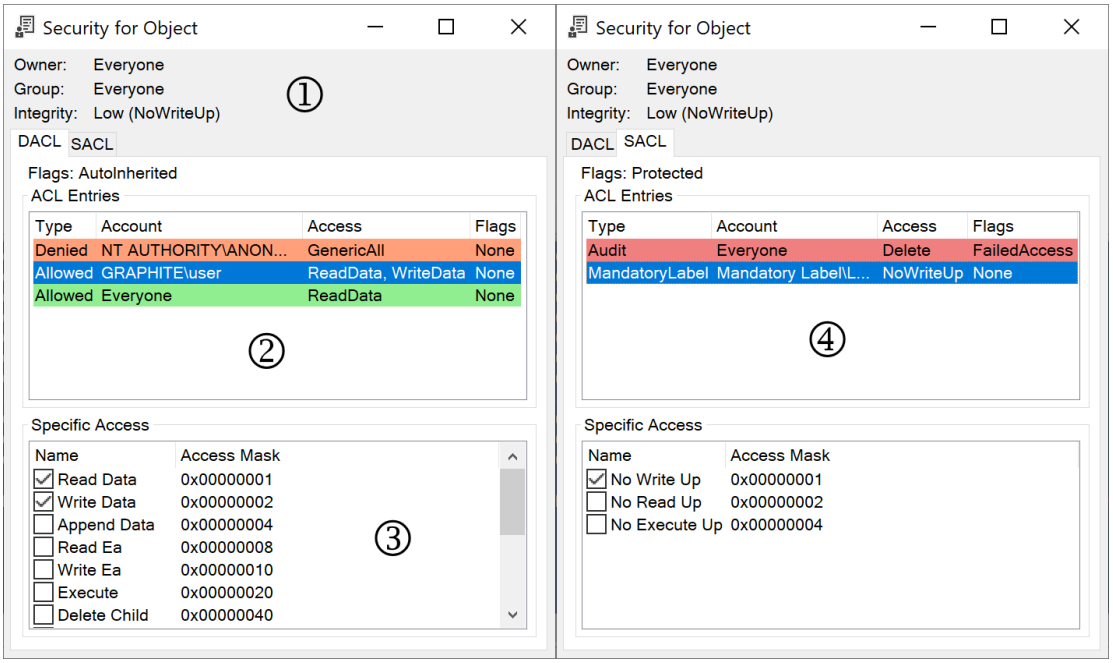

Figure 5-8 A GUI displaying the security descriptor

The dialog summarizes the security descriptor's important data. At the top 1 are the owner and group SIDs resolved into names, as well as the security descriptor's integrity level and mandatory policy. These match the values we specified when creating the security descriptor. In the middle 2 is the list of ACEs in the DACL, with the ACL flags at the top. Each entry in the list includes the type of ACE, the SID, the access mask in generic form and the ACE flags. At the bottom 3 is the decoded access. The list populates when you select an ACE in ACL list. You can also select the SACL by choosing the other tab, where we can see our audit entry, as well the mandatory label 4.

Converting Between Absolute and Relative Security Descriptors

We can convert the security descriptor to a relative security descriptor using the `ConvertFrom-NtSecurityDescriptor` command. We can then format its bytes to the console to see what the underlying structure really is (Listing 5-13).

```
PS> $ba = ConvertFrom-NtSecurityDescriptor $sd
PS> $ba | Out-HexDump -ShowAll
          00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  - 0123456789ABCDEF
-------------------------------------------------------------------------------
00000000: 01 00 14 A4 98 00 00 00 A4 00 00 00 14 00 00 00  - ................
00000010: 44 00 00 00 02 00 30 00 02 00 00 00 02 80 14 00  - D.....0.........
00000020: 00 00 01 00 01 01 00 00 00 00 00 01 00 00 00 00  - ................
00000030: 11 00 14 00 01 00 00 00 01 01 00 00 00 00 00 10  - ................
00000040: 00 10 00 00 02 00 54 00 03 00 00 00 01 00 14 00  - ......T.........
00000050: 00 00 00 10 01 01 00 00 00 00 00 05 07 00 00 00  - ................
00000060: 00 00 24 00 03 00 00 00 01 05 00 00 00 00 00 05  - ..$.............
00000070: 15 00 00 00 F4 AC 30 8A BD 09 92 D1 73 DC ED 0C  - ......0.....s...
00000080: EA 03 00 00 00 00 14 00 01 00 00 00 01 01 00 00  - ................
00000090: 00 00 00 01 00 00 00 00 01 01 00 00 00 00 00 01  - ................
000000A0: 00 00 00 00 01 01 00 00 00 00 00 01 00 00 00 00  - ................
```

Listing 5-13 Converting a security descriptor to a relative security descriptor and displaying its bytes

You can convert the byte array back to an security descriptor object using `New-NtSecurityDescriptor` and the `Byte` parameter:

```
PS> New-NtSecurityDescriptor -Byte $ba
```

As an exercise, I'll leave it to you to pick apart the hex output to find the various structures of the security descriptor based on the descriptions provided in this chapter. To get you started, Figure 5-9 highlights the major structures.

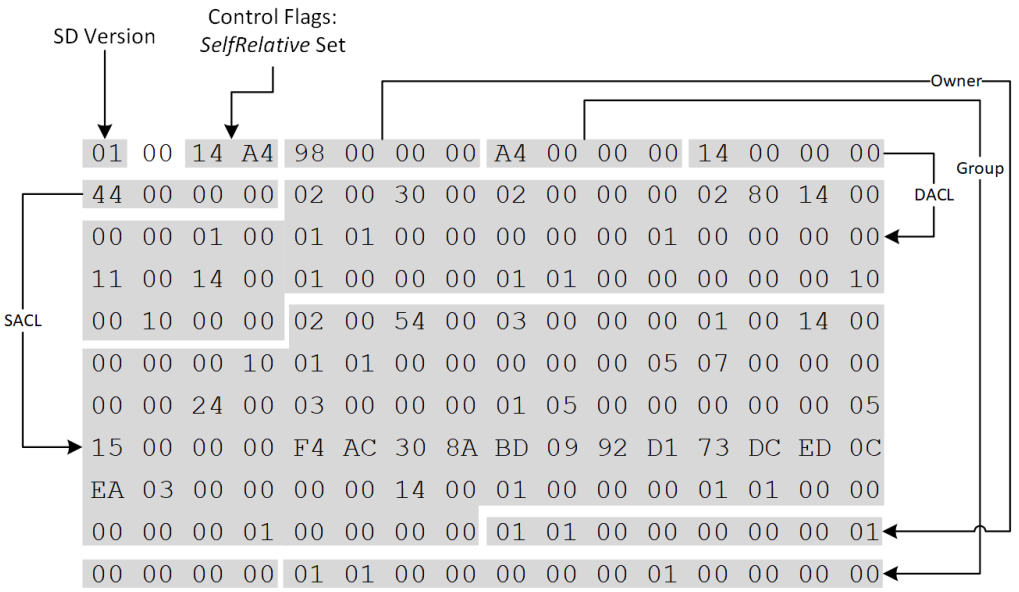

Figure 5-9 An outline of the major structures in the self-relative security descriptor hex output

You'll need to refer to the layout of the ACL and SID structures to manually decode the rest.

The Security Descriptor Definition Language

In Chapter 2, we discussed the basics of the security descriptor definition language (SDDL) format for representing SIDs. The SDDL format can represent the entire security descriptor, too. As the SDDL version of a security descriptor uses ASCII text, it's somewhat human readable and it can be easily copied, unlike the binary data we showed in Listing 5-13. Because it's common to see SDDL strings used throughout Windows, let's describe how to represent a security descriptor in SDDL and how you can read it.

You can convert a security descriptor to the SDDL format by specifying the `ToSddl` parameter to `Format-NtSecurityDescriptor`, as shown in Listing 5-14, where we pass the security descriptor we built in the previous section.

You can also create a security descriptor from an SDDL string using `New-NtSecurityDescriptor` with the `-Sddl` parameter.

```
PS> $sddl = Format-NtSecurityDescriptor $sd -ToSddl -ShowAll
PS> $sddl
O:WDG:WDD:AI(D;;GA;;;AN)(A;;CCDC;;;S-1-5-21-2318445812-3516008893-216915059-
1002)(A;;CC;;;WD)S:P(AU;FA;SD;;;WD)(ML;;NW;;;LW)
```

Listing 5-14 Converting a security descriptor to SDDL

The SDDL version of the security descriptor contains four optional components. You can identity the start of each component by a letter followed by a colon:

O
Owner SID

G
Group SID

D
DACL

S
SACL

In Listing 5-15, we split the output from Listing 5-14 into its components to make it easier to read.

```
PS> $sddl -split "(?=O:)|(?=G:)|(?=D:)|(?=S:)|(?=\()"
O:WD
G:WD
D:AI
  (D;;GA;;;AN)
  (A;;CCDC;;;S-1-5-21-2318445812-3516008893-216915059-1002)
  (A;;CC;;;WD)
S:P
  (AU;FA;SD;;;WD)
  (ML;;NW;;;LW)
```

Listing 5-15 Splitting up the SDDL components

The first two lines represent the owner and group SIDs in SDDL format. You might notice that these don't look like the SDDL SIDs we're used to seeing, as they don't start with `S-1-`.

That is because these strings are two-character aliases that Windows uses for well-known SIDs to reduce the size of an SDDL string. For example, the owner string is `WD`, which we could convert back to the full SID using `Get-NtSid` (Listing 5-16).

```
PS> Get-NtSid -Sddl "WD"
Name     Sid
----     ---
Everyone S-1-1-0
```

Listing 5-16 Converting an alias to a name and SID

As you can see, the `WD` alias represents the *Everyone* group. Table 5-6 shows the aliases for a few well-known SIDs. You can find a more comprehensive list of all supported SDDL aliases in Appendix B.

Table 5-6 Examples of Well-Known SIDs and Their Aliases

SID Alias	Name	SDDL SID
AU	*NT AUTHORITY\Authenticated Users*	S-1-5-11
BA	*BUILTIN\Administrators*	S-1-5-32-544
IU	*NT AUTHORITY\INTERACTIVE*	S-1-5-4
SY	*NT AUTHORITY\SYSTEM*	S-1-5-18
WD	*Everyone*	S-1-1-0

If a SID has no alias, `Format-NtSecurityDescriptor` will emit the SID in the SDDL format, as shown in Listing 5-15. Even SIDs without aliases can have names defined by *LSASS*. For example, the SID in Listing 5-15 belongs to the current user, as shown in Listing 5-17.

```
PS> Get-NtSid -Sddl "S-1-5-21-2318445812-3516008893-216915059-1002" -ToName
GRAPHITE\user
```

Listing 5-17 Looking up the name of the SID

Next in Listing 5-15 is the representation of the DACL. After the `D:` prefix, the ACL in SDDL format looks as follows:

```
ACLFlags(ACE0)(ACE1)…(ACEn)
```

The ACL flags are optional; the DACL's are set to `AI` and the SACL's are set to `P`. These values map to security descriptor control flags and can be one or more of the strings in Table 5-7.

Table 5-7 ACL Flag Strings Mapped to Security Descriptor Control Flags

ACL flag string	DACL control flag	SACL control flag
P	DaclProtected	SaclProtected
AI	DaclAutoInherited	SaclAutoInherited
AR	DaclAutoInheritReq	SaclAutoInheritReq

We'll describe the uses of these three control flags in Chapter 6. Each ACE is enclosed in parentheses and is made up of multiple strings separated by semicolons, following this general format:

```
(Type;Flags;Access;ObjectType;InheritedObjectType;SID[;ExtraData])
```

The `Type` is a short string that maps to an ACE type. Table 5-8 shows these mappings. Note that SDDL format does not support certain ACE types, so they're omitted from the table.

Table 5-8 Mappings of `Type` Strings to ACE Types

ACE type string	ACE type
A	Allowed
D	Denied
AU	Audit
AL	Alarm
OA	AllowedObject
OD	DeniedObject
OU	AuditObject
OL	AlarmObject
XA	AllowedCallback
XD	DeniedCallback
ZA	AllowedCallbackObject
XU	AuditCallback
ML	MandatoryLabel
RA	ResourceAttribute
SP	ScopedPolicyId
TL	ProcessTrustLabel
FL	AccessFilter

The next component is `Flags`, which represents the ACE flags. The audit entry in the SACL from Listing 5-15 shows the flag string `FA`, which represents `FailedAccess`. Table 5-9 shows other mappings.

Table 5-9 Mappings of `Flag` Strings to ACE Flags

ACE flag string	ACE flag
OI	ObjectInherit
CI	ContainerInherit
NP	NoPropagateInherit
IO	InheritOnly
ID	Inherited
CR	Critical
SA	SuccessfulAccess
FA	FailedAccess
TP	TrustProtected

Next is `Access`, which represents the access mask in the ACE. This can be a number in hexadecimal (0x1234), octal (011064), or decimal (4660) format, or a list of short access strings. If no string is specified, the access mask is set to 0. Table 5-10 shows the access strings.

Table 5-10 Mappings of Access Strings to Access Masks

Access string	Access name	Access mask
GR	Generic Read	0x80000000
GW	Generic Write	0x40000000
GX	Generic Execute	0x20000000
GA	Generic All	0x10000000
WO	Write Owner	0x00080000
WD	Write DAC	0x00040000
RC	Read Control	0x00020000
SD	Delete	0x00010000
CR	Control Access	0x00000100
LO	List Object	0x00000080
DT	Delete Tree	0x00000040
WP	Write Property	0x00000020
RP	Read Property	0x00000010
SW	Self Write	0x00000008
LC	List Children	0x00000004
DC	Delete Child	0x00000002
CC	Create Child	0x00000001

Note that the available access strings do not cover the entire access mask range. This is because SDDL was designed to represent the masks for directory service objects, which don't define access mask values outside of a limited range. This is also why the names of the rights are slightly confusing; for example, `Delete Child` does not necessarily map to an arbitrary object type's idea of deleting a child, and you can see in Listing 5-15 that the `File` type's specific access maps to directory service object access, even though it's got nothing to do with Active Directory.

To better support other types, the SDDL format provide access strings for common file and registry key access masks, as shown in Table 5-11.

Table 5-11 Access Strings for File and Registry Key Types

Access string	Access name	Access mask
`FA`	`File All Access`	0x001F01FF
`FX`	`File Execute`	0x001200A0
`FW`	`File Write`	0x00120116
`FR`	`File Read`	0x00120089
`KA`	`Key All Access`	0x000F003F
`KR`	`Key Read`	0x00020019
`KX`	`Key Execute`	0x00020019
`KW`	`Key Write`	0x00020006

If the available access strings can't represent the entire mask, the only option is to represent it as a number. For example, we represent the `ObjectType` and `InheritedObjectType` components, used with object ACEs, as the string forms of GUIDs. The GUIDs can be any value. For example, Table 5-12 contains a few well-known ones used by Active Directory.

Table 5-12 Well-Known Object Type GUIDs Used in Active Directory

GUID	Directory object
19195a5a-6da0-11d0-afd3-00c04fd930c9	Domain
bf967a86-0de6-11d0-a285-00aa003049e2	Computer
bf967aba-0de6-11d0-a285-00aa003049e2	User
bf967a9c-0de6-11d0-a285-00aa003049e2	Group

Here is an example ACE string for an `AllowedObject` ACE with the `ObjectType` set:

```
(OA;;CC;2f097591-a34f-4975-990f-00f0906b07e0;;WD)
```

After the `InheritedObjectType` component in the ACE is the SID. As detailed earlier in this chapter, this can be a short alias if it's a well-known SID, or the full SDDL format if not.

In the final component, which is optional for most ACE types, you can specify a conditional expression if using a callback ACE or a security attribute if using a `ResourceAttribute` ACE. The conditional expression defines a Boolean expression that compares token security attributes values. When evaluated, the result of the expression should be true or false. We saw a simple example in Listing 5-4, `WIN://TokenId == "XYZ"`, which compares the value of the security attribute `WIN://TokenId` with the string value `XYZ`; if they're equal, the expression evaluates to true. The SDDL expression syntax has four different attribute-name formats for the security attribute you want to refer to:

Simple
Used for local security attributes; for example, `WIN://TokenId`

@Device
Used for device claims; for example, `@Device.ABC`

@User
Used for user claims; for example, `@User.XYZ`

@Resource
User for resource attributes; for example, `@Resource.QRS`

The comparison value in the conditional expression can accept several different types, as well. When converting from SDDL to a security descriptor, the condition expression will be parsed, but because the type of the security attribute won't be known at this time, no validation of the value's type can occur. Table 5-13 shows examples for each conditional expression type.

Table 5-13 Example Values for Each Conditional Expression Type

Type	Examples
Number	Decimal: `100`, `-100`, Octal: `0100`, Hexadecimal: `0x100`
String	`"ThisIsAString"`
Fully qualified binary name	`{"O=MICROSOFT CORPORATION, L=REDMOND, S=WASHINGTON",1004}`
SID	`SID(BA)`, `SID(S-1-0-0)`
Octet string	`#0011223344`

The syntax then defines operators to evaluate an expression, starting with unary operators in Table 5-14.

Table 5-14 Unary Operators for Conditional Expressions

Operator	Description
`Exists` *ATTR*	Checks whether the security attribute ATTR exists
`Not_Exists` *ATTR*	Inverse of `Exists`
`Member_of` *{SIDLIST}*	Checks whether the token groups contain all SIDs in list
`Not_Member_of` *{SIDLIST}*	Inverse of `Member_of`
`Device_Member_of` *{SIDLIST}*	Checks whether the token device groups contain all SIDs in list
`Not_Device_Member_of` *{SIDLIST}*	Inverse of `Device_Member_of`
`Member_of_Any` *{SIDLIST}*	Checks whether the token groups contain any SIDs in list
`Not_Member_of_Any` *{SIDLIST}*	Inverse of `Not_Member_of_Any`
`Device_Member_of_Any` *{SIDLIST}*	Checks whether the token device groups contain any SIDs in list
`Not_Device_Member_of_Any` *{SIDLIST}*	Inverse of `Device_Member_of_Any`
`!(`*EXPR*`)`	The logical not of an expression

In Table 5-14, *ATTR* is the name of an attribute to test, *SIDLIST* is a list of SID values enclosed in braces `{}`, and *EXPR* is another conditional sub-expression. Table 5-15 shows the infix operators.

Table 5-15 Infix Operations for Conditional Expressions

Operator	Description
ATTR `Contains` *VALUE*	Checks whether the security attribute contains the value
ATTR `Not_Contains` *VALUE*	Inverse of `Contains`
ATTR `Any_of` {*VALUELIST*}	Checks whether the security attribute contains any of the values

ATTR Not_Any_of *{VALUELIST}*	Inverse of Any_of
ATTR == *VALUE*	Checks whether the security attribute equals the value
ATTR != *VALUE*	Checks whether the security attribute does not equal the value
ATTR < *VALUE*	Checks whether the security attribute is less than the value
ATTR <= *VALUE*	Checks whether the security attribute is less than or equal to the value
ATTR > *VALUE*	Checks whether the security attribute is greater than the value
ATTR >= *VALUE*	Checks whether the security attribute is greater than or equal to the value
EXPR && *EXPR*	Logical AND between two expressions
EXPR || *EXPR*	Logical OR between two expressions

In Table 5-15, *VALUE* can be either a single value from Table 5-13 or a list of values enclosed in braces. The Any_of/Not_Any_of operators work only on lists, and the conditional expression must always be placed in parentheses in the SDDL ACE. For example, if you wanted to use the conditional expression shown back in Listing 5-4 with an AccessCallback ACE, the ACE string would be as follows:

```
(ZA;;GA;;;WD;(WIN://TokenId == "XYZ"))
```

The final component represents a security attribute for the ResourceAttribute ACE. Its general format is as follows:

```
"AttrName",AttrType,AttrFlags,AttrValue(,AttrValue...)
```

The AttrName value is the name of the security attribute, AttrFlags is a hexadecimal number that represents the security attribute flags, and AttrValue is one or more values specific to AttrType separated by commas. The AttrType is a short string that indicates the type of data contained in the security attribute. Table 5-16 shows the strings with examples.

Table 5-16 Security Attribute SDDL Type Strings

Attribute type	Type name	Example Value
TI	Int64	Decimal: 100, -100, Octal: 0100, Hexadecimal: 0x100
TU	UInt64	Decimal: 100, Octal: 0100, Hexadecimal: 0x100

TS	String	"XYZ"
TD	SID	BA, S-1-0-0
TB	Boolean	0, 1
RX	OctetString	#0011223344

To give an example, the following SDDL string represents a ResourceAttribute ACE with the name Classification. It contains two string values, TopSecret and MostSecret, and has the CaseSensitive and NonInheritable flags set:

```
S:(RA;;;;;WD;("Classification",TS,0x3,"TopSecret","MostSecret"))
```

The last field to define in Listing 5-15 is the SACL. The structure is the same as that described for the DACL, although the types of ACEs supported differ. If you try to use an type that is not allowed in the specific ACL, parsing the string will fail. In the SACL example in Listing 5-15, the only ACE is the mandatory label. The mandatory label ACE has its own access strings used to represent the mandatory policy, as shown in the Table 5-17.

Table 5-17 Mandatory Label Access Strings

Access String	Access Name	Access Mask
NX	No Execute Up	0x00000004
NR	No Read Up	0x00000002
NW	No Write Up	0x00000001

The SID represents the integrity level of the mandatory label; again, special SID aliases are defined. Anything outside the list shown in Table 5-18 needs to be represented as a full SID.

Table 5-18 Mandatory Label Integrity Level SIDs

SID Alias	Name	SDDL SID
LW	Low integrity level	S-1-16-4096
ME	Medium integrity level	S-1-16-8192
MP	Medium Plus integrity level	S-1-16-8448
HI	High integrity level	S-1-16-12288
SI	System integrity level	S-1-16-16384

The SDDL format doesn't preserve all information you can store in a security descriptor. For example, the SDDL format can't represent the `OwnerDefaulted` or `GroupDefaulted` control flags, so these are discarded. SDDL also doesn't support some ACEs types, so I omitted these from Table 5-8.

If an unsupported ACE type is encountered while we're converting a security descriptor to SDDL, the conversion process will fail. For this reason, the `ConvertFrom-NtSecurityDescriptor` PowerShell command can convert a security descriptor in relative format to base64. Using base64 preserves the entire security descriptor and allows it to be copied easily, as shown in Listing 5-18.

```
PS> ConvertFrom-NtSecurityDescriptor $sd -AsBase64 -InsertLineBreaks
AQAUpJgAAACkAAAAFAAAAEQAAAACADAAAgAAAAKAFAAAAAEAAQEAAAAAAAEAAAAAEQAUAAEAAAAB
AQAAAAAAEAAQAAACAFQAAwAAAAEAFAAAAAAQAQEAAAAAAAUHAAAAAAAkAAMAAAABBQAAAAAABRUA
AAD0rDCKvQmS0XPc7QzqAwAAAAAUAAEAAAABAQAAAAAAAQAAAAABAQAAAAAAQAAAAABAQAAAAAA
AQAAAAA=
```

Listing 5-18 Converting a security descriptor to a base64 representation

To retrieve the security descriptor, you can pass `New-NtSecurityDescriptor` the `Base64` parameter.

Worked Examples

Let's finish this chapter with some worked examples that use the commands you've learned about in this chapter.

Manually Parsing a Binary SID

The PowerShell module comes with commands you can use to parse SIDs that are structured in various forms. One of those forms is a raw byte array. You can convert an existing SID to a byte array using the `ConvertFrom-NtSid` command:

```
PS> $ba = ConvertFrom-NtSid -Sid "S-1-1-0"
```

You can also convert the byte array back to a SID using the `Byte` parameter on the `Get-NtSid` command, as shown below. The module will parse the byte array and return the SID:

```
PS> Get-NtSid -Byte $ba
```

Although PowerShell can perform these conversions for you, you'll find it valuable to understand how the data is structured at a low level. For example, you might identify code that parses SIDs incorrectly, which could lead to memory corruption; through this discovery, you might find a security vulnerability.

The best way to learn how to parse a binary structure is to write a parser, as we do in Listing 5-19.

```
1 PS> $sid = Get-NtSid -SecurityAuthority Nt -RelativeIdentifier 100, 200, 300
  PS> $ba = ConvertFrom-NtSid -Sid $sid
  PS> $ba | Out-HexDump -ShowAll
            00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  - 0123456789ABCDEF
  ------------------------------------------------------------------------------
  00000000: 01 03 00 00 00 00 00 05 64 00 00 00 C8 00 00 00  - ........d.......
  00000010: 2C 01 00 00                                      - ,...

2 PS> $stm = [System.IO.MemoryStream]::new($ba)
  PS> $reader = [System.IO.BinaryReader]::new($stm)

3 PS> $revision = $reader.ReadByte()
  PS> if ($revision -ne 1) {
      throw "Invalid SID revision"
  }

4 PS> $rid_count = $reader.ReadByte()
5 PS> $auth = $reader.ReadBytes(6)
```

```
PS> if ($auth.Length -ne 6) {
        throw "Invalid security authority length"
    }

PS> $rids = @()
PS> while($rid_count -gt 0) {
        $rids += $reader.ReadUInt32()
        $rid_count--
    }

PS> $new_sid = Get-NtSid -SecurityAuthorityByte $auth -RelativeIdentifier
$rids
PS> $new_sid -eq $sid
True
```

Listing 5-19 Manually parsing a binary SID

For demonstration purposes, we start by creating an arbitrary SID and converting it to a byte array 1. Typically, though, you'll receive a SID to parse in some other way, such as from the memory of a process. We also print the SID as hex. (If you refer to the SID structure shown in Figure 5-1, you might already be able to pick out its various components.)

Next, we create a `BinaryReader` to parse the byte array in a structured form 2. Using the reader, we first check whether the revision value is set to 1 3; if it isn't, we throw an error. Next in the structure is the RID count as a byte 5, followed by the six-byte security authority 5. The `ReadBytes` method can return a short reader, so you'll want to check that you read all six bytes.

We now enter a loop to read the RIDs from the binary structure and append them to an array 6. Next, using the security authority and the RIDs, we can run `Get-NtSid` to construct a new SID object 7 and verify that the new SID matches the one we started with.

This should have given you an example of how to manually parse a SID (or, in fact, any binary structure) using PowerShell. If you're adventurous, you could implement your own parser for the binary security descriptor formats, but that is outside the scope of this book. It's simpler to use the `New-NtSecurityDescriptor` command to do the parsing for you.

Enumerating SIDs

The *LSASS* service does not provide a publicly exposed method for querying every SID-to-name mapping it knows about. While documentation such as MSDN provides a list of known SIDs, these aren't always up to date and won't include the SIDs specific to a computer or enterprise network. However, we can try to enumerate the mappings using brute force. Listing 5-20 defines a function, `Get-AccessSids`, to brute force a list of the SIDs for which *LSASS* has a name.

```
PS> function Get-AccountSids {
    param(
        [parameter(Mandatory)]
        $BaseSid,
        [int]$MinRid = 0,
        [int]$MaxRid = 256
    )

    $i = $MinRid

    while($i -lt $MaxRid) {
        $sid = Get-NtSid -BaseSid $BaseSid -RelativeIdentifier $i
        $name = Get-NtSidName $sid
      1 if ($name.Source -eq "Account") {
            [PSCustomObject]@{
                Sid = $sid;
                Name = $name.QualifiedName;
                Use = $name.NameUse
            }
        }
        $i++
    }
}

2 PS> $sid = Get-NtSid -SecurityAuthority Nt
  PS> Get-AccountSids -BaseSid $sid
  Sid          Name                          Use
  ----         ----                          ---
  S-1-5-1      NT AUTHORITY\DIALUP           WellKnownGroup
  S-1-5-2      NT AUTHORITY\NETWORK          WellKnownGroup
  S-1-5-3      NT AUTHORITY\BATCH            WellKnownGroup
  --snip--

3 PS> $sid = Get-NtSid -BaseSid $sid -RelativeIdentifier 32
  PS> Get-AccountSids -BaseSid $sid -MinRid 512 -MaxRid 1024
  Sid          Name                          Use
  ----         ----                          ---
  S-1-5-32-544 BUILTIN\Administrators        Alias
  S-1-5-32-545 BUILTIN\Users                 Alias
```

```
S-1-5-32-546 BUILTIN\Guests                Alias
--snip--
```

Listing 5-20 Brute forcing known SIDs

The function accepts a base SID and the range of RID values to test. It then creates each SID in the list and queries for its name. If the name's source is `Account`, which indicates the name was retrieved from LSASS, we output the SID's details 1.

To test the function, we call it with the base SID, which contains the `Nt` authority but no RIDs 2. We get the list of retrieved names and SIDs from *LSASS*. Notice that the SIDs in the output are not domain SIDs, as you might expect, but `WellKnownGroup` SIDs. For our purposes, the distinction between `WellKnownGroup`, `Group`, and `Alias` is not important; they're all groups.

We can repeat the brute force with the *BUILTIN* domain SID 3. In this case, we've changed the RID range based on our pre-existing knowledge of the valid range, but you're welcome to try any other range you like. Note that you could automate the search, by inspecting the `NameUse` property in the returned objects and calling `Get-AccountsSids` when its value is `Domain`. I leave this as an exercise for the reader.

Wrapping Up

We started this chapter by delving into the structure of the security descriptor. We detailed its binary structures, such as SIDs, but also its access control lists (ACL) and access control entries (ACE), which make up the discretionary and system ACLs. We then discussed the differences between absolute and relative security descriptors and why the two formats exist.

Next, we described the use of the `New-NtSecurityDescriptor` and `Add-NtSecurityDescriptorAce` commands to create and modify a security descriptor so that it contains whatever entries we require. We also displayed security descriptors in a convenient

form using the `Format-NtSecurityDescriptor` command.

Finally, we covered the SDDL format used for representing security descriptors. We discussed how to represent the various types of security descriptor values, such as ACEs, and how you can write your own. Some tasks we haven't yet covered are how to query a security descriptor from a kernel object and assign a new one. We'll cover these topics in the next chapter.

6

READING AND ASSIGNING SECURITY DESCRIPTORS

In the previous chapter, we discussed the various structures that make up a security descriptor. You also learned how to manipulate security descriptors in PowerShell and how to represent them using the SDDL format. In this chapter, we'll discuss how to read security descriptors from kernel objects, as well as the more complex process of assigning security descriptors to these objects.

Note that we'll focus our discussion on the security descriptors assigned to kernel objects. However, it's possible to

store a security descriptor in persistent storage, such as in a file or as a registry key value. In this case, the security descriptor must be stored in the relative format and read as a stream of bytes before we can convert it into a format we can inspect.

Reading Security Descriptors

To access a kernel object's security descriptor, you can call the `NtQuerySecurityObject` system call. This system call accepts a handle to the kernel object, as well as a set of flags that describe the components of the security descriptor you want to access. The `SecurityInformation` enumeration represents these flags.

Table 6-1 shows the list of available flags on the latest Windows, as well as the location of the information in the security descriptor and the handle access required to query it.

Table 6-1 The `SecurityInformation` Flags and Their Required Access

Flag Name	Description	Location	Handle Access Required
`Owner`	Query the owner SID	Owner	`ReadControl`
`Group`	Query the group SID	Group	`ReadControl`
`Dacl`	Query the DACL	DACL	`ReadControl`
`Sacl`	Query the SACL (auditing ACEs only)	SACL	`AccessSystemSecurity`
`Label`	Query the mandatory label/integrity level	SACL	`ReadControl`
`Attribute`	Query the System Resource Attribute	SACL	`ReadControl`
`Scope`	Query the Scoped Policy ID	SACL	`ReadControl`
`ProcessTrustLabel`	Query the Process Trust Label	SACL	`ReadControl`
`AccessFilter`	Query the Access Filter	SACL	`ReadControl`
`Backup`	Query everything except `ProcessTrustLabel` and `AccessFilter`	All	`ReadControl` and `AccessSystemSecurity`

You need only `ReadControl` access to read most of this information, with the exception of the auditing ACEs from the SACL, which require `AccessSystemSecurity`. (Other ACEs stored in the SACL need only `ReadControl` access.)

The only way to get `AccessSystemSecurity` access is to first enable the `SeSecurityPrivilege` privilege, then

explicitly request the access when opening a kernel object. Listing 6-1 shows this behavior. You must run these commands as an administrator.

```
PS> $dir = Get-NtDirectory "\BaseNamedObjects" -Access AccessSystemSecurity
Get-NtDirectory : (0xC0000061) - A required privilege is not held by the
client.
--snip--

PS> Enable-NtTokenPrivilege SeSecurityPrivilege
PS> $dir = Get-NtDirectory "\BaseNamedObjects" -Access AccessSystemSecurity
PS> $dir.GrantedAccess
AccessSystemSecurity
```

Listing 6-1 Requesting `AccessSystemSecurity` access and enabling `SeSecurityPrivilege`

We first try to open the BNO directory for `AccessSystemSecurity` without having `SeSecurityPrivilege` and receive an error. Next, we enable the `SeSecurityPrivilege` and try again. This time, we successfully open the BNO directory and print its granted access to confirm we've been granted `AccessSystemSecurity`.

It's not entirely clear why the designers of Windows made the decision to guard the reading of audit information with `SeSecurityPrivilege`. While we should consider the modifying or removing of auditing information to be privileged actions, there is no obvious reason that reading that information should be. Unfortunately, we're stuck with this design.

You can query an object' s security descriptor using the `Get-NtSecurityDescriptor` PowerShell command, which calls `NtQuerySecurityObject`. The system call returns the security descriptor in the relative format as a byte array, which the PowerShell command parses into a `SecurityDescriptor` object and returns to the caller. The command accepts either an object or a path to the resource you want to query, as shown in Listing 6-2, which displays the security descriptor of the BNO directory.

```
PS> Use-NtObject($d = Get-NtDirectory "\BaseNamedObjects" -Access ReadControl)
{
   Get-NtSecurityDescriptor -Object $d
```

```
}
Owner                  DACL ACE Count SACL ACE Count Integrity Level
-----                  -------------- -------------- ---------------
BUILTIN\Administrators 4              1              Low
```

Listing 6-2 Querying the security descriptor for BNO directory

We open the BNO directory with `ReadControl` access, then use `Get-NtSecurityDescriptor` to query the security descriptor from the open `Directory` object.

By default, the command queries for the owner, group, DACL, mandatory label, and process trust label. If you want to query any other field (or omit some of the returned information), you need to specify it through the `SecurityInformation` parameter, which accepts the values in Table 6-1. For example, Listing 6-3 uses a path instead of an object and requests only the owner field.

```
PS> Get-NtSecurityDescriptor "\BaseNamedObjects" -SecurityInformation Owner
Owner                  DACL ACE Count SACL ACE Count Integrity Level
-----                  -------------- -------------- ---------------
BUILTIN\Administrators NONE           NONE           NONE
```

Listing 6-3 Querying the owner of the BNO directory

In the output, you can see that only the `Owner` column contains valid information; all other columns now have the value `NONE`, which indicates that no value is present, because we haven't requested that information.

Assigning Security Descriptors

Reading a security descriptor is easy; you just need the correct access to a kernel resource and the ability to parse the relative security descriptor format returned from the `NtQuerySecurityObject` system call. Assigning a security descriptor is a more complex operation. The security descriptor assigned to a resource depends on multiple factors:

- Is the resource being created?

- Has the creator specified a security descriptor during creation?
- Is the new resource stored in a container, such as a file directory or registry key?
- Is the new resource a container or an object?
- What control flags are set on the parent or current security descriptor?
- What user principal is assigning the security descriptor?
- What ACEs does the existing security descriptor contain?
- What kernel object type is being assigned?

As you can see from the list, this process involves many variables and is one of the big reasons Windows security can be so complex.

We can assign a resource's security at creation time or via an open handle. Let's start with the more complex case first: assignment at creation time.

Assigning a Security Descriptor During Resource Creation

When creating a new resource, the kernel needs to assign it a security descriptor. Also, it must store the security descriptor differently depending on the kind of resource being created. For example, object manager resources are ephemeral, so the kernel will store their security descriptors in memory. In contrast, a filesystem driver's security descriptor must be persisted to the disk; otherwise, it will disappear when you reboot your computer.

While the mechanism to store the security descriptor might differ, the kernel must still follow many common procedures when handling it, such as enforcing the rules of inheritance. To provide a consistent implementation, the kernel exports a couple of APIs that calculate the security descriptor to assign to a new resource. The most used of these APIs is

`SeAssignSecurityEx`, which takes the following seven parameters:

Creator Security Descriptor
An optional security descriptor on which to base the new, assigned security descriptor

Parent Security Descriptor
An optional security descriptor for the new resource's parent object

Object Type
An optional GUID that represents the type of object being created

Container
A Boolean value indicating whether the new resource is a container

Auto-Inherit
A set of bit flags that define the auto-inheritance behavior

Token
A handle to the token to use as the creator's identity

Generic Mapping
A mapping from generic access to specific access rights for the kernel type

Based on these parameters, the API calculates a new security descriptor and returns it to the caller. By investigating how these parameters interact, we can understand how the kernel assigns security descriptors to new objects.

Let's consider this assignment process for a `Mutant` object. (This object will delete once the PowerShell instance closes, ensuring that we don't accidentally leave unnecessary files or registry keys around.) Table 6-2 provides an example of how we might set the parameters when creating a new `Mutant` object with `NtCreateMutant`.

Table 6-2 Parameters Passed to `SeAssignSecurityEx` When Creating a `Mutant` Object

Parameter	Setting Value
Creator Security Descriptor	The value of the `SecurityDescriptor` field in the Object Attributes structure
Parent Security Descriptor	The security descriptor of the parent `Directory`; if not set, an unnamed `Mutant`
Object Type	Not set
Container	Set to false, as a `Mutant` isn't a container
Auto-Inherit	Set to `AutoInheritDacl` if the parent security descriptor's control flags have the `DaclAutoInherited` flag and the creator DACL is missing or there is no creator security descriptor Set to `AutoInheritSacl` if the parent security descriptor's control flags have the `SaclAutoInherited` flag and creator SACL is missing or there is no creator security descriptor
Token	If the caller is impersonating, set to an impersonation token; otherwise, the primary token of the caller's process
Generic Mapping	Set to the generic mapping for the `Mutant` type

You might be wondering why the object type isn't set in Table 6-2. Well, the API supports the parameter, but neither the object manager nor NTFS use it. Its primary purpose is to let Active Directory control inheritance, so we'll discuss it separately later in this chapter in "Determining Object Inheritance" on page XX.

Table 6-2 shows only two possible auto-inherit flags, but we can pass many others to the API. Table 6-3 lists the available auto-inherit flags, some of which we'll encounter in this chapter's examples.

Table 6-3 The Auto-Inherit Flags

Flag name	Description
`DaclAutoInherit`	Auto-inherit the DACL
`SaclAutoInherit`	Auto-inherit the SACL
`DefaultDescriptorForObject`	Use the default security descriptor for the new security descriptor
`AvoidPrivilegeCheck`	Don't check for privileges when setting the mandatory label or SACL
`AvoidOwnerCheck`	Avoid checking whether the owner is valid for the current token
`DefaultOwnerFromParent`	Copy the owner SID from the parent security descriptor
`DefaultGroupFromParent`	Copy the group SID from the parent security descriptor
`MaclNoWriteUp`	Auto-inherit the mandatory label with the `NoWriteUp` policy
`MaclNoReadUp`	Auto-inherit the mandatory label with the `NoReadUp` policy

MaclNoExecuteUp	Auto-inherit the mandatory label with the NoExecuteUp policy
AvoidOwnerRestriction	Ignore restrictions placed on the new DACL by the parent security descriptor
ForceUserMode	Enforce all checks as if called from user-mode; only applicable for kernel callers

The most important SeAssignSecurityEx parameters to consider are the values assigned to the parent and creator security descriptors. Let's go through various configurations of these two security descriptor parameters to understand the different outcomes.

Setting Only the Creator Security Descriptor

In the first configuration we'll consider, we call NtCreateMutant with the object attribute's SecurityDescriptor field set to a valid security descriptor. If the new Mutant object is not given a name, it will be created without a parent directory, and the corresponding parent security descriptor won't be set. If there is no parent security descriptor, the auto-inherit flags won't be set, either.

Let's test this behavior to see the security descriptor generated when we create a new Mutant object. Rather than creating the object itself, we'll use the user-mode implementation of SeAssignSecurityEx, which NTDLL exports as RtlNewSecurityObjectEx. We can access RtlNewSecurityObjectEx using the New-NtSecurityDescriptor PowerShell command, as shown in Listing 6-4.

```
PS> $creator = New-NtSecurityDescriptor -Type Mutant
PS> Add-NtSecurityDescriptorAce $creator -Name "Everyone" -Access GenericRead
PS> Format-NtSecurityDescriptor $creator
Type: Mutant
Control: DaclPresent
<DACL>
 - Type  : Allowed
 - Name  : Everyone
 - SID   : S-1-1-0
 - Mask  : 0x80000000
 - Access: GenericRead
 - Flags : None

PS> $token = Get-NtToken -Effective -Pseudo
```

```
3 PS> $sd = New-NtSecurityDescriptor -Token $token -Creator $creator -Type
  Mutant
  PS> Format-NtSecurityDescriptor $sd
  Type: Mutant
  Control: DaclPresent
4 <Owner>
   - Name   : GRAPHITE\user
   - Sid    : S-1-5-21-2318445812-3516008893-216915059-1002

5 <Group>
   - Name   : GRAPHITE\None
   - Sid    : S-1-5-21-2318445812-3516008893-216915059-513

  <DACL>
   - Type   : Allowed
   - Name   : Everyone
   - SID    : S-1-1-0
   - Mask   : 0x00020001
6 - Access: QueryState|ReadControl
   - Flags : None
```

Listing 6-4 Creating a new security descriptor from a creator security descriptor

We build a creator security descriptor with only a single ACE, granting the *Everyone* group `GenericRead` access 1. By formatting the security descriptor 2, we can confirm that only the DACL is present in the formatted output. Next, using the creator security descriptor, we call the `New-NtSecurityDescriptor` command 3, passing the current effective token and specifying the final object type as `Mutant`. This object type determines the generic mapping. Finally, we format the new security descriptor.

You might notice that the security descriptor has changed during the creation process. First, it gained `Owner` 4 and `Group` values 5. Also, the specified access mask has gone from `GenericRead` to `QueryState|ReadControl` 6.

Let's start by considering where those new owner and group values come from. When we don't specify an `Owner` or `Group` value, the creation process copies these from the supplied token's `Owner` and `PrimaryGroup` SIDs. We can confirm this by checking the `Token` object's properties using the `Format-NtToken` PowerShell command (Listing 6-5).

```
PS> Format-NtToken $token -Owner -PrimaryGroup
```

```
OWNER INFORMATION
-----------------
Name              Sid
----              ---
GRAPHITE\user     S-1-5-21-2318445812-3516008893-216915059-1002

PRIMARY GROUP INFORMATION
-------------------------
Name           Sid
----           ---
GRAPHITE\None S-1-5-21-2318445812-3516008893-216915059-513
```

Listing 6-5 Displaying the `Owner` and `PrimaryGroup` SIDs for the current effective token

If you compare the output in Listing 6-5 with the security descriptor values in Listing 6-4, you can see that the owner and group SIDs match.

In Chapter 4, you learned that it's not possible to set an arbitrary owner SID on a token; this value must be either the user's SID or an SID marked with the `Owner` flag. You might wonder: as the token's SID is being used to set the security descriptor's default owner, could we use this behavior to specify an arbitrary owner SID in the security descriptor? Let's check. In Listing 6-6, we first set the security descriptor to the *SYSTEM* user SID, then try to create the security descriptor again.

```
PS> Set-NtSecurityDescriptorOwner $creator -KnownSid LocalSystem
PS> New-NtSecurityDescriptor -Token $token -Creator $creator -Type Mutant
New-NtSecurityDescriptor : (0xC000005A) - Indicates a particular Security ID
may not be assigned as the owner of an object.
```

Listing 6-6 Setting the *SYSTEM* user as the `Mutant` object's security descriptor owner

This time, the creation fails with an exception and the status code `STATUS_INVALID_OWNER`. This is because the API checks whether the owner SID being assigned is valid for the supplied token. This SID doesn't have to be the `Token` object's owner SID, but it must be either the user's SID or a group SID with the `Owner` flag set.

You can set an arbitrary owner SID only when the token used to create the security descriptor has the `SeRestorePrivilege` enabled. Note that this token doesn't necessarily have to belong to the caller of the

`SeAssignSecurityEx` API. You can also disable the owner check by specifying the `AvoidOwnerCheck` auto-inherit flag; however, the kernel will never specify this flag when creating a new object, so it will always enforce the owner check.

This is not to say that there's no way to set a different owner as a normal user. However, any method of setting an arbitrary owner that you discover is a security vulnerability that Microsoft will likely fix. An example of such a bug is CVE-2018-0748, which allowed users to set an arbitrary owner when creating a file. The user had to create the file via a local filesystem share, causing the owner check to be bypassed.

There are no restrictions on the value of the group SID, as the group doesn't contribute to the access check. However, restrictions apply to the SACL. If you specify any audit ACEs in the SACL as part of the creator security descriptor, the kernel will require `SeSecurityPrivilege`.

Remember that, when we created the security descriptor, the access mask went from `GenericRead` to `QueryState|ReadControl`. This is because the security descriptor assignment process maps all generic access rights in the access mask to type-specific access using the object type's generic mapping information. In this case, the `Mutant` type's `GenericRead` mapping converts the access mask to `QueryState|ReadControl`. There is one exception to this rule: if the ACE has the `InheritOnly` flag set, then generic access rights won't be mapped. You'll understand why the exception exists in a moment, when we discuss inheritance.

We can confirm this mapping behavior by using `New-NtSecurityDescriptor` to create an unnamed `Mutant` object, as shown in Listing 6-7.

```
PS> $creator = New-NtSecurityDescriptor -Type Mutant
PS> Add-NtSecurityDescriptorAce $creator -Name "Everyone" -Access GenericRead
PS> Use-NtObject($m = New-NtMutant -SecurityDescriptor $creator){
   Format-NtSecurityDescriptor $m
}
Type: Mutant
Control: DaclPresent
```

```
<Owner>
 - Name   : GRAPHITE\user
 - Sid    : S-1-5-21-2318445812-3516008893-216915059-1002

<Group>
 - Name   : GRAPHITE\None
 - Sid    : S-1-5-21-2318445812-3516008893-216915059-513

<DACL>
 - Type   : Allowed
 - Name   : Everyone
 - SID    : S-1-1-0
 - Mask   : 0x00020001
 - Access: QueryState|ReadControl
 - Flags  : None
```

Listing 6-7 Verifying security descriptor assignment rules by creating a `Mutant` object

As you can see, the output security descriptor is the same as the one created in Listing 6-4.

Setting Neither the Creator Nor the Parent Security Descriptor

Let's explore another simple case. In this scenario, neither the creator nor the parent security descriptor is set. This case corresponds to calling `NtCreateMutant` without a name or a specified `SecurityDescriptor` field. The script to test this case is even simple than the previous, as shown in Listing 6-8.

```
PS> $token = Get-NtToken -Effective -Pseudo
1 PS> $sd = New-NtSecurityDescriptor -Token $token -Type Mutant
PS> Format-NtSecurityDescriptor $sd -HideHeader
2 <Owner>
 - Name   : GRAPHITE\user
 - Sid    : S-1-5-21-2318445812-3516008893-216915059-1002

<Group>
 - Name   : GRAPHITE\None
 - Sid    : S-1-5-21-2318445812-3516008893-216915059-513

3 <DACL>
 - Type   : Allowed
 - Name   : GRAPHITE\user
 - SID    : S-1-5-21-2318445812-3516008893-216915059-1002
 - Mask   : 0x001F0001
 - Access: Full Access
 - Flags  : None
```

```
- Type  : Allowed
- Name  : NT AUTHORITY\SYSTEM
- SID   : S-1-5-18
- Mask  : 0x001F0001
- Access: Full Access
- Flags : None

- Type  : Allowed
- Name  : NT AUTHORITY\LogonSessionId_0_137918
- SID   : S-1-5-5-0-137918
- Mask  : 0x00120001
- Access: QueryState|ReadControl|Synchronize
- Flags : None
```

Listing 6-8 Creating a new security descriptor with no creator or parent security descriptor

This call to `New-NtSecurityDescriptor` requires only the token and kernel object type 1. The `Owner` and `Group` fields in the final security descriptor are set to default values based on the token's `Owner` and `PrimaryGroup` properties 2.

But where did the DACL 3 come from? We haven't specified either a parent or a creator security descriptor, so it couldn't have come from either of those. Instead, the DACL comes from the `Token` object's *default DACL*, an ACL stored in the token that acts as a fallback when there is no other DACL specified. You can display a token's default DACL by passing the token to `Format-NtToken` with the `DefaultDacl` parameter, as in Listing 6-9.

```
PS> Format-NtToken $token -DefaultDacl
DEFAULT DACL
------------
GRAPHITE\user: (Allowed)(None)(GenericAll)
NT AUTHORITY\SYSTEM: (Allowed)(None)(GenericAll)
NT AUTHORITY\LogonSessionId_0_137918:
(Allowed)(None)(GenericExecute|GenericRead)
```

Listing 6-9 Displaying the token's default DACL

Other than its `Mutant`-specific access rights, the DACL in Listing 6-9 matches the one in Listing 6-8. We can conclude that, if we specify neither the parent nor the creator security descriptor during creation, we'll create a new security descriptor based on the token's owner, primary group, and default DACL. However,

just to be certain, let's verify this behavior by creating an unnamed `Mutant` with no security descriptor (Listing 6-10).

```
PS> Use-NtObject($m = New-NtMutant) {
   Format-NtSecurityDescriptor $m
}
Type: Mutant
Control: None
<NO SECURITY INFORMATION>
```

Listing 6-10 Creating an unnamed `Mutant` to verify the default security-descriptor creation behavior

You can see that the new `Mutant` object has no security information at all! This is not what we expected.

The kernel allows certain object types to have no security when the object doesn't have a name. You can learn whether an object requires security by using the `SecurityRequired` property (Listing 6-11).

```
PS> Get-NtType "Mutant" | Select-Object SecurityRequired
SecurityRequired
----------------
           False
```

Listing 6-11 Querying for the `Mutant` type's `SecurityRequired` property

As you can see, the `Mutant` type doesn't require security. So, if we specify neither the creator nor parent security descriptor when creating an unnamed `Mutant` object, the kernel won't generate a default security descriptor.

Why would the kernel support the ability to create an object without a security descriptor? Well, if applications won't share that object with each other, the security descriptor would serve no purpose; it would only use up additional kernel memory. If you assigned the object a name later, the kernel would upgrade it to requiring security so it could be shared.

DUPLICATING UNNAMED OBJECT HANDLES

You can duplicate a handle to the unmanned resource and share it with another process without giving the resource a name. However, this should be done with care. While handle duplication allows you to remove access from a handle if the object has no security descriptor, the receiving process can easily reduplicate the handle to retrieve the access that was removed.

Prior to Windows 8, there was no way assign security to an unnamed object that had

`SecurityRequired` set to `False`. This has changed, and if you specify a security descriptor during creation, you'll assign it to the resulting object. Windows 8 also introduced a new, undocumented flag to `NtDuplicateObject` to separately deal with the issue. Specifying the `NoRightsUpgrade` flag while duplicating a handle tells the kernel to deny any further duplication operations that request additional access rights.

To verify the generation of a default security descriptor, let's now create an object that requires security, such as a `Directory` object (Listing 6-12).

```
PS> Get-NtType Directory | Select-Object SecurityRequired
SecurityRequired
----------------
            True

PS> Use-NtObject($dir = New-NtDirectory) {
   Format-NtSecurityDescriptor $dir -Summary
}
GRAPHITE\user: (Allowed)(None)(Full Access)
NT AUTHORITY\SYSTEM: (Allowed)(None)(Full Access)
NT AUTHORITY\LogonSessionId_0_137918:
(Allowed)(None)(Query|Traverse|ReadControl)
```

Listing 6-12 Creating an unnamed directory to verify the default security descriptor

Listing 6-12 shows that the default security descriptor matches our assumptions.

Setting Only the Parent Security Descriptor

The next case we'll consider is much more complex. Say we call `NtCreateMutant` with a name but without specifying the `SecurityDescriptor` field. Because a named `Mutant` must be live within a `Directory` object (which, as we've just seen, requires security), the parent security descriptor will be set.

Yet when we specify a parent security descriptor, we also bring something else into play: *inheritance*, a process by which the new security descriptor copies a part of the parent security descriptor. Inheritance rules determine which parts of the parent get passed to the new security descriptor, and we call a parent security descriptor whose parts can be inherited *inheritable*.

The purpose of inheritance is to define a hierarchical security configuration for a tree of resources. Without inheritance, we would have to explicitly assign a security descriptor for each new object in the hierarchy, which would become unmanageable rather quickly. It would also makes the resource tree impossible to manage, as each application might choose to behave differently.

Let's test the inheritance rules that apply when we create new kernel resources. We'll focus on the DACL, but these concepts apply to the SACL, as well. To minimize code duplication, Listing 6-13 defines a few functions that run a test with the parent security descriptor and implement various options.

```
PS> function New-ParentSD($AceFlags = 0, $Control = 0) {
  $owner = Get-NtSid -KnownSid BuiltinAdministrators
1 $parent = New-NtSecurityDescriptor -Type Directory -Owner $owner -Group
$owner
2 Add-NtSecurityDescriptorAce $parent -Name "Everyone" -Access GenericAll
  Add-NtSecurityDescriptorAce $parent -Name "Users" -Access GenericAll
-Flags $AceFlags
3 Add-NtSecurityDescriptorControl $parent -Control $Control
4 Edit-NtSecurityDescriptor $parent -MapGeneric
  return $parent
}

PS> function Test-NewSD($AceFlags = 0,
                        $Control = 0,
                        $Creator = $null,
                        [switch]$Container) {
  $parent = New-ParentSD -AceFlags $AceFlags -Control $Control
  Write-Output "-= Parent SD =-"
  Format-NtSecurityDescriptor $parent -Summary

  if ($Creator -ne $null) {
    Write-Output "`r`n-= Creator SD =-"
    Format-NtSecurityDescriptor $creator -Summary
  }

5 $auto_inherit_flags = @()
  if (Test-NtSecurityDescriptor $parent -DaclAutoInherited) {
    $auto_inherit_flags += "DaclAutoInherit"
  }
  if (Test-NtSecurityDescriptor $parent -SaclAutoInherited) {
    $auto_inherit_flags += "SaclAutoInherit"
  }
  if ($auto_inherit_flags.Count -eq 0) {
    $auto_inherit_flags += "None"
  }
```

```
  $token = Get-NtToken -Effective -Pseudo
  $sd = New-NtSecurityDescriptor -Token $token -Parent $parent -Creator
$creator
5 -Type Mutant -Container:$Container -AutoInherit $auto_inherit_flags
  Write-Output "`r`n-= New SD =-"
  Format-NtSecurityDescriptor $sd -Summary
}
```

Listing 6-13 Test function definitions for `New-ParentSD` and `Test-NewSD`

We define two functions, `New-ParentSD` and `Test-NewSD`. The `New-ParentSD` function creates a new security descriptor with the `Owner` and `Group` fields set to the *Administrators* group 1. This will allow us to check for any inheritance of the `Owner` or `Group` fields in any new security descriptor we create from this parent. We also set the `Type` to `Directory`, as expected for the object manager. We add two `Allow` ACEs, one for the `Everyone` group and one for the `Users` group 2, differentiated by their SIDs. We assign both ACEs `GenericAll` access, but add extra flags for the `Users` ACE only.

The function then sets some optional security descriptor control flags 3. Normally. when we assign a security descriptor to a parent, the generic access rights get mapped to type-specific access rights. Here, we use `Edit-NtSecurityDescriptor` with the `MapGeneric` parameter to do this mapping for us 4.

In the `Test-NewSD` function, we create the parent security descriptor and then calculate any auto-inherit flags 5. Finally, we create a new security descriptor, setting the `Container` property if required, as well as the auto-inherit flags we calculated 6. You can specify a creator security descriptor for this function to use to create the new security descriptor. For now, we'll leave this value as `$null`, but we'll come back to it in the next section. We print the parent, the creator (if specified), and new security descriptors to the console to verify the input and output.

Let's start by testing the default case: running the `Test-NewSD` command with no additional parameters. The command

will create a parent security descriptor with no control flags set, so there should be no auto-inherit flags present in the call to `SeAssignSecurityEx` (Listing 6-14).

```
1 PS> Test-NewSD
  -= Parent SD =-
  <DACL>
  Everyone: (Allowed)(None)(Full Access)
  BUILTIN\Users: (Allowed)(None)(Full Access)

  -= New SD =-
2 <Owner> : GRAPHITE\user
  <Group> : GRAPHITE\None
  <DACL>
3 GRAPHITE\user: (Allowed)(None)(Full Access)
  NT AUTHORITY\SYSTEM: (Allowed)(None)(Full Access)
  NT AUTHORITY\LogonSessionId_0_137918:
  (Allowed)(None)(ModifyState|ReadControl|...)
```

Listing 6-14 Creating a new security descriptor with a parent security descriptor and no creator security descriptor

We call `Test-NewSD` with no parameters 1. We can see that the `Owner` and `Group` do not derive from the parent security descriptor 2; instead, they're the defaults we observed earlier in this chapter. This makes sense: the caller, and not the user who created the parent object, should own the new resource.

However, the new DACL doesn't look as we might have expected 3. It's set to the default DACL we saw earlier, and bears no relation to the DACL we built in the parent security descriptor. The reason we didn't get any ACEs from the parent's DACL is that we did not specify the ACEs as inheritable. To do so, you need to set one or both of the `ObjectInherit` and `ContainerInherit` ACE flags. The former flag applies only to non-container objects such as `Mutant` objects, while the latter applies to container objects such as `Directory` objects. The distinction between the two types is important, because they affect how the inherited ACEs propagate to child objects.

The `Mutant` object is a non-container, so let's add the `ObjectInherit` flag to the ACE in the parent security descriptor (Listing 6-15).

```
1 PS> Test-NewSD -AceFlags "ObjectInherit"
```

```
-= Parent SD =-
<Owner> : BUILTIN\Administrators
<Group> : BUILTIN\Administrators
<DACL>
Everyone: (Allowed)(None)(Full Access)
BUILTIN\Users: (Allowed)(ObjectInherit)(Full Access)

-= New SD =-
<Owner> : GRAPHITE\user
<Group> : GRAPHITE\None
<DACL>
BUILTIN\Users:
(Allowed)(None)(ModifyState|Delete|ReadControl|WriteDac|WriteOwner)
```

Listing 6-15 Adding an `ObjectInherit` ACE to the parent security descriptor

We specify the `ObjectInherit` ACE flag to the test function 1. Observe that the `Owner` and `Group` fields have not changed 2 but that the DACL is no longer the default 3. Instead, it contains a single ACE that grants the `Users` group `ModifyState|Delete|ReadControl|WriteDac|WriteOwner` access. This is the ACE that we set to be inherited.

However, you might notice a problem: the parent security descriptor's ACE was granted `Full Access`, while the new security descriptor's ACE is not. Why has the access mask changed? In fact, it hasn't; the inheritance process has merely taken the raw `Directory` access mask for the parent security descriptor's ACE, (the value `0x000F000F`) and copied it to the inherited ACE. A `Mutant` object's valid access bits are `0x001F0001`. Therefore, the inheritance process uses the closest mapping, `0x000F0001`, as shown in Listing 6-16.

```
PS> Get-NtAccessMask (0x0001F0001 -band 0x0000F000F) -ToSpecificAccess Mutant
ModifyState, Delete, ReadControl, WriteDac, WriteOwner
```

Listing 6-16 Checking the inherited access mask

This is a pretty serious issue. Notice, for example, that the `Mutant` type is missing the `Synchronize` access right, which it needs for a caller to wait on the lock. Without this access, the `Mutant` object would be useless to an application.

We can solve this access mask problem by specifying a generic access mask in the ACE. This generic access will map to

a type-specific access when the new security descriptor is created. There is only one complication: we've taken the parent security descriptor from an existing object, so the generic access was already mapped when the security descriptor was assigned. We simulated this behavior in our test function with `Edit-NtSecurityDescriptor` call.

To resolve this issue, the ACE can set the `InheritOnly` flag. As a result, any generic access will remain untouched during the initial assignment. The `InheritOnly` flag marks the ACE for inheritance only, which prevent the generic access from being an issue for access checking. In Listing 6-17, we check this behavior by modifying the call to the test function.

```
1 PS> Test-NewSD -AceFlags "ObjectInherit, InheritOnly"
  -= Parent SD =-
  <Owner> : BUILTIN\Administrators
  <Group> : BUILTIN\Administrators
  <DACL>
  Everyone: (Allowed)(None)(Full Access)
2 BUILTIN\Users: (Allowed)(ObjectInherit, InheritOnly)(GenericAll)

  -= New SD =-
  <Owner> : GRAPHITE\user
  <Group> : GRAPHITE\None
  <DACL>
3 BUILTIN\Users: (Allowed)(None)(Full Access)
```

Listing 6-17 Adding an `InheritOnly` ACE

We change the ACE flags to `ObjectInherit` and `InheritOnly` 1. In the parent security descriptor's output, we can see that the access mask is no longer mapped from `GenericAll` 2. As a result, the inherited ACE is now granted `Full Access`, as we require.

Presumably, the `ContainerInherit` flag works in the same way as `ObjectInherit`, right? Not quite. We test its behavior in Listing 6-18.

```
1 PS> Test-NewSD -AceFlags "ContainerInherit, InheritOnly" -Container
  -= Parent SD =-
  <Owner> : BUILTIN\Administrators
  <Group> : BUILTIN\Administrators
  <DACL>
  Everyone: (Allowed)(None)(Full Access)
```

```
  BUILTIN\Users: (Allowed)(ContainerInherit, InheritOnly)(GenericAll)

  -= New SD =-
  <Owner> : GRAPHITE\user
  <Group> : GRAPHITE\None
  <DACL>
2 BUILTIN\Users: (Allowed)(None)(Full Access)
3 BUILTIN\Users: (Allowed)(ContainerInherit, InheritOnly)(GenericAll)
```

Listing 6-18 Creating a new security descriptor with the `ContainerInherit` flag

We add the `ContainerInherit` and `InheritOnly` flags to the ACE and then pass the function the `Container` parameter 1. Unlike in the `ObjectInherit` case, we now end up with two ACEs in the DACL. The first ACE 2 grants access to the new resource based on the inheritable ACE. The second 3 is a copy of the inheritable ACE, with `GenericAll` access.

NOTE You might wonder how we can create a security descriptor for a `Container` type when we're using the `Mutant` type. The answer is that the API it doesn't care about the final type, as it uses only the generic mapping, but when creating a real `Mutant` object, the kernel would never specify the container flag.

The ACE's automatic propagation is useful, as it allows you to build a hierarchy of containers without needing to manually grant them access rights. However, you might sometimes want to disable this automatic propagation by specifying the `NoPropagateInherit` ACE flag (Listing 6-19).

```
PS> $ace_flags = "ContainerInherit, InheritOnly, NoPropagateInherit"
PS> Test-NewSD -AceFlags $ace_flags -Container
--snip--
-= New SD =-
<Owner> : GRAPHITE\user
<Group> : GRAPHITE\None
<DACL>
1 BUILTIN\Users: (Allowed)(None)(Full Access)
```

Listing 6-19 Using `NoPropagateInherit` to prevent the automatic inheritance of ACE

When we specify this flag, the ACE that grants access to the resource remains present, but the inheritable ACE disappears 1.

Let's try another ACE flag configuration to see what happens to `ObjectInherit` ACEs when they're inherited by a container (Listing 6-20).

```
PS> Test-NewSD -AceFlags "ObjectInherit" -Container
--snip--
-= New SD =-
<Owner> : GRAPHITE\user
<Group> : GRAPHITE\None
<DACL>
1 BUILTIN\Users: (Allowed)(ObjectInherit, InheritOnly)(ModifyState|…)
```

Listing 6-20 Testing the `ObjectInherit` flag on a container

You might not expect the container to inherit the ACE at all, but in fact, it receives the ACE with the `InheritOnly` flag automatically set 1. This allows the container to pass the ACE to non-container child objects.

Table 6-4 summarizes the inheritance rules for container and non-container objects based on the parent ACE flags.

Table 6-4 The Parent ACE Flags and Flags Set on the Inherited ACEs

Parent ACE flags	Non-container object	Container object
`None`	No inheritance	No inheritance
`ObjectInherit`	None	`InheritOnly` `ObjectInherit`
`ContainerInherit`	No inheritance	`ContainerInherit`
`ObjectInherit` `NoPropagateInherit`	None	No inheritance
`ContainerInherit` `NoPropagateInherit`	No inheritance	None
`ContainerInherit` `ObjectInherit`	None	`ContainerInherit` `ObjectInherit`

`ContainerInherit`
`ObjectInherit`
`NoPropagateInherit`
	None	None

Finally, consider *auto-inherit flags*. If you return to Table 6-3, you can see that if the DACL has the `DaclAutoInherited` control flag set, the kernel will pass the `DaclAutoInherit` flag to `SeAssignSecurityEx`, as there is no creator security descriptor. (The SACL has a corresponding `SaclAutoInherit` flag, but we'll focus on the DACL here.)

What does the `DaclAutoInherit` flag do? In Listing 6-21, we perform a test to find out.

```
   PS> $ace_flags = "ObjectInherit, InheritOnly"
1  PS> Test-NewSD -AceFlags $ace_flags -Control "DaclAutoInherited"
   -= Parent SD =-
   <Owner> : BUILTIN\Administrators
   <Group> : BUILTIN\Administrators
2  <DACL> (Auto Inherited)
   Everyone: (Allowed)(None)(Full Access)
   BUILTIN\Users: (Allowed)(ObjectInherit, InheritOnly)(GenericAll)

   -= New SD =-
   <Owner> : GRAPHITE\user
   <Group> : GRAPHITE\None
3  <DACL> (Auto Inherited)
4  BUILTIN\Users: (Allowed)(Inherited)(Full Access)
```

Listing 6-21 Setting the `DaclAutoInherited` control flag in the parent security descriptor

We set the parent security descriptor's control flags to contain the `DaclAutoInherited` flag 1, and confirm that it is set by looking at the formatted DACL 2. As a consequence, the new security descriptor contains flag as well 3. Also, the inherited ACE has the `Inherited` flag 4.

How do the auto-inherit flags differ from the inheritance flags we discussed earlier? Microsoft conserves both inheritance types for compatibility reasons (as it didn't introduce the `Inherited` flag until Windows 2000). From the kernel's perspective, the two types of inheritance are not very different other than determining whether the new security has the `DaclAutoInherited` flag set and whether any inherited ACE gets the `Inherited` flag. But from a user-mode perspective, this inheritance model indicates which parts of the DACL were inherited from a parent security descriptor. That's important information, and various Win32 APIs use it, as we'll discuss in "Win32 APIs" on page XX.

Setting Both the Creator and Parent Security Descriptors

In the final case, we call `NtCreateMutant` with a name and specify the `SecurityDescriptor` field, setting both the

creator and parent security descriptor parameters. To witness the resulting behavior, let's define some test code. Listing 6-22 writes a function to generate a creator security descriptor. We'll reuse the `Test-NewSD` function we wrote earlier to run the test.

```
PS> function New-CreatorSD($AceFlags = 0, $Control = 0, [switch]$NoDacl) {
  1 $creator = New-NtSecurityDescriptor -Type Mutant
  2 if (!$NoDacl) {
  3 Add-NtSecurityDescriptorAce $creator -Name "Network" -Access GenericAll
    Add-NtSecurityDescriptorAce $creator -Name "Interactive"
-Access GenericAll -Flags $AceFlags
  }
  Add-NtSecurityDescriptorControl $creator -Control $Control
  Edit-NtSecurityDescriptor $creator -MapGeneric
  return $creator
}
```

Listing 6-22 The `New-CreatorSD` test function

This function differs from the `New-ParentSD` function created in Listing 6-13 in the following ways: we use the `Mutant` type when creating the security descriptor 1, we allow the caller to not specify a DACL 2, and we set a different SID for the DACL if it is used 3. These changes will allow us to distinguish the parts of a new security descriptor that come from the parent and those that come from the creator.

In some simple cases, the parent security descriptor has no inheritable DACL, and the API follows the same rules it uses when only the creator security descriptor is set. In other words, if the creator specifies the DACL, the new security descriptor will use it. Otherwise, it will use the default DACL.

If the parent security descriptor contains an inheritable DACL, the new security descriptor will inherit it, unless the creator security descriptor also has a DACL. Even an empty or NULL DACL will override the inheritance from the parent.

In Listing 6-23, we verify this behavior.

```
1 PS> $creator = New-CreatorSD -NoDacl
2 PS> Test-NewSD -Creator $creator -AceFlags "ObjectInherit, InheritOnly"
-= Parent SD =-
<Owner> : BUILTIN\Administrators
<Group> : BUILTIN\Administrators
<DACL>
```

```
   Everyone: (Allowed)(None)(Full Access)
3  BUILTIN\Users: (Allowed)(ObjectInherit, InheritOnly)(GenericAll)

   -= Creator SD =-
4  <NO SECURITY INFORMATION>

   -= New SD =-
   <Owner> : GRAPHITE\user
   <Group> : GRAPHITE\None
   <DACL>
5  BUILTIN\Users: (Allowed)(None)(Full Access)
```

Listing 6-23 Testing parent DACL inheritance with no creator DACL

We build a creator security descriptor with no DACL 1, then run the test with an inheritable parent security descriptor 2. In the output, we confirm the inheritable ACE for the *Users* group 3 and that the creator has no DACL set 4. When we create the new security descriptor, it receives the inheritable ACE 5.

Let's also check what happens when we set a creator DACL (Listing 6-24).

```
1  PS> $creator = New-CreatorSD
2  PS> Test-NewSD -Creator $creator -AceFlags "ObjectInherit, InheritOnly"
   -= Parent SD =-
   <Owner> : BUILTIN\Administrators
   <Group> : BUILTIN\Administrators
   <DACL>
   Everyone: (Allowed)(None)(Full Access)
   BUILTIN\Users: (Allowed)(ObjectInherit, InheritOnly)(GenericAll)

   -= Creator SD =-
   <DACL>
   NT AUTHORITY\NETWORK: (Allowed)(None)(Full Access)
   NT AUTHORITY\INTERACTIVE: (Allowed)(None)(Full Access)

   -= New SD =-
   <Owner> : GRAPHITE\user
   <Group> : GRAPHITE\None
   <DACL>
3  NT AUTHORITY\NETWORK: (Allowed)(None)(Full Access)
   NT AUTHORITY\INTERACTIVE: (Allowed)(None)(Full Access)
```

Listing 6-24 Testing the overriding of the parent DACL inheritance by the creator DACL

We build the creator security descriptor with a DACL 1 and keep the same inheritable parent security descriptor as in Listing

6-23 2. In the output, we see that the ACEs from the creator's DACL have been copied to the new security descriptor 3.

The previous two tests haven't specified any auto-inherit flags. If we specify the `DaclAutoInherited` control flag on the parent security descriptor but include no creator DACL, then the inheritance proceeds in the same way as in Listing 6-24, except that it sets the inherited ACE flags.

However, something interesting happens if we specify both a creator DACL and the control flag (Listing 6-25).

```
1 PS> $creator = New-CreatorSD -AceFlags "Inherited"
  PS> Test-NewSD -Creator $creator -AceFlags "ObjectInherit, InheritOnly"
2 -Control "DaclAutoInherited"
  -= Parent SD =-
  <Owner> : BUILTIN\Administrators
  <Group> : BUILTIN\Administrators
  <DACL> (Auto Inherited)
  Everyone: (Allowed)(None)(Full Access)
  BUILTIN\Users: (Allowed)(ObjectInherit, InheritOnly)(GenericAll)

  -= Creator SD =-
  <DACL>
  NT AUTHORITY\NETWORK: (Allowed)(None)(Full Access)
  NT AUTHORITY\INTERACTIVE: (Allowed)(Inherited)(Full Access)

  -= New SD =-
  <Owner> : GRAPHITE\user
  <Group> : GRAPHITE\None
  <DACL> (Auto Inherited)
3 NT AUTHORITY\NETWORK: (Allowed)(None)(Full Access)
4 BUILTIN\Users: (Allowed)(Inherited)(Full Access)
```

Listing 6-25 Testing the parent DACL inheritance when the creator DACL and `DaclAutoInherited` control flag are set

We build a creator security descriptor and set one of the ACE's flags to include the `Inherited` flag 1. Next, we run the test with the `DaclAutoInherited` control flag on the parent security descriptor 2. In the output, notice that there are two ACEs. The first ACE was copied from the creator 3, while the second is the inherited ACE from the parent 4. Figure 6-1 shows this auto-inheritance behavior.

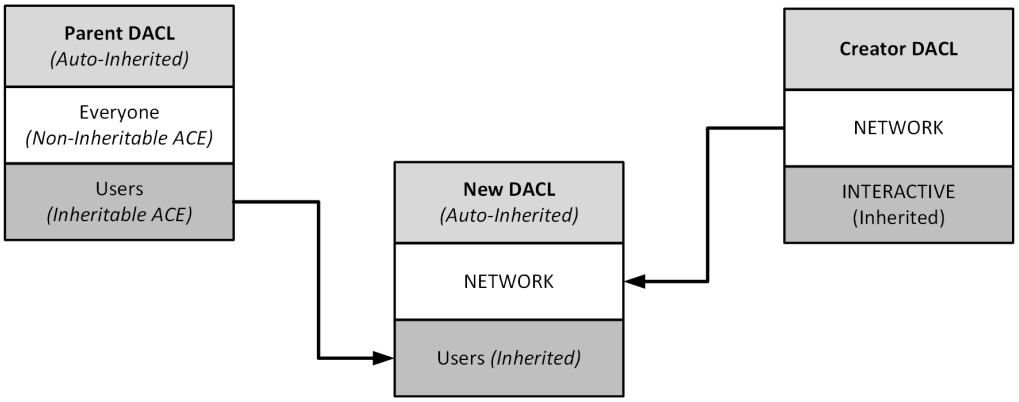

Figure 6-1 The auto-inheritance behavior when the parent and creator security descriptors are both set

When `DaclAutoInherit` is set, the new security descriptor's DACL merges the non-inherited ACEs from the creator security descriptor with the inheritable ACEs from the parent. This auto-inherit behavior allows you to rebuild a child's security descriptor based on its parent without losing any ACEs that the user has explicitly added to the DACL. Additionally, the automatic setting of the `Inherited` ACE flag lets us differentiate between these explicit and inherited ACEs.

Note that normal operations in the kernel do not set the `DaclAutoInherit` flag, which is enabled only if the parent security descriptor has the `DaclAutoInherited` control flag and the DACL isn't present. In our test, we specified a DACL, so the auto-inherit flag was not set. The Win32 APIs use this behavior, as we'll discuss later in this chapter.

If you want to suppress the merging of the explicit ACEs and the parent 's inheritable ACEs, you can set the `DaclProtected` or `SaclProtected` security descriptor control flags. If a protected control flag is set, the inheritance rules leave the DACL alone, other than setting the `AutoInherited` control flag for the ACL and clearing any inherited ACE flags. In Listing 6-26, we test this behavior.

```
1 PS> $creator = New-CreatorSD  -AceFlags "Inherited" -Control "DaclProtected"
  PS> Test-NewSD -Creator $creator -AceFlags "ObjectInherit, InheritOnly"
2 -Control "DaclAutoInherited"
  -= Parent SD =-
```

```
<Owner> : BUILTIN\Administrators
<Group> : BUILTIN\Administrators
<DACL> (Auto Inherited)
Everyone: (Allowed)(None)(Full Access)
BUILTIN\Users: (Allowed)(ObjectInherit, InheritOnly)(GenericAll)

-= Creator SD =-
<DACL> (Protected)
NT AUTHORITY\NETWORK: (Allowed)(None)(Full Access)
NT AUTHORITY\INTERACTIVE: (Allowed)(Inherited)(Full Access)

-= New SD =-
<Owner> : GRAPHITE\user
<Group> : GRAPHITE\None
<DACL> (Protected, Auto Inherited)
NT AUTHORITY\NETWORK: (Allowed)(None)(Full Access)
NT AUTHORITY\INTERACTIVE: (Allowed)(None)(Full Access)
```

Listing 6-26 Testing the `DaclProtected` control flag

We start by generating a creator security descriptor with the `DaclProtected` flag, and set one of the ACE's flags to `Inherited` 1. We then create a new security descriptor with an auto-inherited parent 2. Without the `DaclProtected` flag, the new security descriptor's DACL would have been a merged version of the creator DACL and the inheritable ACEs from the parent. Instead, we see only the creator DACL's ACEs. Also, the `Inherited` flag on the second ACE has been cleared 3.

What if we don't know whether the parent security descriptor will have inheritable ACEs, and we don't want to end up with the default DACL? This might be important for permanent objects, such as files or keys, as the default DACL contains the ephemeral logon SID, which shouldn't really be persisted to disk. After all, reusing the logon SID could end up granting access to an unrelated user.

In this scenario, we can't set a DACL in the creator security descriptor; according to inheritance rules, this would overwrite any inherited ACEs. Instead, we can handle this scenario using the `DaclDefaulted` security descriptor control flag. This flag indicates that the provided DACL is a default, and Listing 6-27 demonstrates its use.

```
PS> $creator = New-CreatorSD -Control "DaclDefaulted"
```

```
PS> Test-NewSD -Creator $creator -AceFlags "ObjectInherit, InheritOnly"
= Parent SD =-
<Owner> : BUILTIN\Administrators
<Group> : BUILTIN\Administrators
<DACL>
Everyone: (Allowed)(None)(Full Access)
BUILTIN\Users: (Allowed)(ObjectInherit, InheritOnly)(GenericAll)

-= Creator SD =-
<DACL> (Defaulted)
NT AUTHORITY\NETWORK: (Allowed)(None)(Full Access)
NT AUTHORITY\INTERACTIVE: (Allowed)(None)(Full Access)

-= New SD =-
<Owner> : GRAPHITE\user
<Group> : GRAPHITE\None
<DACL>
BUILTIN\Users: (Allowed)(None)(Full Access)

PS> Test-NewSD -Creator $creator
-= Parent SD =-
<Owner> : BUILTIN\Administrators
<Group> : BUILTIN\Administrators
<DACL>
Everyone: (Allowed)(None)(Full Access)
BUILTIN\Users: (Allowed)(None)(Full Access)

-= Creator SD =-
<DACL> (Defaulted)
NT AUTHORITY\NETWORK: (Allowed)(None)(Full Access)
NT AUTHORITY\INTERACTIVE: (Allowed)(None)(Full Access)

-= New SD =-
<Owner> : GRAPHITE\user
<Group> : GRAPHITE\None
<DACL>
NT AUTHORITY\NETWORK: (Allowed)(None)(Full Access)
NT AUTHORITY\INTERACTIVE: (Allowed)(None)(Full Access)
```

Listing 6-27 Testing the `DaclDefaulted` flag

If the parent does not contain any inheritable DACL ACEs, the new security descriptor will use creator's DACL instead of the default. If the parent does contain inheritable ACEs, the inheritance process will overwrite the DACL, following as the rules we've already outlined.

To implement similar behavior for the SACL, you can use the `SaclDefaulted` control flag. However, tokens don't contain any default SACL, so this flag is somewhat less important.

Replacing the Creator Owner and Creator Group SIDs

We've seen that, during inheritance, the inherited ACE retains the same SID as the original. In some scenarios, this isn't desirable. For example, you might have a shared directory that allows any user to create a child directory. What security descriptor could you set on this shared directory so that only the creator of the child directory has access to it?

One solution would be to remove all inheritable ACEs. As a result, the new directory would use the default DACL. This would almost certainly secure the directory to prevent other users from accessing it. However, the default DACL is designed for ephemeral resources, such as those in the object manager. Persistent security descriptors shouldn't use it.

To accommodate features such as shared directories, the inheritance implementation supports four special creator SIDs. When a security descriptor inherits an ACE with any of these SIDs, the inheritance implementation will replace the creator SID with a specific SID from the creator's token:

CREATOR OWNER (S-1-3-0)
Replaced by the token's owner

CREATOR GROUP (S-1-3-1)
Replaced by the token's primary group

CREATOR OWNER SERVER (S-1-3-2)
Replaced by the server's owner

CREATOR GROUP SERVER (S-1-3-3)
Replaced by the server's primary group

We use the server SIDs only when creating a server security descriptor, which we'll discuss in "Server Security Descriptors

and Compound ACEs" on page XX. The conversion from the creator SID to a specific SID is a one-way process: once the SID has been replaced, you can't tell it apart from a SID you set explicitly. However, if a container has inherited the ACE, it will keep the creator SID in the `InheritOnly` ACE. Listing 6-28 provides an example.

```
PS> $parent = New-NtSecurityDescriptor -Type Directory
PS> Add-NtSecurityDescriptorAce $parent -KnownSid CreatorOwner
-Flags ContainerInherit, InheritOnly -Access GenericWrite
PS> Add-NtSecurityDescriptorAce $parent -KnownSid CreatorGroup
-Flags ContainerInherit, InheritOnly -Access GenericRead
PS> Format-NtSecurityDescriptor $parent -Summary -SecurityInformation Dacl
<DACL>
1  CREATOR OWNER: (Allowed)(ContainerInherit, InheritOnly)(GenericWrite)
CREATOR GROUP: (Allowed)(ContainerInherit, InheritOnly)(GenericRead)

PS> $token = Get-NtToken -Effective -Pseudo
2 PS> $sd = New-NtSecurityDescriptor -Token $token -Parent $parent
-Type Directory -Container
PS> Format-NtSecurityDescriptor $sd -Summary -SecurityInformation Dacl
<DACL>
3 GRAPHITE\user: (Allowed)(None)(CreateObject|CreateSubDirectory|ReadControl)
CREATOR OWNER: (Allowed)(ContainerInherit, InheritOnly)(GenericWrite)
4 GRAPHITE\None: (Allowed)(None)(Query|Traverse|ReadControl)
CREATOR GROUP: (Allowed)(ContainerInherit, InheritOnly)(GenericRead)
```

Listing 6-28 Testing creator SIDs during inheritance

We add two ACEs with the `CREATOR OWNER` and `CREATOR GROUP` SIDs to a parent security descriptor, and give the ACES different levels of access, to make them easy to distinguish 1. We then create a new security descriptor based on the parent, specifying that we'll use it for a container 2. In the formatted output, we see that the user's SID has replaced the `CREATOR OWNER` SID. This SID is based on the owner SID in the token 3.

As we've created the security descriptor for a container, we also see that there are two `InheritOnly` ACEs whose creator SID has not been changed. This behavior allows the creator SID to propagate to any future children.

Assigning Mandatory Labels

The mandatory label ACE contains the integrity level of a resource. But when we create a new security descriptor using creator's token whose integrity level is greater than or equal to `Medium`, the new security descriptor won't receive a mandatory label by default. This behavior explains why we haven't seen any mandatory label ACE in our tests so far.

If the token's integrity level is less than `Medium`, however, this label is automatically assigned to the new security descriptor, as shown in Listing 6-29.

```
PS> $token = Get-NtToken -Duplicate -IntegrityLevel Low
PS> $sd = New-NtSecurityDescriptor -Token $token -Type Mutant
PS> Format-NtSecurityDescriptor $sd -SecurityInformation Label -Summary
<Mandatory Label>
Mandatory Label\Low Mandatory Level: (MandatoryLabel)(None)(NoWriteUp)
PS> $token.Close()
```

Listing 6-29 Assigning the mandatory label of the creator's token

We duplicate the current token and assign it a `Low` integrity level. When we create a new security descriptor based on the token, we see that it has a mandatory label with the same integrity level.

An application can set a mandatory label ACE explicitly when creating a new resource through the creator security descriptor. However, the integrity level in the mandatory label ACE must be less than or equal to the token's integrity level; otherwise, the creation will fail, as shown in Listing 6-30.

```
PS> $creator = New-NtSecurityDescriptor -Type Mutant
PS> Set-NtSecurityDescriptorIntegrityLevel $creator System
PS> $token = Get-NtToken -Duplicate -IntegrityLevel Medium
 PS> New-NtSecurityDescriptor -Token $token -Creator $creator -Type Mutant
1 New-NtSecurityDescriptor : (0xC0000061) - A required privilege is not held by
  the client.
2 PS> $sd = New-NtSecurityDescriptor -Token $token -Creator $creator -Type
  Mutant
  -AutoInherit AvoidPrivilegeCheck
  PS> Format-NtSecurityDescriptor $sd -SecurityInformation Label -Summary
  <Mandatory Label>
  Mandatory Label\System Mandatory Level: (MandatoryLabel)(None)(NoWriteUp)
  PS> $token.Close()
```

Listing 6-30 Assigning a mandatory label based on the creator security descriptor

We attempt to create a new security descriptor and add a mandatory label with the `System` integrity level to it. We then get the caller's token and set its integrity level to `Medium`. Because the `System` integrity level is greater than `Medium`, creating the new security descriptor fails with a `STATUS_PRIVILEGE_NOT_HELD` error 1.

To set a higher integrity level, you either need to have `SeRelabelPrivilege` enabled on the creator token or specify the `AvoidPrivilegeCheck` auto-inherit flag, as shown at 2. The creation will now succeed, and we can see the mandatory label in the formatted output.

We can make the mandatory label ACE inheritable by setting its `ObjectInherit` or `ContainerInherit` flags. It's also possible to specify its `InheritOnly` flag, which prevents the integrity level from being used as part of an access check, reserving it for inheritance only.

Keep in mind, though, that integrity-level restrictions apply to inherited mandatory label ACEs, too. The inherited ACE must have an integrity level that is less than or equal to the token's; otherwise, the security descriptor assignment will fail. We can bypass this restriction with either the `SeRelabelPrivilege` or the `AvoidPrivilegeCheck` auto-inherit flags. Listing 6-31 shows an example in which a security descriptor inherits the mandatory label ACE.

```
PS> $parent = New-NtSecurityDescriptor -Type Mutant
1 PS> Set-NtSecurityDescriptorIntegrityLevel $parent Low -Flags ObjectInherit
PS> $token = Get-NtToken -Effective -Pseudo
PS> $sd = New-NtSecurityDescriptor -Token $token -Parent $parent -Type Mutant
PS> Format-NtSecurityDescriptor $sd -SecurityInformation Label -Summary
2 <Mandatory Label>
Mandatory Label\Low Mandatory Level: (MandatoryLabel)(Inherited)(NoWriteUp)
```

Listing 6-31 Assigning a mandatory label from a parent security descriptor through inheritance

First, we create a parent security descriptor and assign it a mandatory label ACE with a `Low` integrity level and the `ObjectInherit` flag set 1. We then create a new security

descriptor using the parent. The new security descriptor inherits the mandatory label 2, as indicated by the `Inherited` flag.

Certain kernel object types might receive the mandatory label automatically, even if the caller's token has an integrity level greater than or equal to `Medium`. Using auto-inherit flags, you can also configure various kernel types to use a specific integrity level when you're creating a new security descriptor for the resource. These flags include `MaclNoWriteUp`, `MaclNoReadUp`, and `MaclNoExecuteUp`, which auto-inherit the token's integrity level and set the mandatory policy to `NoWriteUp`, `NoReadUp`, and `NoExecuteUp`, respectively. By combining these flags, you can get the desired mandatory policy.

On the latest version of Windows, only four types are registered to use these auto-inherit flags, as shown in Table 6-5.

Table 6-5 Types with the Integrity-Level Auto-Inherit Flags Enabled

Type name	Auto-inherit flags
`Process`	`MaclNoWriteUp`, `MaclNoReadUp`
`Thread`	`MaclNoWriteUp`, `MaclNoReadUp`
`Job`	`MaclNoWriteUp`
`Token`	`MacNoWriteUp`

We can test the behavior of these auto-inherit flags by specifying them when we create a security descriptor. In Listing 6-32, we specify the `MaclNoReadUp` and `MaclNoWriteUp` auto-inherit flags.

```
PS> $token = Get-NtToken -Effective -Pseudo
PS> $sd = New-NtSecurityDescriptor -Token $token -Type Mutant
-AutoInherit MaclNoReadUp, MaclNoWriteUp
PS> Format-NtSecurityDescriptor $sd -SecurityInformation Label -Summary
<Mandatory Label>
Mandatory Label\Medium Mandatory Level:
(MandatoryLabel)(None)(NoWriteUp|NoReadUp)
```

Listing 6-32 Assigning a mandatory label auto-inherit flag

In the output, we can see a mandatory label ACE with a `Medium` integrity level, even though we mentioned at the start of this section that the `Medium` level wouldn't normally be assigned. We can also see that the mandatory policy has been set

to `NoWriteUp|NoReadUp`, which matches the auto-inherit flags we specified.

Determining Object Inheritance

When we specify an object ACE type, such as `AllowedObject`, in a parent security descriptor, the inheritance rules change slightly. This is because each object ACE can contain two optional GUIDs: `ObjectType`, used for access checking, and `InheritedObjectType`, used for inheritance.

The `SeAssignSecurityEx` API uses the `InheritedObjectType` GUID in an ACE to calculate whether a new security descriptor should inherit that ACE. If this GUID exists and its value matches the `ObjectType` GUID, the new security descriptor will inherit the ACE. By contrast, if the values don't match, the ACE won't be copied. Table 6-6 shows the possible combinations of the `ObjectType` parameter and `InheritedObjectType` and whether the ACE is inherited.

Table 6-6 Whether to Inherit the ACE Based on `InheritedObjectType`

`ObjectType` parameter specified?	**`InheritedObjectType` in ACE?**	**Inherited**
No	No	Yes
No	Yes	**No**
Yes	No	Yes
Yes	Yes (and the values match)	Yes
Yes	Yes (and the values don't match)	**No**

I've bolded the cases in Table 6-6 where inheritance doesn't happen. Note that this doesn't supersede any other inheritance decision: the flags must have `ObjectInherit` or `ContainerInherit` to be considered for inheritance.

In Listing 6-33, we verify this behavior by adding some object ACEs to a security descriptor and using it as the parent.

```
PS> $owner = Get-NtSid -KnownSid BuiltinAdministrators
PS> $parent = New-NtSecurityDescriptor -Type Directory -Owner $owner -Group
$owner

1 PS> $type_1 = New-Guid
```

```
PS> $type_2 = New-Guid

PS> Add-NtSecurityDescriptorAce $parent -Name "SYSTEM" -Access GenericAll
-Flags ObjectInherit -Type AllowedObject -ObjectType $type_1
PS> Add-NtSecurityDescriptorAce $parent -Name "Everyone" -Access GenericAll
-Flags ObjectInherit -Type AllowedObject -InheritedObjectType $type_1
PS> Add-NtSecurityDescriptorAce $parent -Name "Users" -Access GenericAll
-Flags ObjectInherit -InheritedObjectType $type_2 -Type AllowedObject
PS> Format-NtSecurityDescriptor $parent -Summary -SecurityInformation Dacl
<DACL>
NT AUTHORITY\SYSTEM:
(AllowedObject)(ObjectInherit)(GenericAll)(OBJ:f5ee1953...)
Everyone: (AllowedObject)(ObjectInherit)(GenericAll)(IOBJ:f5ee1953...)
BUILTIN\Users: (AllowedObject)(ObjectInherit)(GenericAll)(IOBJ:0b9ed996...)

PS> $token = Get-NtToken -Effective -Pseudo
PS> $sd = New-NtSecurityDescriptor -Token $token -Parent $parent
-Type Directory -ObjectType $type_2
PS> Format-NtSecurityDescriptor $sd -Summary -SecurityInformation Dacl
<DACL>
NT AUTHORITY\SYSTEM: (AllowedObject)(None)(Full Access)(OBJ:f5ee1953...)
BUILTIN\Users: (Allowed)(None)(Full Access)
```

Listing 6-33 Verifying the behavior of the `InheritedObjectType` GUID

We first generate a couple of random GUIDs to act as our object types 1. Next, we add three inheritable `AllowedObject` ACEs to the parent security descriptor. In the first ACE, we set `ObjectType` to the first GUID we created 2. This ACE demonstrates that the `ObjectType` GUID is not considered when inheriting the ACE. The second ACE sets the `InheritedObjectType` to the first GUID 3. The final ACE uses the second GUID 4.

We then create a new security descriptor, passing the second GUID to the `ObjectType` parameter 5. When we check the new security descriptor, we can see that it inherited the ACE without the `InheritedObjectType` 6. The second ACE in the output is a copy of the ACE with an `InheritedObjectType` GUID that matches 7. Notice that, based on the output, the `InheritedObjectType` has been removed, as the ACE is no longer inheritable.

Having a single `ObjectType` GUID parameter is somewhat inflexible, so Windows provides the `SeAssignSecurityEx2` kernel API and the

`RtlNewSecurityObjectWithMultipleInheritance` user-mode API. These APIs takes a list of GUIDs rather than a single GUID. Any ACE with the `InheritedObjectType` in the list will be inherited; otherwise, the inheritance rules are basically the same as those covered here.

This concludes our discussion on assigning security descriptors during creation. As you've seen, the assignment process is complex, especially with regards to inheritance. We'll now discuss assigning a security descriptor to an existing resource, a considerably simpler process.

Assigning a Security Descriptor to an Existing Resource

If a resource already exists, it's not possible to set the security descriptor by calling a creation system call such as `NtCreateMutant` and specifying the `SecurityDescriptor` field in the object attributes. Instead, you need to open a handle to the resource with one of three access rights, depending on what part of the security descriptor you want to modify. Once you have this handle, you can call the `NtSetSecurityObject` system call to set specific security descriptor information. Table 6-7 shows the access rights needed to set each security descriptor field based on the `SecurityInformation` enumeration.

Table 6-7 `SecurityInformation` Flags and Required Access for Security Descriptor Creation

Flag name	Description	Location	Handle access required
`Owner`	Set the owner SID	Owner	`WriteOwner`
`Group`	Set the group SID	Group	`WriteOwner`
`Dacl`	Set the DACL	DACL	`WriteDac`
`Sacl`	Set the SACL; for auditing ACEs only	SACL	`AccessSystemSecurity`
`Label`	Set the mandatory label	SACL	`WriteOwner`
`Attribute`	Set a system resource attribute	SACL	`WriteDac`
`Scope`	Set a scoped policy ID	SACL	`AccessSystemSecurity`
`ProcessTrustLabel`	Set the process trust label	SACL	`WriteDac`
`AccessFilter`	Set an access filter	SACL	`WriteDac`

Backup	Set everything except ProcessTrustLabel and AccessFilter	All	WriteDac, WriteOwner, and AccessSystemSecurity

You might notice that the handle access required for setting this information is more complex than the access needed to merely query it, as it is split across three access rights instead of two. Instead of trying to memorize these access rights, you can retrieve them using the Get-NtAccessMask PowerShell command, specifying the parts of the security descriptor you want to set with the SecurityInformation parameter, as shown in Listing 6-34.

```
PS> Get-NtAccessMask -SecurityInformation AllBasic -ToGenericAccess
ReadControl

PS> Get-NtAccessMask -SecurityInformation AllBasic -ToGenericAccess -
SetSecurity
WriteDac, WriteOwner
```

Listing 6-34 Discovering the access mask needed to query or set specific security descriptor information

To set a security descriptor, the NtSetSecurityObject system call invokes a type-specific security function. This type-specific function allows the kernel to support the different storage requirements for security descriptors; for example, a file must persist its security descriptor to disk, while the object manager can store a security descriptor in memory.

These type-specific functions eventually call the SeSetSecurityDescriptorInfoEx kernel API to build the updated security descriptor. User mode exports this kernel API as RtlSetSecurityObjectEx. Once the security descriptor has been updated, the type-specific function can store it using its preferred mechanism.

The SeSetSecurityDescriptorInfoEx API accepts five parameters and returns a new security descriptor:

Modification Security Descriptor
A new security descriptor passed to NtSetSecurityObject

Object Security Descriptor

The current security descriptor for the object being updated

Security Information
Flags to specify what parts of the security descriptor to update, described in Table 6-7

Auto-Inherit
A set of bit flags that define the auto-inheritance behavior

Generic Mapping
The generic mapping for the type being created

No kernel code uses the auto-inherit flags; therefore, the behavior of this API is simple. It merely copies the parts of the security descriptor that we've specified in the security information parameter to the new security descriptor. It also maps any generic access to the type-specific access using the generic mapping, excluding `InheritOnly` ACEs.

Some security descriptor control flags introduce special behavior. For example, it's not possible to explicitly set `DaclAutoInherited`. However, you can specify it along with `DaclAutoInheritReq` to set it on the new security descriptor.

We can test out the `RtlSetSecurityObjectEx` API using the `Edit-NtSecurityDescriptor` command, as shown in Listing 6-35.

```
PS> $owner = Get-NtSid -KnownSid BuiltinAdministrators
PS> $obj_sd = New-NtSecurityDescriptor -Type Mutant -Owner $owner -Group
$owner
PS> Add-NtSecurityDescriptorAce $obj_sd -KnownSid World -Access GenericAll
PS> Format-NtSecurityDescriptor $obj_sd -Summary -SecurityInformation Dacl
<DACL>
Everyone: (Allowed)(None)(Full Access)

PS> Edit-NtSecurityDescriptor $obj_sd -MapGeneric
PS> $mod_sd = New-NtSecurityDescriptor -Type Mutant
PS> Add-NtSecurityDescriptorAce $mod_sd -KnownSid Anonymous -Access
GenericRead
PS> Set-NtSecurityDescriptorControl $mod_sd DaclAutoInherited,
DaclAutoInheritReq
PS> Edit-NtSecurityDescriptor $obj_sd $mod_sd -SecurityInformation Dacl
PS> Format-NtSecurityDescriptor $obj_sd -Summary -SecurityInformation Dacl
```

```
<DACL> (Auto Inherited)
NT AUTHORITY\ANONYMOUS LOGON: (Allowed)(None)(QueryState|ReadControl)
```

Listing 6-35 Using `Edit-NtSecurityDescriptor` to modify an existing security descriptor

You can set the security for a kernel object using the `Set-NtSecurityDescriptor` command. The command can accept either an object handle with the required access or an OMNS path to the resource. For example, the following commands will try to modify the object *\BasedNamedObject\ABC* by setting a new DACL.

```
PS> $new_sd = New-NtSecurityDescriptor -Sddl "D:(A;;GA;;;WD)"
PS> Set-NtSecurityDescriptor -Path "\BaseNamedObjects\ABC"
-SecurityDescriptor $new_sd -SecurityInformation Dacl
```

Even if you can open a resource with the required access to set a security descriptor component, such as `WriteOwner` access, this doesn't mean the kernel will let you do it. The same rules regarding owner SIDs and mandatory labels apply here as when assigning a security descriptor at creation time.

The `SeSetSecurityDescriptorInfoEx` API enforces these rules. If no object security descriptor is specified, then the API returns the `STATUS_NO_SECURITY_ON_OBJECT` status code. Therefore, you can't set the security descriptor for a type with `SecurityRequired` set to false; that object won't have a security descriptor, so any attempt to modify it causes the error.

NOTE One ACE flag I haven't mentioned yet is `Critical`. The Windows kernel contains code to check the `Critical` flag and block the removal of ACEs that have the flag set. However, which ACEs to deem `Critical` is up to the code assigning the new security descriptor, and APIs such as `SeSetSecurityInformationEx` do not enforce it. Therefore, do not rely on the `Critical` flag to do anything specific. If you're using security descriptors in user mode, you can handle the flag any way you like.

What happens if you change the inheritable ACEs on a container? Will the changes in the security descriptor propagate to all existing children? In a word, no. Technically, a type could implement this automatic propagation behavior, but none do.

Instead, it's up to the user-mode components to handle it. Let's now look at the user-mode Win32 APIs that implement this propagation.

Win32 Security APIs

Most applications don't directly call the kernel system calls to read or set security descriptors. Instead, they'll use a range of Win32 APIs. While we won't discuss every API you could use, we'll cover some of the additional functionality added by the APIs to the underlying system calls.

Win32 implements the `GetKernelObjectSecurity` and `SetKernelObjectSecurity` APIs, which wrap `NtQuerySecurityObject` and `NtSetSecurityObject`. Likewise, the `CreatePrivateObjectSecurityEx` and `SetPrivateObjectSecurityEx` Win32 APIs wrap `RtlNewSecurityObjectEx` and `RtlSetSecurityObjectEx`, respectively. Every property of the native APIs discussed in this chapter applies equally to these Win32 APIs.

However, Win32 also provides some higher-level APIs: most notably, `GetNamedSecurityInfo` and `SetNamedSecurityInfo`. These APIs allow an application to query or set a security descriptor by providing a path and the type of resource that path refers to, rather than a handle. The use of a path and type allows the function to be more general; for example, the API supports getting and setting the security of files and registry keys but also services, printers, and Active Directory DS entries.

To query or set the security descriptor, the API must open the specified resource and then call the appropriate API to perform the operation. For example, to query a file's security descriptor, the API would open the file using the `CreateFile` Win32 API and then call the `NtQuerySecurityObject` system call. However, to query a printer's security descriptor, it needs to open the printer using the `OpenPrinter` print spooler API and then

call the `GetPrinter` API on the opened printer handle to query the security descriptor.

PowerShell already uses the `GetNamedSecurityInfo` API through the `Get-Acl` command; however, the built-in command doesn't support reading certain security descriptor ACEs, such as mandatory labels. Therefore, the `NtObjectManager` module implements `Get-Win32SecurityDescriptor`, which calls `GetNamedSecurityInfo` and returns a `SecurityDescriptor` object.

If you merely want to display the security descriptor, you can use the `Format-Win32SecurityDescriptor` command, which takes the same parameters but doesn't return a `SecurityDescriptor` object. Listing 6-36 provides a couple of examples of commands that leverage the underlying Win32 security APIs.

```
PS> Get-Win32SecurityDescriptor "$env:WinDir"
Owner                       DACL ACE Count SACL ACE Count Integrity Level
-----                       -------------- -------------- ---------------
NT SERVICE\TrustedInstaller 13             NONE           NONE

PS> Format-Win32SecurityDescriptor "MACHINE\SOFTWARE" -Type RegistryKey -
Summary
<Owner> : NT AUTHORITY\SYSTEM
<Group> : NT AUTHORITY\SYSTEM
<DACL> (Protected, Auto Inherited)
BUILTIN\Users: (Allowed)(ContainerInherit)(QueryValue|...)
--snip--
```

Listing 6-36 An example usage of `Get-Win32SecurityDescriptor` and `Format-Win32SecurityDescriptor`

We start by using `Get-Win32SecurityDescriptor` to query the security descriptor for the *Windows* folder, in this case `$env:WinDir`. Note that we don't specify the type of resource we want to query, as it defaults to a file. In the second example, we use `Format-Win32Security` to display the security descriptor for the *MACHINE\SOFTWARE* key. This key path corresponds to the Win32 *HKEY_LOCAL_MACHINE\SOFTWARE* key path. We need to indicate that we're querying a registry key by specifying the

`Type` parameter; otherwise, the command will try to open the path as a file, which is unlikely to work.

NOTE To find the path format for every supported type, consult the API documentation for `GetNamedSecurityInfo` and `SetNamedSecurityInfo`, which provides numerous examples.

The `SetNamedSecurityInfo` API is more complex, as it implements auto-inheritance across hierarchies (for example, across a file directory tree). As we discussed earlier, if you use the `NtSetSecurityObject` system call to set a file's security descriptor, any new inheritable ACEs won't get propagated to any existing children.

If you set a security descriptor on a file directory with `SetNamedSecurityInfo`, the API will enumerate all child files and directories and attempt to update each child's security descriptor. The API generates the new security descriptor by querying the child security descriptor and using it as the creator security descriptor in a call to `RtlNewSecurityObjectEx`, taken the parent security descriptor from the parent directory. The `DaclAutoInherit` and `SaclAutoInherit` flags are always set, to merge any explicit ACEs in the creator security descriptor into the new security descriptor.

PowerShell exposes the `SetNamedSecurityInfo` API through the `Set-Win32SecurityDescriptor` command, shown in Listing 6-37.

```
PS> $path = Join-Path "$env:TEMP" "TestFolder"
1 PS> Use-NtObject($f = New-NtFile $path -Win32Path -Options DirectoryFile
-Disposition OpenIf) {
    Set-NtSecurityDescriptor $f "D:AIARP(A;OICI;GA;;;WD)" Dacl
}

PS> $item = Join-Path $path test.txt
PS> "Hello World!" | Set-Content -Path $item
PS> Format-Win32SecurityDescriptor $item -Summary -SecurityInformation Dacl
<DACL> (Auto Inherited)
2 Everyone: (Allowed)(Inherited)(Full Access)

PS> $sd = Get-Win32SecurityDescriptor $path
PS> Add-NtSecurityDescriptorAce $sd -KnownSid Anonymous -Access GenericAll
```

```
    -Flags ObjectInherit,ContainerInherit,InheritOnly
3 PS> Set-Win32SecurityDescriptor $path $sd Dacl
  PS> Format-Win32SecurityDescriptor $item -Summary -SecurityInformation Dacl
  <DACL> (Auto Inherited)
  Everyone: (Allowed)(Inherited)(Full Access)
4 NT AUTHORITY\ANONYMOUS LOGON: (Allowed)(Inherited)(Full Access)
```

Listing 6-37 Testing auto-inheritance with `Set-Win32SecurityDescriptor`

Listing 6-37 demonstrates the auto-inheritance behavior of `SetNamedSecurityInfo` for files. We first create the *TestFolder* directory in the root of the system drive, then set the security descriptor so that it contains one inheritable ACE for the *Everyone* group and has the `DaclAutoInherited` and `DaclProtected` flags set 1. Next, we create a text file inside the directory and print its security descriptor 2. The DACL contains the single ACE inherited from the parent by the text file.

We then get the security descriptor from the directory and add a new inheritable ACE to it for the *Anonymous* user. We use this security descriptor to set the DACL of the parent using `Set-Win32SecurityDescriptor` 3. Printing the text file's security descriptor again, we now see that it's got two ACEs, as the *Anonymous* user ACE has been added. If we had used `Set-NtSecurityDescriptor` to set the parent directory's security descriptor, this inheritance would not have taken place.

As `SetNamedSecurityInfo` always uses auto-inheritance, applying a protected security descriptor control flag, such as `DaclProtected` or `SaclProtected`, becomes an important way to block the automatic propagation of ACEs.

Oddly, the API doesn't allow you to specify the `DaclProtected` and `SaclProtected` control flags directly in the security descriptor you specify. Instead, it introduces some additional `SecurityInformation` flags to handle setting and unsetting the control flags. To set a protected security descriptor control flag, you can use the `ProtectedDacl` and `ProtectedSacl` flags for `SecurityInformation`. To remove a flag, use `UnprotectedDacl` and `UnprotectedSacl`. Listing 6-38 provides examples of setting and unsetting the protected control flag for the DACL.

```
PS> $path = Join-Path "$env:TEMP\TestFolder" "test.txt"
1 PS> $sd = New-NtSecurityDescriptor "D:(A;;GA;;;AU)"
PS> Set-Win32SecurityDescriptor $path $sd Dacl,ProtectedDacl
PS> Format-Win32SecurityDescriptor $path -Summary -SecurityInformation Dacl
<DACL> (Protected, Auto Inherited)
NT AUTHORITY\Authenticated Users: (Allowed)(None)(Full Access)

2 PS> Set-Win32SecurityDescriptor $path $sd Dacl,UnprotectedDacl
PS> Format-Win32SecurityDescriptor $path -Summary -SecurityInformation Dacl
<DACL> (Auto Inherited)
NT AUTHORITY\Authenticated Users: (Allowed)(None)(Full Access)
Everyone: (Allowed)(Inherited)(Full Access)
NT AUTHORITY\ANONYMOUS LOGON: (Allowed)(Inherited)(Full Access)
```

Listing 6-38 Testing the `ProtectedDacl` and `UnprotectedDacl` `SecurityInformation` flag

This script assumes you've run Listing 6-37 already, as we reuse the file we created there. We create a new security descriptor with a single ACE for the *Authenticated Users* group and assign it to the file with the `ProtectedDacl` and `Dacl` flags. As a result, the protected control flag for the DACL is now set on the file 1. Note that the inherited ACEs from Listing 6-37 have been removed; only the new, explicit ACE is left.

Finally, if we assign the security descriptor again with the `UnprotectedDacl` flag, the new security descriptor no longer has the protected control flag set 2. Also, the API restores the inherited ACEs from the parent directory and merges them with the explicit ACE for the *Authenticated Users* group.

The behavior of the command when we specify the `UnprotectedDacl` flag shows you how you can restore the inherited ACEs for any file. If you specify an empty DACL, so that no explicit ACEs will be merged, and additionally specify the `UnprotectedDacl` flag, you'll reset the security descriptor to the version based on its parent. To simplify this operation, the PowerShell module contains the `Reset-Win32SecurityDescriptor` command (Listing 6-39).

```
PS> $path = Join-Path "$env:TEMP\TestFolder" "test.txt"
1 PS> Reset-Win32SecurityDescriptor $path Dacl
PS> Format-Win32SecurityDescriptor $path -Summary -SecurityInformation Dacl
<DACL> (Auto Inherited)
Everyone: (Allowed)(Inherited)(Full Access)
NT AUTHORITY\ANONYMOUS LOGON: (Allowed)(Inherited)(Full Access)
```

Listing 6-39 Resetting the security of a directory using `Reset-Win32SecurityDescriptor`

In Listing 6-39, we call `Reset-Win32SecurityDescriptor` with the path to the file, and request that the DACL be reset 1. When we display the security descriptor of the file, we now find that it's been reset based on the parent directory's security descriptor, in Listing 6-37.

THE DANGERS OF AUTO-INHERITANCE

The auto-inheritance features of the Win32 security APIs are convenient for applications, which can merely set an inheritable security descriptor to apply it to any child resources. However, auto-inheritance introduces a security risk, especially if used by privileged applications or services.

The risk occurs if the privileged application can be tricked into resetting the inherited security for a hierarchy when a malicious user has control over the parent security descriptor. For example, CVE-2018-0983 was a security vulnerability in the privileged storage service: it called `SetNamedSecurityInfo` to reset the security of a file with the path specified by the user. By using some filesystem tricks, an attacker could link the file being reset to a system file that was writeable by an administrator only. However, the `SetNamedSecurityInfo` API thought the file was in a directory controlled by the user, so it reset the security descriptor based on that directory's security descriptor, granting the malicious user full access to the system file.

Microsoft has fixed this issue, and Windows no longer supports the filesystem tricks necessary to exploit it. However, there are other potential ways for a privileged service to be tricked. Therefore, if you're writing code to set or reset the security descriptor of a resource, pay careful attention to where the path comes from. If it's from an unprivileged user, make sure you impersonate the caller before calling any of the Win32 security APIs.

One final API to cover is `GetInheritanceSource`, which allows you to identify the source of a resource's inherited ACEs. One reason the inherited ACEs are marked with the `Inherited` flag is to facilitate the analysis of inherited ACEs. Without the flag, the API would have no way of distinguishing between inherited and non-inherited ACEs.

Based on the state of the `Inherited` ACE flag, the API works its way up the parent hierarchy until it finds an inheritable ACE that doesn't have the `Inherited` flag set but contains the same SID and access mask. Of course, there is no guarantee that the found ACE is the actual source of the inherited ACE, which could potentially live multiple levels down the hierarchy from the parent. Thus, treat the output of `GetInheritanceSource` as purely informational, and don't use it for security-critical decisions.

As with the other Win32 APIs, `GetInheritanceSource` supports different types. However, it's limited to resources that have a child-parent relationship, such as files, registry keys, and DS objects. You can access the API through the `Search-Win32SecurityDescriptor` command (Listing 16-40).

```
PS> $path = Join-Path "$env:TEMP" "TestFolder"
1 PS> Search-Win32SecurityDescriptor $path | Format-Table
Name Depth User                           Access
---- ----- ----                           ------
     0     Everyone                       GenericAll
     0     NT AUTHORITY\ANONYMOUS LOGON   GenericAll

PS> $path = Join-Path $path "new.txt"
PS> "Hello" | Set-Content $path
2 PS> Search-Win32SecurityDescriptor $path | Format-Table
Name                 Depth User                           Access
----                 ----- ----                           ------
C:\Temp\TestFolder\ 1     Everyone                       GenericAll
C:\Temp\TestFolder\ 1     NT AUTHORITY\ANONYMOUS LOGON   GenericAll
```

Listing 6-40 Enumerating inherited ACEs using `Search-Win32SecurityDescriptor`

We call `Search-Win32SecurityDescriptor` with the path to the directory we created in Listing 6-38 1. The output is a list of the ACEs in the resource's DACL, including the name of the resource from which the ACE was inherited and the depth of the hierarchy. We set two explicit ACEs on the directory. The output reflects this as a `Depth` value of `0`, which indicates that the ACE wasn't inherited. Also, the name column is empty.

In contrast, if we create a new file in the directory and rerun the command, we get different output 2. As you might have expected, the ACEs show that they were both inherited from the parent folder, with a depth of `1`.

We've covered the basics of the Win32 APIs. Keep in mind that there are clear differences in behavior between the low-level system calls and these Win32 APIs, especially regarding inheritance. When you interact with the security of resources via a GUI, it's almost certainly calling one of these APIs.

Server Security Descriptors and Compound ACEs

Let's finish this chapter with a topic we briefly mentioned when we discussed creator SIDs: server security descriptors. The kernel supports two, very poorly documented security descriptor control flags for servers: `ServerSecurity` and `DaclUntrusted`. We use these flags only when generating a new security descriptor, either at object creation time or when assigning a security descriptor explicitly. The main control flag, `ServerSecurity`, indicates to the security descriptor generation code that the caller is expecting to impersonate another user.

When a new security descriptor is created during impersonation, the owner and group SIDs will default to the values from the impersonation token. This might not be desirable, as being the owner of a resource can grant a caller additional access to it. However, the caller can't set the owner to an arbitrary SID, because the SID must be able to pass the owner check, which is based on the impersonation token.

This is where `ServerSecurity` control flag comes in. If you set the flag on the creator security descriptor when creating a new security descriptor, the owner and group SIDs default to the primary token of the caller, and not to the impersonation token. This flag also replaces all `Allowed` ACEs in the DACL with `AllowedCompound` ACEs, the structure of which we defined back in Chapter 5. In the compound ACE, the server SID is set to the owner SID from the primary token. Listing 6-41 shows an example.

```
1 PS> $token = Get-NtToken -Anonymous
  PS> $creator = New-NtSecurityDescriptor -Type Mutant
  PS> Add-NtSecurityDescriptorAce $creator -KnownSid World -Access GenericAll
  PS> $sd = New-NtSecurityDescriptor -Token $token -Creator $creator
  PS> Format-NtSecurityDescriptor $sd -Summary -SecurityInformation
  Owner,Group,Dacl
2 <Owner> : NT AUTHORITY\ANONYMOUS LOGON
  <Group> : NT AUTHORITY\ANONYMOUS LOGON
  <DACL>
  Everyone: (Allowed)(None)(Full Access)

3 PS> Set-NtSecurityDescriptorControl $creator ServerSecurity
```

```
PS> $sd = New-NtSecurityDescriptor -Token $token -Creator $creator
PS> Format-NtSecurityDescriptor $sd -Summary -SecurityInformation
Owner,Group,Dacl
<Owner> : GRAPHITE\user
<Group> : GRAPHITE\None
<DACL>
Everyone: (AllowedCompound)(None)(Full Access)(Server:GRAPHITE\user)
```

Listing 6-41 Testing the `ServerSecurity` security descriptor control flag

We first create a new security descriptor using the `Anonymous` user token 1. This initial test doesn't set the `ServerSecurity` flag. As expected, the `Owner` and `Group` default to values based on the `Anonymous` user token, and the single ACE we added remains intact 2. Now we add the `ServerServer` control flag to the creator security descriptor 3. After calling `New-NtSecurityDescriptor` again, we now find that the `Owner` and `Group` are set to the defaults for the primary token, not to those of the `Anonymous` user token 4. Also, the single ACE has been replaced with a compound ACE, whose server SID is set to the primary token owner SID 5. We'll discuss how changes to compound ACEs impacts access checking in Chapter 7.

The `DaclUntrusted` control flags works in combination with `ServerSecurity`. By default, `ServerSecurity` assumes that any compound ACE in the DACL is trusted and will copy it verbatim into the output. When the `DaclUntrusted` control flag is set, all compounds ACEs instead have their server SID values set to the primary token owner SID.

If the `ServerSecurity` control flag is set on the creator security descriptor and the new security descriptor inherits ACEs from a parent, we can convert the `CREATOR OWNER SERVER` and `CREATOR GROUP SERVER` SIDs to their respective primary token values. Also, any inherited `Allowed` ACEs will be converted to compound ACEs, except for those of the default DACL.

A Summary of Inheritance Behavior

As inheritance is a very important topic to understand, Table 6-8 summarizes the ACL inheritance rules we've discussed in this chapter, to help you make sense of them.

Table 6-8 Summary of Inheritance Rules for the DACL

Parent ACL	Creator ACL	AutoInherit Set	AutoInherit Not Set
None	None	Default	Default
None	Present	Creator	Creator
Non-inheritable	None	Default	Default
Inheritable	None	Parent	Parent
Non-inheritable	Present	Creator	Creator
Inheritable	Present	Parent and Creator	Creator
Non-inheritable	Protected	Creator	Creator
Inheritable	Protected	Creator	Creator
Non-inheritable	Defaulted	Creator	Creator
Inheritable	Defaulted	Parent	Parent

Table 6-8 includes four columns: the state of the parent ACL, the creator ACL, and two columns describing the resulting ACL, depending on whether the `DaclAutoInherit` and `SaclAutoInherit` auto-inherit flags were set or not. There are six ACL types to consider:

None
The ACL isn't present in the security descriptor

Present
The ACL is present in the security descriptor (even if it is a NULL or empty ACL)

Non-Inheritable
The ACL has no inheritable ACEs

Inheritable
The ACL has one or more inheritable ACEs

Protected
The security descriptor has the `DaclProtected` or `SaclProtect` control flag set

Defaulted

The security descriptor has the `DaclDefaulted` or `SaclDefaulted` control flag set

Additionally, there are four possible resulting ACLs:

Default
The default DACL from the token, or nothing in the case of a SACL

Creator
All ACEs from the creator ACL

Parent
The inheritable ACEs from the parent ACL

Parent and Creator
The inheritable ACEs from the parent and explicit ACEs from the creator

When the auto-inherit flag is set, the new security descriptor will have the `DaclAutoInherited` or `SaclAutoInherited` control flag set. Also, all ACEs that were inherited from the parent ACL will have the `Inherited` ACE flag set. Note that this table doesn't consider the behavioral changes due to object ACEs, mandatory labels, server security, and creator SIDs, which add more complexity.

Worked Examples

Let's walk through some worked examples that use the commands you've learned about in this chapter.

Finding Object Manager Resource Owners

As you've seen in this chapter, the owner of a resource's security descriptor is usually the user SID of the resource's creator. For administrators, however, it's typically the built-in *Administrators* group. The only way to set a different owner SID is to use another token group SID that has the `Owner` flag set, or

to enable `SeRestorePrivilege`. Neither option is available to non-administrator users.

Thus, knowing the owner of a resource can indicate whether a more privileged user created and used the resource. This could help you identify potential misuses of the Win32 security APIs in privileged applications, or find shared resources that a lower-privileged user might write to; a privileged user could mishandle these, causing a security issue.

Listing 6-42 shows a simple example: finding object manager resources whose owner SID differs from the caller's.

```
1 PS> function Get-NameAndOwner {
      [CmdletBinding()]
      param(
          [parameter(Mandatory, ValueFromPipeline)]
          $Entry,
          [parameter(Mandatory)]
          $Root
      )

      begin {
        2 $curr_owner = Get-NtSid -Owner
      }

      process {
        3 $sd = Get-NtSecurityDescriptor -Path $Entry.Name -Root $Root
  -TypeName $Entry.NtTypeName -ErrorAction SilentlyContinue
          if ($null -ne $sd -and $sd.Owner.Sid -ne $curr_owner) {
              [PSCustomObject] @{
                  Name = $Entry.Name
                  NtTypeName = $Entry.NtTypeName
                  Owner = $sd.Owner.Sid.Name
                  SecurityDescriptor = $sd
              }
          }
      }
  }

4 PS> Use-NtObject($dir = Get-NtDirectory \BaseNamedObjects) {
      Get-NtDirectoryEntry $dir | Get-NameAndOwner -Root $dir
  }
  Name                        NtTypeName   Owner
  SecurityDescriptor
  ----                        ----------   -----                    --------------
  ----
  CLR_PerfMon_DoneEnumEvent   Event        NT AUTHORITY\SYSTEM
  O:SYG:SYD:(A;;...
```

```
WAMACAPO;3_Read             Event       BUILTIN\Administrators
O:SYG:SYD:(A;;...
WAMACAPO;8_Mem              Section     BUILTIN\Administrators
O:SYG:SYD:(A;;...
--snip--
```

Listing 6-42 Finding objects in `BaseNamedObjects` that are owned by a different user

We first define a function to query the name and owner of an object manager directory entry 1. The function initializes the `$curr_owner` variable with the owner SID of the caller's token 2. We'll compare this SID with the owner of a resource to return only resources owned by a different user.

For each directory entry, we query its security descriptor using the `Get-NtSecurityDescriptor` command 3. We can specify a path and a root `Directory` object to the command to avoid having to manually open the resource. If we successfully query the security descriptor, and if the owner SID does not match the current user's owner SID, we return the resource's name, object type, and owner SID.

To test the new function, we open a directory (in this case, the global base-named objects 4), and use `Get-NtDirectoryEntry` to query for all entries, piping them through the function we defined. We receive a list of resources not owned by the current user.

For example, the output includes the `WAMACAPO;8_Mem` object, which is a shared memory `Section` object. If a normal user can write to this `Section` object, we should investigate it further, as it might be possible to trick a privileged application into performing an operation that would elevate a normal user's privileges.

We can test our ability to get write access on the `Section` object by using the `Get-NtGrantedAccess` command with the `SecurityDescriptor` property of the object, as shown in Listing 6-43.

```
PS> $entry
Name            NtTypeName   Owner                  SecurityDescriptor
----            ----------   -----                  ------------------
```

```
WAMACAPO;8_Mem  Section      BUILTIN\Administrators  O:SYG:SYD:(A;;...

PS> Get-NtGrantedAccess -SecurityDescriptor $entry.SecurityDescriptor
Query, MapWrite, MapRead, ReadControl
```

Listing 6-43 Getting the granted access for a `Section` object

The `$entry` variable contains the object we want to inspect. We pass its security descriptor to the `Get-NtGrantedAccess` command to return the maximum granted access for that resource. In this case, we can see that `MapWrite` is present, which indicates that the `Section` object could be mapped as writeable.

The example I've shown in Listing 6-42 should provide you with an understanding of how to query for any resource. You can replace the directory with a file or registry key, then call `Get-NtSecurityDescriptor` with the path and the root object to query the owner for each of these resource types.

For the object manager and registry, however, there is a much simpler way of finding the owner SID. For the registry, we can look up the security descriptor for the entries returned from the `NtObject` drive provider using the `SecurityDescriptor` property. For example, we can select the name and owner SID fields for the root registry key using the following script:

```
PS> ls NtKey:\ | Select Name, {$_.SecurityDescriptor.Owner.Sid}
```

You can also specify the `Recurse` parameter to perform the check recursively.

If you want to query the owner SIDs of files, you can't use this technique, as the file provider does not return the security provider in its entries. Instead, you need to use the built-in `Get-Acl` command. Here, for example, we query a file's ACL:

```
PS> ls c:\ | Get-Acl | Select Path, Owner
```

The `Get-Acl` command returns the owner as a username, not a SID. You'll have to look up the SID manually using the `Get-NtSid` command and the `Name` parameter if you need it. Alternatively, you can convert the output of the `Get-Acl` command to a `SecurityDescriptor` object used in the `NtObjectManager` module, as shown in Listing 6-44.

```
PS> (Get-Acl c:\ | ConvertTo-NtSecurityDescriptor).Owner.Sid
Name                           Sid
----                           ---
NT SERVICE\TrustedInstaller S-1-5-80-956008885-3418522649-1831038044-...
```

Listing 6-44 Converting `Get-Acl` output to a `SecurityDescriptor` object

We use the `ConvertTo-NtSecurityDescriptor` PowerShell command to perform the conversion.

Changing the Ownership of a Resource

Administrators commonly take ownership of resources. This allows them to easily modify a resource's security descriptor and gain full access to it. Windows comes with several tools for doing this, such as *takeown.exe*, which sets the owner of a file to the current user. However, you'll find it instructive to go through the process of changing the owner manually, so you can understand exactly how it works. Run the commands in Listing 6-45 as an administrator.

```
PS> $new_dir = New-NtDirectory "ABC" -Win32Path
PS> Get-NtSecurityDescriptor $new_dir | Select {$_.Owner.Sid.Name}
$_.Owner.Sid.Name
-----------------
BUILTIN\Administrators

 PS> Set-NtTokenPrivilege SeRestorePrivilege
PS> Use-NtObject($dir = Get-NtDirectory "ABC" -Win32Path -Access WriteOwner) {
    $sid = Get-NtSid -KnownSid World
    $sd = New-NtSecurityDescriptor -Owner $sid
    Set-NtSecurityDescriptor $dir $sd -SecurityInformation Owner
}

PS> Get-NtSecurityDescriptor $new_dir | Select {$_.Owner.Sid.Name}
$_.Owner.Sid.Name
-----------------
Everyone

PS> $new_dir.Close()
```

Listing 6-45 Setting an arbitrary owner for a `Directory` object

We start by creating a new `Directory` object on which to perform the operations. (We'll avoid modifying an existing resource, which risks breaking your system.) We then query the

resource's current owner SID. In this case, because we're running this script as an administrator, it's set to the *Administrators* group.

Next, we enable the `SeRestorePrivilege` privilege. We need to do this only if we want to set an arbitrary owner SID. If we want to set a permitted SID, we can skip this line. We then open the `Directory` again, but only for `WriteOwner` access.

We can now create a security descriptor with just the owner SID set to the `World` SID. To do this, we call the `Set-NtSecurityDescriptor` PowerShell command, specifying only the `Owner` flag. If you haven't enabled `SeRestorePrivilege`, this operation will fail with a `STATUS_INVALID_OWNER` status code. To confirm that we've changed the owner SID, we query it again, which confirms that it's now set to `Everyone` (the name of the `World` SID).

You can apply this same set of operations to any resource type, including registry keys and files. Simply change the command used to open the resource. Whether you'll be granted `WriteOwner` access depends on the specifics of the access-check process. In Chapter 7, you'll learn about a few cases in which the access check automatically grants `WriteOwner` access based on certain criteria.

Wrapping Up

We started this chapter by giving an overview of how to read the security descriptor of an existing kernel resource using the `Get-NtObjectSecurity` command. We covered the security information flags that define what parts of the security descriptors the command should read and outlined the special rules for accessing audit information stored in the SACL.

We then discussed how we can assign security descriptors to resources, either during the resource creation process or by modifying an existing resource. In the process, you learned about ACL inheritance and auto-inheritance. We also discussed the behavior of the Win32 APIs, specifically

`SetNamedSecurityInfo`, and how the API implements auto-inheritance even though the kernel doesn't explicitly implement it. We concluded with an overview of the poorly documented server security descriptor and compound ACEs. In the next chapter, we'll (finally) discuss how Windows combines the token and security descriptor to check whether a user can access a resource.

7

ACCESS CHECKING

We've covered the first two components of the SRM: the security access token and the security descriptor. Now, we'll define its final component: the access-check process, which accepts the token and the security descriptor and applies a fixed set of rules to determine whether an application can access a resource.

In this chapter, we'll start by discussing the APIs you can call to perform an access check. Then, we'll take a deep dive into the implementation of the access check inside of the Windows kernel, detailing how the access check processes the different parts of the security descriptor and `Token` object to generate a final granted access value for the resource. In doing so, we'll develop our own

basic implementation of the access-check process using a PowerShell script.

Running an Access Check

When a caller attempts to open a resource, the kernel performs an access check based on the caller's identity. The API used to run an access check depends on whether it's being called from kernel mode or user mode. Let's start by describing the kernel-mode API.

Kernel-Mode Access Checks

The `SeAccessCheck` API implements the access-check process in kernel mode. It accepts the following parameters:

Security Descriptor
The security descriptor to use for the check; must contain both owner and group SIDs

Security Subject Context
The primary and impersonation tokens for the caller

Desired Access
An access mask for the access requested by the caller

Access Mode
The caller's access mode, set to either `UserMode` or `KernelMode`

Generic Mapping
The type-specific generic mapping

The API returns four values:

Granted Access
An access mask for the access the user was granted

Access Status Code

An NT status code indicating the result of the access check

Privileges
Any privileges used during the access check

Success Code
A Boolean value; if true, the access check succeeded

If the access check succeeds, the API will set the granted access to the desired access parameter, the success code to true, and the access status code to `STATUS_SUCCESS`. However, if any bit in the desired access is not granted, it will set the granted access to 0, the success code to false, and the access status code to `STATUS_ACCESS_DENIED`.

You might wonder why the API bothers returning the granted access value if all bits in the desired access must be granted for this value to indicate a success. The reason is that this behavior supports the `MaximumAllowed` access mask bit, which the caller can set in the desired access parameter. If the bit is set and the access check grants at least one access, the API returns `STATUS_SUCCESS`, setting the granted access to the maximum allowed access.

The security subject context parameter is a pointer to a `SECURITY_SUBJECT_CONTEXT` structure containing the caller's primary token and any impersonation token of the caller's thread. Typically, kernel code will use the kernel API `SeCaptureSubjectContext` to initialize the structure and gather the correct tokens for the current caller. If the impersonation token is captured, it must be at Impersonation level or above; if it's not at this level, the API will fail, and the access status code will be set to `STATUS_BAD_IMPERSONATION_LEVEL`.

Note that the call to `SeAccessCheck` might not occur in the thread that made the original resource request. For example, the check might have been delegated to a background thread in the `System` process. The kernel can capture the subject context from the original thread, then pass that context to the thread that

calls `SeAccessCheck`, to ensure that the access check uses the correct identity.

The Access Mode

The access mode parameter has two possible values, `UserMode` and `KernelMode`. If you pass `UserMode` to this parameter, all access checks will continue as normal. However, if you pass `KernelMode`, the kernel will disable all access checks. Why would you want to call `SeAccessCheck` without enforcing any security? Well, usually, you won't directly call the API with the `KernelMode` value. Instead, the parameter will be set to the value of the calling thread's `PreviousMode` value, which is stored in the thread's kernel object structure. When you call a system call from a user-mode application, the `PreviousMode` value is set to `UserMode` and passed to any API that needs the `AccessMode` set.

Therefore, the kernel normally enforces all access checks. Figure 7-1 shows the described behavior with a user-mode application calling the `NtCreateMutant` system call.

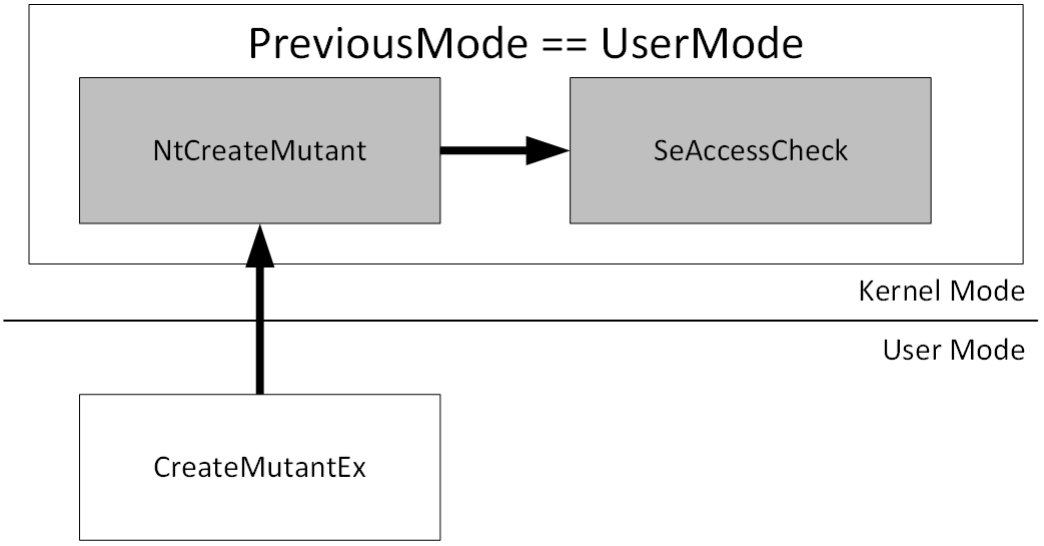

Figure 7-1 A thread's `PreviousMode` value when calling the `NtCreateMutant` system call

Even though the thread calling `SeAccessCheck` in Figure 7-1 is executing kernel code, the thread's `PreviousMode` value reflects the fact that the call was started from `UserMode`. Therefore, the `AccessMode` parameter specified to `SeAccessCheck` will be `UserMode`, and the kernel will enforce the access check.

The most common way of transitioning the thread's `PreviousMode` value from `UserMode` to `KernelMode` is for existing kernel code to call a system call via its `Zw` form, for example `ZwCreateMutant`. When such a call is made, the system call dispatch correctly identifies that the previous execution occurred in the kernel and sets `PreviousMode` to `KernelMode`. Figure 7-2 shows the transition of the thread's `PreviousMode` from `UserMode` to `KernelMode`.

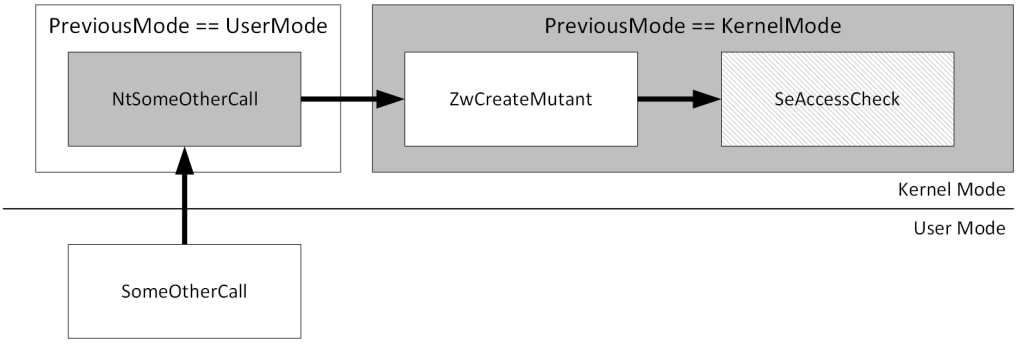

Figure 7-2 A thread's `PreviousMode` value being set to `KernelMode` after a call to `ZwCreateMutant`

In Figure 7-2, the user-mode application calls a hypothetical kernel system call, `NtSomeOtherCall`, that internally calls `ZwCreateMutant`. The code executing in the `NtSomeOtherCall` function runs with the `PreviousMode` value set to `UserMode`. However, once it calls `ZwCreateMutant`, the mode changes to `KernelMode` for the duration of the system call. Because `ZwCreateMutant` would call `SeAccessCheck` to determine whether the caller had access to a `Mutant` object, the API would receive the `AccessMode` set to `KernelMode`, disabling access checking.

This behavior could introduce a security issue if the hypothetical `NtSomeOtherCall` allowed the user-mode application to influence where the `Mutant` object was created. Once the access check is disabled, it might be possible to create or modify the `Mutant` in a location that the user would not normally be allowed to access.

Memory Pointer Checking

The access mode parameter has a second purpose: when `UserMode` is specified, the kernel will check any pointers passed as parameters to a kernel API to ensure that they do not point to kernel memory locations. This is an important security restriction: it prevents an application in user-mode from forcing a kernel API to read or write to kernel memory it should not have access to.

Specifying `KernelMode` disables these pointer checks at the same time as it disables the access checking. This mixing of behavior can introduce security issues; a kernel-mode driver might want to disable only pointer checking but inadvertently disable access checking as well.

How a caller can indicate these different uses of the access mode parameter depends on the kernel APIs being used. For example, you can sometimes specify two `AccessMode` values, one for the pointer checking and one for the access checking. A more common method is to specify a flag to the call; for example, the `OBJECT_ATTRIBUTES` structure passed to system calls has a flag, `ForceAccessCheck`, that disables pointer checking but leaves access checking enabled.

If you're analyzing a kernel driver, it's worth paying attention to the use of `Zw` APIs in which the `ForceAccessCheck` flag is not set. If a non-administrator user can control the target object manager path for the call, then there's likely to be a security vulnerability. For example, CVE-2020-17136 is a vulnerability in a kernel driver responsible for implementing the Microsoft OneDrive remote filesystem. The issue occurred because the API that the driver exposed to the Explorer shell did not set the `ForceAccessCheck` flag when creating a cloud-based file.

Because of that, a user calling these same APIs could create an arbitrary file anywhere they wanted on the filesystem, allowing them to gain administrator privileges.

User-Mode Access Checks

To support user-mode applications, the kernel exposes its access check implementation through the `NtAccessCheck` system call. This system call uses the same access check algorithm as the `SeAccessCheck` API; however, it's tailored to the unique behavior of user-mode callers. The parameters for the system call are as follows:

Security Descriptor
The security descriptor to use for the check; must contain owner and group SIDs

Client Token
A handle to an impersonation token for the caller

Desired Access
An access mask for the access requested by the caller

Generic Mapping
The type-specific generic mapping

The API returns four values:

Granted Access
An access mask for the access the user was granted

Access Status Code
An NT status code indicating the result of the access check

Privileges
Any privileges used during the access check

NT Success Code
A separate NT status code indicating the status of the system call

Some of the parameters present in the kernel API are missing here. For example, there is no reason to specify the access mode, as it will always be set to the caller's mode (`UserMode`, for a user-mode caller). Also, the caller's identity is now a handle to an impersonation token rather than a subject context. This handle must have `Query` access to be used for the access check. If you want to perform the access check against a primary token, you'll need to duplicate that token to an impersonation token first.

Another difference is that the impersonation token used in user-mode can be as low as Identification level. The reason for this disparity in impersonation level is that the system call is designed for user services that want to check a caller's permissions, and it's possible that the caller granted access to an Identification-level token only, which must be accounted for.

The system call returns an additional NT status code instead of the Boolean value returned by the kernel API. The return value indicates whether there was a problem with the parameters passed to the system call. For example, if the security descriptor doesn't have the owner or group SIDs set, the system call will return `STATUS_INVALID_SECURITY_DESCR`.

The NtAccessCheck PowerShell Command

Let's use `NtAccessCheck` from PowerShell to determine the caller's granted access based on a security descriptor and an access token. The PowerShell module wraps the call to `NtAccessCheck` with the `Get-NtGrantedAccess` command, shown in Listing 7-1.

```
PS> $sd = New-NtSecurityDescriptor -EffectiveToken -Type Mutant
PS> Format-NtSecurityDescriptor $sd -Summary
<Owner> : GRAPHITE\user
<Group> : GRAPHITE\None
<DACL>
GRAPHITE\user: (Allowed)(None)(Full Access)
NT AUTHORITY\SYSTEM: (Allowed)(None)(Full Access)
NT AUTHORITY\LogonSessionId_0_795805: (Allowed)(None)(ModifyState|...)

PS> Get-NtGrantedAccess $sd -AsString
Full Access
PS> Get-NtGrantedAccess $sd -Access ModifyState -AsString
```

```
ModifyState

PS> Clear-NtSecurityDescriptorDacl $sd
PS> Format-NtSecurityDescriptor $sd -Summary
<Owner> : GRAPHITE\user
<Group> : GRAPHITE\None
<DACL> - <EMPTY>

PS> Get-NtGrantedAccess $sd -AsString
ReadControl|WriteDac
```

Listing 7-1 Using the `Get-NtGrantedAccess` command

We start by creating the default security descriptor using the `EffectiveToken` parameter, and confirm that it is correct by formatting it. In simplistic terms, the system call will check this security descriptor's DACL for an `Allowed` ACE that matches one of the token's SIDs; if such an ACE exists, it will grant the access mask. As the first ACE in the DACL grants the User SID `Full Access`, we'd expect the result of the check to also grant `Full Access`.

We specify the security descriptor to `Get-NtGrantedAccess`. By not specifying an explicit token, we use the current effective token. We also do not specify an access mask, which means that the command checks `MaximumAllowed` access, converting the result to a string. The result is `Full Access`, as we expected based on the DACL.

We also test the command when supplied an explicit access mask using the `Access` parameter. The command will work out the access mask enumeration for the security descriptor's type to allow us to specify type-specific values. We requested to check for `ModifyState`, so receive only that access. For example, if we were opening a handle to a `Mutant` object, then the handle's access mask would grant only `ModifyState`.

To test an Access Denied case, we next removed all the ACEs from the DACL. If there is no `Allowed` ACE, then no access should be granted. But when we re-run `Get-NtGrantedAccess`, we get a surprise: we were granted `ReadControl` and `WriteDac` instead of nothing. To understand why we received these access levels, we need to dig

into the internals of the access check process. We do so in the next section.

Implementing the Access-Check Process in PowerShell

The access-check process in Windows has changed substantially since the first version of Windows NT. This has resulted in a complex set of algorithms that calculate what access a user is granted based on the combination of the security descriptor and the token. The flowchart in Figure 7-3 shows the major components of the access-check process.

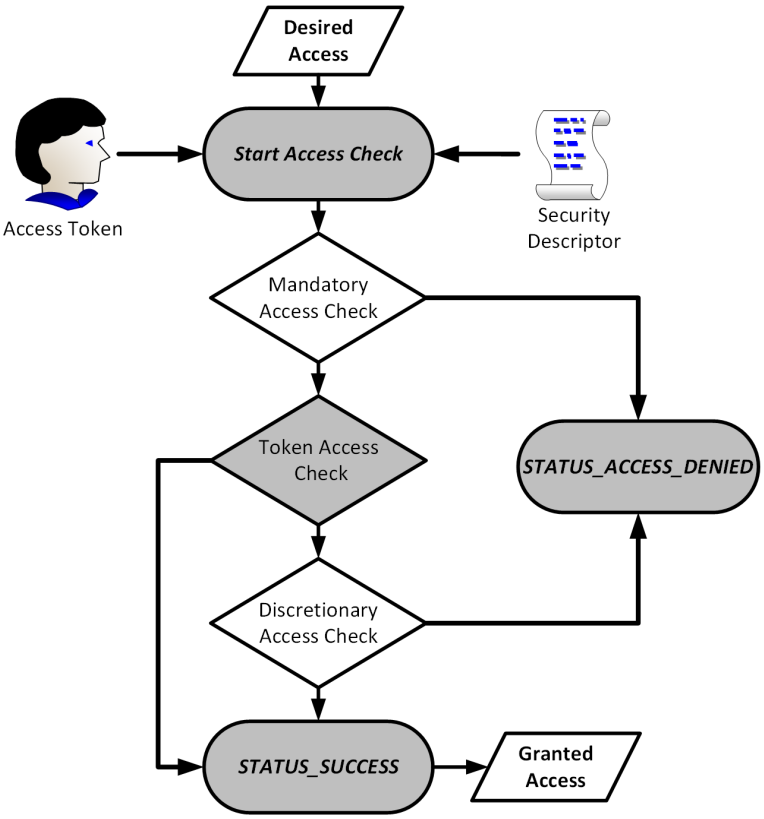

Figure 7-3 The access-check process

The access-check process starts by combining the token, the security descriptor, and the desired access mask. It then uses this

information in the following three main checks to determine whether access should be granted or denied:

Mandatory Access
Denies access to resources when the token does not meet a set policy

Token Access
Grants access based on the token's owner and privileges

Discretionary Access
Grants or denies access based on the DACL

To explore these steps in more detail, let's write a basic implementation of the access-check process in PowerShell. This PowerShell implementation won't replace the `Get-NtGrantedAccess` command, as, for simplicity, it won't check for maximum allowed access and might not include newer features. Even so, having an implementation that you can analyze and debug can help you gain a greater understanding of the access-check process.

The implementation of the access check is quite complex; therefore, we'll build it in stages. You can access the full implementation in the *chapter7_access_check_impl.psm1* script included with the book's example code. To use the script, import it as a module with this command:

```
PS> Import-Module .\chapter7_access_check_impl.psm1
```

Let's begin by defining the function to perform the access-check process.

Defining the Access-Check Function

The module exports a single top-level access-check function, `Get-PSGrantedAccess`, shown in Listing 7-2.

```
function Get-PSGrantedAccess {
    param(
        $Token = (Get-NtToken -Effective -Pseudo),
        $SecurityDescriptor,
        $GenericMapping,
```

```
        $DesiredAccess
    )

  1 $context = @{
        Token = $Token
        SecurityDescriptor = $SecurityDescriptor
        GenericMapping = $GenericMapping
        RemainingAccess = Get-NtAccessMask $DesiredAccess
        Privileges = @()
    }

    ## Test-MandatoryAccess defined below
  2 if (!(Test-MandatoryAccess $context)) {
        return Get-AccessResult STATUS_ACCESS_DENIED
    }

    ## Get-TokenAccess defined below
  3 Resolve-TokenAccess $context
  4 if (Test-NtAccessMask $context.RemainingAccess -Empty) {
        return Get-AccessResult STATUS_SUCCESS $context.Privileges
$DesiredAccess
    }

  5 if (Test-NtAccessMask $context.RemainingAccess AccessSystemSecurity) {
        return Get-AccessResult STATUS_PRIVILEGE_NOT_HELD
    }

  6 Get-DiscretionaryAccess $context
    if (Test-NtAccessMask $context.RemainingAccess -Empty) {
        return Get-AccessResult STATUS_SUCCESS $context.Privileges
$DesiredAccess
    }

  7 return Get-AccessResult STATUS_ACCESS_DENIED
}
```

Listing 7-2 The top-level access-check function

The function accepts the four parameters we defined earlier in the chapter: a token, the security descriptor, the type's generic mapping, and the desired access. If the caller doesn't specify a token, we'll use their effective token for the rest of the access check.

The first task tackled in the function is building a context that represents the current state of the access check process 1. The most important property used here is `RemainingAccess`. We initially set this property is to the `DesiredAccess` parameter,

then remove bits from the property as they're granted during the access-check process.

The rest of the function follows the flowchart in Figure 7-3. First, it makes the mandatory access check 2. We'll describe what this check does in the next section. If the check fails, then the function completes with `STATUS_ACCESS_DENIED`. To simplify the code, the full script defines a helper function, `Get-AccessResult` to build the result of the access check. Listing 7-3 shows this function definition.

```
function Get-AccessResult {
    param(
        $Status,
        $Privileges = @(),
        $GrantedAccess = 0
    )

    $props = @{
      Status = Get-NtStatus -Name $Status -PassStatus
      GrantedAccess = $GrantedAccess
      Privileges = $Privileges
    }
    return [PSCustomObject]$props
}
```

Listing 7-3 Implementing `Get-AccessResult` helper function

Next, the token access check will update the `RemainingAccess` property in the context 3. If `RemainingAccess` becomes empty, then we can conclude we've been granted all access rights and can return `STATUS_SUCCESS` 4.

We must now make a second check: if the caller requested `AccessSystemSecurity` and the token check didn't grant that right, we need to fail 5. Finally, we perform the discretionary access check. As with the token access check, we check the `RemainingAccess` property: if it's empty, the caller has received all the accesses they've requested 6. Otherwise, they've been denied access 8. Let's delve into the details of each check in turn.

Performing the Mandatory Access Check

Windows Vista introduced a feature called *mandatory integrity control (MIC)* that uses the token's integrity level, as well as the mandatory label ACE, to control resource access based on a general policy. MIC is a type of *mandatory access check (MAC)*. The key behavior of a MAC is that it cannot grant access to resource; it can only deny access. If the caller requests more access than the policy permits, the access check will immediately deny access, and if the MAC denies access, the DACL will never be checked. Because there is no way for a non-privileged user to circumvent the check, it's considered mandatory.

In the latest version of Windows, the access-check process performs two additional checks along with the MIC. These checks implement similar behavior, and so we'll group them together. Listing 7-4 define the `Test-MandatoryAccess` function we called in Listing 7-2.

```
function Test-MandatoryAccess {
    param($Context)

    ## Test-ProcessTrustLevel is defined below.
    if (!(Test-ProcessTrustLevel $Context)) {
        return $false
    }

    ## Test-ProcessTrustLevel is defined below.
    if (!(Test-AccessFilter $Context)) {
        return $false
    }

    ## Test-ProcessTrustLevel is defined below.
    if (!(Test-MandatoryIntegrityLevel $Context)) {
        return $false
    }

    return $true
}
```

Listing 7-4 Implementing the `Test-MandatoryAccess` function

This function performs three checks: `Test-ProcessTrustLevel`, `Test-AccessFilter`, and `Test-MandatoryIntegrityLevel`. If any of these checks fail,

then the entire access-check process fails, returning `STATUS_ACCESS_DENIED`. Let's detail each check in turn.

The Process Trust Level Check

Windows Vista introduced *protected processes*, which are processes that even an administrator can't manipulate and compromise. The original purpose of protected processes was to protect media content. However, Microsoft has since expanded them to cover a range of uses, such as protecting anti-virus services and virtual machines.

A token can be assigned a *process trust-level SID*. This SID depends on the protection level of a protected process and is assigned when such a process is created. To restrict access to a resource, the access-check process consults a defined order of these process trust-level SIDs. When one SID is considered more trusted than another, it's said to *dominate*.

To check whether one process trust-level SID dominates another, you can call the `RtlSidDominatesForTrust` API or the `Compare-NtSid` command with the `Dominates` parameter. Listing 7-5 translates the algorithm for checking the process trust label into PowerShell.

```
function Test-ProcessTrustLevel {
    param($Context)

  1 $trust_level = Get-NtTokenSid $Token -TrustLevel
    if ($null -eq $trust_level) {
        $trust_level = Get-NtSid -TrustType None -TrustLevel None
    }

  2 $access = Get-NtAccessMask 0xFFFFFFFF
    $sacl = Get-NtSecurityDescriptorSacl $Context.SecurityDescriptor
    foreach($ace in $sacl) {
     3 if (!$ace.IsProcessTrustLabelAce -or $ace.IsInheritOnly) {
            continue
        }

     4 if (!(Compare-NtSid $trust_level $ace.Sid -Dominates)) {
            $access = Get-NtAccessMask $ace
        }
        break
    }
```

```
    $access = Grant-NtAccessMask $access AccessSystemSecurity
  5 return Test-NtAccessMask $access $Context.RemainingAccess -All
}
```

Listing 7-5 The process trust-level checking algorithm

To check the process trust level, we need to query the SID for the current token 1. If the token does not have a trust-level SID, then we define the lowest possible SID. Next, we initialize an access mask to all bits set 2.

We then enumerate the values in the SACL, checking any process trust-label ACE other than `InheritOnly` 3. When we find a relevant ACE, we compare its SID to the SID queried for the token 4. If the ACE SID dominates, then the token has a lower protection level, and the access mask is set to the value from the ACE.

Finally, we compare the access mask to the remaining access the caller requested 5. If the access mask doesn't contain all remaining access, then the function returns `False`, which indicates that the process trust-level check failed. Note that the check always adds `AccessSystemSecurity`, regardless of the mask in the ACE.

Let's test the behavior of the process trust-label ACE. Rather than create a new protected process, we'll use the process trust-level SID of the anonymous user's token for the access check. To simplify testing, we'll define a helper function that we can reuse. This function in Listing 7-6 will create a default security descriptor that grants access to both the current user and the anonymous user. Whenever we need a security descriptor for a test, we can call this function and use the returned value.

```
PS> function New-BaseSD {
  $owner = Get-NtSid -KnownSid LocalSystem
  $sd = New-NtSecurityDescriptor -Owner $owner -Group $owner -Type Mutant
  Add-NtSecurityDescriptorAce $sd -KnownSid Anonymous -Access GenericAll
  $sid = Get-NtSid
  Add-NtSecurityDescriptorAce $sd -Sid $sid -Access GenericAll
  Set-NtSecurityDescriptorIntegrityLevel $sd Untrusted
  Edit-NtSecurityDescriptor $sd -MapGeneric
  return $sd
```

```
}
```

Listing 7-6 Defining a helper function for testing

The `New-BaseSD` function creates a basic security descriptor with the owner and group set to the *SYSTEM* user. It then adds an `Allowed` ACE for the anonymous and current user SIDs, granting them full access. It also sets the mandatory label to the `Untrusted` integrity level; you'll learn why the integrity level is important in "The Mandatory Integrity Level Check" on page XX. Finally, it maps any generic access to `Mutant` type-specific access. Let's now test the process trust label using the code in Listing 7-7.

```
1 PS> $sd = New-BaseSD
  PS> $trust_sid = Get-NtSid -TrustType ProtectedLight -TrustLevel Windows
  PS> Add-NtSecurityDescriptorAce $sd -Type ProcessTrustLabel -Access
  ModifyState
  2 -Sid $trust_sid
  PS> Get-NtGrantedAccess $sd -AsString
3 ModifyState

  PS> $token = Get-NtToken -Anonymous
  PS> $anon_trust_sid = Get-NtTokenSid -Token $token -TrustLevel
  4 PS> Compare-NtSid $anon_trust_sid $trust_sid -Dominates
  True
  PS> Get-NtGrantedAccess $sd -Token $token -AsString
  5 Full Access
```

Listing 7-7 Testing the process trust label ACE

The first thing we do is create our base security descriptor 1 and add a process trust label, granting `ModifyState` access only to tokens whose process trust level does not dominate the process trust label 2. When we run the access check, we see that the effective token, which doesn't have any process trust level, gets `ModifyState` access only 3, indicating that the process trust label is being enforced.

Now, using `Get-NtToken`, we can get a handle to an anonymous user's token, query its process trust-level SID, and compare it to the SID we added to the security descriptor 4. The call to `Compare-NtSid` returns `True`, which indicates the token's process trust level SID dominates the one in the security descriptor. To confirm this, we run the access check and find that

the anonymous user's token is granted `Full Access`, which means the process trust label did not limit its access [5].

You might wonder whether you could impersonate the anonymous token to bypass the process trust label. Remember that, in user-mode, we're calling `NtAccessCheck`, which takes only a single `Token` handle, but that the kernel's `SeAccessCheck` takes both a primary token and an impersonation token. Before the kernel verifies the process trust label, it checks both tokens and chooses the one with the lower trust level. Therefore, if the impersonation token is trusted but your primary token is untrusted, the effective trust level will be untrusted.

Windows applies a secondary security check when assigning the process trust-label ACE to a resource. While you need only `WriteDac` access to set the process trust label, you cannot change or remove the ACE if your effective trust level does not dominate the label's trust level. This prevents you from setting a new, arbitrary process trust-label ACE. Microsoft uses this ability to check certain files related to Windows applications for modifications and verify that the files were created by a protected process.

The Access Filter ACE

The second mandatory access check is the access filter ACE. It works in a manner similar to the process trust label, except that instead of using a process trust level to determine whether to apply a restricting access mask, it uses a conditional expression that evaluates to either `True` or `False`. If the conditional evaluates to `False`, the ACE's access mask limits the maximum granted access for the access check; if it evaluates to `True`, the access filter is ignored.

You can have multiple access filter ACEs in the SACL. Every conditional expression that evaluates to `False` removes more of the access mask. Therefore, if you match one ACE, but don't match a second ACE that restricts to `GenericRead`, you'll get a

maximum access of `GenericRead`. We can express this logic in a PowerShell function, as shown in Listing 7-8.

```
function Test-AccessFilter {
    param($Context)

    $access = Get-NtAccessMask 0xFFFFFFFF
    $sacl = Get-NtSecurityDescriptorSacl $Context.SecurityDescriptor
    foreach($ace in $sacl) {
        if (!$ace.IsAccessFilterAce -or $ace.IsInheritOnly) {
            continue
        }
      1 if (!(Test-NtAceCondition $ace -Token $token)) {
          2 $access = $access -band $ace.Mask
        }
    }

    $access = Grant-NtAccessMask $access AccessSystemSecurity
  3 return Test-NtAccessMask $access $Context.RemainingAccess -All
}
```

Listing 7-8 The access filter check algorithm

This algorithm resembles the one we implemented to check the process trust level. The only difference is that we check a conditional expression rather than the SID 1. The function supports multiple access filter ACEs, and for each matching ACE, the access mask is bitwise ANDed with the final access mask, which starts with all access mask bits set 3. As the masks are ANDed, each ACE can only remove access, not add it. Once we've checked all ACEs, we check the remaining access to determine whether the check succeeded or failed 3.

In Listing 7-9, we check the behavior of the actual access-filter algorithm to ensure it works as expected.

```
PS> $sd = New-BaseSD
PS> Add-NtSecurityDescriptorAce $sd -Type AccessFilter -KnownSid World `
  1 -Access ModifyState -Condition "Exists TSA://ProcUnique" -MapGeneric
PS> Format-NtSecurityDescriptor $sd -Summary -SecurityInformation AccessFilter
<Access Filters>
Everyone: (AccessFilter)(None)(ModifyState)(Exists TSA://ProcUnique)

2 PS> Show-NtTokenEffective -SecurityAttributes
SECURITY ATTRIBUTES
-------------------
Name             Flags                 ValueType Values
----             -----                 --------- ------
```

```
  TSA://ProcUnique NonInheritable, Unique UInt64    {187, 365588953}

  PS> Get-NtGrantedAccess $sd -AsString
3 Full Access

  PS> Use-NtObject($token = Get-NtToken -Anonymous) {
      Get-NtGrantedAccess $sd -Token $token -AsString
  }
4 ModifyState
```

Listing 7-9 Testing the access filter ACE

We add the `AccessFilter` ACE to the security descriptor with the conditional expression `"Exists TSA://ProcUnique"` 1. The expression checks whether the `TSA://ProcUnique` security attribute is present in the token. For a normal user, this check should always return `True`; however, the attribute doesn't exist in the anonymous user's token. We set the mask to be `ModifyState` and the SID to the `Everyone` group. Note that the SID isn't verified, so it can have any value, but using the `Everyone` group is conventional.

We can check the current effective token's security attributes using `Show-NtTokenEffective` 2. Getting the maximum access for the effective token results in `Full Access`, meaning the access filter check passes without restricting access 3. However, when use the anonymous user's token, the access filter check fails, and the access is restricted to `ModifyState` only 4.

To set an access filter, you need only `WriteDac` access. So, what's to prevent a user removing the filter? Obviously, the access filter shouldn't grant `WriteDac` in the first place, but if it does, you can limit any changes to a protected-process trust level. To do this, set the ACE SID to a process trust-level SID, and set the `TrustProtected` ACE flag. Now a caller with a lower process trust level won't be able to remove or modify the access filter ACE.

The Mandatory Integrity Level Check

Finally, we'll implement the mandatory integrity level check. In the SACL, a mandatory label ACE's SID represents the

security descriptor's integrity level. Its mask, which expresses the mandatory policy, combines the `NoReadUp`, `NoWriteUp` and `NoExecuteUp` policies to determine the maximum access the system can grant the caller based on the `GenericRead`, `GenericWrite`, and `GenericExecute` values from the generic mapping structure.

To determine whether to enforce the policy, the check compares the integrity level SIDs of the security descriptor and token. If the token's SID dominates the security descriptor's, then no policy is enforced, and any access is permitted. However, if the token's SID doesn't dominate, then any access requested outside of the value for the policy causes the access check to fail with `STATUS_ACCESS_DENIED`.

Calculating whether one integrity level SID dominates another is much simpler than calculating the equivalent value for the process trust-level SID. To do so, we extract the last RID from each SID and compare these as numbers. If one integrity level SID's RID level is greater than or equal to the other, it dominates.

However, calculating the access mask for the policy based on the generic mapping is much more involved, as it requires a consideration of shared access rights. I won't implement the code for calculating the access mask, as we can use an option on `Get-NtAccessMask` to calculate it for us.

In Listing 7-10, we implement the mandatory integrity level check.

```
function Test-MandatoryIntegrityLevel {
    param($Context)

    $token = $Context.Token
    $sd = $Context.SecurityDescriptor
    $mapping = $Context.GenericMapping

  1 $policy = Get-NtTokenMandatoryPolicy -Token $token
    if (($policy -band "NoWriteUp") -eq 0) {
        return $true
    }

    if ($sd.HasMandatoryLabelAce) {
```

```
            $ace = $sd.GetMandatoryLabel()
            $sd_il_sid = $ace.Sid
          2 $access = Get-NtAccessMask $ace.Mask -GenericMapping $mapping
        } else {
          3 $sd_il_sid = Get-NtSid -IntegrityLevel Medium
            $access = Get-NtAccessMask -ManadatoryLabelPolicy NoWriteUp `
                    -GenericMapping $GenericMapping
        }

      4 if (Test-NtTokenPrivilege -Token $token SeRelabelPrivilege) {
            $access = Grant-NtAccessMask $access WriteOwner
        }

        $il_sid = Get-NtTokenSid -Token $token -Integrity
        if (Compare-NtSid $il_sid $sd_il_sid -Dominates) {
            return $true
        }

        return Test-NtAccessMask $access $Context.RemainingAccess -All
    }
```

Listing 7-10 Implementing the mandatory integrity level check algorithm

We start by checking the token's mandatory policy 1. In this case, we check if the `NoWriteUp` flag is set or not. If the flag is not set, then we disable integrity level checking for this token and return `True`. This flag is rarely turned off, however, and it requires `SeTcbPrivilege` to disable, so in almost all cases, the integrity level check will continue.

Next, we need to capture the security descriptor's integrity level and mandatory policy from the mandatory label ACE. If the ACE exists, we extract these values and map the policy to the maximum access mask using `Get-NtAccessMask` 2. If the ACE doesn't exist, the algorithm uses a `Medium` integrity level and a `NoWriteUp` policy by default 3.

If the token has the `SeRelabelPrivilege` privilege, we add the `WriteOwner` access back to the maximum access, even if the policy removed it 4. This allows a caller with `SeRelabelPrivilege` to change the security descriptor's mandatory integrity label ACE.

Then, we query the token's integrity level SID and compare it to the security descriptor's. If the token's SID dominates, then the

check passes and allows any access. Otherwise, the calculated policy access mask must grant the entirety of the remaining access mask requested. Note that we don't treat `AccessSystemSecurity` differently here, as we did in the process trust level or access filter checks. We remove it if the policy contains `NoWriteUp`, the default for all resource types.

Let's verify the behavior of the mandatory integrity level check by considering the real access-check process (Listing 7-11).

```
PS> $sd = New-BaseSD
PS> Format-NtSecurityDescriptor $sd -SecurityInformation Label -Summary
<Mandatory Label>
1 Mandatory Label\Untrusted Mandatory Level: (MandatoryLabel)(None)(NoWriteUp)

PS> Use-NtObject($token = Get-NtToken -Anonymous) {
        Format-NtToken $token -Integrity
        Get-NtGrantedAccess $sd -Token $token -AsString
    }
2 INTEGRITY LEVEL
  ---------------
  Untrusted
3 Full Access

4 PS> Remove-NtSecurityDescriptorIntegrityLevel $sd
  PS> Use-NtObject($token = Get-NtToken -Anonymous) {
         Get-NtGrantedAccess $sd -Token $token -AsString
    }
5 ModifyState|ReadControl|Synchronize
```

Listing 7-11 Testing the mandatory label ACE

We create a security descriptor and check its mandatory integrity label. We can see that it's set to the `Untrusted` integrity level, which is the lowest level, and that its policy is `NoWriteUp` 1. We then get the maximum access for the anonymous user's token. We can show that that the token has an integrity level of `Untrusted` 2. As this integrity level matches the security descriptor's integrity level, the token is allowed full access 3.

To test access mask restrictions, we remove the mandatory label ACE from the security descriptor so that the access check will default to the `Medium` integrity level 4. Running the check

again, we now get `ModifyState|ReadControl|Synchronize`, which is the `Mutant` object's full access without the `GenericWrite` access mask [5].

This concludes the implementation of the mandatory access check. We've seen that this algorithm is really three separate checks for integrity, process trust level, and the access filter. Each check can only deny access; it never grants additional access.

Performing the Token Access Check

The second main check, the *token access check*, uses properties of the caller's token to determine whether to grant certain specific access rights. More specifically, it checks for any special privileges, as well as for the owner of the security descriptor.

Unlike the mandatory access check, the token access check can grant access to a resource if it has removed all bits from the token's access mask. Listing 7-12 implements the top-level `Result-TokenAccess` function.

```
Function Result-TokenAccess {
    param($Context)

    Resolve-TokenPrivilegeAccess $Context
    if (Test-NtAccessMask $Context.RemainingAccess -Empty) {
        return
    }
    return Resolve-TokenOwnerAccess $Context
}
```

Listing 7-12 Token access check algorithm

The check is simple. First, we check the token's privileges using a function we'll define next, `Resolve-TokenPrivilegeAccess`, passing it the current context. If certain privileges are enabled, this function modifies the token's remaining access, and if the remaining access is empty, meaning no access remains to be granted, we can return immediately. We then call `Resolve-TokenOwnerAccess`, which checks

whether the token owns the resource and can also update `RemainingAccess`. Let's dig into these individual checks.

The Privilege Check

The *privilege check* determines whether the `Token` object has three different privileges enabled. If one of these privileges is enabled, we'll grant an access mask and the bits from the remaining access (Listing 7-13).

```
function Resolve-TokenPrivilegeAccess {
    param($Context)

    $token = $Context.Token
    $access = $Context.RemainingAccess

    if ((Test-NtAccessMask $access AccessSystemSecurity) -and
      1 (Test-NtTokenPrivilege -Token $token SeSecurityPrivilege)) {
        $access = Revoke-NtAccessMask $access AccessSystemSecurity
        $Context.Privileges += "SeSecurityPrivilege"
    }

    if ((Test-NtAccessMask $access WriteOwner) -and
        (Test-NtTokenPrivilege -Token $token SeTakeOwnershipPrivilege)) {
        $access = Revoke-NtAccessMask $access WriteOwner
        $Context.Privileges += "SeTakeOwnershipPrivilege"
    }

    if ((Test-NtAccessMask $access WriteOwner) -and
        (Test-NtTokenPrivilege -Token $token SeRelabelPrivilege)) {
        $access = Revoke-NtAccessMask $access WriteOwner
        $Context.Privileges += "SeRelabelPrivilege"
    }

  2 $Context.RemainingAccess = $access
}
```

Listing 7-13 The token-privilege checking algorithm

First, we check whether the caller has requested `AccessSystemSecurity`; if so, and if `SeSecurityPrivilege` is enabled, we remove `AccessSystemSecurity` from the remaining access 1. We also update the list of privileges we've used so that we can return it to the caller.

Next, we perform similar checks for `SeTakeOwnershipPrivilege` and `SeRelabelPrivilege` and remove `WriteOwner` from the remaining access if they're enabled. Lastly, we update the `RemainingAccess` value with the final access mask (2).

Granting `WriteOwner` to both `SeTakeOwnershipPrivilege` and `SeRelabelPrivilege` makes sense from the kernel's perspective, as you need `WriteOwner` to modify the owner SID and integrity level. However, this implementation also means that a token with only `SeRelabelPrivilege` can take ownership of the resource, which we might not always intend. Fortunately, even administrators don't get `SeRelabelPrivilege` by default, making this a minor issue.

Let's check this function against the real access check process. Run the script in Listing 7-14 as an administrator.

```
PS> $owner = Get-NtSid -KnownSid Null
PS> $sd = New-NtSecurityDescriptor -Type Mutant -Owner $owner
(1) -Group $owner -EmptyDacl
(2) PS> Set-NtTokenPrivilege SeTakeOwnershipPrivilege
(3) PS> Get-NtGrantedAccess $sd -Access WriteOwner -PassResult
Status                Granted Access Privileges
------                -------------- ----------
STATUS_SUCCESS        WriteOwner     (4) SeTakeOwnershipPrivilege

(5) PS> Set-NtTokenPrivilege SeTakeOwnershipPrivilege -Disable
PS> Get-NtGrantedAccess $sd -Access WriteOwner -PassResult
Status                Granted Access Privileges
------                -------------- ----------
(6) STATUS_ACCESS_DENIED None           NONE
```

Listing 7-14 Testing the token privilege check

Listing 7-14 starts by creating a security descriptor that should grant no access to the current user (1). We then enable `SeTakeOwnershipPrivilege` (2). Next, we request an access check for `WriteOwner` access and specify the `PassResult` parameter, which outputs the full access check result (3). The result shows that the access check succeeded, granting `WriteOwner` access, but also that the check used the `SeTakeOwnershipPrivilege` (4). To verify that we weren't

granted `WriteOwner` for another reason, we disable the privilege 5 and rerun the check. The check now denies us access 6.

The Owner Check

The *owner check* exists to grant `ReadControl` and `WriteDac` access to the owner of the resource, even if the DACL doesn't grant that owner any other access. The purpose of this check is to prevent a user from locking themselves out of their own resources. If they accidentally change the DACL so that they no longer have access, they can still use `WriteDac` access to return the DACL to its previous state.

The check compares the owner SID in the security descriptor with all enabled token groups (not just the token owner), checking them for the user's SID. We demonstrated this behavior at the start of this chapter, in Listing 7-1. In Listing 7-15, we'll implement the `Resolve-TokenOwnerAccess` function.

```
function Resolve-TokenOwnerAccess {
    param($Context)

    $token = $Context.Token
    $sd = $Context.SecurityDescriptor
    $sd_owner = Get-NtSecurityDescriptorOwner $sd
  1 if (!(Test-NtTokenGroup -Token $token -Sid $sd_owner.Sid)) {
        return
    }

    $sids = Select-NtSecurityDescriptorAce $sd
2 -KnownSid OwnerRights -First -AclType Dacl
    if ($sids.Count -gt 0) {
        return
    }

    $access = $Context.RemainingAccess
  3 $Context.RemainingAccess = Revoke-NtAccessMask $access ReadControl,
WriteDac
}
```

Listing 7-15 The token owner access-check algorithm

We use the `Test-NtTokenGroup` to check whether the security descriptor's owner SID is an enabled member of the

token 1. If the owner SID is not a member, we simply return. If it is an owner, the code then needs to check whether there are any *OWNER RIGHTS* SIDs (S-1-3-4) in the DACL 2. If there are, then we don't follow the default process, and instead rely on the DACL check to grant access to the owner. Finally, if both checks pass, we can remove `ReadControl` and `WriteDac` from the remaining access 3.

In Listing 7-16, we verify this behavior in the real access-check process.

```
1 PS> $owner = Get-NtSid -KnownSid World
  PS> $sd = New-NtSecurityDescriptor -Owner $owner -Group $owner
      -Type Mutant -EmptyDacl
  PS> Get-NtGrantedAccess $sd
2 ReadControl, WriteDac

  PS> Add-NtSecurityDescriptorAce $sd -KnownSid OwnerRights -Access ModifyState
  PS> Get-NtGrantedAccess $sd
  ModifyState
```

Listing 7-16 Testing the owner check process

We start by creating a security descriptor with the owner and group set to *Everyone* 1. We also create a security descriptor with an empty DACL, which means the access check process will consider only the owner check when calculating the granted access. When we run the access check, we get `ReadControl` and `WriteDac` 2.

We then add a single ACE with the *OWNER RIGHTS* SID. This disables the default owner access and causes the access check to grant only the access specified in the ACE (in this case, `ModifyState`). When we run the access check again, we now find that the only granted access is `ModifyState` and that we no longer have `ReadControl` or `WriteDac`.

This concludes the token access check. As we demonstrated, the algorithm can grant certain access rights to a caller before any significant processing of the security descriptor takes place. This is primarily to allow users to maintain access to their own resources, and for administrators to take ownership of other user's files. Let's continue to the final check.

Performing the Discretionary Access Check

We've relied on the behavior of the DACL for a few of our tests. Now we'll explore exactly how the DACL check works. Checking the DACL may seem simple, but the devil is in the details. Listing 7-17 implements the algorithm:

```
function Get-DiscretionaryAccess {
    param($Context)

    $token = $Context.Token
    $sd = $Context.SecurityDescriptor
    $access = $Context.RemainingAccess
    $resource_attrs = $null
    if ($sd.ResourceAttributes.Count -gt 0) {
        $resource_attrs = $sd.ResourceAttributes.ResourceAttribute
    }

  1 if (!(Test-NtSecurityDescriptor $sd -DaclPresent) `
        -or (Test-NtSecurityDescriptor $sd -DaclNull)) {
        $Context.RemainingAccess = Get-NtAccessMask 0
        return
    }

    $owner = Get-NtSecurityDescriptorOwner $sd
    $dacl = Get-NtSecurityDescriptorDacl $sd
  2 foreach($ace in $dacl) {
    3 if ($ace.IsInheritOnly) {
        continue
      }
    4 $sid = Get-AceSid $ace -Owner $owner
      $continue_check = $true
      switch($ace.Type) {
        "Allowed" {
        5 if (Test-NtTokenGroup -Token $token $sid) {
            $access = Revoke-NtAccessMask $access $ace.Mask
          }
        }
        "Denied" {
          6 if (Test-NtTokenGroup -Token $token $sid -DenyOnly) {
              if (Test-NtAccessMask $access $ace.Mask) {
                $continue_check = $false
            }
          }
        }
        "AllowedCompound" {
          $server_sid = Get-AceSid $ace -Owner $owner
        7 if ((Test-NtTokenGroup -Token $token $sid)
-and (Test-NtTokenGroup -Sid $server_sid)) {
            $access = Revoke-NtAccessMask $access $ace.Mask
          }
```

```
            }
            "AllowedCallback" {
              if ((Test-NtTokenGroup -Token $token $sid)
-and (Test-NtAceCondition $ace -Token $token
8 -ResourceAttributes $resource_attrs)) {
                  $access = Revoke-NtAccessMask $access $ace.Mask
                }
              }
          }

      9 if (!$continue_check -or (Test-NtAccessMask $access -Empty)) {
          break
        }
      }

    0 $Context.RemainingAccess = $access
}
```

Listing 7-17 The discretionary access-check algorithm

We begin by checking whether the DACL is present; if it is, we check whether it's a NULL ACL 1. If there is no DACL or only a NULL ACL, there is no security to enforce, so the function clears the remaining access and returns, granting the token any access to the resource that the mandatory access check hasn't restricted.

Once we've confirmed that there is a DACL to check, we can enumerate each of its ACEs 2. If an ACE is `InheritOnly`, it won't take part in the check, so we ignore it 3. Next, we need to map the SID in the ACE to the SID we're checking using a helper function we'll define next, `Get-AceSid` 4. This function converts the *OWNER RIGHTS* SID for the ACE to the current security descriptor's owner, as shown in Listing 7-18.

```
function Get-AceSid {
    param(
        $Ace,
        $Owner
    )

    $sid = $Ace.Sid
    if (Compare-NtSid $sid -KnownSid OwnerRights) {
        $sid = $Owner.Sid
    }

    return $sid
}
```

Listing 7-18 The implementation of `Get-AceSid`

With the SID in hand, we can now evaluate each ACE based on its type. For the simplest type, `Allowed`, we check whether the SID is in the token's `Enabled` groups. If so, we grant the access represented by the ACE's mask and can remove those bits from remaining access 5.

The `Denied` type also checks whether the SID is in the token's groups; however, it must include both `Enabled` and `DenyOnly` groups, so we pass the `DenyOnly` parameter 6. Note that it's possible to configure the token user SID as a `DenyOnly` group as well, and `Test-NtTokenGroup` takes this into account. A `Denied` ACE doesn't modify the remaining access; instead, the function compares the mask against the current remaining access, and if any bit of remaining access is also set in the mask, then the function denies that access and immediately returns the remaining access.

The final two ACE types we'll cover are variations on the `Allowed` type. The first, `AllowedCompound`, contains the additional server SID. To perform this check, the function compares both the normal SID and the server SID with the caller token's groups, as these values might be different 7. (Note that the server SID should be mapped to the owner if the *OWNER RIGHTS* SID is used.) The ACE condition is met only if both SIDs are enabled.

Finally, we check the `AllowedCallback` ACE type. To do so, we again check the SID, as well as whether a conditional expression matches the token using `Test-NtAceCondition` 8. If the expression returns `True`, the ACE condition is met, and we remove the mask from the remaining access. To fully implement the conditional check, we also need to pass in any resource attributes from the security descriptor; we'll describe resource attributes in more detail in the Central Access Policy section later in the chapter. Notice, we're intentionally not checking `DenyCallback`, as the kernel does not support `DenyCallback` ACEs, although the user-mode-only `AuthzAccessCheck` API does.

After we've processed the ACE, we check the remaining access 9. If the remaining access is empty, we've been granted the entire requested access and can stop processing ACEs. This is why we have a canonical ACL ordering. If `Denied` ACEs were placed after `Allowed` ACEs, the remaining access could become empty, and the loop might exit before ever checking a `Denied` ACE.

Lastly, this function sets the `RemainingAccess` 0. If the value of `RemainingAccess` is non-empty, the access check fails with access denied. Therefore, an empty DACL blocks all access; if there are no ACEs, the `RemainingAccess` never changes, so it won't be empty at the end of the function.

We've covered all three access checks, and you should now have a better understanding of their structure. However, there is more to the access check process. In the next section, we discuss how the access check process supports the implementation of sandboxes.

Sandboxing

In Chapter 4, we covered two types of sandbox tokens: restricted and lowbox. These sandbox tokens modify the access check process by adding additional checks. Let's discuss each token type in more detail, starting with restricted tokens.

Restricted Tokens

Using a restricted token affects the access-checking process by introducing a second owner and a discretionary access check against the list of restricted SIDs. In Listing 7-15, we modify the owner SID check in the `Resolve-TokenOwnerAccess` function to account for this.

```
1 if (!(Test-NtTokenGroup -Token $token -Sid $sd_owner.Sid)) {
      return
  }

  if ($token.Restricted -and
  2 !(Test-NtTokenGroup -Token $token -Sid $sd_owner.Sid -Restricted)) {
```

```
        return
    }
```

Listing 7-19 The modified `Get-TokenOwner` access check for restricted tokens

We first perform the existing SID check 1. If the owner SID isn't in the list of token groups, then we don't grant `ReadControl` or `WriteDac` access. At 2 is the additional check: if the token is restricted, then we also check the list of restricted SIDs for the owner SID, and grant the token `ReadControl` and `WriteDac` access only if the owner SID is in both the main group list and the restricted SID list.

We'll follow the same pattern for the discretionary access check, although for simplicity, we'll add a Boolean `Restricted` switch parameter to the `Get-DiscretionaryAccess` function and pass it to any call to `Test-NtTokenGroup`. For example, we can modify the allowed-ACE check implemented in Listing 7-17 so it looks as shown in Listing 7-20.

```
"Allowed" {
    if (Test-NtTokenGroup -Token $token $sid -Restricted:$Restricted) {
        $access = Revoke-NtAccessMask $access $ace.Mask
    }
}
```

Listing 7-20 The modified allowed-ACE type for restricted tokens

In Listing 7-20, we set the `Restricted` parameter to the value of a parameter passed into `Get-DiscretionaryAccess`. We now need to modify the `Get-PSGrantedAccess` function defined in Listing 7-2 to call `Get-DiscretionaryAccess` twice for a restricted token (Listing 7-21).

```
1 $RemainingAccess = $Context.RemainingAccess
  Get-DiscretionaryAccess $Context
2 $success = Test-NtAccessMask $Context.RemainingAccess -Empty

3 if ($success -and $Token.Restricted) {
    4 if (!$Token.WriteRestricted -or
  (Test-NtAccessMask $RemainingAccess -WriteRestricted $GenericMapping)) {
        $Context.RemainingAccess = $RemainingAccess
      5 Get-DiscretionaryAccess $Context -Restricted
        $success = Test-NtAccessMask $Context.RemainingAccess -Empty
```

```
        }
    }

6 if ($success) {
        return Get-AccessResult STATUS_SUCCESS $Context.Privileges $DesiredAccess
    }
    return Get-AccessResult STATUS_ACCESS_DENIED
```

Listing 7-21 The `Get-PSGrantedAccess` function modified to account for restricted tokens

We first capture the existing `RemainingAccess` value, as the discretionary access check will modify it, and we want to repeat that check a second time 1. We then run the discretionary access check and save the result in a variable 2. If this first check succeeded and the token is restricted, we must perform a second check 3. We also need to consider whether the token is write-restricted, and whether the remaining access includes write access 4. We look for write access by checking the passed generic mapping. (Note that the owner check doesn't perform a write check, so in theory it could grant the token `WriteDac` access, which is considered a write access).

We run the check a second time, this time with the `Restricted` parameter to indicate that the restricted SIDs should be checked 5. If this second check also passes, we set the success variable to `True` and grant access to the resource 6.

Keep in mind that the restricted SID check applies to both allow and deny ACE types. This means that if the DACL contains a denied ACE that references a SID in the restricted SID list, the function will deny access, even if the SID isn't in the normal group list.

Lowbox Tokens

The access check process for a lowbox token resembles that for a restricted token. A lowbox token can contain a list of capability SIDs used to perform a second check, similar to the check we performed with the list of restricted SIDs. Likewise, if the access check process doesn't grant access through both normal and capability checks, the access check fails. However,

the lowbox token's access check contains some subtle differences:

- It will consider the token's package SID in addition to its list of capability SIDs.
- The checked capability SIDs must have the enabled attribute flag set to be considered active.
- The check applies only to `Allowed` ACE types, not to `Denied` ACE types.
- NULL DACLs do not grant full access.

In addition, two special package SIDs will match any token's package SID for the purposes of the package SID check:

- `ALL APPLICATION PACKAGES` (S-1-15-2-1)
- `ALL RESTRICTED APPLICATION PACKAGES` (S-1-15-2-2)

You might wonder why we need these two separate special package SIDs. Well, Windows 8 originally included only the `ALL APPLICATION PACKAGES` special SID, but during the development of the Edge web browser, Microsoft realized that many secured resources used it to allowed access, increasing the attack surface.

To combat this expanded attack surface, Microsoft decided that the access check would ignore the `ALL APPLICATION PACKAGES` SID if the token had the `WIN://NOALLAPPPKG` security attribute with a single value of 1. In those cases, it would consider only the `ALL RESTRICTED APPLICATION PACKAGES` SID. If the security attribute wasn't present or was set to 0, the access check would consider both SIDs. Microsoft refer to processes with this security attribute as running a *less privileged AppContainer (LPAC)*.

Because setting a token's security attribute requires `SeTcbPrivilege`, the process-creation APIs have an option for applying the attribute to a new process. Listing 7-22 shows a

basic implementation of the lowbox access check for `Allowed` ACE types. You should add this code to the end of discretionary access check in Listing 7-17.

```
## Add to start of Get-DiscretionaryAccess
$ac_access = $context.DesiredAccess
if (!$token.AppContainer) {
   $ac_access = Get-NtAccessMask 0
}

## Add in switch ACE switch statement.
"Allowed" {
    if (Test-NtTokenGroup -Token $token $sid -Restricted:$Restricted) {
      1 $access = Revoke-NtAccessMask $access $ace.Mask
    } else {
      2 if ($Restricted) {
            break
      }

      3 if (Test-NtTokenGroup -Token $token $sid -Capability) {
          4 $ac_access = Revoke-NtAccessMask $ac_access $ace.Mask
      }
    }
}

## Add at end of ACE loop.
$effective_access = $access -bor $ac_access
```

Listing 7-22 An implementation of the lowbox access check for `Allowed` ACEs

The first test verifies whether the SID is in the normal group (and thus not a restricted SID). If it finds the group, it removes the mask from the remaining access check 1. If the group test fails, we check whether it's a package or capability SID. We must ensure that we're not checking whether we're in the restricted SID mode, as this mode doesn't define lowbox checks 2.

Our check for the capability SIDs includes the package SID and the `ALL APPLICATION PACKAGES` SIDs 3. If we find a match, we remove the mask from the remaining access 4. However, we need to maintain separate remaining access values for normal SIDs and AppContainer SIDs. Therefore, we create two variables, `$access` and `$ac_access`. We initialize the `$ac_access` variable to the value the original `DesiredAccess`, not the current remaining access, as we won't grant owner rights such as `WriteDac` unless the SID also

matches an `Allowed` package or capability SID ACE. We also modify the loop's exit condition to consider both remaining access values; they must both be empty before we exit.

We also add some additional checks to better isolate AppContainer processes from existing `Low`-integrity-level sandboxes such as Internet Explorer's protected mode. After all, AppContainer processes should have strong isolation mechanisms, so why not try to isolate them from `Low`-integrity-level sandbox processes?

The first change we implement affects the mandatory access check. If the check fails for a lowbox token, we then check the security descriptor's integrity level a second time. If the integrity level is less than or equal to `Medium`, we assume that the check succeeds. This is despite the fact that lowbox tokens have a `Low` integrity level, as demonstrated in Chapter 4. This behavior blocks access to any resources that the lowbox token created, as these would have inherited the token's `Low` integrity level.

In Listing 7-23, we check this behavior.

```
1 PS> $sd = New-NtSecurityDescriptor -Owner "BA" -Group "BA" -Type Mutant
  PS> Add-NtSecurityDescriptorAce $sd -KnownSid World -Access GenericAll
  PS> Add-NtSecurityDescriptorAce $sd -KnownSid AllApplicationPackages
      -Access GenericAll
  PS> Edit-NtSecurityDescriptor $sd -MapGeneric
2 PS> Set-NtSecurityDescriptorIntegrityLevel $sd Medium

  PS> Use-NtObject($token = Get-NtToken -Duplicate -IntegrityLevel Low) {
      Get-NtGrantedAccess $sd -Token $token -AsString
  }
3 ModifyState|ReadControl|Synchronize

  PS> $sid = Get-NtSid -PackageName "mandatory_access_lowbox_check"
  PS> Use-NtObject($token = Get-NtToken -LowBox -PackageSid $sid) {
      Get-NtGrantedAccess $sd -Token $token -AsString
  }
4 Full Access
```

Listing 7-23 Verifying the behavior of a mandatory access check against a lowbox token

We start by building a security descriptor that grants `GenericAll` access for the `Everyone` and `ALL APPLICATION PACKAGES` groups 1. We also set an explicit

integrity level of `Medium` 2, although this isn't necessary, as `Medium` is the default for security descriptors without a mandatory label ACE. We then perform an access check using a `Low`-integrity-level token, and we receive only read access to the security descriptor 3. We try the access check again with a lowbox token; although the token's integrity level is still `Low`, the token is granted `Full Access`.

The second change in behavior is this: if the DACL contains a package SID, we deny access to the `Low`-integrity-level token regardless of the security descriptor's integrity level or other groups. This mechanism blocks access to resources that are assigned the default DACL, as the package SID is added to the default DACL when a lowbox token is created. Listing 7-24 tests this behavior.

```
PS> $sid = Get-NtSid -PackageName 'package_sid_low_il_test'
1 PS> $token = Get-NtToken -LowBox -PackageSid $sid
2 PS> $sd = New-NtSecurityDescriptor -Token $token -Type Mutant
PS> Format-NtSecurityDescriptor $sd -Summary -SecurityInformation Dacl, Label
<DACL>
3 GRAPHITE\user: (Allowed)(None)(Full Access)
NT AUTHORITY\SYSTEM: (Allowed)(None)(Full Access)
NT AUTHORITY\LogonSessionId_0_109260: (Allowed)(None)(ModifyState|...)
4 package_sid_low_il_test: (Allowed)(None)(Full Access)
<Mandatory Label>
5 Mandatory Label\Low Mandatory Level: (MandatoryLabel)(None)(NoWriteUp)

PS> Get-NtGrantedAccess $sd -Token $token -AsString
6 Full Access
PS> $token.Close()

PS> $low_token = Get-NtToken -Duplicate -IntegrityLevel Low
PS> Get-NtGrantedAccess $sd -Token $low_token -AsString
7 None
```

Listing 7-24 Verifying the behavior of the package SID for `Low`-integrity-level tokens

In Listing 7-24, we start by creating a lowbox token 1. The token does not have any added capability SIDs, only the package SID. Next, we build a default security descriptor from the lowbox token 2. When inspecting the entries in the security descriptor, we see that the current user SID 3 and the package SID 4 have been granted `Full Access`. As a lowbox token has `Low` integrity

level, the security descriptor inheritance rules require the integrity level to be added to the security descriptor [5].

We then request the granted access for the security descriptor based on the lowbox token and receive `Full Access` [6]. Next, we create a duplicate of the current token but set its integrity level to `Low`. We now get a granted access of `None` [7], even though we expected to receive `Full Access` based on the integrity level ACE in the security descriptor. In this case, the presence of the package SID in the security descriptor blocked access.

One final thing to note: as the sandbox access checks are orthogonal, it's possible to create a lowbox token from a restricted token, causing both lowbox checks and restricted SID checks to occur. The resulting access is the most restrictive of all, making for a stronger sandbox primitive.

Enterprise Access Checks

Enterprise deployments of Windows often perform some additional access checks. You won't typically need these checks on standalone installations of Windows, but you should still understand how they modify the access-check process if present.

The Object-Type Access Check

For simplicity's sake, one thing I intentionally removed from the discretionary access-check algorithm was the handling of object ACEs. To support object ACEs, you must use a different access-check API: either `SeAccessCheckByType` in kernel mode or the `NtAccessCheckByType` system call. These APIs introduces two additional parameters to the access-check process:

`Principal`
An SID used to replace the *SELF* SID in ACEs

`ObjectTypes`
A list of GUIDs that are valid for the check

The `Principal` is easy to define: when we're processing the DACL and encounter an ACE's SID set to the *SELF* SID (`S-1-5-10`), we replace the SID with a value from the `Principal` parameter. Listing 7-25 shows an adjusted version of the `Get-AceSid` function that takes this into account.

```
function Get-AceSid {
    Param (
        $Ace,
        $Owner,
        $Principal
    )

    $sid = $Ace.Sid
    if (Compare-NtSid $sid -KnownSid OwnerRights) {
        $sid = $Owner
    }
    if ((Compare-NtSid $sid -KnownSid Self) -and ($null -NE $Principal)) {
        $sid = $Principal
    }
    return $sid
}
```

Listing 7-25 Adding the principal SID to the `Get-AceSid` function

You'll also have to modify the `Get-PSGrantedAccess` function to receive the `Principal` parameter by adding it to the `$Context` value.

Microsoft introduced the *SELF* SID for use in Active Directory, we'll discuss its purpose in more detail in Chapter 11. Listing 7-26 tests the behavior of the `Principal` SID.

```
PS> $owner = Get-NtSid -KnownSid LocalSystem
1 PS> $sd = New-NtSecurityDescriptor -Owner $owner -Group $owner -Type Mutant
PS> Add-NtSecurityDescriptorAce $sd -KnownSid Self -Access GenericAll -
MapGeneric
2 PS> Get-NtGrantedAccess $sd -AsString
None

PS> $principal = Get-NtSid
3 PS> Get-NtGrantedAccess $sd -Principal $principal -AsString
Full Access
```

Listing 7-26 Testing the `Principal` SID replacement

We start by creating a security descriptor with the owner and group set to the *SYSTEM* user SID and a single `Allowed` ACE

that grants the `SELF` SID `GenericAll` access 1. Based on the access-checking rules, this should not grant the user any access to the resource. We can confirm that this is the case with a call to `Get-NtGrantedAccess` 2.

Next, we get the effective token's User SID and pass it in the `Principal` parameter to `Get-NtGrantedAccess` 3. The DACL check will then replace the `SELF` SID with the `Principal` SID, which matches the current user and therefore grants `Full Access`. This check replaces SIDs in the DACL and SACL only; for example, setting *SELF* as the owner SID won't grant any access.

The other parameter, `ObjectTypes`, is much trickier to implement. The parameter is a list of GUIDs that are valid for the access-check process. Each GUID represents the type of an object to be accessed; for example, you might have a GUID associated with a computer object and a different one for a user object.

Each GUID also has an associated level, turning the list into a hierarchical tree. Each node maintains its own remaining access, which it initializes to the main `RemainingAccess` value. Active Directory uses this hierarchy to implement a concept of properties and property sets, as shown in Figure 7-4.

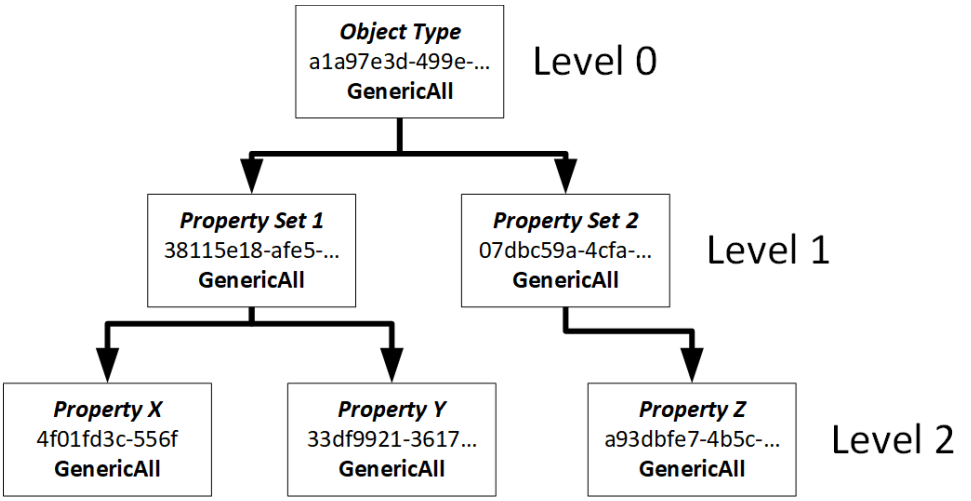

Figure 7-4 Active Directory-style properties

Each node in Figure 7-4 shows the name we've given it, a portion of the object type GUID, and the current `RemainingAccess` value (in this case, `GenericAll`). Level 0 corresponds to the top-level object, of which there can be only in the list. At level 1 are the property sets, numbered 1 and 2. Below each property set are the individual properties, at level 2.

By setting up the object types in a hierarchy, we can configure a security descriptor to grant access to multiple properties using a single ACE by setting the access on the property set. If you granted a property set some access, you'd also grant that access to all properties contained in that set. Conversely, if you were to deny access to a single property, the deny status would propagate up the tree and deny access to the entire property set and object as a whole.

Let's consider a basic implementation of object-type access. The code in Listing 7-27 relies on an `ObjectTypes` property added to the access context. We can generate the values for this parameter using the `New-ObjectTypeTree` and `Add-ObjectTypeTree` commands, whose use we'll cover in "The Object-Type Access Check" on page XX.

Listing 7-27 shows the access check implementation for the `AllowedObject` ACE type. Add it to the ACE enumeration code from Listing 7-17.

```
"AllowedObject" {
  1 if (!(Test-NtTokenGroup -Token $token $sid)) {
        break
    }

  2 if ($null -eq $Context.ObjectTypes -or $null -eq $ace.ObjectType) {
        break
    }

  3 $object_type = Select-ObjectTypeTree $Context.ObjectTypes
    if ($null -eq $object_type) {
        break
    }

  4 Revoke-ObjectTypeTreeAccess $object_type $ace.Mask
  5 $access = Revoke-NtAccessMask $access $ace.Mask
}
```

Listing 7-27 An implementation of the `AllowedObject` ACE access-check algorithm

We start with the SID check 1. If the SIDs don't match, we don't process the ACE. Next, we check whether the `ObjectTypes` property exists in the context, and whether the ACE defines an `ObjectType` 2. (In Chapter 5, you learned that the `ObjectType` on the ACE is optional.) Again, if these checks fail, we ignore the ACE. Finally, we check whether there is an entry in the `ObjectTypes` parameter for the `ObjectType` GUID 3.

If all checks pass, we consider the ACE for the access check. First, we revoke the access from the entry in the tree of objects. This removes the access from the `ObjectType` entry we found at 3, but also from any children of that entry. We also revoke the access we're maintaining for this function.

Let's apply this behavior to the tree shown in Figure 7-4. If the `AllowedObject` ACE granted `GenericAll` to property set 1, the new tree would look like the one in Figure 7-5.

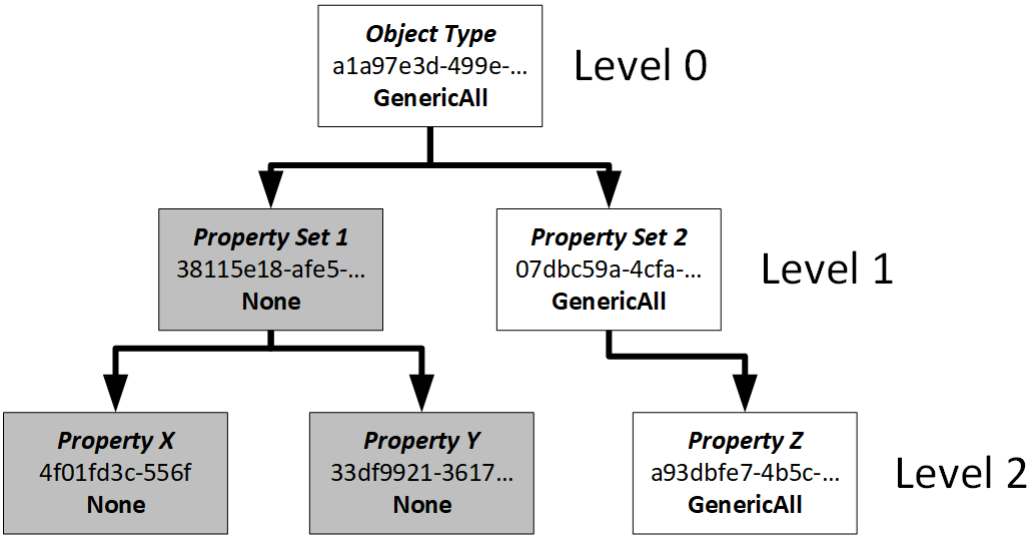

Figure 7-5 The object type tree after access is granted to property set 1

Because we revoked the `RemainingAccess` from property set 1, we also removed that access from properties X and Y. I've

highlighted the nodes with an empty `RemainingAccess`. Note that, for `Allowed` ACEs, only the main `RemainingAccess` matters, as the tree's purpose is to handle `Denied` ACEs correctly. This means that not every object type must have a `RemainingAccess` of zero for the access check to succeed.

Now let's handle the `DeniedObject` ACE. Add the code in Listing 7-28 to the existing ACE enumeration code in Listing 7-17.

```
"DeniedObject" {
    if (!(Test-NtTokenGroup -Token $token $sid -DenyOnly)) {
        break
    }

  1 if ($null -ne $Context.ObjectTypes) {
        if ($null -eq $ace.ObjectType) {
            break;
        }

        $object_type = Select-ObjectTypeTree $Context.ObjectTypes
$ace.ObjectType
        if ($null -eq $object_type) {
            break
        }

      2 if (Test-NtAccessMask $object_type.RemainingAccess $ace.Mask) {
            $continue_check = $false
            break
        }
    }
  3 if (Test-NtAccessMask $access $ace.Mask) {
        $continue_check = $false
    }
}
```

Listing 7-28 An implementation of the `DeniedObject` ACE access-check algorithm

As usual, we begin by checking all ACEs with the `DeniedObject` type. If the check passes, we next check the `ObjectTypes` context property 1. When we handled the `AllowedObject` ACE, we stopped the check if the property was missing. However, we handle the `DeniedObject` ACEs differently. If there is no `ObjectTypes` property, the check will continue as if it were a normal `Denied` ACE, by considering the main `RemainingAccess` 3.

If the ACE's access mask contains bits in the `RemainingAccess`, we deny access 2. If this check passes, we check the value against the main `RemainingAccess`. This demonstrates the purpose of maintaining the tree: if the `Denied` ACE matched property X in Figure 7-5, the denied mask would have no effect. However, if the `Denied` ACE matched property Z, then that object type, and by association property set 2 and the root object type, would be denied as well. In Figure 7-6, you can see that the hatched nodes are all now denied, even though the property set 1 branch is still allowed.

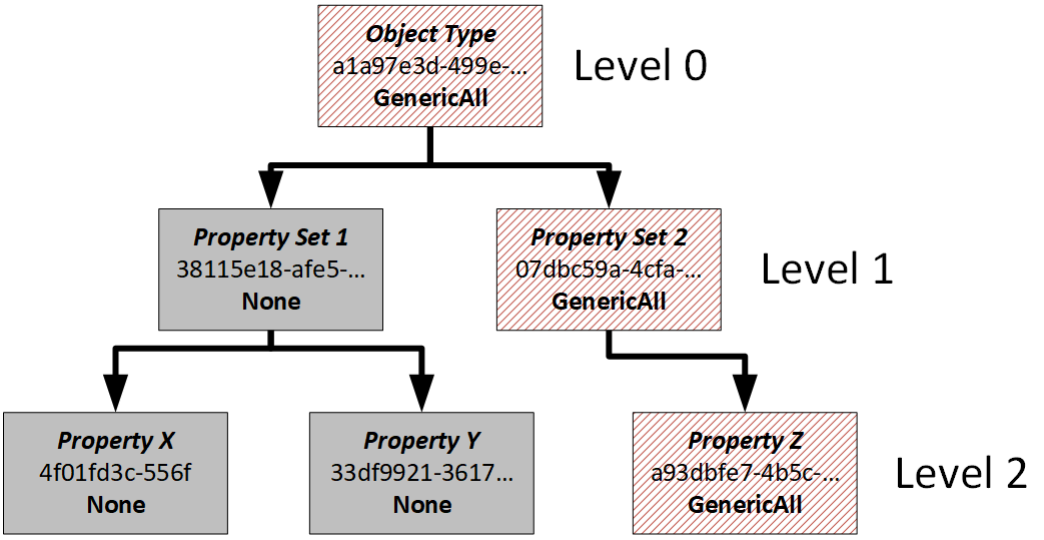

Figure 7-6 The object type tree after denying access to property Z

The `NtAccessCheckByType` system call returns a single status and granted access for the entire list of object types, reflecting the access specified at the root of the object type tree. Therefore, in the case of Figure 7-6, the whole access check would fail.

However, you can figure out which particular object types failed the access check using the `NtAccessCheckByTypeResultList` system call, which returns a status and the granted access for every entry in the object type list. We can use this system call by specifying the

`ResultList` parameter to `Get-NtGrantedAccess`, as shown in Listing 7-29.

```
1 PS> $tree = New-ObjectTypeTree (New-Guid) -Name "Object"
  PS> $set_1 = Add-ObjectTypeTree $tree (New-Guid) -Name "Property Set 1" -
  PassThru
  PS> $set_2 = Add-ObjectTypeTree $tree (New-Guid) -Name "Property Set 2" -
  PassThru
  PS> Add-ObjectTypeTree $set_1 (New-Guid) -Name "Property X"
  PS> Add-ObjectTypeTree $set_1 (New-Guid) -Name "Property Y"
  PS> $prop_z = New-Guid
  PS> Add-ObjectTypeTree $set_2 $prop_z -Name "Property Z"

  PS> $owner = Get-NtSid -KnownSid LocalSystem
  PS> $sd = New-NtSecurityDescriptor -Owner $owner -Group $owner -Type Mutant
  PS> Add-NtSecurityDescriptorAce $sd -KnownSid World -Access WriteOwner
2 -MapGeneric -Type DeniedObject -ObjectType $prop_z
  PS> Add-NtSecurityDescriptorAce $sd -KnownSid World
  -Access ReadControl, WriteOwner -MapGeneric
  PS> Edit-NtSecurityDescriptor $sd -CanonicalizeDacl
3 PS> Get-NtGrantedAccess $sd -PassResult -ObjectType $tree
  -Access ReadControl, WriteOwner | Format-Table Status, SpecificGrantedAccess,
  Name
                Status SpecificGrantedAccess Name
                ------ --------------------- ----
4 STATUS_ACCESS_DENIED                  None Object

5 PS> Get-NtGrantedAccess $sd -PassResult -ResultList -ObjectType $tree
  -Access ReadControl, WriteOwner | Format-Table Status, SpecificGrantedAccess,
  Name
              6 Status   SpecificGrantedAccess   Name
                ------   ---------------------    ----
  STATUS_ACCESS_DENIED                eadControl  Object
        STATUS_SUCCESS   ReadControl, WriteOwner  Property Set 1
        STATUS_SUCCESS   ReadControl, WriteOwner  Property X
        STATUS_SUCCESS   ReadControl, WriteOwner  Property Y
  STATUS_ACCESS_DENIED              ReadControl   Property Set 2
  STATUS_ACCESS_DENIED              ReadControl   Property Z
```

Listing 7-29 Example showing the difference between normal and list results

We start by building the object type tree to match the tree in Figure 7-4 1. We don't care about the specific GUID values except for that of property Z, which we'll need for the `DeniedObject` ACE, so we generate random GUIDs. Next, we build the security descriptor, creating an ACE that denies `ReadControl` access to property Z 2. We also include a non-object ACE to grant `ReadControl` and `WriteOwner`.

We first run the access check with the object type tree but without the `ResultList` parameter, requesting both `ReadControl` and `WriteOwner` [3]. We use the `Deny` ACE, as it matches an object type GUID in the object type tree. As we expected, this causes the access-check process to return `STATUS_ACCESS_DENIED`, with `None` as the granted access [4].

When we execute the access check again, this time with `ResultList`, we receive a list of access check results [5]. The top-level object entry still indicates that access was denied, but Property Set 1 and its children were considered a success [6]. This result corresponds to the tree shown in Figure 7-6. Also note that the entries for which access was denied don't show an empty granted access; instead, they indicate that `ReadControl` would have been granted. This is an artifact of how the access check is implemented under the hood and almost certainly shouldn't be used.

Central Access Policy

Central access policy, a feature added in Window 8 and Windows Server 2012 for use in enterprise networks, is the core security mechanism behind a Windows feature called *dynamic access control*. It relies on device- and user-claim attributes in the token. While it's an enterprise-focused feature, its changes to the access check process exist in all versions of Windows since.

In Chapter 4, we mentioned user and device claims when discussing the conditional expression format. A *user claim* is a security attribute added to the token for a specific user. For example, you might have a claim that represents the country in which a user is employed. We can sync the value of the claim with values stored in Active Directory, so that if the user, say, moves to another country, their user claim will update the next time they authenticate.

A *device claim* belongs to the computer used to access the resource. For example, a device claim might indicate whether the computer is located in a secure room or is running a specific version of Windows. Figure 7-7 shows a common use of a central

access policy: restricting access to files on a server in an enterprise network.

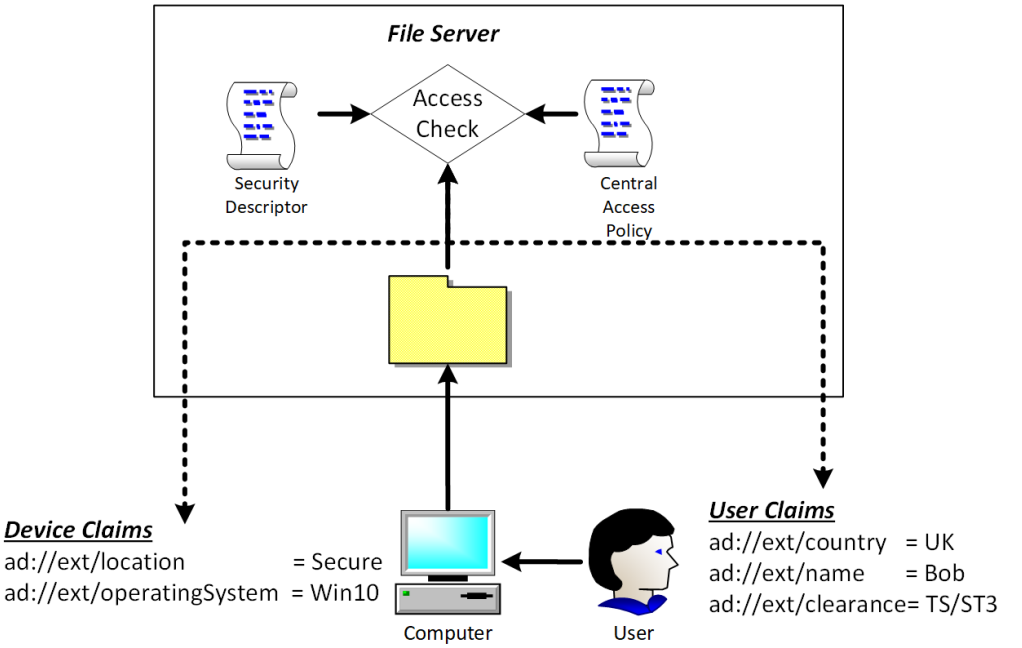

Figure 7-7 A central access policy on a file server

This central access policy contains one or more additional security descriptors that the access check will consider in addition to a file's security descriptor. The final granted access is the most restrictive of the access checks. While not strictly necessary, the additional security descriptors can rely on user and device claims in `AllowedCallback` ACEs to determine the granted access. To send the user and device claims over the network, the enterprise's Kerberos authentication must be configured to support the claims. We'll come back to Kerberos authentication in Chapter 12.

You might wonder how using a central access policy differs from simply configuring the security of the files to use the device and user claims. The main difference is that the central access policy is managed centrally using policies in the enterprise domain group policy. This means an administrator can change the policy in one place to update it across the enterprise.

A second difference is that the central access policy works more like a mandatory access control mechanism. For example, a user might typically be able to modify the security descriptor for the file; however, the central access policy could further restrict their access or block it outright if, for example, the user moved to a new country or used a different computer not accounted for in the rules.

We won't discuss how to configure a central access policy, as the topic is more appropriate for a book on Windows enterprise management. Instead, we'll explore how it's enforced by the kernel's access-check process. The Windows registry stores the central access policy when the computer's group policy is updated, and you can find the key at the following location: *HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Lsa\CentralizedAccessPolicies*

There can be more than one configured policy. Each policy contains the following information:

- The name and description of the policy
- An SID that uniquely identifies the policy
- One or more policy rules

Each policy rule contains the following information:

- The name and a description of the rule
- A conditional expression that determines when the rule should be enforced
- The security descriptor to use in the central access policy access check
- An optional staging security descriptor used to test new policy rules

You can use the `Get-CentralAccessPolicy` PowerShell command to display the list of policies and rules. For most Windows systems, the command won't return any

information. To see results like those in Listing 7-30, you'll need to join a domain that is configured to use a central access policy.

```
PS> Get-CentralAccessPolicy
Name               CapId                              Description
----               -----                              -----------
Secure Room Policy S-1-17-3260955821-1180564752-...  Only for Secure Computers
Main Policy        S-1-17-76010919-1187351633-...
PS> $rules = Get-CentralAccessPolicy | Select-Object -ExpandProperty Rules
PS> $rules | Format-Table
Name        Description  AppliesTo
----        -----------  ---------
Secure Rule Secure!      @RESOURCE.EnableSecure == 1
Main Rule   NotSecure!

PS> $sd = $rules[0].SecurityDescriptor
PS> Format-NtSecurityDescriptor $sd -Type File -SecurityInformation Dacl
<DACL> (Auto Inherit Requested)
 - Type  : AllowedCallback
 - Name  : Everyone
 - SID   : S-1-1-0
 - Mask  : 0x001F01FF
 - Access: Full Access
 - Flags : None
- Condition: @USER.ad://ext/clearance == "TS/ST3" &&
             @DEVICE.ad://ext/location = "Secure"
```

Listing 7-30 Displaying the central access policy using `Get-CentralAccessPolicy`

We run `Get-CentralAccessPolicy` and see two policies, `Secure Room Policy` and `Main Policy`. Each policy has a `CapId` SID and a `Rules` property, which we can expand to see the individual rules. The output table contains the following fields: `Name`, `Description`, and `AppliesTo`, which is a conditional expression used to select whether the rule should be enforced. If the `AppliesTo` field is empty, the rule will always be enforced. The `AppliesTo` field for the Secure Rule selects on a resource attribute, which we'll come back to in Listing 7-32.

Let's display the security descriptor for one of the rules, Secure Rule. The DACL contains a single `AllowedCallback` ACE that grants full access to the *Everyone* group if the condition matches. In this case, the clearance user claim must be set to the value `TS/ST3`, and the device claim location must be set to `Secure`.

We'll walk through a basic implementation of the central access policy access check to better understand what the policy is being used for. Add the code in Listing 7-31 to the end of the `Get-PSGrantedAccess` function from Listing 7-2.

```
1 if (!$success) {
    return Get-AccessResult STATUS_ACCESS_DENIED
  }

2 $capid = $SecurityDescriptor.ScopedPolicyId
  if ($null -eq $capid) {
      return Get-AccessResult STATUS_SUCCESS $Context.Privileges $DesiredAccess
  }

3 $policy = Get-CentralAccessPolicy -CapId $capid.Sid
  if ($null -eq $policy){
      return Get-AccessResult STATUS_SUCCESS $Context.Privileges $DesiredAccess
  }

4 $effective_access = $DesiredAccess
  foreach($rule in $policy.Rules) {
    if ($rule.AppliesTo -ne "") {
      $resource_attrs = $null
      if ($sd.ResourceAttributes.Count -gt 0) {
        $resource_attrs = $sd.ResourceAttributes.ResourceAttribute
      }
      if (!(Test-NtAceCondition -Token $Token -Condition $rule.AppliesTo
5 -ResourceAttribute $resource_attrs)) {
              continue
      }
    }
    $new_sd = Copy-NtSecurityDescriptor $SecurityDescriptor
  6 Set-NtSecurityDescriptorDacl $rule.Sd.Dacl

    $Context.SecurityDescriptor = $new_sd
    $Context.RemainingAccess = $DesiredAccess

  7 Get-DiscretionaryAccess $Context
  8 $effective_access = $effective_access -band (-bnot $Context.RemainingAccess)
  }

9 if (Test-NtAccessMask $effective_access -Empty) {
    return Get-AccessResult STATUS_ACCESS_DENIED
  }
0 return Get-AccessResult STATUS_SUCCESS $Context.Privileges $effective_access
```

Listing 7-31 An implementation of central access policy enforcement

Listing 7-31 begins immediately after the discretionary access check. If the discretionary access check fails, the `$success`

variable will be `False,` and we should return access denied 1. To start the process of enforcing a central access policy, we need to query the `ScopedPolicyId` ACE from the SACL 2. If there is no `ScopedPolicyId` ACE, we can return success. We also return success if there is no central access policy with a `CapId` that matches the ACE's SID 3.

Within the central access policy check, we first set the effective access to the original `DesiredAccess` 1. We'll use the effective access to determine how much of the `DesiredAccess` we can grant after processing all the policy rules. Next, we check the `AppliesTo` conditional expression for each rule. If there is no value, the rule applies to all resources and tokens. If there is a conditional expression, we must check it using `Test-NtAceCondition`, passing any resource attributes from the security descriptor 5. If the test doesn't pass, the check should skip to the next rule.

We build a new security descriptor using the owner, group, and SACL from the original security descriptor but the DACL from the rule's security descriptor 6. If the rule applies, we do another discretionary access check for the `DesiredAccess` 7. After the discretionary access check, we remove any bits that we weren't granted from the `effective_access` variable 8.

After we've checked all the applicable rules, we test whether the effective access is empty. If it is, the central access policy has not granted the token any access, so it should be denied 9. Otherwise, we return success, but return only the remaining effective access that grants less access than the result of the first access check 0.

How can we enable the central access policy for a resource? If we're enabling it on a file server, we can use the Windows GUI to set up the policy using file properties in Explorer and the Advanced Security Settings dialog.

While most central access policies are designed to check files, we can modify any resource type to enforce a policy. To enable it for another resource, we need to do two things: set a scoped

policy ID ACE with the SID of the policy to enable, and add any resource attribute ACEs to match the `AppliesTo` condition, if there is any. We perform these tasks in Listing 7-32.

```
  PS> $sd = New-NtSecurityDescriptor
1 PS> $attr = New-NtSecurityAttribute "EnableSecure" -LongValue 1
2 PS> Add-NtSecurityDescriptorAce $sd -Type ResourceAttribute -Sid "WD"
  -SecurityAttribute $attr -Flags ObjectInherit, ContainerInherit
  PS> $capid = "S-1-17-3260955821-1180564752-1365479606-2616254494"
3 PS> Add-NtSecurityDescriptorAce $sd -Type ScopedPolicyId -Sid $capid
  -Flags ObjectInherit, ContainerInherit
  PS> Format-NtSecurityDescriptor $sd -SecurityInformation Attribute, Scope
  Type: Generic
  Control: SaclPresent
  <Resource Attributes>
   - Type  : ResourceAttribute
   - Name  : Everyone
   - SID   : S-1-1-0
   - Mask  : 0x00000000
   - Access: Full Access
   - Flags : ObjectInherit, ContainerInherit
   - Attribute: "EnableSecure",TI,0x0,1

  <Scoped Policy ID>
   - Type  : ScopedPolicyId
   - Name  : S-1-17-3260955821-1180564752-1365479606-2616254494
   - SID   : S-1-17-3260955821-1180564752-1365479606-2616254494
   - Mask  : 0x00000000
   - Access: Full Access
   - Flags : ObjectInherit, ContainerInherit

  PS> Set-NtTokenPrivilege SeSecurityPrivilege
4 PS> Set-Win32SecurityDescriptor $sd MACHINE\SOFTWARE\PROTECTED
  -Type RegistryKey -SecurityInformation Scope, Attribute
```

Listing 7-32 Enabling the secure room policy for a registry key

The first thing we need to do is add a resource attribute ACE to satisfy the `AppliesTo` condition for the Secure Rule. We create a security attribute object with the name `EnableSecure` and a single Int64 value of 1 1. We add this security attribute to an ACE of type `ResourceAttribute` in the security descriptor's SACL 2. We then need to set the SID of the central access policy, which you can get from the output of the `Get-CentralAccessPolicy` command in a `ScopedPolicyId` ACE 3. We can format the security descriptor to check the ACEs are correct.

We now set the two ACEs to the resource. In this case, the resource we'll pick is a registry key [4]. Note that you must have previously created this registry key for the operation to succeed. The `SecurityInformation` must be set to `Scope` and `Attribute`. As we observed in Chapter 5, to set the `ScopedPolicyId` ACE, we need `AccessSystemSecurity` access, which means we need to first enable `SeSecurityPrivilege`.

If you access the registry key, you should find the policy to be enforced. Note that, as the central access policy is configured for use with filesystems, the access mask in the security descriptor might not work correctly with other resources, such as registry keys. You could manually configure the attributes in Active Directory if you really wanted to support this behavior.

One final thing to mention: central access policy rules support specifying a staging security descriptor as well as the normal security descriptor. We can use this staging security descriptor to test an upcoming security change before deploying it widely. The staging security descriptor is checked in the same way as the normal security descriptor, except the result of the check is used only to compare against the real granted access, and an audit log is generated if the two access masks differ.

Worked Examples

Let's finish with some worked examples using the commands you've learned about in this chapter.

Using the Get-PSGrantedAccess Command

Throughout this chapter, we've built our own implementation of the access check process: the `Get-PSGrantedAccess` command. In this section, let's explore the use of this command. You can retrieve the module from the *chapter7_access_check_impl.psm1* file included with the online additional materials for this book.

Because `Get-PSGrantedAccess` is a simple implementation of the access check, it's missing some features, such as support for calculating maximum access. However, can help you understanding the access check process. You can, for example, use a PowerShell debugger in the PowerShell Integrated Scripting Environment (ISE) or Visual Studio Code to step through the access check and see how it functions based on different input.

Run the commands in Listing 7-33 as a non-administrator split-token user.

```
1 PS> Import-Module ".\chapter_7_access_check_impl.psm1"
2 PS> $sd = New-NtSecurityDescriptor "O:SYG:SYD:(A;;GR;;;WD)"
  -Type File -MapGeneric
  PS> $type = Get-NtType File
  PS> $desired_access = Get-NtAccessMask -FileAccess GenericRead -
  MapGenericRights
3 PS> Get-PSGrantedAccess -SecurityDescriptor $sd
  -GenericMapping $type.GenericMapping -DesiredAccess $desired_access
  Status         Privileges  GrantedAccess
  ------         ----------  -------------
  STATUS_SUCCESS {}          1179785

4 PS> $desired_access = Get-NtAccessMask -FileAccess WriteOwner
  PS> Get-PSGrantedAccess -SecurityDescriptor $sd
  -GenericMapping $type.GenericMapping -DesiredAccess $desired_access
  Status               Privileges  GrantedAccess
  ------               ----------  -------------
  STATUS_ACCESS_DENIED {}          0

5 PS> $token = Get-NtToken -Linked
6 PS> Set-NtTokenPrivilege -Token $token -Privilege SeTakeOwnershipPrivilege
  PS> Get-PSGrantedAccess -Token $token -SecurityDescriptor $sd
  -GenericMapping $type.GenericMapping -DesiredAccess $desired_access
  Status         Privileges                 GrantedAccess
  ------         ----------                 -------------
  STATUS_SUCCESS {SeTakeOwnershipPrivilege} 524288
```

Listing 7-33 Using the `Get-PSGrantedAccess` command

First, we import the module containing the `Get-PSGrantedAccess` command 1. The import assumes the module file is saved in your current directory; if it's not, modify the path as appropriate. We then build a restrictive security

descriptor, granting read access to the *Everyone* group and nobody else 2.

We call `Get-PSGrantedAccess`, requesting `GenericRead` access, along with the `File` object type's generic mapping 3. We don't specify a `Token` parameter, which means the check will use the caller's effective token. The command returns `STATUS_SUCCESS`, and the granted access matches the desired access we originally passed to it.

Next, we change the desired access to `WriteOwner` access only 4. Based on the restrictive security descriptor, only the owner of the security descriptor should be granted this access, which was set to the *SYSTEM* user. If we rerun the access check, we get `STATUS_ACCESS_DENIED`, and no granted access.

To show how we can bypass these restrictions, we query for the caller's linked token 5. As described in Chapter 4, UAC uses the linked token to expose the full administrator token. This command won't work unless you're running the script as a split-token administrator. However, we can enable the `SeTakeOwnershipPrivilege` on the linked token 6, which should bypass the owner check for `WriteOwner`. The access check should now return `STATUS_SUCCESS` and grant the desired access. The privileges column shows that `SeTakeOwnershipPrivilege` was used to grant the access right.

As mentioned, it's worth running this script in a debugger and stepping into `Get-PSGrantedAccess` to follow the access check process so that you understand it better. I also recommend trying different combinations of values in the security descriptor.

Calculating Granted Access for Resources

If you really need to know the granted access of a resource, you're better off using the `Get-NtGrantedAccess` command over the PowerShell implementation we've developed. Let's see how we can use this command to get the granted access for a list of resources. In Listing 7-34, we'll take the script we used in

Chapter 6 to find the owners of objects and calculate the full granted access.

```
PS> function Get-NameAndGrantedAccess {
    [CmdletBinding()]
    param(
        [parameter(Mandatory, ValueFromPipeline)]
        $Entry,
        [parameter(Mandatory)]
        $Root
    )

    PROCESS {
        $sd = Get-NtSecurityDescriptor -Path $Entry.Name -Root $Root
-TypeName $Entry.NtTypeName -ErrorAction SilentlyContinue
        if ($null -ne $sd) {
          1 $granted_access = Get-NtGrantedAccess -SecurityDescriptor $sd
            if (!(Test-NtAccessMask $granted_access -Empty)) {
                $props = @{
                    Name = $Entry.Name;
                    NtTypeName = $Entry.NtTypeName
                    GrantedAccess = $granted_access
                }

                New-Object -TypeName PSObject -Prop $props
            }
        }
    }
}

PS> Use-NtObject($dir = Get-NtDirectory \BaseNamedObjects) {
    Get-NtDirectoryEntry $dir | Get-NameAndGrantedAccess -Root $dir
}
Name                                NtTypeName  GrantedAccess
----                                ----------  -------------
SM0:8924:120:WilError_03_p0         Semaphore   QueryState, ModifyState, ...
CLR_PerfMon_DoneEnumEvent           Event       QueryState, ModifyState, ...
msys-2.0S5-1888ae32e00d56aa         Directory   Query, Traverse, ...
SyncRootManagerRegistryUpdateEvent Event       QueryState, ModifyState, ...
--snip--
```

Listing 7-34 Enumerating objects and getting their granted access

We've modified the script you created in Listing 6-37. Now, instead of merely checking the owner SID, we call `Get-NtGrantedAccess` with the security descriptor 1. This should retrieve the granted access for the caller. Another strategy would have been to check the granted access for any impersonation token at the Identification level with query access on the handle,

then pass it as the `Token` parameter. In the next chapter, we'll show an easier way to do large-scale access checking without having to write your own scripts.

Wrapping Up

In this chapter, we detailed the implementation of the access-checking process in Windows at length. This included describing the operating system's mandatory access checks, token owner and privilege checks, and discretionary access checks. We also built our own implementation of the access check so that you can better understand the process.

We also covered how the two types of sandboxing tokens, restricted and lowbox, affect the access-checking process to restrict resource access. We then discussed object-type checking and central access policies, important features of enterprise security for Windows.

8

OTHER ACCESS CHECKING USE CASES

Access checks determine what access a caller should have when opening a kernel resource. However, we sometimes perform access checks for other reasons, as they can serve as additional security checks. This chapter details some examples of using access checks as a secondary security mechanism.

We'll start the chapter with traversal checking, which determines whether a caller has access to a hierarchy of resources. Then, we'll discuss how access checks are used when a handle is duplicated. We'll also discuss how an access check can limit

access to kernel information, such as process listings, from sandboxed applications. Finally, we'll describe some additional PowerShell commands that automate the access checking of resources.

Traversal Checking

When accessing a hierarchical set of resources, such as an object directory tree, the user must traverse the hierarchy until they reach the target resource. But for every directory or container in the hierarchy, the system performs an access check to determine whether a caller can proceed to the next container.

This check is called a *traversal check*, and it's performed whenever code looks up a path inside the I/O manager or object manager. For example, Figure 8-1 shows the traversal checks needed to access an OMNS object using the path *\ABC\QRS\XYZ\OBJ*.

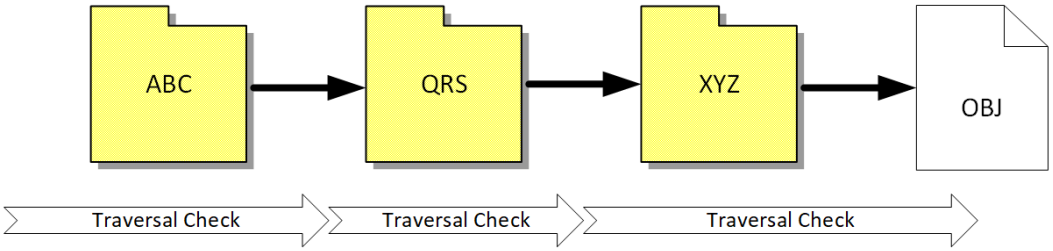

Figure 8-1 Traversal checks needed to access *OBJ*

As you can see, we need three access checks to determine if *OBJ* exists. Each access check extracts the security descriptor from the container and then checks the type-specific access to see if traversal is allowed. Both the OMNS and file directories can have a `Traverse` access right they can grant. If, for example, *QRS* denied `Traverse` access to the caller, the traversal check would fail, as shown in Figure 8-2.

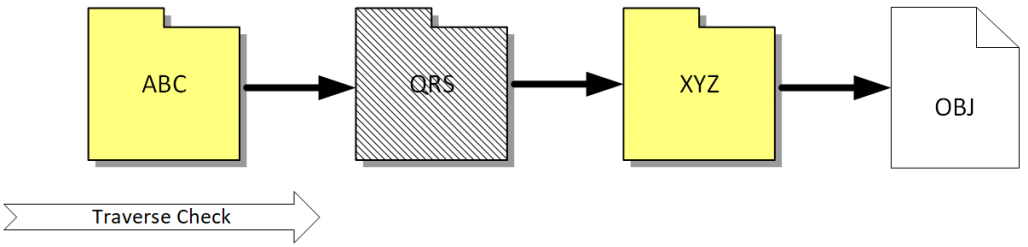

Figure 8-2 Traversal checks blocked at *QRS*

Even though the caller can access *XYZ* and *OBJ*, the traversal check will fail because *QRS* now denies access, so it's no longer possible to access *OBJ* using the *\ABC\QRS\XYZ\OBJ* path.

This behavior has two implications: first, it prevents a user from accessing their resources if any parent container blocks their access. However, if a user must have access to a resource multiple levels deep in a hierarchy, they probably also have access to all of the resource's parents. As it's common for the user to have access to these parent containers, doing the access check for each container is unnecessary and wastes processing time.

Almost every `Token` object has `SeChangeNotifyPrivilege` enabled, which solves both of these issues.

The SeChangeNotifyPrivilege Privilege

If the `SeChangeNotifyPrivilege` privilege is enabled, the system bypasses the entire traversal check and lets users access resources that an inaccessible parent would otherwise block. Let's verify the privilege's behavior using OMNS directory objects (Listing 8-2).

```
PS> $path = "\BaseNamedObjects\ABC\QRS\XYZ\OBJ"
PS> $os = New-NtMutant $path -CreateDirectories
PS> Enable-NtTokenPrivilege SeChangeNotifyPrivilege
PS> Test-NtObject $path
True

PS> $sd = New-NtSecurityDescriptor -EmptyDacl
PS> Set-NtSecurityDescriptor "\BaseNamedObjects\ABC\QRS" $sd Dacl
```

```
PS> Test-NtObject $path
True

PS> Disable-NtTokenPrivilege SeChangeNotifyPrivilege
PS> Test-NtObject $path
False

PS> Test-NtObject "OBJ" -Root $os[1]
True
```

Listing 8-1 Testing `SeChangeNotifyPrivilege` to bypass traversal checks

We first create a `Mutant` object and all of its parent directories. We can automate this directory-creation by using the `CreateDirectories` property. We ensure the privilege is enabled and then use the `Test-NtObject` command to check whether the `Mutant` object can be opened. In the output we can see we're able to open the `Mutant object`.

We then set a security descriptor with an empty DACL on the *QRS* directory. This should block all access to the directory object, including `Traverse` access. But when we checking our access again, we can still access the `Mutant` object because we have the privilege enabled.

We now disable the privilege and try again to open the `Mutant` object. This time, the directory traversal fails. Without the privilege or access to the *QRS* directory, we can no longer open the `Mutant` object. However, our final check demonstrates that if you have access to a parent after *QRS*, such as *XYZ*, you can access the `Mutant` object via a relative open by using the directory as the `Root`.

Limited Checks

The kernel contains an additional performance improvement for traversal checks. If the `SeChangeNotifyPrivilege` is disabled, the kernel will call the `SeFastTraverseCheck` function, which performs a more limited check rather than a full access check. For completeness, I have reimplemented the `SeFastTraverseCheck` function in PowerShell so that we

can explore its behavior in more detail. Listing 8-2 shows the implementation.

```
function Get-FastTraverseCheck {
    Param(
      1 $TokenFlags,
        $SecurityDescriptor,
        $AccessMask
    )

  2 if ($SecurityDescriptor.DaclNull) {
        return $true
    }
  3 if (($TokenFlags -band "IsFiltered, IsRestricted") -ne 0) {
        return $false
    }
    $sid = Get-Ntsid -KnownSid World
    foreach($ace in $SecurityDescriptor.Dacl) {
      4 if ($ace.IsInheritedOnly -or !$ace.IsAccessGranted($AccessMask)) {
            continue
        }
      5 if ($ace.IsDeniedAce) {
            return $false
        }
      6 if ($ace.IsAllowedAce -and $ace.Sid -eq $sid) {
            return $true
        }
    }
  6 return $false
}
```

Listing 8-2 A PowerShell implementation of `SeFastTraverseCheck`

First, we define the three parameters the function takes: the token's flags, a directory object's security descriptor, and the traverse access rights to check 1. We specify the access rights because the object manager and the I/O manager use this function for `Directory` and `File` objects, and the `Traverse` access right has a different value between the two object types, so specifying the access as a parameter allows the check function to handle both cases.

Next, we check whether the security descriptor's DACL is NULL; if it is, we grant access 2. We follow this with a check on two token flags 3. If the flags indicate that the token is filtered or restricted, then the fast check fails. The kernel copies these flags from the caller's token object. You can get the flags from user-

mode using the `Flags` property on a token object, as shown in Listing 8-3.

```
PS> $token = Get-NtToken -Pseudo -Primary
PS> $token.Flags
VirtualizeAllowed, IsFiltered, NotLow
PS> $token.ElevationType
Limited
```

Listing 8-3 Querying token flags

Listing 8-3 also shows something interesting: the flags include the `IsFiltered` flag. If you're not running in a restricted token sandbox, why would this flag be set?

The token elevation type is `Limited`, which means it's the default token for a UAC administrator. To convert the full administrator token to the default token, LSASS uses the `NtFilterToken` system, which will set the `IsFiltered` flag but not `IsRestricted`, as it's only removing groups, not adding restricted SIDs. This means that while a UAC admin running code as the default user can never pass the fast traverse check, a normal user could. This behavior doesn't have any security implication, but it does mean that if `SeChangeNotifyPrivilege` is disabled, resource lookup performance will suffer.

The final check in Listing 8-2 consists of enumerating the DACL's ACEs. If the ACE is inherit-only or doesn't contain the required `Traverse` access mask, it's skipped 4. If it's a deny ACE, the fast traverse check fails 5, and the ACE's SID is not checked at all in this case. Finally, if the ACE is an allow ACE and the SID equals the *Everyone* group SID, the fast check succeeds 6. If there are no more ACEs, the check fails 7.

Note that this fast check doesn't consider whether the caller's token has the *Everyone* group enabled. This is because, typically, the only way to remove the *Everyone* group would be to filter the token. The big exception to this is the anonymous token, which doesn't have any groups but is also not filtered in any way.

Let's turn to another use for the access check: considering the granted access when assigning a duplicated handle.

Handle Duplication Access Checks

The system always performs an access check when creating or opening a kernel resource that returns a handle. But what about when that handle is duplicated? In the simplest case, when the new handle has the same granted access mask as the original, the system won't perform any checks. It's also possible to drop some parts of the granted access mask, and doing so won't trigger an additional access check, either. However, if you want to add additional access rights to the duplicated handle, the kernel will query the security descriptor from the object and perform a new access check to determine whether to allow access.

When you duplicate a handle, you must specify both the source and destination process handles, and the access check occurs in the context of the destination process. This means the access check considers the destination process's primary token, not the source process's, which could be an issue if a privileged process tried to duplicate a handle to a less privileged process with additional access. Such an operation would fail with access denied.

Listing 8-4 demonstrates this handle duplication access check behavior.

```
PS> $sd = New-NtSecurityDescriptor -EmptyDacl
PS> $m = New-NtMutant -Access ModifyState, ReadControl -SecurityDescriptor $sd
PS> Use-NtObject($m2 = Copy-NtObject -Object $m) {
    $m2.GrantedAccess
}
QueryState, ReadControl

PS> $mask = Get-NtAccessMask -MutantAccess QueryState
PS> Use-NtObject($m2 = Copy-NtObject -Object $m -DesiredAccessMask $mask) {
   $m2.GrantedAccess
}
QueryState

PS> Use-NtObject($m2 = Copy-NtObject -Object $m -DesiredAccess GenericAll) {
    $m2.GrantedAccess
```

```
}
Copy-NtObject : (0xC0000022) - {Access Denied}
A process has requested access to an object, ...
```

Listing 8-4 Testing the handle duplication access check behavior

We first create a new `Mutant` object with an empty DACL and request only `QueryState` and `ReadControl` access on the handle. This will block all users from accessing the `Mutant`, with the exception of the owner, who can be granted `ReadControl` and `WriteDac` access thanks to the owner check we described in the previous chapter.

We now test the duplication by requesting the same access, which the new handle returns. Next, we request `QueryState` access only. As the mutant's DACL is empty, this access right wouldn't be granted during an access check, and because we get `QueryState` on the new handle, we know that no access check took place. Finally, we try to increase our access by requesting `GenericAll`. An access check must now take place, as we're requesting additional access rights than the handle currently has. This check results in `Access Denied`.

If we hadn't set a security descriptor when creating the `Mutant`, there would be no security associated with the object, and this last check would have succeeded, granting full access. As mentioned in Chapter 5, you need to be careful when duplicating unnamed handles to less privileged processes if you're dropping access; the destination process might be able to reduplicate the handle to one with more access. In Listing 8-5, we test the `NtDuplicateObject` `NoRightsUpgrade` flag to see how it affects handle duplication access checking.

```
PS> $m = New-NtMutant -Access ModifyState
PS> Use-NtObject($m2 = Copy-NtObject -Object $m -DesiredAccess GenericAll) {
   $m2.GrantedAccess
}
QueryState, Delete, ReadControl, WriteDac, WriteOwner, Synchronize

PS> Use-NtObject($m2 = Copy-NtObject -Object $m -NoRightsUpgrade) {
   Use-NtObject($m3 = Copy-NtObject -Object $m2 -DesiredAccess GenericAll) {}
}
Copy-NtObject : (0xC0000022) - {Access Denied}
A process has requested access to an object, ...
```

Listing 8-5 Testing the `NtDuplicateObject` `NoRightsUpgrade` flag

We start by creating an unnamed `Mutant` object, which will have no associated security descriptor. We request the initial handle with `QueryState` access only. However, duplicating a new handle with `GenericAll` succeeds, granting us complete access.

Now, we test the `NoRightsUpgrade` flag, and because we don't specify any access mask, the handle will be duplicated with `QueryState` access. With the new handle, we perform another duplication, this time requesting `GenericAll` access. We can observe that the handle duplication fails. This isn't due to an access check; instead, it's because of a flag set on the handle entry in the kernel indicating that any request for more access should fail immediately. This prevents the handle from being used to gain additional access rights.

The incorrect handling of duplicate handles can lead to vulnerabilities; for example, I discovered CVE-2019-0943, an issue in a privileged service responsible for caching the details of font files on Windows. The service duplicated a `Section` object handle to a sandbox process with read-only access. However, sandbox process could convert the handle back to a writeable section handle, and the section could be mapped into memory as writeable. This allowed the sandbox process to modify the state of the privileged service and escape the sandbox. Windows fixed the vulnerability by duplicating the handle using the `NoRightsUpgrade` flag.

THE THREAD PROCESS CONTEXT

Every thread is associated with a process. Normally, when an access check occurs, the kernel extracts the process object from the calling thread's object structure and uses it to look up the primary token for the access check. But the thread has a second process object associated with it: the current *process context*, which indicates the process in which the thread is currently executing code.

Normally, these process objects are the same; however, the kernel sometimes switches the current process context to another process to save time during certain tasks, such as handle or virtual memory access. When the process switch has occurred, any access check on the thread will look up the primary token of the switched-to process rather than the token belonging to the process associated with the thread. Handle duplication operations use this process context switch; the kernel first queries the source process's handle table, then switches the process context for the calling thread to the

destination process to create the new handle in that process's handle table.

A handle can abuse this behavior to duplicate a handle with more access to a less-privileged process. If you call the `NtDuplicateObject` system call while impersonating your own token with access to the object, then when the access check runs, it will capture the `SECURITY_SUBJECT_CONTEXT` for the thread, setting the primary token for the destination process. Crucially, though, it also sets the impersonation token to the identity being impersonated. The result is that the access check will run against the caller's impersonation token rather than the destination process's primary token. This allows a handle to be duplicated with additional granted access rights even if the destination process's primary token could not pass an access check for those rights. You probably shouldn't rely on this behavior in practice; it's an implementation detail and might be subject to change.

The access checks that occur during traversal checking and handle duplication are typically hidden from view, but both relate to the security of an individual resource. Next, we'll discuss how access checks limit the information we can extract, and the operations we can perform, for a group of resources. These restrictions occur based on the caller's token, regardless of the individual access set for those resources.

Sandbox Token Checks

Beginning in Windows 8, Microsoft has tried to make it harder to compromise the system by escaping sandbox token restrictions. This is especially important for software such as web browsers or document readers, which process untrusted content from the internet.

The kernel implements two APIs that use an access check to determine whether the caller is in a sandbox: `ExIsRestrictedCaller`, introduced in Windows 8, and `RtlIsSandoxToken`, introduced in Windows 10. These APIs produce equivalent results; the difference between them is that `ExIsRestrictedCaller` checks the token of the caller, while `RtlIsSandoxToken` checks a specified token object that doesn't have to be the caller's.

Internally, these APIs perform an access check for the token and grants access only if the token is not in a sandbox. Listing 8-6 shows a reimplementation of this access check in PowerShell.

```
PS> $type = New-NtType -Name "Sandbox" -GenericRead 0x20000 -GenericAll
0x1F0001
PS> $sd = New-NtSecurityDescriptor -NullDacl -Owner "SY" -Group "SY" -Type
$type
PS> Set-NtSecurityDescriptorIntegrityLevel $sd Medium -Policy NoReadUp
PS> Get-NtGrantedAccess -SecurityDescriptor $sd -Access 0x20000 -PassResult
Status                Granted Access Privileges
------                -------------- ----------
STATUS_SUCCESS        GenericRead    NONE

PS> Use-NtObject($token = Get-NtToken -Duplicate -IntegrityLevel Low) {
    Get-NtGrantedAccess -SecurityDescriptor $sd -Access 0x20000
-Token $token -PassResult
}
Status                Granted Access Privileges
------                -------------- ----------
STATUS_ACCESS_DENIED None            NONE
```

Listing 8-6 An access check for a sandbox token

First, we need to define a dummy kernel object type using the `New-NtType` command. This allows us to specify the generic mapping for the access check. We specify only the generic-read and generic-all values, as write and execute access are not important in this context. Note that the new type is local to PowerShell; the kernel doesn't know anything about it.

We then define a security descriptor with a NULL DACL and the owner and group SIDs set to the *SYSTEM* user. The use of a NULL DACL will deny access to lowbox tokens, as we described in the previous chapter, but not to any other sandbox token type, such as restricted tokens.

To handle other token types, we add a `Medium` mandatory label ACE with a `NoReadUp` policy. As a result, any token with an integrity level lower than `Medium` will be denied access to the mask specified in the generic mapping's generic read field. Lowbox tokens ignore the `Medium` mandatory label, but we've covered these tokens using the NULL DACL. Note that this security descriptor doesn't consider restricted tokens with a `Medium` integrity level to be sandbox tokens. It's not clear if this is an intentional oversight or a bug in the implementation.

We can now perform an access check with the `Get-NtGrantedAccess` command, using the current, non-sandboxed token. The access check succeeds, granting us `GenericRead` access. If we repeat the check with a token that has a `Low` integrity level, the system denies us access, indicating that the token is sandboxed.

Behind the scenes, the kernel APIs call the `SeAccessCheck` API, which will return an error if the caller has an Identification-level impersonation token. Therefore, the kernel will consider some impersonation tokens to be sandboxed even if the implementation in Listing 8-6 would indicate otherwise.

When either API indicates that the caller is sandbox, the kernel changes its behavior to do the following:

- Listing only processes and threads that can be directly accessed
- Blocking access to loaded kernel modules
- Enumerating open handles and their kernel object addresses
- Creating arbitrary file and object manager symbolic links
- Creating a new restricted token with more access

For example, in Listing 8-7, we query for handles while impersonating a `Low` integrity level token, and are denied access.

```
PS> Invoke-NtToken -Current -IntegrityLevel Low {
    Get-NtHandle -ProcessId $pid
}
Get-NtHandle : (0xC0000022) - {Access Denied}
A process has requested access to an object,...
```

Listing 8-7 Querying for handle information while impersonating a `Low` integrity level token

While only kernel-mode code can access `ExIsRestrictedCaller`, you can access `RtlIsSandboxToken` in user-mode, as it's also exported in *NDTLL*. This allows you to query the kernel using a token handle to find out whether the kernel thinks it is a sandbox token. The

`RtlIsSandboxToken` API exposes its result in the token object's `IsSandbox` property, as shown in Listing 8-8.

```
PS> Use-NtObject($token = Get-NtToken) {
   $token.IsSandbox
}
False

PS> Use-NtObject($token = Get-NtToken -Duplicate -IntegrityLevel Low) {
   $token.IsSandbox
}
True
```

Listing 8-8 Checking the sandbox status of tokens

The process object returned by `Get-NtProcess` has an `IsSandboxToken` property. Internally, this property opens the process's token and calls `IsSandbox`. We can use this property to easily discover which processes are sandboxed, by using the script in Listing 8-9, for example.

```
PS> Use-NtObject($ps = Get-NtProcess -FilterScript {$_.IsSandboxToken}) {
    $ps | ForEach-Object { Write-Host "$($_.ProcessId) $($_.Name)" }
}
7128 StartMenuExperienceHost.exe
7584 TextInputHost.exe
4928 SearchApp.exe
7732 ShellExperienceHost.exe
1072 Microsoft.Photos.exe
7992 YourPhone.exe
```

Listing 8-9 Enumerating all sandboxed processes for the current user

These sandbox checks are an important feature for limiting information disclosure and restricting dangerous functionality such as symbolic links, which improve an attacker's chances of escaping the sandbox and gaining additional privileges. For example, blocking access to the handle table prevents the disclosure of kernel object addresses that could be used to exploit kernel memory-corruption vulnerabilities.

We've now covered three uses of the access check for purposes not related to opening a resource. We'll finish this chapter by describing some commands that simplify access checking over a range of individual resources.

Automating Access Checks

The previous chapter provided a worked example that used `Get-NtGrantedAccess` to determine the granted access for a collection of kernel objects. If you want to check a different type of resource, such as files, you'll need to modify that script to use file commands.

Because checking for the granted access across a range of resources is such a useful operation, the PowerShell module comes with several commands to automate the process. The commands are designed to allow you to quickly assess the security attack surface of available resources on a Windows system. They all start with `Get-Accessible`, and you can use `Get-Command` to list them, as shown in Listing 8-10.

```
PS> Get-Command Get-Accessible* | Format-Wide
Get-AccessibleAlpcPort                  Get-AccessibleDevice
Get-AccessibleEventTrace                Get-AccessibleFile
Get-AccessibleHandle                    Get-AccessibleKey
Get-AccessibleNamedPipe                 Get-AccessibleObject
Get-AccessibleProcess                   Get-AccessibleScheduledTask
Get-AccessibleService                   Get-AccessibleToken
Get-AccessibleWindowStation             Get-AccessibleWnf
```

Listing 8-10 Listing the `Get-Accessible` commands

We'll come back to some of these commands in later chapters. Here, we'll demonstrate the `Get-AccessibleObject` command, which we can use to automate access checking over the entire OMNS. The command lets you specify an OMNS path to check, then enumerates the OMNS and reports either the maximum granted access or whether a specific access mask can be granted.

You can also specify what tokens to use for the access checking. The command can source tokens from the following locations:

- The list of token objects
- The list of process objects

- The list of process names
- The list of process IDs
- The list of process command lines

If you specify no options when running the command, it will use the current primary token. It will then enumerate all objects based on an OMNS path and perform an access check for every token specified. If the access check succeeds, then the command generates a structured object containing the details of the access check result. Listing 8-11 shows us an example.

```
PS> Get-AccessibleObject -Path "\"
TokenId Access                      Name
------- ------                      ----
C5856B9 GenericExecute|GenericRead    \
```

Listing 8-11 Getting accessible objects from the OMNS root

We run the command against the root of the OMNS and receive three columns in the output:

`TokenId`
The unique identifier of the token used for the access check

`Access`
The granted access, mapped to generic access rights

`Name`
The name of the checked resource

We can use the `TokenId` to distinguish the results for the different tokens specified to the command.

The result contains much more information than is shown by default, and you can extract it using commands like `Format-List`. You can also display the copy of the security descriptor used to perform the access check with the `Format-NtSecurityDescriptor` PowerShell command, as shown in Listing 8-12.

```
PS> Get-AccessibleObject -Path \ | Format-NtSecurityDescriptor -Summary
<Owner> : BUILTIN\Administrators
<Group> : NT AUTHORITY\SYSTEM
```

```
<DACL>
Everyone: (Allowed)(None)(Query|Traverse|ReadControl)
NT AUTHORITY\SYSTEM: (Allowed)(None)(Full Access)
BUILTIN\Administrators: (Allowed)(None)(Full Access)
NT AUTHORITY\RESTRICTED: (Allowed)(None)(Query|Traverse|ReadControl)
```

Listing 8-12 Displaying the security descriptor used for the access check

As we've run the command against a directory, you might wonder if it will also list the objects contained within the directory. By default, no; the command opens the path as an object and does an access check. If you want to recursively check all objects in the directory, you need to specify the `Recurse` parameter. The command also accepts a `Depth` parameter you can use to specify the maximum recursive depth. If you run a recursive check as a non-administrator user, you might see a lot of warnings, as in Listing 8-13.

```
PS> Get-AccessibleObject -Path "\" -Recurse
WARNING: Couldn't access \PendingRenameMutex - Status: STATUS_ACCESS_DENIED
WARNING: Couldn't access \ObjectTypes - Status: STATUS_ACCESS_DENIED
--snip--
```

Listing 8-13 Warnings when recursively enumerating objects.

You can turn off warnings by setting the `WarningAction` parameter to `Ignore`, but keep in mind that they're trying to tell you something. For the command to work, it needs to open each object and query its security descriptor. From user mode, this requires passing the access check during the opening, so if you don't have permission to open for `ReadControl` access, the command can't perform an access check. For better results, you can run the command as an administrator, and for the best results, run it as the *SYSTEM* user by using the `Start-Win32ChildProcess` command to start a *SYSTEM* PowerShell shell.

By default, the command will perform the access check using the caller's token. But if you're running the command as an administrator, you probably don't want to do this, as almost all resources will allow administrators full access. Instead, consider specifying arbitrary tokens to check against the resource. For example, when run as a UAC administrator, the following

command recursively opens the resources as the administrator token but performs the access check with the non-administrator token from the Explorer process:

```
PS> Get-AccessibleObject -Path \ -ProcessName explorer.exe -Recurse
```

It's common to want to filter the list of objects to check. You could run the access check against all the objects and then filter the list afterward, but this would require a lot of work that you'll then just throw away. To save you some time, the `Get-AccessibleObject` command supports multiple filter parameters:

`TypeFilter`
A list of NT type names to check

`Filter`
A name filter used to restrict which objects are opened; can contain wildcards

`Include`
A name filter used to determine which results to include in the output

`Exclude`
A name filter to determine which results to exclude from the output

`Access`
An access mask to limit the output to only objects with specific granted access

For example, the following command will find all the `Mutant` objects that can be accessed for `GenericAll` access:

```
PS> Get-AccessibleObject -Path \ -TypeFilter Mutant -Access GenericAll -
Recurse
```

By default, the `Access` parameter requires that all access be granted before outputting a result. You can be modify this by specifying `AllowPartialAccess`, which would output any result that partially matches the specified access. If you want to

see all results regardless of the granted access, specify `AllowEmptyAccess`.

Worked Examples

Let's wrap up with some worked examples that use the commands you've learned about in this chapter.

Simplifying an Access Check for an Object

In the previous chapter, we used the `Get-NtGrantedAccess` command to automate an access check against kernel objects and determine their maximum granted access. To accomplish this, we first needed to query for an object's security descriptor. Then, we passed this value to the command along with the type of kernel object to check.

If you have a handle to an object, you can simplify the call to the `Get-NtGrantedAccess` command by specifying the object with the `Object` parameter, as shown in Listing 8-14.

```
PS> $key = Get-NtKey HKLM\Software -Win32Path -Access ReadControl
PS> Get-NtGrantedAccess -Object $key
QueryValue, EnumerateSubKeys, Notify, ReadControl
```

Listing 8-14 Running an access check on an object

Using the `Object` parameter eliminates having to manually extract the security descriptor from the object and will automatically select the correct generic mapping structure for the kernel object type. This reduces the chance that you'll make mistakes when performing an object access check.

Finding Writeable Section Objects

The system uses `Section` objects to share memory between processes. If a privileged process sets a weak security descriptor, it might be possible for a less-privileged process to open and modify the contents of the section. This can lead to security issues

if that section contains trusted parameters that can trick the privileged process into performing privileged operations.

I discovered a vulnerability of this class, CVE-2014-6349, in Internet Explorer's sandbox configuration. The configuration incorrectly secured a shared `Section` object, allowing sandboxed Internet Explorer processes to open it and disable the sandbox entirely. To discover this issue, I checked all named `Section` objects, performing an access check for the `MapWrite` access right. Once I had identified all sections with this access right, I manually determined whether any of them were exploitable from the sandbox. In Listing 8-15, we automate the discovery of writeable sections using the `Get-AccessibleObject` command.

```
PS> $access = Get-NtAccessMask -SectionAccess MapWrite -AsGenericAccess
PS> $objs = Use-NtObject($token = Get-NtToken -Duplicate -IntegrityLevel Low)
{
  Get-AccessibleObject -Win32Path "\" -Recurse -Token $token
-TypeFilter Section -Access $access
}
PS> $token.Close()
PS> $objs | ForEach-Object {
  Use-NtObject($sect = Get-NtSection -Path $_.Name) {
    Use-NtObject($map = Add-NtSection $sect -Protection ReadWrite -ViewSize
4096) {
      Write-Host "$($sect.FullPath)"
        Out-HexDump -ShowHeader -ShowAscii -HideRepeating -Buffer $map | Out-
Host
    }
  }
}
\Sessions\1\BaseNamedObjects\windows_ie_global_counters
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  - 0123456789ABCDEF
------------------------------------------------------------------------
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  - ................
-> REPEATED 1 LINES
00 00 00 00 00 00 00 00 00 00 00 00 1C 00 00 00  - ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  - ................
--snip--
```

Listing 8-15 Enumerating writeable `Section` objects for a `Low` integrity level token

Listing 8-15 starts by calculating the access mask for the `MapWrite` access and converting it into a generic access enumeration. We do this because the `Get-AccessibleObject` command takes only generic access, as it

doesn't know ahead of time what objects you're likely to want to check for. We then duplicate the current user's token and set its integrity level to `Low`, producing a simple sandbox.

We pass the token and access mask to `Get-AccessibleObject`, performing a recursive check in the user's base-named objects directory by specifying a single path separator to the `Win32Path` parameter. The results returned from the command should contains only sections that can be opened for `MapWrite` access.

Finally, we enumerate the list of discovered sections, displaying their names and initial contents of any discovered writeable section object. We open the named section, map up to the first 4,096 bytes into memory, and then output the contents as a hex dump. We map the section writeable, as it's possible the `Section` object's security descriptr grants `MapWrite` access but that the section was created read-only. In this case, mapping `ReadWrite` will fail with an error.

You can use this script as-is to find interesting writeable sections. You don't have to use a sandbox token; it can be interesting to see the sections available for a normal user that are owned by privileged processes. It should also give you the basics for how you could do the same check for any kernel object type.

Wrapping Up

In this chapter, we looked at some examples of the uses of access checking outside of opening a resource. We first described traversal access, which is used to determine if a user can traverse a hierarchical list of containers, such as object directories. Then we discussed how the access check is used when handles are duplicated between processes, including how this can create security issues if the object has no name or security descriptor configured.

We then described how an access check is used to determine if a caller's token is sandboxed. The kernel does this to limit

access to information or certain operations to make it more difficult to exploit certain classes of security vulnerabilities. Finally, we described how to automate access checks for various resource types of `Get-Accessible` commands. We described the basic parameters common to all commands and how to use them to enumerate accessible named kernel objects.

That's the end of our description of the access-checking process. In the next chapter, we'll cover the last remaining responsibility of the SRM: security auditing.

9

SECURITY AUDITING

Intertwined with the access-checking process is the auditing process. An administrator can configure the system's auditing mechanism to generate a log of accessed resources. Each log event would include details about the user and application that opened the resource and whether the access succeeded or failed. This information could help us identify incorrect security settings or detect malicious access to sensitive resources.

In this short chapter, we'll first discuss where the resource access log gets stored once the kernel generates it. We'll then

describe how a system administrator can configure the audit mechanism. Finally, we'll detail how to configure individual resources to generate audit log events through the SACL.

The Security Event Log

Windows generates log events whenever an access check succeeds or fails. The kernel writes these log events to the *security event log*, which only administrators can access.

When performing access checks on kernel resources, Windows will generate the following types of audit events. The security event log represents these by using the event ID included in parentheses:

- Object handle opened (4656)
- Object handle closed (4658)
- Object deleted (4660)
- Object handle duplicated (4690)
- SACL changed (4717)

When we access resources via kernel system calls such as `NtCreateMutant`, the auditing mechanism generates these events automatically. But for the object-related audit events, we must first configure two aspects of the system: we must set the system policy to generate audit events, and we must enable audit ACEs in the resource's SACL. Let's discuss each of these configuration requirements in turn.

Configuring the System Audit Policy

Most Windows users don't need to capture auditing information for kernel resources, so the audit policy is disabled by default. Enterprise environments commonly configure the audit policy through a *domain security policy*, which the enterprise network distributes to the individual devices.

Users not in an enterprise network can enable the audit policy manually. One way to do so is to edit the *local security policy*, which looks the same as the domain security policy but applies only to the current system. There are two types of audit policy: the legacy policy used prior to Windows 7 and the advanced audit policy. On the latest version of Windows, the advanced audit policy is recommended, as it provides more granular configuration, so we won't discuss the legacy policy further.

If you open the local security policy editor by running the `secpol.msc` command in PowerShell, you can view the current configuration of the advanced audit policy, as shown in Figure 9-1.

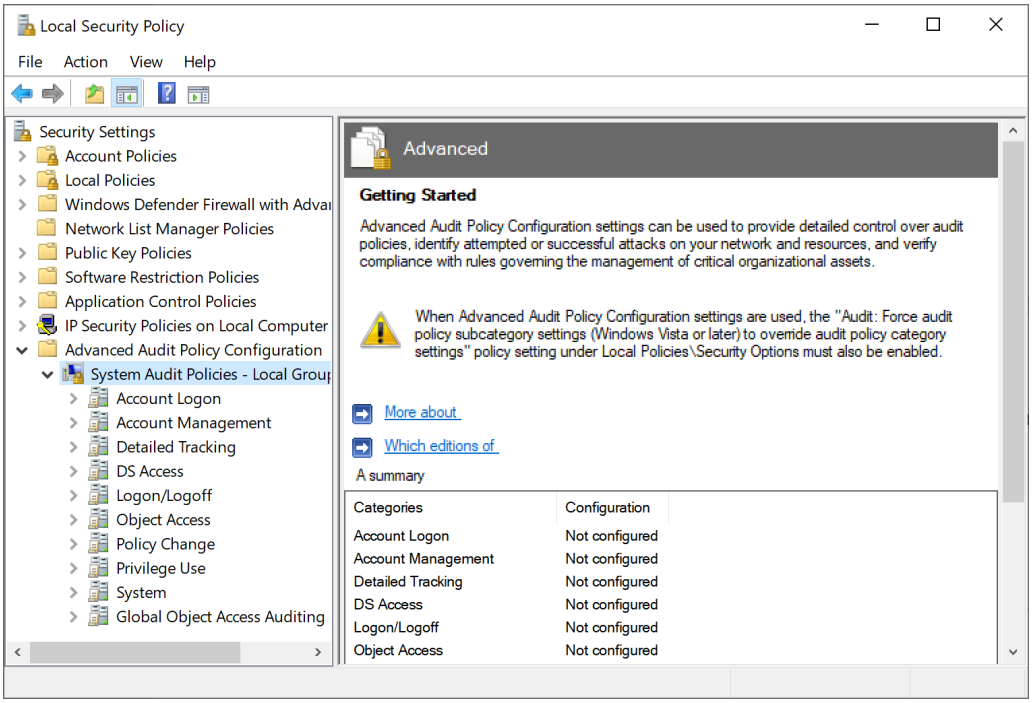

Figure 9-1 The security policy editor showing the advanced audit policy

As you can see, the categories in the audit policy aren't currently configured. To explore how audit events are generated, we'll use PowerShell to enable the required audit policy temporarily and run some example code. Any changes you make

with PowerShell won't be reflected in the local security policy, which will revert the next time it synchronizes (for example, during a reboot or when the group policy is updated on an enterprise network). You can force the settings to synchronize by running the command `gpupdate.exe /force` as an administrator in PowerShell or the command prompt.

Advanced audit policies have two levels: a top-level category and multiple sub-categories. You can query for the top-level categories using `Get-NtAuditPolicy`, as in Listing 9-1.

```
PS> Get-NtAuditPolicy
Name                SubCategory Count
----                ----------------
System              5
Logon/Logoff        11
Object Access       14
Privilege Use       3
Detailed Tracking   6
Policy Change       6
Account Management  6
DS Access           4
Account Logon       4
```

Listing 9-1 The top-level audit policy categories

In the output, you can see the name of each category and its number of sub-categories. Each category also has an associated GUID, but this value is hidden by default. To see it, select the `Id` property from the command's output, as shown in Listing 9-2.

```
PS> Get-NtAuditPolicy | Select-Object Name, Id
Name                Id
----                --
System              69979848-797a-11d9-bed3-505054503030
Logon/Logoff        69979849-797a-11d9-bed3-505054503030
Object Access       6997984a-797a-11d9-bed3-505054503030
--snip--
```

Listing 9-2 Displaying category GUIDs

You can display the sub-categories by using the `ExpandCategory` parameter. In Listing 9-3, we specify the `System` category by name and then expand the output to show its sub-categories.

```
PS> Get-NtAuditPolicy -Category System -ExpandCategory
```

```
Name                      Policy
----                      ------
Security State Change     Unchanged
Security System Extension Unchanged
System Integrity          Unchanged
IPsec Driver              Unchanged
Other System Events       Unchanged
```

Listing 9-3 Displaying the audit policy's sub-categories

You can also select a category by specifying its GUID using the `CategoryGuid` parameter.

The auditing policy is based on these sub-categories; here, they all show the value `Unchanged`, which means no policy has been configured. The sub-category policy can have one or more of the following values:

`Unchanged`
The policy is not configured, and should not be changed.

`Success`
The policy should generate audit events when an auditable resource is opened successfully.

`Failure`
The policy should generate audit events when an auditable resource can't be opened.

`None`
The policy should never generate an audit event.

We can enable kernel object auditing by running the commands shown in Listing 9-4 in an administrator PowerShell console.

```
PS> Enable-NtTokenPrivilege SeSecurityPrivilege
PS> Set-NtAuditPolicy -Category ObjectAccess -Policy Success, Failure -
PassThru
Name                          Policy
----                          ------
File System                   Success, Failure
Registry                      Success, Failure
Kernel Object                 Success, Failure
SAM                           Success, Failure
Certification Services        Success, Failure
```

```
Application Generated          Success, Failure
Handle Manipulation            Success, Failure
File Share                     Success, Failure
Filtering Platform Packet Drop Success, Failure
Filtering Platform Connection  Success, Failure
Other Object Access Events     Success, Failure
Detailed File Share            Success, Failure
Removable Storage              Success, Failure
Central Policy Staging         Success, Failure
```

Listing 9-4 Setting the policy and viewing the resulting `ObjectAccess` audit policy list

We've enabled the `Success` and `Failure` auditing policies for all sub-categories under `Object Access`. To make this modification, we need the `SeSecurityPrivilege` privilege. You can set a single sub-category rather than the entire category by name by using the `SubCategoryName` parameter or specifying the GUID using `SubCategoryGuid`.

We can confirm that the audit policy has been configured correctly by specifying the `PassThru` parameter, which lists the modified sub-category objects. The output displays some important audit policies, including `File System`, `Registry`, and `Kernel Object`, which enable auditing on files, registry keys, and other kernel objects, respectively.

You can run the following command as an administrator to disable the change we made in Listing 9-4:

```
PS> Set-NtAuditPolicy -Category ObjectAccess -Policy None
```

Unless you need to enable the audit policy for some reason, it's best to disable it once you're finished.

Configuring the Per-User Audit Policy

We've shown how to configure the system-wide policy, but it's also possible to configure the audit policy on a per-user basis. You could use this feature to add auditing to a specific user account in cases when the system does not define an overall auditing policy. You could also use it to exclude a specific user account from auditing. To facilitate this behavior, the policy settings differ slightly for per-user policies:

`Unchanged`
The policy is not configured. When set the policy should not be changed.

`SuccessInclude`
Generate audit events on success regardless of the system policy.

`SuccessExclude`
Never generate audit events on success regardless of the system policy.

`FailureInclude`
Generate audit events on failure regardless of the system policy.

`FailureExclude`
Never generate audit events on failure regardless of the system policy.

`None`
Never generate an audit event. Removes the per-user audit entry.

To configure a per-user policy, you can specify an SID to the `User` parameter when using the `Set-NtAuditPolicy` command. This SID must belong to a user account; it can't be a group SID, such as *Administrator*, or a service account, such as *SYSTEM*, or you'll receive an error when setting the policy.

Listing 9-5 configures a per-user policy for the current user. You must run these commands as an administrator.

```
PS> Enable-NtTokenPrivilege SeSecurityPrivilege
PS> $sid = Get-NtSid
PS> Set-NtAuditPolicy -Category ObjectAccess -User $sid -UserPolicy
SuccessExclude
PS> Get-NtAuditPolicy -User $sid -Category ObjectAccess -ExpandCategory
Name              User              Policy
----              ----              ------
File System       GRAPHITE\admin    SuccessExclude
Registry          GRAPHITE\admin    SuccessExclude
Kernel Object     GRAPHITE\admin    SuccessExclude
SAM               GRAPHITE\admin    SuccessExclude
--snip--
```

Listing 9-5 Configuring a per-user audit policy

We specify the user's SID to the `User` parameter and then specify the `SuccessExclude` user policy. This will exclude success audit events for only this user. If you want to remove the per-user policy for a user, you can specify the `None` user policy. You can also enumerate all users who have configured policies using the `AllUser` parameter of `Get-NtAuditPolicy`, as shown in Listing 9-6.

```
PS> Get-NtAuditPolicy -AllUser
Name                User                SubCategory Count
----                ----                ------------------
System              GRAPHITE\admin      5
Logon/Logoff        GRAPHITE\admin      11
Object Access       GRAPHITE\admin      14
--snip--
```

Listing 9-6 Querying per-user policies for all users

To clear the per-user audit policy, run the following command:

```
PS> Set-NtAuditPolicy -Category ObjectAccess -User $sid -UserPolicy None
```

You now know how to query and set policies for the system and for a specific user. Now let's look at how to grant users the access needed to query and set these policies on the system.

Audit Policy Security

To query or set a policy, the caller can enable `SeSecurityPrivilege` on their token. If the privilege is not enabled, LSASS will perform an access check based on a security descriptor in the system configuration. We can configure the following access rights in the security descriptor:

`SetSystemPolicy`
Enables setting the system audit policy

`QuerySystemPolicy`
Enables querying the system audit policy

`SetUserPolicy`
Enables setting a per-user audit policy

`QueryUserPolicy`
Enables querying a per-user audit policy

`EnumerateUsers`
Enables enumerating all per-user audit policies

`SetMiscPolicy`
Enables setting a miscellaneous audit policy

`QueryMiscPolicy`
Enables querying a miscellaneous audit policy

These access rights grant a user the ability to query or set the policy for the system a single user. No standard auditing API seems to use the `SetMiscPolicy` and `QueryMiscPolicy` access rights, but because they are defined in the Windows SDK, I've added them here for completeness.

As an administrator, you can query the currently configured security descriptor by enabling `SeSecurityPrivilege` and using the `Get-NtAuditSecurity` command (Listing 9-7).

```
PS> Enable-NtTokenPrivilege SeSecurityPrivilege
PS> $sd = Get-NtAuditSecurity
PS> Format-NtSecurityDescriptor $sd -Summary -MapGeneric
<DACL>
1 BUILTIN\Administrators: (Allowed)(None)(GenericRead)
  NT AUTHORITY\SYSTEM: (Allowed)(None)(GenericRead)
```

Listing 9-7 Querying and displaying the audit security descriptor

We pass the queried security descriptor to `Format-NtSecurityDescriptor` to display the DACL. Notice that only the *Administrators* and *SYSTEM* groups can access the policy 1. Also, this access is limited to `GenericRead` access, which allows users to query the policy but not modify it. Thus, even administrators would need to enable `SeSecurityPrivilege` to modify the audit policy, as that privilege bypasses any access check.

Note A user who has not been granted read access to the policy can still query the advanced audit categories and sub-categories, which

ignore the security descriptor. However, they won't be granted access to query the configured settings. The `Get-NtAuditPolicy` will return the value of `Unchanged` for audit settings the user wasn't able to query.

If you want to allow non-administrators to change the advanced audit policy, you can change the security descriptor using the `Set-NtAuditSecurity` command. Run the commands in Listing 9-8 as an administrator.

```
PS> Enable-NtTokenPrivilege SeSecurityPrivilege
PS> $sd = Get-NtAuditSecurity
PS> Add-NtSecurityDescriptorAce $sd -Sid "LA" -Access GenericAll
PS> Set-NtAuditSecurity $sd
```

Listing 9-8 Modifying the audit security descriptor

We first query the existing security descriptor for the audit policy and grant the local administrator all access rights. Then, we set the modified security descriptor using the `Set-NtAuditSecurity` command. Now the local administrator can query and modify the audit policy without needing to enable `SeSecurityPrivilege`.

You shouldn't normally reconfigure the security of the audit policy, and you certainly shouldn't grant all users write access. Note that the security descriptor doesn't affect who can query or set the security descriptor itself; only callers with `SeSecurityPrivilege` enabled can do this, no matter the values in the security descriptor.

Configuring the Resource SACL

Just enabling the audit policies isn't enough to start generating audit events. We also need to configure an object's SACL to specify the auditing rules to use . To set the SACL on an object, we'll need to enable `SeSecurityPrivilege`, which can only be done as an administrator (Listing 9-9).

```
PS> $sd = New-NtSecurityDescriptor -Type Mutant
PS> Add-NtSecurityDescriptorAce $sd -Type Audit -Access GenericAll
-Flags SuccessfulAccess, FailedAccess -KnownSid World -MapGeneric
```

```
PS> Enable-NtTokenPrivilege SeSecurityPrivilege
PS> Clear-EventLog -LogName "Security"
PS> Use-NtObject($m = New-NtMutant "ABC" -Win32Path -SecurityDescriptor $sd) {
     Use-NtObject($m2 = Get-NtMutant "ABC" -Win32Path) {
    }
}
```

Listing 9-9 Creating a `Mutant` object with an audit ACE and opening it to generate an event

We create an empty security descriptor and add a single `Audit` ACE to the SACL. Other ACE types you could add include `AuditObject` and `AuditCallback`.

The processing of `Audit` ACEs looks a lot like the discretionary access check we described in Chapter 7. The SID must match a group in the calling token (including any `DenyOnly` SIDs), and the access mask must match one or more bits of the granted access. The *Everyone* group SID is a special case; it will always match, regardless of whether the SID is available in the token.

In addition to any of the usual inheritance ACE flags, such as `InheritOnly`, the `Audit` ACE must specify one or both of the `SuccessfulAccess` and `FailedAccess` flags, which provide the auditing code with the conditions in which it should generate the audit entry.

We'll assign the SACL to a `Mutant` object. But before creating the `Mutant` object, we need to enable `SeSecurityPrivilege`. If we don't, the creation will fail. To make it easier to see the generated audit event, we can also clear the security event log. Next, we create the object, passing it the SACL we built, and then reopen it to trigger the generation of an audit log.

Now we can query the security event log using `Get-WinEvent`, passing it the event ID 4656 to find the generated audit event (Listing 9-10).

```
PS> $filter = @{logname='Security';id=@(4656)}
PS> Get-WinEvent -FilterHashtable $filter | Select-Object -ExpandProperty
Message
A handle to an object was requested.
Subject:
```

```
        Security ID:    S-1-5-21-2318445812-3516008893-216915059-1002
        Account Name:   user
        Account Domain: GRAPHITE
        Logon ID:       0x524D0

Object:
        Object Server:  Security
        Object Type:    Mutant
        Object Name:    \Sessions\2\BaseNamedObjects\ABC
        Handle ID:      0xfb4
        Resource Attributes:    -

Process Information:
        Process ID:     0xaac
        Process Name:
C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe

Access Request Information:
        Transaction ID:         {00000000-0000-0000-0000-000000000000}
        Accesses:               DELETE
                                READ_CONTROL
                                WRITE_DAC
                                WRITE_OWNER
                                SYNCHRONIZE
                                Query mutant state

        Access Reasons:         -
        Access Mask:            0x1F0001
        Privileges Used for Access Check:       -
        Restricted SID Count:   0
```

Listing 9-10 Viewing the open audit event for the `Mutant` object

We first set up a filter for the security event log and event ID 4656, which corresponds to the opening of a handle. We then use the filter with `Get-WinEvent` and select the event's textual message.

The output begins with this textual description of the event, which confirms that it was generated in response to a handle being opened. After this comes the `Subject`, which includes the user's information, including their SID and username. To look up the username, the kernel sends the audit event to the LSASS process.

Next are the details of the opened object. These include the object server (`Security`, representing the SRM), the object type

(`Mutant`), and the native path to the object. One key value, `HandleId`, is the handle number for the object. If you query the handle value returned from the `NtCreateMutant` system call, it should match this value. We then get some basic process information, and finally, information about the access granted to the handle.

How can distinguish between success and failure events? The best way to do this is to extract the `KeywordsDisplayNames` property, which contains either `Audit Success` if the handle was opened or `Audit Failure` if the handle could not be opened. Listing 9-11 shows an example.

```
PS> Get-WinEvent -FilterHashtable $filter | Select-Object KeywordsDisplayNames
KeywordsDisplayNames
--------------------
{Audit Success}
{Audit Failure}
--snip--
```

Listing 9-11 Extracting `KeywordsDisplayName` to view the success or failure status

When you close the handle to the object, you'll get another audit event, with the event ID 4658, as shown in Listing 9-12.

```
PS> $filter = @{logname='Security';id=@(4658)}
PS> Get-WinEvent -FilterHashtable $filter | Select-Object -ExpandProperty
Message
The handle to an object was closed.
Subject :
        Security ID:    S-1-5-21-2318445812-3516008893-216915059-1002
        Account Name:   user
        Account Domain: GRAPHITE
        Logon ID:       0x524D0

Object:
        Object Server:  Security
        Handle ID:      0xfb4

Process Information:
        Process ID:     0xaac
        Process Name:
C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
```

Listing 9-12 Viewing the audit event generated when the `Mutant` object handle is closed

You might notice that the information provided about the closing of the object handle is slightly less detailed than the

information generated when the handle was opened. You can manually correlate the open and close handle events by using the handle IDs, which should match.

It's possible to generate object audit events manually from user-mode using some additional system calls. To do so, you need the `SeAuditPrivilege`, which is typically only granted to the *SYSTEM* account, and not to normal administrators.

You can generate the audit event at the same time as an access check using the `NtAccessCheckAndAuditAlarm` system call. This access check system call has all of the same object ACE variants as the normal access checks do. You can access it using the `Get-NtGrantedAccess` PowerShell command with the `Audit` parameter.

You can also generate events manually using the `NtOpenObjectAuditAlarm` and `NtCloseObjectAuditAlarm` system calls, which PowerShell exposes through the `Write-NtAudit` command. Run the commands in Listing 9-13 as the *SYSTEM* user to manually generate audit log events.

```
1 PS> Enable-NtTokenPrivilege SeAuditPrivilege -WarningAction Stop
  PS> $owner = Get-NtSid -KnownSid Null
  PS> $sd = New-NtSecurityDescriptor -Type Mutant -Owner $owner -Group $owner
  PS> Add-NtSecurityDescriptorAce $sd -KnownSid World -Access GenericAll -
  MapGeneric
  PS> Add-NtSecurityDescriptorAce $sd -Type Audit -Access GenericAll
2 -Flags SuccessfulAccess, FailedAccess -KnownSid World -MapGeneric

3 PS> $handle = 0x1234
4 PS> $r = Get-NtGrantedAccess $sd -Audit -SubsystemName "SuperSecurity"
  -ObjectTypeName "Badger" -ObjectName "ABC" -ObjectCreation
  -HandleId $handle -PassResult
5 PS> Write-NtAudit -Close -SubsystemName "SuperSecurity" -HandleId $handle
  -GenerateOnClose:$r.GenerateOnClose
```

Listing 9-13 Manually generating audit log events

We must first enable `SeAuditPrivilege`; otherwise, the rest of the script will fail 1. This privilege must be enabled on the primary token, as you can't impersonate a token with the

privilege, which is why you must run the PowerShell instance as the *SYSTEM* user.

After enabling the privilege, we build a security descriptor with a SACL to audit success and failure access [2]. We generate a fake handle ID [3]; this value would be the kernel handle in a normal audit event, but when we generate an event from user mode, it can be any value we like. We can then run the access check, specifying the `Audit` parameter, which enables the other auditing parameters [4]. We need to specify the `SubsystemName`, `ObjectTypeName`, and `ObjectName` parameters, which can be completely arbitrary. We also specify the handle ID.

In the output, we receive an access check result with one additional property: `GenerateOnClose`, which indicates whether we need to write a closed handle event. Calling the `Write-NtAudit` command and specifying the `Close` parameter will call the `NtCloseObjectAuditAlarm` system call to generate the event. We do so, specifying the `GenerateOnClose` value from the result [5]. If `GenerateOnClose` were false, we would still need to write the close event to complete the audit, but the actual close event would not be written to the audit log.

If you don't receive any auditing events when you run the commands in Listing 9-13, ensure that you've enabled object auditing, as we did in Listing 9-4.

THE MYSTERIOUS ALARM ACE

If you've look at the lists of ACE types, you might have noticed an `Alarm` ACE type that is related to auditing. (Also, all of the system calls we've described in this chapter for manually generating audit events in user mode end with the term `AuditAlarm`.) But if you read the online MSDN documentation for this ACE type, you'll see the phrase, "The `SYSTEM_ALARM_ACE` structure is reserved for future use." What is the purpose of this ACE type if it has always been reserved?

It's hard to tell. Kernel code checked for the `Alarm` ACE type starting in Windows NT 3.1 until Microsoft removed the check in Windows XP. The Windows developers even defined `AlarmCallback`, `AlarmObject` and `AlarmObjectCallback` variants, though code doesn't seem to have checked these in the Windows 2000 kernel, where object ACEs were introduced. It is clear from old kernels that the `Alarm` ACE type was handled; less clear is whether an `Alarm` ACE could generate an event to be monitored. Even in the MSDN documentation for versions of Windows that handled the

`Alarm` ACE type, it is marked as unsupported.

As to what the `Alarm` ACE might have done, it's likely a holdover from Windows NT's VMS roots. VMS had a similar security model to Windows NT, including the use of ACLs and ACEs. In VMS, audit ACEs wrote to an audit log file, as on Windows, and the `Alarm` ACEs would generate real-time ephemeral security events on the operator's terminal once a user enabled alarms using the `REPLY/ENABLE=SECURITY` command. It's likely that Microsoft added support to the Windows kernel for this ACE type but never implemented the ability to send these real-time events. With modern logging alternatives such as Event Tracing for Windows (ETW), which provides much more comprehensive security information in real-time, the chances of Microsoft reintroducing the `Alarm` ACE in the future are slim.

Configuring the Global SACL

Correctly configuring the SACL for every resource can be difficult, as well as time consuming. For this reason, the advanced audit policy allows you to configure a global SACL for files or registry keys. The system will use this global SACL if no SACL exists for a resource, and if a resource already has a SACL, it will merge the global and resource SACLs. Because these broad auditing configurations can swamp your logging output and impede your ability to monitor events, use global SACLs sparingly.

You can query the global SACL by specifying either the `File` or `Key` value to the `GlobalSacl` parameter of the `Get-NtAuditSecurity` PowerShell command. You can also modify the global SACL with the `Set-NtAuditSecurity` command, specifying the same `GlobalSacl` parameter. To test this behavior, run the commands in Listing 9-14 as an administrator.

```
PS> Enable-NtTokenPrivilege SeSecurityPrivilege
PS> $sd = New-NtSecurityDescriptor -Type File
PS> Add-NtSecurityDescriptorAce $sd -Type Audit -KnownSid World
-Access WriteData -Flags SuccessfulAccess
PS> Set-NtAuditSecurity -GlobalSacl File -SecurityDescriptor $sd
PS> Get-NtAuditSecurity -GlobalSacl File |
Format-NtSecurityDescriptor -SecurityInformation Sacl -Summary
<SACL>
Everyone: (Audit)(SuccessfulAccess)(WriteData)
```

Listing 9-14 Setting and querying the global file SACL

We start by building a security descriptor containing a SACL with a single `Audit` ACE, as we've done earlier in the chapter. We then call `Set-NtAuditSecurity` to set the global SACL for the `File` type. Finally, we query the global SACL to make sure it's set correctly.

You can remove the global SACL by passing a security descriptor with a NULL SACL to `Set-NtAuditSecurity`. To create this security descriptor, use the following command:

```
PS> $sd = New-NtSecurityDescriptor -NullSacl
```

Then use the command we covered in Listing 9-14 to clear the SACL.

Worked Examples

Let's wrap up with some worked examples that use the commands you learned about in this chapter.

Verifying Audit Access Security

When we're checking whether malicious code has compromised an untrusted Windows system, it's a good idea to verify that the security settings haven't been modified. One check you might want to perform is determining whether a non-administrator user has the access needed to change the audit policy on the system. If a non-administrator user can change the policy, they could disable auditing and hide their access to sensitive resources.

You can inspect the audit policy's security descriptor manually, or do so using the `Get-NtGrantedAccess` PowerShell command. Run the commands in Listing 9-15 as an administrator.

```
PS> Enable-NtTokenPrivilege SeSecurityPrivilege
PS> $sd = Get-NtAuditSecurity
PS> Set-NtSecurityDescriptorOwner $sd -KnownSid LocalSystem
PS> Set-NtSecurityDescriptorGroup $sd -KnownSid LocalSystem
PS> Get-NtGrantedAccess $sd -PassResult
Status          Granted Access Privileges
```

```
------          -------------- ----------
STATUS_SUCCESS  GenericRead    NONE

PS> Use-NtObject($token = Get-NtToken -Filtered -Flags LuaToken) {
    Get-NtGrantedAccess $sd -Token $token -PassResult
}
Status               Granted Access Privileges
------               -------------- ----------
STATUS_ACCESS_DENIED 0              NONE
```

Listing 9-15 Performing an access check on the audit policy security descriptor

We start by querying for the audit policy security descriptor and setting the owner and group fields. The security descriptor returned from `Get-NtAuditSecurity` does not contain these fields, but they're required for the access check process, so we set them here.

We can then pass the security descriptor to the `Get-NtGrantedAccess` command to check it against the current administrator token. The result indicates the caller has `GenericRead` access to the audit policy, which allows them to query the policy but not set it without enabling `SeSecurityPrivilege`.

Finally, we can remove the administrator group from the token by creating a filtered token with the `LuaToken` flag. Running the access check with the filtered token indicates that it has no granted access to the audit policy (not even read access). If this second check returns a status other than *access denied*, you can conclude that the default audit policy security descriptor has been changed, and it's worth checking whether this occurred intentionally or maliciously.

Finding Resources with Audit ACEs

Most resources aren't configured with a SACL. So, you might want to enumerate the resources on the system that have a SACL, as this can help you understand what resources might generate audit log events. Listing 9-16 provides a simple example in which we find these resources. Run the commands as an administrator.

```
PS> Enable-NtTokenPrivilege SeDebugPrivilege, SeSecurityPrivilege
```

```
1 PS> $ps = Get-NtProcess -Access QueryLimitedInformation, AccessSystemSecurity
-FilterScript {
  2 $sd = Get-NtSecurityDescriptor $_ -SecurityInformation Sacl
    $sd.HasAuditAce
}

3 PS> $ps | Format-NtSecurityDescriptor -SecurityInformation Sacl
Path: \Device\HarddiskVolume3\Windows\System32\lsass.exe
Type: Process
Control: SaclPresent

<SACL>
 - Type  : Audit
 - Name  : Everyone
 - SID   : S-1-1-0
 - Mask  : 0x00000010
4 - Access: VmRead
 - Flags : SuccessfulAccess, FailedAccess

PS> $ps.Close()
```

Listing 9-16 Finding processes with configured SACLs

We focus on process objects here, but you can apply this same approach to other resource types.

We first open all processes for `QueryLimitedInformation` and `AccessSystemSecurity` access 1. We apply a filter to the processes, querying for the SACL from the process object, then returning the value of the `HasAuditAce` property 2. This property indicates whether the security descriptor has at least one audit ACE.

We then pipe the results returned from the `Get-NtProcess` command into `Format-NtSecurityDescriptor` to display the SACLs 3. In this case, there is only a single entry, for the LSASS process. We can see that the audit ACE logs an event whenever the LSASS process is opened for `VmRead` access 4.

This policy is a default audit configuration on Windows, used to detect access to the LSASS process. The `VmRead` access right allows a caller to read the virtual memory of a process, and this ACE aims to detect the extraction the LSASS memory contents, which can include passwords and other authentication credentials.

If the process is opened for any other access right, no audit log entry will be generated.

Wrapping Up

In this chapter, we covered the basics of security auditing. We started with a description of the security event log and the types of log entries you might find when auditing resource access. Next, we checked the auditing policy configuration, and set advanced auditing policies with the `Set-NtAuditPolicy` command. We also discussed how Windows controls access to the auditing policy and showed the importance of the `SeSecurityPrivilege` privilege, used for almost all audit-related configuration.

To enable auditing on an object, we must modify the SACL to define rules for generating the events enabled by the policy. We walked through examples of generating audit events automatically, using the SACL, and manually, during a user-mode access check.

We've now covered all aspects of the SRM: security access tokens, security descriptors, access checking, and auditing. In the rest of this book, we'll explore the various parts of the Windows platform that the SRM secures.