НАВЧАЛЬНІ ВИДАННЯ

Гладких В.М, Криворучко О.В., Зінченко О.В.

БЕЗПЕКА ОПЕРАЦІЙНОЇ СИСТЕМИ WINDOWS

Навчальний посібник
частина 1

Гладких В.М, Криворучко О.В., Зінченко О.В.

БЕЗПЕКА ОПЕРАЦІЙНОЇ СИСТЕМИ WINDOWS

Навчальний посібник
частина 1

КИЇВ
2025

УДК 004.45
ББК 32.972.11
П43

Рекомендовано до друку Вченою радою Національного університету біоресурсів і природокористування України (протокол № 5 від 27 листопада 2025 р.)

Рецензенти:

Остапов С. Е., доктор фізико-математичних наук, професор, професор кафедри програмного забезпечення комп'ютерних систем Чернівецького національного університету імені Ю. Федьковича.

Вишнівський В. В., доктор технічних наук, завідувач кафедри комп'ютерних наук Державного університету інформаційно-комунікаційних технологій.

Цюцюра М. І., доктор технічних наук, професор, професор кафедри комп'ютерних наук НУБіП України.

П43 Операційні системи : БЕЗПЕКА ОПЕРАЦІЙНОЇ СИСТЕМИ WINDOWS. Навчальний посібник частина 1 // В.М. Гладких, О.В. Криворучко, О.В. Зінченко. Київ: НУБіП України, 2025.- 306 с.

ISBN

Навчальний посібник «БЕЗПЕКА ОПЕРАЦІЙНОЇ СИСТЕМИ WINDOWS» призначений для здійснення самостійної теоретичної та практичної підготовки студентів закладів вищої освіти за спеціальностями F5 «Кібербезпека та захист інформації», F3 «Комп'ютерні науки» і F2 «Інженерія програмного забезпечення» В посібнику розглядається основні принципи, методи та алгоритми забезпечення безпеки даних, сервісів та додатків на сучасних версіях операційної системи Windows. функції безпеки. Кожен розділ закінчується декількома практичними прикладами, написаними на PowerShell, які допоможуть краще зрозуміти команди, представлені в розділі.

УДК 004.45

ISBN

ВІДОМОСТІ ПРО АВТОРІВ

Гладких Валерій Миколайович
доцент кафедри комп'ютерних систем, мереж та кібербезпеки Національного університету біоресурсів і природокористування України, Україна. Кандидат технічних наук, доцент.
Коло наукових інтересів: Аналіз на ідентифікація мережевого трафіку, кібербезпека, захист інформації, телекомунікаційні системи.
Електронна адреса: v.hladkykh@nubip.edu.ua

Криворучко Олена Володимирівна
професор кафедри комп'ютерних систем, мереж та кібербезпеки Національного університету біоресурсів і природокористування України, Україна. Доктор технічних наук, професор.
Коло наукових інтересів: управління проектами, Product IT, Project IT, інформаційні технології, інтелектуальні системи, захист інформації, інформаційна безпека, аудит інформаційних систем, менеджмент проектів програмного забезпечення, якість та стандартизація програмного забезпечення.
Електронна адреса: o.kryvoruchko@nubip.edu.ua

Зінченко Ольга Валеріївна
завідувач кафедри штучного інтелекту Державного університету інформаційно-комунікаційних технологій, доктор технічних наук, доцент, Україна.
Коло наукових інтересів: методи та засоби штучного інтелекту, проектування систем штучного інтелекту, технології прийняття рішень, аналіз і обробка великих даних.
Електронна адреса: zinchenkoov@gmail.com

ЗМІСТ

ПЕРЕДМОВА

Мільйони пристроїв по всьому світу використовують платформу Microsoft Windows. Багато найбільших світових компаній покладаються на її безпеку для захисту своїх даних і комунікацій.

Windows відіграє важливу роль у забезпеченні безпеки сучасного Інтернету. З іншого боку ця операційна система вона є популярною мішенню для атак.

Починаючи з Windows NT компанія Microsoft почала включати безпеку у свій дизайн. Починаючи з 1993 року в цій ОС було запроваджено облікові записи користувачів та контроль над ресурсами. За понад 30 років, що минули з того часу, багато чого змінилося в безпеці Windows. Microsoft значно захистила платформу від атак, замінила свій початковий процес автентифікації сучасними технологіями, надала механізму контролю доступу додаткові можливості та.

Сьогодні безпека платформи Windows є досить складною, і багато атак базуються на зловживанні цією складністю. На жаль, документація Microsoft у цій галузі може бути недостатньою. Оскільки Windows не є відкритим програмним забезпеченням, іноді єдиним способом зрозуміти його безпеку є проведення поглиблених досліджень та аналізів.

Цій посібник передбачає достатнє знайомство з інтерфейсом користувача Windows та його основними операціями, наприклад, такими як робота з файлами. Проте, вам не потрібно бути експертом з Windows низького рівня.

Ми значною мірою покладаємось на використання сценаріїв PowerShell, тому вам буде корисно мати певний досвід роботи з цією мовою, а також з фреймворком .NET, на якому вона базується.

Щоб ви могли швидко ознайомитися з усім, перший розділ містить стислий огляд деяких функцій PowerShell.

У кожному розділі розглядається основні функції безпеки, реалізовані в сучасних версіях Windows. Розділ закінчується декількома практичними прикладами, написаними на PowerShell, які допоможуть вам краще зрозуміти команди, представлені в розділі. Ось короткий огляд того, що розглядається в кожному розділі.

Розділ 1: Налаштування середовища тестування PowerShell. У цьому розділі ви налаштуєте PowerShell для запуску прикладів, що містяться в наступних розділах. У розділі також наведено огляд мови сценаріїв PowerShell.

Розділ 2: *Ядро Windows*. У цьому розділі розглядаються основи ядра Windows та його інтерфейсу системних викликів — тема, яка має ключове значення для формування глибокого розуміння безпеки Windows.

Розділ 3: *Додатки користувального режиму*. Більшість додатків не використовують безпосередньо інтерфейс системних викликів ядра. Замість цього вони користуються набором інтерфейсів високого рівня. У цьому розділі розглядаються такі функціональні можливості Windows, як обробка файлів і реєстр.

Розділ 4: Токени доступу. Windows призначає кожному запущеному процесу токен доступу, який являє собою ідентифікатор користувача в системі. У цьому розділі описано різні компоненти, що зберігаються в токені і використовуються для перевірки доступу.

Розділ 5: Дескриптор безпеки. Кожен захищений ресурс потребує опису того, хто може отримати доступ і який тип доступу йому надається. Саме для цього призначені дескриптори безпеки. У цьому розділі ми розглянемо їх внутрішню структуру та способи їх створення і редагування.

Розділ 6: Читання та призначення дескрипторів безпеки. Для перевірки безпеки необхідно мати можливість запитувати дескриптор безпеки ресурсу. У цьому розділі пояснюється, як відбувається такий запит для різних типів ресурсів. Також розглядаються численні способи, якими Windows призначає дескриптори безпеки ресурсам.

Розділ 7: Процес перевірки доступу. Windows використовує перевірку доступу, для визначення, який доступ надати користувачеві до ресурсу. Ця операція бере токен і дескриптор безпеки та використовує певний алгоритм для визначення наданого доступу. У цьому розділі розглядається такий алгоритм та його реалізацію в PowerShell.

Розділ 8: Інші випадки використання перевірки доступу. У цьому розділі розглядаються ці альтернативні випадки використання перевірки доступу.

Розділ 9: Аудит безпеки. Процес перевірки доступу також може створювати журнали з інформацією про ресурси, до яких користувач отримав доступ, та про рівень доступу. У цьому розділі розглядаються політики аудиту системи.

Частка внеску кожного з авторів: Гладких В.М. – розділи 6-9 та частково розділи 2, 5; Криворучко О.В. – розділи 1, 2, 4, 5 та частково розділи 8, 9; Зінченко О.В. – розділ 3 та частково розділи 5, 7.

РОЗДІЛ 1. НАЛАШТУВАННЯ СЕРЕДОВИЩА ТЕСТУВАННЯ POWERSHELL

У цьому розділі розглянемо налаштування PowerShell. Це необхідно для роботи з прикладами коду. Також ми коротко, повторимо мову PowerShell, включаючи його типи, змінні та вирази. Основні питання:

— як виконуються команди,
— як отримати довідку,
— як експортувати дані для подальшого використання.

1.1. ВИБІР ВЕРСІЇ POWERSHELL

Найважливішим інструментом, який вам необхідний для читання та розуміння використання цього посібника, є PowerShell. Ця програма встановлена в операційній системі (починаючи з Windows 7) за замовчанням.

Існує безліч різних версій цієї програми. Версія, встановлена за замовчанням на час написання посібника, версія PowerShell - 5.1.

Microsoft більше не підтримує її повністю, хоча вона повністю підходить для вивчення. Пізніші версії PowerShell є кроссплатформенними, мають відкритий вихідний код, але ці версії повинні бути встановлені окремо у Windows зі сторінки (Microsoft).

Код, представлений у цьому посібнику, працює як у PowerShell 5.1, так і в останній версії (на момент написання посібника PowerShell-7.5.0-win-x64 - 2025-02-24) з відкритим вихідним кодом, тому не має значення, яку ви виберете. Якщо ви хочете використовувати PowerShell з відкритим вихідним кодом, відвідайте сторінку проекту на GitHub (Github.com), щоб отримати допомогу в інсталяції для вашої версії Windows.

1.2. НАЛАШТУВАННЯ POWERSHELL

Починати роботу в PowerShell необхідно з установки політики виконання скриптів, яка визначатиме, які типи скриптів може виконувати ця програма. Для версій Windows, що працюють під керуванням PowerShell 5.1, блокується виконання всіх скриптів, не підписаних довіреним сертифікатом, тобто за замовчуванням встановлено значення *Restricted*.

Оскільки скрипти в цьому посібнику не підписані, необхідно змінити політику виконання на *RemoteSigned*. Ця політика дозволить нам запускати непідписані скрипти PowerShell, якщо вони створені локально. Але скрипти завантажені з мережі Internet або прикріплені до електронної пошти виконуватись не будуть.

Щоб встановити політику виконання, необхідно виконати наступну команду:

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

Ця команда змінить політику виконання скриптів лише для поточного користувача, а не системи. Якщо потрібно змінити її для всіх користувачів, необхідно запустити PowerShell від імені адміністратора, а потім виконати команду, вилучивши параметр *-Scope*.

На версіях операційної системи Windows Server, політика виконання скриптів за замовчуванням *RemoteSigned*, і немає необхідності змінювати цю політику.

Тепер, коли ми можемо запускати непідписані скрипти, можна встановити модуль PowerShell, який будемо використовувати в цьому посібнику.

Модуль PowerShell – це пакет скриптів та двійкових файлів .NET, які експортують команди PowerShell. Кожна інсталяція PowerShell постачається з встановленими декількома модулями для завдань, починаючи від налаштування програм і закінчуючи налаштуванням Windows Update. Ви можете встановити модуль вручну, скопіювавши його файли, (Покажчик_місця_заповнення1) але найпростіший підхід використовувати галерею (Powershell), онлайн-репозиторій модулів.

Щоб встановити модуль із репозиторію PowerShell, ми використовуємо команду PowerShell *Install-Module*. Для цього посібника нам потрібно встановити модуль *NtObject Manager*, що можна зробити за допомогою наступної команди:

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

Обов'язково відповідайте «так», якщо інсталятор додатку поставить вам будь-які питання (звичайно, після того, як ви прочитаєте і зрозумієте питання). Якщо у вас вже встановлено модуль, ви можете переконатися, що у вас встановлена остання версія, використовуючи команду *Update-Module*:

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

Після встановлення ви можете завантажити модуль за допомогою команди *Import-Module*:

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

Якщо ви бачите будь-які помилки після імпорту модуля, двічі перевірте, чи правильно ви встановили політику виконання, це найбільш поширена причина неправильного завантаження модуля.

Як останній тест давайте запустимо команду, яка йде з модулем, щоб перевірити його роботу. Виконайте команду в лістингу 1.1 та переконайтеся, що висновок відповідає тому, що ви бачите у консолі PowerShell. Ми розглянемо призначення цієї команди у наступному розділі.

Лістинг 1.1. Перевірка працездатності модуля *NtObjectManager*

```
PS C:\Users\hladk> New-NtSecurityDescriptor

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

Якщо все працює і ви добре володієте PowerShell, можете переходити до наступного розділу. Якщо вам потрібно швидко оновити свої знання про мову PowerShell, продовжуйте читати.

1.3. ОГЛЯД МОВИ POWERSHELL

Повне введення PowerShell виходить за рамки даного посібника.

Щоб використати посібник максимально ефективно, у цьому підрозділі розглянемо мовні можливості.

1.3.1. Типи, змінних та виразів

PowerShell підтримує багато різних типів, від простих цілих чисел і рядків до складних об'єктів. У таб.1.1 показані деякі з поширених вбудованих типів, а також базові типи середовища виконання .NET.

Таблиця 1.1.

Загальні базові типи PowerShell з типами .NET та приклади

Type	.NET type	Приклади
int	System.Int32	142, 0x8E,0216
long	System.Int64	142L, Ox8EF
string	System. String	”Hello World!'
double	System.Double	1.00
bool	System.Boolean	Strue, Sfalse
array	System.Object[]	@ (1, "ABC", Strue)
hashtable	System.Collections.Hashtable	@ {A=1; B="ABC"}

Для виконання обчислень над базовими типами оператори можуть бути перевантажені, наприклад, + використовується для складання, а також для конкатенації рядків і масивів.

Таб. 1.2 містить прості поширені оператори з простими прикладами та результатами використання оператора.

Самостійно протестуйте приклади, щоб перевірити виведення кожного оператора.

Надавати значення змінним можна за допомогою оператора присвоєння =.

Змінна має буквено-цифрове ім'я із префіксом у вигляді символу $. Наприклад, у лістингу 1.2 показано, як можна захопити масив у змінній та використовувати оператор індексації для пошуку значення.

Таблиця 1.2.
Приклади обчислень з поширеними операторами

Оператор	Назва	Приклад	Результат
+	Addition or concatenation	1 + 2, "Hello" + "World!"	3, "HelioWorld!"
-	Subtraction	2-1	1
*	Multiplication	2*4	8
/	Division	8/4	2
%	Modulus	6 % 4	2
[]	Index	@(3, 2, 1, 0)[l]	2
-f	String formatter	”Ox{O:X} {1}" -f 42, 123	"0X2A 123"
-band	Bitwise AND	OxlFF -band OxFF	255
-bor	Bitwise OR	0x100 -bor 0x20	288
-bxor	Bitwise XOR	OxCC -bxor OxDD	17
-bnot	Bitwise NOT	-bnot OxEE	-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
-1t	Less than	4 -It 10	$true
-gt	Greater than	4 -gt 10	$false

Лістинг 1.2. Захоплення масиву в змінну та індексація його на ім'я змінної

```
PS C:\Users\hladk> $var = 3, 2, 1, 0
PS C:\Users\hladk> $var[1]
2
```

Є також деякі наперед визначені змінні, які ми будемо використовувати в частині нашого посібника:

$null - Вказує на значення NULL, яке свідчить на відсутність значення в порівняннях;

$pwd - містить поточний робочий каталог;

$pid - містить ідентифікатор процесу оболонки;

$env - отримує доступ до середовища процесу (наприклад, *$env: WinDir* - для отримання каталогу Windows).

Є можливість перерахувати всі змінні за допомогою *Get-Variable*.

У таб.1.1 ви могли помітити, що було два приклади рядків, один із використанням подвійних лапок, а інший з використанням одинарних лапок. Одна з відмінностей між ними полягає в тому, що рядок у подвійних лапках підтримує інтерполяцію рядків, коли ви вставляєте ім'я змінної в рядок як заповнювач, а PowerShell включає його значення в результат. У лістингу 1.3 показано, що відбувається, коли ви робите це в рядках у подвійних та одинарних лапках.

Лістинг 1.3. Приклади інтерполяції рядків

```
PS C:\Users\hladk> $var = 42
PS C:\Users\hladk> "The magic number is $var"
The magic number is 42
PS C:\Users\hladk> 'It is not $var'
It is not $var
```

Спочатку ми визначаємо змінну значення 42 для вставки в рядок. Потім ~~ми~~ створюємо рядок у подвійних лапках з ім'ям змінної всередині. Результатом є рядок з ім'ям змінної, заміненим на його значення, відформатований як рядок. (Якщо вам потрібен більший контроль над форматуванням, ви можете використовувати оператор форматування рядка, переліченим у таб.1.2.) Щоб продемонструвати різну поведінку рядка в одинарних лапках, ми визначаємо одну з них з ім'ям змінної в рядку. Ми можемо помітити, що у цьому випадку ім'я змінної копіюється дослівно і замінюється значенням.

Інша відмінність полягає в тому, що рядок у подвійних лапках може містити *escape* символи, які ігноруються в одинарних лапках. Ці символи використовують синтаксис, схожий на синтаксис мови програмування C, але замість символу зворотний слеш (\) PowerShell використовує зворотну лапку (`).

Це пов'язано з тим, що Windows використовує зворотний слеш як роздільник шляху, і записування шляхів до файлів було б дуже незручним, якби доводилося використовувати *escape* символи для кожного зворотного слеша. У таб.1.3 наведено список *escape* символів, які можна використовувати в PowerShell.

Якщо ви хочете вставити подвійну лапку в рядок, що міститься в подвійних лапках, вам потрібно використовувати символ `" escape. Щоб вставити одинарну лапку в рядок, вкладений в одинарні лапки, ви подвоюєте символ лапки. Наприклад: 'Hello' 'There' перетворюється в Hello'There. Зверніть увагу на згадку символу NUL у цій таблиці. На відміну від мови C, додавання NUL не призведе до передчасного завершення рядка.

Таблиця 1.3.
Escape символи

Escapes символ	Назва
'0	Символ NUL
'a	Bell
'b	Backspace
'n	Line feed
'r	Carriage return
't	Horizontal tab
'v	Vertical tab
''	Символ зворотного апострофа
' “	Подвійні лапки

Оскільки всі значення є типами .NET, ми можемо викликати методи та отримувати доступ до властивостей об'єкта. Наприклад, наступний код викликає метод *ToCharArray* для рядка, щоб перетворити його на масив окремих символів:

```
PS C:\Users\hladk> "Hello".ToCharArray()
H
e
l
l
o
```

За допомогою PowerShell можна створити майже будь-який тип .NET. Найпростіший спосіб зробити це — перетворити значення в цей тип, вказавши тип .NET у квадратних дужках. Під час перетворення PowerShell спробує знайти відповідний конструктор для типу, який потрібно викликати. Наприклад, наступна команда перетворить рядок в об'єкт *System.Guid.* PowerShell знайде конструктор, який приймає рядок, і викличе його:

```
PS C:\Users\hladk> [System.Guid]"6c0a3a17-4459-4339-a3b6-1cdb1b3e8973"

Guid
----
6c0a3a17-4459-4339-a3b6-1cdb1b3e8973
```

Ви також можете явно викликати конструктор, викликавши новий метод для типу. Попередній приклад можна переписати наступним чином:

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

Guid
----
6c0a3a17-4459-4339-a3b6-1cdb1b3e8973
```

Ця синтаксична конструкція також може використовуватися для виклику статичних методів типу. Наприклад, наступний код викликає статичний метод *NewGuid* для створення нового випадкового глобальна унікального ідентифікатора (GUID):

```
PS C:\Users\hladk> [System.Guid]::NewGuid()

Guid
----
0688856e-8661-4673-9092-bf566523a073
```

Можна створювати нові об'єкти, використовуючи команду *New-Object:*

```
PS C:\Users\hladk> New-Object -TypeName Guid -ArgumentList "6c0a3a17-4459-4339-a3b6-1cdb1b3e8973"

Guid
----
6c0a3a17-4459-4339-a3b6-1cdb1b3e8973
```

Цей приклад еквівалентний виклику статичної функції *new.*

1.3.2. Виконання команд

Майже всі команди PowerShell іменуються з використанням загального шаблону: дієслово та іменник, розділені тире. Наприклад, розглянемо команду *Get-Item*. Дієслово *Get* має на увазі отримання існуючого ресурсу, в той же час як *Item* - це тип ресурсу, що повертається.

Кожна команда може приймати список параметрів, які керують поведінкою команди. Наприклад, команда *Get-Item* приймає параметр *Path*, який вказує існуючий ресурс для отримання, як показано нижче:

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

Параметр Path є *позиційним* параметром. Це означає, що ви можете опустити ім'я параметра, і PowerShell зробить все можливе, щоб вибрати найкращий варіант. Отже, попередню команду також можна записати так:

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

Якщо параметр приймає значення рядка, а рядок не містить спеціальних символів або пробілів, то не потрібно використовувати лапки навколо рядка. Наприклад, команда *Get-Item* також працюватиме з наступним:

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


    Каталог: C:\


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----         1/14/2025   5:43 AM                Windows
```

Результатом виконання однієї команди є нуль або більше значень, які можуть бути простими або складними типами об'єктів. Ви можете передати результат виконання однієї команди в іншу як вхідні дані за допомогою конвеєра, який позначається вертикальною рискою |. Приклади використання конвеєра ми розглянемо пізніше в цьому розділі, коли будемо говорити про фільтрування, групування та сортування.

Ви можете зберегти результат всієї команди або конвеєра в змінній, а потім інтерактивна працювати. Наприклад, наступний код зберігає результат команди *Get-Item* і запитує властивість *FullName*:

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

Якщо ви не хочете зберігати результат у змінній, ви можете взяти команду в дужки і безпосередньо отримати доступ до її властивостей та методів:

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

Довжина командного рядка є фактично нескінченною. Однак, для кращої читабельності команд, бажано розділяти довгі рядки. Оболонка автоматично розділяє рядок на символи. Якщо вам потрібно розділити довгий рядок без спеціальних символів, ви можете використовувати символ зворотної лапки, а потім почати новий рядок. Зворотна лапка повинна бути останнім символом у рядку, інакше під час аналізу скрипта виникне помилка.

1.3.3. Пошук команд та отримання довідки

Стандартна інсталяція PowerShell містить сотні команд на вибір. Це означає, що знайти команду для виконання конкретного завдання може бути складно, і навіть якщо ви знайдете команду, може бути незрозуміло, як її використовувати. Для допомоги ви можете скористатися двома вбудованими командами: *Get-Command* і *Get-Help*.

Команда *Get-Command* може бути використана для переліку всіх доступних вам команд. У найпростішій формі ви можете виконати її без будь-яких параметрів, і вона виведе всі команди з усіх модулів. Однак, ймовірно, корисніше буде відфільтрувати конкретне слово, яке вас цікавить. Наприклад, лістинг 1.4 виведе тільки команди, в назві яких є слово SecurityDescriptor.

Ця команда використовує синтаксис підстановки для лістингу тільки тих команд, імена яких містять вказане слово. Синтаксис підстановки використовує символ * для позначення будь-якого символу або серії символів. Тут ми

поставили * по обидва боки від SecurityDescriptor, щоб вказати, що перед ним або після нього може бути будь-який текст.

Лістинг 1.4. Використання *Get-Command* для перерахування команд

```
PS C:\Users\hladk> Get-Command -Name *SecurityDescriptor*

CommandType     Name                                               Version    Source
-----------     ----                                               -------    ------
Function        Add-NtSecurityDescriptorControl                    2.0.1      NtObjectManager
Function        Clear-NtSecurityDescriptorDacl                     2.0.1      NtObjectManager
Function        Clear-NtSecurityDescriptorSacl                     2.0.1      NtObjectManager
Function        ConvertFrom-NtSecurityDescriptor                   2.0.1      NtObjectManager
Function        Copy-NtSecurityDescriptor                          2.0.1      NtObjectManager
Function        Edit-NtSecurityDescriptor                          2.0.1      NtObjectManager
Function        Format-NtSecurityDescriptor                        2.0.1      NtObjectManager
Function        Format-Win32SecurityDescriptor                     2.0.1      NtObjectManager
Function        Get-NtSecurityDescriptor                           2.0.1      NtObjectManager
Function        Get-NtSecurityDescriptorControl                    2.0.1      NtObjectManager
Function        Get-NtSecurityDescriptorDacl                       2.0.1      NtObjectManager
Function        Get-NtSecurityDescriptorGroup                      2.0.1      NtObjectManager
Function        Get-NtSecurityDescriptorIntegrityLevel             2.0.1      NtObjectManager
Function        Get-NtSecurityDescriptorOwner                      2.0.1      NtObjectManager
Function        Get-NtSecurityDescriptorSacl                       2.0.1      NtObjectManager
Function        Get-Win32ServiceSecurityDescriptor                 2.0.1      NtObjectManager
Function        Remove-NtSecurityDescriptorControl                 2.0.1      NtObjectManager
Function        Remove-NtSecurityDescriptorDacl                    2.0.1      NtObjectManager
Function        Remove-NtSecurityDescriptorGroup                   2.0.1      NtObjectManager
Function        Remove-NtSecurityDescriptorIntegrityLevel          2.0.1      NtObjectManager
Function        Remove-NtSecurityDescriptorOwner                   2.0.1      NtObjectManager
Function        Remove-NtSecurityDescriptorSacl                    2.0.1      NtObjectManager
Function        Set-NtSecurityDescriptor                           2.0.1      NtObjectManager
Function        Set-NtSecurityDescriptorControl                    2.0.1      NtObjectManager
```

Ви також можете перелічити команди, доступні в модулі. Наприклад, лістинг 1.5 перелічить тільки команди, які експортуються модулем *NtObjectManager* і починаються з дієслова *Start*.

Лістинг 1.5. Використання *Get-Command* для перерахування команд у модулі *NtObjectManager*

```
PS C:\Users\hladk> Get-Command -Module NtObjectManager -Name Start-*

CommandType     Name                                               Version    Source
-----------     ----                                               -------    ------
Function        Start-AccessibleScheduledTask                      2.0.1      NtObjectManager
Function        Start-AppModelApplication                          2.0.1      NtObjectManager
Function        Start-FwNetEventListener                           2.0.1      NtObjectManager
Function        Start-NtFileOplock                                 2.0.1      NtObjectManager
Function        Start-Win32ChildProcess                            2.0.1      NtObjectManager
Function        Start-Win32DebugConsole                            2.0.1      NtObjectManager
Function        Start-Win32Service                                 2.0.1      NtObjectManager
Cmdlet          Start-NtDebugWait                                  2.0.1      NtObjectManager
Cmdlet          Start-NtWait                                       2.0.1      NtObjectManager
```

Знайшовши команду, яка виглядає перспективною, ви можете скористатися командою *Get-Help*, щоб перевірити її параметри та отримати приклади використання.

Лістинг 1.6. Відображення довідки для команди *Start-NtWait*

```
PS C:\WINDOWS\system32> Get-Help Start-NtWait

NAME
    Start-NtWait

SYNOPSIS
    Wait on one or more NT objects to become signaled.


SYNTAX
    Start-NtWait [-Object] <NtObject[]> [-Alertable <SwitchParameter>] [-Hour <int>] [-MilliSecond <long>] [-Minute <in
    t>] [-Second <int>] [-WaitAll <SwitchParameter>] [<CommonParameters>]

    Start-NtWait [-Object] <NtObject[]> [-Alertable <SwitchParameter>] [-Infinite <SwitchParameter>] [-WaitAll <SwitchP
    arameter>] [<CommonParameters>]


DESCRIPTION
    This cmdlet allows you to issue a wait on one or more NT objects until they become signaled. This is used for examp
    le to acquire a Mutant, decrement a Semaphore or wait for a Process to exit. The timeout value is a combination of
    all the allowed time parameters, e.g. if you specify 1 second and 1000 milliseconds it will wait 2 seconds in total
    . Specifying -Infinite overrides the time parameters and will wait indefinitely.


RELATED LINKS
    about_ManagingNtObjectLifetime

REMARKS
    To see the examples, type: "get-help Start-NtWait -examples".
    For more information, type: "get-help Start-NtWait -detailed".
    For technical information, type: "get-help Start-NtWait -full".
    For online help, type: "get-help Start-NtWait -online"
```

У лістингу 1.6 ми беремо команду Start-NtWait з лістингу 1.5 і передаємо її до Get-Help.

В лістингу 1.6 за замовчанням Get-Help виводить ім'я команди, короткий опис, синтаксис команди та детальніший опис.

У розділі синтаксису команди ви можете побачити кілька можливих режимів її роботи. У нашому випадку це або вказання часу в годинах, хвилинах, секундах та/або мілісекундах, або вказання Infinite для очікування без обмеження часу. Коли будь-яка частина синтаксису показана в дужках [], це означає, що вона є необов'язковою. Наприклад, єдиним обов'язковим параметром є Object, який приймає масив значень NtObject. Навіть ім'я цього параметра є необов'язковим, оскільки -Object знаходиться в дужках.

Ви можете отримати більше інформації про параметр, використовуючи команду *Parameter*. Лістинг 1-7 показує деталі для параметра *Object*.

Лістинг 1.7: Запит інформації про параметр об'єкта за допомогою команди *Parameter*

```
PS C:\WINDOWS\system32> 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
```

Ви можете використовувати синтаксис підстановки для вибору групи подібних імен параметрів. Наприклад, якщо ви вкажете Obj*, то отримаєте інформацію про всі параметри, імена яких починаються з префікса Obj.

Якщо ви хочете отримати приклади використання команди, використовуйте параметр *Examples*, як показано в лістингу 1.8.

Кожен приклад повинен містити один або два рядки фрагмента скрипта PowerShell та опис його функції. Ви також можете переглянути повну довідку щодо команди, вказавши параметр Full. Щоб переглянути цю інформацію в окремому діалоговому вікні, скористайтеся параметром ShowWindow. Наприклад, спробуйте виконати таку команду:

```
PS C:\Users\hladk> Get-Help Start-NtWait -ShowWindow
```

Останнє, що варто згадати про команди, — це можливість налаштовувати псевдоніми, або альтернативні імена для команд. Наприклад, ви можете використовувати псевдонім, щоб скоротити довжину команди для введення. PowerShell має багато попередньо визначених псевдонімів, а ви можете створити власні за допомогою команди New-Alias.

Лістинг 1.8. Демонстрація прикладів для *Start-NtWait*

```
PS C:\WINDOWS\system32> Get-Help Start-NtWait -Examples

NAME
    Start-NtWait

SYNOPSIS
    Wait on one or more NT objects to become signaled.


    ----------  EXAMPLE 1  ----------

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

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

    $ev = Get-NtEvent \BaseNamedObjects\ABC
    $ev | Start-NtWait -Infinite

    Get an event and wait indefinitely for it to be signaled.
    ----------  EXAMPLE 3  ----------

    $ev = Get-NtEvent \BaseNamedObjects\ABC
    $ev | Start-NtWait -Infinite -Alertable

    Get an event and wait indefinitely for it to be signaled or alerted.
    ----------  EXAMPLE 4  ----------

    $evs = @($ev1, $ev2)$
    Start-NtWait $evs -WaitAll -Second 100

    Get a list of events and wait 100 seconds for all events to be signaled.
```

Ви повинні побачити діалогове вікно, показане на рис. 1.1.

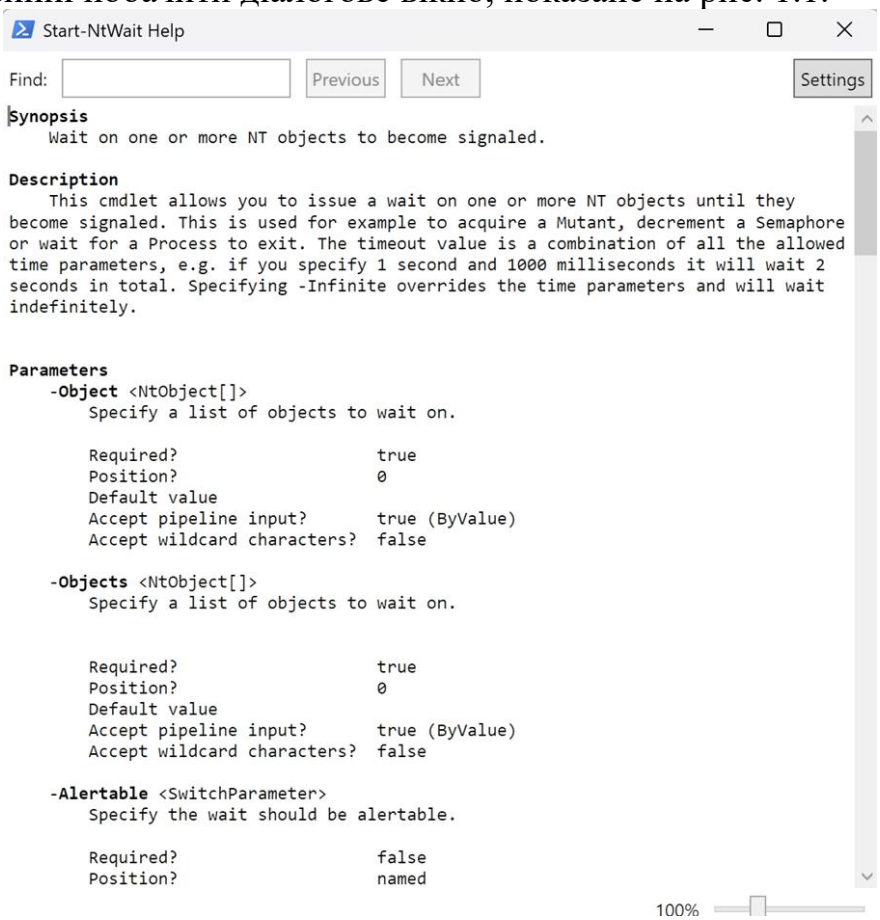


Рисунок 1.1. Діалогове вікно, що відображає інформацію *Get-Help* за допомогою параметра *ShowWindow*.

Наприклад, ми можемо встановити для команди *Start-NtWait* псевдонім *swt*, виконавши такі дії:

```
PS C:\Users\hladk> New-Alias -Name swt -Value Start-NtWait
```

Щоб відобразити список усіх певних псевдонімів, використовуйте команду *Get-Alias*.

Ми будемо уникати використання псевдонімів без необхідності протягом усього посібника, оскільки це може зробити скрипти більш заплутаними, якщо ви не знаєте, що означає той чи інший псевдонім.

1.4. ВИЗНАЧЕННЯ ФУНКЦІЙ

Як і в усіх мовах програмування, в PowerShell доцільно зменшувати складність. Одним із способів зменшення складності є об'єднання загального коду в функцію. Після визначення функції скрипт PowerShell може викликати цю функцію, замість того щоб повторювати один і той самий код у кількох місцях. Основний синтаксис функцій у PowerShell є простим; приклад наведено в лістингу 1.9.

Лістинг 1.9. Визначення простої функції PowerShell з ім'ям *Get- NameValue*

```
PS C:\Users\hladk> function Get-NameValue { param( [string]$Name = "", $Value )
>> return "We've got $Name with value $Value" }
PS C:\Users\hladk> Get-NameValue -Name "Hello" -Value "World"
We've got Hello with value World
PS C:\Users\hladk> Get-NameValue "Goodbye" 12345
We've got Goodbye with value 12345
```

Синтаксис для визначення функції починається з ключового слова *function*, за яким слідує назва функції, яку ви хочете визначити. Хоча використання стандартної конвенції іменування команд PowerShell, що складається з дієслова, за яким слідує іменник, не є обов'язковим, це варто робити, оскільки це дає користувачеві чітке уявлення про те, що робить ваша функція.

Далі ви визначаєте іменовані параметри функції. Як і змінні, параметри визначаються за допомогою імені з префіксом *$*, як показано в лістингу 1.9. Ви можете вказати тип у дужках, але це необов'язково. У цьому прикладі $*Name* є рядком, але параметр *$Value* може приймати будь-яке значення від виклику. Вказувати іменовані параметри не обов'язково. Якщо блок *param* не включений, то всі передані аргументи розміщуються в масиві *$args*. Перший параметр знаходиться в *$args[0]*, другий в *$args[1]* і так далі.

Тіло функції *Get-NameValue* приймає параметри і будує рядок за допомогою інтерполяції рядків. Функція повертає рядок за допомогою ключового слова *return*, яке також негайно завершує роботу функції. У цьому випадку можна опустити ключове слово *return*, оскільки PowerShell поверне всі значення, які не були захоплені в змінні.

Після визначення функції ми її викликаємо. Ви можете явно вказати імена параметрів. Однак, якщо виклик однозначний, вказувати імена параметрів не обов'язково. Лістинг 1.9 показує обидва підходи.

Якщо є потреба запустити невеликий блок коду без визначення функції, ви можете створити блок скрипту. Блок скрипта - це один або кілька операторів, вкладених у фігурні дужки **{.}**. Цей блок можна позначити змінною та виконати

за необхідності за допомогою команди *Invoke-Command* або оператора *&*, як показано у лістингу 1.10.

Лістинг 1.10. Створення блоку скрипту та його виконання

```
PS C:\Users\hladk> $script = { Write-Output "Hello" }
PS C:\Users\hladk> & $script
Hello
```

1.5. ВІДОБРАЖЕННЯ ТА КЕРУВАННЯ ОБ'ЄКТАМИ

Якщо ви виконуєте команду і не зберігаєте результати в змінній, результати передаються до консолі PowerShell. Консоль використовує форматувальник для відображення результатів у вигляді таблиці або списку (формат вибирається автоматично залежно від типів об'єктів, що містяться в результатах).

Також можна вказати власний форматувальник. Наприклад, якщо ви використовуєте вбудовану команду *Get-Process* для отримання списку запущених процесів, PowerShell використовує власний форматувальник для відображення записів у вигляді таблиці, як показано в лістингу 1.11.

Лістинг 1.11. Виведення списку процесів у вигляді таблиці

```
PS C:\Users\hladk> Get-Process

Handles  NPM(K)    PM(K)      WS(K)     CPU(s)     Id  SI ProcessName
-------  ------    -----      -----     ------     --  -- -----------
    741     118    15488      13152               1132   0 afwServ
    196      12     5244       9952               9696   0 AggregatorHost
    427      23    57240      46560       0.25   15644   4 ApplicationFrameHost
    977      47   210316     118612               3624   0 aswEngSrv
   1079      37    73452      63716               3492   0 aswidsagent
   1688      53   125144      56064               4160   0 aswToolsSvc
   6389     170   278660     257516               3376   0 AvastSvc
    581      29    15600      38084       0.08    3004   4 AvastUI
   2319      62    45040      62172     275.63    3732   4 AvastUI
    771      34    22812      46892       0.23   10144   4 AvastUI
    638      32    16536      42364       0.27   13456   4 AvastUI
     76       7     1092       1120               9124   0 conhost
    137      10     1640       9744       0.03   20772   4 conhost
    862      30     2700       3296               1144   0 csrss
    744      25     3392       8180              12236   4 csrss
    536      23     9988      36712       3.98    5016   4 ctfmon
    317      16     9528       4140               6940   0 dasHost
    483      18     6424       6328               5744   0 DAX3API
```

Лістинг 1.12. Вибір лише властивостей *Id* та *ProcessName*

```
PS C:\Users\hladk> Get-Process | Select-Object Id, ProcessName

   Id ProcessName
   -- -----------
 1132 afwServ
 9696 AggregatorHost
15644 ApplicationFrameHost
 3624 aswEngSrv
 3492 aswidsagent
 4160 aswToolsSvc
 3376 AvastSvc
 3004 AvastUI
 3732 AvastUI
10144 AvastUI
13456 AvastUI
 8036 backgroundTaskHost
15368 backgroundTaskHost
 9124 conhost
18880 conhost
 1144 csrss
12236 csrss
 5016 ctfmon
 6940 dasHost
 5744 DAX3API
 6288 dllhost
21512 dllhost
 5664 DSAService
 5680 DSAUpdateService
```

Якщо є потреба зменшити кількість стовпців у виведенні результату, ви можете використовувати команду Select-Object, щоб вибрати лише потрібні властивості. Наприклад, Лістинг 1.12 вибирає властивості Id та ProcessName .

Можна змінити поведінку за замовчуванням, використовуючи команду *Format-Table* або *Format-List,* яка примусово відформатує таблицю або список відповідно. Наприклад, у лістингу 1.13 показано, як використовувати команду *Format-List* для зміни виведення до списку.

Щоб знайти імена доступних властивостей, можна використовувати команду Get-Member на одному з об'єктів, що повертаються Get-Process.

Лістинг 1.13. Використання *Format-List* для відображення процесів у вигляді списку

```
PS C:\Users\hladk> Get-Process | Format-List


Id      : 1132
Handles : 741
CPU     :
SI      : 0
Name    : afwServ

Id      : 9696
Handles : 196
CPU     :
SI      : 0
Name    : AggregatorHost

Id      : 15644
Handles : 427
CPU     : 0.25
SI      : 4
Name    : ApplicationFrameHost

Id      : 3624
Handles : 977
CPU     :
SI      : 0
Name    : aswEngSrv
```

Наприклад, у лістингу 1.14 перелічені властивості об'єкта Process.

Лістинг 1.14. Використання команди *Get-Member* для отримання списку властивостей об'єкта *Process*

```
PS C:\Users\hladk> 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;}
Handle                     Property   System.IntPtr Handle {get;}
HandleCount                Property   int HandleCount {get;}
HasExited                  Property   bool HasExited {get;}
Id                         Property   int Id {get;}
MachineName                Property   string MachineName {get;}
MainModule                 Property   System.Diagnostics.ProcessModule MainModule {get;}
MainWindowHandle           Property   System.IntPtr MainWindowHandle {get;}
MainWindowTitle            Property   string MainWindowTitle {get;}
MaxWorkingSet              Property   System.IntPtr MaxWorkingSet {get;set;}
MinWorkingSet              Property   System.IntPtr MinWorkingSet {get;set;}
Modules                    Property   System.Diagnostics.ProcessModuleCollection Modules {get;}
NonpagedSystemMemorySize   Property   int NonpagedSystemMemorySize {get;}
NonpagedSystemMemorySize64 Property   long NonpagedSystemMemorySize64 {get;}
PagedMemorySize            Property   int PagedMemorySize {get;}
PagedMemorySize64          Property   long PagedMemorySize64 {get;}
PagedSystemMemorySize      Property   int PagedSystemMemorySize {get;}
```

Ви могли помітити, що є інші властивості, не включені до висновку. Щоб відобразити їх, необхідно перевизначити форматування користувача.

Найпростіший спосіб отримати доступ до прихованих властивостей - використовувати *Select-Object* для явного отримання значень або вказати

властивості для відображення в команді *Format-Table* або *Format-List*. Як правило, використовують, щоб відобразити всі властивості, як у лістингу 1.15.

Лістинг 1.15. Відображення всіх властивостей об'єкта *Process* у списку

```
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--
```

Багато об'єктів також мають методи, які можна викликати для виконання деяких дій над ним. Лістинг 1.16 показує, яким чином використовується *Get-Member* для запиту методів

Лістинг 1.16. Відображення методів об'єкта *Process*

```
PS C:\Users\hladk> 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()
CloseMainWindow           Method     bool CloseMainWindow()
CreateObjRef              Method     System.Runtime.Remoting.ObjRef CreateObjRef(type requestedType)
Dispose                   Method     void Dispose(), void IDisposable.Dispose()
Equals                    Method     bool Equals(System.Object obj)
GetHashCode               Method     int GetHashCode()
GetLifetimeService        Method     System.Object GetLifetimeService()
GetType                   Method     type GetType()
InitializeLifetimeService Method     System.Object InitializeLifetimeService()
Kill                      Method     void Kill()
Refresh                   Method     void Refresh()
Start                     Method     bool Start()
ToString                  Method     string ToString()
WaitForExit               Method     bool WaitForExit(int milliseconds), void WaitForExit()
WaitForInputIdle          Method     bool WaitForInputIdle(int milliseconds), bool WaitForInputIdle()
```

Якщо вихідні дані команди занадто великі, щоб вміститися на екрані, можна розділити їх на сторінки, щоб відображалася тільки перша частина, а консоль чекала натискання клавіші, перш ніж відображати решту.

Лістинг 1.17. Виведення сторінок за допомогою *Out-Host*

```
PS C:\Users\hladk> Get-Process | Out-Host -Paging

Handles  NPM(K)    PM(K)      WS(K)     CPU(s)      Id  SI ProcessName
-------  ------    -----      -----     ------      --  -- -----------
    737     117    15424      13056               1132   0 afwServ
    196      12     5368       8796               9696   0 AggregatorHost
    427      23    57240      43216       0.25   15644   4 ApplicationFrameHost
    977      47   211068      98440               3624   0 aswEngSrv
   1094      37    73452      61108               3492   0 aswidsagent
   1681      53   125060      43784               4160   0 aswToolsSvc
   6470     170   279564     258896               3376   0 AvastSvc
    581      29    15628      35996       0.08    3004   4 AvastUI
   2335      63    43776      69188     289.89    3732   4 AvastUI
    771      34    22844      45992       0.23   10144   4 AvastUI
    638      32    16568      41460       0.28   13456   4 AvastUI
    141      10     1640       9792       0.02    5396   4 conhost
     76       7     1092       1120               9124   0 conhost
    902      31     2696       3208               1144   0 csrss
    746      25     3392       8136              12236   4 csrss
    536      23    10528      36804       5.77    5016   4 ctfmon
    317      16     9528       4140               6940   0 dasHost
    483      18     6424       6132               5744   0 DAX3API
    300      25     6600      17900       0.34    6288   4 dllhost
    125       9     1532       8780       0.02   21512   4 dllhost
   1185      73   106128      33716               5664   0 DSAService
    626      31    35408      14460               5680   0 DSAUpdateService
   2317      48   218508     139228              24872   4 dwm
    221      18     3476      13904       0.31   20120   4 EnduranceGamingProcess
    377      19     7544      27872      37.20    8764   4 EoAExperiences
   4665     133   320924     343496      93.11   26020   4 explorer
<ПРОБЕЛ> следующая страница; <CR> следующая строка; Q выход
    867      82   285668     167328      80.59   22364   4 FineReader
<ПРОБЕЛ> следующая страница; <CR> следующая строка; Q выход
```

Можна увімкнути розділення на сторінки, перенаправивши вихідні дані на команду Out-Host і вказавши параметр Paging або за допомогою команди more. Приклад наведено в лістингу 1.17.

Ви можете писати безпосередньо у вікно консолі, використовуючи команду Write-Host у своїх власних скриптах. Це дозволяє змінювати кольори виводу на свій розсуд, використовуючи параметри ForegroundColor та BackgroundColor.

Також це має перевагу у тому, що за замовчуванням об'єкти не вставляються у конвеєр, як показано нижче:

```
PS C:\Users\hladk> $output = Write-Host "Hello"
Hello
```

Це означає, що за замовчуванням ви не можете перенаправити вихідні дані у файл або в конвеєр. Однак ви можете перенаправити вихідні дані хоста, перенаправивши його потік у стандартний потік вихідних даних за допомогою такої команди:

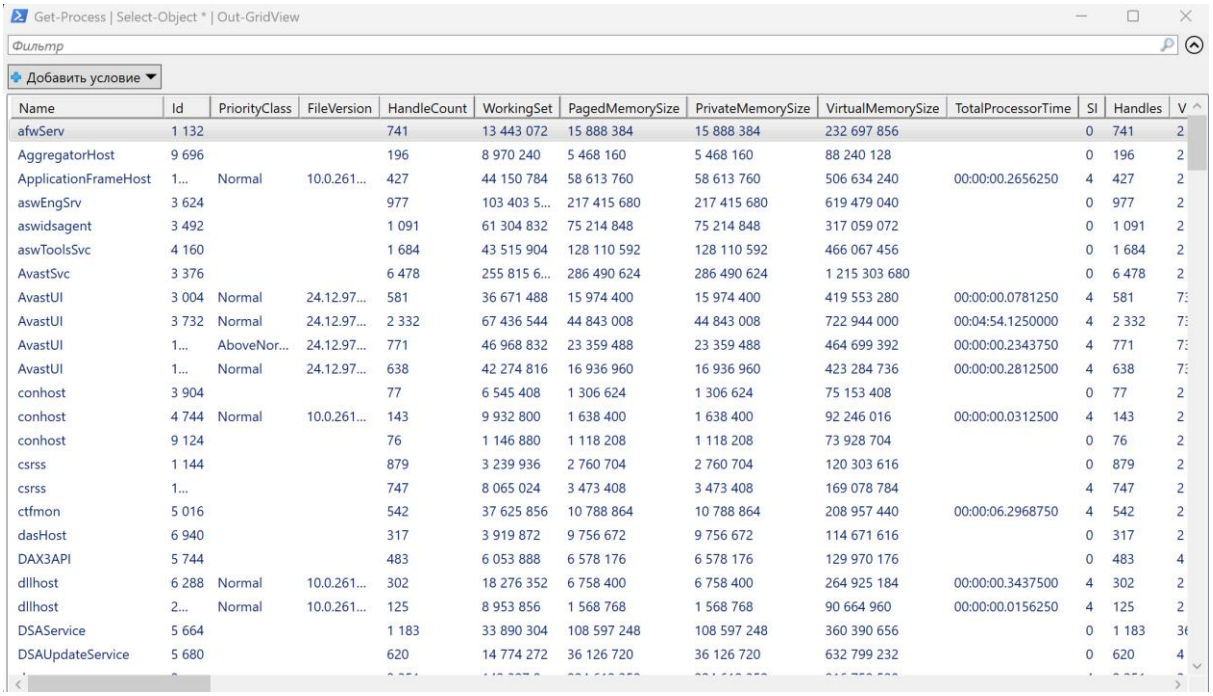
```
PS C:\Users\hladk> $output = Write-Host "Hello" 6>&1
PS C:\Users\hladk> $output
Hello
```

PowerShell також підтримує базовий графічний інтерфейс для відображення таблиць об'єктів.

Щоб отримати до нього доступ, скористайтеся командою *Out-GridView*. Зверніть увагу, що власне форматування все одно обмежуватиме колонки, які відображає PowerShell. Якщо ви хочете переглянути інші колонки, скористайтеся командою *Select-Object* у конвеєрі, щоб вибрати властивості. У наведеному нижче прикладі відображаються всі властивості в графічному інтерфейсі Grid View:

```
PS C:\Users\hladk> Get-Process | Select-Object * | Out-GridView
```

Виконання цієї команди повинно відкрити діалогове вікно, як показано на рис. 1.2.



Name	Id	PriorityClass	FileVersion	HandleCount	WorkingSet	PagedMemorySize	PrivateMemorySize	VirtualMemorySize	TotalProcessorTime	SI	Handles	V
afwServ	1 132			741	13 443 072	15 888 384	15 888 384	232 697 856		0	741	2
AggregatorHost	9 696			196	8 970 240	5 468 160	5 468 160	88 240 128		0	196	2
ApplicationFrameHost	1...	Normal	10.0.261...	427	44 150 784	58 613 760	58 613 760	506 634 240	00:00:00.2656250	4	427	2
aswEngSrv	3 624			977	103 403 5...	217 415 680	217 415 680	619 479 040		0	977	2
aswidsagent	3 492			1 091	61 304 832	75 214 848	75 214 848	317 059 072		0	1 091	2
aswToolsSvc	4 160			1 684	43 515 904	128 110 592	128 110 592	466 067 456		0	1 684	2
AvastSvc	3 376			6 478	255 815 6...	286 490 624	286 490 624	1 215 303 680		0	6 478	2
AvastUI	3 004	Normal	24.12.97...	581	36 671 488	15 974 400	15 974 400	419 553 280	00:00:00.0781250	4	581	7
AvastUI	3 732	Normal	24.12.97...	2 332	67 436 544	44 843 008	44 843 008	722 944 000	00:04:54.1250000	4	2 332	7
AvastUI	1...	AboveNor...	24.12.97...	771	46 968 832	23 359 488	23 359 488	464 699 392	00:00:00.2343750	4	771	7
AvastUI	1...	Normal	24.12.97...	638	42 274 816	16 936 960	16 936 960	423 284 736	00:00:00.2812500	4	638	7
conhost	3 904			77	6 545 408	1 306 624	1 306 624	75 153 408		0	77	2
conhost	4 744	Normal	10.0.261...	143	9 932 800	1 638 400	1 638 400	92 246 016	00:00:00.0312500	4	143	2
conhost	9 124			76	1 146 880	1 118 208	1 118 208	73 928 704		0	76	2
csrss	1 144			879	3 239 936	2 760 704	2 760 704	120 303 616		0	879	2
csrss	1...			747	8 065 024	3 473 408	3 473 408	169 078 784		4	747	2
ctfmon	5 016			542	37 625 856	10 788 864	10 788 864	208 957 440	00:00:06.2968750	4	542	2
dasHost	6 940			317	3 919 872	9 756 672	9 756 672	114 671 616		0	317	2
DAX3API	5 744			483	6 053 888	6 578 176	6 578 176	129 970 176		0	483	4
dllhost	6 288	Normal	10.0.261...	302	18 276 352	6 758 400	6 758 400	264 925 184	00:00:00.3437500	4	302	2
dllhost	2...	Normal	10.0.261...	125	8 953 856	1 568 768	1 568 768	90 664 960	00:00:00.0156250	4	125	2
DSAService	5 664			1 183	33 890 304	108 597 248	108 597 248	360 390 656		0	1 183	3
DSAUpdateService	5 680			620	14 774 272	36 126 720	36 126 720	632 799 232		0	620	4

Рисунок 1.2. Відображення об'єктів процесу у вигляді сітки

Ви можете фільтрувати та обробляти дані в графічному інтерфейсі *Grid View*. Спробуйте попрацювати з елементами керування. Ви також можете вказати параметр *PassThru* для *Out-GridView*, що змусить команду чекати, поки ви натиснете кнопку OK у графічному інтерфейсі.

Усі рядки в перегляді, які будуть вибрані на момент натискання кнопки OK, будуть записані в командний конвеєр.

1.6. ФІЛЬТРУВАННЯ, СОРТУВАННЯ ТА ГРУПУВАННЯ ОБ'ЄКТІВ

Традиційна оболонка передає необроблений текст між командами. PowerShell передає об'єкти. Передача об'єктів дозволяє отримати доступ до окремих властивостей об'єктів і легко фільтрувати конвеєр. Ви навіть можете легко впорядковувати та групувати об'єкти.

Ви можете фільтрувати об'єкти за допомогою команди *Where-Object*, яка має псевдоніми *Where* і *?*. Найпростіший фільтр - це перевірка значення параметра, як показано в лістингу 1.18, де ми фільтруємо вихідні дані вбудованої команди *Get-Process*, щоб знайти процес *explorer*.

Лістинг 1.18. Фільтрування списку процесів за допомогою *Where-Object*

```
PS C:\Users\hladk> Get-Process | Where-Object ProcessName -EQ "explorer"

Handles  NPM(K)    PM(K)      WS(K)     CPU(s)     Id  SI ProcessName
-------  ------    -----      -----     ------     --  -- -----------
   4642     133   292288     312500     104.70  26020   4 explorer
```

У лістингу 1.18 ми відбираємо тільки об'єкти *Process*, у яких Process *Name* дорівнює (-EQ) «explorer». Існує безліч операторів, які можна використовувати для фільтрації, деякі з них наведені в таб.1.4.

Таблиця 1.4.

Загальні оператори для *Where-Object*

Оператор	Приклад	Опис
-EQ	ProcessName -EQ "explorer"	Дорівнює значенню
-NE	ProcessName -NE "explorer"	Не дорівнює значенню
-Match	ProcessName -Match "ex.*"	Зіставляє рядок із регулярним виразом
- NotMatch	ProcessName - NotMatch "ex.*"	Зворотній оператор -*Match*
-Like	ProcessName -Like "ex*"	Відповідає рядку символу підстановки
- NotLike	ProcessName - NotLike "ex*"	Оператор, зворотний оператору - *Like*
-GT	ProcessName -GT "ex"	Більше, ніж порівняння
-LT	ProcessName -LT "ex"	Менше порівняння

Ви можете переглянути всі підтримувані оператори, скориставшись командою *Get-Help* у команді *Where-Object*.

Якщо умова для фільтрування є складнішою, ніж просте порівняння, ви можете скористатися блоком сценарію. Блок сценарію повинен повертати значення *True*, щоб залишити об'єкт у конвеєрі, або *False*, щоб відфільтрувати його.

Наприклад, ви також можете написати лістинг 1-18 наступним чином:

```
PS C:\Users\hladk> Get-Process | Where-Object { $_.ProcessName -eq "explorer" }

Handles  NPM(K)    PM(K)      WS(K)     CPU(s)      Id  SI ProcessName
-------  ------    -----      -----     ------      --  -- -----------
   4719     134   326960     349168     110.66   26020   4 explorer
```

Змінна *$_*, що передається до блоку скрипту, представляє поточний об'єкт у конвеєрі. Використовуючи блок скрипту, ви можете отримати доступ до всієї мови у вашому фільтруванні, включаючи виклик функцій.

Для впорядкування об'єктів використовуйте команду *Sort-Object*. Якщо об'єкти можна впорядкувати, як у випадку з рядками або числами, то вам просто потрібно передати об'єкти в команду.

В іншому випадку вам потрібно буде вказати властивість, за якою буде здійснюватися сортування. Наприклад, ви можете впорядкувати список процесів за кількістю їхніх дескрипторів, представлених властивістю *Handles*, як показано в лістингу 1.19.

Лістинг 1.19. Сортування процесів за кількістю дескрипторів

```
PS C:\Users\hladk> Get-Process | Sort-Object Handles

Handles  NPM(K)    PM(K)      WS(K)     CPU(s)      Id  SI ProcessName
-------  ------    -----      -----     ------      --  -- -----------
      0      24    13028      65536                280   0 Registry
      0       0      172      63056                236   0 Secure System
      0       0       60          8                  0   0 Idle
      0       0     2480     661404               3296   0 Memory Compression
     42      12     5196      10244              25260   4 fontdrvhost
     42       7     1632        884               1488   0 fontdrvhost
     49       6     1356        680               4948   0 NgcIso
     58      16     1852       1060               1312   0 LsaIso
     58       4     1152        664                784   0 smss
     64       6      844       1168               5752   0 FpcSessionService
     76       7     1092       1120               9124   0 conhost
    105      12     1456       1824               2020   0 svchost
    106       8     1404       7268       0.02   23196   4 svchost
    108       8     1552       1652              21600   0 svchost
    115       8     1432       1632               4596   0 svchost
    115       9     2428       3404              15616   0 svchost
    117       9     1660       1724               2236   0 svchost
    119       8     1484       3820              11180   0 svchost
    125      10     1916       8340       6.77   15520   4 ipf_helper
    125       9     1844       1796              24048   0 svchost
    125       9     1532       8700       0.02   21512   4 dllhost
    125       8     1944       4504               3680   0 svchost
    140      11     1536       2064               5768   0 svchost
    141      13     1808       2140               5696   0 ipfsvc
    141      11     2040      10640       0.03    2144   4 UserOOBEBroker
    143      10     1496       1656               5812   0 jhi_service
```

Щоб виконати сортування за спаданням замість зростання, використовуйте параметр *Descending*, як показано у лістингу 1.20.

Лістинг 1.20. Сортування процесів за кількістю дескрипторів у порядку зменшення

```
PS C:\Users\hladk> Get-Process | Sort-Object Handles -Descending

Handles  NPM(K)    PM(K)      WS(K)     CPU(s)      Id  SI ProcessName
-------  ------    -----      -----     ------      --  -- -----------
  35846       0       56        132                  4   0 System
   6468     170   280220     236572               3376   0 AvastSvc
   4669     133   325752     345456     116.53   26020   4 explorer
   3318      22     7776      16408               8200   0 svchost
   2472      49   257900     178664              24872   4 dwm
   2338      63    43980      64680     305.83    3732   4 AvastUI
   2187      80   441316     389256     335.95    9344   4 WINWORD
   2021      72   153048     176728      98.48   20084   4 opera
   1899     138   417764     182876      40.84    8520   4 Viber
   1857      38    15572      26900               1332   0 lsass
   1741      53   464040     322596      46.58   18224   4 SnippingTool
   1714      30    39820      47844               1456   0 svchost
   1687      53   125116      40896               4160   0 aswToolsSvc
   1529     112   169648     250924       7.63   25520   4 SearchHost
   1468      70   139060       2708       2.13   12224   4 SystemSettings
   1416      20    12936      16616               1588   0 svchost
```

На цьому етапі також можна відфільтрувати дублікати записів, вказавши параметр *Unique* для *Sort-Object*.

Нарешті, ви можете згрупувати об'єкти на основі імені властивості за допомогою команди *Group-Object*. Лістинг 1.21 показує, що ця команда повертає список об'єктів, кожен з яких має властивості *Count*, *Name* і *Group*.

Лістинг 1.21.Групування об'єктів *Process* за *ProcessName*

```
PS C:\Users\hladk> Get-Process | Group-Object ProcessName

Count Name                      Group
----- ----                      -----
    1 afwServ                   {System.Diagnostics.Process (afwServ)}
    1 AggregatorHost            {System.Diagnostics.Process (AggregatorHost)}
    1 ApplicationFrameHost      {System.Diagnostics.Process (ApplicationFrameHost)}
    1 aswEngSrv                 {System.Diagnostics.Process (aswEngSrv)}
    1 aswidsagent               {System.Diagnostics.Process (aswidsagent)}
    1 aswToolsSvc               {System.Diagnostics.Process (aswToolsSvc)}
    1 AvastSvc                  {System.Diagnostics.Process (AvastSvc)}
    4 AvastUI                   {System.Diagnostics.Process (AvastUI), System.Diagnostics.Process (AvastUI), System....
    2 conhost                   {System.Diagnostics.Process (conhost), System.Diagnostics.Process (conhost)}
    2 csrss                     {System.Diagnostics.Process (csrss), System.Diagnostics.Process (csrss)}
    1 ctfmon                    {System.Diagnostics.Process (ctfmon)}
    1 dasHost                   {System.Diagnostics.Process (dasHost)}
    1 DAX3API                   {System.Diagnostics.Process (DAX3API)}
    2 dllhost                   {System.Diagnostics.Process (dllhost), System.Diagnostics.Process (dllhost)}
    1 DSAService                {System.Diagnostics.Process (DSAService)}
    1 DSAUpdateService          {System.Diagnostics.Process (DSAUpdateService)}
    1 dwm                       {System.Diagnostics.Process (dwm)}
    1 EnduranceGamingProcess    {System.Diagnostics.Process (EnduranceGamingProcess)}
    1 EoAExperiences            {System.Diagnostics.Process (EoAExperiences)}
    1 explorer                  {System.Diagnostics.Process (explorer)}
```

Як інший варіант, ви можете використовувати всі ці команди разом в одному конвеєрі, як показано в лістингу 1.22.

Лістинг 1.22. Об'єднання *Where-Object*, *Group-Object* та *Sort -Object*

```
PS C:\Users\hladk> Get-Process | Group-Object ProcessName | Where-Object Count -GT 10 | Sort-Object Count

Count Name                      Group
----- ----                      -----
   21 opera                     {System.Diagnostics.Process (opera), System.Diagnostics.Process (opera), System.Diag...
   87 svchost                   {System.Diagnostics.Process (svchost), System.Diagnostics.Process (svchost), System....
```

1.7. ЕКСПОРТ ДАНИХ

Отримавши ідеальний набір об'єктів, які ви хочете перевірити, ви можете зберегти цю інформацію у файлі на диску. PowerShell надає для цього численні опції, деякі з яких розглянемо тут.

Лістинг 1.23. Запис вмісту в текстовий файл та його повторне читання

```
PS C:\Users\hladk> Get-Process | Out-File processes.txt
PS C:\Users\hladk> Get-Content processes.txt

Handles  NPM(K)    PM(K)      WS(K)     CPU(s)      Id  SI ProcessName
-------  ------    -----      -----     ------      --  -- -----------
    739     117    15456      12940               1132   0 afwServ
    194      12     4220       8680               9696   0 AggregatorHost
    427      23    57240      43016       0.30   15644   4 ApplicationFrameHost
    977      48   221156     107544               3624   0 aswEngSrv
   1073      37    77564      61740               3492   0 aswidsagent
   1678      53   125064      40296               4160   0 aswToolsSvc
    422      22   103388     107560      21.08   15876   0 audiodg
   6334     172   282460     227040               3376   0 AvastSvc
    581      29    15600      35924       0.09    3004   4 AvastUI
   2335      63    47388      69312     314.25    3732   4 AvastUI
    771      34    22828      46524       0.31   10144   4 AvastUI
    642      32    16564      42108       0.30   13456   4 AvastUI
     76       7     1092       1120               9124   0 conhost
    144      10     1732       9768       0.02    9928   4 conhost
    853      30     2696       3028               1144   0 csrss
    764      25     3392       8096              12236   4 csrss
    550      23    10680      36688       8.53    5016   4 ctfmon
    317      16     9528       3500               6940   0 dasHost
    483      18     6624       7572               5744   0 DAX3API
    310      26     6760      17700       0.34    6288   4 dllhost
```

Перший варіант – вивести об'єкти у файл у вигляді тексту за допомогою Out-File. Ця команда захоплює відформатований текстовий результат і записує його у файл. Ви можете використовувати Get-Content, щоб знову прочитати файл, як показано в лістингу 1.23.

Ви також можете використовувати оператор «більше ніж» для відправлення вихідних даних у файл, як і в інших оболонках. Наприклад:

```
PS C:\Users\hladk> Get-Process > processes.txt
```

Якщо вам потрібний більш структурований формат, ви можете скористатися командою *Export-Csv* для перетворення об'єкта в табличний формат із розділенням значень комами (CSV).

Далі ви можете імпортувати цей файл у програму для роботи з електронними таблицями щоб проаналізувати його в автономному режимі.

У прикладі в лістингу 1.24 вибираються деякі властивості об'єкта *Process* і експортуються в CSV-файл processes.csv.

Лістинг 1.24 Експорт об'єктів у файл CSV

```
PS C:\Users\hladk> Get-Process | Select-Object Id, ProcessName | Export-Csv processes.csv -NoTypeInformation
PS C:\Users\hladk> Get-Content processes.csv
"Id","ProcessName"
"1132","afwServ"
"9696","AggregatorHost"
"15644","ApplicationFrameHost"
"3624","aswEngSrv"
"3492","aswidsagent"
"4160","aswToolsSvc"
"15876","audiodg"
"3376","AvastSvc"
"3004","AvastUI"
"3732","AvastUI"
"10144","AvastUI"
"13456","AvastUI"
"9124","conhost"
"9928","conhost"
"1144","csrss"
"12236","csrss"
"5016","ctfmon"
```

За допомогою команди *Import-Csv* можливо повторно імпортувати дані CSV.

Однак, якщо ви плануєте експортувати дані, а потім імпортувати їх пізніше, вам, ймовірно, більше сподобається формат CLI XML.

Цей формат може включати структуру і тип вихідного об'єкта, що дозволяє відновити його під час імпорту даних.

Лістинг 1.25 показує, як можна використовувати команди *Export-CliXml* та *Import-CliXml* для експорту об'єктів у цьому форматі, а потім імпортувати їх.

Лістинг 1.25. Експорт та повторний імпорт XML-файлів CLI

```
PS C:\Users\hladk> Get-Process | Select-Object Id, ProcessName | Export-CliXml processes.xml
PS C:\Users\hladk> Get-Content processes.xml
<Objs Version="1.1.0.1" xmlns="http://schemas.microsoft.com/powershell/2004/04">
  <Obj RefId="0">
    <TN RefId="0">
      <T>Selected.System.Diagnostics.Process</T>
      <T>System.Management.Automation.PSCustomObject</T>
      <T>System.Object</T>
    </TN>
    <MS>
      <I32 N="Id">1132</I32>
      <S N="ProcessName">afwServ</S>
    </MS>
  </Obj>
  <Obj RefId="1">
    <TNRef RefId="0" />
    <MS>
      <I32 N="Id">9696</I32>
      <S N="ProcessName">AggregatorHost</S>
    </MS>
  </Obj>
  <Obj RefId="2">
    <TNRef RefId="0" />
    <MS>
      <I32 N="Id">15644</I32>
```

1.8. ВИСНОВКИ ДО РОЗДІЛУ 1

У цьому розділі наведено короткий огляд того, як Налаштування середовища PowerShell, щоб виконати приклади коду, наведені в цій книзі. Ми обговорили налаштування PowerShell для виконання скриптів та встановлення необхідного зовнішнього модуля PowerShell.

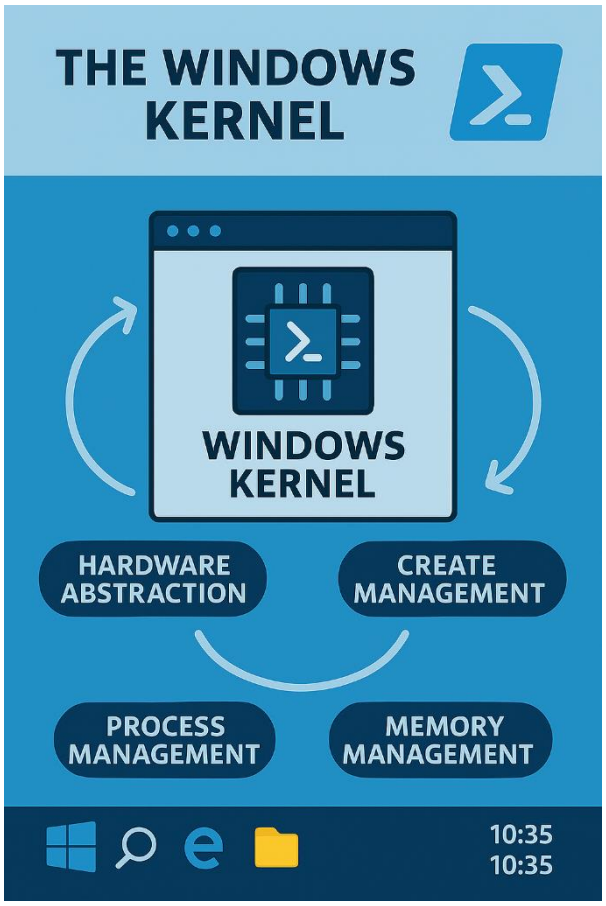
У подальшій частині розділу наведено коротку інформацію про мову PowerShell. Це включало основи синтаксису PowerShell, а також пошук команд за допомогою Get-Command, отримання довідки за допомогою Get-Help, а також відображення, фільтрування, групування та експорт об'єктів PowerShell.

Вивчивши основи PowerShell, ми можемо перейти до внутрішньої роботи операційної системи Windows. У наступному розділі ми обговоримо ядро Windows і те, як ви можете взаємодіяти з ним за допомогою PowerShell.

ПИТАННЯ ДЛЯ САМОКОНТРОЛЮ

1. Доведіть необхідність та доцільність захисту операційної системи Windows.
2. Що дозволяє політика *RemoteSigned*, якою командою ця політика встановлюється?
3. Яка команда дозволяє встановити додаткові модулі з репозиторію PowerShell?
4. Наведіть приклад базового типу *hashtable*.
5. Наведіть приклад використання оператора *-eq*.
6. Що містить змінна *$pwd*?
7. За допомогою якої команди можна перерахувати всі змінні?
8. Наведіть приклад інтерполяції рядків.
9. Яка команда дозволяє отримати допомогу?
10. Якщо необхідно отримати приклади використання команди, який параметр необхідно використовувати?
11. Найпростіший спосіб отримати доступ до прихованих властивостей об'єкта – використовувати команду ...
12. Як можна використовувати команди Export-CliXml та Import-CliXml?

РОЗДІЛ 2. ЯДРО ОПЕРАЦІЙНОЇ СИСТЕМИ WINDOWS



Windows — це безпечна багатофункціональна операційна система. Однак це також одна із найскладніших сучасних операційних систем для детального розуміння.

Розглянемо дві частини операційної системи: ядро і додатки в режимі користувача. Ядро приймає рішення щодо безпеки. Ці рішення визначають, що може робити в системі користувач. Однак більшість програм, які ви запускаєте на машині з операційною системою Windows, працюють в режимі користувача. Відповідно, цей розділ буде посвячений ***ядру***, а наступний - додаткам режиму ***користувача***.

Познайомимось із підсистемами, що входять до складу ядра Windows. Для кожної підсистеми пояснимо її призначення та спосіб використання. Почнемо з об'єктів диспетчера, які передбачають системні виклики, що дозволяють додатку режиму користувача отримувати доступ до об'єктів ядра.

2.1. ВИКОНАВЧИЙ МОДУЛЬ ЯДРА WINDOWS

Повний модуль ядра Windows NTOS, або скорочене ядро, є серцем Windows. Він забезпечує всі привілейовані функції операційної системи, а також інтерфейси, через які додатки користувача можуть взаємодіяти з обладнанням. Ядро розділено на кілька підсистем, кожна з яких має свою визначену ціль. На рис. 2.1 показана схема компонентів, які є найбільш цікаві.

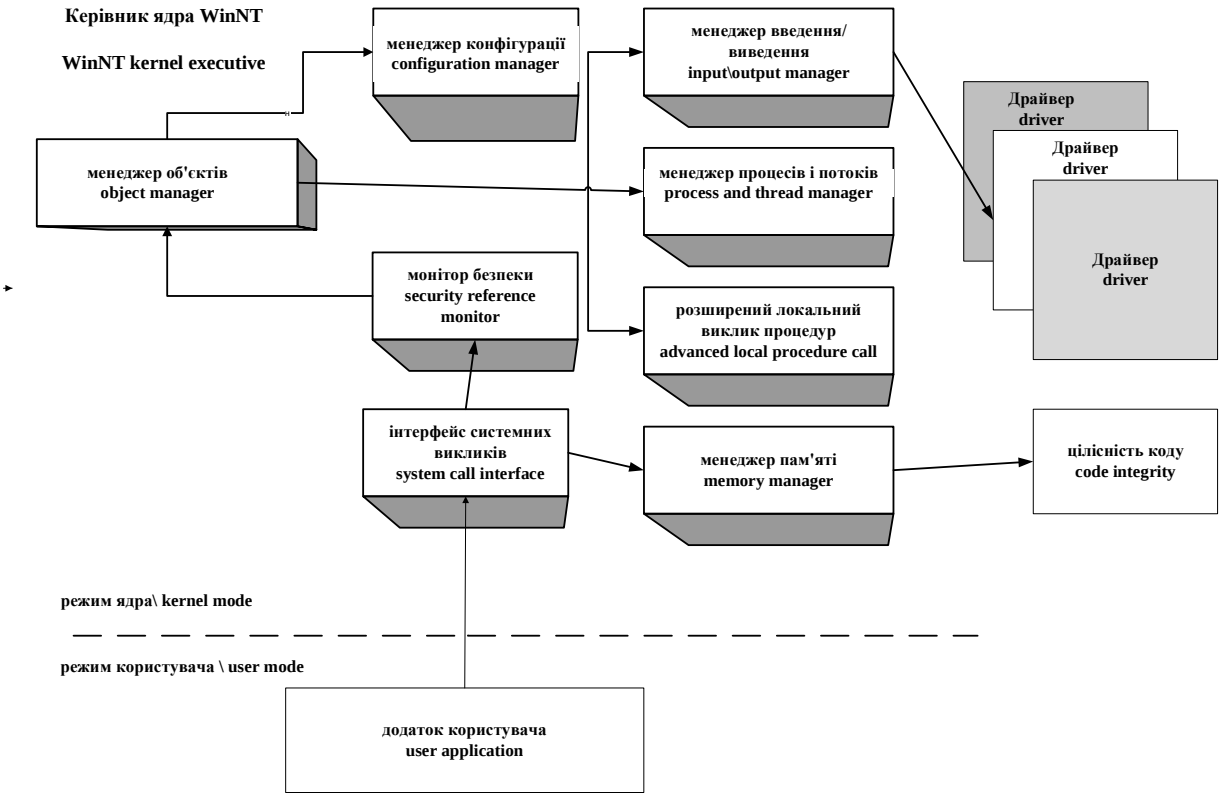


Рисунок 2.1. Виконавчі модулі ядра Windows

Кожна підсистема в виконавчій системі ядра надає API для виклику іншими підсистемами. Якщо є можливість переглянути код ядра, ви швидко визначите, до якої підсистеми відноситься кожен API, використовуючи його двосимвольний префікс. Префікси для підсистеми на рис. 2.1 показані в таб.2.1.

Таблиця 2.1.

Відображення префіксів API і підсистем

Префікс	Підсистема	Приклад
Nt або Zw	Системний виклик	NtOpenFile / ZwOpenFile
Se	Довідковий монітор	SeAccessCheck
Ob	Менеджер об'єктів	ObReferenceObjectByHandle
Ps	Процес і нитка менеджер	PsGetCurrentProcess
Cm	Менеджер конфігурації	CmRegisterCallback
Mm	Менеджер пам'яті	MmMapIoSpace
Io	Вхід / вихід менеджер	IoCreateFile
Ci	Цілісність коду	CiValidateFileObject

2.2. МОНІТОР БЕЗПЕКИ

Монітор безпеки (SRM) є самою важливою підсистемою в ядрі. Він реалізує механізми безпеки, які обмежують доступ користувачів до різних ресурсів. Без SRM не можна було б заборонити іншим користувачам отримувати доступ до ваших файлів.

На рис. 2.2 показано SRM і пов'язані з ним системні компоненти.

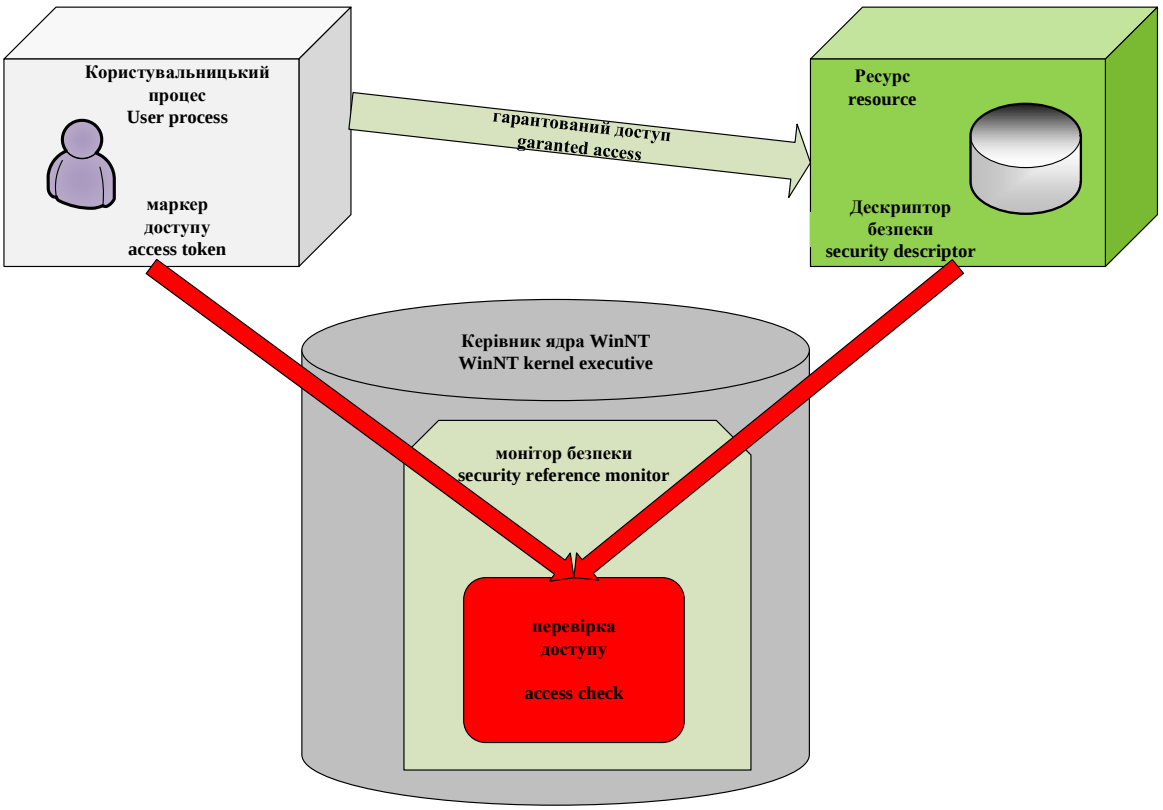


Рисунок 2.2. Компоненти монітора *Security Reference*

Кожному процесу, при його створенні, призначається маркер доступу. Цим доступом керує SRM і визначає особистість користувача, пов'язаного з цим процесом. Потім SRM може виконати операцію, яку називають перевіркою доступу. Ця операція запитує дескриптор безпеки доступу до ресурсу, порівнює його з токеном процесу або враховує рівень наданого доступу, або вказує, що доступ заборонено для сторони, що викликає процес.

SRM також відповідає за аудит створення подій кожен раз, коли користувач отримує доступ до ресурсу. За замовчуванням аудит відключений, через обсяг подій, які він може створити. Тому адміністратор повинен спочатку ввімкнути його. Події аудиту можуть використовуватися для виявлення шкідливих даних у системі, а також для діагностики невірних параметрів безпеки.

SRM передбачає, що користувачі та групи будуть представлені у вигляді подвійних структур, які мають назву ідентифікатори безпеки (SID). Однак передача необроблених подвійних SID не дуже зручна для користувачів. Користувачі, зазвичай, звертаються до інших користувачів і груп за осмисленими іменами (наприклад, користувач *Taras* або група *Users*). Ці імена необхідно перетворити в SID, перш ніж SRM зможе їх використовувати. Задача перетворення імені в SID виконується в локальній підсистемі управління безпекою (LSASS), яка працює всередині привілейованого процесу, незалежного від будь-яких користувачів які зайшли в систему.

Неможливо представити всі можливі SID як ім’я, тому Microsoft визначає формат мови визначення безпеки дескрипторів (SDDL) для представлення SID у вигляді рядка. SDDL може представляти собою весь ресурс безпеки дескриптора, але на разі просто використовуємо його для представлення SID. У лістингу 2.1 використовуємо PowerShell для пошуку імен групи користувачів за допомогою команди Get -NtSid, це має надати нам SDDL для SID.

Лістинг 2.1. Запит SID групи користувачів за допомогою *Get-NtSid*

```
PS C:\Users\hladk> Get-NtSid -Name "Users"

Name          Sid
----          ---
BUILTIN\Users S-1-5-32-545
```

Передаємо ім’я групи користувачів у *Get-NtSid* , яке повертає повне ім’я з прикріпленим локальним доменом BUILTIN.BUILTIN\Users та SID.

Інформація надана в лістингу також містить SID у форматі SDDL, який можна розбити на наступні частини, розділені дефісом:

— **Префікс символ S**. Це вказує на те, що далі слід SDDL SID.
— **Версія структури SID** в десятковому форматі. Має фіксоване значення 1.
— Служба безпеки. Центр 5 вказує на вбудований центр NT.
— **Два відносних ідентифікатора** (RID) у десятковому форматі. RID (маємо 32 і 545) представляють групу центрів NT.

Також можемо використовувати *Get-NtSid* для виконання зворотної операції, відтворюючи SDDL SID навпаки в імені, як показано в лістингу 2.2.

Лістинг 2.2. Використання *Get-NtSid* для пошуку імені, пов’язаного з SID

```
PS C:\Users\hladk> Get-NtSid -Sddl "S-1-5-32-545"

Name          Sid
----          ---
BUILTIN\Users S-1-5-32-545
```

Пам'ятайте, що SID представляє користувачів і групи, і що ми можемо відобразити їх у вигляді рядка у формі SDDL.

2.3. ДИСПЕТЧЕР ОБ'ЄКТІВ

В ***Unix***-подібних операційних системах все є файлом. У Windows все є *об’єктом*, тобто кожен файл, процес і потік представлені в пам’яті ядра як об’єктна структура. Що важливо для безпеки, кожному з цих об’єктів може бути призначений дескриптор безпеки, який обмежує доступ користувачів до об’єкта та визначає тип доступу, який вони мають (наприклад, читання або запис).

Менеджер об’єктів — це компонент ядра, який відповідає за керування об’єктами ресурсів, виділенням пам’яті та терміном їх життя.

2.3.1. Типи об'єктів

Ядро веде список усіх типів об'єктів, які воно підтримує. Це необхідно, оскільки кожен тип об'єкта має різні підтримувані операції та властивості безпеки. У лістингу 2.3 показано, як використовувати команду *Get-NtType* для виведення списку всіх підтримуваних типів у ядрі.

В запропонованому скороченому списку типів є декілька цікавих записів. Перший запис у згенерованому списку — *Type*, як ми бачимо навіть список типів ядра, побудованого з об'єктів! Інші типи, які наведені в лістингу — *Process* і *Thread.* Ці типи представляють об’єкти ядра для процесу та потоку, відповідно. Далі розглянемо детальніше інші типи об’єктів.

Лістинг 2.3. Виповнення *Get-NtType*

```
PS C:\Users\hladk> Get-NtType

Name
----
Type
Directory
SymbolicLink
Token
Session
Job
Process
Thread
Partition
UserApcReserve
IoCompletionReserve
ActivityReference
ProcessStateChange
ThreadStateChange
CpuPartition
SchedulerSharedData
PsSiloContextPaged
PsSiloContextNonPaged
DebugObject
Event
Mutant
Callback
Semaphore
Timer
```

Відобразити властивість типу можемо за допомогою *Format-List,* який повертає додаткову інформацію про цей тип. Для вирішення питання - як отримати доступ до кожного з цих типів, потрібно розглянути простір імен диспетчера об'єктів.

2.3.2. Простір імен диспетчера об'єктів

Як користувач Windows, ви зазвичай бачите диски своєї файлової системи в Провіднику. Але існує додаткова файлова система лише для об'єктів ядра. Вона знаходиться під інтерфейсом користувача. Доступ до цієї файлової системи, яку називають простором іменем об’єктів диспетчера (OMNS), погано документований, що робить його ще більш цікавим.

OMNS побудовано з об'єктів каталогу. Об'єкти діють таким чином, якби ніби вони знаходилися у файловій системі, тому кожен каталог містить інші об'єкти, які ви можете вважати файлами. Однак вони відрізняються від звичайних каталогів файлів. Кожен каталог створено за допомогою дескриптора безпеки, який визначає, які користувачі можуть тільки переглядати його, а які користувачі можуть створювати нові підкаталоги та об’єкти всередині нього. Ви можете побачити повний шлях до об’єкта за допомогою рядків, розділених зворотною косою межею.

Ми можемо отримати перелік OMNS. Як показано в лістингу 2.4.

Лістинг 2.4. Лістинг кореневого каталогу OMNS

```
PS C:\Users\hladk> ls NtObject:\ | Sort-Object Name

Name                                                                                                  TypeNam
                                                                                                      e
----                                                                                                  -------
ArcName                                                                                               Dire...
aswFsBlkPort                                                                                          Filt...
aswPort                                                                                               Filt...
BaseNamedObjects                                                                                      Dire...
BindFltPort                                                                                           Filt...
Callback                                                                                              Dire...
CLDMSGPORT                                                                                            Filt...
clfs                                                                                                  Device
Container_Microsoft.ScreenSketch_11.2409.25.0_x64__8wekyb3d8bbwe-S-1-5-21-2687063813-2248694510-27868876-1001  Job
Container_Microsoft.WindowsTerminal_1.21.3231.0_x64__8wekyb3d8bbwe-S-1-5-21-2687063813-2248694510-27868876-1001 Job
Container_Microsoft.YourPhone_1.24112.110.0_x64__8wekyb3d8bbwe-S-1-5-21-2687063813-2248694510-27868876-1001    Job
CsrSbSyncEvent                                                                                        Event
Device                                                                                                Dire...
Dfs                                                                                                   Symb...
DosDevices                                                                                            Symb...
Driver                                                                                                Dire...
DriverData                                                                                            Symb...
DriverStore                                                                                           Dire...
DriverStores                                                                                          Symb...
EFSInitEvent                                                                                          Event
Fat                                                                                                   Device
FatCdrom                                                                                              Device
FileSystem                                                                                            Dire...
GLOBAL??                                                                                              Dire...
KernelObjects                                                                                         Dire...
```

Лістинг 2.4 показує короткий фрагмент кореневого каталогу OMNS. За замовчуванням, цей результат включає ім'я кожного об'єкта і його тип. Ми бачимо кілька об'єктів **Directory.** Ви можете створити їх список, якщо маєте на це дозвіл.

Ми також бачимо інший важливий тип, *SymbolicLink*.

Ви можете використовувати символічні посилання для перенаправлення одного шляху OMNS на інший. Об'єкт *SymbolicLink* містить властивість *SymbolicLinkTarget*, яка сама містить об'єкт, який має відкрити посилання. Наприклад, лістинг 2-5 показує об'єкт для символічного посилання в корені OMNS.

Лістинг 2.5. Відображення цілих символічних посилань

```
PS C:\Users\hladk> ls NtObject:\Dfs | Select-Object SymbolicLinkTarget

SymbolicLinkTarget
------------------
\Device\DfsClient
```

Тут ми зазначаємо шлях \Dfs OMNS, а потім отримуємо властивість *SymbolicLinkTarget*, щоб знайти справжню мету.

Таблиця 2.2.

Загальні каталоги об'єктів та їх описи

шлях	опис
\ BaseNamedObjects	Глобальний каталог для об'єктів користувача
\Device	Каталог, що містить пристрої, змонтовані файлові системи
\GLOBAL??	Глобальний каталог для символічних посилань, включаючи відображення дисків
\KnownDlls	Каталог, що містить спеціальні, відомі відображення DLL
\ObjectTypes	Каталог, що містить іменовані типи об’єктів
\Sessions	Каталог для окремих консольних сеансів
\Windows	Каталог для об'єктів, пов'язаних з Window Manager
\RPC Control	Каталог для кінцевих точок виклику віддалених процедур

Далі ми перевіряємо шлях до мети, Device\DfsClient, щоб показати, що це тип Device, до якого можна отримати доступ за допомогою символічного посилання.

Windows попередньо налаштовує кілька важливих каталогів об’єктів, показаних у таб. 2-2.

Перший каталог у таб.2.2, *BaseNamedObjects* (BNO), є важливим у контексті диспетчера об'єктів. Він дозволяє будь-якому користувачеві створювати іменовані об'єкти ядра. Цей єдиний каталог дозволяє спільно використовувати ресурси різних користувачів у локальній системі. Зверніть увагу, що вам не обов'язково створювати об'єкти в каталозі BNO. Це лише загальноприйнята практика.

Ми детальніше опишемо інші каталоги об'єктів пізніше в цьому розділі. Наразі ви можете вивести їх у PowerShell, додавши префікс NtObject: до шляху, як показано в лістингу 2.5.

2.3.3. Системні виклики

Інколи виникає питання: яким чином можемо отримати доступ до іменованих об’єктів в OMNS із додатків режиму користувача? Якщо додаток працює в режимі користувача, потрібно ядро для доступу до об’єктів. Ми можемо викликати код ядра режиму, використовуючи інтерфейс системних викликів. Більшість системних викликів виконують операцію над певним типом об'єкта ядра, що надається диспетчером об'єктів. Наприклад, системний виклик *NtCreateMutant* створює об’єкт *Mutant*, який є примітивом взаємного виключення, що використовується для блокування та синхронізації потоків.

Ім'я системного виклику відповідає загальному шаблону. Воно починається з Nt або Zw. Для виклику в режимі користувача ці два префікси еквівалентні, однак, якщо системний виклик активується кодом, що виконується в ядрі, префікс Zw змінює процес перевірки безпеки.

Після префікса йде дієслово операції: *Create*, у випадку *NtCreate Mutant*. Інша частина імені відноситься до типу об'єкта ядра, з яким працює системний виклик. Загальні дієслова системного виклику, які виконують операцію над об'єктом ядра, включають:

Create **–** Створює новий об'єкт. Відповідає командам PowerShell *New-Nt<Type>*;

Open **-** Відкриває існуючий об'єкт. Відповідає командам PowerShell *Get-Nt<Type>*;

QueryInformation - Запитує інформацію та властивості об'єкта;

SetInformation **-** Встановлює інформацію та властивості об'єкта.

Деякі системні виклики виконують операції, специфічні для певного типу. Наприклад, *NtQueryDirectoryFile* використовується для запиту записів у файлі каталогу об’єктів.

Перегляньте прототип на мові C для системного виклику *NtCreateMutant*, щоб отримати, параметри які необхідно передати в типовий виклик. Як показано в лістингу 2.6, системний виклик *NtCreateMutant* створює новий об'єкт *Mutant* .

Лістинг 2.6. Прототип C для *NtCreateMutant*

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

Перший параметр для системного виклику — це вихідний вказівник на HANDLE. Зазвичай у багатьох системних викликах цей параметр використовується для отримання відкритого дескриптора об’єкта (в цьому випадку *Mutant*) при успішному виконанні функції. Використаємо дескриптори разом з іншими системними викликами для доступу до властивостей і виконання операцій. У випадку нашого об’єкта дескриптор *Mutant* дозволяє нам отримувати та знімати блокування для синхронізації потоків.

Далі йде DesiredAccess, який представляє операції, які викликаючий хоче мати можливість виконувати над *Mutant* за допомогою дескриптора. Наприклад, ми могли б запитати доступ, який дозволяє розблокувати *Mutant*. Якщо б ми не запитали цей доступ, будь-який додаток, який спробував очікувати *Mutant*, негайно зазнав би невдачі. Наданий доступ залежить від результатів перевірки доступу SRM. Дескриптори та *DesiredAccess* докладніше розглянемо далі.

Третій — параметр *ObjectAttributes*, який визначає атрибути для об'єкта, які потрібно відкрити або створити. Структура OBJECT_ATTRIBUTES визначена, як показано в лістингу 2.7.

Лістинг 2.7. Структура OBJECT_ATTRIBUTES

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

Ця структура мови C починається з *Length*, яка представляє довжину структури. Вказівка довжини структур на початку є ідіомою стилю C, щоб гарантувати, що правильна структура була передана системному виклику.

Далі йдуть *RootDirectory* і *ObjectName*. Вони розглядаються разом, оскільки ~~вонн~~ вказують, де системний виклик повинен шукати ресурс до якого передбачається доступ. *RootDirectory* — це дескриптор відкритого ядра об'єкта, який можна використовувати в якості основи для пошуку об'єкта. Поле *ObjectName* — це покажчик на структуру UNICODE_STRING. Це розрахунковий рядок, визначений в лістингу 2.8 як структура мови C.

Лістинг 2.8. Структура UNICODE_STRING

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

Структура посилається на строкові дані через *Buffer*, який є вказівником на масиви 16-бітних символів *Unicode*. Рядок представлений в кодуванні UCS-2. Взагалі Windows пропонує багато змін в *Unicode* , такі як UTF-8 і UTF-16.

Структура *UNICODE_STRING* також містить два поля довжини, *Length* і *MaximumLength*. Перше поле довжини містить загальну допустиму довжину рядка, на яку вказує *Buffer*, у байтах (не в символах *Unicode*). Якщо ви маєте досвід програмування на C, ця довжина не включає символи закінчення NUL. Насправді в іменах об’єктів дозволено використовувати символ NUL.

Друге поле довжини є максимальною довжиною рядка, на яку вказує *Buffer*, у байтах. Оскільки структура має дві окремі довжини, можна виділити пустий рядок з великою максимальною довжиною та допустимою довжиною, який дорівнює нулю, а потім оновити значення рядка за допомогою вказівника *Buffer*. Зверніть увагу, що довжини зберігаються як значення *USHORT*, які є 16-бітними цілими числами без знаку. У поєднанні з байтами, що представляють довжину, це означає, що довжина рядка може бути не більше 32 767 символів.

Таблиця 2.3.
Опис прапорців атрибутів об'єкта

Назва PowerShell	Опис
Inherit	Позначає handle як успадкований.
Permanent	Позначає handle як постійний.
Exclusive	Позначає handle як ексклюзивний у разі створення нового об’єкта, тобто лише той самий процес може відкрити handle об’єкта.
CaseInsensitive	Шукає назву об’єкта без урахування регістру.
OpenLink	Відкриває об’єкт, якщо він є посиланням на інший об’єкт, в іншому випадку перейде за посиланням. Використовується лише менеджером конфігурації.
OpenIf	Якщо використовується виклик Create, відкриває дескриптор існуючого об’єкта, якщо він доступний.
KernelHandle	Відкриває дескриптор як дескриптор ядра при використанні в режимі ядра. Це запобігає користувальницьким програмам прямий доступ до handle.
ForceAccessCheck	При використанні в режимі ядра забезпечує виконання всіх перевірок доступу, навіть якщо викликається Zw-версія системного виклику.
IgnorelpersonatedDeviceMap	Вказує на те, що слід не переходити за жодним шляхом, який містить символічне посилання.
DontReparse	Вказує ні до слідувати будь-який шлях що містить символічне посилання .

Щоб вказати ім’я об’єкта, є два варіанти: ви можете задати ObjectName як абсолютний шлях, наприклад: \BaseNamedObjects\ABC, або ви можете задати: RootDirectory як каталог об’єкта для \BaseNamedObjects і змінити ABC як ObjectName . Ці дві дії відкривають один і той же об'єкт.

Повертаючись до лістингу 2-7, після параметра ObjectName йде Attributes, який є набором прапорців для модифікації процесу пошуку імені об'єкта або зміни властивостей поверненого дескриптора. У таб.2.3 наведено допустимі значення для поля Attributes.

Останні два поля в структурі OBJECT_ATTRIBUTES дозволяють вказати якість обслуговування безпеки (SQoS) і дескриптор безпеки для об’єкта.

Далі в системному виклику *NtCreateMutant* (лістинг 2.6) міститься логічний параметр *InitialOwner* , який є специфічним для цього типу. У нашому випадку він вказує, чи належить створений *Mutant* абоненту, що викликається або ні. Багато інших системних викликів, особливо для файлів, мають більш складні параметри.

2.3.4. Коди NTSTATUS

Усі системні виклики повертають 32-розрядний код NTSTATUS. Цей код стану складається з кількох компонентів, упакованих у 32 біта, як показано на Рисунок 2.3.

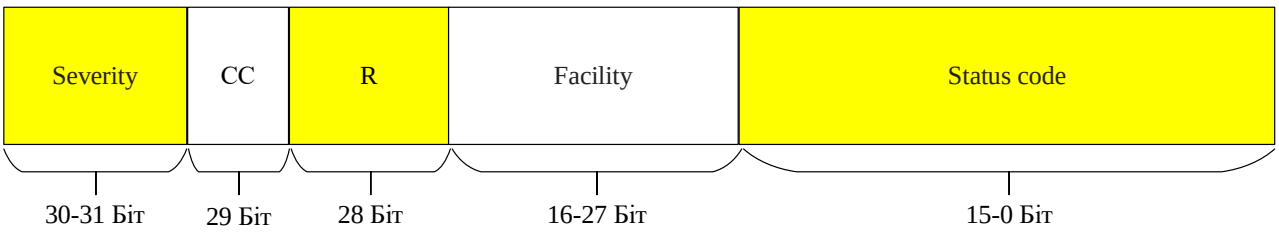


Рисунок 2.3. Структура коду NTStatus

Самі значимі два біта (31 і 30) вказують на серйозність стану коду. Таб. 2.4 показує доступні значення.

Таблиця 2.4.
Коди серйозності *NTStatus*

Назва	Значення
STATUS_SEVERITY_SUCCESS	0
ІНФОРМАЦІЙНИЙ СТАТУС СУВНОСТІ	1
STATUS_SEVERITY_WARNING	2
STATUS_SEVERITY_ERROR	3

Якщо рівень серйозності вказує на попередження або помилку, то біт 31 коду стану буде встановлений на 1. Якщо код стану обробляється як 32-бітне ціле число зі знаком, цей біт являє собою від'ємне значення. Загальноприйнятою практикою кодування є припущення, що якщо код статусу є від’ємним, він позначає помилку, а якщо він є додатним, він позначає успіх. Як ми бачимо з таблиці, це припущення не є повністю вірним — від’ємний код статусу також може позначати попередження — але на практиці воно працює досить добре.

Наступний компонент на рис. 2.3, CC, є кодом клієнта. Це прапор, який вказує, чи код статусу визначений Microsoft (значення 0) або визначений третьою стороною (значення 1). Треті сторони не зобов'язані дотримуватися цієї специфікації, тому не слід розглядати це як факт.

Після коду клієнта йде біт *R*, зарезервований біт, який повинен бути встановлений на 0.

Наступні 12 бітів вказують на об'єкт, тобто компонент або підсистему, пов'язану з кодом статусу. Microsoft заздалегідь визначила близько 50 об'єктів для своїх власних цілей. Треті сторони повинні визначити свої власні об'єкти і поєднати їх з кодом клієнта, щоб відрізнити себе від Microsoft. У таб.2.5 наведено кілька найпоширеніших об'єктів.

Таблиця 2.5.
Значення загального статусу Facility

Об'єкт назва	Значення	Опис
FACILITY_DEFAULT	0	За замовчуванням використовується для загальних кодів стану
FACILITY_DEBUGGER	1	Використовується для кодів, пов’язаних із налагоджувачем
FACILITY_NTWIN32	7	Використовується для кодів Win32 API

Останній компонент, код стану, являє собою 16-бітне число, унікальне для об'єкта. Розробник має визначити значення кожного числа. Модуль PowerShell містить список відомих станів кодів, які ми можемо запитати командою *Get-NtStatus* без параметрів (лістинг 2.9).

Лістинг 2.9. Приклад виконання *Get-NtStatus*

```
PS C:\Users\hladk> Get-NtStatus

Status   StatusName                                                      Message
------   ----------                                                      -------
00000000 STATUS_SUCCESS                                                  STATUS_SUCCESS
00000001 STATUS_WAIT_1                                                   STATUS_WAIT_1
00000002 STATUS_WAIT_2                                                   STATUS_WAIT_2
00000003 STATUS_WAIT_3                                                   STATUS_WAIT_3
0000003F STATUS_WAIT_63                                                  STATUS_WAIT_63
00000080 STATUS_ABANDONED_WAIT_0                                         STATUS_ABANDONED_WAIT_0
000000BF STATUS_ABANDONED_WAIT_63                                        STATUS_ABANDONED_WAIT_63
000000C0 STATUS_USER_APC                                                 STATUS_USER_APC
```

Зверніть увагу на те, що деякі значення статусу, такі як STATUS_PENDING, мають зрозуміле для людини повідомлення. Це повідомлення не вбудовано в модуль PowerShell, замість цього він зберігається в бібліотеці Windows і може бути «витягнутий» під час виконання.

Коли ми хочемо зробити системний виклик за допомогою команди PowerShell, код стану відображається через виняткову ситуацію .NET. Наприклад, якщо спробуємо відкрити неіснуючий об’єкт *Directory*, то побачимо на консолі виняток, показаний у лістингу 2.10.

Лістинг 2.10. Виключення NTSTATUS, згенероване при спробі відкрити неіснуючий каталог

```
PS C:\WINDOWS\system32> Get-NtDirectory \THISDOESNOTEXIST
Get-NtDirectory : (0xC0000034) - Object Name not found.
At line:1 char:1
+ Get-NtDirectory \THISDOESNOTEXIST
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : NotSpecified: (:) [Get-NtDirectory], NtException
    + FullyQualifiedErrorId : NtCoreLib.NtException,NtObjectManager.Cmdlets.Object.GetNtDirectoryCmdlet
```

```
PS C:\WINDOWS\system32> 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
```

У списку 2.10 використовуємо *Get-NtDirectory*, щоб відкрити неіснуючий шлях. Це генерує виключення NTSTATUS 0xC0000034, яке показано із декодованим повідомленням.

Якщо ви хочете отримати більше інформації про код стану, ви можете передати його до *Get-NtStatus* і налаштувати виведення у вигляді списку, щоб переглянути всі його властивості, включаючи Facility і Severity. Код стану NT є цілим числом без знаку, проте часто його також відображають (неправильно) як число зі знаком.

2.3.5. Дескриптори об'єктів

Менеджер об'єктів обробляє посилання на пам'ять ядра. Додаток у режимі користувача не може безпосередньо читати або записувати в пам'ять ядра, тоді як він може отримати доступ до об'єкта? Він робить це за допомогою дескриптора, поверненого системним викликом, як описано в попередньому розділі.

Кожен запущений процес має пов'язану таблицю дескрипторів, що містить три елементи інформації:

— Числовий ідентифікатор дескриптора;

— Наданий доступ до дескриптора, наприклад, читання або запис;

— Покажчик на структуру об'єкта в пам'яті ядра.

Кожен запущений процес має пов'язану таблицю дескрипторів, що містить три елементи інформації:

— Числовий ідентифікатор дескриптора;

— Наданий доступ до дескриптора, наприклад, читання або запис;

— Покажчик на структуру об'єкта в пам'яті ядра.

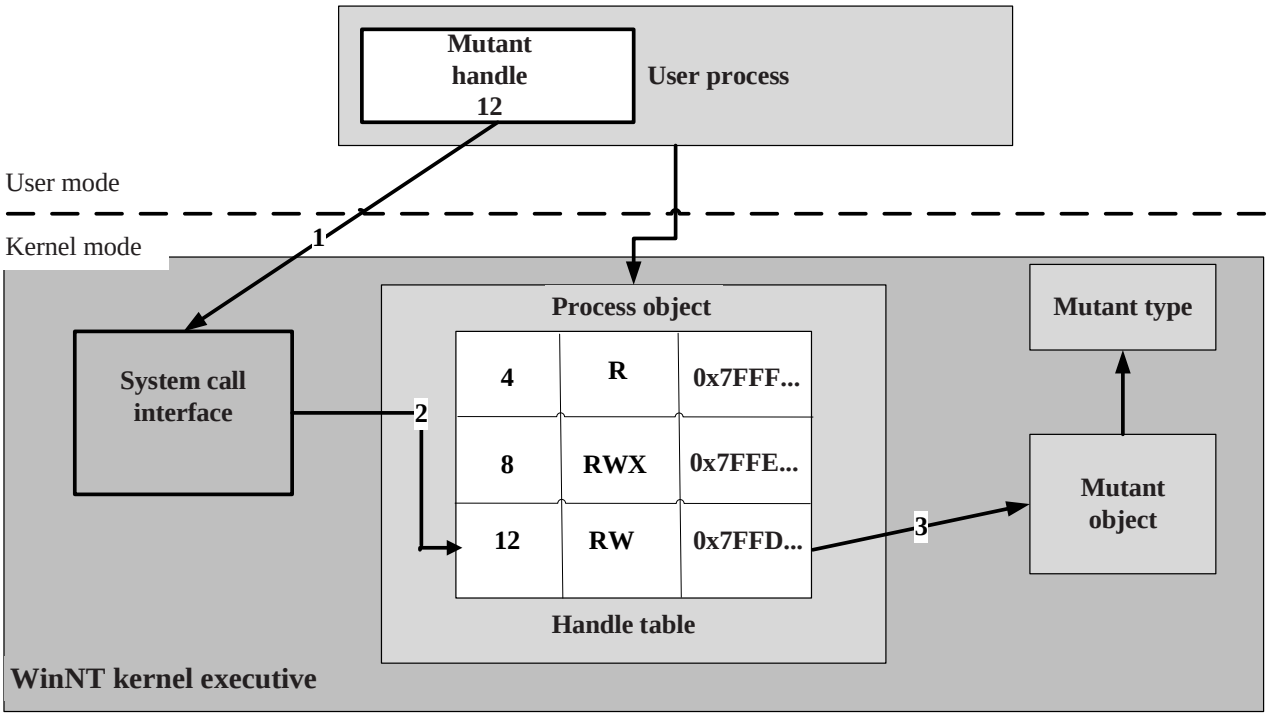


Рисунок 2.4. Процес пошуку в таблиці дескрипторів

Перш ніж ядро зможе використовувати дескриптор, реалізація системного виклику повинна знайти покажчик об'єкта ядра в таблиці дескрипторів за допомогою API ядра, такого як *ObReferenceObjectByHandle*. Надаючи цей дескриптор опосередковано, компонент ядра може повернути номер дескриптора до програми в режимі користувача, не розкриваючи об'єкт ядра безпосередньо. На рис. 2.4 показано процес пошуку дескриптора.

На рис. 2.4 користувач намагається виконати якусь операцію над об'єктом *Mutant*. Коли процес користувача має намір використовувати дескриптор, він повинен спочатку передати значення дескриптора системному виклику, яке ми визначили в попередньому розділі.

Потім реалізація системного виклику викликає API ядра, щоб перетворити дескриптор на покажчик ядра, посилаючись на числове значення дескриптора в таблиці дескрипторів процесу.

Щоб визначити, чи надавати доступ, API перетворення враховує тип доступу, який запитав користувач для виконання операцій системного виклику, а також тип об’єкта, до якого здійснюється доступ. Якщо-доступ за запитом не збігається з наданим доступом, записаним у записі таблиці дескрипторів, API поверне STATUS_ACCESS_DENIED і операція перетворення не вдасться. Так само, якщо типи об’єктів не збігаються з API повернеться STATUS_OBJECT_TYPE_MISMATCH.

Ці дві перевірки мають вирішальне значення для безпеки. Перевірка доступу гарантує, що користувач не зможе виконати операцію з дескриптором, до якого у нього немає доступу (наприклад, запис у файл, до якого у нього є тільки доступ на читання).

Перевірка типу гарантує, що користувач не передав непов’язаний тип об’єкта ядра, що може призвести до плутанини в ядрі, спричиняючи проблеми безпеки, наприклад пошкодження пам’яті. Якщо перетворення вдається, системний виклик тепер має покажчик ядра на об’єкт, який він може використовувати для виконання операції, яку запитує користувач.

2.3.6. Маски доступу

Значення наданого доступу в таблиці дескрипторів представляє собою 32-бітне бітове поле, зване маскою доступу. Це саме бітове поле, яке використовується для параметра *DesiredAccess*, зазначеного в системному виклику.

Маска доступу складається з чотирьох компонентів, як показано на рис. 2.5.

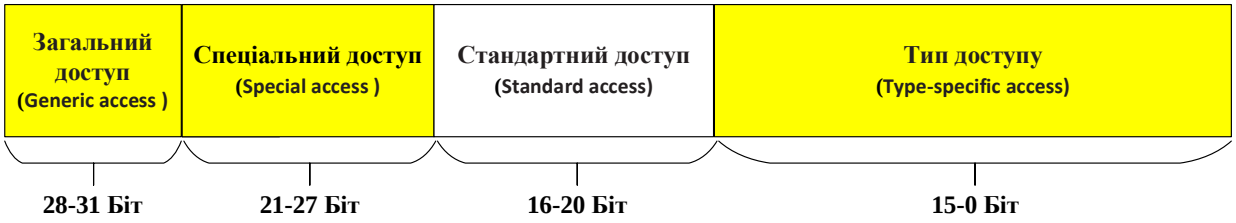


Рисунок 2.5. Структура маски доступу

Найважливішим з них є специфічний для типу 16-бітний компонент доступу, який визначає операції, дозволені для певного типу об’єкта ядра.

Наприклад, об’єкт *File* може мати окремі біти, щоб вказати, чи дозволено файл читати або записувати під час використання *handle*. На відміну від цього, подія синхронізації може мати лише один біт, який дозволяє сигналізувати про подію.

У зворотному напрямку *standard access* доступу маски доступу визначає операції, які можна застосовувати до будь-якого типу об’єкта. Ці операції включають:

Delete - Видаляє об'єкт, наприклад, видаляючи його з диска або з реєстру;

ReadControl - Зчитує інформацію дескриптора безпеки для об'єкта;

WriteDac - Записує дискреційне керування доступом (DAC) дескриптора безпеки об’єкта;

WriteOwner - Записує інформацію про власника об'єкта;

Synchronize - Очікує на об'єкті; наприклад, чекає завершення процесу або розблокування *Mutant*.

Перед цим ідуть зарезервовані і спеціальні біти доступу. Вони включають два значення доступу:

AccessSystemSecurity – Зчитує або записує інформацію аудиту на об'єкті;

MaximumAllowed - Запитує максимальний доступ до об'єкта при виконанні перевірки доступу.

Нарешті, чотири старших біта маски доступу (універсальний доступ до компонентів) використовуються лише при запиті доступу до об'єкта ядра з використанням параметра *DesiredAccess* системного виклику. Існує чотири універсальні категорії доступу: *GenericRead, GenericWrite, GenericExecute* і *GenericAll*.

Лістинг 2.11. Відображення загальної таблиці розміщення для типів об’єктів

```
PS C:\Users\hladk> 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
Session                       R:00020001 W:00020002 E:00120001 A:000F0003
Job                           R:00020004 W:0002000B E:00120000 A:001F003F
Process                       R:00020410 W:00020BEA E:00121001 A:001FFFFF
Thread                        R:00020048 W:00020437 E:00121800 A:001FFFFF
Partition                     R:00020001 W:00020002 E:00120001 A:001F0003
UserApcReserve                R:00020001 W:00020002 E:00020000 A:000F0003
IoCompletionReserve           R:00020001 W:00020002 E:00020000 A:000F0003
ActivityReference             R:00020000 W:00020000 E:00020000 A:001F0000
ProcessStateChange            R:00020000 W:00020001 E:00020001 A:000F0001
ThreadStateChange             R:00020000 W:00020001 E:00020001 A:000F0001
CpuPartition                  R:00020001 W:00020002 E:00020004 A:000F0007
SchedulerSharedData           R:00020000 W:00020001 E:00020001 A:000F0001
PsSiloContextPaged            R:00020000 W:00020000 E:00020000 A:000F0000
PsSiloContextNonPaged         R:00020000 W:00020000 E:00020000 A:000F0000
DebugObject                   R:00020001 W:00020002 E:00120000 A:001F000F
Event                         R:00020001 W:00020002 E:00120000 A:001F0003
Mutant                        R:00020001 W:00020000 E:00120000 A:001F0001
Callback                      R:00020000 W:00020001 E:00120000 A:001F0001
Semaphore                     R:00020001 W:00020002 E:00120000 A:001F0003
Timer                         R:00020001 W:00020002 E:00120000 A:001F0003
```

Якщо користувач потребує один з цих універсальних прав доступу, SRM спочатку має перевірити доступ до відповідного *type-specific access*. Це означає, що користувач ніколи не отримаєте доступ до дескриптора за допомогою *GenericRead*. Замість цього буде надано доступ до певної маски доступу, яка представляє операції читання для цього типу. Щоб полегшити перетворення, кожен тип містить загальну таблицю відображення, яка відображає чотири загальні категорії для доступу до певного типу. Таблицю відображення за допомогою *Get-NtType*, показано в лістингу 2.11.

Дані типу не містять імен для кожної конкретної маски доступу.

Однак для всіх поширених типів модуль PowerShell надає нумерований тип, який представляє доступ до певного типу. Є можливість отримати доступ до цього типу за допомогою команди *Get-NtTypeAccess*. Лістинг 2.12 показує приклад для типу *File*.

Вихідні дані команди *Get-NtTypeAccess* показують значення маски доступу, ім’я доступу, яке відоме модулю PowerShell, і загальний доступ, з яким він буде порівняний.

Лістинг 2.12. Відображення маски доступу для типу *File* об'єкта

```
PS C:\Users\hladk> 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
```

Для підвищення зручності використання модуля PowerShell було змінено вихідні назви прав доступу, в порівнянні тими, що знаходяться в комплекті розробки програмного забезпечення Windows (SDK). Ви можете переглянути еквівалентні назви SDK, використовуючи властивість *SDKName* з командою *Get-NtTypeAccess* :

```
PS C:\Users\hladk> Get-NtTypeAccess -Type File | Select SDKName, Value

SDKName                       Value
-------                       -----
FILE_READ_DATA                ReadData
FILE_WRITE_DATA              WriteData
FILE_APPEND_DATA            AppendData
FILE_READ_EA                    ReadEa
FILE_WRITE_EA                  WriteEa
FILE_EXECUTE                   Execute
FILE_DELETE_CHILD          DeleteChild
FILE_READ_ATTRIBUTES    ReadAttributes
FILE_WRITE_ATTRIBUTES WriteAttributes
DELETE                         Delete
READ_CONTROL                ReadControl
WRITE_DAC                      WriteDac
WRITE_OWNER                 WriteOwner
SYNCHRONIZE                 Synchronize
```

Ці компоненти є корисними для перенесення власного коду в PowerShell.

Перетворення виконуються між числовою маскою доступу та визначеними типами об’єктів за допомогою команди *Get-NtAccessMask*, як показано в лістингу 2.13.

Лістинг 2.13. Перетворення масок доступу за допомогою *Get-NtAccessMask*

```
PS C:\Users\hladk> Get-NtAccessMask -FileAccess ReadData, ReadAttributes, ReadControl

Access
------
00020081

PS C:\Users\hladk> Get-NtAccessMask -FileAccess GenericRead

Access
------
80000000

PS C:\Users\hladk> Get-NtAccessMask -FileAccess GenericRead -MapGenericRights

Access
------
00120089

PS C:\Users\hladk> Get-NtAccessMask 0x120089 -AsTypeAccess File
ReadData, ReadEa, ReadAttributes, ReadControl, Synchronize
```

На прикладі лістингу 2.13 видно, що спочатку відбувається запит маски доступу з набору імен доступу *File* і, відповідно, отримуємо числову маску

доступу в шістнадцятковому форматі. Потім отримуємо маску доступу для доступу *GenericRead.*

Отже, значення що повертається — це просто числове значення *GenericRead* .

Потім відбувається запит маски доступу для *GenericRead*, але необхідно вказати, що хотілося б створити загальний доступ за допомогою параметра *MapGenericRights*. Оскільки вказано доступ для типу *File*, то ця команда використовує загальне розміщення типу *File* для перетворення в певну маску доступу. Нарешті, перетворюємо необроблену маску доступу навпаки в тип доступу з параметром *AsTypeAccess*, вказавши тип ядра для використання.

Як показано в лістингу 2.14, може відбутись запит маски наданого доступу дескриптора об’єкта через властивість *GrantedAccess* об’єкта PowerShell. Це повертає формат перерахованого типу для маски доступу. Щоб отримати числове значення, використовуйте властивість *GrantedAccessMask* .

Лістинг 2.14. Відображення числового значення маски доступу за допомогою *GrantedAccessMask*

```
PS C:\Users\hladk> $mut = New-NtMutant
PS C:\Users\hladk> $mut.GrantedAccess
ModifyState, Delete, ReadControl, WriteDac, WriteOwner, Synchronize
PS C:\Users\hladk> $mut.GrantedAccessMask

Access
------
001F0001
```

Ядро надає можливість завантажити всі записи таблиць дескрипторів у систему через системний виклик *NtQuerySystemInformation*. Доступ до таблиці дескрипторів з PowerShell отримуємо за допомогою команди *Get-NtHandle*, як показано в лістингу 2.15.

Кожний запис дескриптора містить тип об’єкта, адресу ядра об’єкта в пам’яті ядра та надану маску доступу.

Лістинг 2.15. Відображення таблиці дескрипторів для поточного процесу за допомогою *Get-NtHandle*

```
PS C:\Users\hladk> Get-NtHandle -ProcessId $pid

ProcessId Handle ObjectType           Object           GrantedAccess
--------- ------ ----------           ------           -------------
19128     4      Event                0000000000000000 001F0003
19128     8      Event                0000000000000000 001F0003
19128     12     IRTimer              0000000000000000 00100002
19128     16     TpWorkerFactory      0000000000000000 000F00FF
19128     20     IoCompletion         0000000000000000 001F0003
19128     24     WaitCompletionPacket 0000000000000000 00000001
19128     28     IRTimer              0000000000000000 00100002
19128     32     WaitCompletionPacket 0000000000000000 00000001
19128     36     Event                0000000000000000 001F0003
19128     40     WaitCompletionPacket 0000000000000000 00000001
19128     44     Event                0000000000000000 001F0003
19128     48     WaitCompletionPacket 0000000000000000 00000001
19128     52     EtwRegistration      0000000000000000 00000804
19128     56     EtwRegistration      0000000000000000 00000804
19128     60     EtwRegistration      0000000000000000 00000804
19128     64     EtwRegistration      0000000000000000 00000804
19128     68     Directory            0000000000000000 00000003
19128     72     File                 0000000000000000 00120089
19128     76     Event                0000000000000000 001F0003
19128     80     Event                0000000000000000 001F0003
19128     84     File                 0000000000000000 00100020
19128     88     SchedulerSharedData  0000000000000000 00000001
19128     92     Key                  0000000000000000 00000001
19128     96     File                 0000000000000000 0012019F
19128     100    ALPC Port            0000000000000000 001F0001
```

Після того, як додаток завершив роботу з дескриптором, його можна закрити за допомогою API *NtClose*. Якщо-отримали об’єкт PowerShell із виклику *Get* або *New*, то викликаєте метод *Close* для об’єкта, щоб відкрити дескриптор. Також можна автоматично закривати дескриптор об’єкта в PowerShell, використовуючи команду *Use-NtObject* для виклику блоку сценарію, який закриває дескриптор після завершення його виконання. У лістингу 2.16 наведені приклади обох підходів.

Якщо не закриваєте дескриптори вручну, сміттєзбірник .NET автоматично закриває їх для об’єктів, на яких немає посилань (наприклад, що містяться в змінній PowerShell). Однак слід звикати закривати дескриптори вручну. Інакше, доведеться довго чекати на звільнення ресурсів, бо збірник сміття може запуститися в будь-який час.

Лістинг 2.16. Закриття дескриптора об'єкта

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

Якщо на структуру об'єкта ядра більше не посилаються, або через дескриптор, або через компонент ядра, то об'єкт також буде видалений.

Після остаточного видалення об'єкта вся пам'ять, яку він займав, очищається, а, якщо вона існує, її ім'я в OMNS видаляється.

2.3.7. Дублювання дескрипторів

Ви можете дублювати дескриптори за допомогою системного виклику *NtDuplicateObject*. Основною причиною, по якій ви можете захотіти це зробити, є надання процесу додаткового посилання на об'єкт ядра. Об'єкт ядра не буде знищений, поки всі дескриптори до нього не будуть закриті, тому створення нового дескриптора зберігає об'єкт ядра.

Дублювання дескрипторів також може бути використано для передачі дескрипторів між процесами, якщо дескриптори вихідного та кінцевого процесів мають доступ *DupHandle*. Ви також можете використовувати дублювання дескрипторів для зменшення прав доступу до дескриптора. Наприклад, коли ви передаєте дескриптор файлу новому процесу, ви можете надати дубльованому дескриптору тільки доступ для читання, запобігаючи запису нового процесу в об'єкт. Однак не слід покладатися на цей підхід для зменшення наданого доступу. Якщо процес з дескриптором має доступ до ресурсу, він може просто відкрити його знову, щоб отримати доступ для запису.

У лістингу 2.17 показані деякі приклади використання команди *Copy-NtObject*, яка обробляє *NtDuplicateObject*, щоб виконати інше дублювання в цьому ж процесі.

Спочатку створюємо новий об'єкт *Mutant* для перевірки дублювання дескрипторів і залучаємо поточний наданий доступ, який показує шість прав доступу.

Для первинного дублювання зберігаємо наданий доступ. У першому стовбці можете побачити, що дескриптори відрізняються. Однак виклик *Compare-NtObject* для визначення того, чи посилаються два дескриптора на той же базовий об’єкт ядра, повертає *True*. Потім отримуємо маску доступу для доступу *Mutant ModifyState* і дублюємо дескриптор, запитуючи доступ. В решті решт бачимо, що наданий доступ тепер лише *ModifyState* .

Однак значення Compare-NtObject на разі вказує, що дескриптори посилаються на той же єдиний об'єкт.

Лістинг 2.17. Використання Copy-NtObject для дублювання дескрипторів

```
PS> $mut = New-NtMutant "\BaseNamedObjects\ABC"
PS> $mut.GrantedAccess ModifyState, Delete, ReadControl, WriteDac, WriteOwner, Synchronize
PS> Use-NtObject($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
PS> $mask = Get-NtAccessMask -MutantAccess ModifyState
PS> Use-NtObject($dup = Copy-NtObject $mut -DesiredAccessMask $mask) {
        $dup.GrantedAccess
        Compare-NtObject $mut $dup
}
ModifyState
True
```

Також для дублювання дескрипторів важливі атрибути дескриптора *Inherit* і *ProtectFromClose*. Установка *Inherit* дозволяє новому процесу успадкувати дескриптор під час його створення. Це також дозволяє передавати дескриптори нового процесу для виконання таких завдань, як перенаправлення тексту виведення консолі у файл.

Встановлення *ProtectFromClose* захищає дескриптор від закриття. За потреби є можливість встановити цей атрибут, встановивши властивість *ProtectFromClose* на об’єкті. У лістингу 2.18 показано приклад його використання.

Лістинг 2.18. Тестування атрибута дескриптора *ProtectFromClose*

```
PS C:\Users\hladk> $mut = New-NtMutant
PS C:\Users\hladk> $mut.ProtectFromClose = $true
PS C:\Users\hladk> Close-NtObject -SafeHandle $mut.Handle -CurrentProcess
STATUS_HANDLE_NOT_CLOSABLE
```

Будь-яка спроба відкрити дескриптор завершується невдало з кодом стану STATUS_HANDLE_NOT_CLOSABLE, і дескриптор залишиться відкритим.

2.3.8. Системні виклики Query і Set Information

Об'єкт ядра зазвичай зберігає інформацію про свій стан. Наприклад, об'єкт *Process* зберігає тимчасову мітку часу, коли він був створений. Щоб дозволити вилучити цю інформацію, ядро могло б реалізувати спеціальний системний виклик «отримати час створення процесу». Однак, через обсяг інформації, що зберігається для різних типів об'єктів, цей підхід може швидко стати непрацездатним.

Замість цього ядро реалізує загальні системні виклики *Query* і *Set* інформація , параметри яких наступні загальному шаблону для всіх типів об'єктів ядра. У лістингу 2.19 показаний шаблон системного виклику Query інформація на прикладі типу *Process*; для інших типів просто замініть *Process* на ім'я типу ядра.

Лістинг 2.19. Приклад системного виклику інформації про запит для типу процесу

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

Усі системні виклики інформації *Query* приймають дескриптор об'єкта в якості першого параметра. Другий параметр, *InformationClass*, описує тип інформації про процес запиту. Клас інформації — це значення, що перераховується. SDK вказує імена класів інформації, які можна вилучити та реалізувати в PowerShell. Для запиту певних видів інформації можуть знадобитися спеціальні привілеї або доступ адміністратора.

Для кожного класу інформації необхідно вказати непрозорий буфер для отримання запитуваної інформації, а також довжину буфера. Системний виклик також повертає значення довжини, яка служить двом цілям: він вказує, яка частина буфера була заповнена, якщо системний виклик був успішним, і якщо системний виклик не відбувся - вказує, наскільки більшим повинен бути буфер за допомогою STATUS_INFO_LENGTH_MISMATCH або STATUS_BUFFER_TOO_SMALL. Однак слід бути обережним, при використанні довжини, що повернулась, для визначення розміру буфера, що передається в запит. Деякі класи та типи інформації неправильно встановлюють необхідну довжину, якщо наданий буфер, занадто малий. Це ускладнює запит даних, тому, що не знаєте заздалегідь їх формат. На жаль, навіть SDK документує точні потрібні розміри дуже рідко. Як показано в лістингу 2.20, виклик *Set information* працює за аналогічним шаблоном. Основні відмінності полягають у

тому, що немає параметра довжини, що повертається, і в цьому випадку буфер є входом для системного виклику, а не виходом.

Лістинг 2.20. Приклад системного виклику *Set* інформація для типу *Process*

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

У модулі PowerShell є можливість на запит імені класів інформації типу за допомогою команди *Get-NtObjectInformationClass*, як показано в лістингу 2.21. Майте на увазі, що деякі імена класів інформації можуть бути відсутні в списку, оскільки Microsoft не завжди їх документує.

Лістинг 2.21. Список класів інформації для типу процесу

```
PS C:\Users\hladk> Get-NtObjectInformationClass Process

Key                                 Value
---                                 -----
ProcessBasicInformation                 0
ProcessQuotaLimits                      1
ProcessIoCounters                       2
ProcessVmCounters                       3
ProcessTimes                            4
ProcessBasePriority                     5
ProcessRaisePriority                    6
ProcessDebugPort                        7
ProcessExceptionPort                    8
ProcessAccessToken                      9
ProcessLdtInformation                  10
ProcessLdtSize                         11
ProcessDefaultHardErrorMode            12
ProcessIoPortHandlers                  13
ProcessPooledUsageAndLimits            14
ProcessWorkingSetWatch                 15
ProcessUserModeIOPL                    16
ProcessEnableAlignmentFaultFixup       17
ProcessPriorityClass                   18
ProcessWx86Information                 19
ProcessHandleCount                     20
ProcessAffinityMask                    21
ProcessPriorityBoost                   22
```

Для системного виклику *Query information* використовуйте *Get-NtObjectInformation*, вказавши відкритий дескриптор об'єкта та клас інформації. Для виклику *Set Information* використовуйте *Set-NtObjectInformation*. У лістингу 2.22 показано приклад того, як використовувати *Get-NtObjectInformation*. Тип Process не встановлює довжину, що повертається, для інформаційного класу *ProcessTimes*, тому, якщо не буде вказано довжину, операція генерує помилку STATUS_BUFFER_TOO_SMALL. Однак, шляхом перевірки або перебору є можливість дізнатися, що довжина даних становить 32 байта. Кількісне визначення цього значення за допомогою параметра *Length* дозволяє запиту успішно виконати та повернути дані у вигляді масиву байтів.

Лістинг 2.22. Запит базової інформації до об’єкта *Process*

```
PS C:\Users\hladk> $proc = Get-NtProcess -Current
PS C:\Users\hladk> Get-NtObjectInformation $proc ProcessTimes
Get-NtObjectInformation : (0xC0000023) - {Буфер слишком мал}
Размер буфера слишком мал для размещения данных. Данные в буфер не записаны.
строка:1 знак:1
+ Get-NtObjectInformation $proc ProcessTimes
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : NotSpecified: (:) [Get-NtObjectInformation], NtException
    + FullyQualifiedErrorId : NtCoreLib.NtException,NtObjectManager.Cmdlets.Object.GetNtObjectInfoCmdlet

PS C:\Users\hladk> Get-NtObjectInformation $proc ProcessTimes -Length 32
170
212
31
241
103
102
219
1
0
0
PS C:\Users\hladk> Get-NtObjectInformation $proc ProcessTimes -AsObject

CreateTime         ExitTime KernelTime UserTime
----------         -------- ---------- --------
133813210550621354 0        5468750    8437500
```

Для багатьох інформаційних класів команда *Get-NtObjectInformation* знає розмір і структуру даних запиту. Якщо вказано параметр *AsObject*, отримаєте попередньо відформатований об’єкт, а не масив байтів.

Крім того, для багатьох інформаційних класів дескриптор об’єкта вже надає властивості та методи для встановлення чи запиту значень. Значення будуть декодовані у форматі використання; наприклад, у лістингу 2.22 час знаходиться у внутрішньому форматі. Властивість *CreationTime* об'єкта створює цей внутрішній формат і перетворює його в зрозумілий людині формат дати і часу.

Маємо можливість легко перевірити властивості, звернувшись за допомогою команди *Format-List*. Наприклад, лістинг 2.23 перераховує всі властивості об'єкта *Process*, а потім запитує відформатований *CreationTime* .

Лістинг 2.23. Запит дескриптора об’єкта для властивостей і перевірка *CreationTime*

```
PS> $proc | Format-List
SessionId      : 2
ProcessId      : 5484
ParentProcessId : 8108
PebAddress     : 46725963776
--snip--
PS> $proc.CreationTime Saturday,
October 24, 17:12:58
```

Класи *QueryInformation* і *SetInformation* для певного типу зазвичай мають однакові перераховані значення.

Ядро може обмежити перераховані значення класу інформації одним типом операції, повертаючи код стану STATUS_INVALID _INFO_CLASS, якщо це недопустиме значення. Для деяких типів, таких як ключі реєстру, клас інформації відрізняється між запитом і налаштуванням, як прописано в лістингу 2.24.

Лістинг 2.24. Перевірка класів *QueryInformation* і *SetInformation* для типу Key

```
PS C:\Users\hladk> Get-NtObjectInformationClass Key

Key                          Value
---                          -----
KeyBasicInformation              0
KeyNodeInformation               1
KeyFullInformation               2
KeyNameInformation               3
KeyCachedInformation             4
KeyFlagsInformation              5
KeyVirtualizationInformation     6
KeyHandleTagsInformation         7
KeyTrustInformation              8
KeyLayerInformation              9


PS C:\Users\hladk> Get-NtObjectInformationClass Key -Set

Key                            Value
---                            -----
KeyWriteTimeInformation            0
KeyWow64FlagsInformation           1
KeyControlFlagsInformation         2
KeySetVirtualizationInformation    3
KeySetDebugInformation             4
KeySetHandleTagsInformation        5
KeySetLayerInformation             6
```

Виклик *Get-NtObjectInformationClass* тільки з іменем типу повертає клас *QueryInformation*. Якщо вказати ім'я типу і параметр *Set*, то отримаємо клас *SetInformation*. Зверніть увагу, що два показані записи мають різні імена і, відповідно, представляють різну інформацію.

2.4. МЕНЕДЖЕР ВВОДУ-ВИВОДУ

Менеджер вводу-виводу (I/O) забезпечує доступ до пристроїв вводу-виводу через драйвери пристроїв. Наприклад, коли відкриваєте документ на своєму комп’ютері, файл стає доступним через драйвер файлової системи. Менеджер вводу-виводу підтримує інші види драйверів для таких пристроїв, як клавіатура або відеокарти.

Можливо самостійно завантажити новий драйвер через системний виклик *NtLoadDriver* або зробити це автоматично за допомогою менеджера *PlugandPlay* (PnP). Для кожного драйвера менеджер вводу-виводу створює запис у каталозі *Driver*. Ви можете переглянути вміст цього каталогу, тільки якщо ви є адміністратором. До речі, як звичайному користувачеві, не потрібно отримувати доступ ~~ні~~ до чогось в каталозі *Driver*.

Натомість є можливість взаємодіяти з драйвером через об’єкт *Device*, який зазвичай створюється в каталозі *Device*.

Лістинг 2.25. Відображення об'єктів *Device*

```
PS C:\Users\hladk> ls NtObject:\Device

Name                                               TypeName
----                                               --------
HDSDEV_{6ac1c144-0bba-4822-b48d-543b752f4ada}      Device
0000006a                                           Device
00000058                                           Device
GPIO_1                                             Device
aswSP_Open                                         Device
AswVmm                                             Device
00000044                                           Device
NTPNP_PCI0002                                      Device
00000030                                           Device
NDMP3                                              Device
0000007a                                           Device
00000068                                           Device
00000054                                           Device
UcmCx0                                             Device
00000040                                           Device
NTPNP_PCI0003                                      Device
{df098b79-7c3a-4a44-890c-eaf78c6eda2a}             SymbolicLink
NDMP4                                              Device
0000008a                                           Device
00000078                                           Device
00000064                                           Device
USBPDO-5                                           Device
MSGpioClassExt0                                    Device
00000050                                           Device
VidExo                                             Device
```

Драйвери відповідають для створення нових об'єктів *Device* за допомогою API *IoCreateDevice*. Драйвер може мати більше одного зв'язку з об'єктом *Device*. Якщо він не вимагає взаємодії з користувачем, він може не мати жодного об’єкту. Як показано в лістингу 2.25, можемо перерахувати склад каталогу *Device* з повноваженнями звичайного користувача через OMNS.

У висновку ми бачимо, що всі імена типів об’єктів — *Device*.

Однак, якщо будете потреба знайти системний виклик з *Device* в імені, ви нічого не знайдете.

Лістинг 2.26. Відкриття пристрою об’єкта та відображення його шляху до нього

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

\Device\HarddiskVolume3\Windows\notepad.exe File
PS> Get-Item NtObject:\Device\HarddiskVolume3
Name           TypeName
----           --------
HarddiskVolume3 Device
```

Це відбувається тому, що не взаємодіємо з менеджером вводу-виводу з виділеними системними викликами, замість цього використовуємо системні виклики об'єкта File, наприклад NtCreateFile. Можемо отримати доступ до цього системного виклику через New-NtFile і Get-NtFile, які створюють і відкривають файли, відповідно, як показано в лістингу 2.26.

В цьому прикладі відкриваємо notepad.exe з каталогу Windows. Символічне посилання *SystemRoot* вказує на каталог Windows на системному диску. Якщо символічне посилання *SystemRoot* є частиною OMNS, OMNS спочатку обробляє доступ до файлу. З відкритим дескриптором можемо вибрати повний шлях до файлу та ім’я типу.

Поглянувши результат, бачимо, що повний шлях починається з *Device\HarddiskVolume3*, за яким йде *Windows\notepad.exe*. Якщо спробуємо відобразити пристрій, то-дізнаємося, що він має тип *Device*. Тільки но диспетчер об'єктів знаходить об'єкт *Device*, він передає відповідальність за решту частини

шляху диспетчера вводу-виводу, який викликає відповідний метод всередині драйвера ядра.

Перерахувати драйвери, завантажені в ядро, можемо за допомогою команди *Get-NtKernelModule* (лістинг 2.27).

На відміну від інших операційних систем, таких як Linux, Windows не реалізує основні мережеві протоколи, такі як протоколи із стеку TCP/IP, за допомогою вбудованих системних викликів.

Лістинг 2.27. Перерахунок всіх завантажених драйверів ядра

```
PS C:\Users\hladk> Get-NtKernelModule

Name                 ImageBase        ImageSize
----                 ---------        ---------
ntoskrnl.exe         0000000000000000 21295104
hal.dll              0000000000000000 24576
kd.dll               0000000000000000 45056
symcryptk.dll        0000000000000000 45056
cng.sys              0000000000000000 864256
CLFS.SYS             0000000000000000 548864
tm.sys               0000000000000000 176128
winaccel.sys         0000000000000000 86016
PSHED.dll            0000000000000000 110592
BOOTVID.dll          0000000000000000 53248
FLTMGR.SYS           0000000000000000 634880
msrpc.sys            0000000000000000 401408
ksecdd.sys           0000000000000000 180224
clipsp.sys           0000000000000000 1052672
cmimcext.sys         0000000000000000 61440
werkernel.sys        0000000000000000 90112
ntosext.sys          0000000000000000 53248
CI.dll               0000000000000000 1064960
globmerger.sys       0000000000000000 126976
Wdf01000.sys         0000000000000000 978944
WppRecorder.sys      0000000000000000 77824
WDFLDR.SYS           0000000000000000 94208
PRM.sys              0000000000000000 61440
acpiex.sys           0000000000000000 172032
```

Натомість в Windows є драйвер диспетчера вводу-виводу, драйвер додаткових функцій (AFD), який забезпечує доступ до мережевих служб та програм. Не потрібно мати справу з драйвером напряму, Win32 надає API у стилі сокетів BSD, званий WinSock , для керування доступом до нього.

На додачу до стандартного набору інтернет-протоколів, AFD також реалізує інші типи мережевих сокетів, такі як сокети Unix і спеціальні сокети Hyper-V для зв'язку з віртуальними машинами.

Звернімося до іншої важливої підсистеми, диспетчеру процесів і потоків.

2.5. ДИСПЕТЧЕР ПРОЦЕСІВ І ПОТОКІВ

Весь код режиму користувача живе в контексті процесу, кожен з яких має один або декілька потоків, які керують виконанням коду. Процеси і потоки захищаються ресурсами. Це має сенс: якщо ви могли б отримати доступ до процесу, ви могли б змінити свій код і додати його в контекст іншого користувача. Таким чином, на відміну від більшості інших об'єктів ядра неможливо відкрити процес або потік по імені. Натомість необхідно відкрити їх через унікальний числовий ідентифікатор процесу (PID) або ідентифікатор потоку (TID).

Щоб отримати список запущених процесів і потоків, можна перебрати простір ідентифікаторів, викликаючи відкритий системний виклик з усіма

можливими ідентифікаторами, але це займає інший час. На разі системний виклик *NtQuerySystemInformation* надає інформаційний клас *SystemProcessInformation* , який дозволяє перераховувати процеси та потоки, які не мають доступу до об'єкта *Process*.

Отримати доступ до списку процесів і потоків маємо можливість, використовуючи команди *Get-NtProcess* і *Get-NtThread* та передавши їм параметр *InfoOnly*, як показано в лістингу 2.28. Також можемо використовувати вбудовану команду *Get-Process* для отримання аналогічного висновку. Кожен із об'єктів що повертаються, мають властивість *Threads*, яку можемо запитувати для отримання інформації про потоки.

Лістинг 2.28. Відображення процесів і потоків, які не мають високих привілей.

```
PS C:\Users\hladk> Get-NtProcess -InfoOnly

PID   PPID  Name                           SessionId
---   ----  ----                           ---------
0     0     Idle                           0
4     0     System                         0
236   4     Secure System                  0
280   4     Registry                       0
784   4     smss.exe                       0
1144  816   csrss.exe                      0
1228  816   wininit.exe                    0
1304  1228  services.exe                   0
1312  1228  LsaIso.exe                     0
1332  1228  lsass.exe                      0
1456  1304  svchost.exe                    0
1488  1228  fontdrvhost.exe                0
1524  1304  WUDFHost.exe                   0
1588  1304  svchost.exe                    0
1640  1304  svchost.exe                    0
1684  1304  WUDFHost.exe                   0
1740  1304  WUDFHost.exe                   0
1796  1304  WUDFHost.exe                   0
2020  1304  svchost.exe                    0
2028  1304  svchost.exe                    0
2044  1304  svchost.exe                    0
1636  1304  svchost.exe                    0
1240  1304  svchost.exe                    0
2084  1304  svchost.exe                    0
2092  1304  svchost.exe                    0
2196  1304  svchost.exe                    0
PS C:\Users\hladk> Get-NtThread -InfoOnly

TID   PID   ProcessName                    StartAddress
---   ---   -----------                    ------------
0     0     Idle                           00000000
32    0     Idle                           00000000
36    0     Idle                           00000000
0     0     Idle                           00000000
40    0     Idle                           00000000
44    0     Idle                           00000000
0     0     Idle                           00000000
48    0     Idle                           00000000
52    0     Idle                           00000000
0     0     Idle                           00000000
56    0     Idle                           00000000
60    0     Idle                           00000000
0     0     Idle                           00000000
64    0     Idle                           00000000
68    0     Idle                           00000000
0     0     Idle                           00000000
72    0     Idle                           00000000
76    0     Idle                           00000000
0     0     Idle                           00000000
80    0     Idle                           00000000
84    0     Idle                           00000000
0     0     Idle                           00000000
88    0     Idle                           00000000
92    0     Idle                           00000000
```

Перші два процеси, перераховані у виведенні, є особистими. Перший — це процес *Idle* з PID 0. Цей процес містить потоки, які виконуються, коли операційна система не виконує жодних дій. Процес *System* з PID 4 важливий,

оскільки він повністю працює в режимі ядра. Коли ядру або драйверу необхідно виконати фоновий потік, потік має справу з процесом *System*.

Щоб відкрити процес або потік, можемо передати *Get-NtProcess* або *Get-NtThread* PID або TID, в залежності від того, які ми хочемо відкрити. Команда верне об’єкт *Process* або *Thread*, з якими ми потім зможемо взаємодіяти. Наприклад, у лістингу 2.29 показано, яким чином запитувати командний рядок та шлях до виконуваного файлу поточного процесу.

Лістинг 2.29. Відкриття поточного процесу за його ідентифікатор процесу

```
PS C:\Users\hladk> $proc = Get-NtProcess -ProcessId $pid
PS C:\Users\hladk> $proc.CommandLine
"C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe"
PS C:\Users\hladk> $proc.Win32ImagePath
C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
```

Під час відкриття об'єкту *Process* або *Thread* , використовуючи його ID, отримуєте дескриптор. Для зручності ядро також підтримує два псевдо дескриптора, які посилаються на поточний процес і поточний потік. Псевдо дескриптор поточного процесу — це значення -1, перетворене в дескриптор, а для поточного потоку — це -2. Можливо отримати доступ до цих псевдо дескрипторів , передавши параметр *Current* замість ID команд *Get-NtProcess* і *Get -NtThread*.

Зверніть увагу, що безпека процесу та його потоків незалежна. Якщо знаєте ідентифікатор потоку, то можете отримати доступ до дескриптора потоку всередині процесу, навіть якщо не маєте можливості отримати доступ до самого процесу.

2.6. ДИСПЕТЧЕР ПАМ'ЯТІ

Кожен процес має власний адресний простір віртуальної пам'яті, який розробник може використовувати на свій розсуд. 32-розрядний процес може отримати доступ до 2 ГБ адресного простору віртуальної пам'яті, в той час, як 64-розрядний процес може отримати доступ до 128 ТБ. Підсистема диспетчера пам'яті ядра керує розподілом цього адресного простору.

Навряд чи на комп'ютері буде 128 ТБ фізичної пам'яті, але диспетчер пам'яті може зробити так, щоб здавалось, що мається більше фізичної пам'яті, ніж насправді.

Наприклад, диспетчер пам'яті може використовувати файл у файловій системі, так званий файлом підкачки, для тимчасового зберігання пам’яті. Оскільки доступний простір зберігання файлів системи набагато більше фізичної пам’яті комп’ютера, це може створити видимість більшого об’єму пам’яті.

Простір віртуальної пам'яті одночасно використовується виділенням пам'яті та зберігає стан виконання кожного процесу, а також його коду, що виконується. Кожне виділення пам’яті може передбачати низку захистів, таких як *ReadOnly* або *ReadWrite*, які повинні бути встановлені відповідно до призначення. Наприклад, для виконання коду пам'яті має бути стан захисту *ExecuteRead* або *ExecuteReadWrite* .

Створити запит на всю інформацію про стан пам'яті для процесу можна скориставшись *NtQueryVirtualMemory*, якщо маєте право доступу *QueryLimitedInformation* до дескриптора процесу. Однак для зчитування і запису даних пам'яті потрібні права доступу *VmRead* і *VmWrite* відповідно, а також виклик *NtReadVirtualMemory* і *NtWriteVirtualMemory* .

Можливо виділити нову пам'ять і звільнити пам'ять в процесі за допомогою *NtAllocateVirtualMemory* і *NtFreeVirtualMemory*, які вимагають права доступу до *VmOperation*. Нарешті, з'являється можливість змінити захист пам'яті за допомогою *NtProtectVirtualMemory*, що також вимагає доступу *VmOperation*.

2.6.1. Команди NtVirtualMemory

PowerShell обробляє ~~ці~~ системні виклики за допомогою *команд Get-, Add-, Read -, Write -, Remove - і Set-NtVirtualMemory*. Зверніть увагу, що ці команди приймають необов'язковий параметр *Process*, який дозволяє отримувати доступ до пам'яті в іншому процесі, який не є поточним. У лістингу 2.30 показані команди в дії.

Лістинг 2.30. Виконання різних операцій з пам'яттю в процесі

```
PS C:\Users\hladk> Get-NtVirtualMemory

Address          Size          Protect          Type    State   Name
-------          ----          -------          ----    -----   ----
000000007FFE0000 4096          ReadOnly         Private Commit
000000007FFEE000 4096          ReadOnly         Private Commit
000000313FBC0000 241664        None             Private Reserve
000000313FBFB000 12288         ReadWrite, Guard Private Commit
000000313FBFE000 8192          ReadWrite        Private Commit
000000313FC00000 1712128       None             Private Reserve
000000313FDA2000 12288         ReadWrite        Private Commit
000000313FDA5000 49152         None             Private Reserve
000000313FDB1000 16384         ReadWrite        Private Commit
000000313FDB5000 49152         None             Private Reserve
000000313FDC1000 8192          ReadWrite        Private Commit
000000313FDC3000 8192          None             Private Reserve
000000313FDC5000 16384         ReadWrite        Private Commit
000000313FDC9000 49152         None             Private Reserve
000000313FDD5000 24576         ReadWrite        Private Commit
000000313FDDB000 40960         None             Private Reserve
000000313FDE5000 49152         ReadWrite        Private Commit
000000313FDF1000 61440         None             Private Reserve
000000313FE00000 450560        None             Private Reserve
000000313FE6E000 20480         ReadWrite, Guard Private Commit
```

```
PS C:\Users\hladk> $addr = Add-NtVirtualMemory -Size 1000 -Protection ReadWrite
PS C:\Users\hladk> Get-NtVirtualMemory -Address $addr

Address          Size Protect   Type    State  Name
-------          ---- -------   ----    -----  ----
000001D7ED5D0000 4096 ReadWrite Private Commit


PS C:\Users\hladk>  Read-NtVirtualMemory -Address $addr -Size 4 | Out-HexDump
00 00 00 00

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

PS C:\Users\hladk> Set-NtVirtualMemory -Address $addr -Protection ExecuteRead -Size 4
ReadWrite
PS C:\Users\hladk> Get-NtVirtualMemory -Address $addr

Address          Size Protect     Type    State  Name
-------          ---- -------     ----    -----  ----
000001D7ED5D0000 4096 ExecuteRead Private Commit
```

```
PS C:\Users\hladk> Remove-NtVirtualMemory -Address $addr
PS C:\Users\hladk> Get-NtVirtualMemory -Address $addr

Address          Size Protect   Type   State  Name
-------          ---- -------   ----   -----  ----
000001D7ED5D0000 4096 ReadWrite Mapped Commit
```

В цьому лістингу виконуємо кілька операцій. В першу чергу використовуємо *Get-NtVirtualMemory* для перерахування всіх областей пам’яті, що використовуються поточним процесом. Створений список буде більшим, але частина лістинга, показана в прикладі, повинна дати орієнтоване уявлення про те, яким чином представлена інформація. Він (список) включає адресу області пам'яті, її розмір, її захист і її стан. Існує три можливі значення стану пам’яті:

Commit - Вказує, що область віртуальної пам'яті виділена та доступна для використання.

Reserve - Вказує, що область віртуальної пам'яті виділена, але в даний час немає резервної пам'яті. Використання зарезервованої області пам'яті приведе до певних проблем.

Free - Вказує, що область віртуальної пам'яті не використовується. Використання вільної області пам'яті також може викликати проблеми.

Може виникнути питання, в чому полягає різниця між *Reserve* і *Free*, якщо використання як зарезервованої, так і вільної області пам'яті приведе до проблем. Стан *Reserve* дозволяє ~~вам~~ зарезервувати область віртуальної пам'яті для того, щоб використання нічого іншого не могло запозичити пам'ять в цьому діапазоні адресу пам'яті. Згодом буде можливо сформувати стан *Reserve* у *Commit*, знову викликавши *NtAllocateVirtualMemory*. Стан *Free* вказує на частини пам’яті, які вільно доступні для виділення.

Далі виділяємо 1000-байтову область читання/запису та зберігаємо адресу в змінній. Передача адреси в *Get-NtVirtualMemory* дозволяє запитувати лише цю, конкретну, область віртуальної пам'яті. Хоча запит був на 1000-байтову область, розмір повернутої області становить 4096 байт. Це пов'язано з тим, що в Windows існує мінімальний розмір виділення в системі користувача - мінімум становить 4096 байт. Тому неможливо виділити менший обсяг пам’яті. З цієї причини ці системні виклики не особливо корисні для загальних програм. Найімовірніше, вони є примітивами, на яких побудовані менеджери пам'яті «кучі», такі як *malloc* з бібліотеки C.

Далі відбувається зчитування і запис у виділену область пам'яті. В першу чергу використовуємо *Read-NtVirtualMemory* для розрахунку 4 байтів області пам'яті та виявляємо, що всі байти дорівнюють нулю. Згодом записуємо байти 1, 2, 3 і 4 в області пам'яті за допомогою *Write-NtVirtualMemory*. Ми зчитуємо байти, щоб підтвердити, що запис операції пройшов успішно. Два значення повинні співпадати, як показано в лістингу.

З виділеною пам'яттю змінюємо захист за допомогою *Set-NtVirtualMemory*. У цьому випадку вказуємо захист виділеної пам'яті, що виконується, як ExecuteRead. Запит поточного стану області пам'яті за допомогою команди *Get-NtVirtualMemory* показує, що захист змінився з *ReadWrite* на *ExecuteRead* . Також зверніть увагу, що, хоча ~~ми~~ просили змінити

захист лише на 4 байти, вся область об’ємом 4096 байт тепер є така, що виконується. Це відбувається через мінімальний розмір виділеної пам'яті.

Потім звільняємо пам'ять за допомогою *Remove-NtVirtualMemory* і перевіряємо, чи знаходиться наразі пам'ять в стані *Free*. Пам'ять, виділена за допомогою *NtAllocate VirtualMemory*, вважається приватною, відповідно, на це вказує значення властивості *Type*, як показано в лістингу 2.30.

2.6.2. Об'єкти Section

Інший спосіб розподілу віртуальної пам'яті — через об'єкти Section. Об'єкт Section — це тип ядра, що реалізує файли, відображені в пам'яті. Об'єкти Section можна використовувати для двох пов'язаних цілей:

— Читання або запис файлу так, ніби він повністю завантажений у пам'ять;

— Спільне використання пам'яті між процесами, щоб зміни в одному процесі відображалися в іншому.

Створюємо об'єкт *Section* за допомогою системного виклику *NtCreateSection* або команди PowerShell *New-NtSection*. Необхідно вказати розмір відображення, захист пам'яті, необов'язковий дескриптор файлу та отримати дескриптор розділу.

Однак таке створення не дозволяє нам отримати автоматичний доступ до пам'яті. Спочатку нам потрібно відобразити його в адресному просторі віртуальної пам'яті за допомогою *NtMapViewOfSection* або *Add-NtSection*.

У лістингу 2.31 наведено приклад, в якому створено анонімний розділ і відображаємо його в пам’яті.

Для початку створюємо об'єкт Section захищений ReadWrite розміром 4096 байт. Не вказуємо параметр File, який означає, що він анонімний і не підтримується будь-яким файлом. Якщо додати об’єкту Section шлях OMNS, анонімна пам’ять, яку він представляє, могла б використовуватися спільно з іншими процесами.

Лістинг 2.31. Створення розділу та відображення його в пам’яті

```
PS C:\Users\hladk> $s = New-NtSection -Size 4096 -Protection ReadWrite
PS C:\Users\hladk> $m = Add-NtSection -Section $s -Protection ReadWrite
PS C:\Users\hladk> Get-NtVirtualMemory $m.BaseAddress

Address          Size Protect   Type   State  Name
-------          ---- -------   ----   -----  ----
000001BEFB670000 4096 ReadWrite Mapped Commit


PS C:\Users\hladk> Remove-NtSection -Mapping $m
PS C:\Users\hladk> Get-NtVirtualMemory -Address 0x1C3DD0E0000

Address          Size            Protect  Type State Name
-------          ----            -------  ---- ----- ----
000001C3DD0E0000 136549697323008 NoAccess None Free


PS C:\Users\hladk> Add-NtSection -Section $s -Protection ExecuteRead
Исключение при вызове "Map" с "9" аргументами: "(0xC000004E) - Представление секции памяти использует режим защиты,
несовместимый с режимом защиты исходного представления."
C:\Users\hladk\Documents\WindowsPowerShell\Modules\NtObjectManager\2.0.1\NtObjectManager.psm1:8565 знак:5
+     $Section.Map($Process, $Protection, $ViewSize, $BaseAddress, `
+     ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : NotSpecified: (:) [], MethodInvocationException
    + FullyQualifiedErrorId : NtException
```

Потім відображаємо його у пам’яті за допомогою *Add-NtSection*, вказуючи необхідний захист для пам’яті, та запитуємо відображену адресу, щоб переконатися, що операція пройшла успішно. Зверніть увагу, що *Type* встановлений у *Mapped* .

Коли закінчимо, ми викликаємо *Remove-NtSection*, щоб скасувати відображення розділу, а потім перевіряємо, чи він тепер вільний.

Нарешті, демонструємо, що не можна відобразити розділ із захистом, відмінним від того, що було надано під час створення об'єкта Section. Коли намагаємося відобразити розділ із правами на читання та виконання, які несумісні, ми бачимо виняток.

Захист, який дозволено використовувати для відображення об'єкта *Section* у пам'яті залежить від двох факторів.

Перший — це захист, визначений під час створення об'єкта *Section*. Наприклад, якщо розділ було створено із захистом *ReadOnly*, ви ніколи не зможете призначити йому статус записуваного.

Друга залежність — це доступ, наданий дескриптору секції, яку ви відображаєте. Якщо ви хочете відобразити секцію як доступну для читання, дескриптор повинен мати доступ *MapRead*. Щоб відобразити її як доступну для запису, вам потрібен доступ як *MapRead*, так і *MapWrite*. (І, звичайно, наявність лише доступу *MapWrite* не є достатньою для відображення секції як доступної для запису, якщо оригінальний об'єкт *Section* не був вказаний із захистом від запису.)

Можна відобразити розділ в іншому процесі, вказавши дескриптор процесу в *Add-NtSection*. Не потрібно вказувати процес в *Remove-NtSection,* оскільки об'єкт відображення знає, в якому процесі він був відображений. У вихідних даних про пам'ять у стовпці «Ім'я» буде вказано ім'я файлу резервної копії, якщо вона існує.

Створений розділ був анонімним, тому ми не бачимо нічого в стовпці *Name*, але ми можемо виконати запит, щоб знайти розділи, які підкріплені файлами, використовуючи команду, показану в лістинг 2.32.

У доповненні до типів *Anonymous* і *Mapped* існує третій тип розділу - тип *Image*. При наданні дескриптора файлу для файлу, що виконується, Windows ядро автоматично проаналізує формат і створює кілька підрозділів, які представляють різні компоненти файлу, що виконуються. Щоб створити відображуваний образ із файлу, потрібно мати лише доступ *Execute* до дескриптора файлу. Файл не обов'язково повинен бути доступний для читання.

Лістинг 2.32. Список спільно поставлених файлів з іменами

```
PS C:\Users\hladk> Get-NtVirtualMemory -Type Mapped | Where-Object Name -ne ""

Address          Size    Protect  Type   State  Name
-------          ----    -------  ----   -----  ----
000001BEF8680000 69632   ReadOnly Mapped Commit C_1251.NLS
000001BEF86A0000 69632   ReadOnly Mapped Commit C_866.NLS
000001BEF86C0000 12288   ReadOnly Mapped Commit l_intl.nls
000001BEF8700000 12288   ReadOnly Mapped Commit l_intl.nls
000001BEF8710000 69632   ReadOnly Mapped Commit C_1251.NLS
000001BEF88B0000 864256  ReadOnly Mapped Commit locale.nls
000001BEF8990000 69632   ReadOnly Mapped Commit C_866.NLS
000001BEF8A20000 24576   ReadOnly Mapped Commit powershell.exe.mui
000001BEFA300000 16384   ReadOnly Mapped Commit cversions.2.db
000001BEFA310000 307200  ReadOnly Mapped Commit {6AF0698E-D558-4F6E-9B3C-3716689AF493}.2.ver0x0000000000000001.db
000001BEFA360000 16384   ReadOnly Mapped Commit cversions.2.db
000001BEFA370000 86016   ReadOnly Mapped Commit propsys.dll.mui
000001BEFA3A0000 118784  ReadOnly Mapped Commit {AFBF9F1A-8EE8-4C77-AF34-C647E37CA0D9}.1.ver0x000000000000000f.db
000001BEFA3D0000 655360  ReadOnly Mapped Commit {DDF571F2-BE98-426D-8288-1A9A39C3FDA2}.2.ver0x0000000000000001.db
000001BEFA4A0000 3387392 ReadOnly Mapped Commit SortDefault.nls
000001BEFA880000 69632   ReadOnly Mapped Commit C_1252.NLS
000001BEFA8B0000 28672   ReadOnly Mapped Commit winnlsres.dll
000001BEFAA30000 86016   ReadOnly Mapped Commit winnlsres.dll.mui
000001BEFAA50000 57344   ReadOnly Mapped Commit crypt32.dll.mui
000001BEFAA90000 434176  ReadOnly Mapped Commit mscorrc.dll
000001BEFAB20000 16384   ReadOnly Mapped Commit mpr.dll.mui
000001BEFB260000 1429504 ReadOnly Mapped Commit KernelBase.dll.mui
000001BEFB6E0000 528384  ReadOnly Mapped Commit ntdll.dll.mui
```

Windows широко використовує розділи образу для спрощення відображення виконуваних файлів у пам'яті. Ми можемо вказати розділ образу, передавши прапор Image під час створення об'єкта Section або за допомогою команди *New-NtSectionImage*, як показано в лістингу 2.33.

Лістинг 2.33. Відображення notepad.exe та перегляд завантаженого образу

```
PS C:\Users\hladk> $sect = New-NtSectionImage -Win32Path "C:\Windows\notepad.exe"
PS C:\Users\hladk> $map = Add-NtSection -Section $sect -Protection ReadOnly
PS C:\Users\hladk> Get-NtVirtualMemory -Address $map.BaseAddress

Address          Size Protect  Type  State  Name
-------          ---- -------  ----  -----  ----
00007FF7944F0000 4096 ReadOnly Image Commit notepad.exe

PS C:\Users\hladk> Get-NtVirtualMemory -Type Image -Name "notepad.exe"

Address          Size   Protect     Type  State   Name
-------          ----   -------     ----  -----   ----
00007FF7944F0000 4096   ReadOnly    Image Commit  notepad.exe
00007FF7944F1000 163840 ExecuteRead Image Commit  notepad.exe
00007FF794519000 45056  ReadOnly    Image Commit  notepad.exe
00007FF794524000 12288  WriteCopy   Image Commit  notepad.exe
00007FF794527000 8192   ReadOnly    Image Commit  notepad.exe
00007FF794529000 4096   WriteCopy   Image Commit  notepad.exe
00007FF79452A000 131072 ReadOnly    Image Commit  notepad.exe
00007FF79454A000 4096   ExecuteRead Image Commit  notepad.exe
00007FF79454B000 8192   None        Image Reserve notepad.exe


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

Як бачите, не потрібно вказувати захист *ExecuteRead* або *ExecuteReadWrite* при відображенні розділу зображення. Будь-який захист, включно *ReadOnly*, буде працювати. Отримавши інформацію про пам’ять для адреси на базі карти, бачимо, що там немає повної пам’яті, і що відведена пам’ять становить усього 4096 байт, що здається занадто малим для notepad.exe. Це пов'язано з тим, що розділ складається з кількох менших відображених областей. Якщо відфільтруємо інформацію про пам'ять для відображуваного імені, побачимо пам'ять що використовується. Застосувавши команду *Out-HexDump*, можемо роздрукувати файл, що містить відображуваний буфер файлу.

2.7. ЦІЛІСНІСТЬ КОДУ

Одним з важливих завдань безпеки є забезпечення того, щоб код, який виконується на вашому комп'ютері, був тим самим кодом, який розробник планував для виконання. Якщо зловмисний користувач змінив файли операційної системи, ви можете зіткнутися з проблемами безпеки, такими як витік приватних даних.

Microsoft вважає повноцінність коду, що працює в Windows, настільки важливим, що для її захисту системних файлів існує ціла підсистема. Ця підсистема перевіряє і обмежує цілісність коду, які файли можуть виконуватися в ядрі та при необхідності, в режимі користувача, перевіряючи цілісність коду. Диспетчер пам'яті може проконсультуватися з підсистемою при завантаженні образу файлу, якщо йому потрібно перевірити, чи правильно підписано виконуваний файл.

Майже кожен файл, що виконується при стандартній установці Windows, створений за допомогою механізму, який називається *Authenticode*. Цей механізм дозволяє вставити криптографічний підпис у виконуваний файл або всередині каталогу файлів. Підсистема повноцінності коду може розраховувати цю підписку, перевіряти її дійсність і приймати на її основі рішення про довіру.

Використовуємо команду *Get-AuthenticodeSignature* для запиту статусу підпису файлу, як показано в лістингу 2.34.

Сформувавши запит підпису статусу виконуваного файлу *notepad.exe*, формується висновок команди у вигляді списку. Виведення інформації починається з інформації про сертифікат підписника X.509. В даному випаду, маємо лише ім’я суб’єкта, яке вказує на те, що цей файл підписаний Microsoft.

Далі йде статус підпису. В нашому випадку статус вказує, що файл дійсний і що підпис перевірено. Можливо, є підписаний файл, підпис якого недійсний, наприклад, коли сертифікат було відкликано. В цьому випадку статус, швидше за все, покаже помилку, наприклад *NotSigned* .

Властивість *SignatureType* показує, що цей підпис був заснований на файлі каталогу, а не був вбудований у файл. Можемо бачити, що цей файл є подвійним файлом операційної системи, яка визначається інформацією, вбудованою в підпис.

Лістинг 2.34. Відображення підписки *Authenticode* для драйвера ядра

```
PS C:\Users\hladk> Get-AuthenticodeSignature "$env:WinDir\system32\notepad.exe" | Format-List


SignerCertificate      : [Subject]
                           CN=Microsoft Windows, O=Microsoft Corporation, L=Redmond, S=Washington, C=US

                         [Issuer]
                           CN=Microsoft Windows Production PCA 2011, O=Microsoft Corporation, L=Redmond, S=Washington,
                         C=US

                         [Serial Number]
                           33000004A882E6B8AC1C5D5FF00000000004A8

                         [Not Before]
                           9/12/2024 11:04:07 PM

                         [Not After]
                           9/11/2025 11:04:07 PM

                         [Thumbprint]
                           B2732A60F9D0E554F756D87E7446A20F216B4F73

TimeStamperCertificate : [Subject]
                           CN=Microsoft Time-Stamp Service, OU=nShield TSS ESN:8603-05E0-D947, OU=Microsoft America
                         Operations, O=Microsoft Corporation, L=Redmond, S=Washington, C=US

                         [Issuer]
                           CN=Microsoft Time-Stamp PCA 2010, O=Microsoft Corporation, L=Redmond, S=Washington, C=US

                         [Serial Number]
                           33000001F1B345F527E8C016D60001000001F1

                         [Not Before]
                           12/6/2023 8:45:55 PM

                         [Not After]
                           3/5/2025 8:45:55 PM

                         [Thumbprint]
                           FB9FB064E20B0CDAD6D4FE2F8CDC1B519CB8F4F7

Status                 : Valid
```

Найпоширенішим рішенням щодо довіри, яке приймає підсистема цілісності коду, є перевірка можливості завантаження драйвера ядра. Кожен файл драйвера повинен мати підпис, який підтверджує його довіру за допомогою ключа, виданого компанією Microsoft. Якщо підпис є недійсним або не походить від ключа, виданого компанією Microsoft, ядро може заблокувати завантаження драйвера, щоб зберегти цілісність системи.

2.8. РОЗШИРЕНИЙ ЛОКАЛЬНИЙ ВИКЛИК

Підсистема розширеного виклику локальних процедур (ALPC) реалізує локальну між процесну комунікацію. Щоб використовувати ALPC, спочатку потрібно створити серверний порт ALPC за допомогою системного виклику *NtCreateAlpcPort* і вказати його ім'я в OMNS. Потім клієнт може використовувати це ім'я, виконавши системний виклик *NtConnectAlpcPort*, щоб підключитися до серверного порту.

На базовому рівні порт ALPC забезпечує безпечну передачу дискретних повідомлень між сервером і клієнтом. ALPC забезпечує базовий механізм передачі для локальних API викликів віддалених процедур, реалізованих у Windows.

2.9. ДИСПЕТЧЕР КОНФІГУРАЦІЇ

Диспетчер конфігурацій, більш відомий як *реєстр*, є важливим компонентом для налаштувань операційної системи. Він зберігає різноманітну інформацію про конфігурації, починаючи з критично важливого для системи списку доступних драйверів пристроїв диспетчера вводу-виводу до (менш важливої) останньої позиції на екрані вікна текстового редактора.

Є можливість створити реєстр як файлову систему, в якій ключі подібні папкам, а значення подібні файлам Отримати до нього доступ можна через OMNS, хоча необхідно використовувати системні виклики, специфічні для реєстру. Корінь реєстру — це шлях OMNS REGISTRY. Перегляд реєстру у PowerShell відбувається з використанням диску *NtObject* , як показано в лістингу 2.35.

Лістинг 2.35. Перелік кореневого ключа реєстру

```
PS C:\Users\hladk> ls NtObject:\REGISTRY

Name    TypeName
----    --------
A       Key
MACHINE Key
USER    Key
WC      Key
```

За потреби можна замінити *NtObject :\REGISTRY* у лістингу 2.35 на *NtKey:*, щоб спростити доступ до реєстру.

Ядро попередньо створює чотири ключі, під час ініціалізації.

Кожен з ключів є спеціальною точкою приєднання, до якої можна приєднати *реєстр*. Реєстр — це ієрархія об'єктів ключів під одним кореневим ключем.

Адміністратор може завантажити нові реєстри з файлу та приєднати їх до цих існуючих ключів.

Зверніть увагу, що PowerShell уже постачається з дисками, які можете використовувати для доступу до реєстру. Однак цей постачальник дисків надає лише представлення реєстру Win32, яке приховує внутрішні відомості про реєстр від представлення. Представлений реєстр Win32 розглянемо окремо, згодом.

Є можливість взаємодіяти з реєстром безпосередньо, використовуючи команди *Get-NtKey* і *New-NtKey* для відкриття та створення ключа об'єктів, відповідно. ~~Ви~~ Також можна використовувати *Get-NtKeyValue* і *Set-NtKeyValue* для отримання та встановлення значень ключів. Щоб видалити ключі чи значення, скористайтеся *Remove-NtKey* або *Remove-NtKeyValue*. У лістингу 2.36 показано деякі з цих команд у дії.

Лістинг 2.36. Відкриття розділу реєстру та запит його значень

```
PS C:\Users\hladk> $key = Get-NtKey \Registry\Machine\SOFTWARE\Microsoft\.NETFramework
PS C:\Users\hladk> Get-NtKeyValue -Key $key

Name        Type   DataObject
----        ----   ----------
Enable64Bit Dword  1
InstallRoot String C:\Windows\Microsoft.NET\Framework64\
UseRyuJIT   Dword  1
```

Ключ об'єкта відкриваємо за допомогою команди *Get-NtKey*. За допомогою команди *Get-NtKeyValue* формується запит про значення, які зберігаються в об’єкті Key. Кожен запис у виведенні показує ім'я значень, тип даних що зберігається.

2.10. ПРАКТИЧНІ ПРИКЛАДИ

Щоб заохотити експерименти, кожна глава завершується набором практичних прикладів, в яких використовуються різні команди, які ви вивчили.

У цих прикладах наведемо випадки, коли за допомогою цього інструменту виявляються вразливості безпеки. Це дасть вам чітке уявлення про те, на що слід звертати увагу в програмах Microsoft або сторонніх розробників, якщо ви хочете стати фахівцем з безпеки.

2.10.1. Пошук відкритих дескрипторів по імені

Об’єкти, що повертаються командою *Get-NtHandle*, мають додаткові властивості, які дозволяють запитувати ім’я об’єкта та дескриптор безпеки. Ці властивості не відображаються за замовчуванням, оскільки їх пошук вимагає великих витрат. Для цього потрібно відкрити процес, що містить дескриптор для доступу *DupHandle*, дублювати дескриптор навпаки у екземплярі PowerShell та запитати властивість.

Якщо продуктивність не важлива, скористайтеся кодом у лістингу 2.37, щоб знайти всі відкриті файли, відповідному-імені файлу.

Лістинг 2.37. Файл пошуку дескрипторів об'єктів, визначене певним ім'ям

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

ProcessId Handle Name
--------- ------ ----
    17648     80 \Device\HarddiskVolume3\Windows\System32\DriverStore\FileRepository\ipf_cpu.inf_amd64_1d0f5f1cb16186f0
    19496     84 \Device\HarddiskVolume3\Windows\System32
    19496   2108 \Device\HarddiskVolume3\Windows\SystemResources\shell32.dll.mun
    19496   2328 \Device\HarddiskVolume3\Windows\System32\ru-RU\propsys.dll.mui
    19496   3084 \Device\HarddiskVolume3\Windows\WinSxS\amd64_microsoft.windows.common-controls_6595b64144ccf1df_6.0...
    19496   3132 \Device\HarddiskVolume3\Windows\System32\ru-RU\KernelBase.dll.mui
    19496   3228 \Device\HarddiskVolume3\Users\hladk\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup
    19496   3244 \Device\HarddiskVolume3\Program Files\WindowsApps\Microsoft.LanguageExperiencePackru-RU_26100.25.43...
    19496   3272 \Device\HarddiskVolume3\Windows\System32\ru-RU\shell32.dll.mui
    19496   3344 \Device\HarddiskVolume3\Program Files\WindowsApps\Microsoft.LanguageExperiencePackru-RU_26100.25.43...
    19496   3624 \Device\HarddiskVolume3\ProgramData\Microsoft\Windows\Start Menu\Programs\Startup
    19496   3856 \Device\HarddiskVolume3\Program Files\WindowsApps\Microsoft.LanguageExperiencePackru-RU_26100.25.43...
    18380     88 \Device\HarddiskVolume3\Windows\System32
    18380    356 \Device\HarddiskVolume3\Program Files\WindowsApps\Microsoft.LanguageExperiencePackru-RU_26100.25.43...
    18380    996 \Device\HarddiskVolume3\Program Files\WindowsApps\Microsoft.LanguageExperiencePackru-RU_26100.25.43...
    18380   1940 \Device\HarddiskVolume3\Program Files\WindowsApps\Microsoft.LanguageExperiencePackru-RU_26100.25.43...
    18380   2044 \Device\HarddiskVolume3\Program Files\WindowsApps\Microsoft.LanguageExperiencePackru-RU_26100.25.43...
    19180     88 \Device\HarddiskVolume3\Windows\System32
    19180    356 \Device\HarddiskVolume3\Program Files\WindowsApps\Microsoft.LanguageExperiencePackru-RU_26100.25.43...
    17756     80 \Device\HarddiskVolume3\Windows\System32
```

Розглянутий скрипт запитує всі дескриптори об'єктів *File* і фільтру їх. Залишає лише ті, у властивості *Name* яких є рядок Windows, що представляє шлях до файлу. Після запиту властивості *Name* створюється його кеш, тому він відобразиться на консолі за допомогою вибору користувача.

Зверніть увагу, оскільки дублюється дескриптор із процесу, цей скрипт може відображати дескриптори лише в процесах, які можуть відкрити виклик. Щоб отримати кращі результати, запустіть його як адміністратор, який може відкрити максимальну кількість процесів.

2.10.3. Пошук загальних об'єктів

Коли записується список дескрипторів за допомогою команди *Get-NtHandle*, отримуєте адресу об’єкта в ядрі пам’яті. Під час відкриття одного й

того ж об'єкта ядра отримуєте різні дескриптори, але вони, як і раніше, будуть показуватися на одну й ту ж адресу об'єкта ядра.

Доцільно використовувати адресу об’єкта для процесів пошуку, які сумісно використовують дескриптори.

Це може бути цікаво для безпеки в двох випадках, коли об’єкт одночасно використовується процесами з різними привілеями. Процес із більш низькими привілеями може мати можливість змінити властивості об’єкта, щоб забезпечити перевірку безпеки в процесі з більш високими привілеями, що дозволяє йому отримати додаткові привілеї.

Насправді використовуємо цю техніку, щоб знайти проблему безпеки CVE-2019-0943 у Windows. В корні проблеми був привілейований процес, кеш шрифтів Windows, який розділяв дескриптори розділів з процесом з низькими привілеями.

Низько привілейований процес може відобразити розділений розділ як доступний для запису та змінити вміст, який привілейований процес вважався незмінним.

Це фактично дозволило низько привілейованому процесу змінити довільну пам'ять у привілейованому процесі, що призвело до виконання привілейованого коду.

У лістингу 2.38 наведено приклад пошуку доступних для запису об’єктів *Section*, які одночасно-використовуються двома процесами.

Лістинг 2.38. Пошук загальних дескрипторів розділу

```
PS> $ss = Get-NtHandle -ObjectType Section -GroupByAddress | Where-Object ShareCount -eq 2
PS> $mask = Get-NtAccessMask -SectionAccess MapWrite
PS> $ss = $ss | Where-Object { Test-NtAccessMask $_.AccessIntersection $mask }
PS> foreach($s in $ss) {
$count = ($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
```

По-перше отримуємо дескриптори, вказуючи параметр *GroupByAddress*. Це повертає список груп, організованих на основі адреси об’єкта ядра, замість списку дескрипторів. Є можливість групувати дескриптори за допомогою вбудованої команди *Group-Object*, однак групи, що повертаються *GroupByAddress*, мають додаткові властивості, включаючи *ShareCount*, що вказує на кількість унікальних процесів з якими об’єкт спільно використовується. Застосовуємо фільтрацію, щоб включити лише дескриптори, які використовуються одночасно двома процесами.

Далі знаходимо об’єкти *Section*, які можуть бути доступні для запису.

Спершу перевіряємо, що всі дескриптори мають доступ до *MapWrite*. Як згадувалося раніше, захист об'єкта *Section* також повинен бути доступним для

запису. Хоч як це не дивно, не запитуємо вихідний захист, який може бути призначений під час створення об’єкта *Section*, але перевірка доступу *MapWrite* є простим проксі. Використовуємо властивість *AccessIntersection*, яка містить надані права доступу, загальні для всіх дескрипторів.

Коли ми маємо потенційні кандидати на розділи *Section*, потрібно визначити, які з них відповідають критеріям, щоб можна було отримати доступ лише до одного з процесів, що містять дескриптор розділу. Відповідно, робимо ще одне припущення: якщо можемо відкрити лише один із двох процесів, які розділяють дескриптор для доступу *DupHandle*, то є розділ, загальний для привілейованого та низько привілейованого процесу. Врешті-решт, якщо був доступ *DupHandle* до обох процесів, можна було б скомпрометувати процеси, викравши всі їх дескриптори або дублювати їх дескриптори процесів.

Результат, показаний у лістингу 2.38, є розділом, який спільно використовують *Chrome* і процес *Audio Device Graph*. Спільний розділ використовується для відтворення аудіо з браузера, і це, ймовірно, не є проблемою безпеки. Однак, якщо запустити сценарій у своїй системі, можна знайти спільні розділи.

Зверніть увагу, що після того, як об'єкт *Section* буде відображено у пам'ять, дескриптор більше не потрібен. Тому ви можете пропустити деякі спільні секції, які було відображено, коли початковий дескриптор було закрито. Також дуже ймовірно, що ви отримаєте хибні спрацьовування, наприклад, об'єкти *Section*, які навмисно відкриті для запису всім бажаючим. Наша мета полягає в тому, щоб знайти потенційну поверхню атаки на Windows.

Після цього ви повинні піти і оглянути дескриптори, щоб переконатися, що спільний доступ до них не призвів до виникнення проблеми з безпекою.

Наступним кроком є необхідність перевірити маркери, щоб дізнатися, чи їхній спільний доступ спричинив проблему безпеки. Треба знайти та перевірити дескриптори, щоб побачити, чи не привело їх спільне використання до проблеми безпеки.

2.10.4. Зміна відображеного розділу

Якщо знайдено цікавий об’єкт *Section* для редагування, то його можна відобразити-в пам’яті за допомогою *Add-NtSection*.

Виникає питання, яким чином змінити відображувану пам'ять.

Найбільш простий підхід, використати з командного рядка команду *Write-NtVirtualMemory*, яка підтримує передачу відображеного розділу та масиву байтів для запису. Лістинг 2.39 демонструє цей метод, припускаючи, що є цікавий дескриптор у змінній *$ handle*.

Лістинг 2.39. Відображення і зміна об'єкта *Section*

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

Спершу викликаємо метод *GetObject* для дескриптора, щоб продублювати його в поточний процес і повернути об’єкт *Section*. Щоб це вдалося, процес, у якому запускаємо цю команду, повинен отримати доступ до процесу з дескриптором. Потім відображаємо дескриптор як *ReadWrite* у пам’яті поточного процесу.

Тепер ми можемо створити випадковий масив байт розміром до розміру відображуваної ділянки і записати їх в область пам'яті за допомогою Write-NtVirtualMemory.

Це швидкий але грубий спосіб модифікації пам'яті, що розділяється. Сподівання полягають у тому, що модифікуючи пам'ять, привілейований процес неправильно поводитиметься з вмістом області пам'яті.

Якщо привілейований процес аварійно завершує роботу, ми повинні визначити, чи зможемо ми контролювати аварійне завершення за допомогою більш цілеспрямованої модифікації пам'яті, що розділяється..

Відобразити пам'ять можна за допомогою *Out-HexDump*. Одна з корисних функцій цієї команди, в порівнянні з вбудованим *Format-Hex,* полягає в тому, що вона виводить адресу в пам'ять на основі створеного файлу, тоді як *Format-Hex* просто виводить розміщення, починаючи з 0.

Ви також можете створити шістнадцятковий редактор графічного інтерфейсу за допомогою команди *Show-NtSection*, вказавши об'єкт секції для редагування. Оскільки секцію можна зіставити з будь-яким процесом, її запис у шістнадцятковому редакторі графічного інтерфейсу також змінить усі інші зіставлення цієї секції.

Нижче наведено команду для відображення шістнадцяткового редактора:

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

2.10.5. Пошук виконавчої та доступної для запису пам'яті

Для того, щоб процес міг виконувати інструкції, у Windows пам'ять повинна бути позначена як така, що виконується. Однак, також можна показати пам'ять як таку, що виконується та записується. Зловмисне ПЗ іноді використовує цю комбінацію для впровадження коду дозволу оболонки в процес і для запуску шкідливого коду з ідентифікатором хост-процесу користувача.

У лістингу 2.40 показано, як перевірити пам'ять у процесі, який є записаним і таким що виконується. Наявність такої пам'яті може вказувати на те, що відбувається щось шкідливе, хоча в більшості випадків це буде не так.

Наприклад, середовище виконання .NET створює записувану та виконувану пам'ять для виконання JIT-компіляції байт-коду .NET у власних інструкціях.

Почнемо з відкриття процесу для доступу QueryLimitedInformation , що є всім, що нам потрібно для перерахування областей віртуальної пам'яті. Тут ми відкриваємо поточний процес PowerShell. Оскільки PowerShell — це .NET, ми знаємо, що в ньому будуть деякі записані та використані області пам’яті, але відкритий вами процес може бути всім, що ви хочете перевірити.

Лістинг 2.40. Пошук виконавчої та доступної для запису пам'яті у процесі

```
PS C:\Users\hladk> $proc = Get-NtProcess -ProcessId $pid -Access QueryLimitedInformation
PS C:\Users\hladk> Get-NtVirtualMemory -Process $proc | Where-Object { $_.Protect -band "ExecuteReadWrite" }

Address          Size   Protect          Type    State  Name
-------          ----   -------          ----    -----  ----
0000028A21200000 8192   ExecuteReadWrite Private Commit
0000028A39480000 36864  ExecuteReadWrite Private Commit
0000028A398D0000 8192   ExecuteReadWrite Private Commit
00007DF4D1420000 4096   ExecuteReadWrite Private Commit
00007DF4D1430000 4096   ExecuteReadWrite Private Commit
00007DF4D1440000 4096   ExecuteReadWrite Private Commit
00007FFF30D33000 4096   ExecuteReadWrite Private Commit
00007FFF30D3D000 12288  ExecuteReadWrite Private Commit
00007FFF30D4D000 4096   ExecuteReadWrite Private Commit
00007FFF30D5B000 4096   ExecuteReadWrite Private Commit
00007FFF30D5D000 4096   ExecuteReadWrite Private Commit
00007FFF30D8C000 8192   ExecuteReadWrite Private Commit
00007FFF30DEC000 57344  ExecuteReadWrite Private Commit
00007FFF30E16000 77824  ExecuteReadWrite Private Commit
00007FFF30E50000 258048 ExecuteReadWrite Private Commit
00007FFF30EF0000 4096   ExecuteReadWrite Private Commit
00007FFF30F00000 24576  ExecuteReadWrite Private Commit
00007FFF30F40000 49152  ExecuteReadWrite Private Commit
00007FFF312C0000 143360 ExecuteReadWrite Private Commit
00007FFF9062E000 4096   ExecuteReadWrite Image   Commit clr.dll


PS C:\Users\hladk> $proc.Close()
```

Потім ми перечислюємо всі області пам'яті за допомогою Get-NtVirtualMemory і фільтруємо за типом захисту ExecuteReadWrite . Нам потрібно використовувати побитову операцію AND, оскільки до захисту можна додати додаткові прапорці, такі як *Guard*, який створює сторінку захисту, яка передбачає виконання прямої перевірки рівня.

2.11. ВИСНОВКИ ДО РОЗДІЛУ 2

У цьому розділі ми ознайомилися з ядром Windows та його взаємодією з іншими системами. Ядро складається з багатьох окремих підсистем, таких як монітор безпеки, диспетчер об'єктів, диспетчер конфігурації (або реєстр), диспетчер вводу/виводу, а також диспетчер процесів і потоків.

Ви дізналися про те, як менеджер об'єктів керує ресурсами і типами ядра, як отримати доступ до ресурсів ядра за допомогою системних викликів і як виділяються дескриптори з певними правами доступу. Ви також отримали доступ до ресурсів диспетчера об'єктів за допомогою провайдера дисків *NtObject* та окремих команд.

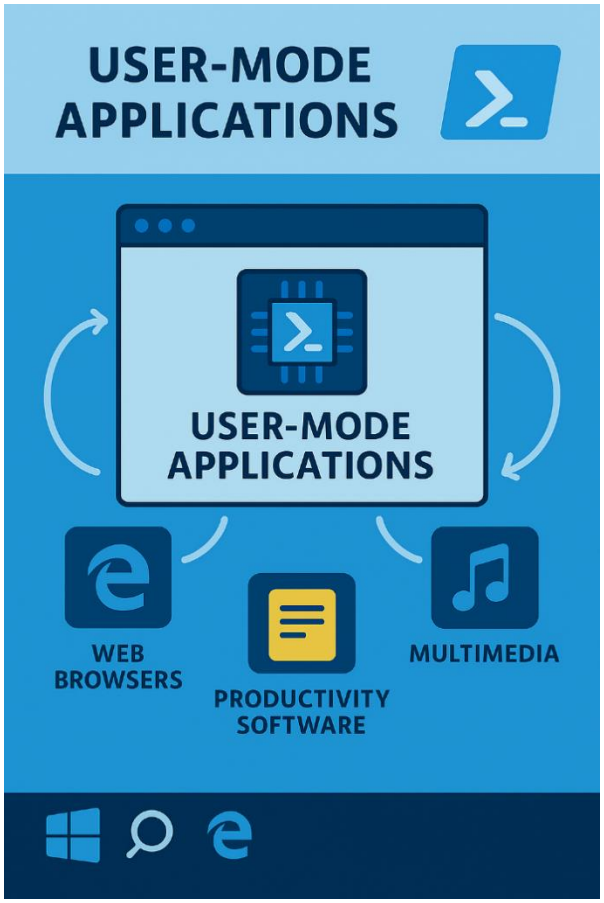
Потім розглянули основи створення процесів і потоків і продемонстрував використання таких команд, як *Get-NtProcess* для запиту інформації про процес у системі. Також показано, як перевіряти віртуальну пам'ять процесу, а також деякі окремі типи пам'яті.

Користувач не взаємодіє безпосередньо з ядром. Натомість, користувацькі програми підтримують взаємодію з ядром. У наступному розділі обговоримо компоненти користувацького режиму більш детально.

ПИТАННЯ ДЛЯ САМОКОНТРОЛЮ

1. На які підсистеми розділено ядро операційної системи Windows?
2. Які виконавчі модулі існують в ядрі Windows?
3. Який виконавчій модуль реалізує механізми безпеки операційної системи Windows?
4. Що таке SID, які його функції ?
5. Яким чином можемо отримати доступ до іменованих об’єктів в OMNS із додатків режиму користувача?
6. Наведіть структуру коду NTStatus.
7. Додаток у режимі користувача не може безпосередньо читати або записувати в пам'ять ядра, тоді як він може отримати доступ до об'єкта?
8. Що таке маска доступу?
9. За допомогою якої команди можливо відображення таблиці дескрипторів для поточного процесу ?
10. Навіщо потрібно дублювати дескриптори ?
11. Які функції виконують системні виклики Query і Set Information ?
12. За допомогою якої команди можливо здійснити перерахунок всіх завантажених драйверів ядра ?
13. За допомогою якої команди можливо отримати доступ до списку процесів і потоків ?
14. Кожен процес має власний адресний простір віртуальної пам'яті, який розробник може використовувати на свій розсуд. Яка підсистема керує розподілом цього адресного простору?
15. Як здійснюється пошук відкритих дескрипторів по імені ?

РОЗДІЛ 3. ДОДАТКИ КОРИСТУВАЛЬНОГО РЕЖИМУ



В попередніх розділах обговорювали ядро Windows. Але користувач зазвичай не взаємодіє безпосередньо з ядром. Натомість він взаємодіє з додатками користувача, такими як текстові або табличні редактори. У цьому розділі буде докладно описано, як створюються ці додатки режиму користувача і як вони взаємодіють з ядром для надання послуг.

Розпочнемо розгляд інтерфейсів програмування Win32 (API), створених для розробки програм режиму користувача, та як вони пов'язані з дизайном операційної системи Windows. Наступним кроком розглянемо структуру інтерфейсу користувача Windows і те, як можливо перевіряти його програмно. Необхідно розглянути сеанси консолі які можуть ізолювати інтерфейс та ресурси програми одного користувача від інтерфейсів та ресурсів інших користувачів у тій же системі.

Щоб зрозуміти, як функціонують програми режиму користувача, важливо розуміти, як надані API взаємодіють з базовим інтерфейсом системних викликів ядра. Також необхідно розглянути яким чином програми Win32 отримують доступ до реєстру, бо Win32 обробляє створення процесів та потоків.

3.1. WIN32 ТА API-ІНТЕРФЕЙСИ WINDOWS У РЕЖИМІ КОРИСТУВАЧА

Більша частина коду, що працює в Windows, безпосередньо не взаємодіє з системними викликами. Це артефакт оригінального дизайну операційної системи Windows NT.

Коли відносини Microsoft з IBM зіпсувалися, і Microsoft взяла набір API, розроблений для Windows 95, Win32, і створила підсистему для його реалізації. В значній мірі підсистема OS/2 була видалена в Windows 2000, в той час як POSIX проіснувала до появи Windows 8.1.

У Windows 10 Win32 залишилася єдиною підсистемою (хоча Microsoft згодом реалізувала рівні сумісності Linux, такі як підсистема Windows для Linux, які не використовують старі точки розширення підсистеми). Ядро Windows реалізує загальний набір системних викликів. Конкретні бібліотеки та служби

кожної підсистеми відповідають за перетворення своїх API на інтерфейс системних викликів низького рівня. На рис. 3.1 надано огляд бібліотек API підсистеми Win32.

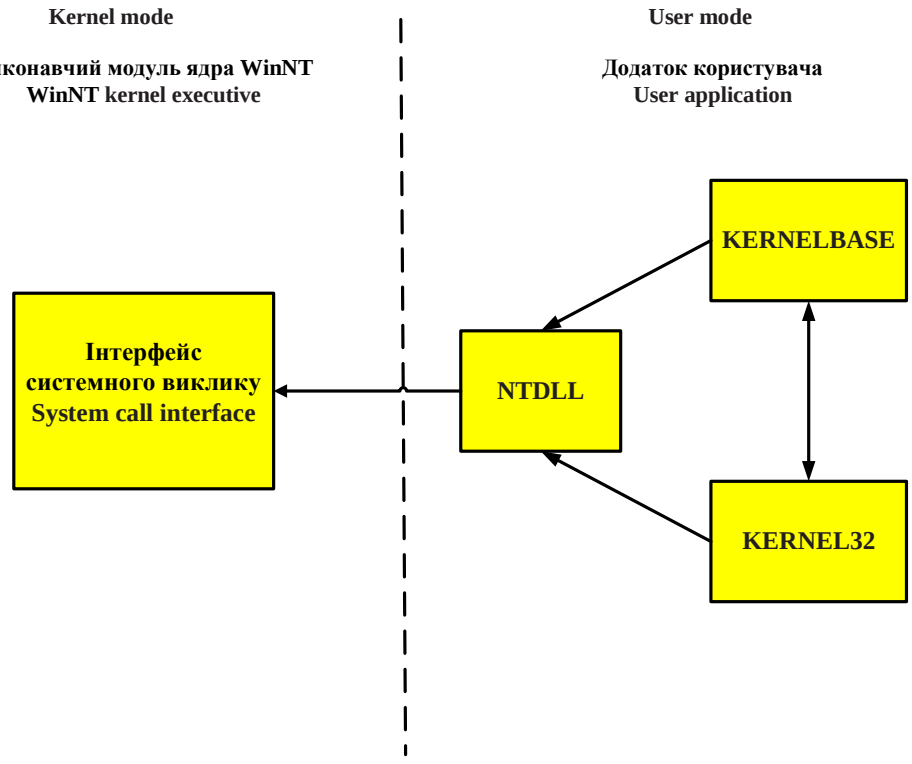


Рисунок 3.1. Модулі Win32 API

Як бачимо, ядро API Win32 реалізовано у бібліотеках KERNEL32 та KERNELBASE. Ці бібліотеки викликають методи в системній динамічній бібліотеці *NT Layer dynamic link library* (NTDLL), яка реалізує диспетчеризацію системних викликів, а також бібліотеки API часу виконання для роботи зі звичайними низькорівневими операціями.

Більшість користувацьких програм не містять безпосередньої реалізації системних API-інтерфейсів Windows. Натомість NTDLL містить завантажувач DLL, який підключає нові бібліотеки на вимогу.

Процес завантаження здебільшого непрозорий для розробника. При створенні програми ви посилаєтесь на набір бібліотек, а компілятор та інструментарій автоматично додають таблицю імпорту до вашого виконавчого файлу, щоб відобразити залежності. Потім завантажувач DLL перевіряє таблицю імпорту, автоматично завантажує всі залежні бібліотеки і виконує імпорт. Ви також можете вказати експортовані функції з вашої програми, щоб інший код міг покладатися на ваші API.

3.1.2. Завантаження нової бібліотеки

Ви можете отримати доступ до експортованих функцій самостійно під час виконання, не використовуючи запис у таблиці імпорту.

Лістинг 3.1. Експорт для бібліотеки KERNEL32

```
PS C:\Users\hladk> $lib = Import-Win32Module -Path "kernel32.dll"
PS C:\Users\hladk> $lib

Name         ImageBase        EntryPoint
----         ---------        ----------
KERNEL32.DLL 00007FFFB59D0000 00007FFFB59FE120


PS C:\Users\hladk> Get-Win32ModuleExport -Module $lib

Ordinal Name                                        Address
------- ----                                        -------
1       AcquireSRWLockExclusive                     NTDLL.RtlAcquireSRWLockExclusive
2       AcquireSRWLockShared                        NTDLL.RtlAcquireSRWLockShared
3       ActivateActCtx                              0x7FFFB5A07D20
4       ActivateActCtxWorker                        0x7FFFB59DE4E0
5       ActivatePackageVirtualizationContext        0x7FFFB5A27C70
6       AddAtomA                                    0x7FFFB5A158D0
7       AddAtomW                                    0x7FFFB5A080A0
8       AddConsoleAliasA                            0x7FFFB5A27990
9       AddConsoleAliasW                            0x7FFFB5A279A0
10      AddDllDirectory                             api-ms-win-core-libraryloader-l1-1-0.AddDllDirectory
11      AddIntegrityLabelToBoundaryDescriptor       0x7FFFB5A151E0
12      AddLocalAlternateComputerNameA              0x7FFFB5A39630
13      AddLocalAlternateComputerNameW              0x7FFFB5A39690
PS C:\Users\hladk> "{0:X}" -f (Get-Win32ModuleExport -Module $lib -ProcAddress "AllocConsole")
7FFFB5A275C0
```

Завантажити нову бібліотеку можна за допомогою Win32 API LoadLibrary, який викликається з PowerShell командою Import-Win32Module. Щоб знайти адресу пам'яті функції, експортованої DLL, використовуйте Win32 API *GetProcAddress*, який викликається командою PowerShell *Get-Win32ModuleExport* (лістинг 3.1).

Тут використовуємо PowerShell для завантаження бібліотеки KERNEL32 і перегляду експортованих та імпортованих API. Спочатку завантажте бібліотеку в пам'ять за допомогою команди *Import-Win32Module*.

Бібліотека KERNEL32 завжди завантажена, тому ця команда просто поверне існуючу завантажену адресу. Для інших бібліотек, однак, завантаження призведе до відображення DLL у пам'ять та ініціалізації.

Команда *Import-Win32Module* завантажить *DLL* у пам'ять і потенційно виконає код. У цьому прикладі це допустимо, оскільки *KERNEL32* є однією з довірених системних бібліотек. Однак ***не використовуйте команду*** для ненадійної *DLL*, особливо якщо аналізуєте шкідливе програмне забезпечення, оскільки це може призвести до виконання шкідливого коду. Щоб бути в безпеці, завжди виконуйте аналіз шкідливого програмного забезпечення в окремій системі, призначеній для цієї мети.

Після завантаження в пам'ять можемо відобразити деякі властивості бібліотеки. До них належать ім'я бібліотеки, а також завантажена адреса пам'яті та адреса *EntryPoint* (точка входу). *DLL* може опціонально визначати функцію *DllMain*, яка запускатиметься при завантаженні. Адреса *EntryPoint* - це перша інструкція в пам'яті, яка буде виконуватись під час завантаження *DLL*.

Далі вивантажуємо всі експортовані функції із *DLL*. У цьому випадку бачимо три фрагменти інформації для кожної: *Ordinal*, *Name* та *Address*.

Ordinal - це число, яке унікально ідентифікує експортовану функцію в *DLL*. Можна імпортувати *API* за його порядковим номером і це означає, що немає потреби експортувати ім'я. Буде видно, що певні імена відсутні в таблицях

експорту в *DLL*, коли *Microsoft* не хоче офіційно підтримувати функцію як загальнодоступний *API.*

Name – це просто ім'я експортованої функції. Воно не обов'язково має відповідати тому, як функція була викликана у вихідному коді, хоча зазвичай це так.

Нарешті, *Address* – це адреса у пам'яті першої інструкції функції. Ви помітите, що перші два експортовані функції містять рядок замість адреси. . Це випадок перенаправлення експорту. Він дозволяє DLL експортувати функцію за назвою, а завантажувачу автоматично перенаправляти її до іншої DLL. У цьому випадку *AcquireSRWLockExclusive* реалізовано як *RtlAcquireSRWLockExclusive* у NTDLL. Ми також можемо скористатися *Get-Win32ModuleExport* для пошуку однієї експортованої функції за допомогою API GetProcAddress

3.1.3. Перегляд імпортованих API

Подібним чином ми можемо переглянути API, які було імпортовано виконуваним файлом з інших DLL за допомогою команди Get-Win32ModuleImport, як показано у лістингу 3.2.

Лістинг 3.2. Перерахування імпорту для бібліотеки KERNEL32

```
PS C:\Users\hladk> Get-Win32ModuleImport -Path "kernel32.dll"

DllName                                    FunctionCount DelayLoaded
-------                                    ------------- -----------
api-ms-win-core-rtlsupport-l1-1-0.dll      12            False
api-ms-win-core-rtlsupport-l1-2-2.dll      1             False
ntdll.dll                                  365           False
KERNELBASE.dll                             93            False
api-ms-win-core-processthreads-l1-1-3.dll  4             False
api-ms-win-core-processthreads-l1-1-2.dll  6             False
api-ms-win-core-processthreads-l1-1-1.dll  11            False
api-ms-win-core-processthreads-l1-1-0.dll  37            False
api-ms-win-core-registry-l1-1-0.dll        42            False
PS C:\Users\hladk> Get-Win32ModuleImport -Path "kernel32.dll" -DllName "ntdll.dll" | Where-Object Name -Match "^Nt"

Name                          Address
----                          -------
NtEnumerateKey                7FFFB6B9FD50
NtTerminateProcess            7FFFB6B9FC90
NtDeleteValueKey              7FFFB6BA12E0
NtSetValueKey                 7FFFB6BA0300
NtQueryInstallUILanguage      7FFFB6BA2280
NtQueryLicenseValue           7FFFB6BA2300
NtMapUserPhysicalPagesScatter 7FFFB6B9F770
NtMapViewOfSection            7FFFB6B9FC10
NtUnmapViewOfSection          7FFFB6B9FC50
```

Спочатку виконайте команду *Get-Win32ModuleImport*, вказавши шлях до бібліотеки KERNEL32 DLL. Коли ви вкажете шлях, команда викличе *Import-Win32Module* і відобразить всі імпортовані дані, включаючи назву DLL для завантаження і кількість імпортованих функцій. В останньому стовпчику вказано, чи було позначено розробником DLL як таку, що завантажується із затримкою. Це оптимізація продуктивності. Вона дозволяє завантажувати DLL лише тоді, коли використовується одна з її експортованих функцій. Ця затримка дозволяє уникнути завантаження всіх DLL у пам'ять під час ініціалізації, що скорочує час запуску процесу і зменшує використання пам'яті під час виконання, якщо імпорт ніколи не використовується.

Далі виводимо дамп імпортованих функцій для бібліотеки DLL.

Оскільки виконавчий файл може імпортувати код з декількох бібліотек, вказуємо потрібну бібліотеку за допомогою властивості *DllName*. Потім фільтруємо всі імпортовані функції, що починаються з префікса *Nt*. Це дозволяє побачити, які саме системні виклики KERNEL32 імпортує з NTDLL.

Ви можете помітити щось підозріле в списку імпортованих імен DLL у лістингу 3.2.

Якщо ви шукаєте файл *api-ms-win-core-rtlsupport-l1-1-0.dll* у своїй файловій системі, ви його не знайдете. Це пов’язано з тим, що ім'я DLL посилається на набір API. Набори API були введені в Windows 7 для створення модулів системних бібліотек і вони абстрагуються від імені набору до DLL, яка експортує API.

Набори API дозволяють виконуваному файлу працювати в різних версіях Windows, таких як клієнтська або серверна, і змінювати його функціональність під час виконання залежно від доступних бібліотек.

Коли завантажувач *DLL* зустрічає одне з цих імен наборів API, він звертається до таблиці, завантаженої в кожний процес, отриманої з файлу *apisetschema.dll*, яка зіставляє ім'я реальної *DLL*. Ви можете запросити відомості про набір API, використовуючи команду *Get-NtApiSet* і вказавши ім'я набору *API:*

```
PS C:\Users\hladk> 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
```

Ви також можете встановити параметр *ResolveApiSet* для команди *Get-Win32ModuleImport*, щоб згрупувати імпорт на основі реальних DLL-файлів:

:

```
PS C:\Users\hladk> Get-Win32ModuleImport -Path "kernel32.dll"  -ResolveApiSet

DllName                          FunctionCount DelayLoaded
-------                          ------------- -----------
ntdll.dll                        379           False
KERNELBASE.dll                   887           False
ext-ms-win-oobe-query-l1-1-0.dll 1             True
daxexec.dll                      7             True
RPCRT4.dll                       10            True
```

Якщо порівняти вихідні дані в лістингу 3.2 з даними тієї ж команди, показаної тут, ви помітите, що список вирішених імпортів набагато коротший, а основні бібліотеки отримали додаткові імпорти функцій.

Також зверніть увагу на заборонене ім’я набору *API*, *ext-ms-win-oobe-query-l1-1-0.dll*. Будь-який набір *API* із префіксом *API* завжди має бути присутнім, тоді як набір із префіксом *EXT* може бути відсутнім.

У даному випадку набір API відсутній, і спроба викликати імпортовану функцію закінчиться невдачею. Однак, оскільки функція позначена як така, що завантажується із затримкою, виконуваний файл може перевірити, чи доступний набір API, перед викликом функції, використовуючи Win32 API *IsApiSetImplemented*.

3.1.3. Пошук DLL

Під час завантаження *DLL* завантажувач створює об’єкт розділу зображення з виконуваного файлу та відображає його в пам’яті. Ядро відповідає за відображення виконуваної пам'яті, однак код режиму користувача все ще потребує аналізу таблиць імпорту та експорту.

Припустімо, ви передаєте рядок *ABC.DLL* до *LoadLibrary API*. Як *API* знає, де знайти цю *DLL?* Якщо абсолютний шлях, *API* реалізує алгоритм пошуку шляху. Алгоритм, спочатку реалізований у *Windows NT 3.1*, шукає файли в такому порядку:

1. Той самий каталог, в якому знаходиться виконуваний файл поточного процесу;
2. Поточний робочий каталог;
3. Каталог *Windows System32*;
4. Каталог *Windows*;
5. Кожне розділене точкою з комою місце розташування в змінному середовищі *PATH*.

Проблема з цим порядком завантаження полягає в тому, що він може призвести до завантаження привілейованим процесом *DLL* із небезпечного розташування. Наприклад, якщо привілейований процес змінив свій поточний робочий каталог за допомогою API *SetCurrentDirectory* на місце, куди може записувати користувач який має менші привілеї, то *DLL* буде завантажена з цього місця раніше за будь-який *DLL* з каталогу *System32*. Ця атака називається перехопленням *DLL*, і це постійна проблема у *Windows*.

Vista змінила порядок завантаження за замовчуванням на наступний, що є безпечніше:

1. Той самий каталог, що й виконуваний файл поточного процесу;
2. Каталог *Windows System32*;
3. Каталог *Windows*;
4. Поточний робочий каталог;
5. Кожне розділене точкою з комою місце розташування в змінному середовищі PATH.

Наразі більше не завантажуємо з поточного робочого каталогу перед *System32* або *Windows*. Однак, якщо зловмисник міг записати в каталог виконуваного файлу, стороннє *DLL* все одно може відбутися. Таким чином, якщо виконуваний файл запускається як привілейований процес, лише адміністратори повинні мати можливість змінювати його каталог, щоб запобігти використанню небезпечних DLL.

Окрема особливість завантаження пов'язана з обробкою розширень файлів у імені файлу DLL. Якщо розширення не вказано, завантажувач DLL автоматично додає розширення .DLL.

Якщо розширення вказано, ім'я файлу обробляється без змін. Нарешті, якщо розширення складається з однієї крапки (наприклад, LIB.), завантажувач

видаляє крапку і намагається завантажити файл без розширення (у даному випадку LIB).

Така поведінка розширення файлу може призвести до невідповідності між DLL, яку програма намагається завантажити, і тією, яку вона фактично завантажує.

Наприклад, програма може перевірити, чи файл LIB є дійсним (тобто чи має він правильний криптографічний підпис). Однак завантажувач DLL завантажить LIB.DLL, який не був перевірений. Це може стати причиною вразливостей системи ***безпеки***, якщо ви зможете змусити привілейовану програму завантажити в пам'ять неправильний DLL, оскільки точка входу буде виконуватися в привілейованому контексті.

Хоча завантажувач DLL зазвичай звертається до диска для отримання бібліотеки, деякі бібліотеки використовуються так часто, що має сенс попередньо ініціалізувати їх. Це підвищує продуктивність та запобігає перехопленню DLL. Два очевидні приклади KERNEL32 та NTDLL.

Перед запуском будь-яких програм користувача в Windows система налаштовує каталог *KnownDlls OMNS*, що містить список попередньо завантажених розділів. Ім'я об'єкта KnownDlls Section – це просто ім'я файлу бібліотеки. Завантажувач DLL спочатку може перевірити *KnownDlls*, перш ніж перейти на диск. Це підвищує продуктивність, оскільки завантажувачеві більше не потрібно створювати новий об'єкт *Section* для файлу. Це також має перевагу в плані безпеки, гарантуючи, що все, що вважається відомим DLL, не може бути перехоплено.

Ми можемо відобразити каталог об'єктів за допомогою команди *NtObject* лістингу 3.3.

Лістинг 3.3: Список вмісту каталогу об'єктів KnownDlls

```
PS C:\Users\hladk> ls NtObject:\KnownDlls

Name                TypeName
----                --------
kernel32.dll        Section
ucrtbase.dll        Section
WS2_32.dll          Section
SHLWAPI.dll         Section
MSCTF.dll           Section
KERNELBASE.dll      Section
wow64.dll           Section
msvcp_win.dll       Section
gdiplus.dll         Section
KnownDllPath        SymbolicLink
user32.dll          Section
```

Розглянуті основи підсистеми Win32 і те, як вона використовує бібліотеки для реалізації API, які додаток режиму користувача може використовувати для взаємодії з операційною системою.

3.2. ГРАФІЧНИЙ ІНТЕРФЕЙС КОРИСТУВАЧА WIN32

Назва «Windows» походить від структури графічного інтерфейсу користувача (GUI) операційної системи. Цей GUI складається з одного або декількох вікон, з якими користувач може взаємодіяти за допомогою елементів керування, таких як кнопки та введення тексту. З часів Windows 1.0 GUI є

найважливішою функцією операційної системи, тому не дивно, що її модель є складною. Реалізація GUI розділена між ядром і режимом користувача, як показано на рис. 3.2.

Зверніть увагу, що ліва частина рисунка 3.2 виглядає дуже схоже на рисунок 3.1, на якому показано модулі для звичайних API Win32. Однак замість NTDLL стоїть WIN32U, який реалізує заглушки системного виклику для виклику ядра. Дві бібліотеки викликають WIN32U: USER32 та GDI32.

USER32 реалізує елементи інтерфейсу вікна та загалом керує графічним інтерфейсом користувача, тоді як GDI32 реалізує примітиви малювання, такі як шрифти та форми.

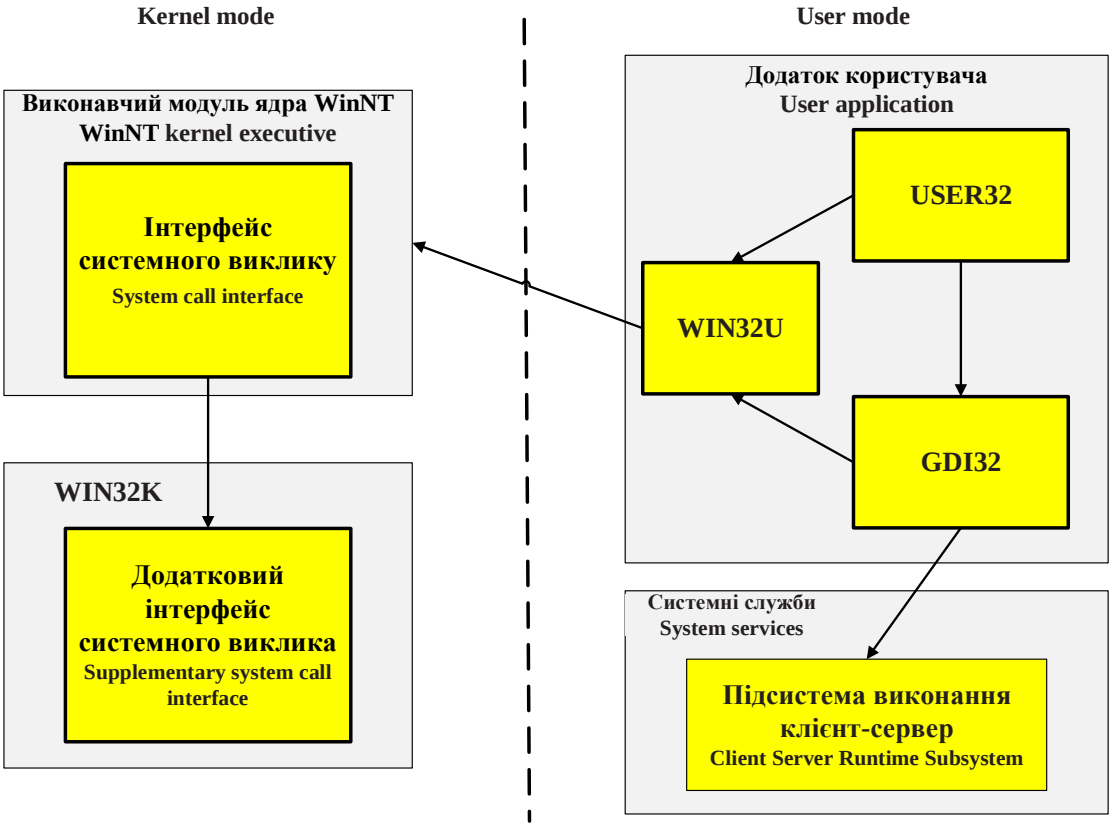


Рисунок 3.2. Модулі графічного інтерфейсу Win32

У Windows до 10 версії код диспетчеризації системних викликів у WIN32U був вбудований безпосередньо в DLL режиму користувача. Це ускладнювало прямий виклик WIN32K без написання програми на мові асемблера .

API графічного інтерфейсу також взаємодіють із спеціальним привілейованим процесом: *Client Server Runtime Subsystem* (CSRSS). Цей процес відповідає за обробку певних привілейованих операцій для клієнтів з нижчими привілеями, такими як налаштування зіставлень дисків для кожного користувача, управління процесами та обробка помилок.

3.2.1. Ресурси ядра GUI

GUI складається з чотирьох типів ресурсів ядра:

— ***Window stations*** - Об'єкти, що представляють сполучення з екраном та інтерфейсом користувача, такі як клавіатура та миша;

— ***Windows GUI elements*** - Елементи графічного інтерфейсу для інтерактивної взаємодії з користувачем, прийому вхідних даних та відображення результатів;

— ***Desktops*** - Об'єкти, які відповідають за відображення робочого столу та виконують роль хоста для вікон;

— ***Drawing resources*** - Растрові зображення, шрифти або будь-що інше, що потрібно відобразити користувачеві.

У той час як ядро Win32 і користувацькі компоненти обробляють вікна, віконні станції та робочі столи доступні через менеджер об'єктів.

Існують типи об'єктів ядра для віконних станцій та робочих столів, як показано у лістингу 3.4.

Лістинг 3.4. Відображення об'єктів типу *WindowStation* та *Desktop*

```
PS C:\Users\hladk> Get-NtType WindowStation,Desktop

Name
----
WindowStation
Desktop
```

Віконна станція призначається процесу коли запускається, або за допомогою API *NtUserSetProcessWindowStation* . Робочі столи призначаються на основі потоків за допомогою *NtUserSetThreadDesktop*. Можливо отримати імена віконних станцій та робочих столів за допомогою команд у лістингу 3.5.

Лістинг 3.5. Відображення всіх поточних віконних станцій та робочих столів

```
PS C:\Users\hladk> Get-NtWindowStationName
WinSta0
Service-0x0-1357df$
PS C:\Users\hladk> Get-NtWindowStationName -Current
WinSta0
PS C:\Users\hladk> Get-NtDesktopName
sbox_alternate_desktop_local_winstation_0x3A1C
Default
PS C:\Users\hladk> Get-NtDesktopName -Current
Default
```

Почнемо із запиту імен усіх доступних віконних станцій.

У цьому прикладі їх два: віконна станція WinSta0 за замовчуванням та Service-0x0-1357df$, створена іншим процесом. Можливість створення окремих віконних станцій дозволяє процесу ізолювати власну взаємодію з графічним інтерфейсом від інших процесів, що працюють в той же час.

Однак WinSta0 є особливим, оскільки це єдиний об'єкт, підключений до консолі користувача.

Далі перевіряємо, яке ім'я даної поточної віконної станції, використовуючи параметр *Current*. Ми можемо бачити, що знаходимося на *WinSta0*.

Далі ми запитуємо імена робочих столів на нашій поточній робочій станції. Бачимо тільки два робочі столи: *Default* і *WinLogon*.

Робочий стіл WinLogon буде видимий тільки в тому випадку, якщо ви запустите команду *Get-NtDesktopName* як адміністратор, оскільки він використовується виключно для відображення екрану входу в систему, до якого звичайна користувацька програма не повинна мати доступу.

Об'єкти робочого столу повинні відкриватися відносно шляху до робочої станції. Для робочих столів не існує спеціального каталогу об'єктів. Тому ім'я робочого столу відображає ім'я об'єкта робочої станції.

Нарешті перевіримо ім'я робочого столу поточного потоку. Робочий стіл, до якого ми підключені, відображається як *Default*, бо це єдиний робочий стіл, доступний для звичайних програм користувача. Можемо перерахувати вікна, створені на робочому столі, за допомогою *Get-NtDesktop* та *Get-NtWindow* (лістинг 3.6).

Можна бачити, кожене вікно має кілька властивостей. По-перше, це його дескриптор, який є унікальним для робочого столу. Це не той тип дескриптора, який розглядали у попередньому розділі для об'єктів ядра. Це значення призначене підсистемою Win32.

Лістинг 3.6. Перелік вікон поточного робочого столу

```
PS C:\Users\hladk> $desktop = Get-NtDesktop -Current
PS C:\Users\hladk> Get-NtWindow -Desktop $desktop

Handle   ProcessId ThreadId ClassName
------   --------- -------- ---------
131252   10908     11016    CursorVisualClass
66242    10900     12276    ForegroundStaging
66182    10900     12120    ForegroundStaging
66094    10900     12004    tooltips_class32
66090    10900     12004    tooltips_class32
66070    10900     12004    Shell_TrayWnd
264718   5124      13688    Chrome_WidgetWin_1
```

Для функціонування вікно отримує повідомлення від системи. Наприклад, коли натискаєте на кнопку миші у вікні, система надсилає повідомлення, щоб повідомити вікно про дії користувача, та про те, яка кнопка миші була натиснута. Потім вікно може обробити повідомлення та змінити свою поведінку відповідним чином.

Також можна вручну надсилати повідомлення вікна за допомогою API *SendMessage* та *PostMessage*.

Кожне повідомлення складається з числового ідентифікатора, наприклад 0x10, який представляє повідомлення WM_CLOSE для закриття вікна та двох додаткових параметрів. Значення двох параметрів залежить від повідомлення.

Наприклад, якщо повідомлення WM_CLOSE, то жоден з параметрів не використовується. Для інших повідомлень вони можуть представляти покажчики на рядки або цілі. Повідомлення можуть бути надіслані або опубліковані. Різниця між відправкою та публікацією повідомлення полягає в тому, що відправка чекає, поки вікно обробить повідомлення і поверне значення, тоді як публікація просто відправляє повідомлення у вікно та негайно повертає його.

У лістингу 3.6 стовпці *ProcessId* та *ThreadId* ідентифікують процес і потік, які створили вікно за допомогою API, такого як *CreateWindowEx*. Вікно має те, що називається прив'язкою до потоку, і це означає, що тільки потік, який створює може керувати станом вікна і обробляти його повідомлення.

Однак будь-який потік може надсилати повідомлення у вікно. Для обробки повідомлень потік, який створює повідомлення повинен запустити цикл повідомлень, які викликає API *GetMessage* для отримання наступного доступного повідомлення, а потім відправляє його у функцію зворотного

виклику обробника вікна за допомогою API DispatchMessage . Коли програма не виконує цикл, можна побачити зависання програм Windows, оскільки без циклу графічний інтерфейс не може бути оновлений.

Останній стовпець у лістингу 3.6 - це *ClassName* . Це ім'я класу вікна, який діє як шаблон нового вікна. При виклику *CreateWindowEx* вказується *ClassName* і вікно ініціалізується зі значеннями за замовчуванням із шаблону, такими як стиль рамки або розмір за замовчуванням. Зазвичай програма реєструє власні класи для обробки унікальних вікон. Як альтернатива, воно може використовувати системні класи для таких речей, як кнопки та інші загальні елементи управління.

3.2.2. Віконні повідомлення

Давайте розглянемо простий приклад у лістингу 3.7, в якому відправляється повідомлення вікна, щоб знайти текст заголовка для всіх вікон на робочому столі.

Лістинг 3.7. Надсилання повідомлення WM_GETTEXT всім вікнам на робочому столі

```
PS C:\Users\hladk> $ws = Get-NtWindow
PS C:\Users\hladk> $char_count = 2048
PS C:\Users\hladk> $buf = New-Win32MemoryBuffer -Length ($char_count*2)
PS C:\Users\hladk> foreach($w in $ws) { $len = Send-NtWindowMessage -Window $w -Message 0xD -LParam $buf.DangerousGetHan
dle() -WParam $char_count -Wait
>> $txt = $buf.ReadUnicodeString($len.ToInt32())
>> if ($txt.Length -eq 0) { continue }
>> "PID: $($w.ProcessId) - $txt" }
PID: 10900 - Индикатор батарей
PID: 5124 - Картинка в картинке
PID: 2732 - HardwareMonitorWindow
PID: 5124 - Картинка в картинке
PID: 16048 - Intel® Graphics Software
PID: 16048 - Intel® Graphics Software
PID: 16048 - Фоновый процессор Intel® Graphics Software
PID: 18660 - Windows PowerShell
```

Спочатку перераховуємо всі вікна на поточному робочому столі за допомогою команди *Get-NtWindow*. Потім виділяємо буфер пам'яті для зберігання 2048 символів. Майте на увазі, що будемо використовувати цей буфер для зберігання 16-бітних символів Unicode, тому кількість символів має бути помножена на 2, щоб визначити розмір буфера в байтах.

У циклі потім відправляємо повідомлення WM_GETTEXT (яке є номером повідомлення 0xD) кожному вікну, щоб запросити заголовок вікна повідомлень. Чекаємо на отримання відправки повідомлення, який вказує кількість символів, скопійованих в буфер.

Потім можемо прочитати рядок заголовка та вивести його на екран, ігноруючи будь-які вікна з порожнім заголовком.

Далі ми розглянемо, як кілька користувачів можуть використовувати власні інтерфейси користувача в одній системі за допомогою створення сеансів консолі.

3.2.3. Сеанси консолі

Перша версія Windows NT дозволяла кільком користувачам проходити аутентифікацію одночасно, і кожен із них запускав процеси. Однак до появи служб віддалених робочих столів (RDS) неможливо було, щоб різні інтерактивні робочі столи одночасно запускали кілька облікових записів користувачів однією машиною.

Всі зареєстровані користувачі повинні були спільно використовувати одну фізичну консоль. Windows NT4 представила підтримку кількох консолей як додаткову функцію, доступну лише Windows Server, стандартною вона стала лише в Windows XP.

RDS – це служба на робочих станціях та серверах Windows, яка дозволяє віддалено підключатися до графічного інтерфейсу та взаємодіяти з системою. Вона використовується для віддаленого адміністрування та надання загального хостингу для кількох користувачів в одній системі, підключеної до мережі. Крім того, її функціональність було переорієнтовано на підтримку механізму, який може перемикатися між користувачами в одній системі без необхідності їх виходу з системи.

Щоб підготуватися до входу нового користувача Windows, служба менеджера сеансів створює новий сеанс на консолі. Цей сеанс використовується для організації віконної станції та об'єктів робочого столу користувача, щоб вони були відокремлені від об'єктів, що належать будь-якому іншому зареєстрованому користувачу.

Ядро створює об'єкт *Session* для відстеження ресурсів, а іменоване посилання на об'єкт зберігається у каталозі *KernelObjects* OMNS. Однак об'єкт Session зазвичай відображається користувачеві лише як ціле число. Це ціле число не є випадковим, воно просто збільшується на 1 при створенні кожної нової сесії консолі.

Менеджер сеансів запускає кілька процесів у новому сеансі до того, як будь-який користувач увійде в систему. До таких процесів відносяться виділена копія CSRSS і процес *Winlogon*, які відображають інтерфейс облікових даних і обробляють аутентифікацію нового користувача.

Консольний сеанс, до якого належить процес, призначається під час запуску процесу. (Технічно сеанс консолі вказується в токені доступу). Можемо спостерігати за процесами, запущеними в кожному сеансі, виконавши деякі команди PowerShell, як показано у лістингу 3.8.

Лістинг 3.8. Відображення процесів у кожному сеансі консолі за допомогою *Get-NtProcess*

```
PS C:\Users\hladk> Get-NtProcess -InfoOnly | Group-Object SessionId

Count Name                      Group
----- ----                      -----
  129 0                       {Idle, System, Secure System, Registry...}
  113 1                       {csrss.exe, winlogon.exe, fontdrvhost.exe, dwm.exe...}
```

Windows має лише одну фізичну консоль, яка підключена до клавіатури, миші та монітора. Однак можна створити новий віддалений робочий стіл через

мережу, використовуючи клієнт, який взаємодіє за допомогою протоколу віддаленого робочого столу (RDP).

Також можна переключити користувача, що увійшов до системи, на фізичну консоль. Це включає підтримку функції швидкого перемикання користувачів Windows. Коли фізична консоль переключається на нового користувача, попередній користувач все ще перебуває в системі та працює у фоновому режимі, але ви не можете взаємодіяти з робочим столом цього (відключеного) користувача.

Кожен сеанс консолі має власну спеціальну область пам'яті ядра. Наявність дубльованих ресурсів гарантує, що сеанси консолі поділені. Сеанс номер 0 є особливим, оскільки він призначений лише для привілейованих служб та управління системою. Зазвичай неможливо використовувати графічний інтерфейс із процесами, запущеними у цьому сеансі.

До Windows Vista служби та фізична консоль працювали в сеансі 0.

Оскільки будь-який процес міг надсилати повідомлення вікна до будь-якого іншого процесу в тому самому сеансі, це спричинило слабкість безпеки, яка називається атакою **«*shatter*»**.

Атака ***«shatter»*** відбувається, коли звичайний користувач, щоб підвищити привілеї, може надіслати повідомлення вікна до програми з більшими привілеями в тому самому сеансі.

Наприклад, повідомлення WM_TIMER могло приймати довільний вказівник функції, який програма з більш високим рівнем привілеїв викликала б після отримання повідомлення. Звичайний користувач міг надіслати це повідомлення з ретельно підібраним покажчиком функції, щоб уможливити виконання довільного коду в контексті програми з високим рівнем привілеїв.

Windows Vista зменшила ризик атак типу ***«shatter»*** за допомогою двох пов'язаних функцій безпеки, які досі присутні в останніх версіях Windows.

Першою була ізоляція сеансу 0, яка перемістила фізичну консоль із сеансу 0, щоб звичайна користувацька програма не могла надсилати повідомлення до служб.

Друга, ізоляція привілеїв користувацького інтерфейсу (UIPI), запобігає взаємодії процесів із нижчими привілеями з вікнами з вищими привілеями. Тому, навіть якщо служба створює вікно на робочому столі користувача, система відхилить будь-які повідомлення, надіслані користувачем до привілейованої служби.

Ще однією важливою особливістю, пов'язаною з консольними сеансами, є розділення іменованих об'єктів. У попередньому розділі ми обговорили каталог *BaseNamedObjects*, який є глобальним розташуванням для іменованих об'єктів, що забезпечує засоби для спільного використання ресурсів кількома користувачами.

Однак, якщо кілька користувачів можуть одночасно увійти до системи, ви можете легко отримати конфлікти імен. Windows вирішує цю проблему,

створюючи окремий каталог BNO для кожного консольного сеансу за адресою \Sessions\<N>\BaseNamedObjects, де <N> – ідентифікатор консольного сеансу.

Каталог \Sessions також містить каталог для станцій вікон у розділі \Sessions\<N>\Windows, що гарантує, що ресурси вікон також розділені. Ви можете перерахувати каталог BNO поточного консольного сеансу з диском *NtObjectSession*, як показано в лістингу 3.9.

Лістинг 3.9 - Вміст каталогу BNO сеансу

```
PS C:\Users\hladk> ls NtObjectSession:\ | Group-Object TypeName

Count Name                   Group
----- ----                   -----
  342 Semaphore              {SM0:10968:304:WilStaging_02_p0, SM0:20460:304:WilStaging_02_p0, PDFTransformerLC-{A...
  225 Mutant                 {SM0:956:304:WilStaging_02, SM0:16060:304:WilStaging_02, {ECB2FF1A-ED67-4CB3-894A-5D...
  317 Event                  {AnimatedFacadePropertyChangesComplete.20460.20304, {3072C704-9EB1-4426-9DA5-003ACED...
  107 Section                {App.599fd15346270a75_Module, 1404HWNDInterface:70376, 3b70HWNDInterface:200ca, aswA...
    4 SymbolicLink           {AppContainerNamedObjects, Local, Session, Global}
    1 ALPC Port              {SIPC_{2819B8FF-EB1C-4652-80F0-7AB4EFA88BE4}}
    1 Job                    {WinlogonAccess}
    1 Directory              {Restricted}
```

Для сеансу 0 немає сеансового BNO для кожної консолі. Він використовує глобальний каталог BNO.

3.3. ПОРІВНЯННЯ WIN32 API ТА СИСТЕМНИХ ВИКЛИКІВ

Не всі системні виклики безпосередньо доступні через Win32, і в деяких випадках Win32 API знижує функціональність відкритих системних викликів. Розглянемо деякі спільні риси та різницю між системними викликами та їх еквівалентами Win32 API.

Як приклад, розглянемо *CreateMutexEx* API, версію Win32 системного виклику *NtCreateMutant*, який розглядали в попередньому розділі. API має прототип C, показаний у лістингу 3.10.

Лістинг 3.10. Прототип для *CreateMutexEx* Win32 API

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

Порівняйте його з прототипом *NtCreateMutant*, показаним у лістингу 3.11.

Лістинг 3.11. Прототип системного виклику *NtCreateMutant*

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

Перша різниця між прототипами полягає в тому, що Win32 API повертає дескриптор об'єкта ядра, тоді як системний виклик повертає код NTSTATUS (і отримує дескриптор через покажчик як перший параметр).

Виникає питання: яким чином помилки передаються зворотній стороні API, якщо не через код NTSTATUS? У цьому плані Win32 API не завжди послідовні. Якщо API повертає дескриптор, зазвичай повертається значення NULL. Однак деякі API, такі як файлові API, натомість повертають значення -1.

Якщо дескриптор не повертається, то зазвичай повертається логічне значення, де TRUE вказує на успіх, а FALSE вказує на помилку.

Але що якщо є потреба дізнатися, чому API не вдалося? Для цього API визначають набір кодів помилок. На відміну від кодів NTSTATUS ці коди помилок не мають жодної структури. Це просто числа. Якщо трапився збій, API Windows можна запитати цей код помилки, викликавши API *GetLastError* .

NTDLL надає API *RtlNtStatusToDosError* для перетворення коду NTSTATUS на визначений код помилки Win32. API CreateMutexEx може перетворити код NTSTATUS на код помилки Win32 під час збою, а потім записати його в останнє розташування помилки для поточного потоку за допомогою API *SetLastError* .

Можемо шукати коди помилок у PowerShell за допомогою *Get-Win32Error*, як показано у лістингу 3.12.

Лістинг 3.12. Пошук коду помилки Win32

```
PS C:\Users\hladk> Get-Win32Error 5

ErrorCode Name                Message
--------- ----                -------
        5 ERROR_ACCESS_DENIED Отказано в доступе.
```

Друга велика різниця між системним викликом і Win32 API полягає в тому, що API не приймає структуру OBJECT_ATTRIBUTES. Натомість він поділяє атрибути між двома параметрами: *lpName*, що використовується для вказівки імені об'єкта, та *lpMutexAttributes*, який є вказівником на структуру SECURITY_ATTRIBUTES.

Параметр *lpName* є рядком з NUL-термінатором, що складається з 16-бітових символів Unicode. Незважаючи на те, що менеджер об'єктів використовує послідовність UNICODE_STRING, Win32 API використовує послідовність, що закінчується символом *C-style*. Це означає, що хоча символ NUL є допустимим символом для імені об'єкта, його неможливо вказати за допомогою Win32 API.

Інша відмінність полягає в тому, що ім'я не є повним шляхом розташування OMNS для об'єкта, замість цього, воно є відносним щодо каталогу BNO поточного сеансу. Це означає, що якщо ім'я ABC, то остаточний шлях - \Sessions\<N>\BaseNamedObjects\ABC, де <N> - ідентифікатор сеансу консолі.

Якщо ви хочете створити об'єкт у глобальному каталозі BNO, можна додати до імені префікс Global (наприклад, Global\ABC). Це працює, оскільки Global — це символічне посилання \BaseNamedObjects, яке автоматично створюється разом з каталогом BNO для кожного сеансу. Якщо необхідно імітувати цю поведінку за допомогою команд PowerShell Get і New, передайте їм параметр -Win32Path, як показано у лістингу 3.13.

Лістинг 3.13. Створення нового мутанта за допомогою Win32Path

```
PS C:\Users\hladk> $m = New-NtMutant ABC -Win32Path
PS C:\Users\hladk> $m.FullPath
\Sessions\1\BaseNamedObjects\ABC
```

У лістингу 3.14 показано структуру SECURITY_ATTRIBUTES.

Лістинг 3.14. Структура SECURITY_ATTRIBUTES

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

Це дозволяє вказати дескриптор безпеки нового об'єкта, а також вказати, чи дескриптор повинен бути успадкованим. CreateMutexEx Win32 API не надає жодних інших опцій з OBJECT_ATTRIBUTES.

Цей процес призводить до останніх двох параметрів у лістингу 3.10. *dwDesiredAccess* безпосередньо зіставляється з *DesiredAccess*, а власний параметр *InitialOwner* вказується через dwFlags з прапором CREATE_MUTEX_INITIAL_OWNER.

Один момент, з яким можна зіткнутися, може виникнути, якщо ви спробуєте знайти адресу CreateMutexEx API у таблиці експорту KERNEL32 DLL (лістинг 3.15).

Лістинг 3.15. Отримання CreateMutexEx з KERNEL32

```
PS C:\Users\hladk> Get-Win32ModuleExport "kernel32.dll" -ProcAddress CreateMutexEx
Исключение при вызове "GetProcAddress" с "2" аргументами: "(0x8007007F) - Не найдена указанная процедура."
C:\Users\hladk\Documents\WindowsPowerShell\Modules\NtObjectManager\2.0.1\NtObjectManager.psm1:17888 знак:13
+             $Module.GetProcAddress($ProcAddress, $true).Result.ToInt6 ...
+             ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : NotSpecified: (:) [], MethodInvocationException
    + FullyQualifiedErrorId : NtException
```

Замість того, щоб отримати адресу - отримуємо виняток. Чи вибрали неправильну бібліотеку? Спробуємо знайти API, вивантаживши всі експортовані елементи та відфільтрувавши їх за іменем, як показано в лістингу 3.16.

Лістинг 3.16. Пошук API *CreateMutexEx* шляхом створення лістингу всіх експортованих функцій

```
PS C:\Users\hladk> Get-Win32ModuleExport "kernel32.dll" | Where-Object Name -Match CreateMutexEx

Ordinal Name           Address
------- ----           -------
241     CreateMutexExA 0x7FFCC8376C60
242     CreateMutexExW 0x7FFCC8376C70
```

Як можна побачити, API *CreateMutexEx* є в списку не один раз, а двічі. Кожна функція має суфікс A або W. Це пов'язано з тим, що Windows 95 (де було створено більшість API) спочатку не підтримувала рядки Unicode, тому API використовували одно символьні рядки в поточному текстовому кодуванні.

З появою Windows NT ядро стало на 100 відсотків Unicode, але воно надало два API для однієї функції, щоб увімкнути старі програми Windows 95.

API із суфіксом A приймають одно символьні рядки або рядки ANSI.

Ці API перетворять свої рядки на рядки Unicode для передачі ядру, і вони перетворять їх назад, якщо потрібно повернути рядок.

З іншого боку, програми, створені для Windows NT, можуть використовувати API з суфіксом W для широких рядків. Їм не потрібно виконувати жодних перетворень рядків.

3.4. ШЛЯХИ РЕЄСТРУ WIN32

Попередньо було розглянуто основи того, як отримати доступ до реєстру за допомогою власних системних викликів, використовуючи шляхи OMNS. API Win32, що використовуються для доступу до реєстру, такі як *RegCreateKeyEx*, не розкривають ці шляхи OMNS.

Натомість ви отримуєте доступ до розділів реєстру щодо визначених кореневих розділів рис. 3.3.

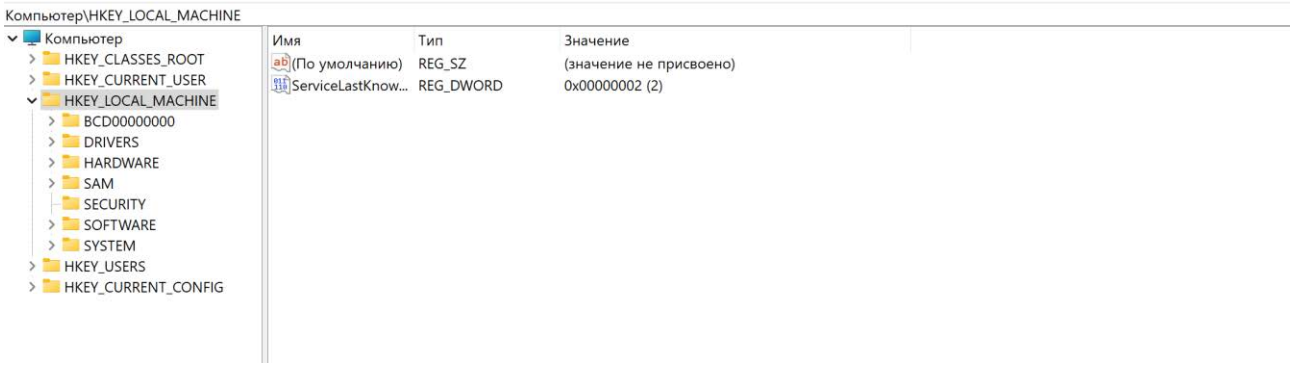


Рисунок 3.3. Основний вид утиліти **Regedit**

Значення дескрипторів, показані на рис.3.3, перераховані у таб.3.1 разом із відповідними їм шляхами OMNS.

Таблиця 3.1.
Зумовлені дескриптори реєстру та їх власні еквіваленти

Попередньо визначена назва дескриптора	OMNS шлях
HKEY_LOCAL_MACHINE	\REGISTRY\MACHINE
HKEY_USERS	\REGISTRY\USER
HKEY_CURRENT_CONFIG	\REGISTRY\MACHINE\SYSTEM\ CurrentControlSet \ Hardware Profiles\Current
HKEY_CURRENT_USER	\REGISTRY\USER\<SDDL SID>
HKEY_CLASSES_ROOT	Merged view of \REGISTRY\MACHINE\SOFTWARE\ Classes and \REGISTRY\USER\<SDDL SID>_Classes

Перші три зумовлені дескриптори, HKEY_LOCAL_MACHINE, HKEY_USERS і HKEY_CURRENT_CONFIG, не є чимось особливим. Вони безпосередньо порівнюються з одним шляхом ключа реєстру OMNS. Наступний дескриптор, HKEY_CURRENT_USER, цікавіший. Він зіставляється з кущем, завантаженим для поточного авторизованого користувача. Ім'я ключа – це рядок SDDL ідентифікатора безпеки користувача (SID).

Останній ключ, HKEY_CLASSES_ROOT, який зберігає таку інформацію, як зіставлення розширень файлів, є об'єднаним уявленням куща класів користувача і куща машини. Кущ користувача має пріоритет над кущем машини, що дозволяє користувачеві змінювати розширення файлів без потреби в адміністраторі.

3.4.1. Відкриття ключів

При використанні команд *Get-NtKey* та *New-NtKey* ми можемо вказати шлях Win32 за допомогою Win32Path (листинг 3.17).

Лістинг 3.17. Взаємодія з реєстром за допомогою шляхів Win32

```
PS C:\Users\hladk> Use-NtObject($key = Get-NtKey \REGISTRY\MACHINE\SOFTWARE) { $key.Win32Path }
HKEY_LOCAL_MACHINE\SOFTWARE
PS C:\Users\hladk> Use-NtObject($key = Get-NtKey -Win32Path "HKCU\SOFTWARE") { $key.FullPath }
\REGISTRY\USER\S-1-5-21-2687063813-2248694510-27868876-1001\Software
```

Почнемо з відкриття об'єкта Key за допомогою команди *Get-NtKey*. Ми використовуємо шлях OMNS для відкриття ключа, потім конвертуємо шлях у його версію Win32 за допомогою властивості *Win32Path*. І тут бачимо, що \REGISTRY\ MACHINE\SOFTWARE зіставлений з HKEY_LOCAL_MACHINE\SOFTWARE.

Потім робимо зворотне і відкриваємо ключ за допомогою Win32, вказавши параметр Win32Path і вивівши його власний шлях OMNS. Тут ми використовуємо кущ поточного користувача. Зверніть увагу, що ~~ми~~ використовуємо скорочену форму визначеного імені ключа: HKCU замість HKEY_CURRENT_USER. Всі інші зумовлені ключі мають схожі скорочені форми. Наприклад, HKLM посилається на HKEY_LOCAL_MACHINE.

У вихідних даних можна побачити рядок SDDL SID, який представляє поточного користувача. Як показує цей приклад, використовувати шлях Win32 для доступу до куща поточного користувача набагато простіше, ніж шукати поточного користувача SID і відкривати його за допомогою шляху OMNS.

3.4.2. Список вмісту реєстру

У попередньому розділі ви бачили, як вивести список вмісту реєстру шляхом постачальника диска *NtObject* або *NtKey*. Для реєстру Win32 ви маєте кілька додаткових опцій. Щоб спростити доступ до куща поточного користувача, можна використовувати *NtKeyUser*. Наприклад, можна вивести список програмного ключа поточного користувача за допомогою наступного:

```
PS C:\Users\hladk> ls NtKeyUser:\SOFTWARE
```

PowerShell також має вбудовані диски HKLM і HKCU для локального комп'ютера та поточного користувача відповідно. Наприклад, еквівалентом попередньої команди є наступне:

```
PS C:\Users\hladk> ls HKCU:\SOFTWARE
```

Розробники модуля PowerShell мають ту перевагу, що дозволяють переглядати весь реєстр. Вони також використовують власний API, який використовує обмежені рядки та підтримує використання символів NUL в іменах ключів та значень реєстру. На відміну від цього, API Win32 використовує рядки типу C, що закінчуються символом NUL, який не можуть обробляти вбудовані символи NUL. Тому, якщо NUL вбудований в ім'я, API Win32 не може отримати доступ до цього ключа або значення. Лістинг 3.18 показує це.

Лістинг 3.18. Додавання та доступ до розділу реєстру з символом NUL

```
PS C:\Users\hladk> $key = New-NtKey -Win32Path "HKCU\ABC`0XYZ"
PS C:\Users\hladk> Get-Item "NtKeyUser:\ABC`0XYZ"

Name    TypeName
----    --------
ABCXYZ Key


PS C:\Users\hladk> Get-Item "HKCU:\ABC`0XYZ"
Get-Item : Не удается найти путь "HKCU:\ABCXYZ", так как он не существует.
строка:1 знак:1
+ Get-Item "HKCU:\ABC`0XYZ"
+ ~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : ObjectNotFound: (HKCU:\ABCXYZ:String) [Get-Item], ItemNotFoundException
    + FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.GetItemCommand

PS C:\Users\hladk> Remove-NtKey $key
PS C:\Users\hladk> $key.Close()
```

Лістинг починається зі створення нового ключа із символом NUL в імені, позначеного `0 escape. Якщо отримаєте доступ до цього шляху через *NtKeyUser*, то зможете успішно отримати ключ. Однак, якщо спробуєте це з вбудованим постачальником диска, то це не спрацює.

Така поведінка API Win32 може призвести до проблем безпеки. Наприклад, шкідливий код може приховати ключі та значення реєстру від будь-якого програмного забезпечення, що використовує API Win32, шляхом впровадження символів NUL в ім'я.

Також можливо отримати невідповідність, якщо деяке програмне забезпечення використовує власні системні виклики, а інше програмне забезпечення використовує API Win32. Наприклад, якщо якийсь код перевіряє шлях ABC`0XYZ, щоб переконатися, що він правильно налаштований, а потім передає його іншій програмі, яка використовує цей шлях з API Win32, нова програма натомість отримуватиме доступ до непов'язаного ключа ABC, який не було перевірено. Це може призвести до проблем із розкриттям інформації, якщо вміст ABC буде повернуто службі, що викликала.

Вбудований реєстр також має перевагу: його можна використовувати без встановлення зовнішнього модуля. Він також дозволяє створювати нові ключі та додавати значення.

3.5. ШЛЯХИ ДО ФАЙЛІВ ПРИСТРОЇВ DOS

Ще одна істотна відмінність між API Win32 та вбудованими системними викликами полягає в тому, як вони обробляють шляхи до файлів. У попередньому розділі ми бачили, що ми можемо отримати доступ до змонтованої файлової системи, використовуючи шлях Device\<VolumeName>. Однак ми не можемо вказати цей шлях за допомогою Win32 API. Замість цього ми використовуємо добре відомі шляхи, такі як C:\Windows, які мають літери дисків. Оскільки шляхи з літерами дисків є залишком MS-DOS, ми називаємо їх *шляхами пристроїв DOS*.

Звичайно, API Win32 необхідно передати системному виклику свій шлях, щоб системний виклик працював правильно. API NTDLL *RtlDosPathNameToNtPathName* обробляє цей процес перетворення. Даний API бере шлях пристрою DOS і повертає повністю перетворений власний шлях. Найпростіше перетворення відбувається, коли сторона, що викликає, вказала

повний шлях до диска: наприклад, C:\Windows. У цих випадках процес перетворення просто додає до шляху префікс із зумовленим компонентом шляху \?? для отримання результату \??\C:\Windows.

Шлях \??, також званий префіксом карти пристроїв DOS, зазначає, що диспетчер об'єктів повинен використовувати двоетапний процес пошуку літери диска.

Диспетчер об'єктів спочатку перевірить каталог картки пристроїв DOS для кожного користувача шляхом Sessions\0\ DosDevices \<AUTHID>. Оскільки диспетчер об'єктів спочатку перевіряє розташування кожного користувача, кожен користувач може створювати свої власні зіставлення дисків. Компонент <AUTHID> пов'язаний із сеансом аутентифікації токена зухвалої сторони, його значення є унікальним для кожного користувача. Зверніть увагу, що використання 0 для ідентифікатора сеансу консолі не є друкарською помилкою: всі зіставлення пристроїв DOS розміщуються в одному місці, незалежно від того, в який сеанс консолі увійшов користувач.

Якщо літера диска не знайдена для кожного користувача, диспетчер об'єктів перевірить глобальний каталог GLOBAL??. Якщо його там немає, пошук файлу завершується невдачею. Літера диска — це символічне посилання менеджера об'єктів, яке вказує на підключений том.

Можем це побачити в процесі роботи, використовуючи команду Get-NtSymbolicLink для відкриття літер дисків і відображення їх властивостей. (лістинг 3.19).

Лістинг 3.19. Відображення символічних посилань для дисків C: та Z:

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

FullPath     Target
--------     ------
\GLOBAL??\C: \Device\HarddiskVolume3


PS C:\Users\hladk> Add-DosDevice Z: C:\Windows
PS C:\Users\hladk> Use-NtObject($zdrive = Get-NtSymbolicLink "\??\Z:") { $zdrive | Select-Object FullPath, Target }

FullPath                                  Target
--------                                  ------
\Sessions\0\DosDevices\00000000-001357df\Z: \??\C:\Windows


PS C:\Users\hladk> Remove-DosDevice Z:
```

Спочатку відкриваємо символічне посилання диска C: і відображаємо властивості *FullPath* і *Target*. Повний шлях знаходиться в каталозі \GLOBAL?, а метою є шлях до того. Потім ми створюємо новий диск Z: за допомогою команди *Add-DosDevice*, яка вказує диск на каталог Windows.

Зверніть увагу, що диск Z: доступний в будь-якій програмі користувача, а не тільки в PowerShell. Відображення властивостей диска Z: показує, що він знаходиться у карті пристроїв DOS кожному за користувача, а метою є власний шлях до каталогу Windows. Це показує, що ціль літери диска не обов'язково повинна вказувати безпосередньо, якщо вона врешті-решт туди потрапить (у даному випадку після переходу за символічним посиланням диска C:). Далі, для повноти картини видаляємо диск Z: за допомогою *Remove-DosDevice*.

3.5.1. Типи шляхів

У таб.3.2 показано кілька різних типів шляхів, підтримуваних API Win32, і навіть приклади власних шляхів після перетворення.

Таблиця 3.2.
Типи шляхів Win32

DOS шлях	Native шлях	Опис
some\path	\??\C:\ABC\some\path	Відносний шлях до поточного каталогу
C:\some\path	\??\C:\some\path	Абсолютний шлях
C: some \path	\??\C:\ABC\some\path	Відносний шлях диска
\some\path	\??\C:\some\path	Рутується до поточного диска
\\ C:\some\.\path	\??\C:\path	Шлях пристрою, канонізований
\\?\ C:\some\.\path	\??\C:\some\.\path	Шлях до пристрою, не канонізований
\??\C:\some\path	\??\C:\some\path	Шлях до пристрою, не канонізований
\\server\share\path	\??\UNC\server\share\path	Шлях UNC для спільного використання на сервері

Через спосіб визначення шляхів DOS, кілька шляхів DOS можуть представляти один і той самий власний шлях. Щоб гарантувати правильність остаточного власного шляху, шлях DOS повинен пройти процес канонізації, щоб перетворити ці різні уявлення в ту саму канонічну форму.

Однією простою операцією, яка виконується при канонізації, є обробка роздільників шляху. Для власних шляхів є лише один роздільник шляху, знак зворотної косої межі (\). Якщо ви використовуєте пряму косу рису (/), менеджер об'єктів розглядатиме її як просто ще один символ імені файлу. Проте шляхи DOS підтримують як прямі, і зворотні косі межі як роздільників шляху. Процес канонізації дбає про це, гарантуючи, що всі прямі косі межі перетворюються на зворотні косі межі. Таким чином, C:\Windows та C:/Windows еквівалентні.

Інша операція канонізації - це дозвіл посилань на батьківські каталоги. При написанні шляху DOS можна вказати ім'я файлу з однією крапкою (.) або двома крапками (..), кожна з яких має особливе значення.

Одна крапка відноситься до поточного каталогу, та процес канонізації видалить її зі шляху. Подвійна крапка відноситься до батьківського каталогу, тому батьківський каталог буде видалено. Таким чином, шлях C:\ABC\.\XYZ буде перетворений на C:\ABC\XYZ, а C:\ABC\..\XYZ буде перетворений на C:\XYZ. Як і у випадку з косою межею, власні API не знають про ці спеціальні імена файлів і припускають, що це імена файлу для пошуку.

Якщо шлях DOS починається з \\?\ або \??\, то шлях не канонізується і натомість використовується дослівно, включаючи будь-які посилання на батьківські каталоги або прямі межі. У деяких випадках префікс \??\ може спантеличити API Win32 з поточного кореневого шляху диска, що призведе до відкриття шляху, такого як \??\C:\??\ Path . Незрозуміло, чому Microsoft додала цей тип шляху DOS з огляду на його потенційну плутанину.

Доречно вручну перетворити шлях Win32 у власний шлях за допомогою команди *Get-NtFilePath*. Також можливо перевірити тип шляху за допомогою команди *Get-NtFilePathType*.

Лістинг 3.20. Приклади перетворення шляху до файлу Win32

```
PS C:\Users\hladk> 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
PS C:\WINDOWS>
```

При використанні команди *Get-NtFile* або *New-NtFile* можна використовувати властивості Win32Path, щоб обробляти шлях як шлях Win32 і автоматично перетворювати його.

3.5.2. Максимальна довжина шляху

Максимальна довжина імені файлу, підтримувана Windows, обмежена максимальною кількістю символів, які можна зберегти в UNICODE_STRING (32,767). Однак, API Win32 мають суворіші вимоги.

За замовчуванням, як показано в лістингу 3.21, будь-яка спроба передати шлях, довший за значення MAX_PATH, визначене як 260 символів, завершиться невдачею. Ця поведінка реалізована всередині NTDLL API RtlDosPathNameToNtPathName під час перетворення шляху з Win32 у рідний формат.

Лістинг 3.21. Тестування обмеження шляху Win32 MAX_PATH

```
PS C:\Users\hladk> $path = "C:\$('A'*256)"
PS C:\Users\hladk> $path.Length
259
PS C:\Users\hladk> Get-NtFilePath -Path $path
\??\C:\AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAA
PS C:\Users\hladk> $path += "A"
PS C:\Users\hladk> $path.Length
260
PS C:\Users\hladk> Get-NtFilePath -Path $path
Get-NtFilePath : Исключение при вызове "ResolveWin32Path" с "2" аргументами: "(0xC0000106) - Длина указанной строки
имени слишком велика для предполагаемого применения."
строка:1 знак:1
+ Get-NtFilePath -Path $path
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : NotSpecified: (:) [Write-Error], WriteErrorException
    + FullyQualifiedErrorId : Microsoft.PowerShell.Commands.WriteErrorException,Get-NtFilePath

PS C:\Users\hladk> $path = "\\?\" + $path
PS C:\Users\hladk> $path.Length
264
PS C:\Users\hladk> Get-NtFilePath -Path $path
\??\C:\AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAA
```

Викликаємо API RtlDosPathNameToNtPathName через команду *Get-NtFilePath*. Перший створений шлях має довжину 259 символів, і можемо успішно перетворити його у власний шлях. Потім додаємо до шляху ще один символ, роблячи шлях довжиною 260 символів. Ця спроба завершується помилкою STATUS_NAME_TOO_LONG. Якщо MAX_PATH дорівнює 260,

можна поставити запитання: хіба шлях довжиною в 260 символів не повинен бути успішним? На жаль, ні. API включають завершальний символ NUL як частину довжини, тому максимальна довжина насправді становить лише 259 знаків.

Лістинг 3.21 також показує в який спосіб обійти це обмеження. Якщо додати префікс пристрою до шляху, перетворення буде успішним, навіть якщо довжина шляху тепер становить 264 символи. Це відбувається тому, що префікс замінюється на префікс пристрою DOS \??\, а шлях, що залишився, залишається дослівним. Хоча цей метод працює, зверніть увагу, що він також вимикає корисні функції, такі як канонізація шляху. У поточних версіях Windows є спосіб вибору довгих імен файлів, як показано в лістингу 3.22.

Лістинг 3.22. Перевірка та тестування довгих програм, що враховують шляхи

```
PS C:\Users\hladk> $path = "HKLM\SYSTEM\CurrentControlSet\Control\FileSystem"
PS C:\Users\hladk> Get-NtKeyValue -Win32Path $path -Name "LongPathsEnabled"

Name             Type  DataObject
----             ----  ----------
LongPathsEnabled Dword 0


PS C:\Users\hladk> (Get-Process -Id $pid).Path | Get-Win32ModuleManifest | Select-Object LongPathAware

LongPathAware
-------------
         True


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

Перше, що робимо в цьому лістингу, це перевіряємо, що значення реєстру *LongPathsEnabled* встановлено на 1. Значення має бути встановлене на 1 до початку процесу, оскільки воно буде прочитане лише один раз під час ініціалізації процесу. Однак простого включення функції довгого шляху недостатньо: файл процесу, що виконується, повинен включитися, вказавши властивість маніфесту.

Можемо встановити цю властивість, використовуючи команду *Get-ExecutableManifest* і вибравши *LongPathAware*. Слід зазначити, що у PowerShell ця опція маніфесту дорівнює 1. Тепер можемо успішно перетворювати набагато більш довгі шляхи, як показано з шляхом з 303 символів.

Виникає питання, чи є довгі шляхи проблемою безпеки.

Проблеми безпеки часто виникають у місцях, де є межа інтерфейсу. У цьому випадку той факт, що файлова система може підтримувати довгі шляхи, може призвести до невірного припущення, що шлях до файлу ніколи не може бути довшим за 260 символів. Можлива проблема може виникнути, коли програма запитує повний шлях до файлу, а потім копіює цей шлях у буфер пам'яті з фіксованим розміром 260 символів. Якщо довжина шляху до файлу не перевіряється заздалегідь, ця операція може призвести до пошкодження пам'яті після буфера, що може дозволити зловмисникові отримати контроль над виконанням програми.

3.6. СТВОРЕННЯ ПРОЦЕСУ

Процеси є основним способом виконання компонентів режиму користувача та їх ізоляції з метою безпеки, тому важливо, щоб ми докладно вивчили, як їх створювати. У попередньому розділі згадували, що можливо створити процес за допомогою системного виклику *NtCreateUserProcess* . Однак більшість процесів не буде створено безпосередньо за допомогою цього системного виклику. Натомість вони будуть створені за допомогою Win32 *CreateProcess* API, що діє як оболонка.

Системний виклик нечасто використовується безпосередньо, оскільки більшості процесів необхідно взаємодіяти з іншими компонентами режиму користувача. Особливо це стосується CSRSS, для взаємодії з робочим столом користувача. CreateProcess API зареєструє новий процес, створений системним викликом, з відповідними службами, необхідні для правильної ініціалізації.

3.6.1. Аналіз командного рядка

Найпростіший спосіб створити новий процес - вказати рядок, що представляє файл, необхідний для запуску. Потім API CreateProcess проаналізує командний рядок, щоб знайти виконуваний файл для передачі ядру.

Щоб протестувати цей розбір командного рядка, давайте створимо новий процес за допомогою команди PowerShell *New-Win32Process*, яка виконує CreateProcess.

Для цього можна використовувати вбудовану команду, таку як *Start-Process*, але New-Win32Process корисніший, оскільки він розкриває повний набір функцій API CreateProcess. Можемо запустити процес за допомогою наступної команди:

```
PS C:\Users\hladk> $proc = New-Win32Process -CommandLine "notepad test.txt"
```

Вказуємо командний рядок, що містить ім'я виконуваного файлу, який потрібно запустити, Notepad, та ім'я файлу, який потрібно відкрити, *test.txt*. Цей рядок не обов'язково повинен містити повний шлях до файлу, що виконується. Команда *New-Win32Process* проаналізує командний рядок, щоб спробувати відрізнити ім'я вихідного файлу образу від файлу що відкривається. Це не такий простий процес, як здається.

Перше, що зробить *New-Win32Process*, — це проаналізує командний рядок за допомогою алгоритму, який розділяє пробіли, якщо вони не взяті в подвійні лапки. У цьому випадку він розбере командний рядок на два рядки: notepad і *test.txt*. Потім команда бере перший рядок і намагається знайти відповідний процес. Однак є невелика складність: не існує виконуваного файлу notepad, є тільки notepad.exe. Хоча це не є обов'язковим, виконувані файли Windows зазвичай мають розширення .exe, тому алгоритм пошуку автоматично додає це розширення, якщо його ще немає.

Потім команда виконає пошук виконуваного файлу в наступних місцях, що дуже схоже на пошук шляху DLL, який ми обговорювали в розділі "Пошук DLL".

Зверніть увагу, що шлях пошуку файлу, що виконується, збігається з небезпечним шляхом пошуку DLL:

1. Той самий каталог, що й виконуваний файл поточного процесу;
2. Поточний робочий каталог;
3. Каталог Windows System32;
4. Каталог Windows;
5. Кожне розділене точкою з комою розташування в змінному середовищі PATH.

Якщо *New-Win32Process* не може знайти notepad.exe, він спробує знайти файл notepad test.txt. Оскільки ім'я файлу вже має розширення, він не замінить його на .exe. Якщо *New-Win32Process* не може знайти файл, він повертає помилку. Зверніть увагу, що якщо передати блокнот, укладений у подвійні лапки, як у "notepad" test.txt, то *New-Win32Process* буде шукати лише notepad. exe і ніколи не повернеться до спроб всіх комбінацій імені з пробілами.

Така поведінка аналізу командного рядка має два наслідки для безпеки.

По-перше, якщо процес створюється більш привілейованим процесом, а менш привілейований користувач може записати файл у місце, розташоване раніше у списку пошуку шляху, процес може бути перехоплений.

По-друге, наслідок безпеки полягає в тому, що алгоритм пошуку шляху змінюється, якщо перше значення містить роздільник шляху. У цьому випадку замість використання правил пошуку шляху *New-Win32Process* розбиває шлях по пробілах, а потім пробує кожен компонент, якби це був шлях, шукаючи ім'я або з розширенням .exe або без нього.

Розглянемо приклад. Розглянемо приклад. Якщо ми зазначемо командний рядок C:\Program Files \abc.exe, то будуть перевірені наступні шляхи для пошуку виконуваного файлу::

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

Якщо користувач зможе записати файл C:\Program або C:\Program.exe, він зможе перехопити виконання. На щастя, при установці Windows за замовчанням звичайний користувач не може записувати файли до кореня системного диска. Однак деякі конфігурації іноді дозволяють це. Крім того, виконуваний шлях може бути на іншому диску, який дозволяє запис в корінь.

Щоб уникнути обох наслідків безпеки, що викликають може вказати повний шлях виконуваного файлу, встановивши властивість *ApplicationName* під час виклику *New-Win32Process*:

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

Якщо ми задаємо шлях таким чином, команда буде передавати його без змін новому процесу.

3.6.2. API оболонки

Якщо ви двічі натиснете на файл, який не є виконуваним, наприклад текстовий документ, в **Провіднику**, він запустить для вас редактор. Однак, якщо ви спробуєте запустити документ за допомогою New-Win32Process, ви отримаєте помилку, показану тут:

```
PS C:\Users\hladk> New-Win32Process -CommandLine "document.txt"
Исключение при вызове "Create" с "0" аргументами: "(0x80070002) - Не удается найти указанный файл."
C:\Users\hladk\Documents\WindowsPowerShell\Modules\NtObjectManager\2.0.1\NtObjectManager.psm1:18460 знак:5
+      $p = $config.Create()
+      ~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : NotSpecified: (:) [], MethodInvocationException
    + FullyQualifiedErrorId : NtException
```

Ця помилка вказує на те, що текстовий файл не є допустимою програмою Win32.

Причина, через яку Explorer може запустити редактор, полягає в тому, що він не використовує базовий API CreateProcess безпосередньо. Натомість він використовує API оболонки. Основна API оболонка, яка використовується для запуску редактора для файлу - *ShellExecuteEx*, що реалізована у бібліотеці SHELL32. Цей API і його більш проста версія *ShellExecute* надто складні, щоб докладно описувати їх тут. Натомість дамо лише короткий огляд останнього.

Вказуємо три параметри для ShellExecute:

— Шлях до файлу для виконання;

— Дієслово для використання у файлі;

— Будь-які додаткові аргументи.

Перше, що робить ShellExecute , це шукає обробник для розширення файлу для виконання. Наприклад, якщо файл — test.txt, йому потрібно знайти обробник для розширення .txt. Обробники реєструються в реєстрі під ключом HKEY_CLASSES_ROOT. У лістингу 3.23 запитуємо обробник.

Почнемо із запиту ключа класу машини щодо розширення *.txt.*

Хоча можна було б перевірити наявність ключа, специфічного користувача, перевірка ключа класу машини гарантує, що перевіримо системне значення за замовчанням. Ключ реєстру .txt безпосередньо не містить обробник. Натомість значення за замовчанням, представлене порожнім ім'ям, посилається на інший ключ: у цьому випадку на txtfile. Потім перераховуємо субключі txtfile і знаходимо: open, print та printto. Буде доцільно передати ці дієслова на ім'я ShellExecute.

Лістинг 3.23. Запит обробника оболонки для файлів .txt

```
PS> $base_key = "NtKey:\MACHINE\SOFTWARE\Classes"
PS> Get-Item "$base_key\.txt" | Select-Object -ExpandProperty Values
Name                Type       DataObject
----                ----       ----------
Content Type        String     text/plain
PerceivedType       String     text
                    String     txtfile
PS> Get-ChildItem "$base_key\txtfile\Shell" | Format-Table
Name    TypeName
----    --------
open    Key
print   Key
printto Key
PS> Get-Item "$base_key\txtfile\Shell\open\Command" |
Select-Object -ExpandProperty Values | Format-Table
Name   Type       DataObject
----   ----       ----------
       ExpandString %SystemRoot%\system32\NOTEPAD.EXE %1
```

Кожене із цих ключів дієслів може мати субключ, званий *Command*, який містить командний рядок для виконання. Бачимо, що за замовчанням, для файлу .txt відкривається Notepad. %1 замінюється на шлях до виконуваного файлу. (Команда також може містити %*, що включає будь-які додаткові аргументи, передані ShellExecute). API CreateProcess тепер може запустити виконуваний файл та обробити його.

Існує багато різних стандартних дієслів, які можна передати до *ShellExecute*.

У таб.3.3 наведено список найпоширеніших з них.

Може здатися дивним, що є як дієслово **open**, так і дієслово **edit**. Наприклад, якщо відкрили файл .txt, файл відкриється в Блокноті, і зможете його редагувати. Але ця відмінність корисна для таких файлів, як пакетні файли, де дієслово відкриття запустить файл, а дієслово редагування відкриє його у текстовому редакторі.

Таблиця 3.3.
Поширені дієслова-оболонки

Verb	Опис
open	Відкрийте файл. Зазвичай це значення за замовчанням.
edit	Відредагуйте файл.
print	Роздрукуйте файл.
printto	Роздрукувати на вказаному принтері.
explore	Ознайомлення з каталогом. Команда використовується для відкриття каталогу у вікні провідника.
runas	Відкрити файл від імені адміністратора. Як правило, визначено тільки для виконуваних файлів.
runasuser	Відкрити файл як інший користувач. Як правило, визначено тільки для виконуваних файлів.

Щоб використовувати ShellExecute з PowerShell, можна запустити команду *Start-Process*. За замовчуванням ShellExecute буде використовувати дієслово **open**, але можете вказати своє власне дієслово за допомогою Verb. У наступному коді друкуємо файл **.txt** як адміністратор за допомогою дієслова **print**:

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

Зміни дієслів також можуть підвищити безпеку. Наприклад, скрипти PowerShell з розширенням **.ps1** мають зареєстрований дієслово **open**. Проте клік по скрипту відкриє файл скрипта в Блокноті, а не виконає його. Тому, якщо двічі клікнути файл скрипта у Провіднику, він не виконається. Замість цього необхідно викликати контекстне меню і явно вибрати команду *Запустити* з PowerShell.

3.7. СИСТЕМНІ ПРОЦЕСИ

Ядро може вирішувати деякі з таких завдань, як: очікування автентифікації, управління обладнанням та зв'язок по мережі. Проте, написання коду ядра складніше, ніж коду режиму користувача, з ряду причин: ядро не має такого широкого спектра доступних API, його ресурси обмежені, особливо у плані пам'яті. Будь-яка помилка кодування може призвести до збою системи або вразливості безпеки.

Щоб уникнути цих проблем, Windows запускає різні процеси поза режимом ядра з високим рівнем привілеїв.

3.7.1. Диспетчер сеансів

Підсистема диспетчера сеансів (SMSS) – це перший процес режиму користувача, який запускає ядро після завантаження. Він відповідає за налаштування робочого середовища для наступних процесів. Деякі з його обов'язків включають :

- Завантаження відомих DLL та створення об'єктів Section;
- Запуск процесів підсистеми, таких як CSRSS;
- Ініціалізація базових пристроїв DOS, таких як послідовні порти;
- Запуск автоматичних перевірок цілісності диска.

3.7.2. Процес входу до Windows

Процес входу до Windows відповідає за налаштування нового сеансу консолі, а також за відображення входу інтерфейсу користувача (насамперед через додаток LogonUI). Він також відповідає за запуск процесу драйвера шрифтів режиму користувача (UMFD), який відображає шрифти на екрані та процесу диспетчера вікон робочого столу (DWM), який виконує операції з компонування робочого столу, щоб забезпечити красиві прозорі вікна та сучасні штрихи графічного інтерфейсу .

3.7.3. Підсистема локальною безпеки

Ми вже згадували LSASS у контексті SRM. Проте необхідно підкреслити його важливу роль в аутентифікації. Без LSASS користувач не зміг би увійти до системи.

3.7.4. Диспетчер управління службами

Диспетчер управління службами (SCM) відповідає за запуск більшості привілейованих системних процесів у Windows. Він керує цими процесами, необхідними службами, і може запускати та зупиняти їх за потреби. Наприклад, SCM може запустити службу на основі певних умов, таких як доступність мережі.

Кожна служба є ресурсом з детальним контролем, який визначає, які користувачі можуть маніпулювати її станом. За замовчуванням тільки адміністратор може маніпулювати службою. Нижче наведено деякі з найбільш важливих служб, що працюють у будь-якій системі Windows:

Підсистема віддаленого виклику процедур (RPCSS). Служба RPCSS керує реєстрацією кінцевих точок віддаленого виклику процедур, надаючи реєстрацію як локальним клієнтам, так і по мережі, системі, справі. Вихід з ладу цього процесу змусить Windows перезавантажитися.

Запуск процесу сервера DCOM. Запуск процесу сервера DCOM є аналогом RPCSS (і раніше був частиною тієї ж служби).

Він використовується для запуску процесів сервера Component Object Model (COM) від імені локальних або віддалених клієнтів.

Планувальник завдань. Можливість запланувати виконання певної дії на певний час та дату є корисний функцією операційної системи. Наприклад, щоб переконатися, що видаляєте невикористовувані файли за певним розкладу. Можна налаштувати дію за допомогою служби Планувальника завдань.

Інсталятор Windows. Цю службу можна використовувати для встановлення нових програм та функцій. Працюючи як привілейована служба, буде дозволено установку і зміну зазвичай захищених місцях файловий системи.

Windows Update. Наявність повністю оновленою операційної системи має вирішальне значення для безпеки Windows. Коли Microsoft випускає нові виправлення безпеки, їх слід встановлювати як можна швидше. Щоб не вимагати від користувача перевірки оновлень, ця служба працює у фоновому режимі, періодично прокидаючись та перевіряє в Інтернет наявність нових виправлень.

Інформація про програму. Ця служба надає механізм перемикання між адміністратором та користувачем без прав адміністратора на одному робочому столі. Ця функція зазвичай називається контролем облікових записів користувачів (UAC). Можна запустити процес адміністратора, використовуючи дієслово **runas** з API оболонки. Розглянемо, як UAC працює.

Можемо запросити статус усіх служб, що контролюються SCM, використовуючи різні інструменти. PowerShell має вбудовану команду *Get-Service*. Однак модуль PowerShell, надає більш повну команду Get-*Win32Service*, яка може перевіряти налаштовану безпеку служби, а також додаткові властивості, що не відображаються за допомогою команди за замовчанням.

У лістингу 3.24 показано, як запросити всі поточні служби .

Лістинг 3.24. Відображення всіх служб за допомогою *Get-Win32Service*

```
PS C:\Users\hladk> Get-Win32Service

Name                                    Status      ProcessId
----                                    ------      ---------
AarSvc                                  Stopped     0
ABBYY.Licensing.FineReader.16.0         Running     7288
ALG                                     Stopped     0
AppIDSvc                                Stopped     0
Appinfo                                 Running     12068
AppReadiness                            Stopped     0
AppXSvc                                 Running     12548
ApxSvc                                  Stopped     0
aswbIDSAgent                            Running     16424
AudioEndpointBuilder                    Running     2584
Audiosrv                                Running     2008
autotimesvc                             Stopped     0
avast! Antivirus                        Running     4456
avast! Firewall                         Running     6508
avast! Tools                            Running     5064
AvastWscReporter                        Running     3748
AxInstSV                                Stopped     0
BcastDVRUserService                     Stopped     0
BDESVC                                  Running     1628
BFE                                     Running     5164
BITS                                    Running     12884
BluetoothUserService                    Stopped     0
BrokerInfrastructure                    Running     1468
BTAGService                             Running     1932
BthAvctpSvc                             Running     2000
```

Висновок показує ім'я служби, її статус (Зупинена або Працює), та якщо вона запущена, ідентифікатор процесу служби. Якщо перерахувати властивості служби за допомогою *Format-List*, то можна побачити додаткову інформацію, як повний опис служби .

3.8. ПРАКТИЧНІ ПРИКЛАДИ

Розглянемо кілька практичних прикладів, щоб на практиці застосувати різні команди, розглянуті в цьому розділі, для дослідження безпеки або аналізу систем.

3.8.1. Пошук виконуваних файлів, які імпортують певні API

Вже розглянуто, як використовувати команду *Get-Win32ModuleImport* для перегляду імпортованих API виконуваного файлу. Одне із застосування цієї команди, яке є особливо корисним, коли намагаються відстежити проблеми безпеки, полягає в тому, щоб визначити всі виконувані файли, які використовують певний API, наприклад *CreateProcess*, а потім можна використовувати цей список для зменшення файлів, які потрібні для зворотного проектування. Такий пошук можна виконати за допомогою базового сценарію PowerShell, показаного в лістингу 3.25.

Лістинг 3.25. Пошук виконавчих файлів, які імпортують *CreateProcess*

```
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
```

Розпочинаємо з перерахування всіх файлів .exe в каталозі Windows.

Для кожного виконавчого файлу викликаємо команду *Get-Win32ModuleImport*. Це завантажить модуль і розбере його імпорт. Можна вимагати відносно велику кількість ресурсів, тому найкраще зафіксувати результати в змінній.

Далі вибираємо лише ті імпортовані файли, які містять API *CreateProcessW*. Властивість *Names* - це список, що містить імпортовані імена для однієї DLL. Щоб отримати кінцевий список виконавич файлів, які імпортують певний API, можемо вибрати властивість *ModulePath*, яка містить початкове завантажене ім’я шляху.

Можна використовувати ту саму техніку для перерахування файлів DLL або драйверів та швидкого виявлення цілей для зворотнього проектування.

3.8.2. Пошук прихованих ключів або значень реєстру

Одна з великих переваг використання власних системних викликів над API Win32 для взаємодії з реєстром полягає в тому, що вони дозволяють отримувати доступ до ключів і значень із символами NUL у їхніх іменах. Було б корисно мати можливість знайти ці ключі та значення, щоб можна було спробувати виявити програмне забезпечення у системі, яке активно намагається приховати ключі або значення реєстру від користувача (відомо, що деякі сімейства зловмисних програм, такі як **Kovter** і **Poweliks**, використовують цю техніку).

Почнемо з пошуку ключів із символами NUL в імені (лістинг 3.26).

Лістинг 3.26. Пошук прихованих ключів реєстру

```
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()
```

Спочатку ми створимо ключ у поточній гілці користувача з символом NUL. Якщо ви спробуєте знайти цей ключ за допомогою вбудованого постачальника реєстру, це не вдасться.

Замість цього ми виконуємо рекурсивний аналіз поточної гілки користувача і вибираємо всі ключі, які мають символ NUL в імені. У вихідних даних ви можете побачити, що прихований ключ був виявлений.

Щоб знайти приховані значення, ми можемо запитати список значень ключа, перелічивши його властивість Values. Кожне значення містить ім'я ключа та значення даних (лістинг 3.27).

Лістинг 3.27. Пошук прихованих значень реєстру

```
PS> $key = New-NtKey -Win32Path "HKCU\SOFTWARE\ABC"
PS> Set-NtKeyValue -Key $key -Name "`0HIDDEN" -String "HELLO"
PS> function Select-HiddenValue {
                        [CmdletBinding()] param( [parameter(ValueFromPipeline)] $Key )
                        Process {
                        foreach($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()
```

Починаємо зі створення звичайного ключа, а потім додаємо значення з символом NUL в імені. Потім визначаємо функцію *Select-HiddenValue*, яка перевірить ключі в конвеєрі та вибере будь-яке значення з символом NUL в назві, повернувши спеціальний об’єкт у конвеєр.

Далі рекурсивну перераховуємо гілку поточного користувача та фільтруємо ключі за допомогою функції *Select-HiddenValue*. У вихідних даних можна побачити, що виявлено приховане значення.

3.9. ВИСНОВКИ ДО РОЗДІЛУ 3

У цьому розділі було надано короткий огляд компонентів режиму користувача Windows. Розпочали з вивчення API Win32 та завантаження DLL-файлів.

Розуміння цієї теми є важливим, оскільки воно розкриває, як програми користувацького режиму взаємодіють з ядром і реалізують загальні функції.

Далі наведено огляд графічного інтерфейсу Win32, включаючи опис окремої таблиці системних викликів, що використовується для WIN32K, який є компонентом режиму ядра підсистеми Win32.

Після цього ознайомили читачів з типами об'єктів вікна станції та робочого столу і описали призначення сеансу консолі, а також те, як сеанс відповідає робочому столу, який ви бачите як користувач.

Потім повернулися до теми Win32 API, детально описавши відмінності та подібності між Win32 API (у нашому випадку *CreateMutexEx*) та базовим системним викликом (*NtCreateMutant*). Це обговорення мало б дати вам краще розуміння того, як Win32 API взаємодіють з рештою операційної системи.

Ознайомив вас з відмінності між шляхами до пристроїв DOS і системними шляхами, як їх розуміє системний виклик. Ця тема є важливою для розуміння того, як програми в режимі користувача взаємодіють з файловою системою.

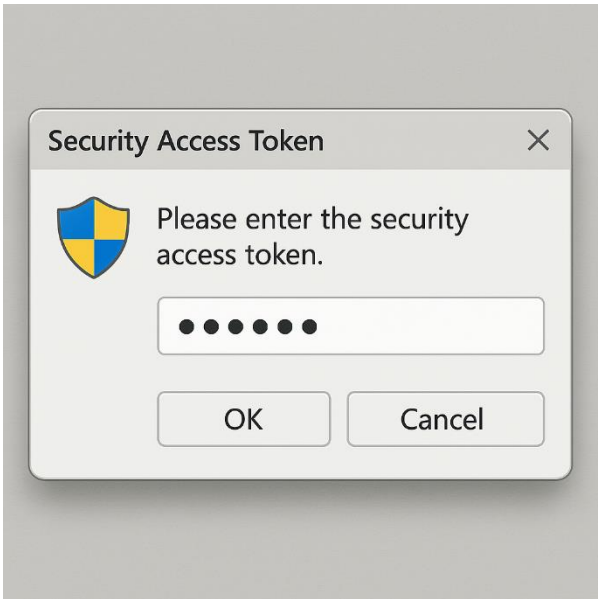
На завершення обговорили кілька тем, пов'язаних із процесами та потоками Win32, розглянувши API, що використовуються для створення процесів безпосередньо або через оболонку, та надавши огляд відомих системних

процесів. У наступних розділах ми зосередимося на тому, як Windows реалізує безпеку за допомогою SRM.

ПИТАННЯ ДЛЯ САМОКОНТРОЛЮ

1. За допомогою якого Win32 API можна завантажити нову бібліотеку?
2. В якому порядку API шукала файли. Які проблеми з цим порядком. Як вони були вирішені?
3. Чи можна вручну надсилати повідомлення вікна за допомогою API?
4. Який об'єкт створює ядро для відстеження ресурсів?
5. В чому полягає атака типу «shatter»?
6. Яку проблему вирішує Windows створюючи окремий каталог BNO для кожного консольного сеансу?
7. Наведіть найпростіший спосіб створити новий процес.
8. За що відповідає Диспетчер управління службами (SCM)?
9. Як здійснити пошук виконуваних файлів, які імпортують певні API?
10. Як здійснити пошук прихованих ключів або значень реєстру?

РОЗДІЛ 4. ТОКЕНИ ДОСТУПУ



Токен безпеки доступу (**security access token**), або коротко ***токен***, є основою безпеки Windows. SRM використовує токени для відображення ідентифікаторів, таких як облікові записи користувачів, а потім надає їм доступ до ресурсів або відмовляє в ньому. Windows працює з токенами за допомогою об'єктів ядра **Token**.

Як і інші об'єкти ядра, токени підтримують системні виклики *Query* та *Set information*, які дозволяють користувачеві як перевіряти властивості токена так і встановлювати певні властивості. Деякі API Win32 також надають ці системні виклики *Set* та *Query*, наприклад, *GetTokenInformation* та *SetTokenInformation*.

Почнемо з огляду двох основних типів токенів, з якими ви зіткнетеся під час аналізу безпеки системи Windows: *первинних токенів* (primary token) та токенів *імперсонації* (impersonation token). Потім ми детально розглянемо важливі властивості, які містить токен. Вам потрібно буде зрозуміти їх, перш ніж ми зможемо обговорити перевірку доступу в наступних розділах.

4.1. ПЕРВИННІ ТОКЕНИ

Кожному процесу призначено токен, який описує його особливість для будь-якої операції доступу до ресурсів. Коли SRM виконує перевірку доступу, він запитує токен процесу та використовує відповідь для визначення того, який тип доступу надати. Коли токен використовується для процесу, він називається *первинним* токеном.

Можна відкрити токен процесу за допомогою системного виклику *NtOpenProcessToken*, який поверне дескриптор, який слід використовувати для запиту інформації про токен.

Оскільки об'єкт токена є ресурсом, який захищається, той що викликає повинен пройти перевірку доступу, щоб отримати дескриптор. Зверніть увагу, також потрібний дескриптор процесу з доступом *QueryLimitedInformation*, щоб мати можливість запросити токен.

При відкритті об'єкта Token можна запитати такі права доступу:

AssignPrimary - Призначає об'єкт Token як основний токен;

Duplicate - Дублює об'єкт Token**;**

Impersonate - Імперсоналізує об'єкт Token;

Query - Запитує властивості об'єкта Token, такі як його групи та привілеї;
QuerySource - Запитує джерело об'єкта Token;
AdjustPrivileges - Налаштовує список привілеїв об'єкта Token;
AdjustGroups - Налаштовує список груп об'єкта Token;
AdjustDefault - Налаштовує властивості об'єкта Token, які не охоплені іншими правами доступу;
AdjustSessionId - Налаштовує ідентифікатор сеансу об'єкта Token.

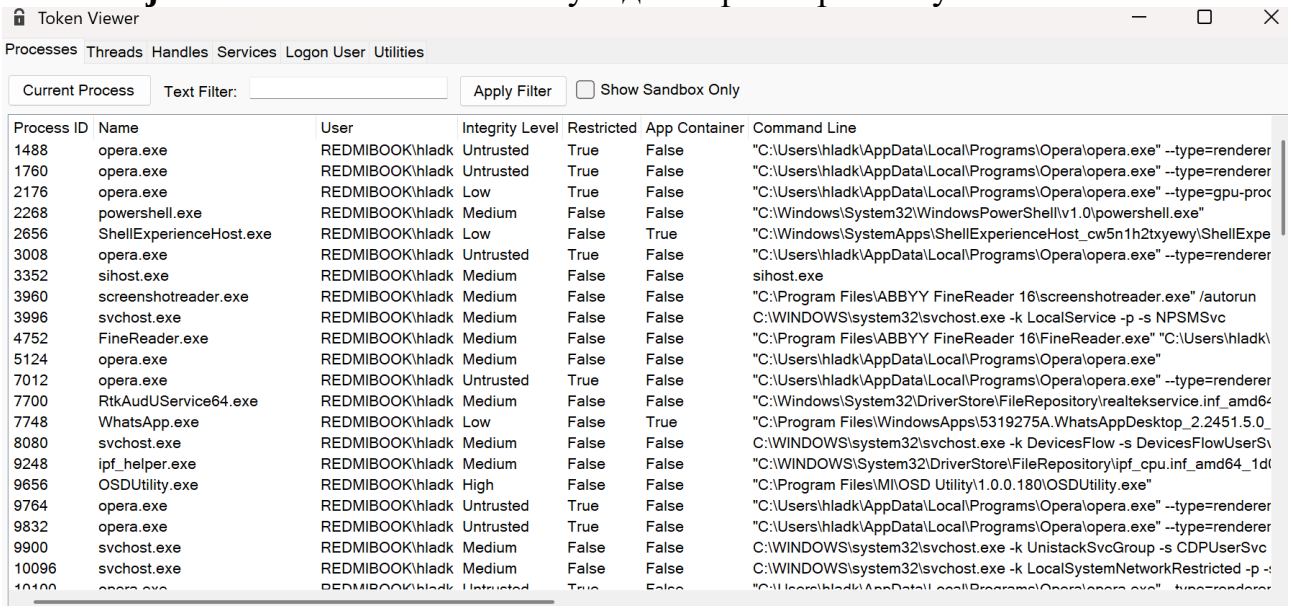


Рисунок 4.1. Засіб перегляду токенів. Відображає список усіх доступних процесів та їх токенів.

Ви можете переглянути список доступних процесів та їх токенів, виконавши команду PowerShell *Show-NtToken -All*. Ця команда має відкрити програму Token Viewer, як показано на рис. 4.1.

Список надає лише простий огляд доступних токенів. Якщо ви хочете побачити більше інформації, двічі натисніть на один із записів процесу, щоб відкрити детальний перегляд токена, як показано на рис. 4-2.

Виділимо кілька важливих елементів інформації в цьому вікні. У верхній частині знаходяться ім'я користувача та SID. Об'єкт Token зберігає тільки SID, але вікно токена відображає ім'я, якщо воно доступне. Наступне поле вказує тип токена. Оскільки ми перевіряємо первинний токен, тип встановлено як Primary. Елемент «Рівень інперсонації» використовується тільки для токенів інперсонації, про які ми поговоримо в наступному розділі. Він не потрібен для первинних токенів, тому встановлено значення N/A.

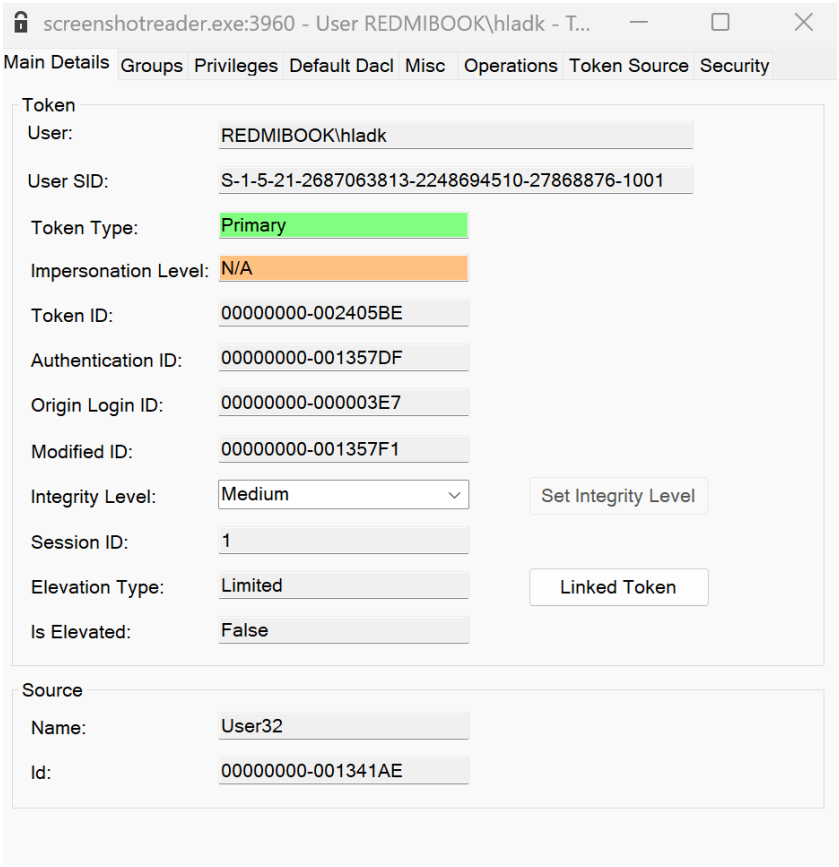


Рисунок 4.2. Детальний вигляд об'єкту Token процесу

У середині діалогового вікна знаходиться список з чотирьох 64-бітних цілочисельних ідентифікаторів:

Token ID - ідентифікатор токена. Унікальне значення, призначене під час створення об'єкта токен.

Authentication ID - ідентифікатор аутентифікації. Значення, що вказує сеанс входу, до якого належить токен.

Origin Login ID - ідентифікатор входу до системи. Ідентифікатор автентифікації батьківського сеансу входу.

Modified ID - унікальне значення, яке оновлюється при зміні певних значень токена.

LSASS створює сеанс входу до системи, коли користувач проходить автентифікацію на комп'ютері під керуванням Windows.

Сеанс входу в систему відстежує ресурси, пов'язані з автентифікацією, наприклад, він зберігає копію облікових даних користувача, щоб їх можна було використовувати повторно.

Під час створення сеансу входу в систему SRM генерує унікальне значення ідентифікатора автентифікації, яке можна використовувати для посилання на сеанс. Отже, для даного сеансу входу в систему всі токени користувача будуть мати той самий ідентифікатор аутентифікації. Якщо користувач проходить автентифікацію двічі на тому самому комп'ютері, SRM згенерує різні ідентифікатори автентифікації.

Ідентифікатор походження входу вказує, хто створив сеанс входу токена. Якщо автентифікуєте іншого користувача на своєму робочому столі (наприклад, викликаючи *API LogonUser* з ім'ям користувача та паролем), то вихідний ідентифікатор походження буде ідентифікатором автентифікації токена. Зауважте, що це поле на рис. 4.2 показує значення 00000000-000003E7. Це один із чотирьох фіксованих ідентифікаторів автентифікації, визначених SRM, що в даному випадку вказує на сеанс входу SYSTEM. Таб. 4.1 показує чотири фіксовані значення разом із SID для облікових записів користувачів, пов'язаних із сеансами.

Таблиця 4.1.

Ідентифікатори аутентифікації та SID користувача для фіксованих сеансів входу до системи

Ідентифікатор автентифікації	SID користувача	Ім’я користувача сеансу входу
00000000-000003E4	S-l-5-20	NT AUTHORITY\NETWORK SERVICE
00000000-000003E5	S-l-5-19	NT AUTHORITY\LOCAL SERVICE
00000000-000003E6	S-l-5-7	NT AUTHORITY\ANONYMOUS LOGON
00000000-000003E7	S-l-5-18	NT AUTHORITY\SYSTEM

Після ідентифікаторів у докладному поданні знаходиться поле, що вказує на рівень цілісності токена. Windows Vista вперше додала рівень цілісності для реалізації простого механізму обов'язкового контролю доступу. За допомогою такого контролю загальносистемні політики забезпечують доступ до ресурсів і не дозволяють окремому ресурсу вказувати свій доступ.

За цим слідує ідентифікатор сеансу, номер, призначений сеансу консолі, до якого приєднано процес. Незважаючи на те, що сеанс консолі є властивістю процесу значення вказується в токені процесу.

Ми згадували, що ідентифікатори токена – це 64-бітові цілі числа. Технічно, це структури локально унікальних ідентифікаторів (**LUID**), що містять два 32-бітові значення.

LUID – це загальний системний тип, і SRM використовує їх, коли йому потрібне унікальне значення.

Наприклад, вони використовуються для унікальної ідентифікації значень привілеїв.

Ви можете призначити свій власний **LUID**, використовуючи системний виклик *NtAllocateLocallyUniqueId* або виконавши команду PowerShell *Get-NtLocallyUniqueId.*

Коли ви використовуєте системний виклик, Windows гарантує, що він має центральний орган для генерації наступного унікального ідентифікатора. Це важливо, оскільки повторне використання значення може мати катастрофічні наслідки. Наприклад, якщо **LUID** був повторно використаний як ідентифікатор аутентифікації для токена, він може перекриватися з одним із ідентифікаторів, визначених у таб.4.1.

Це може змусити систему думати, що до ресурсу звертається користувач, який має великі привілеї.

Графічний інтерфейс Token Viewer чудово підходить, якщо є потреба вручну перевірити інформацію токена. Для програмного доступу є сенс відкрити об'єкт Token у PowerShell за допомогою команди *Get-NtToken*.

```
PS C:\Users\hladk> $token = Get-NtToken
```

За необхідності відкрити токен для певного процесу, можна використовувати цю команду, замінивши <PID> на ідентифікатор цільового процесу:

```
PS C:\Users\hladk> $token = Get-NtToken -ProcessId <PID>
```

Результатом команди *Get-NtToken* є об'єкт Token, властивості якого слід запросити. Наприклад, можна відобразити користувача токена, як показано в лістингу 4.1.

Лістинг 4.1. - Відображення користувача через властивості об'єкта Token

```
PS> $token.User

Name              Attributes
----              ----------
GRAPHITE\user     None
```

Використовуйте команду *Format-NtToken* для виведення базової інформації на консоль, як показано у лістингу 4.2.

Лістинг 4.2. - Відображення властивостей токена за допомогою *Format-NtToken.*

```
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--
```

Передати відкритий об'єкт Token у *Show-NtToken* можливо для відображення того ж графічного інтерфейсу, що показано на рис. 4.2.

4.2. ТОКЕНИ ІМПЕРСОНАЦІЇ

Інший тип токена - це токен імперсонації (*impersonation token*). Токени імперсонації більш важливі для системних служб, оскільки вони дозволяють процесу з однією ідентифікацією тимчасово уособлювати іншу ідентифікацію для перевірки доступу. Наприклад, службі може знадобитися відкрити файл, який належить іншому користувачеві, під час виконання будь-якої операції. Дозволяючи цій службі імітувати користувача, що викликає її, система надає їй доступ до файлу, навіть якщо служба не може відкрити файл безпосередньо.

Токени уособлення призначаються потокам, а не процесам. Це означає, що тільки код, що працює у цьому потоці, прийме уособлену ідентифікацію.

Існує три способи призначення токена імперсонації потоку:

• Явно надавши об'єкту Token доступ *Impersonate* та об'єкту Thread доступ *SetThreadToken;*

• Явно надавши об'єкту Thread доступ *DirectImpersonation*;

• Неявно, шляхом імітації запиту RPC.

Швидше за все, ви зіткнетеся з неявним присвоєнням токена, оскільки це найпоширеніший випадок для системних служб, які використовують механізми RPC. Наприклад, якщо служба створює сервер іменованого каналу, вона може імітувати клієнтів, які підключаються до каналу, за допомогою API *ImpersonateNamedPipe*. Коли виконується виклик на іменованому каналі, ядро фіксує контекст імперсонації на основі потоку та процесу, що виконують виклик. Цей контекст імперсонації використовується для призначення токена імперсонації потоку, що викликає *ImpersonateNamedPipe*.

Контекст імперсонації може базуватися на існуючому токені імперсонації в потоці або на копії основного токена процесу.

4.2.1. Security Quality of Service (SQoS)

SRM підтримує функцію під назвою Quality of Service Security (SQoS). Відкриваючи іменований канал за допомогою API файлової системи, можна передати структуру SECURITY_QUALITY_OF_SERVICE у поле *SecurityQualityOfService* структури OBJECT_ATTRIBUTES.

Структура SQoS містить три значення конфігурації: рівень імперсонації, режим відстеження контексту та ефективний режим токена.

Рівень імперсонації в SQoS є найважливішим полем для керування тим, що служба може робити з вашою особистістю. Він визначає рівень доступу, який надається службі, коли вона неявно видає себе за того хто викликав. Рівень може бути одним із чотирьох значень у порядку зростання привілеїв:

1. Anonymous - Анонімний: забороняє службі відкривати об'єкт Token та запитувати особистість користувача. Це найнижчий рівень. Якщо абонент вказав цей рівень, функціонуватиме лише обмежений набір послуг.

2. Identification - Ідентифікації: дозволяє службі відкривати об'єкт Token та вимагати ідентифікацію, групи та привілеї користувача. Однак потік не може відкривати будь-які захищені ресурси, уособлюючи користувача.

3. Impersonation - Імперсонації: дозволяє службі повністю використовувати ідентифікацію користувача у локальній системі. Служба може відкривати локальні ресурси, захищені користувачем, та маніпулювати ними. Вона також може отримати доступ до віддалених ресурсів для користувача, якщо користувач пройшов локальну автентифікацію в системі. Однак, якщо користувач пройшов аутентифікацію через мережеве з'єднання, наприклад, за протоколом Server Message Block (SMB), то служба не може використовувати об'єкт Token для доступу до віддалених ресурсів.

4. Delegation - Делегування: дозволяє службі відкривати всі локальні та віддалені ресурси, начебто вони були користувачем. Це найвищий рівень. Однак

для доступу до віддаленого ресурсу з користувачів, що пройшли мережну автентифікацію, недостатньо мати цей рівень уособлення. Домен Windows також має бути налаштований на дозвіл цих дій.

Ви можете вказати рівень імперсонації в SQoS під час виклику служби або під час створення копії існуючого токена. Щоб обмежити те, що може робити служба, вкажіть рівень ідентифікації або анонімності.

Запустимо тест за допомогою команди PowerShell *Invoke-NtToken*. У лістингу 4.3 уособлюємо токен на двох різних рівнях та намагаємось виконати скрипт, який відкриває захищений ресурс. Далі вказуємо рівень імперсонації за допомогою властивості *ImpersonationLevel*.

Лістинг 4.3. Видання себе за токен на різних рівнях та відкриття захищеного ресурсу

```
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--
```

Перша команда, яку виконуємо, отримує дескриптор первинного токена поточного процесу. Потім викликаємо *Invoke-NtToken*, щоб створити токен на рівні імперсонації, і запускаємо скрипт, який викликає *Get-NtDirectory*, щоб відкрити кореневий каталог OMNS. Операція відкриття проходить успішно, і виводимо об'єкт каталогу на консоль.

Потім необхідно повторити операцію на рівні ідентифікації, але в цей раз отримуємо помилку STATUS_BAD_IMPERSONATION_LEVEL. Зверніть увагу, що операція відкриття не повертає помилку «доступ заборонено», бо SRM не заходить достатньо далеко, щоб перевірити, чи може уособлений користувач отримати доступ до ресурсу.

Визначення рівня анонімного інперсонації не є тим самим, що запуск як користувача ANONYMOUS LOGON, зазначений у таб.4.1. Можливо запустити з ідентифікацією анонімного користувача і отримати доступ до ресурсу за допомогою перевірки доступу, але токен рівня Anonymous не може пройти жодної перевірки доступу, незалежно від того, як налаштована безпека ресурсу.

Ядро реалізує системний виклик *NtImpersonateAnonymousToken*, який буде імітувати анонімного користувача у вказаному потоці. Ви також можете отримати доступ до токена анонімного користувача за допомогою *Get-NtToken*:

```
PS C:\WINDOWS\system32> Get-NtToken -Anonymous | Format-NtToken
NT AUTHORITY\ANONYMOUS LOGON
```

Два інші поля в SQoS використовуються не часто, але вони важливі. Режим відстеження контексту визначає, чи можна статично захоплювати

ідентифікаційні дані користувача при підключенні до служби. Якщо ідентифікація не захоплена статично, а потім абонент, що викликає, видає себе за іншого користувача перед викликом служби, нова видана ідентифікація стане доступною службі, а не процесу. Зверніть увагу, що видана ідентифікація може бути передана службі лише тоді, коли вона знаходиться на рівні інперсонації або делегування. Якщо виданий токен знаходиться на рівні ідентифікації або анонімності, SRM створює помилку безпеки і відхиляє операцію інперсонації.

Режим ефективного токена змінює токен, переданий на сервер, в інший спосіб. Можна вимкнути групи та привілеї перед виконанням виклику, і якщо режим ефективного токена вимкнено, сервер може повторно включити ці групи та привілеї та використовувати їх. Однак, якщо режими ефективного токена активовано, SRM видалить групи та привілеї, щоб сервер не міг їх знову встановити або використовувати.

За замовчуванням, якщо при відкритті каналу міжпроцесної комунікації (IPC) не вказано структуру SQoS, рівень того, хто викликає є Impersonation зі статичним відстеженням і неефективним токеном.

Якщо контекст імперсонації захоплений, а виклик вже імперсонується, то рівень імперсонації токена потоку повинен бути більшим або дорівнювати рівню імперсонації; інакше захоплення не відбудеться. Це застосовується навіть у випадку, якщо SQoS вимагає рівня ідентифікації. Це важлива функція безпеки; вона запобігає виклику через канал RPC викликом на рівні ідентифікації або нижче, який видає себе за іншого користувача.

Ми описали, як SQoS визначається на рівні системних викликів, оскільки структура SECURITY _QUALITY_OF_SERVICE не відображається безпосередньо через API Win32.

Замість цього, зазвичай його вказують за допомогою додаткових прапорців; наприклад, CreateFile використовує SQoS, вказавши прапорець SECURITY_SQOS_PRESENT..

4.2.2. Явна імперсонація токена

Існує два способи явної імперсонації токена. Якщо у вас є дескриптор об'єкта імперсонації **Token** з доступом **Impersonate**, можна призначити його потоку за допомогою системного виклику *NtSetInformationThread* та інформаційного класу *ThreadImpersonationToken*.

Якщо натомість у вас є потік, який ви хочете уособити з доступом **Direct Impersonation**, ви можете використовувати інший механізм. З дескриптором вихідного потоку можна викликати системний виклик *NtImpersonateThread* і призначити токен імперсонації іншому потоку. Використання *NtImpersonateThread* є сумішшю явного і неявного уособлення. Ядро захопить контекст уособлення, якби вихідний потік викликав іменованим каналом.

Можна навіть вказати структуру SQoS для системного виклику.

Ви можете помилково вважати, що імперсонація відкриває величезну вразливість у системі безпеки. Якщо я налаштую власний іменований канал і

переконаю привілейований процес підключитися до мене, а виклик не встановлює SQoS для обмеження доступу, чи не зможу я отримати підвищені привілеї? Ми повернемося до того, як це можна запобігти в наступних розділах.

4.3. ПЕРЕТВОРЕННЯ МІЖ ТИПАМИ ТОКЕНІВ

Можна робити перетворення між двома типами токенів за допомогою дублювання. Під час дублювання токена ядро створює новий об'єкт Token та робить повну копію всіх властивостей об'єкта. Поки токен дублюється, можна змінити його тип.

Ця операція дублювання відрізняється від дублювання дескриптора, яке розглядалося раніше. Дублювання дескриптора токена просто створить новий дескриптор, який вказує на той самий об'єкт Token. Щоб дублювати фактичний об'єкт Token, потрібно мати права доступу **Duplicate** для дескриптора.

Якщо користувач має таке право, то можливо використовувати системний виклик *NtDuplicateToken* або команду PowerShell *Copy-NtToken* для дублювання токена. Наприклад, щоб створити токен імперсонації на рівні делегування на основі існуючого токена, скористайтеся скриптом у лістингу 4.4.

Лістинг 4.4. Дублювання токена для створення токена імперсонації

```
PS C:\WINDOWS\system32> $imp_token = Copy-NtToken -Token $token -ImpersonationLevel Delegation
PS C:\WINDOWS\system32> $imp_token.ImpersonationLevel
Delegation
PS C:\WINDOWS\system32> $imp_token.TokenType
Impersonation
```

Можна перетворити токен імперсонації назад на основний токен, знову використовуючи *Copy-NtToken*, як показано в лістингу 4.5.

Лістинг 4.5. Перетворення токена імперсонації в основний токен

```
PS C:\WINDOWS\system32> $pri_token = Copy-NtToken -Token $imp_token -Primary
PS C:\WINDOWS\system32> $pri_token.TokenType
Primary
PS C:\WINDOWS\system32> $pri_token.ImpersonationLevel
Delegation
```

Зверніть увагу на цікаву особливість у вихідних даних: новий первинний токен має той самий рівень імперсонації, що й оригінальний токен. Це пов'язано з тим, що SRM враховує лише властивість *TokenType*; якщо токен є первинним, рівень імперсонації ігнорується.

Оскільки ми можемо перетворити токен імперсонації назад у первинний токен, ви можете запитати: чи можна перетворити токен рівня ідентифікації або анонімного рівня назад у первинний токен, створити новий процес і обійти налаштування SQoS? Спробуємо це в лістингу 4.6.

Лістинг 4.6. Дублювання токена рівня ідентифікації назад до первинного токена

```
PS C:\WINDOWS\system32> $imp_token = Copy-NtToken -Token $token -ImpersonationLevel Identification
PS C:\WINDOWS\system32> $pri_token = Copy-NtToken -Token $imp_token -Primary
Исключение при вызове "DuplicateToken" с "5" аргументами: "(0xC00000A5) - Указан неверный уровень олицетворения.
Либо не предоставлен требуемый уровень олицетворения."
C:\Users\hladk\Documents\WindowsPowerShell\Modules\NtObjectManager\2.0.1\NtObjectManager.psm1:12282 знак:9
+         $Token.DuplicateToken($tokentype, $ImpersonationLevel, $Acces ...
+         ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : NotSpecified: (:) [], MethodInvocationException
    + FullyQualifiedErrorId : NtException
```

Цей список показує, що *не* можемо дублювати токен рівня ідентифікації назад до первинного токена. Другий рядок створює виняток, оскільки операція порушує гарантію безпеки SRM (зокрема, те, що SQoS дозволяє абоненту контролювати використання його ідентифікація).

Останнє зауваження: якщо відкриваєте токен за допомогою *Get-NtToken*, можна виконати операцію дублювання за один крок, вказавши параметр *Duplicate*.

4.4. ПСЕВДО ДЕСКРИПТОРИ ТОКЕНІВ

Щоб отримати доступ до токена, потрібно відкрити дескриптор об'єкта **Token**, а потім не забути закрити дескриптор після використання. У Windows 10 було представлено три псевдо дескриптори, які дозволяють запитувати інформацію про токен, не відкриваючи повний дескриптор для об'єкта ядра. Ось ці три дескриптори з їхніми значеннями у дужках:

Первинний (**Primary**) (-4) Первинний токен для поточного процесу;

Уособлення (**Impersonation**)(-5) Токен уособлення для поточного потоку; не вдається, якщо потік не уособлює себе;

Ефективний(**Effective**) (-6) Токен уособлення для поточного потоку, якщо він уособлює себе. Інакше, первинний токен.

На відміну від псевдо дескрипторів поточного процесу та поточного потоку, не можна дублювати ці дескриптори токенів. Є сенс використовувати їх лише для певних обмежених цілей, таких як запит інформації або виконання перевірок доступу. Команда *Get-NtToken* може повернути ці дескриптори, якщо вказати параметр *Pseudo*, як показано в лістингу 4.7.

Лістинг 4.7. Запит псевдо токенів

```
PS C:\WINDOWS\system32> Invoke-NtToken -Anonymous {Get-NtToken -Pseudo -Primary | Get-NtTokenSid}

Name            Sid
----            ---
REDMIBOOK\hladk S-1-5-21-2687063813-2248694510-27868876-1001


PS C:\WINDOWS\system32> Invoke-NtToken -Anonymous {Get-NtToken -Pseudo -Impersonation | Get-NtTokenSid}

Name                         Sid
----                         ---
NT AUTHORITY\ANONYMOUS LOGON S-1-5-7


PS C:\WINDOWS\system32> Invoke-NtToken -Anonymous {Get-NtToken -Pseudo -Effective | Get-NtTokenSid}

Name                         Sid
----                         ---
NT AUTHORITY\ANONYMOUS LOGON S-1-5-7


PS C:\WINDOWS\system32> Invoke-NtToken -Anonymous {Get-NtToken -Pseudo -Effective} | Get-NtTokenSid

Name            Sid
----            ---
REDMIBOOK\hladk S-1-5-21-2687063813-2248694510-27868876-1001
```

У цьому прикладі ми запитуємо три типи псевдо-токенів, імітуючи анонімного користувача. Перша команда запитує первинний токен і витягує його SID користувача. Наступна команда запитує токен імітації, який повертає SID анонімного користувача. Потім ми запитуємо ефективний токен, який, оскільки

ми імітуємо анонімного користувача, також повертає SID анонімного користувача. Нарешті, ми знову запитуємо ефективний токен, цього разу чекаючи, поки блок сценарію буде виконаний, щоб витягти SID користувача. Ця операція повертає SID користувача первинного токена, демонструючи, що псевдо токен є контекстна-залежним.

4.5. ГРУПИ ТОКЕНІВ

Якщо адміністратори мали б захищати кожен ресурс для кожного можливого користувача, управління безпекою облікових записів стало б надто складним. Групи дозволяють користувачам спільно використовувати більш широкі облікові записи безпеки. Більшість операцій контролю доступу в Windows надають доступ групам, а не окремим користувачам.

З точки зору SRM, група є просто ще одним SID, який потенційно може визначати доступ до ресурсу. Ми можемо відобразити групи в консолі PowerShell за допомогою команди *Get-NtTokenGroup*, як показано в лістингу 4,8.

Лістинг 4.8. Запит груп поточного токена

```
PS C:\WINDOWS\system32> Get-NtTokenGroup $token

Name                                                    Attributes
----                                                    ----------
Mandatory Label\High Mandatory Level                    Integrity, IntegrityEnabled
Everyone                                                Mandatory, EnabledByDefault, Enabled
NT AUTHORITY\Локальний обліковий запис і член групи адміністраторів Mandatory, EnabledByDefault, Enabled
BUILTIN\Administrators                                  Mandatory, EnabledByDefault, Enabled, Owner
BUILTIN\Performance Log Users                           Mandatory, EnabledByDefault, Enabled
BUILTIN\Users                                           Mandatory, EnabledByDefault, Enabled
NT AUTHORITY\INTERACTIVE                                Mandatory, EnabledByDefault, Enabled
CONSOLE LOGON                                           Mandatory, EnabledByDefault, Enabled
NT AUTHORITY\Authenticated Users                        Mandatory, EnabledByDefault, Enabled
NT AUTHORITY\This Organization                          Mandatory, EnabledByDefault, Enabled
MicrosoftAccount\hladkykh.v@gmail.com                   Mandatory, EnabledByDefault, Enabled
NT AUTHORITY\Локальний обліковий запис                  Mandatory, EnabledByDefault, Enabled
NT AUTHORITY\LogonSessionId_0_339658873                 Mandatory, EnabledByDefault, Enabled, LogonId
LOCAL                                                   Mandatory, EnabledByDefault, Enabled
NT AUTHORITY\Автентифікація облікового запису хмари     Mandatory, EnabledByDefault, Enabled
```

Можна використовувати *Get-NtTokenGroup* для фільтрації певних прапорів атрибутів, вказавши параметр *Attributes*. У таб.4.2 наведено можливі прапорці атрибутів, які можемо передати команді.

Таблиця 4.2.

Атрибути групи у форматі SDK і PowerShell

Ім'я атрибута SDK	Ім'я атрибута PowerShell
SE_GROUP_ENABLED	Enabled
S E_GROU P_E NAB L E D_BY_DE FAU LT	EnabledByDefault
SE_GROUP_MANDATORY	Mandatory
SE_GROUP_LOGON_ID	Logonld
SE_GROUP_OWNER	Owner
SE_GROUP_USE_FOR_DENY_ONLY	UseForDenyOnly
SE_GROUP_INTEGRITY	Integrity
SE_GROUP_INTEGRITY_ENABLED	IntegrityEnabled
SE_GROUP_RESOURCE	Resource

Означення кожного із властивостей зазначених прапорів розглянемо далі.

4.5.1. Атрибути прапорів Enabled, EnabledByDefault та Mandatory

Найважливішим атрибутом прапорця є **Enabled**. Коли прапорець встановлено, SRM враховує групу під час процесу перевірки доступу. Інакше він ігноруватиме групу.

Будь-яка група з встановленим атрибутом **EnabledByDefault** вмикається автоматично.

Можливо вимкнути групу (виключивши її з процесу перевірки доступу) за допомогою системного виклику *NtAdjustGroupsToken*, якщо є доступ *AdjustGroups* до дескриптора токена. Команда PowerShell *Set-NtTokenGroup* надає доступ до цього системного виклику. Однак, якщо не можете вимкнути групи, для яких встановлено прапорець **Mandatory** (Обов'язковий). Цей прапорець встановлено для всіх груп у токені звичайного користувача, але деякі системні токени мають необов'язкові групи. Якщо групу вимкнено під час передачі токена уособлення через RPC, а в SQoS встановлено прапорець ефективного режиму токена, токен уособлення видалить групу.

4.5.2. LogonId

Прапор LogonId ідентифікує будь-який SID, який надається всім токенам на одному робочому столі. Наприклад, якщо ви запускаєте процес як інший користувач за допомогою утиліти *runas*, токен нового процесу матиме той самий SID входу, що і виклик, навіть якщо це інша ідентичність. Така поведінка дозволяє SRM надавати доступ до ресурсів, специфічних для сеансу, таких як каталог об'єктів сеансу.

SID завжди має формат S-1-4-4-X-Y, де X і Y — це два 32-бітні значення LUID, які були призначені під час створення сеансу автентифікації. Ми повернемося до SID входу в систему та його застосування в наступному розділі.

4.5.3. Owner

Всі захищені ресурси в системі належать або груповому SID, або користувацькому SID. Токени мають властивість **Owner** (власник), яка містить SID, що використовується як власник за замовчуванням при створенні ресурсу. SRM дозволяє вказувати у властивості **Owner** тільки певний набір SID користувачів: або SID користувача, або будь-який груповий SID, який позначений прапорцем Owner.

Ви можете отримати або встановити поточне властивість Owner токена за допомогою команди *Get -NtTokenSid* або *Set-NtTokenSid*. Наприклад, в лістинг 4.9 ми отримуємо SID власника з поточного токена, а потім намагаємося встановити власника.

Лістинг 4.9. Отримання та встановлення SID власника токена

```
PS C:\WINDOWS\system32> Get-NtTokenSid $token -Owner

Name                  Sid
----                  ---
BUILTIN\Administrators S-1-5-32-544


PS C:\WINDOWS\system32> 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 ob
ject."
At C:\Users\hladk\Documents\WindowsPowerShell\Modules\NtObjectManager\2.0.1\NtObjectManager.psm1:13024 char:23
+             "Owner" { $Token.Owner = $Sid }
+                       ~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : NotSpecified: (:) [], SetValueInvocationException
    + FullyQualifiedErrorId : ExceptionWhenSetting
```

У цьому випадку спроба встановити для властивості *Owner* значення SID S-1-2-3-4 завершується невдачею з винятком, оскільки це не наш поточний SID користувача або не входить до нашого списку груп.

4.5.4. UseForDenyOnly

Перевірка доступу SRM дозволяє або забороняє доступ до SID. Але коли SID вимкнено, він більше не бере участі в перевірках дозволу або заборони, що може призвести до неправильної перевірки доступу.

Розглянемо простий приклад. Уявіть, що є дві групи: «Співробітники» та «Віддалений доступ». Користувач створює документ, який він хоче, щоб могли читати всі співробітники, крім тих, хто отримує віддалений доступ до системи, оскільки вміст документа є конфіденційним і користувач не хоче, щоб він потрапив у відкритий доступ. Документ налаштований так, щоб надавати доступ усім членам групи «Співробітники», але відмовляти в доступі користувачам групи «Віддалений доступ».

Тепер уявіть, що користувач, який належить до обох цих груп, може вимкнути групу під час доступу до ресурсу. Він може просто вимкнути «Віддалений доступ», щоб отримати доступ до документа на основі свого членства в групі «Співробітники», легко обійшовши обмеження доступу.

З цієї причини користувачеві майже ніколи не дозволяється вимикати групи. Однак у деяких випадках, наприклад, у пісочниці, ви можете захотіти вимкнути групу, щоб її не можна було використовувати для доступу до ресурсу. Прапорець *UseForDenyOnly* вирішує цю проблему. Коли SID позначений цим прапорцем, він не буде враховуватися при перевірці дозволу на доступ, але все одно буде враховуватися при перевірці відмови в доступі. Користувач може позначити свої власні групи як *UseForDenyOnly*, відфільтрувавши свій токен і використовуючи його для створення нового процесу. Ми обговоримо фільтрування токенів, коли розглянемо обмежені токени в розділі «Токени пісочниці».

4.5.5. Integrity and IntegrityEnabled

Прапорці атрибутів **Integrity** та **IntegrityEnabled** вказують на те, що SID представляє рівень цілісності токена та є ввімкненим. Групові SID, позначені

прапорцем атрибута **Integrity**, зберігають цей рівень цілісності як 32-бітове число у своєму кінцевому RID.

RID може мати будь-яке довільне значення; однак у SDK є сім попередньо визначених рівнів, як показано в таб.4.3. Тільки перші шість є загальновживаними та доступними з користувацького процесу. Для позначення SID цілісності SRM використовує центр безпеки **MandatoryLabel** (який має значення 16).

Таблиця 4.3.

Визначені значення рівня цілісності

Рівень цілісності	Назва рівня цілісності SDK	Назва PowerShell
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

Рівень за замовчуванням для користувача – *Середній*. Адміністраторам зазвичай призначається *Високий*, а службам – *Система*. Можна зробити запит SID цілісності токена за допомогою *Get-NtTokenSid*, як показано в лістингу 4.10.

Лістинг 4.10. Отримання SID рівня цілісності токена

```
PS C:\WINDOWS\system32> Get-NtTokenSid $token -Integrity

Name                               Sid
----                               ---
Mandatory Label\High Mandatory Level S-1-16-12288
```

Можливо також встановити новий рівень цілісності токена, за умови, що він менший або дорівнює поточному значенню. Рівень також можна збільшити, але для цього потрібні спеціальні привілеї та увімкнення *SeTcbPrivilege*.

Хоча можна встановити весь SID, зазвичай зручніше встановити лише значення. Наприклад, скрипт у лістингу 4.11 встановить для токена низький рівень цілісності.

Лістинг 4.11. Встановлення низького рівня цілісності токена

```
PS C:\WINDOWS\system32> Set-NtTokenIntegrityLevel Low -Token $token
PS C:\WINDOWS\system32> Get-NtTokenSid $token -Integrity

Name                              Sid
----                              ---
Mandatory Label\Low Mandatory Level S-1-16-4096
```

Якщо ви запустите цей скрипт, ви можете помітити, що в консолі PowerShell починають з'являтися помилки через блокування доступу до файлів. Ми обговоримо, чому доступ до файлів блокується, коли будемо розглядати обов'язковий контроль цілісності.

4.5.6. Resource

Останній атрибут-прапор заслуговує лише на коротке згадування. Атрибут-прапор «Ресурс» вказує, що SID групи є локальним SID домену.

4.5.7. Групи пристроїв

Токен також може мати окремий список груп пристроїв. Ці групові SID додаються, коли користувач проходить автентифікацію на сервері через мережу в корпоративному середовищі, як показано в лістингу 4.12.

Ви можете отримати інформацію про групи за токеном, використовуючи *Get-NtTokenGroup* і передаючи параметр *Device*.

Лістинг 4.12. Відображення груп пристроїв за допомогою *Get-NtTokenGroup*

```
PS> Get-NtTokenGroup -Device -Token $token
Name                          Attributes
 ----                         ----------
BUILTIN\Users                 Mandatory, EnabledByDefault, Enabled
AD\CLIENT1$                   Mandatory, EnabledByDefault, Enabled
AD\Domain Computers           Mandatory, EnabledByDefault, Enabled
NT AUTHORITY\Claims Value     Mandatory, EnabledByDefault, Enabled
--snip--
```

4.6. ПРИВІЛЕЇ

Групи дозволяють системним адміністраторам контролювати доступ користувачів до певних ресурсів. На відміну від груп, привілеї надаються користувачам, щоб вони могли обходити певні перевірки безпеки для всіх типів ресурсів, наприклад, обходити перевірку доступу. Привілей також може застосовуватися до певних дій, що вимагають особливих прав, таких як зміна системного годинника. Ви можете переглянути привілеї токена в консолі за допомогою *Get-NtTokenPrivilege* (лістинг 4.13).

Лістинг 4.13. Токени привілеїв

```
PS C:\WINDOWS\system32> Get-NtTokenPrivilege $token

Name                                      Luid             Enabled
----                                      ----             -------
SeIncreaseQuotaPrivilege                  00000000-00000005 False
SeSecurityPrivilege                       00000000-00000008 False
SeTakeOwnershipPrivilege                  00000000-00000009 False
SeLoadDriverPrivilege                     00000000-0000000A False
SeSystemProfilePrivilege                  00000000-0000000B False
SeSystemTimePrivilege                     00000000-0000000C False
SeProfileSingleProcessPrivilege           00000000-0000000D False
SeIncreaseBasePriorityPrivilege           00000000-0000000E False
SeCreatePageFilePrivilege                 00000000-0000000F False
SeBackupPrivilege                         00000000-00000011 False
SeRestorePrivilege                        00000000-00000012 False
SeShutdownPrivilege                       00000000-00000013 False
SeDebugPrivilege                          00000000-00000014 True
SeSystemEnvironmentPrivilege              00000000-00000016 False
SeChangeNotifyPrivilege                   00000000-00000017 True
SeRemoteShutdownPrivilege                 00000000-00000018 False
SeUndockPrivilege                         00000000-00000019 False
SeManageVolumePrivilege                   00000000-0000001C False
SeImpersonatePrivilege                    00000000-0000001D True
SeCreateGlobalPrivilege                   00000000-0000001E True
SeIncreaseWorkingSetPrivilege             00000000-00000021 False
SeTimeZonePrivilege                       00000000-00000022 False
SeCreateSymbolicLinkPrivilege             00000000-00000023 False
SeDelegateSessionUserImpersonatePrivilege 00000000-00000024 False
```

Вихідні дані розділені на три стовпці. Перший стовпець містить загальну назву привілею. Як і у випадку з SID, SRM не використовує цю назву безпосередньо, а замість цього використовує значення LUID привілею, яке ми бачимо у другому стовпці. Останній стовпець вказує, чи привілей наразі активний.

Привілеї можуть бути в активному або неактивному стані.

Будь-яка перевірка привілею повинна переконатися, що привілей активний, а не просто присутній. У певних обставинах, таких як пісочниця, токен може мати привілей, що зазначений у лістингу, але обмеження пісочниці можуть заважати його позначенню як увімкненого. Прапор *Enabled* насправді є набором прапорів атрибутів, як атрибути для групових SID. Ми можемо переглянути ці атрибути, якщо відформатуємо вихідні дані *Get-NtTokenPrivilege* як список (лістинг 4.14).

Лістинг 4.14. Відображення всіх властивостей привілею *SeChangeNotifyPrivilege*

```
PS C:\WINDOWS\system32> Get-NtTokenPrivilege $token -Privileges SeChangeNotifyPrivilege | Format-List


Name        : SeChangeNotifyPrivilege
Luid        : 00000000-00000017
Attributes  : EnabledByDefault, Enabled
Enabled     : True
DisplayName : Bypass traverse checking
```

У вихідних даних зараз ми бачимо атрибути, які включають як *Enabled*, так і *EnabledByDefault*. Атрибут *EnabledByDefault* визначає, чи має бути привілей активним за замовчуванням. Також ми бачимо додаткову властивість *DisplayName*, яка використовується для надання додаткової інформації користувачеві.

Щоб змінити стан привілеїв токена, вам потрібен доступ *AdjustPrivileges* до дескриптора токена, далі ви можете використовувати системний виклик *NtAdjustPrivilegesToken* для налаштування атрибутів та увімкнення або вимкнення привілею. Команди PowerShell *Enable-NtTokenPrivilege* та *Disable-NtTokenPrivilege* використовують цей системний виклик, як показано в лістингу 4.15.

За допомогою API *NtAdjustPrivilegesToken* також можна повністю видалити привілей, вказавши атрибут Remove, що можна зробити за допомогою команди PowerShell *Remove-NtTokenPrivilege*.

Лістинг 4.15. Включення та відключення привілею *SeTimeZonePrivilege*

```
PS C:\WINDOWS\system32> Enable-NtTokenPrivilege SeTimeZonePrivilege -Token $token -PassThru

Name                Luid             Enabled
----                ----             -------
SeTimeZonePrivilege 00000000-00000022 True


PS C:\WINDOWS\system32> Disable-NtTokenPrivilege SeTimeZonePrivilege -Token $token -PassThru

Name                Luid             Enabled
----                ----             -------
SeTimeZonePrivilege 00000000-00000022 False
```

Видалення привілею гарантує, що токен більше ніколи не зможе його використовувати. Якщо ви лише вимкнете привілей, його можна буде випадково увімкнути знову. У лістингу 4.16 показано, як видалити привілей.

Лістинг 4.16. Видалення привілею з токена

```
PS C:\WINDOWS\system32> Get-NtTokenPrivilege $token -Privileges SeTimeZonePrivilege

Name                Luid              Enabled
----                ----              -------
SeTimeZonePrivilege 00000000-00000022 False


PS C:\WINDOWS\system32> Remove-NtTokenPrivilege SeTimeZonePrivilege -Token $token
PS C:\WINDOWS\system32> Get-NtTokenPrivilege $token -Privileges SeTimeZonePrivilege
WARNING: Couldn't get privilege SeTimeZonePrivilege
```

Щоб перевірити привілеї, користувацька програма може звернутися до системного виклику *NtPrivilegeCheck*, а код ядра може звернутися до API *SePrivilegeCheck*. Ви можете запитати, чи можна просто вручну перевірити, чи активний привілей, замість того, щоб використовувати спеціальний системний виклик. У цьому випадку так, але завжди варто використовувати системні засоби, якщо це можливо. Це робиться на випадок, якщо ви зробите помилку в реалізації або не врахуєте якийсь крайній випадок. Команда PowerShell *Test-NtTokenPrivilege* обробляє системний виклик, як показано в лістингу 4.17.

Лістинг 4.17. Виконання перевірки привілеїв

```
PS C:\WINDOWS\system32> Remove-NtTokenPrivilege SeTimeZonePrivilege -Token $token
PS C:\WINDOWS\system32> Get-NtTokenPrivilege $token -Privileges SeTimeZonePrivilege
WARNING: Couldn't get privilege SeTimeZonePrivilege
PS C:\WINDOWS\system32> Enable-NtTokenPrivilege SeChangeNotifyPrivilege
PS C:\WINDOWS\system32> Disable-NtTokenPrivilege SeTimeZonePrivilege
WARNING: Couldn't set privilege SeTimeZonePrivilege
PS C:\WINDOWS\system32> Test-NtTokenPrivilege SeChangeNotifyPrivilege
True
PS C:\WINDOWS\system32> Test-NtTokenPrivilege SeTimeZonePrivilege, SeChangeNotifyPrivilege -All
False
PS C:\WINDOWS\system32> Test-NtTokenPrivilege SeTimeZonePrivilege, SeChangeNotifyPrivilege -All -PassResult

EnabledPrivileges         AllPrivilegesHeld
-----------------         -----------------
{SeChangeNotifyPrivilege} False
```

Цей лістинг демонструє кілька прикладів перевірки привілеїв за допомогою *Test-NtTokenPrivilege*. Почнемо з увімкнення *SeChangeNotifyPrivilege* та вимкнення *SeTimeZonePrivilege*. Це загальні привілеї, надані всім користувачам, але, можливо, вам доведеться змінити приклад, якщо ваш токен їх не має.

Потім ми перевіряємо лише *SeChangeNotifyPrivilege*. Він увімкнений, тому цей тест повертає **True**. Далі ми перевіряємо як *SeTimeZonePrivilege*, так і *SeChangeNotifyPrivilege*. Ми бачимо, що не маємо всіх привілеїв, тому *Test-NtTokenPrivilege* повертає **False**. Нарешті, ми виконуємо ту саму команду, але вказуємо опцію *-PassResult*, щоб повернути повний результат перевірки. У стовпці EnabledPrivileges ми бачимо, що ввімкнена лише *SeChangeNotifyPrivilege*.

Нижче наведено деякі з привілеїв, доступних у системі:

SeChangeNotifyPrivilege - Назва цього привілею вводить в оману. Він дозволяє користувачеві отримувати повідомлення про зміни у файловій системі або реєстрі, але також використовується для обходу перевірки доступу.

SeasSignPrimaryTokenPrivilege та SeimpersonatePrivilege - Ці привілеї дозволяють користувачеві обійти перевірки присвоєння первинного токена та імперсонації відповідно. На відміну від більшості привілеїв у цьому списку, ці привілеї повинні бути ввімкнені на первинному токені поточного процесу, а не на токені імперсонації.

SeBackupPrivilege та **SeRestorePrivilege** - Ці привілеї дозволяють користувачеві обходити перевірку доступу під час відкриття певних ресурсів, таких як файли або ключі реєстру. Це дозволяє користувачеві створювати резервні копії та відновлювати ресурси без необхідності отримання явного доступу до них. Ці привілеї також були адаптовані для інших цілей, наприклад, привілей відновлення дозволяє користувачеві завантажувати довільні гілки реєстру.

SeSecurityPrivilege и **SeAuditPrivilege** - Перший з цих привілеїв дозволяє надати користувачеві право доступу *AccessSystemSecurity* до ресурсу. Це дає користувачеві можливість змінювати конфігурацію аудиту ресурсу. Привілей *SeAuditPrivilege* дозволяє користувачеві генерувати довільні повідомлення аудиту об'єктів з користувацької програми.

SeCreateTokenPrivilege - Даний привілей повинен бути наданий тільки дуже обмеженій групі користувачів, оскільки він надає можливість створювати довільні токени за допомогою системного виклику *NtCreateToken*.

SeDebugPrivilege - Назва цього привілею передбачає, що він необхідний для налагодження процесів. Однак це не зовсім так, оскільки налагоджувати процес можна і без нього. Цей привілей дозволяє користувачеві обходити будь-які перевірки доступу під час відкриття процесу або об'єкта потоку.

SeTcbPrivilege - Назва цього привілею походить від терміну «довірча обчислювальна база» (trusted computing base, TCB), який використовується для позначення привілейованого ядра операційної системи Windows. Це загальний термін для привілейованих операцій, які не охоплюються більш конкретним привілеєм. Наприклад, він дозволяє користувачам обходити перевірку на підвищення рівня цілісності токена (до межі рівня системи), а також вказувати резервний обробник винятків для процесу.

SeLoadDriverPrivilege - Ми можемо завантажити новий драйвер ядра за допомогою системного виклику *NtLoadDriver*, хоча частіше використовується SCM. Цей привілей необхідний для успішного виконання цього системного виклику. Зверніть увагу, що наявність цього привілею не дозволяє обійти перевірки драйвера ядра, такі як підписання коду.

SeTakeOwnershipPrivilege та **SeRelabelPrivilege** - Ці привілеї мають однаковий безпосередній ефект: вони дозволяють надати користувачеві доступ *WriteOwner* до ресурсу, навіть якщо звичайний контроль доступу цього не дозволяє. *SeTakeOwnershipPrivilege* дозволяє користувачеві отримати право власності на ресурс, оскільки для цього необхідний доступ *WriteOwner*. *SeRelabelPrivilege* обходить перевірки обов'язкової мітки ресурсу. Зазвичай ви можете встановити мітку, яка дорівнює або нижча за рівень цілісності виклику.

Встановлення обов'язкової мітки також вимагає доступу *WriteOwner* до дескриптора.

Ми розглянемо конкретні приклади використання цих привілеїв у наступних розділах, коли будемо обговорювати дескриптори безпеки та перевірки доступу. А зараз давайте перейдемо до способів обмеження доступу за допомогою пісочниці.

4.7. ТОКЕНИ «ПІСОЧНИЦІ»

У нашому взаємопов'язаному світі ми змушені обробляти велику кількість ненадійних даних. Зловмисники можуть створювати дані, наприклад, щоб скористатися вразливістю безпеки веб-браузера або програми для читання документів. Щоб протидіяти цій загрозі, Windows надає метод обмеження ресурсів, до яких користувач може отримати доступ, розміщуючи будь-які процеси, що обробляють ненадійні дані, у «пісочниці» (**sandbox**). Якщо процес буде скомпрометований, зловмисник матиме лише обмежений доступ до системи і не зможе отримати доступ до конфіденційної інформації користувача.

Windows реалізує пісочниці за допомогою трьох спеціальних типів токенів: токени з обмеженим доступом (restricted tokens), токени з обмеженим доступом для запису(write-restricted tokens) та токени низького рівня (lowbox tokens).

4.7.1. Обмежені токени

Тип токена з обмеженим доступом є найстарішим токеном пісочниці в Windows. Він був введений як частина Windows 2000, але не використовувався широко як пісочниця до появи веб-браузера Google Chrome. Інші браузери, такі як Firefox, з того часу скопіювали реалізацію пісочниці Chrome, як і програми для читання документів, такі як Adobe Reader.

Ви можете створити обмежений токен за допомогою системного виклику *NtFilterToken* або Win32 API *CreateRestrictedToken*, кожен з яких дозволяє вам вказати список обмежених SID, щоб обмежити ресурси, до яких токен матиме доступ. SID не обов'язково повинні бути вже доступними в токені. Наприклад, найбільш обмежувальна пісочниця Chrome визначає NULL SID (S-1-0-0) як єдиний обмежений SID. NULL SID ніколи не надається токену як звичайній групі.

Будь-яка перевірка доступу повинна дозволяти як звичайний список груп, так і список обмежених SID. Інакше користувачеві буде відмовлено в доступі. Системний виклик *NtFilterToken* також може маркувати звичайні групи прапорцем атрибуту *UseForDenyOnly* і привілеями на видалення. Ми можемо поєднати можливість фільтрувати токен з обмеженими SID або використовувати її окремо, щоб створити токен з меншими привілеями без створення більш комплексної пісочниці.

Лістинг 4.18. Створення обмеженого токену та відображення груп та привілеїв

```
PS C:\WINDOWS\system32> $token = Get-NtToken -Filtered -RestrictedSids RC -SidsToDisable WD -Flags DisableMaxPrivileges
PS C:\WINDOWS\system32> Get-NtTokenGroup $token -Attributes UseForDenyOnly

Name     Attributes
----     ----------
Everyone UseForDenyOnly


PS C:\WINDOWS\system32> Get-NtTokenGroup $token -Restricted

Name                   Attributes
----                   ----------
NT AUTHORITY\RESTRICTED Mandatory, EnabledByDefault, Enabled


PS C:\WINDOWS\system32> Get-NtTokenPrivilege $token

Name                   Luid              Enabled
----                   ----              -------
SeChangeNotifyPrivilege 00000000-00000017 True


PS C:\WINDOWS\system32> $token.Restricted
True
```

Легко створити обмежений токен, який не може отримати доступ до будь-яких ресурсів. Таке обмеження створює хорошу пісочницю, але також унеможливлює використання токена як основного токена процесу, оскільки процес не зможе запуститися.

Лістинг 4.18 демонструє, як створити обмежений токен і отримати результати.

Почнемо зі створення обмеженого токена за допомогою команди *Get-NtToken*.

Вказуємо один обмежений SID, RC, який відображається на спеціальний NT AUTHORITY\ RESTRICTED SID, який зазвичай налаштовується для системних ресурсів, щоб дозволити доступ для читання. Також вказуємо, що хочемо перетворити групу *Everyone* (WD) на *UseForDenyOnly*. Нарешті, вказуємо прапорець, щоб вимкнути максимальну кількість привілеїв.

Далі ми відображаємо властивості токена, починаючи з усіх звичайних груп, використовуючи атрибут *UseForDenyOnly*. Вихідні дані показують, що тільки група Everyone має встановлений прапорець. Потім ми відображаємо список обмежених SID, який показує NT AUTHORITY\RESTRICTED SID.

Після цього ми відображаємо привілеї. Зверніть увагу, що навіть незважаючи на те, що ми вимагали вимкнути максимальні привілеї, *SeChangeNotifyPrivilege* все ще присутній. Цей привілей не видаляється, оскільки без нього може бути дуже складно отримати доступ до ресурсів. Якщо ви дійсно хочете позбутися його, ви можете явно вказати його в *NtFilterToken* або видалити після створення токена.

Нарешті, ми запитуємо властивість токена, яка вказує, чи є він обмеженим токеном.

4.7.2. Токени з обмеженим доступом для запису

Токен з обмеженням запису забороняє доступ на запис до ресурсу, але дозволяє доступ на читання та виконання. Можна створити токен з обмеженням запису, активізувавши прапорець WRITE_RESTRICTED в *NtFilterToken*.

Microsoft вперше представив цей тип токена у Windows XP SP2 для захисту системних служб. Його набагато простіше використовувати як «пісочницю», ніж обмежений токен, оскільки не потрібно турбуватися про те, що токен не зможе читати критичні ресурси, такі як бібліотеки DLL. Однак він створює менш корисну «пісочницю». Наприклад, якщо можете читати файли для користувача, можете запозичити його особисту інформацію, таку як паролі, збережені браузером, без необхідності виходу з «пісочниці».

Для повноти давайте створимо токен з обмеженням запису та переглянемо його властивості (лістинг 4.19).

Лістинг 4.19. Створення токена з обмеженням запису

```
PS C:\WINDOWS\system32> $token = Get-NtToken -Filtered -RestrictedSids WR -Flags WriteRestricted
PS C:\WINDOWS\system32> Get-NtTokenGroup $token -Restricted

Name                          Attributes
----                          ----------
NT AUTHORITY\WRITE RESTRICTED Mandatory, EnabledByDefault, Enabled


PS C:\WINDOWS\system32> $token.Restricted
True
PS C:\WINDOWS\system32> $token.WriteRestricted
True
```

Почнемо зі створення токена за допомогою команди *Get-NtToken*. Вказуємо один обмежений SID, **WR**, який відповідає спеціальному NT AUTHORITY\WRITE RESTRICTED SID, що еквівалентний NT AUTHORITY\RESTRICTED, але призначений для доступу на запис до певних системних ресурсів. Також вказуємо прапорець *WriteRestricted*, щоб зробити цей токен токеном з обмеженим доступом на запис, а не звичайним обмеженим токеном.

Далі ми відображаємо властивості токена. У списку обмежених SID бачимо NT AUTHORITY\WRITE RESTRICTED. Відображення властивості *Restricted* показує, що токен вважається обмеженим, однак бачимо, що він також позначений як *WriteRestricted*.

4.7.3. Токени AppContainer та Lowbox

У Windows 8 було впроваджено пісочницю **AppContainer** для захисту нової моделі додатків Windows. AppContainer реалізує свою безпеку за допомогою токена **lowbox**. Ви можете створити токен lowbox з існуючого токена за допомогою системного виклику *NtCreateLowBoxToken*. Не існує прямого еквівалента Win32 API для цього системного виклику, але ви можете створити процес AppContainer за допомогою API CreateProcess. Не будемо тут детально зупинятися на тому, як створити процес за допомогою цього API, замість цього зосередимося лише на токені lowbox.

При створенні токена lowbox, вам потрібно вказати пакет SID і список можливостей SID. Обидва типи SID видаються за допомогою пакета повноважень програми (який має значення 15). Ви можете розрізнити пакет SID і можливості SID, перевіривши їх перші RID, які повинні бути 2 і 3, відповідно. SID пакета працює як SID користувача в звичайному токені, тоді як SID можливостей діють як обмежені SID.

SID можливостей модифікують процес перевірки доступу, але вони також можуть мати значення окремо. Наприклад, існують можливості, що дозволяють доступ до мережі, які спеціально обробляються брандмауером Windows, навіть якщо це не пов'язано безпосередньо з перевіркою доступу.

Таблиця 4.4.

Ідентифікатори SID - коди традиційних функцій

Назва можливості	SID
Ваше інтернет-з’єднання	S-l-15-3-1
Ваше інтернет-з’єднання, включаючи вхідні з’єднання з інтернету	S-l-15-3-2
Ваші домашні або робочі мережі	S-l-15-3-3
Ваша бібліотека зображень	S-l-15-3-4
Ваша відеотека	S-l-15-3-5
Ваша музична бібліотека	S-l-15-3-6
Ваша бібліотека документів	S-l-15-3-7
Ваші облікові дані Windows	S-l-15-3-8
Сертифікати програмного та апаратного забезпечення або смарт-картка	S-l-15-3-9
Знімний носій	S-l-15-3-10
Ваші зустрічі	S-l-15-3-11
Ваші контакти	S-l-15-3-12
Internet Explorer	S-l-15-3-4096

Існує два типи ідентифікаторів SID:

Legacy - Невеликий набір попередньо визначених SID, введених у Windows8;

Named - RID походять від текстового імені.

Таб. 4-4 показує можливості Legacy.

Можна використовувати *Get-NtSid* для запиту ідентифікаторів безпеки пакетів та можливостей, як показано у лістингу 4.20.

Лістинг 4.20. Створення SID пакетів та можливостей

```
PS C:\WINDOWS\system32> Get-NtSid -PackageName 'my_package' -ToSddl
S-1-15-2-4047469452-4024960472-3786564613-914846661-3775852572-3870680127-2256146868
PS C:\WINDOWS\system32> Get-NtSid -PackageName 'my_package' -RestrictedPackageName "CHILD" -ToSddl
S-1-15-2-4047469452-4024960472-3786564613-914846661-3775852572-3870680127-2256146868-951732652-158068026-753518596-3921317197
PS C:\WINDOWS\system32> Get-NtSid -KnownSid CapabilityInternetClient -ToSddl
S-1-15-3-1
PS C:\WINDOWS\system32> Get-NtSid -CapabilityName registryRead -ToSddl
S-1-15-3-1024-1065365936-1281604716-3511738428-1654721687-432734479-3232135806-4053264122-3456934681
PS C:\WINDOWS\system32> Get-NtSid -CapabilityName registryRead -CapabilityGroup  -ToSddl
S-1-5-32-1065365936-1281604716-3511738428-1654721687-432734479-3232135806-4053264122-3456934681
```

Тут створюємо два SID пакета і два SID можливостей. Перший SID пакета генеруємо, вказавши його ім'я в *Get-NtSid* і отримавши в результаті SID. Цей SID

пакета походить від імені в нижньому регістрі, хешованого за допомогою алгоритму SHA256. 256-бітний дайджест розбивається на сім 32-бітних частин, які виступають в ролі RID. Кінцеве 32-бітне значення дайджесту відкидається.

Windows також підтримує обмежений пакет SID, який призначений для того, щоб пакет міг створювати нові безпечні дочірні пакети, які не можуть взаємодіяти між собою. Класичний веб-браузер використовував цю функцію для розділення дочірніх пакетів, що працюють в Інтернеті та Інтранеті, щоб у разі компрометації одного з них він не міг отримати доступ до даних іншого. Для створення дочірнього пакета ми використовуємо оригінальне ім'я групи пакетів та ідентифікатор дочірнього пакета. Створений SID розширює оригінальний SID пакета ще чотирма RID, як ви можете бачити у вихідних даних.

Перший SID є застарілою функцією для доступу до Інтернету. Зверніть увагу, що отриманий SDDL SID має одне додаткове значення RID (1). Другий SID походить від імені, в даному випадку *registryRead*, яке використовується для надання доступу на читання до групи ключів системного реєстру. Як і в разі пакетного SID, іменовані RID можливостей генеруються з хешу SHA256 імені, написаного малими літерами. Щоб розрізнити старі та іменовані SID можливостей, другий RID встановлюється на 1024, а потім додається хеш SHA256. Ви можете генерувати власні SID можливостей за допомогою цього методу, якщо якийсь ресурс налаштований на її використання.

Windows також підтримує групу можливостей, груповий SID, який можна додати до звичайного списку груп. Група можливостей встановлює перший RID на 32, а решту RID на той самий хеш SHA256, який був отриманий з імені можливості.

Лістинг 4.21. Створення токена lowbox та його властивості

```
PS C:\WINDOWS\system32> $token = Get-NtToken -LowBox -PackageSid 'my_package' -CapabilitySid "registryRead", "S-1-15-3-1"
PS C:\WINDOWS\system32> Get-NtTokenGroup $token -Capabilities | Select-Object Name

Name
----
NAMED CAPABILITIES\Registry Read
APPLICATION PACKAGE AUTHORITY\Підключення до Інтернету


PS C:\WINDOWS\system32> $package_sid = Get-NtTokenSid $token -Package -ToSddl
PS C:\WINDOWS\system32> $package_sid
S-1-15-2-4047469452-4024960472-3786564613-914846661-3775852572-3870680127-2256146868
PS C:\WINDOWS\system32> Get-NtTokenIntegrityLevel $token
Low
PS C:\WINDOWS\system32> $token.Close()
```

Спочатку викликаємо *Get-NtToken*, передаючи йому ім'я пакета (SID як SDDL) і список можливостей, які потрібно призначити токену lowbox. Потім можемо запитати список можливостей. Зверніть увагу, що імена двох SID можливостей різні: SID, отриманий від імені, має префікс NAMED CAPABILITIES. Немає можливості перетворити SID іменованої можливості назад на ім'я, з якого він був отриманий. Модуль PowerShell повинен генерувати ім'я на основі великого списку відомих можливостей.

Другий SID є застарілим SID, тому LSASS може перетворити його назад на ім'я.

Далі запитуємо SID пакета. Оскільки SID пакета походить від імені, що використовує SHA256, неможливо перетворити його назад на ім'я пакета. Знову

ж таки, модуль PowerShell має список імен, які він може використовувати для визначення оригінального імені.

Токен lowbox завжди встановлюється на низький рівень цілісності. Фактично, якщо привілейований користувач змінює рівень цілісності на середній або вище, всі властивості lowbox, такі як SID пакета та SID можливостей, видаляються, а токен повертається до токена, що не є пісочницею.

В цій частині обговорили зменшення привілеїв користувача шляхом перетворення його токена в токен пісочниці. Тепер перейдемо до іншого аспекту і розглянемо, що робить користувача достатньо привілейованим для адміністрування системи Windows.

4.8. ЩО РОБИТЬ КОРИСТУВАЧА АДМІНІСТРАТОРОМ?

Якщо ви маєте досвід роботи з **Unix**, то знаєте, що ідентифікатор користувача 0 — це обліковий запис адміністратора або **root**. Як root, ви можете отримати доступ до будь-яких ресурсів і налаштувати систему на свій розсуд. Під час інсталяції Windows перший обліковий запис, який ви налаштовуєте, буде обліковим записом адміністратора. Однак, на відміну від root, цей обліковий запис не матиме спеціального SID, який система обробляє по-іншому.

То що ж робить обліковий запис у Windows обліковим записом адміністратора?

Основна відповідь полягає в тому, що Windows налаштована так, щоб надавати певним групам і привілеям спеціальний доступ. Доступ адміністратора є за своєю суттю дискреційним, тобто можна бути адміністратором, але все одно не мати доступу до ресурсів. Реального еквівалента облікового запису **root** не існує (хоча користувач **SYSTEM** є близьким до нього).

Адміністратори, як правило, мають три характеристики.

По-перше, коли ви налаштовуєте користувача як адміністратора, ви зазвичай додаєте його до групи BUILTIN\Administrators, а потім налаштовуєте Windows, щоб дозволити доступ до групи під час перевірки доступу. Наприклад, системні папки, такі як C:\Windows, налаштовані так, щоб дозволити групі створювати нові файли та каталоги.

По-друге, адміністраторам надається доступ до додаткових привілеїв, які ефективно обходять засоби контролю безпеки системи. Наприклад, *SeDebugPrivilege* дозволяє користувачеві отримати повний доступ до будь-якого іншого процесу або потоку в системі, незалежно від того, які засоби безпеки йому призначені. Маючи повний доступ до процесу, можна змінити код, щоб отримати привілеї іншого користувача.

По-третє, адміністратори зазвичай працюють на високому рівні цілісності, тоді як системні служби працюють на системному рівні. Підвищуючи рівень цілісності адміністратора, ускладнюємо випадкове надання доступу до ресурсів адміністратора (особливо процесів і потоків) користувачам, які не є адміністраторами.

Слабкий контроль доступу до ресурсів є поширеною помилкою конфігурації. Однак, якщо ресурс також позначений рівнем цілісності вище середнього, то користувачі, які не є адміністраторами, не зможуть здійснювати запис в цей ресурс.

Швидкий спосіб перевірити, чи є токен адміністратором, це перевірити властивість *Elevated* на об'єкті **Token**. Ця властивість вказує, чи має токен певні доступні привілеї, що містяться у фіксованому списку в ядрі.

Лістинг 4.22 показує приклад для користувача, який не є адміністратором.

Лістинг 4.22. Властивість *Elevated* для користувача, що не є адміністратором

```
PS C:\WINDOWS\system32> $token = Get-NtToken
PS C:\WINDOWS\system32> $token.Elevated
True
```

Якщо токен має один із наступних привілеїв, він автоматично вважається підвищеним:

— SeCreateTokenPrivilege;
— SeTcbPrivilege;
— SeTakeOwnershipPrivilege;
— SeLoadDriverPrivilege;
— SeBackupPrivilege;
— SeRestorePrivilege;
— SeDebugPrivilege;
— SeImpersonatePrivilege;
— SeRelabelPrivilege;
— SeDelegateSessionUserImpersonatePrivilege.

Привілея не обов'язково повинна бути включена, вона просто повинна бути доступна у токені.

Для груп з підвищеними правами ядро не має фіксованого списку SID; натомість воно перевіряє лише останній RID SID. Якщо RID має одне з наступних значень, SID вважається підвищеним: **114, 498, 512, 516, 517, 518, 519, 520, 521, 544, 547, 548, 549, 550, 551, 553, 554, 556** або **569**. Наприклад, SID групи BUILTIN\Administrators — S-1-4-32-544. Оскільки 544 є в цьому списку, SID вважається підвищеним. (Зверніть увагу, що SID S-1-1-2-3-4-544 також вважатиметься підвищеним, хоча в ньому немає нічого особливого).

Поширеною ***помилкою*** є думка, що якщо токен має високий рівень цілісності, то це токен адміністратора. Однак властивість *Elevated* не перевіряє рівень цілісності токена, а лише його привілеї та групи. Група BUILTIN\Administrators все одно функціонуватиме з нижчим рівнем цілісності, надаючи доступ до таких ресурсів, як каталог файлової системи Windows. Єдине обмеження полягає в тому, що певні привілеї високого рівня, такі як *SeDebugPrivilege*, не можуть бути ввімкнені, якщо рівень цілісності нижчий за *високий*.

Також можливий випадок, коли користувач, який не є адміністратором, працює з високим рівнем цілісності, як у випадку процесів доступу до

інтерфейсу користувача, які іноді працюють на цьому рівні цілісності, але не мають спеціальних привілеїв або груп, що роблять їх адміністраторами.

4.9. КОНТРОЛЬ ОБЛІКОВИХ ЗАПИСІВ КОРИСТУВАЧІВ

При встановленні нової копії Windows перший користувач, якого створюєте, завжди є адміністратором. Важливо налаштувати користувача таким чином. Інакше було б неможливо змінювати систему та встановлювати нове програмне забезпечення.

Однак до Windows Vista така поведінка за замовчанням була величезною проблемою безпеки, оскільки звичайні користувачі встановлювали обліковий запис за замовчанням і, швидше за все, ніколи не змінювали його. Це означало, що більшість людей використовували повний обліковий запис адміністратора для повсякденних дій, таких як серфінг в Інтернеті. Якщо зловмисник міг скористатися вразливістю безпеки в браузері користувача, він отримував повний контроль за машиною Windows. До поширення «пісочниці» ця загроза була серйозною.

У Vista компанія Microsoft змінила цю поведінку за замовчуванням, запровадивши контроль облікових записів користувачів (UAC) та адміністратора з розділеним токеном. У цій моделі користувач залишається адміністратором, однак за замовчуванням усі програми виконуються з токеном, з якого видалено групи адміністраторів та привілеї. Коли користувачеві потрібно виконати завдання, система підвищує процес до повного адміністратора та показує запит, як на рис. 4.3, вимагаючи підтвердження користувача перед продовженням..

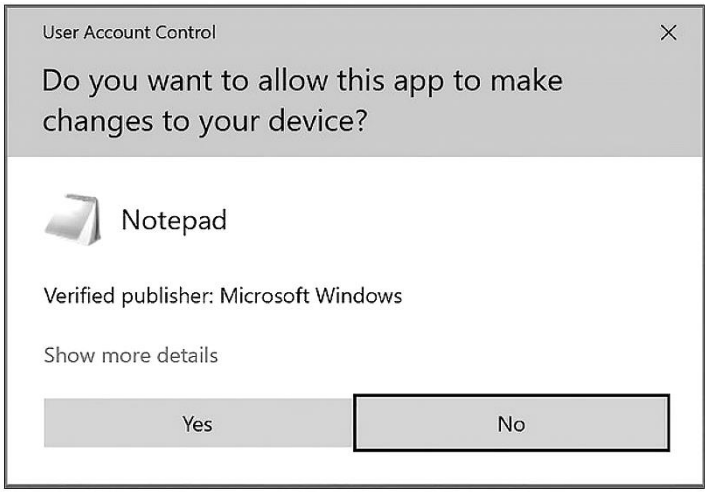


Рисунок 4.3. Діалогове вікно згоди UAC на підвищення привілеїв

Щоб полегшити користувачам роботу з Windows, можна налаштувати програму так, щоб вона примусово підвищувала свої права під час запуску. Властивість підвищення прав програми зберігається в XML-файлі маніфесту, вбудованому в виконуваний образ. Запустіть приклад у лістингу 4.23, щоб отримати інформацію маніфесту для всіх виконуваних файлів у каталозі System32.

Якщо це спеціальна програма, затверджена Microsoft, маніфест може вказувати, чи повинна програма автоматично і непомітно підвищувати свої права (що позначається значенням True у стовпці AutoElevate).

Лістинг 4.23. Запит інформації про маніфест, що виконується.

```
PS C:\WINDOWS\system32> ls C:\Windows\System32\*.exe | Get-Win32ModuleManifest

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
ARP.EXE                             False    False       asInvoker
at.exe                              False    False       asInvoker
attrib.exe                          False    False       asInvoker
audiodg.exe                         False    False       asInvoker
auditpol.exe                        False    False       asInvoker
AuthHost.exe                        False    False       asInvoker
autochk.exe                         False    False       asInvoker
autofstx.exe                        False    False       asInvoker
AxInstUI.exe                        False    False       asInvoker
backgroundTaskHost.exe              False    False       asInvoker
BackgroundTransferHost.exe          False    False       asInvoker
bcdboot.exe                         False    False       asInvoker
bcdedit.exe                         False    False       asInvoker
BdeUISrv.exe                        False    False       asInvoker
bdeunlock.exe                       False    False       asInvoker
BitLockerDeviceEncryption.exe       False    False       requireAdministrator
BitLockerWizardElev.exe             False    True        requireAdministrator
bitsadmin.exe                       False    False       asInvoker
```

Маніфест також вказує, чи може процес виконуватися з доступом до інтерфейсу користувача, про що ми поговоримо пізніше. Стовпець *ExecutionLevel* може мати три можливі значення:

asInvoker - Виконати процес від імені користувача, який його створив. Це значення є заданим за замовчуванням.

highestAvailable - Якщо користувач є адміністратором з розділеним токеном, примусово підвищити рівень до токена адміністратора. Якщо ні, запустити як користувача, який створив процес.

requireAdministrator - Примусово підвищити рівень, незалежно від того, чи є користувач адміністратором з розділеним токеном. Якщо користувач не є адміністратором, йому буде запропоновано ввести пароль облікового запису адміністратора.

Коли якийсь об'єкт створює виконуваний файл із підвищеним рівнем виконання, оболонка викликає метод RPC *RAiLaunchAdminProcess*. Цей метод перевіряє маніфест і запускає процес підвищення, включаючи відображення діалогового вікна згоди. Також можна вручну підвищити будь-яку програму за допомогою API *ShellExecute*, і запитуючи операцію *runas*. PowerShell відображає цю поведінку за допомогою команди Start-Process, як показано нижче:

```
PS C:\WINDOWS\system32> Start-Process notepad -Verb runas
```

Під час запуску цієї команди повинні побачити запит UAC. Якщо натиснути "Так" у діалоговому вікні згоди, *notepad.exe* повинен запуститися від імені адміністратора на робочому столі.

4.9.1. Зв'язані токени та рівень доступу

Коли адміністратор проходить автентифікацію на робочому столі, система відстежує два токени для користувача:

Обмежений (**Limited) -** Токен без підвищених прав, який використовується для більшості запущених процесів

Повний (**Full) -** Повний токен адміністратора, який використовується тільки після підвищення прав.

Лістинг 4.24. Відображення властивостей зв'язаного токена

```
PS C:\WINDOWS\system32> Use-NtObject($token = Get-NtToken -Linked) { Format-NtToken $token -Group -Privilege -Integrity -Information }
GROUP SID INFORMATION
---------------
Name                                                        Attributes
----                                                        ----------
Mandatory Label\Medium Mandatory Level                      Integrity, IntegrityEnabled
Everyone                                                    Mandatory, EnabledByDefault, Enabled
NT AUTHORITY\Локальний обліковий запис і член групи адміністраторів UseForDenyOnly
BUILTIN\Administrators                                      UseForDenyOnly
BUILTIN\Performance Log Users                               Mandatory, EnabledByDefault, Enabled
BUILTIN\Users                                               Mandatory, EnabledByDefault, Enabled
NT AUTHORITY\INTERACTIVE                                    Mandatory, EnabledByDefault, Enabled
CONSOLE LOGON                                               Mandatory, EnabledByDefault, Enabled
NT AUTHORITY\Authenticated Users                            Mandatory, EnabledByDefault, Enabled
NT AUTHORITY\This Organization                              Mandatory, EnabledByDefault, Enabled
MicrosoftAccount\hladkykh.v@gmail.com                       Mandatory, EnabledByDefault, Enabled
NT AUTHORITY\Локальний обліковий запис                      Mandatory, EnabledByDefault, Enabled
NT AUTHORITY\LogonSessionId_0_339658873                     Mandatory, EnabledByDefault, Enabled, LogonId
LOCAL                                                       Mandatory, EnabledByDefault, Enabled
NT AUTHORITY\Автентифікація облікового запису хмари         Mandatory, EnabledByDefault, Enabled

PRIVILEGE INFORMATION
---------------------
Name                          Luid              Enabled
----                          ----              -------
SeShutdownPrivilege           00000000-00000013 False
SeChangeNotifyPrivilege       00000000-00000017 True
SeUndockPrivilege             00000000-00000019 False
SeIncreaseWorkingSetPrivilege 00000000-00000021 False
SeTimeZonePrivilege           00000000-00000022 False

INTEGRITY LEVEL
---------------
Medium

TOKEN INFORMATION
-----------------
Type          : Impersonation
Imp Level     : Identification
ID            : 00000000-1785E96E
Auth ID       : 00000000-143ECB74
Origin ID     : 00000000-000003E7
Modified ID   : 00000000-143ECB80
Session ID    : 2
Elevated      : False
Elevation Type: Limited
Flags         : VirtualAllowed, IsFiltered, NotLow
```

Повна назва split-token administrator походить від цих двох токенів, оскільки наданий користувачеві доступ розділений між обмеженим і повним токенами.

Об'єкт **Token** має поле, яке використовується для зв'язування двох токенів між собою.

Зв'язаний токен можна отримати за допомогою системного виклику *NtQueryInformationToken* і класу інформації *TokenLinkedToken*.

У лістингу 4.24 перевіряємо деякі властивості цих зв'язаних токенів за допомогою PowerShell.

У цьому лістингу отримуємо доступ до пов'язаного токена, передаючи параметр *Linked* до *Get-NtToken*, і форматуємо токен для відображення його груп, привілеїв, рівня цілісності та інформації про токен. У списку груп бачимо, що група BUILTIN\Administrators увімкнена. Також бачимо, що список привілеїв містить деякі привілеї високого рівня, такі як *SeSecurityPrivilege*. Комбінація груп і привілеїв підтверджує, що це токен адміністратора.

Рівень цілісності токена встановлений на **High**, що, як ми вже обговорювали раніше, запобігає випадковому залишенню токена чутливих ресурсів доступними для користувачів, які не є адміністраторами. В інформації

про токен бачимо, що на рівні Identification є токен імперсонації. Щоб отримати токен, який може створити новий процес, необхідний привілей *SeTcbPrivilege*, що означає, що тільки системні служби, такі як служба *Application Information*, можуть отримати токен. Нарешті, бачимо, що токен позначений як підвищений і що тип підвищення токена вказує, що це повний токен.

Давайте проведемо порівняння з обмеженим токеном (лістинг 4.25).

Спочатку отримуємо дескриптор поточного токена і форматуємо його так само, як у лістингу 4.24. У списку груп бачимо, що BUILTIN\Administrators було перетворено на групу UseForDenyOnly.

Будь-яка інша група, яка відповідатиме підвищеній перевірці RID, буде перетворена таким самим чином.

Список привілеїв містить лише п'ять елементів. Це єдині п'ять привілеїв, які може мати обмежений токен. Рівень цілісності токена встановлено на Середній (*Medium*), що нижче за *High* у повному токені. В інформації про токен бачимо, що токен не підвищений, а тип підвищення вказує, що це обмежений токен.

Лістинг 4.25. Відображення властивостей обмеженого токена

```
PS C:\WINDOWS\system32> Use-NtObject($token = Get-NtToken) { Format-NtToken $token -Group -Privilege -Integrity -Information }
GROUP SID INFORMATION
---------------
Name                                                        Attributes
----                                                        ----------
Mandatory Label\High Mandatory Level                        Integrity, IntegrityEnabled
Everyone                                                    Mandatory, EnabledByDefault, Enabled
NT AUTHORITY\Локальний обліковий запис і член групи адміністраторів Mandatory, EnabledByDefault, Enabled
BUILTIN\Administrators                                      Mandatory, EnabledByDefault, Enabled, Owner
BUILTIN\Performance Log Users                               Mandatory, EnabledByDefault, Enabled
BUILTIN\Users                                               Mandatory, EnabledByDefault, Enabled
NT AUTHORITY\INTERACTIVE                                    Mandatory, EnabledByDefault, Enabled
CONSOLE LOGON                                               Mandatory, EnabledByDefault, Enabled
NT AUTHORITY\Authenticated Users                            Mandatory, EnabledByDefault, Enabled
NT AUTHORITY\This Organization                              Mandatory, EnabledByDefault, Enabled
MicrosoftAccount\hladkykh.v@gmail.com                       Mandatory, EnabledByDefault, Enabled
NT AUTHORITY\Локальний обліковий запис                      Mandatory, EnabledByDefault, Enabled
NT AUTHORITY\LogonSessionId_0_339658873                     Mandatory, EnabledByDefault, Enabled, LogonId
LOCAL                                                       Mandatory, EnabledByDefault, Enabled
NT AUTHORITY\Автентифікація облікового запису хмари         Mandatory, EnabledByDefault, Enabled

PRIVILEGE INFORMATION
---------------------
Name                                Luid             Enabled
----                                ----             -------
SeIncreaseQuotaPrivilege            00000000-00000005 False
SeSecurityPrivilege                 00000000-00000008 False
SeTakeOwnershipPrivilege            00000000-00000009 False
SeLoadDriverPrivilege               00000000-0000000A False
SeSystemProfilePrivilege            00000000-0000000B False
SeSystemTimePrivilege               00000000-0000000C False
SeProfileSingleProcessPrivilege     00000000-0000000D False
SeIncreaseBasePriorityPrivilege     00000000-0000000E False
SeCreatePageFilePrivilege           00000000-0000000F False
SeBackupPrivilege                   00000000-00000011 False
SeRestorePrivilege                  00000000-00000012 False
SeShutdownPrivilege                 00000000-00000013 False
SeDebugPrivilege                    00000000-00000014 True
SeSystemEnvironmentPrivilege        00000000-00000016 False
SeChangeNotifyPrivilege             00000000-00000017 True
SeRemoteShutdownPrivilege           00000000-00000018 False
SeUndockPrivilege                   00000000-00000019 False
SeManageVolumePrivilege             00000000-0000001C False
SeImpersonatePrivilege              00000000-0000001D True
SeCreateGlobalPrivilege             00000000-0000001E True
SeIncreaseWorkingSetPrivilege       00000000-00000021 False
SeTimeZonePrivilege                 00000000-00000022 False
SeCreateSymbolicLinkPrivilege       00000000-00000023 False
SeDelegateSessionUserImpersonatePrivilege 00000000-00000024 False

INTEGRITY LEVEL
---------------
High

TOKEN INFORMATION
-----------------
Type          : Primary
ID            : 00000000-1783CA4F
Auth ID       : 00000000-143ECB28
Origin ID     : 00000000-000003E7
Modified ID   : 00000000-1783E255
Session ID    : 2
Elevated      : True
Elevation Type: Full
Flags         : NotLow
```

Нарешті, зверніть увагу, що прапорці містять значення *IsFiltered*. Цей прапорець вказує, що токен був відфільтрований за допомогою системного виклику *NtFilterToken*.

Це пов'язано з тим, що для створення обмеженого токена LSASS спочатку створює новий повний токен, щоб його ідентифікатор автентифікації мав унікальне значення. (Якщо порівняти значення Auth ID в лістингах 4.24 і 4.25, ви помітите, що вони дійсно різні). Це дозволяє SRM розглядати два токени як окремі сеанси входу.

Потім LSASS передає токен до *NtFilterToken* з прапором параметра *LuaToken*, щоб перетворити будь-яку групу на *UseForDenyOnly* і видалити всі привілеї, крім п'яти дозволених. Однак *NtFilterToken* не знижує рівень цілісності з високого до середнього. Це потрібно робити окремо. Нарешті, LSASS викликає *NtSetInformationToken*, щоб зв'язати два токени разом за допомогою класу інформації *TokenLinkedToken*.

Існує третій тип підвищення, за замовчуванням, який використовується для будь-якого токена, не пов'язаного з адміністратором розділеного токена:

```
PS C:\WINDOWS\system32> Use-NtObject($token = Get-NtToken -Anonymous) { $token.ElevationType }
Default
```

У цьому прикладі *анонімний* користувач не є адміністратором розділеного токена, тому токен має тип підвищення за замовчуванням.

4.9.2. Доступ до інтерфейсу користувача

Однією з інших функцій безпеки, впроваджених у Windows Vista, є ізоляція привілеїв користувацького інтерфейсу (**UIPI**), яка запобігає програмній взаємодії процесів з нижчими привілеями з користувацьким інтерфейсом процесів з вищими привілеями. Це забезпечується за допомогою рівнів цілісності, і це ще одна причина, чому адміністратори **UAC** працюють на високому рівні цілісності.

Але UIPI створює проблему для додатків, які призначені для взаємодії з інтерфейсом користувача, таких як програми для читання з екрану та сенсорні клавіатури. Щоб обійти це обмеження, не надаючи процесу занадто багато привілеїв, токен може встановити прапорець *доступу до інтерфейсу користувача*.

Чи надається процесу доступ до інтерфейсу користувача, залежить від налаштування **UiAccess** у маніфест-файлі виконуваного файлу.

Цей прапор доступу до інтерфейсу користувача сигналізує робочому середовищу, що воно повинно вимкнути перевірки UIPI. У лістингу 4.26 запитуємо цей прапор у відповідному процесі, екранній клавіатурі (OSK).

Лістинг 4.26. Запит прапорця доступу до інтерфейсу користувача в основному токені екранної клавіатури

```
PS C:\WINDOWS\system32> $process = Start-Process "osk.exe" -PassThru
PS C:\WINDOWS\system32> $token = Get-NtToken -ProcessId $process.Id
PS C:\WINDOWS\system32> $token.UIAccess
True
```

Запускаємо OSK і відкриваємо його об'єкт **Token**, щоб запитати прапор доступу до інтерфейсу користувача.

Щоб встановити цей прапор, користувачеві потрібно мати привілей *SeTcbPrivilege*. Єдиний спосіб створити процес доступу до інтерфейсу користувача як звичайний користувач — це скористатися службою **UAC**.

Тому будь-який процес доступу до інтерфейсу користувача потрібно запускати за допомогою *ShellExecute*, саме тому використовували *Start-Process* у лістингу 4.26. Все це відбувається за фоном під час створення програми доступу до інтерфейсу користувача.

4.9.3. Віртуалізація

Іншою проблемою, що з'явилася в Vista через UAC, є питання про те, як поводитися зі старими програмами, які очікують, що зможуть записувати в місця, доступні тільки адміністратору, такі як каталог Windows або розділ реєстру локального комп'ютера.

Vista реалізувала спеціальне обхідне рішення: якщо прапорець віртуалізації увімкнене на первинному токені, він без повідомлення перенаправить записи з цих місць у сховище для кожного користувача. Це створює враження, ніби користувач успішно додав ресурси до захищених місць.

За замовчуванням прапор *віртуалізації* вмикається в застарілих програмах автоматично. Однак ви можете вказати його вручну, встановивши властивість на первинному токені. Виконайте команди в лістингу 4.27 у оболонці, що має прав адміністратора.

Лістинг 4.27. Включення віртуалізації для об’єкту **Token** та створення файлу в C:\Windows

```
PS C:\WINDOWS\system32> $file = New-NtFile -Win32Path C:\Windows\hello.txt -Access GenericWrite
PS C:\WINDOWS\system32> $token = Get-NtToken
PS C:\WINDOWS\system32> $token.VirtualizationEnabled = $true
PS C:\WINDOWS\system32> $file = New-NtFile -Win32Path C:\Windows\hello.txt -Access GenericWrite
New-NtFile : (0xC0000035) - Object Name already exists.
At line:1 char:9
+ $file = New-NtFile -Win32Path C:\Windows\hello.txt -Access GenericWri ...
+         ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : NotSpecified: (:) [New-NtFile], NtException
    + FullyQualifiedErrorId : NtCoreLib.NtException,NtObjectManager.Cmdlets.Object.NewNtFileCmdlet

PS C:\WINDOWS\system32> $file.Win32PathName
C:\Windows\hello.txt
```

У цьому лістингу спочатку намагаємося створити файл для запису, C:\Windows\hello.txt.

Ця операція завершується з помилкою «доступ заборонено». Потім отримуємо поточний основний токен і встановлюємо властивість *VirtualizationEnabled* у значення *True*.

Коли повторюємо операцію створення файлу, вона тепер виконується успішно. Якщо ми запитуємо розташування файлу, ми бачимо, що він знаходиться в каталозі користувача у віртуальному сховищі. Тільки *звичайні* токени без привілеїв можуть увімкнути віртуалізацію. У токенів системних служб та адміністраторів віртуалізація вимкнена. Ви можете дізнатися, чи дозволена віртуалізація, запитавши властивість *VirtualizationAllowed* об'єкта Token.

4.10. АТРИБУТИ БЕЗПЕКИ

Атрибути безпеки токена – це список пар «ім'я/значення», які містять дані. Існує три типи атрибутів безпеки, пов'язаних з токеном: *локальні (*local*)*, вимоги *користувача* (user claims*)* та вимоги *пристрою* (device claims). Кожен атрибут безпеки може мати одне або кілька значень, які повинні бути одного типу.

У таб.4.5 наведено допустимі типи для атрибута безпеки.

Майже кожен токен процесу має атрибут безпеки TSA://ProcUnique.

Цей атрибут безпеки містить унікальний LUID, який присвоюється під час створення процесу. Ми можемо відобразити його значення для ефективного токена за допомогою Show-NtToken Effective, як показано в лістингу 4.28.

Лістинг 4.28. Запит атрибутів безпеки для поточного процесу

```
PS C:\WINDOWS\system32> Show-NtTokenEffective -SecurityAttributes
SECURITY ATTRIBUTES
-------------------
Name            Flags                ValueType Values
----            -----                --------- ------
TSA://ProcUnique NonInheritable, Unique UInt64    {295, 394512977}
```

Таблиця 4.5.

Типи атрибутів безпеки

Назва типу	Опис
Int64	Знакове 64-бітове ціле число
UInt64	Без знакове 64-бітове ціле число
String	Рядок Unicode
Fqbn	Повне двійкове ім'я; містить номер версії та рядок Unicode
Sid	SID
Boolean	Значення true або false, що зберігається як Int64, де 0 означає false, а 1 – true
Octetstring	Довільний масив байтів

Набір прапорців може бути призначений атрибуту безпеки для зміни певних аспектів його поведінки, наприклад, чи можуть нові токени успадковувати його. У таб.4.6 наведено визначені прапорці.

Таблиця 4.6.

Прапорці атрибутів безпеки

Назва прапора	Опис
NonInheritable	Атрибут безпеки не може бути успадкований дочірнім токеном процесу.
CaseSensitive	Якщо атрибут безпеки містить рядкове значення, порівняння має бути чутливим до регістру.
UseForDenyOnly	Атрибут безпеки використовується лише під час перевірки заборони доступу.
DisabledByDefault	Атрибут безпеки вимкнено за замовчуванням.
Disabled	Атрибут безпеки вимкнено.
Mandatory	Атрибут безпеки є обов'язковим.
Unique	Атрибут безпеки має бути унікальним у локальній системі.
InheritOnce	Атрибут безпеки може бути успадкований дочірнім об'єктом один раз, після чого його слід встановити як неуспадкований.

З вихідних даних ми бачимо, що ім'я атрибуту — TSA:// ProcUnique. Він має два значення UInt64, які в сукупності утворюють LUID.

Нарешті, він має два прапорці: *NonInheritable*, що означає, що атрибут безпеки не буде переданий новим токенам процесу, та *Unique*, що означає, що

ядро не повинно намагатися об'єднати атрибут безпеки з будь-яким іншим атрибутом у системі з таким самим ім'ям.

Щоб встановити локальні атрибути безпеки, викликаюча сторона повинна мати привілей *SeTcbPriv* перед викликом *NtSetInformationToken*. Вимоги користувачів і пристроїв повинні бути встановлені під час створення токена.

4.11. СТВОРЕННЯ ТОКЕНІВ

Зазвичай LSASS створює токени, коли користувач проходить автентифікацію на комп'ютері.

Однак LSASS також може створювати токени для користувачів, яких не існує, наприклад для віртуальних облікових записів, що використовуються для служб.

Ці токени можуть бути інтерактивними, для використання в сеансі консолі, або мережевими токенами для використання в локальній мережі. Локально автентифікований користувач може створити токен іншого користувача, викликавши Win32 API, такий як *LogonUser*, який викликає LSASS для створення токена.

Варто зрозуміти, як LSASS створює токени. Для цього LSASS викликає системний виклик *NtCreateToken*. Як вже згадувалось раніше, цей системний виклик вимагає привілею *SeCreateTokenPrivilege*, який надається обмеженій кількості процесів. Цей привілей є максимально можливим, оскільки ви можете використовувати його для створення довільних токенів з будь-яким SID групи або користувача та отримати доступ до будь-якого ресурсу на локальній машині.

Хоча вам не доведеться часто викликати *NtCreateToken* з PowerShell, ви можете зробити це за допомогою команди *New-NtToken*, якщо у вас є *SeCreateTokenPrivilege*. Системний виклик *NtCreateToken* приймає такі параметри:

Тип токена **(Token type**) - Первинний або імперсонації;

Ідентифікатор аутентифікації **(Authentication ID) -** Ідентифікатор аутентифікації LUID. Може бути встановлений будь-яке бажане значення;

Час закінчення терміну дії **(Expiration time) -** Дозволяє закінчуватися терміну дії токена через встановлений період;

Користувач **(User) -** Ідентифікатор безпеки користувача;

Групи**(Groups) -** Список ідентифікаторів безпеки груп;

Привілеї **(Privileges) -** Список привілеїв;

Власник **(Owner) -** Ідентифікатор безпеки власника;

Первинна група **(Primary group)-** Ідентифікатор безпеки первинної групи;

Джерело (Source)- Ім'я джерела інформації.

Крім того, починаючи Windows 8 до системного виклику були додані наступні нові функції, доступ до яких можна отримати за допомогою системного виклику *NtCreateTokenEx*:

Групи пристроїв **(Device groups) -** Список додаткових ідентифікаторів безпеки для пристрою;

Атрибути вимог до пристроїв **(Device claim attributes) -** Список атрибутів безпеки для визначення вимог до пристроїв;

Атрибути вимог користувачів **(User claim attributes) -** Список атрибутів безпеки для визначення вимог користувачів;

Обов'язкова політика**(Обов'язкова політика) -** Набір прапорців, що вказують на обов'язкову політику цілісності токена.

Все, чого немає в цих двох списках, можна налаштувати лише викликом NtSetInformationToken після створення нового токена. Залежно від того, яка властивість токена встановлюється, може знадобитися інший привілей, наприклад SeTcbPrivilege. Продемонструємо, як створити новий токен за допомогою скрипта в лістингу 4.29, який потрібно запустити від імені адміністратора.

Лістинг 4.29 - Створення нового токена

```
PS> Enable-NtTokenPrivilege SeDebugPrivilege
PS> $imp = Use-NtObject($p = Get-NtProcess -Name lsass.exe) {
Get-NtToken -Process $p -Duplicate
}
PS> Enable-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
       ----                           ----------
Everyone                       Mandatory, EnabledByDefault, Enabled
Mandatory Label\System Mandatory Level Integrity, IntegrityEnabled
```

Звичайний адміністратор за замовчуванням не має привілею *SeCreateTokenPrivilege*. Тому нам потрібно позичити токен в іншого процесу, який має цей привілей. У більшості випадків найпростішим процесом для позичання є LSASS.

Ми відкриваємо процес LSASS і його токен, дублюючи його в токен імперсонації. Далі переконуємося, що *SeCreateTokenPrivilege* увімкнена на токені. Потім ми можемо імперсонувати токен і викликати *New-NtToken*, передаючи йому SID для користувача однієї групи. Нарешті, можемо вивести деталі нового токена, включаючи його набір SID користувача та набір груп. Команда *New-NtToken* також додає SID рівня цілісності системи за замовчуванням, який ви можете побачити в списку груп.

4.11. ПРИЗНАЧЕННЯ ТОКЕНІВ

Якщо звичайний обліковий запис користувача може призначати довільні первинні токени або токени імперсонації, він може підвищити свої привілеї для доступу до ресурсів інших користувачів.

Це буде особливо проблематично, коли мова йде про імперсонацію, оскільки іншому обліковому запису користувача потрібно лише відкрити

іменований канал, щоб ненавмисно дозволити серверу отримати токен імперсонації.

З цієї причини SRM накладає обмеження на те, що може робити звичайний користувач без привілеїв SeAssignPrimaryTokenPrivilege та SeImpersonationPrivilege.

Давайте розглянемо критерії, які повинні бути виконані для призначення токена звичайному користувачу.

4.11.1. Призначення основного токена

Новому процесу можна призначити первинний токен одним з трьох способів:

—Він може успадкувати токен від батьківського процесу.

—Токен можна призначити під час створення процесу (наприклад, за допомогою API CreateProcessAsUser).

—Токен може бути встановлений після створення процесу за допомогою NtSetInformationProcess, до запуску процесу.

Успадкування токена від батьківського процесу є найпоширенішим способом призначення токенів. Наприклад, коли запускаєте програму з меню «Пуск», новий процес успадкує токен від процесу Explorer.

Якщо процес не успадковує токен від свого батьківського процесу, йому буде передано токен як дескриптор, який повинен мати право доступу *AssignPrimary*.

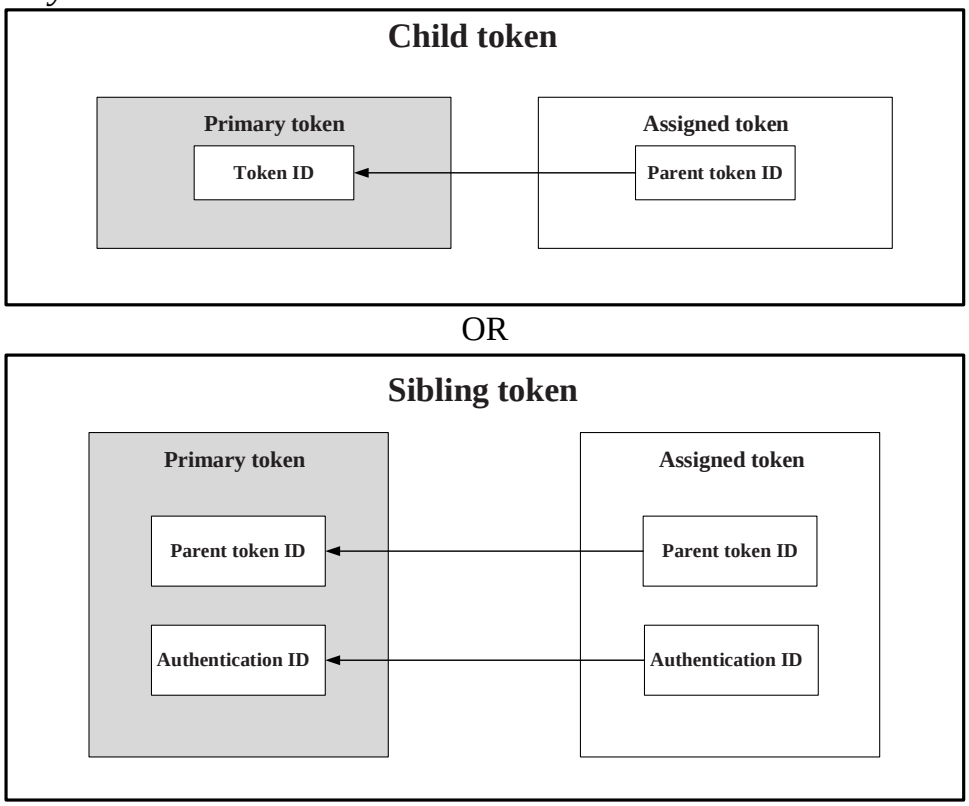


Рисунок 4.4 - Основні критерії призначення токенів *SeIsTokenAssignableToProcess*

Якщо доступ до об'єкта Token надано, SRM накладає додаткові критерії до токена, щоб запобігти призначенню більш привілейованого токена (якщо тільки первинний токен користувача не має увімкненого привілею *SeAssignPrimaryTokenPrivilege*).

Функція ядра *SeIsTokenAssignableToProcess* встановлює вимоги до токену. Спочатку функція перевіряє, чи рівень цілісності призначеного токена менший або дорівнює рівню цілісності первинного токена поточного процесу. Якщо цей критерій виконано, то перевіряється, чи відповідає токен одному з критеріїв, показаних на рис. 4.4. А саме, чи є токен дочірнім для первинного токена користувача або спорідненим з первинним токеном.

Спочатку розглянемо випадок дочірнього токена. Процес користувача може створити новий токен на основі існуючого. Коли це відбувається, властивість *ParentTokenId* в об'єкті ядра нового токена встановлюється на ідентифікатор батьківського токена. Якщо *ParentTokenId* нового токена збігається з поточним значенням ідентифікатора первинного токена, то присвоєння буде дозволено. Обмежені токени є прикладами дочірніх токенів. Коли ви створюєте обмежений токен за допомогою *NtFilterToken*, ідентифікатор батьківського токена нового токена встановлюється рівним ідентифікатору первинного токена.

Токен-брат - це токен, створений як частина тієї самої автентифікації, що й існуючий токен. Для перевірки цього критерію функція порівнює ідентифікатор батьківського токена та ідентифікатори автентифікації двох токенів. Якщо вони збігаються, то токен можна призначити. Ця перевірка також перевіряє, чи сеанси автентифікації є спеціальними сеансами споріднених сеансів, встановлених ядром (досить рідкісна конфігурація). Поширеними прикладами споріднених токенів є токени, які дублюють токен поточного процесу, а також токени lowbox.

Зверніть увагу, що функція не перевіряє користувача, якого представляє токен, і якщо токен відповідає одному з критеріїв, його можна призначити новому процесу. Якщо він не відповідає критеріям, то під час призначення токена буде повернуто помилку STATUS_PRIVILEGE_NOT _HELD.

Як утиліта *runas* створює новий процес як звичайний користувач з цими обмеженнями? Вона використовує API *CreateProcessWithLogon*, який автентифікує користувача і запускає процес від системної служби, яка має необхідні привілеї для обходу цих перевірок.

Якщо ми спробуємо призначити токен процесу, то побачимо, як легко операція може завершитися невдачею, навіть якщо ми призначаємо токени для одного і того ж користувача. Запустіть код у лістингу 4.30 від імені користувача без прав адміністратора..

Лістинг 4.30. Створення процесу з використанням обмежених токенів

```
PS C:\WINDOWS\system32> $token = Get-NtToken -Filtered -Flags DisableMaxPrivileges
PS C:\WINDOWS\system32> Use-NtObject($proc = New-Win32Process notepad -Token $token) { $proc | Out-Host }


Process            : Notepad.exe
Thread             : thread:1040 - process:13868
Pid                : 13868
Tid                : 1040
TerminateOnDispose : False
ExitStatus         : 259
ExitNtStatus       : STATUS_PENDING



PS C:\WINDOWS\system32> $token = Get-NtToken -Filtered -Flags DisableMaxPrivileges -Token $token
PS C:\WINDOWS\system32> $proc = New-Win32Process notepad -Token $token
```

У цьому прикладі створимо два обмежених токени і використаємо їх для створення екземпляра **Notepad**. У першій спробі створюємо токен на основі поточного первинного токена. Поле ID батьківського токена в новому токені встановлюється на ID первинного токена, і коли використовуємо токен під час створення процесу, операція проходить успішно.

Під час другої спроби створюємо ще один токен, але вже на основі раніше створеного. Створення процесу з цим токеном завершується невдачею з помилкою привілеїв. Це відбувається тому, що ідентифікатор батьківського токена другого токена встановлений на ідентифікатор створеного токена, а не первинного токена. Оскільки токен не відповідає ні критерію спорідненого, ні критерію рідного брата, ця операція завершиться невдачею під час присвоєння.

Ви можете встановити маркер після створення процесу за допомогою системного виклику *NtSetInformationProcess* або *ProcessAccessToken*, який PowerShell відкриває за допомогою команди *Set-NtToken* (показано у лістингу 4.31).

Лістинг 4.31. Встановлення токену доступу після запуску процесу

```
PS C:\WINDOWS\system32> $proc = New-Win32Process notepad -Token $token
PS C:\WINDOWS\system32> $proc = Get-NtProcess -Current
PS C:\WINDOWS\system32> $token = Get-NtToken -Duplicate -TokenType Primary
PS C:\WINDOWS\system32> Set-NtToken -Process $proc -Token $token
Set-NtToken : (0xC00000BB) - The request is not supported.
At line:1 char:1
+ Set-NtToken -Process $proc -Token $token
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : NotSpecified: (:) [Set-NtToken], NtException
    + FullyQualifiedErrorId : NtCoreLib.NtException,NtObjectManager.Cmdlets.Object.SetNtTokenCmdlet
```

Ця операція призначення не оминає жодної з перевірок призначення, які ми обговорювали. Як тільки початковий потік процесу починає виконуватися, можливість встановити первинний токен відключається, тому, коли ми намагаємося встановити токен у запущеному процесі, ми отримуємо помилку STATUS_UNSUPPORTED.

4.11.2. Призначення токена імперсонації

Як і у випадку з первинними токенами, SRM вимагає, щоб призначений токен імперсоналізації відповідав певному набору критеріїв. В іншому випадку він відхилить призначення токена потоку. Важливо, що ці критерії відрізняються від критеріїв для призначення первинних токенів.

Це може призвести до ситуацій, коли можна призначити токен імперсоналізації, але не можна призначити первинний токен, і навпаки.

Якщо токен вказано явно, то дескриптор повинен мати право доступу *Impersonate*. Якщо імперсоналізація відбувається неявно, то ядро вже підтримує токен, і він не потребує спеціальних прав доступу.

Функція *SeTokenCanImpersonate* в ядрі виконує перевірку на відповідність критеріям імперсоналізації. Як показано на рис. 4.5, ця перевірка є значно складнішою, ніж при призначенні первинних токенів.

Розглянемо кожну перевірку і опишемо, що вона враховує як для токену імперсоналізації, так і для первинного токена. Зауважте, що оскільки можна призначити токен імперсоналізації потоку в іншому процесі (якщо у вас є відповідний дескриптор для цього потоку), первинним токеном, який перевіряється, є токен, призначений процесу, який інкапсулює потік, а не первинний токен потоку, що викликає потік.

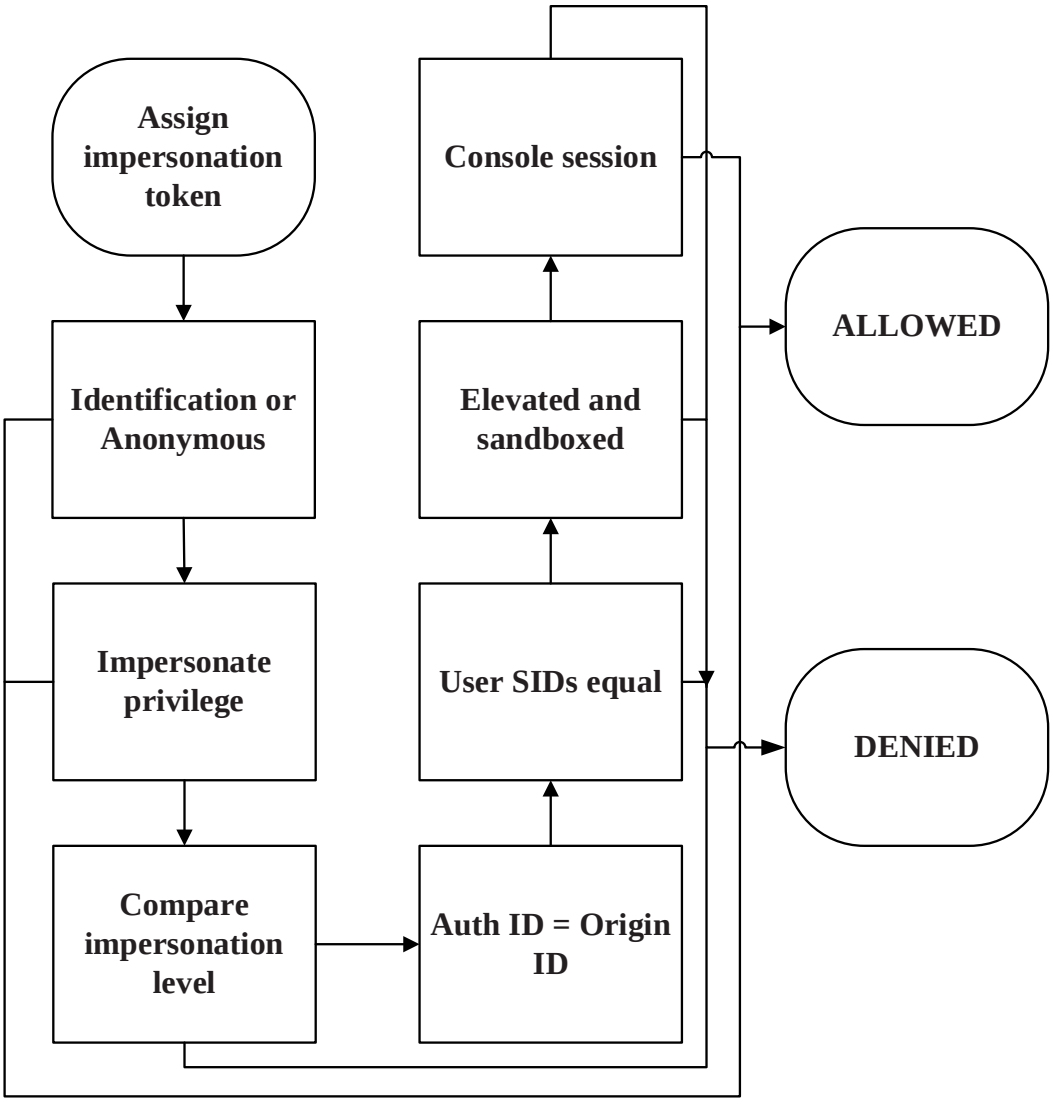


Рисунок 4.5. *SeTokenCanImpersonate* виконує перевірку токену імперсоналізації

Функція виконує наступні кроки перевірки:

1. Перевірка рівня імперсоналізації - *Імперсоналізації* або *Анонімний*. Якщо токен імперсоналізації має один з цих рівнів, призначення його потоку не становить загрози безпеці, і SRM негайно дозволяє призначення.

Ця перевірка також дозволяє призначення, якщо токен імперсоналізації представляє анонімного користувача на основі його ID автентифікації.

2. Перевірка наявності привілеїв на імперсоналізацію. Якщо *SeImpersonatePrivilege* увімкнена, SRM дозволить призначення невідкладно.

3. Порівняння рівнів цілісності основного токену та токену-дублікату. Якщо рівень цілісності основного токена менший за рівень цілісності токена-дубліката, у призначенні буде відмовлено. Якщо вони однакові або більші, перевірка продовжується.

4. Перевірка того, чи ID автентифікації дорівнює ID оригіналу. Якщо ідентифікатор входу в систему оригінального токена дорівнює ідентифікатору автентифікації первинного токена, SRM дозволяє призначення.

В іншому випадку, він продовжує перевірку.

Зауважте, що ця перевірка має цікавий наслідок. Як обговорювалося раніше у цьому розділі, початковий ідентифікатор входу у систему для токенів звичайних користувачів встановлюється як ідентифікатор автентифікації користувача SYSTEM. Це пов'язано з тим, що процес автентифікації виконується від імені користувача SYSTEM. Тому користувач SYSTEM може видавати себе за будь-який інший токен у системі, якщо він відповідає вимогам рівня цілісності, навіть якщо привілей SeImpersonatePrivilege не увімкнена.

5. Перевірка рівності ідентифікаторів користувачів. Якщо ідентифікатор користувача основного токена не дорівнює ідентифікатору користувача токена-дубліката, SRM відмовляє у призначенні. В іншому випадку, він продовжує перевірку. Цей критерій дозволяє користувачеві використовувати свій власний обліковий запис, але блокує використання облікового запису іншого користувача, якщо у нього немає облікових даних іншого користувача. Під час автентифікації іншого користувача LSASS повертає токен імперсоналізації з початковим ідентифікатором входу, встановленим на ідентифікатор автентифікації абонента, тому токен пройде попередню перевірку, і SID користувачів ніколи не будуть порівнюватися.

6. Перевірка прапорця *Elevated* (Підвищений). Ця перевірка гарантує, що зловмисник не зможе підмінити собою більш привілейований токен. Якщо на токені-дублікаті встановлено прапорець *Elevated*, а на первинному токені - ні, то імітація буде відхилена. У версіях Windows до 10 ця перевірка не виконувалася, тому раніше можна було видавати себе за токен адміністратора UAC, якщо попередньо знизити рівень цілісності.

7. Перевірка на наявність «пісочниці». Ця перевірка гарантує, що користувач не зможе імітувати менш захищений токен. Щоб імітувати *lowbox* - токен, новий токен повинен або збігатися з SID пакета, або бути обмеженим SID пакета первинного токена. В іншому випадку буде відмовлено в уособленні.

Список можливостей не перевіряється. Для обмеженого токена достатньо, щоб новий токен також був обмеженим токеном, навіть якщо список обмежених SID відрізняється. Те ж саме стосується і токенів з обмеженням на запис.

SRM має різні механізми захисту, щоб ускладнити отримання більш привілейованого токена пісочниці.

8. Перевірка сеансу консолі. На цьому завершальному кроці перевіряється, чи є сеанс консолі сеансом 0 чи ні. Це запобігає тому, що користувач може видавати себе за токен у сеансі 0, який може надавати підвищені привілеї (наприклад, можливість створювати глобальні об'єкти розділів).

Ви можете припустити, що якщо функція відмовить у призначенні, вона поверне помилку STATUS_PRIVILEGE_NOT_HELD, але це не так. Замість цього SRM дублює токен імперсоніфікації як токен рівня Identifikation і призначає його. Це означає, що навіть якщо уособлення не вдасться, потік все одно може перевірити властивості токена..

Перевірити, чи можна імперсоналізувати токен, можна за допомогою команди *Test-NtTokenImpersonation* PowerShell. Ця команда уособлює токен і повторно відкриває його в потоці. Потім вона порівнює рівень цілісності оригінального токена і повторно відкритого токена і повертає булеве значення. У лістингу 4.32 наведено простий приклад, який не відповідає критеріям перевірки рівня цілісності. Зауважте, що краще не запускати цей сценарій у PowerShell, про яку ви дбаєте, оскільки ви не зможете відновити початковий рівень цілісності.

Лістинг 4.32. Перевірка імперсоналізації токену

```
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
```

Ці перевірки досить прості. Спочатку ми отримуємо дублікат токена поточного процесу і передаємо його в *Test-NtTokenImpersonation*. Результат буде **True**, що вказує на те, що ми можемо уособлювати токен на рівні **Impersonation**. Для наступної перевірки ми знизимо рівень цілісності первинного токена поточного процесу до **Low** і запустимо тест знову. Цього разу він повертає значення **False**, оскільки видавати себе за токен на рівні **Impersonation** більше неможливо.

Нарешті, ми перевіряємо, чи можемо ми видавати себе за токен на рівні ідентифікації, який також повертає значення **True**.

4.12. ПРАКТИЧНІ ПРИКЛАДИ

Розглянемо кілька робочих прикладів, щоб ви могли дізнатися, як застосовувати різні команди, представлені в цьому розділі, для дослідження безпеки або аналізу системи.

4.12.1. Пошук процесів з доступом до UI

У деяких випадках корисно переглянути всі процеси, до яких ви маєте доступ, і перевірити властивості їхніх первинних токенів. Це може допомогти

вам знайти процеси, запущені від імені певних користувачів або з певними властивостями. Наприклад, ви можете виявити процеси із встановленим прапорцем доступу до інтерфейсу користувача. Раніше у цьому розділі ми обговорювали, як перевірити прапорець доступу до інтерфейсу користувача окремо. У лістингу 4.33 ми виконаємо перевірку для всіх процесів, до яких ми можемо отримати доступ.

Лістинг 4.33. Пошук процесів з доступом до інтерфейсу користувача

```
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()
```

Спочатку викликаємо команду *Get-NtProcess*, щоб відкрити всі процеси з доступом *QueryLimitedInformation*. Далі створюємо скрипт для фільтрації. Якщо скрипт поверне значення **True**, команда поверне процес, інакше вона закриває дескриптор процесу.

У цьому скрипті ми відкриваємо токен процесу для доступу до *Query* і повертаємо властивість *UIAccess*. Результат фільтрує список процесів, залишаючи лише ті, що працюють з токенами доступу до інтерфейсу користувача. Відображаємо знайдені процеси.

4.12.2. Пошук токенів для імперсоналізації

Існує кілька офіційних способів отримати доступ до токена для імперсоналізації, наприклад, за допомогою віддаленого виклику процедури або відкриття первинного токена процесу. Інший підхід полягає у пошуку існуючих дескрипторів об'єктів токенів, які ви можете скопіювати і використовувати.

Цей метод може бути корисним, якщо ви працюєте від імені користувача, який не є адміністратором і має привілей *SeImpersonatePrivilege* (як у випадку службового облікового запису, наприклад, LOCAL SERVICE), або для оцінки безпеки пісочниці, щоб переконатися, що пісочниця не може відкритися і видати себе за більш привілейований токен.

Лістинг 4.34. Пошук дескрипторів токенів з підвищеними правами

```
PS> function Get-ImpersonationTokens {
$hs = Get-NtHandle -ObjectType Token
                foreach($h in $hs) {
                   try {
                Use-NtObject($token = Copy-NtObject -Handle $h) {
                    if (Test-NtTokenImpersonation -Token $token) {
                          Copy-NtObject -Object $token }
                                  }
                      }
catch {
        }
      }
    }
PS> $tokens = Get-ImpersonationTokens 5 PS> $tokens | Where-Object Elevated
```

Ви також можете використовувати цю техніку для доступу до ресурсів іншого користувача, дочекавшись, поки він підключиться до комп'ютера Windows, наприклад, по мережі. Якщо ви захопите токен користувача, ви зможете повторно використовувати його ідентифікатор без необхідності знати його пароль. У лістингу 4.34 показано просту реалізацію цієї ідеї.

У функції *Get-ImpersonationTokens* отримуємо список усіх дескрипторів типу **Token** за допомогою команди *Get-NtHandle*. Потім для кожного дескриптора намагаємось дублювати дескриптор для поточного процесу за допомогою *Copy-NtObject*. Якщо це вдається, то перевіряємо, чи можемо успішно уособити токен. Якщо так, то робимо ще одну копію токена, щоб вона не була закрита.

Запуск функції *Get-ImpersonationTokens* повертає всі доступні дескриптори токенів, які можна уособлювати. За допомогою цих об'єктів токенів ми можемо запитувати властивості, які нас цікавлять. Наприклад, ми можемо перевірити, чи є токен підвищеним чи ні, що може свідчити про те, що ми зможемо використати токен для отримання додаткових привілейованих груп.

4.12.3. Видалення привілеїв адміністратора

Під час запуску програми від імені адміністратора вам може знадобитися тимчасово скасувати свої привілеї, щоб ви могли виконати певну операцію без пошкодження системи, наприклад, випадково видалити системні файли. Для виконання операції ви можете скористатися тим самим підходом, що і UAC для створення відфільтрованого токена з меншими привілеями. Запустіть код у лістингу 4.35 від імені адміністратора..

Лістинг 4.35. Відключення прав адміністратора

```
PS> $token = Get-NtToken -Filtered -Flags LuaToken
PS> Set-NtTokenIntegrityLevel Medium -Token $token
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()
```

Почнемо з перевірки поточного токена і встановлення прапорця *LuaToken*.

Цей прапор видаляє всі групи адміністраторів і додаткові привілеї, які не може мати обмежений токен. Прапор *LuaToken* не знижує рівень цілісності токена, тому ми повинні встановити його на **Medium** вручну. Ми можемо переконатися, що токен більше не вважається адміністраторським, перевіривши, що властивість *Elevated* має значення **False**.

Щоб побачити ефект в дії, тепер ми можемо записати файл в місце, доступне тільки для адміністратора, наприклад, в каталог Windows. Якщо ми спробуємо зробити це за допомогою токена поточного процесу, операція пройде успішно. Однак, коли ми намагаємося виконати операцію, видаючи себе за токен, вона зазнає невдачі з помилкою відмови у доступі. Ви також можете використовувати токен за допомогою команди *New-Win32Process PowerShell* для запуску нового процесу з менш привілейованим токеном.

4.13. ВИСНОВКИ ДО РОЗДІЛУ 4

У цьому розділі ми познайомилися з двома основними типами токенів: первинними токенами, які асоціюються з процесом, та токенами імперсоналізації, які асоціюються з потоком і дозволяють процесу тимчасово видавати себе за іншого користувача. Ми розглянули важливі властивості обох типів токенів, такі як групи, привілеї та рівні цілісності, а також те, як ці властивості впливають на безпеку ідентичності, яку надає токен. Потім ми обговорили два типи токенів «пісочниці» (*restricted* та *lowbox*), які використовуються такими програмами, як веб-браузери та програми для читання документів, щоб обмежити шкоду від потенційного експлойту для віддаленого виконання коду.

Далі ми розглянули, як токени використовуються для представлення привілеїв адміністратора, включаючи те, як Windows реалізує контроль облікових записів користувачів і адміністраторів з розділеними токенами для звичайних користувачів настільних комп'ютерів. В рамках цього обговорення ми розглянули особливості того, що операційна система вважає адміністратором або підвищеним токеном.

Нарешті, ми обговорили кроки, пов'язані з призначенням токенів процесам і потокам. Ми визначили конкретні критерії, яким повинен відповідати звичайний користувач, щоб призначити токен, і чим відрізняються перевірки первинних токенів і токенів імперсоналізації.

У наступному розділі ми обговоримо дескриптори безпеки. Вони визначають, який доступ буде надано до ресурсу на основі ідентифікаційних даних та груп, присутніх у токені доступу користувача..

ПИТАННЯ ДЛЯ САМОКОНТРОЛЮ

1. З якою метою Windows використовує токени?
2. Які існують типи токенів?
3. Яку функцію виконують токени *імперсонації* (impersonation token)?
4. Які права доступу можна запитати при відкритті об'єкта Token?
5. Що таке локально унікальний ідентифікатор?
6. Які існують способи призначення токена імперсонації потоку?
7. Надайте опис структури SQoS. Яку функцію відіграє рівень імперсонації?
8. Які існують способи явної імперсонації токена?
9. Чим відрізняється операція дублювання токену від дублювання дескриптора?
10. Функції «пісочниці» (sandbox)?
11. Як Windows реалізує функції «пісочниці» (sandbox)?
12. Що робить користувача адміністратором?
13. Як Windows реалізує контроль облікових записів користувачів?
14. Які атрибути безпеки токена ви знаєте?
15. Як здійснити пошук процесів з доступом до UI?

РОЗДІЛ 5. ДЕСКРИПТОР БЕЗПЕКИ

У попередньому розділі ми обговорили токен безпеки, який описує користувача для SRM. У цьому розділі ви дізнаєтесь, як дескриптори безпеки визначають безпеку ресурсу.

Дескриптор безпеки виконує декілька функцій. Він вказує власника ресурсу, дозволяючи SRM надавати певні права користувачам, які отримують доступ до своїх власних даних. Він також містить дискреційний контроль доступу (discretionary access control, DAC) та обов'язковий контроль доступу (mandatory access control, MAC), які надають або забороняють доступ користувачам та групам доступ до ресурсів. Нарешті дескриптор безпеки може містити записи, які генерують події аудиту.

Майже кожен ресурс ядра має дескриптор безпеки. Програми користувальницького режиму можуть реалізовувати власний контроль доступу через дескриптори безпеки без необхідності створення ресурсу ядра. Розуміння структури дескрипторів безпеки має вирішальне значення для розуміння безпеки Windows, оскільки вони використовуються для захисту об'єкта ядра і багатьох компонентів режиму користувача, таких, наприклад, як служби. Ви навіть знайдете, що дескриптори безпеки використовуються через мережні кордони для захисту віддалених ресурсів.

При розробці програми Windows або дослідженні безпеки Windows вам неминуче доведеться перевіряти або створювати дескриптор безпеки, тому чітке розуміння того, що містить дескриптор безпеки, заощадить багато часу. Щоб допомогти з цим, ми почнемо з докладного опису структури дескриптора безпеки.

5.1. СТРУКТУРА ДЕСКРИПТОРА БЕЗПЕКИ

Windows зберігає дескриптори безпеки у вигляді двійкових структур на диску або в пам'яті. Хоча вам не часто доведеться вручну аналізувати ці структури, варто зрозуміти, що вони містять. Дескриптор безпеки складається з наступних семи компонентів:

— Ревізія;
— Необов'язкові прапорці диспетчера ресурсів;
— Прапорці контролю;
— Необов'язковий SID власника;
— Необов'язковий SID групи;

— Необов'язковий список дискреційного контролю доступу;

— Необов'язковий список системного контролю доступу.

Розглянемо кожен з них по черзі. Першим компонентом будь-якого дескриптора безпеки є ревізія, яка вказує на версію двійкового формату дескриптора безпеки. На сьогодні існує лише одна версія, тому ревізія завжди має значення 1. Далі йде додатковий набір прапорів для використання диспетчера ресурсів.

Ви майже ніколи не зіткнетеся зі встановленням цих прапорців, однак їх використовує Active Directory.

За прапорами менеджера ресурсів слідує набір прапорів управління. Вони мають три призначення:

— визначають, які необов'язкові компоненти дескриптора безпеки є дійсними;

— як були створені дескриптори та компоненти безпеки;

— як обробляти дескриптор безпеки під час його застосування до об'єкта.

В таб.5.1 надано список допустимих прапорів та його описи.

Таблиця 5.1.

Допустимі прапорці управління

Назва	Значення	Опис
OwnerDefaulted	0x0001	SID власника було призначено за замовчуванням.
GroupDefaulted	0x0002	SID групи було призначено за замовчуванням.
DaclPresent	0x0004	DACL присутній у дескрипторі безпеки.
DaclDeTaulted	0x0008	DACL призначений за замовчуванням.
SaclPresent	0x0010	SACL присутній у дескрипторі безпеки.
SaclDefaulted	0x0020	SACL призначений за замовчуванням.
DaclUntrusted	0x0040	У поєднанні з ServerSecurity DACL є ненадійним.
ServerSecurity	0x0080	DACL змінено на ACL сервера.
DaclAutoInheritReq	0x0100	Запитується автоспадкування DACL для дочірніх об'єктів.
SaclAutoInheritReq	0x0200	Запитується автоспадкування SACL для дочірніх об'єктів.
DaclAutoInherited	0x0400	DACL підтримує автоспадкування.
SaclAutoInherited	0x0800	SACL підтримує автоспадкування.
DaclProtected	0x1000	DACL захищений від успадкування.
SaclProtected	0x2000	SACL захищений від успадкування.
RmControlValid	0x4000	Прапорці менеджера ресурсів дійсні.
SelfRelative	0x8000	Дескриптор безпеки має відносний формат.

Після прапорів керування йде SID власника. Зазвичай це SID користувача, проте право власності також може бути призначене групі, такій як група

«Адміністратори». Будучи власником ресурсу, ви отримуєте певні привілеї, включаючи можливість змінювати дескриптор безпеки ресурсу. Забезпечуючи власнику таку можливість, система запобігає блокуванню користувачем доступу до власних ресурсів.

SID групи схожий на ідентифікатор безпеки власника, але використовується нечасто. Він існує в основному для забезпечення сумісності з POSIX (що було проблемою в ті часи, коли Windows ще мала підсистему POSIX) і не відіграє жодної ролі в контролі доступу для додатків Windows.

Найважливішою частиною дескриптора безпеки є дискреційний список контролю доступу (discretionary access control list, DACL). DACL містить список записів контролю доступу (access control entries, ACE), які визначають, який доступ надається SID. Він вважається дискреційним, оскільки користувач або системний адміністратор може вибрати рівень наданого доступу. Існує багато різних типів ACE. Основна інформація в кожному ACE включає наступне:

— SID користувача або групи, до якої застосовується ACE;

— Тип ACE;

— Маска доступу, до якої SID буде дозволено або заборонено доступ.

Останнім компонентом дескриптора безпеки є список контролю доступу безпеки (security access control list, SACL), який містить правила аудиту. Як і DACL, він містить список ACE, але замість визначення доступу на основі того, чи збігається визначений SID з поточним користувачем, він визначає правила для створення подій аудиту під час доступу до ресурсу. Починаючи з Windows Vista, SACL також є кращим місцем для зберігання додаткових ACE, не пов'язаних з аудитом, таких як обов'язкова позначка ресурсу.

Два останні елементи, яким слід приділити увагу в DACL та SACL, це прапорці керування *DaclPresent* та *SaclPresent*. Ці прапорці вказують на те, що DACL та SACL відповідно присутні в дескрипторі безпеки. Використання прапорів дозволяє встановити NULL ACL, де встановлено прапорець "поточний", але значення для поля ACL в дескрипторі безпеки не вказано. NULL ACL вказує на те, що для цього ACL не визначено жодної безпеки, і SRM фактично ігнорує його. Це відрізняється від порожнього ACL, де встановлено прапорець "поточний" і вказано значення для ACL, але ACL не містить ACE.

5.2. СТРУКТУРА SID

До цього часу ми говорили про SID як про непрозорі двійкові значення або рядки чисел. Детальніше розглянемо, що містить SID. На рис. 5.1 показано SID, який зберігається в пам’яті.

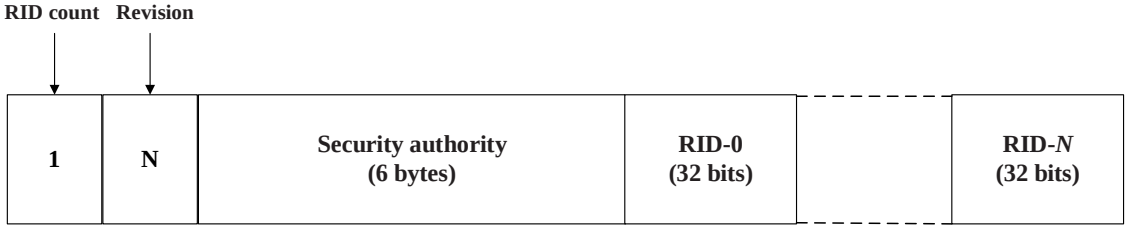


Рисунок 5.1. Структура SID в памяти

У двійковому SID є чотири компоненти:

Revision (Ревізія). Значення, яке завжди встановлено на 1, тому що на сьогодні немає іншої версії;

Relative identifier count (Кількість відносних ідентифікаторів), Кількість RID у SID;

Security authority (Служба безпеки), Значення, що представляє сторону, що видала SID;

Relative identifiers (Відносні ідентифікатори), Нуль або більше 32-бітних чисел, які представляють користувача чи групу.

У Windows є попередньо визначені деякі поширені значення. Усі відомі повноваження починаються з п’яти байтів 0, за якими йде значення з таб.5.2.

Таблиця 5.2.

Загальновідомі повноваження авторизації

Назва	Кінцеве значення	Приклад назви
Null	0	NULL SID
World	1	Everyone
Local	2	CONSOLE LOGON
Creator	3	CREATOR OWNER
Nt	5	BUILTIN\Users
Package	15	APPLICATION PACKAGE AUTHORTTY\Your Internet connection
MandatoryLabel	16	Mandatory Label\Medium Mandatory Level
ScopedPolicyId	17	N/A
ProcessTrust	19	TRUST LEVEL\ProtectedLight-Windows

Після цього йдуть відносні ідентифікатори. SID може містити один або декілька RID, де RID домену йдуть за RID користувача.

Розглянемо, як створюється SID для відомої групи BUILTIN\Users. Зверніть увагу, що компонент домену відокремлюється від назви групи зворотною скісну рискою. У цьому випадку домен — BUILTIN. Це попередньо визначений домен, представлений одним RID, 32. У лістингу 5.1 створюється SID домену для домену BUILTIN з його компонентів за допомогою команди PowerShell *Get-NtSid*, а потім використовується команда *Get-NtSidName* для отримання системного імені SID.

Лістинг 5.1. Запит SID домену BUILTIN

```
PS C:\WINDOWS\system32> $domain_sid = Get-NtSid -SecurityAuthority Nt -RelativeIdentifier 32
PS C:\WINDOWS\system32> Get-NtSidName $domain_sid

Domain  Name    Source  NameUse
------  ----    ------  -------
BUILTIN BUILTIN Account Domain
```

SID домену BUILTIN є членом системи безпеки Nt.

Вказуємо цю систему безпеки за допомогою параметра *SecurityAuthority* і вказуємо єдиний RID за допомогою параметра *RelativeIdentifier*.

Потім передаємо SID команді *Get-NtSidName*. У перших двох стовпцях вихідних даних показано ім'я домену та ім'я SID.

У даному випадку ці значення збігаються. Це просто особливість реєстрації домену BUILTIN.

Наступний стовпець позначає місце, звідки було отримано ім'я. У даному прикладі джерело *Account* вказує, що ім'я було отримано з LSASS. Якщо джерелом було *WellKnown*, це означало б, що PowerShell знав ім'я завчасно і не потребував запиту до LSASS.

Четвертий стовпець, *NameUse*, вказує тип SID. У даному випадку це *Domain*, що ми і очікували. Останній стовпець — це SID у форматі SDDL.

Будь-які RID, вказані для SID після доменного SID, ідентифікують конкретного користувача або групу. Для групи «Користувачі» ми використовуємо єдиний RID із значенням 545 (заздалегідь визначеним Windows). У лістингу 5.2 створюється новий SID шляхом додавання RID 545 до SID базового домену.

Лістинг 5.2. Створення SID на основі служби безпеки та RID

```
PS C:\WINDOWS\system32> $user_sid = Get-NtSid -BaseSid $domain_sid -RelativeIdentifier 545
PS C:\WINDOWS\system32> Get-NtSidName $user_sid

Domain  Name  Source  NameUse
------  ----  ------  -------
BUILTIN Users Account Alias


PS C:\WINDOWS\system32> $user_sid.Name
BUILTIN\Users
```

У вихідних даних тепер відображається *Users* як ім'я SID. Також зверніть увагу, що *NameUse* в цьому випадку встановлено на *Alias*. Це означає, що SID представляє локальну вбудовану групу, на відміну від *Group*, яка представляє групу, визначену користувачем.

Коли ми виводимо властивість *Name* на SID, вона виводить повне ім'я, де домен і ім'я розділені зворотним слешем.

Ви можете знайти списки відомих SID у технічній документації Microsoft та на інших веб-сайтах. Однак Microsoft іноді додає SID без їх документування. Тому я рекомендую вам перевірити кілька значень безпеки та RID, щоб побачити, яких інших користувачів та груп ви можете знайти.

Проста перевірка різних SID не завдає ніякої шкоди. Наприклад, спробуйте замінити RID користувача в лістингу 5.2 на 544. Цей новий SID представляє групу BUILTIN\Administrators, як показано в лістингу 5.3.

Лістинг 5.3. Запит SID групи адміністраторів

```
PS C:\WINDOWS\system32> Get-NtSid -BaseSid $domain_sid -RelativeIdentifier 544

Name                  Sid
----                  ---
BUILTIN\Administrators S-1-5-32-544
```

Запам'ятати службу безпеки та RID для конкретного SID може бути складно, і ви можете не згадати точну назву для запиту за допомогою параметра *Name*. Тому *Get-NtSid* реалізує режим, який може запитувати SID із відомого набору. Наприклад, щоб запитати SID групи Administrators, ви можете використати команду, показану в лістингу 5.4.

Лістинг 5.4. Запит відомого SID групи адміністраторів

```
PS C:\WINDOWS\system32> Get-NtSid -KnownSid BuiltinAdministrators

Name                  Sid
----                  ---
BUILTIN\Administrators S-1-5-32-544
```

SID використовуються в усій операційній системі Windows. Дуже важливо розуміти, як вони структуровані, оскільки це дозволить вам швидко оцінити, що може означати SID. Наприклад, якщо ви ідентифікуєте SID з службою безпеки Nt, а його перший RID дорівнює 32, ви можете бути впевнені, що він представляє вбудованого користувача або групу. Знання структури також дозволяє ідентифікувати та витягувати SID з дампів помилок або пам'яті в випадках, коли немає кращих інструментів.

5.3. АБСОЛЮТНІ ТА ВІДНОСНІ ДЕСКРИПТОРИ БЕЗПЕКИ

Ядро підтримує два формати бінарного представлення для опису безпеки: абсолютний і відносний. У цьому розділі ми розглянемо обидва формати та проаналізуємо переваги і недоліки кожного з них.

Обидва формати починаються з тих самих трьох значень: версія, прапорці диспетчера ресурсів і прапорці керування. Прапорець *SelfRelative* у прапорцях керування визначає, який формат використовувати, як показано на рис. 5.2.

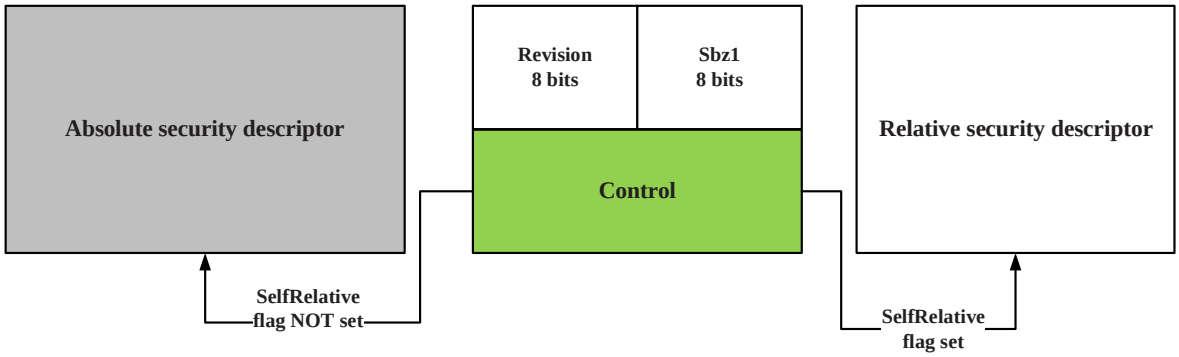


Рисунок 5.2. Вибір формату дескриптора безпеки

Загальний розмір заголовка дескриптора безпеки становить 32 біти, розділених двома 8-бітними значеннями, *ревізією* та *Sbz1*, і 16-бітними прапорцями контролю. Прапорці диспетчера ресурсів дескриптора безпеки зберігаються в *Sbz1*; вони дійсні лише в тому випадку, якщо встановлено прапорець контролю *RmControlValid*, хоча значення буде присутнє в обох випадках. Решта дескриптора безпеки зберігається відразу після заголовка.

Найпростіший формат - абсолютний дескриптор безпеки, використовується, коли прапор *SelfRelative* не встановлений. Після загального заголовка абсолютний формат визначає чотири покажчики для посилання в пам'яті:

— SID власника;
— SID групи;
— DACL;
— SACL.

Ця послідовність показана на рис. 5.3.

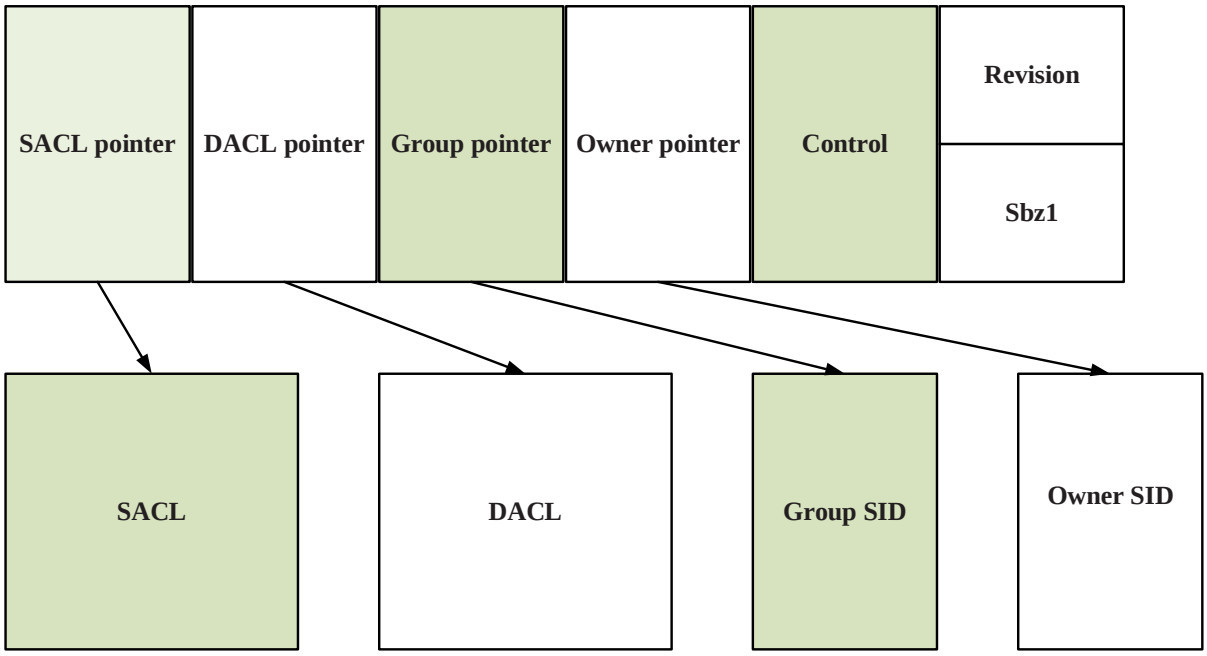


Рисунок 5.3. Структура абсолютного дескриптора безпеки

Кожен покажчик посилається на абсолютну адресу пам'яті, за якою зберігаються дані. Тому розмір покажчика залежить від того, чи є додаток 32- або 64-розрядним. Також можна вказати значення NULL для покажчика, щоб позначити, що значення відсутнє. Значення SID власника та групи зберігаються у двійковому форматі, визначеному в попередньому розділі.

Коли встановлено прапор *SelfRelative*, дескриптор безпеки замість *абсолютного* використовує *відносний* формат. Замість посилання на свої значення за допомогою абсолютних адрес пам'яті, відносний дескриптор безпеки зберігає ці місця як відносні зміщення відносно початку свого заголовка. На рис. 5.4 показано, як побудований відносний дескриптор безпеки.

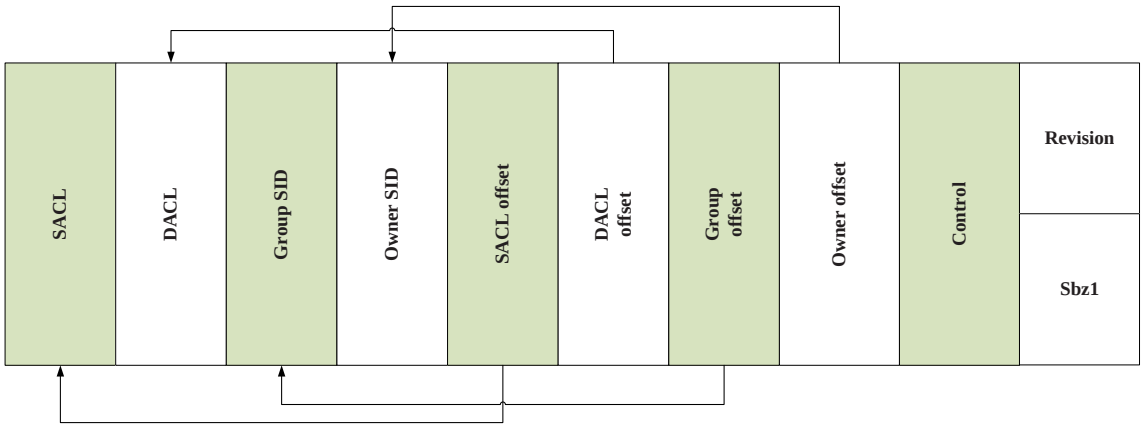


Рисунок 5.4. Структура відносного дескриптора безпеки

Ці значення зберігаються в суміжній пам'яті.

Формат ACL, який ми розглянемо в наступному розділі, вже є відносним форматом і тому не вимагає спеціального оброблення при використанні у відносному дескрипторі безпеки.

Кожне зміщення завжди має довжину 32 біти, незалежно від розряду системи. Якщо зміщення встановлено на 0, значення не існує, як у випадку NULL для абсолютного дескриптора безпеки.

Головною перевагою абсолютного дескриптора безпеки є те, що ви можете легко оновлювати його окремі компоненти. Наприклад, щоб замінити SID власника, ви повинні виділити новий SID в пам'яті та призначити його адресу пам'яті покажчику власника. Для порівняння, модифікація відносного дескриптора безпеки таким самим чином може вимагати коригування виділеної пам'яті, якщо нова структура SID власника більша за стару.

З іншого боку, великою перевагою відносного дескриптора безпеки є те, що його можна розмістити в одному суцільному блоці пам'яті. Це дозволяє перетворити дескриптор безпеки в постійний формат, такий як файл або ключ реєстру. Коли ви намагаєтеся визначити безпеку ресурсу, вам може знадобитися отримати його дескриптор безпеки з пам'яті або постійного сховища. Розуміючи обидва формати, ви можете визначити, як прочитати дескриптор безпеки в форматі, який можна переглянути або обробити.

Більшість API та системних викликів приймають будь-який формат дескриптора безпеки, автоматично визначаючи спосіб обробки шляхом перевірки значення прапора *SelfRelative*. Однак є деякі винятки, коли API приймає тільки один формат. У цьому випадку, якщо ви передаєте API дескриптор безпеки в неправильному форматі, зазвичай ви отримаєте помилку, наприклад STATUS_INVALID_SECURITY_DESCR. Дескриптори безпеки, що повертаються з API, майже завжди будуть у відносному форматі через простоту управління пам'яттю. Система надає API *RtlAbsoluteToSelfRelativeSD* і *RtlSelfRelativeToAbsoluteSD* для перетворення між двома форматами, якщо це необхідно.

Модуль PowerShell обробляє всі дескриптори безпеки за допомогою об'єкта *SecurityDescriptor*, незалежно від формату. Цей об'єкт написаний на мові .NET і перетворюється на відносний або абсолютний дескриптор безпеки тільки тоді, коли це необхідно для взаємодії з власним кодом. Ви можете визначити, чи був об'єкт *SecurityDescriptor* створений з відносного дескриптора безпеки, перевіривши властивість *SelfRelative*.

5.4. ЗАГОЛОВКИ ТА ЗАПИСИ У СПИСКУ КОНТРОЛЮ ДОСТУПУ

DACL і SACL складають більшу частину даних в дескрипторі безпеки. Хоча ці елементи мають різні призначення, вони мають однакову базову структуру. У цьому розділі ми розглянемо, як вони розташовані в пам'яті.

5.4.1. Заголовок

Всі ACL складаються з заголовка ACL, за яким слідує список з нуля або більше ACE в одному суцільному блоці пам'яті. На рис. 5.5 наведено формат верхнього рівня.

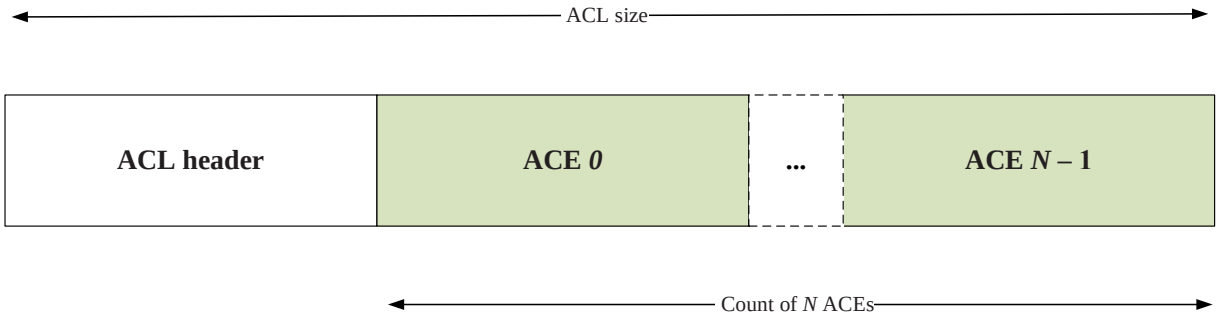


Рисунок 5.5. Загальний огляд структури ACL

Заголовок ACL містить ревізію, загальний розмір ACL у байтах та кількість записів ACE, що йдуть за заголовком. На рис. 5.6 показано структуру заголовка.

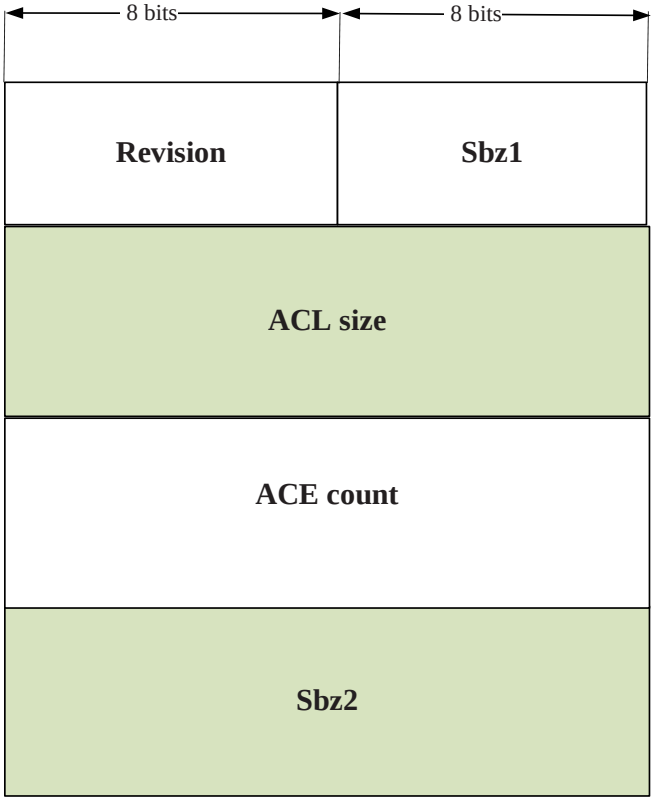


Рисунок 5.6. Структура заголовка ACL

Заголовок ACL також містить два зарезервовані поля, *Sbz1* і *Sbz2*, які завжди повинні мати значення 0. Вони не виконують жодної функції в сучасних версіях Windows і призначені для розширення структури ACL у разі потреби.

Наразі поле Revision може мати одне з трьох значень, які визначають дійсні ACE ACL. Якщо ACL використовує ACE, який не підтримується цією версією, ACL не буде вважатися дійсним.

Windows підтримує такі версії:

Default - Версія ACL за замовчуванням . Підтримує всі основні типи ACE, такі як «Дозволено» та «Заборонено». Позначається *Revision 2*.

Compound - Додає підтримку комбінованих ACE до стандартної версії ACL. Позначається *Revision 3*.

Object - Додає підтримку об'єктних ACE. Позначається *Revision 4*.

5.4.2. Список ACE

Після заголовка ACL розташований список ACE, який визначає, який доступ має SID. ACE мають змінну довжину, але завжди починаються з заголовка, що містить тип ACE, додаткові прапорці та загальний розмір ACE. За заголовком йдуть дані, специфічні для типу ACE. Рисунок 5.7 показує цю структуру.

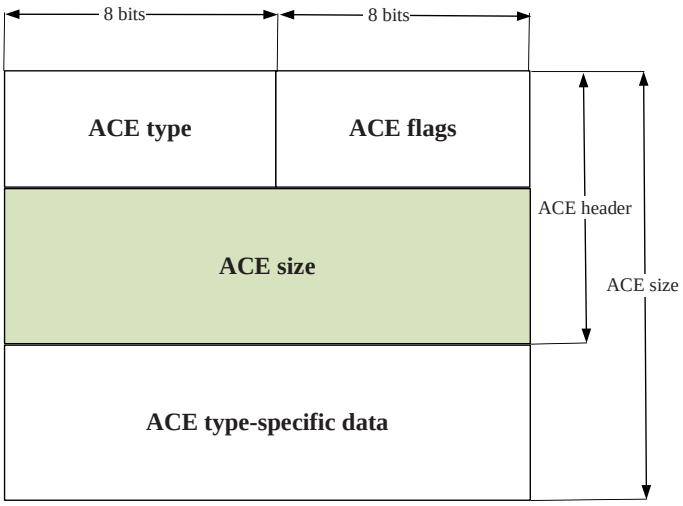


Рисунок 5.7. Структура ACE

Заголовок ACE є однаковим для всіх типів ACE. Це дозволяє додатку безпечно отримувати доступ до заголовка під час обробки ACL. Значення типу ACE можна потім використовувати для визначення точного формату даних, специфічних для типу ACE. Якщо додаток не розуміє тип ACE, він може використовувати поле *ACE size*, щоб повністю пропустити ACE.

У таб.5.3 наведено лістинг підтримуваних типів ACE, мінімальну версію ACE, в якій вони є дійсними, та чи є вони дійсними в DACL або SACL.

Таблиця 5.3.

Підтримувані типи ACE, мінімальні версії ACL та розташування

ACE тип	Значення	Мінімальна ревізія	ACL	Опис
Allowed	0x0	Default	DACL	Надає доступ до ресурсу

ACE тип	Значення	Мінімальна ревізія	ACL	Опис
Denied	0x1	Default	DACL	Забороняє доступ до ресурсу
Audit	0x2	Default	SACL	Перевіряє доступ до ресурсу
Alarm	0x3	Default	SACL	Сповіщення про доступ до ресурсу. Не використовується
AllowedCompound	0x4	Compound	DACL	Надає доступ до ресурсу під час імперсонації
AllowedObject	0x5	Object	DACL	Надає доступ до ресурсу певним типам об'єкта
DeniedObject	0x6	Object	DACL	Забороняє доступ до ресурсу певним типам об'єкта
Auditobject	0x7	Object	SACL	Аудит доступу до ресурсу певним типам об'єкта
AlarmObject	0x8	Object	SACL	Тривога при доступі певними типами об'єкта. Не використовується
AllowedCallback	0x9	Default	DACL	Надає доступ до ресурсу зі зворотним викликом
DeniedCallback	0xA	Default	DACL	Забороняє доступ до ресурсу зі зворотним викликом
AllowedCallbackObject	0xB	Object	DACL	Надає доступ зі зворотним викликом та типом об'єкта
DeniedCallbackObject	0xC	Object	DACL	Забороняє доступ зі зворотним викликом та типом об'єкта
Auditcallback	0xD	Default	SACL	Аудит доступу зі зворотним викликом
AlarmCallback	0xE	Default	SACL	Тривога при доступі зі зворотним викликом. Не використовується
AuditCallbackObject	0xF	Object	SACL	Аудит доступу зі зворотним викликом та типом об'єкта
AlarmCallbackObject	0x10	Object	SACL	Тривога при доступі зі зворотним викликом та типом об'єкта. Не використовується
MandatoryLabel	0x11	Default	SACL	Визначає обов'язкову мітку
ResourceAttribute	0x12	Default	SACL	Визначає атрибути для ресурсу

ACE тип	Значення	Мінімальна ревізія	ACL	Опис
ScopedPolicyId	0x13	Default	SACL	Визначає централізований ідентифікатор політики доступу для ресурсу
ProcessTrustLabel	0x14	Default	SACL	Визначає мітку довіри процесу для обмеження доступу до ресурсу
AccessFilter	0x15	Default	SACL	Визначає фільтр доступу для ресурсу

Хоча Windows офіційно підтримує всі ці типи ACE, ядро не використовує типи *Alarm*. Програми користувача можуть вказувати власні типи ACE, але різні API в режимі користувача та ядра перевіряють дійсність типів і генерують помилку, якщо тип ACE невідомий.

Дані, специфічні для типу ACE, в основному відповідають одному з трьох форматів: стандартні ACE, такі як *Allowed* і *Denied*; комбіновані ACE; та об'єктні ACE.

Стандартні ACE містять наступні поля після заголовка, розмір поля вказано в дужках:

Маска доступу (Access mask, 32 бита) - Маска доступу, яка надається або відхиляється на основі типу ACE;

SID (змінний розмір) - SID у двійковому форматі.

Комбіновані ACE використовуються під час імперсонації. Ці ACE можуть надавати доступ одночасно як імперсонованому виклику, так і процесу користувача. Єдиним прийнятним типом для них є *AllowedCompound*. Хоча останні версії Windows все ще підтримують комбіновані ACE, вони фактично не задокументовані і, ймовірно, застарілі. Ми додали їх до цього посібника для повноти інформації.

Комбіновані ACE мають наступний формат:

Маска доступу (Access mask, 32-біт) - Маска доступу, який буде надано;

Комбінований тип (ACE Compound ACE type, 16-біт) - Якщо встановити значення 1, то це означатиме, що ACE використовується для імперсонації;

Зарезервовано (Reserved, 16-біт) - Завжди 0;

SID сервера (Server SID, змінний розмір) - SID сервера в двійковому форматі, відповідає службі;

SID користувача (SID, змінний розмір) SID в двійковому форматі; відповідає імперсонованому користувачеві.

Компанія Microsoft запровадила формат *об'єкта* ACE для підтримки контролю доступу до служб домену Active Directory. Active Directory використовує 128-бітний GUID для представлення типу об'єкта служби каталогів. *Об'єкт* ACE визначає доступ для певних типів об'єктів, таких як комп'ютери або користувачі. Наприклад, використовуючи єдиний дескриптор

безпеки, каталог може надати SID доступ, необхідний для створення одного типу об'єкта, але не іншого.

Формат об'єкта ACE є таким:

Маска доступу (32 біти) - Маска доступу, яка надається або відхиляється на основі типу ACE;

Прапорці (32 біти) - Використовуються для зазначення, які з наведених нижче GUID знаходяться в наявності;

Тип об'єкта (16 байт) - GUID ObjectType. Використовується тільки якщо встановлено прапорець 0;

Тип успадкованого об'єкта (16 байт) - GUID успадкованого об'єкта; Використовується тільки якщо встановлено прапорець 1;

SID (змінний розмір)- SID у двійковому форматі.

ACE можуть бути більшими за визначені структури своїх типів і можуть використовувати додатковий простір для зберігання неструктурованих даних. Найчастіше вони використовують ці неструктуровані дані для типів ACE з зворотним викликом, таких як *AllowedCallback*, який задає умовний вираз, що визначає, чи повинен ACE бути активним під час перевірки доступу. Ми можемо перевірити дані, які будуть згенеровані з умовного виразу, використовуючи команду PowerShell *ConvertFrom-NtAceCondition*, як показано в лістингу 5.5.

Лістинг 5.5. Аналіз умовного виразу та відображення бінарних даних

```
PS C:\WINDOWS\system32> 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...
```

Ці ACE позначаємо як ACE *зворотного виклику*, оскільки до Windows 8 для їх обробки програмам потрібно було викликати API *AuthzAccessCheck*. API приймав функцію зворотного виклику, яка використовувалася для визначення, чи слід включати ACE зворотного виклику в перевірку доступу. Починаючи з Windows 8, перевірка доступу ядра має вбудовану підтримку умовних ACE у форматі, показаному в лістингу 5.5, хоча користувацькі програми можуть вільно визначати свої власні формати та обробляти ці ACE вручну.

Таблиця 5.4.
Прапорці ACE

ACE прапор	Значення	Опис
Objectlnherit	0x1	ACE може бути успадкований об'єктом.
Containerlnherit	0x2	ACE може бути успадкований контейнером.
NoPropagatelnherit	0x4	Прапорці успадкування ACE не поширюються на дочірні об'єкти.
InheritOnly	0x8	ACE використовується лише для успадкування, а не для перевірки доступу.
Inherited	0x10	ACE було успадковане від батьківського контейнера.
Critical	0x20	ACE є критично важливим і не може бути видалений. Стосується лише дозволених ACE.

SuccessfulAccess	0x40	Подія аудиту має бути згенерована для успішного доступу.
FailedAccess	0x80	Подія аудиту має бути згенерована для невдалого доступу.
TrustProtected	0x40	При використанні з ACE AccessFilter цей прапорець запобігає модифікації.

Основне використання прапорців ACE полягає у визначенні правил успадкування для ACE. У таб.5.4 показано визначені прапорці ACE.

Прапорці успадкування займають тільки молодші 5 бітів, залишаючи старші 3 біти для прапорців, специфічних для ACE.

5.5. СТВОРЕННЯ ТА РЕДАГУВАННЯ ДЕСКРИПТОРІВ БЕЗПЕКИ

Оскільки ви вже ознайомилися зі структурою дескриптора безпеки, давайте розглянемо, як створювати та редагувати їх за допомогою PowerShell. Найпоширенішою причиною для цього є перегляд вмісту дескриптора безпеки, щоб зрозуміти, який доступ застосовується до ресурсу. Іншим важливим випадком використання є необхідність створення дескриптора безпеки для блокування ресурсу. Модуль PowerShell, який використовується в цьому посібнику, має на меті максимально спростити створення та перегляд дескрипторів безпеки..

5.5.1. Створення нового дескриптора безпеки

Щоб створити новий дескриптор безпеки, можна скористатися командою *New-NtSecurityDescriptor*. За замовчуванням вона створює новий об'єкт *SecurityDescriptor* без власника, групи, DACL або SACL. Ви можете скористатися параметрами команди, щоб додати частини дескриптора безпеки, як показано в лістингу 5.6.

Лістинг 5.6. Створення нового дескриптора безпеки з вказаним власником

```
PS C:\WINDOWS\system32> $world = Get-NtSid -KnownSid World
PS C:\WINDOWS\system32> $sd = New-NtSecurityDescriptor -Owner $world -Group $world -Type File
PS C:\WINDOWS\system32> $sd | Format-Table

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

Спочатку ми отримуємо SID для групи *World*. При виклику *New-NtSecurityDescriptor* для створення нового дескриптора безпеки використовуємо цей SID для визначення його власника та групи. Ми також визначаємо ім'я типу об'єкта ядра, з яким буде пов'язаний цей дескриптор безпеки. Цей крок спрощує використання деяких наступних команд. У цьому випадку ми припустимо, що це дескриптор безпеки об'єкта *File*.

Потім відображаємо дескриптор безпеки, форматуючи вихідні дані у вигляді таблиці. Як бачите, поле *Owner* встановлено на *Everyone*. Значення *Group* за замовчуванням не відображається, оскільки воно не є настільки важливим. Наразі в дескрипторі безпеки немає ні DACL, ні SACL, і рівень цілісності не вказано.

Щоб додати деякі ACE, ми можемо використовувати команду *Add-NtSecurityDescriptorAce*. Для звичайних ACE нам потрібно вказати тип ACE, SID і маску доступу. За бажанням ми також можемо вказати прапорці ACE. Скрипт у лістингу 5.7 додає деякі ACE до нашого нового дескриптора безпеки.

Спочатку отримуємо SID поточного користувача за допомогою Get-NtSid. Використовуємо цей SID, щоб додати новий ACE «Дозволено» до DACL. Також додаємо ACE «Відмовлено» для анонімного користувача, вказавши параметр Type, а потім ще один ACE «Дозволено» для групи «Усі». Потім модифікуємо SACL, щоб додати ACE «Аудит» і встановити обов'язкову позначку на низький рівень цілісності.

Лістинг 5.7. Додавання ACE до нового дескриптора безпеки

```
PS> $user = Get-NtSid
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 PS> Add-NtSecurityDescriptorAce $sd -
KnownSid World -Access Delete -Type Audit -Flags FailedAccess
PS> Set-NtSecurityDescriptorIntegrityLevel $sd Low
PS> Set-NtSecurityDescriptorControl $sd DaclAutoInherited, SaclProtected 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-NtSecurityDescriptorDacl $sd | Format-Table
Type                User                                  Flags       Mask
----                ----                                  -----       ----
Audit               Everyone                              FailedAccess 00010000
MandatoryLabel Mandatory Label\Low Mandatory Level None       00000001
```

Щоб завершити створення дескриптора безпеки, ми встановлюємо прапорці контролю DaclAutoInherited і SaclProtected.

Тепер ми можемо вивести детальну інформацію про дескриптор безпеки, який щойно створили. Відображення дескриптора безпеки показує, що DACL тепер містить три ACE і два SACL, а рівень цілісності є низьким. Ми також відображаємо прапорці контролю і списки ACE в DACL і SACL.

5.5.2. Впорядковування ACE

З огляду на принципи роботи перевірки доступу, існує канонічний порядок розміщення ACE в ACL. Наприклад, всі ACE з дозволом «Відмовлено» повинні розміщуватися перед ACE з дозволом «Дозволено», оскільки в іншому випадку система може надати доступ до ресурсу некоректно, виходячи з того, які ACE розміщені першими. SRM не забезпечує дотримання цього канонічного порядку. SRM покладається на те, що будь-яка програма правильно впорядкувала ACE перед передачею їх для перевірки доступу. ACL повинні впорядковувати свої ACE відповідно до таких правил:

1. Всі ACE типу «Заборонено» повинні знаходитися перед типами «Дозволено».
2. ACE типу «Дозволено» повинні знаходитися перед ACE типу «Дозволено» для об'єктів.
3. ACE типу «Заборонено» повинні знаходитися перед ACE типу «Заборонено» для об'єктів.
4. Всі ACE, які не успадковуються, повинні знаходитися перед ACE з встановленим прапорцем «Успадковано».

У лістингу 5.7 ми додали ACE «Відмовлено» до DACL після додавання ACE «Дозволено», порушивши правило першочерговості. Ми можемо переконатися, що DACL канонізовано, використовуючи команду *Edit-NtSecurity* з параметром *CanonicalizeDacl*. Ми також можемо перевірити, чи DACL вже канонізовано, використовуючи команду PowerShell *Test-NtSecurityDescriptor* з параметром *DaclCanonical*. Лістинг 5.8 ілюструє використання обох команд..

Лістинг 5.8. Канонізація DACL

```
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
```

Якщо порівняти список ACE в лістингу 5.8 зі списком в лістингу 5.7, можна помітити, що ACE з дозволом «Відмовлено» було переміщено з середини на початок ACL. Це гарантує, що воно буде оброблено раніше, ніж будь-які ACE з дозволом «Дозволено».

5.5.3. Форматування дескрипторів безпеки

Ви можете вручну вивести значення в дескрипторі безпеки за допомогою команди *Format-Table*, але це займає багато часу. Інша проблема ручного форматування полягає в тому, що маски доступу не будуть декодовані, тому замість *ReadData*, наприклад, ви побачите 00000001. Було б добре мати простий

спосіб виведення деталей дескриптора безпеки та їх форматування на основі типу об'єкта. Саме для цього призначена команда *Format-NtSecurityDescriptor*. Ви можете передати їй дескриптор безпеки, і команда виведе його на консоль. Приклад наведено в лістингу 5.9.

Ми передаємо параметр *ShowAll* до *Format-NtSecurityDescriptor*, щоб забезпечити відображення всього вмісту дескриптора безпеки. За замовчуванням він не виводить SACL або менш поширені ACE, такі як *ResourceAttribute*.

Зверніть увагу, що тип об'єкта ядра виводу відповідає типу файлу, який ми вказали при створенні дескриптора безпеки в лістингу 5.6. Вказання типу об'єкта ядра дозволяє відобразити декодовану маску доступу для типу, а не загальне hex-значення.

Наступний рядок у вихідних даних показує поточні прапорці керування. Вони обчислюються на основі поточного стану дескриптора безпеки. Пізніше ми обговоримо, як змінити ці прапорці керування, щоб змінити поведінку дескриптора безпеки. За прапорцями керування стоять SID власника та групи, а також DACL, які складають більшу частину вихідних даних.

Лістинг 5.9. Відображення дескриптора безпеки

```
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
```

Будь-які прапорці, специфічні для DACL, з'являються поруч із заголовком. У цьому випадку вони вказують, що ми встановили прапорець *DaclAutoInherited*. Далі у вихідних даних перелічуються всі ACE в ACL у порядку, починаючи з

типу ACE. Оскільки команда знає тип об'єкта, вона виводить декодовану маску доступу для цього типу, а також оригінальну маску доступу в шістнадцятковому форматі.

Далі йде SACL, який показує наш ACE єдиного аудиту, а також прапор *SaclProtected*. Останній компонент, що відображається, – це обов'язкова мітка. Маска доступу для обов'язкової мітки – це обов'язкова політика.

Обов'язкова політика може бути встановлена на один або кілька бітових прапорів, показаних у таб.5.5.

Таблиця 5.5.
Значення обов'язкових політик

Назва	Значення	Опис
NoWriteUp	0x00000001	Користувач з нижчим рівнем цілісності не може записувати дані в цей ресурс.
NoReadllp	0x00000002	Користувач з нижчим рівнем цілісності не може читати цей ресурс.
NoExecuteUp	0x00000004	Користувач з нижчим рівнем цілісності не може виконати цей ресурс

За замовчуванням, *Format-NtSecurityDescriptor* може бути надто докладним.

Лістинг 5.10. Відображення дескриптора безпеки в стислому форматі

```
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)
```

Щоб скоротити його вихідні дані, вкажіть параметр *Summary*, який видалить якомога більше даних, зберігаючи при цьому важливу інформацію. Лістинг 5.10 показує це на прикладі.

Для зручності використання PowerShell, вживає прості імена для найбільш поширених прапорців, але ви можете відобразити повні імена SDK, якщо бажаєте (наприклад, для порівняння виводу з базовим кодом).

Лістинг 5.11. Форматування дескриптора безпеки з іменами SDK

```
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
```

Щоб відобразити імена SDK під час перегляду вмісту дескриптора безпеки за допомогою *Format-NtSecurityDescriptor*, використовуйте властивість *SDKName*, як показано в лістингу 5.11.

Однією з особливостей об'єктів *File* є те, що їхні маски доступу мають дві конвенції іменування: одну для файлів і одну для каталогів. Ви можете вимагати, щоб *Format-NtSecurityDescriptor* виводив версію маски доступу для каталогів, використовуючи параметр *Container* або, більш загально, встановивши властивість *Container* об'єкта опису безпеки на True. Лістинг 5.12 показує вплив налаштування параметра *Container* на вихідні дані.

Лістинг 5.12. Форматування дескриптора безпеки як контейнера

```
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--
```

Зверніть увагу, як рядок вихідних даних змінюється з ReadData|WriteData на List Directory|AddFile, коли ми форматуємо його як контейнер. Тип *File* є єдиним типом об'єкта з такою поведінкою в Windows. Це важливо для безпеки, оскільки ви можете легко неправильно інтерпретувати права доступу до файлів, якщо ви форматуєте дескриптор безпеки для каталогу як файл або навпаки.

Якщо вам більше подобається графічний інтерфейс, ви можете його запустити за допомогою команди *Show-NtSecurityDescriptor*:

```
PS> Show-NtSecurityDescriptor $sd
```

Виконання команди повинно відкрити діалогове вікно, показане на рис. 5.8.

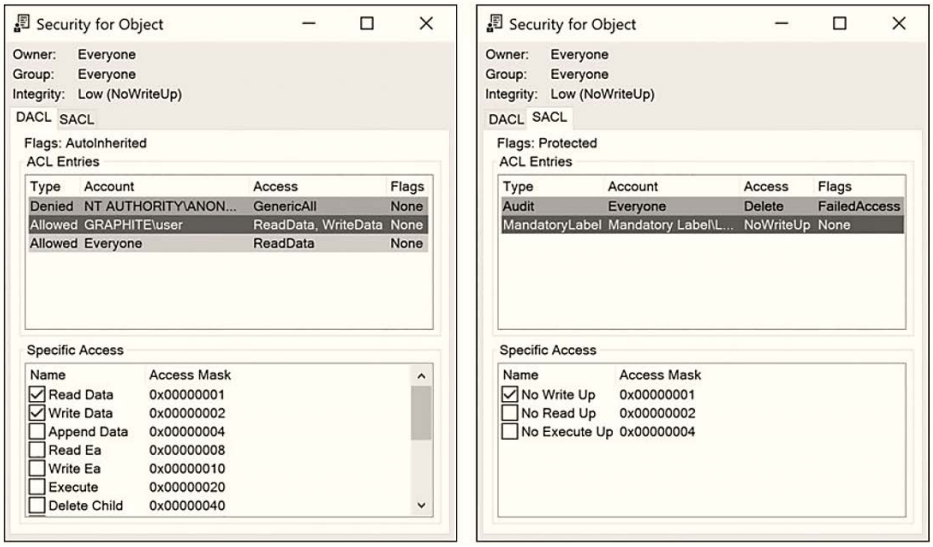


Рисунок 5.8. Графічний інтерфейс, що відображає дескриптор безпеки

У діалоговому вікні підсумовуються важливі дані дескриптора безпеки. У верхній частині знаходяться ідентифікатори власника та групи, перетворені в імена, а також рівень цілісності дескриптора безпеки та обов'язкова політика. Вони відповідають значенням, які ми вказали під час створення дескриптора безпеки. У середині знаходиться список ACE в DACL (зліва) або SACL (праворуч), залежно від того, яку вкладку ви вибрали, з прапорцями ACL у верхній частині. Кожен запис у списку включає тип ACE, SID, маску доступу в загальній формі та прапорці ACE.

У нижній частині знаходиться декодований доступ. Список заповнюється, коли ви вибираєте ACE у списку ACL.

5.5.4. Перетворення дескрипторів безпеки

За допомогою команди *ConvertFrom-NtSecurityDescriptor* можете перетворити об'єкт дескриптора безпеки в масив байтів у відповідному форматі. Потім можливо вивести його вміст, щоб побачити, якою насправді є базова структура, як показано в лістингу 5.13.

Можемо перетворити масив байтів в об'єкт дескриптора безпеки за допомогою New-NtSecurityDescriptor і параметра Byte:

```
PS> New-NtSecurityDescriptor -Byte $ba
```

Як вправу, залишимо вам завдання розібрати вихідні дані у шістнадцятковому форматі, щоб знайти різні структури дескриптора безпеки на основі описів, наведених у цьому розділі.

Лістинг 5.13. Перетворення абсолютного дескриптора безпеки у відносний формат та відображення його байтів

```
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       - ................
```

Для початку на рис. 5.9 наведено основні структури.

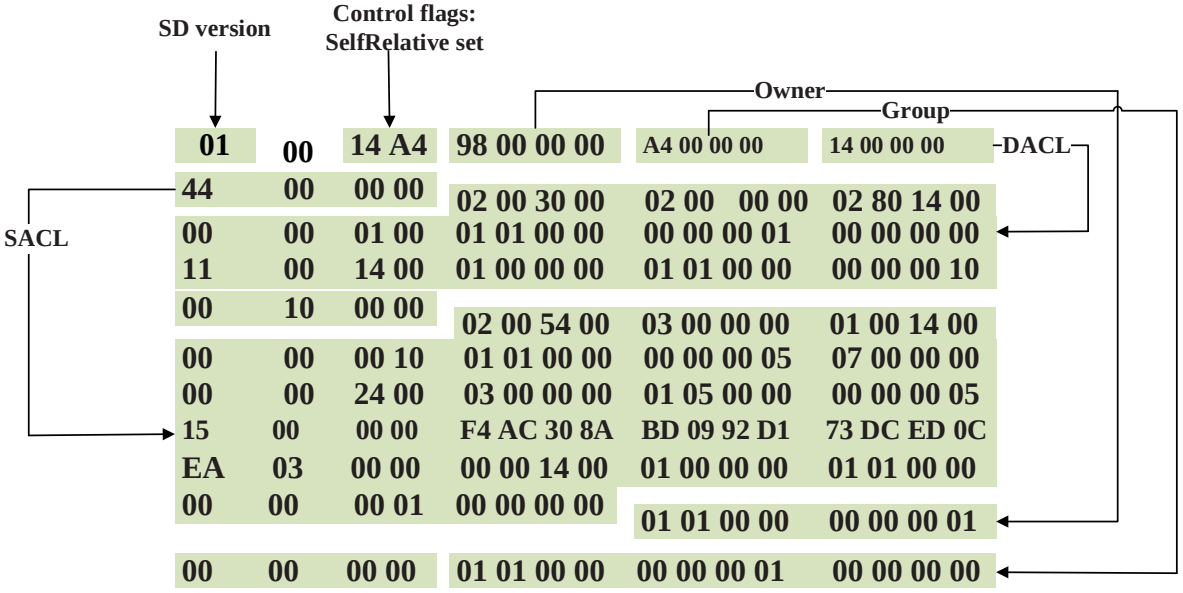


Рисунок 5.9. Схема основних структур відносного дескриптора безпеки у форматі hex

Вам потрібно буде звернутися до схеми структур ACL і SID, щоб вручну розшифрувати іншу частину.

5.6. МОВА ВИЗНАЧЕННЯ ДЕСКРИПТОРІВ БЕЗПЕКИ

В попередніх розділах було розглянуто основи формату мови визначення дескрипторів безпеки (SDDL) для представлення SID. Формат SDDL також здатний представляти повний дескриптор безпеки. Оскільки версія SDDL дескриптора безпеки використовує текст у форматі ASCII, вона є більш читабельною для людини і, на відміну від бінарних даних, показаних у лістингу

5.13, її можна легко скопіювати. Оскільки рядки SDDL часто використовуються в Windows, давайте розглянемо, як представити дескриптор безпеки в SDDL і як його прочитати.

Ви можете перетворити дескриптор безпеки у формат SDDL, вказавши параметр *ToSddl* для *Format-NtSecurityDescriptor*. Це продемонстровано в лістингу 5.14, де ми передаємо дескриптор безпеки, який ми створили в попередньому розділі. Ви також можете створити дескриптор безпеки з рядка SDDL, використовуючи *New-NtSecurityDescriptor* з параметром *ToSddl*.

Лістинг 5.14. Перетворення дескриптора безпеки в SDDL

```
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)
```

Версія SDDL дескриптора безпеки містить чотири необов'язкові компоненти. Початок кожного компонента можна визначити за такими префіксами:

O: SID власника
G: SID групи
D: DACL
S: SACL

У лістингу 5.15 ми розділили вихідні дані з лістингу 5.14 на компоненти, щоб полегшити їх читання.

Лістинг 5.15. Поділ компонентів SDDL

```
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)
```

Перші два рядки представляють SID власника та групи у форматі SDDL. Ви можете помітити, що вони не схожі на SID SDDL, до яких ми звикли, оскільки не починаються з S-1-. Це тому, що ці рядки є двосимвольними псевдонімами, які Windows використовує для відомих SID, щоб зменшити розмір рядка SDDL. Наприклад, рядок власника - це WD, який ми можемо перетворити назад у повний SID за допомогою *Get-NtSid* (лістинг 5.16).

Лістинг 5.16. Перетворення псевдоніма в ім'я та SID

```
PS C:\WINDOWS\system32> Get-NtSid -Sddl "WD"

Name     Sid
----     ---
Everyone S-1-1-0
```

Як бачите, псевдонім WD представляє групу Everyone. У таб.5.6 наведено псевдоніми для декількох відомих SID. Більш повний перелік усіх підтримуваних псевдонімів SDDL наведено в додатку.

Таблиця 5.6.
Відомі SID та їх псевдоніми

SID псевдонім	Ім'я	SDDL SID
AU	NT AUTHORITY\Authenticated Users	S-l-5-11
BA	BUILTIN\Administrators	S-l-5-32-544
IU	NT AUTHORITY\INTERACTIVE	S-l-5-4
SY	NT AUTHORITY\SYSTEM	S-l-5-18
WD	Everyone	S-l-1-0

Якщо SID не має псевдоніма, *Format-NtSecurityDescriptor* виведе SID у форматі SDDL, як показано в лістингу 5.5. Навіть SID без псевдонімів можуть мати імена, визначені LSASS. Наприклад, SID у лістингу 5.15 належить поточному користувачеві, як показано в лістингу 5.17.

Лістинг 5.17. Пошук імені SID

```
PS> Get-NtSid -Sddl "S-1-5-21-2318445812-3516008893-216915059-1002" -ToName
GRAPHITE\user
```

Далі в лістингу 5.15 наведено представлення DACL. Після префікса D: ACL у форматі SDDL виглядає наступним чином:

```
ACLFlags(ACE0)(ACE1)...(ACEn)
```

Прапорці ACL є необов'язковими. DACL встановлені на AI, а SACL — на P. Ці значення відповідають прапорцям контролю дескриптора безпеки і можуть бути одним або декількома рядками з таб.5.7.

Таблиця 5.7.
Строки прапорців ACL, що відповідають прапорцям керування дескрипторами безпеки

ACL прапорець	DACL контрольний прапор	SACL контрольний прапор
P	DaclProtected	SaclProtected
AI	DaclAutoInherited	SaclAutoInherited
AR	DaclAutoInheritReq	SaclAutoInheritReq

Кожен ACE знаходиться в дужках і складається з декількох рядків, розділених крапками з комою, відповідно до такого загального формату:

```
(Type;Flags;Access;ObjectType;InheritedObjectType;SID[;ExtraData])
```

Type — це короткий рядок, який відповідає типу ACE. У таб.5.8 наведено перелік відповідностей. Зверніть увагу, що формат SDDL не підтримує певні типи ACE, тому вони не вказані в таблиці.

Наступним компонентом є Flags, який представляє прапорці ACE. Запис аудиту в SACL з лістингу 5.15 показує прапор FA, який представляє FailedAccess. Таблиця 5.9 містить інші відповідності.

Таблиця 5.8.

Відповідності типів рядків типам ACE

ACE тип string	ACE тип
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	ScopedPolicyld
TL	ProcessTrustLabel
FL	AccessFilter

Таблиця 5.9.

Відповідності рядків прапорців прапорцям ACE

ACE рядок прапора	ACE прапор
01	Objectlnherit
CI	Containerlnherit
NP	NoPropagatelnherit
10	InheritOnly
ID	Inherited
CR	Critical(continued)
SA	SuccessfulAccess
FA	FailedAccess
TP	TrustProtected

Далі йде Access, який представляє маску доступу в ACE. Це може бути число в шістнадцятковому (0x1234), вісімковому (011064) або десятковому (4660) форматі, або список коротких рядків доступу. Якщо рядок не вказано, то використовується порожня маска доступу. Таб. 5.10 демонструє рядки доступу.

Зверніть увагу, що доступні рядки доступу не охоплюють весь діапазон масок доступу. Це пов'язано з тим, що SDDL був розроблений для представлення масок для об'єктів служб каталогів, які не визначають значення масок доступу за межами обмеженого діапазону.

Це також є причиною того, що назви прав є дещо заплутаними. Наприклад, *Delete Child* не обов'язково відповідає ідеї видалення дочірнього об'єкта довільного типу, і ви можете побачити в лістингу 5.15, що доступ, специфічний для типу File, відповідає доступу до об'єкта служби каталогів, навіть якщо він не має нічого спільного з Active Directory.

Таблиця 5.10.

Відповідність рядків доступу маскам доступу

Рядок доступу	Ім'я доступу	Маска доступу
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

Для кращої підтримки інших типів формат SDDL надає рядки доступу для загальних масок доступу до файлів і ключів реєстру, як показано в таб.5.11.

Якщо доступні рядки доступу не можуть представити всю маску, єдиним варіантом є представлення її у вигляді числового рядка, зазвичай у шістнадцятковому форматі.

Для компонентів ObjectType та InheritedObjectType, що використовуються з об'єктами ACE, SDDL записуються у форматі рядка для GUID. GUID можуть мати будь-яке значення. Наприклад, таб. 5.12 містить кілька відомих GUID, що використовуються Active Directory.

Таблиця 5.11.

Рядки доступу для типів файлів та ключів реєстру

Рядок доступу	Ім'я доступу	Маска доступу
FA	File Ail 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

Таблиця 5.12.

Відомі GUID ObjectType, що використовуються в Active Directory

GUID	Об'єкт каталогу
19195a5a-6daO-lldO-a-fd3-OOcO4fd93Oc9	Domain
bf967a86-Ode6-lldO-a285-OOaaOO3O49e2	Computer
bf967aba-Ode6-lldO-a285-OOaaOO3O49e2	User
bf967a9c-Ode6-lldO-a285-OOaaOO3O49e2	Group

Ось приклад ACE-рядка для *AllowedObject* ACE з встановленим ObjectType:

```
(OA;;CC;2f097591-a34f-4975-990f-00f0906b07e0;;WD)
```

Після компонента *InheritedObjectType* в ACE знаходиться SID. Як уже зазначалося раніше в цьому розділі, це може бути короткий псевдонім, якщо це загальновідомий SID, або повний формат SDDL, якщо це не так.

В останньому компоненті, який є необов'язковим для більшості типів ACE, ви можете визначити умовний вираз, якщо використовуєте ACE зворотного виклику, або атрибут безпеки, якщо використовуєте ACE *ResourceAttribute*. Умовний вираз визначається як булевий вираз, який зіставляє значення атрибуту безпеки токена.

При оцінці результат виразу повинен бути істинним або хибним. Ми бачили простий приклад у лістингу 5.5: WIN://TokenId == «XYZ», який порівнює значення атрибуту безпеки WIN://TokenId із рядковим значенням XYZ і оцінюється як істинне, якщо вони збігаються. Синтаксис виразу SDDL має чотири різні формати імен атрибутів для атрибуту безпеки, на який ви бажаєте посилатися:

Simple - Для локальних атрибутів безпеки, наприклад, WIN://TokenId;
Device - Для запитів пристроїв, наприклад, @Device.ABC;
User - Для запитів користувачів, наприклад, @User.XYZ;
Resource - Для атрибутів ресурсу, наприклад, @Resource.QRS.

Значення порівняння в умовних виразах також можуть приймати кілька різних типів. При перетворенні з SDDL в дескриптор безпеки умовний вираз буде проаналізовано, але оскільки тип атрибута безпеки на цей момент буде невідомий, перевірка типу значення не відбудеться. У таб.5.13 наведено приклади для кожного типу умовного виразу.

Таблиця 5.13.

Приклади значень для різних типів умовних виразів

Тип	Приклад
Number	Decimal: 100, -100; octal: 0100; hexadecimal: 0x100
String	"ThisIsAString"
Fully qualified binary name	{"0=MICR0S0FT CORPORATION, L=REDMOND, S=WASHINGTON",1004}
SID	SID(BA), SID(S-l-O-O)
Octet string	#0011223344

Потім синтаксис визначає оператори для обчислення виразу, починаючи з унарних операторів показаних у таб.5.14.

У таб.5.14 ATTR — це ім'я атрибута, що перевіряється, SIDLIST — це список значень SID, укладених у фігурні дужки {}, а EXPR — це інший умовний підвираз. У таб.5.15 наведено вставні оператори, які визначає синтаксис.

```
(ZA;;GA;;;WD;(WIN://TokenId == "XYZ"))
```

Останній компонент представляє атрибут безпеки для ACE *ResourceAttribute*. Його загальний формат такий:

```
"AttrName",AttrType,AttrFlags,AttrValue(,AttrValue...)
```

Таблиця 5.14.

Унарні оператори для умовних виразів

Оператор	Опис
Exists ATTR	Перевіряє, чи існує атрибут безпеки ATTR
Not_Exists ATTR	Інверсія Exists
Member_of {SIDLIST}	Перевіряє, чи групи токенів містять усі SID у SIDLIST
Not_Member_of {SIDLIST}	Інверсія до Member_of
Device_Member_of {SIDLIST}	Перевіряє, чи групи токенів містять усі SID у SIDLIST
Not_Device_Member_of {SIDLIST}	Інверсія до Device_Member_of
Member_of_Any {SIDLIST}	Перевіряє, чи групи токенів містять будь-які SID у SIDLIST
Not_Member_of_Any {SIDLIST}	Інверсія до Not_Member_of_Any
Device_Member_of_Any {SIDLIST}	Перевіряє, чи групи токенів містять будь-які SID у SIDLIST
Not_Device_Member_of_Any {SIDLIST}	Інверсія до Device_Member_of_Any
! (EXPR)	Логічне НЕ виразу

У таб.5.15 VALUE може бути або окремим значенням з таб.5.13, або списком значень, укладених у фігурні дужки. Оператори Any_of і Not_Any_of працюють тільки з списками, і умовний вираз завжди повинен бути розміщений в дужках в SDDL ACE. Наприклад, якщо ви хочете використовувати умовний вираз, показаний в лістингу 5.5, з AccessCallback ACE, рядок ACE буде наступним:

Значення AttrName — це ім'я атрибута безпеки, AttrFlags — шестнадцяткове число, що позначає прапорці атрибута безпеки, а AttrValue — одне або кілька значень, характерних для AttrType, розділених комами.

Таблиця 5.15.

Вставні оператори для умовних виразів

Оператор	Опис
ATTR Contains VALUE	Перевіряє, чи містить атрибут безпеки будь-яке значення
ATTR Not_Contains VALUE	Інверсія Contains
ATTR Any_oi {VALUELIST}	Перевіряє, чи містить атрибут безпеки будь-яке з наведених значень
ATTR Not_Any_of {VALUELIST}	Інверсія до Any_of
ATTR == VALUE	Перевіряє, чи атрибут безпеки збігається із зазначеним значенням
ATTR! = VALUE	Перевіряє, чи атрибут безпеки НЕ збігається із зазначеним значенням
ATTR < VALUE	Перевіряє, чи атрибут безпеки менший за значення
ATTR <= VALUE	Перевіряє, чи атрибут безпеки менший або дорівнює значенню
ATTR > VALUE	Перевіряє, чи атрибут безпеки більший за значення
ATTR >= VALUE	Перевіряє, чи атрибут безпеки більший або дорівнює значенню
EXPR && EXPR	Логічне І між двома виразами
EXPR | | EXPR	Логічне АБО між двома виразами

AttrType — це короткий рядок, що позначає тип даних, які містяться в атрибуті безпеки. У таб.5.16 наведено приклади таких рядків.

Таблиця 5.16.

Строки типу SDDL атрибутів безпеки

Тип атрибуту	Назва типу	Приклад значення
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

Наприклад, наступний рядок SDDL представляє атрибут ресурсу ACE з назвою *Classification*. Він містить два рядкові значення, *TopSecret* і *MostSecret*, і має встановлені прапорці *CaseSensitive* і *NonInheritable*:

```
S:(RA;;;;;WD;("Classification",TS,0x3,"TopSecret","MostSecret"))
```

Останнє поле в лістингу 5.15, яке потрібно визначити, - це SACL. Структура така ж, як і для DACL, хоча типи підтримуваних ACE відрізняються. Якщо ви спробуєте використати тип, який не дозволений у конкретному ACL, розбір рядка завершиться невдачею. У прикладі SACL у лістингу 5.15 єдиним ACE. Це обов'язкова мітка. Обов'язкова мітка ACE має власні рядки доступу, які використовуються для представлення обов'язкової політики, як показано в таб.5.17.

Таблиця 5.17.

Обов'язкові рядки доступу до міток

Рядок доступу	Ім'я доступу	Маска доступу
NX	No Execute Up	0X00000004
NR	No Read Up	0X00000002
NW	No Write Up	0X00000001

SID представляє рівень цілісності обов'язкового мітки. Слід зазначити, що для кожного рівня цілісності мітки існують спеціальні псевдоніми SID. Все, що не входить до списку, наведеного в таб.5.18, повинно бути представлено у вигляді повного SID.

Таблиця 5.18.

SID для рівнів цілісності обов'язкових міток

SID псевдонім	Ім'я	SDDL SID
LW	Низький рівень цілісності	S-l-16-4096
ME	Середній рівень цілісності	S-l-16-8192
MP	Рівень цілісності MediumPlus	S-l-16-8448
HI	Високий рівень цілісності	S-l-16-12288
SI	Рівень цілісності системи	S-l-16-16384

Формат SDDL не зберігає всю інформацію, яку ви можете зберігати в дескрипторі безпеки. Наприклад, формат SDDL не може відображати контрольний прапор *OwnerDefaulted* або *GroupDefaulted*, тому вони відкидаються. SDDL також не підтримує деякі типи ACE, тому їх не включено до таб.5.8.

Як згадувалося раніше, якщо під час перетворення дескриптора безпеки в SDDL зустрічається непідтримуваний тип ACE, процес перетворення завершиться невдачею. Щоб обійти цю проблему, команда PowerShell *ConvertFrom-NtSecurityDescriptor* може перетворити дескриптор безпеки у відносному форматі в *base64*, як показано в лістингу 5.18. Використання *base64* зберігає весь дескриптор безпеки і дозволяє легко його скопіювати.

Лістинг 5.18-Перетворення дескриптора безпеки в формат base64

```
PS C:\WINDOWS\system32> ConvertFrom-NtSecurityDescriptor $sd -AsBase64 -InsertLineBreaks
AQAAgAAAAAAAAAAAAAAAAAAAAAA=
```

Щоб отримати дескриптор безпеки, ви можете передати аргумент *Base64* до команди *New-NtSecurityDescriptor*.

5.7. ПРАКТИЧНІ ПРИКЛАДИ

Завершимо цю розділ декількома практичними прикладами, в яких використовуються команди, про які ви дізналися.

5.7.1.яяяяяяя Розбір бінарного SID у ручному режимі

PowerShell містить команди, за допомогою яких можна аналізувати SID, які мають різну структуру. Однією з таких структур є необроблений масив байтів.

Існуючий SID можна перетворити на масив байтів за допомогою команди *ConvertFrom-NtSid*:

```
PS> $ba = ConvertFrom-NtSid -Sid "S-1-1-0"
```

Також можна перетворити масив байтів у SID за допомогою параметра *Byte* команди *Get-NtSid*, як показано нижче. PowerShell проаналізує масив байтів і поверне SID:

```
PS> Get-NtSid -Byte $ba
```

Хоча PowerShell може виконувати ці перетворення за вас, вам буде корисно зрозуміти, як структуруються дані на низькому рівні. Наприклад, ви можете виявити код, який неправильно аналізує SID, що може призвести до пошкодження пам'яті. Завдяки цьому відкриттю ви можете виявити вразливість безпеки.

Найкращий спосіб навчитися аналізувати бінарну структуру — це написати аналізатор, як ми це робимо в лістинг 5.19.

Для демонстрації ми починаємо з створення довільного SID і перетворення його в масив байтів. Зазвичай, однак, ви отримаєте SID для аналізу іншим способом, наприклад, з пам'яті процесу. Ми також друкуємо SID у вигляді шістнадцяткового числа. (Якщо ви звернетеся до структури SID, показаної на рис. 5.1, ви, можливо, вже зможете виділити його основні компоненти). Далі ми створюємо *BinaryReader* для аналізу масиву байтів у структурованій формі. За допомогою *reader* ми спочатку перевіряємо, чи значення *revision* встановлено на 1. Якщо ні, ми генеруємо помилку. Далі в структурі йде RID count як байт, за яким слідує 6-байтовий *security authority*. Метод *ReadBytes* може повернути короткий reader, тому вам слід перевірити, чи ви прочитали всі шість байтів.

Тепер ми входимо в цикл, щоб прочитати RID з бінарної структури та додати їх до масиву. Далі, використовуючи повноваження безпеки та RID, ми можемо запустити Get-NtSid, щоб створити новий об'єкт SID і перевірити, чи новий SID відповідає тому, з якого ми почали.

Цей лістинг показує приклад того, як вручну розібрати SID (або, фактично, будь-яку бінарну структуру) за допомогою PowerShell. Якщо ви любите ризикувати, ви можете реалізувати власний аналізатор для бінарних форматів дескрипторів безпеки, але це виходить за рамки даного посібника. Простіше використовувати команду *New-NtSecurityDescriptor*, яка зробить цю роботу за вас.

Лістинг 5.19, Ручний аналіз бінарного SID

```
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                                           - ,...
PS> $stm = [System.IO.MemoryStream]::new($ba)
PS> $reader = [System.IO.BinaryReader]::new($stm)
PS> $revision = $reader.ReadByte()
PS> if ($revision -ne 1) { throw "Invalid SID revision" }
PS> $rid_count = $reader.ReadByte()
PS> $auth = $reader.ReadBytes(6)
PS> if ($auth.Length -ne 6) {
          throw "Invalid security authority length" }
PS> $rids = @() 6 PS> while($rid_count -gt 0) {
          $rids += $reader.ReadUInt32()
          $rid_count—
}
PS> $new_sid = Get-NtSid -SecurityAuthorityByte $auth -RelativeIdentifier $rids
PS> $new_sid -eq $sid
True
```

5.7.2. Перерахування SID

Служба LSASS не надає загальнодоступного методу для запиту кожного відомих нею відображень SID-імен. Хоча офіційна документація Microsoft містить список відомих SID, вони не завжди є актуальними і не включають SID, специфічні для певного комп'ютера або корпоративної мережі. Однак ми можемо спробувати перерахувати відображення, використовуючи метод перебору.

У лістингу 5.20 визначимо функцію *Get-AccountSids* для перебору списку SID, для яких LSASS має ім'я.

Функція приймає базовий SID та діапазон значень RID для тестування.

Потім у кожного SID зі списку запитує ім'я. Якщо джерелом імені є *Account*, що вказує на те, що ім'я було отримано з LSASS. Отримуємо деталі SID.

Щоб перевірити функцію, ми викликаємо її з базовим SID, який містить повноваження *Nt*, але не містить RID. Одержуємо список знайдених імен та SID з LSASS.

Зверніть увагу, що SID у вихідних даних не є доменними SID, як можна було б очікувати, а SID WellKnownGroup. Для наших потреб різниця між WellKnownGroup, Group та Alias не є важливою. Усі вони є групами.

Лістинг 5.20. Перебір відомих SID

```
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 $sid
if ($name.Source -eq "Account") {
[PSCustomObject]@{
Sid = $sid;
Name = $name.QualifiedName;
Use = $name.NameUse }
}
$i++ }
}
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--
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--
```

Далі ми спробуємо зламати SID домену BUILTIN методом грубої сили. У цьому випадку ми змінили діапазон RID на основі наших попередніх знань про дійсний діапазон, але ви можете спробувати будь-який інший діапазон, який вам подобається. Зверніть увагу, що ви можете автоматизувати пошук, перевіривши властивість *NameUse* у повернутих об'єктах і викликавши *Get-AccountSids*, коли його значення дорівнює *Domain*. Залишаємо це як вправу для читача.

5.8. ВИСНОВКИ ДО РОЗДІЛУ 5

Розпочали цей розділ із детального вивчення структури дескриптора безпеки. Ми докладно описали його бінарні структури, такі як SID, та розглянули списки контролю доступу та записи контролю доступу, що складають дискреційні та системні ACL. Потім обговорили відмінності між абсолютними та відносними дескрипторами безпеки та причини існування цих двох форматів.

Далі дослідили використання команд *New-NtSecurityDescriptor* та *Add-NtSecurityDescriptorAce* для створення та модифікації дескриптора безпеки, щоб

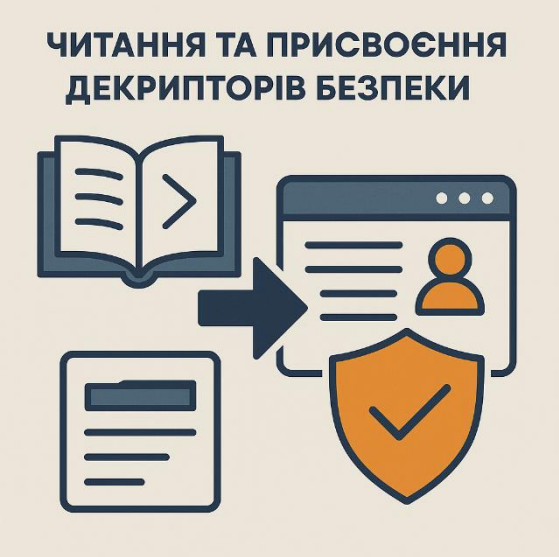
він містив усі необхідні записи. Також розглянули, як відображати дескриптори безпеки у зручній формі за допомогою команди *Format-NtSecurityDescriptor*.

Нарешті, розглянули формат SDDL, який використовується для представлення дескрипторів безпеки. Обговорили, як представляти різні типи значень дескрипторів безпеки, такі як ACE, і як ви можете написати свої власні. Деякі завдання, які ми ще не розглянули, — це як запитувати дескриптор безпеки з об'єкта ядра і як призначити новий. Розглянемо ці теми в наступному розділі.

ПИТАННЯ ДЛЯ САМОКОНТРОЛЮ

1. Які функцій виконує дескриптор безпеки?
2. Яка структура дескриптора безпеки?
3. Перерахуйте допустимі прапорці управління дескриптора безпеки.
4. Як створити SID на основі служби безпеки та RID?
5. Ядро підтримує два формати бінарного представлення для опису безпеки. Які саме?
6. Наведіть структуру абсолютного дескриптора безпеки.
7. Наведіть структуру відносного дескриптора безпеки.
8. Що визначає список ACE?
9. Який тип ACE офіційно не використовує ядро Windows ?
10. Як створити новий дескриптор безпеки?
11. Для чого використовують обов'язкову політику?
12. Наведіть значення обов'язкових політик.
13. Що таке SDDL?

РОЗДІЛ 6. ЧИТАННЯ ТА ПРИЗНАЧЕННЯ ДЕКРИПТОРІВ БЕЗПЕКИ



У попередньому розділі ми обговорили різні структури, що складають дескриптор безпеки. Ви також дізналися, як маніпулювати дескрипторами безпеки в PowerShell і як представляти їх за допомогою формату SDDL. У цьому розділі ми обговоримо, як читати дескриптори безпеки з об'єктів ядра, а також більш складний процес призначення дескрипторів безпеки цим об'єктам.

Ми зосередимося на дескрипторах безпеки, призначених об'єктам ядра. Однак, дескриптор безпеки також можна зберігати в постійній пам'яті, наприклад у файлі або як значення ключа реєстру. У цьому випадку дескриптор безпеки потрібно зберігати у відносному форматі та читати як потік байтів, перш ніж ми зможемо перетворити його у формат, який можна перевірити.

6.1. ЧИТАННЯ ДЕСКРИПТОРІВ БЕЗПЕКИ

Щоб отримати доступ до дескриптора безпеки об'єкта ядра, можна викликати системний виклик *NtQuerySecurityObject*. Цей системний виклик приймає дескриптор об'єкта ядра, а також набір прапорців, що описують компоненти дескриптора безпеки, до яких потрібно отримати доступ. Ці прапорці представлені у переліку *SecurityInformation*.

У таб.6.1 наведено перелік доступних прапорців в останніх версіях Windows, а також розташування інформації в дескрипторі безпеки та дескриптор доступу, необхідний для її запиту.

Таблиця 6.1.

Прапорці *SecurityInformation* та необхідний доступ до них

Назва прапора	Опис	Розташування	Handle
Owner	Запит SID власника.	Owner	ReadControl
Group	Запит SID групи.	Group	ReadControl
Dacl	Запит DACL.	DACL	ReadControl
Sacl	Запит SACL (лише для аудиту ACE).	SACL	AccessSystemSecurity
Label	Запит обов'язкової мітки.	SACL	ReadControl
Attribute	Запит атрибута системного ресурсу.	SACL	ReadControl

Назва прапора	Опис	Розташування	Handle
Scope	Запит ідентифікатора політики ID.	SACL	ReadControl
ProcessTrustLabel	Запит мітки довіри процесу.	SACL	ReadControl
AccessFilter	Запит фільтра доступу.	SACL	ReadControl
Backup	Запит усього, крім мітки довіри процесу та фільтра доступу.	All	ReadControl and AccessSystemSecurity

Для читання більшості цієї інформації вам потрібен лише доступ *ReadControl*, за винятком аудиторських ACE з SACL, для яких потрібен доступ *AccessSystemSecurity*. (Для інших ACE, що зберігаються в SACL, достатньо доступу *ReadControl*.) Єдиний спосіб отримати доступ *AccessSystemSecurity* — це спочатку увімкнути привілей *SeSecurityPrivilege*, а потім явно запросити доступ під час відкриття об'єкта ядра. Лістинг 6.1 ілюструє цю поведінку. Ви повинні виконати ці команди як адміністратор.

Лістинг 6.1. Запит на доступ *AccessSystemSecurity* та увімкнення *SeSecurityPrivilege*

```
PS C:\WINDOWS\system32> $dir = Get-NtDirectory "\BaseNamedObjects" -Access AccessSystemSecurity
Get-NtDirectory : (0xC0000061) - A required privilege is not held by the client.
At line:1 char:8
+ $dir = Get-NtDirectory "\BaseNamedObjects" -Access AccessSystemSecuri ...
+        ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : NotSpecified: (:) [Get-NtDirectory], NtException
    + FullyQualifiedErrorId : NtCoreLib.NtException,NtObjectManager.Cmdlets.Object.GetNtDirectoryCmdlet

PS C:\WINDOWS\system32> Enable-NtTokenPrivilege SeSecurityPrivilege
PS C:\WINDOWS\system32> $dir = Get-NtDirectory "\BaseNamedObjects" -Access AccessSystemSecurity
PS C:\WINDOWS\system32> $dir.GrantedAccess
AccessSystemSecurity
```

Наша перша спроба відкрити каталог BNO з доступом *AccessSystemSecurity* не вдалася, оскільки ми не маємо необхідного привілею *SeSecurityPrivilege*. Далі ми вмикаємо цей привілей і спробуємо ще раз. Цього разу нам вдається відкрити каталог, а виведення його параметра *GrantedAccess* підтверджує, що нам надано доступ *AccessSystemSecurity*.

Не зовсім зрозуміло, чому розробники Windows вирішили захистити читання аудиторської інформації за допомогою *SeSecurityPrivilege*. Хоча ми повинні вважати модифікацію та видалення аудиторської інформації привілейованими діями, немає очевидної причини, чому читання цієї інформації також має бути привілейованим. На жаль, ми змушені миритися з таким рішенням.

Ви можете отримати дескриптор безпеки об'єкта за допомогою команди PowerShell Get-*NtSecurityDescriptor*, яка викликає *NtQuerySecurityObject*. Системний виклик повертає дескриптор безпеки у відносному форматі як масив байтів, який команда PowerShell перетворює в об'єкт SecurityDescriptor і повертає викликаючому. Команда приймає або об'єкт, або шлях до ресурсу, який ви хочете переглянути, як показано в лістингу 6.2, який відображає дескриптор безпеки для каталогу BNO.

Лістинг 6.2. Запит до дескриптора безпеки для каталогу BNO

```
PS C:\WINDOWS\system32> 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
```

Тут ми відкриваємо каталог BNO з доступом *ReadControl*, а потім використовуємо *Get-NtSecurityDescriptor* для запиту дескриптора безпеки з відкритого об'єкта Directory.

За замовчуванням команда *Get-NtSecurityDescriptor* запитує власника, групу, DACL, обов'язкову мітку та мітку довіри процесу. Якщо ви хочете запитати будь-яке інше поле (або пропустити частину повернутої інформації), вам потрібно вказати це через параметр *SecurityInformation*, який приймає значення з таб.6.1. Наприклад, лістинг 6.3 використовує шлях замість об'єкта і запитує тільки поле *Owner*.

Лістинг 6.3. Запит про власника каталогу BNO

```
PS C:\WINDOWS\system32> Get-NtSecurityDescriptor "\BaseNamedObjects" -SecurityInformation Owner

Owner                DACL ACE Count SACL ACE Count Integrity Level
-----                -------------- -------------- ---------------
BUILTIN\Administrators NONE           NONE           NONE
```

У вихідних даних ви можете побачити, що тільки стовпець «Owner» містить дійсну інформацію. Усі інші стовпці тепер мають значення NONE, що вказує на відсутність значення, оскільки ми не запитували цю інформацію.

6.2. СТВОРЕННЯ ДЕСКРИПТОРІВ БЕЗПЕКИ

Читати дескриптор безпеки легко. Потрібно лише мати відповідний доступ до ресурсу ядра та вміти аналізувати формат відносного дескриптора безпеки, який повертається з системного виклику *NtQuerySecurityObject*. Призначення дескриптора безпеки є більш складною операцією. Дескриптор безпеки, призначений ресурсу, залежить від багатьох факторів:

—Чи ресурс створюється?

—Чи створювач вказав дескриптор безпеки під час створення?

—Чи новий ресурс зберігається в контейнері, такому як каталог або ключ реєстру?

—Чи новий ресурс є контейнером або об'єктом?

—Які прапорці контролю встановлені на батьківському або поточному дескрипторі безпеки?

—Який користувач призначає дескриптор безпеки?

—Які ACE містить існуючий дескриптор безпеки?

—Який тип об'єкта ядра призначається?

Як видно зі списку, цей процес включає в себе багато змінних і є однією з головних причин, чому безпека Windows може бути настільки складною. Ми можемо визначити безпеку ресурсу під час створення або через відкритий дескриптор.

Почнемо з більш складного випадку: визначення під час створення.

6.2.1. Створення дескриптора безпеки під час створення ресурсу

При створенні нового ресурсу ядро повинно призначити йому дескриптор безпеки. Крім того, воно повинно зберігати дескриптор безпеки по-різному, залежно від типу створюваного ресурсу. Наприклад, ресурси диспетчера об'єктів

є тимчасовими, тому ядро зберігає дескриптори безпеки в пам'яті. На відміну від цього, дескриптор безпеки драйвера файлової системи повинен зберігатися на диску, інакше він зникне при перезавантаженні комп'ютера.

Хоча механізм зберігання дескриптора безпеки може відрізнятися, ядро все одно повинно дотримуватися багатьох загальних процедур при роботі з ним, таких як застосування правил успадкування. Для забезпечення послідовної реалізації ядро експортує кілька API, які обчислюють дескриптор безпеки, що призначається новому ресурсу. Найбільш використовуваним з цих API є *SeAssignSecurityEx*, який приймає наступні сім параметрів:

Creator security descriptor - Дескриптор безпеки створювача. Необов'язковий дескриптор безпеки, на якому базується новий призначений дескриптор безпеки;

Parent security descriptor - Батьківський дескриптор безпеки. Необов'язковий дескриптор безпеки для базового об'єкта нового ресурсу;

Object type - Тип об'єкта. Необов'язковий GUID, що представляє тип об'єкта, який створюється;

Container – Контейнер. Булеве значення, що вказує, чи є новий ресурс контейнером;

Auto-inherit - Автоматичне успадкування. Набір бітових прапорців, що визначають поведінку автоматичного успадкування;

Token – Токен. Дескриптор токена, що використовується як ідентифікатор створювача;

Generic mapping - Загальне відображення. Відображення загального доступу до конкретних прав доступу для типу ядра.

На основі цих параметрів API обчислює новий дескриптор безпеки та повертає його викликаючому. Дослідивши взаємодію цих параметрів, ми можемо зрозуміти, як ядро призначає дескриптори безпеки новим об'єктам.

Таблиця 6.2.

Приклад параметрів для нового об'єкта **Mutant**

Параметр	Значення параметра
Creator security descriptor	Значення поля SecurityDescriptor у структурі атрибутів об'єкта.
Parent security descriptor	Опис безпеки батьківського каталогу. Не встановлено для анонімного Mutant.
Object type	Не встановлюється.
Container	Встановлюється як False, оскільки Mutant не є контейнером.
Auto-inherit	Встановити значення *AutoInheritDacl*, якщо прапорці керування батьківського дескриптора безпеки містять прапорець *DaclAutoInherited*, а DACL створювача відсутній або немає дескриптора безпеки створювача. Встановити значення *AutoInheritSacl*, якщо прапорці керування батьківського дескриптора безпеки містять прапорець SaclAutoInherited, а SACL створювача відсутній або немає дескриптора безпеки створювача.
Token	Якщо виклик здійснюється від імені іншої особи, встановіть токен імперсонації. В іншому випадку встановіть основний токен процесу того, хто здійснює виклик.
Generic mapping	Встановіть загальне відображення для типу Mutant.

Розглянемо цей процес призначення для об'єкта Mutant. (Цей об'єкт буде видалено після закриття екземпляра PowerShell, щоб ми випадково не залишили непотрібні файли або ключі реєстру.) У таб.6.2 наведено приклад того, як ми можемо встановити параметри під час створення нового об'єкта Mutant за допомогою NtCreateMutant.

Ви можете задатися питанням, чому тип об'єкта не вказаний у таб.6.2. API підтримує цей параметр, але ні менеджер об'єктів, ні менеджер вводу-виводу його не використовують. Його основне призначення — дозволити Active Directory контролювати успадкування.

У таб.6.2 показано лише два можливі прапорці автоматичного успадкування, але ми можемо передати API багато інших. У таб.6.3 наведено лістинг доступних прапорців автоматичного успадкування, деякі з яких ми зустрінемо в прикладах цього розділу.

Найважливішими параметрами *SeAssignSecurityEx*, які слід враховувати, є значення, присвоєні дескрипторам безпеки батьківського елемента та створювача.

Таблиця 6.3.
Прапорці автоматичного успадкування

Назва прапора	Опис
DaclAutoInherit	Автоматичне успадкування DACL.
SaclAutoInherit	Автоматичне успадкування SACL.
DefaultDescriptorForObject	Використовувати дескриптор безпеки за замовчуванням для нового дескриптора безпеки.
AvoidPrivilegeCheck	Не перевіряти привілеї під час встановлення обов'язкової мітки або SACL.
AvoidOwnerCheck	Не перевіряти, чи є власник дійсним для поточного маркера.
DefaultOwnerFromParent	Копіювати SID власника з батьківського дескриптора безпеки.
DefaultGroupFromParent	Копіювати SID групи з батьківського дескриптора безпеки.
MaclNoWriteUp	Автоматично успадковувати обов'язкову мітку з політикою NoWriteUp.
MaclNoReadUp	Автоматично успадковувати обов'язкову мітку з політикою NoReadUp.
MaclNoExecuteUp	Автоматично успадковувати обов'язкову мітку з політикою NoExecutellp.
AvoidOwnerRestriction	Ігнорувати обмеження, накладені на новий DACL батьківським дескриптором безпеки.
ForceUserMode	Виконувати всі перевірки так, ніби вони викликаються з режиму користувача (застосовується тільки для викликів ядра).

Давайте розглянемо кілька варіантів конфігурації цих двох параметрів дескриптора безпеки, щоб зрозуміти різні наслідки.

6.2.2. Створення тільки дескриптора безпеки створювача

У першій конфігурації, яку будемо розглядати, ми викликаємо *NtCreateMutant* із полем *SecurityDescriptor* атрибуту об'єкта, встановленим на допустимий дескриптор безпеки. Якщо новому об'єкту **Mutant** не надано імені, він буде створений без батьківського каталогу, і відповідний батьківський дескриптор безпеки не буде встановлений.

Якщо немає батьківського дескриптора безпеки, прапорці автоматичного успадкування також не будуть встановлені.

Давайте перевіримо цю поведінку, щоб побачити дескриптор безпеки, який генерується при створенні нового об'єкта **Mutant**. Замість того, щоб створювати сам об'єкт, ми будемо використовувати реалізацію *SeAssignSecurityEx* в режимі користувача, яку NTDLL експортує як *RtlNewSecurityObjectEx*. Ми можемо отримати доступ до *RtlNewSecurityObjectEx* за допомогою команди PowerShell *New-NtSecurityDescriptor*, як показано в лістингу 6.4.

Лістинг 6.4. Створення нового дескриптора безпеки з дескриптора безпеки створювача

```
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
PS> $sd = New-NtSecurityDescriptor -Token $token -Creator $creator
-Type Mutant
PS> Format-NtSecurityDescriptor $sd
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 6 - Access: ModifyState|ReadControl
- Flags : None
```

Спочатку ми будуємо дескриптор безпеки створювача з єдиним ACE, надаючи групі **Everyone** доступ **GenericRead**. Форматуючи дескриптор безпеки, ми можемо підтвердити, що у форматованому виведенні присутній тільки DACL. Далі, використовуючи дескриптор безпеки створювача, ми викликаємо команду *New-NtSecurityDescriptor*, передаючи поточний ефективний токен і вказуючи

кінцевий тип об'єкта як **Mutant**. Цей тип об'єкта визначає загальне відображення. Нарешті, ми форматуємо новий дескриптор безпеки.

Ви можете помітити, що дескриптор безпеки змінився під час процесу створення. Він отримав значення **Owner** і **Group**, а вказана маска доступу змінилася з *GenericRead* на ModifyState|ReadControl.

Почнемо з того, що розглянемо, звідки беруться ці нові значення власника та групи. Коли ми не вказуємо значення *Owner* або *Group*, процес створення копіює їх із SID власника та *PrimaryGroup* вказаного токена.

Ми можемо підтвердити це, перевіривши властивості об'єкта Token за допомогою команди PowerShell *Format-NtToken*, як показано в лістингу 6.5.

Лістинг 6.5. Відображення SID власника та *PrimaryGroup* для поточного ефективного токена

```
PS C:\WINDOWS\system32> Format-NtToken $token -Owner -PrimaryGroup
OWNER INFORMATION
-----------------
Name                  Sid
----                  ---
BUILTIN\Administrators S-1-5-32-544

PRIMARY GROUP INFORMATION
-------------------------
Name            Sid
----            ---
REDMIBOOK\hladk S-1-5-21-2687063813-2248694510-27868876-1001
```

Якщо порівняти вихідні дані в лістингу 6.5 із значеннями дескриптора безпеки в лістингу 6.4, можна побачити, що SID власника та групи збігаються. Раніше ви дізналися, що неможливо встановити довільний SID власника для токена. Це значення має бути або SID користувача, або SID, позначений прапорцем *Owner*.

Ви можете запитати: оскільки SID токена використовується для встановлення власника дескриптора безпеки за замовчуванням, чи можна використовувати цю поведінку для визначення довільного SID власника в дескрипторі безпеки? Давайте перевіримо. У лістингу 6.6 ми спочатку встановлюємо дескриптор безпеки на SID користувача **SYSTEM**, а потім намагаємося створити дескриптор безпеки повторно.

Лістинг 6.6. Встановлення користувача *SYSTEM* як власника дескриптора безпеки об'єкта **Mutant**

```
PS C:\WINDOWS\system32> Set-NtSecurityDescriptorOwner $creator -KnownSid LocalSystem
PS C:\WINDOWS\system32> 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.
At line:1 char:1
+ New-NtSecurityDescriptor -Token $token -Creator $creator -Type Mutant
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : NotSpecified: (:) [New-NtSecurityDescriptor], NtException
    + FullyQualifiedErrorId : NtCoreLib.NtException,NtObjectManager.Cmdlets.Object.NewNtSecurityDescriptorCmdlet
```

На цей раз створення завершується з винятком і кодом стану STATUS_INVALID_OWNER. Це відбувається тому, що API перевіряє, чи є призначений SID власника дійсним для вказаного токена. Це не обов'язково має бути SID власника об'єкта Token, але це має бути або SID користувача, або SID групи з встановленим прапором *Owner*.

Ви можете встановити довільний SID власника тільки тоді, коли токен, який використовується для створення дескриптора безпеки, має увімкнений привілей *SeRestorePrivilege*.

Зверніть увагу, що цей токен не обов'язково повинен належати виклику API SeAssignSecurityEx. Ви також можете вимкнути перевірку власника, вказавши прапор автоматичного успадкування *AvoidOwnerCheck*, однак ядро

ніколи не вказує цей прапор під час створення нового об'єкта, тому воно завжди буде виконувати перевірку власника.

Це не означає, що немає можливості встановити іншого власника як звичайного користувача. Однак будь-який спосіб встановлення довільного власника, який ви виявите, є вразливістю безпеки, яку Microsoft, ймовірно, виправить. Прикладом такої помилки є CVE-2018-0748, яка дозволяла користувачам встановлювати довільного власника під час створення файлу. Користувач мав створити файл через спільний доступ до локальної файлової системи, що призводило до обходу перевірки власника.

Немає обмежень щодо значення SID групи, оскільки група не бере участі в перевірці доступу. Однак обмеження застосовуються до SACL. Якщо ви вкажете будь-які ACE аудиту в SACL як частину дескриптора безпеки створювача, ядро вимагатиме SeSecurityPrivilege.

Пам'ятаєте, що під час створення дескриптора безпеки маска доступу змінилася? Це відбувається тому, що процес призначення дескриптора безпеки відображає всі загальні права доступу за допомогою загальної інформації про відображення типу об'єкта. У цьому випадку відображення *GenericRead* типу **Mutant** перетворює маску доступу на *ModifyState|ReadControl*. Є одне виключення з цього правила - якщо ACE має встановлений прапор *InheritOnly*, то загальні права доступу не будуть відображатися. Ви зрозумієте, чому існує це виключення, коли ми обговоримо спадкування.

Ми можемо підтвердити цю поведінку відображення, використовуючи New-*NtSecurityDescriptor* для створення безіменного об'єкта **Mutant**, як показано в лістинг 6.7.

Лістинг 6.7. Перевірка правил призначення дескрипторів безпеки шляхом створення об'єкта **Mutant**

```
PS C:\WINDOWS\system32> $creator = New-NtSecurityDescriptor -Type Mutant
PS C:\WINDOWS\system32> Add-NtSecurityDescriptorAce $creator -Name "Everyone" -Access GenericRead
PS C:\WINDOWS\system32> Use-NtObject($m = New-NtMutant -SecurityDescriptor $creator) { Format-NtSecurityDescriptor $m }
Type: Mutant
Control: DaclPresent

<Owner>
 - Name  : BUILTIN\Administrators
 - Sid   : S-1-5-32-544

<Group>
 - Name  : REDMIBOOK\hladk
 - Sid   : S-1-5-21-2687063813-2248694510-27868876-1001

<DACL>
 - Type  : Allowed
 - Name  : Everyone
 - SID   : S-1-1-0
 - Mask  : 0x00020001
 - Access: ModifyState|ReadControl
 - Flags : None
```

Як бачите, вихідний дескриптор безпеки є таким самим, як і той, що створено в лістингу 6.4.

6.2.3. Не встановлювати дескриптор безпеки створювача та батьківського об'єкта

Розглянемо ще один простий випадок. У цьому сценарії не встановлено ні створювача, ні батьківського дескриптора безпеки. Цей випадок відповідає виклику *NtCreateMutant* без імені або вказаного поля *SecurityDescriptor*.

Скрипт для його тестування ще простіший, ніж попередній, як показано в лістингу 6.8.

Цей виклик функції *New-NtSecurityDescriptor* вимагає лише токен і тип об'єкта ядра. Поля *Owner* і *Group* в кінцевому дескрипторі безпеки встановлюються на значення за замовчуванням на основі властивостей *Owner* і *PrimaryGroup* токена. Але звідки взявся DACL? Ми не вказали ні батьківський дескриптор безпеки, ні дескриптор створювач, тому він не міг походити від одного з них.

Лістинг 6.8. Створення нового дескриптора безпеки без дескриптора створювача або батьківського

```
PS C:\WINDOWS\system32> $token = Get-NtToken -Effective -Pseudo
PS C:\WINDOWS\system32> $sd = New-NtSecurityDescriptor -Token $token -Type Mutant
PS C:\WINDOWS\system32> Format-NtSecurityDescriptor $sd -HideHeader
<Owner>
 - Name  : BUILTIN\Administrators
 - Sid   : S-1-5-32-544

<Group>
 - Name  : REDMIBOOK\hladk
 - Sid   : S-1-5-21-2687063813-2248694510-27868876-1001

<DACL>
 - Type  : Allowed
 - Name  : BUILTIN\Administrators
 - SID   : S-1-5-32-544
 - 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_339658873
 - SID   : S-1-5-5-0-339658873
 - Mask  : 0x00120001
 - Access: ModifyState|ReadControl|Synchronize
 - Flags : None
```

Для цього використовується стандартний DACL об'єкта Token, ACL, що зберігається в токені і використовується як запасний варіант, якщо не вказано інший DACL. Стандартний DACL токена можна відобразити, передавши токен в *Format-NtToken* з параметром *DefaultDacl*, як показано в лістингу 6.9.

Лістинг 6.9. Відображення DACL за замовчуванням для токена

```
PS C:\WINDOWS\system32> Format-NtToken $token -DefaultDacl
DEFAULT DACL
------------
BUILTIN\Administrators: (Allowed)(None)(GenericAll)
NT AUTHORITY\SYSTEM: (Allowed)(None)(GenericAll)
NT AUTHORITY\LogonSessionId_0_339658873: (Allowed)(None)(GenericExecute|GenericRead)
```

Крім прав доступу, специфічних для **Mutant**, DACL в лістингу 6.9 збігається з тим, що в лістингу 6.8. Можна зробити висновок, що якщо під час створення не вказати ні дескриптор безпеки батьківського об'єкта, ні дескриптор безпеки створювача, буде створено новий дескриптор безпеки на основі власника токена, основної групи та DACL за замовчуванням. Однак, щоб переконатися в цьому, перевіримо таку поведінку, створивши анонімний **Mutant** без дескриптора безпеки (лістинг 6.10).

Лістинг 6.10. Створення анонімного **Mutant** для перевірки поведінки створення дескриптора безпеки за замовчуванням

```
PS C:\WINDOWS\system32> Use-NtObject($m = New-NtMutant) { Format-NtSecurityDescriptor $m }
Type: Mutant
Control: None

<NO SECURITY INFORMATION>
```

Зачекайте — новий об'єкт **Mutant** взагалі не має інформації про безпеку! Це не те, чого ми очікували.

Проблема полягає в тому, що ядро дозволяє певним типам об'єктів не мати захисту, коли об'єкт не має імені. Ви можете дізнатися, чи вимагає об'єкт захисту, запитавши його властивість *SecurityRequired*, як показано в лістингу 6.11.

Лістинг 6.11. Запит властивості *SecurityRequired* об'єктів типу **Mutant**

```
PS C:\WINDOWS\system32> Get-NtType "Mutant" | Select-Object SecurityRequired

SecurityRequired
----------------
           False
```

Як бачите, тип Mutant не вимагає безпеки. Отже, якщо при створенні безіменного об'єкта Mutant ми не вказуємо ні дескриптор безпеки створювача, ні дескриптор безпеки батьківського об'єкта, ядро не буде генерувати дескриптор безпеки за замовчуванням.

Чому ядро підтримує можливість створення об'єкта без дескриптора безпеки? Ну, якщо програми не будуть ділитися цим об'єктом між собою, дескриптор безпеки не матиме сенсу. Він лише займатиме додаткову пам'ять ядра. Тільки якщо ви створили об'єкт з іменем, щоб його можна було спільно використовувати, ядро вимагатиме безпеки.

Ви можете дублювати дескриптор неіменованого ресурсу та надавати йому спільний доступ іншому процесу. Однак це слід робити з обережністю. Хоча дублювання дескриптора дозволяє видалити доступ з дескриптора, процес-одержувач може легко повторно дублювати дескриптор, щоб відновити видалений доступ.

До Windows 8 не було можливості призначити безпеку неіменованому об'єкту, для якого *SecurityRequired* було встановлено на *False*. Windows 8 також ввів новий, недокументований прапор до *NtDuplicateObject*, щоб окремо вирішити цю проблему. Вказання прапора *NoRightsUpgrade* під час дублювання дескриптора повідомляє ядру відмовити в будь-яких подальших операціях дублювання, які вимагають додаткових прав доступу.

Щоб перевірити створення дескриптора безпеки за замовчуванням, давайте створимо об'єкт, який вимагає безпеки, наприклад об'єкт *Directory* (лістинг 6.12).

Лістинг 6.12. Створення безіменного об'єкту *Directory* для перевірки дескриптора безпеки за замовчуванням

```
PS C:\WINDOWS\system32> Get-NtType Directory | Select-Object SecurityRequired

SecurityRequired
----------------
            True


PS C:\WINDOWS\system32> Use-NtObject($dir = New-NtDirectory) { Format-NtSecurityDescriptor $dir -Summary }
<Owner> : BUILTIN\Administrators
<Group> : REDMIBOOK\hladk
<DACL>
BUILTIN\Administrators: (Allowed)(None)(Full Access)
NT AUTHORITY\SYSTEM: (Allowed)(None)(Full Access)
NT AUTHORITY\LogonSessionId_0_339658873: (Allowed)(None)(Query|Traverse|ReadControl)
```

Лістинг 6.12 показує, що дескриптор безпеки за замовчуванням відповідає нашим очікуванням.

6.2.4. Встановлення тільки батьківського дескриптора безпеки

Наступний випадок, який ми розглянемо, є набагато складнішим. Припустимо, що ми викликаємо *NtCreate* **Mutant** з іменем, але без вказання поля *SecurityDescriptor*.

Оскільки іменований **Mutant** повинен бути створений в об'єкті *Directory* (який, як ми щойно бачили, вимагає безпеки), буде встановлено батьківський дескриптор безпеки.

Однак, коли ми вказуємо батьківський дескриптор безпеки, ми також вводимо в дію успадкування. Процес, за допомогою якого новий дескриптор безпеки копіює частину батьківського дескриптора безпеки. Правила успадкування визначають, які частини батьківського дескриптора передаються новому дескриптору безпеки, і ми називаємо батьківський дескриптор безпеки таким, що успадковується, якщо його частини можуть бути успадковані.

Мета успадкування полягає у визначенні ієрархічної конфігурації безпеки для дерева ресурсів. Без успадкування нам довелося б явно призначати дескриптор безпеки для кожного нового об'єкта в ієрархії, що досить швидко стало б некерованим. Це також унеможливило б управління деревом ресурсів, оскільки кожна програма могла б поводитися по-різному.

Давайте перевіримо правила успадкування, які застосовуються при створенні нових ресурсів ядра. Ми зосередимося на DACL, але ці концепції застосовуються і до SACL. Щоб мінімізувати дублювання коду, в лістингу 6.13 визначено кілька функцій, які виконують тест з батьківським дескриптором безпеки та реалізують різні опції.

Лістинг 6.13. Визначення тестових функцій для *New-ParentSD* і *Test-NewSD*

```
PS> function New-ParentSD($AceFlags = 0, $Control = 0) {
$owner = Get-NtSid -KnownSid BuiltinAdministrators
$parent = New-NtSecurityDescriptor -Type Directory -Owner $owner -Group $owner
Add-NtSecurityDescriptorAce $parent -Name "Everyone" -Access GenericAll
Add-NtSecurityDescriptorAce $parent -Name "Users" -Access GenericAll -Flags $AceFlags
Add-NtSecurityDescriptorControl $parent -Control $Control
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
        }
$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 7
$sd = New-NtSecurityDescriptor -Token $token -Parent $parent -Creator $creator -Type Mutant -Container:$Container -AutoInherit
$auto_inherit_flags
Write-Output "`r`n-= New SD =-"
Format-NtSecurityDescriptor $sd -Summary
}
```

Функція *New-ParentSD* створює новий дескриптор безпеки з полями *Owner* і *Group*, встановленими для групи *Administrators*. Це дозволить нам перевірити успадкування поля *Owner* або *Group* у будь-якому новому дескрипторі безпеки, який ми створюємо з цього батьківського об'єкта. Ми також встановлюємо *Type* на *Directory*, як і очікується для менеджера об'єктів. Далі ми додаємо два дозволені ACE, один для групи *Everyone* і один для групи *Users*, які відрізняються своїми SID. Ми призначаємо обом ACE доступ *GenericAll* і додаємо кілька додаткових прапорців для ACE *Users*.

Потім функція встановлює кілька необов'язкових прапорців контролю дескриптора безпеки.

Зазвичай, коли ми призначаємо дескриптор безпеки батьківському об'єкту, загальні права доступу відображаються на права доступу, специфічні для типу. Тут ми використовуємо *Edit-NtSecurityDescriptor* з параметром *MapGeneric*, щоб виконати це відображення за нас.

У функції *Test-NewSD* ми створюємо батьківський дескриптор безпеки та обчислюємо будь-які прапорці автоматичного успадкування. Потім ми створюємо новий дескриптор безпеки, встановлюючи властивість *Container*, якщо це необхідно, а також обчислені нами прапорці автоматичного успадкування. Ви можете вказати дескриптор безпеки створювача, який ця функція буде використовувати для створення нового дескриптора безпеки. Наразі ми залишимо це значення як *$null*, але повернемося до нього в наступному розділі.

Нарешті, ми виводимо на консоль батьківський, дескриптор створювача та нові дескриптори безпеки, щоб перевірити вхідні та вихідні дані.

Почнемо з тестування типового випадку: запустимо команду *Test-NewSD* без додаткових параметрів. Команда створить батьківський дескриптор безпеки без встановлених прапорців контролю, тому у виклику *SeAssignSecurityEx* (лістинг 6.14) не повинно бути прапорців автоматичного успадкування.

Лістинг 6.14. Створення нового дескриптора безпеки з батьківським дескриптором безпеки і без дескриптора безпеки створювача

```
PS> Test-NewSD
-= Parent SD =-
<DACL>
Everyone: (Allowed)(None)(Full Access)
BUILTIN\Users: (Allowed)(None)(Full Access)
-= New SD =-
<Owner>:  GRAPHITE\user
<Group> : GRAPHITE\None
<DACL>
GRAPHITE\user: (Allowed)(None)(Full Access)
NT AUTHORITY\SYSTEM: (Allowed)(None)(Full Access)
NT AUTHORITY\LogonSessionId_0_137918: (Allowed)(None)(ModifyState|ReadControl|...)
```

У вихідних даних ми бачимо, що *Owner* і *Group* не походять від батьківського дескриптора безпеки. Замість цього вони є типовими значеннями, які ми розглядали раніше в цьому розділі. Це логічна: власником нового ресурсу повинен бути той, хто його викликає, а не користувач, який створив батьківський об'єкт. Однак новий DACL виглядає не так, як ми очікували.

Він встановлений на стандартний DACL, який ми бачили раніше, і не має жодного відношення до DACL, який ми створили в батьківському дескрипторі безпеки. Причина, по якій ми не отримали жодних ACE з DACL батьківського елемента, полягає в тому, що ми не вказали ACE як успадковуванні. Для цього нам потрібно встановити один або обидва прапорці ACE *ObjectInherit* і *ContainerInherit*.

Перший застосовується тільки до об'єктів, що не є контейнерами, таких як об'єкти **Mutant**, а другий застосовується до об'єктів-контейнерів, таких як об'єкти **Directory**. Різниця між цими двома типами є важливою, оскільки вони впливають на те, як успадковані ACE поширюються на дочірні об'єкти.

Об'єкт Mutant не є контейнером, тому додаємо прапор *ObjectInherit* до ACE в батьківському дескрипторі безпеки (лістинг 6.15).

Лістинг 6.15. Додавання ACE *ObjectInherit* до батьківського дескриптора безпеки

```
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)
```

У цьому лістингу ми вказуємо прапор ACE *ObjectInherit* для тестової функції. Зверніть увагу, що поля *Owner* і *Group* не змінилися, але DACL більше не є за замовчуванням. Замість цього він містить єдиний ACE, який надає групі Users доступ ModifyState|Delete|ReadControl|WriteDac|WriteOwner .

Це ACE, який ми встановили для успадкування.

Однак ви можете помітити проблему: ACE батьківського дескриптора безпеки отримав повний доступ, а ACE нового дескриптора безпеки — ні.

Чому змінилася маска доступу? Насправді вона не змінилася. Процес успадкування просто взяв необроблену маску доступу до *Directory* для ACE батьківського дескриптора безпеки (значення 0x000F000F) і скопіював її до успадкованого ACE.

Діючі біти доступу об'єкта **Mutant** — 0x001F0001. Тому процес успадкування використовує найближче відображення, 0x000F0001, як показано в лістингу 6.16.

Лістинг 6.16. Перевірка успадкованої маски доступу

```
PS> Get-NtAccessMask (0x0001F0001 -band 0x0000F000F) -ToSpecificAccess Mutant
ModifyState, Delete, ReadControl, WriteDac, WriteOwner
```

Це досить серйозна проблема. Зверніть увагу, наприклад, що типу **Mutant** не вистачає права доступу *Synchronize*, яке йому потрібно, щоб викликаючий об'єкт міг очікувати на блокування. Без цього доступу об'єкт **Mutant** буде марним для програми.

Щоб вирішити цю проблему, ACE може встановити прапор *InheritOnly*. В результаті будь-який загальний доступ залишиться незмінним під час початкового призначення.

Прапор *InheritOnly* позначає ACE тільки для успадкування, що запобігає виникненню проблем під час перевірки доступу. У лістингу 6.17 ми перевіряємо цю поведінку, змінюючи виклик тестової функції..

Лістинг 6.17. Додавання *InheritOnly* в ACE

```
PS> Test-NewSD -AceFlags "ObjectInherit, InheritOnly"
-= Parent SD =-
<Owner> : BUILTIN\Administrators
<Group> : BUILTIN\Administrators
<DACL>
Everyone: (Allowed)(None)(Full Access)
BUILTIN\Users: (Allowed)(ObjectInherit, InheritOnly)(GenericAll)

-= New SD =-
<Owner> : GRAPHITE\user
 <Group> : GRAPHITE\None
<DACL>
BUILTIN\Users: (Allowed)(None)(Full Access)
```

У цьому лістингу ми змінюємо прапорці ACE на *ObjectInherit* та *InheritOnly*. У вихідних даних батьківського дескриптора безпеки ми бачимо, що маска доступу більше не відображається як *GenericAll*. Як результат, успадкований ACE тепер має повний доступ, як і потрібно.

Можна припустити, що прапор *ContainerInherit* працює так само, як *ObjectInherit*, чи не так? Не зовсім. Ми перевіряємо його поведінку в лістингу 6.18.

Лістинг 6.18. Створення нового дескриптора безпеки з прапором *ContainerInherit*

```
PS> Test-NewSD -AceFlags "ObjectInherit, InheritOnly" -Container
-= Parent SD =-
<Owner>: BUILTIN\Administrators
<Group>: BUILTIN\Administrators
<DACL>
Everyone: (Allowed)(None)(Full Access)
BUILTIN\Users: (Allowed)(ObjectInherit, InheritOnly) (GenericAll)

-= New SD =-
<Owner> : GRAPHITE\user
 <Group> : GRAPHITE\None
<DACL>
BUILTIN\Users: (Allowed)(None)(Full Access)
BUILTIN\Users: (Allowed)(ContainerInherit, InheritOnly)(GenericAll)
```

Тут ми додаємо прапорці *ContainerInherit* та *InheritOnly* до ACE, а потім передаємо функції параметр *Container*. На відміну від випадку *ObjectInherit*, ми отримуємо два ACE в DACL. Перший ACE надає доступ до нового ресурсу на основі успадкованого ACE. Другий є копією успадкованого ACE з доступом GenericAll..

Ви можете запитати, як ми можемо створити дескриптор безпеки для типу контейнера, коли використовуємо тип **Mutant**. Відповідь полягає в тому, що API не звертає уваги на кінцевий тип, оскільки використовує лише загальне відображення. Однак під час створення реального об'єкта **Mutant** ядро ніколи не вказує прапор *Container*.

Автоматичне поширення ACE є корисним, оскільки дозволяє будувати ієрархію контейнерів без необхідності вручну надавати їм права доступу. Однак іноді може знадобитися відключити це автоматичне поширення, вказавши прапор *NoPropagateInherit* ACE, як показано в лістингу 6.19.

Лістинг 6.19. Використання *NoPropagateInherit* для запобігання автоматичного успадкування ACE

```
PS> $ace_flags = "ContainerInherit, InheritOnly, NoPropagateInherit"
PS> Test-NewSD -AceFlags $ace_flags -Container
--snip--
-= New SD =-
<Owner> : GRAPHITE\user
<Group> : GRAPHITE\None

<DACL>
BUILTIN\Users: (Allowed)(None)(Full Access)
```

Кожен раз, коли вказуємо цей прапор, ACE, що надає доступ до ресурсу, залишається присутнім, але успадковуваний ACE зникає.

Давайте спробуємо іншу конфігурацію прапора ACE, щоб побачити, що відбувається з ACE *ObjectInherit*, коли вони успадковуються контейнером (Лістинг 6.20).

Лістинг 6.20. Тестування прапора *ObjectInherit* для контейнера

```
PS> Test-NewSD -AceFlags "ObjectInherit" -Container
--snip--
-= New SD =-
<Owner> : GRAPHITE\user
<Group> : GRAPHITE\None
<DACL>
BUILTIN\Users: (Allowed)(ObjectInherit, InheritOnly)(ModifyState|...)
```

Ви, можливо, не очікуєте, що контейнер успадкує ACE взагалі, але насправді він отримує ACE з автоматично встановленим прапорцем *InheritOnly*. Це дозволяє контейнеру передавати ACE дочірнім об'єктам, що не є контейнерами.

У таб.6.4 підсумовано правила успадкування для об'єктів-контейнерів та об'єктів, що не є контейнерами, на основі прапорців батьківського ACE. Об'єкти, для яких успадкування не відбувається, виділено жирним шрифтом.

Таблиця 6.4.

Прапорці батьківського ACE і прапорці, встановлені на успадкованих ACE

Батьківські прапори ACE	Неконтейнерний об'єкт	Контейнерний об'єкт
None	**No inheritance**	**No inheritance**
ObjectInherit	None	InheritOnly ObjectInherit
ContainerInherit	**No inheritance**	ContainerInherit
Parent ACE flags	Non-container object	Container object
ObjectInherit NoPropagateInherit	None	**No inheritance**
ContainerInherit NoPropagateInherit	**No inheritance**	None
ContainerInherit ObjectInherit	None	ContainerInherit ObjectInherit
ContainerInherit ObjectInherit NoPropagateInherit	None	None

Розглянемо прапорці автоматичного успадкування. Якщо повернутися до таб.6.3, можна побачити, що якщо DACL має встановлений прапорець управління DaclAutoInherited, ядро буде передавати прапорець DaclAutoInherit до SeAssignSecurityEx, оскільки немає дескриптора безпеки створювача. (SACL має відповідний прапорець SaclAutoInherit, але тут ми зосередимося на DACL). Що робить прапор DaclAutoInherit?

У лістингу 6.21 ми проводимо тест, щоб це з'ясувати.

Лістинг 6.21. Встановлення прапора контролю *DaclAutoInherited* у батьківському дескрипторі безпеки

```
PS> $ace_flags = "ObjectInherit, InheritOnly"
PS> Test-NewSD -AceFlags $ace_flags -Control "DaclAutoInherited"
-= Parent SD =-
<Owner> : BUILTIN\Administrators
<Group> : BUILTIN\Administrators
<DACL> (Auto Inherited)
Everyone: (Allowed)(None)(Full Access)
BUILTIN\Users: (Allowed)(ObjectInherit, InheritOnly)(GenericAll)

-= New SD =-
<Owner> : GRAPHITE\user
<Group> : GRAPHITE\None

<DACL> (Auto Inherited)
BUILTIN\Users: (Allowed)(Inherited)(Full Access)
```

Встановлюємо прапорці контролю батьківського дескриптора безпеки так, щоб вони містили прапор *DaclAutoInherited*, і переконуємося, що він встановлений, переглядаючи форматований DACL. Бачимо, що новий дескриптор безпеки також містить цей прапорець. Крім того, успадкований ACE має прапорець *Inherited*.

Чим прапорці автоматичного успадкування відрізняються від прапорів успадкування, про які ми говорили раніше? Microsoft зберігає обидва типи успадкування з міркувань сумісності (оскільки прапорець *Inherited* був введений тільки в Windows 2000). З точки зору ядра, ці два типи успадкування не дуже відрізняються, за винятком визначення того, чи встановлений прапор *DaclAutoInherited* у новій системі безпеки і отримав будь-який успадкований ACE прапор *Inherited*. Але з точки зору режиму користувача ця модель успадкування вказує, які частини DACL були успадковані від батьківського дескриптора безпеки. Це важлива інформація, і різні API Win32 використовують її.

6.2.5. Встановлення дескрипторів безпеки як створювача, так і батьківського

В цьому випадку ми викликаємо *NtCreateMutant* з іменем і вказуємо поле *SecurityDescriptor*, встановлюючи параметри дескриптора безпеки створювача і батьківського об'єкта. Щоб побачити результат, давайте створимо тестовий код.

У лістингу 6.22 записано функцію для отримання дескриптора безпеки створювача. Будемо використовувати функцію *Test-NewSD*, яку ми написали раніше, для виконання тесту.

Лістинг 6.22. Тестова функція *New-CreatorSD*

```
PS> function New-CreatorSD($AceFlags = 0, $Control = 0, [switch]$NoDacl) {
$creator = New-NtSecurityDescriptor -Type Mutant
        if (!$NoDacl) {
                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
```

Ця функція відрізняється від функції *New-ParentSD*, що була створена в лістингу 6.13, наступним чином: ми використовуємо тип **Mutant** при створенні дескриптора безпеки, дозволяємо виклику не вказувати DACL, і встановлюємо інший SID для DACL, якщо він використовується. Ці зміни дозволять нам розрізнити частини нового дескриптора безпеки, що походять від батьківського, і ті, що походять від створювача.

У деяких простих випадках батьківський дескриптор безпеки не має спадкового DACL, і API дотримується тих самих правил, що й у випадку, коли встановлено лише дескриптор безпеки створювача. Іншими словами, якщо створювач вказує DACL, новий дескриптор безпеки буде його використовувати. В іншому випадку буде використовуватися DACL за замовчуванням.

Якщо батьківський дескриптор безпеки містить спадковий DACL, новий дескриптор безпеки успадкує його, якщо дескриптор безпеки створювача також не має DACL. Навіть порожній або NULL DACL замінить спадкування від батьківського. У лістингу 6.23 ми перевіряємо цю поведінку.

Лістинг 6.23. Тестування успадкування батьківського DACL без DACL створювача

```
PS> $creator = New-CreatorSD -NoDacl
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 =-
 <NO SECURITY INFORMATION>

-= New SD =-
<Owner> : GRAPHITE\user
<Group> : GRAPHITE\None
<DACL>
BUILTIN\Users: (Allowed)(None)(Full Access)
```

Спочатку ми створюємо дескриптор безпеки створювача без DACL, а потім виконуємо тест із успадкованим батьківським дескриптором безпеки. У вихідних даних ми підтверджуємо успадкований ACE для групи *Users* і те, що створювач не має встановленого DACL. Коли ми створюємо новий дескриптор безпеки, він отримує успадкований ACE.

Давайте також перевіримо, що відбувається, коли ми встановлюємо DACL створювача (лістинг 6.24).

У цьому прикладі створюємо дескриптор безпеки створювача з DACL і зберігаємо той самий спадковий батьківський дескриптор безпеки, що і в лістингу 6.23. У вихідних даних ми бачимо, що ACE з DACL створювача були скопійовані до нового дескриптора безпеки.

Лістинг 6.24. Тестування перевизначення спадкування батьківського DACL створювачем DACL

```
PS> $creator = New-CreatorSD
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>
 NT AUTHORITY\NETWORK: (Allowed)(None)(Full Access)
NT AUTHORITY\INTERACTIVE: (Allowed)(None)(Full Access)
```

У двох попередніх тестах не було вказано жодних прапорців автоматичного успадкування. Якщо ми визначимо прапор управління *DaclAutoInherited* на батьківському дескрипторі безпеки, але не включимо DACL створювача, то успадкування відбуватиметься так само, як у лістингу 6.24, за винятком того, що встановлюються прапорці успадкованих ACE.

Однак, якщо ми визначимо і DACL створювача, і прапор управління (лістинг 6.25), відбудеться щось цікаве.

Лістинг 6.25. Тестування успадкування батьківського DACL, коли встановлено прапор створювача DACL і прапор управління *DaclAutoInherited*

```
PS> $creator = New-CreatorSD -AceFlags "Inherited"
PS> Test-NewSD -Creator $creator -AceFlags "ObjectInherit, InheritOnly" -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)
NT AUTHORITY\NETWORK: (Allowed)(None)(Full Access)
BUILTIN\Users: (Allowed)(Inherited)(Full Access)
```

У цьому лістингу визначаємо дескриптор безпеки створювача і встановлюємо INTERACTIVE SID ACE, щоб включити прапор Inherited. Далі запускаємо тест з прапором управління DaclAutoInherited на батьківському дескрипторі безпеки.

У вихідних даних зверніть увагу, що є два ACE. Перший ACE був скопійований зі створювача, а другий - це успадкований ACE від батьківського.

На рис. 6.1 показано таку поведінку автоматичного успадкування.

Коли встановлено *DaclAutoInherit*, DACL нового дескриптора безпеки поєднує не успадковані ACE з дескриптора безпеки створювача з успадкованими ACE від батьківського елемента. Така поведінка автоматичного успадкування дозволяє перебудувати дескриптор безпеки дочірнього елемента на основі його батьківського елемента без втрати будь-яких ACE, які користувач явно додав до DACL. Крім того, автоматичне встановлення прапорця *Inherited* ACE дозволяє нам розрізняти ці явні та успадковані ACE.

Зверніть увагу, що звичайні операції в ядрі не встановлюють прапор *DaclAutoInherit*, який активується тільки в тому випадку, якщо батьківський дескриптор безпеки має встановлений прапор контролю *DaclAutoInherited*, а DACL відсутній. У нашому тесті ми вказали DACL, тому прапор автоматичного успадкування не був встановлений. API Win32 використовують цю поведінку, про що ми поговоримо далі в цьому розділі.

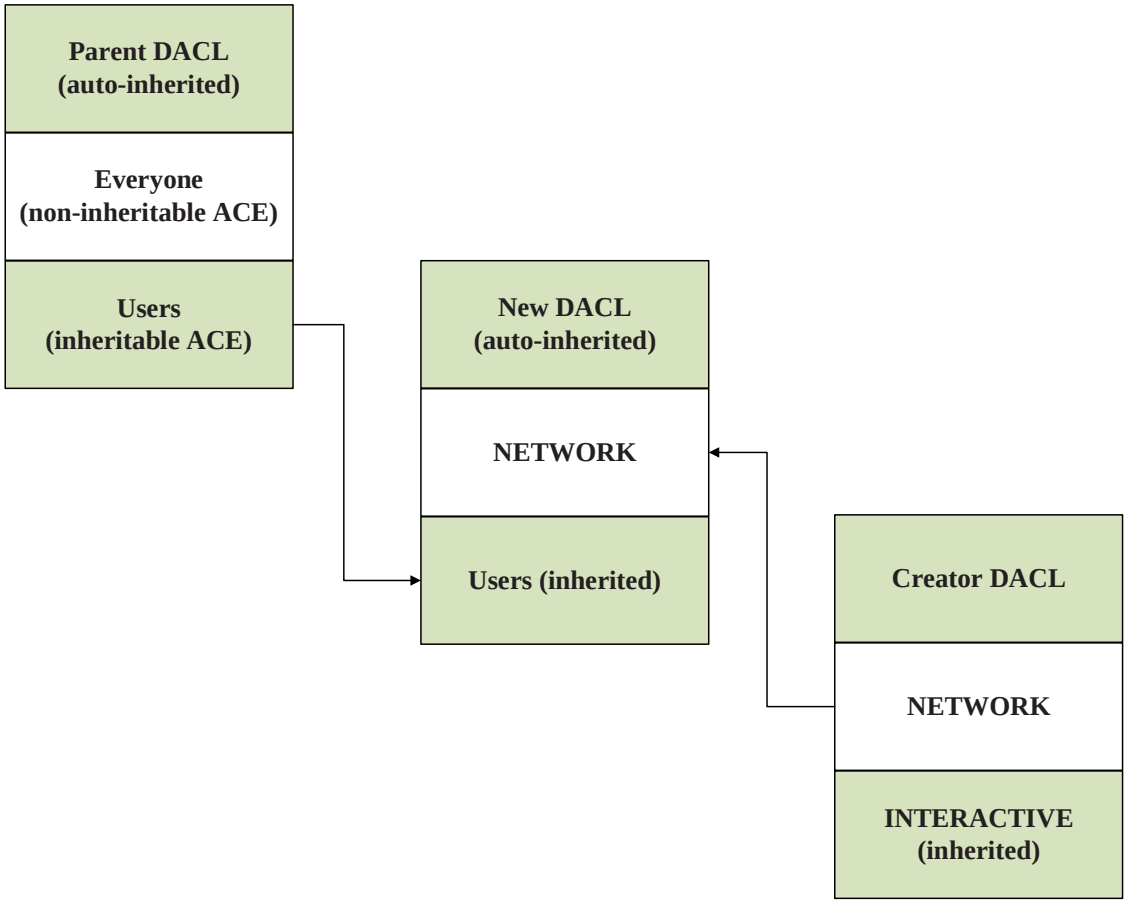


Рисунок 6.1. Поведінка автоматичного успадкування, коли встановлені дескриптори безпеки батьківського об'єкта і створювача

Якщо ви хочете заборонити об'єднання явних ACE та успадкованих ACE батьківського елемента, ви можете встановити прапорці контролю дескриптора безпеки *DaclProtected* та/або *SaclProtected*. Якщо встановлено прапор захисту, правила успадкування залишають відповідний ACL без змін, крім встановлення прапорця контролю *AutoInherited* для ACL та очищення будь-яких успадкованих прапорів ACE. У лістингу 6.26 перевіряємо цю поведінку для DACL.

Лістинг 6.26. Тестування прапора контролю *DaclProtected*

```
PS> $creator = New-CreatorSD -AceFlags "Inherited" -Control "DaclProtected"
PS> Test-NewSD -Creator $creator -AceFlags "ObjectInherit, InheritOnly" -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)
```

Почнемо з генерації дескриптора безпеки створювача з прапором *DaclProtected* і встановлення одного з прапорів ACE на *Inherited*. Потім створюємо новий дескриптор безпеки з автоматично успадкованим батьком. Без прапорця *DaclProtected* DACL нового дескриптора безпеки був би об'єднаною версією DACL створювача та успадкованих ACE від батька. Замість цього ми бачимо лише ACE DACL створювача. Також прапор *Inherited* на другому ACE був очищений.

Що якщо ми не знаємо, чи буде батьківський дескриптор безпеки мати успадковувань ACE, і не хочемо отримати стандартний DACL?

Це може бути важливо для постійних об'єктів, таких як файли або ключі, оскільки стандартний DACL містить тимчасовий SID входу в систему, який не повинен зберігатися на диску. Адже повторне використання SID входу в систему може призвести до надання доступу сторонньому користувачеві.

У цьому випадку ми не можемо встановити DACL у дескрипторі безпеки створювача. Згідно з правилами успадкування, це призведе до перезапису будь-яких успадкованих ACE.

Але можемо вирішити цю проблему за допомогою прапора контролю опису безпеки DaclDefaulted, який вказує, що наданий DACL є типовим. Лістинг 6.27 демонструє використання DaclDefaulted.

Якщо батьківський елемент не містить успадковуваних ACE DACL, новий дескриптор безпеки буде використовувати DACL створювача замість значення за замовчуванням. Якщо батьківський елемент містить успадковуванні ACE, процес успадкування перезапише DACL, дотримуючись правил, описаних вище.

Щоб реалізувати подібну поведінку для SACL, можна використовувати прапор контролю Sacl *Defaulted*. Однак токени не містять SACL за замовчуванням, тому цей прапор є дещо менш важливим

Лістинг 6.27. Тестування прапора *DaclDefaulted*

```
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)
```

6.2.6. Заміна SID CREATOR OWNER та CREATOR GROUP

Під час успадкування в успадкованому ACE зберігається той самий SID, що й оригіналі. У деяких сценаріях це не є бажаним. Наприклад, у вас може бути спільний каталог, який дозволяє будь-якому користувачеві створювати дочірні каталоги. Який дескриптор безпеки ви могли б встановити для цього спільного каталогу, щоб доступ до нього мав лише створювач дочірнього каталогу?

Одним із рішень може бути видалення всіх успадкованих ACE. В результаті новий каталог буде використовувати DACL за замовчуванням. Це майже напевно забезпечить захист каталогу, запобігаючи доступу до нього інших користувачів. Однак, як згадувалося вище, DACL за замовчуванням призначений для тимчасових ресурсів, таких як ресурси в диспетчері об'єктів. Постійні дескриптори безпеки не повинні його використовувати..

Для підтримки таких функцій, як спільні каталоги, реалізація успадкування підтримує чотири спеціальні SID створювача. Коли дескриптор безпеки успадковує ACE з будь-яким із цих SID, реалізація успадкування замінить SID створювача на конкретний SID із токена створювача:

CREATOR OWNER (S-1-3-0)- Замінюється на власника токена;

CREATOR GROUP (S-1-3-1)- Замінюється на основну групу токена;

CREATOR OWNER SERVER (S-1-3-2)- Замінюється на власника серверу;

CREATOR GROUP SERVER (S-1-3-3)- Замінюється основною групою серверу.

Перетворення SID створювача в конкретний SID є одностороннім процесом. Після того, як SID був замінений, ви не зможете відрізнити його від SID, який ви встановили явно. Однак, якщо контейнер успадкував ACE, він збереже SID створювача в ACE *InheritOnly*. Лістинг 6.28 надає приклад.

Лістинг 6.28. Тестування SID створювача під час успадкування

```
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>
CREATOR OWNER: (Allowed)(ContainerInherit, InheritOnly)(GenericWrite)
CREATOR GROUP: (Allowed)(ContainerInherit, InheritOnly)(GenericRead)
PS> $token = Get-NtToken -Effective -Pseudo
PS> $sd = New-NtSecurityDescriptor -Token $token -Parent $parent -Type Directory -Container
PS> Format-NtSecurityDescriptor $sd -Summary -SecurityInformation Dacl
 <DACL>
GRAPHITE\user: Allowed)(None)(CreateObject|CreateSubDirectory|ReadControl)
CREATOR OWNER: (Allowed)(ContainerInherit, InheritOnly)(GenericWrite)
GRAPHITE\None: (Allowed)(None)(Query|Traverse|ReadControl)
CREATOR GROUP: (Allowed)(ContainerInherit, InheritOnly)(GenericRead)
```

Спочатку додаємо два ACE з CREATOR OWNER і CREATOR GROUP SID до батьківського дескриптора безпеки, надаючи ACE різні рівні доступу, щоб їх було легко розрізнити. Потім створюємо новий дескриптор безпеки на основі батьківського, вказуючи, що будемо використовувати його для контейнера.

У відформатованому вигляді бачимо, що SID користувача замінив CREATOR OWNER SID. Цей SID базується на SID власника в токені. Також бачимо, що CREATOR GROUP SID був замінений на SID групи з токена.

Оскільки ми створили дескриптор безпеки для контейнера, ми також бачимо, що є два ACE *InheritOnly*, SID створювача яких не було змінено.

Така поведінка дозволяє SID створювача поширюватися на будь-яких майбутніх дочірніх елементах.

6.2.7. Призначення обов'язкових міток

Обов'язковий мітка ACE містить рівень цілісності ресурсу. Але коли ми створюємо новий дескриптор безпеки, використовуючи токен, рівень цілісності якого більший або дорівнює *Medium*, новий дескриптор безпеки не отримає

обов'язкову мітку за замовчуванням. Ця поведінка пояснює, чому ми досі не бачили жодних обов'язкових міток ACE в наших тестах.

З іншого боку, ця мітка автоматично присвоюється новому дескриптору безпеки якщо рівень цілісності токена менше *Medium*, як показано в лістингу 6.29.

Лістинг 6.29. Призначення обов'язкової мітки токену

```
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()
```

У цьому лістингу ми дублюємо поточний токен і встановлюємо для нього рівень цілісності *Low*. При створенні нового дескриптора безпеки на основі цього токена, ми бачимо, що він має обов'язкову мітку з того ж рівня цілісності.

Додаток може явно встановити обов'язковий мітку ACE при створенні нового ресурсу через дескриптор безпеки створювача. Однак рівень цілісності в обов'язковій мітці ACE повинен бути меншим або дорівнювати рівню цілісності токена. В іншому випадку створювання завершиться невдачею, як показано в лістингу 6.30.

Лістинг 6.30. Визначення обов'язкової мітки на основі дескриптора безпеки створювача

```
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
 New-NtSecurityDescriptor : (0xC0000061) - A required privilege is not held by the client.

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()
```

Спочатку ми створюємо новий дескриптор безпеки створювача і додаємо до нього обов'язкову мітку з рівнем цілісності *System*. Потім ми отримуємо токен викликаючого і встановлюємо його рівень цілісності на *Medium*. Оскільки рівень цілісності системи вищий за *Medium*, якщо ми спробуємо використовувати дескриптор безпеки створювача для призначення нового дескриптора безпеки, операція завершиться невдачею з помилкою STATUS_PRIVILEGE_NOT_HELD.

Щоб встановити вищий рівень цілісності, необхідно надати привілей *SeRelabelPrivilege* токену створювача або вказати прапор автоматичного успадкування *AvoidPrivilegeCheck*. У нашому прикладі ми встановили прапор автоматичного успадкування під час створення нового дескриптора безпеки. Завдяки цьому створення проходить успішно, і ми бачимо обов'язкову мітку у відформатованому вигляді.

Ми можемо зробити обов'язкову мітку ACE успадкованою, встановивши прапор *ObjectInherit* або *ContainerInherit*. Також можна вказати прапор *Inherit Only*, який запобігає використанню рівня цілісності як частини перевірки доступу, зарезервувавши його тільки для успадкування.

Однак слід пам'ятати, що обмеження рівня цілісності застосовуються також до успадкованих обов'язкових міток ACE. Успадкований ACE повинен мати рівень цілісності, який є меншим або дорівнює рівню токена. Інакше призначення дескриптора безпеки не буде успішним. Як і раніше, ми можемо обійти це обмеження за допомогою привілею *SeRelabelPrivilege* або прапора автоматичного успадкування *AvoidPrivilegeCheck*.

Лістинг 6.31 показує приклад, в якому дескриптор безпеки успадковує обов'язкові ACE мітки.

Лістинг 6.31. Визначення обов’язкової мітки через успадкування батьківського дескриптора безпеки

```
PS> $parent = New-NtSecurityDescriptor -Type Mutant
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

<Mandatory Label>
Mandatory Label\Low Mandatory Level: (MandatoryLabel)(Inherited)(NoWriteUp)
```

Спочатку ми створюємо батьківський дескриптор безпеки і призначаємо йому обов'язкову мітку ACE з рівнем цілісності *Low*, та встановленим прапором *ObjectInherit*. Потім створюємо новий дескриптор безпеки, використовуючи батьківський. Новий дескриптор безпеки успадковує обов'язкову мітку, як зазначено прапором *Inherited*.

Деякі типи об'єктів ядра можуть автоматично отримувати обов'язкову мітку, навіть якщо токен викликаючого має рівень цілісності, більший або рівний *Medium*. Вказавши відповідні прапорці автоматичного успадкування, ви завжди можете призначити рівень цілісності викликаючого під час створення нового дескриптора безпеки для ресурсу.

До таких прапорів належать *MaclNoWriteUp*, *MaclNoReadUp* та *MaclNoExecuteUp*, які автоматично успадковують рівень цілісності токена та встановлюють обов'язкову політику відповідно до *NoWriteUp*, *NoReadUp* та *NoExecuteUp*. Комбінуючи ці прапорці, ви можете отримати бажану обов'язкову політику.

В останніх версіях Windows зареєстровано лише чотири типи об’єктів які використовують ці прапорці автоматичного успадкування, як показано в таб.6.5.

Таблиця 6.5.

Типи з автоматичним успадкуванням рівня цілісності

Назва типу	Прапорці автоматичного успадкування
Process	MaclNoWriteUp, MaclNoReadUp
Thread	MaclNoWriteUp, MaclNoReadUp
Job	MaclNoWriteUp
Token	MaclNoWriteUp

Ви можете перевірити поведінку цих прапорів автоматичного успадкування, вказавши їх під час створення дескриптора безпеки. У лістингу

6.32 вказуємо прапорці автоматичного успадкування *MaclNoReadUp* і *MaclNoWriteUp*.

Лістинг 6.32. Визначення обов'язкової мітки шляхом зазначення прапорів автоматичного успадкування

```
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)
```

У вихідних даних ми бачимо обов'язкову мітку ACE із рівнем цілісності *Medium*, хоча на початку цього розділу було зазначено, що рівень *Medium* зазвичай не призначається. Ми також бачимо, що обов'язкова політика встановлена на NoWriteUp|NoReadUp, що відповідає заданим нами прапорам автоматичного успадкування.

6.2.8. Визначення успадкування об'єктів

Коли ми вказуємо тип ACE об'єкта, такий як *AllowedObject*, у батьківському дескрипторі безпеки, правила успадкування дещо змінюються. Це пов'язано з тим, що кожен ACE об'єкта може містити два необов'язкові GUID: *ObjectType*, який використовується для перевірки доступу, та *InheritedObjectType*, який використовується для успадкування.

API *SeAssignSecurityEx* використовує GUID *InheritedObjectType* в ACE для визначення, чи повинен новий дескриптор безпеки успадковувати цей ACE. Якщо цей GUID існує і його значення збігається з GUID *ObjectType*, новий дескриптор безпеки успадковує ACE. Якщо ж значення не збігаються, ACE не копіюється. У таб.6.6 наведено можливі комбінації параметрів *ObjectType* та *InheritedObjectType* і зазначено, чи успадковується ACE.

В таб.6.6 випадки, коли успадкування не відбувається позначено іншим ко кольором.

Зверніть увагу, що це не замінює жодного іншого рішення про успадкування: ACE повинен мати встановлений прапор *ObjectInherit* та/або *ContainerInherit*, щоб бути врахованим для успадкування.

Таблиця 6.6.
Чи буде успадковувати ACE на основі *InheritedObjectType*

Параметр ObjectType визначений?	InheritedObjectType в ACE?	Буде успадкований?
Ні	Ні	Так
Ні	Так	**Ні**
Так	Ні	Так
Так	Так (і значення збігаються)	Так
Так	Так (і значення не збігаються)	**Ні**

У лістингу 6.33 ми перевіряємо цю поведінку, додаючи деякі об'єкти ACE до дескриптора безпеки та використовуючи його як батьківський.

Лістинг 6.33. Перевірка поведінки GUID InheritedObjectType

```
PS> $owner = Get-NtSid -KnownSid BuiltinAdministrators
PS> $parent = New-NtSecurityDescriptor -Type Directory -Owner $owner -Group $owner
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)
```

Спочатку генеруємо кілька випадкових GUID, які будуть виступати в ролі наших типів об'єктів.

Далі ми додаємо три успадковуванні ACE *AllowedObject* до батьківського дескриптора безпеки. У першому ACE ми встановлюємо *ObjectType* для першого GUID, який ми створили.

Цей ACE демонструє, що GUID *ObjectType* не враховується при успадкуванні. Другий ACE встановлює *InheritedObjectType* першого GUID.

Потім ми створюємо новий дескриптор безпеки, передаючи другий GUID до параметра *ObjectType*. Коли ми перевіряємо новий дескриптор безпеки, ми бачимо, що він успадкував ACE без *InheritedObjectType*. Другий ACE у вихідних даних є копією ACE з відповідним GUID *InheritedObjectType*. Зверніть увагу, що, судячи з вихідних даних, *InheritedObjectType* було видалено, оскільки ACE більше не є успадкованим.

Існування єдиного параметра *ObjectType* GUID є дещо негнучким, тому Windows також надає два API, які приймають список GUID, а не єдиний GUID: API ядра *SeAssignSecurityEx2* та API режиму користувача *RtlNewSecurityObject WithMultipleInheritance*. Будь-який ACE у списку з *InheritedObjectType* буде успадкований. В іншому випадку правила успадкування є в основному однаковими з тими, що розглядаються тут.

На цьому ми завершуємо обговорення призначення дескрипторів безпеки під час створення. Як ви бачили, процес призначення є складним, особливо в частині успадкування. Нижче ми розглянемо призначення дескриптора безпеки існуючому ресурсу, що є значно простішим процесом.

6.2.9. Встановлення дескриптора безпеки існуючому ресурсу

Якщо ресурс вже існує, неможливо встановити дескриптор безпеки, викликавши системний виклик створення, такий як *NtCreateMutant*, і вказавши поле *SecurityDescriptor* в атрибутах об'єкта. Замість цього потрібно відкрити дескриптор ресурсу з одним із трьох прав доступу, залежно від того, яку частину дескриптора безпеки ви хочете змінити.

Таблиця 6.7.
Прапорці *SecurityInformation* та необхідний доступ для створення дескриптора безпеки

Назва прапора	Опис	Розташування	Необхідний доступ до дескриптора
Owner	Визначити власника SID.	Owner	WriteOwner
Group	Визначити власника SID групи.	Group	WriteOwner
Dacl	Визначити DACL.	DACL	WriteDac
Sacl	Визначити SACL (тільки для аудиту ACE).	SACL	AccessSystemSecurity
Label	Визначити обов'язкову мітку.	SACL	WriteOwner
Attribute	Визначити атрибут системного ресурсу.	SACL	WriteDac
Scope	Визначити ідентифікатор політики з обмеженою дією.	SACL	AccessSystemSecurity
ProcessTrustLabel	Визначити мітку довіри процесу.	SACL	WriteDac
AccessFilter	Визначити фільтр доступу.	SACL	WriteDac
Backup	Визначити все, крім мітки довіри процесу та фільтра доступу.	All	WriteDac, WriteOwner, and AccessSystemSecurity

Отримавши цей дескриптор, ви можете використати системний виклик *NtSetSecurityObject*, для того щоб задати конкретну інформацію дескриптора безпеки. У таб.6.7 наведено права доступу, необхідні для встановлення полів дескриптора безпеки на основі переліку *SecurityInformation*.

Ви можете помітити, що доступ до дескриптора, необхідний для встановлення цього параметра, є більш складним, ніж доступ, необхідний для простого запиту (описаний у таб.6.1), оскільки права доступу розділені на три рівні замість двох. Замість того, щоб намагатися запам'ятати ці права доступу, ви можете отримати їх за допомогою команди PowerShell *Get-NtAccessMask*, вказавши частини дескриптора безпеки, які ви хочете встановити, за допомогою параметра *SecurityInformation*, як показано в лістингу 6.34.

Лістинг 6.34. Визначення маски доступу, необхідної для запиту або налаштування певної інформації дескриптора безпеки

```
PS> Get-NtAccessMask -SecurityInformation AllBasic -ToGenericAccess ReadControl

PS> Get-NtAccessMask -SecurityInformation AllBasic -ToGenericAccess -SetSecurity
WriteDac, WriteOwner
```

Для встановлення дескриптора безпеки системний виклик *NtSetSecurityObject* активує функцію безпеки, специфічну для даного типу. Ця функція дозволяє ядру підтримувати різні вимоги до зберігання дескрипторів. Наприклад, файл повинен зберігати свій дескриптор безпеки на диску, тоді як менеджер об'єктів може зберігати дескриптор безпеки в пам'яті.

Ці функції, специфічні для кожного типу, в кінцевому підсумку викликають API ядра *SeSetSecurityDescriptorInfoEx* для створення оновленого дескриптора безпеки. Користувацький режим експортує цей API ядра як *RtlSetSecurityObjectEx*. Після оновлення дескриптора безпеки функція, специфічна для кожного типу, може зберегти його за допомогою свого бажаного механізму.

API *SeSetSecurityDescriptorInfoEx* приймає наступні п'ять параметрів і повертає новий дескриптор безпеки::

Modification security descriptor - Новий дескриптор безпеки, переданий до *NtSetSecurityObject*;

Object security descriptor - Поточний дескриптор безпеки для об'єкта, що оновлюється;

Security information - Прапорці для визначення частин дескриптора безпеки, що підлягають оновленню (наведені в таб.6.7);

Auto-inherit - Набір прапорів, які визначають поведінку автоматичного успадкування;

Generic mapping - Загальне відображення для типу, що створюється.

Лістинг 6.35. Використання *Edit-NtSecurityDescriptor* для модифікації існуючого дескриптора безпеки

```
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)(ModifyState|ReadControl)
```

Код ядра не використовує прапорці автоматичного успадкування, тому поведінка цього API є простою. Він просто копіює частини дескриптора безпеки, зазначені в інформації про безпеку, до нового дескриптора.

Він також відображає будь-який загальний доступ до типу-специфічного доступу за допомогою загального відображення, за винятком ACE InheritOnly.

Деякі прапорці керування дескрипторами безпеки запроваджують особливу поведінку. Наприклад, неможливо явно встановити *DaclAutoInherited*, але ви можете вказати його разом із *DaclAutoInheritReq*, щоб налаштувати його нових дескрипторах безпеки.

Можемо протестувати API *RtlSetSecurityObjectEx* за допомогою команди *Edit-NtSecurityDescriptor*, як показано в лістингу 6.35.

Ви можете встановити захист для об'єкта ядра за допомогою команди *Set-NtSecurityDescriptor*. Команда може приймати або дескриптор об'єкта з необхідним доступом, або шлях OMNS до ресурсу. Наприклад, ви можете використовувати наступні команди, щоб спробувати змінити об'єкт \BaseNamedObjects\ABC, встановивши новий DACL:

```
PS> $new_sd = New-NtSecurityDescriptor -Sddl "D:(A;;GA;;;WD)"
PS> Set-NtSecurityDescriptor -Path "\BaseNamedObjects\ABC"
-SecurityDescriptor $new_sd -SecurityInformation Dacl
```

Зверніть увагу на слово «спробувати»: навіть якщо ви зможете відкрити ресурс з необхідним доступом для встановлення компонента дескриптора безпеки, такого як доступ *WriteOwner*, це не означає, що ядро дозволить вам це зробити. Тут застосовуються ті самі правила щодо SID власника та обов'язкових міток, що й під час призначення дескриптора безпеки.

API *SeSetSecurityDescriptorInfoEx* забезпечує дотримання цих правил. Якщо дескриптор безпеки об'єкта не вказано, API повертає код стану STATUS_NO_SECURITY _ON_OBJECT. Тому ви не можете встановити дескриптор безпеки для типу, для якого *SecurityRequired* встановлено на *False*. Цей об'єкт не має дескриптора безпеки, тому будь-яка спроба його зміни призведе до помилки.

Один прапор ACE, про який ще не згадувалося, — це *Critical*. Ядро Windows містить код для перевірки прапора *Critical* і блокування видалення ACE, для яких встановлено цей прапор.

Однак, які ACE вважати критичними, залежить від коду, що призначає новий дескриптор безпеки, і API, такі як *SeSetSecurityInformationEx*. Тому не покладайтеся на прапор *Critical* для виконання будь-яких конкретних дій. Якщо ви використовуєте дескриптори безпеки в режимі користувача, ви можете обробляти прапор будь-яким зручним для вас способом.

Що відбудеться, якщо ви зміните успадковані ACE в *контейнері*? Чи поширяться зміни в дескрипторі безпеки на всі існуючі дочірні елементи?

Одним словом, ні. Технічно цей тип міг би реалізувати таку автоматичну поведінку поширення, але цього не робить. Це завдання покладено на компоненти користувацького режиму. Далі розглянемо API Win32 користувацького режиму, які це реалізують.

6.3. API БЕЗПЕКИ WIN32

Більшість програм не використовують безпосередньо системні виклики ядра для читання або встановлення дескрипторів безпеки. Замість цього вони використовують різноманітні API Win32. Ми розглянемо деякі додаткові функції, які API додають до базових системних викликів.

Win32 реалізує API-інтерфейси *GetKernelObjectSecurity* та *SetKernelObject Security*, які об'єднують *NtQuerySecurityObject* та *NtSetSecurityObject*. Аналогічно, API-інтерфейси *CreatePrivateObjectSecurityEx* та *SetPrivateObjectSecurityEx* Win32 об'єднують *RtlNewSecurityObjectEx* та *RtlSetSecurityObjectEx*, відповідно. Кожна властивість вбудованих API-інтерфейсів, розглянутих у цьому розділі, також застосовується до цих API-інтерфейсів Win32.

Однак Win32 також надає деякі API вищого рівня: зокрема, *GetNamedSecurityInfo* та *SetNamedSecurityInfo*. Ці API дозволяють додатку запитувати або встановлювати дескриптор безпеки, надаючи шлях та тип ресурсу, на який вказує цей шлях. Використання шляху та типу дозволяє функціям бути більш загальними. Наприклад, ці API підтримують отримання та встановлення безпеки не тільки файлів та ключів реєстру, але й служб, принтерів та записів Active Directory Domain Services (ADDS).

Щоб отримати або встановити дескриптор безпеки, API повинен відкрити вказаний ресурс, а потім викликати відповідний API для виконання операції.

Наприклад, щоб отримати дескриптор безпеки файлу, API відкриває файл за допомогою Win32 API *CreateFile*, а потім створює системний виклик *NtQuerySecurityObject*. Однак, щоб отримати дескриптор безпеки принтера, Win32 API повинен відкрити принтер за допомогою API черги друку *OpenPrinter*, а потім викликати API *GetPrinter* на відкритому дескрипторі принтера (оскільки принтер не є об'єктом ядра).

PowerShell вже використовує API *GetNamedSecurityInfo* через команду *Get-Acl*. Однак вбудована команда не підтримує читання певних ACE-дескрипторів безпеки, таких як обов'язкові мітки. Тому модуль *NtObjectManager* реалізує *Get-Win32SecurityDescriptor*, який викликає *GetNamedSecurityInfo* і повертає об'єкт *SecurityDescriptor*.

Якщо ви просто хочете відобразити дескриптор безпеки, ви можете використовувати команду *Format-Win32SecurityDescriptor*, яка приймає ті ж параметри, але не повертає об'єкт *SecurityDescriptor*. Лістинг 6.36 наводить кілька прикладів команд, які використовують базові API безпеки Win32.

Спочатку використовуємо Get-*Win32SecurityDescriptor*, щоб отримати дескриптор безпеки для каталогу Windows, в даному випадку **$env:WinDir**. Зверніть увагу, що ми не вказуємо тип ресурсу, який хочемо отримати, оскільки за замовчуванням це файл.

Список 6.36. Приклад використання *Get-Win32SecurityDescriptor* та *Format-Win32SecurityDescriptor*

```
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--
```

У другому прикладі ми використовуємо *Format-Win32SecurityDescriptor*, щоб відобразити дескриптор безпеки для ключа MACHINE\SOFTWARE. Цей шлях до ключа відповідає шляху до ключа Win32 HKEY_LOCAL_MACHINE\SOFTWARE. Ми повинні вказати, що запитуємо ключ реєстру, вказавши параметр *Type*. Інакше команда спробує відкрити шлях як файл, що навряд чи спрацює.

Щоб знайти формат шляху для кожного підтримуваного типу об'єкта, зверніться до документації API для переліку SE_OBJECT_TYPE, який використовується для визначення типу ресурсу в API *GetNamedSecurityInfo* та *SetNamedSecurityInfo*..

API *SetNamedSecurityInfo* є більш складним, оскільки він реалізує автоматичне успадкування по ієрархіях (наприклад, по дереву каталогів файлів). Як обговорювали раніше, якщо ви використовуєте системний виклик *NtSetSecurityObject* для встановлення дескриптора безпеки файлу, будь-які нові успадковані ACE не будуть поширюватися на існуючі дочірні елементи. Якщо ви встановите дескриптор безпеки на каталозі файлів за допомогою *SetNamedSecurityInfo*, API перелічить усі дочірні файли та каталоги і спробує оновити дескриптор безпеки кожного дочірнього об'єкта.

Лістинг 6.37. Тестування автоматичного успадкування за допомогою *Set-Win32SecurityDescriptor*

```
PS> $path = Join-Path "$env:TEMP" "TestFolder"
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)
Everyone: (Allowed)(Inherited)(Full Access)
PS> $sd = Get-Win32SecurityDescriptor $path
PS> Add-NtSecurityDescriptorAce $sd -KnownSid Anonymous -Access GenericAll -Flags ObjectInherit, ContainerInherit,
InheritOnly
PS> Set-Win32SecurityDescriptor $path $sd Dacl
PS> Format-Win32SecurityDescriptor $item -Summary -SecurityInformation Dacl
<DACL> (Auto Inherited)
Everyone: (Allowed)(Inherited)(Full Access)
NT AUTHORITY\ANONYMOUS LOGON: (Allowed)(Inherited)(Full Access)
```

API SetNamedSecurityInfo створить новий дескриптор безпеки, запитуючи дочірній дескриптор безпеки і використовуючи його як дескриптор безпеки

створювача у виклику RtlNewSecurityObjectEx, беручи батьківський дескриптор безпеки з батьківського каталогу. Прапорці DaclAutoInherit і SaclAutoInherit встановлюються завжди, щоб об'єднати будь-які явні ACE в дескрипторі безпеки створювача в новий дескриптор безпеки.

PowerShell надає доступ до API SetNamedSecurityInfo через команду Set-Win32SecurityDescriptor, як показано в лістингу 6.37.

Лістинг 6.37 демонструє поведінку автоматичного успадкування *SetNamed SecurityInfo* для файлів. Спочатку створюємо каталог *TestFolder* в корені системного диска, потім встановлюємо дескриптор безпеки так, щоб він містив один успадковуваний ACE для групи **Everyone** і мав встановлені прапорці *DaclAutoInherited* і *DaclProtected*. Далі ми створюємо текстовий файл у каталозі і виводимо його дескриптор безпеки. DACL містить єдиний ACE, успадкований текстовим файлом від батьківського елемента.

Потім ми отримуємо дескриптор безпеки з каталогу і додаємо до нього новий успадковуваний ACE для анонімного користувача. Використовуємо цей дескриптор безпеки для встановлення DACL батьківського елемента за допомогою *Set-Win32SecurityDescriptor*.

Знову виводячи дескриптор безпеки текстового файлу, ми бачимо, що він має два ACE, оскільки було додано ACE анонімного користувача. Якби ми використовували *Set-NtSecurityDescriptor* для встановлення дескриптора безпеки батьківського каталогу, це успадкування відбувається.

Лістинг 6.38. Тестування прапорів *ProtectedDacl* та *UnprotectedDacl SecurityInformation*

```
PS> $path = Join-Path "$env:TEMP\TestFolder" "test.txt"
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)

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)
```

Оскільки SetNamedSecurityInfo завжди використовує автоматичне успадкування, застосування захищеного прапора контролю дескриптора безпеки, такого як DaclProtected або SaclProtected, стає важливим способом блокування автоматичного поширення ACE. Як не дивно, API не дозволяє вказувати прапорці контролю DaclProtected і SaclProtected безпосередньо в дескрипторі безпеки. Замість цього він вводить деякі додаткові прапорці SecurityInformation для обробки, встановлення та скасування прапорів контролю. Щоб встановити захищений прапор контролю дескриптора безпеки, можна використовувати прапор ProtectedDacl та ProtectedSacl для SecurityInformation. Щоб скасувати прапорець, використовуйте UnprotectedDacl та UnprotectedSacl.

Лістинг 6.38 надає приклади встановлення та скасування захищеного прапору контролю для DACL.

Цей скрипт розраховує, що ви вже запустили *Лістинг* 6.37, оскільки повторно використовує файл, створений там. Ми створюємо новий дескриптор безпеки з одним ACE для групи *Authenticated Users* і призначаємо його файлу з прапорами *ProtectedDacl* і *Dacl*. В результаті прапор захисту для DACL тепер встановився на файлі. Зверніть увагу, що успадковані ACE з лістингу 6.37 були видалені. Залишився тільки новий, явний ACE.

Потім ми знову призначаємо дескриптор безпеки з прапором *UnprotectedDacl*. Цього разу, коли ми виводимо дескриптор безпеки, бачимо, що прапор захищеного контролю більше не встановлений. Крім того, API відновлює успадковані ACE з батьківського каталогу та об'єднує їх з явним ACE для групи *Authenticated Users*.

Поведінка команди, коли ми вказуємо прапор *UnprotectedDacl*, показує, як можна відновити успадковані ACE для будь-якого файлу. Якщо ви вкажете порожній DACL, щоб явні ACE не об'єднувалися, і додатково вкажете прапор *UnprotectedDacl*, ви скинете дескриптор безпеки до версії, яка базується на його батьківському каталозі. Для спрощення цієї операції модуль PowerShell містить команду *Reset-Win32SecurityDescriptor* (лістинг 6.39).

Лістинг 6.39. Перезавантаження безпеки каталогу за допомогою *Reset-Win32SecurityDescriptor*

```
PS> $path = Join-Path "$env:TEMP\TestFolder" "test.txt"
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)
```

У цьому лістингу ми викликаємо *Reset-Win32SecurityDescriptor* із шляхом до файлу і запитуємо, чи можна скинути DACL. Коли ми відображаємо дескриптор безпеки файлу, ми бачимо, що він збігається з дескриптором безпеки батьківського каталогу, як показано в лістингу 6.37.

Функції автоматичного успадкування API безпеки Win32 зручні для додатків, які можуть просто встановити успадковуваний дескриптор безпеки, щоб застосувати його до будь-яких дочірніх ресурсів. Однак автоматичне успадкування створює ризик для безпеки, особливо якщо його використовують привілейовані додатки або служби.

Ризик виникає, якщо привілейована програма може бути змушена скинути успадковану безпеку для ієрархії, коли зловмисний користувач має контроль над батьківським дескриптором безпеки. Наприклад, CVE-2018-0983 була вразливістю безпеки в привілейованій службі зберігання: вона викликала *SetNamedSecurityInfo*, щоб скинути безпеку файлу із зазначеним користувачем шляхом. Використовуючи деякі хитрощі файлової системи, зловмисник міг пов'язати файл, що скидається, із системним файлом, доступним для запису тільки адміністратору. Однак API *SetNamedSecurityInfo* вважав, що файл знаходиться в каталозі, контрольованому користувачем, тому скидав дескриптор

безпеки на основі дескриптора безпеки цього каталогу, надаючи зловмисному користувачеві повний доступ до системного файлу.

Microsoft виправила цю проблему, і Windows більше не підтримує прийоми роботи з файловою системою, необхідні для використання вразливості. Однак існують інші потенційні способи обману привілейованої служби. Тому, якщо ви пишете код для встановлення або скидання дескриптора безпеки ресурсу, зверніть особливу увагу на те, *звідки походить шлях*. Якщо він походить від непривілейованого користувача, переконайтеся, що ви імітуєте виклик перед викликом будь-якого з API безпеки Win32.

Останній API, про який необхідно згадати, — це *GetInheritanceSource*, який дозволяє ідентифікувати джерело успадкованих ACE ресурсу. Однією з причин, чому ACE позначаються прапором *Inherited*, є полегшення аналізу успадкованих ACE. Без цього прапора API не мав би можливості розрізняти успадковані та не успадковані ACE.

Для кожного ACE з встановленим прапорцем *Inherited* API переміщується вгору по ієрархії батьківських елементів, поки не знайде успадковуваний ACE, який не має цього прапорця, але містить той самий SID і маску доступу. Звичайно, немає гарантії, що знайдений ACE є фактичним джерелом успадкованого ACE, який потенційно може знаходитися вище в ієрархії. Тому розглядайте вихідні дані *GetInheritanceSource* як суто інформаційні і не використовуйте їх для прийняття рішень, критичних для безпеки.

Подібно до інших API Win32, *GetInheritanceSource* підтримує різні типи. Однак, він обмежений ресурсами, які мають відношення нащадок-батько, такими як файли, ключі реєстру та об'єкти ADDS.

Ви можете отримати доступ до API за допомогою команди *Search-Win32SecurityDescriptor*, як показано в лістинг 6.40.

Лістинг 6.40. Перелік успадкованих ACE за допомогою *Search-Win32SecurityDescriptor*

```
PS> $path = Join-Path "$env:TEMP" "TestFolder"
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
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
```

Спочатку викликаємо *Search-Win32SecurityDescriptor* із шляхом до каталогу, який ми створили в лістингу 6.38. Результатом є список ACE в DACL ресурсу, включаючи ім'я ресурсу, від якого було успадкована кожна ACE, та глибину ієрархії. Встановлюємо два явні ACE для каталогу. Результат відображає це як значення глибини 0, що вказує на те, що ACE не було успадкована. Ви також можете побачити, що стовпець «Ім'я» порожній.

Потім створюємо новий файл у каталозі та повторно виконуємо команду. У цьому випадку, як і слід було очікувати, ACE показують, що вони були успадковані від батьківського каталогу, з глибиною 1.

У цій частині розглянуто основи API Win32. Майте на увазі, що між цими API та низькорівневими системними викликами є суттєві відмінності в поведінці, особливо щодо успадкування. Коли ви взаємодієте з безпекою ресурсів через графічний інтерфейс, майже напевно викликається один із API Win32.

6.4. ДЕСКРИПТОРИ БЕЗПЕКИ СЕРВЕРУ ТА КОМБІНОВАНІ ACE

Завершимо розділ темою, яку вже коротко обговорювали, коли ми обговорювали SID створювачів: дескриптори безпеки сервера. Ядро підтримує два дуже погано задокументовані прапорці управління дескрипторами безпеки для серверів: *ServerSecurity* і *DaclUntrusted*. Будемо використовувати ці прапорці тільки під час створювання нового дескриптора безпеки, об'єкта, або під час явного призначення дескриптора безпеки. Основний прапор контролю, *ServerSecurity*, вказує коду створювання дескриптора безпеки, що виклик очікує на імперсонацію іншого користувача.

Коли під час імперсонації створюється новий дескриптор безпеки, SID власника та групи за замовчуванням приймають значення з токена імперсонації. Це може бути небажаним, оскільки власник ресурсу може надати викликаючому додатковий доступ до нього. Однак викликаючий не може встановити власника на довільний SID, оскільки SID повинен пройти перевірку власника, яка базується на токені імперсонації.

Саме тут і з'являється прапор контролю *ServerSecurity*. Якщо ви встановите прапор на дескрипторі безпеки створювача під час створення нового дескриптора безпеки, SID власника та групи за замовчуванням будуть відповідати первинному токену виклику, а не токену імперсонації. Цей прапор також замінює всі дозволені ACE в DACL на дозволені *комбіновані* ACE. У комбінованому ACE SID сервера встановлюється на SID власника з первинного токена. Лістинг 6.41 демонструє приклад.

Спочатку ми створюємо новий дескриптор безпеки, використовуючи токен анонімного користувача. Цей початковий тест не встановлює прапор *ServerSecurity*. Як і очікувалося, власник і група за замовчуванням мають значення, засновані на токені анонімного користувача, а єдиний ACE, який ми додали, залишається незмінним. Далі ми додаємо прапор контролю *ServerSecurity* до дескриптора безпеки створювача. Після повторного виклику New-*NtSecurityDescriptor* ми бачимо, що *Owner* і *Group* встановлені на значення за замовчуванням для основного токена, а не для токена анонімного користувача. Крім того, єдиний ACE був замінений на комбінований ACE, SID сервера якого встановлений на SID власника основного токена.

Лістинг 6.41. Тестування прапора контролю дескриптора безпеки ServerSecurity

```
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
 <Owner> : NT AUTHORITY\ANONYMOUS LOGON
<Group> : NT AUTHORITY\ANONYMOUS LOGON
<DACL>
Everyone: (Allowed)(None)(Full Access)
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)
```

Прапорець управління *DaclUntrusted* працює у поєднанні з *ServerSecurity*. За замовчуванням *ServerSecurity* припускає, що будь-який комбінований ACE в DACL є надійним і копіює його без змін у вихідні дані. Коли встановлено прапор управління *DaclUntrusted*, усі комбіновані ACE замість цього мають значення SID сервера, встановлені на SID власника первинного токена.

Якщо прапор контролю *ServerSecurity* встановлено на дескрипторі безпеки створювача, а новий дескриптор безпеки успадковує ACE від батьківського, ми можемо перетворити SID CREATOR OWNER SERVER і CREATOR GROUP SERVER на відповідні значення первинного токена. Крім того, будь-які успадковані ACE *Allowed* будуть перетворені на комбіновані ACE, за винятком тих, що належать до DACL за замовчуванням.

6.5. ЗВЕДЕННА ІНФОРМАЦІЯ ПРО ТЕ, ЯК ПРАЦЮЄ УСПАДКУВАННЯ

Успадкування є дуже важливою темою, яку необхідно розуміти. У таб.6.8 підсумовано правила успадкування ACL, які ми розглянули в цьому розділі, щоб допомогти вам їх зрозуміти.

Перші два стовпці в цій таблиці описують стан батьківського ACL і ACL створювача. Останні два описують результуючий ACL, залежно від того, чи був встановлений прапор DaclAutoInherit і/або SaclAutoInherit. Існує шість типів ACL, які слід враховувати:

- **Відсутній** - ACL відсутній у дескрипторі безпеки.
- **Присутній** - ACL присутній у дескрипторі безпеки (навіть якщо це NULL або порожній ACL).
- **Не успадковуваний** - ACL не має успадковуваний ACE.
- **Успадковуваний** - ACL має один або кілька успадковуваних ACE.
- **Закритий** - Дескриптор безпеки має встановлений прапор контролю *DaclProtected* або *SaclProtected*.

Таблиця 6.8.
Зведенні правила успадкування для DACL

Батьківський ACL	ACL створювача	Автоматичне успадкування встановлено	Автоматичне успадкування не встановлено
відсутній	відсутній	за замовчуванням	за замовчуванням
відсутній	присутній	створювач	створювач
не успадковуваний	відсутній	за замовчуванням	за замовчуванням
успадковуваний	відсутній	батько	батько
не успадковуваний	присутній	створювач	створювач
успадковуваний	присутній	батько і створювач	створювач
не успадковуваний	закритий	створювач	створювач
успадковуваний	закритий	створювач	створювач
не успадковуваний	встановлений прапор контролю	створювач	створювач
успадковуваний	встановлений прапор контролю	батько	батько

—**Встановлений прапор контролю** - Дескриптор безпеки має встановлений прапор контролю *DaclDefaulted* або *SaclDefaulted*.

Крім того, існує чотири можливі варіанти ACL:

— **За замовчуванням** - DACL за замовчуванням з токена або порожній у випадку SACL

— **Створювач** - Всі ACE від ACL створювача;

— **Батько** - Успадковуванні ACE з батьківського ACL;

— **Батько і створювач** - Успадковуванні ACE з батьківського і явні ACE зі створювача.

Коли встановлено прапорець автоматичного успадкування, новий дескриптор безпеки матиме відповідний прапорець контролю *DaclAutoInherited* або *SaclAutoInherited*. Крім того, всі ACE, які були успадковані від батьківського ACL, матимуть встановлений прапорець Inherited ACE.

Зверніть увагу, що ця таблиця не враховує зміни поведінки, пов'язані з ACE об'єктів, обов'язковими мітками, безпекою сервера та SID створювачів, які значно збільшили складність.

6.6. ПРАКТИЧНІ ПРИКЛАДИ

Розглянемо кілька практичних прикладів, в яких використовуються команди, про які ви дізналися в цьому розділі.

6.6.1. Пошук власників ресурсів диспетчера об'єктів

Як ви дізналися з цього розділу, власником дескриптора безпеки ресурсу зазвичай є користувач, який створив цей ресурс. Однак для адміністраторів це зазвичай вбудована група **Administrators**.

Лістинг 6.42-Пошук об'єктів у *BaseNamedObjects*, які є власністю іншого користувача

```
PS> function Get-NameAndOwner {
                        [CmdletBinding()]
                        param(
                                        [parameter(Mandatory, ValueFromPipeline)]
                                        $Entry,
                                        [parameter(Mandatory)]
                                        $Root
                                )
begin {
                        $curr_owner = Get-NtSid -Owner
}
process {
                        $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 = $sd
                        }
            }
    }
}
PS> Use-NtObject($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--
```

Єдиний спосіб встановити інший SID власника — це використати інший SID групи токенів, для якого встановлено прапорець *Owner*, або ввімкнути *SeRestorePrivilege*.

Жоден із цих варіантів недоступний для користувачів, які не є *адміністраторами*.

Таким чином, якщо відомо, хто є власником ресурсу, можна визначити, чи був цей ресурс створений і використовувався користувачем з більш високим рівнем привілеїв. Це може допомогти виявити потенційні зловживання API безпеки Win32 у привілейованих програмах або знайти спільні ресурси, в які може записувати користувач з нижчим рівнем привілеїв. Користувач з високим рівнем привілеїв може неправильно поводитися з цими ресурсами, що може спричинити проблему безпеки.

Лістинг 6.42 демонструє простий приклад: пошук ресурсів диспетчера об'єктів, SID власника яких відрізняється від SID викликаючого.

Спочатку визначаємо функцію для запиту імені та власника об'єкта в каталозі диспетчера. Функція ініціалізує змінну **$curr_owner** із SID власника токена виклику. Ми порівняємо цей SID із власником ресурсу, щоб повернути тільки ресурси, що належать іншому користувачеві.

Для кожного об'єкта каталогу ми запитуємо його дескриптор безпеки за допомогою команди *Get-NtSecurityDescriptor*. Можемо вказати шлях і кореневий об'єкт каталогу для цієї команди, щоб уникнути необхідності вручну відкривати ресурс. Якщо успішно отримали дескриптор безпеки і якщо SID власника не збігається з SID власника поточного користувача, повертаємо ім'я ресурсу, тип об'єкта та SID власника.

Щоб перевірити нову функцію, ми відкриваємо каталог (у нашому випадку глобальний каталог *BaseNamedObjects*) і використовуємо *Get-NtDirectoryEntry* для запиту всіх записів, передаючи їх через функцію, яку ми визначили. Ви отримуєте список ресурсів, які не належать поточному користувачеві.

Наприклад, вихідні дані містять об'єкт WAMACAPO;8_Mem, який є об'єктом розділу спільної пам'яті. Якщо звичайний користувач може записувати в цей об'єкт розділу, ми повинні це дослідити, може оскільки може бути можливим змусити привілейовану програму виконати операцію, яка підвищить привілеї звичайного користувача.

Ми можемо перевірити нашу здатність отримати доступ на запис до об'єкта *Section*, використовуючи команду *Get-NtGrantedAccess* із властивістю *SecurityDescriptor* об'єкта, як показано в лістингу 6.43.

Лістинг 6.43. Отримання прав доступу для об'єкта *Section*

```
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
```

Змінна **$entry** містить об'єкт, який ми хочемо перевірити. Передаємо його дескриптор безпеки команді *Get-NtGrantedAccess*, щоб повернути максимальний наданий доступ для цього ресурсу. У цьому випадку ми бачимо, що присутній *MapWrite*, що вказує на те, що об'єкт *Section* може бути відображений як доступний для запису.

Приклад, який показано у лістингу 6.42, повинен дати вам розуміння того, як запитувати будь-який ресурс. Ви можете замінити каталог на файл або ключ реєстру, а потім викликати *Get-NtSecurityDescriptor* із шляхом і кореневим об'єктом, щоб запитати власника для кожного з цих типів ресурсів.

Однак для диспетчера об'єктів і реєстру існує набагато простіший спосіб пошуку SID власника. Для реєстру ми можемо переглянути дескриптор безпеки для записів, повернуті з постачальника драйвера *NtObject*, використовуючи властивість *SecurityDescriptor*. Наприклад, ми можемо вибрати поля *Name* і *Owner* SID для кореневого ключа реєстру, використовуючи наступний скрипт:

```
ls NtKey:\ | Select Name, {$_.SecurityDescriptor.Owner.Sid}
```

Також можемо вказати параметр *Recurse*, щоб виконати перевірку рекурсивна. Якщо ви хочете запитати SID власника файлів, ви не можете використовувати цю техніку, оскільки постачальник файлів не повертає постачальника безпеки у своїх записах. Замість цього вам потрібно використовувати вбудовану команду *Get-Acl*. Наприклад, тут ми запитуємо ACL файлу:

```
ls C:\ | Get-Acl | Select Path, Owner
```

Команда *Get-Acl* повертає власника як ім'я користувача, а не SID. Якщо це необхідно, вам доведеться вручну шукати SID за допомогою команди *Get-NtSid* і параметра *Name*.

Як альтернатива, ви можете перетворити вихідні дані команди *Get-Acl* в об'єкт *SecurityDescriptor*, який використовується в модулі *NtObjectManager*, як показано в лістинг 6.44.

Лістинг 6.44. Перетворення вихідних даних *Get-Acl* в об'єкт *SecurityDescriptor*

```
PS> (Get-Acl C:\ | ConvertTo-NtSecurityDescriptor).Owner.Sid
                    Name                Sid
                    ----                ---
NT SERVICE\TrustedInstaller S-1-5-80-956008885-3418522649-1831038044-...
```

Для перетворення ми використовуємо команду PowerShell *ConvertTo-NtSecurityDescriptor*.

6.6.2. Зміна права власності на ресурс

Адміністратори зазвичай отримують права власності на ресурси. Це дозволяє їм легко змінювати дескриптор безпеки ресурсу та отримувати повний доступ до нього.

Лістинг 6.45. Встановлення довільного власника для об'єкта *Directory*

```
PS> $new_dir = New-NtDirectory "ABC" -Win32Path
PS> Get-NtSecurityDescriptor $new_dir | Select {$_.Owner.Sid.Name} $_.Owner.Sid.Name
----------------
BUILTIN\Administrators
PS> Enable-NtTokenPrivilege SeRestorePrivilege
PS> Use-NtObject($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()
```

Windows має кілька інструментів для цього, таких як *takeown.exe*, який призначає власником файлу поточного користувача. Однак, вам буде корисно пройти процес зміни власника вручну, щоб ви могли точно зрозуміти, як це працює. Виконайте команди в лістингу 6.45 як адміністратор.

Починаємо з створення нового об'єкта *Directory*, на якому будемо здійснювати операції. (Ми не будемо змінювати існуючий ресурс, щоб не ризикувати пошкодити вашу систему.) Потім запитуємо поточний SID власника ресурсу. У нашому випадку, оскільки ми запускаємо цей скрипт як адміністратор, він встановлений на групу *Administrators*.

Далі вмикаємо привілей *SeRestorePrivilege*. Це потрібно зробити лише в тому випадку, якщо хочемо встановити довільний SID власника. Якщо хочемо встановити *дозволений* SID, ми можемо пропустити цей рядок. Потім знову відкриваємо каталог, але тільки для доступу *WriteOwner*.

Тепер можемо створити дескриптор безпеки, в якому SID власника встановлено на *World* SID. Для цього викликаємо команду PowerShell *Set-NtSecurityDescriptor*, вказавши тільки прапор *Owner*. Якщо ви не ввімкнули *SeRestorePrivilege*, ця операція завершиться невдачею з кодом статусу STATUS_INVALID_OWNER. Щоб підтвердити, що ми змінили SID власника, ми знову робимо запит, який підтверджує, що тепер SID встановлений на *Everyone* (ім'я *World* SID).

Ви можете застосувати цей самий набір операцій до будь-якого типу ресурсів, включаючи ключі реєстру та файли. Просто змініть команду, яка використовується для відкриття ресурсу. Чи буде вам надано доступ *WriteOwner*, залежить від особливостей процесу перевірки доступу.

6.7. ВИСНОВКИ ДО РОЗДІЛУ 6

Цей розділ розпочався з огляду того, як читати дескриптор безпеки існуючого ресурсу ядра за допомогою команди *Get-NtObjectSecurity*. Ми розглянули прапорці, які визначають, які частини дескрипторів безпеки повинна читати команда, та окреслили спеціальні правила доступу до інформації аудиту, що зберігається в SACL.

Далі обговорили, як можна призначити дескриптори безпеки ресурсам під час створення або шляхом модифікації існуючого ресурсу. У цій частині ви дізналися про успадкування ACL та автоматичне успадкування. Також обговорили поведінку API Win32, зокрема *SetNamedSecurityInfo*, та те, як цей API реалізує автоматичне успадкування, навіть якщо ядро явно його не реалізує.

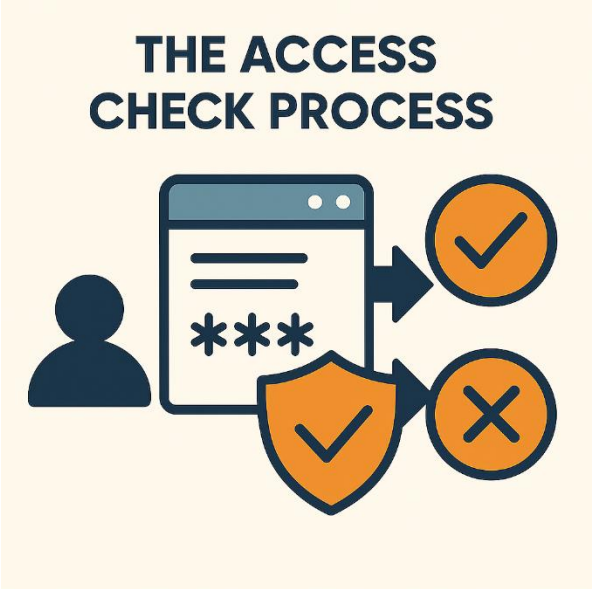
Завершили оглядом погано задокументованого дескриптора безпеки сервера та складених ACE. У наступному розділі обговоримо, як Windows поєднує токен та дескриптор безпеки, щоб перевірити, чи може користувач отримати доступ до ресурсу.

ПИТАННЯ ДЛЯ САМОКОНТРОЛЮ

1. Як отримати доступ до дескриптора безпеки об'єкта ядра?
2. Напишіть запит до дескриптора безпеки для каталогу BNO.
3. Від яких факторів залежить дескриптор безпеки, призначений ресурсу?
4. Наведіть приклад прапорця автоматичного успадкування.

5. Як можливо перевірити правила призначення дескрипторів безпеки?
6. Чи можливо не встановлювати дескриптор безпеки створювача та батьківського об'єкта?
7. Як встановити дескриптори безпеки як створювача, так і батьківського?
8. Чи можливо замінити SID CREATOR OWNER та CREATOR GROUP?
9. Яке призначення має обов'язкова мітка ACE?
10. Як визначити успадкування об'єктів?
11. Як встановити дескриптор безпеки існуючому ресурсу?
12. Як працює успадкування?
13. Як знайти власників ресурсів диспетчера об'єктів?
14. Як змінити права власності на ресурс?

РОЗДІЛ 7. ПРОЦЕС ПЕРЕВІРКИ ДОСТУПУ



Ми розглянули перші два компоненти SRM: токен доступу системи безпеки та дескриптор безпеки. Тепер ми визначимо останній компонент: процес перевірки доступу, який приймає токен і дескриптор безпеки та застосовує фіксований набір правил для визначення, чи може додаток отримати доступ до ресурсу.

Почнемо з обговорення API, які можна викликати для виконання перевірки доступу. Потім ми детально розглянемо реалізацію перевірки доступу в ядрі Windows, як ця перевірка обробляє різні частини дескриптора безпеки та об'єкта токена, щоб згенерувати остаточне значення дозволеного доступу до ресурсу. При цьому ми розробимо власну базову реалізацію процесу перевірки доступу за допомогою скрипта PowerShell.

7.1. ЗАПУСК ПЕРЕВІРКИ ДОСТУПУ

Коли викликаючий намагається відкрити ресурс, ядро виконує перевірку доступу на основі облікових даних викликаючого. API, що використовується для виконання перевірки доступу, залежить від того, чи викликається вона з режиму ядра, чи з режиму користувача. Почнемо з опису API режиму ядра.

7.1.1. Перевірки доступу в режимі ядра

API *SeAccessCheck* реалізує процес перевірки доступу в режимі ядра.

Він приймає наступні параметри:

Security descriptor (Дескриптор безпеки) - Дескриптор безпеки, який використовується для перевірки. Повинен містити як SID власника, так і SID групи;

Security subject context (Контекст суб'єкта безпеки) - Первинні токени та токени для імперсонації для викликаючого;

Desired access (Очікуваний доступ) - Маска доступу, що відповідає запиту на доступ;

Access mode (Режим доступу) - Режим доступу виклику, встановлений як *UserMode* або *KernelMode*;

Generic mapping (Загальне відображення) - Специфічне для даного типу загальне відображення.

API повертає чотири значення:

Granted access (Наданий доступ) - Маска доступу для доступу, наданого користувачеві;

Access status code (Код стану доступу) - Код стану NT, який вказує на результат перевірки доступу;

Privileges (Привілеї) - Будь-які привілеї, використані під час перевірки доступу;

Success code (Код успіху) - Булеве значення. Якщо значення *TRUE*, перевірка доступу пройшла успішно.

Якщо перевірка доступу пройде успішно, API встановить код успіху на true, а код статусу доступу на STATUS_SUCCESS. Однак, якщо хоча б один біт не буде влаштовувати API, він встановить наданий доступ на 0, код успіху на false, а код статусу доступу на STATUS_ACCESS_DENIED.

Ви можете запитати, чому API повертає значення наданого доступу, якщо для того, щоб це значення вказувало на успіх, повинні бути надані всі біти в бажаному доступі. Причина полягає в тому, що така поведінка підтримує біт маски доступу *MaximumAllowed*, який викликаючий може встановити в параметрі бажаного доступу.

Якщо біт встановлено і перевірка доступу надає принаймні один доступ, API повертає STATUS_SUCCESS, встановлюючи наданий доступ на максимально дозволений доступ.

Якщо біт встановлений і перевірка доступу забезпечує хоча б один доступ, API повертає STATUS_SUCCESS, встановлюючи наданий доступ на максимально дозволений рівень.

Параметр контексту суб'єкта безпеки є покажчиком на структуру SECURITY _SUBJECT_CONTEXT, що містить основний токен виклику та будь-який токен імперсонації потоку виклику. Зазвичай код ядра використовує API ядра *SeCaptureSubjectContext* для ініціалізації структури та збору правильних токенів для поточного виклику. Якщо токен імперсонації захоплено, він повинен бути на рівні *Impersonation* або вище. В іншому випадку робота API завершиться з помилкою, а код статусу доступу буде встановлений на STATUS_BAD_IMPERSONATION_LEVEL.

Зверніть увагу, що виклик *SeAccessCheck* може відбуватися не в тому потоці, який зробив початковий запит на ресурс. Наприклад, перевірка може бути делегована фоновому потоку в системному процесі. Ядро може захопити контекст об'єкта з початкового потоку, а потім передати цей контекст потоку, який викликає *SeAccessCheck*, щоб гарантувати, що перевірка доступу використовує правильну ідентичність.

7.1.3. Рівень доступу

Параметр access-mode може мати два значення: *UserMode* і *KernelMode*. Якщо ви встановите для цього параметра значення *UserMode*, всі перевірки

доступу будуть виконуватися в звичайному режимі. Однак, якщо ви встановите значення *KernelMode*, ядро вимкне всі перевірки доступу.

Навіщо викликати *SeAccessCheck* без застосування будь-яких заходів безпеки? Зазвичай ви не будете безпосередньо викликати API із значенням *KernelMode*. Замість цього параметр буде встановлено на значення *PreviousMode* викличного потоку, яке зберігається в структурі об'єкта ядра потоку. Коли ви виконуєте системний виклик із програми в режимі користувача, значення *PreviousMode* встановлюється на *UserMode* і передається до будь-якого API, якому потрібно встановити *AccessMode*.

Тому ядро зазвичай виконує всі перевірки доступу. На рис. 7.1 показано наведену поведінку з програмою в режимі користувача, яка виконує системний виклик *NtCreateMutant*.

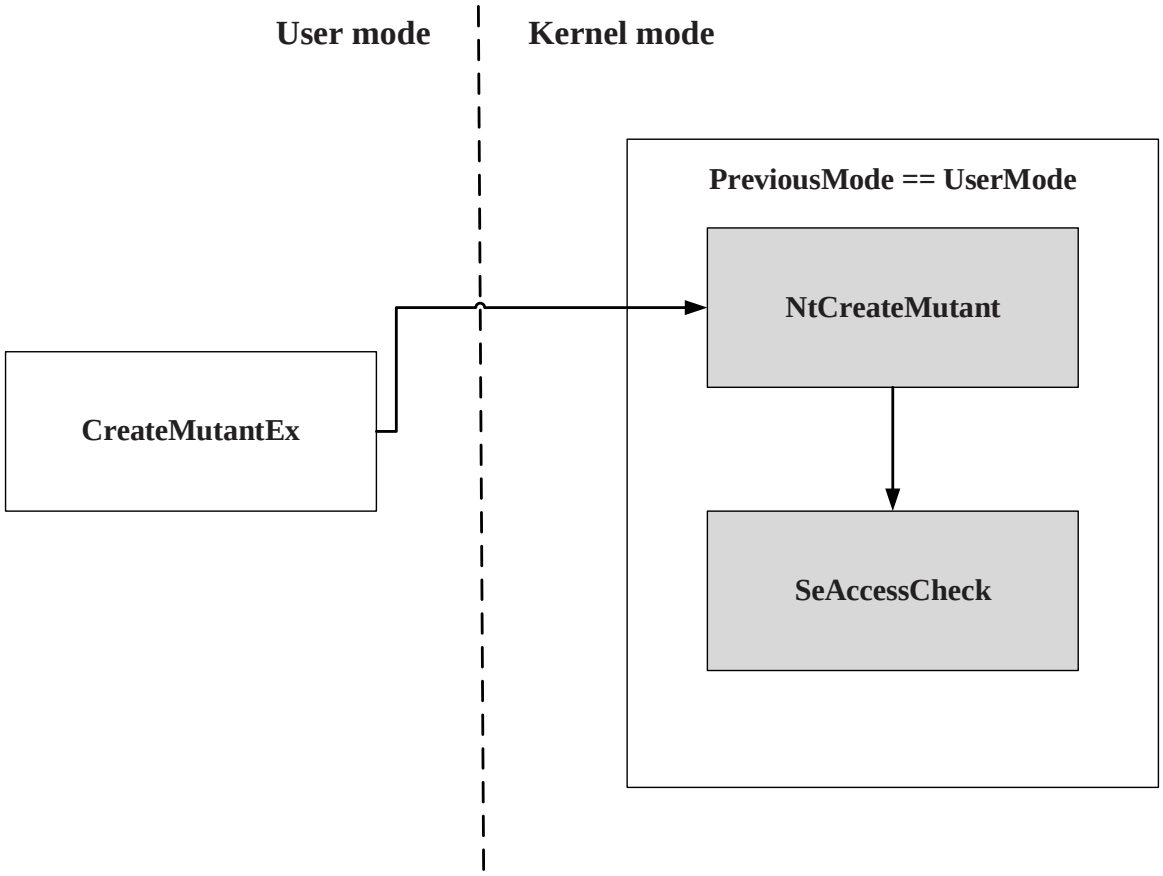


Рисунок 7.1. Значення *PreviousMode* потоку при виконанні системного виклику *NtCreateMutant*

Незважаючи на те, що потік, який викликає *SeAccessCheck* на рис. 7.1, виконує код ядра, значення *PreviousMode* потоку відображає той факт, що виклик був розпочатий з *UserMode*. Тому параметр *AccessMode*, вказаний для *SeAccessCheck*, буде *UserMode*, і ядро буде примусово виконувати перевірку доступу.

Найпоширеніший спосіб переходу значення *PreviousMode* потоку з *UserMode* в *KernelMode* полягає в тому, що існуючий код ядра викликає

системний виклик через його форму **Zw**, наприклад, *ZwCreateMutant*. Коли такий виклик здійснюється, диспетчер системних викликів безпомилково визначає, що попереднє виконання відбулося в ядрі, і встановлює *PreviousMode* в *KernelMode*. На рис. 7.2 показано перехід *PreviousMode* потоку з *UserMode* в *KernelMode*.

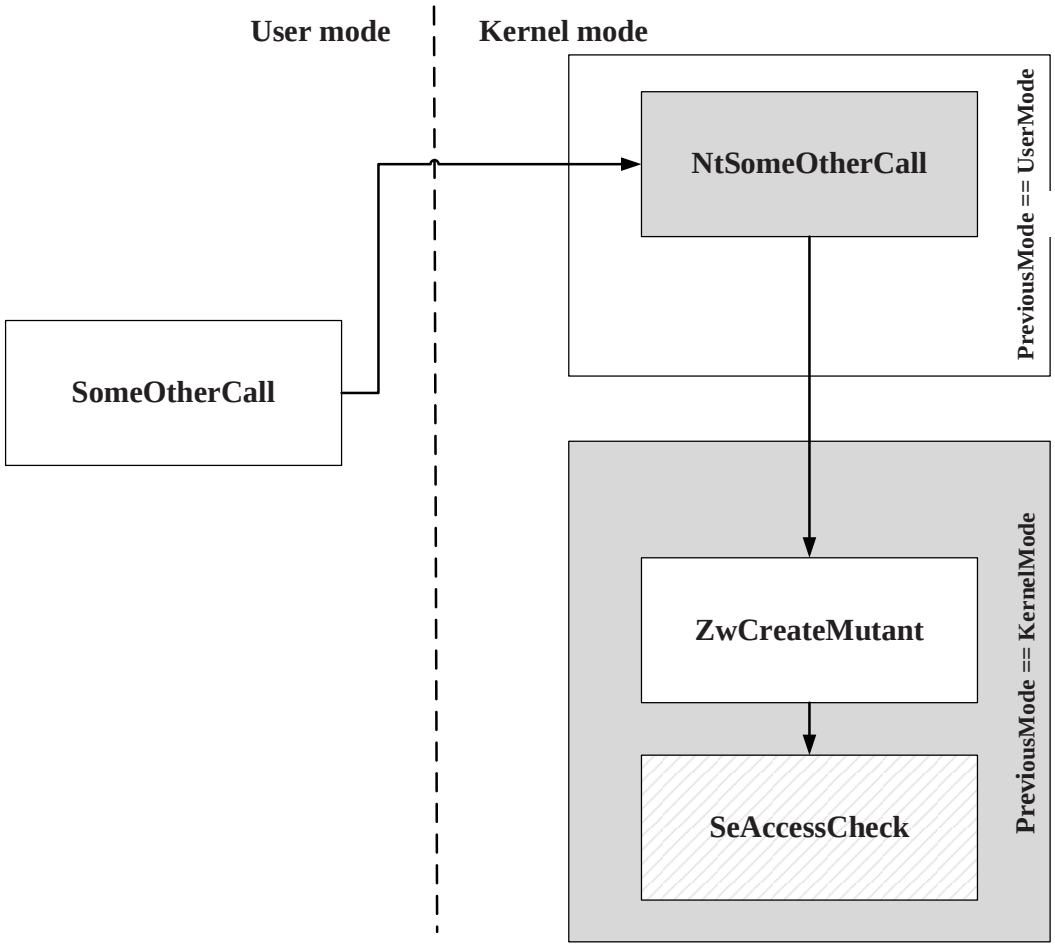


Рисунок 7.2. Значення *PreviousMode* потоку встановлюється на *KernelMode* після виклику *ZwCreateMutant*

На рис. 7.2 програма в режимі користувача викликає умовний системний виклик ядра *NtSomeOtherCall*, який внутрішньо викликає *ZwCreateMutant*. Код, що виконується у функції *NtSomeOtherCall*, працює із значенням *PreviousMode*, встановленим на *UserMode*. Однак після виклику *ZwCreateMutant* режим змінюється на *KernelMode* на час виконання системного виклику. У цьому випадку, оскільки *ZwCreateMutant* викликає *SeAccessCheck* для визначення, чи має викликаючий доступ до об'єкта **Mutant**, API отримує *AccessMode*, встановлений на *KernelMode*, що дозволяє відключити перевірку доступу.

Така поведінка може створювати проблему безпеки, якщо гіпотетична функція *NtSomeOtherCall* дозволяє програмі в режимі користувача впливати на місце створення об'єкта **Mutant**. Після вимкнення перевірки доступу може з'явитися можливість створювати або змінювати об'єкт **Mutant** у розташуванні, до якого користувач зазвичай не має доступу.

7.1.3. Перевірка покажчика пам'яті

Параметр access-mode має ще одне призначення: коли вказано *UserMode*, ядро перевіряє всі покажчики, передані як параметри до API ядра, щоб переконатися, що вони не вказують на місця в пам'яті ядра. Це важливе обмеження безпеки. Воно запобігає тому, щоб програма в режимі користувача змушувала API ядра читати або записувати інформацію в пам'ять ядра, до якої вона не повинна мати доступу.

Зазначення *KernelMode* вимикає ці перевірки покажчиків одночасно з вимиканням перевірки доступу. Таке поєднання поведінки може спричинити проблеми з безпекою: драйвер у режимі ядра може вимкнути лише перевірку покажчиків, але випадково вимкнути також перевірку доступу.

Те, як викликаючий може вказати ці різні параметри режиму доступу, залежить від використовуваних API ядра. Наприклад, іноді можна вказати два значення *AccessMode*: одне для перевірки покажчика, а інше для перевірки доступу. Більш поширеним методом є вказання прапора для виклику. Наприклад, структура OBJECT_ATTRIBUTES, що передається системним викликам, має прапор *ForceAccessCheck*, який вимикає перевірку покажчика, але залишає увімкненою перевірку доступу.

Якщо ви аналізуєте драйвер ядра, варто звернути увагу на використання **Zw** API, в яких не встановлено прапор *ForceAccessCheck*. Якщо користувач, який не є адміністратором, може контролювати шлях до об'єкта-менеджера для виклику, то, ймовірно, існує вразливість безпеки. Наприклад, CVE-2020-17136 — це вразливість у драйвері ядра, який відповідає за реалізацію віддаленої файлової системи Microsoft **OneDrive**. Проблема виникла через те, що API, який драйвер відкрив для оболонки **Explorer**, не встановив прапор *ForceAccessCheck* під час створення хмарного файлу. Через це користувач, який викликав API в драйвері ядра, міг створити довільний файл у будь-якому місці файлової системи, що давало йому можливість отримати права адміністратора.

7.1.4. Перевірка доступу в режимі користувача

Для підтримки додатків у режимі користувача ядро надає доступ до своєї реалізації перевірки доступу через системний виклик *NtAccessCheck*. Цей системний виклик використовує той самий алгоритм перевірки доступу, що й API *SeAccessCheck*, однак він адаптований до унікальної поведінки викликів у режимі користувача. Параметри системного виклику такі:

Security descriptor (Дескриптор безпеки) - Дескриптор безпеки, який використовується для перевірки, повинен містити ідентифікатори власника та групи (SID);

Client token (Токен клієнта) - Дескриптор токена імперсонації для викликаючого;

Desired access (Очікуваний доступ) - Маска доступу, запитуваного викликом;

Generic mapping (Загальне відображення) - Специфічне для даного типу загальне відображення.

API повертає чотири значення:

Granted access (Наданий доступ) - Маска доступу для доступу, наданого користувачеві;

Access status code (Код стану доступу) - Код стану NT, який вказує на результат перевірки доступу;

Privileges (Привілеї) - Будь-які привілеї, використані під час перевірки доступу;

NT success code (Код успіху) - Окремий код стану NT, який вказує стан системного виклику.

Ви помітите, що деякі параметри, присутні в API ядра, тут відсутні. Наприклад, немає необхідності вказувати режим доступу, оскільки він завжди буде встановлений на режим виклику (*UserMode* для виклику в режимі користувача).

Крім того, ідентифікатор викликаючого тепер є дескриптором токена імперсонації, а не контекстом суб'єкта. Цей дескриптор повинен мати доступ *Query*, щоб його можна було використовувати для перевірки доступу. Якщо ви хочете виконати перевірку доступу для первинного токена, спочатку потрібно продублювати цей токен у токен імперсонації.

Інша відмінність полягає в тому, що токен імперсонації, який використовується в режимі користувача, може бути на рівні *Identification*. Причиною такої відмінності є те, що системний виклик призначений для служб користувача, які хочуть перевірити дозволи того, хто викликає, і можливо, що він отримав доступ до токена рівня *Identification*. Ця умова повинна бути врахована.

Системний виклик також повертає додатковий код стану NT замість булевого значення, яке повертає API ядра. Завдяки цьому можна визначити, чи були проблеми з параметрами, переданими системному виклику.

Наприклад, якщо в дескрипторі безпеки не встановлено ідентифікатори власника та групи, системний виклик поверне STATUS_INVALID_SECURITY_DESCR.

Ми можемо використовувати системний виклик *NtAccessCheck* для визначення дозволеного доступу викликаючого на основі дескриптора безпеки та токена доступу. Модуль PowerShell обробляє виклик *NtAccessCheck* за допомогою команди *Get-NtGrantedAccess*, як показано в лістингу 7.1.

Розпочинаємо з створення дескриптора безпеки за замовчуванням, використовуючи параметр EffectiveToken, і підтверджуємо його правильність, форматуючи його. Простіше кажучи, системний виклик перевірить DACL цього дескриптора безпеки на наявність ACE Allowed, який відповідає одному з SID токена. Якщо такий ACE існує, він надає маску доступу. Оскільки перший ACE в DACL надає повний доступ поточному SID користувача, ми очікуємо, що результат перевірки також забезпечить повний доступ.

Лістинг 7.1. Визначення дозволеного доступу користувача, що здійснює виклик

```
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
```

Потім ми викликаємо Get-*NtGrantedAccess*, передаючи йому дескриптор безпеки. Ми не вказуємо явний токен, тому він використовує поточний *ефективний* токен. Ми також не вказуємо маску доступу, що означає, що команда перевіряє доступ *MaximumAllowed*, перетворюючи результат у рядок. Як і очікувалося, повертається повний доступ на основі DACL.

Далі ми перевіряємо команду Get-*NtGrantedAccess*, коли надається явна маска доступу за допомогою параметра *Access*. Команда виконає перелік масок доступу для типу дескриптора безпеки, щоб ми могли вказати значення, специфічні для нашого типу. Ми запросили перевірку *ModifyState*, тому отримуємо тільки цей доступ. Наприклад, якщо ми відкривали дескриптор об'єкта **Mutant**, то маска доступу дескриптора надавала б тільки *ModifyState*.

Нарешті, щоб перевірити випадок відмови в доступі, ми видаляємо всі ACE з DACL. Якщо немає ACE *Allowed*, то доступ не повинен бути наданий. Але коли ми знову запускаємо *Get-NtGrantedAccess*, нас чекає несподіванка: нам надано доступ *ReadControl* і *WriteDac* замість повної відмови. Щоб зрозуміти, чому ми отримали ці рівні доступу, нам потрібно заглибитися в внутрішні процеси перевірки доступу. Ми зробимо це в наступному розділі.

7.2. ПРОЦЕС ПЕРЕВІРКИ ДОСТУПУ В POWERSHELL

Процес перевірки доступу в Windows істотно змінився з часів першої версії Windows NT. Ця еволюція призвела до створення складного набору алгоритмів, які розраховують, який доступ надається користувачеві на основі комбінації дескриптора безпеки і токена. Блок-схема на рис. 7.3 показує основні етапи процесу перевірки доступу.

Першим кроком є об'єднання токена, дескриптора безпеки та бажаної маски доступу. Після цього процес перевірки доступу використовує цю

інформацію у трьох основних перевірках, щоб визначити, чи слід надати доступ або відмовити в ньому:

Mandatory access check (Обов'язкова перевірка доступу) - Відмовляє в доступі до ресурсів, якщо токен не відповідає встановленій політиці;

Token access check (Перевірка доступу на основі токена) - Надає доступ на основі власника токена та його привілеїв;

Discretionary access check (Дискреційна перевірка доступу) - Надає або відмовляє в доступі на основі DACL.

Щоб детальніше розглянути ці кроки, давайте напишемо базову реалізацію процесу перевірки доступу в PowerShell. Ця реалізація PowerShell не замінить команду *Get-NtGrantedAccess*, оскільки для простоти вона не перевірятиме максимально дозволений доступ і може не включати новіші функції.

Проте наявність реалізації, яку можна аналізувати та налагоджувати, допоможе вам краще зрозуміти загальний процес.

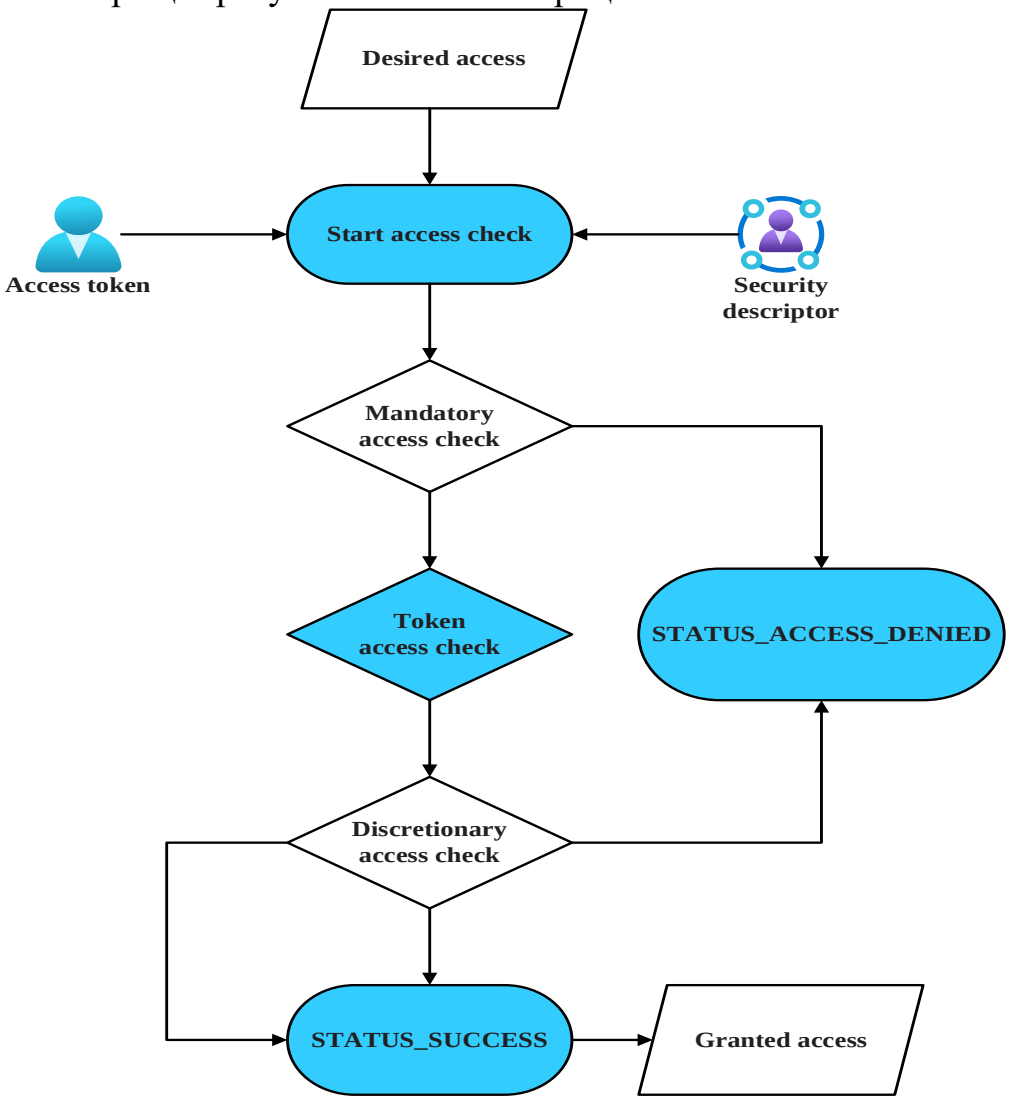


Рисунок 7.3. Процес перевірки доступу

7.2.1. Визначення функції перевірки доступу

Модуль експортує єдину функцію верхнього рівня для виконання перевірки доступу, *Get-PSGrantedAccess*, показану в лістингу 7.2.

Функція приймає чотири параметри, які ми визначили раніше в цьому розділі: токен, дескриптор безпеки, загальне відображення типу та бажаний доступ. Якщо викликаючий не вказав токен, для решти перевірки доступу будемо використовувати його *ефективний* токен.

Листинг 7.2. Функция проверки доступа верхнего уровня

```
function Get-PSGrantedAccess {
    param(
        $Token = (Get-NtToken -Effective -Pseudo),
        $SecurityDescriptor,
        $GenericMapping,
        $DesiredAccess
    )

        $context = @{
        Token = $Token
        SecurityDescriptor = $SecurityDescriptor
        GenericMapping = $GenericMapping
        RemainingAccess = Get-NtAccessMask
        $DesiredAccess Privileges = @()
    }

    ## Test-MandatoryAccess defined below.
        if (!(Test-MandatoryAccess $context)) {
            return Get-AccessResult STATUS_ACCESS_DENIED
            }
    ## Get-TokenAccess defined below.
    Resolve-TokenAccess $context
        if (Test-NtAccessMask $context.RemainingAccess -Empty) {
        return Get-AccessResult STATUS_SUCCESS $context.Privileges
         $DesiredAccess
            }

        if (Test-NtAccessMask $context.RemainingAccess AccessSystemSecurity) {
        return Get-AccessResult STATUS_PRIVILEGE_NOT_HELD
        }
        Get-DiscretionaryAccess $context
        if (Test-NtAccessMask $context.RemainingAccess -Empty) {
        return Get-AccessResult STATUS_SUCCESS
        $context.Privileges $DesiredAccess
        }
        return Get-AccessResult STATUS_ACCESS_DENIED
```

Перше завдання, яке виконує ця функція, — це створення контексту, що відображає поточний стан процесу перевірки доступу. Найважливішою властивістю, яка тут використовується, є *RemainingAccess*. Спочатку ми встановлюємо для цієї властивості параметр *DesiredAccess*, а потім видаляємо біти з властивості, коли вони надаються під час процесу перевірки доступу.

Решта функції відповідає блок-схемі на рис. 7.3. Спочатку вона здійснює обов'язкову перевірку доступу. Ми опишемо, що робить ця перевірка, у

наступному розділі. Якщо перевірка не пройдена, функція завершується з STATUS_ACCESS_DENIED. Для спрощення коду повний скрипт визначає допоміжну функцію *Get-AccessResult* для отримання результату перевірки доступу. Лістинг 7.3 демонструє визначення цієї функції.

Лістинг 7.3. Реалізація допоміжної функції Get-*AccessResult*

```
function Get-AccessResult {
            param(
            $Status,
            $Privileges = @(),
            $GrantedAccess = 0
            )

            $props = @{
            Status = Get-NtStatus -Name $Status -PassStatus
            GrantedAccess = $GrantedAccess
            Privileges = $Privileges
            }
return [PSCustomObject]$props
}
```

Наступним кроком перевірка доступу за токеном оновлює властивість *RemainingAccess* у контексті. Якщо *RemainingAccess* стає порожньою, ми можемо зробити висновок, що нам надано всі права доступу, і повернути STATUS_SUCCESS. Якщо вона **не** порожня, ми виконуємо другу перевірку: якщо викликаючий запитував *AccessSystemSecurity*, а токен не надав цього права, ця перевірка завершується невдачею.

Нарешті, ми виконуємо перевірку дискреційного доступу. Як і при перевірці доступу за токеном, ми перевіряємо властивість *RemainingAccess*: якщо вона порожня, то викликаючий отримав всі запитувані ним дозволи на доступ. В іншому випадку йому було відмовлено в доступі. Маючи на увазі цей огляд, давайте по черзі розглянемо деталі кожної перевірки.

7.2.2. Виконання обов'язкової перевірки доступу

У Windows Vista було впроваджено функцію під назвою **Mandatory Integrity Control** «Обов'язковий контроль цілісності» (MIC), яка використовує рівень цілісності маркера та обов'язкову мітку ACE для контролю доступу до ресурсів на основі загальної політики.

MIC є типом обов'язкової перевірки доступу (MAC). Ключовою особливістю MAC є те, що вона не може надавати доступ до ресурсу, а лише відмовляти в доступі. Якщо викликаючий запитує доступ, що перевищує дозволений політикою, перевірка доступу негайно відмовить у доступі, і якщо MAC відмовляє в доступі, DACL ніколи не перевірятиметься. Оскільки у непривілейованого користувача немає можливості обійти перевірку, вона вважається обов'язковою.

В останніх версіях Windows процес перевірки доступу виконує дві додаткові обов'язкові перевірки разом з MIC. Ці перевірки реалізують подібну поведінку, тому ми об'єднаємо їх в одну групу. Лістинг 7.4 визначає функцію *Test-MandatoryAccess*, яку ми використовували в лістингу 7.2.

Лістинг 7.4. Реалізація функції *Test-MandatoryAccess*

```
function Test-MandatoryAccess {
        param($Context)

        ## Test-ProcessTrustLevel is defined below.
        if (!(Test-ProcessTrustLevel $Context)) {
        return $false
        }

        ## Test-AccessFilter is defined below.
        if (!(Test-AccessFilter $Context)) {
        return $false
        }

        ## Test-MandatoryIntegrityLevel is defined below.
        if (!(Test-MandatoryIntegrityLevel $Context)) {
        return $false
        }
        return $true
}
```

Ця функція виконує три перевірки: *Test-ProcessTrustLevel*, *Test-Access Filter* та *Test-MandatoryIntegrityLevel*. Якщо будь-яка з цих перевірок не пройде, то весь процес перевірки доступу завершиться невдачею, повертаючи STATUS_ACCESS_DENIED. Давайте по черзі детально розглянемо кожну перевірку.

7.2.3. Перевірка рівня довіри до процесу

Windows Vista запровадила захищені процеси, якими навіть адміністратор не може маніпулювати та порушувати їхню цілісність. Спочатку захищені процеси були призначені для захисту медіа-вмісту. Однак згодом Microsoft розширила їхнє застосування, включивши до них захист антивірусних служб та віртуальних машин.

Токену може бути присвоєно SID рівня довіри процесу. Цей SID залежить від рівня захисту захищеного процесу і присвоюється під час створення такого процесу. Щоб обмежити доступ до ресурсу, процес перевірки доступу визначає, чи SID токена є рівноцінним або більш надійним, ніж SID рівня довіри в дескрипторі безпеки.

Коли один SID вважається рівноцінним або більш надійним, ніж інший, кажуть, що він домінує. Щоб перевірити, чи домінує один SID рівня довіри процесу над іншим, можна викликати API *RtlSidDominatesForTrust* або команду *Compare-NtSid* з параметром *Dominates*. У лістингу 7.5 алгоритм перевірки рівня довіри процесу, який зберігається в мітці довіри процесу ACE, передається в PowerShell.

Щоб перевірити рівень довіри процесу, нам потрібно запитати SID для поточного токена. Якщо токен не має SID рівня довіри, ми визначаємо найнижчий можливий SID. Далі ми виставляємо маску доступу на всі встановлені біти.

Лістинг 7.5. Алгоритм перевірки рівня довіри процесу

```
function Test-ProcessTrustLevel {
param($Context)
        $trust_level = Get-NtTokenSid
        $Token -TrustLevel
        if ($null -eq $trust_level) {
           $trust_level = Get-NtSid -TrustType None -TrustLevel None
   }

        $access = Get-NtAccessMask 0xFFFFFFFF
        $sacl = Get-NtSecurityDescriptorSacl
        $Context.SecurityDescriptor
                foreach($ace in $sacl) {
                if (!$ace.IsProcessTrustLabelAce -or $ace.IsInheritOnly) {
                continue
                }
        if (!(Compare-NtSid $trust_level $ace.Sid -Dominates)) {
        $access = Get-NtAccessMask $ace
        }
        break
        }
        $access = Grant-NtAccessMask $access AccessSystemSecurity
        return Test-NtAccessMask $access $Context.RemainingAccess -All
}
```

Потім ми перераховуємо значення в SACL, перевіряючи будь-яку мітку довіри процесу ACE, крім *InheritOnly*. Коли ми знаходимо відповідну ACE, ми порівнюємо її SID з SID, який запитується для токена. Якщо ACE SID домінує, то токен має нижчий рівень захисту, і маска доступу встановлюється на значення з ACE.

Нарешті, ми порівнюємо маску доступу з рештою доступу, який запросив виклик. Якщо всі біти в масці доступу присутні, то функція повертає True, що вказує на успішну перевірку рівня довіри процесу. Зверніть увагу, що перевірка завжди додає *AccessSystemSecurity*, незалежно від маски в ACE.

Наконец, мы сравниваем маску доступа с оставшимся доступом, запрошенным вызывающей стороной. Если все биты в маске доступа присутствуют в оставшемся доступе, то функция возвращает True, что указывает на то, что проверка уровня доверия процесса прошла успешно. Обратите внимание, что проверка всегда добавляет AccessSystemSecurity, независимо от маски в ACE.

Давайте перевіримо поведінку ACE мітки довіри процесу. Замість того, щоб створювати новий захищений процес, ми будемо використовувати для перевірки доступу SID рівня довіри процесу токена анонімного користувача. Для спрощення тестування ми визначимо допоміжну функцію, яку можна використовувати повторно. Ця функція в лістингу 7.6 створить дескриптор безпеки за замовчуванням, який надає доступ як поточному користувачеві, так і анонімному користувачеві. Всякий раз, коли нам потрібен дескриптор безпеки

для тестування, ми можемо викликати цю функцію і використовувати повернуте значення.

Функція *New-BaseSD* створює базовий дескриптор безпеки, в якому власником і групою є користувач SYSTEM. Потім вона додає ACE *Allowed* для анонімних і поточних SID користувачів, надаючи їм повний доступ.

Лістинг 7.6. Визначення допоміжної функції для тестування

```
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
}
```

Також встановлює обов'язкову мітку на рівень цілісності *Untrusted*. Нарешті, вона відображає будь-який загальний доступ, специфічним для типу Mutant.

Давайте тепер протестуємо мітку довіри процесу, як показано в лістингу 7.7.

Лістинг 7.7. Тестування мітки довіри процесу

```
PS> $sd = New-BaseSD
PS> $trust_sid = Get-NtSid -TrustType ProtectedLight -TrustLevel Windows
PS> Add-NtSecurityDescriptorAce $sd -Type ProcessTrustLabel
-Access ModifyState -Sid $trust_sid
PS> Get-NtGrantedAccess $sd -AsString
ModifyState
PS> $token = Get-NtToken -Anonymous
PS> $anon_trust_sid = Get-NtTokenSid -Token $token -TrustLevel
PS> Compare-NtSid $anon_trust_sid $trust_sid -Dominates
True
PS> Get-NtGrantedAccess $sd -Token $token -AsString
Full Access
```

Спочатку ми створюємо базовий дескриптор безпеки та додаємо мітку довіри процесу, надаючи доступ *ModifyState* тільки тим токенам, рівень довіри процесу яких не домінує над міткою довіри процесу. Коли ми проводимо перевірку доступу, ми бачимо, що *ефективний* токен, який не має рівня довіри процесу, отримує тільки доступ *ModifyState*, що вказує на те, що мітка довіри процесу застосовується.

Далі ми отримуємо дескриптор токена анонімного користувача за допомогою *Get-NtToken*, запитуємо його SID рівня довіри процесу і порівнюємо його з SID, який ми додали до дескриптора безпеки. Виклик *Compare-NtSid* повертає *True*, що вказує на те, що SID рівня довіри процесу токена домінує над SID в дескрипторі безпеки. Щоб підтвердити це, ми перевіряємо доступ і виявляємо, що токен анонімного користувача має повний доступ, що означає, що мітка довіри процесу не обмежувала його доступ.

Ви можете запитати, чи можна підробити анонімний токен, щоб обійти мітку довіри процесу. Пам'ятайте, що в режимі користувача ми викликаємо *NtAccessCheck*, який приймає тільки один дескриптор токена, але *SeAccessCheck* ядра приймає як основний токен, так і токен імперсонації. Перш ніж ядро перевіряє мітку довіри процесу, воно перевіряє обидва токени і вибирає той, який має нижчий рівень довіри. Тому, якщо токен імперсонації є довіреним, але ваш основний токен є недовіреним, ефективний рівень довіри буде недовіреним.

Windows застосовує додаткову перевірку безпеки під час присвоєння процесу мітки довіри ACE ресурсу. Хоча для встановлення мітки довіри процесу потрібно лише право доступу *WriteDac*, ви не можете змінити або видалити ACE, якщо ваш ефективний рівень довіри не перевищує рівень довіри мітки. Це запобігає встановленню нової довільної мітки довіри процесу ACE. Microsoft використовує цю можливість для перевірки певних файлів, пов'язаних із програмами Windows, на наявність змін та підтвердження того, що файли були створені захищеним процесом.

7.2.4. Фільтр доступу ACE

Другою обов'язковою перевіркою доступу є фільтр доступу ACE. Він працює аналогічно до мітки довіри процесу ACE, за винятком того, що замість використання рівня довіри процесу для визначення, чи застосовувати маску обмеження доступу, він використовує умовний вираз, який приймає значення *True* або *False*. Якщо умовний вираз приймає значення *False*, маска доступу ACE обмежує максимальний доступ, наданий для перевірки доступу. Якщо він приймає значення *True*, фільтр доступу ігнорується.

У SACL можна мати кілька ACE фільтрів доступу. Кожне умовне вираження, яке оцінюється як *False*, видаляє частину маски доступу. Тому, якщо збігається один ACE, але не збігається другий ACE, який обмежує до *GenericRead*, ви отримаєте максимальний доступ до *GenericRead*. Ми можемо відобразити цю логіку у функції PowerShell, як показано в лістингу 7.8.

Лістинг 7.8. Алгоритм перевірки фільтра доступу

```
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
                }
                if (!(Test-NtAceCondition $ace -Token $token)) {
                $access = $access -band $ace.Mask
                }
        }
        $access = Grant-NtAccessMask
        $access AccessSystemSecurity
return Test-NtAccessMask $access $Context.RemainingAccess -All
}
```

Цей алгоритм подібний до того, який ми реалізували для перевірки рівня довіри процесу. Єдина відмінність полягає в тому, що ми перевіряємо умовний вираз, а не SID. Функція підтримує кілька ACE фільтрів доступу. Для кожного відповідного ACE маска доступу побітно поєднується з кінцевою маскою доступу. Оскільки маски поєднуються, кожен ACE може тільки видалити доступ, а не додати його. Після перевірки всіх ACE ми перевіряємо доступ, що залишився, щоб визначити, чи перевірка пройшла успішно чи ні.

У лістингу 7.9 ми перевіряємо поведінку алгоритму фільтра доступу, щоб переконатися, що він працює як і очікувалося.

Ми доповнюємо дескриптор безпеки фільтром доступу ACE з умовним виразом «Exists TSA://ProcUnique». Цей вираз перевіряє, чи присутній у токені атрибут безпеки TSA://ProcUnique. Для звичайного користувача ця перевірка завжди повинна повертати значення *True*, однак у токені анонімного користувача цей атрибут відсутній. Ми встановлюємо маску *ModifyState* і SID для групи *Everyone*. Зверніть увагу, що SID не перевіряється, тому він може мати будь-яке значення, але використання групи *Everyone* є загальноприйнятим.

Ми можемо перевірити поточні атрибути безпеки *ефективного* токена за допомогою *Show-NtTokenEffective*. Отримання максимального доступу для ефективного токена призводить до повного доступу, що означає, що перевірка фільтра доступу проходить без обмеження. Однак, коли ми повторюємо це, використовуючи токен анонімного користувача, перевірка фільтра доступу не проходить, і доступ обмежується лише *ModifyState*.

Лістинг 7.9. Тестування фільтра доступу ACE

```
PS> $sd = New-BaseSD
PS> Add-NtSecurityDescriptorAce $sd -Type AccessFilter -KnownSid World
-Access ModifyState -Condition "Exists TSA://ProcUnique" -MapGeneric
PS> Format-NtSecurityDescriptor $sd -Summary -SecurityInformation AccessFilter
<Access Filters>
Everyone: (AccessFilter)(None)(ModifyState)(Exists TSA://ProcUnique)
PS> Show-NtTokenEffective -SecurityAttributes
SECURITY ATTRIBUTES
-------------------
Name                Flags                    ValueType Values
 ----                -----                    --------- ------
TSA://ProcUnique   NonInheritable,       Unique UInt64    {187, 365588953}

PS> Get-NtGrantedAccess $sd -AsString
Full Access

PS> Use-NtObject($token = Get-NtToken -Anonymous) {
Get-NtGrantedAccess $sd -Token $token -AsString }
ModifyState
```

Щоб встановити фільтр доступу, вам потрібен лише доступ *WriteDac*. То що ж заважає користувачеві видалити фільтр? Очевидно, що фільтр доступу не повинен надавати доступ *WriteDac*, але якщо це все-таки сталося, ви можете обмежити будь-які зміни до рівня довіри захищеного процесу. Для цього встановіть ACE SID на рівень довіри процесу SID і встановіть прапорець

TrustProtected ACE. Тепер абонент з нижчим рівнем довіри процесу не зможе видалити або змінити фільтр доступу ACE.

7.2.5. Обов'язкова перевірка рівня цілісності

Нарешті, ми реалізуємо обов'язкову перевірку рівня цілісності. У SACL обов'язковий SID мітки ACE представляє рівень цілісності дескриптора безпеки. Його маска, яка виражає обов'язкову політику, поєднує політики *NoReadUp, NoWriteUp* і *NoExecuteUp* для визначення максимального доступу, який система може надати викликаючому на основі значень *GenericRead, GenericWrite* і *GenericExecute* із загальної структури відображення.

Щоб визначити, чи застосовувати цю політику, виконується порівняння SID рівнів цілісності дескриптора безпеки та токена. Якщо SID токена домінує над SID дескриптора безпеки, політика не застосовується і будь-який доступ дозволяється. Однак, якщо SID токена не домінує, будь-який запит на доступ, що виходить за межі значення політики, призводить до невдачі перевірки доступу з кодом STATUS_ACCESS_DENIED.

Обчислення того, чи один SID рівня цілісності домінує над іншим, набагато простіше, ніж обчислення еквівалентного значення для SID рівня довіри процесу.

Для цього ми витягуємо останній RID з кожного SID і порівнюємо їх як числа. Якщо RID одного SID рівня цілісності більший або дорівнює іншому, він домінує.

Однак обчислення маски доступу для політики на основі загального відображення є набагато складнішим, оскільки вимагає врахування спільних прав доступу. Ми не будемо реалізовувати код для обчислення маски доступу, оскільки можемо скористатися параметрами Get-*NtAccessMask*, яка обчислює її за нас.

Почнемо з перевірки обов'язкової політики токена. У нашому випадку перевіряємо, чи встановлено прапор *NoWriteUp*. Якщо прапор не встановлено, вимикаємо перевірку рівня цілісності для цього токена і повертаємо *True*. Однак цей прапор майже ніколи не вимикається, тому що для цього потрібні права *SeTcbPrivilege*. Тому у більшості випадків перевірка рівня цілісності продовжується.

Далі нам потрібно отримати рівень цілісності дескриптора безпеки та обов'язкову політику з обов'язкової мітки ACE. Якщо ACE існує, витягуємо ці значення та відображаємо політику на маску максимального доступу за допомогою Get-*NtAccessMask*. Якщо ACE не існує, алгоритм за замовчуванням використовує середній рівень цілісності та політику *NoWriteUp*.

Якщо токен має привілей *SeRelabelPrivilege*, ми додаємо доступ WriteOwner до максимального доступу, навіть якщо політика його видалила. Це дозволяє виклику з увімкненим *SeRelabelPrivilege* змінювати обов'язкову мітку цілісності ACE дескриптора безпеки.

Потім ми запитуємо SID рівня цілісності токена і порівнюємо його з дескриптором безпеки. Якщо SID токена домінує, перевірка проходить успішно і дозволяється будь-який доступ. Зверніть увагу, що ми не розглядаємо AccessSystemSecurity інакше, ніж у процесі перевірки рівня довіри та фільтра доступу. Ми видаляємо його, якщо політика містить NoWriteUp, що є типовим значенням для всіх типів ресурсів.

Лістинг 7.10. Реалізація обов'язкової перевірки рівня цілісності

```
function Test-MandatoryIntegrityLevel {
        param($Context)
        $token = $Context.Token
        $sd = $Context.SecurityDescriptor
        $mapping = $Context.GenericMapping
         $policy = Get-NtTokenMandatoryPolicy -Token $token
        if (($policy -band "NoWriteUp") -eq 0) {
        return $true
        }
        if ($sd.HasMandatoryLabelAce) {
                $ace = $sd.GetMandatoryLabel()
                $sd_il_sid = $ace.Sid
                $access = Get-NtAccessMask $ace.Mask -GenericMapping $mapping
        } else {
                $sd_il_sid = Get-NtSid -IntegrityLevel Medium
                $access = Get-NtAccessMask -MandatoryLabelPolicy NoWriteUp
                -GenericMapping $GenericMapping
        }
        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
}
```

Давайте перевіримо поведінку обов'язкової перевірки рівня цілісності в процесі перевірки доступу (лістинг 7.11).

Спочатку ми створюємо дескриптор безпеки і перевіряємо його обов'язкову мітку цілісності. Ми бачимо, що вона встановлена на рівень цілісності *Untrusted*, який є найнижчим рівнем, і відповідна політика - *NoWriteUp*. Потім отримуємо максимальний доступ для токена *анонімного* користувача, який, як ми бачимо, має рівень цілісності *Untrusted*. Оскільки цей рівень цілісності відповідає рівню цілісності дескриптора безпеки, токен отримує повний доступ.

Щоб перевірити обмеження маски доступу, ми видаляємо обов'язкову мітку ACE з дескриптора безпеки, щоб перевірка доступу за замовчуванням відповідала рівню цілісності *Medium*. Виконавши перевірку ще раз, ми отримуємо ModifyState|ReadControl|Synchronize, що є повним доступом до об'єкта **Mutant** без маски доступу *GenericWrite*.

Лістинг 7.11. Тестування обов'язкової мітки ACE

```
PS> $sd = New-BaseSD
PS> Format-NtSecurityDescriptor
$sd -SecurityInformation Label -Summary
<Mandatory Label>
 Mandatory Label\Untrusted Mandatory Level: (MandatoryLabel)(None)(NoWriteUp)
PS> Use-NtObject($token = Get-NtToken -Anonymous) {
            Format-NtToken $token -Integrity
            Get-NtGrantedAccess $sd -Token $token -AsString
            }
INTEGRITY LEVEL
---------------
Untrusted Full Access
PS> Remove-NtSecurityDescriptorIntegrityLevel $sd
PS> Use-NtObject($token = Get-NtToken -Anonymous) {
      Get-NtGrantedAccess $sd -Token $token -AsString
}
ModifyState|ReadControl|Synchronize
```

На цьому завершується реалізація обов'язкової перевірки доступу.

Як ми бачимо, що цей алгоритм насправді складається з трьох окремих перевірок: рівня довіри процесу, фільтра доступу та рівня цілісності. Кожна перевірка може лише **відмовити в доступі**, але ніколи не надає додаткового доступу.

7.2.6. Перевірка доступу до токена

Друга основна перевірка, перевірка токена доступу, використовує властивості токена викликаючого, щоб визначити, чи надавати певні права доступу. Більш конкретно, вона перевіряє наявність будь-яких спеціальних привілеїв, а також власника дескриптора безпеки.

На відміну від обов'язкової перевірки доступу, перевірка токена доступу може надати доступ до ресурсу, якщо він видалив усі біти з маски доступу токена.

Лістинг 7.12 реалізує функцію Result-*TokenAccess* верхнього рівня.

Лістинг 7.12. Алгоритм перевірки токену доступу

```
Function Result-TokenAccess {
            param($Context)

            Resolve-TokenPrivilegeAccess $Context
            if (Test-NtAccessMask $Context.RemainingAccess -Empty) {
            return
            }
            return Resolve-TokenOwnerAccess $Context
}
```

Перевірка є простою. Спочатку перевіряємо привілеї токена за допомогою функції, яку ми визначимо далі, Resolve-*TokenPrivilegeAccess*, передаючи їй поточний контекст. Якщо певні привілеї увімкнені, ця функція змінює поточний доступ токена. Якщо поточний доступ порожній, тобто доступ не надається, ми можемо негайно повернутися. Потім ми викликаємо *Resolve-Token OwnerAccess*, яка перевіряє, чи належить токену ресурс, а також має можливість оновлювати *RemainingAccess*. Давайте розглянемо ці перевіри окремо.

7.2.7. Перевірка привілеїв

Перевірка привілеїв (лістинг 7.13) визначає, чи має об'єкт Token три різні привілеї, що ввімкнені. Для кожного з них, якщо привілей увімкнена, ми надаємо маску доступу та біти з решти доступу.

Спочатку ми перевіряємо, чи викликаючий об'єкт запросив *AccessSystemSecurity*. Якщо так, і якщо *SeSecurityPrivilege* увімкнена, ми видаляємо *AccessSystemSecurity* із решти доступу. Ми також оновлюємо список привілеїв, які ми використовували, щоб ми могли повернути його викликаючому об'єкту.

Далі ми виконуємо подібні перевірки для *SeTakeOwnershipPrivilege* та *SeRelabelPrivilege* і видаляємо *WriteOwner* із залишкового доступу, якщо вони активовані. І нарешті, ми оновлюємо значення RemainingAccess із остаточною маскою доступу.

Лістинг 7.13. Алгоритм перевірки доступу до привілеїв токена

```
function Resolve-TokenPrivilegeAccess {
        param($Context)
        $token = $Context.Token
        $access = $Context.RemainingAccess

        if ((Test-NtAccessMask $access AccessSystemSecurity) -and
                (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"
        }

        $Context.RemainingAccess = $access
        }
```

Надання *WriteOwner* доступу до *SeTakeOwnershipPrivilege* і *SeRelabelPrivilege* має сенс з точки зору ядра, оскільки доступ *WriteOwner* необхідний для зміни SID власника і рівня цілісності. Однак така реалізація також означає, що токен, який має тільки *SeRelabelPrivilege*, може отримати право власності на ресурс, що не завжди є бажаним.

На щастя, навіть адміністратори не отримують *SeRelabelPrivilege* за замовчуванням, тому це не є серйозною проблемою.

Давайте перевіримо цю функцію на відповідність реальному процесу перевірки доступу. Запустіть скрипт у лістингу 7.14 з правами адміністратора.

Починаємо з створення дескриптора безпеки, який не повинен надавати доступ поточному користувачеві. Потім увімкнемо SeTakeOwnershipPrivilege.

Далі запитуємо перевірку доступу для WriteOwner і вказуємо параметр PassResult, який виводить повний результат перевірки доступу.

Результат показує, що перевірка доступу пройшла успішно, надавши доступ WriteOwner, але також, що перевірка використовувала SeTakeOwnershipPrivilege.

Лістинг 7.14. Тестування перевірки привілеїв токена

```
PS> $owner = Get-NtSid -KnownSid Null
PS> $sd = New-NtSecurityDescriptor -Type Mutant -Owner $owner -Group $owner -EmptyDacl
PS> Enable-NtTokenPrivilege SeTakeOwnershipPrivilege
PS> Get-NtGrantedAccess $sd -Access WriteOwner -PassResult
Status                  Granted Access      Privileges
------                  --------------      ----------
STATUS_SUCCESS          WriteOwner          SeTakeOwnershipPrivilege
PS> Disable-NtTokenPrivilege SeTakeOwnershipPrivilege
PS> Get-NtGrantedAccess $sd -Access WriteOwner -PassResult
Status                  Granted Access      Privileges
------                  --------------      ----------
STATUS_ACCESS_DENIED    None                NONE
```

Щоб переконатися, що нам не було надано доступ *WriteOwner* з іншої причини, вимикаємо привілей і повторно запускаємо перевірку. Цього разу нам відмовляють у доступі.

7.2.8. Перевірка власника

Перевірка власника існує для надання доступу *ReadControl* і *WriteDac* власнику ресурсу, навіть якщо DACL не надає цьому власнику жодного іншого доступу.

Мета цієї перевірки — запобігти блокуванню користувачем власних ресурсів. Якщо користувач випадково змінить DACL так, що більше не матиме доступу, він все одно зможе скористатися доступом *WriteDac*, щоб повернути DACL до попереднього стану.

Перевірка порівнює SID власника в дескрипторі безпеки з усіма увімкненими групами токенів (а не тільки з токеном власника), і якщо знаходить збіг, надає доступ. Таку поведінку ми демонстрували на початку цього розділу, в лістингу 7.1. У лістингу 7.15 реалізуємо функцію *Resolve-TokenOwnerAccess*.

Ми використовуємо *Test-NtTokenGroup*, щоб перевірити, чи є SID власника дескриптора безпеки активним елементом токена.

Лістинг 7.15. Алгоритм перевірки доступу токена власника

```
function Resolve-TokenOwnerAccess {
        param($Context)
        $token = $Context.Token
        $sd = $Context.SecurityDescriptor
        $sd_owner = Get-NtSecurityDescriptorOwner $sd
         if (!(Test-NtTokenGroup -Token $token -Sid $sd_owner.Sid)) {
                return
        }
        $sids = Select-NtSecurityDescriptorAce $sd
        -KnownSid OwnerRights -First -AclType Dacl
        if ($sids . Count -gt 0) {
                return
        }
        $access = $Context.RemainingAccess
        $Context.RemainingAccess = Revoke-NtAccessMask $access ReadControl,
WriteDac
}
```

Якщо SID власника не є елементом, ми просто повертаємося. Якщо він є елементом, код повинен перевірити, чи є в DACL будь-які SID з правами власника (S-1-3-4). Якщо такі є, не дотримуємося стандартного процесу, а покладаємося на перевірку DACL, щоб надати доступ власнику. Нарешті, якщо обидві перевірки пройшли успішно, ми можемо видалити *ReadControl* і *WriteDac* з решти доступу.

У лістингу 7.16 підтверджуємо цю поведінку в реальному процесі перевірки доступу.

Лістинг 7.16. Тестування токена власника

```
PS> $owner = Get-NtSid -KnownSid World
PS> $sd = New-NtSecurityDescriptor -Owner $owner -Group $owner -Type Mutant -EmptyDacl
PS> Get-NtGrantedAccess $sd
ReadControl, WriteDac
PS> Add-NtSecurityDescriptorAce $sd -KnownSid OwnerRights -Access ModifyState
PS> Get-NtGrantedAccess $sd
ModifyState
```

Спочатку створюємо дескриптор безпеки, в якому власником і групою є *Everyone*. Також створюємо дескриптор безпеки з порожнім DACL, що означає, що процес перевірки доступу враховуватиме лише перевірку власника під час обчислення наданого доступу. Коли ми виконуємо перевірку доступу, ми отримуємо *ReadControl* і *WriteDac*.

Потім додаємо один ACE з OWNER RIGHTS SID. Це вимикає доступ власника за замовчуванням і змушує перевірку доступу надавати тільки доступ, зазначений в ACE (в даному випадку *ModifyState*). Коли ми знову запускаємо перевірку доступу, ми бачимо, що єдиним наданим доступом є *ModifyState* і що ми більше не маємо доступу *ReadControl* або *WriteDac*.

На цьому перевірка доступу токена завершена. Як показано вище, алгоритм може надати певні права доступу виклику перед тим, як відбудеться будь-яка значна обробка дескриптора безпеки. Це робиться в першу чергу для того, щоб користувачі могли зберегти доступ до своїх ресурсів, а адміністратори могли отримати права власності на файли інших користувачів. Тепер перейдемо до остаточної перевірки.

7.2.9. Перевірка довільного доступу

У декількох наших тестах ми покладалися на поведінку DACL. Тепер ми докладно розглянемо, як саме працює перевірка DACL. Перевірка DACL може здаватися простою, але складнощі приховані в деталях. Алгоритм реалізовано в лістингу 7.17.

Спочатку ми перевіряємо, чи присутній DACL. Якщо так, то перевіряємо, чи це NULL ACL. Якщо DACL відсутній або є тільки NULL ACL, то безпека не застосовується, тому функція очищає залишковий доступ і повертається, надаючи токену будь-який доступ до ресурсу, який не обмежений обов'язковою перевіркою доступу.

Після підтвердження наявності DACL для перевірки ми можемо перерахувати кожний з його ACE. Якщо ACE є *InheritOnly*, він не братиме участі в перевірці, тому ми його ігноруємо. Далі нам потрібно зіставити SID в ACE з SID, який ми перевіряємо, за допомогою допоміжної функції, яку ми визначимо далі, Get-*AceSid*.

Лістинг 7.17. Алгоритм перевірки дискреційного доступу

```
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
	}
	if (!(Test-NtSecurityDescriptor $sd -DaclPresent) -or (Test-NtSecurityDescriptor $sd -DaclNull)) {
		$Context.RemainingAccess = Get-NtAccessMask 0
			return
	}
	$owner = Get-NtSecurityDescriptorOwner $sd
	$dacl = Get-NtSecurityDescriptorDacl $sd
	foreach($ace in $dacl) {
		if ($ace.IsInheritOnly) {
		continue
		}
		$sid = Get-AceSid $ace -Owner $owner
		$continue_check = $true
		switch($ace.Type) {
		"Allowed" {
			 if (Test-NtTokenGroup -Token $token $sid) {
			$access = Revoke-NtAccessMask $access $ace.Mask
			}
		}
		"Denied" {
			if (Test-NtTokenGroup -Token $token $sid -DenyOnly) {
				if (Test-NtAccessMask $access $ace.Mask) {
				$continue_check = $false
				}
			}
		}
		AllowedCompound" {
			$server_sid = Get-AceSid $ace -Owner $owner
				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 -
			ResourceAttributes $resource_attrs)) {
			$access = Revoke-NtAccessMask $access $ace.Mask
			}
		}
	}
		if (!$continue_check -or (Test-NtAccessMask $access -Empty)) {
			break
			}
		}
		$Context.RemainingAccess = $access
}
```

Ця функція перетворює OWNER RIGHTS SID для ACE на власника поточного дескриптора безпеки, як показано в лістингу 7.18.

Маючи SID, ми можемо оцінити кожен ACE на основі його типу.

Для найпростішого типу, Allowed, ми перевіряємо, чи є SID у групах Enabled токена. Якщо так, ми надаємо доступ, представлений маскою ACE, і можемо видалити ці біти з решти доступу.

Лістинг 7.18. Реалізація Get-AceSid

```
function Get-AceSid {
        param(
                $Ace,
                $Owner
        )
        $sid = $Ace.Sid
                if (Compare-NtSid $sid -KnownSid OwnerRights) {
                        $sid = $Owner.Sid
         }
        return $sid
}
```

Для типу *Denied* ми також перевіряємо, чи є SID у групах токена. Однак ця перевірка повинна включати групи *Enabled* і *DenyOnly*, тому ми передаємо параметр *DenyOnly*. Зверніть увагу, що можна також налаштувати SID користувача токена як групу *DenyOnly*, функція *Test-NtTokenGroup* враховує це. ACE типу *Denied* не змінює доступ, що залишився. Натомість функція порівнює маску з поточним доступом, що залишився, і якщо будь-який біт доступу, що залишився, також встановлений у масці, функція відмовляє в цьому доступі та негайно повертає доступ, що залишився.

Останні два типи ACE, які ми розглянемо, є варіаціями типу *Allowed*.

Перший, *AllowedCompound*, містить додатковий SID сервера. Для виконання цієї перевірки функція порівнює як звичайний SID, так і SID сервера з групами токена виклику, оскільки ці значення можуть відрізнятися. (Зверніть увагу, що SID сервера повинен бути пов'язаний з власником, якщо використовується SID OWNER RIGHTS). Умова ACE виконується тільки в тому випадку, якщо обидва SID увімкнені.

Нарешті, ми перевіряємо тип ACE *AllowedCallback*. Для цього ми знову перевіряємо SID, а також те, чи відповідає умовний вираз токену за допомогою *Test-NtAceCondition*. Якщо вираз повертає *True*, умова ACE виконується, і ми видаляємо маску з решти доступу. Щоб повністю реалізувати умовну перевірку, нам також потрібно передати будь-які атрибути ресурсу з дескриптора безпеки.

Зверніть увагу, що ми навмисно не перевіряємо *DenyCallback*. Це пов'язано з тим, що ядро не підтримує ACE *DenyCallback*, хоча API *AuthzAccessCheck*, що працює тільки в режимі користувача, підтримує.

7.3. ПІСОЧНИЦЯ

У розділі 4 ми розглянули два типи токенів пісочниці: обмежені та низькорівневі. Ці токени пісочниці модифікують процес перевірки доступу,

додаючи більше перевірок. Давайте обговоримо кожен тип токенів більш детально, починаючи з обмежених токенів.

7.3.1. Обмежені токени

Використання обмеженого токена впливає на процес перевірки доступу, вводячи другого власника і перевірку дискреційного доступу до списку обмежених SID. У лістингу 7.19 ми модифікуємо перевірку SID власника у функції *Resolve-TokenOwnerAccess*, щоб врахувати це.

Лістинг 7.19. Модифікована перевірка доступу Get-TokenOwner для обмежених токенів

```
if (!(Test-NtTokenGroup -Token $token -Sid $sd_owner.Sid)) {
        return
}
if ($token.Restricted -and !(Test-NtTokenGroup -Token $token -Sid $sd_owner.Sid -Restricted)) {
        return
}
```

Спочатку ми виконуємо перевірку існуючого SID. Якщо SID власника відсутній у списку токенів груп, ми не надаємо доступ *ReadControl* або *WriteDac*. Далі виконується додаткова перевірка: якщо токен обмежений, ми перевіряємо список обмежених SID на наявність SID власника і надаємо токену доступ *ReadControl* і *WriteDac* тільки в тому випадку, якщо SID власника присутній як у основному списку груп, так і в списку обмежених SID.

Ми будемо дотримуватися того ж самого шаблону для перевірки дискреційного доступу, хоча для простоти ми додаємо параметр *Boolean Restricted* switch до функції *Get-DiscretionaryAccess* і передаємо його до будь-якого виклику *Test-NtTokenGroup*. Наприклад, ми можемо змінити перевірку дозволеного ACE, реалізовану в лістинг 7.17, так, щоб вона виглядала, як показано в лістинг 7.20.

Лістинг 7.20. Модифікований тип ACE *Allowed* для обмежених токенів

```
"Allowed" {
        if (Test-NtTokenGroup -Token $token $sid -Restricted:$Restricted) {
                $access = Revoke-NtAccessMask $access $ace.Mask
        }
}
```

У лістингу 7.20 ми встановлюємо параметр Restricted на значення параметра, переданого в *Get-DiscretionaryAccess*. Тепер нам потрібно змінити функцію *Get-PSGrantedAccess*, визначену в лістингу 7.2, щоб двічі викликати Get-*DiscretionaryAccess* для обмеженого токена (лістинг 7.21).

Спочатку ми фіксуємо існуюче значення RemainingAccess, оскільки перевірка дискреційного доступу змінить його, а ми хочемо повторити цю перевірку вдруге. Потім виконуємо перевірку дискреційного доступу і зберігаємо результат у змінній. Якщо ця перша перевірка пройшла успішно і токен обмежений, ми повинні виконати другу перевірку. Ми також повинні врахувати, чи токен обмежений для запису і чи включає залишковий доступ для запису.

Лістинг 7.21. Функція Get-*PSGrantedAccess*, модифікована для урахування обмежених токенів

```
$RemainingAccess = $Context.RemainingAccess
Get-DiscretionaryAccess $Context
$success = Test-NtAccessMask $Context.RemainingAccess -Empty
if ($success -and $Token.Restricted) {
            if (!$Token.WriteRestricted -or
            (Test-NtAccessMask $RemainingAccess -WriteRestricted $GenericMapping)) {
             $Context.RemainingAccess = $RemainingAccess
            Get-DiscretionaryAccess $Context -Restricted
            $success = Test-NtAccessMask $Context.RemainingAccess -Empty
            }
}
            if ($success) {
            return Get-AccessResult STATUS_SUCCESS $Context.Privileges $DesiredAccess
            }
return Get-AccessResult STATUS_ACCESS_DENIED
```

Далі ми знову виконуємо перевірку, цього разу з параметром *Restricted*, щоб вказати, що слід перевірити обмежені SID. 5. Якщо ця друга перевірка також проходить успішно, ми встановлюємо змінну *$success* на *True* і надаємо доступ до ресурсу.

Майте на увазі, що перевірка обмежених SID застосовується як до типів ACE *Allowed*, так і до *Denied*. Це означає, що якщо DACL містить *Denied* ACE, який посилається на SID у списку обмежених SID, функція відмовить у доступі, навіть якщо SID не знаходиться у звичайному списку груп.

7.3.2. Токени Lowbox

Процес перевірки доступу для токена **lowbox** схожий на процес перевірки для токена з обмеженим доступом. Токен **lowbox** може містити список SID, що використовуються для виконання другої перевірки, подібної до тієї, яку ми виконували зі списком SID з обмеженим доступом.

Аналогічно, якщо процес перевірки доступу не надає доступу як за допомогою звичайної перевірки, так і за допомогою перевірки можливостей, перевірка доступу завершується невдачею. Однак перевірка доступу токена **lowbox** має деякі тонкі відмінності:

—Крім списку SID можливостей, буде враховуватися пакет SID токена.

—Перевірені SID можливостей повинні мати прапорець атрибуту *enabled*, щоб вважатися активними.

—Перевірка застосовується тільки до типів ACE Allowed, а не до типів ACE Denied.

—NULL DACL не надають повного доступу.

Крім того, два спеціальні SID пакета будуть відповідати будь-якому SID пакета токена для цілей перевірки SID:

• ALL APPLICATION PACKAGES (S-1-15-2-1);

• ALL RESTRICTED APPLICATION PACKAGES (S-1-15-2-2).

Перевірка ALL APPLICATION PACKAGES SID під час перевірки SID пакета може бути вимкнена, якщо токен, який використовується для перевірки доступу, має атрибут безпеки WIN://NOALLAPPPKG, встановлений на єдине значення 1. У цьому випадку перевірка SID пакета враховуватиме лише ALL RESTRICTED APPLICATION PACKAGES SID. Якщо атрибут безпеки відсутній або встановлений на 0, перевірка доступу враховує обидва спеціальні пакети SID. Microsoft називає процеси з цим атрибутом безпеки процесами, що виконуються в *AppContainer з обмеженими правами* (*Less Privileged AppContainer*, LPAC).

Оскільки для встановлення атрибуту безпеки токена потрібні привілеї *SeTcbPrivilege,* API-інтерфейси створення процесів мають опцію додавання атрибуту безпеки WIN:// NOALLAPPPKG до токена нового процесу. Лістинг 7.22 показує базову реалізацію перевірки доступу **lowbox** для типів ACE *Allowed*. Ви повинні додати цей код до перевірки дискреційного доступу в лістингу 7.17, у місцях, вказаних у коментарях.

Лістинг 7.22. Реалізація перевірки доступу до **lowbox** для *Allowed* ACE

```
## Add to start of Get-DiscretionaryAccess.
$ac_access = $context.DesiredAccess if (!$token.AppContainer) {
$ac_access = Get-NtAccessMask 0
}
## Replace the Allowed case in the ACE switch statement.
"Allowed" {
        if (Test-NtTokenGroup -Token $token $sid -Restricted:$Restricted) {
                $access = Revoke-NtAccessMask $access $ace.Mask
        } else {
        if ($Restricted) {
                break
        }
        if (Test-NtTokenGroup -Token $token $sid -Capability) {
                 $ac_access = Revoke-NtAccessMask $ac_access $ace.Mask
                }
        }
}
# Add at end of ACE loop.
$effective_access = $access -bor $ac_access
```

Перший тест перевіряє, чи є SID у списку груп токена. Якщо він знаходить SID у списку груп, тест видаляє маску з решти перевірок доступу. Якщо тест групи не знаходить, перевіряємо, чи це пакет або SID можливостей. Повинні переконатися, що не перевіряємо, чи перебуваємо в режимі обмеженого SID, оскільки цей режим не передбачає перевірки **lowbox**.

Перевірка SID-кодів можливостей включає SID-код пакета та SID-код ALL APPLICATION PACKAGES. Якщо знаходимо збіг, видаляємо маску з решти доступу. Однак нам потрібно зберегти окремі значення решти доступу для звичайних SID-кодів та SID-кодів *AppContainer*.

Тому ми створюємо дві змінні, *$access* та *$ac_access*. Ініціалізуємо змінну *$ac_access* значенням оригінального *DesiredAccess*, а не поточного залишкового доступу. Оскільки ми не надаємо права власника, такі як *WriteDac*, якщо SID також не відповідає пакету *Allowed* або SID ACE можливості. Ми також

змінюємо умову виходу з циклу, щоб врахувати обидва залишкові значення доступу. Вони повинні бути порожніми, перш ніж ми вийдемо.

Далі ми зробимо кілька додаткових перевірок, щоб краще ізолювати процеси *AppContainer* від існуючих пісочниць з низьким рівнем цілісності, таких як захищений режим Internet Explorer. Перша зміна, яку ми впроваджуємо, стосується обов'язкової перевірки доступу. Якщо перевірка токена **lowbox** не проходить, ми перевіряємо рівень цілісності дескриптора безпеки вдруге. Якщо рівень цілісності менше або дорівнює середньому, ми припускаємо, що перевірка пройшла успішно. Це відбувається навіть незважаючи на те, що токени **lowbox** мають низький рівень цілісності, що зазвичай перешкоджає доступу на запис до ресурсу.

Така поведінка дозволяє додатку з більшими привілеями надати токену **lowbox** доступ до ресурсу, блокуючи пісочниці з низьким рівнем цілісності.

Лістинг 7.23 демонструє таку поведінку.

Спочатку ми створюємо дескриптор безпеки, який надає доступ *GenericAll* для груп *Everyone* та ALL APPLICATION PACKAGES. Також встановлюємо явний рівень цілісності *Medium*, хоча це не є обов'язковим, оскільки *Medium* є типовим значенням для дескрипторів безпеки без обов'язкового мітки ACE.

Потім виконуємо перевірку доступу за допомогою токена з низьким рівнем цілісності і отримуємо тільки доступ для читання до дескриптора безпеки. Далі ми знову пробуємо перевірити доступ за допомогою токена **lowbox**. Хоча рівень цілісності токена все ще є низьким, токен отримує повний доступ.

Лістинг 7.23. Перевірка обов'язкового доступу до токена **lowbox**

```
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
PS> Set-NtSecurityDescriptorIntegrityLevel $sd Medium

PS> Use-NtObject($token = Get-NtToken -Duplicate -IntegrityLevel Low) { Get-NtGrantedAccess $sd -Token $token -
AsString }
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
}
Full Access
```

Друга зміна, яку ми впроваджуємо, стосується того, що якщо DACL містить SID пакета, ми відмовляємо в доступі токену з низьким рівнем цілісності, незалежно від рівня цілісності дескриптора безпеки або DACL. Цей механізм блокує доступ до ресурсів, яким призначено DACL за замовчуванням, оскільки SID пакета додається до DACL за замовчуванням під час створення токена lowbox. Лістинг 7.24 протестує цю поведінку.

Почнемо зі створення токена **lowbox**. Токен не має жодних додаткових SID можливостей, окрім SID пакета. Далі ми створюємо дескриптор безпеки за замовчуванням з токена lowbox. При перевірці записів у дескрипторі безпеки ми бачимо, що поточний SID користувача і SID пакета отримали повний доступ.

Оскільки токен **lowbox** має низький рівень цілісності, правила успадкування дескриптора безпеки вимагають додавання рівня цілісності до дескриптора.

Лістинг 7.24. Перевірка поведінки пакета SID для токенів з низьким рівнем цілісності

```
PS> $sid = Get-NtSid -PackageName 'package_sid_low_il_test'
PS> $token = Get-NtToken -LowBox -PackageSid $sid
PS> $sd = New-NtSecurityDescriptor -Token $token -Type Mutant
PS> Format-NtSecurityDescriptor $sd -Summary -SecurityInformation Dacl, Label
<DACL>
GRAPHITE\user: (Allowed)(None)(Full Access)
NT AUTHORITY\SYSTEM: (Allowed)(None)(Full Access)
NT AUTHORITY\LogonSessionId_0_109260: (Allowed)(None)(ModifyState|...)
package_sid_low_il_test: (Allowed)(None)(Full Access)
<Mandatory Label>
Mandatory Label\Low Mandatory Level: (MandatoryLabel)(None)(NoWriteUp)

PS> Get-NtGrantedAccess $sd -Token $token -AsString
Full Access

PS> $token.Close()
PS> $low_token = Get-NtToken -Duplicate -IntegrityLevel Low
PS> Get-NtGrantedAccess $sd -Token $low_token -AsString
None
```

Потім ми запитуємо наданий доступ для дескриптора безпеки на основі токена **lowbox** і отримуємо повний доступ. Далі створюємо дублікат поточного токена, але встановлюємо його рівень цілісності на низький. Тепер ми отримуємо наданий доступ *None*, хоча очікували отримати повний доступ на основі рівня цілісності ACE в дескрипторі безпеки. У цьому випадку наявність пакета SID в дескрипторі безпеки заблокувала доступ.

Останнє, на що слід звернути увагу: оскільки перевірки доступу до пісочниці є *ортогональними*, можна створити токен **lowbox** з обмеженого токена, що призведе до виконання як перевірок **lowbox**, так і перевірок обмеженого SID. Результуючий доступ є найбільш обмежувальним з усіх, що робить пісочницю більш надійною.

7.4. КОНТРОЛЬ ДОСТУПУ ДО КОРПОРАТИВНИХ РЕСУРСІВ

У корпоративних версіях Windows часто виконуються додаткові перевірки доступу. Зазвичай ці перевірки не потрібні в автономних версіях Windows, але все одно слід розуміти, як вони змінюють процес перевірки доступу, якщо вони присутні.

7.4.1. Перевірка доступу за типом об'єкта

Для спрощення було навмисно вилучено алгоритм перевірки дискреційного доступу обробки об'єктних ACE. Для підтримки об'єктних ACE необхідно використовувати інший API перевірки доступу: або

SeAccessCheckByType у режимі ядра, або системний виклик *NtAccessCheckByType*.

Ці API додають два додаткові параметри до процесу перевірки доступу:

Principal - SID, що використовується для заміни SELF SID в ACE;

ObjectTypes - Список GUID, які є допустимими для перевірки.

Principal легко визначити: коли ми обробляємо DACL і зустрічаємо SID ACE, який встановлений на SELF SID (S-1-5-10), ми замінюємо SID значенням з параметра Principal. (Microsoft ввів SELF SID для використання в Active Directory.) Лістинг 7.25 показує скориговану версію функції *Get-AceSid*, яка враховує це. Вам також доведеться змінити функцію *Get-PSGrantedAccess*, щоб отримати параметр **Principal**, додавши його до значення *$Context*.

Лістинг 7.25. Додавання основного SID до функції *Get-AceSid*

```
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
}
```

Лістинг 7.26 перевіряє поведінку Principal SID.

Лістинг 7.26. Тестування заміни Principal SID

```
PS> $owner = Get-NtSid -KnownSid LocalSystem
PS> $sd = New-NtSecurityDescriptor -Owner $owner -Group $owner -Type Mutant
PS> Add-NtSecurityDescriptorAce $sd -KnownSid Self -Access GenericAll -MapGeneric
PS> Get-NtGrantedAccess $sd -AsString
None
```

Почнемо з створення дескриптора безпеки, в якому власник і група встановлені як SID користувача SYSTEM, а також єдиний ACE *Allowed*, що надає SELF SID доступ *GenericAll*. Згідно з правилами перевірки доступу, це не повинно надавати користувачеві будь-який доступ до ресурсу. Ми можете підтвердити це за допомогою виклику *Get-NtGrantedAccess*.

Далі ми отримуємо SID користувача *ефективного* токена і передаємо його в параметр **Principal** до *Get-NtGrantedAccess*. Перевірка DACL замінить SID SELF на SID **Principal**, який відповідає поточному користувачеві, отже, надає повний доступ. Ця перевірка замінює SID тільки в DACL і SACL. Встановлення SELF як SID власника не буде мати наслідком надання доступу.

Інший параметр, *ObjectTypes*, набагато складніший у реалізації. Він надає список GUID, які є дійсними для процесу перевірки доступу. Кожен GUID представляє тип об'єкта, до якого необхідно отримати доступ. Наприклад, ви можете мати GUID, пов'язаний з об'єктом комп'ютера, і відмінний GUID для об'єкта користувача.

Кожен GUID також має власний рівень, перетворюючи список в ієрархічне дерево. Кожен вузол підтримує свій власний доступ, який він ініціалізує до основного значення *RemainingAccess*. Active Directory використовує цю ієрархію для реалізації концепції власних властивостей і груп властивостей, як показано на рис. 7.4.

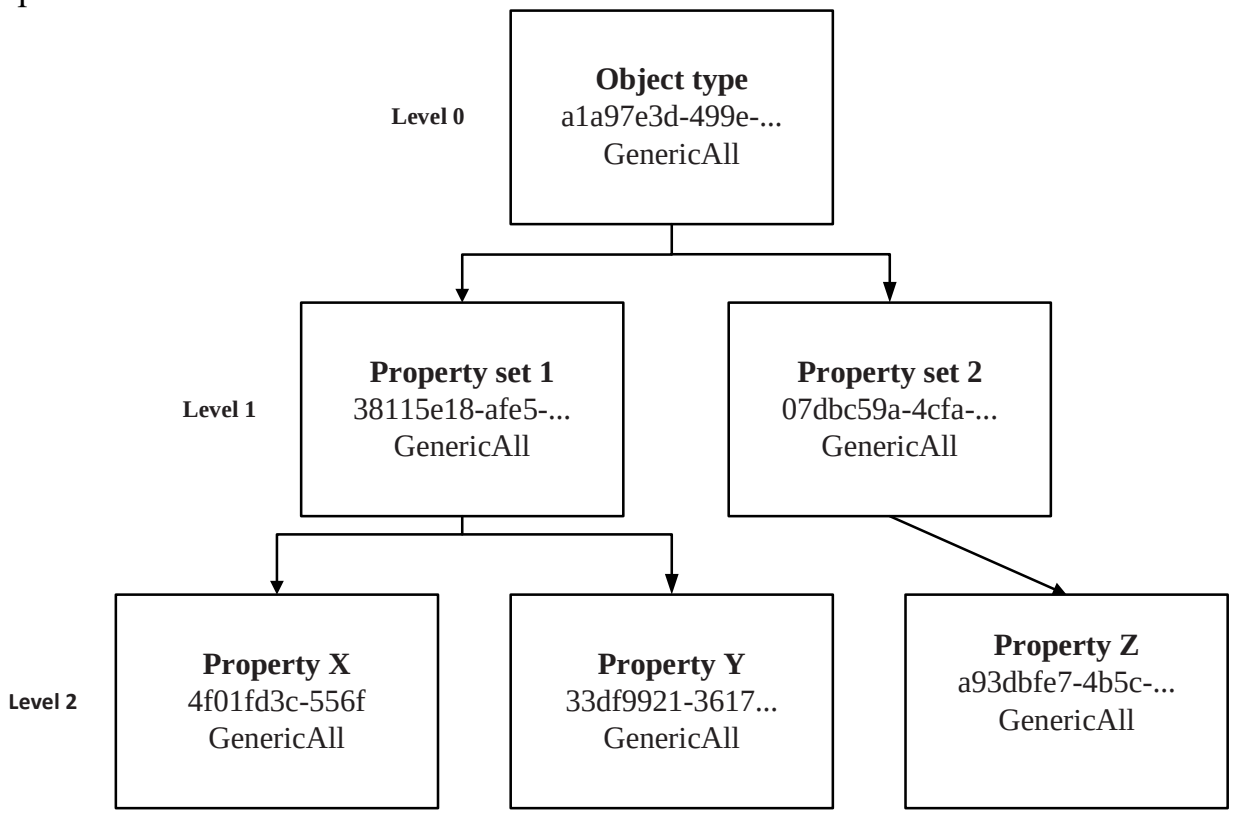


Рисунок 7.4. Властивості в форматі Active Directory

Кожен вузол на рис. 7.4 відображає ім'я, яке йому призначили, частину GUID *ObjectType* і поточне значення *RemainingAccess* (в даному випадку *GenericAll*). Рівень 0 відповідає об'єкту верхнього рівня, який може бути лише один у списку. На рівні 1 знаходяться набори властивостей, в даному випадку мають номери 1 та 2. Під кожним набором властивостей, на рівні 2, містяться окремі властивості.

Налаштування типів об'єктів в ієрархії дозволяє нам конфігурувати дескриптор безпеки для надання доступу до декількох властивостей за допомогою одного ACE, встановивши доступ до набору властивостей. Якщо ми надаємо набору властивостей певний доступ, ми також надаємо цей доступ до всіх властивостей, що містяться в цьому наборі.

І навпаки, якщо ми відмовляємо в доступі до однієї властивості, статус відмови поширюється вгору по дереву і відмовляє в доступі до всього набору властивостей і об'єкта в цілому.

Розглянемо базову реалізацію доступу до типу об'єкта. Код у лістингу 7.27 базується на властивості *ObjectTypes*, доданій до контексту доступу. Ми можемо

згенерувати значення для цього параметра за допомогою команд *New-ObjectTypeTree* та *Add-ObjectTypeTree*.

Лістинг 7.27 показує реалізацію перевірки доступу для типу ACE *AllowedObject*. Додайте його до коду переліку типів ACE з лістингу 7.17.

Лістинг 7.27. Реалізація алгоритму перевірки доступу *AllowedObject* ACE

```
"AllowedObject" {
            if (!(Test-NtTokenGroup -Token $token $sid)) {
                        break
            }
            if ($null -eq $Context.ObjectTypes -or $null -eq $ace.ObjectType) {
                        break
            }
            $object_type = Select-ObjectTypeTree $Context.ObjectTypes
            if ($null -eq $object_type) {
                        break
            }
            Revoke-ObjectTypeTreeAccess $object_type $ace.Mask
            $access = Revoke-NtAccessMask $access $ace.Mask
}
```

Починаємо з перевірки SID. Якщо SID не збігаються, ми не обробляємо ACE. Далі перевіряємо, чи існує властивість *ObjectTypes* у контексті та чи визначає ACE тип об'єкта (*ObjectType* в ACE є необов'язковим). Якщо ці перевірки завершуються невдало, ми ігноруємо ACE. Нарешті, перевіряємо, чи є запис у параметрі *ObjectTypes* для GUID *ObjectType*.

Якщо всі перевірки пройшли успішно, для перевірки доступу беремо до уваги ACE. Спочатку ми анулюємо доступ із запису в дереві об'єктів. Це видаляє доступ не тільки із *ObjectType*, але й із усіх його дочірніх записів.

Проілюструємо цю поведінку на прикладі дерева, показаного на рис. 7.4. Якщо ACE *AllowedObject* надає доступ *GenericAll* до набору властивостей 1, нове дерево буде виглядати так, як показано на рис. 7.5.

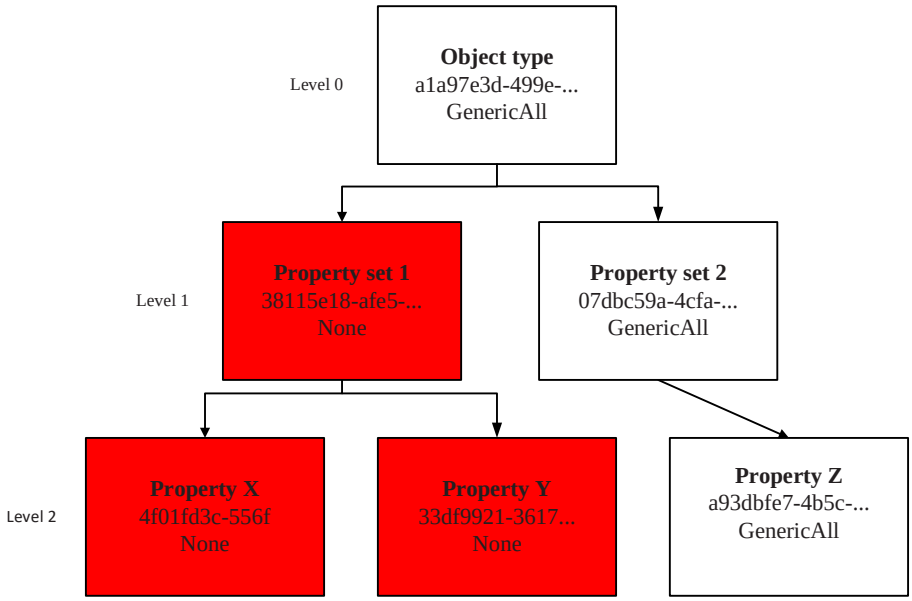


Рисунок 7.5. Дерево типів об'єктів після надання доступу до групи властивостей1

Оскільки доступ *GenericAll* був видалений з *RemainingAccess* для набору властивостей 1, він також був видалений для властивостей **X** і **Y**. Ці вузли тепер мають порожній *RemainingAccess*. Зверніть увагу, що для дозволених ACE важливий тільки основний *RemainingAccess*, оскільки мета дерева полягає в правильній обробці заборонених ACE. Це означає, що не кожен тип об'єкта повинен мати *RemainingAccess* рівний 0, щоб перевірка доступу була успішною.

Тепер давайте обробимо ACE *DeniedObject*. Додайте код у лістинг 7.28 до існуючого коду переліку ACE у лістинг 7.17.

Лістинг 7.28. Реалізація алгоритму перевірки доступу *DeniedObject* ACE

```
"DeniedObject" {
        if (!(Test-NtTokenGroup -Token $token $sid -DenyOnly)) {
                break
                }

        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
                }
        if (Test-NtAccessMask $object_type.RemainingAccess $ace.Mask) {
                 $continue_check = $false
                        break
                }
        }
         if (Test-NtAccessMask $access $ace.Mask) {
                $continue_check = $false
        }
}
```

Починаємо з перевірки всіх ACE типу *DeniedObject*. Якщо перевірка проходить успішно, перевіряємо властивість контексту *ObjectTypes*. При обробці ACE типу *AllowedObject* ми припиняли перевірку, якщо властивість *ObjectType* була відсутня. Однак ACE типу *DeniedObject* ми обробляємо іншим чином.

Якщо властивість *ObjectTypes* відсутня, перевірка продовжується, як у випадку зі звичайним ACE типу *Denied*, з урахуванням основного *RemainingAccess*.

Якщо маска доступу ACE містить біти в *RemainingAccess*, відмовляємо в доступі. Якщо ця перевірка проходить успішно, перевіряємо значення щодо основного *RemainingAccess*. Це демонструє мету підтримки дерева: якщо ACE *Denied* відповідала властивості **X** на рис. 7.5, маска відмови не мала б ніякого ефекту. Однак, якщо ACE *Denied* відповідає властивості **Z**, то цей тип об'єкта, а також властивість набору 2 і тип кореневого об'єкта також будуть відхилені. Це продемонстровано на рис. 7.6. Як бачимо, всі ці вузли тепер відхилені, хоча гілка набору властивостей 1 все ще дозволена..

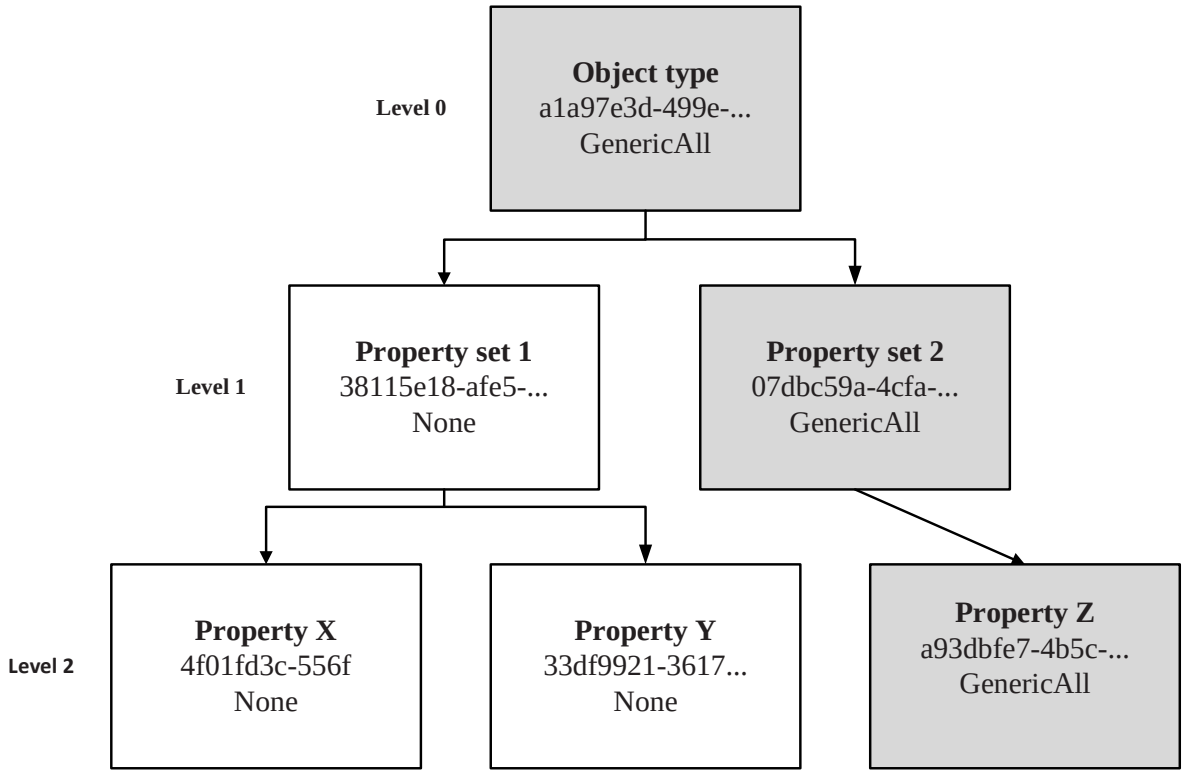


Рисунок 7.6. Дерево типу об'єкта після відмови в доступі до властивості **Z**

Системний виклик *NtAccessCheckByType* повертає єдиний статус і наданий доступ для всього списку типів об'єктів, відображаючи доступ, вказаний у корені дерева типів об'єктів. Тому, у випадку з рисунком 7.6, вся перевірка доступу буде невдалою.

Щоб з'ясувати, які саме типи об'єктів не пройшли перевірку доступу, можна використати системний виклик *NtAccessCheckByTypeResultList*, який повертає статус і наданий доступ для кожного запису в списку типів об'єктів. Лістинг 7.29 показує, як можна використати цей системний виклик, вказавши параметр *ResultList* для *Get-NtGrantedAccess*.

Для початку ми будуємо дерево типів об'єктів, яке відповідає дереву на рис. 7.4.

Нас не цікавлять конкретні значення GUID, за винятком значення властивості Z, яке нам знадобиться для ACE *DeniedObject*, тому ми генеруємо випадкові GUID. Далі ми будуємо дескриптор безпеки, створюючи ACE, який забороняє доступ *ReadControl* до властивості **Z**. Також додаємо ACE, що не є об'єктом, для надання доступу *ReadControl* і *WriteOwner*.

Виконуємо перевірку доступу з деревом типів об'єктів, але без параметра ResultList, запитуючи доступ як для ReadControl, так і для WriteOwner.

Використовуємо Denied ACE, оскільки він відповідає GUID ObjectType в дереві типів об'єктів. Як і очікувалося, це призводить до того, що процес перевірки доступу повертає STATUS _ACCESS_DENIED, з None в якості наданого доступу.

Лістинг 7.29. Приклад, який показує різницю між звичайними результатами та результатами списку

```
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 -MapGeneric -Type DeniedObject -
ObjectType $prop_z
PS> Add-NtSecurityDescriptorAce $sd -KnownSid World -Access ReadControl, WriteOwner -MapGeneric
PS> Edit-NtSecurityDescriptor $sd -CanonicalizeDacl
PS> Get-NtGrantedAccess $sd -PassResult -ObjectType $tree -Access ReadControl, WriteOwner | Format-Table Status,
SpecificGrantedAccess, Name
Status                    SpecificGrantedAccess        Name
 ------                    ---------------------         ----
STATUS_ACCESS_DENIED            None                   Object
PS> Get-NtGrantedAccess $sd -PassResult -ResultList -ObjectType $tree -Access ReadControl, WriteOwner | Format-
Table Status, SpecificGrantedAccess, Name
Status                    SpecificGrantedAccess        Name
 ------                    ---------------------         ----
STATUS_ACCESS_DENIED          ReadControl 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
```

Коли знову виконуємо перевірку доступу, цього разу з *ResultList*, отримуємо список результатів перевірки доступу. Запис об'єкта верхнього рівня все ще вказує, що в доступі було відмовлено, для набору властивостей 1 та його дочірніх елементів. Цей результат відповідає дереву, показаному на рис. 7.6. Також зверніть увагу, що записи, для яких доступ був відхилений, не показують порожній доступ. Натомість вони показують, що доступ *ReadControl* був би наданий, якби запит був успішним. Це наслідок того, як перевірка доступу реалізована під капотом, і напевно не повинна використовуватися.

7.4.2. Політика central access

Політика *central access*, функція, додана в Windows 8 і Windows Server 2012 для використання в корпоративних мережах, є основним механізмом безпеки, що лежить в основі функції Windows під назвою *Dynamic Access Control* (Динамічний контроль доступу). Вона покладається на атрибути пристрою і користувача в токені.

Ми коротко говорили про вимоги користувачів і пристроїв у розділі 4, коли обговорювали формат умовного виразу. Вимога користувача — це атрибут безпеки, доданий до токена для певного користувача. Наприклад, у вас може бути вимога, яка визначає країну, в якій працює користувач.

Ви можете синхронізувати значення вимоги із значеннями, що зберігаються в Active Directory, щоб, наприклад, якщо користувач переїжджає в іншу країну, його вимога оновлювалася під час наступної автентифікації.

Вимога пристрою стосується комп'ютера, який використовується для доступу до ресурсу. Наприклад, вимога щодо пристрою може вказувати, чи знаходиться комп'ютер у захищеній кімнаті або чи працює він під управлінням певної версії Windows. На рис. 7.7 показано типове *central access* політики доступу: обмеження доступу до файлів на сервері в корпоративній мережі.

Політика доступу *central access* містить один або кілька дескрипторів безпеки, які перевірка доступу буде враховувати на додаток до дескриптора безпеки файлу.

Остаточний наданий доступ є найбільш обмежувальним результатом перевірок доступу. Хоча це не є строго необхідним, додаткові дескриптори безпеки можуть покладатися на вимоги користувачів і пристроїв в ACE *AllowedCallback* для визначення наданого доступу.

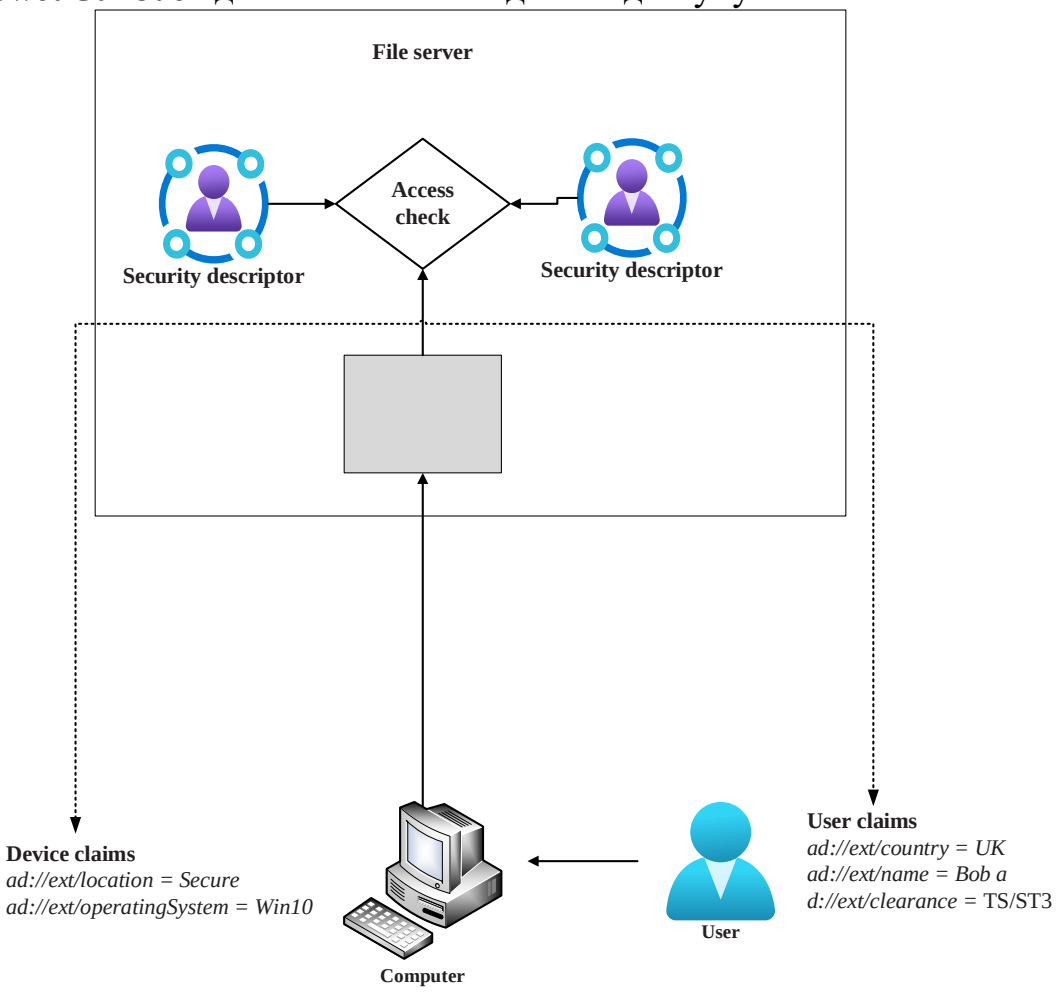


Рисунок 7.7. Політика *central access* на файловому сервері

Ви можете запитати, чим використання політики доступу *central access* відрізняється від простої настройки безпеки файлів для використання пристроєм та обліковими записами користувачів. Основна відмінність полягає в тому, що

вона централізовано управляється за допомогою принципів політики групи домену підприємства. Це означає, що адміністратору потрібно змінити *central access* політику доступу в одному місці, щоб оновити її в усьому підприємстві.

Друга відмінність полягає в тому, що *central access* політика доступу працює більше як механізм обов'язкового контролю доступу. Наприклад, користувач зазвичай може змінювати дескриптор безпеки файлу, однак центральна політика доступу може обмежити його доступ або повністю заблокувати його, якщо, наприклад, користувач переїхав до іншої країни або використовує інший комп'ютер, який не врахований у правилах.

Не будемо обговорювати, як налаштувати *central access* політику доступу, оскільки ця тема більше підходить для посібника про управління Windows в корпоративному середовищі.

Натомість ми розглянемо, як вона застосовується процесом перевірки доступу ядра.

Реєстр Windows зберігає центральну політику доступу під час оновлення групової політики комп'ютера, і ви можете знайти ключ у такому місці:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Lsa\ CentralizedAccessPolicies
```

Може бути налаштовано більше ніж одну політику, кожна з яких містить таку інформацію:

—Назва та опис політики.

—SID, що однозначно ідентифікує політику.

—Одне або декілька правил політики.

У свою чергу, кожне правило політики містить таку інформацію:

—Назва та опис правила.

—Умовний вираз, що визначає, коли правило повинно застосовуватися.

—Дескриптор безпеки, що використовується в *central access* політики доступу.

—Необов'язковий проміжний дескриптор безпеки, що використовується для тестування нових правил політики.

Ви можете використовувати команду PowerShell *Get-CentralAccessPolicy* для відображення списку політик і правил. Для більшості систем Windows команда не поверне жодної інформації. Щоб побачити результати, подібні до тих, що наведені в лістингу 7.30, вам потрібно приєднатися до домену, який налаштований на використання central access.

Тут, коли ми запускаємо Get-CentralAccessPolicy, ми бачимо дві політики: *SecureRoomPolicy* та *MainPolicy*. Кожна політика має *CapId* SID та властивість *Rules*, яку ми можемо розгорнути, щоб побачити окремі правила. Таблиця виводу містить такі поля: *Name*, *Description* та *AppliesTo*, яке є умовним виразом, що використовується для вибору, чи слід застосовувати правило. Якщо поле *AppliesTo* порожнє, правило будуть застосовувати завжди. Поле *AppliesTo* для *SecureRule* вибирає атрибут ресурсу, до якого ми повернемося в лістингу 7.32.

Лістинг 7.30. Відображення політики central access

```
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"
```

Давайте відобразимо дескриптор безпеки для цього правила. DACL містить єдиний ACE *AllowedCallback*, який надає повний доступ групі *Everyone*, якщо умова відповідає. У цьому випадку вимога користувача на допуск повинна бути встановлена на значення TS/ST3 а місце розташування пристрою повинно бути встановлено на Secure.

Ми розглянемо базову реалізацію перевірки доступу до політики *central access* , щоб краще зрозуміти, для чого використовується ця політика. Додайте код з лістингу 7.31 в кінець функції *Get-PSGrantedAccess* з лістингу 7.2.

Лістинг 7.31 починається відразу після перевірки дискреційного доступу.

Якщо ця перевірка не пройде, змінна *$success* буде *False*, і нам повернуто STATUS_ACCESS_DENIED. Щоб розпочати процес застосування *central access* політики, нам потрібно запитати *ScopedPolicyId* ACE з SACL. Якщо *ScopedPolicyId* ACE відсутній, можемо повернути успіх. Ми також повертаємо успіх, якщо немає *central access* політики з *CapId*, що відповідає SID ACE.

В рамках перевірки central access політики спочатку встановлюємо ефективний доступ до початкового DesiredAccess. Будемо використовувати ефективний доступ, щоб визначити, яку частину DesiredAccess ми можемо призначити після обробки всіх правил політики.

Далі перевіряємо умовний вираз AppliesTo для кожного правила. Якщо значення відсутнє, правило застосовується до всіх ресурсів і токенів. Якщо є умовний вираз, ми повинні перевірити його за допомогою Test-NtAceCondition, передаючи будь-які атрибути ресурсу з дескриптора безпеки. Якщо тест не пройдено, перевірка повинна перейти до наступного правила.

Ми створюємо новий дескриптор безпеки, використовуючи власника, групу та SACL з оригінального дескриптора безпеки, але DACL з дескриптора безпеки правила. Якщо правило застосовується, ми виконуємо ще одну перевірку

дискреційного доступу для DesiredAccess. Після цієї перевірки ми видаляємо всі біти, які нам не були надані, із змінної effective_access.

Лістинг 7.31. Перевірка *central access* політики

```
if (!$success) {
        return Get-AccessResult STATUS_ACCESS_DENIED
}
$capid = $SecurityDescriptor.ScopedPolicyId
if ($null -eq $capid) {
        return Get-AccessResult STATUS_SUCCESS $Context.Privileges $DesiredAccess
}
$policy = Get-CentralAccessPolicy -CapId $capid.Sid
if ($null -eq $policy){
        return Get-AccessResult STATUS_SUCCESS $Context.Privileges $DesiredAccess
}
$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 -ResourceAttribute
                        $resource_attrs)) {
                                continue
                                }
                        }
                $new_sd = Copy-NtSecurityDescriptor $SecurityDescriptor
                Set-NtSecurityDescriptorDacl $rule.Sd.Dacl
                $Context.SecurityDescriptor = $new_sd
                $Context.RemainingAccess = $DesiredAccess

                Get-DiscretionaryAccess $Context 8 $effective_access = $effective_access -band (-bnot
                $Context.RemainingAccess)
        }
        if (Test-NtAccessMask $effective_access -Empty) {
        return Get-AccessResult STATUS_ACCESS_DENIED
        }
return Get-AccessResult STATUS_SUCCESS $Context.Privileges $effective_access
```

Перевіривши всі діючі правила, ми перевіряємо, чи ефективний доступ є порожнім. Якщо так, *central access* політика не надала токену жодного доступу, тому ми повертаємо STATUS_ACCESS_DENIED. В іншому випадку повертаємо успіх, але повертаємо лише залишковий ефективний доступ, який надає менше доступу, ніж результат першої перевірки.

Хоча більшість *central access* політик призначені для перевірки файлів, ми можемо модифікувати будь-який тип ресурсу для примусового застосування політики. Щоб увімкнути її для іншого ресурсу, нам потрібно зробити дві речі: встановити ідентифікатор політики ACE з SID політики, та додати будь-які атрибути ресурсу ACE, щоб відповідати умові *AppliesTo*, якщо така існує. Виконаємо ці дії в лістингу 7.32.

Перше, що нам потрібно зробити, це додати атрибут ресурсу ACE, щоб задовольнити умову AppliesTo для правила безпеки. Створюємо об'єкт атрибуту безпеки з іменем EnableSecure і єдиним значенням Int64, рівним 1. Додаємо цей атрибут безпеки до ACE типу ResourceAttribute в SACL дескриптора безпеки.

Потім нам потрібно встановити SID central access політики, яку можемо отримати з результату команди Get-CentralAccessPolicy в ACE ScopedPolicyId. Можемо відформатовати дескриптор безпеки, щоб перевірити правильність ACE.

Тепер встановлюємо два ACE для ресурсу. У цьому випадку виберемо ресурс у вигляді ключа реєстру. Зверніть увагу, що для успішного виконання операції цей ключ реєстру повинен бути попередньо створений. Параметр SecurityInformation повинен бути встановлений на Scope і Attribute. Для встановлення ACE ScopedPolicyId нам потрібен доступ AccessSystemSecurity, а це означає, що спочатку потрібно ввімкнути SeSecurityPrivilege.

Якщо ви отримаєте доступ до ключа реєстру, ви повинні знайти політику, яка буде застосовуватися.

Лістинг 7.32. Увімкнення Secure Room Policy для ключа реєстру

```
PS> $sd = New-NtSecurityDescriptor
PS> $attr = New-NtSecurityAttribute "EnableSecure" -LongValue 1
PS> Add-NtSecurityDescriptorAce $sd -Type ResourceAttribute -Sid "WD"
-SecurityAttribute $attr -Flags ObjectInherit, ContainerInherit PS> $capid = "S-1-17-3260955821-1180564752-1365479606-
2616254494"
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> Enable-NtTokenPrivilege  SeSecurityPrivilege
PS> Set-Win32SecurityDescriptor $sd MACHINE\SOFTWARE\PROTECTED
-Type RegistryKey -SecurityInformation Scope, Attribute
```

Зверніть увагу, що оскільки *central access* політика налаштована для використання з файловими системами, маска доступу в дескрипторі безпеки може не працювати правильно з іншими ресурсами, такими як ключі реєстру. Ви можете вручну налаштувати атрибути в Active Directory, якщо ви дійсно хочете підтримати таку поведінку.

Останнє, про що варто зазначити, це те, що правила *central access* політики підтримують визначення проміжного дескриптора безпеки, а також звичайного дескриптора безпеки. Можемо використовувати цей проміжний дескриптор безпеки для тестування майбутніх змін безпеки перед їх широким розгортанням.

Проміжний дескриптор безпеки перевіряється так само, як і звичайний дескриптор безпеки, за винятком того, що результат перевірки використовується тільки для порівняння з реальним наданим доступом, і якщо дві маски доступу відрізняються, створюється журнал аудиту.

.

7.5. ПРАКТИЧНІ ПРИКЛАДИ

На завершення розглянемо кілька прикладів використання команд, про які ви дізналися в цьому розділі.

7.5.1. Використання команди Get-PSGrantedAccess

У цьому розділі ми реалізували власний процес перевірки доступу: команду Get-PSGrantedAccess. Розглянемо використання цієї команди.

Оскільки Get-PSGrantedAccess є простою реалізацією перевірки доступу, у ній відсутні деякі функції, такі як підтримка обчислення максимального доступу. Однак вона все одно може допомогти вам зрозуміти процес перевірки доступу. Наприклад, ви можете використовувати відладчик PowerShell в інтегрованому середовищі сценаріїв PowerShell (ISE) або Visual Studio Code, щоб пройти перевірку доступу і побачити, як вона функціонує на прикладі різних вхідних даних.

Виконайте команди в лістингу 7.33 як користувач із розділеним токеном, якому не надано прав адміністратора.

Лістинг 7.33. Використання команди Get-PSGrantedAccess

```
PS> Import-Module ".\chapter_7_access_check_impl.psm1"
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
PS> Get-PSGrantedAccess -SecurityDescriptor $sd
-GenericMapping $type.GenericMapping -DesiredAccess $desired_access
Status                  Privileges          GrantedAccess
------                  ----------          -------------
STATUS_SUCCESS          {}                  1179785
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
PS> $token = Get-NtToken -Linked
PS> Enable-NtTokenPrivilege -Token $token SeTakeOwnershipPrivilege
PS> Get-PSGrantedAccess -Token $token -SecurityDescriptor $sd
-GenericMapping $type.GenericMapping -DesiredAccess $desired_access
Status                  Privileges                      GrantedAccess
------                  ----------                      -------------
STATUS_SUCCESS          {SeTakeOwnershipPrivilege} 524288
```

Спочатку імпортуємо модуль, що містить команду *Get-PSGrantedAccess*. Імпорт передбачає, що файл модуля збережено у вашому поточному каталозі.

Якщо це не так, змініть шлях відповідно. Потім створюємо обмежувальний дескриптор безпеки, надаючи доступ на читання групі *Everyone* і нікому іншому.

Далі викликаємо *Get-PSGrantedAccess*, запитуючи доступ *GenericRead* разом із загальним відображенням типу об'єкта *File*. Не вказуємо параметр *Token*, що означає, що перевірка буде використовувати ефективний токен виклику.

Команда повертає STATUS_SUCCESS, і наданий доступ відповідає бажаному доступу.

Потім змінюємо бажаний доступ на доступ *WriteOwner*. На основі обмежувального дескриптора безпеки. Тільки власник дескриптора безпеки, який був встановлений як SYSTEM, повинен отримати цей доступ. Коли повторно виконуємо перевірку доступу, ми отримуємо STATUS_ACCESS_DENIED, тобто жодного наданого доступу.

Щоб показати, як можна обійти ці обмеження, ми запитуємо пов'язаний токен виклику. Ця команда не працюватиме, якщо ви не запускаєте скрипт як адміністратор з розділеним токеном. Однак ми можемо увімкнути привілей *SeTakeOwnershipPrivilege* на пов'язаному токені, що повинно обійти перевірку власника для *WriteOwner*. Тепер перевірка доступу повинна повернути STATUS_SUCCESS і надати бажаний доступ. У стовпці *Privileges* показано, що для надання права доступу було використано *SeTakeOwnershipPrivilege*.

Як уже згадувалося, варто запустити цей скрипт у відладчику і перейти до *Get-PSGrantedAccess*, щоб простежити процес перевірки доступу. також радимо спробувати різні комбінації значень у дескрипторі безпеки.

7.5.2. Обчислення наданого доступу до ресурсів

Якщо вам дійсно потрібно знати наданий доступ до ресурсу, краще використовувати команду *Get-NtGrantedAccess* замість розробленої нами реалізації PowerShell. Давайте подивимося, як можна використовувати цю команду для отримання наданого доступу до списку ресурсів. У лістингу 7.34 ми візьмемо скрипт, який використовували для пошуку власників об'єктів, і розрахуємо повний наданий доступ.

У цій модифікованій версії скрипта, створеного в лістингу 6.37, замість простої перевірки SID власника, викликаємо Get-NtGrantedAccess з дескриптором безпеки. За допомогою цього скрипта отримаємо доступ. Іншою стратегією було б перевірити наданий доступ для будь-якого токена імперсонації на рівні ідентифікації за допомогою доступу Query дескриптора, а потім передати його як параметр Token. У подальшому розглянемо простіший спосіб виконання перевірки доступу повному обсязі без необхідності писати власні скрипти.

Лістинг 7.34. Перелік об'єктів з правами доступу до них

```
PS> function Get-NameAndGrantedAccess {
        [CmdletBinding()]
        param(
                [parameter(Mandatory, ValueFromPipeline)]
                $[Epnatrraym,eter(Mandatory)]
                $Root
        )
        PROCESS {
                $sd = Get-NtSecurityDescriptor -Path $Entry.Name -Root $Root -TypeName $Entry.NtTypeName -
                ErrorAction SilentlyContinue
                if ($null -ne $sd) {
                        $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--
```

7.6. ВИСНОВКИ ДО РОЗДІЛУ 7

У цьому розділі детально описано процес перевірки доступу в Windows. Зокрема, розглянули обов'язкові перевірки доступу операційної системи, перевірки власника токена і привілеїв, а також дискреційні перевірки доступу. Також ми створили власну реалізацію процесу перевірки доступу, щоб полегшити його розуміння.

Далі обговорили, яким чином два типи токенів пісочниці (*обмежений* і **lowbox**) впливають на процес перевірки доступу для обмеження доступу до ресурсів.

Нарешті, обговорили перевірку типу об'єкта та *central access* політики, важливі функції корпоративної безпеки для Windows.

ПИТАННЯ ДЛЯ САМОКОНТРОЛЮ

1. В чому полягає процес перевірки доступу, який приймає токен і дескриптор безпеки?
2. API SeAccessCheck реалізує процес перевірки доступу в режимі ядра. Які параметри він приймає?
3. Навіщо викликати SeAccessCheck без застосування будь-яких заходів безпеки?
4. Як перевірити покажчик пам'яті?
5. Мета використання системного виклика *NtAccessCheck*?
6. Як визначити функції перевірки доступу?
7. Як перевірити рівень довіри до процесу?
8. З якою метою використовується фільтр доступу ACE?
9. Реалізуйте обов'язкову перевірку рівня цілісності.
10. Що визначає перевірка привілеїв?
11. Наведіть алгоритм перевірки доступу до привілеїв токена.
12. Мета перевірки власника?
13. Мета використання обмеженого токена?
14. Які відмінності має перевірка доступу токена lowbox?
15. Наведіть алгоритм перевірки обов'язкового доступу до токена lowbox.
16. Які додаткові перевірки доступу виконуються у корпоративних версіях Windows?
17. В чому полягає політика central access?
18. Як знати наданий доступ до ресурсу?

РОЗДІЛ 8. ІНШІ ВИПАДКИ ВИКОРИСТАННЯ ПЕРЕВІРКИ ДОСТУПУ

Перевірки доступу визначають, який доступ повинен мати виклик при відкритті ресурсу ядра. Однак іноді ми виконуємо їх з інших причин, оскільки вони можуть служити в якості додаткових перевірок безпеки.

У цьому розділі наведено кілька прикладів використання перевірок доступу як додаткового механізму безпеки.

Почнемо з перевірки, яка визначає, чи має користувач доступ до ієрархії ресурсів. Далі обговоримо, як перевірки доступу використовуються при дублюванні дескриптора. Також розглянемо, як перевірка доступу може обмежити доступ до інформації ядра, такої як лістинг процесів. Нарешті, розповімо про деякі додаткові команди PowerShell, які автоматизують перевірку доступу до ресурсів.

8.1. ПЕРЕВІРКА ОБХОДУ

При доступі до ієрархічного набору ресурсів, такого як дерево каталогів об'єктів, користувач повинен пройти всю ієрархію, поки не досягне цільового ресурсу. Для кожного каталогу або контейнера в ієрархії система виконує перевірку доступу, щоб визначити, чи може користувач перейти до наступного контейнера. Ця перевірка називається перевіркою обходу (*traversal check*) і виконується щоразу, коли здійснюється пошук шляху всередині диспетчера вводу-виводу або диспетчера об'єктів.

Наприклад, на рис. 8.1 показано перевірки обходу, необхідні для доступу до об'єкта OMNS за допомогою шляху ABC\QRS\XYZ\OBJ.

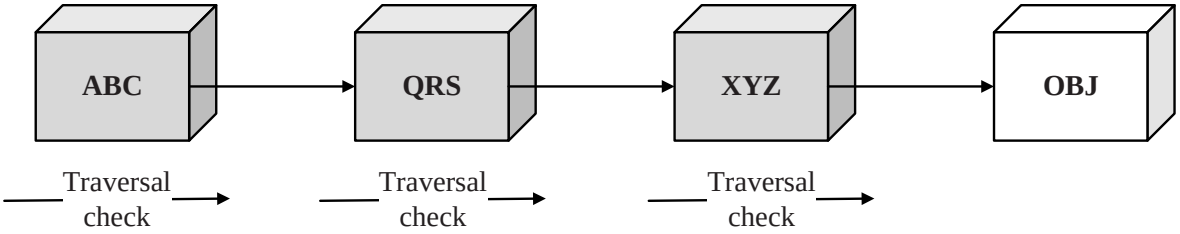


Рисунок 8.1. Перевірки доступу до OBJ

Як бачите, перед тим, як отримати доступ до OBJ, необхідно виконати три перевірки доступу. Кожна перевірка доступу зчитує дескриптор безпеки з контейнера, а потім перевіряє доступ для конкретного типу, щоб визначити, чи дозволено обхід. Як OMNS, так і каталоги файлів можуть надавати або відмовляти в доступі до обходу. Якщо, наприклад, QRS відмовив у доступі до обходу тому, хто здійснює виклик, перевірка обходу завершиться невдачею, як показано на рис. 8.2.

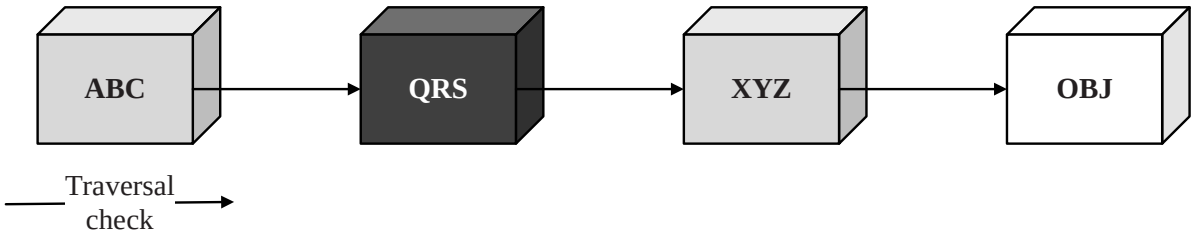


Рисунок 8.2. Перевірки обходу заблоковані в QRS

Навіть якщо користувач пройде перевірку доступу для XYZ та OBJ, оскільки QRS тепер відмовляє в доступі через перевірку обходу, він більше не зможе отримати доступ до OBJ за допомогою шляху ABC\QRS\XYZ\OBJ.

Перевірка обходу запобігає доступу користувача до своїх ресурсів, якщо будь-який батьківський контейнер відмовляє в доступі до обходу. Це несподівана поведінка — чому користувач не може отримати доступ до своїх власних ресурсів? Це також викликає занепокоєння щодо продуктивності. Якщо користувач повинен мати доступ до кожного батьківського контейнера, щоб отримати доступ до своїх файлів, то ядро повинно витратити час і зусилля на перевірку доступу для кожного контейнера, коли з точки зору безпеки важливо лише те, чи має користувач доступ до ресурсу, який він хоче відкрити.

8.1.1. Привілей SeChangeNotifyPrivilege

Щоб поведінка перевірки обходу була ближчою до очікуваної та зменшила вплив на продуктивність, SRM визначає привілей *SeChangeNotifyPrivilege*, який за замовчуванням увімкнена майже для кожного об'єкта Token. Коли цей привілей увімкнена, система обходить всю перевірку обходу та дозволяє користувачам отримати доступ до ресурсів, які інакше були б заблоковані недоступним батьківським об'єктом. У лістингу 8.1 ми перевіряємо поведінку привілею за допомогою об'єктів каталогу OMNS.

Спочатку ми створюємо об'єкт Mutant і всі його батьківські каталоги, автоматизуючи створення каталогів за допомогою властивості CreateDirectories. Ми переконуємося, що привілей увімкнена, а потім використовуємо команду Test-NtObject, щоб перевірити, чи можна відкрити об'єкт Mutant. У результаті ми бачимо, що об'єкт Mutant можна відкрити.

Потім ми встановлюємо дескриптор безпеки з порожнім DACL у каталозі QRS. Це повинно заблокувати весь доступ до об'єкта каталогу, включаючи доступ для переходу. Але коли ми знову перевіряємо наш доступ, ми бачимо, що все ще можемо отримати доступ до об'єкта Mutant, оскільки у нас увімкнене привілей SeChangeNotifyPrivilege.

Лістинг 8.1. Тестування *SeChangeNotifyPrivilege*

```
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
```

Тепер ми вимикаємо привілей і знову намагаємося відкрити об'єкт **Mutant**. Цього разу перехід по каталогах не вдається. Без привілею *SeChangeNotifyPrivilege* або доступу до каталогу QRS ми більше не можемо відкрити об'єкт **Mutant**. Однак наша остаточна перевірка показує, що якщо ми маємо доступ до батьківського каталогу після QRS, наприклад XYZ, ми можемо отримати доступ до об'єкта **Mutant** за допомогою відповідного відкриття, використовуючи каталог як параметр *Root*.

8.1.2. Обмежені перевірки

Ядро містить додаткове поліпшення продуктивності для перевірок обходу. Якщо привілей *SeChangeNotifyPrivilege* вимкнено, ядро викличе функцію *SeFastTraverseCheck*, яка виконує більш обмежену перевірку замість повної перевірки доступу.

Лістинг 8.2. Реалізація SeFastTraverseCheck у PowerShell

```
function Get-FastTraverseCheck {
        Param(
                $TokenFlags,
                $SecurityDescriptor,
                $AccessMask
        )
        if ($SecurityDescriptor.DaclNull) {
                return $true
        }
        if (($TokenFlags -band "IsFiltered, IsRestricted") -ne 0) {
                return $false
        }
        $sid = Get-Ntsid -KnownSid World
                foreach($ace in $SecurityDescriptor.Dacl) {
                        if ($ace.IsInheritedOnly -or !$ace.IsAccessGranted($AccessMask)) {
```

Для повноти ми повторно реалізували функцію SeFastTraverseCheck у PowerShell, щоб ми могли докладніше дослідити її поведінку. Лістинг 8.2 показує реалізацію.Спочатку ми визначаємо три параметри, які приймає функція: прапорці токена, дескриптор безпеки об'єкта каталогу та права доступу *traverse* для перевірки. Ми вказуємо права доступу, оскільки менеджер об'єктів та менеджер вводу-виводу використовують цю функцію для об'єктів каталогу та файлу, а значення права доступу *traverse* відрізняється для цих двох типів об'єктів. Визначення доступу як параметра дозволяє функції перевірки обробляти обидва випадки.

Далі ми перевіряємо, чи DACL дескриптора безпеки дорівнює NULL, і якщо це так, то надаємо доступ. Далі ми перевіряємо два прапорці токена. Якщо прапорці вказують, що токен відфільтрований або обмежений, то швидка перевірка завершується невдачею. Ядро копіює ці прапорці з об'єкта Token об'єкта, що викликає. Ми можемо отримати прапорці з режиму користувача, використовуючи властивість *Flags* об'єкта Token, як показано в лістингу 8.3.

Лістинг 8.3. Перевірка прапорців токенів

```
PS> $token = Get-NtToken -Pseudo -Primary
PS> $token.Flags
VirtualizeAllowed, IsFiltered, NotLow

PS> $token.ElevationType
Limited
```

Зверніть увагу, що прапорці включають *IsFiltered*. Якщо ви не працюєте в обмеженій пісочниці токенів, чому цей прапор встановлений? Запит типу підвищення токена показує, що він обмежений, що означає, що це токен за замовчуванням для адміністратора UAC. Щоб перетворити повний токен

адміністратора на токен за замовчуванням, LSASS використовує систему *NtFilterToken*, яка встановлює прапор *IsFiltered*, але не *IsRestricted*, оскільки вона тільки видаляє групи, а не додає обмежені SID. Це означає, що хоча адміністратор UAC, який виконує код як користувач за замовчуванням, ніколи не може пройти перевірку швидкого обходу, звичайний користувач може це зробити. Така поведінка не має жодних наслідків для безпеки, але означає, що якщо *SeChangeNotifyPrivilege* вимкнено, продуктивність пошуку ресурсів погіршиться.

Фінальна перевірка в лістингу 8.3 полягає в переліку ACE DACL. Якщо ACE є тільки успадкованим або не містить необхідної маски доступу *Traverse*, він пропускається. Якщо це ACE *Denied*, швидка перевірка траверсу не проходить, і SID ACE взагалі не перевіряється. Нарешті, якщо ACE є дозволеним ACE, а SID дорівнює SID групи *Everyone*, швидка перевірка проходить успішно. Якщо більше немає ACE, перевірка провалюється.

Зверніть увагу, що ця швидка перевірка не враховує, чи у токені виклику увімкнена групу *Everyone*. Це пов’язано з тим, що зазвичай єдиний спосіб видалити групу *Everyone* — відфільтрувати токен. Винятком є анонімний токен, який не має жодних груп, але також не фільтрується жодним чином.

Тепер перейдемо до іншого використання перевірки доступу: врахування наданого доступу під час призначення дубльованого дескриптора.

8.2. ПЕРЕВІРКА ДОСТУПУ ДО ДУБЛЮВАННЯ ДЕСКРИПТОРІВ

Система завжди виконує перевірку доступу під час створення або відкриття ресурсу ядра, який повертає дескриптор. Але що відбувається, коли цей дескриптор дублюється? У найпростішому випадку, коли новий дескриптор має ту саму маску доступу, що й оригінальний, система не виконує жодних перевірок. Також можна видалити деякі частини наданої маски доступу, і це також не спричинить додаткової перевірки доступу. Однак, якщо ви хочете додати додаткові права доступу до дубльованого дескриптора, ядро запитає дескриптор безпеки з об'єкта і виконає нову перевірку доступу, щоб визначити, чи дозволити доступ.

При дублюванні дескриптора необхідно вказати дескриптори як вихідного, так і цільового процесів, а перевірка доступу відбувається в контексті цільового процесу. Це означає, що перевірка доступу враховує первинний токен цільового процесу, а не вихідного, що може стати проблемою, якщо привілейований процес спробує дублювати дескриптор для менш привілейованого процесу з додатковим доступом. Така операція завершиться з помилкою *Access Denied*.

Лістинг 8.4 демонструє поведінку перевірки доступу при дублюванні дескриптора.

Лістинг 8.4. Приклад тестування поводження перевірки доступу при дублюванні дескриптора

```
PS> $sd = New-NtSecurityDescriptor -EmptyDacl
PS> $m = New-NtMutant -Access ModifyState, ReadControl -SecurityDescriptor $sd
PS> Use-NtObject($m2 = Copy-NtObject -Object $m) {
        $m2.GrantedAccess
}
ModifyState, ReadControl
PS> $mask = Get-NtAccessMask -MutantAccess ModifyState
PS> Use-NtObject($m2 = Copy-NtObject -Object $m -DesiredAccessMask $mask) {
        $m2.GrantedAccess
}
ModifyState
PS> Use-NtObject($m2 = Copy-NtObject -Object $m -DesiredAccess GenericAll) {
        $m2.GrantedAccess
}
Copy-NtObject : (0xC0000022) - {Access Denied}
A process has requested access to an object, ...
```

Спочатку ми створюємо новий об'єкт **Mutant** з порожнім DACL і запитуємо тільки доступ *ModifyState* і *ReadControl* на дескрипторі. Це заблокує доступ до об'єкта всім користувачам, крім власника, якому може бути надано доступ *ReadControl* і *WriteDac* завдяки перевірці власника, описаній у попередньому розділі. Ми перевіряємо дублювання, запитуючи той самий доступ, який повертає новий дескриптор.

Далі ми запитуємо тільки доступ *ModifyState*. Оскільки DACL **Mutant** порожній, це право доступу не буде надано під час перевірки доступу, і оскільки ми отримуємо *ModifyState* на новому дескрипторі, ми знаємо, що перевірка доступу не відбулася. Нарешті, ми намагаємося збільшити наш доступ, запитуючи доступ *GenericAll*. Тепер повинна відбутися перевірка доступу, оскільки ми запитуємо більші права доступу, ніж має дескриптор. Ця перевірка призводить до помилки *Access Denied*.

Якби ми не встановили дескриптор безпеки під час створення **Mutant**, об'єкт не мав би жодних обмежень безпеки, і ця остання перевірка була б успішною, надавши повний доступ. Потрібно бути обережним під час дублювання безіменних дескрипторів для процесів з меншими привілеями, якщо ви відмовляєтеся від доступу. Процес призначення може бути здатний повторно дублювати дескриптор для отримання більшого доступу. У лістингу 8.5 протестуємо прапор *NtDuplicateObject NoRightsUpgrade*, щоб побачити, як він впливає на перевірку доступу при дублюванні дескриптора.

Лістинг 8.5. Тестування прапора *NtDuplicateObject NoRightsUpgrade*

```
PS> $m = New-NtMutant -Access ModifyState
PS> Use-NtObject($m2 = Copy-NtObject -Object $m -DesiredAccess GenericAll) {
        $m2.GrantedAccess
}
ModifyState, Delete, ReadControl, WriteDac, WriteOwner, Synchronize
PS> Use-NtObject($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, ...
```

Почнемо з створення об'єкта **Mutant** без імені, який не матиме пов'язаного дескриптора безпеки. Ми запитуємо початковий дескриптор лише з доступом *ModifyState*. Однак наша спроба дублювати новий дескриптор з доступом *GenericAll* є успішною, що надає нам повний доступ.

Тепер ми перевіряємо прапор *NoRightsUpgrade*. Оскільки ми не вказуємо жодної маски доступу, дескриптор буде дубльований з доступом *ModifyState*. З новим дескриптором ми виконуємо ще одне дублювання, цього разу запитуючи доступ *GenericAll*. Ми можемо помітити, що дублювання дескриптора не вдалося. Це не через перевірку доступу, а через прапор, встановлений на дескрипторі в ядрі, який вказує, що будь-який запит на додатковий доступ повинен негайно відхилятися. Це запобігає використанню дескриптора для отримання додаткових прав доступу.

Неналежне оброблення дублікатів дескрипторів може призвести до появи вразливостей. Наприклад, CVE-2019-0943 — проблема, яка виявлена у привілейованій службі, відповідальній за кешування деталей файлів шрифтів у Windows. Служба дублювала дескриптор об'єкта Section у процесі пісочниці з доступом тільки для читання. Однак процес пісочниці міг перетворити дескриптор назад на дескриптор розділу з правом запису, і розділ міг бути відображений у пам'яті як доступний для запису. Це дозволяло процесу пісочниці змінювати стан привілейованої служби та виходити за межі пісочниці. Windows виправила цю вразливість, дублюючи дескриптор за допомогою прапора *NoRightsUpgrade*.

Перевірки доступу, що відбуваються під час перевірки обходу та дублювання дескрипторів, зазвичай приховані від очей, але вони стосуються безпеки окремого ресурсу. Далі ми обговоримо, як перевірки доступу обмежують інформацію, яку ми можемо витягти, та операції, які ми можемо виконати для групи ресурсів. Ці обмеження відбуваються на основі токена виклику, незалежно від індивідуального доступу, встановленого для цих ресурсів.

8.3. ПЕРЕВІРКА ТОКЕНІВ ПІСОЧНИЦІ

Починаючи з Windows 8, Microsoft намагається ускладнити можливість компрометації системи шляхом обходу обмежень токенів пісочниці. Це

особливо важливо для програмного забезпечення, такого як веб-браузери та програми для читання документів, які обробляють ненадійний вміст з Інтернету.

Ядро реалізує два API, які використовують перевірку доступу для визначення, чи знаходиться виклик у пісочниці: *ExIsRestrictedCaller*, представлений у Windows 8, та *RtlIsSandboxToken*, представлений у Windows 10. Ці API дають еквівалентні результати. Різниця між ними полягає в тому, що *ExIsRestrictedCaller* перевіряє токен виклику, а *RtlIsSandboxToken* перевіряє вказаний об'єкт Token, який не обов'язково має бути токеном виклику.

Внутрішньо ці API виконують перевірку доступу для токена і надають доступ тільки в тому випадку, якщо токен не знаходиться в пісочниці. Лістинг 8.6 показує реалізацію цієї перевірки доступу в PowerShell.

Спочатку нам потрібно визначити фіктивний тип об'єкта ядра за допомогою команди *New-NtType*. Це дозволяє нам вказати загальне відображення для перевірки доступу. Ми вказуємо тільки значення *GenericRead* і *GenericAll*, оскільки доступ для запису та виконання в цьому контексті не є важливим. Зверніть увагу, що новий тип є локальним для PowerShell. Ядро нічого про нього не знає.

Лістинг 8.6. Перевірка доступу для токена пісочниці

```
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
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
```

Потім ми визначаємо дескриптор безпеки з NULL DACL і SID власника та групи, встановленими для користувача SYSTEM. Використання NULL DACL заборонить доступ до токенів *lowbox*, як описано в попередньому розділі, але не до будь-яких інших типів токенів пісочниці, таких як обмежені токени.

Для обробки інших типів токенів ми додаємо обов'язкову мітку ACE рівня **Medium** із політикою *NoReadUp*. В результаті будь-який токен із рівнем цілісності нижче **Medium** отримає відмову в доступі до маски, вказаної в полі *GenericRead* загального відображення. Токени *lowbox* ігнорують обов'язкову мітку **Medium**, але забезпечили захист цих токенів за допомогою NULL DACL. Зверніть увагу, що цей дескриптор безпеки не розглядає обмежені токени з рівнем цілісності **Medium** як токени пісочниці. Незрозуміло, чи це навмисне упущення, чи помилка в реалізації.

Тепер ми можемо виконати перевірку доступу за допомогою команди *Get-NtGrantedAccess*, використовуючи поточний токен, що не є токеном пісочниці.

Перевірка доступу проходить успішно, надаючи нам доступ *GenericRead*. Якщо ми повторимо перевірку з токеном, що має низький рівень цілісності, система відмовить нам у доступі, вказавши, що токен є пісочницею.

За кулісами API ядра викликають API *SeAccessCheck*, який поверне помилку, якщо у виклику є токен імперсонації рівня ідентифікації.

Тому ядро буде вважати деякі токени імперсонації пісочницею, навіть якщо реалізація в лістингу 8.6 свідчить про протилежне..

Коли будь-який API вказує, що виклик знаходиться в пісочниці, ядро змінює свою поведінку, щоб зробити наступне:

—Відображати тільки процеси та потоки, до яких можна отримати прямий доступ.

—Блокувати доступ до завантажених модулів ядра.

—Визначати відкриті дескриптори та адреси їхніх об'єктів ядра.

—Створювати довільні символічні посилання на файли та об'єкти диспетчера.

—Створювати новий токен з розширеними правами доступу.

Наприклад, у лістингу 8.7 ми запитуємо дескриптори, імітуючи токен з низьким рівнем цілісності, і отримуємо відмову в доступі.

Лістинг 8.7. Запит інформації про дескриптор під час імперсонації токена з низьким рівнем цілісності

```
PS> Invoke-NtToken -Current -IntegrityLevel Low { Get-NtHandle -ProcessId $pid }
Get-NtHandle : (0xC0000022) - {Access Denied}
A process has requested access to an object, ...
```

Хоча доступ до *ExIsRestrictedCaller* має лише код режиму ядра, ви можете отримати доступ до *RtlIsSandboxToken* у режимі користувача, оскільки він також експортується в NTDLL. Це дозволяє вам запитувати ядро за допомогою дескриптора токена, щоб дізнатися, чи вважає ядро його токеном пісочниці. API *RtlIsSandboxToken* відображає свій результат у властивості *IsSandbox* об'єкта Token, як показано в лістингу 8.8.

Об'єкт **Process**, повернений *Get-NtProcess*, має властивість *IsSandboxToken*. Внутрішньо ця властивість відкриває токен процесу і викликає IsSandbox.

Лістинг 8.8. Перевірка статусу токенів пісочниці

```
PS> Use-NtObject($token = Get-NtToken) {
        $token.IsSandbox
}
False
PS> Use-NtObject($token = Get-NtToken -Duplicate -IntegrityLevel Low) {
        $token.IsSandbox
}
True
```

Ми можемо використовувати цю властивість, щоб легко виявити, які процеси знаходяться в пісочниці, використовуючи, наприклад, скрипт в лістинг 8.9.

Лістинг 8.9. Перелік усіх процесів у пісочниці для поточного користувача

```
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
```

Ці перевірки пісочниці є важливою функцією для обмеження витоку інформації та обмеження небезпечних функцій, таких як символічні посилання, які збільшують шанси зловмисника вийти з пісочниці та отримати додаткові привілеї. Наприклад, блокування доступу до таблиці дескрипторів запобігає витоку адрес об'єктів ядра, які можуть бути використані для експлуатації вразливостей пошкодження пам'яті ядра.

Наразі ви вивчили три способи використання перевірки доступу, які не пов'язані з відкриттям ресурсу. На завершення цього розділу ми опишемо деякі команди, які спрощують перевірку доступу до низки окремих ресурсів.

8.4. АВТОМАТИЗАЦІЯ ПЕРЕВІРКИ ДОСТУПУ

В попередньому розділі наведено приклад, в якому за допомогою *Get-NtGranted Access* було визначено права доступу для набору об'єктів ядра. Якщо ви хочете перевірити інший тип ресурсів, наприклад файли, вам потрібно буде змінити цей скрипт, щоб використовувати команди для роботи з файлами.

Лістинг 8.10. Перелік команд *Get-Accessible*

```
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
```

Оскільки перевірка наданого доступу до ряду ресурсів є дуже важливою операцією, модуль PowerShell містить кілька команд для автоматизації цього процесу. Ці команди дозволяють швидко оцінити поверхню атаки на доступні ресурси в системі Windows. Всі вони починаються з Get-Accessible, і ви можете використовувати Get-Command для їх переліку, як показано в лістинг 8.10.

Ми повернемося до деяких із цих команд у наступних розділах. Тут ми зосередимося на команді *Get-AccessibleObject*, яку можна використовувати для автоматизації перевірки доступу до всієї OMNS. Ця команда дозволяє вказати

шлях OMNS для перевірки, а потім перелічує OMNS і повідомляє про максимальний наданий доступ або про те, чи можна надати певну маску доступу.

Ви також можете вказати, які токени використовувати для перевірки доступу. Команда може отримувати токени з наступного списку:

—Токени об'єктів;

—Токени процесів;

—Імена процесів;

—Ідентифікатори процесів;

—Рядки команд процесів.

Якщо під час виконання команди не вказано жодних опцій, вона буде використовувати поточний основний токен. Потім вона зробить перелік усіх об'єктів на основі шляху OMNS і виконає перевірку доступу для кожного вказаного токена. Якщо перевірка доступу пройде успішно, команда згенерує структурований об'єкт, що містить деталі результату. Приклад наведено в лістингу 8.11.

Лістинг 8.11. Отримання доступних об'єктів з каталогу OMNS

```
PS> Get-AccessibleObject -Path "\"

TokenId   Access                         Name
-------   ------                         ----
C5856B9 GenericExecute|GenericRead        \
```

Тут ми виконуємо команду щодо кореня OMNS і отримуємо три стовпці у вихідних даних:

TokenId - Унікальний ідентифікатор токена, що використовується для перевірки доступу;

Access - Наданий доступ, що відповідає загальним правам доступу;

Name - Назва перевіреного ресурсу.

Ми можемо використовувати **TokenId** для того, щоб розглянути результати для різних токенів, вказаних у команді.

Цей результат є лише частиною результату, отриманого за допомогою команди *Get-AccessibleObject*. Ви можете отримати решту інформації за допомогою таких команд, як Format-List. Ви також можете відобразити копію дескриптора безпеки, який використовується для перевірки доступу, за допомогою команди PowerShell *Format-NtSecurityDescriptor*, як показано в лістингу 8.12.

Оскільки ми запустили команду для каталогу, ви можете запитати, чи буде вона також відображати об'єкти, що містяться в каталозі. За замовчуванням, ні. Команда відкриває шлях як об'єкт і виконує перевірку доступу. Якщо ви хочете рекурсивна перевірити всі об'єкти в каталозі, вам потрібно вказати параметр Recurse.

Лістинг 8.12. Відображення дескриптора безпеки, що використовується для перевірки доступу

```
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)
```

Команда *Get-AccessibleObject* також приймає параметр *Depth*, який ви можете використовувати для вказання максимальної глибини рекурсії. Якщо ви виконуєте рекурсивну перевірку як користувач, що не є адміністратором, ви можете побачити багато попереджень, як у лістингу 8.13.

Лістинг 8.13. Попередження при рекурсивному переліку об'єктів

```
PS> Get-AccessibleObject -Path "\" -Recurse
WARNING: Couldn't access \PendingRenameMutex - Status: STATUS_ACCESS_DENIED
WARNING: Couldn't access \ObjectTypes - Status: STATUS_ACCESS_DENIED


--snip--
```

Ви можете вимкнути попередження, встановивши для параметра *WarningAction* значення *Ignore*, але майте на увазі, що вони намагаються вам щось повідомити. Щоб команда працювала, вона повинна відкрити кожен об'єкт і запитати його дескриптор безпеки. У режимі користувача для цього потрібно пройти перевірку доступу під час відкриття. Якщо у вас немає дозволу на відкриття об'єкта для доступу *ReadControl*, команда не може виконати перевірку доступу. Для отримання кращих результатів ви можете запустити команду як адміністратор, а для отримання оптимальних результатів запустіть її як користувач SYSTEM, використовуючи команду *Start-Win32ChildProcess* для запуску оболонки SYSTEM PowerShell.

За замовчуванням команда виконує перевірку доступу, використовуючи токен виклику. Але якщо ви виконуєте команду як адміністратор, вам, ймовірно, не потрібна така поведінка, оскільки майже всі ресурси надають адміністраторам повний доступ. Замість цього розгляньте можливість вказання довільних токенів для перевірки ресурсу. Наприклад, при виконанні як користувач з правами адміністратора наступна команда рекурсивна відкриває ресурси, використовуючи токен адміністратора, але виконує перевірку доступу з токеном не адміністратора з процесу *Explorer*:

```
PS> Get-AccessibleObject -Path \ -ProcessName explorer.exe -Recurse
```

Часто виникає потреба відфільтрувати список об'єктів для перевірки. Ви можете виконати перевірку доступу для всіх об'єктів, а потім відфільтрувати список, але це вимагатиме багато роботи, яку ви потім просто викинете. Щоб заощадити ваш час, команда Get-AccessibleObject підтримує кілька параметрів фільтрації:

TypeFilter - Список імен типів NT для перевірки;

Filter - Іменний фільтр, що використовується для обмеження об'єктів, які відкриваються. Може містити символ підстановки;

Include - Іменний фільтр, що використовується для визначення результатів, які слід включити у вихідні дані;

Exclude - Іменний фільтр, що використовується для визначення результатів, які слід виключити з вихідних даних;

Access - Маска доступу, що використовується для обмеження вихідних даних лише об'єктами з певним наданим доступом.

Наприклад, наступна команда знайде всі об'єкти **Mutant**, які можна відкрити за допомогою доступу *GenericAll*:

```
PS> Get-AccessibleObject -Path \ -TypeFilter Mutant -Access GenericAll -Recurse
```

За замовчуванням параметр *Access* вимагає, щоб перед виведенням результату був наданий повний доступ. Ви можете змінити це, вказавши *AllowPartial Access*, що дозволить вивести будь-який результат, який частково відповідає вказаному доступу. Якщо ви хочете побачити всі результати незалежно від наданого доступу, вкажіть *AllowEmptyAccess*.

8.5. ПРАКТИЧНІ ПРИКЛАДИ

Підсумуємо кілька прикладів, в яких використовуються команди, про які ви дізналися в цьому розділі.

8.5.1. Спрощення перевірки доступу до об'єкта

У попередньому розділі ми використовували команду *Get-NtGrantedAccess* для автоматизації перевірки доступу до об'єктів ядра та визначення їх максимального дозволеного доступу. Для цього спочатку потрібно було отримати дескриптор безпеки об'єкта. Потім ми передали це значення команді разом із типом об'єкта ядра, який потрібно перевірити.

Якщо у вас є дескриптор доступу до об'єкта, ви можете спростити виклик команди *Get-NtGrantedAccess*, вказавши об'єкт за допомогою параметра Object, як показано в лістингу 8.14.

Лістинг 8.14.Виконання перевірки доступу до об'єкта

```
PS> $key = Get-NtKey HKLM\Software -Win32Path -Access ReadControl
PS> Get-NtGrantedAccess -Object $key
QueryValue, EnumerateSubKeys, Notify, ReadControl
```

Використання параметру *Object* дозволяє уникнути необхідності вручну витягувати дескриптор безпеки з об'єкта і автоматично вибирає правильну загальну структуру відображення для типу об'єкта ядра. Це зменшує ризик помилок під час перевірки доступу до об'єкта.

8.5.2. Пошук об'єктів з розділами, доступними для запису

Система використовує об'єкти Section для спільного використання пам'яті процесами. Якщо привілейований процес встановлює слабкий дескриптор безпеки, процес з меншими привілеями може відкрити та змінити вміст розділу. Це може призвести до проблем із безпекою, якщо цей розділ містить довірені параметри, які можуть змусити привілейований процес виконати привілейовані операції.

Так було виявлено уразливість цього класу, CVE-2014-6349, у конфігурації пісочниці Internet Explorer. Ця конфігурація неправильно захищала спільний об'єкт *Section*, дозволяючи процесам Internet Explorer у пісочниці відкривати його та вимикати пісочницю повністю. Цю проблему було виявлено, виконавши перевірку доступу *MapWrite* до всіх іменованих об'єктів *Section*. У лістингу 8.15 ми автоматизуємо виявлення секцій, доступних для запису, за допомогою команди *Get-AccessibleObject*.

Лістинг 8.15. Перелік об'єктів *Section*, доступних для запису, для токена з низьким рівнем цілісності

```
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> $objs | ForEach-Object {
      Use-NtObject($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--
```

Почнемо з обчислення маски доступу для доступу *MapWrite* і перетворення її в загальний перелік доступу. Команда *Get-AccessibleObject* приймає тільки загальний доступ, оскільки вона заздалегідь не знає, які об'єкти ви, ймовірно, будете перевіряти. Потім ми дублюємо токен поточного користувача і встановлюємо його рівень цілісності на **Low**, створюючи просту пісочницю.

Ми передаємо токен і маску доступу до *Get-AccessibleObject*, виконуючи рекурсивну перевірку в каталозі *BaseNamedObjects* користувача, вказавши єдиний роздільник шляху для параметра Win32Path. Результати, повернуті

командою, повинні містити тільки розділи, які можна відкрити для доступу *MapWrite*.

Нарешті, ми перераховуємо список виявлених розділів, відображаючи їх імена та початковий вміст будь-якого виявленого об'єкта *Section*, доступного для запису.

Ми відкриваємо кожен названий розділ, відображаємо перші 4096 байт у пам'яті, а потім виводимо вміст у вигляді шістнадцяткового дампа. Відображаємо розділ як доступний для запису, оскільки можливо, що дескриптор безпеки об'єкта *Section* надає доступ *MapWrite*, але розділ був створений як доступний тільки для читання. У цьому випадку відображення *ReadWrite* завершиться помилкою. Ви можете використовувати цей скрипт для пошуку важливих розділів, доступних для запису. Вам не потрібно використовувати токен пісочниці. Може бути цікаво побачити розділи, доступні для звичайного користувача, які належать привілейованим процесам. Ви також можете використовувати це як шаблон для виконання такої ж перевірки для будь-якого іншого типу об'єкта ядра.

8.6. ВИСНОВКИ ДО РОЗДІЛУ 8

У цьому розділі ми розглянули кілька прикладів використання перевірки доступу поза межами відкриття ресурсу. Спочатку ми розглянули перевірки обходу, які використовуються для визначення, чи може користувач переходити по ієрархічному списку контейнерів, таких як каталоги об'єктів. Потім ми обговорили, як перевірки доступу використовуються, коли дескриптори дублюються між процесами, включаючи те, як це може створити проблеми безпеки, якщо об'єкт не має імені або налаштованого дескриптора безпеки.

Далі ми розглянули, як перевірка доступу може бути використана для визначення, чи токен викликаючого знаходиться в пісочниці. Ядро робить це для обмеження доступу до інформації або певних операцій, щоб ускладнити використання певних класів вразливостей безпеки. Нарешті, розглянули, як автоматизувати перевірки доступу для різних типів ресурсів за допомогою команд *Get-Accessible*. Було розглянуто основні параметри, спільні для всіх команд, і як їх використовувати для переліку доступних іменованих об'єктів ядра.

На цьому ми завершуємо розгляд процесу перевірки доступу. У наступному розділі ми розглянемо останню функцію SRM: аудит безпеки.

ПИТАННЯ ДЛЯ САМОКОНТРОЛЮ

1. Яка перевірка називається перевіркою обходу (traversal check)?
2. Коли виконується перевірка обходу (traversal check)?
3. Чому в деяких випадках користувач не може отримати доступ до своїх власних ресурсів?
4. Що визначає SRM привілей SeChangeNotifyPrivilege?

5. Як провести тестування *SeChangeNotifyPrivilege*?

6. Якщо привілей SeChangeNotifyPrivilege вимкнено, яку функцію викличе ядро для обмеженої перевірки замість повної перевірки доступу?

7. Що відбувається, коли цей дескриптор дублюється?

8. Які два API реалізує ядро, для визначення, чи знаходиться виклик у пісочниці?

9. Які команди містить модуль PowerShell для автоматизації перевірки доступу?

10. Як спростити перевірки доступу до об'єкта?

РОЗДІЛ 9. АУДИТ БЕЗПЕКИ

Процес перевірки доступу тісно пов'язаний з процесом аудиту. Адміністратор може налаштувати механізм аудиту системи для створення журналу ресурсів, до яких було здійснено доступ.

Кожна подія в журналі буде містити детальну інформацію про користувача та програму, які відкрили ресурс, а також про те, чи був доступ успішним чи ні. Ця інформація може допомогти нам виявити неправильні налаштування безпеки або зловмисний доступ до конфіденційних ресурсів.

У цьому короткому розділі ми спочатку обговоримо, де зберігається журнал доступу до ресурсів після його створення ядром. Потім ми опишемо, як системний адміністратор може налаштувати механізм аудиту. Нарешті, ми детально розповімо, як налаштувати окремі ресурси для створення подій журналу аудиту за допомогою SACL.

9.1. ЖУРНАЛ ПОДІЙ БЕЗПЕКИ

Windows генерує події журналу щоразу, коли перевірка доступу проходить успішно або завершується невдачею.

Ядро записує ці події журналу в журнал подій безпеки, доступ до якого мають лише адміністратори.

Під час перевірки доступу до ресурсів ядра Windows генерує такі типи подій аудиту. Журнал подій безпеки відображає їх за допомогою ідентифікатора події, вказаного в дужках:

- Відкрито дескриптор об'єкта (4656)
- Закрито дескриптор об'єкта (4658)
- Об'єкт видалено (4660)
- Дескриптор об'єкта дубльовано (4690)
- SACL змінено (4717)

Коли ми отримуємо доступ до ресурсів за допомогою системних викликів ядра, таких як NtCreateMutant, механізм аудиту автоматично генерує ці події. Але для подій аудиту, пов'язаних з об'єктами, ми повинні спочатку налаштувати два аспекти системи: ми повинні встановити системну політику для генерації подій аудиту і ми повинні ввімкнути ACE аудиту в SACL ресурсу. Давайте по черзі обговоримо кожну з цих вимог до конфігурації.

9.1.1. Налаштування політики аудиту системи

Більшість користувачів Windows не потребують збору інформації про аудит ресурсів ядра, тому політика аудиту за замовчуванням вимкнена. У корпоративних середовищах політика аудиту зазвичай налаштовується за допомогою політики безпеки домену, яку корпоративна мережа розподіляє між окремими пристроями.

Користувачі, які не входять до корпоративної мережі, можуть увімкнути політику аудиту вручну. Один із способів зробити це — редагувати локальну політику безпеки, яка виглядає так само, як політика безпеки домену, але застосовується тільки до поточної системи. Існує два типи політики аудиту: застаріла політика, яка використовувалася до Windows 7, і розширена політика аудиту. Рекомендується використовувати розширену політику аудиту, оскільки вона забезпечує більш детальне налаштування; ми не будемо далі обговорювати застарілу політику.

Якщо відкрити редактор локальної політики безпеки, виконавши команду *secpol.msc* у PowerShell, можна переглянути поточну конфігурацію розширеної політики аудиту, як показано на рис. 9.1.

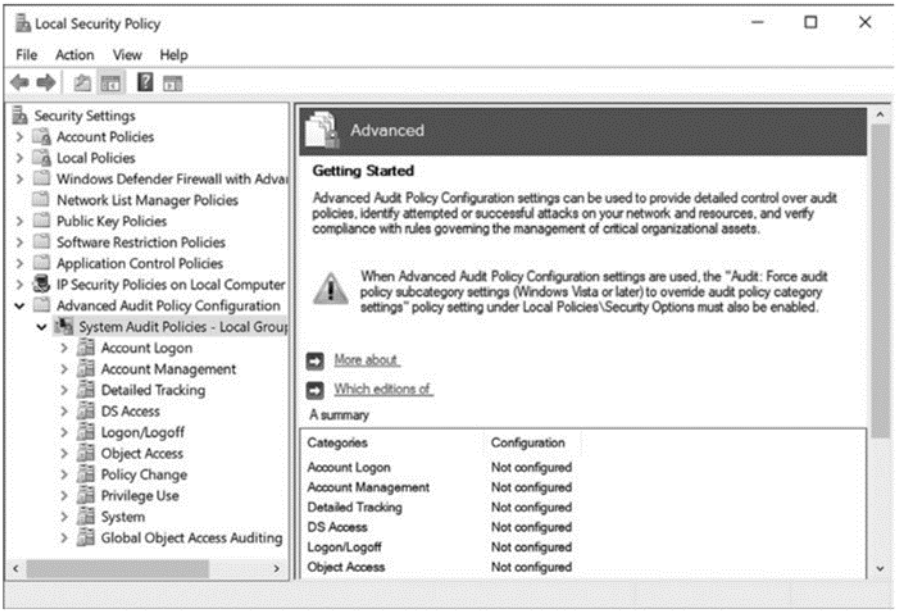


Рисунок 9.1. Редактор політики безпеки, що відображає розширену політику аудиту

Як бачите, категорії в політиці аудиту наразі не налаштовані. Щоб дізнатися, як генеруються події аудиту, ми використаємо PowerShell, щоб тимчасово ввімкнути необхідну політику аудиту та запустити приклад коду. Будь-які зміни, внесені за допомогою PowerShell, не будуть відображені в локальній політиці безпеки, яка буде відновлена під час наступної синхронізації (наприклад, під час перезавантаження або оновлення групової політики в корпоративній мережі). Ви можете примусово синхронізувати налаштування,

виконавши команду gpupdate.exe/force як адміністратор в PowerShell або в командному рядку.

Лістинг 9.1. Категорії політики аудиту верхнього рівня

```
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
```

Розширені політики аудиту мають два рівні: категорію верхнього рівня та кілька підкатегорій. Ви можете запитувати категорії верхнього рівня за допомогою Get-NtAuditPolicy, як показано в лістингу 9.1.

У вихідних даних ви можете побачити назву кожної категорії та кількість її підкатегорій. Кожна категорія також має пов'язаний GUID, але це значення за замовчуванням приховане. Щоб його побачити, виберіть властивість Id з вихідних даних команди, як показано в лістингу 9.2.

Список 9.2. Відображення GUID категорій

```
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--
```

Ви можете відобразити підкатегорії, використовуючи параметр *ExpandCategory*. У лістингу 9.3 вкажіть категорію System за ім'ям, а потім розгорніть вихідні дані, щоб відобразити підкатегорії.

Лістинг 9.3. Відображення підкатегорій політики аудиту

```
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
```

Ви також можете вибрати категорію, вказавши її GUID за допомогою параметра *CategoryGuid*. Політика аудиту базується на цих підкатегоріях.

Кожна політика підкатегорії може мати одне або кілька з наступних значень:

Unchanged - Політика не налаштована і не повинна змінюватися;

Success - Політика повинна генерувати події аудиту, коли ресурс, що підлягає аудиту, відкривається успішно;

Failure - Політика повинна генерувати події аудиту, коли ресурс, що підлягає аудиту, не може бути відкритий;

None - Політика ніколи не повинна генерувати події аудиту.

У лістингу 9.3 всі підкатегорії мають значення *Unchanged*, що означає, що політика не була налаштована. Аудит об'єктів ядра можна увімкнути, виконавши команди, показані в лістингу 9.4, від імені адміністратора.

Лістинг 9.4. Встановлення політики та перегляд отриманого списку політик аудиту *ObjectAccess*

```
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
```

У цьому випадку ми ввімкнули політики аудиту успішності та невдачі для всіх підкатегорій в *ObjectAccess*. Щоб виконати цю зміну, нам потрібні права *SeSecurityPrivilege*. Замість усієї категорії за назвою ми можемо встановити окрему підкатегорію, використовуючи параметр *SubCategoryName* або вказавши GUID за допомогою *SubCategoryGuid*.

Переконуємося, що політика аудиту налаштована правильно, вказавши параметр PassThru в якому міститься перелік змінених об'єктів *SubCategory*.

У вихідних даних відображаються деякі важливі політики аудиту, включаючи *FileSystem*, *Registry* і *KernelObject*, які дозволяють проводити аудит файлів, ключів реєстру та інших об'єктів ядра відповідно.

Ви можете виконати наступну команду як адміністратор, щоб відключити зміни, які ми зробили в лістингу 9.4:

```
PS> Set-NtAuditPolicy -Category ObjectAccess -Policy None
```

Якщо вам з якихось причин, не потрібно вмикати політику аудиту, краще вимкнути її після завершення експерименту.

9.1.2. Налаштування політики аудиту для окремих користувачів

Крім налаштування політики для всієї системи, можна також налаштувати політику аудиту для окремих користувачів. Цю функцію можна використовувати для налаштування аудиту для певного облікового запису користувача в тих випадках, коли система не визначає загальну політику аудиту. Її також можна використовувати для відключення аудиту для певного облікового запису.

Налаштування політики для окремих користувачів дещо відрізняються від налаштування політики для всієї системи:

Unchanged - Політика не налаштована. Після налаштування політику не рекомендується змінювати;

SuccessInclude - Політика повинна генерувати події аудиту в разі успіху, незалежно від системної політики;

SuccessExclude - Політика ніколи не повинна генерувати події аудиту в разі успіху, незалежно від системної політики;

FailureInclude - Політика повинна генерувати події аудиту в разі невдачі, незалежно від системної політики;

FailureExclude - Політика ніколи не повинна генерувати події аудиту в разі невдачі, незалежно від системної політики;

None - Політика ніколи не повинна генерувати події аудиту.

Щоб налаштувати політику для окремого користувача, можна вказати SID для параметра *User* під час використання команди *Set-NtAuditPolicy*. Цей SID повинен представляти обліковий запис користувача. Він не може представляти групу, таку як *Administrators*, або обліковий запис служби, такий як SYSTEM, інакше під час налаштування політики з'явиться помилка.

Лістинг 9.5 налаштовує політику для поточного користувача. Ви повинні запустити ці команди як користувач з правами адміністратора.

Лістинг 9.5. Налаштування політики аудиту для окремого користувача

```
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--
```

Тут ми вказуємо SID користувача в параметрі *User*, а потім вказуємо політику користувача *SuccessExclude*. Це дозволить виключити події аудиту успішності тільки для цього користувача. Якщо ви хочете видалити політику для окремого користувача, ви можете вказати політику користувача *None*:

```
PS> Set-NtAuditPolicy -Category ObjectAccess -User $sid -UserPolicy None
```

Ви також можете отримати список всіх користувачів, які налаштували політики, ви можете це зробити за допомогою параметра *AllUser* команди *Get-NtAuditPolicy*, як показано в лістингу 9.6.

Лістинг 9.6. Запит політик для усіх користувачів

```
PS> Get-NtAuditPolicy -AllUser
Name            User                 SubCategory Count
----            ----                 -----------------
System          GRAPHITE\admin       5
Logon/Logoff    GRAPHITE\admin       11
Object Access   GRAPHITE\admin       14
--snip--
```

Тепер ви знаєте, як перевіряти та встановлювати політики для системи та для окремого користувача. Далі ми розглянемо, як надати користувачам доступ, необхідний для перевірки та встановлення цих політик у системі.

9.2. ПОЛІТИКА АУДИТУ БЕЗПЕКИ

Для запиту або встановлення політики викликаюча сторона повинна мати у своєму *токені* увімкнену опцію *SeSecurityPrivilege*. Якщо ця опція не увімкнена, LSASS виконає перевірку доступу на основі дескриптора безпеки в конфігурації системи. Можна налаштувати такі права доступу в дескрипторі безпеки, щоб надати користувачеві можливість запитувати або встановлювати політику для системи або окремого користувача:

SetSystemPolicy - Дозволяє налаштувати політику аудиту системи;

QuerySystemPolicy - Дозволяє запитувати політику аудиту системи;

SetUserPolicy - Дозволяє налаштувати політику аудиту для окремого користувача;

QueryUserPolicy - Дозволяє запитувати політику аудиту для окремого користувача;

EnumerateUsers - Дозволяє отримувати перелік усіх політик аудиту для окремих користувачів;

SetMiscPolicy - Дозволяє налаштувати політику аудиту для різних типів запитів;

QueryMiscPolicy - Дозволяє запитувати політику аудиту для різних типів запитів.

Стандартний API аудиту не використовує права доступу *SetMiscPolicy* і *QueryMiscPolicy*, але оскільки вони визначені в Windows SDK, вони згадали їх тут для повноти.

Як адміністратор, ви можете отримати поточний дескриптор безпеки, увімкнувши *SeSecurityPrivilege* і використавши команду *Get-NtAuditSecurity*, як показано в лістинг 9.7.

Лістинг 9.7. Запит і відображення дескриптора аудиту безпеки

```
PS> Enable-NtTokenPrivilege SeSecurityPrivilege
PS> $sd = Get-NtAuditSecurity
PS> Format-NtSecurityDescriptor $sd -Summary -MapGeneric
<DACL>
BUILTIN\Administrators: (Allowed)(None)(GenericRead)
NT AUTHORITY\SYSTEM: (Allowed)(None)(GenericRead)
```

Передаємо запитуваний дескриптор безпеки до Format-*NtSecurityDescriptor*, щоб відобразити DACL. Зверніть увагу, що доступ до політики мають лише адміністратори та SYSTEM. Крім того, вони обмежені доступом GenericRead, який дозволяє користувачам запитувати політику, але не змінювати її. Таким чином, навіть адміністраторам потрібно буде ввімкнути *SeSecurityPrivilege*, щоб змінити політику аудиту, оскільки цей привілей обходить будь-яку перевірку доступу.

Користувач, якому не надано доступ для зчитування політики, все одно може запитувати розширені категорії та підкатегорії аудиту, які ігнорують дескриптор безпеки. Однак йому не буде надано доступ для зчитування налаштованих параметрів. *Get-NtAuditPolicy* поверне значення *Unchanged* для параметрів аудиту, які користувач не зміг зчитати.

Якщо ви хочете дозволити користувачам, які не є адміністраторами, змінювати розширену політику аудиту, ви можете змінити дескриптор безпеки за допомогою команди *Set-NtAuditSecurity*. Виконайте команди в лістингу 9.8 як користувач з правами адміністратора.

Лістинг 9.8. Модифікація дескриптора аудиту безпеки

```
PS> Enable-NtTokenPrivilege SeSecurityPrivilege
PS> $sd = Get-NtAuditSecurity
PS> Add-NtSecurityDescriptorAce $sd -Sid "LA" -Access GenericAll
PS> Set-NtAuditSecurity $sd
```

Спочатку перевіряємо існуючий дескриптор безпеки для політики аудиту і надаємо локальному адміністратору всі права доступу. Потім ми встановлюємо змінений дескриптор безпеки за допомогою команди *Set-NtAuditSecurity*. Тепер локальний адміністратор може перевіряти і змінювати політику аудиту без необхідності включення *SeSecurityPrivilege*.

Зазвичай не слід змінювати конфігурацію політики аудиту безпеки та, звичайно, не слід надавати всім користувачам права на запис. Зверніть увагу, що дескриптор безпеки не впливає на те, хто може запитувати або встановлювати сам дескриптор безпеки. Це можуть робити тільки абоненти з увімкненим *SeSecurityPrivilege*, незалежно від значень у дескрипторі безпеки.

9.2.1. Конфігурація ресурсу SACL

Для початку генерації подій аудиту недостатньо просто ввімкнути політики аудиту. Також потрібно налаштувати SACL об'єкта, щоб вказати правила аудиту, які слід використовувати. Щоб встановити SACL для об'єкта, нам знову потрібно ввімкнути *SeSecurityPrivilege*, а це може зробити лише адміністратор. Лістинг 9.9 демонструє процес створення об'єкта **Mutant** зі SACL.

Лістинг 9.9. Створення об'єкта Mutant зі SACL

```
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) {
    }
}
```

Почнемо з створення порожнього дескриптора безпеки, а потім додаємо один ACE аудиту до SACL. Інші типи ACE, які ми можемо додати, включають *AuditObject* та *AuditCallback*.

Обробка ACE аудиту дуже схожа на перевірку дискреційного доступу. SID повинен відповідати групі в **токені** виклику (включаючи будь-які SID *DenyOnly*),

а маска доступу повинна відповідати одному або декільком бітам наданого доступу. SID групи *Everyone* є особливим випадком. Він завжди буде збігатися, незалежно від того, чи доступний SID в токені.

На додаток до будь-яких звичайних прапорців ACE успадкування, таких як *InheritOnly*, *Audit* ACE повинен вказувати один або обидва прапорці *SuccessfulAccess* і *FailedAccess*, які надають аудиту умови, за яких він повинен генерувати запис аудиту.

Створюємо об'єкт **Mutant** з дескриптором безпеки, що містить SACL. Перед створенням об'єкта нам потрібно ввімкнути *SeSecurityPrivilege*. Якщо цього не зробити, створення об'єкта не вдасться. Щоб полегшити перегляд згенерованої події аудиту, ми також очищаємо журнал подій безпеки. Далі ми створюємо об'єкт, передаючи йому створений нами SACL, а потім знову відкриваємо його, щоб ініціювати генерацію журналу аудиту.

Тепер можемо запитати журнал подій безпеки за допомогою *Get-WinEvent*, передавши йому ідентифікатор події 4656, щоб знайти згенеровану подію аудиту (лістинг 9.10).

Лістинг 9.10. Перегляд події аудиту для об'єкта Mutant

```
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
```

Спочатку налаштовуємо фільтр для журналу подій безпеки та ідентифікатора події 4656. Потім використовуємо фільтр із *Get-WinEvent* і вибираємо текстове повідомлення події.

Вихідні дані починаються з цього текстового опису події, який підтверджує, що вона була згенерована у відповідь на відкриття дескриптора.

Далі йде розділ *Subject*, який містить інформацію про користувача, включаючи його SID та ім'я користувача. Щоб знайти ім'я користувача, ядро надсилає подію аудиту до процесу LSASS.

Далі йдуть деталі відкритого об'єкта. До них відносяться сервер об'єкта (Security, що представляє SRM), тип об'єкта (Mutant) і власний шлях до об'єкта, а також ідентифікатор дескриптора (номер дескриптора для об'єкта). Якщо ви запитуєте значення дескриптора, повернуте з системного виклику *NtCreateMutant*, воно повинно відповідати цьому значенню. Потім ми отримуємо деяку базову інформацію про процес і, нарешті, деяку інформацію про доступ, наданий дескриптору.

Як можна відрізнити успішні події від невдалих? Найкращий спосіб – скористатися властивістю *KeywordsDisplayNames*, яка має значення *Audit Success*, якщо дескриптор було відкрито, або *Audit Failure*, якщо дескриптор не вдалося відкрити. Приклад наведено в лістингу 9.11.

Лістинг 9.11. Отримання ключових DisplayNames для перегляду статусу

```
PS> Get-WinEvent -FilterHashtable $filter | Select-Object KeywordsDisplayNames
KeywordsDisplayNames
--------------------
{Audit Success}
{Audit Failure}
--snip--
```

Коли ви будете закривати дескриптор об'єкта, ви отримаєте подію аудиту з ідентифікатором події 4658, як показано в лістингу 9.12.

Лістинг 9.12. Перегляд події аудиту, що генеруються при закритті дескриптора об'єкта Mutant

```
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
```

Ви можете помітити, що інформація про закриття дескриптора об'єкта є дещо менш детальною, ніж інформація, яка генерується під час його відкриття. Ви можете вручну зіставити події відкриття та закриття дескриптора, використовуючи ідентифікатори дескрипторів, які повинні збігатися.

Можна вручну генерувати події аудиту об'єктів з режиму користувача за допомогою деяких додаткових системних викликів. Однак для цього потрібно мати привілей *SeAuditPrivilege*, який зазвичай надається тільки обліковому запису SYSTEM, а не звичайним адміністраторам.

Ви можете створити подію аудиту одночасно з перевіркою доступу за допомогою системного виклику *NtAccessCheckAndAuditAlarm*, який має всі ті самі параметри ACE об'єкта, що й звичайні перевірки доступу. Ви можете отримати доступ до нього за допомогою команди PowerShell *Get-NtGrantedAccess* з параметром *Audit*.

Ви також можете створити події вручну за допомогою системних викликів *NtOpenObjectAuditAlarm* і *NtCloseObjectAuditAlarm*, які PowerShell надає через команду *Write-NtAudit*. Виконайте команди в лістингу 9.13 як користувач SYSTEM, щоб вручну створити події журналу аудиту.

Лістинг 9.13. Ручне створення подій журналу аудиту

```
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 -Flags SuccessfulAccess, FailedAccess -KnownSid
World -MapGeneric
PS> $handle = 0x1234
PS> $r = Get-NtGrantedAccess $sd -Audit -SubsystemName "SuperSecurity"
 -ObjectTypeName "Badger"
-ObjectName "ABC" -ObjectCreation -HandleId $handle -PassResult
PS> Write-NtAudit -Close -SubsystemName "SuperSecurity" -HandleId $handle -GenerateOnClose:$r.GenerateOnClose
```

Почнемо з увімкнення *SeAuditPrivilege*, інакше решта скрипту не працюватиме. Цей привілей має бути увімкнений на основному токені. Ви не можете імітувати токен із привілеєм, тому ви повинні запустити екземпляр PowerShell як користувач SYSTEM.

Після увімкнення необхідних прав, створюємо дескриптор безпеки з SACL для аудиту успішних і невдалих спроб доступу. Генеруємо фальшивий ідентифікатор дескриптора. Це значення буде дескриптором ядра в звичайній події аудиту, але коли ми генеруємо подію з режиму користувача, це може бути будь-яке значення, яке нам подобається.

Потім ми можемо запустити перевірку доступу, вказавши параметр *Audit*, який вмикає інші параметри аудиту. Нам потрібно вказати параметри *SubsystemName*, *ObjectTypeName* та *ObjectName*, які можуть бути повністю довільними. Ми також вказуємо ID дескриптора.

У вихідних даних ми отримуємо результат перевірки доступу з одним додатковим властивістю: *GenerateOnClose*, яке вказує, чи потрібно записувати подію закриття дескриптора. Виклик команди *Write-NtAudit* із зазначенням параметра *Close* призведе до виклику системної функції *NtCloseObjectAuditAlarm* для генерації події. Ми робимо це, вказавши значення *GenerateOnClose* з результату. Якщо *GenerateOnClose* було *False*, нам все одно потрібно було б записати подію закриття, щоб завершити аудит, але фактична подія закриття не була б записана в журнал аудиту.

Якщо ви не отримуєте жодних подій аудиту при виконанні команд з лістингу 9.13, переконайтеся, що ви увімкнули аудит об'єктів, як ми це зробили в лістингу 9.4.

У списку типів ACE в таб.5.3 ви, мабуть, помітили тип *Alarm*, який пов'язаний з аудитом. У таблиці заначено, що ядро не використовує цей тип, і якщо ви прочитаєте технічну документацію Microsoft щодо типу ACE *Alarm*, то побачите фразу «Структура SYSTEM_ALARM_ACE зарезервована для майбутнього використання». Яке ж її призначення, якщо вона завжди була зарезервована?

Код ядра перевіряв тип Alarm ACE, починаючи з Windows NT 3.1, поки Microsoft не видалила цю перевірку в Windows XP. Розробники Windows навіть визначили варіанти *AlarmCallback*, *AlarmObject* та *AlarmObjectCallback*, хоча код не перевіряв їх у ядрі Windows 2000, де були введені об'єкти ACE. Зі старих версій ядра зрозуміло, що тип *Alarm* ACE оброблявся. Менш зрозуміло, чи міг *Alarm* ACE генерувати подію, яка підлягала моніторингу. Навіть у документації для версій Windows, які обробляли тип *Alarm* ACE, він позначений як непідтримуваний.

Ймовірно, Microsoft додала підтримку цього типу ACE до ядра Windows, але ніколи не реалізувала можливість передачі цих подій у реальному часі. З сучасними альтернативами ведення журналів, такими як Event Tracing for Windows (ETW), які надають набагато більш вичерпну інформацію про безпеку в реальному часі, шанси на те, що Microsoft знову введе *Alarm* ACE (або реалізує його варіанти) у майбутньому, є невеликими.

9.2.2. Конфігурація глобального SACL

Коректне налаштування SACL для кожного ресурсу може бути складним і трудомістким завданням. З цієї причини розширена політика аудиту дозволяє налаштувати глобальний SACL для файлів або ключів реєстру. Система буде використовувати цей глобальний SACL, якщо для ресурсу не існує SACL. А для ресурсів, які вже мають SACL, вона об'єднає глобальний SACL і SACL ресурсу. Оскільки ці широкі конфігурації аудиту можуть переповнити ваш журнал і перешкоджати вашій здатності відстежувати події, радимо використовувати глобальні SACL з обережністю.

Ви можете отримати глобальний SACL, вказавши значення *File* або *Key* для параметра *GlobalSacl* команди PowerShell *Get-NtAuditSecurity*. Ви також можете змінити глобальний SACL за допомогою команди *Set-NtAuditSecurity*, вказавши той самий параметр *GlobalSacl*. Щоб перевірити таку поведінку, запустіть команди в лістинг 9.14 як користувач з правами адміністратора.

Лістинг 9.14. Налаштування та запит глобального файлу SACL

```
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)
```

Спочатку ми створюємо дескриптор безпеки, що містить SACL з одним *Audit* ACE. Потім ми викликаємо *Set-NtAuditSecurity*, щоб встановити глобальний

SACL для файлу. Нарешті, ми запитуємо глобальний SACL, щоб переконатися, що він встановлений правильно.

Ви можете видалити глобальний SACL, передавши дескриптор безпеки з NULL SACL до *Set-NtAuditSecurity*. Щоб створити цей дескриптор безпеки, використовуйте наступну команду:

```
PS> $sd = New-NtSecurityDescriptor -NullSacl
```

9.3. ПРАКТИЧНІ ПРИКЛАДИ

Давайте підсумуємо на прикладах, що використовують команди, про які ви дізналися в цьому розділі.

9.3.1. Перевірка доступу до аудиту

Коли ви перевіряєте, чи не було порушено безпеку незахищеної частини системи Windows шкідливим кодом, варто переконатися, що налаштування безпеки не були змінені. Однією з перевірок, яку ви можете виконати, є визначення того, чи має користувач, який не є адміністратором, доступ, необхідний для зміни політики аудиту в системі.

Якщо користувач, який не є адміністратором, може змінити політику, він може вимкнути аудит і приховати свій доступ до конфіденційних ресурсів.

Можемо перевірити дескриптор безпеки політики аудиту вручну або за допомогою команди PowerShell *Get-NtGrantedAccess*. Виконайте команди в лістингу 9.15 як користувач з правами адміністратора.

Лістинг 9.15. Перевірка доступу за допомогою дескриптора

```
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
```

Спочатку ми запитуємо дескриптор безпеки політики аудиту та встановлюємо поля «Owner» і «Group». Ці поля необхідні для процесу перевірки доступу, але дескриптор безпеки, повернутий командою *Get-NtAuditSecurity*, їх не містить.

Потім ми можемо передати дескриптор безпеки команді *Get-NtGrantedAccess*, щоб перевірити його відносно поточного токена адміністратора. Результат вказує, що викликаючий має доступ *GenericRead* до

політики аудиту, що дозволяє йому запитувати політику, але не встановлювати її без активації *SeSecurityPrivilege*.

Нарешті, ми можемо видалити групу **Administrators** з токена, створивши відфільтрований токен із прапором *LuaToken*. Виконання перевірки доступу з фільтрованим токеном вказує на те, що він не має доступ до політики аудиту (навіть доступ для читання). Якщо ця друга перевірка повертає статус, відмінний від STATUS_ACCESS_DENIED, можна зробити висновок, що дескриптор безпеки політики аудиту за замовчуванням був змінений, і варто перевірити, чи було це зроблено навмисно або навить зловмисно.

9.3.2. Пошук ресурсів за допомогою Audit ACEs

Більшість ресурсів не налаштовані з SACL, тому ви можете бути зацікавлені в переліку ресурсів вашої системи, які мають SACL. Це допоможе вам зрозуміти, які ресурси можуть генерувати події журналу аудиту. Лістинг 9.16 надає простий приклад, в якому ми знаходимо ці ресурси. Виконайте команди від імені адміністратора.

Лістинг 9.16. Пошук процесів із налаштованими SACL

```
PS> Enable-NtTokenPrivilege SeDebugPrivilege, SeSecurityPrivilege
PS> $ps = Get-NtProcess -Access QueryLimitedInformation, AccessSystemSecurity -FilterScript {
          $sd = Get-NtSecurityDescriptor $_ -SecurityInformation Sacl
          $sd.HasAuditAce
}
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
          - Access: VmRead
          - Flags : SuccessfulAccess, FailedAccess
PS> $ps.Close()
```

Тут зосереджуємося на об'єктах **Process**, але ви можете застосувати цей самий підхід до інших типів ресурсів.

Спочатку ми відкриваємо всі процеси для *QueryLimitedInformation* та *AccessSystemSecurity access*. Застосовуємо фільтр до процесів, запитуючи SACL з об'єкта **Process**, а потім повертаємо значення властивості *HasAuditAce*. Ця властивість вказує, чи має дескриптор безпеки хоча б один ACE аудиту.

Потім ми передаємо результати, отримані за допомогою команди *Get-NtProcess*, в *Format-NtSecurityDescriptor* для відображення SACL. У даному випадку є тільки один запис для процесу LSASS. Ми бачимо, що ACE аудиту реєструє подію щоразу, коли процес LSASS відкривається для доступу *VmRead*.

Дана політика є стандартною конфігурацією аудиту в Windows, яка використовується для виявлення доступу до процесу LSASS. Право доступу *VmRead* дозволяє користувачеві читати віртуальну пам'ять процесу, а даний ACE

призначений для виявлення отримання вмісту пам'яті LSASS, який може містити паролі та інші дані для автентифікації. Якщо процес відкритий для будь-якого іншого права доступу, запис в журналі аудиту не буде створено.

9.4. ВИСНОВКИ ДО РОЗДІЛУ 9

У цьому розділі розглянули основи аудиту безпеки. Розпочали з опису журналу подій безпеки та типів записів журналу, які ви можете знайти під час аудиту доступу до ресурсів. Далі розглянули налаштування політики аудиту та встановлення розширених політик аудиту за допомогою команди *Set-NtAuditPolicy*. Також обговорили, як Windows контролює доступ до політики аудиту, та важливість привілею *SeSecurityPrivilege*, який використовується майже для всіх налаштувань, пов'язаних з аудитом.

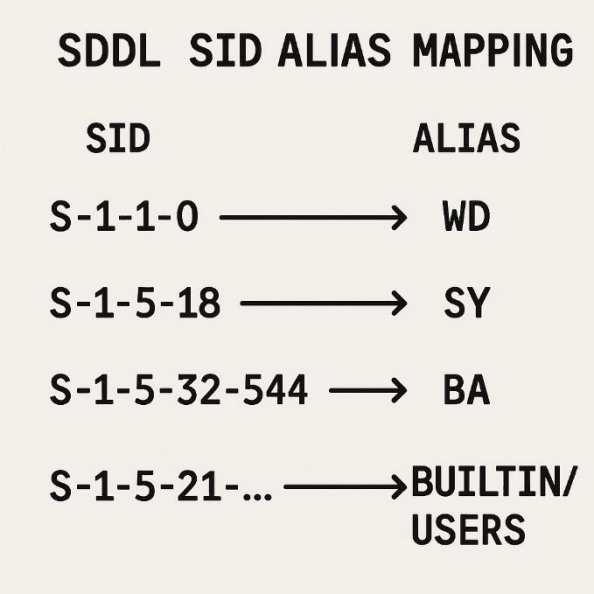
Щоб увімкнути аудит об'єкта, ми повинні змінити SACL, щоб визначити правила для генерації подій, дозволених політикою. Ознайомилися з прикладами автоматичної генерації подій аудиту за допомогою SACL та вручну під час перевірки доступу в режимі користувача.

Наразі в перший частині посібника розглянуто всі аспекти SRM: *токени доступу безпеки, дескриптори безпеки, перевірку доступу та аудит*. У наступній частині розглянемо різні механізми автентифікації в системі Windows.

ПИТАННЯ ДЛЯ САМОКОНТРОЛЮ

1. Коли Windows генерує події журналу?
2. Як генеруються події аудиту?
3. Які рівні мають розширені політики аудиту?
4. Як переконатися, що політика аудиту налаштована правильно?
5. Як налаштувати політики аудиту для окремих користувачів?
6. Для чого потрібно налаштувати SACL об'єкта?
7. Як переглянути події аудиту, що генеруються при закритті дескриптора об'єкта?
8. Як налаштувати та виконати запит глобального файлу SACL?
9. Як здійснити пошук ресурсів за допомогою Audit ACEs?
10. Як перевірити доступу до аудиту?

ДОДАТОК ВІДПОВІДНОСТІ ПСЕВДОНІМІВ SID SDDL ДО SID



У розділі 5 було представлено формат мови визначення дескрипторів безпеки (SDDL) для подання дескриптора безпеки у вигляді рядка та наведено кілька прикладів двосимвольних псевдонімів, які Windows підтримує для відомих SID SDDL. Хоча Microsoft документує формат SDDL для SID, вона не надає жодного ресурсу з лістингом усіх коротких рядків псевдонімів SID. Єдиним доступним ресурсом є заголовок *sddl.h* у Windows SDK.

Цей заголовок визначає API Windows, які програміст може використовувати для маніпулювання рядками формату SDDL, і надає список коротких рядків псевдонімів SID.

Таб. А містить короткі псевдоніми разом з іменами та повними SID, які вони представляють. Таблиця була витягнута з заголовка, наданого з SDK для Windows 11 (build 22621), який повинен бути канонічним списком на момент написання. Зверніть увагу, що деякі псевдоніми SID працюють тільки якщо ви підключені до доменної мережі. Ви можете ідентифікувати їх за допомогою заповнювача <DOMAIN> в імені SID, який слід замінити на ім'я домену, до якого підключена система. Також замініть заповнювач <DOMAIN> в рядку SDDL SID на унікальний доменний SID.

Таблиця А

Підтримувані відповідності псевдонімів SID SDDL до SID

SID псевдонім	Ім'я	SDDL SID
AA	BUILTIN\Access Control Assistance Operators	S-l-5-32-579
AC	APPLICATION PACKAGE AUTHORITY\ALL APPLICATION PACKAGES	S-l-15-2-1
AN	NT AUTHORITY\ANONYMOUS LOGON	S-l-5-7
AO	BUILTIN\Account Operators	S-l-5-32-548
AP	<DOMAIN>\Protected Users	S-l-5-21-<DOMAIN>-525
AS	Authentication authority asserted identity	S-l-18-1
AU	NT AUTHORITY\Authenticated Users	S-l-5-11
BA	BUILTIN\Administrators	S-l-5-32-544
BG	BUILTIN\Guests	S-l-5-32-546

SID псевдонім	Ім'я	SDDL SID
BO	BUILTIN\Backup Operators	S-l-5-32-551
BU	BUILTIN\Users	S-l-5-32-545
CA	<DOMAIN>\Cert Publishers	S-l-5-21-<DOMAIN>S17
CD	BUILTIN\Certificate Service DCOM Access	S-l-5-32-574
CG	CREATOR GROUP	S-l-3-1
CN	<DOMAIN>\Cloneable Domain Controllers	S-l-5-21-<DOMAIN>-522
CO	CREATOR OWNER	S-l-3-0
CY	BUILTIN\Cryptographic Operators	S-l-5-32-569
DA	<DOMAIN>\Domain Admins	S-l-5-21-<DOMAIN>-512
DC	<DOMAIN>\Domain Computers	S-l-5-21-<DOMAIN>-515
DD	<DOMAIN>\Domain Controllers	S-l-5-21-<DOMAIN>S16
DG	<DOMAIN>\Domain Guests	S-l-5-21-<DOMAIN>-51A
DU	<DOMAIN>\Domain Users	S-l-5-21-<DOMAIN>-513
EA	<DOMAIN>\Enterprise Admins	S-l-5-21-<DOMAIN>S19
ED	NT AUTHORITY\ENTERPRISE DOMAIN CONTROLLERS	S-l-5-9
EK	<DOMAIN>\Enterprise Key Admins	S-l-5-21-<DOMAIN>-527
ER	BUILTIN\Event Log Readers	S-l-5-32-573
ES	BUILTIN\RDS Endpoint Servers	S-l-5-32-576
HA	BUILTIN\Hyper-V Administrators	S-l-5-32-578
HI	Mandatory Label\High Mandatory Level	S-l-16-12288
IS	BUILTINMISJUSRS	S-l-5-32-568
IU	NT AUTHORITY\INTERACTIVE	S-l-5-4
KA	<DOMAIN>\Key Admins	S-l-5-21-<DOMAIN>-526
LA	<DOMAIN>\Administrator	S-l-5-21-<D0MAIN>-500
LG	<DOMAIN>\Guest	S-l-5-21-<D0MAIN>-501
LS	NT AUTHORITY\LOCAL SERVICE	S-l-5-19
LU	BUILTIN\Performance Log Users	S-l-5-32-559
LW	Mandatory Label\Low Mandatory Level	S-l-16-4096

SID псевдонім	Ім'я	SDDL SID
ME	Mandatory Label\Medium Mandatory Level	S-l-16-8192
MP	Mandatory Labe\Medium Plus Mandatory Level	S-l-16-8448
MS	BUILTIN\RDS Management Servers	S-l-5-32-577
MU	BUILTIN\Performance Monitor Users	S-l-5-32-558
NO	BUILTIN\Network Configuration Operators	S-l-5-32-556
NS	NT AUTHORITY\NETWORK SERVICE	S-l-5-20
NU	NT AUTHORITY\NETWORK	S-l-5-2
OW	OWNER RIGHTS	S-l-3-4
PA	<DOMAIN>\Group Policy Creator Owners	S-l-5-21-<DOMAII\l>-52O
PO	BUILTIN\Print Operators	S-I-5-32-55O
PS	NT AUTHORITY\SELF	S-1-5-1O
PU	BUILTIN\Power Users	S-l-5-32-547
RA	BUILTIN\RDS Remote Access Servers	S-l-5-32-575
RC	NT AUTHORITY\RESTRICTED	S-l-5-12
RD	BUILTIN\Remote Desktop Users	S-l-5-32-555
RE	BUILTIN\Replicator	S-l-5-32-552
RM	BUILTIN\Remote Management Users	S-I-5-32-58O
RO	<DOMAIN>\Enterprise Read-only Domain Controllers	S-l-5-21-<DOMAIN>-49B
RS	<DOMAIN>\RAS and IAS Servers	S-l-5-21-<DOMAIN>-553
RU	BUILTIN\Pre-Windows 2000 Compatible Access	S-l-5-32-554
SA	<DOMAIN>\Schema Admins	S-1-5-21-<D0MAIN>-518
SI	Mandatory Label\System Mandatory Level	S-l-16-16384
SO	BUILTIN\Server Operators	S-l-5-32-549

SID псевдонім	Ім'я	SDDL SID
SS	Service asserted identity	S-l-18-2
SU	NT AUTHORITY\SERVICE	S-l-5-6
SY	NT AUTHORITY\SYSTEM	S-l-5-18
UD	NT AUTHORITY\USER MODE DRIVERS	S-l-5-84-0-0-0-0-0
WD	Everyone	S-l-1-0
WR	NT AUTHORITYWVRITE RESTRICTED	S-l-5-33

СПИСОК ЛІТЕРАТУРНИХ ДЖЕРЕЛ ДЛЯ ПОГЛИБЛЕНОГО ВИВЧЕННЯ

1. Микитишин А.Г. Операційні системи: консп. лекц. / укл. А.Г. Микитишин, І.В. Чихіра. – Тернопіль: ТНТУ імені Івана Пулюя, 2016. – 107 с.
2. PowerShell. URL: https://learn.microsoft.com/uk-ua/powershell/scripting/learn/ps101/00-introduction?view=powershell-7.5 (дата звернення: 10.08.2025)
3. PowerShell Documentation. URL: https://learn.microsoft.com/uk-ua/powershell. (дата звернення: 10.08.2025)
4. Гаркуша І.М. Конспект лекцій з дисципліни “Операційні системи” для студентів галузі знань 12 “Інформаційні технології” спеціальності 126 “Інформаційні си- стеми та технології”. – Д.: НТУ «ДП», 2020. – 73 с.
5. Зайцев В. Г. Операційні системи: навч. посіб. для студ. / В. Г. Зайцев, І. П. Дробязко; КПІ ім. Ігоря Сікорського. – Електронні текстові дані – Київ: КПІ ім. Ігоря Сікорського, 2019. – 240 с.
6. Авраменко В. С., Авраменко А. С. Основи операційних систем. Навчальний посібник. – Черкаси: ЧНУ імені Богдана Хмельницького, 2018. – 524 с.: іл. ISBN 966-552-157-8
7. Шеховцов В. А. Операційні системи / В. А. Шеховцов. – Київ : Видавнича група BHV, 2005. – 576 с.: іл.
8. Погребняк Б. І. Операційні системи : навч. посібник / Б. І. Погребняк, М. В. Булаєнко ; Харків. нац. ун-т міськ. госп-ва ім. О. М. Бекетова. – Харків : ХНУМГ ім. О. М. Бекетова, 2018. – 104 с.
9. Федотова-Півень І.М. Операційні системи: навчальний посібник. [за ред. В.М. Рудницького] / І.М. Федотова-Півень, І.В. Миронець, О.Б. Півень, С.В. Сисоєнко, Т.В. Миронюк; Черкаський державний технологічний університет. – Харків: ТОВ «ДІСА ПЛЮС», 2019. – 216 с.
10. Бондаренко М.Ф. Операційні системи: навчальний посібник. / М.Ф. Бондаренко, О.Г. Качко - Харків: Компанія СМІТ, 2008 - 432 с.
11. Каплун, В. А. Захист програмного забезпечення : навчальний посібник. Ч. 2 / В. А. Каплун, О. В. Дмитришин, Ю. В. Баришев ; ВНТУ. – Вінниця : ВНТУ, 2016. – 106 с.
12. Байдачний, С. Windows 10 для C# розробників [Текст]. Кн. 2 / С. Байдачний, М. Остапчук. - Київ : ІТ-книга, 2016. - 312 с.
13. Вишня В. Б. Основи інформаційної безпеки : навч. посібник / В. Б. Вишня, О. С. Гавриш, Е. В. Рижков. Дніпро : Дніпроп. держ. ун-т внутріш. справ, 2020. 128 с.
14. Кібербезпека : сучасні технології захисту. Навчальний посібник для студентів вищих навчальних закладів. / С. Е. Остапов, С. П. Євсеєв, О.Г. Король. – Львів: «Новий Світ- 2000», 2020 . – 678 с.

15. Бурячок В. Л. Основи інформаційної та кібернетичної безпеки. [Навчальний посібник]. / В. Л. Бурячок , Р. В. Киричок, П. М. Складанний – К. , 2018. – 320 с.
16. William Stallings. Operating Systems: Internals and Design Principles, 9th Edition. – Print-ISBN: 978-1-292-45966-0, E-ISBN: 978-1-292-72789-9 Pages: 1184.

Додатковий список літературних джерел

17. Andrew Tanenbaum, Herbert Bos, Modern Operating Systems, Global Edition 5rd Edition. – Pearson, 2007. – 1104 p. ISBN-10: 0136006639, ISBN-13: 978-0136006633.
18. Pavel Yosifovich, Mark Russinovich, David Solomon, Alex Ionescu. Windows Internals, Part 1: System architecture, processes, threads, memory management, and more, 7th Edition – Microsoft Press, 2017. – 800 p. ISBN-10: 9780735684188, ISBN-13: 978-0735684188.
19. Evi Nemeth. UNIX and Linux System Administration Handbook, 5th Edition / Evi Nemeth, Garth Snyder, Trent Hein, Ben Whaley, Dan Mackin. – Addison-Wesley Professional, 2017. – 1232 p. ISBN-10: 0134277554, ISBN-13: 978-0134277554.
20. Kevin Wilson. MacOS Fundamentals: Catalina Edition: The Step-by-step Guide to Using your Mac. – Independently published, 2019. – 335 p. ISBN-10: 1708721118, ISBN-13: 978-1708721114.
21. Алгоритми і структури даних: посібник / Н. Б. Шаховська, Р. О. Голощук; за заг. ред. В. В.Пасічника. - К.: Магнолія 2006, 2010. - 215 с.: рис., схеми. - (Комп'ютинг). - Бібліогр.: с. 200
22. CIS Microsoft Windows 10 Enterprise (Release 1809) Benchmark v1.6.0 URL: https://www.cisecurity.org/benchmark/microsoft_windows_desktop/ (дата звернення: 10. 08.2025)
23. Bhatt A. Introduction to Hashing and how to retrieve Windows 10 password hashes. URL: https://medium.com/@anunayb007/introduction-to-hashing- and-how-to-retrieve-windows-10-password-hashes-9c8637decaef . (дата звернення: 10. 08.2025)
24. Shepherd J.Ultimate authentication playbook. URL: https://www.okta.com/blog/2019/02/the-ultimate-authentication-playbook/ (дата звернення: 10.08.2025).
25. Authentication type. URL: https://www.sciencedirect.com/topics/computer-science/authentication-type (дата звернення: 10. 08.2025)
26. Registry Key Security and Access Rights. Електронний ресурс. URL: https://docs.microsoft.com/en-us/windows/win32/sysinfo/registry-key-security-and-access-rights (дата звернення: 10. 08.2025)
27. Cryptography Hash functions. Електронний ресурс. URL: https://www.tutorialspoint.com/cryptography/cryptography_hash_functions.htm (дата звернення: 10. 08.2025)

28. Microsoft Cryptographic Service Providers. Електронний ресурс. URL: https://docs.microsoft.com/en-us/windows/win32/seccrypto/microsoft-cryptographic-service-providers . (дата звернення: 10. 08.2025)
29. Kambire, M. K., Gaikwad, P. H., Gadilkar, S. Y., & Funde, Y. A: An improved framework for tamper detection in databases. Int. J. Comput. Sci. Inform. Technol. 6 (2015).
URL: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.664.9099& rep=rep1&type=pdf (дата звернення: 10. 08.2025)
30. Introduction to Microsoft Security Development Life Cycle (SDL) Threat Modeling.
Електронний ресурс. URL: https://download.microsoft.com/download/9/3/5/935520EC- D9E2-413E-BEA7-0B865A79B18C/Introduction_to_Threat_Modeling.ppsx (дата звернення: 10. 08.2025)
31. Сучасні інформаційні війни в соціальних онлайн-мережах / О. В. Курбан // Інформаційне суспільство. - 2016. - Вип. 23. - С. 85-90. URL: http://nbuv.gov.ua/UJRN/is_2016_23_15 (дата звернення: 10. 08.2025)
32. Указ Президента України від 25 лютого 2017 року N47/2017 «Про рішення Ради національної безпеки і оборони України від 29 грудня 2016 ро- ку «Про Доктрину інформаційної безпеки України»». Президент Украї- ни.URL: https://www.president.gov.ua/documents/472017-21374 (дата звернення: 10. 08.2025)
33. Stallings, William. Operating systems: internals and design principles / William Stallings. – 7th ed. Prentice Hall, New Jersey, 2012, p.769. ISBN-13:978-0-13-230998-1

НАВЧАЛЬНЕ ВИДАННЯ

Гладких Валерій Миколайович
Криворучко Олена Володимирівна
Зінченко Ольга Валеріївна

БЕЗПЕКА ОПЕРАЦІЙНОЇ СИСТЕМИ WINDOWS

НАВЧАЛЬНИЙ ПОСІБНИК
ЧАСТИНА 1

306