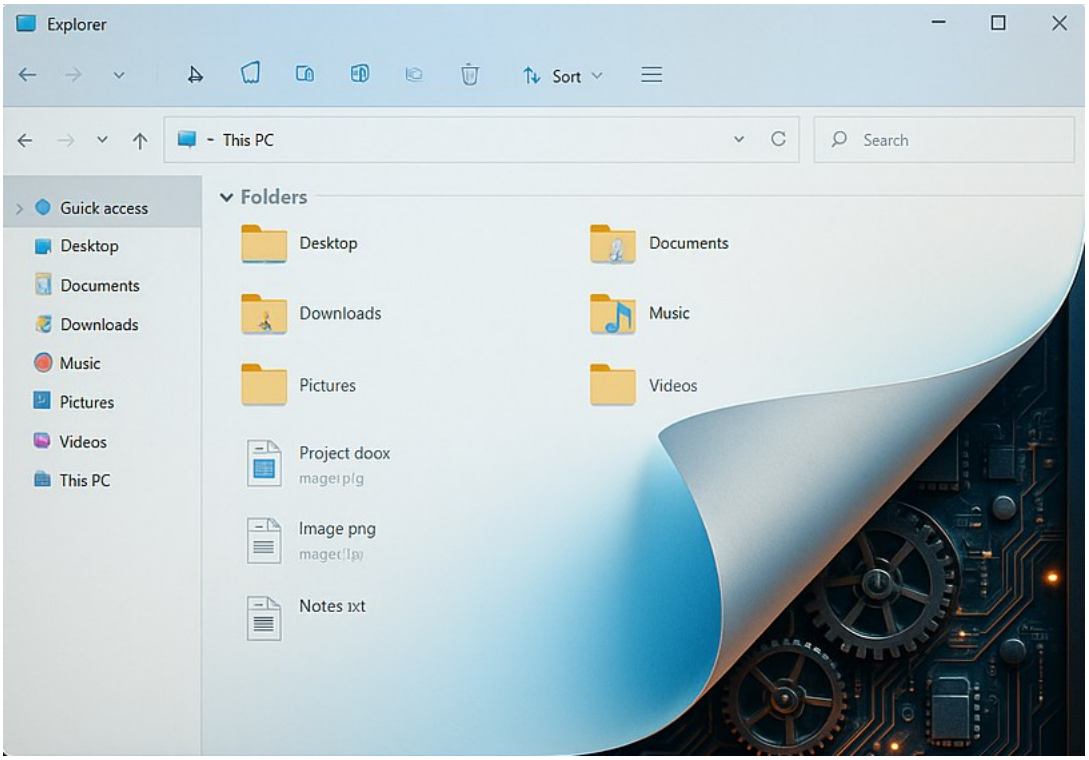
Обзор неявных возможностей дисковой подсистемы Windows 11



В своей повседневной жизни мы, как правило, пользуемся только самыми примитивными функциями Windows по работе с данными. Мы создаём документы, копируем файлы, переименовываем каталоги и делаем множество других операций, доступных на расстоянии пары кликов в стандартном интерфейсе Windows.

Но операционная способна на большее. В ней скрыт набор потрясающих возможностей, с помощью которых она элегантно решает свои внутренние задачи и которые могут существенно облегчить вам жизнь, особенно при решении нестандартных задач.

В этой статье мы поговорим о таком функционале Windows, как виртуальные жёсткие диски, ссылки, точки повторной обработки (reparse points), расширенные атрибуты (extended attributes), альтернативные потоки (alternative data streams), теневые копии (volume shadow copy) и кое-чем ещё.

ОГЛАВЛЕНИЕ

1. Предполётная подготовка

Материал статьи по каждому разделу будет включать небольшую описательную часть и набор практических примеров, из расчёта того, что вы будете воспроизводить их по ходу чтения статьи.

Во избежание ненужных проблем все учебные эксперименты следует проводить на виртуальной машине, работающей под управлением Microsoft Windows 11 Enterprise 25H2 с английской локализацией.

Приведённые в статье примеры кода рассчитаны на запуск из консоли PowerShell. В большинстве случаев для их выполнения потребуются права системного администратора.

2. Виртуальные жёсткие диски

Мало кто знает, но Windows имеет встроенные средства для создания и монтирования виртуальных жёстких дисков. Используемые при этом файлы-образы имеют формат .VHD / .VHDX, что делает их совместимыми с гипервизором Hyper-V и встроенной системой резервного копирования «Windows Image Backup» («Control panel > Create system image»).

Операционная система предлагает несколько способов работы с виртуальными жёсткими дисками:

1. Из графического пользовательского интерфейса, с помощью оснастки «Disk management» («Start > Disk Management > Action > Create VHD / Attach VHD»).
2. Из командной строки с помощью утилиты diskpart.

Есть, конечно, ещё и специализированные PowerShell-командлеты, такие как New-VHD, Mount-VHD и другие, но мы рассматривать их не будем, поскольку они часть Hyper-V, а ставить гипервизор только для работы с образами дисков не всегда удобно или допустимо.

Вне зависимости от способа работы с виртуальными жёсткими дисками, вам для этого необходимо будет обладать правами системного администратора.

С графическим пользовательским интерфейсом всё довольно просто: открыл одно окно – создал файл-образ, открыл другое – подключил его как устройство. Всё. Для разовой ручной работы этого более чем достаточно, а вот если необходима автоматизация, то без консольной утилиты diskpart не обойтись. На ней и сосредоточим дальнейшее внимание.

Первое что бросается в глаза при использовании diskpart – это то, что утилита предназначена для работы в интерактивном режиме. Что это значит? А то, что, запустив утилиту, вы как бы беседуете с ней. Вы пишите ей команды, а она пишет вам результаты выполнения.

По понятным причинам, интерактивный режим не подходит для автоматического использования, но, к счастью, утилиту можно запустить, передав ей скрипт на вход.

По большому счёту, скрипты diskpart – это просто последовательность команд, которые бы вы вводили руками, работая в интерактивном режиме, а так они хранятся в текстовом файле, и diskpart как бы набирает их за вас.

Формат запуска утилиты со скриптом на входе выглядит так:

```
diskpart.exe /s <имя файла скрипта>
```

Практическое изучение работы с виртуальными жёсткими дисками проведём на сквозном примере, где вначале создадим и инициализируем виртуальный жёсткий диск, а затем научимся отключать его и подключать снова.

2.1. Создание и первичная инициализация виртуального жёсткого диска

План работ:

- создадим виртуальный жёсткий диск размером 10 MB, с файлом-образом «c:\test\image.vhd»;
- затем подключим виртуальный жёсткий диск к операционной системе;
- создадим на нём файловую систему NTFS;
- сделаем его доступным из интерфейса Windows через букву X.

diskpart-скрипт, выполняющий всё задуманное, будет выглядеть так:

```
create vdisk file=c:\test\image.vhd maximum=10 type=fixed
select vdisk file=c:\test\image.vhd
attach vdisk
attributes disk clear readonly
convert gpt
create partition primary
format quick fs=ntfs label="vdisk"
assign letter="X"
```

По окончании работы скрипта в проводнике Windows и оснастке «Disk management» мы увидим созданный и готовый к работе виртуальный жёсткий диск (Рисунок 1).

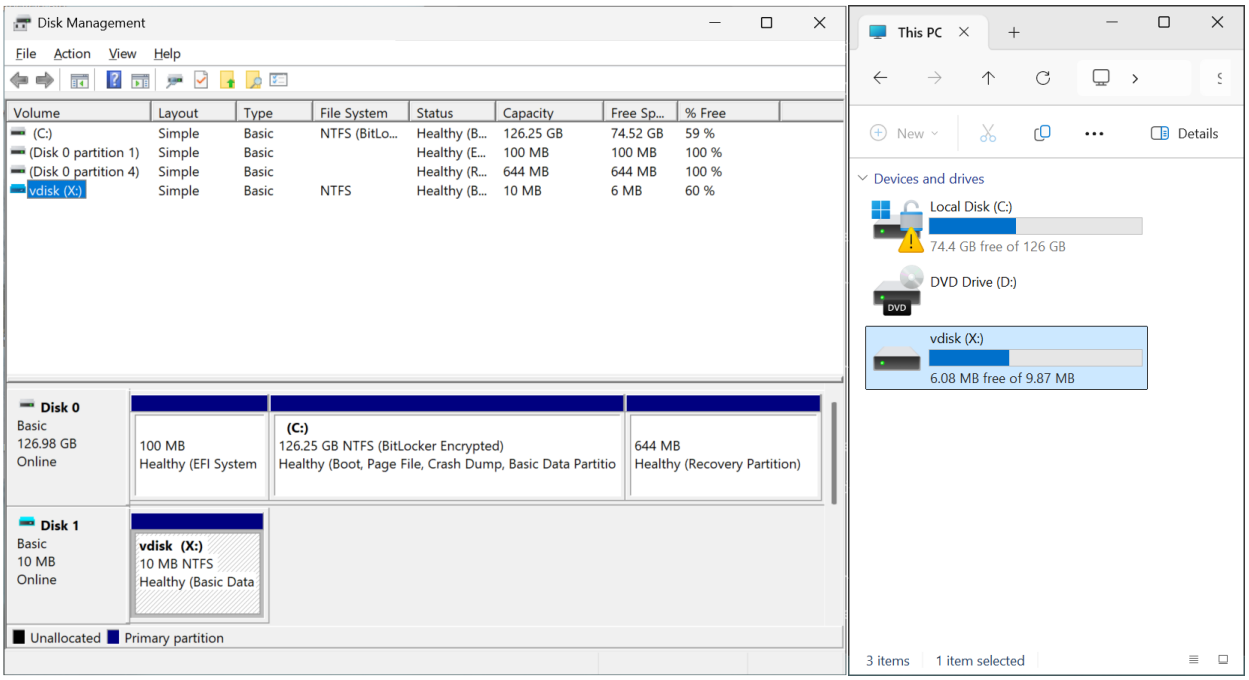


Рисунок 1

2.2. Отключение виртуального жёсткого диска

Смонтированный виртуальный жёсткий диск автоматически отключается при перезагрузке операционной системы. Если же потребуется отключить диск раньше, то в этом поможет diskpart-скрипт:

```
select vdisk file=c:\test\image.vhd
detach vdisk
```

2.3. Подключение виртуального жёсткого диска

Для повторного подключения виртуального жёсткого диска следует воспользоваться diskpart-скриптом:

```
select vdisk file=c:\test\image.vhd
attach vdisk
```

3. Ссылки и точки повторной обработки

В экосистеме Linux ссылки на файлы и каталоги – обычная вещь, с которой пользователи встречаются практически каждый день. В Windows же механизмом ссылок пользуются значительно реже, хотя по своим возможностям он ничуть не хуже того, что есть в Linux, а может быть даже и лучше.

Говоря о ссылках в Windows, практически любой пользователь первым делом вспомнит про файлы-ярлыки (*.lnk). Они очень удобны, когда на рабочем столе нужно сделать кнопку для быстрого запуска программы или открытия каталога, лежащего где-то в недрах файловой системы. Но, несмотря на все удобства, ярлыки всё же ненастоящие ссылки. Они не могут использоваться программами или скриптами, как новый путь до объекта файловой системы. Например, сделав ярлык на каталог «C:\Windows», мы не сможем сказать скрипту, чтобы он использовал его, как часть полного пути файла «C:\Windows\notepad.exe». Для этой цели нам необходимо вместо ярлыка воспользоваться настоящей ссылкой, то есть такой, которая сама является частью файловой системы.

В NTFS реализованы два класса ссылок:

1. Жёсткие ссылки (hard links).
2. Точки повторной обработки (reparse points), на базе которых реализованы:

   a) Символические ссылки (symbolic links).

   b) Точки соединения (junction points).

3.1. Жёсткие ссылки

Жёсткие ссылки NTFS – это альтернативные имена одного и того же файла (Рисунок 2).

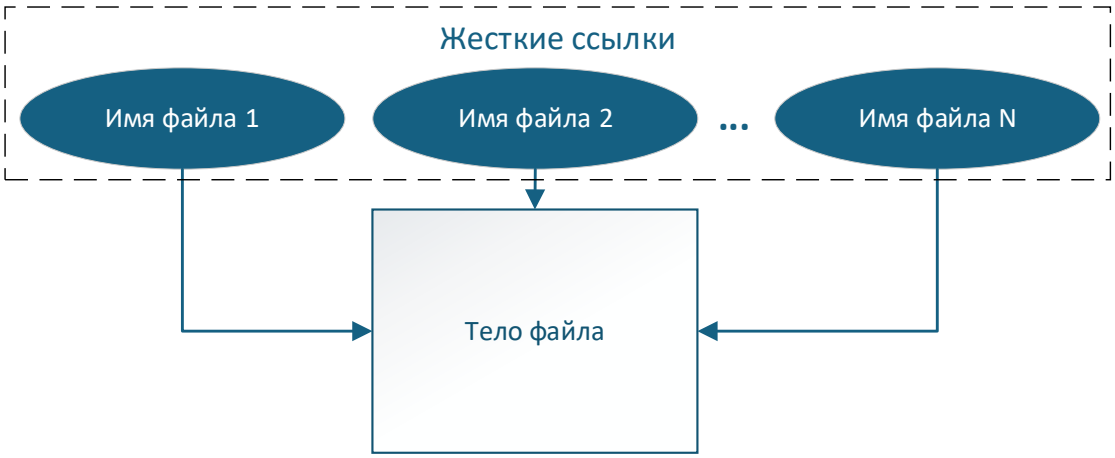


Рисунок 2

Если к обычному файлу сделать жёсткую ссылку, то операционная система начнёт показывать и старое, и новое имя файла как жёсткую ссылку. С точки зрения Windows, все жёсткие ссылки равноправны. Среди них нет какой-то одной, которую можно назвать главным именем файла и набора других, которые являются альтернативными именами. Все жёсткие ссылки одного и того же файла указывают на одни и те же данные.

3.1.1. Создание жёсткой ссылки

Пусть в каталоге «C:\TEST2» необходимо создать жёсткую ссылку с именем «hardlink_image.vhd» на файл «C:\TEST\image.vhd».

Чтобы это провернуть, нам потребуются права системного администратора, а с их наличием у нас есть как минимум два варианта:

1) Создание жёсткой ссылки с помощью встроенной в интерпретатор cmd команды mklink:

```
cmd /c mklink /H C:\test2\hardlink_image.vhd C:\test\image.vhd
```

2) Создание жёсткой ссылки с помощью PowerShell-командлета New-Item:

```
New-Item -ItemType HardLink -Path C:\test2\hardlink_image.vhd -Target
C:\test\hardlink_image.vhd
```

3.1.2. Перечисление всех жёстких ссылок в каталоге

Windows активно использует жёсткие ссылки в своей работе. Давайте перейдём в каталог «C:\Windows» и выведем список всех жёстких ссылок в этом каталоге и его подкаталогах.

```
Get-ChildItem -Recurse -Force -ErrorAction SilentlyContinue | Where-Object
LinkType -eq "HardLink" | Select-Object FullName
```

Результаты работы скрипта (Рисунок 3) содержат множество имён файлов, являющимися жёсткими ссылками. Секрет в том, что Windows использует механизм жёстких ссылок для реализации подсистемы Windows component storage (каталог WinSxS), предназначенной для управления версиями компонентов операционной системы. Одной из основных особенностей подсистемы является экономное использование дискового пространства, достигаемое за счёт того, что одинаковые файлы замещаются одним, а на месте дубликатов размещаются жёсткие ссылки.

Рисунок 3

Чтобы лучше это понять, выведем перечень исполняемых файлов, имеющих более двух жёстких ссылок (Рисунок 4).

```
Get-ChildItem *.exe -Recurse -Force -ErrorAction SilentlyContinue | Where-
Object { $_.Target.Count -gt 2 } | Select-Object FullName
```

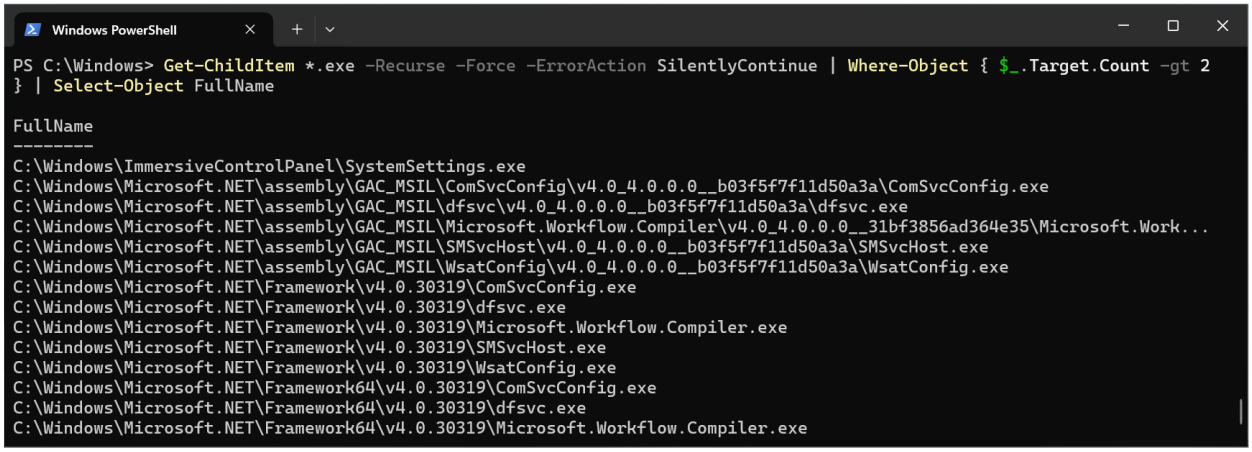


Рисунок 4

3.1.3. Просмотр всех жёстких ссылок файла

Теперь посмотрим перечень всех жёстких ссылок первого файла из списка – «C:\Windows\ImmersiveControlPanel\SystemSettings.exe»:

```
Get-Item C:\Windows\ImmersiveControlPanel\SystemSettings.exe | Select-Object
-ExpandProperty Target
```

Как мы видим из результатов работы скрипта (Рисунок 5), анализируемый файл идентичен файлам из нескольких обновлений, хранящихся в каталоге WinSxS, и для устранения дублирования они замещены жёсткими ссылками.

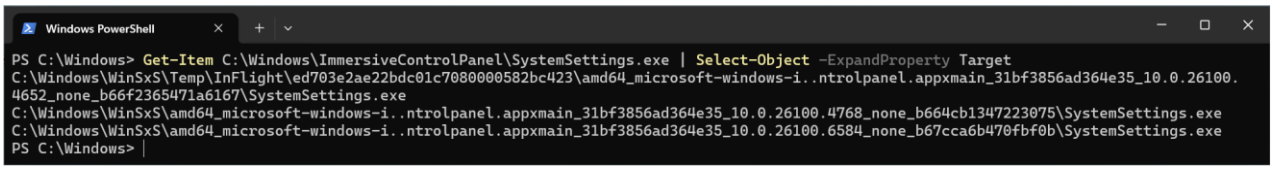


Рисунок 5

В то же время, если мы поищем все версии файла «C:\Windows\ImmersiveControlPanel\SystemSettings.exe», то увидим, что их больше, чем жёстких ссылок и часть из них отличается размером (Рисунок 6).

```
Get-ChildItem SystemSettings.exe -Recurse -Force -ErrorAction
SilentlyContinue | Select-Object Length, FullName
```

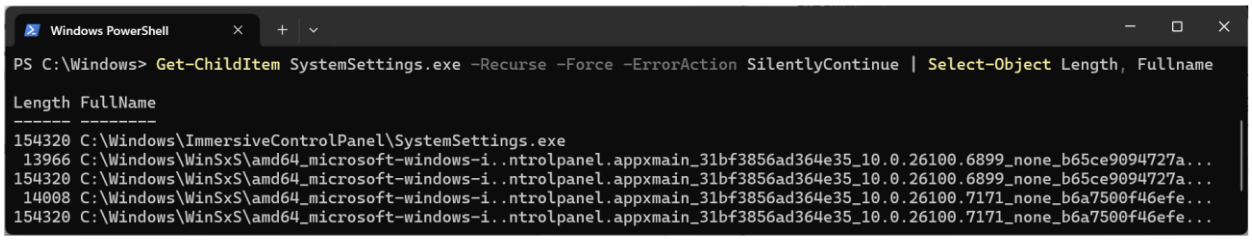


Рисунок 6

Вот таким интересным образом магия жёстких ссылок позволяет экономить дисковое пространство.

3.1.4. Важные особенности использования жёстких ссылок

1. Можно создать жёсткую ссылку только на файл. Жёсткие ссылки на каталоги не поддерживаются.
2. Создание жёсткой ссылки не уменьшает общий размер свободного места на диске. В то же время размер каталога, где хранится жёсткая ссылка, будет увеличен на размер «тела файла», адресуемого жёсткой ссылкой.
3. Жёсткие ссылки «живут» только в рамках одного тома (volume). Следовательно, нельзя создать на одном диске жёсткую ссылку на файл, находящийся на другом диске. Копирование жёсткой ссылки с одного диска на другой приведёт к копированию файла целиком.
4. Чтобы удалить файл, имеющий несколько жёстких ссылок, нужно удалить все эти ссылки.

3.2. Точки повторной обработки

Стандартный алгоритм операционной системы по предоставлению данных из файла включает: получение запроса от прикладного ПО, считывание необходимых данных с накопителя и передачу полученной информации запросившей программе.

При использовании NTFS стандартное поведение операционной системы можно изменить и делается это следующим образом (Рисунок 7):

0) К требуемому файлу или каталогу прикрепляется особая структура данных, называемая точкой повторной обработки. Важнейшей частью этой структуры является тег, определяющий, что будет делать операционная система при попытке считывания помеченного им объекта. В операционную систему инсталлируется файловый драйвер, обрабатывающий точки повторной обработки с указанным тегом.

1) Пользовательское приложение пытается прочитать файл или каталог, с прикреплённой точкой повторной обработки.

2) Операционная система получает запрос на доступ к данным, обнаруживает, что запрашиваемый объект помечен точкой повторной обработки.

3) Операционная система считывает тег, а затем передаёт обработку запроса в ассоциированный с тегом драйвер.

4) Драйвер отрабатывает запрос и возвращает результат операционной системе.

5) Операционная система передаёт полученные от драйвера данные прикладной программе.

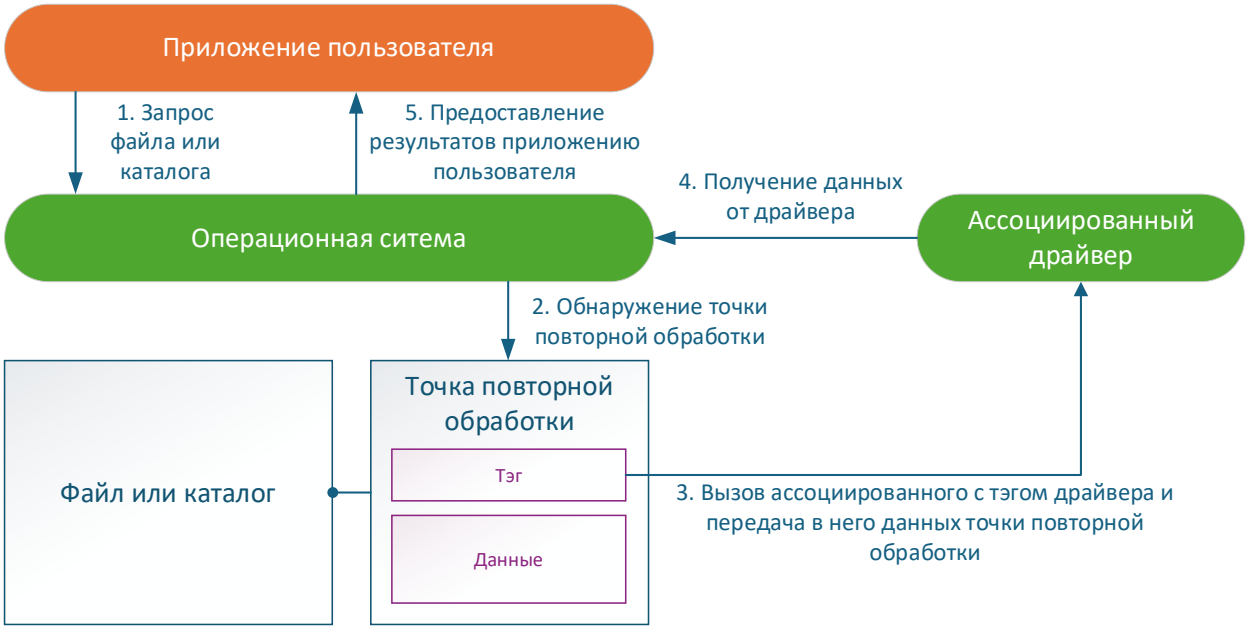


Рисунок 7

Рассмотрим работу этой схемы на примере работы облачного диска OneDrive.

Пользователь указывает OneDrive каталог, который хочет синхронизировать с облаком. OneDrive помечает в нём файлы точками повторной обработки. Затем, когда пользователь попытается открыть из этого каталога файл в текстовом редакторе, то операционная система вместо того, чтобы извлечь данные с диска, передаст запрос в драйвер OneDrive, который извлечёт данные из облачного хранилища или локального кэша и передаст их в операционную систему, а та, в свою очередь, вернёт их текстовому редактору. В результате всей этой внутренней работы пользователь может работать с синхронизируемыми файлами, как обычно, не задумываясь, где те находятся – в облаке или на локальном накопителе.

Точки повторной обработки относятся к данным, определяемым пользователем. Прикладное программное обеспечение может создавать произвольные точки повторной обработки, но довольно внушительный объём тегов Microsoft зарезервировала для своих нужд. К слову говоря, рассматриваемые далее символические ссылки и точки соединения реализуются с помощью тегов: «IO_REPARSE_TAG_SYMLINK» и «IO_REPARSE_TAG_MOUNT_POINT» соответственно.

Создание точек повторной обработки требует прав администратора, но есть и исключения – точки соединения, которые может создать рядовой пользователь.

3.2.1. Просмотр перечня точек повторной обработке в текущем каталоге

Посмотреть перечень всех объектов в текущем каталоге, к которым подсоединены точки повторной обработки, можно командой:

```
Get-ChildItem -Recurse -Force -Attributes ReparsePoint -ErrorAction
SilentlyContinue | Select-Object FullName
```

Если выполнить этот скрипт с административными правами в корне диска «C:», то увидим (Рисунок 8), что Windows очень активно использует точки повторной обработки в своей работе. Что это за точки и зачем они нужны, мы разберём далее.

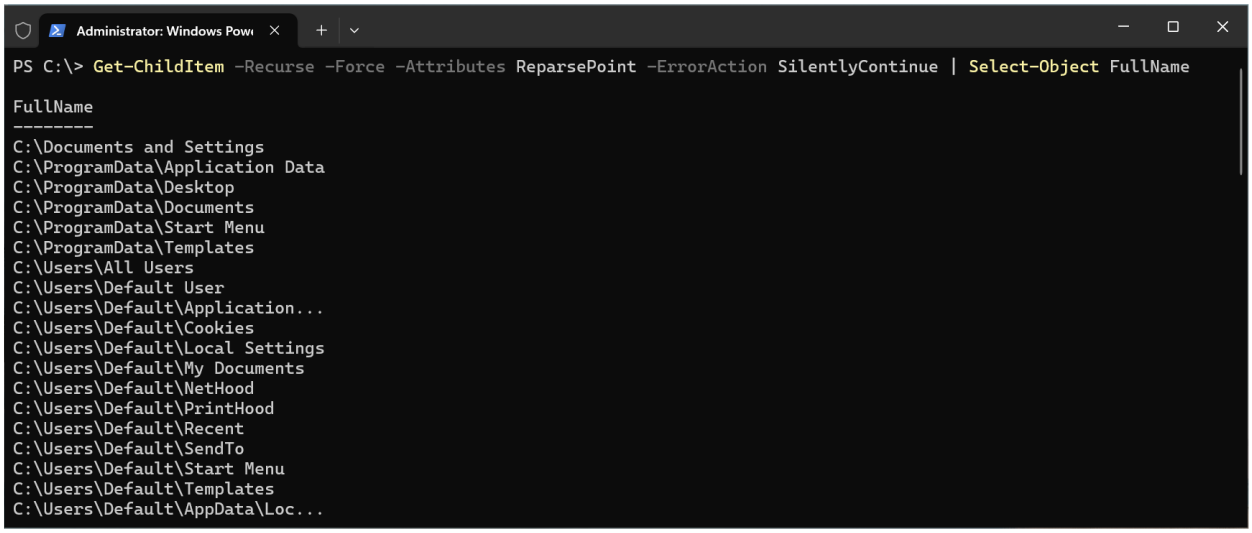


Рисунок 8

3.2.2. Информационная безопасность точек повторной обработки

Точки повторной обработки могут использоваться злоумышленниками во вредоносных целях, например:

1) Для блокирования доступа к файлам или каталогам.
Выбирается такой тег точек повторной обработки, с которым не ассоциирован ни один из драйверов в операционной системе. Затем файлы или каталоги, доступ к которым хотят заблокировать, помечаются точками повторной обработки, содержащими данный тег. В результате любые попытки чтения помеченных объектов закончатся неудачей.

2) Для скрытого хранения информации в области данных точки повторной обработки.

Область данных точки повторной обработки может содержать до 16 Кб информации, что позволяет в ней самой скрыто хранить данные.

3) Для подмены содержимого файлов.
Есть такое понятие «чёрная бухгалтерия», когда компания показывает перед налоговой одну отчётность, а по факту совсем другая. Так вот, с точками повторной обработки можно создать «чёрную файловую систему». Смысл этого явления в том, что неавторизованный пользователь будет видеть в файлах одну информацию, а авторизованный – совершенно другую. Для этого потребуется создать и настроить специфический драйвер файловой системы.

4) Для закрепления вредоносного кода в заражённой системе.
Во взломанный компьютер может устанавливаться вредоносный файловый драйвер, который будет получать управление всякий раз, когда открывается один из файлов, помеченный точкой повторной обработки ассоциированный с этим драйвером.

За более глубоким техническим разбором работы точек повторной обработки можно обратиться к этой статье.

3.2.3 Символические ссылки

Первой практической реализацией точек повторной обработки, которые мы рассмотрим, будут символические ссылки. И да, это те же символические ссылки, что и в Linux, но только круче. Изюминка в том, что Windows, в отличие от Linux, позволяет создавать символические ссылки не только на файлы и каталоги, но и на сетевые папки (SMB shares).

3.2.3.1. Создание символической ссылки на файл

Перейдём к практике. Пусть нам необходимо создать символическую ссылку «C:\test3\symlink_image.vhd» на файл «C:\test\image.vhd».

Тогда сделать это можно двумя способами:

1) Создание символической ссылки на файл с использованием встроенной в интерпретатор cmd команды mklink:

```
cmd /c mklink C:\test3\symlink_image.vhd C:\test\image.vhd
```

2) Создание символической ссылки на файл с помощью Powershell-командлета New-Item:

```
New-Item -ItemType SymbolicLink -Path C:\test3\symlink_image.vhd -Target
C:\test\image.vhd
```

Вне зависимости от выбранного способа в каталоге «C:\test3\» появился файл «symlink_image.vhd». Проводник Windows покажет (Рисунок 9), что созданный файл имеет тип «.symlink», хотя расширение файла «.vhd», но это никоим образом не помешает пользоваться файлом по назначению.

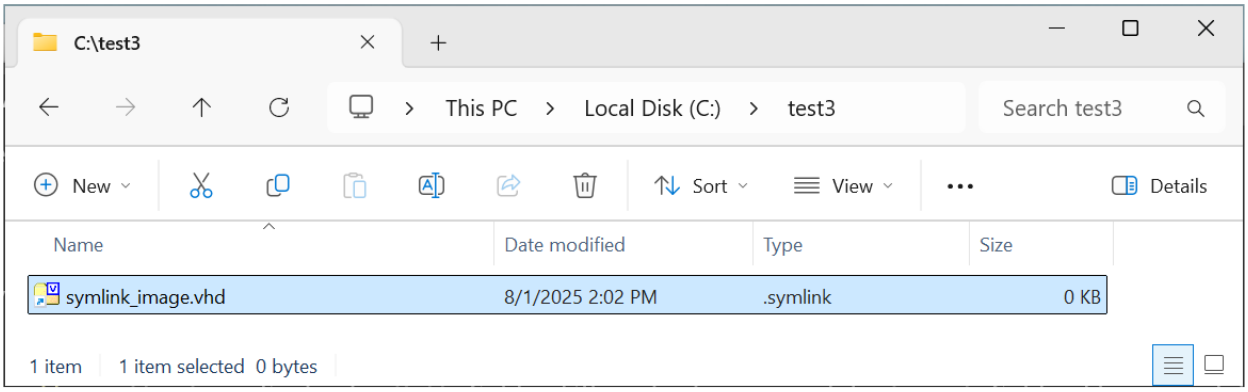


Рисунок 9

3.2.3.2. Перечисление всех символических ссылок в каталоге

Получить список всех символических ссылок в текущем каталоге и его подкаталогах можно с помощью PowerShell-скрипта:

```
Get-ChildItem -Recurse -Force | Where-Object LinkType -eq 'SymbolicLink' |
Select-Object FullName, Attributes, Target
```

Выполнив этот скрипт в каталоге «C:\test3», где ранее мы создали символическую ссылку на файл, мы получим следующие результаты (Рисунок 10):

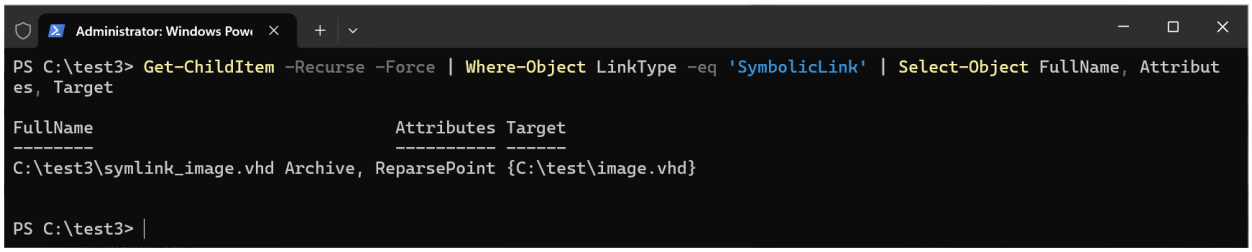


Рисунок 10

3.2.3.3. Анализ символической ссылки с помощью утилиты fsutil

Утилита fsutil позволят проводить подробный анализ точек повторной обработки, включая символические ссылки. Применительно к файлу «C:\test3\symlink_image.vhd» формат вызова утилиты будет следующим:

```
fsutil.exe reparsepoint query C:\test3\symlink_image.vhd
```

После запуска мы увидим (Рисунок 11) тег и область данных точки повторной обработки, которая является символической ссылкой.

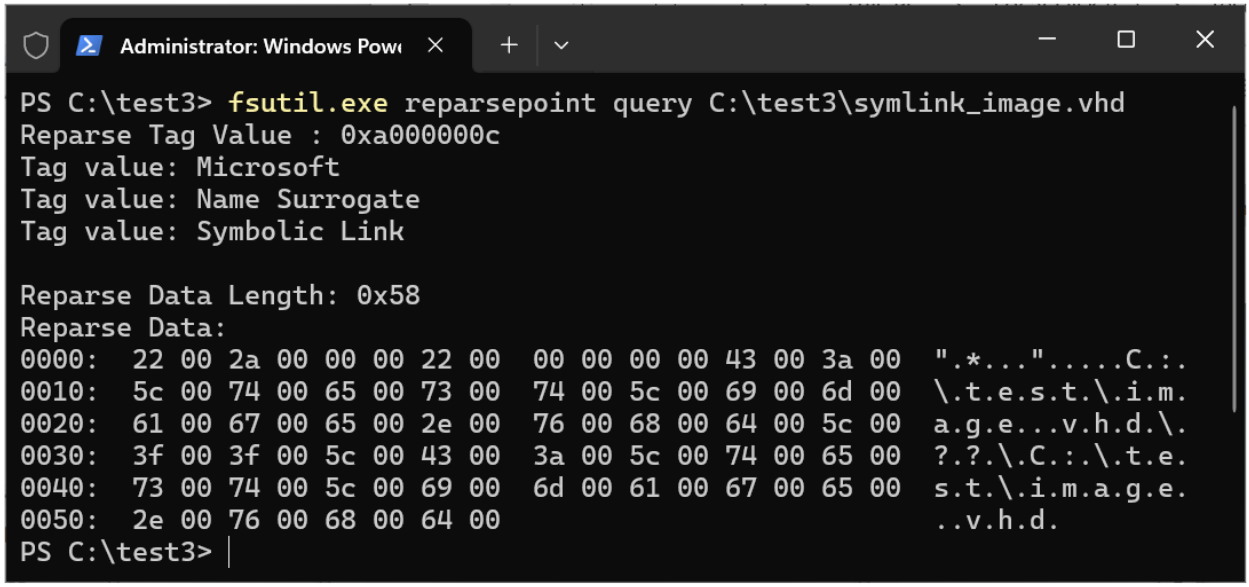


Рисунок 11

3.2.3.4. Удаление символической ссылки

Удалить символические ссылки можно с помощью стандартного командлета «Remove-Item». В результате его работы удаляются как точка повторной обработки, так и файл или каталог, с которыми она была связана.

Если же требуется удалить только саму точку повторной обработки, не удаляя при этом файл или каталог, то поможет в этом утилита «fsutil».

Например, удаление ранее созданной рекурсивной ссылки с помощью «fsutil» будет выглядеть так:

```
fsutil.exe reparsepoint delete .\recurse\
```

3.2.3.5. Сравнение преимуществ символических и жёстких ссылок

Если жёсткая и символическая ссылка на файл по факту создаёт новое альтернативное имя файла, то какую лучше использовать? Однозначного ответа на этот вопрос не существует, поскольку каждый тип ссылок по-своему хорош.

Преимущества символических ссылок над жёсткими:

1. Можно сделать ссылку на файл, находящийся на другом логическом диске.
2. Можно создать ссылку «наперёд» для ещё несуществующего файла.

3. Перенос символической ссылки на другой раздел не приведёт к ее преобразованию в файл.

Преимущества жёстких ссылок над символическими:

1. Жёсткая ссылка – это дополнительное имя одного и того же блока данных. В то время как символическая ссылка – это отдельный блок данных, ссылающийся на другой блок данных. В отношении символических ссылок всегда существует неопределённость: рассматривать их как отдельный файл-ссылку или же как объект, на который они указывают.
2. Удаление целевого файла символической ссылки приведёт к поломке всех символических ссылок, которые на него ссылались, в то время как удаление одной из жёстких ссылок никоим образом не скажется на работоспособности оставшихся символических ссылок.

Поэтому выбор типа ссылки будет зависеть от решаемой с помощью неё задачи.

3.2.4. Точки соединения

В NTFS исторически первым видом функционала, похожего на привычные символические ссылки в Linux, были точки соединения. Они и по сей день широко используются в Windows. Например, каталог «C:\Document and settings», оставленный в Windows 11 для совместимости со старыми версиями операционной системы, содержит точку соединения, указывающую на «C:\Users» (Рисунок 12).

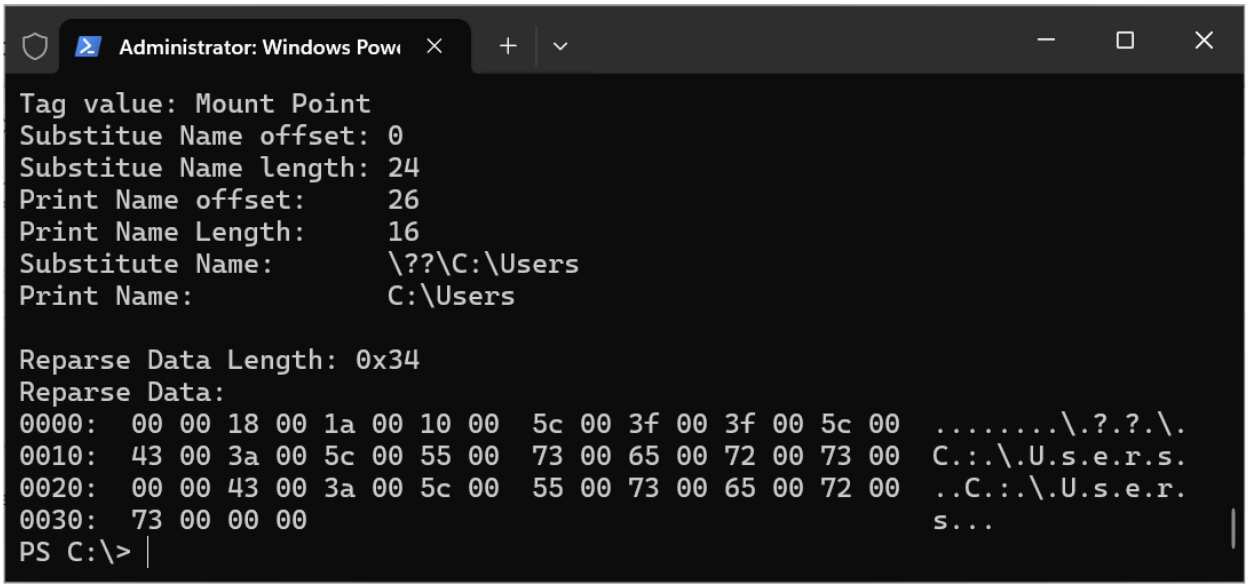


Рисунок 12

Точки соединения могут использоваться для двух целей:

1. Для ссылки одного каталога на другой (directory junction).
2. Для монтирования дисков в каталог (volume junction).

3.2.4.1. Создание точки соединения, указывающей на каталог

Пусть требуется создать точку соединения «C:\test4», указывающую на «C:\test».

Тогда операционная система предоставляет нам два варианта действий:

1) Создание точки соединения, указывающей на каталог, с помощью встроенной в интерпретатор cmd команды mklink:

```
cmd /c mklink /j C:\test4 c:\test
```

2) Создание точки соединения, указывающей на каталог, с помощью PowerShell-командлета «New-Item»:

```
New-Item -ItemType Junction -Path C:\test4 -Target C:\test
```

В результате выполнения любого из вариантов будет создана точка соединения визуально и функционально такая же, как и символическая ссылка (Рисунок 13).

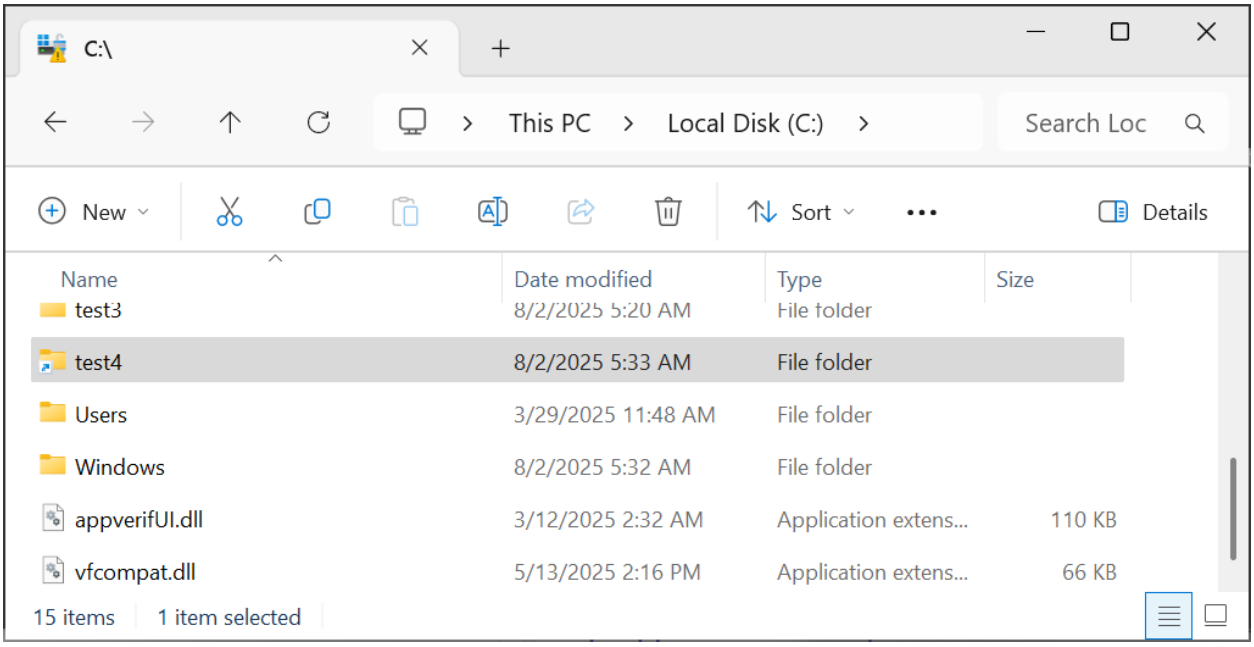


Рисунок 13

3.2.4.2. Создание точки соединения, указывающей на диск

Пусть требуется смонтировать диск «X:» (созданный нами ранее виртуальный жёсткий диск, хранящийся в файле c:\test\image.vhd) в папку «c:\test5».

Тогда вначале нам следует создать папку «C:\test5», а затем выполнить с правами администратора PowerShell-скрипт:

```
Get-Volume -DriveLetter "X" | Get-Partition | Add-PartitionAccessPath -
AccessPath C:\test5
```

Главной частью этого скрипта является командлет Add-PartitionAccessPath, поскольку именно он и создаёт точку соединения, а предшествующие ему командлеты призваны лишь идентифицировать нужный раздел для монтирования.

В примере выше мы для простоты повествования идентифицировали нужный нам раздел по букве, но на практике это будет вряд ли возможным, поскольку нет смысла монтировать диск в каталог, если он уже доступен по букве. В таких случаях придётся «глазами» идентифицировать нужный диск, просматривая перечень всех доступных дисков, выводимый скриптом:

```
Get-Volume | Select-Object *
```

Конечная цель просмотра – определение UniqueId нужного нам тома.

На виртуальной машине, использованной для написания статьи, требуемый том был идентифицирован по метке (File system label) и размеру (Size). Соответственно его UniqueId был равен «\\?\Volume{1d9019dc-8cb2-4190-ab14-06c1ab02a3a9}\» (Рисунок 14).

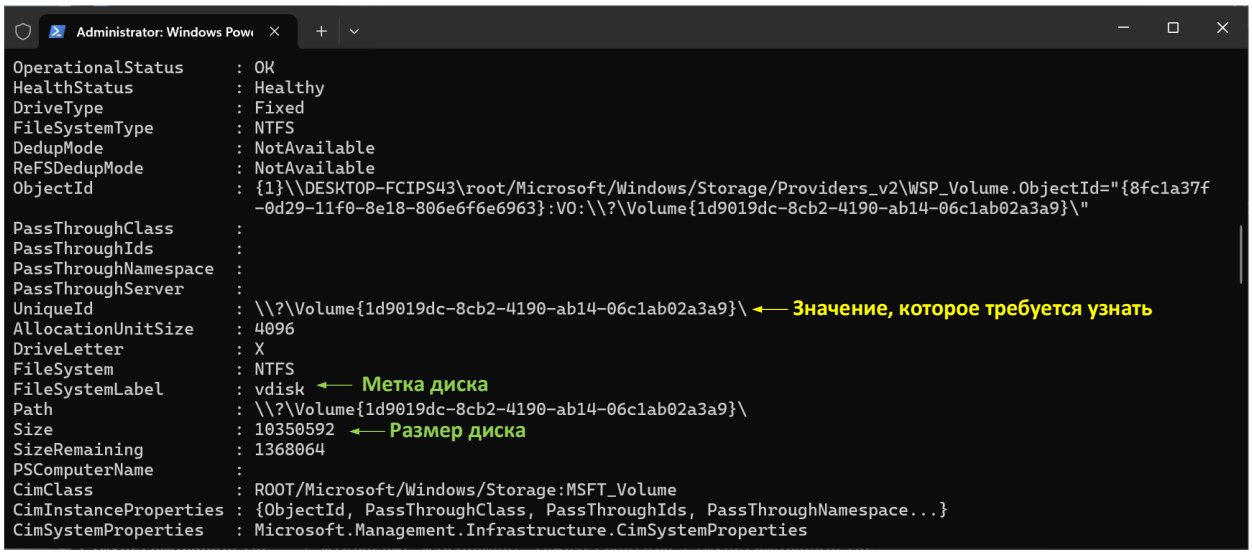


Рисунок 14

У вас же при повторении эксперимента значение UniqueId будет другим.

Определив UniqueId, создадим каталог «c:\test6» и смонтируем туда наш диск:

```
Get-Volume -UniqueId '\\?\Volume{1d9019dc-8cb2-4190-ab14-06c1ab02a3a9}\' |
Get-Partition | Add-PartitionAccessPath -AccessPath C:\test6
```

3.2.5. Сравнение символических ссылок и точек соединения

Логичный вопрос, если точки соединения, могут указывать на каталог и символические ссылки тоже могут указывать на каталог, то, в чём между ними разница? С точки зрения результата – ни в чём, и то и другое даст ссылку на каталог, но вот выбор конкретного механизма имеет ряд нюансов:

1. Точки соединения могут указывать на каталог или том (логический диск), а символические ссылки – на файл, каталог или сетевую папку.
2. Создание символической ссылки требует прав администратора, создание точки соединения – нет.

Таким образом, всё будет зависеть, на что следует ссылаться и есть ли административные привилегии в момент создания ссылки или их нет.

3.2.6. Информационная безопасность символических ссылок и точек соединения

Злоумышленники с помощью символических ссылок или точек соединения могут создавать петли в дереве каталогов, что может привести к отказу в обслуживании уязвимого ПО. Потенциально опасными являются ссылки, указывающие сами на себя, или указывающие на родительский каталог.

Для демонстрации проблемы создадим символическую ссылку-каталог «c:\test3\recurse», который будет указывать на родительский каталог «C:\test3»:

```
New-Item -ItemType SymbolicLink -Path C:\test3\recurse -Target C:\test3
```

В проводнике Windows попробуем несколько раз открыть каталог «.\recurse». После предпринятых действий, изображение содержимого каталога, демонстрируемое проводником Windows (Рисунок 15), не поменялось, а вот адрес текущей директории изменился.

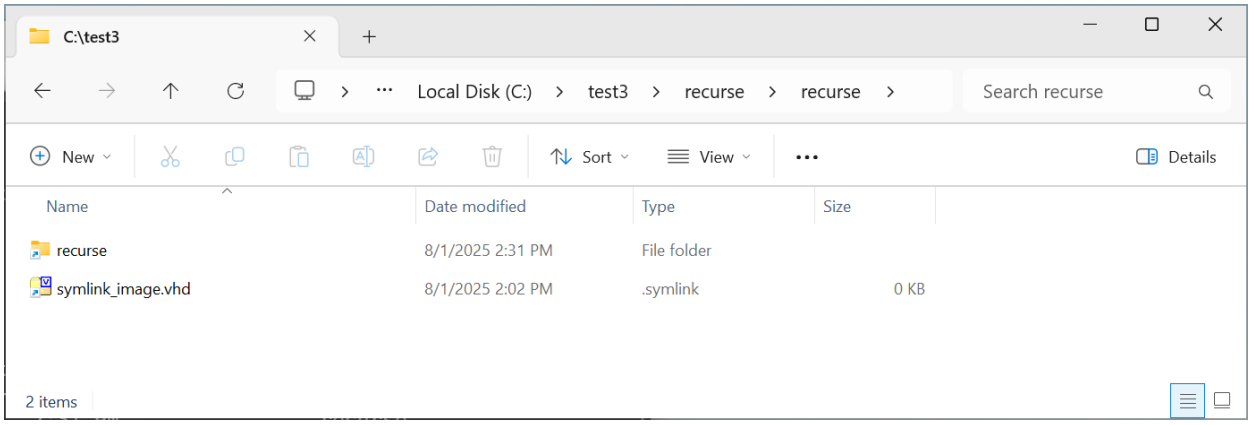


Рисунок 15

Соответственно, если вместо нас пробежаться по всем элементам каталога «C:\Test3\» захочет уязвимая программа, то рано или поздно она зависнет или аварийно завершится.

PowerShell-командлет «Get-ChildItem» по умолчанию не переходит по символическим ссылкам или точкам соединения. Он рассматривает содержащие их объекты как обычные файлы или каталоги. Чтобы заставить «Get-ChildItem» переходить по ссылкам нужно использовать ключ «-FollowSymlink». Благо разработчики защитили командлет от рекурсивных переходов и если «Get-ChildItem» увидит рекурсию, то выдаст предупреждение, о том, что он уже был в том месте и не будет больше туда переходить (Рисунок 16).

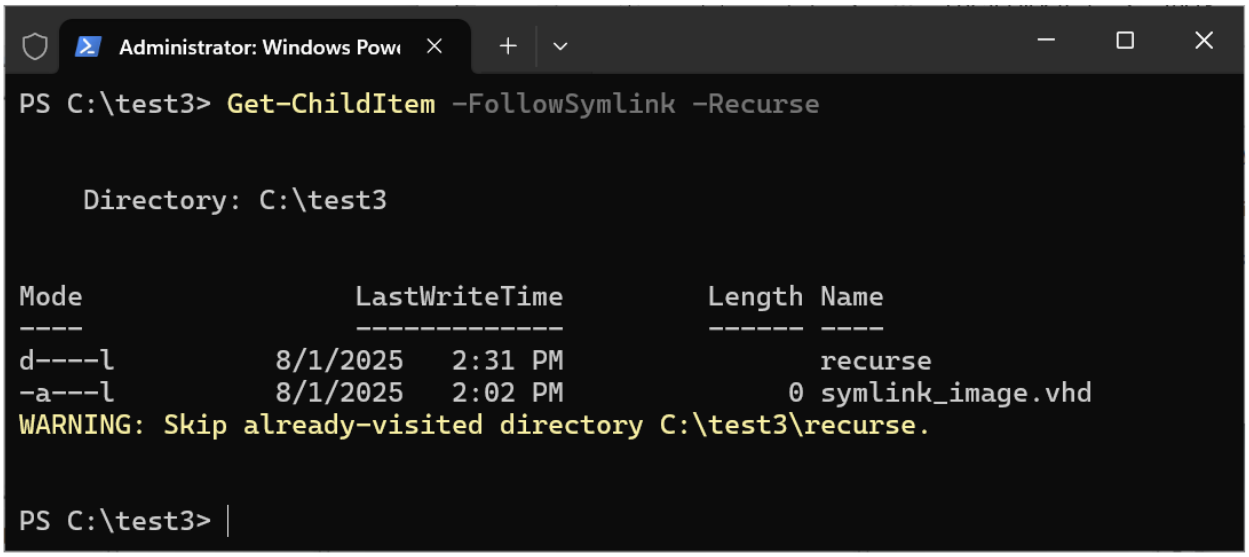


Рисунок 16

4. Расширенные атрибуты файлов (extended attributes)

Мы привыкли, что у файлов есть четыре стандартных атрибута:

- «только для чтения»,
- «скрытый»,
- «системный»,
- «готов для архивирования».

Однако кроме стандартных атрибутов NTFS поддерживает так называемые расширенные атрибуты (extended attributes, EA), невидимые в стандартном пользовательском интерфейсе. Резонный вопрос, раз их не видно, тогда зачем они нужны? Давайте разбираться.

Расширенные атрибуты – это именованные блоки данных, которые могут быть подсоединены к файлу. Максимальный размер расширенного атрибута в NTFS, который может назначить Windows (судя по спецификации структуры FILE_FULL_EA_INFORMATION, используемой Win32 API функции ZwSetEaFile) составляет 64 Кб.

Термин «расширенный» в указании атрибутов зачастую порождает путаницу. Дело в том, что к стандартным атрибутам файла, таким как «Скрытый», «Системный» и так далее в новых версиях Windows добавились атрибуты: «Зашифрованный» (Encrypted), «Исключён из индексирования» (Not content indexed), «Разреженный файл» (Sparse file) и прочие. Так вот, эти новые атрибуты в некоторых публикациях ошибочно называют расширенными, хотя это в корне не верно. Классические и вновь добавленные атрибуты – это битовые маски (см. File Attribute Constants) фиксированной длины и формата, а механизм расширенных атрибутов – это именованные блоки данных, формат и длина которых в определённых пределах может быть произвольными. Файл может вообще не содержать расширенных атрибутов, в то время как стандартные атрибуты в нём будут всегда.

Встроенные средства Windows включают утилиту «fsutil», позволяющую просматривать расширенные атрибуты файлов. Каких-либо штатных средств установки расширенных атрибутов современные версии Windows не имеют.

4.1. Перечисление всех файлов, содержащих расширенные атрибуты

Чтобы найти все файлы, обладающие расширенными атрибутами, можно воспользоваться PowerShell скриптом, представленным ниже.

```
Get-ChildItem -Recurse -Force -ErrorAction SilentlyContinue | ForEach-Object
{
  $Result= $(fsutil.exe file queryea $_.FullName )
```

```
  if ( $Result -match 'Extended Attributes \(EA\) information for file' ) {
    $_.FullName
  }
}
```

Проблема в том, что скрипт построен на базе текстового анализа вывода утилиты «fsutil» и может не работать на инсталляциях Windows, использующих локализацию, отличную от английской (например, русскую).

Перейдя в каталог «C:\Windows» и запустив этот скрипт, мы увидим (Рисунок 17), что операционная система активно использует механизм расширенных атрибутов в своей работе.

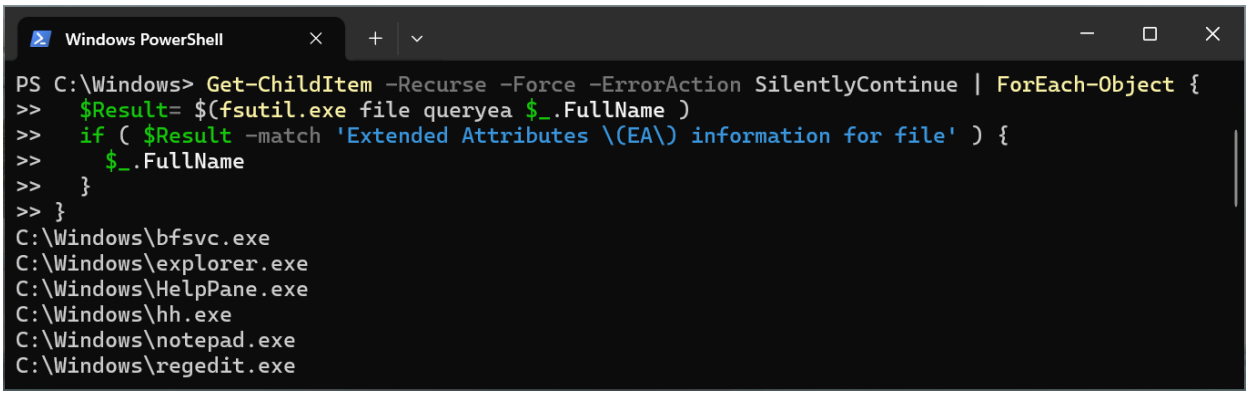


Рисунок 17

4.2. Просмотр расширенных атрибутов файла

Давайте глянем, какие расширенные атрибуты назначены на исполняемый файл стандартной программы «Блокнот» (C:\Windows\notepad.exe):

```
fsutil.exe file queryea C:\Windows\notepad.exe
```

Здесь видно (Рисунок 18), что файл содержит два расширенных атрибута: $KERNEL.PURGE.ESBCACHE и $CI.CATALOGHINT. Причём первый атрибут, судя по началу имени «$KERNEL.» относится к расширенным атрибутам ядра (Kernel Extended Attributes), которые могут быть установлены только из ядра Windows.

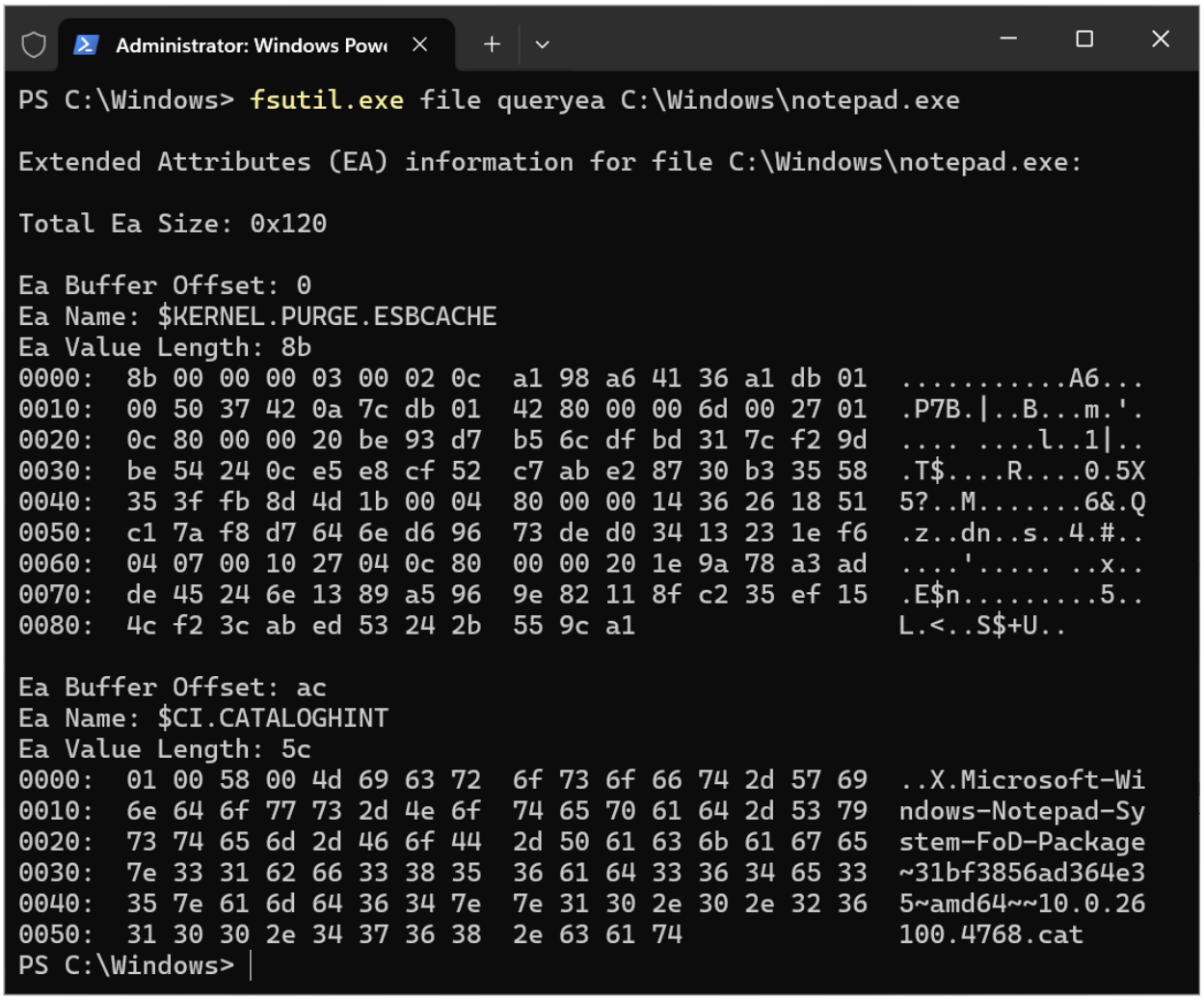


Рисунок 18

4.3. Установка расширенного атрибута файлу

Как отмечалось ранее, Windows не содержит стандартных инструментов для установки расширенных атрибутов файлов. Поэтому для проведения эксперимента по созданию собственного расширенного атрибута, воспользуемся сторонним PowerShell-модулем «PSReflect-Functions», который содержит функцию «NtSetEaFile», являющуюся обёрткой для одноимённой WinAPI-функции, хранящейся в «ntdll.dll».

Используя «NtSetEaFile» напишем минимальный скрипт, добавляющий к файлу «C:\test7\a.txt» (файл и каталог нужно предварительно создать) расширенный атрибут с именем «MyEA» и значением «My_DATA».

```
Import-Module PSReflect-Functions -DisableNameChecking
$FilePath = "C:\test7\a.txt"
$EaName = "MyEA"
$EaValue = [System.Text.Encoding]::UTF8.GetBytes("My_DATA")
```

```
$File = [System.IO.File]::Open(
                                  $FilePath,
                                  [System.IO.FileMode]::Open,
                                  [System.IO.FileAccess]::ReadWrite,
                                  [System.IO.FileShare]::None
                                )

try {
  NtSetEaFile -FileHandle $File.Handle -Name $EaName -Value $EaValue
  Write-Host "Extended attribute '$EaName' successfully set on '$FilePath'."
}
catch {
  Write-Error "Failed to set extended attribute: $($_.Exception.Message)"
}

finally {
  if ( $null -ne $File.Handle ) {
[System.Runtime.InteropServices.Marshal]::Release($File.Handle) }
  if ( $null -ne $File ) { $File.Close() }
}
```

Примечание 1. Перед запуском скрипта необходимо предварительно проинсталлировать модуль «PSReflect-Functions» путём вызова команды (требуется права Администратора):

```
Install-Module PSReflect-Functions
```

Примечание 2. Модуль «PSReflect-Functions» не отличается стабильностью работы. Для установки атрибута, возможно, потребуется запустить скрипт несколько раз.

После запуска скрипта запустим «fsutil» и посмотрим, что получилось.

```
fsutil.exe file queryea C:\test7\a.txt
```

Как мы видим (Рисунок 19), расширенный атрибут был успешно создан.

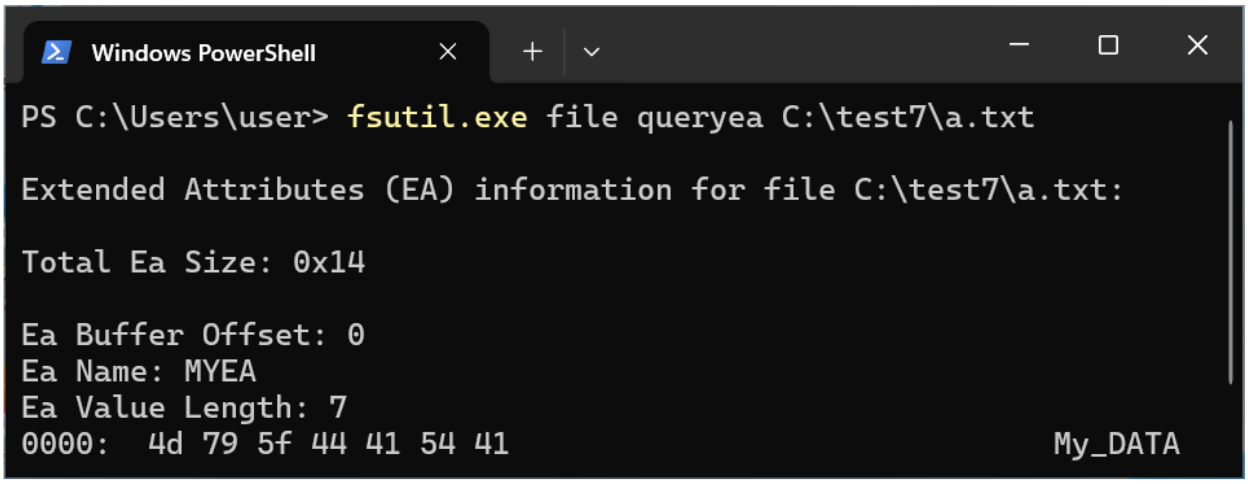


Рисунок 19

5. Альтернативные потоки

Ранее, рассматривая жёсткие ссылки, мы отметили, что у файла может быть несколько имён. Теперь настало время узнать, что у файла может быть ещё и несколько «тел», каждое из которых имеет собственные данные.

Такое вот, «тело» файла в NTFS правильно называется потоком данных. Файл может содержать несколько потоков (Рисунок 20), один из которых будет основным, а остальные – альтернативными.

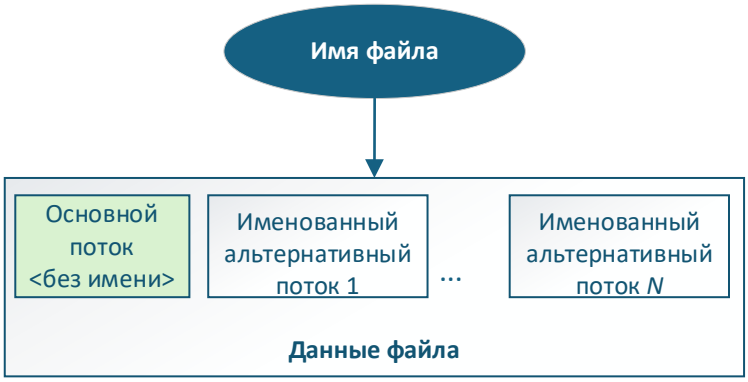


Рисунок 20

Основной поток существует всегда и используется по умолчанию. У него нет отличительного имени. В противовес ему альтернативные потоки нужно явно создавать и каждый из них должен носить своё уникальное (в рамках файла) имя. Файл может вообще не иметь альтернативных потоков, а может иметь их сколь угодно много, главное, чтоб хватило места на диске.

Для чтения или записи информации в альтернативные потоки нужно явным образом указывать их имена, а иногда и тип. Схема именования файла, включающая имя и тип альтернативных потоков, выглядит следующим образом:

```
<имя файла>:<имя потока>:<тип потока>
```

Применительно к обычным файлам тип потока данных всегда будет $DATA. В других случаях, часть которых мы проанализируем ниже, типы потоков будут отличаться.

Файл с альтернативными потоками может напоминать архив, в котором вместе слито несколько файлов. Поначалу это может вызывать недоумение, зачем вообще так сделали, но со временем становится понятным, что альтернативные потоки – это не такая уж плохая штука.

Рассмотрим пример. Предположим, вы собираетесь обеспечивать целостность какого-либо файла. Соответственно, вам необходимо рассчитать его контрольную сумму и периодически её проверять. Это понятно. Вопрос остаётся в том, где хранить контрольную сумму. Для файловой системы exFAT единственным вариантом было бы создание нового файла и размещение её там, но для NTFS элегантным решением является хранение контрольной суммы в альтернативном потоке. Решение с дополнительным файлом приводит к куче проблем:

- Что делать, если защищаемый файл требуется переместить в другой каталог или переименовать?
- Что делать, если вы хотите защищать ещё один файл? Нужно ли при этом создавать новый файл или помещать контрольную сумму в уже существующий?
- И так далее.

В то же время альтернативные потоки полностью избавляют от всех этих проблем, поскольку позволяют реализовать принцип «всё своё ношу с собой». Хотя с ними тоже не всё так просто, но об этом ниже.

Приступая к экспериментам с альтернативными потоками, начнём с того, что в каталоге «C:\Test10» создадим текстовый файл «file.txt» с содержимым «I am the main stream»:

```
"I am the main stream" > C:\Test10\file.txt
```

5.1. Создание альтернативных потоков

Теперь для этого файла создадим альтернативный поток с именем «alter1» и содержимым «I'm alter stream 1»:

```
Set-Content -Path "C:\test10\file.txt" -Value "I'm alter stream 1" -Stream
"alter1"
```

Создадим ещё один альтернативный поток с именем «file.jpg» и для разнообразия поместим в него не скучную строку, а обои рабочего стола Windows:

```
Get-Content "$env:AppData\Microsoft\Windows\Themes\TranscodedWallpaper" -
Encoding Byte | Set-Content -Path C:\Test10\file.txt -Stream "file.jpg" -
Encoding Byte
```

5.2. Просмотр перечня альтернативных потоков файлов

Посмотрим, что у нас получилось. Сначала выведем перечень всех потоков данных файла «C:\Test10\file.txt».

```
Get-Item C:\Test10\file.txt -Stream * | Select-Object PSPath
```

Обратите внимание (Рисунок 21), что имя основного потока пустое и указан только его тип $DATA. Кроме основного потока в файле присутствуют и два альтернативных – «alter1» и «file.jpg», но тип $DATA для них не отображён, хотя он именно таков.

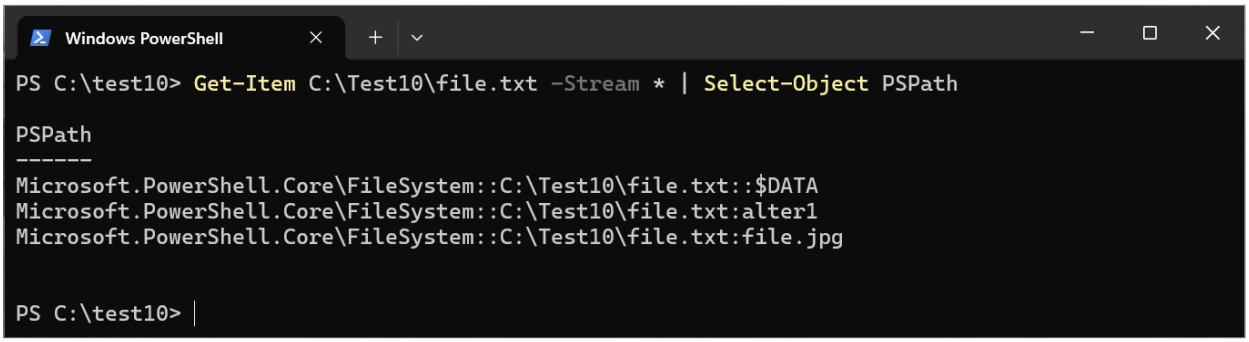


Рисунок 21

5.3. Извлечение данных из альтернативного потока

Просто посмотреть перечень альтернативных потоков – это, конечно, интересно, но малоинформативно. Для отображения содержимого потока «alter1» на консоль нужно выполнить команду (Рисунок 22):

```
Get-Content C:\Test10\file.txt -Stream alter1
# или
Get-Content C:\Test10\file.txt:alter1
```

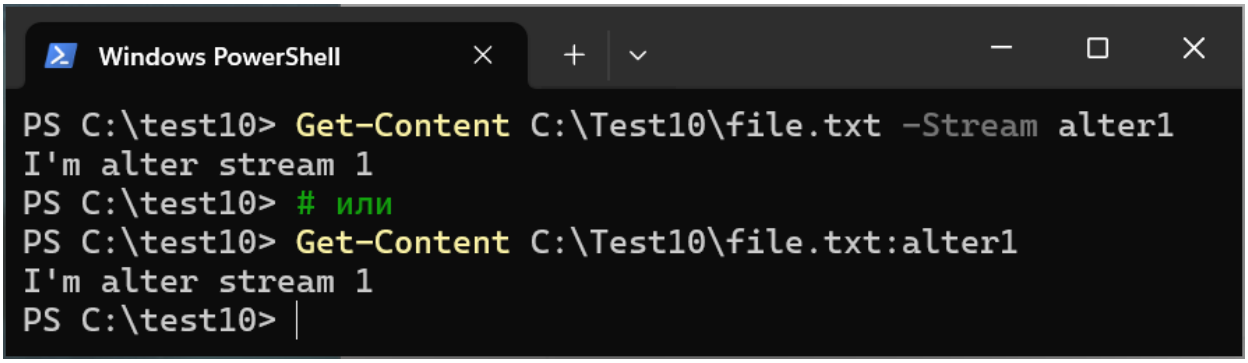


Рисунок 22

Строку, конечно, мы в консоль вывели, но вот просматривать так картинку – не лучшая идея. Вместо этого сохраним её лучше в файл (например, «C:\Test10\out.jpg»):

```
Get-Content C:\Test10\file.txt -Stream file.jpg -Encoding Byte | Set-Content
C:\Test10\out.jpg -Encoding byte
```

А затем посмотрим в любой программе показа графических файлов.

В скриптах выше мы использовали обращение к альтернативным потокам без указания типа. Однако ничто не запрещает нам сделать то же самое, но указав и тип (Рисунок 23).

```
Get-Content C:\Test10\file.txt -Stream 'alter1:$DATA'
```

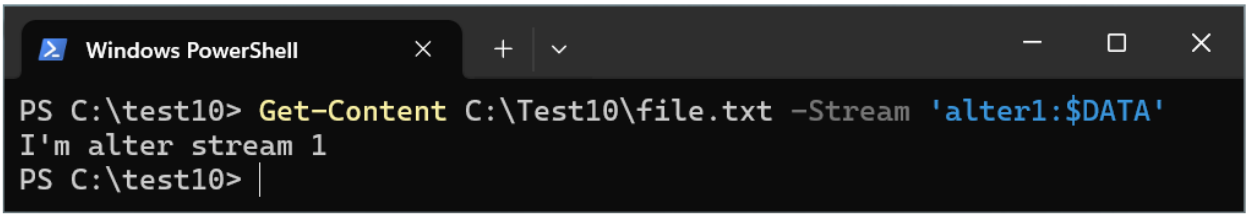


Рисунок 23

По этой же схеме мы можем явно обратиться и к основному потоку данных (Рисунок 24):

```
Get-Content C:\Test10\file.txt -Stream ':$DATA'
```

Рисунок 24

5.4. Удаление альтернативных потоков

Рассматривая работу с альтернативными потоками, нельзя не рассмотреть их удаление. Для того чтобы удалить выбранный поток, например, «file.jpg» можно воспользоваться командлетом «Remove-Item»:

```
Remove-Item C:\Test10\file.txt -Stream file.jpg
```

Удаление всех альтернативных потоков будет выглядеть так:

```
Remove-Item C:\Test10\file.txt -Stream *
```

5.5. Поиск всех файлов, содержащих альтернативные потоки

Для того чтобы найти все файлы, содержащие альтернативные потоки в текущем каталоге можно воспользоваться скриптом:

```
Get-ChildItem -Recurse -Force -ErrorAction Ignore | ForEach-Object { try {
Get-Item $_.FullName -Stream * -ErrorAction Stop } catch {} } | Where-Object
Stream -ne ':$Data' | Select-Object FileName, Stream, Length
```

Существует и более короткий вариант скрипта, правда, он больше подходит для ручного анализа, когда в анализируемом каталоге не очень много файлов.

```
cmd /c "dir /r"
```

5.6. Альтернативные потоки каталогов

В NTFS каталоги – это тоже файлы, только особые, имеющие определённую структуру. Содержимое каталога хранится в основном потоке с именем «$I30» и типом «$INDEX_ALLOCATION». Мы можем совершить листинг каталога с явным указанием имени и типа потока. Например, так:

```
cmd /c 'dir c:\test10:$I30:$INDEX_ALLOCATION'
```

Теперь, раз мы знаем, что каталог – это тоже файл, то, значит, для него тоже можно создавать альтернативные потоки!

В качестве примера создадим для каталога «C:\Test10» альтернативный поток с именем «ads 1» и содержимым «Alternative directory stream 1».

```
Set-Content -Path C:\test10 -Stream 'ads 1' -Value 'Alternative directory
stream 1'
```

Просмотреть альтернативные потоки каталога можно с помощью команды:

```
cmd /c dir C:\test10\ /r
```

Извлечь данные из альтернативного потока каталога можно почти точно так же, как и из альтернативного потока файла. Например, для нашего тестового каталога «C:\test10» команда будет такой:

```
Get-Content C:\test10 -Stream 'ads 1'
```

Вот с удалением альтернативных потоков каталогов есть проблемы. Встроенных в Windows 11 средств для этого нет. Но если очень надо, то можно обновить PowerShell до версии 7.2 или выше, после – можно удалить альтернативный поток с помощью командлета «Remove-Item». Другим вариантом будет использование сторонних утилит, например, «streams» из пакета Sysinternals, которые также позволяют удалять альтернативные потоки.

5.7. Широко известные альтернативные потоки

Перечень широко известных имён потоков, используемых в файловой системе NTFS, можно посмотреть тут. Однако все эти потоки системные и представляют мало интереса для практического использования.

Прикладные программы также активно используют альтернативные потоки (см. тут) и как минимум с одной реализацией вы наверняка знакомы.

Помните ситуацию, когда, скачав документ из Интернета, Word открывал его в режиме защищённого просмотра и не давал редактировать (Рисунок 25)?

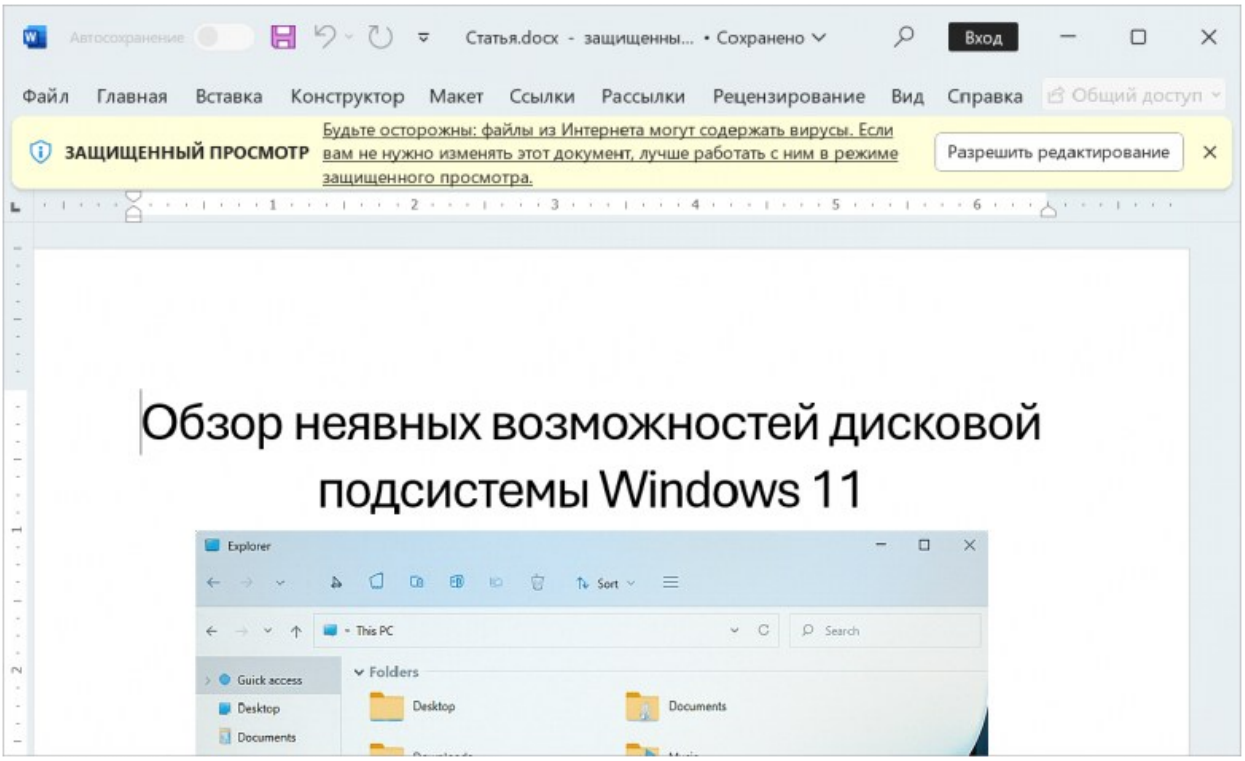


Рисунок 25

Так вот, это происходило из-за того, что браузер (или проводник Windows) добавлял к скачанному файлу альтернативный поток «Zone.Identifier» и указывал в нём, что файл получен из сети (Рисунок 26).

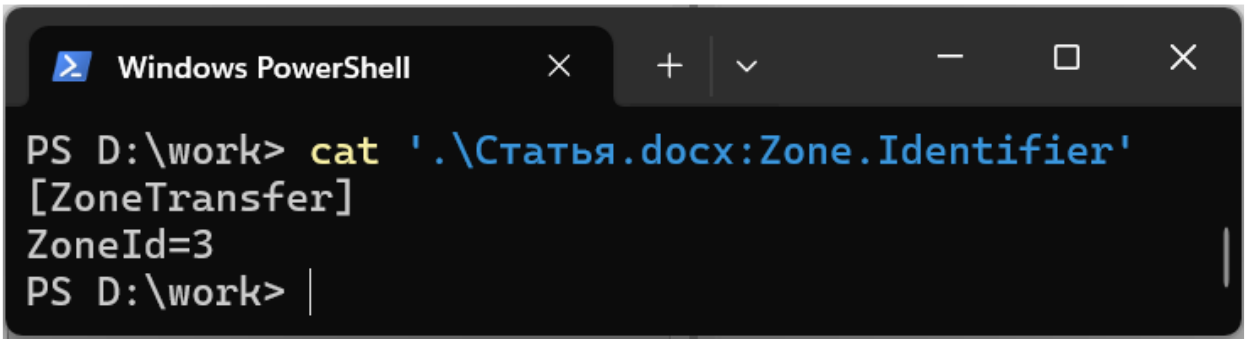


Рисунок 26

Обнаруживая этот альтернативный поток и видя, что файл получен извне, текстовый редактор разводил панику на тему того, что в документе может быть вирус.

Просто и быстро удалить альтернативный поток «Zone.Identifier» можно с помощью командлета «Unblock-File».

5.8. Ограничения альтернативных потоков

Несмотря на свои неоспоримые преимущества, технология альтернативных потоков не лишена недостатков. Первым и главным из которых является то, что альтернативные потоки могут тихо пропасть из файла при его перемещении или копировании.

Например, если вы копируете файл с альтернативными потоками на файловую систему, не поддерживающую эту технологию, например, exFAT, то все потоки пропадут, и вы даже не узнаете об этом. Аналогичная проблема может произойти при передаче файла по сети или даже при использовании буфера обмена для переноса файла на виртуальную машину.

Другими словами, альтернативные потоки – очень неустойчивая конструкция и может использоваться только для временного хранения данных, не играющих существенной роли для функционирования приложений.

5.9. Что лучше альтернативные потоки или расширенные атрибуты

Для ситуаций, когда вместе с файлом необходимо хранить дополнительную информацию, Microsoft рекомендует использовать альтернативные потоки. Технология расширенных атрибутов считается устаревшей, и производитель Windows потихоньку убирает все инструменты для работы с ней из состава дистрибутива.

5.10. Информационная безопасность альтернативных потоков

Тот факт, что альтернативные потоки не видны из проводника Windows, делает их отличным местом для скрытого хранения информации, чем активно пользуется шпионские программы.

До недавнего времени Windows позволяла запускать исполняемые файлы напрямую из альтернативных потоков, чем с превеликим удовольствием пользовались трояны. Однако теперь провернуть такой же фокус в Windows 11 стало значительно тяжелее (хотя всё же возможно).

Работа с альтернативными потоками не требует административных полномочий и может совершаться рядовыми пользователями. Поэтому, при хранении контрольных сумм файлов в альтернативных потоках нужно помнить, что любая программа, обладающая возможностью изменять файл, может также изменить и альтернативный поток с контрольной суммой.

6. Теневые копии

В отличие от всего рассмотренного ранее, механизм теневых копий (Volume Shadow Copy, VSS) – это не особенность NTFS, а сервис операционной системы

Windows. С его помощью можно создать снимок состояния (snapshot) файловой системы – теневую копию, а затем использовать его для отката изменений или доступа к монопольно открытым файлам. Звучит очень круто, но есть нюансы.

Теневые копии – это не резервные копии, ведь когда создаётся теневая копия, никакого дублирования информации не происходит. Вместо этого применяется технология «копирование при записи» (Copy-on-write), суть которой в том, что при создании теневой копии, производится как бы опись состояния файловой системы, а затем, когда в файл или каталог вносятся изменения, исходный (не модифицированный) объект помещается в буфер теневой копии, а модифицированные данные напрямую записываются на диск. В результате пользователь всегда работает с актуальными данными, непосредственно хранящимися на диске, а в случае необходимости может извлечь немодифицированные данные из буфера теневой копии.

По умолчанию теневые копии хранятся в каталоге «System Volume Information» логического диска, который они защищают и под них отводится 10% от максимальной ёмкости тома (volume). Однако с использованием утилиты vssadmin.exe подобное поведение, а равно и многие другие настройки теневых копий можно изменить.

Если говорить о назначении теневых копий, то они, конечно, хороши сами по себе как механизм версионирования данных. Теневые копии при соблюдении ряда условий могут послужить отличным средством от вирусов-шифровальщиков. Но из-за того, что в теневых копиях не производится дублирования информации, они не могут использоваться как замена резервных копий. Однако в сочетании с системами резервного копирования вся мощь теневых копий раскрывается в полной красе (более подробно об этом можно почитать тут). Дело в том, что, создав теневую копию, можно получить доступ к используемым в настоящий момент файлам, а это позволяет:

- проводить резервное копирование, не отрывая пользователя от работы;
- проводить расчёт контрольных сумм работающих исполняемых файлов для обеспечения контроля целостности системы или реализации механизмов замкнутой программной среды;
- и делать другие интересные вещи.

Ну и под конец описательной части отметим, что манипуляции с теневыми копиями всегда требуют прав системного администратора.

6.1. Создание теневой копии

Узнав столько о теневых копиях, вам, наверно, уже не терпится «потрогать их руками». Что же, начнём с простого – создадим теневую копию диска C.

```
Invoke-CimMethod -ClassName Win32_ShadowCopy -MethodName Create -Arguments @{
Volume = "C:\" }
```

В результате выполнения команды (Рисунок 27) система вернёт нам «ShadowID» (в нашем примере он равен {7247233F-08F3-4D1D-AD36-AC90348FAC45 }), который затем понадобится для монтирования теневой копии.

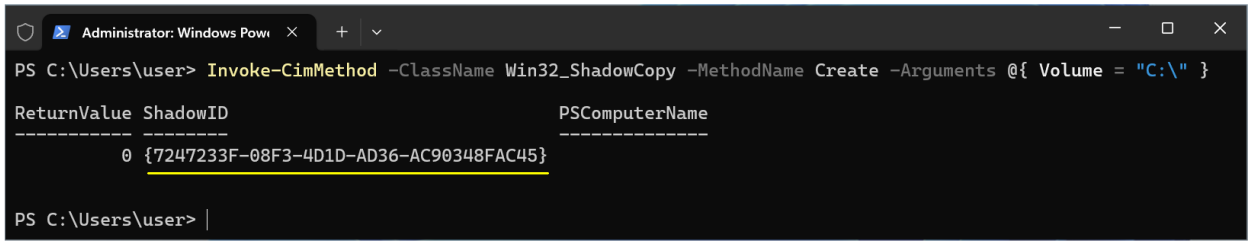


Рисунок 27

6.2. Перечисление теневых копий

Если «ShadowID» был утерян – не беда, его всегда можно восстановить, выполнив перечисление всех теневых копий.

Вариант 1. Перечисление теневых копий с помощью утилиты vssadmin (Рисунок 28):

```
vssadmin.exe list shadows
```

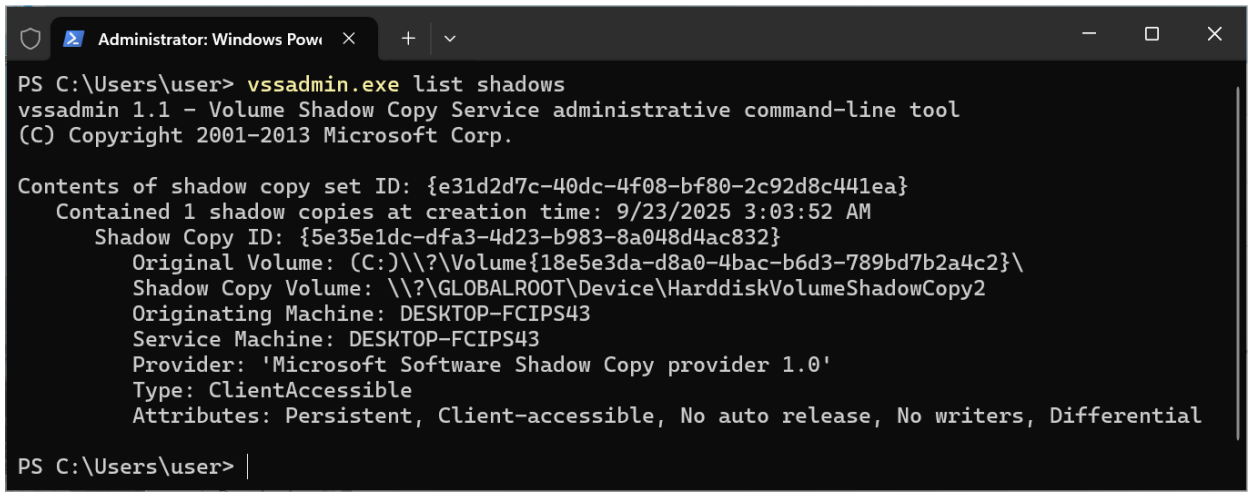


Рисунок 28

Вариант 2. Перечисление теневых копий с помощью PowerShell-скрипта (Рисунок 29):

```
Get-CimInstance -ClassName Win32_ShadowCopy | Select-Object InstallDate,
  @{
    Name = 'DriveLetter'
    Expression = { (Get-Volume -UniqueId $_.VolumeName).DriveLetter }
  }, DeviceObject, ID
```

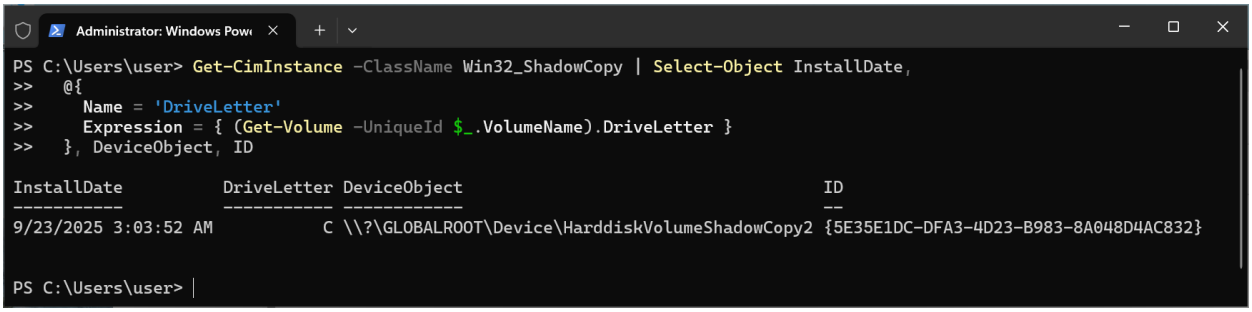


Рисунок 29

Для идентификации нужной теневой копии удобно использовать комбинацию времени создания копии и буквы логического диска. Применительно к первому варианту, эти значения представлены в строках «creation time» и «original volume», для второго варианта – параметры «InstallDate» и «DriveLetter» соответственно.

6.3. Монтирование теневой копии

Для доступа к содержимому теневой копии её нужно смонтировать. Для этого необходимо определить путь до неё в файловой системе.

Для первого варианта получения сведений о теневой копии этот путь находится в строке «Shadow copy volume», а для второго варианта – в параметре «DeviceObject». В дальнейшем к этому параметру нужно добавить «\» в конец и использовать полученное значение, как адрес назначения при создании символической ссылки каталога или точки объединения.

PowerShell-скрипт, монтирующий теневую копию в каталог символическую ссылку, будет выглядеть следующим образом (Рисунок 30):

```
$DeviceObject = "\\?\GLOBALROOT\Device\HarddiskVolumeShadowCopy3\"
cmd /c mklink /D Shadow $DeviceObject
```

Рисунок 30

После этого, перейдя в папку «.\Shadow» можно будет получить доступ к содержимому теневой копии.

6.4. Удаление теневой копии

Для удаления выбранной теневой копии можно воспользоваться скриптом:

```
$ShadowID = "{D74C20B7-D797-4B15-9CF4-27D722D6F8DD}"
Get-CimInstance -Class Win32_ShadowCopy | Where-Object ID -eq $ShadowID |
Remove-CimInstance
```

Для удаления всех теневых копий достаточно выполнить скрипт:

```
Get-CimInstance -ClassName Win32_ShadowCopy | Remove-CimInstance
```

Обратите внимание на эту команду. Если вы хотите с помощью теневых копий защититься от вирусов-шифровальщиков, то никогда не работайте с правами системного администратора и не отключайте UAC. В противном случае подобный миниатюрный скрипт, выполненный трояном, полностью уничтожит вашу защиту.

6.5. Информационная безопасность теневых копий

Теневые копии, как любые сильные механизмы, могут быть использованы как во благо, так и во вред.

Как мы отмечали выше, теневые копии при соблюдении дополнительных условий, являются простой и доступной защитой от вирусов-шифровальщиков. Но в то же время в тени могут скрывать и сами вредоносы.

Возможен следующий сценарий. При заражении машины вредонос создаёт теневую копию и помещает туда своё тело, затем он удаляет себя из локального диска и запускается только из теневой копии. Для этого он создаёт ссылку, как мы делали это в примере выше, а после запуска удаляет её. Получается своеобразный псевдо бесфайловый вариант вируса.

7. Преодоление ограничений на максимальную длину пути

С древних времён, максимальный размер пути в файловой системе ограничен 260 символами. Когда размер имени файла был всего восемь символов, этого, вероятно, было достаточно, но в современных условиях создаёт проблемы.

Благо, с Windows 10 версии 1607 есть возможность существенно ослабить это ограничение. Для этого необходимо добавить в реестр следующее значение:

```
Windows Registry Editor Version 5.00
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\FileSystem]
"LongPathsEnabled"=dword:00000001
```

В итоге программы, декларирующие в своём манифесте элемент «longPathAware» (например, PowerShell), смогут работать с путями в файловой системе, размер которых превышает 260 символов. Однако даже для них сохранится ряд ограничений:

- Максимальный размер пути ограничивается 32,767 символами;
- Длина компонентов составного пути: промежуточных каталогов или имени файла, как правило, не может превышать 255 символов. Проверить актуальное значение можно с помощью утилиты fsutil. Например, для диска C: её вызов будет выглядеть так (требуются права администратора):

```
fsutil fsinfo volumeinfo C:
```

Здесь параметр «Max component Length» как раз и показывает максимальную длину компонента составного пути (Рисунок 31).

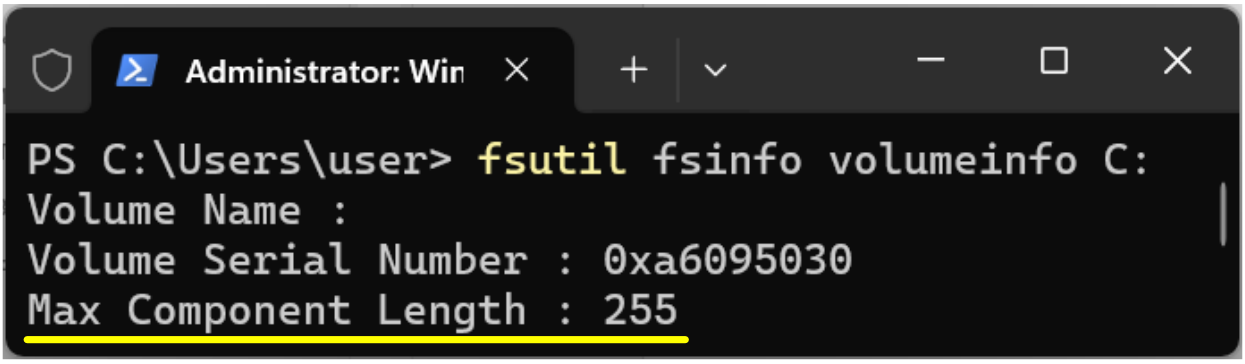


Рисунок 31

8. Тонкости задания файловых путей в командлетах PowerShell

Множество командлетов PowerShell могут использовать маски для указания сразу нескольких файлов. В большинстве случаев, особенно при ручной покомандной работе, это очень удобно, но в скриптах может привести к неожиданным последствиям.

Рассмотрим пример. Создадим два файла «book.txt», «cook.txt». Запишем в них значения: «book» и «cook» соответственно:

```
'book' > book.txt
'cook' > cook.txt
```

Теперь выведем значение этих файлов в консоль. Сделать это мы можем одной командой (Рисунок 32):

```
cat [a-z]ook.txt
```

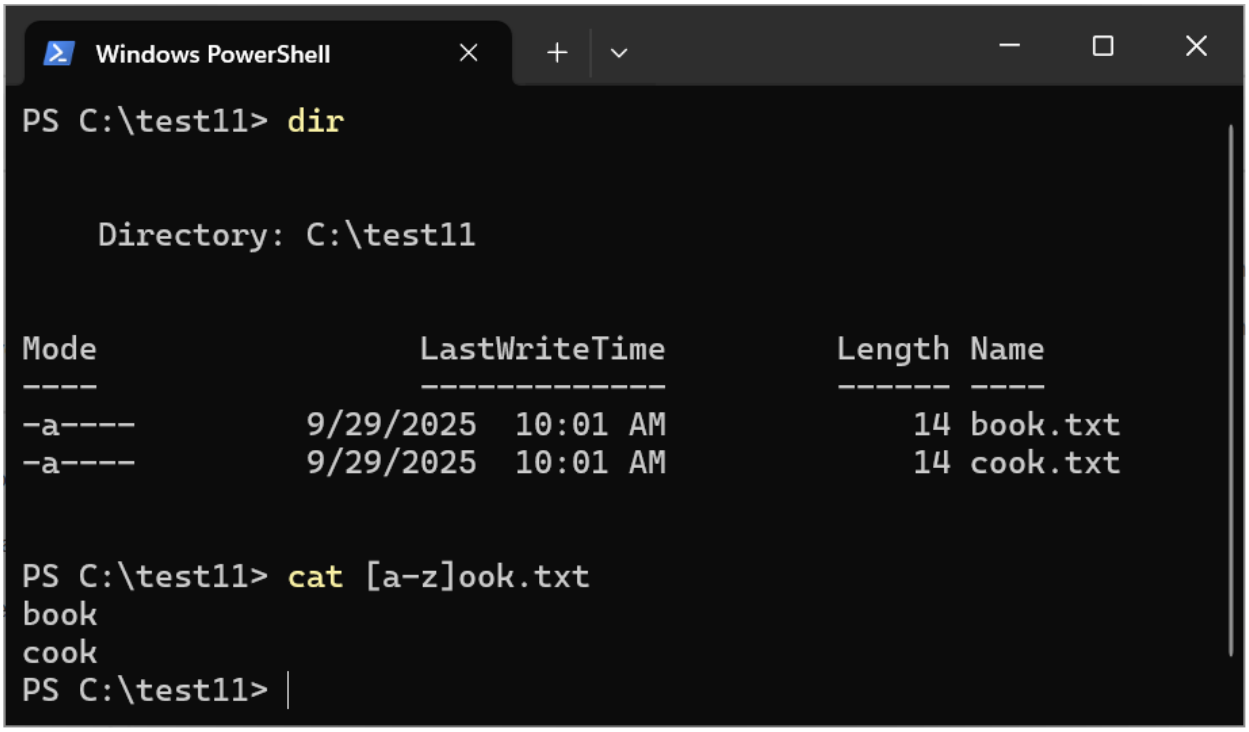


Рисунок 32

Теперь запустим проводник Windows и создадим в том же каталоге файл с именем «[a-z]ook.txt» и содержимым «[a-z]ook» (Рисунок 33).

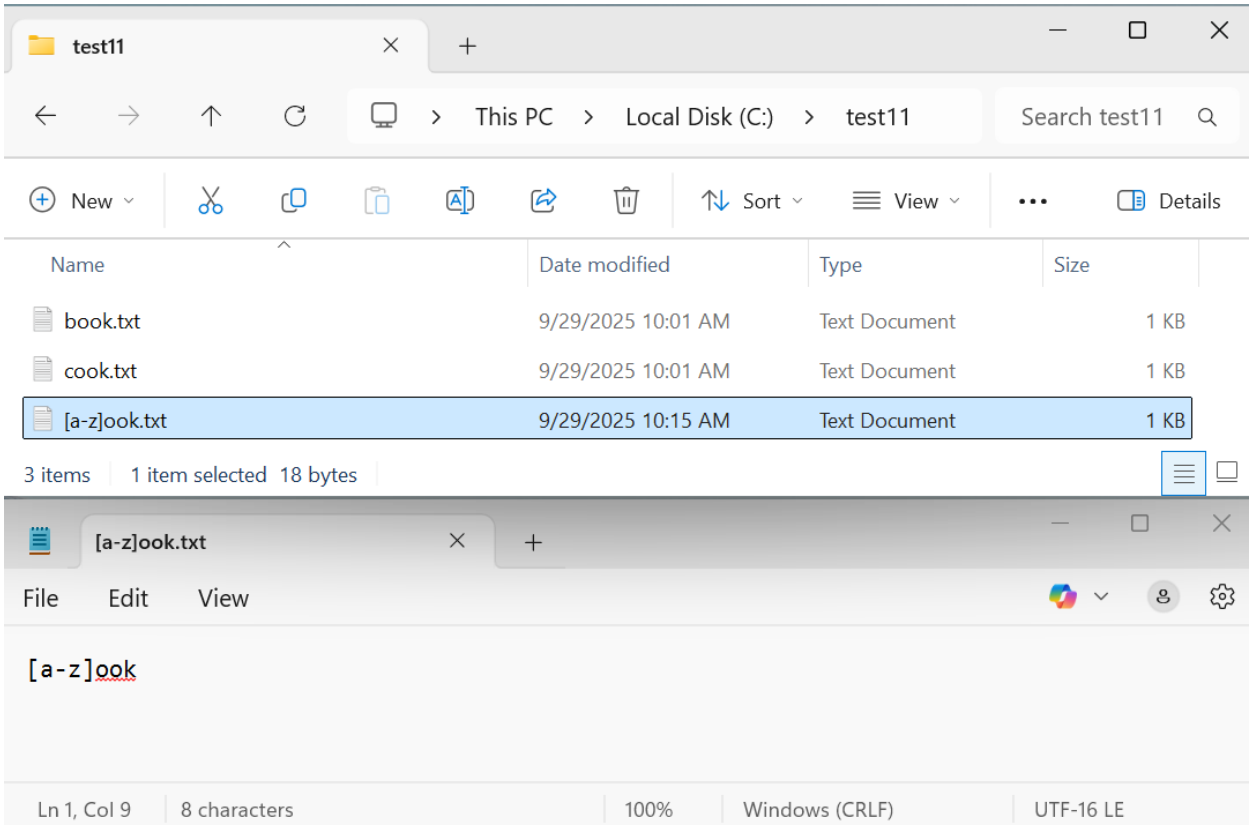


Рисунок 33

Вернёмся в PowerShell и попытаемся просмотреть содержимое файла «[a-z]ook.txt»:

```
cat [a-z]ook.txt
```

Секундочку – мы получаем (Рисунок 34) совершенно неверный ответ! Ведь содержимое файла «[a-z]ook.txt» должно быть «[a-z]ook».

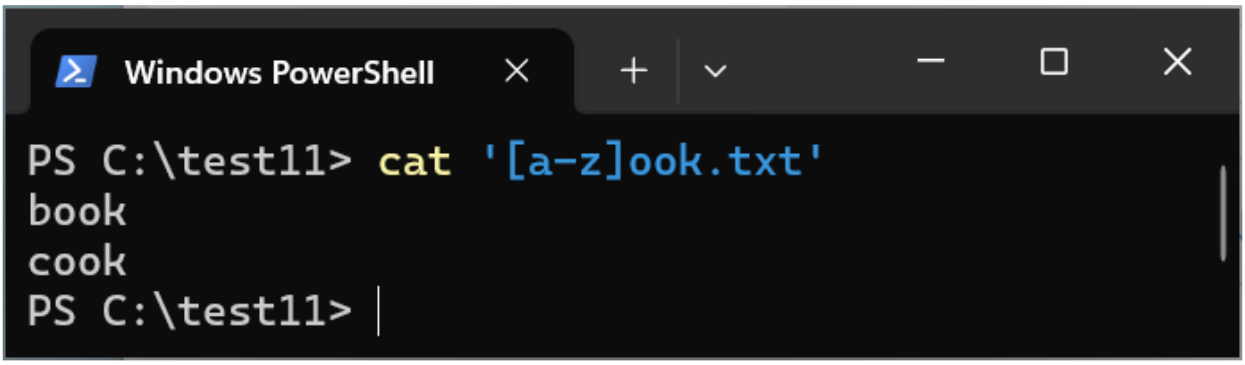


Рисунок 34

Ясно, что проблема в том, что командлет «cat» интерпретирует имя как маску, а нам надо, чтобы он использовал его как явное значение. Для одного файла

исправить ситуацию довольно легко: достаточно экранировать специальные символы «[» и «]», интерпретируемые «cat» как символы маски (Рисунок 35).

```
cat '[a-z]ook.txt'
```

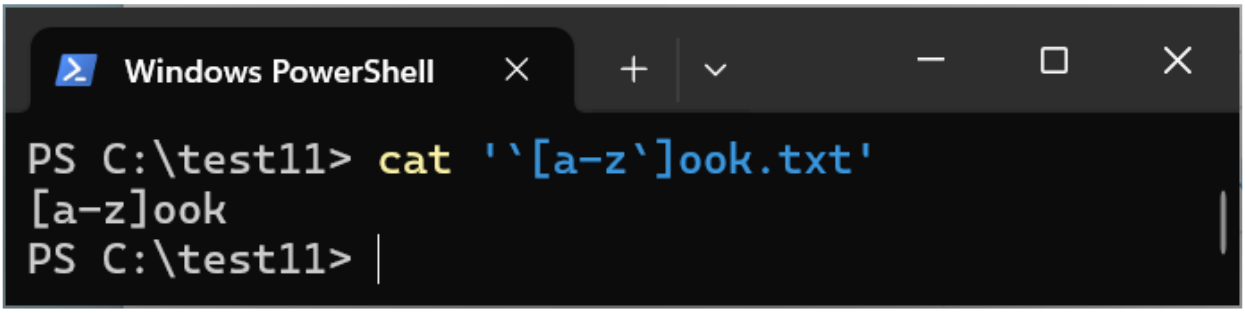


Рисунок 35

Но что делать, если нам нужно написать скрипт, скажем резервного копирования, который обрабатывает множество файлов, и мы понятия не имеем, встречаются ли среди них те, чьи имена могут интерпретироваться как маски, или нет?

Первая идея – написать функцию, экранирующую специальные символы в именах файлов. Однако так делать категорически нельзя, поскольку это радикально снизит производительность массовых файловых операций.

Благо, есть гораздо более простое решение: использовать для указания путей не ключ «Path» (используемый по умолчанию), а ключ «LitteralPath», отключающий использование масок и обрабатывающий имя файла как оно есть. С использованием этого ключа просмотр содержимого файла «[a-z]ook.txt» будет выглядеть так (Рисунок 36):

```
cat -LiteralPath '[a-z]ook.txt'
```

Рисунок 36

Ключ «LiteralPath» поддерживается множеством PowerShell командлетов: Copy-Item, Remove-Item, Set-ItemProperty, Get-Content и прочими. Поэтому можно смело дать совет: если вы пишете скрипты, проводящие массовые файловые операции, используйте везде, где только возможно, указание путей через «LiteralPath». Это

может спасти от бессонных ночей отладки, когда ваша разработка «валится», встретив файл «Sales [august].xlsx».