Глава 11.Обсуждение различных вопросов по разработке драйверов

В этой последней главе книги мы рассмотрим различные темы, которые не соответствовали предыдущем главам.

В этой главе:

• Подпись драйвера.

• Средство проверки драйверов.

• Использование нативных API.

• Драйверы фильтров.

• Монитор устройства.

• Подключение драйвера.

• Библиотеки ядра.

Подпись драйвера

Драйверы ядра - Это единственный официальный механизм для добавления кода в ядро Windows.

Драйверы могут вызвать сбой системы . Ядро Windows не делает различий между «более важными» и «менее важными» драйверами.

Microsoft естественно, хотелось бы, чтобы Windows была стабильной, без сбоев или нестабильности системы.

Начиная с Windows Vista, в 64-битных системах Microsoft требует, чтобы драйверы были подписаны с использованием надлежащего сертификата получено из центра сертификации (CA). Без подписи драйвер не загрузится.

Гарантирует ли подпись драйвера его качество ? Гарантирует ли это, что система не выйдет из строя?

Нет, это только гарантирует, что файлы драйвера не изменились с тех пор, как он покинул издателя драйвера и что сам издатель аутентичен. Это не панацея против ошибок в драйверах, но дает уверенность в самом драйвере.

Также это защищает от руткитов на уровне ядра, ведь после введения такой подписи, практически исчезли руткиты, т. к. авторам вредоносного софта нет смысла покупать подпись драйверов.)

Для аппаратного драйвера Microsoft требует, чтобы он прошел Лабораторию качества оборудования Windows.

WHQL тесты, содержащие строгие тесты на стабильность и функциональность драйвера.

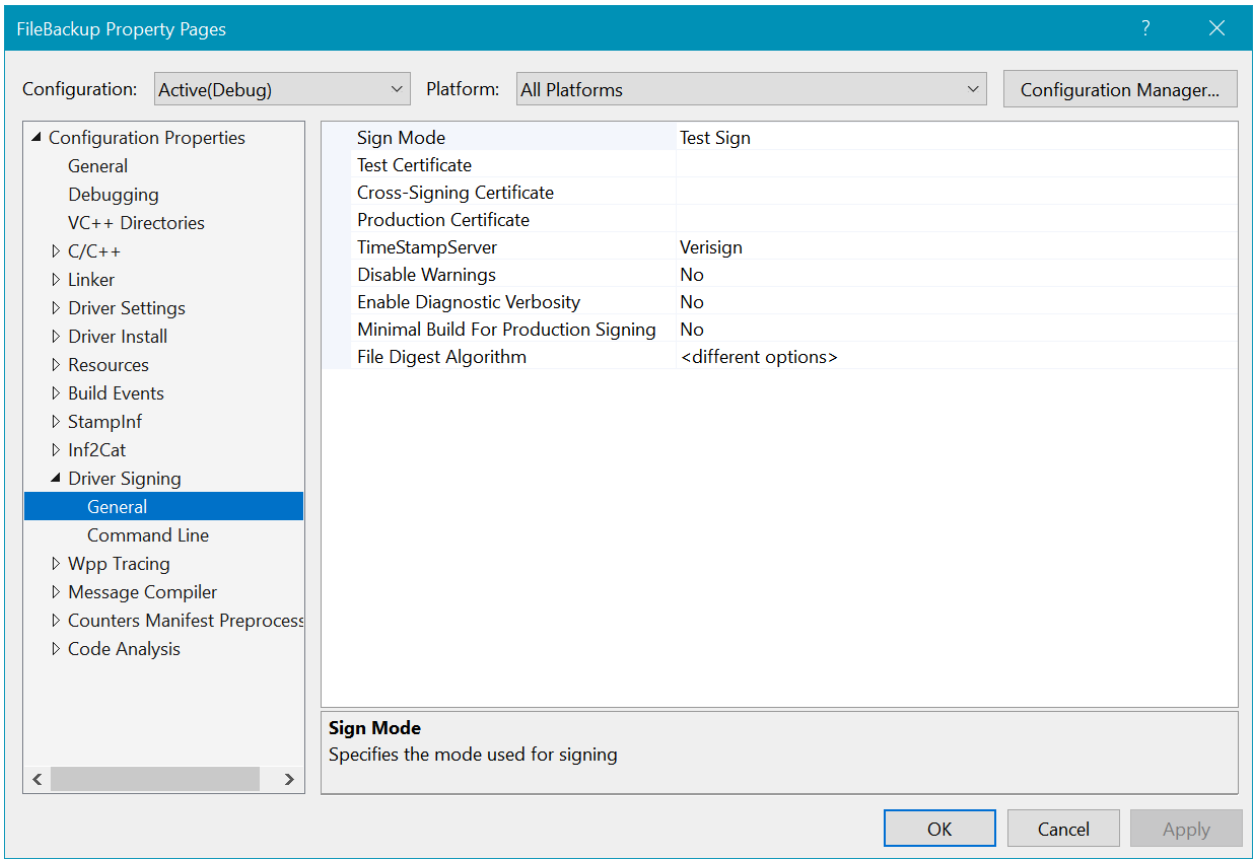
Если драйвер проходит тесты, то он получает знак качества Microsoft, который издатель драйвера может рекламировать как знак качества и доверия. Еще одно последствие передачи WHQL - доступ к драйверу через «Центр обновления Windows», который важен для некоторых издателей.

Первым шагом при подписании драйвера является получение надлежащего сертификата от центра сертификации (например,Verisign, Globalsign, Digicert, Symantec и другие), по крайней мере, для подписи кода ядра.

CA проверит личность запрашивающей компании и, если все в порядке, выдаст сертификат. Загруженный сертификат можно установить в хранилище сертификатов. Поскольку сертификат должен храниться в секрете и не допускать утечки, обычно он устанавливается на выделенном сборочном компьютере и в драйвере процесс подписания выполняется как часть процесса сборки.

Фактическая операция подписи выполняется с помощью инструмента SignTool.exe, входящего в состав Windows SDK. Вы можете использовать Visual Studio для подписи драйвера, если сертификат установлен в хранилище сертификатов на локальной машине.

На рисунке показаны свойства подписи в Visual Studio.

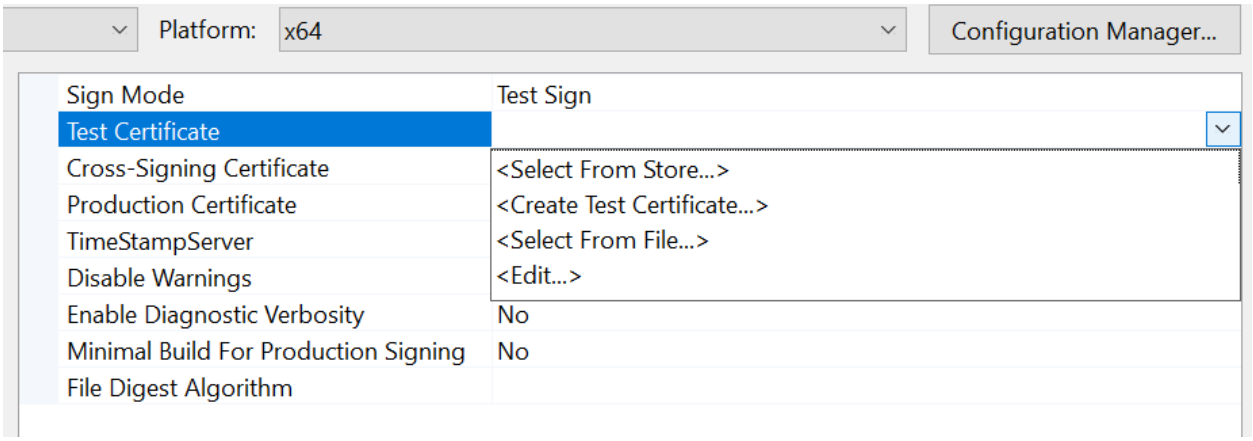


Visual Studio предоставляет два типа подписи: тестовая подпись и рабочая подпись. С тестовым подписанием, обычно используется сертификат который созданан локально и которому не доверяют глобально.

Это позволяет тестирование драйвера в системах, настроенных с включенной тестовой подписью, как мы это делали в книге.

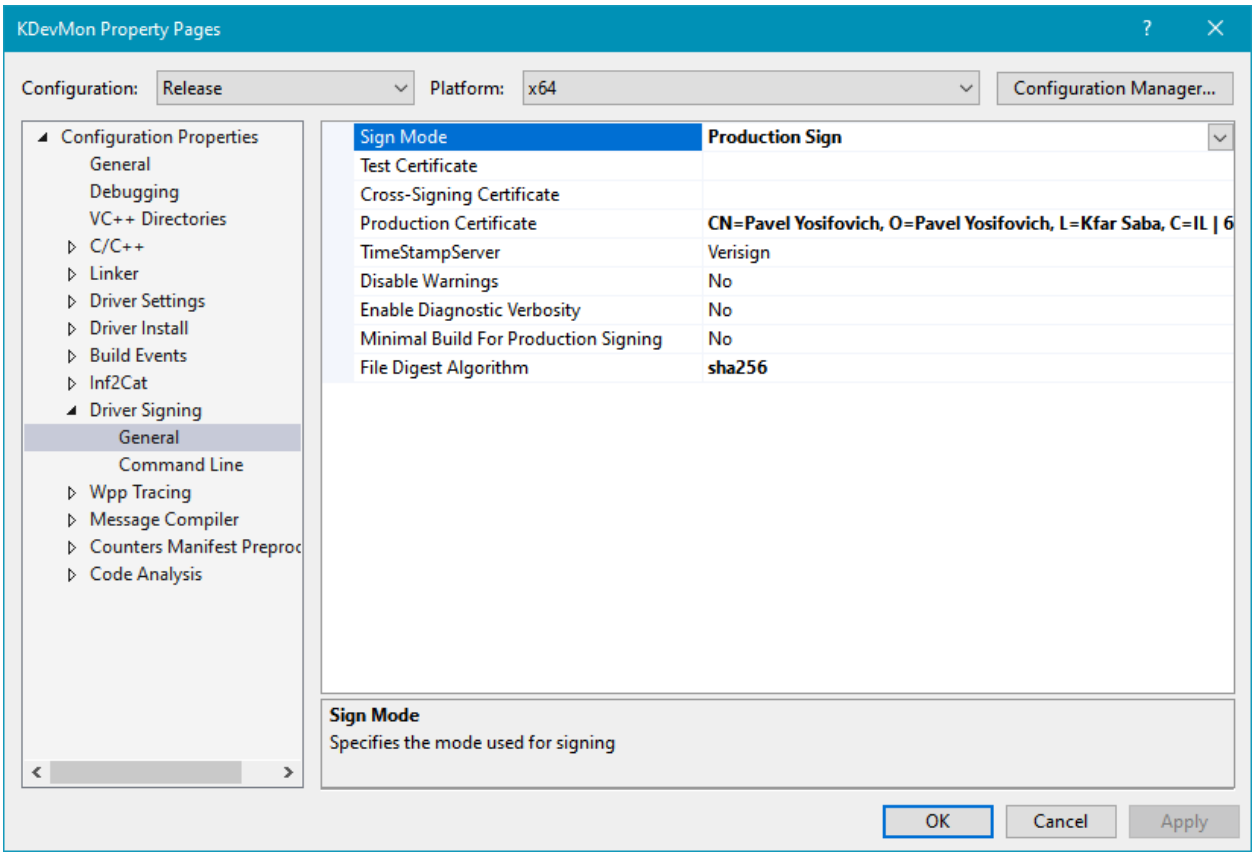
Подпись производственной среды заключается в использовании реального сертификата для подписи драйвера для производственного использования.

Сертификаты тестирования могут быть созданы по желанию с помощью Visual Studio при выборе сертификата, как показано на следующем на рисунке.



На следующем рисунке показан пример подписания производственной сборки драйвера в Visual Studio.

Алгоритм должен быть SHA256, а не более старый и менее безопасный SHA1.



Работа с различными процедурами регистрации и подписания драйверов выходит за рамки этой книги. В последние годы ситуация усложнилась из-за новых правил и процедур Microsoft.

Более подробно про подписание драйверов можно почитать здесь:https://docs.microsoft.com/en-us/windows-hardware/drivers/install/kernel-mode-code-signing-policy--windows-vista-and-later-

Средство проверки драйверов

Driver Verifier - это встроенный инструмент, который существует в Windows, начиная с Windows 2000.

Его цель — помочь выявлять ошибки драйверов и неправильные методы кодирования.

Например, предположим, что ваш драйвер вызывает BSOD каким-то образом, но код драйвера не находится ни в одном стеке вызовов в файле аварийного дампа.

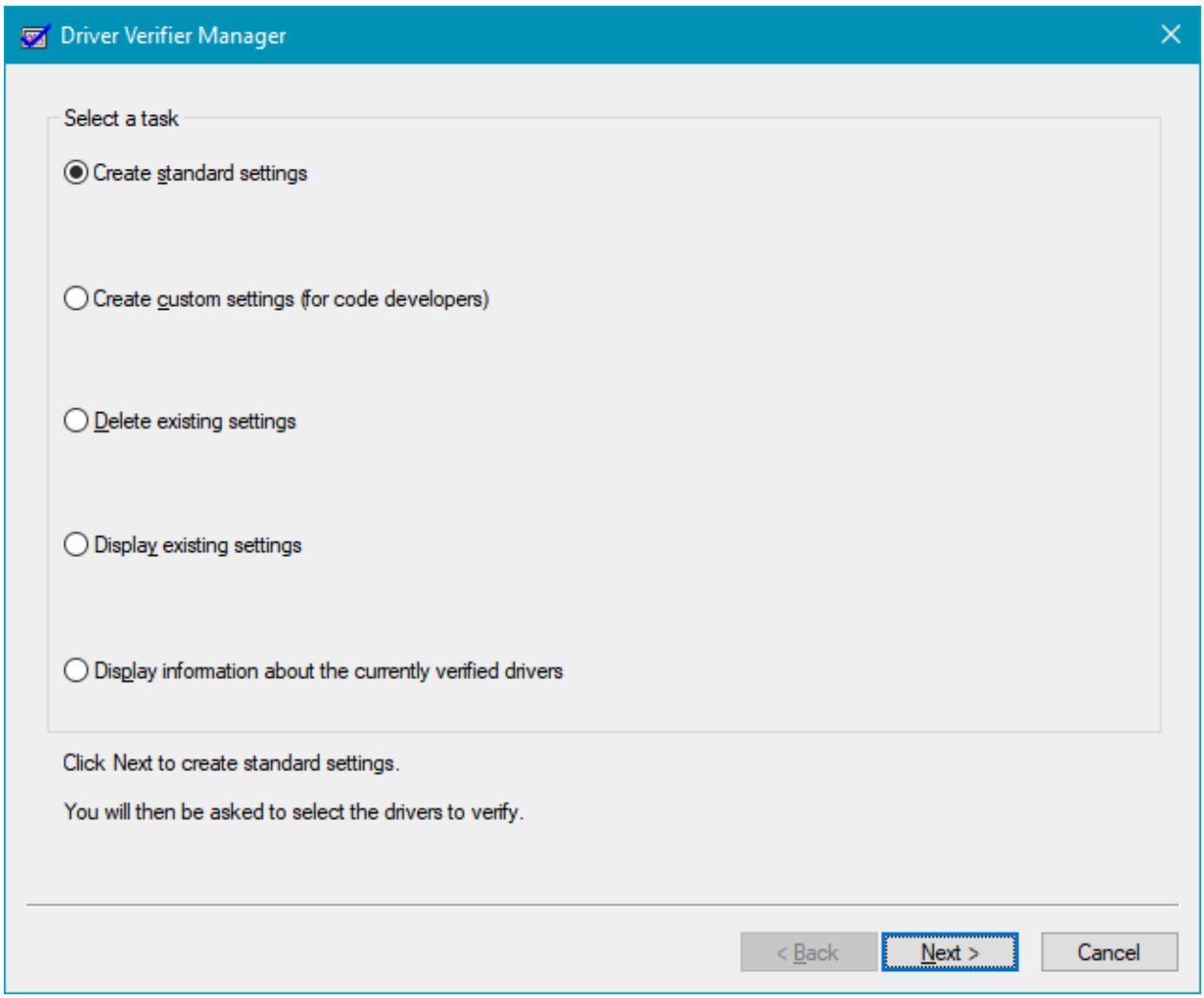
Обычно это означает что ваш драйвер сделал что-то, что не было фатальным в то время, например, записал что-то за пределами одного из своих выделенных буферов, где эта память, к сожалению, была выделена другому драйверу или ядру.

В этот момент сбоя нет. Однако когда-нибудь позже этот драйвер или ядро будет использовать эти данные и, скорее всего, вызовет сбой системы. Нет простого способа связать сбой с драйвером-нарушителем. Средство проверки драйверов предлагает возможность выделить память для драйвера в свой собственный «специальный» пул, в котором страницы с более высокими и низкими адресами недоступны, и поэтому будут вызывать немедленный сбой при переполнении или опустошении буфера, что упрощает выявление проблемных мест.

Средство проверки драйверов имеет графический интерфейс и интерфейс командной строки и может работать с любым драйвером.

Самый простой способ начать работать с верификатором - открыть его, набрав verifier в диалоговом окне «Выполнить».

В любом случае, верификатор представляет свой начальный пользовательский интерфейс, показанный на следующем рисунке.



Необходимо выбрать две вещи: тип проверок, выполняемых верификатором, и драйверы которые следует проверить. Первая страница мастера посвящена самим проверкам.

Варианты проверок:

• Создать стандартные настройки - выбирает заранее определенный набор проверок, которые необходимо выполнить. Мы увидим полный список доступных проверок на второй странице, каждая с флажком Standard или Additional.

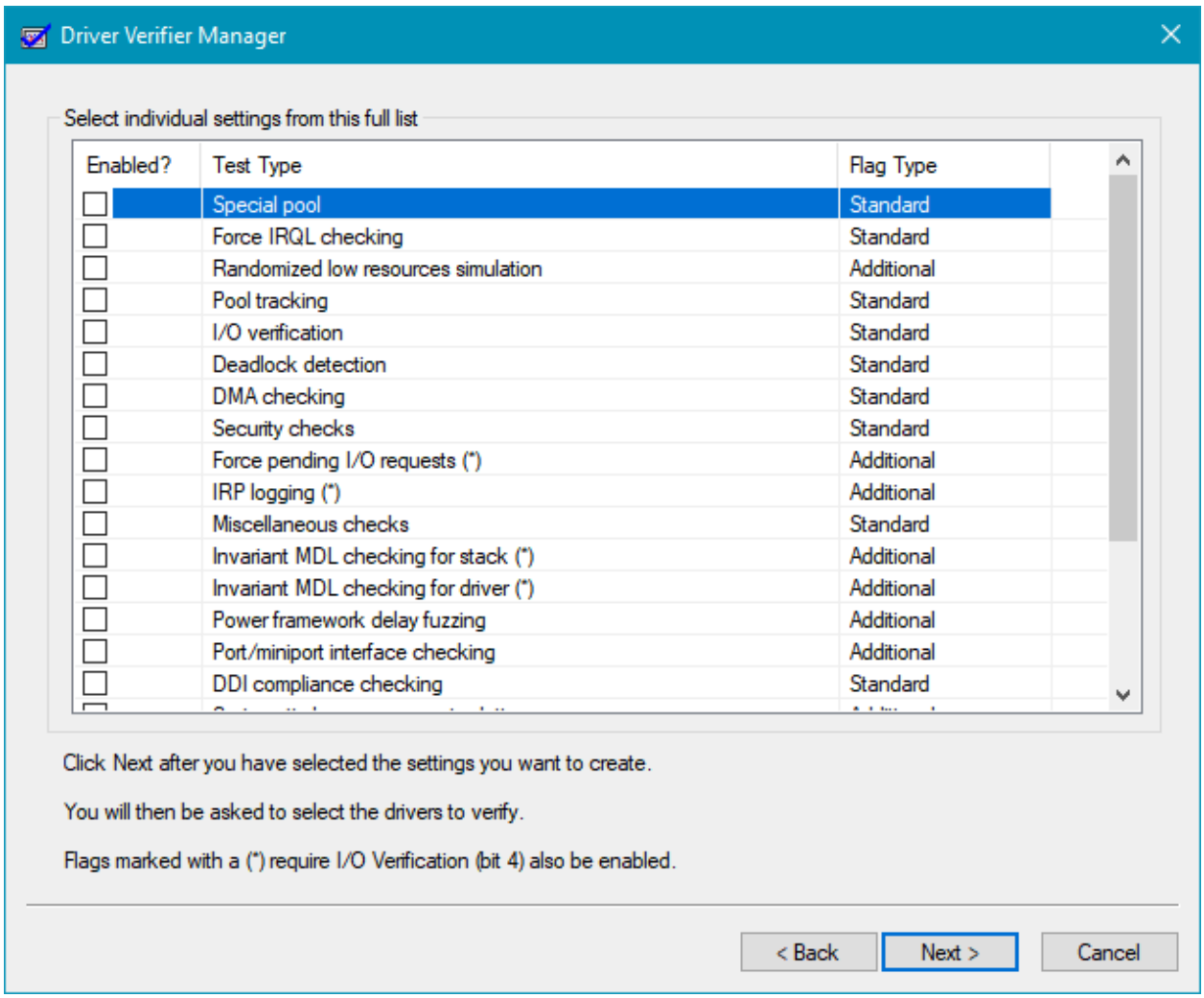
Все отмеченные как Standard выбираются этой опцией автоматически.

• Создание пользовательских настроек, позволяет детально выбирать проверки, перечисляя все доступные проверки.

• Удалить существующие настройки, удаляет все существующие настройки верификатора.

• Показать существующие настройки, показывает текущие настроенные проверки и драйверы, для которых это применяется.

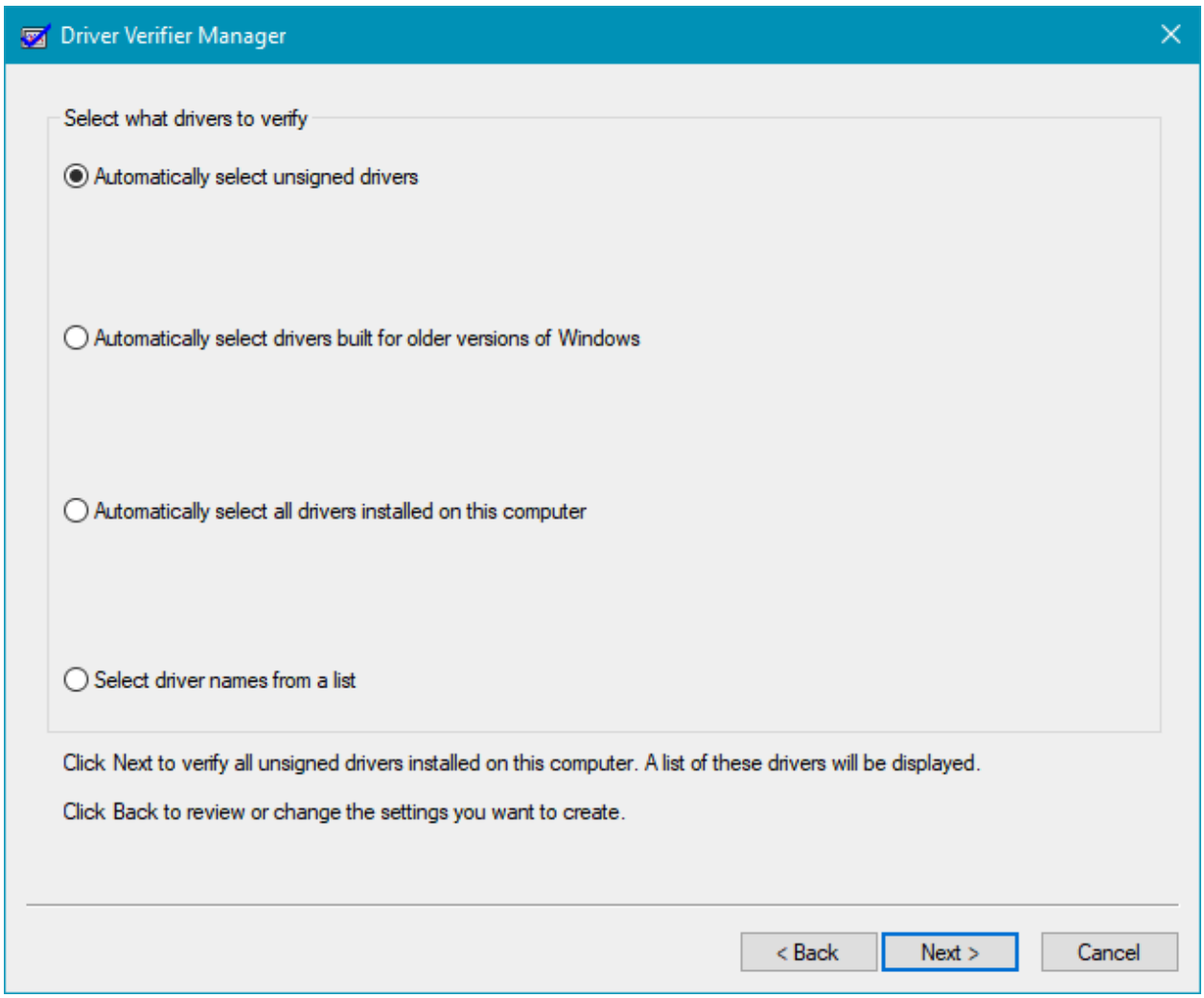
• Показать информацию о проверенных в настоящее время драйверах, показывает собранную информацию для драйверов, работающих под верификатором в предыдущем сеансе.



При выборе Создать пользовательские настройки отображается доступный список настроек верификатора, список, который увеличился значительно, с первых дней Driver Verifier.

Флаг Standard указывает, что эта настройка часть стандартных настроек, которую можно выбрать на первой странице мастера.

После того, как настройки были выбраны, Verifier показывает следующий шаг для выбора драйверов, которые нужно проверить, это показано на следующем рисунке.



Вот возможные варианты:

• Автоматический выбор неподписанных драйверов, наиболее актуален для 32-битных систем, как и для 64-битных систем.

При нажатии кнопки «Далее» будут перечислены такие драйверы.

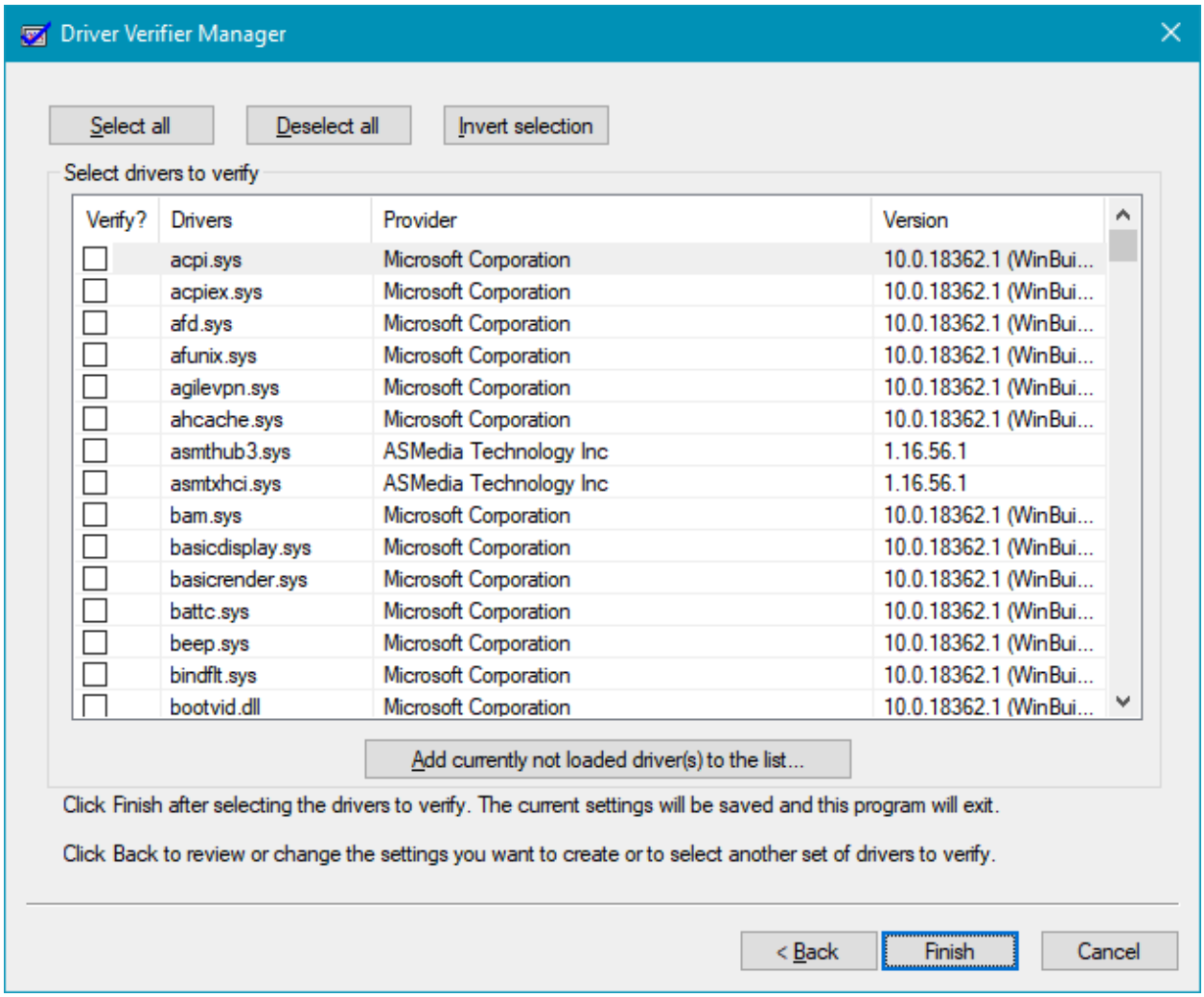
• Автоматический выбор драйверов, созданных для более старых версий Windows, является устаревшей настройкой для NT 4.

• Автоматический выбор всех драйверов, установленных на компьютере, - это универсальный вариант, который выбирает все драйверы. Теоретически это может быть полезно, если у вас есть система, которая дает сбой, но никто не знает, кто вызвал сбой. Однако этот параметр не рекомендуется, так как он замедляет работу машины (у верификатора есть свои затраты), потому что верификатор перехватывает различные операции (на основе предыдущих настроек) и обычно приводит к использованию большего объема памяти.

Так лучше в такой ситуации, чтобы выбрать первые (скажем) 15 драйверов, посмотреть, обнаружит ли проверяющий плохой драйвер, и если нет, выберите следующие 15 драйверов и так далее.

• Выберите имена драйверов из списка * - лучший вариант для использования, где Verifier представляет список драйверов.

Если рассматриваемый драйвер в данный момент не загружен, щелкните по Добавить в список не загруженные драйверы ... позволяет перейти к соответствующим файлу(ам) SYS.

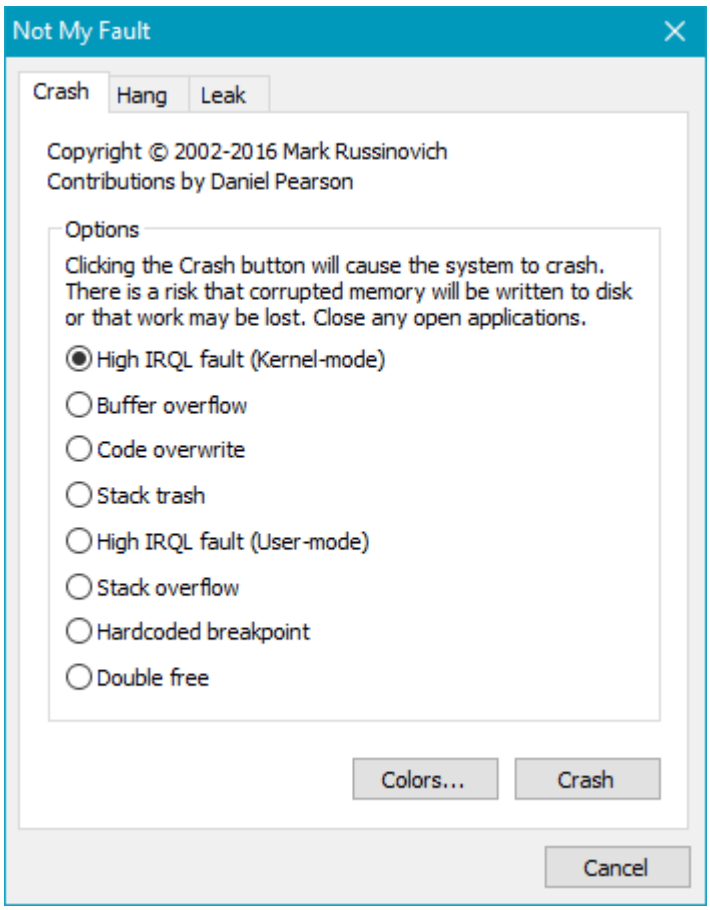


Наконец, нажатие кнопки «Завершить изменения» делает настройки постоянными до тех пор, пока они не будут отозваны, и систему обычно необходимо перезапустить, чтобы верификатор мог инициализировать себя и перехватить драйверы, особенно если они в настоящее время выполняются.

Примеры сеансов проверки драйверов

Начнем с простого примера с использованием инструмента NotMyFault от Sysinternals. Как обсуждалось в главе 6 этот инструмент можно использовать для сбоя системы различными способами.

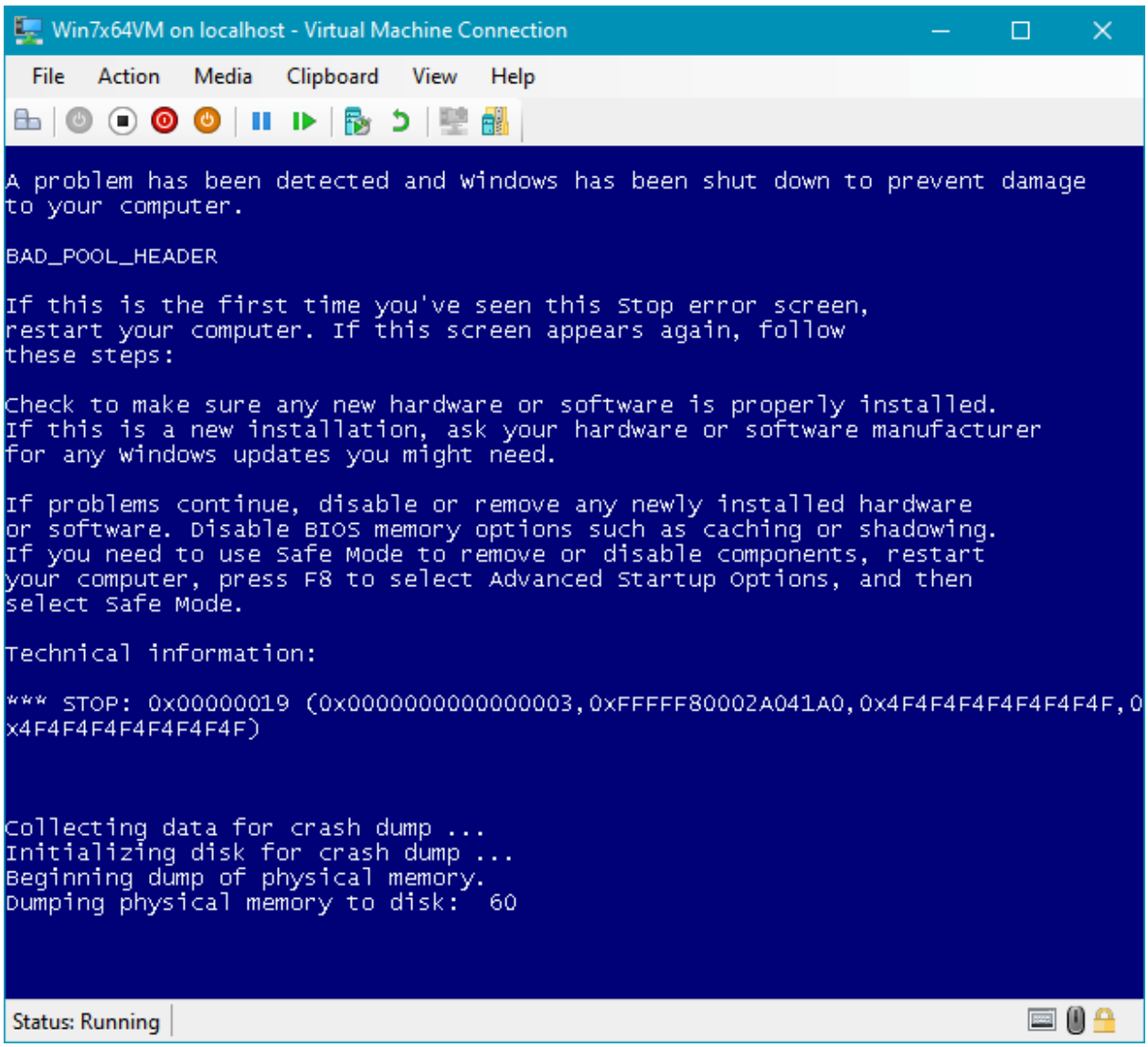
На рисунке показан NotMyFault.



Давайте попробуем это (на виртуальной машине). Может потребоваться несколько щелчков мышью по Crash, чтобы действительно вывести систему из строя.

На рисунке показан результат на виртуальной машине Windows 7 после нескольких нажатий на сбой и нескольких секунд ожиданий.

Обратите внимание на код BSOD (BAD_POOL_HEADER). Хорошее предположение - переполнение буфера.



Загрузим полученный файл дампа и просмотрим стек вызовов:

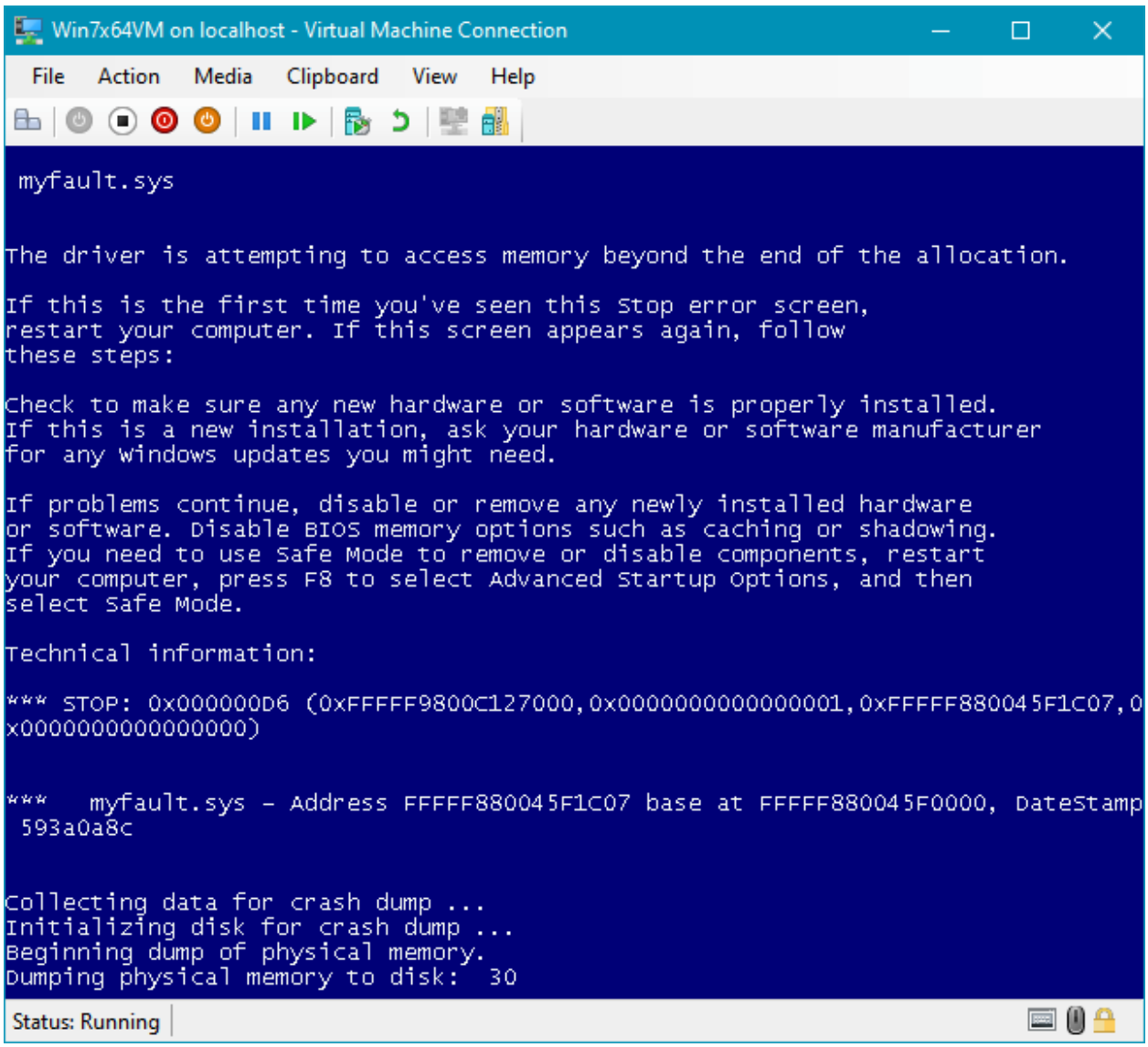
```
1: kd> k
 # Child-SP          RetAddr           Call Site
00 fffff880`054be828 fffff800`029e4263 nt!KeBugCheckEx
01 fffff880`054be830 fffff800`02bd969f nt!ExFreePoolWithTag+0x1023
02 fffff880`054be920 fffff800`02b0669b nt!ObpAllocateObject+0x12f
03 fffff880`054be990 fffff800`02c2f012 nt!ObCreateObject+0xdb
04 fffff880`054bea00 fffff800`02b1a7b2 nt!PspAllocateThread+0x1b2
05 fffff880`054bec20 fffff800`02b20d95 nt!PspCreateThread+0x1d2
06 fffff880`054beea0 fffff800`028aaad3 nt!NtCreateThreadEx+0x25d
07 fffff880`054bf5f0 fffff800`028a02b0 nt!KiSystemServiceCopyEnd+0x13
```

```
08 fffff880`054bf7f8 fffff800`02b29a60 nt!KiServiceLinkage
09 fffff880`054bf800 fffff800`0286ac1a nt!RtlpCreateUserThreadEx+0x138
0a fffff880`054bf920 fffff800`0285c1c0 nt!ExpWorkerFactoryCreateThread+0x92
0b fffff880`054bf9e0 fffff800`02857dd0 nt!ExpWorkerFactoryCheckCreate+0x180
0c fffff880`054bfa60 fffff800`028aaad3 nt!NtReleaseWorkerFactoryWorker+0x1a0
0d fffff880`054bfae0 00000000`76e1ac3a nt!KiSystemServiceCopyEnd+0x13
```

Ясно, что MyFault.sys нигде не найти.

Теперь давайте проведем тот же эксперимент с Driver Verifier. Выберите стандартные настройки и перейдите к System32\Drivers, чтобы найти MyFault.sys (если он в данный момент не запущен). Перезагрузите систему, запустите NotMyFault снова, выберите Buffer overflow и нажмите Crash.

Вы заметите, что система вылетает немедленно с BSOD, аналогичным показанному на рисунке выше.



Сам BSOD сразу говорит о MyFault.sys. Файл дампа подтверждает это следующим стеком вызовов:

```
0: kd> k
 # Child-SP          RetAddr           Call Site
00 fffff880`0651c378 fffff800`029ba462 nt!KeBugCheckEx
01 fffff880`0651c380 fffff800`028ecb96 nt!MmAccessFault+0x2322
02 fffff880`0651c4d0 fffff880`045f1c07 nt!KiPageFault+0x356
03 fffff880`0651c660 fffff880`045f1f88 myfault+0x1c07
04 fffff880`0651c7b0 fffff800`02d63d56 myfault+0x1f88
05 fffff880`0651c7f0 fffff800`02b43c7a nt!IovCallDriver+0x566
06 fffff880`0651c850 fffff800`02d06eb1 nt!IopSynchronousServiceTail+0xfa
07 fffff880`0651c8c0 fffff800`02b98296 nt!IopXxxControlFile+0xc51
```

У нас нет символов для MyFault.sys, но очевидно, что он виноват.

В качестве другого примера мы применим стандартные настройки верификатора к драйверу DelProtect3 из главы 10.

После установки драйвера мы пытаемся добавить папку для защиты от удаления вот так:

delprotectconfig3 add c:\temp

Получаем сбой системы, инициированный верификатором, с кодом BAD_POOL_CALLER (0xc2).

Открытие получившийся файла дампа вызовом analysis -v приводит к анализу, частью которого является следующее:

```
BAD_POOL_CALLER (c2)
The current thread is making a bad pool request.  Typically this is at a bad IRQL le\
vel or double freeing the same allocation, etc.
Arguments:
Arg1: 000000000000009b, Attempt to allocate pool with a tag of zero.  This would mak\
e the pool untrackable and worse, corrupt the existing tag tables.
Arg2: 0000000000000001, Pool type
Arg3: 000000000000000c, Size of allocation in bytes
Arg4: fffff8012dcd1297, Caller's address.

FAULTING_SOURCE_CODE:
   113:    return pUnicodeString;
   114: }
   115:
   116: wchar_t* kstring::Allocate(size_t chars, const wchar_t* src) {
>  117:    auto str = static_cast<wchar_t*>(ExAllocatePoolWithTag(m_Pool,
              sizeof(WCHAR) * (chars + 1), m_Tag));
   118:    if (!str) {
   119:        KdPrint(("Failed to allocate kstring of length %d chars\n", chars));
   120:        return nullptr;
   121:    }
   122:  if (src) {
```

Действительно, объект kstring, используемый в коде, не указывал ненулевой тег для его выделения.

Код ошибки является частью функции ConvertDosNameToNtName:

kstring symLink(L"\\??\\");

Исправить достаточно просто:

kstring symLink(L"\\??\\", PagedPool, DRIVER_TAG);

Возможно, класс kstring следует изменить так, чтобы он требовал тега без значения по умолчанию для нуля.

Использование нативных API

Как мы видели в главах 1 и 10, Windows Native API доступен в пользовательском режиме через NtDll.Dll,

Драйверы ядра могут пользоваться одним и тем же API, что уже было продемонстрировано различными Zw-функциями.

Однако очень мало этих функций задокументировано, некоторые из них практически не задокументированы - например NtQuerySystemInformation, NtQueryInformationProcess, NtQueryInformationThread и подобные функции.

Мы уже встречались NtQueryInformationProcess :

```
NTSTATUS ZwQueryInformationProcess(
            _In_
            HANDLE
            ProcessHandle,
            _In_
            PROCESSINFOCLASS ProcessInformationClass,
            _Out_
            PVOID
            ProcessInformation,
            _In_
            ULONG
            ProcessInformationLength,
            _Out_opt_ PULONG
            ReturnLength);
```

Перечисление PROCESSINFOCLASS обширно и содержит около 70 значений в WDK для сборки 18362.

Если вы внимательно изучите список, вы обнаружите, что некоторые значения отсутствуют.

В официальной документации задокументировано только 6 значений (на момент написания), что очень досадно.

Более того, фактический список поддерживаемых намного длиннее, чем официально предоставляется Microsoft.

К счастью, проект с открытым исходным кодом на Github под названием Process Hacker предоставляет большую часть отсутствующей информации.

Process Hacker имеет все определения нативного API в отдельном проекте, удобном для использования в других проектах.

URL-адрес: https://github.com/processhacker/phnt

PROCESSINFOCLASS в этом репозитории в настоящее время имеет 99 записей, без каких-либо пропущенных.

Единственное предостережение при использовании этих определений заключается в том, что теоретически Microsoft может изменить практически любой из них без предупреждения, поскольку большинство из них недокументировано.

Однако это маловероятно, поскольку может сломать довольно много приложений, некоторые из которых от Microsoft.

Например, Process Explorer использует некоторые из этих «недокументированных» функций.

Тем не менее, лучше всего тестировать приложения и драйверы которые используют эти функции для всех версий Windows, где будут эти приложения или драйверы работать.

Драйверы фильтров

Модель драйвера Windows ориентирована на устройства, как мы уже видели в главе 7.

Устройства могут быть слоями друг - друга.

Эта же модель используется для драйверов файловой системы, которые мы использовали в главе 10

Однако модель фильтрации является универсальной и может использоваться для других типов устройств.

В этом разделе мы подробнее рассмотрим общую модель фильтрации устройств, которую мы сможем применить к широкому спектру устройств.

API ядра предоставляет несколько функций, которые позволяют размещать одно устройство поверх другого.

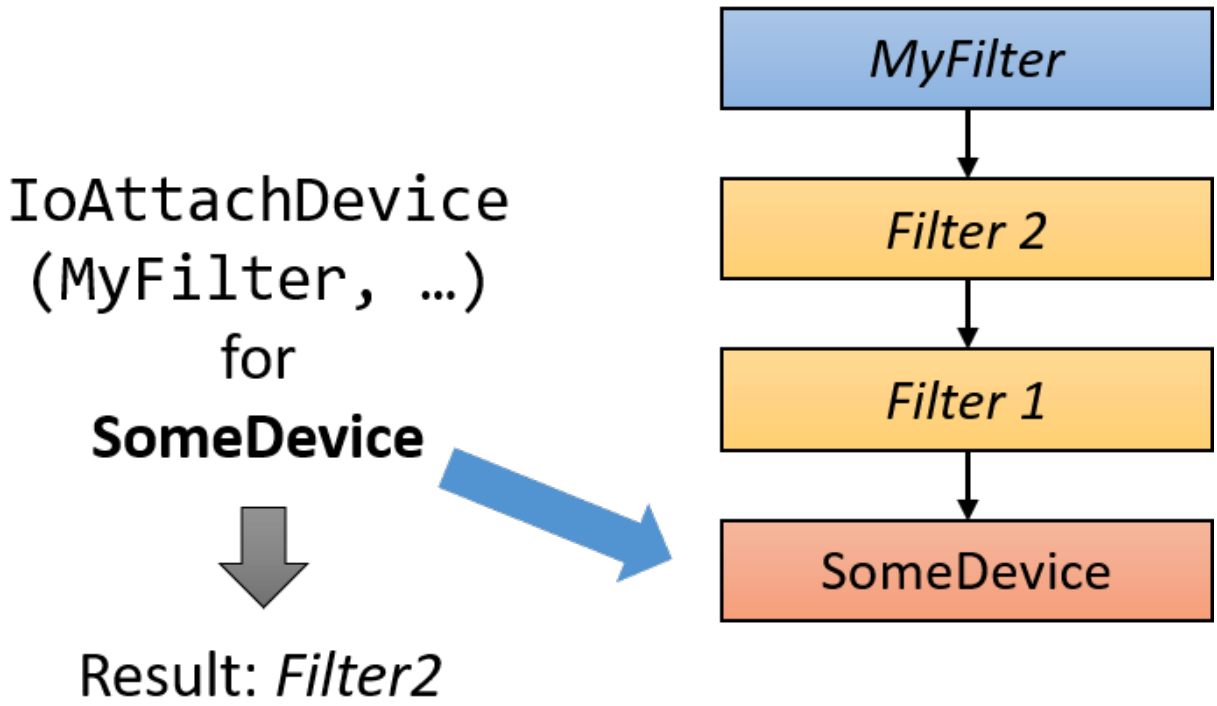
Самым простым, вероятно, является IoAttachDevice, который принимает объект устройства для присоединения и целевой именованный объект устройства для подключения.

Вот его прототип:

```
NTSTATUS IoAttachDevice (
            PDEVICE_OBJECT SourceDevice,
            _In_ PUNICODE_STRING TargetDevice,
            _Out_ PDEVICE_OBJECT *AttachedDevice);
```

Результатом функции (помимо статуса) является другой объект устройства, к которому SourceDevice был фактически прикреплен.

Это показано на следующем рисунке:



К сожалению, прикрепление к объекту устройства требует дополнительной работы. Как обсуждалось в главе 7, устройство может попросить диспетчера ввода-вывода помочь с доступом к пользовательскому буферу с помощью буферизованного ввода-вывода или прямого ввода-вывода (для запросов IRP_MJ_READ и IRP_MJ_WRITE) путем установки соответствующих флагов в DEVICE_OBJECT.

Есть еще несколько настроек, которые необходимо скопировать с подключенного устройства, чтобы убедиться, что новый фильтр выглядит так же для системы ввода-вывода.

Мы увидим эти настройки позже, когда создадим полный пример фильтра.

Какое имя устройства требуется для IoAttachDevice ?

Это именованный объект устройства в пространстве имен диспетчера объектов, доступное для просмотра с помощью инструментов WinObj, которые мы использовали ранее.

Большинство названных объектов устройства расположены в каталоге \Device\, но некоторые находятся в другом месте.

Например, если бы мы должны были прикрепить объект устройства фильтра к объекту устройства Process Explorer, имя было бы \Device\ProcExp152 (имя без учета регистра).

Другие функции для присоединения к другому объекту устройства включают IoAttachDeviceToDeviceStack и IoAttachDeviceToDeviceStackSafe, оба принимают другой объект устройства для присоединения. Эти функции в основном полезны при создании фильтров, зарегистрированных для аппаратных драйверов устройств, где объект целевого устройства предоставляется как часть узла устройства (также частично описано в главе 7).

Оба возвращают фактический объект многоуровневого устройства, как это делает IoAttachDevice.

Функция Safe возвращает правильный NTSTATUS, а первая возвращает NULL в случае ошибки. В остальном эти функции идентичны.

Как правило, код ядра может получить указатель на именованный объект устройства с помощью IoGetDeviceObjectPointer, который возвращает объект устройства и объект файла, открытый для этого устройства на основе имени устройства.

Вот прототип:

```
NTSTATUS IoGetDeviceObjectPointer (
            _In_ PUNICODE_STRING ObjectName,
            _In_ ACCESS_MASK DesiredAccess,
            _Out_ PFILE_OBJECT *FileObject,
            _Out_ PDEVICE_OBJECT *DeviceObject);
```

Желаемый доступ обычно - FILE_READ_DATA или любой другой, допустимый для файловых объектов.

Ссылка на возвращаемый файловый объект увеличивается, поэтому драйвер должен быть осторожен, т. к. нужно потом уменьшить эту ссылку (ObDereferenceObject), чтобы не было утечки ресурсов.

Возвращенное устройство объекта можно использовать в качестве аргумента для IoAttachDeviceToDeviceStack (Safe).

Реализация драйвера фильтра

Драйвер фильтра должен прикрепить объект устройства к целевому устройству, для которого требуется фильтрация.

Мы обсудим позже, когда должно произойти прикрепление.

Поскольку новый объект устройства теперь станет самым верхним устройством в стеке устройств, любой запрос, который драйвер не поддерживает, будет возвращен клиенту с ошибкой «неподдерживаемая операция».

Это означает, что DriverEntry фильтра должен регистрировать все основные коды функций, если он хочет убедиться, что базовый объект устройства продолжит функционировать.

Вот один из способов настроить это:

```
for (int i = 0; i < ARRAYSIZE(DriverObject->MajorFunction); i++)
      DriverObject->MajorFunction[i] = HandleFilterFunction;
```

Приведенный выше фрагмент кода устанавливает все основные коды функций, указывающие на одну и ту же функцию.

`HandleFilterFunction` должна, как минимум, вызывать драйвер нижнего уровня с помощью объекта устройства, полученной одной из функций.

Конечно, будучи фильтром, драйвер захочет выполнять дополнительную работу или другую работу для запросов, которые его интересуют, но все запросы, которые он выполняет должны быть перенаправлено на устройство нижнего уровня, иначе это устройство не будет работать должным образом.

Эта операция «вперед и забыть» очень распространена в фильтрах. Посмотрим, как это реализовать.

Фактический вызов, который передает IRP на другое устройство - это IoCallDriver. Тем не менее, перед его вызовом текущий драйвер должен подготовить следующее расположение стека ввода/вывода для нижнего драйвера.

Помните, что изначально диспетчер ввода-вывода инициализирует только первое местоположение стека ввода-вывода.

Драйвер может вызвать IoGetNextIrpStackLocation, чтобы получить указатель на IO_STACK_LOCATION и инициализировать его.

Однако в большинстве случаев драйвер просто хочет представить на нижний уровень ту же информацию, которую он получил сам.

Одна функция, которая может в этом помочь это IoCopyCurrentIrpStackLocationToNext, что не требует пояснений.

Эта функция, однако не просто слепо копирует расположение стека ввода-вывода:

auto current = IoGetCurrentIrpStackLocation(Irp);

auto next = IoCopyCurrentIrpStackLocationToNext(Irp);

*next = *current;

Почему? Причина связана с процедурой завершения. Вспомните из главы 7, что драйвер может настроить процедуру завершения, чтобы получать уведомление, когда IRP завершается более низким драйвером (IoSetCompletionRoutine / Ex).

Указатель завершения (и определяемый драйвером контекст) хранятся в следующем месте стека ввода-вывода, поэтому слепая копия будет дублировать процедуру завершения высшего уровня (если таковая имеется), чего мы не хотим.

Это именно то, что IoCopyCurrentIrpStackLocationToNext избегает.

Но на самом деле есть лучший способ, если драйверу не нужна процедура завершения и он просто хочет использовать «вперед и забыть», не платя за копирование данных о местоположении стека ввода/вывода.

Это достигается путем пропуска местоположения стека ввода-вывода таким образом, чтобы следующий драйвер нижнего уровня видел то же расположение стека ввода-вывода, что и этот:

IoSkipCurrentIrpStackLocation(Irp);

status = IoCallDriver(LowerDeviceObject, Irp);

IoSkipCurrentIrpStackLocation просто уменьшает расположение стека ввода-вывода внутреннего IRP, a IoCallDriver увеличивает его, по существу, заставляя нижний драйвер видеть тот же ввод-вывод, без какого-либо копирования; это предпочтительный способ, если драйвер не желает вносить изменения в запрос и не требует процедуры завершения.

Присоединение фильтров

Когда драйвер вызывает одну из функций присоединения ? Идеальное время, когда базовое устройство (цель прикрепления) создается, то есть когда создается узел устройства.

Это обычное дело в фильтрах для аппаратных драйверов устройств, где фильтры могут быть зарегистрированы в именованных значениях UpperFilters и LowerFilters, с которыми мы встречались в главе 7.

DriverObject->DriverExtension->AddDevice = FilterAddDevice;

Этот обратный вызов AddDevice вызывается, когда новое аппаратное устройство, принадлежащее драйверу, идентифицированы системой Plug & Play.

Эта процедура имеет следующий прототип:

```
NTSTATUS AddDeviceRoutine (
            _In_ PDRIVER_OBJECT DriverObject,
            _In_ PDEVICE_OBJECT PhysicalDeviceObject);
```

Система ввода-вывода предоставляет драйверу объект устройства в нижней части стека устройств (PhysicalDeviceObject или PDO) для использования при вызове IoAttachDeviceToDeviceStack (Safe).

PDO - одна из причин, по которой DriverEntry не является подходящим местом для выполнения присоединенного вызова - на данный момент PDO еще не предоставлен.

Кроме того, второе устройство того же типа может быть добавлено в систему (например, вторая USB-камера), и в этом случае DriverEntry вообще не будет вызываться.

Вот пример реализации подпрограммы AddDevice для драйвера фильтра (обработка ошибок опущена):

```
struct DeviceExtension {
	PDEVICE_OBJECT LowerDeviceObject;
};
NTSTATUS FilterAddDevice(PDRIVER_OBJECT DriverObject, PDEVICE_OBJECT PDO) {
	PDEVICE_OBJECT DeviceObject;
	auto status = IoCreateDevice(DriverObject, sizeof(DeviceExtension), nullptr,
			FILE_DEVICE_UNKNOWN, 0, FALSE, &DeviceObject);
	auto ext = (DeviceExtension*)DeviceObject->DeviceExtension;
	status = IoAttachDeviceToDeviceStackSafe(
			// device to attach
			DeviceObject,
			PDO,
			// target device
			&ext->LowerDeviceObject);
	// actual device object
	// copy some info from the attached device
	DeviceObject->DeviceType = ext->LowerDeviceObject->DeviceType;
	DeviceObject->Flags |= ext->LowerDeviceObject->Flags &
			(DO_BUFFERED_IO | DO_DIRECT_IO);
	// important for hardware-based devices
	DeviceObject->Flags &= ~DO_DEVICE_INITIALIZING;
	DeviceObject->Flags |= DO_POWER_PAGABLE;
	return status;
```

```
}
```

Несколько важных моментов для приведенного выше кода:

• Объект устройства создается без имени. Имя не требуется, потому что целевое устройство названо и является реальной целью для IRP, поэтому нет необходимости указывать собственное имя. Фильтр собирается быть вызванным независимо.

• В вызове IoCreateDevice мы указываем ненулевой размер для второго аргумента, запрашивая ввод-вывод.

До сих пор мы использовали глобальные переменные для управления состоянием устройства.

Однако драйвер фильтра может создавать несколько объектов устройств и подключаться к нескольким стекам устройств.

Механизм расширения устройства позволяет легко перейти к состоянию, зависящему от устройства, с учетом самого объекта устройства.

В приведенном выше коде мы фиксируем нижний объект устройства как свое состояние, но эту структуру можно расширить, включив дополнительную информацию по мере необходимости.

• Мы копируем некоторую информацию из нижнего объекта устройства, чтобы наш фильтр отображался для ввода/вывода как целевое устройство. В частности, мы копируем тип устройства и метод буферизации..

• Наконец, мы удаляем флаг DO_DEVICE_INITIALIZING (изначально установленный системой ввода-вывода), чтобы указать диспетчеру Plug & Play, что устройство готово к работе.

Учитывая приведенный выше код, вот реализация «вперед и забыть», в которой нижнее устройство используется как описано в предыдущем разделе:

```
NTSTATUS FilterGenericDispatch(PDEVICE_OBJECT DeviceObject, PIRP Irp) {
	auto ext = (DeviceExtension*)DeviceObject->DeviceExtension;
	IoSkipCurrentIrpStackLocation(Irp);
	return IoCallDriver(ext->LowerDeviceObject, Irp);
}
```

Присоединение фильтров в произвольное время

В предыдущем разделе было рассмотрено подключение фильтрующего устройства к функции обратного вызова AddDevice.

Для драйверов, не основанных на оборудовании, этот обратный вызов AddDevice никогда не вызывается.

Для этих более общих случаев драйвер фильтра может теоретически подключать фильтрующие устройства в любое время, создание объекта устройства (IoCreateDevice) с последующим использованием одной из функций «прикрепления».

Это означает целевое устройство уже существует, оно уже работает и в какой-то момент получает фильтр.

Драйверу необходимо убедиться, что это небольшое «прерывание» не оказывает отрицательного воздействия на целевое устройство.

Большинство операций, показанных в предыдущих разделах, также актуальны и здесь, например, копирование некоторых флагов с нижнего устройства. Однако следует проявлять особую осторожность, чтобы убедиться, что работа устройства не нарушена.

Используя IoAttachDevice, следующий код создает объект устройства и прикрепляет его поверх другого именованного объекта устройства (обработка ошибок опущена):

```
// use hard-coded name for illustration purposes

UNICODE_STRING targetName = RTL_CONSTANT_STRING(L"\\Device\\SomeDeviceName");

PDEVICE_OBJECT DeviceObject;

auto status = IoCreateDevice(DriverObject, 0, nullptr,

            FILE_DEVICE_UNKNOWN, 0, FALSE, &DeviceObject);

PDEVICE_OBJECT LowerDeviceObject;

status = IoAttachDevice(DeviceObject, &targetName, &LowerDeviceObject);

// copy information

DeviceObject->Flags |= LowerDeviceObject->Flags & (DO_BUFFERED_IO |
DO_DIRECT_IO);

DeviceObject->Flags &= ~DO_DEVICE_INITIALIZING;

DeviceObject->Flags |= DO_POWER_PAGABLE;

DeviceObject->DeviceType = LowerDeviceObject->DeviceType;
```

Очистка фильтра

После того, как фильтр прикреплен, его необходимо в какой-то момент отсоединить. Вызов IoDetachDevice с указателем объекта устройства выполняет эту операцию.

Обратите внимание, что аргументом является нижний объект устройства, а не собственный объект фильтра. Наконец, следует вызвать IoDeleteDevice для объекта фильтра, так же, как мы поступали со всеми нашими драйверами до сих пор.

Вопрос в том, когда следует вызывать этот код очистки ? Если драйвер выгружен явно, то эти операции очистки должны выполняться при обычной процедуре выгрузки.

Однако некоторые сложности возникает в фильтрах для аппаратных драйверов.

Эти драйверы, возможно, потребуется выгрузить из-за Plug & Play.

Ситуации, например, когда пользователь выдергивает устройство из системы. Драйверы на основе оборудования получают запрос IRP_MJ_PNP с второстепенным IRP IRP_MN_REMOVE_DEVICE, указывающий, что само оборудование выключено, поэтому весь узел устройства не нужен, и он будет снят.

Это ответственность драйвера для правильной обработки этого PnP-запроса, нужно отсоединиться от узла устройство и удалить устройство.

Это означает, что для аппаратных фильтров простой метод «вперед и забыть» для IRP_MJ_PNP не будет хватать.

Для IRP_MN_REMOVE_DEVICE требуется особая обработка. Вот пример кода:

```
NTSTATUS FilterDispatchPnp(PDEVICE_OBJECT fido, PIRP Irp) {
	auto ext = (DeviceExtension*)fido->DeviceExtension;
	auto stack = IoGetCurrentIrpStackLocation(Irp);
	UCHAR minor = stack->MinorFunction;
	IoSkipCurrentIrpStackLocation(Irp);
	auto status = IoCallDriver(ext->LowerDeviceObject, Irp);
	if (minor == IRP_MN_REMOVE_DEVICE) {
		IoDetachDevice(LowerDeviceObject);
		IoDeleteDevice(fido);
	}
	return status;
```

Подробнее об аппаратных драйверах фильтров

Фильтры для аппаратного драйвера имеют некоторые дополнительные сложности.

В FilterDispatchPnp, показанный в предыдущем разделе есть условие гонки. Проблема в том, что пока обрабатывается некоторая IRP, может поступить запрос на удаление устройства (например, обработанный на другом процессоре).

Это вызовет IoDeleteDevice, который являются частью узла устройства, пока фильтр готовится к отправке другова запроса вниз по стеку устройства. Более подробное объяснение этого состояния гонки выходит за рамки объема этой книги, но, тем не менее, нам нужно правильное решение.

Решение - это объект, предоставляемый системой ввода-вывода, который называется снятием блокировки, представленный структурой IO_REMOVE_LOCK.

По сути, эта структура управляет счетчиком ссылок количества невыполненных IRP, которые в настоящее время обрабатываются, и событие, о котором сигнализирует, когда счетчик ввода-вывода равен нулю и выполняется операция удаления.

Использование IO_REMOVE_LOCK можно регламентировать следующим образом:

1. Драйвер выделяет структуру как часть расширения устройства или глобальной переменной и инициализирует это один раз с IoInitializeRemoveLock.

2. Для каждого IRP драйвер получает блокировку удаления с помощью IoAcquireRemoveLock перед передачей это до нижнего устройства, если вызов завершился неудачно (STATUS_DELETE_PENDING), это означает удаление, операция выполняется, и драйвер должен немедленно вернуться.

3. После того, как драйвер более низкого уровня завершится с IRP, снимите блокировку удаления (IoReleaseRemoveLock).

4. При обработке IRP_MN_REMOVE_DEVICE вызовите IoReleaseRemoveLockAndWait перед отсоединением и удалением устройства. Вызов будет успешным, когда все остальные IRP перестанут обрабатываться.

Имея в виду эти шаги, общие запросы на передачу диспетчеризации должны быть изменены следующим образом (при условии, что снятие блокировки уже инициализировано):

```
struct DeviceExtension {
	IO_REMOVE_LOCK RemoveLock;
	PDEVICE_OBJECT LowerDeviceObject;
};
NTSTATUS FilterGenericDispatch(PDEVICE_OBJECT DeviceObject, PIRP Irp) {
	auto ext = (DeviceExtension*)DeviceObject->DeviceExtension;
```

```
	// second argument is unused in release builds
	auto status = IoAcquireRemoveLock(&ext->RemoveLock, Irp);
	if(!NT_SUCCESS(status)) {
		// STATUS_DELETE_PENDING
		Irp->IoStatus.Status = status;
		IoCompleteRequest(Irp, IO_NO_INCREMENT);
		return status;
	}
	IoSkipCurrentIrpStackLocation(Irp);
	status = IoCallDriver(ext->LowerDeviceObject, Irp);
	IoReleaseRemoveLock(&ext->RemoveLock, Irp);
	return status;
}
```

Чтобы правильно использовать снятие блокировки, необходимо изменить обработчик IRP_MJ_PNP:

```
NTSTATUS FilterDispatchPnp(PDEVICE_OBJECT fido, PIRP Irp) {
	auto ext = (DeviceExtension*)fido->DeviceExtension;
	auto status = IoAcquireRemoveLock(&ext->RemoveLock, Irp);
	if(!NT_SUCCESS(status)) {
		// STATUS_DELETE_PENDING
		Irp->IoStatus.Status = status;
		IoCompleteRequest(Irp, IO_NO_INCREMENT);
		return status;
	}
	auto stack = IoGetCurrentIrpStackLocation(Irp);
	UCHAR minor = stack->MinorFunction;
	IoSkipCurrentIrpStackLocation(Irp);
	auto status = IoCallDriver(ext->LowerDeviceObject, Irp);
	if (minor == IRP_MN_REMOVE_DEVICE) {
		// wait if needed
		IoReleaseRemoveLockAndWait(&ext->RemoveLock, Irp);
```

```
            IoDetachDevice(ext->LowerDeviceObject);
            IoDeleteDevice(fido);
      }
      else {
            IoReleaseRemoveLock(&ext->RemoveLock, Irp);
      }
      return status;
}
```

Монитор устройств

С помощью представленной информации можно создать универсальный драйвер, который может подключаться к объектам устройства в качестве фильтров для других устройств. Это позволяет перехватывать запросы к (почти) любому устройству.

Клиент сопутствующего пользовательского режима позволит добавлять и удалять устройства для фильтрации.

Мы создадим новый проект драйвера Empty WDM под названием KDevMon, как мы это делали много раз.

Драйвер должен иметь возможность подключаться к нескольким устройствам и, кроме того, предоставлять свой собственный Control Device Object (CDO) для обработки запросов конфигурации клиента пользовательского режима.

CDO будет создан в DriverEntry, как обычно, но вложения будут управляться отдельно, управляемыми запросами от клиента пользовательского режима.

Чтобы управлять всеми фильтруемыми устройствами, мы создадим вспомогательный класс DevMonManager. Его основная цель - добавлять и удалять устройства для фильтрации.

Каждому устройству будет представлена следующая структура:

```
struct MonitoredDevice {
      UNICODE_STRING DeviceName;
      PDEVICE_OBJECT DeviceObject;
      PDEVICE_OBJECT LowerDeviceObject;
};
```

Для каждого устройства нам нужно сохранить объект устройства фильтра (созданный этим драйвером), нижний объект устройства, к которому он прикреплен, и имя устройства.

Класс DevMonManager содержит фиксированный массив структур MonitoredDevice,

быстрый мьютекс для защиты массива и некоторые вспомогательные функции.

Вот определение класса DevMonManager:

```
const int MaxMonitoredDevices = 32;

class DevMonManager {

public:

	void Init(PDRIVER_OBJECT DriverObject);

	NTSTATUS AddDevice(PCWSTR name);

	int FindDevice(PCWSTR name);

	bool RemoveDevice(PCWSTR name);

	void RemoveAllDevices();

	MonitoredDevice& GetDevice(int index);

	PDEVICE_OBJECT CDO;

private:

	bool RemoveDevice(int index);

private:

	MonitoredDevice Devices[MaxMonitoredDevices];

	int MonitoredDeviceCount;

	FastMutex Lock;

	PDRIVER_OBJECT DriverObject;

};
```

Добавление устройства в фильтр

Самая интересная функция - DevMonManager :: AddDevice, которая выполняет прикрепление. Давайте сделаем это шаг за шагом.

NTSTATUS DevMonManager::AddDevice(PCWSTR name) {

Во-первых, мы должны получить мьютекс на случай, если выполняется более одной операции добавления/удаления/поиска. Затем мы можем быстро проверить, все ли слоты нашего массива заняты и что рассматриваемое устройство еще не фильтруется:

```
AutoLock locker(Lock);
if (MonitoredDeviceCount == MaxMonitoredDevices)
      return STATUS_TOO_MANY_NAMES;
if (FindDevice(name) >= 0)
      return STATUS_SUCCESS;
```

Теперь пора поискать свободный индекс массива, в котором мы можем хранить информацию о новом фильтре:

```
for (int i = 0; i < MaxMonitoredDevices; i++) {
      if (Devices[i].DeviceObject == nullptr) {
```

Свободный слот обозначается указателем объекта устройства, который равен NULL внутри структуры MonitoredDevice.

Далее, мы попытаемся получить указатель на объект устройства, который мы хотим отфильтровать, с помощью IoGetDeviceObjectPointer.

```
UNICODE_STRING targetName;
RtlInitUnicodeString(&targetName, name);
PFILE_OBJECT FileObject;
PDEVICE_OBJECT LowerDeviceObject = nullptr;
auto status = IoGetDeviceObjectPointer(&targetName, FILE_READ_DATA,
            &FileObject, &LowerDeviceObject);
if (!NT_SUCCESS(status)) {
      KdPrint(("Failed to get device object pointer (%ws) (0x%8X)\n", name,
status));
      return status;
}
```

Результатом IoGetDeviceObjectPointer на самом деле является самый верхний объект устройства, который не обязательно объект устройства, на который мы нацеливались.

Это нормально, так как любая операция присоединения в любом случае будет прикрепляться к вершине стека устройств.

Следующим шагом является создание нового объекта фильтрующего устройства и его инициализация, частично на основе объекта устройства.

В то же время нам нужно заполнить структуру MonitoredDevice. Для каждого созданного устройства мы хотим иметь расширение устройства, в котором хранится нижнее устройство.

Для этого мы определяем структуру расширения устройства, которая называется DeviceExtension, в файле DevMonManager.h:

struct DeviceExtension {

PDEVICE_OBJECT LowerDeviceObject;

};

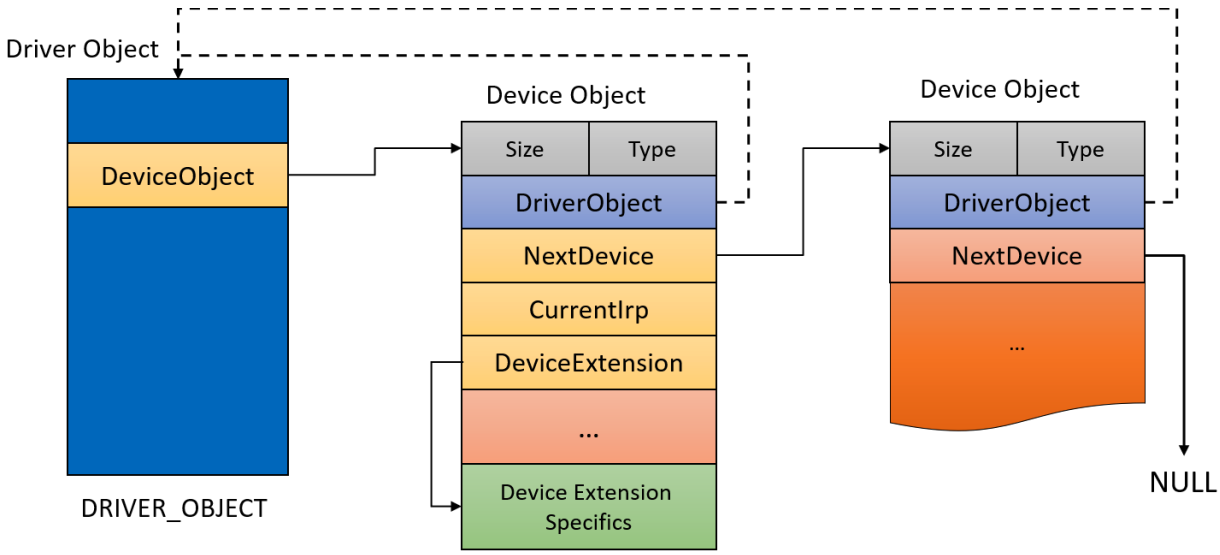
Вернемся к DevMonManager :: AddDevice - давайте создадим объект фильтрующего устройства:

```
PDEVICE_OBJECT DeviceObject = nullptr;

WCHAR* buffer = nullptr;

do {

      status = IoCreateDevice(DriverObject, sizeof(DeviceExtension), nullptr,

                  FILE_DEVICE_UNKNOWN, 0, FALSE, &DeviceObject);

      if (!NT_SUCCESS(status))

            break;
```

IoCreateDevice вызывается с размером структуры расширения устройства, которое будет выделено.

Сама структура DEVICE_OBJECT хранится в поле DeviceExtension поэтому она всегда доступна при необходимости.

На следующем рисунке показан эффект вызова IoCreateDevice.



Теперь мы можем продолжить инициализацию устройства и структуры MonitoredDevice:

```
// allocate buffer to copy device name

buffer = (WCHAR*)ExAllocatePoolWithTag(PagedPool, targetName.Length,
DRIVER_TAG);

if (!buffer) {

      status = STATUS_INSUFFICIENT_RESOURCES;

      break;

}

auto ext = (DeviceExtension*)DeviceObject->DeviceExtension;

DeviceObject->Flags |= LowerDeviceObject->Flags & (DO_BUFFERED_IO |
DO_DIRECT_IO);

DeviceObject->DeviceType = LowerDeviceObject->DeviceType;

Devices[i].DeviceName.Buffer = buffer;

Devices[i].DeviceName.MaximumLength = targetName.Length;

RtlCopyUnicodeString(&Devices[i].DeviceName, &targetName);

Devices[i].DeviceObject = DeviceObject;
```

На этом этапе новый объект устройства готов, осталось только прикрепить его и доделать еще инициализацию:

```
status = IoAttachDeviceToDeviceStackSafe(

            DeviceObject,

            // filter device object

            LowerDeviceObject,

            // target device object

            &ext->LowerDeviceObject);

// result

if (!NT_SUCCESS(status))

      break;

Devices[i].LowerDeviceObject = ext->LowerDeviceObject;

// hardware based devices require this

DeviceObject->Flags &= ~DO_DEVICE_INITIALIZING;

DeviceObject->Flags |= DO_POWER_PAGABLE;

MonitoredDeviceCount++;

} while (false);
```

Устройство будет подключено, и полученный указатель будет немедленно сохранен в расширении устройства. Это важно, поскольку сам процесс присоединения генерирует как минимум два IRP - IRP_MJ_CREATE и IRP_MJ_CLEANUP, поэтому драйвер должен быть подготовлен к их обработке.

Как мы скоро увидим, эта обработка требует, чтобы нижний объект устройства был доступен в расширении устройства.

Теперь осталось только навести порядок:

```
if (!NT_SUCCESS(status)) {
	if (buffer)
		ExFreePool(buffer);
	if (DeviceObject)
		IoDeleteDevice(DeviceObject);
	Devices[i].DeviceObject = nullptr;
}
if (LowerDeviceObject) {
	// dereference - not needed anymore
	ObDereferenceObject(FileObject);
}
return status;
}
}
// should never get here
NT_ASSERT(false);
return STATUS_UNSUCCESSFUL;
}
```

Удаление фильтрующего устройства

Удалить устройство из фильтрации довольно просто - в обратном порядке, что сделал AddDevice:

```
bool DevMonManager::RemoveDevice(PCWSTR name) {
	AutoLock locker(Lock);
	int index = FindDevice(name);
	if (index < 0)
		return false;
	return RemoveDevice(index);
}
bool DevMonManager::RemoveDevice(int index) {
	auto& device = Devices[index];
	if (device.DeviceObject == nullptr)
		return false;
	ExFreePool(device.DeviceName.Buffer);
	IoDetachDevice(device.LowerDeviceObject);
	IoDeleteDevice(device.DeviceObject);
	device.DeviceObject = nullptr;
	MonitoredDeviceCount--;
	return true;
}
```

Важными частями являются отсоединение устройства и его удаление.

FindDevice - простой помощник для поиска устройство по имени в массиве. Он возвращает индекс устройства в массиве или -1, если устройство не найдено:

```
int DevMonManager::FindDevice(PCWSTR name) {
	UNICODE_STRING uname;
	RtlInitUnicodeString(&uname, name);
	for (int i = 0; i < MaxMonitoredDevices; i++) {
		auto& device = Devices[i];
		if (device.DeviceObject &&
			RtlEqualUnicodeString(&device.DeviceName, &uname,
TRUE)) {
			return i;
		}
	}
```

```
	return -1;
}
```

Единственная уловка здесь - убедиться, что быстрый мьютекс получен перед вызовом этой функции.

Инициализация и выгрузка

Процедура DriverEntry довольно стандартна, она создает CDO, который позволяет добавлять и удалять фильтры. Однако есть некоторые отличия. В частности, драйвер должен поддерживать все основные функции устройства.

Поскольку драйвер теперь служит двойной цели: с одной стороны, он обеспечивает конфигурацию для добавления и удаления устройств при вызове CDO, а с другой стороны, основные коды функций будут вызываться самими клиентами фильтруемых устройств.

Мы запускаем DriverEntry, создав CDO и выставляя его через символическую ссылку, как мы видели много раз:

```
DevMonManager g_Data;

extern "C" NTSTATUS

DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING) {

	UNICODE_STRING devName = RTL_CONSTANT_STRING(L"\\Device\\KDevMon");

	PDEVICE_OBJECT DeviceObject;

	auto status = IoCreateDevice(DriverObject, 0, &devName,

			FILE_DEVICE_UNKNOWN, 0, TRUE, &DeviceObject);

	if (!NT_SUCCESS(status))

		return status;

	UNICODE_STRING linkName = RTL_CONSTANT_STRING(L"\\??\\KDevMon");

	status = IoCreateSymbolicLink(&linkName, &devName);

	if (!NT_SUCCESS(status)) {

		IoDeleteDevice(DeviceObject);

		return status;

	}

	DriverObject->DriverUnload = DevMonUnload;
```

В этом фрагменте кода нет ничего нового. Затем мы должны инициализировать все процедуры отправки, чтобы все основные функции поддерживались:

```
for (auto& func : DriverObject->MajorFunction)
	func = HandleFilterFunction;


/*
equivalent to:
for (int i = 0; i < ARRAYSIZE(DriverObject->MajorFunction); i++)
	DriverObject->MajorFunction[i] = HandleFilterFunction;
*/
```

Ранее в этой главе мы видели похожий код. В приведенном выше коде используется C ++ для изменения всех основные функции, указывающие на HandleFilterFunction, с которыми мы скоро познакомимся. Наконец, нам нужносохранить возвращенный объект устройства для удобства в глобальном объекте g_Data (DevMonManager) и инициализируем его:

g_Data.CDO = DeviceObject;

g_Data.Init(DriverObject);

return status;

}

Метод Init просто инициализирует быстрый мьютекс и сохраняет указатель объекта драйвера для дальнейшего использования с IoCreateDevice (который мы рассмотрели в предыдущем разделе).

Прежде чем мы углубимся в эту обычную процедуру отправки, давайте подробнее рассмотрим процедуру выгрузки.

Когда драйвер выгружен, нам нужно удалить символическую ссылку и CDO как обычно, но мы также должны отсоединить от всех активных в данный момент фильтров. Вот код:

```
void DevMonUnload(PDRIVER_OBJECT DriverObject) {
	UNREFERENCED_PARAMETER(DriverObject);
	UNICODE_STRING linkName = RTL_CONSTANT_STRING(L"\\??\\KDevMon");
	IoDeleteSymbolicLink(&linkName);
	NT_ASSERT(g_Data.CDO);
	IoDeleteDevice(g_Data.CDO);
	g_Data.RemoveAllDevices();
}
```

Ключевым моментом здесь является вызов DevMonManager :: RemoveAllDevices. Эта функция довольно проста, опираясь на DevMonManager :: RemoveDevice для своей работы:

```
void DevMonManager::RemoveAllDevices() {
	AutoLock locker(Lock);
	for (int i = 0; i < MaxMonitoredDevices; i++)
		RemoveDevice(i);
}
```

Обработка запросов

Процедура отправки HandleFilterFunction - самая важная часть драйвера.

Она будет вызывается для всех основных функций, предназначенных для одного из фильтрующих устройств или CDO.

Нужно это различать, и именно поэтому мы ранее сохранили указатель CDO. Наш CDO поддерживает: Создание, закрытие и DeviceIoControl.

Вот исходный код:

```
NTSTATUS HandleFilterFunction(PDEVICE_OBJECT DeviceObject, PIRP Irp) {
	if (DeviceObject == g_Data.CDO) {
		switch (IoGetCurrentIrpStackLocation(Irp)->MajorFunction) {
		case IRP_MJ_CREATE:
		case IRP_MJ_CLOSE:
			return CompleteRequest(Irp);
		case IRP_MJ_DEVICE_CONTROL:
			return DevMonDeviceControl(DeviceObject, Irp);
		}
		return CompleteRequest(Irp, STATUS_INVALID_DEVICE_REQUEST);
	}
```

Если целевым устройством является наш CDO, мы включаем саму основную функцию. Для создания и закрытия мы просто успешно завершаем IRP, вызвав вспомогательную функцию, с которой мы познакомились в главе 7:

```
NTSTATUS CompleteRequest(PIRP Irp,
            NTSTATUS status = STATUS_SUCCESS,
            ULONG_PTR information = 0);
NTSTATUS CompleteRequest(PIRP Irp, NTSTATUS status, ULONG_PTR information) {
      Irp->IoStatus.Status = status;
      Irp->IoStatus.Information = information;
      IoCompleteRequest(Irp, IO_NO_INCREMENT);
      return status;
}
```

Для IRP_MJ_DEVICE_CONTROL мы вызываем DevMonDeviceControl, который должен реализовывать наши управляющие коды для добавления и удаления фильтров. Для всех других основных функций мы просто заполняем IRP с ошибкой "неподдерживаемая операция".

Если объект устройства не является CDO, то это должен быть один из наших фильтров. Здесь драйвер может делать с запросом что угодно: регистрировать его, изучать, изменять - все, что угодно.

Для нашего драйвера мы просто отправьте отладчику вывод некоторой информации о запросе, а затем отправим его вниз к устройству под фильтром.

Сначала мы извлечем расширение для нашего устройства, чтобы получить доступ к нижнему устройству:

auto ext = (DeviceExtension*)DeviceObject→DeviceExtension;

Затем мы получим поток, отправивший запрос, углубившись в IRP, а затем получим поток и идентификаторы вызывающего:

```
auto thread = Irp->Tail.Overlay.Thread;
HANDLE tid = nullptr, pid = nullptr;
if (thread) {
      tid = PsGetThreadId(thread);
      pid = PsGetThreadProcessId(thread);
}
```

В большинстве случаев текущий поток совпадает с потоком, отправившим первоначальный запрос.Теперь пришло время вывести идентификаторы потока и процесса, а также тип запрошенной операции:

```
auto stack = IoGetCurrentIrpStackLocation(Irp);

DbgPrint("Intercepted driver: %wZ: PID: %d, TID: %d, MJ=%d (%s)\n",

            &ext->LowerDeviceObject->DriverObject->DriverName,

            HandleToUlong(pid), HandleToUlong(tid),

            stack->MajorFunction, MajorFunctionToString(stack-
>MajorFunction));
```

Вспомогательная функция MajorFunctionToString просто возвращает строковое представление . Например, для IRP_MJ_READ он возвращает «IRP_MJ_READ».

На этом этапе драйвер может продолжить изучение запроса. Если было получено IRP_MJ_DEVICE_CONTROL, он может просматривать управляющий код и входной буфер.

Если это IRP_MJ_WRITE, он может посмотреть на пользовательский буфер и так далее.

Наконец, поскольку мы не хотим нарушать работу целевого устройства, мы передадим запрос без изменений:

```
IoSkipCurrentIrpStackLocation(Irp);

return IoCallDriver(ext->LowerDeviceObject, Irp);

}
```

Упомянутая ранее функция DevMonDeviceControl является обработчиком драйвера для IRP_MJ_DEVICE_CONTROL. Это используется для динамического добавления или удаления устройств из фильтрации.

Коды управления следующие (в KdevMonCommon.h):

```
#define IOCTL_DEVMON_ADD_DEVICE \

            CTL_CODE(0x8000, 0x800, METHOD_BUFFERED, FILE_ANY_ACCESS)

#define IOCTL_DEVMON_REMOVE_DEVICE \

            CTL_CODE(0x8000, 0x801, METHOD_BUFFERED, FILE_ANY_ACCESS)

#define IOCTL_DEVMON_REMOVE_ALL \

            CTL_CODE(0x8000, 0x802, METHOD_NEITHER, FILE_ANY_ACCESS)
```

К настоящему времени код обработки можно довольно легко понять:

```
NTSTATUS DevMonDeviceControl(PDEVICE_OBJECT, PIRP Irp) {
    auto stack = IoGetCurrentIrpStackLocation(Irp);
    auto status = STATUS_INVALID_DEVICE_REQUEST;
    auto code = stack->Parameters.DeviceIoControl.IoControlCode;

    switch (code) {
        case IOCTL_DEVMON_ADD_DEVICE:
        case IOCTL_DEVMON_REMOVE_DEVICE:
        {
            auto buffer = (WCHAR*)Irp->AssociatedIrp.SystemBuffer;
            auto len = stack->Parameters.DeviceIoControl.InputBufferLength;
            if (buffer == nullptr || len < 2 || len > 512) {
                status = STATUS_INVALID_BUFFER_SIZE;
                break;
            }

            buffer[len / sizeof(WCHAR) - 1] = L'\0';
            if (code == IOCTL_DEVMON_ADD_DEVICE)
                status = g_Data.AddDevice(buffer);
            else {
                auto removed = g_Data.RemoveDevice(buffer);
                status = removed ? STATUS_SUCCESS : STATUS_NOT_FOUND;
            }
            break;
        }

        case IOCTL_DEVMON_REMOVE_ALL:
        {
            g_Data.RemoveAllDevices();
            status = STATUS_SUCCESS;
            break;
        }
    }

    return CompleteRequest(Irp, status);
}
```

Тестирование драйвера

Консольное приложение пользовательского режима снова довольно стандартно и принимает несколько команд для добавления и удаление устройств. Вот несколько примеров выполнения команд:

devmon add \device\procexp152

devmon remove \device\procexp152

devmon clear

Вот основная функция клиента пользовательского режима (очень небольшая обработка ошибок):

```
int wmain(int argc, const wchar_t* argv[]) {
	if (argc < 2)
		return Usage();
	auto& cmd = argv[1];
	HANDLE hDevice = ::CreateFile(L"\\\\.\\kdevmon", GENERIC_READ | GENERIC_WRITE,
		FILE_SHARE_READ, nullptr, OPEN_EXISTING, 0, nullptr);
	if (hDevice == INVALID_HANDLE_VALUE)
		return Error("Failed to open device");
	DWORD bytes;
	if (::_wcsicmp(cmd, L"add") == 0) {
		if (!::DeviceIoControl(hDevice, IOCTL_DEVMON_ADD_DEVICE, (PVOID)argv[2],
			static_cast<DWORD>(::wcslen(argv[2]) + 1) * sizeof(WCHAR), nullptr, 0,
			&bytes, nullptr))
			return Error("Failed in add device");
		printf("Add device %ws successful.\n", argv[2]);
		return 0;
	}
	else if (::_wcsicmp(cmd, L"remove") == 0) {
		if (!::DeviceIoControl(hDevice, IOCTL_DEVMON_REMOVE_DEVICE, (PVOID)argv[2],
			static_cast<DWORD>(::wcslen(argv[2]) + 1) * sizeof(WCHAR), nullptr, 0,
			&bytes, nullptr))
			return Error("Failed in remove device");
		printf("Remove device %ws successful.\n", argv[2]);
```

```
            return 0;
        }
        else if (::_wcsicmp(cmd, L"clear") == 0) {
            if (!::DeviceIoControl(hDevice, IOCTL_DEVMON_REMOVE_ALL,
                        nullptr, 0, nullptr, 0, &bytes, nullptr))
                return Error("Failed in remove all devices");
            printf("Removed all devices successful.\n");
        }
        else {
            printf("Unknown command.\n");
            return Usage();
        }
        return 0;
}
```

Мы уже много раз видели подобный код.

Драйвер можно установить так:

sc create devmon type= kernel binpath= c:\book\kdevmon.sys

И стартануть:

sc start devmon

В качестве первого примера мы запустим Process Explorer (он должен работать с повышенными правами, чтобы его драйвер мог устанавливается при необходимости) и фильтрровать поступающие к нему запросы:

devmon add \device\procexp152

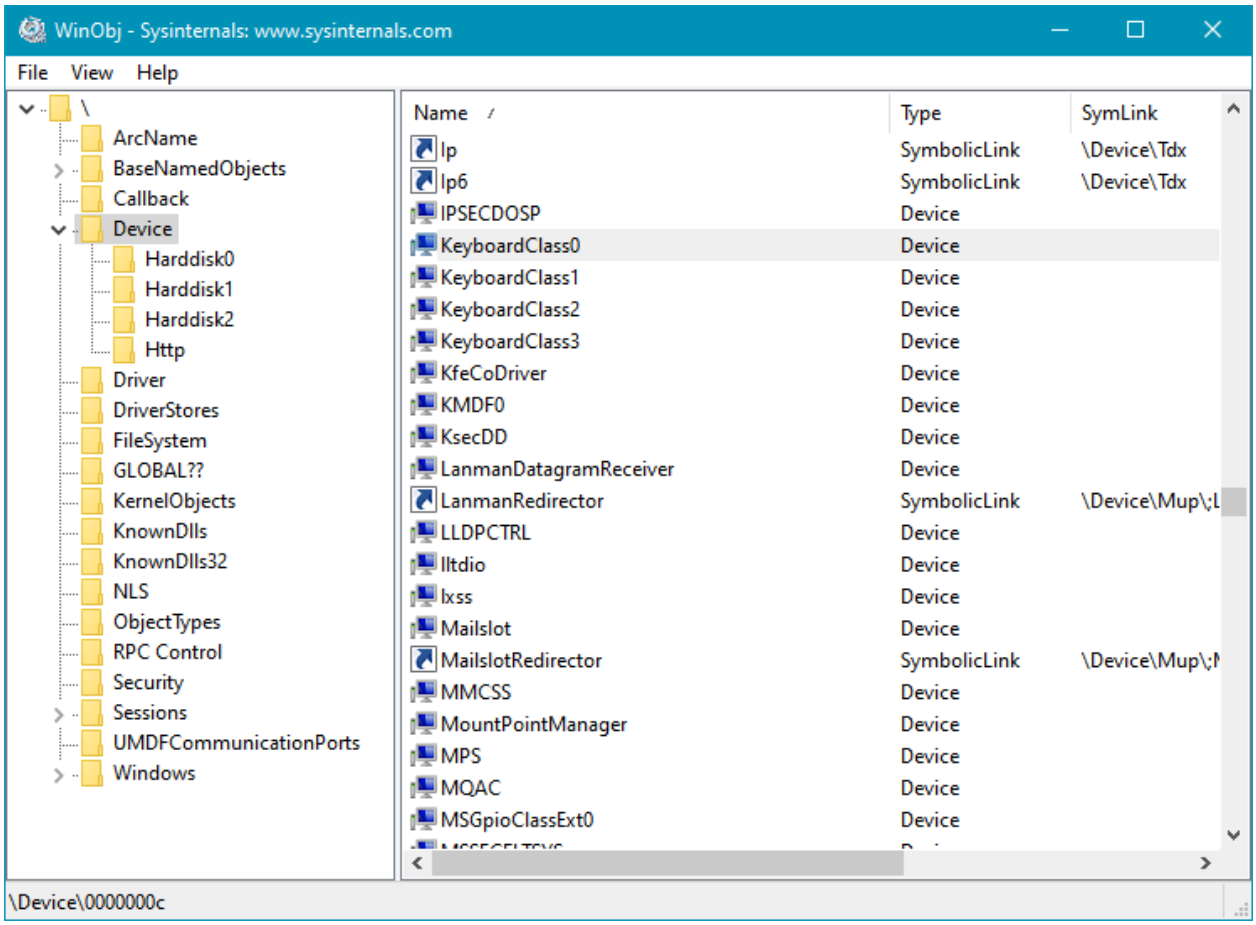
Помните, что WinObj показывает устройство с именем ProcExp152 в каталоге Device объекта. Мы можем запустить DbgView из SysInternals с повышенными правами и настроить его для ведения логов.

Вот пример вывода:

```
1 0.00000000 driver: \Driver\PROCEXP152: PID: 5432, TID: 8820, MJ=14 (IRP_MJ_DEVICE_\
CONTROL)
2 0.00016690 driver: \Driver\PROCEXP152: PID: 5432, TID: 8820, MJ=14 (IRP_MJ_DEVICE_\
CONTROL)
3 0.00041660 driver: \Driver\PROCEXP152: PID: 5432, TID: 8820, MJ=14 (IRP_MJ_DEVICE_\
CONTROL)
4 0.00058020 driver: \Driver\PROCEXP152: PID: 5432, TID: 8820, MJ=14 (IRP_MJ_DEVICE_\
CONTROL)
5 0.00071720 driver: \Driver\PROCEXP152: PID: 5432, TID: 8820, MJ=14 (IRP_MJ_DEVICE_\
CONTROL)
```

Неудивительно, что идентификатор процесса Process Explorer на этом компьютере равен 5432 (и у него есть поток с ID 8820). Очевидно, что Process Explorer своевременно отправляет своим драйверам запросы, и это всегда IRP_MJ_DEVICE_CONTROL.

Устройства, которые мы можем фильтровать, можно просмотреть с помощью WinObj, в основном в каталоге устройств, показанном на рисунке:



Давайте отфильтруем keyboardclass0, которым управляет драйвер класса клавиатуры:

devmon add \device\keyboardclass0

Теперь нажмите несколько клавиш. Вы увидите, что для каждой нажатой клавиши вы получаете строку вывода. Вот как это выглядит:

```
1 11:31:18 driver: \Driver\kbdclass: PID: 612, TID: 740, MJ=3 (IRP_MJ_READ)
2 11:31:18 driver: \Driver\kbdclass: PID: 612, TID: 740, MJ=3 (IRP_MJ_READ)
3 11:31:19 driver: \Driver\kbdclass: PID: 612, TID: 740, MJ=3 (IRP_MJ_READ)
4 11:31:19 driver: \Driver\kbdclass: PID: 612, TID: 740, MJ=3 (IRP_MJ_READ)
5 11:31:20 driver: \Driver\kbdclass: PID: 612, TID: 740, MJ=3 (IRP_MJ_READ)
6 11:31:20 driver: \Driver\kbdclass: PID: 612, TID: 740, MJ=3 (IRP_MJ_READ)
```

Что это за процесс 612? Это экземпляр CSrss.exe, запущенный в пользовательском сеансе.

В его обязанности входит получение данных с устройств ввода. Обратите внимание, что это операция чтения, что означает, что ожидается некоторый буфер от драйвера класса клавиатуры. Но как его получить? Мы вернемся к этому в следующем разделе.

Вы можете попробовать другие устройства. Некоторые могут не подключиться (обычно те, которые открыты для эксклюзивного доступа), а некоторые из них не подходят для такой фильтрации, особенно драйверы файловой системы.

Вот пример с устройством с несколькими поставщиками UNC (MUP):

devmon add \device\mup

Перейдите в какую-нибудь сетевую папку, и вы увидите много активности, как показано здесь:

```
001 11:46:19 driver: \FileSystem\FltMgr: PID: 4, TID: 6236, MJ=2 (IRP_MJ_CLOSE)
002 11:46:25 driver: \FileSystem\FltMgr: PID: 7212, TID: 5600, MJ=0 (IRP_MJ_CREATE)
003 11:46:25 driver: \FileSystem\FltMgr: PID: 7212, TID: 5600, MJ=13 (IRP_MJ_FILE_SY\
STEM_CONTROL)
004 11:46:25 driver: \FileSystem\FltMgr: PID: 7212, TID: 5600, MJ=18 (IRP_MJ_CLEANUP\
)
005 11:46:25 driver: \FileSystem\FltMgr: PID: 7212, TID: 5600, MJ=2 (IRP_MJ_CLOSE)
006 11:47:00 driver: \FileSystem\FltMgr: PID: 7212, TID: 4464, MJ=0 (IRP_MJ_CREATE)
007 11:47:00 driver: \FileSystem\FltMgr: PID: 7212, TID: 4464, MJ=13 (IRP_MJ_FILE_SY\
STEM_CONTROL)
...
054 11:47:25 driver: \FileSystem\FltMgr: PID: 7212, TID: 8272, MJ=13 (IRP_MJ_FILE_SY\
STEM_CONTROL)
055 11:47:25 driver: \FileSystem\FltMgr: PID: 7212, TID: 8272, MJ=18 (IRP_MJ_CLEANUP\
)
```

Обратите внимание, что наслоение расположено поверх Менеджера фильтров, с которым мы познакомились в главе 10. Также обратите внимание, что задействовано несколько процессов (оба экземпляра Explorer.exe). Устройство MUP - это том для удаленной файловой системы. Этот тип устройств лучше всего фильтровать с помощью мини-фильтра файловой системы.

Результаты запросов

Общий обработчик диспетчеризации, который у нас есть для драйвера DevMon, видит только входящие запросы.

Остается интересный вопрос - как мы можем получить результаты запроса?

Какой-то драйвер в стеке устройств вызовет IoCompleteRequest. Если драйвер заинтересован в результате он должен настроить процедуру завершения ввода-вывода.

Как обсуждалось в главе 7, процедуры завершения вызываются в обратном порядке регистрации, когда вызывается IoCompleteRequest.

Каждый уровень в стеке устройств (кроме самого нижнего) может настраивать процедура завершения, которая будет вызываться как часть завершения запроса. В это время драйвер может посмотреть статус IRP, проверить выходные буферы и т. д.

Настройка процедуры завершения выполняется с помощью IoSetCompletionRoutine или (лучше) IoSetCompletionRoutineEx. Вот прототип последнего:

```
NTSTATUS IoSetCompletionRoutineEx (
            _In_ PDEVICE_OBJECT DeviceObject,
            _In_ PIRP Irp,
            _In_ PIO_COMPLETION_ROUTINE CompletionRoutine,
            _In_opt_ PVOID Context,
            _In_ BOOLEAN InvokeOnSuccess,
            _In_ BOOLEAN InvokeOnError,
            _In_ BOOLEAN InvokeOnCancel);
```

Большинство параметров говорят сами за себя. Последние три параметра указывают, для какого IRP статус завершения:

• Если InvokeOnSuccess имеет значение TRUE, процедура завершения вызывается, если статус IRP является NT_SUCCESS.

• Если InvokeOnError имеет значение TRUE, процедура завершения вызывается, если статус IRP не соответствует NT_SUCCESS..

• Если InvokeOnCancel имеет значение TRUE, процедура завершения вызывается, если статус IRP - STATUS_CANCELED, что означает, что запрос был отменен.

Сама процедура завершения должна иметь следующий прототип:

```
NTSTATUS CompletionRoutine (
            _In_ PDEVICE_OBJECT DeviceObject,
            _In_ PIRP Irp,
            _In_opt_ PVOID Context);
```

Процедура завершения вызывается произвольным потоком (тем, который вызвал IoCompleteRequest) в IRQL <= DISPATCH_LEVEL (2). Это означает, что необходимо соблюдать все правила из 6 главы для IRQL 2.

Driver Hooking

Использование драйверов фильтров, описанных в этой главе и в главе 10, обеспечивает большую гибкость для драйверов.

Разработчик может перехватывать запросы практически к любому устройству. В этом разделе я хотел бы упомянуть и другую технику, хотя она и не «официальная», но может быть весьма полезной в определенных случаях.

Этот метод перехвата драйверов основан на идее замены указателей подпрограммы диспетчеризации запущенных драйверов. Это автоматически обеспечивает «фильтрацию» для всех устройств, управляемых этим драйвером.

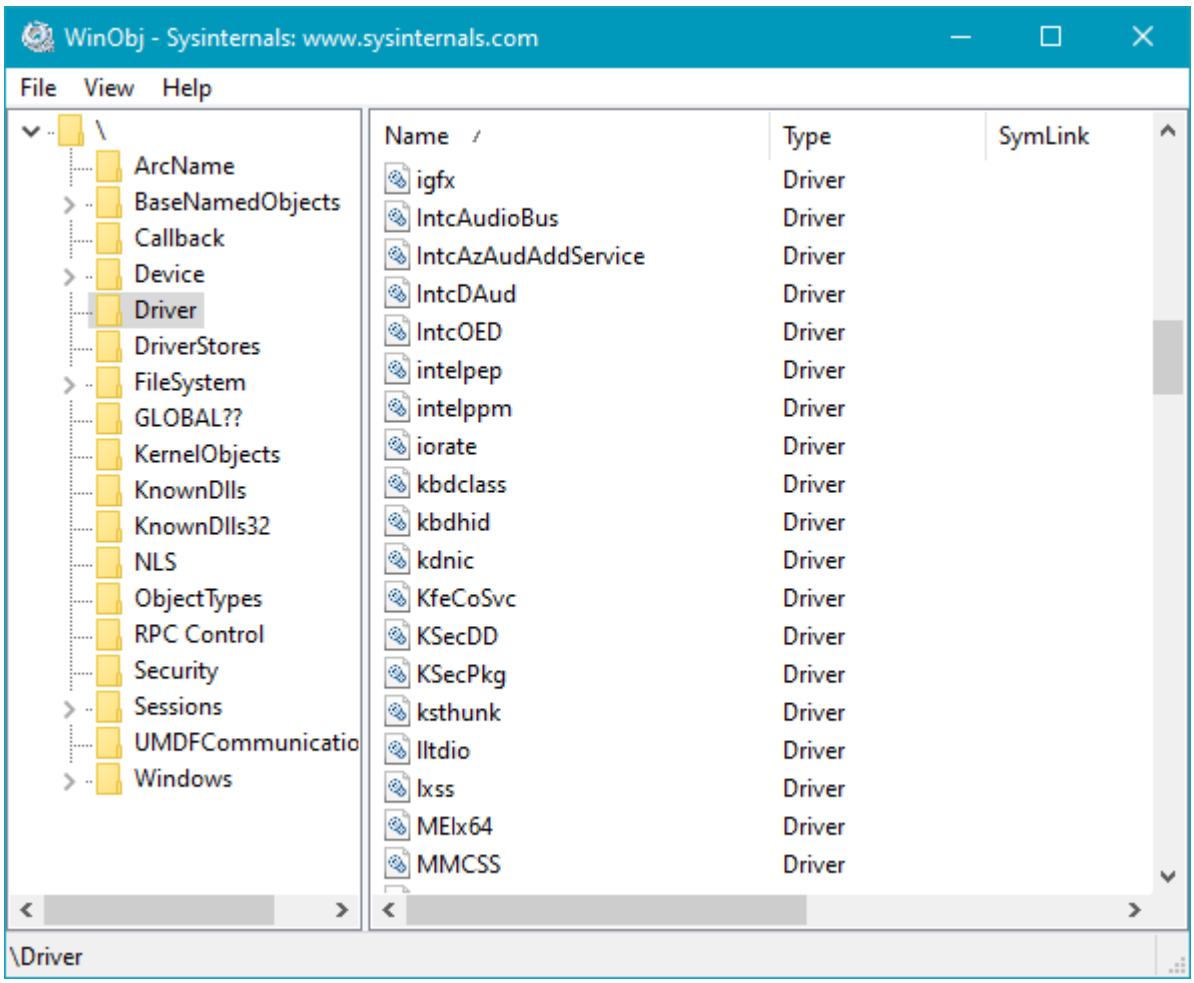
Хук драйвера сохранит старые указатели на функции, а затем заменит основной массив функций в драйвере своими функциями. Теперь любой запрос, поступающий на устройство под управлением подключенного драйвера вызовет процедуры диспетчеризации подключающегося драйвера.

ВАЖНО:

В современной Windows нет возможности сделать это, т. к. существует защита ядра PatchGuard (also known as Kernel Patch Protection), данная манипуляция приведет к краху системы.

У драйверов есть имена, и поэтому они являются частью пространства имен диспетчера объектов.

Это можно посмотреть в WinObj на следующем рисунке:



Чтобы сделать хук на драйвер, нам нужно найти указатель объекта драйвера (DRIVER_OBJECT), и для этого мы может использовать недокументированную, но экспортированную функцию, которая может найти любой объект по его имени:

```
NTSTATUS ObReferenceObjectByName (
            _In_ PUNICODE_STRING ObjectPath,
            _In_ ULONG Attributes,
            _In_opt_ PACCESS_STATE PassedAccessState,
            _In_opt_ ACCESS_MASK DesiredAccess,
            _In_ POBJECT_TYPE ObjectType,
            _In_ KPROCESSOR_MODE AccessMode,
            _Inout_opt_ PVOID ParseContext,
            _Out_ PVOID *Object);
```

Вот пример вызова ObReferenceObjectByName для поиска драйвера kbdclass:

```
UNICODE_STRING name;

RtlInitUnicodeString(&name, L"\\driver\\kbdclass");

PDRIVER_OBJECT driver;

auto status = ObReferenceObjectByName(&name, OBJ_CASE_INSENSITIVE,

            nullptr, 0, *IoDriverObjectType, KernelMode,

            nullptr, (PVOID*)&driver);

if(NT_SUCCESS(status)) {

      // manipulate driver

      ObDereferenceObject(driver);

      // eventually

}
```

Хук на драйвер теперь может заменять указатели основных функций, процедуру выгрузки, добавления устройства и т.д.

Любая такая замена всегда должна сохранять указатели на предыдущие функции для отсоединения при желании и для пересылки запроса реальному драйверу. Поскольку эта замена должна быть произведена атомарно, лучше всего использовать InterlockedExchangePointer для атомарного обмена.

Следующий фрагмент кода демонстрирует эту технику:

```
for (int j = 0; j <= IRP_MJ_MAXIMUM_FUNCTION; j++) {

      InterlockedExchangePointer((PVOID*)&driver->MajorFunction[j],
MyHookDispatch);

}

InterlockedExchangePointer((PVOID*)&driver->DriverUnload, MyHookUnload);
```

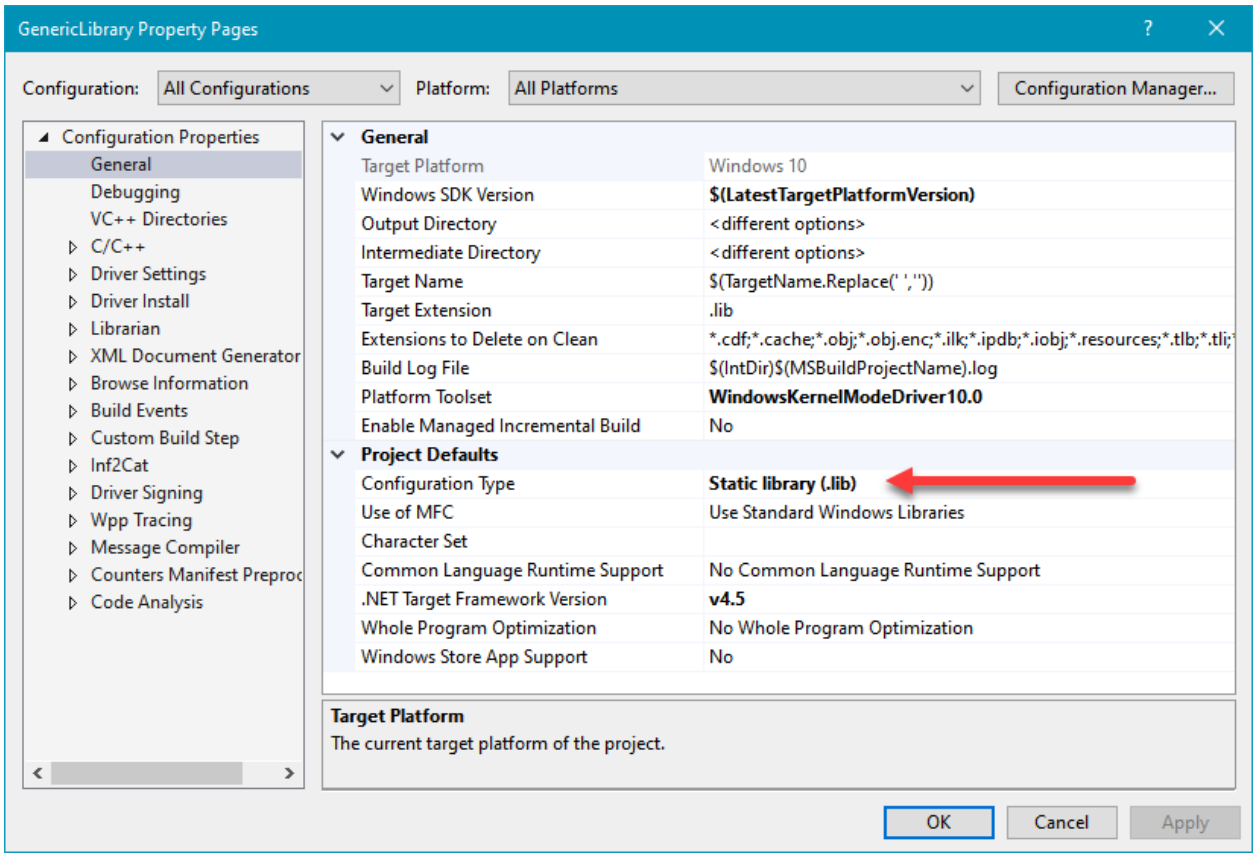
Довольно полный пример этой техники перехвата можно найти в проекте DriverMon на Github по адресу:`https://github.com/zodiacon/DriverMon`

Библиотеки ядра

В процессе написания драйверов мы разработали несколько классов и вспомогательных функций, которые можно использовать в своих драйверах.

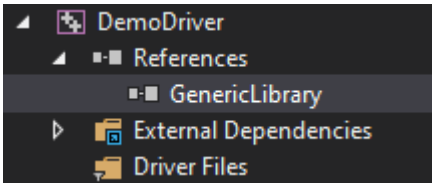
Однако имеет смысл упаковать их в единую библиотеку, чтобы мы могли подключить её, вместо копирования исходных файлов из проекта в проект.

Шаблоны проектов, поставляемые с WDK, явно не предоставляют возможность создать статической библиотеки для драйверов, но сделать его довольно просто. Для этого нужно создать обычный проект драйвера (на основе WDM Empty Driver, например), а затем просто измените тип проекта на статическую библиотеку, как показано на рисунке:



Проект драйвера, который нужно связать с этой библиотекой, просто должен добавить ссылку в Visual Studio, щелкнув правой кнопкой мыши узел «Ссылки» в обозревателе решений, выбрав «Добавить ссылку ...» и выбрав библиотечный проект.

На рисунке ниже показан узел ссылок драйвера после добавления библиотеки:



Резюме

<u>В этой книге мы не рассмотрели ОЧЕНЬ много вопросов, так как мир драйверов ядра очень велик.</u>

Считайте эту книгу введением в мир драйверов устройств ядра.

Некоторых тем у нас не было.

В этой книге не рассматривалось:

• Аппаратные драйверы устройств.

• Сетевые драйверы и фильтры.

• Платформа фильтрации Windows (WFP).

• Дополнительные темы о мини-фильтрах файловой системы

• Другие общие методы разработки: вспомогательные списки, растровые изображения, деревья AVL.

• Конкретные типы драйверов, связанные с технологией: устройство интерфейса пользователя (HID), дисплей, аудио, изображения, bluetooth, хранилище, ...

•Некоторые из этих тем являются хорошими кандидатами для будущей «продвинутой» книги.

Microsoft задокументировала все вышеупомянутые типы драйверов, а официальные образцы есть на Github: https://github.com/Microsoft/Windows-driver-samples , которые регулярно обновляются. Это должно быть ваше первое место для поиска дополнительной информации.

Мы подошли к концу этой книги. Желаю вам удачного программирования драйверов ядра !