Глава 4. Драйвер и взаимодействие с ним из пользовательского режима

Тема на форуме: https://ru-sfera.org/threads/windows-kernel-programming-glava-4-drajver-i-vzaimodejstvie-s-nim-iz-polzovatelskogo-rezhima.3953/

В этой главе мы будем использовать многие концепции, которые мы изучили в предыдущих главах, и создадим простой драйвер.

Также в этой главе создадим клиент, для взаимодействия с нашим драйвером.

Мы установим драйвер и выполним в режиме ядра некоторую операцию, недоступную пользовательский режиме.

В этой главе:

Введение.

Инициализация драйвера.

Код клиента.

Создание и закрытие процедур отправки.

Программа отправки DeviceIoControl

Установка и тестирование.

1)Введение

Проблема, которую мы решим с помощью простого драйвера ядра, заключается в негибкости установки приоритетов потоков используя Windows API.

В пользовательском режиме приоритет потока определяется комбинацией его класса приоритета процесса со смещением для каждого потока, которое имеет ограниченное количество уровней.

Изменение класса приоритета процесса может быть достигнуто с помощью функции SetPriorityClass, которая принимает дескриптор процесса и один из шести поддерживаемых классов приоритетов.

Каждый класс приоритета соответствует уровню приоритета, который является приоритетом по умолчанию для потоков, созданных в этом процессе.

Определенный приоритет потока можно изменить с помощью функции SetThreadPriority, принимая дескриптор потока и один из нескольких констант, соответствующих смещениям вокруг базового класса приоритета.

Таблица 4-1 показывает доступные приоритеты потока на основе класса приоритета процесса и смещения приоритета потока.

Priority Class	- Sat	-2	-1	0 (default)	+1	+2	+ Sat	Comments
Idle (Low)	1	2	3	4	5	6	15	
Below Normal	1	4	5	6	7	8	15	
Normal	1	6	7	8	9	10	15	
Above Normal	1	8	9	10	11	12	15	
High	1	11	12	13	14	15	15	Only six levels are available (not seven).
Real-time	16	22	23	24	25	26	31	All levels between 16 to 31 can be selected.

Значения, приемлемые для SetThreadPriority, определяют смещение.

Пять уровней соответствуют смещениям от -2 до +2:
THREAD_PRIORITY_LOWEST (-2), THREAD_PRIORITY_BELOW_NORMAL (-1),

THREAD_PRIOR-ITY_NORMAL (0), THREAD_PRIORITY_ABOVE_NORMAL (+1), THREAD_PRIORITY_HIGHEST (+2).

оставшиеся два уровня, называемые уровнями насыщенности, устанавливают приоритет двум крайностям, поддерживаемым этот класс приоритета: THREAD_PRIORITY_IDLE (-Sat) и THREAD_PRIORITY_TIME_CRITICAL (+ Sat).

В следующем примере кода изменяется приоритет текущего потока на 11:

SetPriorityClass(GetCurrentProcess(), ABOVE_NORMAL_PRIORITY_CLASS);

SetThreadPriority(GetCurrentThread(),THREAD_PRIORITY_ABOVE_NORMAL);

Класс приоритета в реальном времени не подразумевает, что Windows является ОС реального времени.

Windows не предоставляют некоторые временные гарантии, обычно предоставляемые в режиме реального времени.

Кроме того, поскольку приоритеты в реальном времени очень высоки и конкурируют со многими потоками ядра, выполняющих важную работу, такой процесс должен выполняться с правами администратора.

В противном случае попытка установить класс приоритета в режиме реального времени приводит к значению High.

Существуют и другие различия между приоритетами реального времени и классами с более низким приоритетом.

Обратитесь к книге «Windows Internals» для получения дополнительной информации.

Таблица 4-1 показывает проблему, которую мы решаем достаточно четко. Доступен только небольшой набор приоритетов который можно установить напрямую.

Мы хотели бы создать драйвер, который обойдет эти ограничения и позволит установить приоритет потока на любое число, независимо от его класса приоритета процесса.

2)Инициализация драйвера

Мы начнем создавать драйвер так же, как в главе 2.

Создайте новый «WDM Empty».

Проект с именем PriorityBooster (или другим именем по вашему выбору) и удалите созданный INF-файл.

Затем добавьте в проект новый исходный файл с именем PriorityBooster.cpp. Добавьте основной #include для основного заголовка WDK и пустой DriverEntry:

```
#include <ntddk.h>

extern "C" NTSTATUS

DriverEntry(_In_ PDRIVER_OBJECT DriverObject, _In_ PUNICODE_STRING RegistryPath) {

return STATUS_SUCCESS;

}
```

Большинство драйверов программного обеспечения должны сделать следующее в DriverEntry:

- *Установить процедуру выгрузки.*
- *Установить процедуры отправки, поддерживаемые драйвером.*
- *Создайте объект устройства.*
- *Создайте символическую ссылку на объект устройства.*

Как только все эти операции выполнены, драйвер готов к приему запросов.

Первый шаг - добавить подпрограмму Unload и указать на нее из объекта драйвера. Вот новый DriverEntry с подпрограммой выгрузки:

```
// prototypes
void PriorityBoosterUnload(_In_ PDRIVER_OBJECT DriverObject);
// DriverEntry
extern "C" NTSTATUS
DriverEntry(_In_ PDRIVER_OBJECT DriverObject, _In_ PUNICODE_STRING RegistryPath) {
DriverObject->DriverUnload = PriorityBoosterUnload;
return STATUS_SUCCESS;
}
void PriorityBoosterUnload(_In_ PDRIVER_OBJECT DriverObject) {
}
```

Мы добавим код в процедуру Unload по мере необходимости, когда будем выполнять реальную работу в DriverEntry, которая требует быть отмененным.

Далее нам нужно настроить процедуры отправки, которые мы хотим поддерживать.

Практически все драйверы должны поддерживать IRP_MJ_CREATE и IRP_MJ_CLOSE, иначе не было бы способа открыть дескриптор любого устройства для этого драйвера.

Поэтому мы добавляем в DriverEntry следующее:

```
DriverObject->MajorFunction[IRP_MJ_CREATE] = PriorityBoosterCreateClose;
DriverObject->MajorFunction[IRP_MJ_CLOSE] = PriorityBoosterCreateClose;
```

Мы указываем основные функции «Создать» и «Закрыть» на одну и ту же процедуру. Это потому, что, в нашем примере, они фактически делают то же самое: просто одобрят запрос.

В более сложных случаях это могут быть отдельные функции, где в случае Create драйвер может (например) проверить, чтобы увидеть кто является вызывающим абонентом, и только разрешенным абонентам давать возможность открыть устройство.

Все основные функции имеют один и тот же прототип (они являются частью массива указателей функций), поэтому мы должны добавить прототип для PriorityBoosterCreateClose.

Прототип для этих функций следующей:

NTSTATUS PriorityBoosterCreateClose(_In_ PDEVICE_OBJECT DeviceObject, _In_ PIRP Irp);

Функция должна возвращать NTSTATUS и принимает указатель на объект устройства и указатель на пакет запроса ввод/вывод (IRP).

IRP является основным объектом, где хранится информация запроса, для всех типов запросов.

Мы углубимся в IRP в главе 6.

3)Передача информации драйверу

Настроенные нами операции «Создать» и «Закрыть» обязательны, но, безусловно, недостаточны.

Нам нужен способ сообщить драйверу какой поток и какому значению установить его приоритет.

Из пользовательского режима клиента, есть три основные функции, которые он может использовать: WriteFile, ReadFile и DeviceIoControl.

В целях нашего драйвера мы можем использовать либо WriteFile, либо DeviceIoControl.

В чтении смысла нет, потому что мы передаем информацию драйверу, а не от драйвера.

Так что -же лучше, WriteFile или DeviceIoControl ?

Это в основном дело вкуса, но обычно используется Write, если это действительно операция записи (логически) для всего остального —

DeviceIoControl, так как это общий механизм для передачи данных в драйвер и из него.

Поскольку изменение приоритета потока не является чисто операцией записи, мы будем использовать DeviceIoControl.

Эта функция имеет следующий прототип:

BOOL WINAPI DeviceIoControl(

_In_ HANDLE hDevice,

_In_ DWORD dwIoControlCode,

_In_reads_bytes_opt_(nInBufferSize) LPVOID lpInBuffer,

_In_ DWORD nInBufferSize,

*_Out_writes_bytes_to_opt_(nOutBufferSize,*lpBytesReturned) LPVOID lpOutBuffer,*

_In_ DWORD nOutBufferSize,

_Out_opt_ LPDWORD lpBytesReturned,

_Inout_opt_ LPOVERLAPPED lpOverlapped);

У DeviceIoControl есть три важных элемента:

• Управляющий код.

• Входной буфер.

• Выходной буфер.

Это означает, что DeviceIoControl - это гибкий способ связи с драйвером.

На стороне драйвера DeviceIoControl соответствует основной функции IRP_MJ_DEVICE_CONTROL.

Давайте добавим это к нашей инициализации процедур отправки:

DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = PriorityBoosterDeviceControl;

4)Протокол коммуникации Client/Driver

Учитывая, что мы решили использовать DeviceIoControl для связи клиент/драйвер, теперь мы должны определить фактическую семантику. Понятно, что нам нужен управляющий код и входной буфер.

Этот буфер должен содержать две части информации, необходимые для того, чтобы драйвер мог делать свое дело: идентификатор потока и приоритет для этого потока.

Эти фрагменты информации должны использоваться как драйвером, так и клиентом.

Клиент будет поставлять данные, и драйвер будет действовать на него.

Это означает, что эти определения должны быть в отдельном файле, который должен быть включен в драйвер и код клиента.

Для этого мы добавим файл заголовка PriorityBoosterCommon.h в проект драйвера.

Этот файл также будет использоваться позже клиентом пользовательского режима.

В этом файле нам нужно определить две вещи: структуру данных, которую драйвер ожидает от клиентов, и управляющий код для изменения приоритета потока.

Давайте начнем с объявления структуры, которая захватывает информацию, необходимую драйверу для клиента:

```
struct ThreadData {
ULONG ThreadId;
int Priority;
};
```

Нам нужен уникальный идентификатор потока и приоритет.

Идентификаторы потоков - это 32-разрядные целые числа без знака, поэтому

мы выбираем ULONG в качестве типа (Обратите внимание, что мы не можем обычно использовать DWORD - общий тип, определенный в заголовке режима пользователя - потому что он не определен в заголовках режима ядра. ULONG, с другой стороны, это определяется в обоих).

Приоритет должен быть числом от 1 до 31, поэтому простое 32-разрядное целое число.

Далее нам нужно определить управляющий код. Вы можете подумать, что подойдет любое 32-битное число, но это не тот случай.

Управляющий код должен быть построен с использованием макроса CTL_CODE, который принимает четыре аргумента, которые составляют окончательный контрольный код.

CTL_CODE определяется так:

*#define CTL_CODE(DeviceType, Function, Method, Access) (*

((DeviceType) << 16) | ((Access) << 14) | ((Function) << 2) | (Method))

Вот краткое описание значения этих макро аргументов:

• DeviceType - определяет тип устройства. Это может быть одна из констант FILE_DEVICE_xxx, которая определяется в заголовках WDK, но это в основном для аппаратных драйверов.

Для драйверов программного обеспечения как и у нас, число не имеет большого значения.

Тем не менее, в документации Microsoft указано, что значения для третьих лиц должны начинаться с 0x8000.

• Функция - возрастающий номер, обозначающий конкретную операцию. Если нет ничего другого, этот номер должен отличаться для разных управляющих кодов для одного и того же драйвера.

Опять же, можно любое число, но официальная документация гласит, что сторонние драйверы должны начинаться с 0x800.

• Метод - самая важная часть контрольного кода. Указывает, как предоставленные клиентом буферы передаются драйверу. Мы рассмотрим эти значения подробно в главе 6. Для нашего драйвера мы будем

использовать простейшее значение METHOD_NEITHER. Мы увидим его эффект позже.

• Доступ - указывает, относится ли эта операция к драйверу (FILE_WRITE_ACCESS), или драйверу (FILE_READ_ACCESS) или оба доступа (FILE_ANY_ACCESS).

Типичные драйверы просто используют FILE_ANY_ACCESS и обрабатывают фактический запрос в обработчике IRP_MJ_DEVICE_CONTROL.

Учитывая приведенную выше информацию, мы можем определить наш единственный контрольный код следующим образом:

```
#define PRIORITY_BOOSTER_DEVICE 0x8000
#define IOCTL_PRIORITY_BOOSTER_SET_PRIORITY CTL_CODE(PRIORITY_BOOSTER_DEVICE, \
0x800, METHOD_NEITHER, FILE_ANY_ACCESS)
```

5)Создание объекта устройства

В настоящее время у нас нет объектов устройства и поэтому нет возможности открыть handle и добраться до драйвера.

Типичному программному драйверу нужен только один объект устройства с символической ссылкой, указывающей на него, чтобы клиенты пользовательского режима могли получить дескрипторы.

Создание объекта устройства требует вызова API IoCreateDevice, объявленного следующим образом (некоторые SAL аннотации опущены/упрощены для ясности):

```
NTSTATUS IoCreateDevice(
_In_
PDRIVER_OBJECT DriverObject,
_In_
ULONG DeviceExtensionSize,
_In_opt_
PUNICODE_STRING DeviceName,
_In_
DEVICE_TYPE DeviceType,
_In_
```

ULONG DeviceCharacteristics,

*_In_*

BOOLEAN Exclusive,

*_Outptr_*

*PDEVICE_OBJECT *DeviceObject);*

Аргументы IoCreateDevice описаны ниже:

• DriverObject - объект драйвера, к которому принадлежит этот объект устройства. Это должно быть просто, объект драйвера передается в функцию DriverEntry.

• DeviceExtensionSize - дополнительные байты, которые будут выделены в дополнение к sizeof (DEVICE_OBJECT). Полезно для связи некоторой структуры данных с устройством. Это менее полезно для программных драйверов, создающие только один объект устройства, так как состояние, необходимое для устройства, может просто управляться глобальными переменными.

• DeviceName - имя внутреннего устройства, обычно создаваемое в диспетчере объектов устройства.

• DeviceType - относится к некоторым типам аппаратных драйверов. Для драйверов программного обеспечения должно использоваться значение FILE_DEVICE_UNKNOWN.

• DeviceCharacteristics - набор флагов, актуальных для некоторых конкретных драйверов.

Драйверы программного обеспечения указывают ноль или FILE_DEVICE_SECURE_OPEN, если они поддерживают истинное пространство имен.

• Exclusive — Указывает следует ли разрешить открытию более одного файлового объекта одному и тому же устройству. Большинству драйверов следует указать FALSE, но в некоторых случаях TRUE более уместно.

• DeviceObject - возвращаемый указатель, переданный как указатель на указатель. В случае успеха, IoCreateDevice выделяет структуру из невыгружаемого пула и сохраняет результирующий указатель внутри разыменованного аргумента.

Перед вызовом IoCreateDevice мы должны создать UNICODE_STRING для хранения имени внутреннего устройства:

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

// RtlInitUnicodeString(&devName, L"\\Device\\ThreadBoost");

Имя устройства может быть любым, но оно должно быть в каталоге диспетчера объектов устройств.

Есть два способа инициализировать UNICODE_STRING константной строкой.

Первый использует RtlInitUnicodeString, которая работает просто отлично. Но RtlInitUnicodeString должен считать количество символов в строке для соответствующей инициализации Length и MaximumLength.

В этом случае это долго, но есть более быстрый путь - использование макроса RTL_CONSTANT_STRING, который вычисляет длину строки статически во время компиляции, то есть она может корректно работать только с константной строкой.

Теперь мы можем вызвать функцию IoCreateDevice:

```
PDEVICE_OBJECT DeviceObject;
NTSTATUS status = IoCreateDevice(
DriverObject, // our driver object,
0 ,// no need for extra bytes,
&devName, // the device name,
FILE_DEVICE_UNKNOWN, // device type,
0, // characteristics flags,
FALSE, // not exclusive,
&DeviceObject // the resulting pointer
);
if (!NT_SUCCESS(status)) {
KdPrint(("Failed to create device object (0x%08X)\n", status));
return status;
}
```

Если все идет хорошо, у нас теперь есть указатель на наш объект устройства. Следующий шаг - сделать это устройство объектом доступным для абонентов пользовательского режима путем предоставления символической ссылки. Следующие строки создают символическую ссылку и подключают его к нашему объекту устройства:

```
UNICODE_STRING symLink =
RTL_CONSTANT_STRING(L"\\??\\PriorityBooster");
status = IoCreateSymbolicLink(&symLink, &devName);
if (!NT_SUCCESS(status)) {
KdPrint(("Failed to create symbolic link (0x%08X)\n", status));
IoDeleteDevice(DeviceObject);
return status;
}
```

IoCreateSymbolicLink выполняет свою работу, принимая символическую ссылку и имя устройства.

Обратите внимание, что если создание не удается, мы должны отменить все, что сделано до сих пор путем вызова IoDeleteDevice.

При любом статусе сбоя подпрограмма Unload не вызывается. Если бы у нас было больше шагов инициализации, пришлось-бы помнить все, чтобы отменить все до этого момента в случае неудачи.

Мы увидим более элегантный способ справиться с этим в главе 5.

Как только мы установим символическую ссылку и настроим объект устройства, DriverEntry может вернуть успех и драйвер теперь готов принимать запросы.

Прежде чем двигаться дальше, мы не должны забывать процедуру выгрузки. Предполагая, что DriverEntry завершен успешно, процедура выгрузки должна отменить все, что было сделано в DriverEntry.

В нашем случае есть две вещи: создание объекта устройства и создание символической ссылки. Мы отменим их в обратном порядке:

```
void PriorityBoosterUnload(_In_ PDRIVER_OBJECT DriverObject) {
UNICODE_STRING symLink = RTL_CONSTANT_STRING(L"\\??\\PriorityBooster");
// delete symbolic link
IoDeleteSymbolicLink(&symLink);
// delete device object
IoDeleteDevice(DriverObject->DeviceObject);
}
```

6)Код клиента

На этом этапе стоит написать код клиента в пользовательском режиме.

Все, что нам нужно для клиента уже было определено.

Добавьте новый консольный проект в решение с именем Booster (или другим именем на ваш выбор). Мастер Visual Studio должен создать один исходный файл (Visual Studio 2019), и два предварительно скомпилированных заголовочных файла (pch.h, pch.cpp) в Visual Studio 2017.

Вы можете спокойно игнорировать предварительно скомпилированные заголовочные файлы на данный момент.

В файле Booster.cpp удалите код «hello, world» по умолчанию и добавьте следующее объявление:

#include <windows.h>

#include <stdio.h>

#include "..\PriorityBooster\PriorityBoosterCommon.h"

Обратите внимание, что мы включили общий заголовочный файл, созданный драйвером и совместно используемый с клиентским кодом.

Измените основную функцию, чтобы принимать аргументы командной строки.

Мы примем идентификатор потока и приоритет, используя аргументы командной строки и запросим драйвер, изменить приоритет потока на данное значение.

int main(int argc, const char argv[]) {*

if (argc < 3) {

printf("Usage: Booster <threadid> <priority>\n");

return 0;

}

Далее нам нужно открыть хендл для нашего устройства.

Это «Имя файла» для CreateFile должно быть символической ссылкой с префиксом «\\. \». Весь вызов должен выглядеть так:

HANDLE hDevice = CreateFile(L"\\\\.\\PriorityBooster", GENERIC_WRITE,

FILE_SHARE_WRITE, nullptr, OPEN_EXISTING, 0, nullptr);

if (hDevice == INVALID_HANDLE_VALUE)

return Error("Failed to open device");

Функция Error просто печатает некоторый текст с последней произошедшей ошибкой:

int Error(const char message) {*

printf("%s (error=%d)\n", message, GetLastError());

return 1;

}

Вызов CreateFile должен достигнуть драйвера в его процедуре отправки IRP_MJ_CREATE.

Если драйвер в данный момент не загружен - это означает, что нет объекта устройства и символической ссылки - мы получим ошибку номер 2 (файл не найден).

Теперь, когда у нас есть действительный дескриптор нашего устройства, пришло время настроить вызов в DeviceIoControl.

Сначала нам нужно создать структуру ThreadData и заполнить её:

ThreadData data;

data.ThreadId = atoi(argv[1]); // command line first argument

data.Priority = atoi(argv[2]); // command line second argument

Теперь мы готовы вызвать DeviceIoControl и потом закрыть дескриптор устройства:

```
DWORD returned;
BOOL success = DeviceIoControl(hDevice,
IOCTL_PRIORITY_BOOSTER_SET_PRIORITY, // control code
&data, sizeof(data), // input buffer and length
nullptr, 0, // output buffer and length
&returned, nullptr);
if (success)
printf("Priority change succeeded!\n");
else
Error("Priority change failed!");
CloseHandle(hDevice);
```

DeviceIoControl достигает драйвера, вызывая основную функцию IRP_MJ_DEVICE_CONTROL.

На этом этапе код клиента готов. Все, что остается, - это реализовать процедуры отправки, которые мы реализуем на стороне драйвера.

7) Создание и закрытие процедур отправки

Теперь мы готовы реализовать три процедуры отправки, определенные драйвером.

Самые простые, это процедуры создания и закрытия. Все, что нужно, это выполнить запрос с успешным положение дел.

Вот полная реализация подпрограммы создание/закрытие:

```
_Use_decl_annotations_
NTSTATUS PriorityBoosterCreateClose(PDEVICE_OBJECT DeviceObject, PIRP Irp) {
UNREFERENCED_PARAMETER(DeviceObject);
Irp->IoStatus.Status = STATUS_SUCCESS;
Irp->IoStatus.Information = 0;
IoCompleteRequest(Irp, IO_NO_INCREMENT);
return STATUS_SUCCESS;
}
```

Каждая диспетчерская процедура принимает объект целевого устройства и пакет запроса ввода-вывода (IRP).

Мы не заботимся об объекте устройства, так как у нас он будет только один, так что это будет тот, который мы создали в DriverEntry.

IRP, с другой стороны, чрезвычайно важен. Мы углубимся в IRP в главе 6, но сейчас нам нужно взглянуть на IRP.

IRP - это полу-документированная структура, представляющая запрос, обычно исходящий от одного из менеджера в Executive: Диспетчер ввода/вывода, Plug&Play Manager или Power Manager.

У простого программного драйвера, скорее всего, будет диспетчер ввода-вывода.

Независимо от создателя IRP, задача драйвера - обработать IRP, что означает просмотр деталей запроса и выполнение что нужно сделать, чтобы завершить это.

Каждый запрос к драйверу всегда приходит в IRP, будь то Create, Close, Read или любой другой IRP.

Глядя на члены IRP, мы можем выяснить тип и детали запроса (технически, сама диспетчерская процедура была указана на основе типа запроса, поэтому в большинстве случаев вы уже знаете тип запроса).

Стоит отметить, что IRP никогда не приходит один, он сопровождается одной или несколькими структурами типа IO_STACK_LOCATION.

В простых случаях, таких как наш драйвер, существует один IO_STACK_LOCATION.

В более сложных случаях, когда есть драйверы фильтра, выше или ниже нас, то существует несколько экземпляров IO_STACK_LOCATION, по одному для каждого уровня в стеке устройства.(Мы обсудим это более подробно в главе 6).

Проще говоря, некоторая информация нам нужна в базовой структуре IRP, а некоторая находятся в IO_STACK_LOCATION для нашего «уровня» в стеке устройства.

В случае создания и закрытия нам не нужно заглядывать в какие-либо элементы. Нам просто нужно установить статус IRP в его члене IoStatus (типа IO_STATUS_BLOCK), который имеет два члена:

• **Статус** - указывает на состояние, с которым эти запросы будут завершены.

• **Информация** - полиморфный член, означающий разные вещи в разных запросах.

В случае Create и Close, нулевое значение, в случае успешного завершения.

Чтобы фактически завершить IRP, мы вызываем IoCompleteRequest.

Эта функция имеет много общего, но он передает IRP обратно своему создателю (обычно диспетчеру ввода-вывода), и этот менеджер уведомляет

клиент, который завершил операцию.

Второй аргумент - временное повышение приоритета, которое драйвер может предоставить своему клиенту.

В большинстве случаев нулевое значение является успешным завершением операции (IO_NO_INCREMENT определяется как ноль), потому что запрос завершен синхронно, поэтому вызывающая сторона не должна получать приоритетное повышение. Опять же, дополнительная информация об этой функции представлена в главе 6.

Последняя операция - вернуть тот же статус, который был введен в IRP. Это может показаться бесполезным дублированием, но это необходимо (причина будет яснее в следующей главе).

7)Программа отправки DeviceIoControl

Первое, что нам нужно проверить, это контрольный код. Типичные драйверы могут поддерживать множество управляющих кодов, поэтому мы хотим немедленно отклонить запрос, если контрольный код не распознан:

```
_Use_decl_annotations_
NTSTATUS PriorityBoosterDeviceControl(PDEVICE_OBJECT, PIRP Irp) {
// get our IO_STACK_LOCATION
auto stack = IoGetCurrentIrpStackLocation(Irp); // IO_STACK_LOCATION*
auto status = STATUS_SUCCESS;
switch (stack->Parameters.DeviceIoControl.IoControlCode) {
case IOCTL_PRIORITY_BOOSTER_SET_PRIORITY:
// do the work
break;
default:
status = STATUS_INVALID_DEVICE_REQUEST;
break;
}
```

Ключ к получению информации для любого IRP - заглянуть внутрь связанной IO_STACK_LOCATION с текущим уровнем устройства.

Вызов IoGetCurrentIrpStackLocation возвращает указатель на IO_STACK_LOCATION. В нашем случае действительно есть только один IO_STACK_LOCATION, но в любом случае использовать case IoGetCurrentIrpStackLocation является правильным

Основным компонентом IO_STACK_LOCATION является член с именем Parameters, который содержит набор структур, по одной для каждого типа IRP.

В случае IRP_MJ_DEVICE_CONTROL для использования является DeviceIoControl.

В этой структуре мы можем найти информацию, переданную клиентом, такой как управляющий код, буферы и их длины.

Оператор switch использует член IoControlCode, чтобы определить, понимаем ли мы контрольный код или нет. Если нет, мы просто устанавливаем статус чего-то другого, кроме успеха, и выходим из блока switch.

Последний фрагмент общего кода, который нам нужен, - завершить IRP после блока switch.

В противном случае клиент не получит ответ о завершении:

```
Irp->IoStatus.Status = status;

Irp->IoStatus.Information = 0;

IoCompleteRequest(Irp, IO_NO_INCREMENT);

return status;
```

Мы просто завершаем IRP с любым статусом. Если контрольный код не был распознан, это было бы состоянием отказа.

В противном случае это будет зависеть от фактической работы, проделанной в случае, если мы узнаем контрольный код.

Последний кусок является наиболее интересным и важным: выполнение реальной работы по изменению приоритет потока.

Первый шаг - проверить, достаточно ли велик полученный буфер, который содержит объект ThreadData.

Указатель на предоставленный пользователем входной буфер доступен в Type3InputBuffer и длина входного буфера в InputBufferLength:

```
if (stack->Parameters.DeviceIoControl.InputBufferLength < sizeof(ThreadData))
{

status = STATUS_BUFFER_TOO_SMALL;

break;

}
```

Вам может быть интересно, действительно ли возможен доступ к предоставленному буферу.

Поскольку этот буфер в пользовательском пространстве мы должны быть в контексте процесса клиента.

И действительно, как вызывающий поток клиента, перешел в режим ядра, описано в главе 1.

Далее, мы можем предположить, что буфер достаточно большой, поэтому давайте обработаем его как ThreadData:

auto data = (ThreadData)stack->Parameters.DeviceIoControl.Type3InputBuffer;*

Если указатель NULL, то мы должны выйти со статусом ошибки:

if (data == nullptr) {

status = STATUS_INVALID_PARAMETER;

break;

}

Далее, давайте посмотрим, находится ли приоритет в допустимом диапазоне от 1 до 31, и выйти с ошибкой, если нет:

if (data->Priority < 1 || data->Priority > 31) {

status = STATUS_INVALID_PARAMETER;

break;

}

Мы приближаемся к нашей цели.

API, который мы хотели бы использовать, это KeSetPriorityThread, прототип следующий:

KPRIORITY KeSetPriorityThread(

Inout PKTHREAD Thread,

_In_ KPRIORITY Priority);

Тип KPRIORITY - это просто 8-битное целое число. Сам поток идентифицируется указателем на KTHREAD.

KTHREAD - это одна из частей управления ядром потоками.

У нас есть идентификатор потока от клиента, и нам нужно каким-то образом указать на объект реального потока в пространстве ядра.

Функция, которая может искать поток по его идентификатору называется PsLookupThreadByThreadId.

Чтобы получить его определение, нам нужно добавить еще один #include:

#include <ntifs.h>

Обратите внимание, что вы должны добавить этот #include до <ntddk.h>, иначе вы получите ошибки компиляции.

Теперь мы можем превратить наш идентификатор потока в указатель:

PETHREAD Thread;

status = PsLookupThreadByThreadId(ULongToHandle(data->ThreadId), &Thread);

if (!NT_SUCCESS(status))

break;

В этом фрагменте кода есть несколько важных моментов:

• Функция поиска принимает handle, а не какой-то идентификатор.

Так это handle или ID ? Это идентификатор, как дескриптор.

Причина связана со способом генерации ID потоков. Они генерируются из глобальной частной таблицы дескрипторов ядра, поэтому значение дескрипторов являются фактическими идентификаторами.

Макрос ULongToHandle обеспечивает необходимое приведение. (Помните, что HANDLE 64-битный на 64-битных системах, но ID потока, предоставленный клиентом, всегда 32-битный.).

• Результирующий указатель определяется как PETHREAD или указатель на ETHREAD. Опять же, ETHREAD незадокументированный, несмотря на это, у нас, похоже, есть проблема, так как KeSetPriorityThread принимает PKTHREAD, а не PETHREAD.

Оказывается, это то же самое, потому что первым членом ETHREAD является KTHREAD (член называется Tcb).

Мы докажем все это в следующей главе, когда мы используем отладчик ядра. Суть в том, что мы можем смело переключать PKTHREAD для PETHREAD или наоборот при необходимости без заминки.

• PsLookupThreadByThreadId может завершиться ошибкой по разным причинам, например, из-за недопустимого идентификатора потока.

Теперь мы наконец готовы изменить приоритет. Но подождите - что если после последнего вызова поток завершается, как раз перед тем, как мы установим его новый приоритет?

Будьте уверены, этого не может быть на самом деле.

Технически поток может завершиться в этой точке, но это не сделает наш указатель висящим.

Это потому, что функция поиска, в случае успеха, увеличивает счетчик ссылок объекта потока в ядре, поэтому он не может умереть, пока мы явно не уменьшим счетчик ссылок.

Вот как сделать изменение приоритета:

KeSetPriorityThread((PKTHREAD)Thread, data->Priority);

Все, что осталось сделать, это уменьшить ссылку на объект потока; в противном случае у нас будет утечка, которая будет решена только при следующей загрузке системы.

Функция, которая выполняет это: *ObDereferenceObject(Thread);*

Вот полный обработчик IRP_MJ_DEVICE_CONTROL, с некоторыми незначительными косметическими изменениями:

```
_Use_decl_annotations_
NTSTATUS PriorityBoosterDeviceControl(PDEVICE_OBJECT, PIRP Irp) {
  // get our IO_STACK_LOCATION
  auto stack = IoGetCurrentIrpStackLocation(Irp); // IO_STACK_LOCATION*
  auto status = STATUS_SUCCESS;
  switch (stack->Parameters.DeviceIoControl.IoControlCode) {
    case IOCTL_PRIORITY_BOOSTER_SET_PRIORITY: {
      // do the work
      auto len = stack->Parameters.DeviceIoControl.InputBufferLength;
      if (len < sizeof(ThreadData)) {
        status = STATUS_BUFFER_TOO_SMALL;
        break;
      }
      auto data = (ThreadData*)stack->Parameters.DeviceIoControl.Type3InputBuffer;
      if (data == nullptr) {
        status = STATUS_INVALID_PARAMETER;
        break;
      }
      if (data->Priority < 1 || data->Priority > 31) {
        status = STATUS_INVALID_PARAMETER;
        break;
      }
      PETHREAD Thread;
      status = PsLookupThreadByThreadId(ULongToHandle(data->ThreadId), &Thread);
      if (!NT_SUCCESS(status))
        break;
      KeSetPriorityThread((PKTHREAD)Thread, data->Priority);
      ObDereferenceObject(Thread);
      KdPrint(("Thread Priority change for %d to %d succeeded!\n",
```

```
        data->ThreadId, data->Priority));
      break;
    }
    default:
      status = STATUS_INVALID_DEVICE_REQUEST;
      break;
  }
  Irp->IoStatus.Status = status;
  Irp->IoStatus.Information = 0;
  IoCompleteRequest(Irp, IO_NO_INCREMENT);
  return status;
}
```

8)Установка и тестирование

На данный момент мы можем успешно собрать драйвер и клиент. Наш следующий шаг - установить драйвер и проверить его функциональность. Вы можете попробовать следующее на виртуальной машине, или если вы чувствуете себя смелым - на твоем девайсе.)))

Во-первых, давайте установим драйвер. Откройте окно командной строки с повышенными правами и установите с помощью средства sc.exe как мы сделали еще в главе 2:

sc create booster type= kernel binPath= c:\Test\PriorityBooster.sys

Убедитесь, что binPath содержит полный путь к полученному файлу SYS. Имя драйвера (бустер) в примере, это имя созданного раздела реестра, поэтому оно должно быть уникальным.

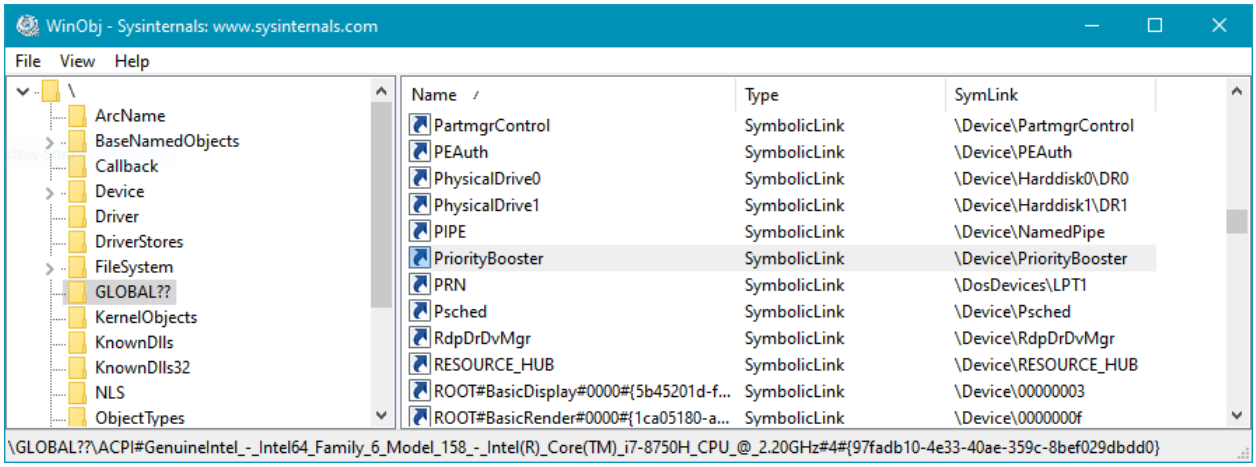
Это не должно быть связанно с именем файла SYS.

Теперь мы можем загрузить драйвер:

sc start booster

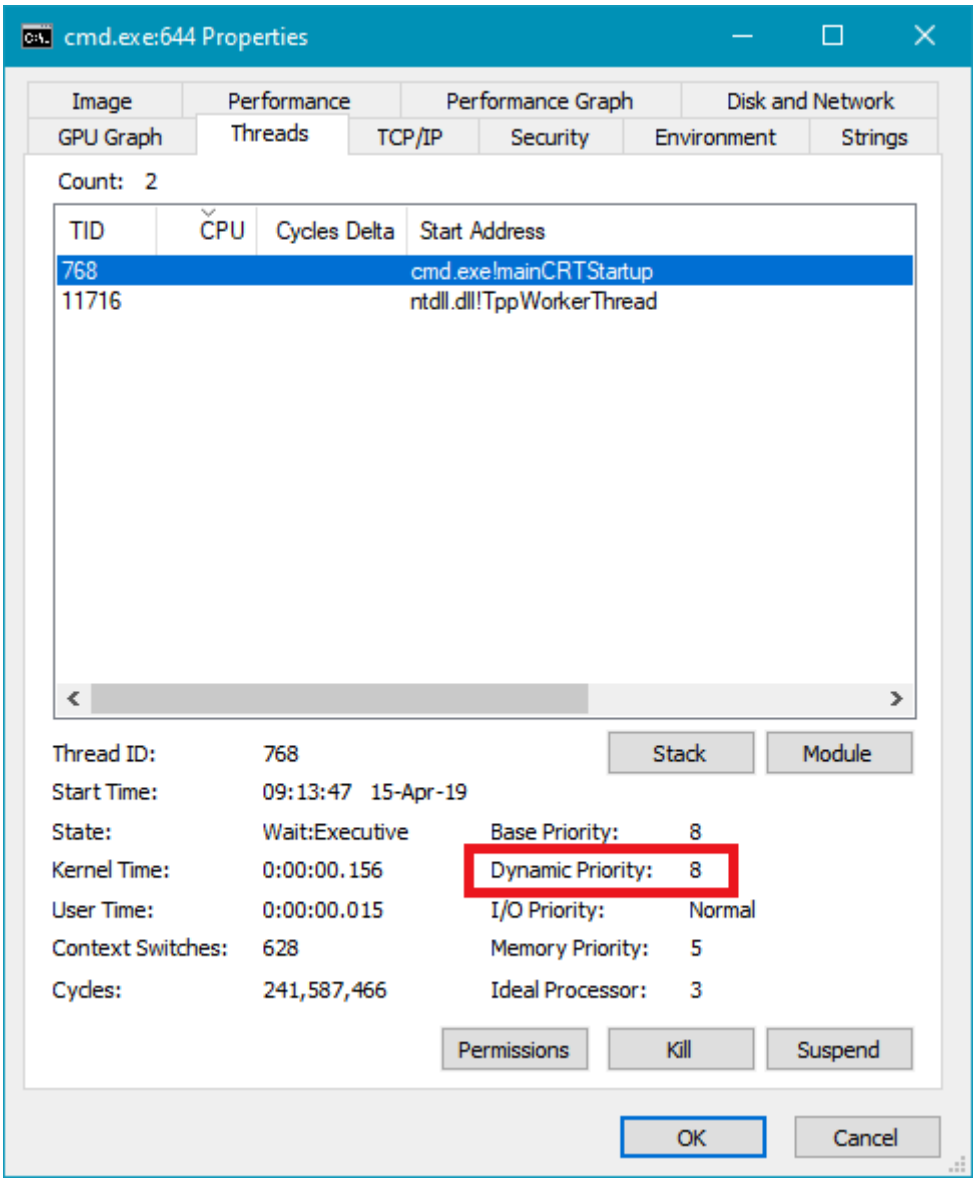
Если все в порядке, драйвер успешно запустится.

Чтобы убедиться в этом, мы можем открыть WinObj и найти название нашего устройства и символическую ссылку. Рисунок 4-1 показывает символическую ссылку в WinObj.



Теперь мы можем наконец запустить исполняемый файл клиента. Рисунок 4-2 показывает поток в Process Explorer.

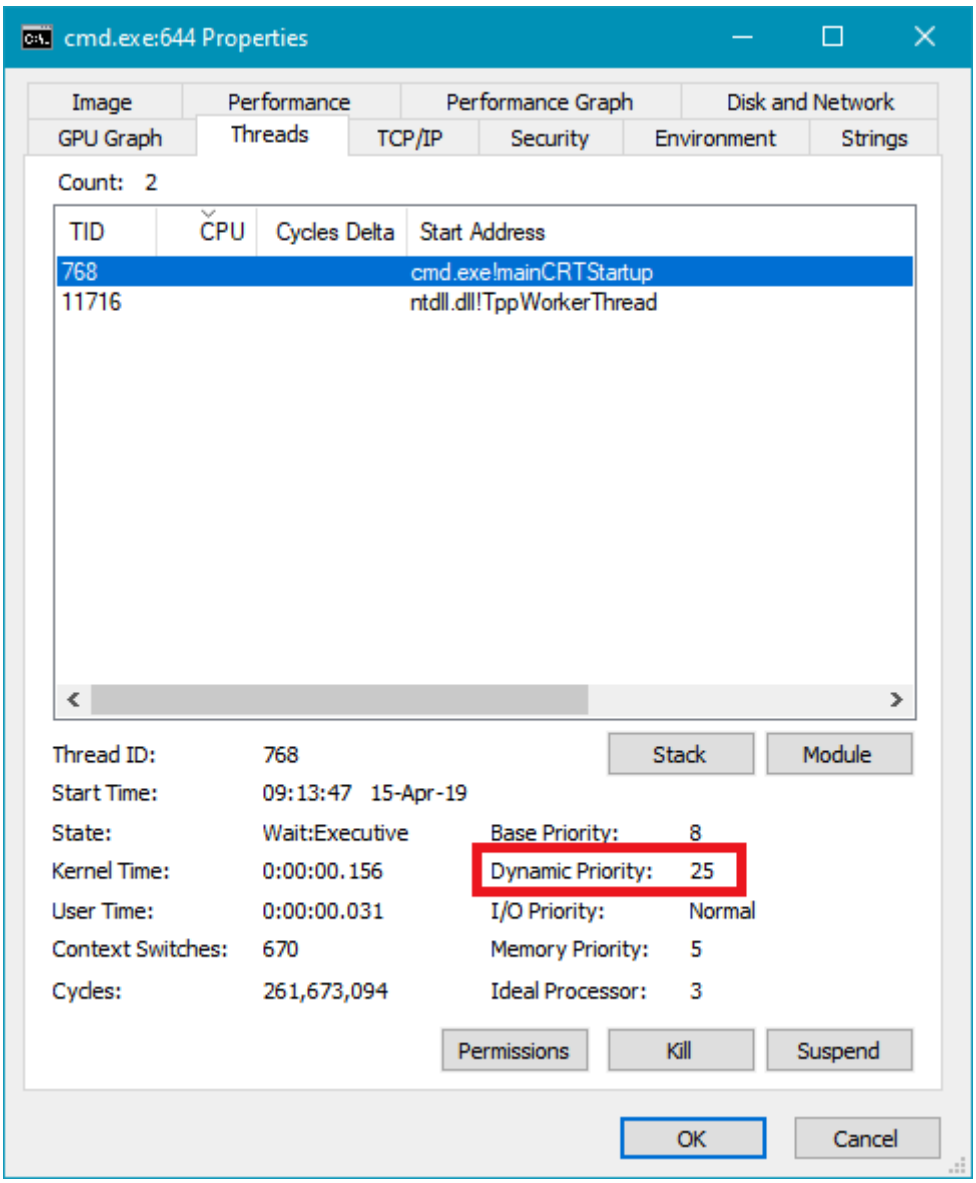
Процесс cmd.exe выбран в качестве примера, для которого мы хотим установить приоритет нового значения.



Запустите клиент с идентификатором потока и желаемым приоритетом (при необходимости замените идентификатор потока):

booster 768 25

И вуаля! Смотрите рисунок 4-3.



8)Резюме

Мы увидели, как создать простой, но полный драйвер, от начала до конца. Мы создали программу для связи с драйвером.

В следующей главе мы рассмотрим отладку, которую мы можем делать при написании драйверов, которые могут вести себя не так, как мы ожидаем.