Глава 9. Уведомления объектов и реестра

Ядро предоставляет больше способов перехвата определенных операций/событий. Сначала мы рассмотрим объект уведомления, где может быть перехвачено и получение дескрипторов некоторых типов объектов. Далее мы рассмотрим перехват операций реестра.

В этой главе:

- Уведомления об объектах.

- Драйвер защиты процессов .

- Уведомления реестра.

- Внедрение в уведомление реестра.

- Упражнения.

Уведомления об объектах

Ядро предоставляет механизм для уведомления заинтересованным драйверам при попытке открыть или скопировать дескриптор определенных типов объектов.

Официально поддерживаемые типы объектов: процесс, поток.

Для регистрации уведомления существует функция апи ObRegisterCallbacks, прототип которого выглядит так:

```
NTSTATUS ObRegisterCallbacks (
            _In_ POB_CALLBACK_REGISTRATION CallbackRegistration,
            _Outptr_ PVOID *RegistrationHandle);
```

Перед регистрацией должна быть инициализирована структура OB_CALLBACK_REGISTRATION, которая обеспечивает необходимые сведения о том, для чего регистрируется драйвер.

RegistrationHandle — это указатель, в котором хранится значение после успешной регистрации.

Вот определение OB_CALLBACK_REGISTRATION:

```
typedef struct _OB_CALLBACK_REGISTRATION {
      _In_ USHORT
      Version;
      _In_ USHORT
      OperationRegistrationCount;
      _In_ UNICODE_STRING
```

```
    Altitude;

    _In_ PVOID

    RegistrationContext;

    _In_ OB_OPERATION_REGISTRATION *OperationRegistration;
} OB_CALLBACK_REGISTRATION, *POB_CALLBACK_REGISTRATION;
```

`Version` - это просто константа, которая должна быть установлена в OB_FLT_REGISTRATION_VERSION (в настоящее время 0x100).

Затем количество операций, которые регистрируются, указывается - OperationRegistrationCount. Это определяет количество структур OB_OPERATION_REGISTRATION, которые указывают на OperationRegistration. Каждый из них предоставляет информацию об интересующем типе объекта (процесс, поток или рабочий стол).

Аргумент `Altitude` интересен. Он определяет число (в строковой форме), которое влияет на порядок обратных вызовов для этого драйвера. Это необходимо, потому что у других драйверов могут быть свои обратные вызовы и за вопрос о том, какой драйвер вызывается первым, отвечает `Altitude` - чем выше `Altitude`, тем раньше в цепочке вызовов вызывается драйвер. Какое значение должно быть `Altitude`? В большинстве случаев это не имеет значения, и все зависит от драйвера.

Указанный `Altitude` не должна совпадать с `Altitude`, который указан ранее зарегистрированными драйверами. `Altitude` не обязательно должна быть целым числом. Фактически, это десятичное число бесконечной точности, и поэтому он указан как строка.

Драйвер может даже генерировать случайные цифры, чтобы избежать конфликтов.

Если регистрация дало сбой со статусом STATUS_FLT_INSTANCE_ALTITUDE_COLLISION, это означает конфликты `Altitude`, так что осторожный драйвер может отрегулировать `Altitude` и попробовать еще раз.

Наконец, RegistrationContext - это определяемое драйвером значение, которое передается как есть подпрограмме обратного вызова. Структура OB_OPERATION_REGISTRATION - это то место, где драйвер устанавливает свои обратные вызовы, определяет какие типы объектов и операции представляют интерес. Это определяется так:

```
typedef struct _OB_OPERATION_REGISTRATION {
    _In_ POBJECT_TYPE

    *ObjectType;

    _In_ OB_OPERATION

    Operations;

    _In_ POB_PRE_OPERATION_CALLBACK PreOperation;

    _In_ POB_POST_OPERATION_CALLBACK PostOperation;
```

```
} OB_OPERATION_REGISTRATION, *POB_OPERATION_REGISTRATION;
```

ObjectType - это указатель на тип объекта для регистрации этого экземпляра - процесс, поток или рабочий стол.

`Operations` представляет собой перечисление битовых флагов с выбором создания/открытия (OB_OPERATION_HANDLE_CREATE) и/или дублировать (OB_OPERATION_HANDLE_DUPLICATE).

OB_OPERATION_HANDLE_CREATE, относится к вызовам функций пользовательского режима, таких как CreateProcess, OpenProcess, CreateThread, OpenThread, CreateDesktop, OpenDesktop и аналогичные функции для этих типов объектов.

OB_OPERATION_HANDLE_DUPLICATE относится к дублированию обработки для этих объектов (API пользовательского режима DuplicateHandle).

Каждый раз, когда выполняется один из этих вызовов (кстати, тоже из ядра), один или два обратных вызова могут быть зарегистрированы: обратный вызов до операции (поле PreOperation) и обратный вызов после операции (PostOperation).

Pre-Operation Callback

Обратный вызов перед операцией вызывается до завершения фактической операции создания/открытия/дублирования, дает возможность драйверу внести изменения в результат операции. Обратный вызов перед операцией получает структуру OB_PRE_OPERATION_INFORMATION, определенную, как показано здесь:

```
typedef struct _OB_PRE_OPERATION_INFORMATION {
      _In_ OB_OPERATION
      Operation;
      union {
            _In_ ULONG Flags;
            struct {
                  _In_ ULONG KernelHandle:1;
                  _In_ ULONG Reserved:31;
            };
      };
      _In_ PVOID
      Object;
      _In_ POBJECT_TYPE
      ObjectType;
      _Out_ PVOID
```

```
    CallContext;

    _In_ POB_PRE_OPERATION_PARAMETERS Parameters;
} OB_PRE_OPERATION_INFORMATION, *POB_PRE_OPERATION_INFORMATION;
```

Вот краткое изложение членов структуры:

• Operation - указывает, что это за операция (OB_OPERATION_HANDLE_CREATE или OB_OPERATION_HANDLE_DUPLICATE).

• KernelHandle - указывает, что это дескриптор ядра. Это позволяет драйверу игнорировать запросы ядра.

• Объект - указатель на фактический объект, для которого создается/открывается/дублируется дескриптор. Для процессов это адрес EPROCESS, для потока - это адрес PETHREAD.

• ObjectType - указывает на тип объекта: * PsProcessType, * PsThreadType или * ExDesktopObjectType.

• CallContext - значение, определяемое драйвером, которое передается в пост-обратный вызов для этого экземпляра. (если есть).

• Параметры - объединение, определяющее дополнительную информацию.

Это объединение определяется так:

```
typedef union _OB_PRE_OPERATION_PARAMETERS {
    _Inout_ OB_PRE_CREATE_HANDLE_INFORMATION
    CreateHandleInformation;
    _Inout_ OB_PRE_DUPLICATE_HANDLE_INFORMATION DuplicateHandleInformation;
} OB_PRE_OPERATION_PARAMETERS, *POB_PRE_OPERATION_PARAMETERS;
```

Драйвер должен проверить соответствующее поле в зависимости от операции. Для операций Create драйвер получает следующую информацию:

```
typedef struct _OB_PRE_CREATE_HANDLE_INFORMATION {
    _Inout_ ACCESS_MASK
    DesiredAccess;
    _In_ ACCESS_MASK
    OriginalDesiredAccess;
} OB_PRE_CREATE_HANDLE_INFORMATION, *POB_PRE_CREATE_HANDLE_INFORMATION;
```

OriginalDesiredAccess - это маска доступа, указанная вызывающей стороной.

Рассмотрим этот код пользовательского режима чтобы открыть дескриптор существующего процесса:

```
HANDLE OpenHandleToProcess(DWORD pid) {
	HANDLE hProcess = OpenProcess(PROCESS_QUERY_INFORMATION | PROCESS_VM_READ,
			FALSE, pid);
	if(!hProcess) {
		// failed to open a handle
	}
	return hProcess;
}
```

В этом примере клиент пытается получить дескриптор процесса с указанной маской доступа, указывая, каковы его «намерения» по отношению к этому процессу.

Обратный вызов драйвера перед операцией получает это значение в поле OriginalDesiredAccess. Это значение также копируется в DesiredAccess. Как обычно, ядро определит на основе контекста безопасности клиента и дескриптора безопасности процесса, может ли клиент получить желаемый доступ.

Драйвер может, исходя из своей собственной логики, изменить DesiredAccess, например, удалив некоторые объекты из доступа по запросу клиента:

```
OB_PREOP_CALLBACK_STATUS OnPreOpenProcess(PVOID /* RegistrationContext */,
		POB_PRE_OPERATION_INFORMATION Info) {
	if(/* some logic */) {
		Info->Parameters->CreateHandleInformation.DesiredAccess &= ~PROCESS_VM_READ;
	}
	return OB_PREOP_SUCCESS;
}
```

Приведенный выше фрагмент кода удаляет маску доступа PROCESS_VM_READ перед тем, как разрешить операцию. Если это в конечном итоге удастся, клиент вернет действительный дескриптор, но только с другой маской доступа.

Для дублирующих операций драйверу предоставляется следующая информация:

```
typedef struct _OB_PRE_DUPLICATE_HANDLE_INFORMATION {
      _Inout_ ACCESS_MASK
      DesiredAccess;
      _In_ ACCESS_MASK
      OriginalDesiredAccess;
      _In_ PVOID
      SourceProcess;
      _In_ PVOID
      TargetProcess;
} OB_PRE_DUPLICATE_HANDLE_INFORMATION, *POB_PRE_DUPLICATE_HANDLE_INFORMATION;
```

Поле DesiredAccess можно изменить, как и раньше. Предоставленная дополнительная информация является источником процесса (из которого дублируется дескриптор) и целевой процесс (новый дескриптор будет продублирован). Это позволяет драйверу запрашивать различные свойства этих процессов перед принятием решения о том, как изменить желаемую маску доступа.

Как мы можем получить дополнительную информацию о процессе, учитывая его адрес? Поскольку EPROCESS структура недокументирована, и есть только несколько экспортированных и задокументированных функций которые имеют дело с такими указателями напрямую - получение подробной информации может показаться проблематичным. Альтернативой является использование ZwQueryInformationProcess для получения необходимой информации, но функции требуется дескриптор, который можно получить, вызвав ObOpenObjectByPointer. Мы обсудим эту технику более подробно в главе 11.

Post-Operation Callback

Обратные вызовы после операции, вызываются после завершения операции. На этом этапе драйвер не может вносить какие-либо доработки, можно только проводить операции по имеющимся данным.

Обратный вызов после операции получает следующая структура:

```
typedef struct _OB_POST_OPERATION_INFORMATION {
      _In_ OB_OPERATION Operation;
      union {
            _In_ ULONG Flags;
            struct {
                  _In_ ULONG KernelHandle:1;
```

```
                    _In_ ULONG Reserved:31;
            };
        };
        _In_ PVOID
        Object;
        _In_ POBJECT_TYPE
        ObjectType;
        _In_ PVOID
        CallContext;
        _In_ NTSTATUS
        ReturnStatus;
        _In_ POB_POST_OPERATION_PARAMETERS Parameters;
} OB_POST_OPERATION_INFORMATION,*POB_POST_OPERATION_INFORMATION;
```

Это похоже на информацию обратного вызова перед операцией, за исключением следующего:

• Окончательный статус операции возвращается в ReturnStatus. В случае успеха клиент вернет действительный дескриптор (возможно, с другой маской доступа).

• Предоставляемое объединение параметров содержит только одну часть информации: маску доступа, предоставленную клиентом (при условии, что статус успешен).

Драйвер защиты процессов

Драйвер Process Protector является примером использования обратных вызовов объекта. Его цель - защитить определенные процессы от завершения, путем отказа в маске доступа PROCESS_TERMINATE.

Драйвер должен вести список защищенных процессов. В этом драйвере мы будем использовать простой ограниченный массив для хранения идентификаторов процессов под защитой драйвера. Вот структура (определенные в ProcessProtect.h):

```
#define DRIVER_PREFIX "ProcessProtect: "
#define PROCESS_TERMINATE 1
#include "FastMutex.h"
const int MaxPids = 256;
struct Globals {
        int PidsCount; // currently protected process count
        ULONG Pids[MaxPids]; // protected PIDs
        FastMutex Lock;
        PVOID RegHandle; // object registration cookie
        void Init() {
                Lock.Init();
        }
};
```

В основном файле (ProcessProtect.cpp) объявляется глобальная переменная типа Globals с именем g_Data (и вызывает Init в начале DriverEntry).

Регистрация объекта уведомления

Подпрограмма DriverEntry для драйвера защиты процесса должна включать регистрацию для обратных вызывов для процессов. Сначала готовим конструкции к регистрации:

```
OB_OPERATION_REGISTRATION operations[] = {
		{
				PsProcessType, // object type
				OB_OPERATION_HANDLE_CREATE |
OB_OPERATION_HANDLE_DUPLICATE,
				OnPreOpenProcess, nullptr // pre, post
		}
};
OB_CALLBACK_REGISTRATION reg = {
		OB_FLT_REGISTRATION_VERSION,
		1, // operation count
		RTL_CONSTANT_STRING(L"12345.6171"), // altitude
		nullptr, // context
		operations
};
```

Регистрация предназначена только для объектов процесса с возможностью предварительного обратного вызова. Этот обратный вызов должен удалить PROCESS_TERMINATE из желаемого доступа, запрошенного любым клиентом. Теперь мы готовы к фактической регистрации:

```
do {
	status = ObRegisterCallbacks(&reg, &g_Data.RegHandle);
	if (!NT_SUCCESS(status)) {
		break;
	}
```

Управление защищенными процессами

Драйвер поддерживает массив идентификаторов процессов для процессов, находящихся под его защитой.

Драйвер выставляет три управляющих кода ввода/вывода, позволяющие добавлять и удалять PID, а также очищать весь список. Управляющие коды определены в ProcessProtectCommon.h:

```
#define PROCESS_PROTECT_NAME L"ProcessProtect"
#define IOCTL_PROCESS_PROTECT_BY_PID \
		CTL_CODE(0x8000, 0x800, METHOD_BUFFERED, FILE_ANY_ACCESS)
#define IOCTL_PROCESS_UNPROTECT_BY_PID \
		CTL_CODE(0x8000, 0x801, METHOD_BUFFERED, FILE_ANY_ACCESS)
#define IOCTL_PROCESS_PROTECT_CLEAR \
		CTL_CODE(0x8000, 0x802, METHOD_NEITHER, FILE_ANY_ACCESS)
```

Для защиты и снятия защиты процессов обработчик IRP_MJ_DEVICE_CONTROL принимает массив PID (не обязательно только один).

Код обработчика является стандартным переключателем для известные управляющих кодов:

```
NTSTATUS ProcessProtectDeviceControl(PDEVICE_OBJECT, PIRP Irp) {
	auto stack = IoGetCurrentIrpStackLocation(Irp);
	auto status = STATUS_SUCCESS;
	auto len = 0;
	switch (stack->Parameters.DeviceIoControl.IoControlCode) {
	case IOCTL_PROCESS_PROTECT_BY_PID:
		//...
		break;
	case IOCTL_PROCESS_UNPROTECT_BY_PID:
		//...
		break;
	case IOCTL_PROCESS_PROTECT_CLEAR:
		//...

	break;
	default:
	status = STATUS_INVALID_DEVICE_REQUEST;
	break;
	}
	// complete the request
	Irp->IoStatus.Status = status;
	Irp->IoStatus.Information = len;
	IoCompleteRequest(Irp, IO_NO_INCREMENT);
	return status;
	}
```

Чтобы помочь с добавлением и удалением PID, мы создадим для этой цели две вспомогательные функции:

```
bool AddProcess(ULONG pid) {
	for(int i = 0; i < MaxPids; i++)
		if (g_Data.Pids[i] == 0) {
			// empty slot
			g_Data.Pids[i] = pid;
			g_Data.PidsCount++;
			return true;
		}
	return false;
}
bool RemoveProcess(ULONG pid) {
	for (int i = 0; i < MaxPids; i++)
		if (g_Data.Pids[i] == pid) {
			g_Data.Pids[i] = 0;
			g_Data.PidsCount--;
			return true;
		}
	return false;
}
```

Обратите внимание, что в этих функциях не используется быстрый мьютекс, что означает, что вызывающий должен получить быстрый мьютекс перед вызовом AddProcess или RemoveProcess.

В качестве последней функции мы будем использовать поиск идентификатора процесса в массиве и возвращать true, если он найден:

```
bool FindProcess(ULONG pid) {
	for (int i = 0; i < MaxPids; i++)
		if (g_Data.Pids[i] == pid)
			return true;
	return false;
```

```
}
```

Теперь мы готовы реализовать управляющие коды ввода/вывода. Для добавления процесса нам нужно найти пустой «Слот» в массиве идентификаторов процесса и запрошенный PID; конечно, мы можем получить более одного PID.

```
case IOCTL_PROCESS_PROTECT_BY_PID:
{
	auto size = stack->Parameters.DeviceIoControl.InputBufferLength;
	if (size % sizeof(ULONG) != 0) {
		status = STATUS_INVALID_BUFFER_SIZE;
		break;
	}
	auto data = (ULONG*)Irp->AssociatedIrp.SystemBuffer;
	AutoLock locker(g_Data.Lock);
	for (int i = 0; i < size / sizeof(ULONG); i++) {
		auto pid = data[i];
		if (pid == 0) {
			status = STATUS_INVALID_PARAMETER;
			break;
		}
		if (FindProcess(pid))
			continue;
		if (g_Data.PidsCount == MaxPids) {
			status = STATUS_TOO_MANY_CONTEXT_IDS;
			break;
		}
		if (!AddProcess(pid)) {
			status = STATUS_UNSUCCESSFUL;
			break;
		}
		len += sizeof(ULONG);
	}
	break;
}
```

Сначала код проверяет размер буфера, который должен быть кратен четырем байтам (PID), а не нулю.

Затем извлекается указатель на системный буфер (управляющий код использует METHOD_BUFFERED - см. главу 7, если вам нужно что-то напомнить). Теперь быстрый мьютекс получен, и цикл начинается. Цикл перебирает все PID, указанные в запросе, и, если все следующее верно, добавляет PID к массив:

• PID не равен нулю (Ноль всегда недопустимый PID, зарезервированный для незанятого процесса).

• PID еще нет в массиве (FindProcess определяет это).

• Количество управляемых PID не превышает MaxPids. Удаление PID аналогично. Мы должны найти его, а затем «удалить», поместив ноль в этот слот (это это задача для RemoveProcess):

```
case IOCTL_PROCESS_UNPROTECT_BY_PID:
{
	auto size = stack->Parameters.DeviceIoControl.InputBufferLength;
	if (size % sizeof(ULONG) != 0) {
		status = STATUS_INVALID_BUFFER_SIZE;
		break;
```

```
        }
        auto data = (ULONG*)Irp->AssociatedIrp.SystemBuffer;
        AutoLock locker(g_Data.Lock);
        for (int i = 0; i < size / sizeof(ULONG); i++) {
                auto pid = data[i];
                if (pid == 0) {
                        status = STATUS_INVALID_PARAMETER;
                        break;
                }
                if (!RemoveProcess(pid))
                        continue;
                len += sizeof(ULONG);
                if (g_Data.PidsCount == 0)
                        break;
        }
        break;
}
```

Наконец, очистить список довольно просто, если это делается при удерживании блокировки:

```
case IOCTL_PROCESS_PROTECT_CLEAR:
{
        AutoLock locker(g_Data.Lock);
        ::memset(&g_Data.Pids, 0, sizeof(g_Data.Pids));
        g_Data.PidsCount = 0;
        break;
}
```

The Pre-Callback

Самая важная часть драйвера - это удаление PROCESS_TERMINATE для PID, которые в настоящее время защищено от завершения:

```
OB_PREOP_CALLBACK_STATUS
OnPreOpenProcess(PVOID, POB_PRE_OPERATION_INFORMATION Info) {
        if(Info->KernelHandle)
                return OB_PREOP_SUCCESS;
        auto process = (PEPROCESS)Info->Object;
        auto pid = HandleToULong(PsGetProcessId(process));
        AutoLock locker(g_Data.Lock);
        if (FindProcess(pid)) {
                // found in list, remove terminate access
                Info->Parameters->CreateHandleInformation.DesiredAccess &=
                                ~PROCESS_TERMINATE;
        }
        return OB_PREOP_SUCCESS;
}
```

Если дескриптор является дескриптором ядра, мы позволяем операции продолжаться нормально. Это имеет смысл, поскольку мы хотим, чтобы код ядра работал правильно.

Теперь нам нужен идентификатор процесса, для которого открывается дескриптор. Данные, предоставленные в обратном вызове как указатель объекта. К счастью, получить PID просто с помощью API PsGetProcessId.

Последняя часть, проверяющая, действительно ли мы защищаем этот конкретный процесс или нет, поэтому мы вызываем FindProcess. В случае обнаружения удаляем доступ PROCESS_TERMINATE.

Клиентское приложение

Клиентское приложение должно иметь возможность добавлять, удалять и очищать процессы, выдавая правильные DeviceIoContro.

Интерфейс командной строки демонстрируется следующими командами (при условии, что исполняемый файл Protect.exe):

Protect.exe add 1200 2820 (protect PIDs 1200 and 2820)

Protect.exe remove 2820 (remove protection from PID 2820)

Protect.exe clear (remove all PIDs from protection)

Вот основная функция:

```
int wmain(int argc, const wchar_t* argv[]) {
	if(argc < 2)
		return PrintUsage();
	enum class Options {
		Unknown,
		Add, Remove, Clear
	};
	Options option;
	if (::_wcsicmp(argv[1], L"add") == 0)
		option = Options::Add;
	else if (::_wcsicmp(argv[1], L"remove") == 0)
		option = Options::Remove;
	else if (::_wcsicmp(argv[1], L"clear") == 0)
		option = Options::Clear;
	else {
		printf("Unknown option.\n");
		return PrintUsage();
	}
	HANDLE hFile = ::CreateFile(L"\\\\.\\" PROCESS_PROTECT_NAME,
			GENERIC_WRITE | GENERIC_READ, 0, nullptr, OPEN_EXISTING, 0,
nullptr);
	if (hFile == INVALID_HANDLE_VALUE)
		return Error("Failed to open device");
	std::vector<DWORD> pids;
	BOOL success = FALSE;
	DWORD bytes;
	switch (option) {
	case Options::Add:
		pids = ParsePids(argv + 2, argc - 2);
		success = ::DeviceIoControl(hFile, IOCTL_PROCESS_PROTECT_BY_PID,
				pids.data(), static_cast<DWORD>(pids.size()) *
sizeof(DWORD),
				nullptr, 0, &bytes, nullptr);
		break;
	case Options::Remove:
		pids = ParsePids(argv + 2, argc - 2);
		success = ::DeviceIoControl(hFile, IOCTL_PROCESS_UNPROTECT_BY_PID,
				pids.data(), static_cast<DWORD>(pids.size()) *
sizeof(DWORD),
				nullptr, 0, &bytes, nullptr);
		break;
	case Options::Clear:
		success = ::DeviceIoControl(hFile, IOCTL_PROCESS_PROTECT_CLEAR,
				nullptr, 0, nullptr, 0, &bytes, nullptr);
```

```
            break;
        }
        if (!success)
            return Error("Failed in DeviceIoControl");
        printf("Operation succeeded.\n");
        ::CloseHandle(hFile);
        return 0;
}
```

Вспомогательная функция ParsePids анализирует идентификаторы процессов и возвращает их как std :: vector <DWORD>, который легко передать как массив, используя метод data () в std :: vector <T>:

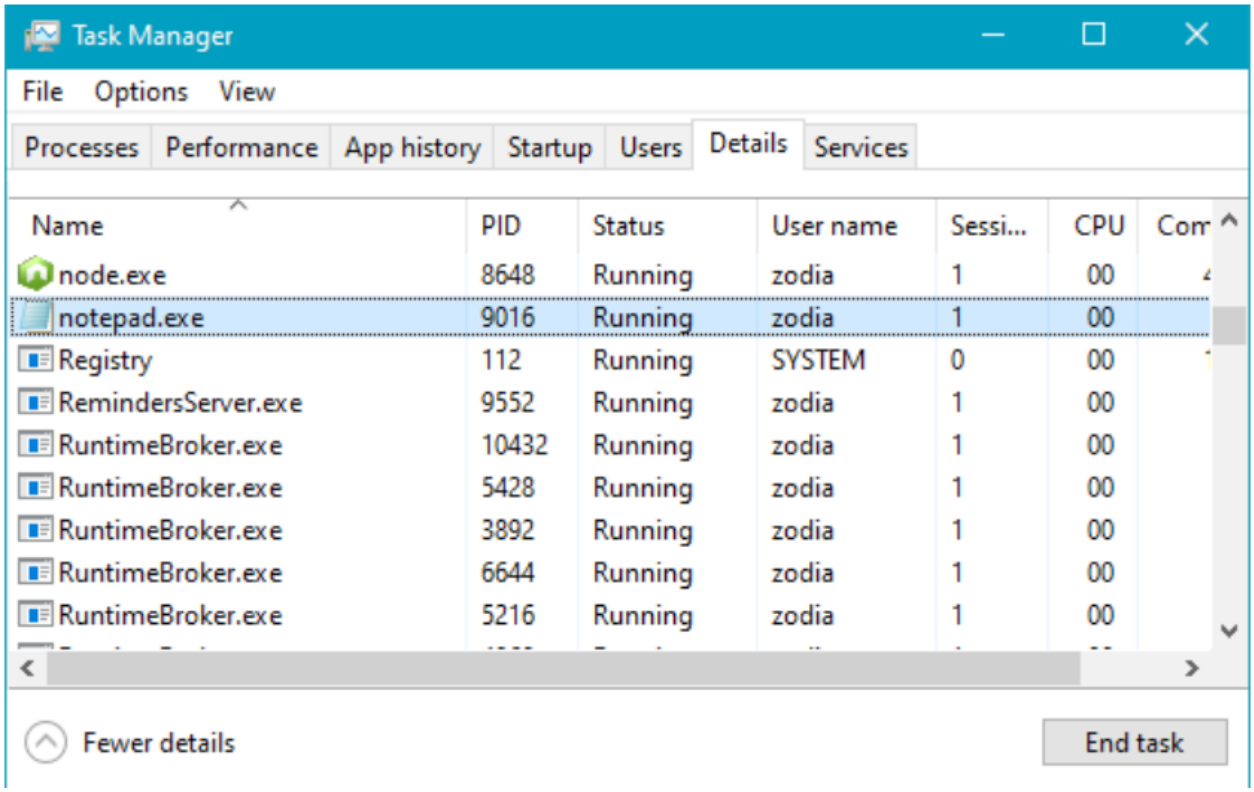
```
std::vector<DWORD> ParsePids(const wchar_t* buffer[], int count) {
        std::vector<DWORD> pids;
        for (int i = 0; i < count; i++)
            pids.push_back(::_wtoi(buffer[i]));
        return pids;
}
```

Наконец, функция Error та же, что и в предыдущих проектах, а PrintUsage просто отображает простую информация об использовании. Драйвер устанавливается, а потом запускается:

sc create protect type= kernel binPath= c:\book\processprotect.sys

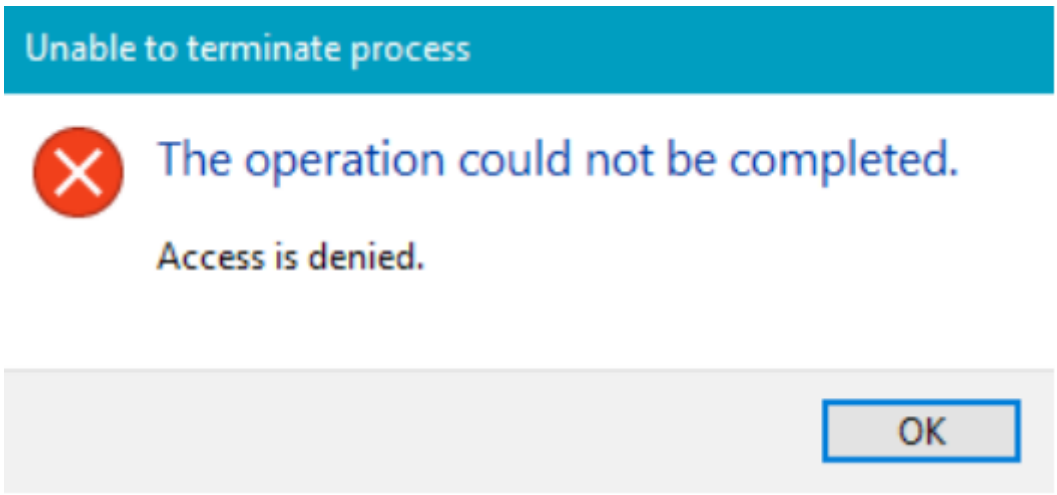
sc start protect

Давайте проверим его, запустив в качестве примера процесс (Notepad.exe), защитив его, а затем попробовав убить это с диспетчером задач. На рисунке показан запущенный экземпляр блокнота.



protect add 9016

При нажатии Завершить задачу в диспетчере задач появляется сообщение об ошибке:



Мы можем снять защиту и попробовать еще раз. На этот раз процесс завершен, как и ожидалось.

protect remove 9016

Интересно:

В случае с блокнотом, даже с защитой, нажав кнопку закрытия окна или выбрав Файл/Выход из меню прервет процесс.

Это потому, что это делается внутренне, вызывая ExitProcess, который не включает никаких дескрипторов. Это означает Механизм защиты, который мы здесь разработали, по существу хорош для процессов без пользовательского интерфейса.

Уведомления реестра

В чем-то похожий на уведомления об объектах.

Диспетчер конфигураций может использоваться для регистрации уведомлений о доступе к ключам реестра.

CmRegisterCallbackEx используется для регистрации таких уведомлений. Его прототип такой:

```
NTSTATUS CmRegisterCallbackEx (
    _In_        PEX_CALLBACK_FUNCTION   Function,
    _In_        PCUNICODE_STRING        Altitude,
    _In_        PVOID                   Driver,     // PDRIVER_OBJECT
    _In_opt_    PVOID                   Context,
    _Out_       PLARGE_INTEGER          Cookie,
    _Reserved_  PVOID                   Reserved
```

Function - это сам обратный вызов, который мы рассмотрим чуть позже.

Context - это значение, определяемое драйвером, передаваемое как есть к обратному вызову.

Наконец, Cookie - это результат успешной регистрации.

Функция обратного вызова довольно общая, она показана здесь:

```
NTSTATUS RegistryCallback (
            _In_ PVOID CallbackContext,
            _In_opt_ PVOID Argument1,
            _In_opt_ PVOID Argument2);
```

CallbackContext - это аргумент контекста, переданный в CmRegisterCallbackEx.

Argument1 на самом деле является перечислением REG_NOTIFY_CLASS, описывающий операцию, для которой обратный вызов вызывается, независимо от того, происходит ли это до или после уведомления.

Argument2 - указатель конкретной структуры, относящейся к этому типу уведомления. Драйвер обычно включает тип уведомления так:

```
NTSTATUS OnRegistryNotify(PVOID, PVOID Argument1, PVOID Argument2) {
      switch ((REG_NOTIFY_CLASS)(ULONG_PTR)Argument1) {
      //...
      }
```

В таблице показаны некоторые значения из перечисления REG_NOTIFY_CLASS, данная структура передается как Argument2.

Notification	Associated structure
RegNtPreDeleteKey	REG_DELETE_KEY_INFORMATION
RegNtPostDeleteKey	REG_POST_OPERATION_INFORMATION
RegNtPreSetValueKey	REG_SET_VALUE_KEY_INFORMATION
RegNtPostSetValueKey	REG_POST_OPERATION_INFORMATION
RegNtPreCreateKey	REG_PRE_CREATE_KEY_INFORMATION
RegNtPostCreateKey	REG_POST_CREATE_KEY_INFORMATION

Обработка предварительных уведомлений

Обратный вызов вызывается для предварительных операций, прежде чем они будут выполнены Configuration Manager.

У драйвера есть следующие возможности:

• Возвращение STATUS_SUCCESS из обратного вызова, тогда будет продолжаться обработка операции в обычном режиме.

• Возврат некоторого состояния отказа из обратного вызова. В этом случае Configuration Manager возвратит ошибку и операция не будет выполнена.

• Обработать запрос каким-либо образом, а затем вернуть STATUS_CALLBACK_BYPASS из обратного вызова. Configuration Manager возвращает вызывающему объекту успешное выполнение и не вызывает операцию. Драйвер должен позаботиться о том, чтобы установить правильные значения в REG_xxx_KEY_INFORMATION.

Обработка постопераций

Выполняется после завершения операции и при условии, что драйвер не препятствовал выполнению пост-операции.

Если это происходит, обратный вызов вызывается после того, как Configuration Manager выполнил операцию.

Вот прототип обратного вызова:

```
typedef struct _REG_POST_OPERATION_INFORMATION {
    PVOID     Object;         // input
    NTSTATUS  Status;         // input
    PVOID     PreInformation; // The pre information
    NTSTATUS  ReturnStatus;   // callback can change the outcome of the operation
    PVOID     CallContext;
    PVOID     ObjectContext;
    PVOID     Reserved;
} REG_POST_OPERATION_INFORMATION,*PREG_POST_OPERATION_INFORMATION;
```

Вы можете сделать следующие операции:

• Посмотрите на результат операции и сделайте что-нибудь (например, запишите лог и т.д.).

• Измените статус возврата, установив новое значение статуса в поле ReturnStatus структуры пост-операции, а затем возврат STATUS_CALLBACK_BYPASS. Конфигурационный менеджер возвращает этот новый статус вызывающему абоненту.

• Измените выходные параметры в структуре REG_xxx_KEY_INFORMATION и верните STATUS_SUCCESS. Configuration Manager возвращает эти новые данные вызывающей стороне.

Вопрос производительности

Обратный вызов реестра вызывается для каждой операции реестра, нет способа отфильтровать только определенные операции.

Это означает, что обратный вызов должен быть как можно быстрее, поскольку вызывающий ожидает обработки.

Кроме того, в цепочке обратных вызовов может быть более одного драйвера.

Некоторые операции с реестром, особенно операции чтения, выполняются в большом количестве, так что лучше для драйвера, это избежать обработки операций чтения, если это возможно.

Если он должен обрабатывать операцию чтения, то он должен как минимум ограничить свою обработку определенными интересующими ключами, например, HKLM \System\CurrentControlSet (просто пример).

Операции записи и создания используются гораздо реже, поэтому в этих случаях драйвер может делать больше, если необходимо.

Внедрение уведомлений реестра

Мы расширим наш драйвер SysMon из главы 8, включив в него уведомления для некоторых операций реестра.

Например, мы добавим уведомления о записи куда-нибудь в HKEY_LOCAL_MACHINE. Сначала мы определим структуру данных, которая будет включать в себя сообщаемую информацию (в SysMonCom.h):

```
struct RegistrySetValueInfo : ItemHeader {
    ULONG ProcessId;
    ULONG ThreadId;
    WCHAR KeyName[256];      // full key name
    WCHAR ValueName[64];     // value name
    ULONG DataType;          // REG_xxx
    UCHAR Data[128];         // data
    ULONG DataSize;          // size of data
};
```

Для простоты
мы будем использовать массивы фиксированного размера для сообщаемой информации.

В драйвере лучше сделать это динамическим, чтобы сэкономить память и предоставить полную информацию там, где это необходимо.

Массив данных - это фактически записанные данные. Естественно, мы должны каким-то образом ограничить его.

DataType - одна из констант типа REG_xxx, например REG_SZ, REG_DWORD, REG_BINARY и т. д., значения одинаковы в пользовательском режиме и режиме ядра.

Затем мы добавим новый тип события для этого уведомления:

```
enum class ItemType : short {
      None,
      ProcessCreate,
      ProcessExit,
      ThreadCreate,
      ThreadExit,
      ImageLoad,
      // new value
      RegistrySetValue
};
```

В DriverEntry нам нужно добавить регистрацию обратного вызова реестра как часть блока do/while(false).

Возвращенный файл cookie, представляющий регистрацию, сохраняется в нашей структуре Globals:

```
UNICODE_STRING altitude = RTL_CONSTANT_STRING(L"7657.124");

status = CmRegisterCallbackEx(OnRegistryNotify, &altitude, DriverObject,

            nullptr, &g_Globals.RegCookie, nullptr);

if(!NT_SUCCESS(status)) {

      KdPrint((DRIVER_PREFIX "failed to set registry callback (%08X)\n",

                  status));

      break;

}
```

Конечно, мы должны отменить регистрацию в процедуре выгрузки:

```
CmUnRegisterCallback(g_Globals.RegCookie);
```

Обработка обратного вызова реестра

Наш обратный вызов должен учитывать только записи, сделанные в HKEY_LOCAL_MACHINE. Сначала мы включаем нужную операцию:

```
NTSTATUS OnRegistryNotify(PVOID context, PVOID arg1, PVOID arg2) {

      UNREFERENCED_PARAMETER(context);

      switch ((REG_NOTIFY_CLASS)(ULONG_PTR)arg1) {

      case RegNtPostSetValueKey:

            //...

      }

      return STATUS_SUCCESS;

}
```

В этом драйвере нас не заботят никакие другие операции, поэтому после переключения мы просто возвращаем успешный статус.

Обратите внимание, что мы исследуем пост-операцию.

```
auto args = (REG_POST_OPERATION_INFORMATION*)arg2;

if (!NT_SUCCESS(args->Status))

break;
```

Если операция не удалась, мы просто выходим.

Это просто произвольное решение для этого драйвера, на самом деле, другой драйвер может обрабатывать эти неудачные попытки.

Затем нам нужно проверить, находится ли рассматриваемый ключ под HKLM.

Если нет, мы просто пропускаем этот ключ.

Внутренний пути реестра, просматриваемые ядром, всегда начинаются с \ REGISTRY \ в качестве корня. После этого идет MACHINE \ - то же самое, что HKEY_LOCAL_MACHINE в коде пользовательского режима.

Это означает, что нам нужно проверить, начинается ли рассматриваемый ключ с \REGISTRY\MACHINE\.

Путь ключа не сохраняется в постструктуре и даже не сохраняется напрямую в предварительной структуре.

Вместо этого сам объект ключа реестра предоставляется как часть структуры постинформации. Тогда нам необходимо извлечь имя ключа с помощью CmCallbackGetKeyObjectIDEx и посмотреть, начинается ли оно с \REGISTRY\MACHINE\:

```
static const WCHAR machine[] = L"\\REGISTRY\\MACHINE\\";

PCUNICODE_STRING name;

if (NT_SUCCESS(CmCallbackGetKeyObjectIDEx(&g_Globals.RegCookie, args->Object,

            nullptr, &name, 0))) {

      // filter out none-HKLM writes

      if (::wcsncmp(name->Buffer, machine, ARRAYSIZE(machine) - 1) == 0) {


            HANDLE OpenHandleToProcess(DWORD pid) {

                  HANDLE hProcess = OpenProcess(PROCESS_QUERY_INFORMATION |
PROCESS_VM_READ,

                              FALSE, pid);

                  if(!hProcess) {

                        // failed to open a handle

                  }

                  return hProcess;

            }
```

Если условие выполняется, нам необходимо зафиксировать информацию об операции в нашем уведомлении.

Эта информация (тип данных, имя значения, фактическое значение и т.д.) снабжена структурой предварительной информации, которая, к счастью, доступна как часть постинформации.

```
auto preInfo = (REG_SET_VALUE_KEY_INFORMATION*)args->PreInformation;

NT_ASSERT(preInfo);

auto size = sizeof(FullItem<RegistrySetValueInfo>);
```

```
auto info = (FullItem<RegistrySetValueInfo>*)ExAllocatePoolWithTag(PagedPool,
            size, DRIVER_TAG);
if (info == nullptr)
      break;
// zero out structure to make sure strings are null-terminated when copied
RtlZeroMemory(info, size);
// fill standard data
auto& item = info->Data;
KeQuerySystemTimePrecise(&item.Time);
item.Size = sizeof(item);
item.Type = ItemType::RegistrySetValue;
// get client PID/TID (this is our caller)
item.ProcessId = HandleToULong(PsGetCurrentProcessId());
item.ThreadId = HandleToULong(PsGetCurrentThreadId());
// get specific key/value data
::wcsncpy_s(item.KeyName, name->Buffer, name->Length / sizeof(WCHAR) - 1);
::wcsncpy_s(item.ValueName, preInfo->ValueName->Buffer,
            preInfo->ValueName->Length / sizeof(WCHAR) - 1);
item.DataType = preInfo->Type;
item.DataSize = preInfo->DataSize;
::memcpy(item.Data, preInfo->Data, min(item.DataSize, sizeof(item.Data)));
PushItem(&info->Entry);
```

Конкретная структура предварительной информации (REG_SET_VALUE_KEY_INFORMATION) содержит информацию, которую мы ищем.

Код старается не копировать слишком много, чтобы не переполнить статически выделенные буферы. Наконец, если CmCallbackGetKeyObjectIDEx завершается успешно, полученное имя ключа должно быть явно освобождено:

```
CmCallbackReleaseKeyObjectIDEx(name);
```

Модификация кода клиента

Клиентское приложение необходимо изменить для поддержки этого нового типа события. Вот возможная реализация:

```
case ItemType::RegistrySetValue:

{

	DisplayTime(header->Time);

	auto info = (RegistrySetValueInfo*)buffer;

	printf("Registry write PID=%d: %ws\\%ws type: %d size: %d data: ",

			info->ProcessId, info->KeyName, info->ValueName,

			info->DataType, info->DataSize);

	switch (info->DataType) {

	case REG_DWORD:

		printf("0x%08X\n", *(DWORD*)info->Data);

		break;

	case REG_SZ:

	case REG_EXPAND_SZ:

		printf("%ws\n", (WCHAR*)info->Data);

		break;

	case REG_BINARY:

		DisplayBinary(info->Data, min(info->DataSize, sizeof(info-
>Data)));

		break;

		// add other cases... (REG_QWORD, REG_LINK, etc.)

	default:

		DisplayBinary(info->Data, min(info->DataSize, sizeof(info-
>Data)));

		break;

	}

	break;

}
```

DisplayBinary - это простая вспомогательная функция, которая показывает двоичные данные в виде серии шестнадцатеричных значений:

```
void DisplayBinary(const UCHAR* buffer, DWORD size) {
	for (DWORD i = 0; i < size; i++)
		printf("%02X ", buffer[i]);
	printf("\n");
}
```

Вот некоторые результаты для этого расширенного клиента и драйвера:

```
19:22:21.509: Thread 6488 Exited from process 8808
19:22:21.509: Thread 5348 Created in process 8252
19:22:21.510: Thread 5348 Exited from process 8252
19:22:21.531: Registry write PID=7288: \REGISTRY\MACHINE\SOFTWARE\Microsoft\Windows \
Search\FileChangeClientConfigs\{5B8A4E77-3A02-4093-BDDC-B46FAB03AEF5}\FileAttributes\
FilteredOut type: 4 size: 4 data: 0x00000000
19:22:21.531: Registry write PID=7288: \REGISTRY\MACHINE\SOFTWARE\Microsoft\Windows \
Search\FileChangeClientConfigs\{5B8A4E77-3A02-4093-BDDC-B46FAB03AEF5}\UsnSourceFilte\
redOut type: 4 size: 4 data: 0x00000008
19:22:21.531: Registry write PID=7288: \REGISTRY\MACHINE\SOFTWARE\Microsoft\Windows \
Search\FileChangeClientConfigs\{5B8A4E77-3A02-4093-BDDC-B46FAB03AEF5}\UsnReasonFilte\
redOut type: 4 size: 4 data: 0x00000000
19:22:21.531: Registry write PID=7288: \REGISTRY\MACHINE\SOFTWARE\Microsoft\Windows \
Search\FileChangeClientConfigs\{5B8A4E77-3A02-4093-BDDC-B46FAB03AEF5}\ConfigFlags ty\
pe: 4 size: 4 data: 0x00000001
19:22:21.531: Registry write PID=7288: \REGISTRY\MACHINE\SOFTWARE\Microsoft\Windows \
Search\FileChangeClientConfigs\{5B8A4E77-3A02-4093-BDDC-B46FAB03AEF5}\ScopeToMonitor\
 type: 1 size: 270 data: C:\Users\zodia\AppData\Local\Packages\Microsoft.Windows.Con\
tentD19:22:21.531: Registry write PID=7288: \REGISTRY\MACHINE\SOFTWARE\Microsoft\Win\
dows Search\FileChangeClientConfigs\{5B8A4E77-3A02-4093-BDDC-B46FAB03AEF5}\Monitored\
PathRegularExpressionExclusion type: 1 size: 2 data:
19:22:21.531: Registry write PID=7288: \REGISTRY\MACHINE\SOFTWARE\Microsoft\Windows \
Search\FileChangeClientConfigs\{5B8A4E77-3A02-4093-BDDC-B46FAB03AEF5}\ApplicationNam\
e type: 1 size: 36 data: RuntimeBroker.exe
19:22:21.531: Registry write PID=7288: \REGISTRY\MACHINE\SOFTWARE\Microsoft\Windows \
Search\FileChangeClientConfigs\{5B8A4E77-3A02-4093-BDDC-B46FAB03AEF5}\ClientId type:\
 4 size: 4 data: 0x00000001


19:22:21.531: Registry write PID=7288: \REGISTRY\MACHINE\SOFTWARE\Microsoft\Windows \
Search\FileChangeClientConfigs\{5B8A4E77-3A02-4093-BDDC-B46FAB03AEF5}\VolumeIndex ty\
pe: 4 size: 4 data: 0x00000001
19:22:21.678: Thread 4680 Exited from process 6040
19:22:21.678: Thread 4760 Exited from process 6040
```

Упражнения

1. Модифицируйте драйвер, который не будет разрешать внедрение потоков в другие процессы, если целевой процесс отлаживается.

2. Реализуйте драйвер, защищающий ключ реестра от изменений. Клиент может выслать драйверу ключи реестра для защиты или снятия защиты.

3. Модифицируйте драйвер, который перенаправляет операции записи в реестр, поступающие от выбранных процессов (настраивается клиентским приложением) на свой закрытый ключ, если они обращаются к HKEY_LOCAL_MACHINE. Если приложение записывает данные, оно переходит в свое частное хранилище. Если он считывает данные, сначала проверьте частное хранилище, и если там нет значения, перейдите к реальному ключу реестра. Это один из аспектов применения песочницы.

Резюме

В этой главе мы рассмотрели два механизма обратного вызова, поддерживаемые ядром - получение дескрипторов для определенных объектов и доступ к реестру. В следующей главе мы окунемся в новую территорию файловой системы и мини-фильтров.