Министерство образования и науки Российской Федерации

Санкт-Петербургский Политехнический Университет Петра Великого

—

Институт компьютерных наук и кибербезопасности

ЛАБОРАТОРНАЯ РАБОТА № 1

«Реализация моделей безопасности в ОС Windows»

1. по дисциплине «Методы безопасности компьютерных систем»
2. Выполнил
3. студент гр. 5151004/10101 Веремейчук Я.Ю.

<подпись>

1. Преподаватель
2. асс. преподавателя Овасапян Т.Д.

<подпись>

1. Санкт-Петербург
2. 2024
3. Цель работы

Исследование принципов разработки моделей безопасности, функционирующих на уровне ядра ОС Windows.

1. Постановка задачи

Разработать драйвер-фильтр для ОС Windows 7/8/8.1/10, позволяющий ограничить права доступа процессов к объектам файловой системы. Разграничение должно осуществляться по правилам дискреционной модели доступа.

1. Теоретические исследования

Драйвер фильтра файловой системы, или минифильтр, перехватывает запросы, направленные к файловой системе или другому драйверу фильтра файловой системы. Перехватывая запрос до того, как он достигнет намеченной цели, минифильтр может расширить или заменить функциональность, предоставленную исходной целью запроса.

Разработчики драйверов фильтров используют поставляемый системой менеджер фильтров, который предоставляет платформу для разработки драйверов фильтров без необходимости управлять всеми сложностями файлового ввода-вывода. Диспетчер фильтров упрощает разработку драйверов фильтров сторонних производителей и решает многие проблемы, связанные с устаревшей моделью драйверов фильтров, например, возможность управлять порядком загрузки с помощью назначенной высоты.

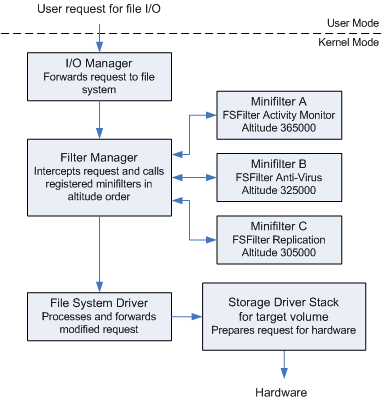


Рисунок 1 – Упрощенный стек ввода-вывода с диспетчером фильтров и тремя драйверами минифильтров

В работе был рассмотрен драйвер Minispy File System Minifilter Driver. Он представляет собой инструмент для мониторинга и регистрации любых операций ввода-вывода и транзакций, которые происходят в системе.

1. Ход работы

Для выполнения работы был скачан исходный код драйвера, запущен проект на хостовой машине Windows 10 в общей папке с виртуальной машиной Windows 10 и запущен драйвер.

Среди образцов драйверов файловой системы с сайта Microsoft.com был выбран наиболее подходящий к данной лабораторной работе, а именно minispy. Данный минифильтр предназначен для мониторинга и регистрации любых операций ввода-вывода и транзакций, происходящих в системе.

Для этого на гостевой системе был установлен режим тестовой подписи. 64-разрядные версии Windows 10 и 8 включают функцию "принудительного использования подписи драйвера". Они будут загружать только драйверы, подписанные Microsoft. Чтобы установить драйверы, не являющиеся официальными, необходимо отключить эту функцию. Для этого необходимо выполнить команду

bcdebit /set testigning on

и перезагрузить машину. Также аналогичного результата можно добиться при загрузке системы с выбором соответствующих опций в разделе «Устранение неполадок».

Файлы конфигурации загрузки (BCD) предоставляют хранилище, которое используется для описания параметров загрузки и загрузки приложений. Объекты и элементы в хранилище фактически заменяют Boot.ini. BCDEdit — это средство командной строки для управления хранилищами BCD.

Изображение выглядит как текст, снимок экрана, Шрифт, программное обеспечение

Автоматически созданное описание

Рисунок 2 – Активирование режима тестовой подписи

Изображение выглядит как текст, снимок экрана, Шрифт, программное обеспечение

Автоматически созданное описание

Рисунок 3 – Активирование режима тестовой подписи

Далее необходимо подготовить среду разработки, которая включает в себя Visual Studio, Window Driver Kit (WDK), Software Development Kit (SDK).

Для установки дополнительных пакетов необходимо зайти в проект VS, зайти в компоненты и установить требующиеся пакеты.

Изображение выглядит как текст, Шрифт, снимок экрана, число

Автоматически созданное описание

Рисунок 4 – Этап перехода к установке необходимых компонентов

Изображение выглядит как текст, снимок экрана, Шрифт, дизайн

Автоматически созданное описание

Рисунок 5 – Установка пакетов

Проект с драйвером до сих пор показывает ошибку при сборке. Чтобы это починить, необходимо изменить свойства для всех компонентов проекта: установить значение Disable в поле Spectre Mitigation.

Изображение выглядит как текст, программное обеспечение, Значок на компьютере, Мультимедийное программное обеспечение

Автоматически созданное описание

Рисунок 6 – Изменение свойств

После успешной сборки (x64) необходимо нажать ПКМ и установить файл .inf.

Изображение выглядит как текст, снимок экрана, программное обеспечение, число

Автоматически созданное описание

Рисунок 7 – Установка драйвера

Далее необходимо разрешить INF установку в открывающемся окне.

Изображение выглядит как текст, электроника, снимок экрана, программное обеспечение

Автоматически созданное описание

Рисунок 8 – Разрешение установки

Затем необходимо выбрать соответствующие файлы для установки.

Изображение выглядит как текст, программное обеспечение, снимок экрана, число

Автоматически созданное описание

Рисунок 9 – Процесс установки

Изображение выглядит как текст, снимок экрана, программное обеспечение, число

Автоматически созданное описание

Рисунок 10 – Процесс установки

Сначала драйвер не запускался, но была выполнена команда *fltmc load minispy*, которая исправила проблему. Также, если возникает ошибка отсутствия библиотеки vcruntime.dll, ее можно скачать и добавить в папку с вызываемым экзешником.

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Рисунок 11 – Запуск драйвера

Каждый драйвер минифильтра файловой системы должен иметь процедуру DriverEntry. Процедура DriverEntry вызывается при загрузке драйвера минифильтра. Программа DriverEntry выполняет глобальную инициализацию, регистрирует драйвер минифильтра и запускает фильтрацию. Эта процедура выполняется в контексте системного потока на уровне IRQL PASSIVE\_LEVEL.

DriverEntry имеет два входных параметра. Первый, DriverObject, это объект драйвера, который был создан при загрузке драйвера минифильтра. Второй, RegistryPath, представляет собой указатель на подсчитанную строку в Юникоде, которая содержит путь к разделу реестра драйвера минифильтра.

Программа DriverEntry драйвера минифильтра должна выполнять следующие шаги:

1. Выполнение глобальной инициализации для драйвера минифильтра.

2. Регистрирование драйвера минифильтра с помощью FltRegisterFilter.

3. Инициализация фильтрации с помощью FltStartFiltering.

4. Возвращение значения NTSTATUS.

Собственно, в код функции DriverEntry() было добавлено считывание конфигурационного файла с помощью ZwCreateFile и ZwReadFile.

Конфигурационный файл представлен в виде: [File] [Process] [Access].

Данные, считанные из конфигурационного файла, хранятся в виде следующей структуры rule.

Изображение выглядит как текст, Шрифт, снимок экрана, число

Автоматически созданное описание

Рисунок 12 – Структура rule

После успешного считывания информации из конфигурационного файла был добавлен участок кода, отвечающий за проверку прав доступа. Для этого необходимо получить имя файла с помощью FltParseFileNameInformation и имя процесса с помощью GetProcessNameByHandle.

Если зарегистрированная операция осуществляется в директории MBKS, то далее идет проверка прав доступа согласно информации, считанной из конфигурационного файла. Если нет необходимой записи, то доступ запрещен.

Для реагирования только на события чтения и записи в массиве обратных вызовов были убраны лишние события.

После всех проделанных действий важно освободить память, иначе будет BSOD.

Для отладки программы использовалась утилита DebugView.

Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание

Рисунок 13 – Запуск драйвера

Для указания папки с конфигурационным файлом необходимо записывать путь в особом формате.

Изображение выглядит как текст, Шрифт, снимок экрана, Графика

Автоматически созданное описание

Рисунок 14 – Путь к конфигурационному файлу

Minispy устанавливает бит “подавлять автоматическое вложение” во флагах экземпляра. Поэтому для того, чтобы правила вступили в силу, необходимо не просто установить и запустить измененный драйвер, но и выполнить команду /a \Device\HarddiskVolume2 (первый из HarddiskVolume, предложенный в подсказке).

Изображение выглядит как текст, снимок экрана, программное обеспечение, Шрифт

Автоматически созданное описание

Рисунок 15 – Запуск драйвера

Для тестирования драйвера была написана простейшая программа для записи текста и чтения из него.

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Рисунок 16 – Тестирование простейшей программы для записи текста в файл и чтения из него

Изображение выглядит как текст, электроника, снимок экрана, программное обеспечение

Автоматически созданное описание

Рисунок 17 – Тестирование программы с запущенным минифильтром

В качестве дополнительного задания было необходимо изменить программу так, чтобы при изменении конфигурационного файла работа драйвера не нарушалась и новые изменения сразу вступали в силу.

Ранее проверка на права доступа осуществлялась в функции SpyPreOperationCallback. Теперь в SpyPostOperationCallback.

Изображение выглядит как текст, снимок экрана, программное обеспечение, Значок на компьютере

Автоматически созданное описание

Рисунок 18 – Изменение конфигурационного файла и применение новых правил

1. Выводы

В ходе работы были исследованы принципы разработки моделей безопасности, функционирующих на уровне ядра ОС Windows, разработан драйвер-фильтр для ОС Windows 10, позволяющий ограничить права доступа процессов к объектам файловой системы по правилам дискреционной модели доступа.

# Приложение 1

Листинг кода minispy.c

/\*++

Copyright (c) 1989-2002 Microsoft Corporation

Module Name:

MiniSpy.c

Abstract:

This is the main module for the MiniSpy mini-filter.

Environment:

Kernel mode

--\*/

#include "mspyKern.h"

#include <stdio.h>

//

// Global variables

//

MINISPY\_DATA MiniSpyData;

NTSTATUS StatusToBreakOn = 0;

//---------------------------------------------------------------------------

// Function prototypes

//---------------------------------------------------------------------------

DRIVER\_INITIALIZE DriverEntry;

NTSTATUS

DriverEntry(

\_In\_ PDRIVER\_OBJECT DriverObject,

\_In\_ PUNICODE\_STRING RegistryPath

);

NTSTATUS

SpyMessage(

\_In\_ PVOID ConnectionCookie,

\_In\_reads\_bytes\_opt\_(InputBufferSize) PVOID InputBuffer,

\_In\_ ULONG InputBufferSize,

\_Out\_writes\_bytes\_to\_opt\_(OutputBufferSize, \*ReturnOutputBufferLength) PVOID OutputBuffer,

\_In\_ ULONG OutputBufferSize,

\_Out\_ PULONG ReturnOutputBufferLength

);

NTSTATUS

SpyConnect(

\_In\_ PFLT\_PORT ClientPort,

\_In\_ PVOID ServerPortCookie,

\_In\_reads\_bytes\_(SizeOfContext) PVOID ConnectionContext,

\_In\_ ULONG SizeOfContext,

\_Flt\_ConnectionCookie\_Outptr\_ PVOID\* ConnectionCookie

);

VOID

SpyDisconnect(

\_In\_opt\_ PVOID ConnectionCookie

);

NTSTATUS

SpyEnlistInTransaction(

\_In\_ PCFLT\_RELATED\_OBJECTS FltObjects

);

//---------------------------------------------------------------------------

// Assign text sections for each routine.

//---------------------------------------------------------------------------

#ifdef ALLOC\_PRAGMA

#pragma alloc\_text(INIT, DriverEntry)

#pragma alloc\_text(PAGE, SpyFilterUnload)

#pragma alloc\_text(PAGE, SpyQueryTeardown)

#pragma alloc\_text(PAGE, SpyConnect)

#pragma alloc\_text(PAGE, SpyDisconnect)

#pragma alloc\_text(PAGE, SpyMessage)

#endif

#define SetFlagInterlocked(\_ptrFlags,\_flagToSet) \

((VOID)InterlockedOr(((volatile LONG \*)(\_ptrFlags)),\_flagToSet))

//---------------------------------------------------------------------------

// ROUTINES

//---------------------------------------------------------------------------

typedef struct rule {

wchar\_t file\_name[32];

wchar\_t proc\_name[32];

int access;

}rule;

int num\_of\_rules = 0;

rule rules[16] = { 0 };

NTSTATUS

DriverEntry(

\_In\_ PDRIVER\_OBJECT DriverObject,

\_In\_ PUNICODE\_STRING RegistryPath

)

/\*++

Routine Description:

This routine is called when a driver first loads. Its purpose is to

initialize global state and then register with FltMgr to start filtering.

Arguments:

DriverObject - Pointer to driver object created by the system to

represent this driver.

RegistryPath - Unicode string identifying where the parameters for this

driver are located in the registry.

Return Value:

Status of the operation.

--\*/

{

PSECURITY\_DESCRIPTOR sd;

OBJECT\_ATTRIBUTES oa;

UNICODE\_STRING uniString;

NTSTATUS status = STATUS\_SUCCESS;

try {

//

// Initialize global data structures.

//

MiniSpyData.LogSequenceNumber = 0;

MiniSpyData.MaxRecordsToAllocate = DEFAULT\_MAX\_RECORDS\_TO\_ALLOCATE;

MiniSpyData.RecordsAllocated = 0;

MiniSpyData.NameQueryMethod = DEFAULT\_NAME\_QUERY\_METHOD;

MiniSpyData.DriverObject = DriverObject;

InitializeListHead(&MiniSpyData.OutputBufferList);

KeInitializeSpinLock(&MiniSpyData.OutputBufferLock);

ExInitializeNPagedLookasideList(&MiniSpyData.FreeBufferList,

NULL,

NULL,

POOL\_NX\_ALLOCATION,

RECORD\_SIZE,

SPY\_TAG,

0);

#if MINISPY\_VISTA

//

// Dynamically import FilterMgr APIs for transaction support

//

#pragma warning(push)

#pragma warning(disable:4055) // type cast from data pointer to function pointer

MiniSpyData.PFltSetTransactionContext = (PFLT\_SET\_TRANSACTION\_CONTEXT)FltGetRoutineAddress("FltSetTransactionContext");

MiniSpyData.PFltGetTransactionContext = (PFLT\_GET\_TRANSACTION\_CONTEXT)FltGetRoutineAddress("FltGetTransactionContext");

MiniSpyData.PFltEnlistInTransaction = (PFLT\_ENLIST\_IN\_TRANSACTION)FltGetRoutineAddress("FltEnlistInTransaction");

#pragma warning(pop)

#endif

//

// Read the custom parameters for MiniSpy from the registry

//

SpyReadDriverParameters(RegistryPath);

//

// Now that our global configuration is complete, register with FltMgr.

//

UNICODE\_STRING fullFileName;

HANDLE fileHandle;

IO\_STATUS\_BLOCK iostatus;

CHAR buffer\_data[1024];

memset(buffer\_data, 0, 1024);

LARGE\_INTEGER byteOffset;

byteOffset.LowPart = byteOffset.HighPart = 0;

DbgPrintEx(0, 0, "[Debug\_print] Start read config\n");

RtlInitUnicodeString(&fullFileName,

L"\\??\\C:\\MBKS\\config.txt");

InitializeObjectAttributes(&oa,

&fullFileName, OBJ\_CASE\_INSENSITIVE | OBJ\_KERNEL\_HANDLE,

NULL,

NULL);

status = ZwCreateFile(&fileHandle,

GENERIC\_READ,

&oa,

&iostatus,

0,

FILE\_ATTRIBUTE\_NORMAL,

0,

FILE\_OPEN,

FILE\_SYNCHRONOUS\_IO\_NONALERT,

NULL,

0);

if (NT\_SUCCESS(status)) {

status = ZwReadFile(fileHandle, NULL, NULL, NULL, &iostatus, buffer\_data, 1024, &byteOffset, NULL);

int pos = 0;

for (char\* c = buffer\_data; \*c; c++) {

int i = 0, j = 0;

if (!(\*c) || \*c == '\r')

break;

rule\* rule\_ = &rules[pos];

while (\*c != ' ')

rule\_->file\_name[j++] = \*c++;

rule\_->file\_name[j] = 0;

c++;

while (\*c != ' ')

rule\_->proc\_name[i++] = \*c++;

rule\_->proc\_name[i] = 0;

c++;

if (\*c == 'r')

rule\_->access = 1;

else if (\*c == 'w')

rule\_->access = 2;

else

rule\_->access = 3;

c += 2;

pos++;

DbgPrintEx(0, 0, "[Debug\_print] %ws %ws %d\n", rule\_->file\_name, rule\_->proc\_name, rule\_->access);

}

num\_of\_rules = pos;

DbgPrintEx(0, 0, "[Debug\_print] The config was read successfully. %d\n", num\_of\_rules);

ZwClose(fileHandle);

}

else {

DbgPrintEx(0, 0, "[Debug\_print] Error open config.\n");

ZwClose(fileHandle);

return STATUS\_UNSUCCESSFUL;

}

status = FltRegisterFilter(DriverObject,

&FilterRegistration,

&MiniSpyData.Filter);

if (!NT\_SUCCESS(status)) {

leave;

}

status = FltBuildDefaultSecurityDescriptor(&sd,

FLT\_PORT\_ALL\_ACCESS);

if (!NT\_SUCCESS(status)) {

leave;

}

RtlInitUnicodeString(&uniString, MINISPY\_PORT\_NAME);

InitializeObjectAttributes(&oa,

&uniString,

OBJ\_KERNEL\_HANDLE | OBJ\_CASE\_INSENSITIVE,

NULL,

sd);

status = FltCreateCommunicationPort(MiniSpyData.Filter,

&MiniSpyData.ServerPort,

&oa,

NULL,

SpyConnect,

SpyDisconnect,

SpyMessage,

1);

FltFreeSecurityDescriptor(sd);

if (!NT\_SUCCESS(status)) {

leave;

}

//

// We are now ready to start filtering

//

status = FltStartFiltering(MiniSpyData.Filter);

}

finally {

if (!NT\_SUCCESS(status)) {

if (NULL != MiniSpyData.ServerPort) {

FltCloseCommunicationPort(MiniSpyData.ServerPort);

}

if (NULL != MiniSpyData.Filter) {

FltUnregisterFilter(MiniSpyData.Filter);

}

ExDeleteNPagedLookasideList(&MiniSpyData.FreeBufferList);

}

}

return status;

}

NTSTATUS

SpyConnect(

\_In\_ PFLT\_PORT ClientPort,

\_In\_ PVOID ServerPortCookie,

\_In\_reads\_bytes\_(SizeOfContext) PVOID ConnectionContext,

\_In\_ ULONG SizeOfContext,

\_Flt\_ConnectionCookie\_Outptr\_ PVOID\* ConnectionCookie

)

/\*++

Routine Description

This is called when user-mode connects to the server

port - to establish a connection

Arguments

ClientPort - This is the pointer to the client port that

will be used to send messages from the filter.

ServerPortCookie - unused

ConnectionContext - unused

SizeofContext - unused

ConnectionCookie - unused

Return Value

STATUS\_SUCCESS - to accept the connection

--\*/

{

PAGED\_CODE();

UNREFERENCED\_PARAMETER(ServerPortCookie);

UNREFERENCED\_PARAMETER(ConnectionContext);

UNREFERENCED\_PARAMETER(SizeOfContext);

UNREFERENCED\_PARAMETER(ConnectionCookie);

FLT\_ASSERT(MiniSpyData.ClientPort == NULL);

MiniSpyData.ClientPort = ClientPort;

return STATUS\_SUCCESS;

}

VOID

SpyDisconnect(

\_In\_opt\_ PVOID ConnectionCookie

)

/\*++

Routine Description

This is called when the connection is torn-down. We use it to close our handle to the connection

Arguments

ConnectionCookie - unused

Return value

None

--\*/

{

PAGED\_CODE();

UNREFERENCED\_PARAMETER(ConnectionCookie);

//

// Close our handle

//

FltCloseClientPort(MiniSpyData.Filter, &MiniSpyData.ClientPort);

}

NTSTATUS

SpyFilterUnload(

\_In\_ FLT\_FILTER\_UNLOAD\_FLAGS Flags

)

/\*++

Routine Description:

This is called when a request has been made to unload the filter. Unload

requests from the Operation System (ex: "sc stop minispy" can not be

failed. Other unload requests may be failed.

You can disallow OS unload request by setting the

FLTREGFL\_DO\_NOT\_SUPPORT\_SERVICE\_STOP flag in the FLT\_REGISTARTION

structure.

Arguments:

Flags - Flags pertinent to this operation

Return Value:

Always success

--\*/

{

UNREFERENCED\_PARAMETER(Flags);

PAGED\_CODE();

//

// Close the server port. This will stop new connections.

//

FltCloseCommunicationPort(MiniSpyData.ServerPort);

FltUnregisterFilter(MiniSpyData.Filter);

SpyEmptyOutputBufferList();

ExDeleteNPagedLookasideList(&MiniSpyData.FreeBufferList);

return STATUS\_SUCCESS;

}

NTSTATUS

SpyQueryTeardown(

\_In\_ PCFLT\_RELATED\_OBJECTS FltObjects,

\_In\_ FLT\_INSTANCE\_QUERY\_TEARDOWN\_FLAGS Flags

)

/\*++

Routine Description:

This allows our filter to be manually detached from a volume.

Arguments:

FltObjects - Contains pointer to relevant objects for this operation.

Note that the FileObject field will always be NULL.

Flags - Flags pertinent to this operation

Return Value:

--\*/

{

UNREFERENCED\_PARAMETER(FltObjects);

UNREFERENCED\_PARAMETER(Flags);

PAGED\_CODE();

return STATUS\_SUCCESS;

}

NTSTATUS

SpyMessage(

\_In\_ PVOID ConnectionCookie,

\_In\_reads\_bytes\_opt\_(InputBufferSize) PVOID InputBuffer,

\_In\_ ULONG InputBufferSize,

\_Out\_writes\_bytes\_to\_opt\_(OutputBufferSize, \*ReturnOutputBufferLength) PVOID OutputBuffer,

\_In\_ ULONG OutputBufferSize,

\_Out\_ PULONG ReturnOutputBufferLength

)

/\*++

Routine Description:

This is called whenever a user mode application wishes to communicate

with this minifilter.

Arguments:

ConnectionCookie - unused

OperationCode - An identifier describing what type of message this

is. These codes are defined by the MiniFilter.

InputBuffer - A buffer containing input data, can be NULL if there

is no input data.

InputBufferSize - The size in bytes of the InputBuffer.

OutputBuffer - A buffer provided by the application that originated

the communication in which to store data to be returned to this

application.

OutputBufferSize - The size in bytes of the OutputBuffer.

ReturnOutputBufferSize - The size in bytes of meaningful data

returned in the OutputBuffer.

Return Value:

Returns the status of processing the message.

--\*/

{

MINISPY\_COMMAND command;

NTSTATUS status;

PAGED\_CODE();

UNREFERENCED\_PARAMETER(ConnectionCookie);

//

// \*\*\*\* PLEASE READ \*\*\*\*

//

// The INPUT and OUTPUT buffers are raw user mode addresses. The filter

// manager has already done a ProbedForRead (on InputBuffer) and

// ProbedForWrite (on OutputBuffer) which guarentees they are valid

// addresses based on the access (user mode vs. kernel mode). The

// minifilter does not need to do their own probe.

//

// The filter manager is NOT doing any alignment checking on the pointers.

// The minifilter must do this themselves if they care (see below).

//

// The minifilter MUST continue to use a try/except around any access to

// these buffers.

//

if ((InputBuffer != NULL) &&

(InputBufferSize >= (FIELD\_OFFSET(COMMAND\_MESSAGE, Command) +

sizeof(MINISPY\_COMMAND)))) {

try {

//

// Probe and capture input message: the message is raw user mode

// buffer, so need to protect with exception handler

//

command = ((PCOMMAND\_MESSAGE)InputBuffer)->Command;

} except(SpyExceptionFilter(GetExceptionInformation(), TRUE)) {

return GetExceptionCode();

}

switch (command) {

case GetMiniSpyLog:

//

// Return as many log records as can fit into the OutputBuffer

//

if ((OutputBuffer == NULL) || (OutputBufferSize == 0)) {

status = STATUS\_INVALID\_PARAMETER;

break;

}

//

// We want to validate that the given buffer is POINTER

// aligned. But if this is a 64bit system and we want to

// support 32bit applications we need to be careful with how

// we do the check. Note that the way SpyGetLog is written

// it actually does not care about alignment but we are

// demonstrating how to do this type of check.

//

#if defined(\_WIN64)

if (IoIs32bitProcess(NULL)) {

//

// Validate alignment for the 32bit process on a 64bit

// system

//

if (!IS\_ALIGNED(OutputBuffer, sizeof(ULONG))) {

status = STATUS\_DATATYPE\_MISALIGNMENT;

break;

}

}

else {

#endif

if (!IS\_ALIGNED(OutputBuffer, sizeof(PVOID))) {

status = STATUS\_DATATYPE\_MISALIGNMENT;

break;

}

#if defined(\_WIN64)

}

#endif

//

// Get the log record.

//

status = SpyGetLog(OutputBuffer,

OutputBufferSize,

ReturnOutputBufferLength);

break;

case GetMiniSpyVersion:

//

// Return version of the MiniSpy filter driver. Verify

// we have a valid user buffer including valid

// alignment

//

if ((OutputBufferSize < sizeof(MINISPYVER)) ||

(OutputBuffer == NULL)) {

status = STATUS\_INVALID\_PARAMETER;

break;

}

//

// Validate Buffer alignment. If a minifilter cares about

// the alignment value of the buffer pointer they must do

// this check themselves. Note that a try/except will not

// capture alignment faults.

//

if (!IS\_ALIGNED(OutputBuffer, sizeof(ULONG))) {

status = STATUS\_DATATYPE\_MISALIGNMENT;

break;

}

//

// Protect access to raw user-mode output buffer with an

// exception handler

//

try {

((PMINISPYVER)OutputBuffer)->Major = MINISPY\_MAJ\_VERSION;

((PMINISPYVER)OutputBuffer)->Minor = MINISPY\_MIN\_VERSION;

} except(SpyExceptionFilter(GetExceptionInformation(), TRUE)) {

return GetExceptionCode();

}

\*ReturnOutputBufferLength = sizeof(MINISPYVER);

status = STATUS\_SUCCESS;

break;

default:

status = STATUS\_INVALID\_PARAMETER;

break;

}

}

else {

status = STATUS\_INVALID\_PARAMETER;

}

return status;

}

//---------------------------------------------------------------------------

// Operation filtering routines

//---------------------------------------------------------------------------

typedef NTSTATUS(\*QUERY\_INFO\_PROCESS) (

\_\_in HANDLE ProcessHandle,

\_\_in PROCESSINFOCLASS ProcessInformationClass,

\_\_out\_bcount(ProcessInformationLength) PVOID ProcessInformation,

\_\_in ULONG ProcessInformationLength,

\_\_out\_opt PULONG ReturnLength

);

QUERY\_INFO\_PROCESS ZwQueryInformationProcess;

NTSTATUS GetProcessNameByHandle(PEPROCESS eProcess, PUNICODE\_STRING\* ProcessImageName) {

NTSTATUS status = STATUS\_SUCCESS;

ULONG returnedLength;

HANDLE hProcess = NULL;

PAGED\_CODE(); // this eliminates the possibility of the IDLE Thread/Process

if (eProcess == NULL) {

return STATUS\_INVALID\_PARAMETER\_1;

}

status = ObOpenObjectByPointer(eProcess, 0, NULL, 0, 0, KernelMode, &hProcess);

if (!NT\_SUCCESS(status)) {

return status;

}

if (ZwQueryInformationProcess == NULL) {

UNICODE\_STRING routineName = RTL\_CONSTANT\_STRING(L"ZwQueryInformationProcess");

ZwQueryInformationProcess = (QUERY\_INFO\_PROCESS)MmGetSystemRoutineAddress(&routineName);

if (ZwQueryInformationProcess == NULL) {

status = STATUS\_UNSUCCESSFUL;

goto cleanUp;

}

}

status = ZwQueryInformationProcess(hProcess,

ProcessImageFileName,

NULL, // buffer

0, // buffer size

&returnedLength);

if (STATUS\_INFO\_LENGTH\_MISMATCH != status) {

goto cleanUp;

}

\*ProcessImageName =

ExAllocatePoolWithTag(NonPagedPoolNx, returnedLength, '2gat');

if (ProcessImageName == NULL) {

status = STATUS\_INSUFFICIENT\_RESOURCES;

goto cleanUp;

}

status = ZwQueryInformationProcess(hProcess,

ProcessImageFileName,

\*ProcessImageName,

returnedLength,

&returnedLength);

if (!NT\_SUCCESS(status)) {

ExFreePoolWithTag(\*ProcessImageName, '2gat');

}

cleanUp:

ZwClose(hProcess);

return status;

}

FLT\_PREOP\_CALLBACK\_STATUS

#pragma warning(suppress: 6262)

SpyPreOperationCallbackRead(

\_Inout\_ PFLT\_CALLBACK\_DATA Data,

\_In\_ PCFLT\_RELATED\_OBJECTS FltObjects,

\_Flt\_CompletionContext\_Outptr\_ PVOID\* CompletionContext

)

/\*++

Routine Description:

This routine receives ALL pre-operation callbacks for this filter. It then

tries to log information about the given operation. If we are able

to log information then we will call our post-operation callback routine.

NOTE: This routine must be NON-PAGED because it can be called on the

paging path.

Arguments:

Data - Contains information about the given operation.

FltObjects - Contains pointers to the various objects that are pertinent

to this operation.

CompletionContext - This receives the address of our log buffer for this

operation. Our completion routine then receives this buffer address.

Return Value:

Identifies how processing should continue for this operation

--\*/

{

FLT\_PREOP\_CALLBACK\_STATUS returnStatus = FLT\_PREOP\_SUCCESS\_NO\_CALLBACK; //assume we are NOT going to call our completion routine

PRECORD\_LIST recordList;

PFLT\_FILE\_NAME\_INFORMATION nameInfo = NULL;

UNICODE\_STRING defaultName;

PUNICODE\_STRING nameToUse;

NTSTATUS status;

#if MINISPY\_VISTA

PUNICODE\_STRING ecpDataToUse = NULL;

UNICODE\_STRING ecpData;

WCHAR ecpDataBuffer[MAX\_NAME\_SPACE / sizeof(WCHAR)];

#endif

#if MINISPY\_NOT\_W2K

WCHAR name[MAX\_NAME\_SPACE / sizeof(WCHAR)];

#endif

PFLT\_FILE\_NAME\_INFORMATION FileNameInfo = NULL;

WCHAR FileName[512] = { 0 };

NTSTATUS status\_ = FltGetFileNameInformation(Data, FLT\_FILE\_NAME\_NORMALIZED | FLT\_FILE\_NAME\_QUERY\_DEFAULT, &FileNameInfo);

if (NT\_SUCCESS(status\_)) {

status\_ = FltParseFileNameInformation(FileNameInfo);

if (NT\_SUCCESS(status\_)) {

if (FileNameInfo->Name.MaximumLength < 512) {

RtlCopyMemory(FileName, FileNameInfo->Name.Buffer, FileNameInfo->Name.Length);

WCHAR buffer[512] = { 0 };

PUNICODE\_STRING processName = NULL;

GetProcessNameByHandle(IoThreadToProcess(Data->Thread), &processName);

RtlCopyMemory(buffer, processName->Buffer, processName->MaximumLength);

int mode = 1, flag = 1;

//if ((wcsstr(FileName, L"\\Device\\HarddiskVolume3\\MBKS\\") != NULL)) { //&& (wcsstr(buffer, L"\\Device\\HarddiskVolume3\\MBKS\\") != NULL)

for (int i = 0; i < num\_of\_rules; i++) {

if ((wcsstr(FileName, rules[i].file\_name) != NULL) && (wcsstr(buffer, rules[i].proc\_name) != NULL)) {

mode = rules[i].access;

DbgPrintEx(0, 0, "[Debug\_print] rule %ws for %ws access %d\n", rules[i].proc\_name, FileName, rules[i].access);

if ((Data->Iopb->MajorFunction == IRP\_MJ\_READ && mode == 2) || (Data->Iopb->MajorFunction == IRP\_MJ\_WRITE && mode == 1)) {

DbgPrintEx(0, 0, "[Debug\_print] ERROR: %ws for %ws blocked\n", rules[i].proc\_name, FileName);

Data->IoStatus.Status = STATUS\_ACCESS\_DENIED;

Data->IoStatus.Information = 0;

FltReleaseFileNameInformation(FileNameInfo);

return FLT\_PREOP\_COMPLETE;

}

flag = 0;

break;

}

}

//if (wcsstr(FileName, L"config.txt") != NULL) {

// OBJECT\_ATTRIBUTES oa;

// UNICODE\_STRING fullFileName;

// HANDLE fileHandle;

// IO\_STATUS\_BLOCK iostatus;

// CHAR buffer\_data[1024];

// memset(buffer\_data, 0, 1024);

// LARGE\_INTEGER byteOffset;

// byteOffset.LowPart = byteOffset.HighPart = 0;

// DbgPrintEx(0, 0, "[Debug\_print] Start update config\n");

// RtlInitUnicodeString(&fullFileName,

// L"\\??\\C:\\MBKS\\config.txt");

// InitializeObjectAttributes(&oa,

// &fullFileName, OBJ\_CASE\_INSENSITIVE | OBJ\_KERNEL\_HANDLE,

// NULL,

// NULL);

// DbgPrintEx(0, 0, "[Debug\_print] Create file\n");

// status = ZwCreateFile(&fileHandle,

// GENERIC\_READ,

// &oa,

// &iostatus,

// 0,

// FILE\_ATTRIBUTE\_NORMAL,

// 0,

// FILE\_OPEN,

// FILE\_SYNCHRONOUS\_IO\_NONALERT,

// NULL,

// 0);

// DbgPrintEx(0, 0, "[Debug\_print] Check statuc\n");

// if (NT\_SUCCESS(status)) {

// DbgPrintEx(0, 0, "[Debug\_print] Read file\n");

// status = ZwReadFile(fileHandle, NULL, NULL, NULL, &iostatus, buffer\_data, 1024, &byteOffset, NULL);

// int pos = 0;

// for (char\* c = buffer\_data; \*c; c++) {

// int i = 0, j = 0;

// if (!(\*c) || \*c == '\r')

// break;

// rule\* rule\_ = &rules[pos];

// while (\*c != ' ')

// rule\_->file\_name[j++] = \*c++;

// rule\_->file\_name[j] = 0;

// c++;

// while (\*c != ' ')

// rule\_->proc\_name[i++] = \*c++;

// rule\_->proc\_name[i] = 0;

// c++;

// if (\*c == 'r')

// rule\_->access = 1;

// else if (\*c == 'w')

// rule\_->access = 2;

// else

// rule\_->access = 3;

// c += 2;

// pos++;

// DbgPrintEx(0, 0, "[Debug\_print] %ws %ws %d\n", rule\_->file\_name, rule\_->proc\_name, rule\_->access);

// }

// num\_of\_rules = pos;

// DbgPrintEx(0, 0, "[Debug\_print] The config was update successfully. %d\n", num\_of\_rules);

// ZwClose(fileHandle);

// }

// else {

// DbgPrintEx(0, 0, "[Debug\_print] Error open config.\n");

// ZwClose(fileHandle);

// }

// DbgPrintEx(0, 0, "[Debug\_print] UPDATED!\n");

//}

/\*if (flag) {

DbgPrintEx(0, 0, "[Debug\_print] ERROR: The process isn't in the config.txt\n");

Data->IoStatus.Status = STATUS\_ACCESS\_DENIED;

Data->IoStatus.Information = 0;

FltReleaseFileNameInformation(FileNameInfo);

return FLT\_PREOP\_COMPLETE;

}\*/

//}

}

}

FltReleaseFileNameInformation(FileNameInfo);

}

//

// Try and get a log record

//

recordList = SpyNewRecord();

if (recordList) {

//

// We got a log record, if there is a file object, get its name.

//

// NOTE: By default, we use the query method

// FLT\_FILE\_NAME\_QUERY\_ALWAYS\_ALLOW\_CACHE\_LOOKUP

// because MiniSpy would like to get the name as much as possible, but

// can cope if we can't retrieve a name. For a debugging type filter,

// like Minispy, this is reasonable, but for most production filters

// who need names reliably, they should query the name at times when it

// is known to be safe and use the query method

// FLT\_FILE\_NAME\_QUERY\_DEFAULT.

//

if (FltObjects->FileObject != NULL) {

status = FltGetFileNameInformation(Data,

FLT\_FILE\_NAME\_NORMALIZED |

MiniSpyData.NameQueryMethod,

&nameInfo);

}

else {

//

// Can't get a name when there's no file object

//

status = STATUS\_UNSUCCESSFUL;

}

//

// Use the name if we got it else use a default name

//

if (NT\_SUCCESS(status)) {

nameToUse = &nameInfo->Name;

//

// Parse the name if requested

//

if (FlagOn(MiniSpyData.DebugFlags, SPY\_DEBUG\_PARSE\_NAMES)) {

#ifdef DBG

FLT\_ASSERT(NT\_SUCCESS(FltParseFileNameInformation(nameInfo)));

#else

FltParseFileNameInformation(nameInfo);

#endif

}

}

else {

#if MINISPY\_NOT\_W2K

NTSTATUS lstatus;

PFLT\_FILE\_NAME\_INFORMATION lnameInfo;

//

// If we couldn't get the "normalized" name try and get the

// "opened" name

//

if (FltObjects->FileObject != NULL) {

//

// Get the opened name

//

lstatus = FltGetFileNameInformation(Data,

FLT\_FILE\_NAME\_OPENED |

FLT\_FILE\_NAME\_QUERY\_ALWAYS\_ALLOW\_CACHE\_LOOKUP,

&lnameInfo);

if (NT\_SUCCESS(lstatus)) {

#pragma prefast(suppress:\_\_WARNING\_BANNED\_API\_USAGE, "reviewed and safe usage")

(VOID)\_snwprintf(name,

sizeof(name) / sizeof(WCHAR),

L"<%08x> %wZ",

status,

&lnameInfo->Name);

FltReleaseFileNameInformation(lnameInfo);

}

else {

//

// If that failed report both NORMALIZED status and

// OPENED status

//

#pragma prefast(suppress:\_\_WARNING\_BANNED\_API\_USAGE, "reviewed and safe usage")

(VOID)\_snwprintf(name,

sizeof(name) / sizeof(WCHAR),

L"<NO NAME: NormalizeStatus=%08x OpenedStatus=%08x>",

status,

lstatus);

}

}

else {

#pragma prefast(suppress:\_\_WARNING\_BANNED\_API\_USAGE, "reviewed and safe usage")

(VOID)\_snwprintf(name,

sizeof(name) / sizeof(WCHAR),

L"<NO NAME>");

}

//

// Name was initialized by \_snwprintf() so it may not be null terminated

// if the buffer is insufficient. We will ignore this error and truncate

// the file name.

//

name[(sizeof(name) / sizeof(WCHAR)) - 1] = L'\0';

RtlInitUnicodeString(&defaultName, name);

nameToUse = &defaultName;

#else

//

// We were unable to get the String safe routine to work on W2K

// Do it the old safe way

//

RtlInitUnicodeString(&defaultName, L"<NO NAME>");

nameToUse = &defaultName;

#endif //MINISPY\_NOT\_W2K

#if DBG

//

// Debug support to break on certain errors.

//

if (FltObjects->FileObject != NULL) {

NTSTATUS retryStatus;

if ((StatusToBreakOn != 0) && (status == StatusToBreakOn)) {

DbgBreakPoint();

}

retryStatus = FltGetFileNameInformation(Data,

FLT\_FILE\_NAME\_NORMALIZED |

MiniSpyData.NameQueryMethod,

&nameInfo);

if (!NT\_SUCCESS(retryStatus)) {

//

// We always release nameInfo, so ignore return value.

//

NOTHING;

}

}

#endif

}

#if MINISPY\_VISTA

//

// Look for ECPs, but only if it's a create operation

//

if (Data->Iopb->MajorFunction == IRP\_MJ\_CREATE) {

//

// Initialize an empty string to receive an ECP data dump

//

RtlInitEmptyUnicodeString(&ecpData,

ecpDataBuffer,

MAX\_NAME\_SPACE / sizeof(WCHAR));

//

// Parse any extra create parameters

//

SpyParseEcps(Data, recordList, &ecpData);

ecpDataToUse = &ecpData;

}

//

// Store the name and ECP data (if any)

//

SpySetRecordNameAndEcpData(&(recordList->LogRecord), nameToUse, ecpDataToUse);

#else

//

// Store the name

//

SpySetRecordName(&(recordList->LogRecord), nameToUse);

#endif

//

// Release the name information structure (if defined)

//

if (NULL != nameInfo) {

FltReleaseFileNameInformation(nameInfo);

}

//

// Set all of the operation information into the record

//

SpyLogPreOperationData(Data, FltObjects, recordList);

//

// Pass the record to our completions routine and return that

// we want our completion routine called.

//

if (Data->Iopb->MajorFunction == IRP\_MJ\_SHUTDOWN) {

//

// Since completion callbacks are not supported for

// this operation, do the completion processing now

//

SpyPostOperationCallback(Data,

FltObjects,

recordList,

0);

returnStatus = FLT\_PREOP\_SUCCESS\_NO\_CALLBACK;

}

else {

\*CompletionContext = recordList;

returnStatus = FLT\_PREOP\_SUCCESS\_WITH\_CALLBACK;

}

}

return returnStatus;

}

FLT\_PREOP\_CALLBACK\_STATUS

#pragma warning(suppress: 6262) // higher than usual stack usage is considered safe in this case

SpyPreOperationCallback(

\_Inout\_ PFLT\_CALLBACK\_DATA Data,

\_In\_ PCFLT\_RELATED\_OBJECTS FltObjects,

\_Flt\_CompletionContext\_Outptr\_ PVOID\* CompletionContext

)

/\*++

Routine Description:

This routine receives ALL pre-operation callbacks for this filter. It then

tries to log information about the given operation. If we are able

to log information then we will call our post-operation callback routine.

NOTE: This routine must be NON-PAGED because it can be called on the

paging path.

Arguments:

Data - Contains information about the given operation.

FltObjects - Contains pointers to the various objects that are pertinent

to this operation.

CompletionContext - This receives the address of our log buffer for this

operation. Our completion routine then receives this buffer address.

Return Value:

Identifies how processing should continue for this operation

--\*/

{

FLT\_PREOP\_CALLBACK\_STATUS returnStatus = FLT\_PREOP\_SUCCESS\_NO\_CALLBACK; //assume we are NOT going to call our completion routine

PRECORD\_LIST recordList;

PFLT\_FILE\_NAME\_INFORMATION nameInfo = NULL;

UNICODE\_STRING defaultName;

PUNICODE\_STRING nameToUse;

NTSTATUS status;

#if MINISPY\_VISTA

PUNICODE\_STRING ecpDataToUse = NULL;

UNICODE\_STRING ecpData;

WCHAR ecpDataBuffer[MAX\_NAME\_SPACE / sizeof(WCHAR)];

#endif

#if MINISPY\_NOT\_W2K

WCHAR name[MAX\_NAME\_SPACE / sizeof(WCHAR)];

#endif

//

// Try and get a log record

//

recordList = SpyNewRecord();

if (recordList) {

//

// We got a log record, if there is a file object, get its name.

//

// NOTE: By default, we use the query method

// FLT\_FILE\_NAME\_QUERY\_ALWAYS\_ALLOW\_CACHE\_LOOKUP

// because MiniSpy would like to get the name as much as possible, but

// can cope if we can't retrieve a name. For a debugging type filter,

// like Minispy, this is reasonable, but for most production filters

// who need names reliably, they should query the name at times when it

// is known to be safe and use the query method

// FLT\_FILE\_NAME\_QUERY\_DEFAULT.

//

if (FltObjects->FileObject != NULL) {

status = FltGetFileNameInformation(Data,

FLT\_FILE\_NAME\_NORMALIZED |

MiniSpyData.NameQueryMethod,

&nameInfo);

}

else {

//

// Can't get a name when there's no file object

//

status = STATUS\_UNSUCCESSFUL;

}

//

// Use the name if we got it else use a default name

//

if (NT\_SUCCESS(status)) {

nameToUse = &nameInfo->Name;

//

// Parse the name if requested

//

if (FlagOn(MiniSpyData.DebugFlags, SPY\_DEBUG\_PARSE\_NAMES)) {

#ifdef DBG

FLT\_ASSERT(NT\_SUCCESS(FltParseFileNameInformation(nameInfo)));

#else

FltParseFileNameInformation(nameInfo);

#endif

}

}

else {

#if MINISPY\_NOT\_W2K

NTSTATUS lstatus;

PFLT\_FILE\_NAME\_INFORMATION lnameInfo;

//

// If we couldn't get the "normalized" name try and get the

// "opened" name

//

if (FltObjects->FileObject != NULL) {

//

// Get the opened name

//

lstatus = FltGetFileNameInformation(Data,

FLT\_FILE\_NAME\_OPENED |

FLT\_FILE\_NAME\_QUERY\_ALWAYS\_ALLOW\_CACHE\_LOOKUP,

&lnameInfo);

if (NT\_SUCCESS(lstatus)) {

#pragma prefast(suppress:\_\_WARNING\_BANNED\_API\_USAGE, "reviewed and safe usage")

(VOID)\_snwprintf(name,

sizeof(name) / sizeof(WCHAR),

L"<%08x> %wZ",

status,

&lnameInfo->Name);

FltReleaseFileNameInformation(lnameInfo);

}

else {

//

// If that failed report both NORMALIZED status and

// OPENED status

//

#pragma prefast(suppress:\_\_WARNING\_BANNED\_API\_USAGE, "reviewed and safe usage")

(VOID)\_snwprintf(name,

sizeof(name) / sizeof(WCHAR),

L"<NO NAME: NormalizeStatus=%08x OpenedStatus=%08x>",

status,

lstatus);

}

}

else {

#pragma prefast(suppress:\_\_WARNING\_BANNED\_API\_USAGE, "reviewed and safe usage")

(VOID)\_snwprintf(name,

sizeof(name) / sizeof(WCHAR),

L"<NO NAME>");

}

//

// Name was initialized by \_snwprintf() so it may not be null terminated

// if the buffer is insufficient. We will ignore this error and truncate

// the file name.

//

name[(sizeof(name) / sizeof(WCHAR)) - 1] = L'\0';

RtlInitUnicodeString(&defaultName, name);

nameToUse = &defaultName;

#else

//

// We were unable to get the String safe routine to work on W2K

// Do it the old safe way

//

RtlInitUnicodeString(&defaultName, L"<NO NAME>");

nameToUse = &defaultName;

#endif //MINISPY\_NOT\_W2K

#if DBG

//

// Debug support to break on certain errors.

//

if (FltObjects->FileObject != NULL) {

NTSTATUS retryStatus;

if ((StatusToBreakOn != 0) && (status == StatusToBreakOn)) {

DbgBreakPoint();

}

retryStatus = FltGetFileNameInformation(Data,

FLT\_FILE\_NAME\_NORMALIZED |

MiniSpyData.NameQueryMethod,

&nameInfo);

if (!NT\_SUCCESS(retryStatus)) {

//

// We always release nameInfo, so ignore return value.

//

NOTHING;

}

}

#endif

}

#if MINISPY\_VISTA

//

// Look for ECPs, but only if it's a create operation

//

if (Data->Iopb->MajorFunction == IRP\_MJ\_CREATE) {

//

// Initialize an empty string to receive an ECP data dump

//

RtlInitEmptyUnicodeString(&ecpData,

ecpDataBuffer,

MAX\_NAME\_SPACE / sizeof(WCHAR));

//

// Parse any extra create parameters

//

SpyParseEcps(Data, recordList, &ecpData);

ecpDataToUse = &ecpData;

}

//

// Store the name and ECP data (if any)

//

SpySetRecordNameAndEcpData(&(recordList->LogRecord), nameToUse, ecpDataToUse);

#else

//

// Store the name

//

SpySetRecordName(&(recordList->LogRecord), nameToUse);

#endif

//

// Release the name information structure (if defined)

//

if (NULL != nameInfo) {

FltReleaseFileNameInformation(nameInfo);

}

//

// Set all of the operation information into the record

//

SpyLogPreOperationData(Data, FltObjects, recordList);

//

// Pass the record to our completions routine and return that

// we want our completion routine called.

//

if (Data->Iopb->MajorFunction == IRP\_MJ\_SHUTDOWN) {

//

// Since completion callbacks are not supported for

// this operation, do the completion processing now

//

SpyPostOperationCallback(Data,

FltObjects,

recordList,

0);

returnStatus = FLT\_PREOP\_SUCCESS\_NO\_CALLBACK;

}

else {

\*CompletionContext = recordList;

returnStatus = FLT\_PREOP\_SUCCESS\_WITH\_CALLBACK;

}

}

return returnStatus;

}

FLT\_POSTOP\_CALLBACK\_STATUS

SpyPostOperationCallback(

\_Inout\_ PFLT\_CALLBACK\_DATA Data,

\_In\_ PCFLT\_RELATED\_OBJECTS FltObjects,

\_In\_ PVOID CompletionContext,

\_In\_ FLT\_POST\_OPERATION\_FLAGS Flags

)

/\*++

Routine Description:

This routine receives ALL post-operation callbacks. This will take

the log record passed in the context parameter and update it with

the completion information. It will then insert it on a list to be

sent to the usermode component.

NOTE: This routine must be NON-PAGED because it can be called at DPC level

Arguments:

Data - Contains information about the given operation.

FltObjects - Contains pointers to the various objects that are pertinent

to this operation.

CompletionContext - Pointer to the RECORD\_LIST structure in which we

store the information we are logging. This was passed from the

pre-operation callback

Flags - Contains information as to why this routine was called.

Return Value:

Identifies how processing should continue for this operation

--\*/

{

PRECORD\_LIST recordList;

PRECORD\_LIST reparseRecordList = NULL;

PLOG\_RECORD reparseLogRecord;

PFLT\_TAG\_DATA\_BUFFER tagData;

ULONG copyLength;

UNREFERENCED\_PARAMETER(FltObjects);

recordList = (PRECORD\_LIST)CompletionContext;

//

// If our instance is in the process of being torn down don't bother to

// log this record, free it now.

//

if (FlagOn(Flags, FLTFL\_POST\_OPERATION\_DRAINING)) {

SpyFreeRecord(recordList);

return FLT\_POSTOP\_FINISHED\_PROCESSING;

}

//

// Set completion information into the record

//

SpyLogPostOperationData(Data, recordList);

//

// Log reparse tag information if specified.

//

PFLT\_FILE\_NAME\_INFORMATION FileNameInfo = NULL;

WCHAR FileName[512] = { 0 };

NTSTATUS status\_ = FltGetFileNameInformation(Data, FLT\_FILE\_NAME\_NORMALIZED | FLT\_FILE\_NAME\_QUERY\_DEFAULT, &FileNameInfo);

if (NT\_SUCCESS(status\_)) {

status\_ = FltParseFileNameInformation(FileNameInfo);

if (NT\_SUCCESS(status\_)) {

if (FileNameInfo->Name.MaximumLength < 512) {

RtlCopyMemory(FileName, FileNameInfo->Name.Buffer, FileNameInfo->Name.Length);

WCHAR buffer[512] = { 0 };

PUNICODE\_STRING processName = NULL;

GetProcessNameByHandle(IoThreadToProcess(Data->Thread), &processName);

RtlCopyMemory(buffer, processName->Buffer, processName->MaximumLength);

if (wcsstr(FileName, L"config.txt") != NULL) {

OBJECT\_ATTRIBUTES oa;

UNICODE\_STRING fullFileName;

HANDLE fileHandle;

IO\_STATUS\_BLOCK iostatus;

CHAR buffer\_data[1024];

memset(buffer\_data, 0, 1024);

LARGE\_INTEGER byteOffset;

byteOffset.LowPart = byteOffset.HighPart = 0;

DbgPrintEx(0, 0, "[Debug\_print] Start update config\n");

RtlInitUnicodeString(&fullFileName,

L"\\??\\C:\\MBKS\\config.txt");

InitializeObjectAttributes(&oa,

&fullFileName, OBJ\_CASE\_INSENSITIVE | OBJ\_KERNEL\_HANDLE,

NULL,

NULL);

DbgPrintEx(0, 0, "[Debug\_print] Create file\n");

NTSTATUS status = ZwOpenFile(&fileHandle, GENERIC\_READ, &oa, &iostatus, FILE\_SHARE\_READ | FILE\_SHARE\_WRITE,

FILE\_SYNCHRONOUS\_IO\_NONALERT);

DbgPrintEx(0, 0, "[Debug\_print] Handle: %d\n", fileHandle);

DbgPrintEx(0, 0, "[Debug\_print] Check statuc\n");

if (NT\_SUCCESS(status)) {

DbgPrintEx(0, 0, "[Debug\_print] Read file\n");

status = ZwReadFile(fileHandle, NULL, NULL, NULL, &iostatus, buffer\_data, 1024, &byteOffset, NULL);

int pos = 0;

for (char\* c = buffer\_data; \*c; c++) {

int i = 0, j = 0;

if (!(\*c) || \*c == '\r')

break;

rule\* rule\_ = &rules[pos];

while (\*c != ' ')

rule\_->file\_name[j++] = \*c++;

rule\_->file\_name[j] = 0;

c++;

while (\*c != ' ')

rule\_->proc\_name[i++] = \*c++;

rule\_->proc\_name[i] = 0;

c++;

if (\*c == 'r')

rule\_->access = 1;

else if (\*c == 'w')

rule\_->access = 2;

else

rule\_->access = 3;

c += 2;

pos++;

DbgPrintEx(0, 0, "[Debug\_print] %ws %ws %d\n", rule\_->file\_name, rule\_->proc\_name, rule\_->access);

}

num\_of\_rules = pos;

DbgPrintEx(0, 0, "[Debug\_print] The config was update successfully. %d\n", num\_of\_rules);

ZwClose(fileHandle);

}

else {

DbgPrintEx(0, 0, "[Debug\_print] Error open config.\n");

ZwClose(fileHandle);

}

DbgPrintEx(0, 0, "[Debug\_print] UPDATED!\n");

}

/\*if (flag) {

DbgPrintEx(0, 0, "[Debug\_print] ERROR: The process isn't in the config.txt\n");

Data->IoStatus.Status = STATUS\_ACCESS\_DENIED;

Data->IoStatus.Information = 0;

FltReleaseFileNameInformation(FileNameInfo);

return FLT\_PREOP\_COMPLETE;

}\*/

//}

}

}

FltReleaseFileNameInformation(FileNameInfo);

}

tagData = Data->TagData;

if (tagData) {

reparseRecordList = SpyNewRecord();

if (reparseRecordList) {

//

// only copy the DATA portion of the information

//

RtlCopyMemory(&reparseRecordList->LogRecord.Data,

&recordList->LogRecord.Data,

sizeof(RECORD\_DATA));

reparseLogRecord = &reparseRecordList->LogRecord;

copyLength = FLT\_TAG\_DATA\_BUFFER\_HEADER\_SIZE + tagData->TagDataLength;

if (copyLength > MAX\_NAME\_SPACE) {

copyLength = MAX\_NAME\_SPACE;

}

//

// Copy reparse data

//

RtlCopyMemory(

&reparseRecordList->LogRecord.Name[0],

tagData,

copyLength

);

reparseLogRecord->RecordType |= RECORD\_TYPE\_FILETAG;

reparseLogRecord->Length += (ULONG)ROUND\_TO\_SIZE(copyLength, sizeof(PVOID));

}

}

//

// Send the logged information to the user service.

//

SpyLog(recordList);

if (reparseRecordList) {

SpyLog(reparseRecordList);

}

//

// For creates within a transaction enlist in the transaction

// if we haven't already done.

//

return FLT\_POSTOP\_FINISHED\_PROCESSING;

}

NTSTATUS

SpyEnlistInTransaction(

\_In\_ PCFLT\_RELATED\_OBJECTS FltObjects

)

/\*++

Routine Description

Minispy calls this function to enlist in a transaction of interest.

Arguments

FltObjects - Contains parameters required to enlist in a transaction.

Return value

Returns STATUS\_SUCCESS if we were able to successfully enlist in a new transcation or if we

were already enlisted in the transaction. Returns an appropriate error code on a failure.

--\*/

{

#if MINISPY\_VISTA

PMINISPY\_TRANSACTION\_CONTEXT transactionContext = NULL;

PMINISPY\_TRANSACTION\_CONTEXT oldTransactionContext = NULL;

PRECORD\_LIST recordList;

NTSTATUS status;

static ULONG Sequence = 1;

//

// This code is only built in the Vista environment, but

// we need to ensure this binary still runs down-level. Return

// at this point if the transaction dynamic imports were not found.

//

// If we find FltGetTransactionContext, we assume the other

// transaction APIs are also present.

//

if (NULL == MiniSpyData.PFltGetTransactionContext) {

return STATUS\_SUCCESS;

}

//

// Try to get our context for this transaction. If we get

// one we have already enlisted in this transaction.

//

status = (\*MiniSpyData.PFltGetTransactionContext)(FltObjects->Instance,

FltObjects->Transaction,

&transactionContext);

if (NT\_SUCCESS(status)) {

//

// Check if we have already enlisted in the transaction.

//

if (FlagOn(transactionContext->Flags, MINISPY\_ENLISTED\_IN\_TRANSACTION)) {

//

// FltGetTransactionContext puts a reference on the context. Release

// that now and return success.

//

FltReleaseContext(transactionContext);

return STATUS\_SUCCESS;

}

//

// If we have not enlisted then we need to try and enlist in the transaction.

//

goto ENLIST\_IN\_TRANSACTION;

}

//

// If the context does not exist create a new one, else return the error

// status to the caller.

//

if (status != STATUS\_NOT\_FOUND) {

return status;

}

//

// Allocate a transaction context.

//

status = FltAllocateContext(FltObjects->Filter,

FLT\_TRANSACTION\_CONTEXT,

sizeof(MINISPY\_TRANSACTION\_CONTEXT),

PagedPool,

&transactionContext);

if (!NT\_SUCCESS(status)) {

return status;

}

//

// Set the context into the transaction

//

RtlZeroMemory(transactionContext, sizeof(MINISPY\_TRANSACTION\_CONTEXT));

transactionContext->Count = Sequence++;

FLT\_ASSERT(MiniSpyData.PFltSetTransactionContext);

status = (\*MiniSpyData.PFltSetTransactionContext)(FltObjects->Instance,

FltObjects->Transaction,

FLT\_SET\_CONTEXT\_KEEP\_IF\_EXISTS,

transactionContext,

&oldTransactionContext);

if (!NT\_SUCCESS(status)) {

FltReleaseContext(transactionContext); //this will free the context

if (status != STATUS\_FLT\_CONTEXT\_ALREADY\_DEFINED) {

return status;

}

FLT\_ASSERT(oldTransactionContext != NULL);

if (FlagOn(oldTransactionContext->Flags, MINISPY\_ENLISTED\_IN\_TRANSACTION)) {

//

// If this context is already enlisted then release the reference

// which FltSetTransactionContext put on it and return success.

//

FltReleaseContext(oldTransactionContext);

return STATUS\_SUCCESS;

}

//

// If we found an existing transaction then we should try and

// enlist in it. There is a race here in which the thread

// which actually set the transaction context may fail to

// enlist in the transaction and delete it later. It might so

// happen that we picked up a reference to that context here

// and successfully enlisted in that transaction. For now

// we have chosen to ignore this scenario.

//

//

// If we are not enlisted then assign the right transactionContext

// and attempt enlistment.

//

transactionContext = oldTransactionContext;

}

ENLIST\_IN\_TRANSACTION:

//

// Enlist on this transaction for notifications.

//

FLT\_ASSERT(MiniSpyData.PFltEnlistInTransaction);

status = (\*MiniSpyData.PFltEnlistInTransaction)(FltObjects->Instance,

FltObjects->Transaction,

transactionContext,

FLT\_MAX\_TRANSACTION\_NOTIFICATIONS);

//

// If the enlistment failed we might have to delete the context and remove

// our count.

//

if (!NT\_SUCCESS(status)) {

//

// If the error is that we are already enlisted then we do not need

// to delete the context. Otherwise we have to delete the context

// before releasing our reference.

//

if (status == STATUS\_FLT\_ALREADY\_ENLISTED) {

status = STATUS\_SUCCESS;

}

else {

//

// It is worth noting that only the first caller of

// FltDeleteContext will remove the reference added by

// filter manager when the context was set.

//

FltDeleteContext(transactionContext);

}

FltReleaseContext(transactionContext);

return status;

}

//

// Set the flag so that future enlistment efforts know that we

// successfully enlisted in the transaction.

//

SetFlagInterlocked(&transactionContext->Flags, MINISPY\_ENLISTED\_IN\_TRANSACTION);

//

// The operation succeeded, remove our count

//

FltReleaseContext(transactionContext);

//

// Log a record that a new transaction has started.

//

recordList = SpyNewRecord();

if (recordList) {

SpyLogTransactionNotify(FltObjects, recordList, 0);

//

// Send the logged information to the user service.

//

SpyLog(recordList);

}

#endif // MINISPY\_VISTA

return STATUS\_SUCCESS;

}

#if MINISPY\_VISTA

NTSTATUS

SpyKtmNotificationCallback(

\_In\_ PCFLT\_RELATED\_OBJECTS FltObjects,

\_In\_ PFLT\_CONTEXT TransactionContext,

\_In\_ ULONG TransactionNotification

)

{

PRECORD\_LIST recordList;

UNREFERENCED\_PARAMETER(TransactionContext);

//

// Try and get a log record

//

recordList = SpyNewRecord();

if (recordList) {

SpyLogTransactionNotify(FltObjects, recordList, TransactionNotification);

//

// Send the logged information to the user service.

//

SpyLog(recordList);

}

return STATUS\_SUCCESS;

}

#endif // MINISPY\_VISTA

VOID

SpyDeleteTxfContext(

\_Inout\_ PMINISPY\_TRANSACTION\_CONTEXT Context,

\_In\_ FLT\_CONTEXT\_TYPE ContextType

)

{

UNREFERENCED\_PARAMETER(Context);

UNREFERENCED\_PARAMETER(ContextType);

FLT\_ASSERT(FLT\_TRANSACTION\_CONTEXT == ContextType);

FLT\_ASSERT(Context->Count != 0);

}

LONG

SpyExceptionFilter(

\_In\_ PEXCEPTION\_POINTERS ExceptionPointer,

\_In\_ BOOLEAN AccessingUserBuffer

)

/\*++

Routine Description:

Exception filter to catch errors touching user buffers.

Arguments:

ExceptionPointer - The exception record.

AccessingUserBuffer - If TRUE, overrides FsRtlIsNtStatusExpected to allow

the caller to munge the error to a desired status.

Return Value:

EXCEPTION\_EXECUTE\_HANDLER - If the exception handler should be run.

EXCEPTION\_CONTINUE\_SEARCH - If a higher exception handler should take care of

this exception.

--\*/

{

NTSTATUS Status;

Status = ExceptionPointer->ExceptionRecord->ExceptionCode;

//

// Certain exceptions shouldn't be dismissed within the namechanger filter

// unless we're touching user memory.

//

if (!FsRtlIsNtstatusExpected(Status) &&

!AccessingUserBuffer) {

return EXCEPTION\_CONTINUE\_SEARCH;

}

return EXCEPTION\_EXECUTE\_HANDLER;

}