A Data Loss Prevention prototype for Windows Platforms

Fotis A. Ailianos

M.Sc Computer Science AUEB

Athens, 2010

Advisors: George Xylomenos, George Polyzos

Contents

Acknowledgements

With this thesis I am actually walking at the end of my student life in Athens University of Economics and Business. After 8 years in AUEB a chapter is being closed. During those years being both an undergraduate student and graduate I have gained a lot of knowledge, experiences, friends and happy memories.

This work that is presented in this thesis is one of the most complex things I have ever done. I wanted my Master Thesis to be a great work, ideal for this closing chapter. I hope that I have succeeded my goals and satisfy my supervisors and colleagues through this attempt

Before starting, I would like to deeply thank to my supervisors Mr. George Polyzos and Mr. George Xylomenos for the trust they showed towards me and the freedom they provided in order to fulfill my work. During those 8 years they have been two great professors, advisors in all aspects of my academic career, mentors and a great source of inspiration. My way of thinking and working has been deeply affected and evolved thanks to them.

I would also like to give many thanks to my friend Pantelis Fragoudis. During those years Pantelis has been a great motivator, a source of inspiration and a great friend. He was always been supportive and helpful. It was always fun working with him and together we have completed some of the greatest projects I have ever had.

Last but not least I would like to thank my friend Commander Spyros Papageorgiou of the Cyber Defense directorate for providing me with great ideas as far as the security issues are concerned. Without him I doubt if this project could have been even started. The cooperation with him the last three years gave me an uncountable amount of experiences and knowledge in the field of security. I deeply thank him for his trust on my capabilities.

1. Introduction

1.1 Motivation

One of the biggest problems a modern organization has to face is the prevention of data loss. Data loss may be due to a plethora of reasons, some of them are:

- Trojans
- System failures
- Users not complying with the predefined security measures

We must pay special attention to the last bullet. Users usually ignore the security policy. For example, they copy classified files to their USB thumb drivers and take their work at home. The problem is that the organization may be perfectly safe as far as their systems are concerned, but from the time the data leave the building there is no guaranteed security at all. Users can plug the thumb drive into their infected home workstation and then the problems begin. Trojans are also a big problem, as hackers can easily remove data from infected workstations. Organizations for that reason use Data Loss Prevention solutions (DLPs) which work just like modern Intrusion Prevention Systems. DLPs monitor the network and hosts and try to identify a potential data loss. These are usually highly expensive and complex solutions. In our work we tried to create a simple working DLP prototype based on simple hacks in order to prove that with simple technical steps our goal can be fulfilled. In the following sections of this text we are going to describe the basic Microsoft Windows driver framework that we used in order to create our driver, the various driver functions, the potential attacks against our system and finally the conclusions.

1.2 Overview of the system

In this section we will briefly present our driver. It monitors a predefined list of file types, for example DOC, PDF, TXT files etc. If an I/O call happens in the system the driver kicks in and checks whether this file is marked or not. By the term mark we refer to the classification attribute of the file *"classified"* or *"non-classified"*. The next step is to determine the I/O type of the call; was it a read or write or rename? Finally it decides whether the call must be completed successfully or canceled.

Some of the rules by which our driver was coded are the following. Firstly, copying data from a local volume to a USB removable volume is strictly prohibited. We have implemented a mechanism that detects the insertion of a USB removable volume. The driver recognizes the insertion and automatically attaches itself to the newly mounted volume. Secondly, no browser application or other application irrelevant to the specific marked file is allowed to access the file. By this rule we block Trojans or other factors that can lead to data leaks in the Internet. Thirdly all files have to be marked; we have enforced a policy that does not allow the user to save a file unless he puts a mark in that file. We have implemented a user space application that a user can use in order to mark files

Apart from the rules, we have given a lot of emphasis in the driver performance and, therefore, in the extra overhead that could affect overall system performance. Clever data structures enhance the driver performance removing the extra stress from the system.

Although our driver is resilient to the majority of attacks, hackers will surely be able to outsmart our implementation and defeat the driver protection. For this reason we created a logging mechanism that logs all actions regarding the monitored files. In case of a data leak an experienced forensics expert can easily find the source and the cause of the data leak which is equally important with prevention.

In the next chapter we are going to describe the Windows driver model and provide more details about the theory behind our driver.

2. Windows Driver Model

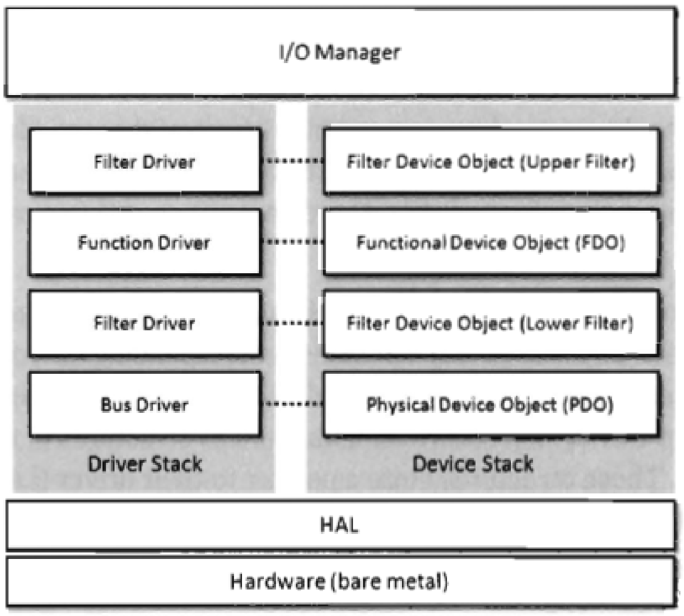
2.1 Drivers

While it is possible for all of the I/O services required by a peripheral device to be instantiated in terms of a single driver, this isn't always the case. Physical devices are sometimes serviced by a whole chain of drivers, which is referred to as a *driver stack.*

Each driver in the stack performs part of the work necessary to support a given I/O operation. The underlying motivation for this layering approach is based on the desire to reduce the amount of redundant code by instituting a division of labor. Several peripheral devices may share the same device controller and underlying I/O bus. Rather than reimplement an identical framework of code for each device driver (which would be a huge waste of time and energy), it makes sense to modularize functionality into distinct components to maximize the opportunity for reuse. This scheme is reflected by the fact that there are three basic types of kernel-mode drivers in the classic Windows driver model (WDM):

- Function drivers
- Bus drivers
- Filter drivers

Function drivers take center stage as the primary drivers for a given device; which is to say that they perform most of the work necessary to service I/O requests. These drivers take the Windows API calls made by a client application and translate them into a discrete series of I/O commands that can then be issued to a device. In a driver chain, function drivers typically reside somewhere in the middle, layered between filter drivers and bus drivers.


I/O Manager
Filter Driver
Filter Device Object (Upper Filter)
Function Driver
Functional Device Object (FDO)
Filter Driver
Filter Device Object (Lower Filter)
Bus Driver
Physical Device Object (PDO)
Driver Stack
Device Stack
HAL
Hardware (bare metal)

Bus drivers implement functionality that is specific to a particular hardware interface (e.g., the USB bus, PCI bus, or SCSI bus). They are essentially low-level function drivers for a particular system bus. Bus drivers always reside at the bottom of the driver stack, where they enumerate and then manage the devices attached to the bus.

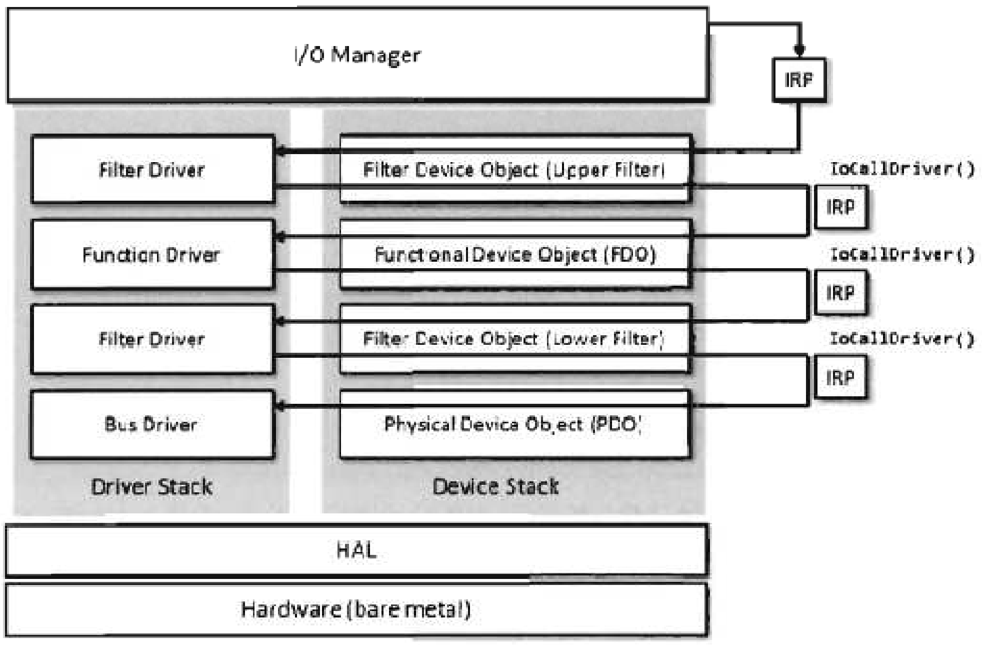
Filter drivers don't manage hardware per se. Instead, they intercept and modify information as it passes through them. For example, filter drivers could be used to encrypt data written to a storage device and also decrypt data read from the storage device. Filter devices can be categorized as upper filters or lower filters, depending upon their position relative to the function driver. Function drivers and bus drivers are often implemented in terms of a driver/minidriver pair. In practice, this can be a class/miniclass driver pair or a port/mini port driver pair.

A *class* driver offers hardware-agnostic support for operations on a certain type (e.g., a certain class) of device. Windows ships with a number of class drivers, like the kbdclass. sys driver that provides support for keyboards. A *miniclass* driver, on the other hand, is usually supplied by a vendor. It supports device-specific operations for a particular device of a given class.

A *port* driver supports general I/O operations for a given peripheral hardware interface. Because the core functionality of these drivers is mandated by the OS, Windows ships with a variety of port drivers. For example, the iSEl42prt.sys port driver services the 8042 microcontroller used to connect PS/2 keyboards to the system's peripheral bus. *Miniport* drivers, like miniclass drivers, tend to be supplied by hardware vendors. They support device-specific operations for peripheral hardware connected to a particular port.

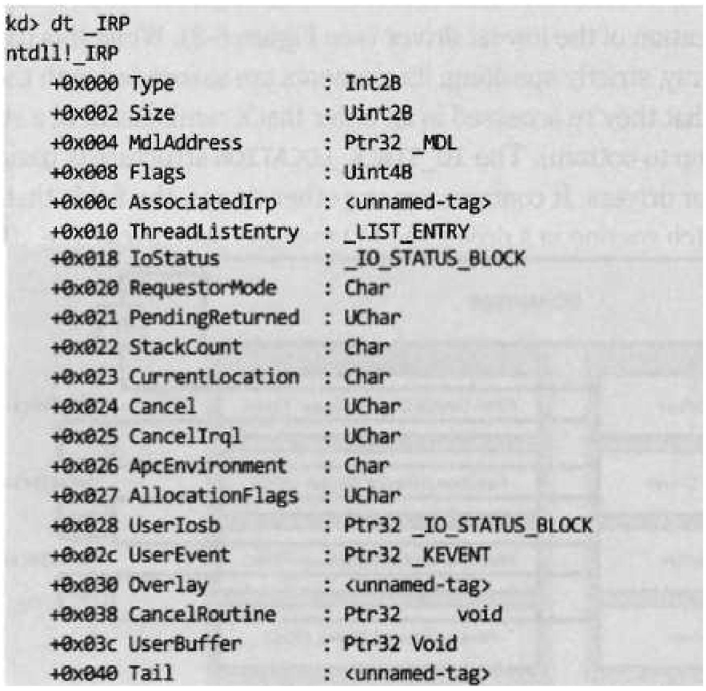
2.2 The I/O Request Packet (IRP)

The drivers in a driver stack pass information down through the stack by the use of IRPs. The fun begins when a client application sends an I/O request which the Windows I/O manager formally packages as an IRP. The I/O manager locates the device object at the top of the device object stack and uses it to route the IRP to the appropriate dispatch routine in the top device's driver. If the top driver can service the request by itself, it completes the I/O request and returns the IRP to the I/O manager. The exact nature of IRP "completion" will be discussed later. For now, just accept that the process of completing the I/O request means that the driver stack did what it was asked to do (or at least attempted to do so).



If the top driver cannot service the request by itself, it does what it can and then locates the device object associated with the next lowest driver. Then the top driver asks the I/O manager to forward the IRP to the next lowest driver via a call to IoCallDriver ( ). This series of steps repeats itself for the next driver, and in this fashion, IRPs can make their way from the top of the driver stack to the bottom. Note that if an IRP actually reaches the driver at the very bottom of the driver stack, it will have to be completed there (there's nowhere else for it go).

To delve deeper into exactly how layered drivers work, we'll have to get a closer look at the structure of IRPs. The official WDK docs describe the IRP structure as being "partially opaque," which is a nice way of them saying that they're not going to tell you everything. When this happens, your instinctive response should be to whip out a kernel debugger and dump the type description of the object. In this case, we end up with:



```
kd> dt _IRP
ntdll!_IRP
   +0x000 Type             : Int2B
   +0x002 Size             : Uint2B
   +0x004 MdlAddress       : Ptr32 _MDL
   +0x008 Flags            : Uint4B
   +0x00c AssociatedIrp    : <unnamed-tag>
   +0x010 ThreadListEntry  : _LIST_ENTRY
   +0x018 IoStatus         : _IO_STATUS_BLOCK
   +0x020 RequestorMode    : Char
   +0x021 PendingReturned  : UChar
   +0x022 StackCount       : Char
   +0x023 CurrentLocation  : Char
   +0x024 Cancel           : UChar
   +0x025 CancelIrql       : UChar
   +0x026 ApcEnvironment   : Char
   +0x027 AllocationFlags  : UChar
   +0x028 UserIosb         : Ptr32 _IO_STATUS_BLOCK
   +0x02c UserEvent        : Ptr32 _KEVENT
   +0x030 Overlay          : <unnamed-tag>
   +0x038 CancelRoutine    : Ptr32     void
   +0x03c UserBuffer       : Ptr32 Void
   +0x040 Tail             : <unnamed-tag>
```

Looking at these fields, all you really need to notice for the time being is that they form a fixed-size header that's used by the I/O manager to store metadata about an I/O request. Think of the previous dump of structure fields as constituting an *IRP header.* If you want to know more about a particular field in an IRP, see the in-code documentation for the IRP structure in the WDK's wdm. h header file.

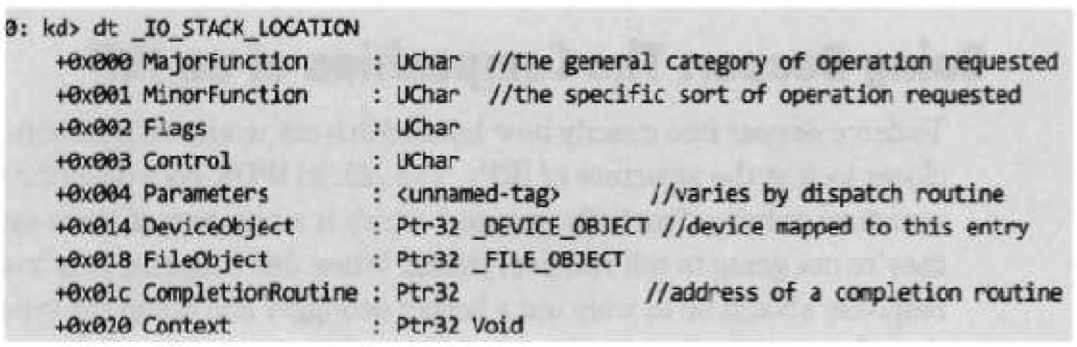
The most important field of the IRP structure is the type. Each IRP that the I/O manager passes down is assigned a major function code of the form IRP MJ _xxx. These codes tell the driver what sort of operation it should perform to satisfy the I/O request. The list of all possible major function codes is defined in the WDK's wdm. h header file. The three most common types of IRPs are:

• IRP MJ_ READ

• IRP MJ WRITE

• IRP MJ_CREATE

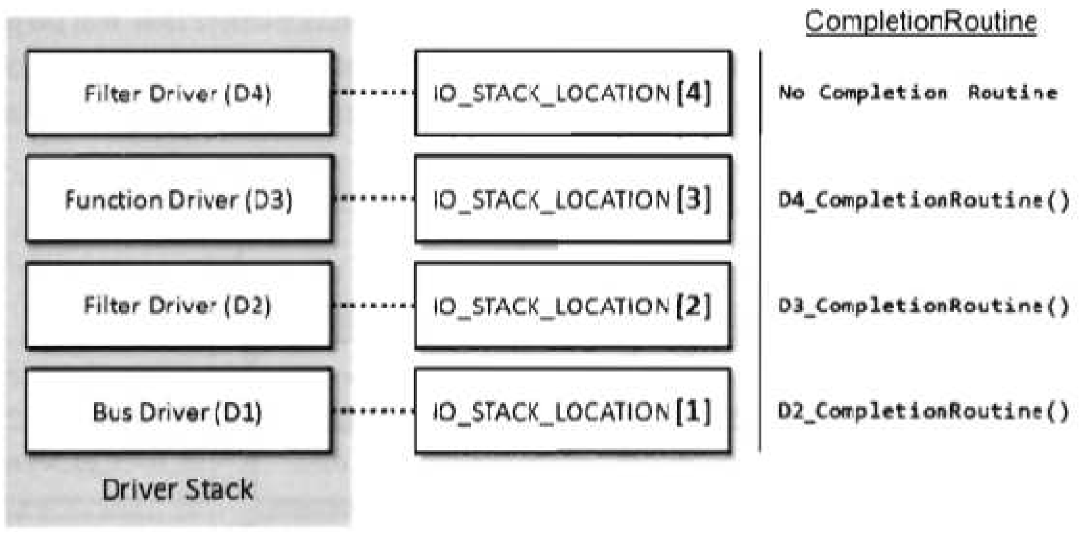
Our driver is based on the above IRP types. One can easily understand that these types of IRPs refer to read, write and file create I/O calls. The driver structure contains an internal array called the *MajorFunction* array. This array contains a pointer to a function for each IRP type existing in the system. These functions referenced by the MajorFunction array are known as *dispatch routines.* When the I/O manager creates an IRP, it allocates additional storage space just beyond the header for each driver in a device's driver stack. When the I/O manager requisitions this storage space, it knows exactly how many drivers are in the stack and this allows it to set aside just enough space. The I/O manager breaks this storage space into an array of structures, where each driver in the driver stack is assigned an instance of the IO_STACK_LOCATION structure:



```
0: kd> dt _IO_STACK_LOCATION
    +0x000 MajorFunction     : UChar  //the general category of operation requested
    +0x001 MinorFunction     : UChar  //the specific sort of operation requested
    +0x002 Flags             : UChar
    +0x003 Control           : UChar
    +0x004 Parameters        : <unnamed-tag>      //varies by dispatch routine
    +0x014 DeviceObject      : Ptr32 _DEVICE_OBJECT //device mapped to this entry
    +0x018 FileObject        : Ptr32 _FILE_OBJECT
    +0x01c CompletionRoutine : Ptr32               //address of a completion routine
    +0x020 Context           : Ptr32 Void
```

An IRP's array of stack locations is indexed starting at 1, which is mapped to the stack location of the lowest driver. While this data structure is, strictly speaking, an array, its elements are associated with the driver stack such that they're accessed in an order that's reminiscent of a stack (e.g., from top to bottom). The IO_STACK_LOCATION structure is basically a cubbyhole for drivers. It contains, among other things, the fields that dictate which dispatch routine in a driver the I/O manager will invoke (i.e., the major and minor IRP function codes) and also the information that will be passed to the driver's dispatch routine (e.g., the Parameters union, whose content varies according to the major and minor function codes). It also has a pointer to the device object that it's associated with.

From the vantage point of the sections that follow, the most salient field in the IO_STACK_LOCATION structure is the CompletionRoutine pointer. This pointer field references a function that resides in the driver *directly above* the driver to which the stack location is assigned. This is an important point to keep in mind, and you'll see why shortly. For instance, when a driver registers a completion routine with an IRP, it does so by storing the address of its completion routine in the stack location allocated for the driver below it on the driver stack. For example, if the lower filter driver is going to register its completion routine with the IRP, it will do so by storing the address of this routine in the stack location allocated to the bus driver. Later on when we describe the IRP path in the system we are going to describe WHY we do need completion routines.



2.3 IRP Path

When a driver's dispatch routine first receives an IRP, it will usually retrieve parameter values from its I/O stack location (and anything else that might be stored there) by making a call to the IoGetCurrentIrpStackLocation() routine. Once this is done, the dispatch routine is free to go ahead and do whatever it was designed to do. Near the end of its lifespan, if the dispatch routine plans on forwarding the IRP to the next lowest driver on the chain, it must:

1. Set up the I/O stack location in the IRP for the next driver lower down in the stack.
2. Register a completion routine (this step is optional).
3. Send off the IRP to the next driver below it.
4. Return a status code (NTSTATUS).

There are a couple of standard ways to set up the stack location for the next IRP. If you're not using the current stack location for anything special and you'd like to simply pass this stack location on to the next driver, use the following routine:

VOID IoSkipCurrentIrpStackLocation(IN PIRP Irp);

This routine produces the desired effect by decrementing the I/O manager's pointer to the IO_STACK_LOCATION array by 1. This way, when the IRP gets forwarded and the aforementioned array pointer is incremented, the net effect is that the array pointer is unchanged. The driver below the current one gets the exact same IO_STACK_LOCATION element as the current driver. Naturally, this means that there will be an I/O stack location that doesn't get utilized because you're essentially sharing an array element between two drivers. This is not a big deal. Too much is always better than not enough. If you want to copy the contents of the current I/O stack element into the next element, with the exception of the completion routine pointer, use the following routine:

VOID IoCopyCurrentIrpStacklocationToNext(IN PIRP Irp)

Registering a completion routine is as easy as invoking the following:

```
VOID IoSetCompletionRoutine
(
        IN PIRP Irp, //pointer to the IRP
        IN PIO_COMPlETION_ROUTINE CompletionRoutine, //completion routine address
        IN PVOID Context, //basically whatever you want
        IN BOOLEAN InvokeOnSuccess,
        IN BOOLEAN InvokeOnError,
        IN BOOLEAN InvokeOnCancel
)
```

The last three Boolean arguments determine under what circumstances the completion routine will be invoked. Most of the time, all three parameters are set to TRUE. Actually firing off the IRP to the next driver is done by invoking the following:

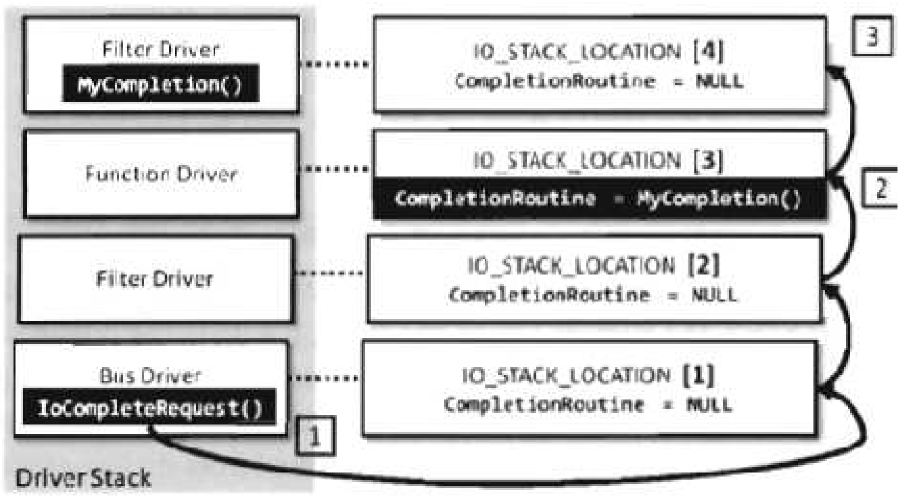
NTSTATUS IoCallDriver(IN POEVICE_OBJECT DeviceObject, IN OUT PIRP Irp)

The first argument accepts a pointer to the device object corresponding to the driver below the current one on the stack. It's up to the dispatch routine to somehow get its hands on this address. There's no standard technique to do so. Most of the time, the NTSTATUS value that a forwarding dispatch routine will return is simply the value that's returned from its invocation of IoCallDriver( ).

2.4 IRP Completion

An IRP cannot be forwarded forever onward. Eventually, it must be completed. It's in the nature of an IRP to seek completion. If an IRP reaches the lowest driver on the stack then it *must* be completed by that driver because it literally has nowhere else to go. On an intuitive level, IRP completion infers that the driver stack has finished its intended I/O operation. For better or for worse, the I/O request is done. From a technical standpoint there's a bit more to this. Specifically, the I/O manager initiates the completion process for an IRP when one of the drivers processing the IRP invokes the IoCompleteRequest () function. During this call, it's assumed that both the Status and Information fields of the IRP's IO_STATUS_BLOCK structure have been initialized. Also, the second argument (the value assigned to the function's PriorityBoost parameter) is almost always set to IO_NO_INCREMENT.

Via the implementation of IoCompleteRequest ( ), the I/O manager then takes things from here. Starting with the current driver's I/O stack location, it begins looking for completion routines to execute. In particular, the I/O manager checks the current stack location to see if the previous driver registered a completion routine. If a routine has not been registered, the I/O manager moves up to the next IO_STACK_LOCATION element and continues the process until it hits the top of the I/O stack array. If the I/O manager does encounter a valid completion routine address, it executes the routine and then keeps moving up the stack so long as the completion routine doesn't return the STATUS_MORE_PROCESSING_REQUIRED status code.



To sum up we have to note that an IRP, the most important element in the system regarding I/O calls, has a two way path. First it descends through the I/O stack. Throughout its descent it stops only at those drivers inside the stack that have a registered dispatch routine. Inside the dispatch routines the driver registers the callback routines. After the end of the path IRP is completed by the IoCompleteRequest and then ascends the stack in reverse. This time the IRP stops only to the drivers of the stack that have registered a completion routine.

3. Implementation

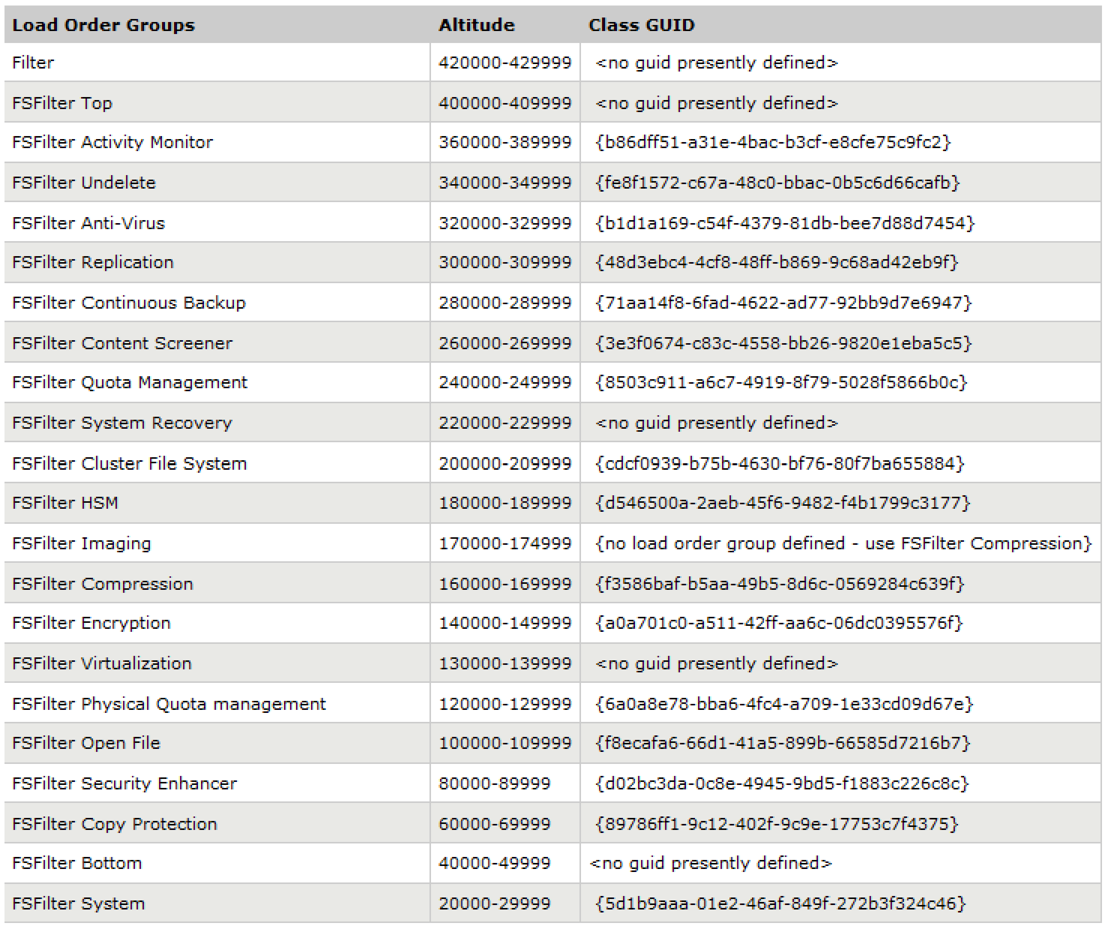
3.1 The driver

In this section we are going to describe the basic components of our driver and the initialization process. As we already mentioned in the introduction, our driver was coded based on the Microsoft mini filter driver. We are going to present the basic steps of our driver initialization process and give details about each step

- List existing volume drives
- Register the mini filter component
- Register which IRP packets our driver is going to monitor
- Register plug ‘n play callback
- Register user space communication port and start listener
- Register teardown and driver shutdown functions
- Start the driver loop
- Attach the driver into discovered modules

The first task in order to initialize our driver is to discover all existing volumes. Consider a system with many volume drives, like C:\, D:\, E:\; the driver must be attached to each one of the discovered volumes in order to monitor the files they contain. During the volume discovery process, our driver must also determine whether a found volume is a removable USB device or not. We have already mentioned the value of that information: it is necessary in order to prevent protected files from being copied to such devices.

The next important step is the registration of the driver into the system. First of all we have to set the driver’s altitude in the system. Every file system filter driver developed to the Filter Manager model (a minifilter) must have a unique identifier called an altitude. The altitude of a minifilter defines its position relative to other minifilters in the I/O stack when the minifilter is loaded. The altitude is an infinite-precision string interpreted as a decimal number. A minifilter that has a low numerical altitude is loaded below a minifilter that has a higher numerical value in the I/O stack. Each load-order group has a defined range of altitudes. Allocation of altitudes to minifilters is managed by Microsoft. In other words, the altitude defines firstly the loading order of the various drivers and secondly their priority during the system calls. Below we provide a table of the available altitudes.



Load Order Groups	Altitude	Class GUID
Filter	420000-429999	<no guid presently defined>
FSFilter Top	400000-409999	<no guid presently defined>
FSFilter Activity Monitor	360000-389999	{b86dff51-a31e-4bac-b3cf-e8cfe75c9fc2}
FSFilter Undelete	340000-349999	{fe8f1572-c67a-48c0-bbac-0b5c6d66cafb}
FSFilter Anti-Virus	320000-329999	{b1d1a169-c54f-4379-81db-bee7d88d7454}
FSFilter Replication	300000-309999	{48d3ebc4-4cf8-48ff-b869-9c68ad42eb9f}
FSFilter Continuous Backup	280000-289999	{71aa14f8-6fad-4622-ad77-92bb9d7e6947}
FSFilter Content Screener	260000-269999	{3e3f0674-c83c-4558-bb26-9820e1eba5c5}
FSFilter Quota Management	240000-249999	{8503c911-a6c7-4919-8f79-5028f5866b0c}
FSFilter System Recovery	220000-229999	<no guid presently defined>
FSFilter Cluster File System	200000-209999	{cdcf0939-b75b-4630-bf76-80f7ba655884}
FSFilter HSM	180000-189999	{d546500a-2aeb-45f6-9482-f4b1799c3177}
FSFilter Imaging	170000-174999	{no load order group defined - use FSFilter Compression}
FSFilter Compression	160000-169999	{f3586baf-b5aa-49b5-8d6c-0569284c639f}
FSFilter Encryption	140000-149999	{a0a701c0-a511-42ff-aa6c-06dc0395576f}
FSFilter Virtualization	130000-139999	<no guid presently defined>
FSFilter Physical Quota management	120000-129999	{6a0a8e78-bba6-4fc4-a709-1e33cd09d67e}
FSFilter Open File	100000-109999	{f8ecafa6-66d1-41a5-899b-66585d7216b7}
FSFilter Security Enhancer	80000-89999	{d02bc3da-0c8e-4945-9bd5-f1883c226c8c}
FSFilter Copy Protection	60000-69999	{89786ff1-9c12-402f-9c9e-17753c7f4375}
FSFilter Bottom	40000-49999	<no guid presently defined>
FSFilter System	20000-29999	{5d1b9aaa-01e2-46af-849f-272b3f324c46}

We have thoroughly described IRP system handling during the first chapter of this text. We mentioned two important components of the IRP lifetime, the dispatch routines and the completion routines. Microsoft has created the Minifilter driver framework that significantly reduces the complexity of the needed code in order to build filter drivers. Specifically, that framework uses only 3 functions: the registration function initializing the driver and two callback functions, the Pre-Operation callback and the Post Operation. Pre operations callback and Post operations callback is the direct analogue for dispatch routine and completion routine respectively. Windows is an event driven operation system. The registration of the callback routines means that whenever the right event occurs in the system the Pre or Post operation callback kicks in. One can easily find similarities of the Windows callback functions with the Java action event listener. In other words, when a specific IRP type for example IRP_MJ_READ occurs after the dispatch function is called, the PreOperation callback is automatically being executed. The same thing happens for the PostOperation callback and the completion routine.

In order for a driver to be registered, one must call the FltRegisterFilter() takes as a parameter a FLT_REGISTRATION structure, which contains an unload routine, instance notification callbacks, a list of the callback function pointers and a list of file system operation callback pointers. This structure is the key for all filter driver operations. As far as the callback functions are concerned we provide the function prototypes.

```
typedef FLT_PREOP_CALLBACK_STATUS (*PFLT_PRE_OPERATION_CALLBACK)
(
       IN OUT PFLT_CALLBACK_DATA Data,
       IN PCFLT_RELATED_OBJECTS FltObjects,
       OUT PVOID *CompletionContext
);
typedef FLT_POSTOP_CALLBACK_STATUS (*PFLT_POST_OPERATION_CALLBACK)
(
       IN OUT PFLT_CALLBACK_DATA Data,
       IN PCFLT_RELATED_OBJECTS FltObjects,
       IN PVOID CompletionContext,
       IN FLT_POST_OPERATION_FLAGS Flags
);
```

On both callback functions we can see an FLT_CALLBACK_DATA structure parameter a FLT_REGISTRATION structure, which contains all the needed information regarding the IRP, the I/O call, the results, even the file name or objects that is being accessed.

The next important step during our driver initialization is the callback registration for the Plug'n Play messages. Plug 'n Play messages are system messages that are broadcasted inside the system whenever a device is inserted in one of the peripheral ports (VGA, USB, Serial port etc). Those messages are useful for automatic device installation but also inform all system components of the new hardware changes. In order for our driver to listen to Plug 'n Play messages the *IoRegisterPlugPlayNotification* call must be used. The **IoRegisterPlugPlayNotification** routine registers a driver callback routine to be called when a PnP event of the specified category occurs.

```
NTSTATUS IoRegisterPlugPlayNotification(
  __in      IO_NOTIFICATION_EVENT_CATEGORY EventCategory,
  __in      ULONG EventCategoryFlags,
  __in_opt  PVOID EventCategoryData,
  __in      PDRIVER_OBJECT DriverObject,
  __in      PDRIVER_NOTIFICATION_CALLBACK_ROUTINE CallbackRoutine,
  __in_opt  PVOID Context,
  __out     PVOID *NotificationEntry);
```

The most important arguments of the above function are the EventCategoryFlags and the EventCategoryData. The first one must be set to EventCategoryDeviceInterfaceChange. PnP events in this category include the arrival (enabling) of a new instance of a device interface class (GUID_DEVICE_INTERFACE_ARRIVAL), or the removal (disabling) of an existing device interface instance (GUID_DEVICE_INTERFACE_REMOVAL). The other important flag is the EventCategoryData which must be set to GUID_DEVINTERFACE_PARTITION. Drivers register an instance of the GUID_DEVINTERAFACE_PARTITION for a partition that is a child of device of a storage device. All the above simply means that whenever a Plug ‘n Play message arrives regarding a storage partition volume arrival the callback function will be called.

The next step is the installation of a user space communication port. As we will describe in the last chapter our driver needs to exchange messages with the user space. This way we call the *FltCreateCommunicationPort* which is the *socket()* of libc direct analogue. There are is also the *FltSendMessage* for sending messages. In the other direction, the user space application can connect to the driver by calling the *FilterConnectCommunication* port and there are also the *FilterSendMessage*, *FilterReplyMessage* and *FilterGetMessage* for sending and receiving messages to the kernel.

One more step is to register the driver exit routines and the teardown routines. In these routines we have to unregister all the aforementioned callbacks, free the buffers and stop the filter loop, otherwise the system will crash and will not be able to shutdown or release its resources. For example a USB volume device cannot be unmounted if the teardown function does not exist. Tear down differs to driver exit because teardown can be called in the following cases

- Driver being unloaded
- Volume being dismounted
- Driver being detached from volume by FltDetachVolume function

The final step of our driver initialization is the volume attach and the start of the main driver loop. The driver has to be attached specifically in each volume. This can be done by the use of FltAttachVolume. So for each of the discovered volumes plus for each newly appearing volume we have to call FltAttachVolume. That is not enough: the filter driver main loop must also be started. The main loop is no loop at all, it just engages the event listener we talked about previously and calls the callbacks whenever this is needed. This can be done by the FltStartFiltering

3.2 Monitoring and action rules

In this section we are going to describe what operating system functions our driver monitors, why we chose that approach and what are the advantages or drawbacks of those decisions. We must mention again that our driver first of all monitors all I/O calls regarding the marked files and secondly kicks in and allows or bans a specific action based on our rules. Although our driver has a very high percentage of efficiency against attacks it cannot be considered as bullet proof under no circumstances. There will always be hackers or conditions that can outsmart our driver and manage to leak data. Therefore we have designed our system for 100% detailed event recording in order to make sure that even if a leak occurs, administrators can quickly find why did the leak happened and who is responsible for this. If our driver did not manage to stop the leak maybe the logs taken will be enough to stop future leaks. We are going now to present the various monitor types

File type monitoring

First of all, which files are going to be monitored? Text documents have to be monitored; those are generally *PDF, DOC, TXT, RTF* files. All other files are being excluded from the monitor process mainly because of performance issues. A system has a significant number of file types, if we monitor each and every one of them, the performance of the system is going to degrade. One could argue that by monitoring only specific files, one can easily escape from our driver monitoring by simply renaming the file extension; for example from .DOC to .EXE. We will analyze further that attack in the next sections, but our driver is capable of monitoring the actions regarding file renaming and does not allow file extension changes of a marked file. The list of the monitored file types can easily be modified based on each organization's needs.

Process monitoring

The second stage of our monitoring is to check which process is accessing the monitored files. Our data leak prevention policy specifically describes that only appropriate processes can monitor those files. For example DOC files can be handled only by MS Office applications, PDFs by Adobe products etc. Any other process trying to access marked files is not allowed to do so. For example one of the main causes of data leakage is sending classified data through the internet, usually by emails. Our driver monitors various web browser of email software like FireFox, Internet Explorer, Outlook etc in order to prevent file leakage on the internet. If a user tries to send an email with a marked file attached the driver will automatically block and record that action. Of course things are not that simple because many system processes need to access those files, for example explorer.exe or some antivirus software. Note that explorer.exe is the main system process that browses files. If we block explorer.exe from accessing our secured files then those files will not be listed making the system unusable. During our driver deployment those processes must be set by the system administrator after careful planning. For the time being we explicitly demand that the explorer.exe process must and has to access any file in the system.

At this point we are going to describe some other problems that can occur regarding the process monitor criterion. First of all modern trojans hide themselves by a method called “DLL Injection”. Each process has each own virtual space where its code and data is stored. This is also where the Dynamic-Link libraries (DLLs) reside. DLLs are libraries that are dynamically called whenever they are needed. Imagine now that a virus is coded as a DLL. If an attacker manages to import that DLL into a target process then all malicious actions of that DLL will be performed inside the target process. For example if an attacker injects a virus DLL inside explorer.exe actually hides the virus inside a benign process and the system cannot directly recognize whether this process is allowed to perform those (malicious) actions or not. After some research we deal with DLL injects by simply monitoring the Thread ID number of explorer.exe that tries to access a file. The system, each time it boots, spawns a specific Thread ID inside explorer.exe that is used for file browsing. On the other hand in order for a DLL injection attack to complete, a new thread must be spawned inside explorer.exe, thus a different Thread ID is going to appear trying to access a file.

Therefore we monitor that Thread ID changes and act accordingly. We have to mention that this is a highly strict policy. There are cases, although rare ones that the system restarts explorer.exe or creates a new thread, so the administrator of our system must carefully decide whether to apply this policy or not.

<u>The confused deputy case</u>

Another problem regarding process monitoring is the process renaming attack. For example we mentioned that Word (winword.exe) is a process that can access a DOC file. What if an attacker installs a process named winword.exe? Our driver will not be able to distinguish between the benign process and the malicious one. Although this attack is outside the scope of our work, we are going to briefly describe this issue. This problem is known as the *Confused Deputy* problem. We are going to provide an example that is a direct analogue to our problem.

In the original example of a confused deputy, there is a program that provides compilation services to other programs. Normally, the client program specifies the name of the input and output files, and the server is given the same access to those files that the client has. The compiler service is pay-per-use, and the compiler program has access to a file (dubbed *BILL*) where it stores billing information. Clients obviously cannot write into the billing file. Now suppose a client calls the service and specifies *BILL* as the name of the output file. The service opens the output file. Even though the client did not have access to that file, the service does, so the open succeeds, and the server writes the compilation output to the file, overwriting it, and thus destroying the billing information. The deputy is our driver and the output file of the compiler is the process name. The solution to the confused deputy problem is the use of *"object capabilities".* The object-capability model is a computer security model based on the Actor model of computation. The name "object-capability model" is due to the idea that the capability to perform an operation can be obtained by the following combination:

- an unforgeable reference (in the sense of object references or protected pointers) that can be sent in messages.
- a message that specifies the operation to be performed.

An example of a possible capability would be to look for a digital signature inside the binary of the process reading the file. Although this solution can work fine it is not always sure that a benign process comes signed. However, usually that is the case and it is actually the case for all the tested document platforms, therefore we consider this a working and acceptable solution

Monitor USB thumb drives

We have defined the files that are monitored and the processes that can access files. The above defenses in general prevent malware and Trojans specifically from exfiltrating files. The driver blocks users from sending data deliberately on the internet. Apart from those causes of data leakage there is one other very common one. Users have a very bad habit, to copy their files into their USB mass storage devices. At first this may seem quite innocent but it is not. Even if we consider user's work environment as a perfectly safe environment as far as the systems are concerned we cannot say the same for the user's personal systems. There is no guarantee that the user will not insert his USB thumb drive into an infected system. For this reason, some organizations strictly prohibit the use of USB thumb drives and sometimes even remove USB thumb drive support from their system.

In order to protect data from these kinds of user actions our driver listens for any Plug 'n Play messages and if those messages refer to a Removable Volume Device (a Microsoft term for USB thumb drives) then our driver is automatically attached to that volume and monitors the newly mounted volume. Plug 'n Play messages are system messages that are broadcasted inside the system whenever a device is inserted in one of the peripheral ports (VGA, USB, Serial port etc). Those messages are useful for automatic device installation but also inform all system components for the new hardware changes. The question is why monitor the new volume. As we have already mentioned in our introduction, we strictly block every I/O call targeting a USB removable volume. Thus any copy, cut – paste, move, or drag 'n drop action that occurs from a local drive to a removable volume is strictly prohibited. This way we are sure that no file is going to be stored one way or another to a USB disk.

We have described in detail the registration of the required plug 'n play callbacks in the Driver initialization section. A potential attack here would be to open an new document file copy paste the contents of the classified file (not the file itself) and save the file directly to the USB drive. As we will describe later this attack is easily prevented thanks to our file marking policy.

Network sharing monitor

Another potential threat for data is data exfiltration through network shares. As it is widely known Windows provide a network file sharing services called SMB (Service Message Block). A user having access to a remote computer can easily copy files through a provided "Network Shared Folder". Our driver is capable of blocking those kinds of actions if they are detected. We achieve the above by looking at the accessed file Token Descriptors. If we detect the existence of a *TokenImpersonation* token, then we are sure that a network service is about to access a marked file and therefore our driver will cancel that operation. But first of all, what is a token? Each file is accessed by the system through a file descriptor. That specific file descriptor has a lot of attributes and one of them is the access token. An *access token* is an object that describes the *security context* of a *process* or thread or object. The information in a token includes the identity and privileges of the user account associated with the process or thread. When a user logs on, the system verifies the user's password by comparing it with information stored in a security database. If the password is *authenticated*, the system produces an access token.

Every process executed on behalf of this user has a copy of this access token. Every process has a primary token that describes the security context of the user account associated with the process. By default, the system uses the primary token when a thread of the process interacts with a securable object. Moreover, a thread can impersonate a client account. Impersonation allows the thread to interact with securable objects using the client's security context. A thread that is impersonating a client has both a primary token and an *impersonation token*. Network operations on Windows are actually being impersonated by the system. During network file access the one that actually initiates the process is the system. There is no such account as "system" so the operating system impersonates itself as the logged on user. There is another advantage with impersonations, system has fully privileges; by impersonating another user, the system drops its privileges in order to prevent unauthorized access.

3.3 Technical details

In this section we are going to provide some high level technical details of our driver implementations. We are going to present the various IRP operations monitored by our driver.

As we already mentioned each IRP packet has a specific flag describing each I/O operation. Our driver monitors the system for the IRPs with the following flags

- IRP_MJ_CREATE
- IRP_MJ_WRITE
- IRP_MJ_SET_INFORMATION

The System sends the IRP_MJ_CREATE request when a new file or directory is being created, or when an existing file, device, directory, or volume is being opened. Normally this IRP is sent on behalf of a user-mode application that has called a Microsoft Win32 function such as **CreateFile** or **OpenFile** or on behalf of a kernel-mode component that has called **IoCreateFile**, **IoCreateFileSpecifyDeviceObjectHint**, **ZwCreateFile**, or **ZwOpenFile**. If the create request is completed successfully, the application or kernel-mode component receives a handle to the file object.

The IRP_MJ_WRITE request is sent by the I/O Manager or by a file system driver. This request can be sent, for example, when a user-mode application has called a Microsoft Win32 function such as **WriteFile** or when a kernel-mode component has called **ZwWriteFile**. We must mention that each IRP packet contains information and flags referring to the result and the type of operation. IRP_MJ_WRITE may refer to a write operation but we must know after it completes what the result was. We are only going to refer to two IRP_MJ_WRITE flags that our driver monitors; the *FO_FILE_MODIFIED* and the *FO_REMOTE_FILE*. Those flags are really important because they help us to distinguish the copy, move and cut operations from the save operations. Recall that the driver monitors “save” operations in order to enforce the file marking policy (details about file marking can be found in the following chapter). The FO_FILE_MODIFIED flag is enabled when a file is modified, therefore during a save operation. The FO_REMOTE_FILE flag is enabled during a file copy, move, cut operation from one disk volume to another.

Finally there is the IRP_MJ_SET_INFORMATION IRP type. This type can be seen in all file operations that change the internal state information of the file. Some examples of such operations are altering the file position pointer because of a file read operation, renaming the file, changing the security attributes or the date properties etc. In general all file attributes that can be changed by the *Properties* windows from the Windows Desktop trigger the IRP_MJ_SET_INFORMATION IRP. This flag is very useful in order to protect our drivers from various attacks. We are going to provide more details about this in the next section. For the time being we are going to stick to the fact that this specific type of IRP appears during file renames but also surprisingly during copy, cut, move from a disk volume to the same disk volume. Consider a file *C:\myfile.txt,* if we move the file to *C:\mymovedfile.txt* for the system this is the same file as before but with a different name, this happens because data exist on this volume. Actually that king of *move* operation is nothing more than a *rename.* On the other hand if we move the file to another volume those data does not exist and therefore have to be written, thus an IRP_MJ_WRITE appears. The same thing happens also for the copy operations. Below there is a table containing all possible IRP types that can be seen based on the I/O operation.

	IRP_MJ_CREATE	IRP_MJ_WRITE	IRP_SET_INFORMATION
OPEN	√		
READ	√	√	
WRITE	√	√	
COPY	√	√	√
MOVE	√	√	√
SAVE	√	√	
RENAME	√		√

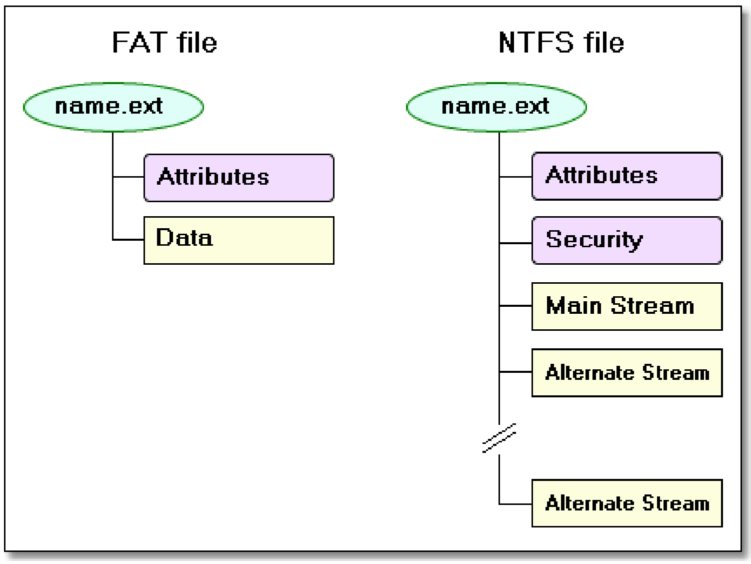
3.4 File marking

We have described in detail the internal functions of our driver regarding the monitoring of marked files; but what is a marked file, and where is that mark stored? During the development of our driver there were four alternatives regarding the above questions. As always we are going to describe the pros and the cons of each option:

- Alternate data streams
- File marking in byte sequence
- File marking as a field in file attributes
- Storing file markings in a central file

Alternate data streams

Alternate data streams have been around since the introduction of NTFS in the Windows NT operating system. What are alternate data streams though? In essence they were created to provide compatibility with HFS, or the old Macintosh Hierarchical File System. The way that the Macintosh's file system works is that it uses both the data and resource forks to store their contents. The data fork is for the contents of the document while the resource fork is to identify file type and other pertinent details. In other words the file *file.txt* may have another stream called *file.txt:stream1* or more. Compared with earlier file systems like FAT, NTFS significantly expands the customary concept of a file as a named portion of data:



The unnamed stream is a mandatory element and is always present. If you are creating an alternate stream and the file does not exist, the system will automatically create a zero length unnamed stream. If you are deleting the unnamed stream, the system considers it as a request to delete the whole file, and all the alternate streams will also be deleted. The security descriptor and the file attributes belong to the file as a whole, not to the unnamed stream. For instance, no stream can be opened for writing if the ***read-only*** attribute is set. A directory also can have alternate streams, which can be accessed exactly the same way as file streams. However a directory can't have an unnamed stream, and any attempt to access it (that is specifying the directory name without a stream name) will result the *Access denied* error. Because the colon character is used also in drive specification, it may cause an ambiguity. For example, **A:B** may represent either a file **B** in the current directory of the **A:** drive, or a stream **B** of the file **A**. The system always resolves this ambiguity as a drive and a name, so if you want it to be interpreted the other way, specify the current directory - in our example the path should look as **.\A:B**.

Now that we have clarified the theory of alternate data streams, we are going to describe how Alternate Data Streams could be used in our file marking. For each file that has to be monitored, i.e. DOC files, we demand the existence or the creation of an Alternate Data stream named *:classification* which contains the word *classified* or *non-classified*. Our driver would try to read the contents of that stream and handle the file as classified, non-classified or non-marked (if the above stream does not exist). Unfortunately this solution is not suitable for two reasons. Firstly Alternate Data streams are only available in the NTFS file system and there is no guarantee that a system will not contain FAT file systems. Secondly whenever DOC files specifically are saved or written, Microsoft Office destroys that alternate stream. Therefore, this solution is not a choice for us.

<u>**File marking in byte sequence**</u>

Another idea is to insert byte sequences inside a file in order to mark the file. The place could be the end of file, where our byte sequence would be some trailing bits or inside the header. For non read-only files, trailing bits was not a solution because if the file is expanded i.e. by adding a lot of pages inside a document the trailing bits would be overwritten. The only solution would be to insert a byte sequence inside the file header. Microsoft DOC files have some byte offsets that are not used by Microsoft Word that would be a great place to store our mark. Unfortunately when a doc file is saved, Microsoft Word rebuilds the file header and those bytes are rewritten to their default dummy values. Moreover it is not always possible to find for all data types byte offsets in the file header that are not being used. This would significantly limit the file types that could be monitored. Therefore this solution is not suitable too.

File marking as a field in file attributes

All document suites like Microsoft Office or Adobe Acrobat have the ability to add custom document attributes that are readable by their products. They also offer a developer's kit (SDK) with which programmers can write their own software. The idea was to create a lightweight library where those files could be read on the fly. The driver would call that library and read the attributes. The problem is that in order for our library to work an instance of Microsoft Word or Acrobat Reader has to be loaded in memory. That adds a significant overhead in our driver performance. Imagine if our driver had to load Microsoft Word for each DOC file. A careful reader would wonder why did we not try to read the file directly from disk without the use of the developer's kit and try to find the attribute keyword. The answer is that each attribute is not in a standard byte offset position and in order to read that specific attribute we had to iteratively read each file offset until we reach the correct one. For sure that would add much less overhead than the SDK solution.

Storing file markings in a central file

The final and selected option was to store a file list externally in another file containing the file paths. This file has the specified format for each line "file path * classification". The '*' is the delimiter. For security reasons that file can only be accessed if the kernel allows it and as we will describe later this can only happen by our own user space manager application. We will name that file from now on as record. This implementation is very flexible because we can easily monitor any type of file without the previous restrictions. Whenever our driver sees a file type of the monitored ones it simply opens the record, reads it and looks up the classification, if it does not exist in the record the file is not marked. We have to note that marking files existing in a removable disk volume is strictly not allowed, otherwise one could mark the file and simply open a Microsoft Word instance and directly save the file to the USB. This action would bypass the copy/move/ protections mechanism because there is no copy or mover operation; it is just a new file creation. Recall, that the driver DOES NOT allow an IRP_MJ_WRITE (file saving action) to complete unless the file is marked. Given that a file with path on a USB cannot be marked sequentially, it cannot be written to disk

3.5 File marking performance issues

Being in kernel mode, given that the kernel is not preemptive, a task must be finished as soon as possible otherwise the performance of our system can be greatly affected. Our driver in a normal system, adds minimal overhead if we consider how many document files can exist in a work station; maybe a hundred? Definitely there can be systems with many more files; therefore we tried to decrease the overhead of our driver. First of all, during driver initialization the record is loaded in a non paged memory in order to be always cached in RAM and to take advantage of the fast read throughput. We have written a generic structure that is being filled by the record file contents. Being that generic the structure can be enhanced with extra indexing capabilities in order to speed up the various searches inside the record structure. String operations can be quite time consuming in some cases, so indexing with some advanced indexing techniques based on hashing can be very helpful.

Even these enhancements may not be useful enough it the record structure has a large number of files. Consider a record of a thousand marked files. For each IRP the driver must iterate in all record contents. At this point it is a good time to mention that for the simplest action in our system (like a mouse click in the file) even five IRP_MJ_CREATE can appear per action; that means five thousand string operations for a simple click. Let us give another example: a PDF file of 10MB needs to be printed. During that operation the file is segmented in blocks as if it was a network operation. That can lead to even 50 IRP_MJ_READs or more for a very simple print operation. Generally for an action in a file can appear many IRPs for the same file. It would be stupid to look for each IRP our record. To solve this problem we created another component as a record cache. It is an ephemeral structure that remembers the last 10 files opened. For each IRP the drive looks up only 10 records instead of the many others contained in our big record. If the file does not exist in our cache our driver looks at the bigger record. The size of the cache size can be easily be changed by configuration in order for each administrator to tweak the diver’s performance as needed.

3.6 Userspace application and policy enforcement

One the biggest problem in the security industry is the role of the users. Highly sophisticated programs and algorithms are incapable of doing anything if the user disables them or does not follow the rules. In our case, a user has to mark the files in order to protect me them. Without the mark our driver is useless so we need a way to force the user to put marks in files. By its nature our driver targets environments where security is the primary priority and user friendliness comes last. So in order to protect our system from lazy users we have coded the driver by the following policy; *"no file can be saved (and therefore the effort of the user) unless it is marked"*. We have already talked about IRPs so when our driver captures an IRP_MJ_WRITE with FO_FILE_MODIFIED flag it looks at its records. If the file is marked it allows the I/O call otherwise it cancels it.

We have thoroughly talked about our record iterations the file marking operations, we have also talked about our user space application, and it is now time to describe it. During the description of the driver initialization we mentioned the users pace communication port, as a channel for direct message exchange between kernel and user space. Our user space application from now on will be mentioned as the controller. The controller directly passes options to the driver, for example the aforementioned cache size. It is also the means for the user to mark the files. Given that our work is a proof of concept, it does not have a special GUI, it is just a command line application. When our driver detects a file type that has to be checked, it sends the file path to the controller and the controller looks at its records. Afterwards the controller determines the file classification and sends the result back to the kernel driver. The question that can rise is whether that function could be merged to the kernel or not. The answer is definitely yes, but the complex record structures, the caching, and even the string operations, are way easier to be written in user space than in the kernel; it is just a matter of ease for the programmer.

3.7 Attacks

It is now time to summarize all the potential attacks we have briefly mentioned in the previous sections.

File extension rename

The driver has a predefined list of file types that it filters, i.e PDF files. An attacker can rename the .PDF file to .ZIP (a file type not included in the list) to evade the driver filtering. We prevent this attack by monitoring file renames (IRP_MJ_SET_INFORMATION IRPs)

File declassification or classified file renaming

This attack can easily be prevented the same way as the previous one. We strictly prohibit the declassification or the renaming of a classified file.

Creating a new file as non-classified and copy the contents of a classified

For this attack there is nothing it can be done. There is no deterministic way of doing that. In real life this can be done by simply photocopying a classified document by hiding the classified label. The only way for this attack to be prevented is to inspect the contents of each document programmatically and conclude about the mark of the file. This can only be done by machine learning and it is outside of the scope of this work.

Confused deputy and process renaming

We have thoroughly presented the problem of confused deputy in the "monitor" section. An attacker can create a process with the name of a benign process relative to the file we monitor in order to bypass our filter. We have also proposed a solution which is based on verifying the signature of the process. Although it has not been implemented it seems that could easily work.

Copy the file, mark it as non classified, and copy it to the USB

The concept of this attack is quite simple. Consider a file test.txt that is classified. We create a record for a newtest.txt marked as non classified then we copy test.txt with a new name "newtest.txt". This is the same file as test.txt but now non classified. Then we copy it to the USB freely because it is non classified. This attack can be prevented because actually the action can be captured by the IRP_MJ_SET_INFORMATION and detect that a classified file is given a new name.

4. Conclusions

We have created a prototype Data Loss Prevention application by simply using the Microsoft mini filter framework. Although not highly sophisticate we proved that by simply using clever hacks we can create a DLP that can work efficiently as one and facing others more expensive DLPs.

Our DLP can prevent a lot of attacks. Only a few attacks cannot be stopped. We consider that the attacker will attack the system as if it was a black box therefore it is quite difficult to discover or guess the defensive mechanisms we have created. Therefore the possibility for these attacks to become successful dramatically drops to nearly zero.

We have also created an easy to use user space application to control the driver and help users mark files. In order to prevent user forgetfulness we enforce the marking policy by not allowing users to store their work in the disk unless they mark it.

Finally we are quite satisfied by the performance of our driver and by the fact that it adds no extra overhead to the system. Plus the performance can also be tweaked by an administrator so as to adapt to each organization's needs.

5. References

Microsoft Developers Network: http://msdn.microsoft.com

Windows Internals 5th Edition, Mark Russinovich, David Solomon ,Alex Ionescu, Microsoft Press

Thre Rootkit Arsenal, Bill Blunden, Wordware Publishing

Rootkits, Subverting the Windows Kernel, Greg Hoglund, James Butler, Addison-Wesley Publications

The Open DLP project http://code.google.com/p/opendlp/

Handling IRPS: What Every Driver Writer Needs To Know, Microsoft Whitepaper

Filter Driver Development Guide, Microsoft Whitepaper