

# Cracking Rename Operations

This article is going to show you how to decode one of the more complex File System IRPs: a rename request. In particular, we will show you how you can filter an NT system API `NTSetInformationFile()` operation for the case `FileInformationClass` equals `FileRenameInformation`. Specifically, we will look at the IRP that represents this operation.

Filter Drivers are one of the many different types of NT Kernel Mode Device Drivers. A filter driver inserts itself above a particular device driver in order to receive all the IO requests targeted at that device driver. The filter driver can then examine and alter these IO requests before passing them down to the target device. In addition a filter driver may also process IO requests as they are completed, examining and perhaps modifying the results of IO processing by the target device.

File System Filter Drivers (FSFD) are a specific type of NT Kernel Mode Filter Drivers that in general operate between IO requests targeted at file systems and the File System Drivers that implement them.

Typically, user processes issue file IO requests which the operating system (NT) directs to the appropriate File System Driver. FSFDs layer themselves on top of one or more File System Drivers, and in doing so, become the effective target of file system IO requests directed at the file system it filters.

What can you do with an FSFD? Well, virus detection and file -replication are two typical commercial uses.

I/O in most cases is communicated to NT drivers through messages called IO Request Packets, or as they are commonly known, IRPs. Although the IRP is a standard well-defined structure, there are many different types of IRPs and in many cases the type and contents of an IRP depend on its target device. In particular, file system IO requests have their own set of IRPs that really are specific to file system I/O.

So a filter driver, in order to do anything useful, must understand exactly what type of IRPs it is going to be filtering. In the case of a FSFD, this means that the driver must understand the way in which the NT operating system packages file system operations into an IRP. For the most part, this packaging is *not documented*.

## Step 1 – Identifying an NTSetInformationFile Operation

Well of course a Rename operation in NT is not its own separate type of file operation, as it is for example in the UNIX VFS architecture. Instead, it is one of many file operations that are all possible from a single NT API call (You do know that there is an *NT System Service API* don't you? No? We are not talking about Win32. We are talking about the *real* system service API.). That API function is `NtSetInformationFile`, and its prototype is shown below:

```
NTSetInformationFile(  
    HANDLE FileHandle,  
    PIO_STATUS_BLOCK IoStatusBlock,  
    PVOID FileInformation,  
    ULONG Length,  
    FILE_INFORMATION_CLASS FileInformationClass);
```

The careful observer may notice that this function looks exactly like `ZwSetInformationFile()`, except for the substitution of Nt for Zw. Just so. They are in fact the same. Of course if you go read the documentation for `ZwSetInformationFile()` you will see absolutely no reference to rename operations. I have no explanation either. Ask Microsoft.

Now we are not going to be talking here about anything other than rename operations, so I'm only giving you enough information about **NtSetInformationFile()** to understand how a rename request is transformed from a procedure call in an application into an IRP. The IRP of course has a major function code, and it turns out that the major function code for file system setinformation operations is **IRP\_MJ\_SET\_INFORMATION**. I'm not kidding. Look it up in the DDK include files.

I'm sure you've always worried that you should have a dispatch entry point in your driver for this IRP. Now you know. If you are a file system filter driver and rename operations are of interest to you then you had better have a dispatch entry point for **IRP\_MJ\_SET\_INFORMATION**.

You can find out what *type* of the many different types of SetInformation operations you have in a particular IRP by looking at the IRP's *IoStack.Parameters.SetFile.FileInformationClass* field because right there is where the IO Manager has placed the value of FileInformationClass from the original request. In our case, we are only interested in class *FileRenameInformation* class operations.

In summary, to filter rename operations you must have a dispatch entry point in your filter driver for **IRP\_MJ\_SET\_INFORMATION**, and in that dispatch entry point you have to further decode the IRP from the *IoStack.Parameters.SetFile.FileInformationClass* field in order to identify a rename operation.

The C code for routing from **IRP\_MJ\_SET\_INFORMATION** to a *FileRenameInformation* operation might look like that below.

```
PIO_STACK_LOCATION irpSp = IoGetCurrentIrpStackLocation(Irp);

BOOLEAN IsRename = (irpSp->Parameters.SetFile.FileInformationClass ==
                    FileRenameInformation);

if (IsRename) {
    ntStatus = FilterProcessRenameOperation(Irp, irpSp);
}
```

### **But Wait, What Exactly Is A File System Rename Operation?**

It may seem obvious that a rename operation changes the name of a file. For example you could change the file C:\frob\nicate.txt to C:\frob\etacin.txt. It is less obvious that you could also change the name of C:\frob\nicate.txt to C:\frob\nicate.txt. Which is to say you can *move* a file from one location to another by renaming it.

A rename operation has four pieces of pathname data to deal with:

- The source directory or pathname
- The source filename
- The target directory or pathname
- The target filename

In addition, there are three different types of renaming that can occur:

- Rename, change the name of the source file, *not* its location
- Move - change a file's location, *not* its name
- Move and rename – change the name of a file *and* its location

In addition, there may or may not be a conflict with an existing file that currently uses the rename target filename. A filter driver or a file system driver might have to understand what is the correct thing to do if a conflict occurs.

Windows NT file system documentation refers to three types of rename operations called **Simple Rename**, **Fully Qualified Rename**, and **Relative Rename**. These only roughly correspond to the 3 cases above. What these three NT types of rename really describe is how the target of the rename operation is described by the SetInformation operation, and therefore how the file system driver, and by extension our file system filter driver, can understand it.

In the case of a Simple Rename, the source file object and the target file object are in fact the same objects. Only the name (not directory) of the file is being changed.

In the Fully Qualified Rename case, an absolute (fully qualified) pathname identifies the target of the rename operation. In this case the target filename may or may not be the same as the source filename, but we can be sure that the fully qualified source pathname is not the same as the fully qualified target pathname.

In the case of a Relative Rename, the target file name is specified as a filename relative to a directory. Once again we can be sure that the two fully qualified pathnames are not the same.

### Yes But Why Do I Care?

File System Filter Driver that is performing replication services might wish to track all rename operations so that these operations can be replayed on another file system, perhaps even on another system. To do so the filter driver would have to record the name of the file being renamed (the source file) and the new name of the file (the target file.)

For example, if I were to rename C:\frob\nicate.txt to C:\frobnicate.txt, my replication filter driver would create the log record:

```
RENAME: C:\frob\nicate.txt C:\frobnicate.txt
```

And my *replication agent*, in communication with my filter driver, would read this log record and then perform whatever miracle is required to replicate the rename elsewhere.

All the filter driver has to do is produce a simple record containing three values:

- The operation (RENAME)
- The fully qualified pathname of the source file
- The fully qualified pathname of the target file

Should be simple eh? Well, nothing is ever as simple as it should be.

### Step 2. Getting at the parameters

The NT DDK defines the parameters relevant to an **IRP\_MJ\_SET\_INFORMATION** in one of the fields in the union *IO\_STACK.Parameters* (shown below).

```
//  
// System service parameters for: NtSetInformationFile  
//
```

```

struct {
    ULONG Length;
    FILE_INFORMATION_CLASS FileInformationClass;
    PFILE_OBJECT FileObject;

    union {

        struct {

            BOOLEAN ReplaceIfExists;
            BOOLEAN AdvanceOnly;

        };

        ULONG ClusterCount;
        HANDLE DeleteHandle;

    };

} SetFile;

```

If you know how to interpret these parameters for a rename operation you are all set. Well almost. There is another structure supplied with a **IRP\_MJ\_SET\_INFORMATION** IRP that contains more of the data required to process the rename operation. Unfortunately, it is NOT defined in any standard NT include file. It might look something like the code segment below. And best of all, it most certainly is located at *Irp->AssociatedIrp.SystemBuffer*.

```

typedef struct {
    BOOLEAN Replace;
    HANDLE RootDir;
    ULONG FileNameLength;
    WCHAR FileName[1];

} FILE_RENAME_INFORMATION, *PFILE_RENAME_INFORMATION;

```

### Step 3. Putting The Pieces Together

Now that you now know where all the pieces are, all that remains is to put them into the context of a rename operation so that your filter driver can do something useful.

We need to look at the two structures from the **SET\_INFORMATION** IRP, the *IO\_STACK\_LOCATION.Parameters.SetFile* structure and the **FILE\_RENAME\_INFORMATION** structure found at *IRP.AssociatedIrp.SystemBuffer*. Our goal is to produce a log record of the rename operation.

#### The Rules

*Simple Rename:* SetFile.FileObject is NULL.

*Fully Qualified Rename:* SetFile.FileObject is non-NULL and FILE\_RENAME\_INFORMATION.RootDir is NULL.

*Relative Rename:* SetFile.FileObject and FILE\_RENAME\_INFORMATION.RootDir are both non-NULL.

Expressed as a C algorithm, these rules look like the code segment below.

```
//
// assume that Irp is a pointer to an IRP and that
// irpSp is a pointer to our IO_STACK location in that IRP.
//

PFILE_RENAME_INFORMATION renameInfo;

renameInfo = (PFILE_RENAME_INFORMATION) Irp-
>AssociatedIrp.SystemBuffer;

if (!irpSp->Parameters.SetFile.FileObject) {

    //
    // simple rename case - call a function that handles this case
    //

    status = processSimpleRename(Irp, irpSp, renameInfo);

} else {

    //
    //

    if (renameInfo->RootDirectory == NULL) {

        //
        //fully qualified rename case - call a function for
this
        //

        status = processFQRename(Irp, irpSp, renameInfo);

    } else {

        //
        // relative rename case - call a function for this
case.
        //

        status = processRelativeRename(Irp, irpSp, renameInfo);

    }

}

}
```

I mentioned earlier that you might need to know what to do if a rename target already exists. The *SetFile.ReplaceIfExists* field informs the driver about the correct action. This is a **BOOLEAN**, and if it has

the value **TRUE**, a rename target can be deleted if it exists, otherwise, if it is **FALSE**, an existing rename target causes the rename operation to fail.

## Starters

Build a fully qualified filename for the source file object. How you do that is not the subject of this article. Suffice it to say that one can always construct a fully qualified pathname for a file object by walking back through the file objects linked to the original file object and collecting each file object's **UNICODE\_STRING** name (i.e. **FILE\_OBJECT.FileName**.) thus building the fully qualified pathname of the original file object.

Having done that, we still need the fully qualified target pathname in order to record the file operation. How we get the target fully qualified pathname depends on which type of rename operation we have: simple, fully qualified or relative. The remainder of this article demonstrates how for each rename type the target pathname can be constructed.

## Simple Rename

Easy – just copy the pathname component of the fully qualified source filename and append the target filename. The target filename is the WCHAR string at **FILE\_RENAME\_INFORMATION.FileName**.

The code for a simple rename might look like that below (note that the following code segments, while based on a functional filter driver are pseudo code only and will not compile without errors).

```
//
// simple rename case
// FileNameBuffer is the fully qualified source file name
//

WCHAR * FileNameBuffer;

//
// renameInfo is a pointer to FILE_RENAME_INFORMATION structure
//

PFILE_RENAME_INFORMATION renameInfo;

//
// NewNameBuffer will contain the fully qualified target file name
//

WCHAR * NewNameBuffer;

//
// prefix is a pointer to the end of the pathname
// component of FileNameBuffer

WCHAR * prefix;

//
// prefixLength is the length of prefix
//
```

```

ULONG prefixLength;

//
// length is the string length of NewNameBuffer
//

ULONG length;

//
// First find out just how big a string we need:
//

length = 0;

//
// find the last pathname component of the source file
//

prefix = wcsrchr(FileNameBuffer, (int) L'\\');

if (prefix == NULL) {

    //
    // just the file system will fail too.
    //
    return (STATUS_SUCCESS);

} else {

    //
    // set prefixlength and initialize length
    //
    length = prefixLength = ((prefix - FileNameBuffer) + 1)*
sizeof(WCHAR);

}

//
// Now add the prefix length to the length of the target FileName
//

length += (wcslen(renameInfo->FileName) + 1) * sizeof(WCHAR);

//
// Great! Now allocate the buffer for this thing
//

NewNameBuffer = ExAllocatePoolWithTag(PagedPool, length, 'tset');

if (!NewNameBuffer) { // ? no memory of any type?

    return (STATUS_INSUFFICIENT_RESOURCES);
}

```

```

}

//
// Ok we have the memory and the length, now construct the string
//

RtlZeroMemory(NewNameBuffer, length);

(void) wcsncpy(NewNameBuffer, FileNameBuffer, prefixLength);

wscat(NewNameBuffer, renameInfo->FileName);

//
// That's it - now just create the log record
//

```

### Fully Qualified Rename

In the Fully Qualified Rename case, *RenameInfo.FileName* is the fully qualified pathname of the rename target. This makes our task relatively simple.

I've introduced one other complexity here, the drive specification of the fully qualified pathname. In the simple rename case I ignored the drive specification portion of the fully qualified pathname. In other words I just left out the fact that to fully log a file system operation we need the fully qualified pathname to include the drive (e.g. "C:"). I left it out for the same reason I am omitting a discussion of exactly how one constructs a fully qualified pathname from a *File Object*: it is in and of itself a complicated enough subject to warrant its own article. In the simple rename case, just assume that the source file name included the drive letter and that the target file name will as well.

Since we are not just copying the source file name, this drive letter has to come from somewhere. In the following example, how we obtain the drive specification is undefined, and a global variable *DriveLetter* just magically has the correct value.

```

//
// fully qualified rename
//
// renameInfo is a pointer to the FILE_RENAME_INFORMATION structure
//

PFILE_RENAME_INFORMATION renameInfo;

//
// NewNameBuffer will contain the fully qualified target file name
//

WCHAR * NewNameBuffer;

//
// startOffset is used to locate the beginning of the
// fully qualified pathname we want to use from the string
// that is passed into us at renameInfo.FileName
//

```

```

ULONG startOffset = 0;

//
// If we need to add a drive spec, prependChars is used
// to account for the space we need to do that.
//

ULONG prependChars = 0;

//
// finally target length is going to be set to the total
// size of the WCHAR string we are going to construct.
//

ULONG targetlength = 0;

//
// DriveLetter has been set to the correct value for this operation.
//

extern WCHAR DriveLetter;

//
// For this case the string at renameInfo.FileName MUST start with '\\'
// and could start with "\DosDevices\". If we find "\DosDevices\" then
// we can assume that this string already has a drive specifier,
// otherwise we will provide one based on the value of DriveLetter.
//
// If the string does not start with a '\\' note this and stop
// trying to process this operation.
//

if (renameInfo->FileName[0] != L'\\') {

    return (STATUS_SUCCESS);

}

//
// See if the prefix of FileName is "\DosDevices\"
// if it is, strip off the \DosDevices\ part, leaving the FQpathname.
// else prepend the Device specification.
//

if (0 == wcsnicmp(L"\\DosDevices\\", renameInfo->FileName, 12)) {
    startOffset = 12;
    // Copy from end of \DosDevices\
} else {
    // put in the device spec (e.g. D:)

    prependChars = 2;

```

```

}

//
// figure out how many WCHARS we need
//

targetLength = (renameInfo->FileNameLength/sizeof(WCHAR)) -
                startOffset + prependChars + 1;
                // make room for NULL too

if (targetLength < 4) {
    // must be at least "D:\"

    return (STATUS_SUCCESS);

}

//
// Allocate the storage for the target file name
//

NewNameBuffer = ExAllocatePool(PagedPool, targetLength * sizeof(WCHAR),
'tset');

//
// if there is no paged pool available fail the operation
//

if (NewNameBuffer == NULL) {

return (STATUS_INSUFFICIENT_RESOURCES);

}

//
// now construct the string by prepending the drive letter if needed
// and appending the filename in renameInfo.
//

if (prependChars) {

    //
    // set the device spec, just assume that DriveLetter is a WCHAR
    // that contains the correct value.
    //

    NewNameBuffer[0] = DriveLetter;
    NewNameBuffer[1] = L':';

}

wcsncpy(&NewNameBuffer[prependChars], &renameInfo-
>FileName[startOffset],
                targetLength );

```

```
//  
// that's all - go log the operation  
//
```

## Relative Rename

Okay, we took care of the easy cases, now lets take a look at the last case, the relative rename operation. In this case the source file is being moved to a new location relative to the directory referenced at **FILE\_RENAME\_INFORMATION.RootDir**. One can easily construct the fully qualified pathname from a file object, but we are given a handle. So we have to call into the object manager to get the object (i.e. the file object) referenced by the handle at RootDir. *This can only work if we are in the same process context as the process that issued the original IO request.*

An example of processing a relative rename operation in order to produce a log record of the operation from a file system filter driver is shown below.

```
//  
// relative rename  
//  
// renameInfo is a pointer to FILE_RENAME_INFORMATION structure  
//  
  
PFILE_RENAME_INFORMATION renameInfo;  
  
//  
// NewNameBuffer will contain the fully qualified target file name  
//  
  
WCHAR * NewNameBuffer;  
  
//  
// status is used to collect return value from standard DDK functions.  
//  
  
NTSTATUS status;  
  
//  
// We need to convert from the RootDir HANDLE to a FILE_OBJECT  
//  
  
PFILE_OBJECT dirObject;  
  
//  
// we will call ConstructFQPname() and this function will  
// magically produce a fully qualified pathname given a file  
// object and a drive specification dirPath will contain the  
// unicode string for the fully qualified pathname  
//  
  
UNICODE_STRING dirPath;
```

```

//
// as usual we need to index into WCHAR strings
//

ULONG wcharOffset;

//
// Our friend the mysterious drive specification
//

extern WCHAR DriveLetter;

//
// We need the fully qualified pathname
// of the directory referenced by Buffer->RootDir
// Once we have that we should then be able to
// append the contents of Buffer->FileName to
// construct the fully qualified target pathname
//

status = ObReferenceObjectByHandle(renameInfo->RootDirectory,
                                  STANDARD_RIGHTS_REQUIRED,
                                  NULL,
                                  KernelMode,
                                  (PVOID *)&dirObject,
                                  NULL);

if (!NT_SUCCESS(status) ) {

    //
    // assume that the file system driver will fail as well
    //
    return (STATUS_SUCCESS);

}

//
// the function ConstructFQpname takes a file object and a driveletter
// as input and produces a UNICODE_STRING containing the fully
// qualified pathname of the input file object as output.
// The buffer for the UNICODE_STRING is allocated by this function, we
// must remember to free it up when we are done
//

if (! ConstructFQpname(dirObject, &dirPath, DriveLetter) ) {

    //
    // assume the file system driver will fail
    //
    return (STATUS_SUCCESS);

}

ObDereferenceObject(dirObject); // we don't need this anymore

```

```

//
// figure out how many WCHARs we need
//

targetLength = ((dirPath.Length + renameInfo->FileNameLength ) /
                sizeof(WCHAR)) + 2; // add in a L'\ ' and a NULL

//
// Allocate the storage for the target file name
//

NewNameBuffer = ExAllocatePool(PagedPool, targetLength * sizeof(WCHAR),
'tset');

if (NewNameBuffer == NULL) {

    //
    // Or we can stop a file operation right here
    //
    return (STATUS_INSUFFICIENT_RESOURCES);

}

//
// copy the relative directory fully qualified pathname to our buffer
//

wcharOffset = dirPath.Length/sizeof(WCHAR);

//
// if we need a path separator, then add it it now
//

if (NewNameBuffer[wcharOffset-1] != L'\\') {

    NewNameBuffer[wcharOffset] = L'\\';

    wcsncpy(&NewNameBuffer[wcharOffset+1],
            renameInfo->FileName,
            renameInfo->FileNameLength/sizeof(WCHAR));

} else {

    //
    // we don't need a path separator
    //
    wcsncpy(&NewNameBuffer[wcharOffset],
            renameInfo->FileName,
            renameInfo->FileNameLength/sizeof(WCHAR));

}

```

```
//  
// since constructFQPname allocated storage, free it up  
//  
  
ExFreePool(dirPath.Buffer);  
  
//  
// that's all - go log the operation  
//
```

## Conclusion

To wrap up rename operations here is what your filter driver needs to do then:

- Filter **IRP\_MJ\_SET\_INFORMATION** operations.
- Decode rename operations by examining the **IO\_STACK.Parameters.SetFile.FileInformation Class field**
- Compute the fully qualified pathnames for the source and target files of the rename operation. To do that you have to understand the rules for deciding if you have a *simple rename*, a *fully qualified rename*, or a *relative rename*, and derive the target file name correctly based on which type of rename you have.

That's all that there really is to logging a rename operation. Of course there is quite a lot more to correctly filtering file system operations that we have not mentioned at all in this article. For example, we have not discussed issues such as completion handling. Yet a file system filter driver that performs replication, as in our example, cannot assume that all operations succeed. It might need to undo a logging operation if it observes that an IO request failed on completion.

And we only touched on the issue of process context indirectly in one of the code samples, yet dealing with process context is frequently a complex design issue in a filter driver. For example, if the underlying device driver assumes that certain IO requests are always processed in the user's process context, then a filter driver must not violate that assumption. And, as we have seen, if a logging operation needs to obtain information from a handle to an object, it must be sure that that handle is valid in the current execution context.