

The Definitive Guide on Win32 to NT Path Conversion

 googleprojectzero.blogspot.com/2016/02/the-definitive-guide-on-win32-to-nt.html

Posted by James Forshaw, path'ological reverse engineer.

How the Win32 APIs process file paths on Windows NT is a tale filled with backwards compatibility hacks, weird behaviour, and beauty[†]. Incorrect handling of Win32 paths can lead to security vulnerabilities. This blog post is to try and give a definitive* guide on the different types of paths supported by the OS. I'm going to try and avoid discussion of quirks in the underlying filesystem implementations (such as NTFS streams and the like), and instead focus on the Win32 to NT conversion layer.

The reason this blog post is needed at all is that the documentation for Win32 paths is incomplete relative to what's actually implemented in the OS. If you find some code which takes an untrusted path and either does no sanitization at all or attempts to sanitize without fully understanding all the possible scenarios there could be trouble. I was reminded when a colleague found an interesting path-handling vulnerability which I realised could be exploited further by abusing weird, and barely documented, behaviour in the Win32 path conversion code.

Caveats: this is based on analysis of Windows 8.1 and Windows 10. As some of this information isn't documented the implementation might change in the future. Also this is based on calling the APIs directly, which a normal application is likely to do. Some or all of these tricks might not work if passed into the Shell APIs or over an SMB share.

** I've done my best to make this definitive, but there's always a chance I've missed something.*

† As it's commonly stated, beauty is in the eye of the beholder.

Background on the Relationship Between Win32 and NT Paths

The Windows kernel's IO manager handles file paths differently from what is exposed through the public user-mode APIs such as [CreateFile](#). The documentation for [CreateFile](#) and related functions generally point to [this](#) page, which describes the different types of paths. Unfortunately it doesn't really delve too deeply into how the paths are converted; of course as [Raymond Chen](#) might describe the situation, it's an implementation detail you shouldn't rely on. In the real world, if you're dealing with paths which might come from untrusted sources, sometimes you have to understand how it works to determine what you need to protect against, so let's get under the hood and delve into the implementation detail.

When you call `CreateFile`, the API must do a conversion between the many different types of paths Win32 supports and the underlying NT IO manager representation. Internally `CreateFile` calls a NTDLL export, `RtlDosPathNameToRelativeNtPathName_U`, which takes the Unicode Win32 path and returns the appropriate NT path form. The fact that the function refers to these paths as Dos Paths gives the game away as to their legacy. The function has the following prototype; if the conversion is successful then the function will return `TRUE`:

```
typedef struct _RTL_RELATIVE_NAME {
    UNICODE_STRING RelativeName;
    HANDLE         ContainingDirectory;
    void*          CurDirRef;
} RTL_RELATIVE_NAME, *PRTL_RELATIVE_NAME;

BOOLEAN NTAPI RtlDosPathNameToRelativeNtPathName_U(
    _In_     PCWSTR DosFileName,
    _Out_    PUNICODE_STRING NtFileName,
    _Out_opt_ PWSTR* FilePath,
    _Out_opt_ PRTL_RELATIVE_NAME RelativeName
);
```

There are actually a few variants of this function. `RtlDosPathNameToNtPathName_U`, for example, won't return the full relative path information (I'll get on to what that means later). There's also the functions suffixed with "WithStatus," which instead of returning `TRUE` or `FALSE` return an `NTSTATUS` code that describes the reason for conversion failure.

There are 7 types of path that the Win32 API distinguishes between, and potentially does different things with. NTDLL has a function, `RtlDetermineDosPathNameType_U`, which, given a Unicode string will return you the path type. We'll go through each one of these types in the next section. The following prototype can be used to call this function:

```
enum RTL_PATH_TYPE {
    RtlPathTypeUnknown,
    RtlPathTypeUncAbsolute,
    RtlPathTypeDriveAbsolute,
    RtlPathTypeDriveRelative,
    RtlPathTypeRooted,
    RtlPathTypeRelative,
    RtlPathTypeLocalDevice,
    RtlPathTypeRootLocalDevice
};
```

```
RTL_PATH_TYPE NTAPI RtlDetermineDosPathNameType_U(_In_ PCWSTR Path);
```

Under the hood most of the heavy lifting of converting these different path types is done using the `RtlGetFullPathName_U` API. This takes a path string and performs conversion, canonicalization and resolving of current directory information. In most cases this function does not verify whether the path exists (that's why you're opening it) but I'll point out situations later where checks are made. We can call this externally with the following prototype; note in this case the function returns the number of bytes of path information converted:

```
ULONG NTAPI RtlGetFullPathName_U(  
    _In_ PWSTR FileName,  
    _In_ ULONG BufferLength,  
    _Out_writes_bytes_(BufferLength) PWSTR Buffer,  
    _Out_opt_ PWSTR *FilePart);
```

This API is actually exposed through the standard Win32 API [GetFullPathName](#) so you don't need to import it directly from NTDLL. I've put together a simple tool to use these APIs to query what the converted NT path is for each path type. I'll use it as we go along. If you want to use it yourself you can find it at the end of this blog post.

Types of DOS Path

Let's look at each type of path in turn to see what they are and how they are converted into an NT path. I'll also point out interesting behaviours as we go along and hopefully correct some assumptions about the types of paths.

A common theme which will come up is canonicalization rules. I'll explain the main rules now before digging into the different paths and I'll point out any odd behavior for each type. The implementation does the following things to a path to make it canonical to pass through to the NT APIs.

- Convert all forward slashes (character U+002F) to backslash path separator (character U+005C).
- Collapse repeating runs of path separators into one.
- Split up path elements and:
 - Remove elements where the name is only a single dot signifying the current directory.
 - Remove the previous path element where the name is two dots, if it's not already at the root of the path type. This is to allow relative paths referring to a parent.
- If the last character is a path separator leave as is in the final result.
- Remove any trailing spaces or dots for the last path element, assuming that it isn't a single or double dot name.

That last rule seems odd, but as we'll see it really does do this for normal paths.

Drive Absolute

This is the simplest of the types of Win32 paths available and in theory the most unambiguous. Everyone should be familiar with this form, it contains a drive and at least one path element.

The following are all valid Drive Absolute paths with the results of calling `RtlGetFullPathName_U` and `RtlDosPathNameToRelativeNtPathName_U` on them (note that `<SP>` refers to the space character).

| Dos Path | Full Path | NT Path |
|------------------|------------|----------------|
| X:\ABC\DEF | X:\ABC\DEF | \??\X:\ABC\DEF |
| X:\ | X:\ | \??\X:\ |
| X:\ABC\ | X:\ABC\ | \??\X:\ABC\ |
| X:\ABC\DEF.<SP>. | X:\ABC\DEF | \??\X:\ABC\DEF |
| X:/ABC/DEF | X:\ABC\DEF | \??\X:\ABC\DEF |
| X:\ABC\..\XYZ | X:\XYZ | \??\X:\XYZ |
| X:\ABC\..\..\ | X:\ | \??\X:\ |

As specified in the canonicalization rules, the trailing space and dots have been removed. Note the final row shows that you cannot create a relative path to replace the drive letter no matter how many parent directory references you use.

Drive Relative

These types of paths specify the drive letter but do not follow the colon with a path separator. For example `C:ABC` is a Drive Relative path. The implementation will replace the drive letter with the current directory set for that drive. The following process is used to determine the current directory for the Drive Relative path.

1. If the drive letter matches the current working directory drive letter then use that directory.
2. If the environment variable `'=?:'` exists (where `?` is replaced with the drive letter) and the path exists then use the environment variable's value.
3. If all else fails just use the drive root path (as in `?:\`) and ensure the environment variable is updated to reflect that state.

Assuming that the current working directory is set to X:\ABC, the Y drive has a variable for Y:\DEF and Z drive is not set to anything. There shouldn't be anything unusual about any of the following results.

| Dos Path | Full Path | NT Path |
|--------------|----------------|--------------------|
| X:DEF\GHI | X:\ABC\DEF\GHI | \??\X:\ABC\DEF\GHI |
| X: | X:\ABC | \??\X:\ABC |
| X:DEF.<SP>. | X:\ABC\DEF | \??\X:\ABC\DEF |
| Y: | Y:\DEF | \??\Y:\DEF |
| Z: | Z:\ | \??\Z:\ |
| X:ABC\..\XYZ | X:\ABC\XYZ | \??\X:\ABC\XYZ |
| X:ABC\..\..\ | X:\ | \??\X:\ |

Of course if you know anything about Win32 programming you'd assume that you can only specify one current directory using the [SetCurrentDirectory](#) API. So how do any of the other environment variables get set? Well, in general they don't; this is a feature implemented in the path conversion for the benefit of command shell cmd.exe. The environment variable is set when you 'cd' to a new directory. These variables are hidden from most tools; however you can see them using a debugger and dumping the PEB (in this case using !peb in WinDBG).

```

C:\Windows\SysWOW64\cmd.exe - WinDbg:10.0.10240.9 X86
File Edit View Debug Window Help
Command
SubSystemData: 00000000
ProcessHeap: 00640000
ProcessParameters: 006411e8
CurrentDirectory: 'C:\Windows\'
WindowTitle: 'C:\Windows\SysWOW64\cmd.exe'
ImageFile: 'C:\Windows\SysWOW64\cmd.exe'
CommandLine: 'C:\Windows\SysWOW64\cmd.exe'
DllPath: '< Name not readable >'
Environment: 00645038
=::=:\
-C:=-C:\Windows
-D:=-D:\OS2
-Z:=-Z:\dev
ALLUSERSPROFILE=C:\ProgramData
APPDATA=C:\Users\user\AppData\Roaming
CommonProgramFiles=C:\Program Files (x86)\Common Files
CommonProgramFiles(x86)=C:\Program Files (x86)\Common Files
CommonProgramW6432=C:\Program Files\Common Files
COMPUTERNAME=DESKTOP-F9T0PCQ
ComSpec=C:\WINDOWS\system32\cmd.exe
FPS_BROWSER_APP_PROFILE_STRING=Internet Explorer
FPS_BROWSER_USER_PROFILE_STRING=Default
HOMEDRIVE=C:
.....
0:002>
Ln 0, Col 0 Sys 0:<Local> Proc 000:c2c Thrd 002:60 ASM OVR CAPS NUM

```

We can see four different drives are set; however, what's up with the first one '=::=:\'? To explain that, it's natural to assume that drive "letters" can only be A through Z. It turns out the `RtlGetFullPathName_U` API does not enforce this requirement, although the Explorer shell and command prompt almost certainly do. Therefore as long as the second character of a path is a colon, the conversion will treat it as a Drive Absolute or Drive Relative path. Of course if the `DosDevices` object directory doesn't have an appropriate symbolic link it's not going to do you much good.

```

C:\WINDOWS\system32\cmd.exe
C:\test>ConvertDosPathToNtPath.exe @:\abc
Converting: '@:\abc'
To: '\\?@\abc'
Type: RtlPathTypeDriveAbsolute
FileName: abc
FullPathName: '@:\abc'

C:\test>ConvertDosPathToNtPath.exe !:\xyz
Converting: '!:\xyz'
To: '\\?!\:xyz'
Type: RtlPathTypeDriveRelative
FileName: xyz
FullPathName: '!:\xyz'

C:\test>

```

We can now explain why you find the weird '=::=:\' entry in the environment block of most applications. The Explorer shell uses a special format to refer to shell objects. For example,

pasting ‘::<20d04fe0-3aea-1069-a2d8-08002b30309d}’ into the run dialog will open the computer folder. Somewhere in Explorer something is passing one of these shell names to a file API, which is interpreting it as a Drive Relative path for the ‘:’ drive. As it doesn’t find an existing environment variable for the drive it adds the default environment variable. As the environment is inherited by default it’s migrated into other processes.

Rooted

A Rooted path is one which starts with a path separator. This creates a path rooted on the drive currently set in the current working directory. Effectively the implementation prepends the root drive (or UNC path if set) and then applies the normal canonicalization rules for that path type. So assuming the current working directory is X:\ABC, we’d get the following:

| Dos Path | Full Path | NT Path |
|---------------|------------|----------------|
| ABC\DEF | X:\ABC\DEF | \??\X:\ABC\DEF |
| \ | X:\ | \??\X:\ |
| ABC\DEF.<SP>. | X:\ABC\DEF | \??\X:\ABC\DEF |
| /ABC/DEF | X:\ABC\DEF | \??\X:\ABC\DEF |
| ABC\..\XYZ | X:\XYZ | \??\X:\XYZ |
| ABC\..\..\. | X:\ | \??\X:\ |

Relative

These paths are relative to the current working directory. The implementation determines a path is relative if it doesn’t start with a path separator and its second character is not a colon (indicating a drive path). The simplest way to think of the implementation is the relative component is appended to the current working directory path with a path separator added and the canonicalization rules are applied. Assuming the currently working directory is X:\XYZ, then:

| Dos Path | Full Path | NT Path |
|----------|----------------|--------------------|
| ABC\DEF | X:\XYZ\ABC\DEF | \??\X:\XYZ\ABC\DEF |
| . | X:\XYZ | \??\X:\XYZ |

| | | |
|---------------|----------------|--------------------|
| ABC\DEF.<SP>. | X:\XYZ\ABC\DEF | \??\X:\XYZ\ABC\DEF |
| ABC/DEF | X:\XYZ\ABC\DEF | \??\X:\XYZ\ABC\DEF |
| ..\ABC | X:\ABC | \??\X:\ABC |
| ABC\..\..\. | X:\ | \??\X:\ |

Note that you can't have an empty path (which includes just spaces or dots). If you want to refer to the current directory you need to specify a single dot.

Relative paths also trigger another behavior when calling `RtlDosPathNameToRelativeNtPathName_U`, which can be used by the Win32 APIs. If the relative path is within the current working directory, or one of its children, then the implementation can return a file handle to the current working directory in the `RTL_RELATIVE_NAME` structure, which also contains only the relative name component. This can be passed as the `RootDirectory` handle in the `OBJECT_ATTRIBUTES` structure to `NtCreateFile`. This can be a performance win as it avoids the object manager needing to parse the entire path, work out the filesystem device, and call it with the subpath to parse. Instead it can call the NTFS driver's parse routine immediately. This handle is opened whenever the current directory is changed and stored in a global variable in NTDLL. This is the reason you can't delete the directory an application has set as its current working directory as it holds a handle with no `SHARE_DELETE` option. We can see this behaviour by running the tool and specifying a relative path.

```

C:\WINDOWS\system32\cmd.exe
C:\test>ConvertDosPathToNtPath.exe abc
Converting: 'abc'
To: '\??\C:\test\abc'
Type: RtlPathTypeRelative
FileName: abc
RelativeName: 'abc'
Directory: 0x14
CurDirRef: 0x591D80
FullPathName: 'C:\test\abc'

```

UNC Absolute

Universal Naming Convention (UNC) paths are a type which is pretty much only found on Windows (although arguably URIs replace their role on everything else). They're used to access remote file systems, typically SMB but can be almost any implementation such as WebDAV (installed by default) or one of the many virtualization shared folder implementations. By convention a UNC path starts with two path separators, a server address (be it a domain name or an IP address), then the name of a share on that server. Finally the relative path to the resource you want is specified afterwards.

The conversion rules are pretty simple; the path is canonicalized as per the usual rules, although in this case the root is considered to be the share name not the drive letter; and finally the leading path separators are replaced with the string '\\??\UNC', which routes to the Multiple UNC Provider (MUP) driver which handles dispatching the request to the appropriate remote file system provider. On to examples:

| Dos Path | Full Path | NT Path |
|---------------------------|------------------------|-------------------------------|
| \\server\share\ABC\DEF | \\server\share\ABC\DEF | \\??\UNC\server\share\ABC\DEF |
| \\server | \\server | \\??\UNC\server |
| \\server\share | \\server\share | \\??\UNC\server\share |
| \\server\share\ABC.<SP> | \\server\share\ABC | \\??\UNC\server\share\ABC |
| //server/share/ABC/DEF | \\server\share\ABC\DEF | \\??\UNC\server\share\ABC\DEF |
| \\server\share\ABC\..\XYZ | \\server\share\XYZ | \\??\UNC\server\share\XYZ |
| \\server\share\ABC\..\..\ | \\server\share | \\??\UNC\server\share |

Local Device

A Local Device path is any path that begins with the sequence '\\.\.'. This looks like a UNC path with a server name of '.', however instead it's used to directly escape to the DosDevices object manager directory. This directory contains things like the symbolic links for drive letters as well as for kernel drivers. It's most commonly used to access devices such as COM ports and named pipes.

| Dos Path | Full Path | NT Path |
|---------------------|-----------------|------------------|
| \\.\COM20 | \\.\COM20 | \\??\COM20 |
| \\.\pipe\mypipe | \\.\pipe\mypipe | \\??\pipe\mypipe |
| \\.\X:\ABC\DEF.<SP> | \\.\X:\ABC\DEF | \\??\X:\ABC\DEF |
| \\.\X:/ABC/DEF | \\.\X:\ABC\DEF | \\??\X:\ABC\DEF |
| \\.\X:\ABC\..\XYZ | \\.\X:\XYZ | \\??\X:\XYZ |

| | | |
|---|-------------------------------|-------------------------------|
| <code>\\.\X:\ABC\..\C:\</code> | <code>\\.\C:\</code> | <code>\??\C:\</code> |
| <code>\\.\pipe\mypipe\..\notmine</code> | <code>\\.\pipe\notmine</code> | <code>\??\pipe\notmine</code> |

Most things are as you'd expect. These paths are still canonicalized so trailing spaces and dots are removed. An odd behavior especially related to normal Drive Absolute or UNC Absolute paths is that you can completely remove the first component of the path. This allows you to change the drive, change a local path to a UNC path, or even open a different named pipe as the underlying APIs have no knowledge of what's being accessed so it can't make any assumptions. I actually exploited this behaviour to create arbitrary named pipes from the Chrome sandbox awhile back (see the fixed issue [here](#)).

Note that `\\localhost\xyz` is not the same as `\\.\xyz` even though some APIs (say `LogonUser`) blur the distinction a little bit by specifying things like the local logon server as a single dot. The former accesses a UNC share on localhost over IP, whereas the latter tries to access the device name `xyz`.

You might make an assumption that you can only access special devices by specifying this form of path. If you read the [page](#) on path formats, though, you'll find the following note, although it doesn't explain why there's a restriction.

- Do not use the following reserved names for the name of a file:

CON, PRN, AUX, NUL, COM1, COM2, COM3, COM4, COM5, COM6, COM7, COM8, COM9, LPT1, LPT2, LPT3, LPT4, LPT5, LPT6, LPT7, LPT8, and LPT9. Also avoid these names followed immediately by an extension; for example, `NUL.txt` is not recommended. For more information, see [Namespaces](#).

The reason for this is due to legacy DOS support. For example, if you wanted to write data to the first COM port you could issue the following command:

```
echo ATDT 2024561414 > COM1
```

The list of supported special case device names are as follows:

- PRN
- AUX
- NUL
- CON
- LPT[1-9]
- COM[1-9]
- CONIN\$
- CONOUT\$

Before we start, note that CONIN\$ and CONOUT\$ are not documented as being reserved. The LPT and COM names can take a number between 1 and 9 to refer to the first 9 configured ports. Actually even this isn't strictly true, as the underlying code passes the character to the `iswdigit` library function and permits anything which is a digit based on the result of the call and not being '0'. If you test all the 16-bit characters as to whether they're considered digits it also includes characters U+00B2, U+00B3, and U+00B9, which are Superscript 2, Superscript 3, and Superscript 1 respectively. So if you're desperate for a port name of COM² then go ahead.

This behavior was emulated in the path conversion process so that these plain names are converted to Local Device paths, which end up with the correct NT path. Now if just specifying these paths explicitly was all that this process handled it would be annoying but not the end of the world. However it's much worse. The conversion process actively tries to convert any path with the device name last, even if the path is a Drive Absolute path. To make matters even worse the device name can have arbitrary trailing characters as long the trailing characters are separated from the device by a dot or a colon. The name can then also have trailing spaces. Let's look at some examples:

| Dos Path | Full Path | NT Path |
|----------------------|----------------|----------------------|
| COM1 | \\.\COM1 | \??\COM1 |
| X:\COM1 | \\.\COM1 | \??\COM1 |
| X:COM1 | \\.\COM1 | \??\COM1 |
| valid\COM1 | \\.\COM1 | \??\COM1 |
| X:\notvalid\COM1 | \\.\COM1 | Error in Conversion |
| X:\COM1.blah | \\.\COM1 | \??\COM1 |
| X:\COM1:blah | \\.\COM1 | \??\COM1 |
| X:\COM1<SP><SP>.blah | \\.\COM1 | \??\COM1 |
| \\.\X:\COM1 | \\.\X:\COM1 | \??\X:\COM1 |
| \\abc\xyz\COM1 | \\abc\xyz\COM1 | \??\UNC\abc\xyz\COM1 |

Plenty of misbehavior here. Drive Absolute, Drive Relative, and Relative paths will be forcefully converted. Note that the preceding drive path must be a valid directory, otherwise the conversion to the NT path fails, although getting the full path works. Why it does the

check is beyond me as it seems to serve no actual purpose. Also note the removal of trailing suffixes, which can come in handy if something is actively trying to guard against this behavior. For example, if an application was mindful and was checking for a filename that matched one of the reserved names you can just bypass that check by appending an arbitrary suffix.

Root Local Device

The final type is the Root Local Device path. This is any path that begins with the characters `\\?\` and acts as an escape into the object manager. It's almost exactly the same as the Local Device path type with one crucial difference: no canonicalization of the path is done. What this means is that forward slashes are not converted to backslashes, relative paths are not collapsed, and trailing spaces/dots are not removed. This lack of canonicalization has a number of useful properties when you can pass it to an application that is trying to secure paths.

| Dos Path | Full Path | NT Path |
|--|-----------------------------|--|
| <code>\\?\X:\ABC\DEF</code> | <code>\\?\X:\ABC\DEF</code> | <code>\\?\X:\ABC\DEF</code> |
| <code>\\?\X:\</code> | <code>\\?\X:\</code> | <code>\\?\X:\</code> |
| <code>\\?\X:</code> | <code>\\?\X:</code> | <code>\\?\X:</code> |
| <code>\\?\X:\COM1</code> | <code>\\?\X:\COM1</code> | <code>\\?\X:\COM1</code> |
| <code>\\?\X:\ABC\DEF.<SP></code> | <code>\\?\X:\ABC\DEF</code> | <code>\\?\X:\ABC\DEF.<SP></code> |
| <code>\\?\X:/ABC/DEF</code> | <code>\\?\X:\ABC\DEF</code> | <code>\\?\X:/ABC/DEF</code> |
| <code>\\?\X:\ABC\..\XYZ</code> | <code>\\?\X:\XYZ</code> | <code>\\?\X:\ABC\..\XYZ</code> |
| <code>\\?\X:\ABC\..\..\</code> | <code>\\?\</code> | <code>\\?\X:\ABC\..\..\</code> |

Note that all canonicalization is skipped, including converting device names such as CON and replacing drive letters with their current directory. But notice the discrepancy with the last four rows. While the resulting NT path has no canonicalization, the Full Path result has canonicalized the paths. This can only mean one thing:

`RtlDosPathNameToRelativeNtPathName_U` must be special casing our path type and not calling `RtlGetFullPathName_U` on them. We can look at the code to find out what it's doing:

```
if (DosPath->Length > 8) {
    WCHAR* buffer = DosPath->Buffer;
    if (*buffer == '\\') {
```

```

if (buffer[1] == '\\ ' || buffer[1] == '?')
    && buffer[2] == '?' && buffer[3] == '\\') {
    return RtlpWin32NtNameToNtPathName(DosPath, ...);
}
}
}
}
// Continue with processing.

```

So if the path starts with the \\?\ prefix we instead call into RtlpWin32NtNameToNtPathName. This explains the discrepancy. But wait, look again at the check, it isn't just checking for \\?\, it also allows the second character to be another '?'. What this means is that CreateFile or similar APIs also accept the form '\\?\ABC' as a valid Root Local Device path. Let's just check to prove to ourselves it works:

```

C:\WINDOWS\system32\cmd.exe
C:\test>ConvertDosPathToNtPath.exe \\?\X:\ABC
Converting:  '\\?\X:\ABC'
To:          '\\?\X:\ABC'
Type:       RtlPathTypeRooted
FileName:   ABC
FullPathName: 'C:\??\X:\ABC'

C:\test>ConvertDosPathToNtPath.exe \\?\X:/ABC\DEF
Converting:  '\\?\X:/ABC\DEF'
To:          '\\?\X:/ABC\DEF'
Type:       RtlPathTypeRooted
FileName:   DEF
FullPathName: 'C:\??\X:\ABC\DEF'

C:\test>

```

If we run the same set of paths as before we'll see the discrepancies (assume that the current drive is the X: drive).

| Dos Path | Full Path | NT Path |
|----------------------|------------------|----------------------|
| \\?\X:\ABC\DEF | X:\??\X:\ABC\DEF | \\?\X:\ABC\DEF |
| \\?\X:\ | X:\??\X:\ | \\?\X:\ |
| \\?\X: | X:\??\X: | \\?\X: |
| \\?\X:\COM1 | X:\??\X:\COM1 | \\?\X:\COM1 |
| \\?\X:\ABC\DEF.<SP>. | X:\??\X:\ABC\DEF | \\?\X:\ABC\DEF.<SP>. |
| \\?\X:/ABC/DEF | X:\??\X:\ABC\DEF | \\?\X:/ABC/DEF |

| | | |
|-------------------|--------------|-------------------|
| \??\X:\ABC\..\XYZ | X:\??\X:\XYZ | \??\X:\ABC\..\XYZ |
| \??\X:\ABC\..\..\ | X:\ | \??\X:\ABC\..\..\ |

This type of path is just begging to be made an example of. If you look at the type returned by the call to `RtlDetermineDosPathNameType_U` and the full path, those APIs think it's a rooted path, not a Root Local Device path. It would be easy to imagine a scenario where this could be abused. I'll give one such example later.

Another thing to note is that using this type of path allows you to specify characters that would normally be considered illegal in a path. Every file system has some sort of limit on what characters it's willing to accept as valid. This is usually for ease of use, such as not allowing NUL characters where your API is based on C-style terminated strings. The two most common file systems used on NT systems, NTFS and FAT, have considerably more limitations on valid characters as we can see in the following table, where anything red is banned from normal filenames.

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|----|----|----|-----|
| 0 | NUL | SOH | STX | ETX | EOT | ENQ | ACK | BEL | BS | TAB | LF | VT | FF | CR | SO | SI |
| 1 | DLE | DC1 | DC2 | DC3 | DC4 | NAK | SYN | ETB | CAN | EM | SUB | ESC | FS | GS | RS | US |
| 2 | SP | ! | " | # | \$ | % | & | ' | (|) | * | + | , | - | . | / |
| 3 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | : | ; | < | = | > | ? |
| 4 | @ | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
| 5 | P | Q | R | S | T | U | V | W | X | Y | Z | [| \ |] | ^ | _ |
| 6 | ` | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o |
| 7 | p | q | r | s | t | u | v | w | x | y | z | { | | } | ~ | DEL |

While it's true that an NTFS/FAT path on disk cannot contain illegal characters, at least added directly via the OS, there's nothing to stop a path from containing these characters as long as they don't end up hitting the NTFS driver. For example, if you get the object manager into the mix (through redirecting via a mount point for example) only the following characters are illegal in the object manager:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|----|----|----|-----|
| 0 | NUL | SOH | STX | ETX | EOT | ENQ | ACK | BEL | BS | TAB | LF | VT | FF | CR | SO | SI |
| 1 | DLE | DC1 | DC2 | DC3 | DC4 | NAK | SYN | ETB | CAN | EM | SUB | ESC | FS | GS | RS | US |
| 2 | SP | ! | " | # | \$ | % | & | ' | (|) | * | + | , | - | . | / |
| 3 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | : | ; | < | = | > | ? |
| 4 | @ | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
| 5 | P | Q | R | S | T | U | V | W | X | Y | Z | [| \ |] | ^ | _ |
| 6 | ` | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o |
| 7 | p | q | r | s | t | u | v | w | x | y | z | { | | } | ~ | DEL |

Quite a difference. Note that even NUL is valid as the NT kernel uses counted strings. The backslash is only invalid because without that there'd be no path separator. Obviously outside of NUL all these characters can be put into a Rooted Local Device path.

Also the canonicalization behavior works to our advantage. On systems such as Linux or OSX, the canonicalization is done during the process of opening the file inside the kernel. Therefore the directory entries '.' and '..' really exist (or at least are faked sufficiently so they exist). However, as we've seen on Windows, the canonicalization is done in user-mode before passing to the kernel, and for the most part no verification is done on that path as to whether it exists. So, for example, if you specify invalid characters as part of a string, they can be removed before the file is actually checked.

```

C:\WINDOWS\system32\cmd.exe
C:\test>ConvertDosPathToMtPath.exe x:\abc\?*<>|..\xyz
Converting: 'x:\abc\?*<>|..\xyz'
To:        '\??\x:\abc\xyz'
Type:      RtlPathTypeDriveAbsolute
FileName:  xyz
FullPathName: 'x:\abc\xyz'

C:\test>

```

As a final note, the documentation for `CreateFile`, for example, explicitly states “In the ANSI version of this function, the name is limited to `MAX_PATH` characters. To extend this limit to 32,767 wide characters, call the Unicode version of the function and prepend “\\?” to the path.” Therefore you’d make the assumption that calling the ANSI version can’t take the Rooted Local Device prefix, or if it can, it can’t support long paths.

This is demonstrably false. While you still need to specify the \\? or similar prefixes otherwise `RtlDosPathNameToNtPathName` gets unhappy, it’ll still work as no API checks the length of the ANSI string before converting to Unicode (and subsequently calling the Unicode version of the API). We can also understand why the API is limited to 32,767 characters, as the underlying NT `UNICODE_STRING` counted string structure represents the length of a string as a 16 bit integer. Because it stores it as a byte rather than character count, we can only store at most $2^{15}-1$ characters, which just happens to be 32,767.

Bypassing Device and UNC Path Checks

Let’s finish up with an overview of how some of the quirks I’ve described can be used to attack real world applications either by tricking the code into opening the wrong file/device or by bypassing path checks. Sometimes an application will allow you to specify a path, but try and restrict which types of resources you access. Common restrictions are blocking access to UNC paths and named pipes. Take for example the UNC case, where a really naïve check would be something like the following:

```
BOOL IsUncPath(LPCWSTR Path) {
    if (wcslen(Path) > 2) {
        return (Path[0] == '\\ ' || Path[0] == '/')
            && (Path[1] == '\\ ' || Path[1] == '/');
    }
    return FALSE;
}
```

At least it checks for forward slashes (otherwise the check is pretty easy to bypass). But it also excludes us from using the \\?\UNC form as it still looks like a UNC path to this check function. So instead we can use the \\?*\UNC prefix, which works just as well and circumvents the check. Of course some checks try to be more clever. For example, the `SHLWAPI` function `PathIsUNC` is available on all Windows systems so it would make sense to call it and you’d assume it should handle all cases. It does have some weird behaviour though:

| Path Specified | Result |
|----------------|--------|
|----------------|--------|

| | |
|-----------------|-------|
| \\abc\xyz | TRUE |
| C:\abc\xyz | FALSE |
| \\.C:\abc\xyz | TRUE |
| \\?\C:\abc\xyz | FALSE |
| \\?\UNC\abc\xyz | TRUE |
| \??\UNC\abc\xyz | FALSE |

Again I've highlighted the odd ones. The Local Device path is always considered a UNC path even though it isn't. On the other hand, the Root Local Device path isn't considered a UNC path unless it's followed by UNC. The alternative form of the Root Local Device path isn't considered a UNC path either.

As an aside, even with the \\?\ version you can bypass the check in PathIsUNC by exploiting the object manager, specifically the standard GLOBALROOT symbolic link, so you use paths like:

| Path Specified | Result |
|--|--------|
| \\?\UNC\abc\xyz | TRUE |
| \\?\GLOBALROOT\??\UNC\abc\xyz | FALSE |
| \\?\GLOBALROOT\DosDevices\UNC\abc\xyz | FALSE |
| \\?\GLOBALROOT\Device\Mup\abc\xyz | FALSE |
| \\?\GLOBALROOT\Device\LanManRedirector\abc\xyz | FALSE |
| \\?\GLOBALROOT\Device\WebDavRedirector\abc\xyz | FALSE |

The last two rows show you how to use this path to explicitly specify either SMB or WebDAV protocols. While we're on the subject, if you look at the LanManRedirector or WebDavRedirector entries in WinObj you'll find them to be symbolic links to \Device\Mup\;NAME where NAME is the name of the entry. Now think back to how UNC Absolute paths are converted from Win32 to NT: the leading path separators are replaced with \??\UNC. As UNC itself is a symbolic link to \Device\Mup you can specify paths like:

| Path | Final Result after Symbolic Link Resolving |
|----------------------------------|--|
| \\;LanmanRedirector\evil.com\xyz | \Device\Mup\;LanmanRedirector\evil.com\xyz |
| \\;WebDavRedirector\evil.com\xyz | \Device\Mup\;WebDavRedirector\evil.com\xyz |

This will no doubt confuse a parser into thinking you're trying to access share evil.com on server ;LanmanRedirector instead of share xyz on evil.com. Fortunately this doesn't seem to work in the Explorer shell, but it does work when passed to native APIs such as CreateFile. For extra bonus points it also breaks the canonicalization; normally you can't canonicalize above the share name but in this case the implementation doesn't realize xyz is a share name (it's just as confused) so will allow it to be canonicalized away. There's even more weirdness with UNC paths, but that's perhaps for another time.

Conclusions

I hope you've seen that Win32 paths are massively more complex than it would seem even from just reading the documentation. There are so many quirks and weird behaviors it's very difficult to write code that validates all possible outcomes. Converting everything to NT paths helps slightly as they are less prone to misbehavior, but even then through symbolic link abuse or changing drive letters it's still possible to be confused. If you ever encounter an application trying to validate a Win32 path, be very skeptical and try and use some of these techniques to bypass the checks.

Example Program

Here's the simple example program which uses all the various API functions on a string passed on the command line. This will allow you to test each example to prove the results. I'll use C# as it's a lot easier to call into NTDLL functions without needing a library or messing around with GetProcAddress. As a bonus every version of Windows since Vista has a version of .NET installed by default which includes the C# CSC compiler so no need to install a C compiler or Python.

```
using System;
using System.ComponentModel;
using System.Runtime.InteropServices;
using System.Text;

class Program
{
    [StructLayout(LayoutKind.Sequential)]
```

```

struct UNICODE_STRING
{
    public ushort Length;
    public ushort MaximumLength;
    public IntPtr Buffer;

    public override string ToString()
    {
        if (Buffer != IntPtr.Zero)
            return Marshal.PtrToStringUni(Buffer, Length / 2);
        return "(null)";
    }
}

[StructLayout(LayoutKind.Sequential)]
class RTL_RELATIVE_NAME
{
    public UNICODE_STRING RelativeName;
    public IntPtr ContainingDirectory;
    public IntPtr CurDirRef;
}

[DllImport("ntdll.dll", CharSet = CharSet.Unicode)]
static extern int RtlDosPathNameToRelativeNtPathName_U_WithStatus(
    string DosFileName,
    out UNICODE_STRING NtFileName,
    out IntPtr ShortPath,
    [Out] RTL_RELATIVE_NAME RelativeName
);

enum RTL_PATH_TYPE
{
    RtlPathTypeUnknown,
    RtlPathTypeUncAbsolute,
    RtlPathTypeDriveAbsolute,
    RtlPathTypeDriveRelative,
    RtlPathTypeRooted,
    RtlPathTypeRelative,
    RtlPathTypeLocalDevice,
    RtlPathTypeRootLocalDevice
}

```

```

[DllImport("ntdll.dll", CharSet = CharSet.Unicode)]
static extern RTL_PATH_TYPE RtlDetermineDosPathNameType_U(string Path);

[DllImport("ntdll.dll", CharSet = CharSet.Unicode)]
static extern int RtlGetFullPathName_UEx(
    string FileName,
    int BufferLength,
    [Out] StringBuilder Buffer,
    IntPtr FilePart,
    out int FinalLength);

[DllImport("ntdll.dll")]
static extern int RtlNtStatusToDosError(int NtStatus);

static void PrintStatus(int status)
{
    Console.WriteLine("Error:    {0}",
        new Win32Exception(RtlNtStatusToDosError(status)).Message);
}

static void ConvertPath(string path)
{
    Console.WriteLine("Converting: '{0}'", path);
    UNICODE_STRING nname = new UNICODE_STRING();
    IntPtr filename = IntPtr.Zero;
    RTL_RELATIVE_NAME relative_name = new RTL_RELATIVE_NAME();
    int status = RtlDosPathNameToRelativeNtPathName_U_WithStatus(
        path,
        out nname,
        out filename,
        relative_name);
    if (status == 0)
    {
        Console.WriteLine("To:      '{0}'",
            nname);
        Console.WriteLine("Type:    {0}",
            RtlDetermineDosPathNameType_U(path));
        Console.WriteLine("FileName: {0}",
            Marshal.PtrToStringUni(filename));
        if (relative_name.RelativeName.Length > 0)
        {
            Console.WriteLine("RelativeName: '{0}'",
                relative_name.RelativeName);
        }
    }
}

```

```

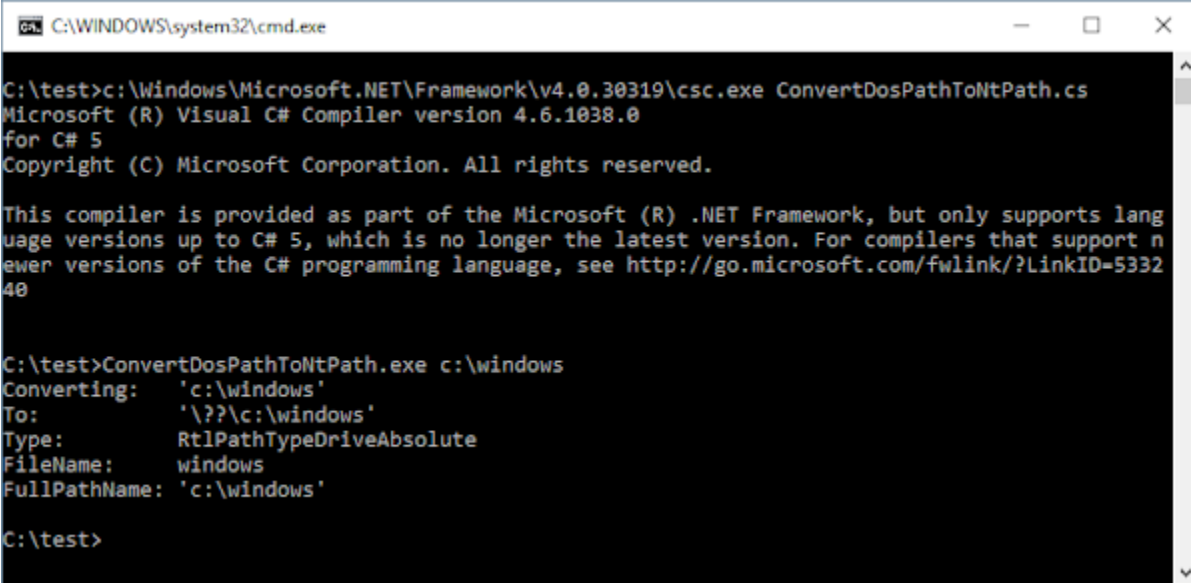
    Console.WriteLine("Directory: 0x{0:X}",
        relative_name.ContainingDirectory.ToInt64());
    Console.WriteLine("CurDirRef: 0x{0:X}",
        relative_name.CurDirRef.ToInt64());
}
}
else
{
    PrintStatus(status);
}

int length = 0;
StringBuilder builder = new StringBuilder(260);
status = RtlGetFullPathName_UEx(
    path,
    builder.Capacity * 2,
    builder,
    IntPtr.Zero,
    out length);
if (status == 0)
{
    Console.WriteLine("FullPathName: '{0}'",
        builder.ToString());
}
else
{
    PrintStatus(status);
}
}

static void Main(string[] args)
{
    if (args.Length < 1)
    {
        Console.WriteLine("Usage: ConvertDosPathToNtPath DosPath");
    }
    else
    {
        ConvertPath(args[0]);
    }
}
}
}

```

Let's just try it and see if it works:



```
C:\WINDOWS\system32\cmd.exe
C:\test>c:\Windows\Microsoft.NET\Framework\v4.0.30319\csc.exe ConvertDosPathToNtPath.cs
Microsoft (R) Visual C# Compiler version 4.6.1038.0
for C# 5
Copyright (C) Microsoft Corporation. All rights reserved.

This compiler is provided as part of the Microsoft (R) .NET Framework, but only supports lang
uage versions up to C# 5, which is no longer the latest version. For compilers that support n
ewer versions of the C# programming language, see http://go.microsoft.com/fwlink/?LinkID=533240

C:\test>ConvertDosPathToNtPath.exe c:\windows
Converting: 'c:\windows'
To:        '??\c:\windows'
Type:      RtlPathTypeDriveAbsolute
FileName:  windows
FullPathName: 'c:\windows'

C:\test>
```