



More ▾

[Create Blog](#) [Sign In](#)

# Project Zero

News and updates from the Project Zero team at Google

Thursday, March 14, 2019

## Windows Kernel Logic Bug Class: Access Mode Mismatch in IO Manager

Posted by James Forshaw, Project Zero

This blog post is an in-depth look at an interesting logic bug class in the Windows Kernel and what I did to try to get it fixed with our partners at Microsoft. The maximum impact of the bug class is local privilege escalation if kernel and driver developers don't take into account how the IO manager operates when accessing device objects. This blog discusses how I discovered the bug class and the technical background. For more information about the further investigation, fixing and avoiding writing new code with the bug class refer to MSRC's [blog post](#).

## Technical Background

I first stumbled upon the bug class while trying to exploit [issue 779](#). This issue was a file TOCTOU which bypassed the [custom font loading mitigation policy](#). The mitigation policy was introduced in Windows 10 to limit the impact of exploitable font memory corruption vulnerabilities. Normally it'd be trivial to exploit a file TOCTOU issue using a combination of file and Object Manager symbolic links. An exploit using symbolic links worked as a normal user, but not inside a sandbox. Rather than spend too much time on this low-impact issue, I exploited it without symbolic links using Shadow Object Directories and Microsoft fixed it as [CVE-2016-3219](#). I added a note of the unexpected behavior to my list of topics to follow up on at a later date.

Search This Blog

Search

Pages

- [Working at Project Zero](#)
- [0day "In the Wild"](#)
- [Vulnerability Disclosure FAQ](#)

Archives

2019

- [A very deep dive into iOS Exploit chains found in ...](#) (Aug)
- [In-the-wild iOS Exploit Chain 1](#) (Aug)
- [In-the-wild iOS Exploit Chain 2](#) (Aug)
- [In-the-wild iOS Exploit Chain 3](#) (Aug)
- [In-the-wild iOS Exploit Chain 4](#) (Aug)
- [In-the-wild iOS Exploit Chain 5](#) (Aug)
- [Implant Teardown](#) (Aug)
- [JSC Exploits](#) (Aug)
- [The Many Possibilities of CVE-2019-8646](#) (Aug)

Fast forward over a year I decided to go back and take a look at the unexpected behavior in more depth. The code which was failing was similar to the following:

```
HANDLE OpenFilePath(LPCWSTR pwzPath) {
    UNICODE_STRING Path;
    OBJECT_ATTRIBUTES ObjectAttributes;
    HANDLE FileHandle;
    NTSTATUS status;

    RtlInitUnicodeString(&Path, pwzPath);
    InitializeObjectAttributes(&ObjectAttributes,
                              &Path,
                              OBJ_KERNEL_HANDLE | OBJ_CASE_INSENSITIVE);

    status = IoCreateFile(
        &FileHandle,
        GENERIC_READ,
        &ObjectAttributes,
        // ...
        FILE_OPEN,
        FILE_NON_DIRECTORY_FILE,
        // ...
        IO_NO_PARAMETER_CHECKING | IO_FORCE_ACCESS_CHECK);

    if (NT_ERROR(status))
        return NULL;
    return FileHandle;
}
```

When this code tries to open a file with an Object Manager symbolic link in the path the call to [IoCreateFile](#) failed with `STATUS_OBJECT_NAME_NOT_FOUND`. Further digging led me to discover the source of the error in `ObpParseSymbolicLink`, which looks like the following:

```
NTSTATUS ObpParseSymbolicLink(POBJECT_SYMBOLIC_LINK Object,
                             PACCESS_STATE AccessState,
                             KPROCESSOR_MODE AccessMode) {
    if (Object->Flags & SANDBOX_FLAG
```

- [Down the Rabbit-Hole...](#) (Aug)
- [The Fully Remote Attack Surface of the iPhone](#) (Aug)
- [Trashing the Flow of Data](#) (May)
- [Windows Exploitation Tricks: Abusing the User-Mode...](#) (Apr)
- [Virtually Unlimited Memory: Escaping the Chrome Sa...](#) (Apr)
- [Splitting atoms in XNU](#) (Apr)
- [Windows Kernel Logic Bug Class: Access Mode Mismat...](#) (Mar)
- [Android Messaging: A Few Bugs Short of a Chain](#) (Mar)
- [The Curious Case of Convexity Confusion](#) (Feb)
- [Examining Pointer Authentication on the iPhone XS](#) (Feb)
- [voucher\\_swap: Exploiting MIG reference counting in...](#) (Jan)
- [Taking a page from the kernel's book: A TLB issue ...](#) (Jan)

---

## 2018

- [On VBScript](#) (Dec)
- [Searching statically-linked vulnerable library fun...](#) (Dec)
- [Adventures in Video Conferencing Part 5: Where Do ...](#) (Dec)
- [Adventures in Video Conferencing Part 4: What Didn...](#) (Dec)
- [Adventures in Video Conferencing Part 3: The Even ...](#) (Dec)
- [Adventures in Video Conferencing Part 2: Fun with ...](#) (Dec)
- [Adventures in Video Conferencing Part 1: The Wild ...](#) (Dec)
- [Injecting Code into Windows Protected Processes us...](#) (Nov)

```

    && !RtlIsSandboxedToken(AccessState->SubjectSecurityContext, AccessMode))
    return STATUS_OBJECT_NAME_NOT_FOUND;

// ...
}

```

The failing check is part of the [symbolic link mitigations](#) Microsoft introduced in Windows 10. I was creating the symbolic link inside a sandbox which would set `SANDBOX_FLAG` in the object's structure. When parsing the symbolic link while opening the font file this check is made. With the sandbox flag set the kernel also calls `RtlIsSandboxedToken` to determine if the caller is still inside a sandbox. As the call to open the font file is in a sandboxed process thread `RtlIsSandboxedToken` should return `TRUE`, and the function would continue. Instead it was returning `FALSE`, which made the kernel think the call was coming from a more privileged process and returns `STATUS_OBJECT_NAME_NOT_FOUND` to mitigate any exploitation.

At this point I understood how my exploit was failing, and yet I didn't understand why. Specifically I didn't understand why `RtlIsSandboxToken` was returning `FALSE`. Digging into the function gave me an important insight:

```

BOOLEAN RtlIsSandboxedToken(PSECURITY_SUBJECT_CONTEXT SubjectSecurityContext,
                             KPROCESSOR_MODE AccessMode) {
    NTSTATUS AccessStatus;
    ACCESS_MASK GrantedAccess;

    if (AccessMode == KernelMode)
        return FALSE;

    if (SeAccessCheck(
        SeMediumDaclSd,
        SubjectSecurityContext,
        FALSE,
        READ_CONTROL,
        0,
        NULL,
        &RtlpRestrictedMapping,
        AccessMode,
        &GrantedAccess,

```

- [Heap Feng Shader: Exploiting SwiftShader in Chrome...](#) (Oct)
- [Deja-XNU](#) (Oct)
- [Injecting Code into Windows Protected Processes us...](#) (Oct)
- [365 Days Later: Finding and Exploiting Safari Bugs...](#) (Oct)
- [A cache invalidation bug in Linux memory managemen...](#) (Sep)
- [OATmeal on the Universal Cereal Bus: Exploiting An...](#) (Sep)
- [The Problems and Promise of WebAssembly](#) (Aug)
- [Windows Exploitation Tricks: Exploiting Arbitrary ...](#) (Aug)
- [Adventures in vulnerability reporting](#) (Aug)
- [Drawing Outside the Box: Precision Issues in Graph...](#) (Jul)
- [Detecting Kernel Memory Disclosure - Whitepaper](#) (Jun)
- [Bypassing Mitigations by Attacking JIT Server in M...](#) (May)
- [Windows Exploitation Tricks: Exploiting Arbitrary ...](#) (Apr)
- [Reading privileged memory with a side-channel](#) (Jan)

## 2017

- [aPAColypse now: Exploiting Windows 10 in a Local N...](#) (Dec)
- [Over The Air - Vol. 2, Pt. 3: Exploiting The Wi-Fi...](#) (Oct)
- [Using Binary Diffing to Discover Windows Kernel Me...](#) (Oct)
- [Over The Air - Vol. 2, Pt. 2: Exploiting The Wi-Fi...](#) (Oct)
- [Over The Air - Vol. 2, Pt. 1: Exploiting The Wi-Fi...](#) (Sep)

```

        &AccessStatus)) {
    return FALSE;
}

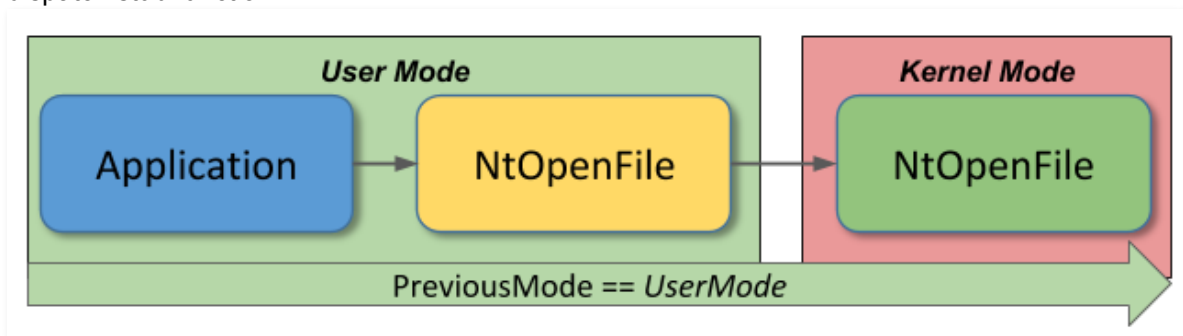
return TRUE;
}

```

The important parameter is *AccessMode*, which is of type *KPROCESSOR\_MODE* and can be set to one of two values *UserMode* or *KernelMode*. If the *AccessMode* parameter was set to the value *KernelMode* then the function would automatically return FALSE indicating the current caller is not in a sandbox. Breakpointing on this function in a kernel debugger confirmed that *AccessMode* was set to *KernelMode* when being called from my exploit. If this parameter was always set to *KernelMode* how would *RtlIsSandboxToken* ever return TRUE? To understand how the kernel is functioning let's go into more depth on what the *AccessMode* parameter represents.

## Previous Access Mode

Every thread in Windows has a [previous access mode](#) associated with it. This previous access mode is stored in the *PreviousMode* member of the *KTHREAD* structure. The member is accessed by third-parties using [ExGetPreviousMode](#) and returns the *KPROCESSOR\_MODE* type. The previous access mode is set to *UserMode* if a user mode thread is running kernel code due to a system call transition. As an example the following diagram shows a call from a user mode application to the system call *NtOpenFile*, via a system call dispatch stub function in NTDLL.



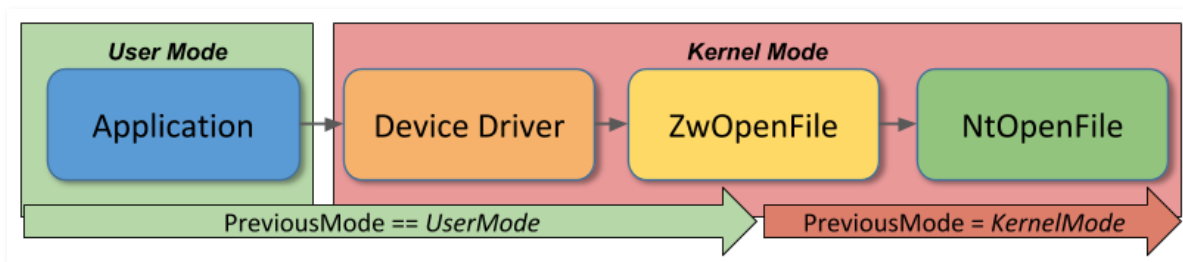
Note how the previous mode is always set to *UserMode* even when code is executing inside the *NtOpenFile* system call in the kernel memory space. In contrast, *KernelMode* is set if the thread is a system thread (in the System process) or due to a system call transition from kernel mode. The following diagram shows the

- [The Great DOM Fuzz-off of 2017](#) (Sep)
- [Bypassing VirtualBox Process Hardening on Windows](#) (Aug)
- [Windows Exploitation Tricks: Arbitrary Directory C...](#) (Aug)
- [Trust Issues: Exploiting TrustZone TEEs](#) (Jul)
- [Exploiting the Linux kernel via packet sockets](#) (May)
- [Exploiting .NET Managed DCOM](#) (Apr)
- [Exception-oriented exploitation on iOS](#) (Apr)
- [Over The Air: Exploiting Broadcom's Wi-Fi Stack \(P...](#) (Apr)
- [Notes on Windows Uniscribe Fuzzing](#) (Apr)
- [Pandavirtualization: Exploiting the Xen hypervisor...](#) (Apr)
- [Over The Air: Exploiting Broadcom's Wi-Fi Stack \(P...](#) (Apr)
- [Project Zero Prize Conclusion](#) (Mar)
- [Attacking the Windows NVIDIA Driver](#) (Feb)
- [Lifting the \(Hyper\) Visor: Bypassing Samsung's Rea...](#) (Feb)

## 2016

- [Chrome OS exploit: one byte overflow and symlinks](#) (Dec)
- [BitUnmap: Attacking Android Ashmem](#) (Dec)
- [Breaking the Chain](#) (Nov)
- [task\\_t considered harmful](#) (Oct)
- [Announcing the Project Zero Prize](#) (Sep)
- [Return to libstagefright: exploiting libutils on A...](#) (Sep)
- [A Shadow of our Former Self](#) (Aug)

transition when a device driver (which is already running in kernel mode) calls the *ZwOpenFile* system call, which results in executing *NtOpenFile*.



In the diagram a user mode application calls a function inside a device driver, for example using the *NtFsControlFile* system call. The previous mode equals *UserMode* during the call to the device driver. However, if the device driver calls *ZwOpenFile* the kernel simulates a system call transition, this results in the previous mode being changed to *KernelMode* and the *NtOpenFile* system call code being executed.

From a security perspective the previous access mode influences two important but fundamentally different security checks in the kernel, Security Access Checking (SecAC) and Memory Access Checking (MemAC). SecAC is used for calls to APIs exposed by the [Security Reference Monitor](#) such as [SeAccessCheck](#) or [SePrivilegeCheck](#). These security APIs are used to determine if the caller has rights to access a resource. Commonly the APIs take an *AccessMode* parameter, if this parameter is *KernelMode* then the access checks pass automatically, which might be a security vulnerability if the resource couldn't normally be accessed by a user. We already saw this use case in *RtlIsSandboxToken*, the API checked explicitly for *AccessMode* being *KernelMode* and returned FALSE. Even without the shortcut, by passing *KernelMode* to *SeAccessCheck* the call will succeed regardless of the caller's access token and *RtlIsSandboxToken* would have returned FALSE.

MemAC is used to ensure the user application can't pass pointers to a kernel address location. If the *AccessMode* is *UserMode* then all memory addresses passed to the system call/operation should be verified that they're less than *MmUserProbeAddress* or via functions such as *ProbeForRead/ProbeForWrite*. Again if this checking is incorrect privilege escalation could occur as the user could trick kernel code into reading or writing into privileged kernel memory locations. Note that not all kernel APIs perform MemAC, for example the *SeAccessCheck* assumes that the caller has already checked parameters, the *AccessMode* parameter is only used to determine whether to bypass the security check.

Storing the previous access mode on the thread creates a problem as there's no way of differentiating between SecAC and MemAC for a kernel API. It's possible for an API to disable SecAC intentionally and

- [A year of Windows kernel font fuzzing #2: the tech...](#) (Jul)
- [How to Compromise the Enterprise Endpoint](#) (Jun)
- [A year of Windows kernel font fuzzing #1: the resu...](#) (Jun)
- [Exploiting Recursion in the Linux Kernel](#) (Jun)
- [Life After the Isolated Heap](#) (Mar)
- [Race you to the kernel!](#) (Mar)
- [Exploiting a Leaked Thread Handle](#) (Mar)
- [The Definitive Guide on Win32 to NT Path Conversio...](#) (Feb)
- [Racing MIDI messages in Chrome](#) (Feb)
- [Raising the Dead](#) (Jan)

## 2015

- [FireEye Exploitation: Project Zero's Vulnerability...](#) (Dec)
- [Between a Rock and a Hard Link](#) (Dec)
- [Windows Sandbox Attack Surface Analysis](#) (Nov)
- [Hack The Galaxy: Hunting Bugs in the Samsung Galax...](#) (Nov)
- [Windows Drivers are True'ly Tricky](#) (Oct)
- [Revisiting Apple IPC: \(1\) Distributed Objects](#) (Sep)
- [Kaspersky: Mo Unpackers, Mo Problems.](#) (Sep)
- [Stagefrightened?](#) (Sep)
- [Enabling QR codes in Internet Explorer, or a story...](#) (Sep)
- [Windows 10^H^H Symbolic Link Mitigations](#) (Aug)
- [One font vulnerability to rule them all #4: Window...](#) (Aug)

disable MemAC accidentally resulting in security issues and vice-versa. Let's look in more detail how the IO Manager tries to solve this mismatched access checking issue.

## IO Manager Access Checking

The IO Manager exposes two main API groups for directly accessing files. The first APIs are the system calls *NtCreateFile/ZwCreateFile* or *NtOpenFile/ZwOpenFile*. The system calls are primarily for use by user-mode applications, but can be called from kernel mode if needed. The other APIs are exposed only for kernel mode callers, *IoCreateFile* and [IoCreateFileEx](#).

If you compare the implementations of the two main APIs you find they're simple forwarding wrappers around the internal function *lopCreateFile*. By default *lopCreateFile* uses the current thread's previous mode to determine whether to perform MemAC and SecAC. For example when calling *lopCreateFile* via *NtCreateFile* from a user-mode process the kernel performs MemAC and SecAC as the previous mode will be *UserMode*. If kernel-mode calls *ZwCreateFile* then the previous mode is set to *KernelMode* and both SecAC and MemAC are disabled.

*IoCreateFile* can only be called from kernel mode code and there's no syscall transition involved, therefore any calls will use whatever previous mode is set on the thread. If *IoCreateFile* is called from a thread with previous mode set to *UserMode* this means that SecAC and MemAC will be performed. Enforcing MemAC is especially problematic as it means that the kernel code can't pass kernel mode pointers to *IoCreateFile* which would make the API very difficult to use. However the caller of *IoCreateFile* can't just change the thread's previous mode to *KernelMode* as then SecAC would be disabled.

*IoCreateFile* solves this problem by specifying special flags that can be passed via the *Options* parameter. This parameter is forwarded to *lopCreateFile* but is not exposed through the *NtCreateFile* system call. Going back to our font issue, WIN32K is calling *IoCreateFile* and passing the option flags *IO\_NO\_PARAMETER\_CHECKING* (*INPC*) and *IO\_FORCE\_ACCESS\_CHECK* (*IFAC*).

*INPC* is documented as:

*"[If specified] the parameters for this call should not be validated before attempting to issue the create request. Driver writers should use this flag with caution as certain invalid parameters can cause a system failure. For more information, see Remarks."*

In the remarks section it expands further:

- [Three bypasses and a fix for one of Flash's Vector...](#) (Aug)
- [Attacking ECMAScript Engines with Redefinition](#) (Aug)
- [One font vulnerability to rule them all #3: Window...](#) (Aug)
- [One font vulnerability to rule them all #2: Adobe ...](#) (Aug)
- [One font vulnerability to rule them all #1: Introd...](#) (Jul)
- [One Perfect Bug: Exploiting Type Confusion in Flas...](#) (Jul)
- [Significant Flash exploit mitigations are live in ...](#) (Jul)
- [From inter to intra: gaining reliability](#) (Jul)
- [When 'int' is the new 'short'](#) (Jul)
- [What is a 'good' memory corruption vulnerability?](#) (Jun)
- [Analysis and Exploitation of an ESET Vulnerability...](#) (Jun)
- [Owning Internet Printing - A Case Study in Modern ...](#) (Jun)
- [Dude, where's my heap?](#) (Jun)
- [In-Console-Able](#) (May)
- [A Tale of Two Exploits](#) (Apr)
- [Taming the wild copy: Parallel Thread Corruption](#) (Mar)
- [Exploiting the DRAM rowhammer bug to gain kernel p...](#) (Mar)
- [Feedback and data-driven updates to Google's discl...](#) (Feb)
- [\(^Exploiting\)\s\\*\(CVE-2015-0318\)\s\\*\(in\)\s\\*\(Flash\\$\)](#) (Feb)
- [A Token's Tale](#) (Feb)
- [Exploiting NVMAP to escape the Chrome sandbox - CV...](#) (Jan)
- [Finding and exploiting ntpd vulnerabilities](#) (Jan)



*“The Options `IO_NO_PARAMETER_CHECKING` flag can be useful if a driver is issuing a kernel-mode create request on behalf of an operation initiated by a user-mode application. Because the request occurs within a user-mode context, the I/O manager, by default, probes the supplied parameter values, which can cause an access violation if the parameters are kernel-mode addresses. This flag enables the caller to override this default behavior and avoid the access violation.”*

This makes its purpose clear, it disables MemAC, allowing the kernel code to pass pointers into kernel memory as function parameters. As a byproduct it also disables most of the validation of the parameters, such as incompatible flag combinations being checked. There is a separate, not properly documented, flag `IO_CHECK_CREATE_PARAMETERS` which turns back on just parameter flag checking, but not MemAC.

*IFAC* on the other hand is documented as:

*“The I/O manager must check the create request against the file's security descriptor.”*

This implies that the flag reenables SecAC. It makes sense if the caller was a system thread with previous mode set to *KernelMode* but why would we need to re-enable SecAC if we're calling from *UserMode*? Here in lies the seeds to understanding the original unexpected behavior, as we can see in some simplified code from *IopCreateFile*.

```
NTSTATUS IopCreateFile(PHANDLE FileHandle, ACCESS_MASK DesiredAccess,
                    POBJECT_ATTRIBUTES ObjectAttributes, ...,
                    ULONG Options) {
    KPROCESSOR_MODE AccessMode;

    if (Options & IO_NO_PARAMETER_CHECKING) {
        AccessMode = KernelMode;
    } else {
        AccessMode = KeGetCurrentThread()->PreviousMode;
    }

    FILE_PARSE_CONTEXT ParseContext = {};
    // Initialize other values
    ParseContext->Options = Options;

    return ObOpenObjectByName(
        ObjectAttributes,
        IoFileObjectType,
```

## 2014

- [Internet Explorer EPM Sandbox Escape CVE-2014-6350...](#) (Dec)
- [pwn4fun Spring 2014 - Safari - Part II](#) (Nov)
- [Project Zero Patch Tuesday roundup, November 2014](#) (Nov)
- [Did the “Man With No Name” Feel Insecure?](#) (Oct)
- [More Mac OS X and iPhone sandbox escapes and kerne...](#) (Oct)
- [Exploiting CVE-2014-0556 in Flash](#) (Sep)
- [The poisoned NUL byte, 2014 edition](#) (Aug)
- [What does a pointer look like, anyway?](#) (Aug)
- [Mac OS X and iPhone sandbox escapes](#) (Jul)
- [pwn4fun Spring 2014 - Safari - Part I](#) (Jul)
- [Announcing Project Zero](#) (Jul)

```
    AccessMode,  
    NULL,  
    DesiredAccess,  
    &ParseContext,  
    &FileHandle);  
}
```

This code shows that if *INPC* is specified then the *AccessMode* for all subsequent calls is set to *KernelMode*. Therefore specifying that option disables not just MemAC but also SecAC. It's worth noting that the thread's previous mode is not changed, just the *AccessMode* value which is passed forward to *ObOpenObjectByName*. *IoCreateFile* is delegating pointer checking to the Object Manager, therefore the only way it can achieve this is to kill all checking. Crucially *IFAC* is not checked, it's only passed forward inside the parsing context structure, something for the rest of the IO manager to deal with.

This isn't the end of the story just yet, it's also possible to call *ZwCreateFile* and pass the special flag *OBJ\_FORCE\_ACCESS\_CHECK* (*OFAC*) inside the *OBJECT\_ATTRIBUTES* structure and ensure access checking is performed, even if the previous access mode is set to *KernelMode*. As you can't pass in *IFAC* via *ZwCreateFile* and there's no checking for the *OFAC* flag in *IoCreateFile*, it must be in *ObOpenObjectByName*. Actually it's slightly deeper, first all the parameters are processed based on the *AccessMode* passed to *ObOpenObjectByName*, then *ObpLookupObjectName* is called which checks the *OFAC* flag, if it's set then *AccessMode* is forced back to *UserMode*.

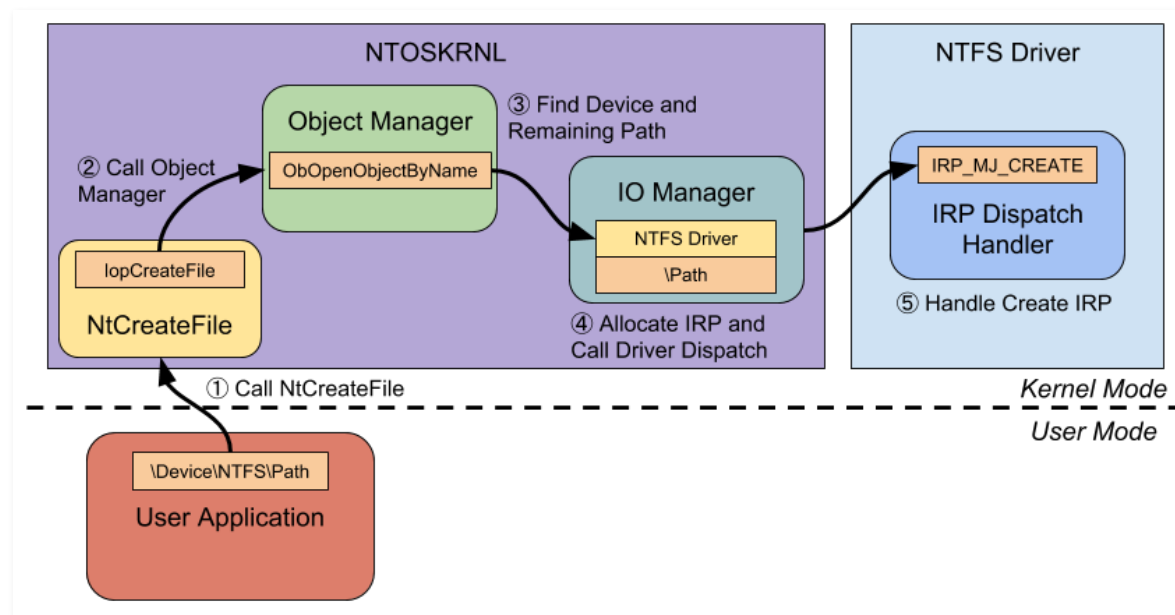
We can now finally understand why we got the unexpected behavior with opening the font file. Parsing the symbolic link happens inside the Object Manager, not the IO Manager, therefore has no knowledge of the *IFAC* flag. *IoCreateFile* has told the Object Manager to do all checks as if the previous access mode is *KernelMode*, so that's the value passed to *ObpParseSymbolicLink*, which gets passed to *RtlIsSandboxToken* which rightly indicates it's not running in a sandboxed process. However, once the file is actually opened the *IFAC* flag kicks in and ensures SecAC is still performed on the file. If the caller had also specified *OFAC* then the symbolic link would have worked, as the parsing would occur during the lookup operation which has been forced to *UserMode*.

This in itself is an interesting result, basically any operation which is called during the Object Manager parsing operation which trusts the value of *AccessMode* will disable security checks unless *OFAC* has been specified. However, that's not the bug that this blog is about, for that we need to go even deeper to work out how *IFAC* works inside the IO Manager.



## IO Device Parsing

The Object Manager's responsibility for opening a file ends once it finds a named device object in the Object Manager namespace. The Object Manager looks up the parse function for the Device type, which is *IoParseDevice*, then passes all the information it knows about. This includes the *AccessMode* value, which we know is set to *KernelMode*, the remaining path to parse and the parsing context buffer which includes the *Options* parameter. The *IoParseDevice* function does some security checks of its own, such as checking device traversal, allocates a new [IO Request Packet](#) (IRP) and calls the driver responsible for the device object.



The IRP structure contains a *RequestorMode* field which reflects the *AccessMode* of the file operation. This reason to have a *RequestorMode* field is the IRP can be dispatched asynchronously. The thread which processes the IO operation might not be the thread which started the IO operation. You might guess now that this is where *IFAC* comes into play, perhaps the IO manager sets *RequestorMode* to *UserMode*? If you actually check this inside a kernel driver when accessed from *IoCreateFile* using *INPC* you'd find this field is still set to *KernelMode*, so it's not the answer.

The operation type being performed and the operation's specific parameters are passed in the [IO Stack Location](#) structure which is located immediately after the IRP structure. In the case of opening a file the major operation type is [IRP\\_MJ\\_CREATE](#) and uses the *Create* union field of the *IO\_STACK\_LOCATION*

structure. This is where *IFAC* comes in, if the flag is specified to *IoCreateFile* then in the IO Stack Location's *Flags* parameter, the new flag *SL\_FORCE\_ACCESS\_CHECK* (*SFAC*) will be set. It's up to the file system driver to ensure it verifies this flag, and not rely on the *RequestorMode* being set to *UserMode*. The NTFS driver knows this, and has the following code:

```
KPROCESSOR_MODE NtfsEffectiveMode(PIRP Irp) {
    PIO_STACK_LOCATION loc = IoGetCurrentIrpStackLocation(Irp);
    if (loc->MajorOperation == IRP_MJ_CREATE
        && loc->Flags & SL_FORCE_ACCESS_CHECK) {
        return UserMode;
    }
    else {
        return Irp->RequestorMode;
    }
}
```

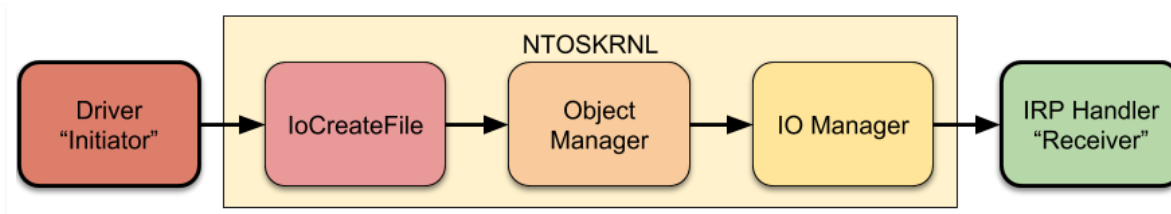
The *NtfsEffectiveMode* is called by any operation which is going to perform a security related function. It ensures that SecAC is still performed even if the caller was in kernel mode, as long as the *IFAC* flag was passed. The NTFS filesystem driver is a fundamental part of the Windows OS, and has close interactions with the IO manager, so it's unsurprising it knows to do the right thing. However, on Windows all drivers are filesystem drivers, even if they don't explicitly implement a filesystem.

I thought it'd be interesting to find out how many Microsoft and third-party drivers actually did the right checks, or did they all just trust *RequestorMode* and base their security decisions on it?

## Defining the Bug Class

Finally we get to define the bug class. In order for a privilege escalation vulnerability to exist there needs to be two separate components.

1. A kernel mode *Initiator* (code which calls *IoCreateFile* or *IoCreateFileEx*) which sets the *INPC* and *IFAC* flags but doesn't set *OFAC*. This could be in a driver or the kernel itself.
2. A vulnerable *Receiver* which uses *RequestorMode* during the handling of *IRP\_MJ\_CREATE* for a security decision but doesn't also check the *Flags* for *SFAC*.



The following table summarises an API when the calling thread's previous mode is set to *UserMode*. The table includes the state of the input options, *INPC*, *IFAC* and *OFAC* and the corresponding IRP's *RequestorMode* and *SFAC* flag. I've highlighted when the calls are a useful initiator.

<i>Method</i>	<i>INPC</i>	<i>IFAC</i>	<i>OFAC</i>	<i>RequestorMode</i>	<i>SFAC</i>	<i>Initiator?</i>
IoCreateFile	NO	NO	NO	User	NO	NO
IoCreateFile	YES	NO	NO	Kernel	NO	YES
IoCreateFile	YES	YES	NO	Kernel	YES	YES
IoCreateFile	N/A	N/A	YES	User	N/A	NO
IoCreateFileEx	NO	NO	NO	Kernel	NO	YES
IoCreateFileEx	YES	NO	NO	Kernel	NO	YES
IoCreateFileEx	YES	YES	NO	Kernel	YES	YES
IoCreateFileEx	N/A	N/A	YES	User	N/A	NO

It's worth noting that any calls where *IFAC* is not passed might be vulnerable to a privileged file access vulnerability as without the generated *SFAC* flag even NTFS would not perform security checks. There are other similar functions to *IoCreateFile* such as [FltCreateFileEx](#) which are used in special cases but they all have similar properties. Also notice in the table the rules are slightly different for *IoCreateFileEx*. While it's not documented, *IoCreateFileEx* always passes the *INPC* option to *IoCreateFile*, therefore unless the *OFAC* flag is specified it will always run its operations with the previous access mode set to *KernelMode*.

The ideal *Initiator* is one which opens an arbitrary path from the user and gives full control over all parameters to *IoCreateFile* and the opened file handle is returned back to user mode. However, depending on the *Receiver* full control might not be necessary.

A *Receiver* could perform a number of actions when receiving the IRP. Common for file system drivers is to parse the remaining filename and perform some further action based on that such as opening a different file. Another possibility is parsing the Extended Attributes (EA) block and performing some action based on that. It might just be the case that opening the device object would normally require an access check which the setting of *RequestorMode* to *KernelMode* bypasses.

## Examples

Here's some examples I've found of both *Initiators* and *Receivers*. This is based on code shipped with Windows 10 1709 which is two versions behind what's available today (1809) but many of these examples still exist in the latest versions of Windows as well as Windows 7 and 8. All the examples are Microsoft code, so a third party developer probably has even less understanding of the behavior.

To discover these examples I didn't use any special static analysis tooling, instead I just searched for them manually. I left deeper investigation to Microsoft.

### Receivers

Finding Receivers can be much harder than Initiators as there's no imported function to search for which gives a clear signal to look for. Instead I looked for drivers which imported *IoCreateDevice* to ensure the driver is exposing a device of some sort. I then filtered the drivers' imported APIs to ones which took an explicit *AccessMode* parameter, such as *SeAccessCheck* or *ObReferenceObjectByHandle*. Of course this didn't really limit the number of drivers very much so ultimately I had to manually analyze drivers which looked the most interesting. In my analysis the "real" file system drivers such as NTFS and FAT always seem to do the right thing.

### WS2IFSL

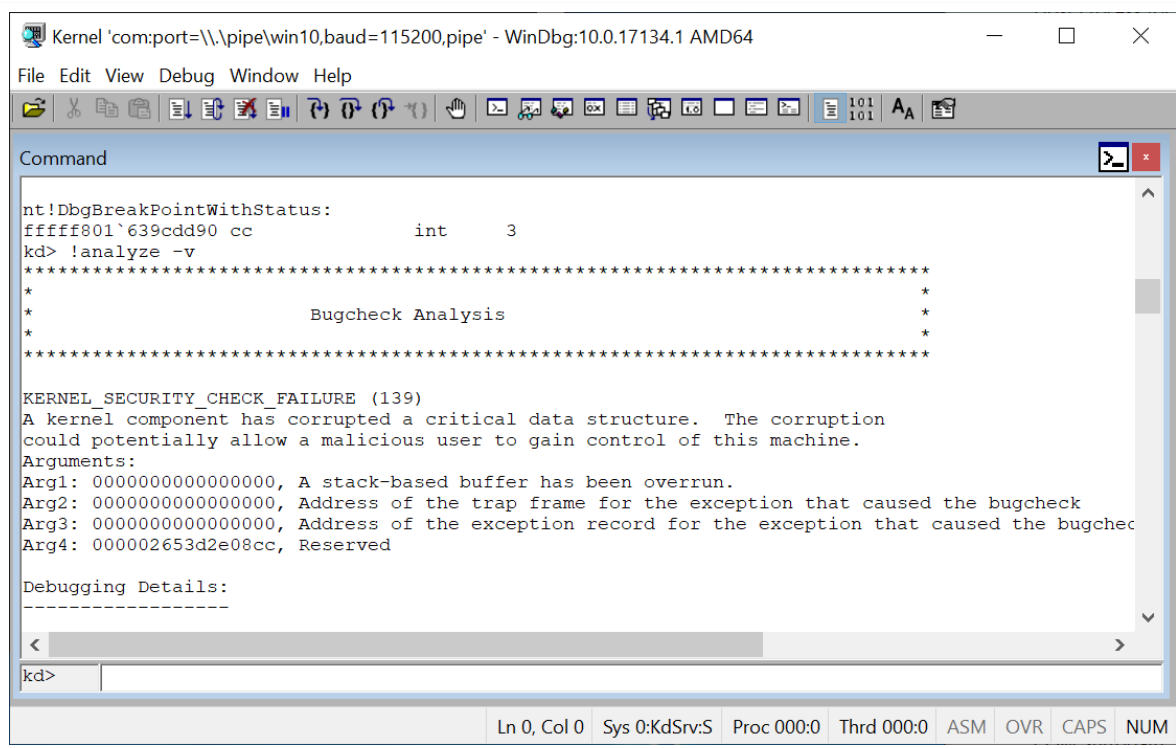
While not always enabled this driver is used to create a file object which passes read/write requests to a user mode application using APCs. When creating a new object you can specify an EA with information to create either a Socket or a Process file. The APC is set up in the *CreateProcessFile* function based on the information in the EA. The driver uses *RequestorMode* without any further checks, this would allow the callback APC to execute in kernel mode. When creating a process file you also pass a handle to a thread which is opened for *THREAD\_SET\_CONTEXT* access for use with the APC. Setting *KernelMode* allows kernel handles to be used with call the call to *ObReferenceObjectByHandle* however as the thread has to be in the calling process this isn't much of a benefit.

```

NTSTATUS DispatchCreate(DEVICE_OBJECT* DeviceObject, PIRP Irp) {
    PFILE_FULL_EA_INFORMATION ea = Irp->AssociatedIrp.SystemBuffer;
    PIO_STACK_LOCATION loc = IoGetCurrentIrpStackLocation(Irp);
    if (ea->EaNameLength != 7)
        return STATUS_INVALID_PARAMETER;
    if (!memcmp(ea->EaName, "NifsSct", 8))
        return CreateSocketFile(lock->FileObject, Irp->RequestorMode, ea);
    if (!memcmp(ea->EaName, "NifsPvd", 8))
        return CreateProcessFile(lock->FileObject, Irp->RequestorMode, ea);
    // ...
}

```

To be exploitable a compatible initiator must be able to provide an EA on the process file. Then a socket file would need to be created which referenced that process file, and a read/write operation must be performed to force the APC to execute. On modern versions of Windows 10, you'll also have to worry about SMEP and Kernel CFG. If you point the APC routine at a user mode address the kernel will bug check when it goes to execute the APC in *KernelMode* as shown in the following screenshot which I created by using a custom initiator to setup WS2IFSL.



The screenshot shows the WinDbg interface with the title bar 'Kernel 'com:port=\\.\pipe\win10,baud=115200,pipe' - WinDbg:10.0.17134.1 AMD64'. The menu bar includes File, Edit, View, Debug, Window, and Help. The toolbar contains various debugging icons. The Command window shows the following text:

```
nt!DbgBreakPointWithStatus:
fffff801`639cdd90 cc          int      3
kd> !analyze -v
*****
*                                     *
*               Bugcheck Analysis   *
*                                     *
*****

KERNEL_SECURITY_CHECK_FAILURE (139)
A kernel component has corrupted a critical data structure. The corruption
could potentially allow a malicious user to gain control of this machine.
Arguments:
Arg1: 0000000000000000, A stack-based buffer has been overrun.
Arg2: 0000000000000000, Address of the trap frame for the exception that caused the bugcheck
Arg3: 0000000000000000, Address of the exception record for the exception that caused the bugcheck
Arg4: 000002653d2e08cc, Reserved

Debugging Details:
-----
<----->
```

The status bar at the bottom shows 'Ln 0, Col 0', 'Sys 0:KdSrv:S', 'Proc 000:0', 'Thrd 000:0', and tabs for ASM, OVR, CAPS, and NUM.

## NPFS

While NPFS correctly checks for the *SFAC* it does use the *RequestorMode* to determine if the caller is allowed to specify an arbitrary EA block. Normally when a named pipe is opened the driver records the calling PID and a session ID. This information can then be exposed via APIs such as [GetNamedPipeClientProcessId](#). If the caller is *UserMode* then the code isn't allowed to set the EA block, but if it's *KernelMode* an arbitrary EA block can be used which means the PID and session ID fields can be spoofed.

```
NTSTATUS NpCreateClientEnd(PIRP Irp, ...) {
    // ...
    PFILE_FULL_EA_INFORMATION ea = Irp->AssociatedIrp.SystemBuffer;
    PVOID Data;
    SIZE_T Length;
```



```

if (!NpLocateEa(ea, "ClientComputerName", &Data, Length))
    return STATUS_INVALID_PARAMETER;

if (!IsValidEaString(Data, Length) || Irp->RequestorMode != KernelMode)
    return STATUS_INVALID_PARAMETER;

NpSetAttributeInList(Irp, CLIENT_COMPUTER_NAME, Data, Length);
NpLocateEa(ea, "ClientProcessId", Data, Length);
NpSetAttributeInList(Irp, CLIENT_PROCESS_ID, Data, Length);
NpLocateEa(ea, "ClientSessionId", Data, Length);
NpSetAttributeInList(Irp, CLIENT_SESSION_ID, Data, Length);

// ...
}

```

This behavior is used to allow the SMB driver to set the computer name field and session ID. If some service trusted this information it could be used to elevate privileges. To exploit this you'd need to be able to set an arbitrary EA as *KernelMode*, and to do something interesting you'd need to access the opened handle.

## Initiators

To find Initiators I looked in the kernel and drivers for any functions which called *IoCreateFile* et al and did basic inspection of the call parameters for the *Options* and object attributes flags. With suitable candidates I was able to do a closer inspection to determine what parameters the user could influence. Finding Initiators is relatively trivial once you understand the bug class as you can quickly narrow down targets just by looking for imported calls to the target methods.

### NTOSKRNL NtSetInformationFile FileRenameInformation Class

When renaming a file you can specify an arbitrary path even though the file can't be on a different volume to the original. The function *IoOpenLinkOrRenameTarget* is called to open the target path first using *IoCreateFileEx* passing *INPC* and normally *IFAC* (it also sets *IO\_OPEN\_TARGET\_DIRECTORY* but that's not important for the operation). This initiator only allows you to specify the full path.

### SMBv2 Server Driver

The SMB servers will open files on a share using *IoCreateFileEx*, for example in *Smb2CreateFile*. It specifies *IFAC* but not *INPC* because the call is made on a system thread so the previous access mode is already

*KernelMode*. Normally it's not possible to redirect the file creation to an arbitrary NT Object Manager path as the server passes a relative path to an open volume handle. While you could add a mount point to a directory on the file system and access the server locally the kernel intentionally limits the target device to a limited set of types.

```
if (ParseContext->ReparseTag == IO_REPARSE_TAG_MOUNT_POINT) {
    switch (ParseContext->TargetDevice) {
        case FILE_DEVICE_DISK:
        case FILE_DEVICE_CD_ROM:
        case FILE_DEVICE_DISK:
        case FILE_DEVICE_TAPE:
            break;
        default:
            return STATUS_IO_REPARSE_DATA_INVALID;
    }
}
```

There was a bug in the implementation which allows you to circumvent the device check which allows a local mount point to be used to redirect the SMB server to open any device file. The SMBv2 driver has special requirements when it comes to NTFS symbolic links, it must return the [link information to the client](#) for processing. To support the symbolic link feature the server passes the *Options* flag *IO\_STOP\_ON\_SYMLINK* as shown below.

```
NTSTATUS Smb2CreateFile(HANDLE VolumeHandle, PUNICODE_STRING Name, ...) {
    // ...
    int ReparseCount = 0;
    OBJECT_ATTRIBUTES ObjectAttributes;
    ObjectAttributes.RootDirectory = VolumeHandle;
    ObjectAttributes.ObjectName = Name;
    IO_STATUS_BLOCK IoStatus = {};
    do {
        status = IoCreateFileEx(
            &FileHandle,
            DesiredAccess,
            &ObjectAttributes,
            &IoStatus,
```

```

        ...
        IO_STOP_ON_SYMLINK | IO_FORCE_ACCESS_CHECK
    );
    if (status == STATUS_STOPPED_ON_SYMLINK) {
        UNICODE_STRING NewName;
        status = SrvGraftName(ObjectAttributes.ObjectName,
            (PREPARSE_DATA_BUFFER)IoStatus.Information, &NewName);
        if (status == STATUS_STOPPED_ON_SYMLINK)
            break;
        ObjectAttributes.RootDirectory = NULL;
        ObjectAttributes.ObjectName = NewName;
        continue;
    }
} while (ReparseCount++ < MAXIMUM_REPARSE_COUNT);
// ...
}

```

If *IoCreateFileEx* returns *STATUS\_STOPPED\_ON\_SYMLINK* the server extracts the returned *REPARSE\_DATA\_BUFFER* structure from *IO\_STATUS\_BLOCK* and passes it to the *SrvGraftName* utility function. The reparse buffer could be either a mount point or a NTFS symbolic link. If it's a symbolic link then *SrvGraftName* returns *STATUS\_STOPPED\_ON\_SYMLINK* again which allows the server to return the buffer to the caller. If the reparse buffer is a mount point then *SrvGraftName* builds a new absolute path based only on the string found in the *REPARSE\_DATA\_BUFFER*, it doesn't check the destination device. The server reissues the open request with the new absolute path which can now point to any device on the system.

This initiator was the best I found during my analysis. You can specify almost all arguments to *IoCreateFileEx* including the EA buffer. This allows you to initialize a driver which requires an EA (such as WS2IFSL) which opens up a lot more attack surface. As it's running on a system thread both the *RequestorMode* and the thread's previous mode are set to *KernelMode* which might introduce other interesting attack surface.

While impressive it's not an ideal Initiator, the opened device needs to support certain valid IRP's such as *IRP\_MJ\_GET\_INFORMATION\_FILE* otherwise the server will not return a valid handle to the caller. Without this check it would be trivial to exploit WS2IFSL as you'd be able to perform a Read/Write operation to get an APC to execute in kernel mode. Even if you can get a handle back to the file the SMB Server intentionally limits the IO control codes you can send which limits many of the interesting things you could do with this

vulnerability. Even so I considered this to be a serious issue, so I reported it directly to MSRC. It was fixed as [CVE-2018-0749](#) by using a special [Extra Creation Parameter](#) which will filter out all other reparse points except for symbolic links.

## Next Steps

While I believed this to be a serious bug class, it turned out that finding a matching Initiator/Receiver pair was very difficult. In my research I didn't find any pair which would give direct privilege escalation. The best pair I identified was combining the SMBv2 Initiator with the NPFS process ID spoofing. While I couldn't identify a service which would use the client PID for any security related operation it's possible that a third-party service exists.

I could have filed this issue back in my list of interesting or unexpected behaviors, to see if I could exploit it at a later date. But instead I decided to talk to my contacts at MSRC to see if we could collaborate instead. With MSRC on side I wrote up a document explaining the bug class and describing some of my findings. At the same time I reported the SMB server issue through the normal channels at it was the most serious issue I'd discovered. This led to meetings with various teams at Bluehat 2017 in Redmond where a plan was formed for Microsoft to use their source code access to discover the extent of this bug class in the Windows kernel and driver code base. Note, I did not have access to the source code, this part of the investigation was delegated to MSRC, the results of which are in their blog post.

It's worth noting that while I applied the standard 90 day disclosure deadline to the SMB server report, I didn't apply an explicit deadline to the bug class report. With no single bug to point to it'd be difficult and likely counter productive to enforce such a deadline. However I did ensure that MSRC agreed to publish the technical details on this issue regardless of the outcome. This is why we're blogging about it now, 12 months after providing the report.

## Conclusions

It's always interesting to find a new bug class in Windows to go hunting for. By luck this hasn't turned out to be anywhere as serious as it could have been. While it would have made sense to specify two separate access modes, one for MemAC and one for SecAC, that's not what the original NT design used. For backwards compatibility it seems unlikely that behavior will be changed. This bug class is as much about poor documentation as it is about technical issues, as while the behavior can't be changed, it's also poorly documented.

If you look at the documentation for *IoCreateFile* you'll now find a new remark:

*"For create requests originating in user mode, if the driver sets both IO\_NO\_PARAMETER\_CHECKING and IO\_FORCE\_ACCESS\_CHECK in the Options parameter of IoCreateFile then it should also set OBJ\_FORCE\_ACCESS\_CHECK in the ObjectAttributes parameter. For info on this flag, see the Attributes member of OBJECT\_ATTRIBUTES."*

This remark was added very recently. With the majority of Microsoft's developer documentation on GitHub you can even find [this commit](#) which introduced it.

It's likely that any bugs identified by MSRC will only be fixed in the latest versions of Windows 10, therefore if you're a developer you should read Microsoft's blog post to understand how to avoid these issues in your drivers as well as techniques to find these sorts of issues in your code base. For security researchers it's another thing to bear in mind when you're reviewing a new Windows kernel driver.

I'd like to thank Steven Hunter and Gavin Thomas from MSRC who were my main points of contact for getting this bug class remediated.

Posted by [Ben](#) at [9:00 AM](#)

No comments:



[Newer Posts](#) · · · · · [Home](#) · · · · · [Older Posts](#)

Subscribe to: [Posts \(Atom\)](#)

---

Simple theme. Powered by [Blogger](#).