

# The Windows Notification Facility

The Most Undocumented Kernel Attack Surface Yet

Gabrielle Viala  
Alex Ionescu

@pwissenlit  
@aionescu

Black Hat USA  
2018

# About Gabrielle Viala

- 🔗 Gaby - @pwissenlit on twitter
- 🔗 Reverse engineer at Quarkslab
- 🔗 Playing with the Windows Internals
- 🔗 Member of the BlackHoodie organization board
- 🔗 Quite new in the field so I don't have much record yet ;)



# About Alex Ionescu

- 🔗 VP of EDR Strategy and Founding Architect at CrowdStrike
- 🔗 Co-author of *Windows Internals 5<sup>th</sup>-7<sup>th</sup> Editions*
- 🔗 Reverse engineering NT since 2000 – was lead kernel developer of ReactOS
- 🔗 Instructor of worldwide Windows internals classes
- 🔗 Author of various tools, utilities and articles
- 🔗 Conference speaker at SyScan, Infiltrate, Offensive Con, Black Hat, Blue Hat, Recon, ...
- 🔗 For more info, see [www.alex-ionescu.com](http://www.alex-ionescu.com) or hit me up on Twitter @ aionescu



# Talk Outline

- WNF Internals
  - APIs
  - Structures
  - Tools & Extensions
- WNF Attack Surface
  - Subscribers and Callbacks
  - Some “Fuzzing” Results
- Interesting/Sensitive WNF State Names
  - System State
  - User Behavior
- Using WNF to Change System State
  - Process Migration
- Key Takeaways & Future Research

# WNF Internals & APIs

# What is WNF?

- 🔗 The Windows Notification Facility is a *pubsub* (Publisher/Subscriber) user/kernel notification mechanism that was added in **Windows 8**, in part to solve some long-standing design constraints in the OS, as well as to serve as a basis for Mobile-based/App-centric Push Notifications similar to iOS/Android
- 🔗 Its key differentiator is that it is a *blind* (basically, registration-less) model which allows for *out-of-order* subscription vs. publishing
  - ✂ By this, we mean that a consumer can subscribe to a notification even before the notification has been published by its producer
  - ✂ And that there is no requirement for the producer to ‘register’ the notification ahead of time
- 🔗 On top of this, it also supports persistent vs. volatile notifications, monotonically increasing unique change stamp IDs, payloads of up to 4KB for each notification event, a thread-pool-based notification model with group-based serialization, and a security model that is both *scope* based and implements Windows Security Descriptors through the standard DACL/SACL mechanism [Run-On Sentence Pwnie Award]

# Why does WNF exist?

🔗 The canonical example is a driver wanting to know if the volumes have been mounted for RW access yet

✂ To indicate this, Autochk (Windows' fsck) signals an event called `VolumesSafeForWriteAccess`

- But to signal an event, you need to first create it

🔗 However, how can we know if Autochk has run, before Autochk has created the event for us wait on?

✂ Ugly solution: Sit in a `sleep()` loop checking for the presence of the event – once the event is known to exist, wait on it

🔗 After a Windows application exits, all handles are closed – and once an object has no handles, it is destroyed

✂ So who would event keep the event around?

🔗 Without WNF, the solution is to have the kernel create the event before any drivers can load, and have Autochk *open* it like a consumer would, but instead, be the one to signal it instead of waiting on it

# WNF State Names

🔗 In the WNF world, a state name is a 64-bit number – but there's a trick to it – it has an encoded structure

🔗 A state name has a *version*, a *lifetime*, a *scope*, a *data permanence* flag, and a *unique sequence* number

```
typedef struct _WNF_STATE_NAME_INTERNAL
{
    ULONG64 Version:4;
    ULONG64 NameLifetime:2;
    ULONG64 DataScope:4;
    ULONG64 PermanentData:1;
    ULONG64 Unique:53;
} WNF_STATE_NAME_INTERNAL, *PWNF_STATE_NAME_INTERNAL;
```

🔗 This data is only accessible if we XOR the 64-bit number with a magic constant:

```
#define WNF_STATE_KEY 0x41C64E6DA3BC0074
```



# State Name Lifetime

- ❧ A WNF state name can be *well-known*, *permanent*, *persistent*, or *temporary* (WNF\_STATE\_NAME\_LIFETIME)
- ❧ The first-three lifetimes are related to the registry location where the state information will be kept
  - ✧ Well-known names live in HKLM\SYSTEM\CurrentControlSet\Control\Notifications
  - ✧ Permanent names live in HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Notifications
  - ✧ Persistent names live in HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\VolatileNotifications
- ❧ Cannot *register* a well-known name – these are relied upon by the kernel and must be provisioned in the registry
- ❧ Permanent and persistent names require SeCreatePermanentPrivilege, just like global named objects
  - ✧ Persistent names persist beyond process (registrar's) exit while *permanent* names persist beyond reboot

# State Scopes

- ❧ The data scope determines the first security boundary around a WNF state name – who has access/visibility to it
  - ✖ A state name's can have *system scope*, *session scope*, *user scope*, *process scope* or *machine scope*
  - ✖ Other than providing a security boundary, WNF scopes can also be used to provide *instantiated data* for the same name
  - ✖ Kernel bypasses state access checks, while TCB-privilege allows cross-scope access of WNF state names
- ❧ System and machine scoped names are global – there is no scope identifier (they use a different *scope map*)
- ❧ Session scoped names use the session ID as the identifier
- ❧ User scoped names use the user SID as the identifier
- ❧ Process scoped names use the EPROCESS object address as the identifier

# Sequence Numbers

🔗 To guarantee uniqueness, each state name has a unique 51-bit sequence number associated with it

✖ Well-known names have a 4-character *family* tag with the remainder 21 bits used as the unique identifier

✖ Permanent names have an increasing sequence number seeded off the registry value “SequenceNumber”

✖ Persistent names and volatile names share an increasing sequence number seeded off a runtime global value

🔗 This data is then managed on a per-silo (container) basis and available in `PspHostSiloGlobals->WnfSiloState`

🔗 Internally, within Microsoft, each WNF name then has a ‘friendly’ identifier that is used in code – sometimes this is stored in a global sharing the same name

✖ `nt!WNF_BOOT_DIRTY_SHUTDOWN - 0x1589012fa3bc0875 => 0x544f4f4200000801`

✖ `BOOT1`, Well-Known Lifetime, System Scope, Version 1

# Registering a WNF State Name


Other than well-known names, as previously mentioned, a WNF state name can be registered at runtime:

NTSTATUS

```
ZwCreateWnfStateName (  
    _Out_ PWNF_STATE_NAME StateName,  
    _In_ WNF_STATE_NAME_LIFETIME NameLifetime,  
    _In_ WNF_DATA_SCOPE DataScope,  
    _In_ BOOLEAN PersistData,  
    _In_opt_ PCWNF_TYPE_ID TypeId, // This is an optional way to get type-safety  
    _In_ ULONG MaximumStateSize, // Cannot be above 4KB  
    _In_ PSECURITY_DESCRIPTOR SecurityDescriptor // *MUST* be present  
);
```

Can also use ZwDeleteWnfStateName to delete the registered state name (other than for well-known ones)

# Publishing WNF State Data

 To modify WNF state name data, the following system call can be used

NTSTATUS

```
ZwUpdateWnfStateData (  
    _In_ PCWNF_STATE_NAME StateName,  
    _In_reads_bytes_opt_(Length) const VOID* Buffer,  
    _In_opt_ ULONG Length, // Must be less than MaximumSize when registered  
    _In_opt_ PCWNF_TYPE_ID TypeId, // Optionally, for type-safety  
    _In_opt_ const PVOID ExplicitScope, // Process handle, User SID, Session ID  
    _In_ WNF_CHANGE_STAMP MatchingChangeStamp, // Expected current change stamp  
    _In_ LOGICAL CheckStamp // Enforce the above or silently ignore it  
);
```

 Can also use ZwDeleteWnfStateData to wipe the current state data buffer

# Consuming WNF Data

🔗 To query WNF state name data, the following system call can be used instead – most parameters are like *update*

NTSTATUS

```
ZwQueryWnfStateData(  
    _In_ PCWNF_STATE_NAME StateName,  
    _In_opt_ PCWNF_TYPE_ID TypeId,  
    _In_opt_ const VOID* ExplicitScope,  
    _Out_ PWNF_CHANGE_STAMP ChangeStamp,  
    _Out_writes_bytes_to_opt_(*BufferSize, *BufferSize) PVOID Buffer,  
    _Inout_ PULONG BufferSize // Can be 0 to receive the current size  
);
```

🔗 The real power, however, is that both the Update and Query APIs don't actually need a registered state name

✂ If the name is non-temporary, and the caller has sufficient access, the name instance can be live-registered!

# WNF Notifications

🔗 So far, we've assumed that the consumer knows *when* to call the API – but there are 'blocking' reads as well

✂ This works using a notification system, closer to a true pub-sub model

🔗 First, the process must register an event (`ZwSetWnfProcessNotificationEvent`)

✂ Then use `ZwSubscribeWnfStateChange`, specifying an event mask -> receive a subscription ID on output

- 1 -> Data Arrival 10 -> Name Destroyed (these are called *data notifications*)
- 2 -> Data Subscriber Arrived, 4 -> Meta Subscriber Arrived, 8 -> Generic Subscriber Arrived (these are called *meta notifications*)

✂ Then, wait on the event that was registered

✂ Whenever it is signaled, `ZwGetCompleteWnfStateSubscription`, which returns an `WNF_DELIVERY_DESCRIPTOR`

🔗 But these low-level APIs have a problem (thanks Gabi!) – only a single per-process notification event can exist

# High Level API

🔗 When it comes to notifications, things get complicated – so Rtl provides a simpler interface:

## NTSTATUS

```
RtlSubscribeWnfStateChangeNotification (  
    _Outptr_ PWNF_USER_SUBSCRIPTION* Subscription,  
    _In_ WNF_STATE_NAME StateName,  
    _In_ WNF_CHANGE_STAMP ChangeStamp,  
    _In_ PWNF_USER_CALLBACK Callback,  
    _In_opt_ PVOID CallbackContext,  
    _In_opt_ PCWNF_TYPE_ID TypeId,  
    _In_opt_ ULONG SerializationGroup,  
    _In_opt_ ULONG Unknown);
```

🔗 Uses single Ntdll.dll-managed event queue with a callback system – no need for using any system calls



# Notification Callback

- Behind the scenes, the contents of the `WNF_DELIVERY_DESCRIPTOR` are converted into the callback parameters

```
typedef NTSTATUS (*PWNF_USER_CALLBACK) (  
    _In_ WNF_STATE_NAME StateName,  
    _In_ WNF_CHANGE_STAMP ChangeStamp,  
    _In_opt_ PWNF_TYPE_ID TypeId,  
    _In_opt_ PVOID CallbackContext,  
    _In_ PVOID Buffer,  
    _In_ ULONG BufferSize);
```

- For each registration, an entry is entered into the `RtlpWnfProcessSubscriptions` global pointer, which has a `LIST_ENTRY` of `WNF_NAME_SUBSCRIPTION` structures (we will play with these a bit later)

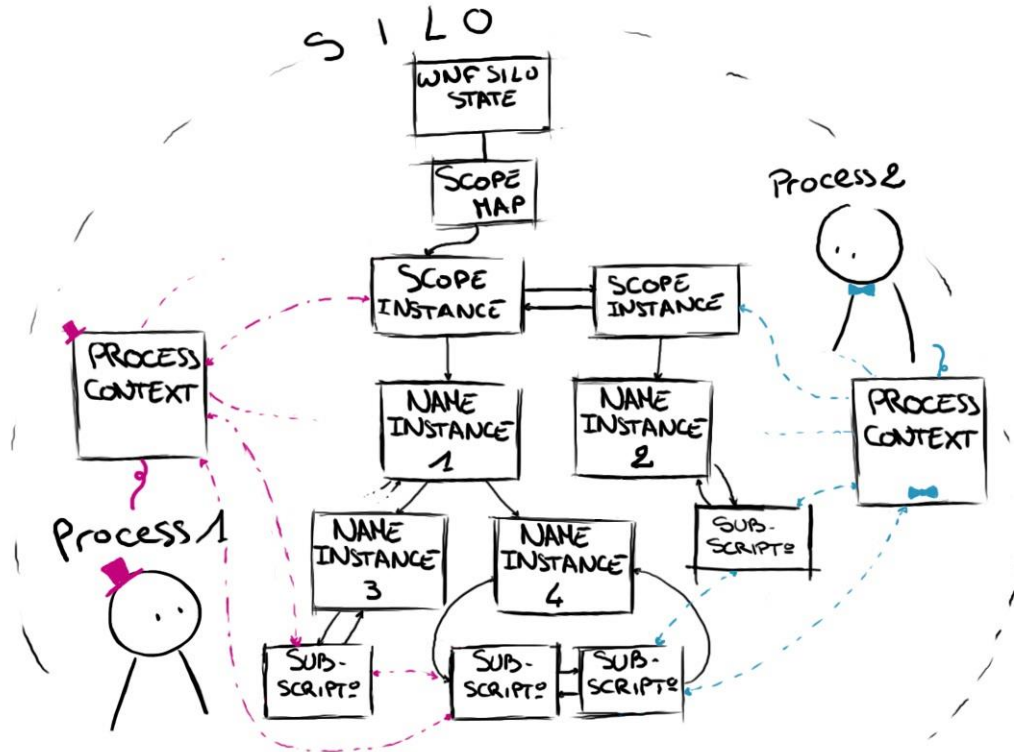
Each one of these, in turn, has a `LIST_ENTRY` of `WNF_USER_SUBSCRIPTION` structures, with the callback and context

# Kernel API

- 🔗 WNF also provides almost identical functionality to kernel-mode callers as well, both through the exported system calls (which can be used from a driver) as well as through high-level APIs in the executive runtime
- 🔗 `ExSubscribeWnfStateChange` -> Given state name, mask, and a callback + context, receives subscription handle
  - ✂ Callbacks receive the signaled name, event mask, change stamp, but not the buffer or its size
- 🔗 `ExQueryWnfStateData` receives the subscription handle, and reads the currently active state data
  - ✂ Each callback ends up calling this to actually get the data associated with the notification
- 🔗 For both kernel and user-mode subscriptions, WNF creates a `WNF_SUBSCRIPTION` data structure to track it
  - ✂ But some fields won't be filled out for user-mode – like the `Callback/Context` – since this is in `Ntdll.dll` struct instead

# WNF Data Structures

# WNF Structure



Analyzing with Tools & WinDbg

# WinDBG Custom Extension

Basically does the same things as !wnf command

Does not rely on private symbols and should (hopefully) work smoothly


```
kd> !wnfhelp
[WnfDbg] extension commands help:
> !wnfsm [Address] = Displays the structure of a _WNF_SCOPE_MAP object.
                    The address can either be a scope map or an eprocess (in case of multiple silo).
                    If no address are provided, it will search for the generic scope map with nt!PspHostSiloGlobals.


> !wnfsi [Address] [0/1/2] = Displays the structure of a _WNF_SCOPE_INSTANCE object.
> !wnfsilist <Address> [0/1/2] = List all the scope instances in a list_entry.
> !wnfni [Address] [0/1/2] = Displays the structure of a _WNF_NAME_INSTANCE object.
> !wnfnilist <Address> [0/1/2] = List all the name instances in a list_entry.
> !wnfsd <Address> = Displays the structure of a _WNF_STATE_DATA object and dump the data.
> !wnfsub [Address] [0/1/2] = Displays the structure of a _WNF_SUBSCRIPTION object.
> !wnfsublist <address> [offsetList] [0/1/2] = List all the subscriptions in a list_entry
                                             [offsetList] indicates the offset of the list to parse in the subscription.
                                             By default, it will parse ProcessSubscriptionListEntry.
                                             If the address provided is the base address of a list_entry, it will parse this list.

> !wnfctx [Address] [0/1/2] = Displays the structure of a _WNF_PROCESS_CONTEXT object.
> !wnfctxlist [Address] [0/1/2] = Displays the structure of a _WNF_SCOPE_INSTANCE object.
                                If no address is provided, it will list the process contexts pointed by nt!ExpWnfProcessesListHead
> !wnfctxhead = Prettyprints the list head pointed by nt!ExpWnfProcessesListHead
> !wnfname <StateName> = Displays the WNF state name information.
> !wnfwhat <Address> = Determines the type of the structure at the provided address and dumps its structure.
> !wnfhelp = meh... >_>'
```


- For most of these commands, the inputted address can either be the base address of the object or the base of a list\_entry in the structure.
- Specify the verbosity level with 0, 1 or 2.


# WnfCom

 (Small) python module that can be used to communicate via WNF

 Enables reading and writing data for existing name instances

 Can also create a temporary state name (server style ☺)


 Got a “client” side that is notified anytime an update happens for a specific name instance

 Easy peasy to use:

```
>>> from wnfcom import Wnfcom
>>> wnfserver = Wnfcom()
>>> wnfserver.CreateServer()
[SERVER] StateName created: 41c64e6da5559945
>>> wnfserver.Write(b“potato soup”)
Encoded Name: 41c64e6da5559945, Clear Name: 6e99931
          Version: 1, Permanent: No, Scope: Machine,
Lifetime: Temporary, Unique: 56627
State update: 11 bytes written
```

```
>>> from wnfcom import Wnfcom
>>> wnfclient = Wnfcom()
>>> wnfclient.SetStateName("41c64e6da5559945")
>>> wnfclient.Listen()
[CLIENT] Event registered: 440
[CLIENT] Timestamp: 0x1 Size: 0xb
      Data:
00000000: 70 6F 74 61 74 6F 20 73  6F 75 70
potato soup
```


# Wnfdump

 Command-line C utility that can be used to discover information about WNF state names


 -d → Dump all WNF state names using registry-based enumeration, displaying vital information


- Add -v → Verbose output, which includes a hexdump of the WNF state data
- Add -s → Security descriptors, which includes an SDDL string of the WNF state name's permissions

 -b → Bruteforce temporary WNF state names (will show how we do this soon)

 -i → Show information about a specific single WNF state name

 -r → Read a particular WNF state name

 -w → Write data into a particular WNF state name

 -n → Register for notifications as a subscriber for the given WNF state name



# StateNameDiffer



Yet another small python script to dump or diff Well-Known State Names in DLLs

Allows quickly spotting differences across versions

Dump output can be used in other scripts (Alex's wnfTool, wnfCom, etc.)

```
$ python .\StateNamediffer.py -h
usage: StateNamediffer.py [-h] [-dump | -diff] [-v] [-o OUTPUT] [-py] file1 [file2]
Stupid little script to dump or diff wnf name table from dll
positional arguments:
  file1
  file2 Only used when diffing
optional arguments:
  -h, --help show this help message and exit
  -dump Dump the table into a file
  -diff Diff two tables and dump the difference into a file
  -v, --verbose Print the description of the keys
  -o OUTPUT, --output OUTPUT Outputs the output file (Default: output.txt)
  -py, --python Change the output language to python (by default it's c)
```

WNF Attack Surface

# The Privileged Disclosure

🔗 When reading the thousands of WNF state names that exist on the system, I noticed several that had interesting-looking buffers

✖ Including some that had what looked to be like pointers or other privileged data

🔗 I did several repros on multiple machines and in some cases discovered heap, stack, and other privileged information data/information disclosures across privilege boundaries

✖ Submitted case(s) with MSRC in July – fixed in November! (sorry I am not James Forshaw it takes 90+ days)

✖ WNF\_AUDC\* events leak(ed) 4KB of stack space!

🔗 The main underlying problem is the same as you've seen in past research from j00ro, tavis0, and others...

✖ Certain WNF state names contain encoded data structures with various padding/alignment issues

✖ Or in certain cases, bona fide uninitialized memory (such as due to `sizeof` misuse or other scenarios)

# Discovering State Names and Permissions

- 🔗 The first approach was to discover all the possible state names that could perhaps be manipulated maliciously
- 🔗 For well-known names, permanent names, and persisted names, this is doable by enumerating the registry keys
  - ✿ We can then associate friendly names with the well-known names if we obtain Microsoft's database
    - There are a few places where this can be found 😊
  - ✿ Then, we can also look in the registry for their security descriptor (this is the first thing in the data buffer)
- 🔗 The security descriptor is a bit tricky because it doesn't have an Owner or Group, so technically it's 'invalid'
  - ✿ We manually fix this up to address the problem with a fake owner/group
- 🔗 But for temporary names – they're not in the registry, and only kernel has the data structures for them (!wnf)

# Discovering Volatile Names

🔗 Volatile names are actually not that hard to brute force

✖ The version is always 1

✖ The lifetime is `WnfTemporaryStateName`

✖ The permanent flag is always zero (temporary names can't have permanent data)

✖ The scope is one of the 4 possible scopes

✖ But the sequence number is 51 bits, Alex!

🔗 Well.... recall that sequence numbers are *monotonically increasing*

✖ And for volatile names, the sequence is reset to 0 each boot

🔗 So we can use `ZwQueryWnfStateNameInformation` with the `WnfInfoStateNameExist` class up to ~1M...

# Brute Forcing Security Descriptors

🔒 Volatile state names only have security descriptors in kernel memory – no way to query this without !wnf

🔒 But we can infer if we have read access by trying to read 😊

🔒 We can infer write access by trying to write!

✖ But you saw how well that went

🔒 So there's a trick: remember that we can enforce a *matching change stamp*

✖ Set this to 0xFFFFFFFF – the match check is made after the access check, so the error value leaks the writeable state

🔒 Doesn't give us the whole security descriptor, but we can run the tool at different privileges to get some idea of AC/Low IL/User/Admin/SYSTEM

# Dumping Kernel/User Subscribers

- 🔗 All subscriptions done by a process are in the `WNF_PROCESS_CONTEXT` as a `LIST_ENTRY` of `WNF_SUBSCRIPTION`
  - ✖ Kernel subscriptions are basically owned by the System process
- 🔗 We can use `!list` to dump the `Callback` and `CallbackContext` in `WNF_SUBSCRIPTION` for the System Process
  - ✖ Note that in some cases, the *event aggregator* (CEA.SYS) is used which hides the real callback(s) in the context
- 🔗 We can repeat this for user-mode processes as well, but the `Callback` will be `NULL`, as these are user subscribers
  - ✖ So we need to attach to user-space, get the `RtlpWnfProcessSubscriptions` table, go over the `WNF_NAME_SUBSCRIPTION`'s and then for each one, dump the list of `WNF_USER_SUBSCRIPTION` structures, each of which has a callback
  - ✖ Unfortunately, this symbol is a *static* which means that it is not in public symbols (but can be found by disassembling)
  - ✖ Certain user-mode callbacks also use the *event aggregator* (`EventAggregation.dll`) which stores callback in context

Interesting/Sensitive WNF State Names



# Inferring System State and User Behavior with WNF

🔗 Some WNF IDs can be used because they reveal interesting information about the machine state:

- ✖ WNF\_WIFI\_CONNECTION\_STATUS – Wireless connection status (many more interesting WNF\* IDs)
- ✖ WNF\_BLTH\_BLUETOOTH\_STATUS – Similar, but for Bluetooth (also WNF\_TETH\_TETHERING\_STATE)
- ✖ WNF\_UBPM\_POWER\_SOURCE – Indicates battery power vs. AC adapter (WNF\_SEB\_BATTERY\_LEVEL has battery level)
- ✖ WNF\_CELL\_\* – On Windows Phone (lol), has information such as network, number, signal strength, EDGE vs 3G, etc.

🔗 Others can be used to infer user behaviors

- ✖ WNF\_AUDC\_CAPTURE/RENDER – Indicates (including the PID) the process that is capturing/playing back audio
- ✖ WNF\_TKBN\_TOUCH\_EVENT – Indicates each mouse click, keyboard press, or touch screen press
- ✖ WNF\_SEB\_USER\_PRESENT / WNF\_SEB\_USER\_PRESENCE\_CHANGED – Utilizes Windows' own user-presence learning

# Avoiding Standard Notification APIs

- ❧ Even in situations where certain user actions already have documented APIs for receiving a notification, these APIs may generate event log/EDR/audit data
- ❧ Corresponding WNF IDs may exist (and sometimes may even be more descriptive) for the same actions
- ❧ For example, `WNF_SHEL_(DESKTOP)_APPLICATION_(STARTED/TERMINATED)` provides information on both modern application launches (including the actual package name that was launched) through DCOM, as well as regular Win32 application launches
  - ❧ As long as the applications were created through `ShellExecute` (and/or interactively through `Explorer.exe` or CLI)
- ❧ Other examples are when there is a user-mode API, but no kernel-mode equivalent
  - ❧ `WNF_EDGE_LAST_NAVIGATED_HOST` – Indicates each URL the user types in (or clicks on) in Edge
  - ❧ `WNF_SHEL_LOCKSCREEN_ACTIVE` – Indicates the lock screen is active






# Controlling the System with WNF


- Some WNF IDs can be written to in order to effect change on the system
  - WNF\_FSRL\_OPLOCK\_BREAK receives a list of PIDs (and the number/size) and terminates each of them!
  - There is a similar API for waiting on the process(es) to exit/resume instead
- Others we haven't quite figured out yet, but look pretty interesting
  - WNF\_SHEL\_DDC\_(WNS/SMS)\_COMMAND – The buffer size is 4KB which indicates potential for parsing bugs
- In a similar vein, there are also certain WNF IDs that indicate that certain things should happen/be done
  - WNF\_CERT\_FLUSH\_CACHE\_TRIGGER – Flushes the certificate store
  - WNF\_BOOT\_MEMORY\_PARTITIONS\_RESTORE – Stores the original memory partitions
  - WNF\_RTDS\_RPC\_INTERFACE\_TRIGGER\_CHANGED – Potentially starts RPC-Interface Trigger Started Services



Using WNF to Change System State

# Injecting Code with WNF

 Common techniques for injecting code into other processes include

-  `WriteProcessMemory` – directly injecting code
-  File Mappings (Section Objects) – mapping a section object into the target, or writing into an already-mapped section
-  Atom Objects – storing data into an atom and then having the target request the atom data
-  Window Messages – using messages such as `WM_COPYDATA` and DDE messages to cause the target process to get data
-  GUI Objects – Changing the title of a Window (or that of its class) to data we wish to have in the target's memory

 Using WNF provides yet another way of transferring data into a target process

-  Either by re-using one of its existing WNF IDs that it reads (especially if it's stored in a persistent fashion in the process)
-  Or by directing the process to call `Rtl/ZwQueryWnfStateData` on a particular WNF ID of interest

# Modifying Callbacks/Contexts for Code Redirection

🔗 Injecting memory is one part of the problem, but to redirect control flow, common solutions include

✖ APCs

✖ Remote Threads

✖ Changing Thread Context

✖ Modifying the “*window long*” of a window to get a custom callback/context associated with the window handler

🔗 Another approach, however, can be to parse the `WNF_USER_SUBSCRIPTION`'s of a remote process (and these are linked to `WNF_NAME_SUBSCRIPTION`'s, which are linked off the `RtlpWnfProcessSubscriptions`)

✖ The callback function can be changed (be wary of CFG) and then the WNF payload + size will be parameters 5 and 6

✖ Alternatively, the callback context can be changed (which is often a V-Table or has another embedded function pointer)

# Conclusion

# Key Takeaways

- 🔗 The Windows Notification Facility is an extremely interesting, well-designed addition to the Windows 8+ kernel
  - ✖ It provides lots of useful functionality to various system applications, services, and drivers
  - ✖ It also acts as the basis for more advanced functionality and notification frameworks
- 🔗 Unfortunately, it is highly undocumented, and provides no real visibility into its behavior
  - ✖ Other than a WinDBG extension which doesn't work since the WNF symbols are private
  - ✖ And some limited ETW events that only work for one particular set of callers
  - ✖ Its ability to persist data across reboots also makes it even more interesting for misuse
- 🔗 There is no magic intellectual property in WNF internals – Microsoft is supposed to publish the symbols to make !wnf work



# Key Takeaways (cont)



WNF has grown beyond just providing notifications



Did its designer really mean for it to be used as a way to kill processes from kernel-mode?



Should it really be used to store PII and every URL ever visited in a world-readable WNF state name?



Do teams that use WNF at Microsoft fully understand the implications, boundaries, etc?

- Such as needing to initialize memory and treating data the same way you'd treat system call parameters



Because it can be used to transfer data from one boundary to another, and because callbacks are involved, both parsing errors as well as code redirection/injection attacks are possible using more novel techniques than usual



Obviously, this implies permissions to exist in the first place, so this is an EDR-evasion technique more than anything



How long until we start seeing WNF state names with pointers in them? :-/



Defenders – start fuzzing, building visibility tools, and poking at WNF – at the very least, you'll have fun!

# Future Research

- 🔗 A big chunk of WNF events all start with SEB\_ which represents the System Events Broker
  - ✂ SystemEventsBrokerServer.dll and SystemEventsBrokerClient.dll are the user-mode high-level APIs
  - ✂ It may be that some of these SEB events are then internally parsed by SEB's clients, masking some true consumers
- 🔗 Many of the registered kernel-mode and user-mode callbacks are owned by CEA.SYS or EventAggregation.dll
  - ✂ These are part of the “Event Aggregation Library”, which allows you to have start/stop callbacks when a certain set of events have accumulated above a certain threshold, or multiple WNF events are happening at the same time, or in a given sequence, or when at least one out of a group of WNF events have occurred
    - Essentially a finite state machine around WNF event IDs that callers can register for
    - So the real consumers are hidden behind the event aggregation library
- 🔗 We are releasing the tools this weekend on GitHub

THANK YOU!

Q & A