Playing with the Windows Notification Facility (WNF)

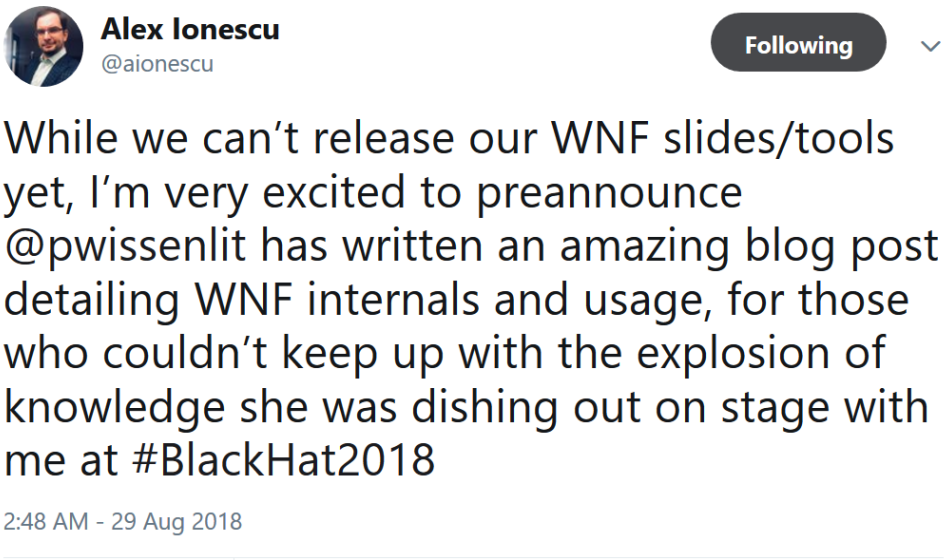
blog.quarkslab.com/playing-with-the-windows-notification-facility-wnf.html

October 25, 2018

This blogpost briefly presents the Windows Notification Facility and provides a write-up for a nice exercise that was given by Bruce Dang during his workshop at Recon Montreal 2018.

Preamble

This post was mainly written few days after the end of Bruce Dang's Recon Montreal training but I decided to postpone its publication for various reasons. I almost forgot about it until Alex reminded me that it was still pending. As Bruce already wrote a really nice post about WNF (if you missed it, I encourage you to read it now) I felt that mine wouldn't bring anything new to the table and that there wasn't much point for me to finish it properly but someone didn't really give me the choice... :')



Introduction

Some time ago, Bruce Dang invited five BlackHoodie ladies to attend his Windows Kernel Rootkit training at Recon Montreal. Barbie, Priya, Oryan, Aneal and I had the chance to be there during these four days of intensive work.

In this blog post I won't describe the content of the class (trust me, it was great) but I will focus on one of the exercises I really enjoyed: reversing and (mis)using WNF!

Input data

I didn't know this component at all and very few information on it is available on the Internet. The only input at my disposal was the following prompt:

```
14. Reverse engineer the following Windows kernel functions.
    The result of this exercise will be used in the next exercise.

    • ExSubscribeWnfStateChange
    • ExQueryWnfStateData

15. Write a driver that gets notified when an application is using the microphone.
    Print its pid.

Hints:
    • check contentDeliveryManager_Utilities.dll for the WNF StateName.
    • some interesting info is available here:

http://redplait.blogspot.com/2017/08/wnf-ids-from-perfntcdll.html
```

Some background

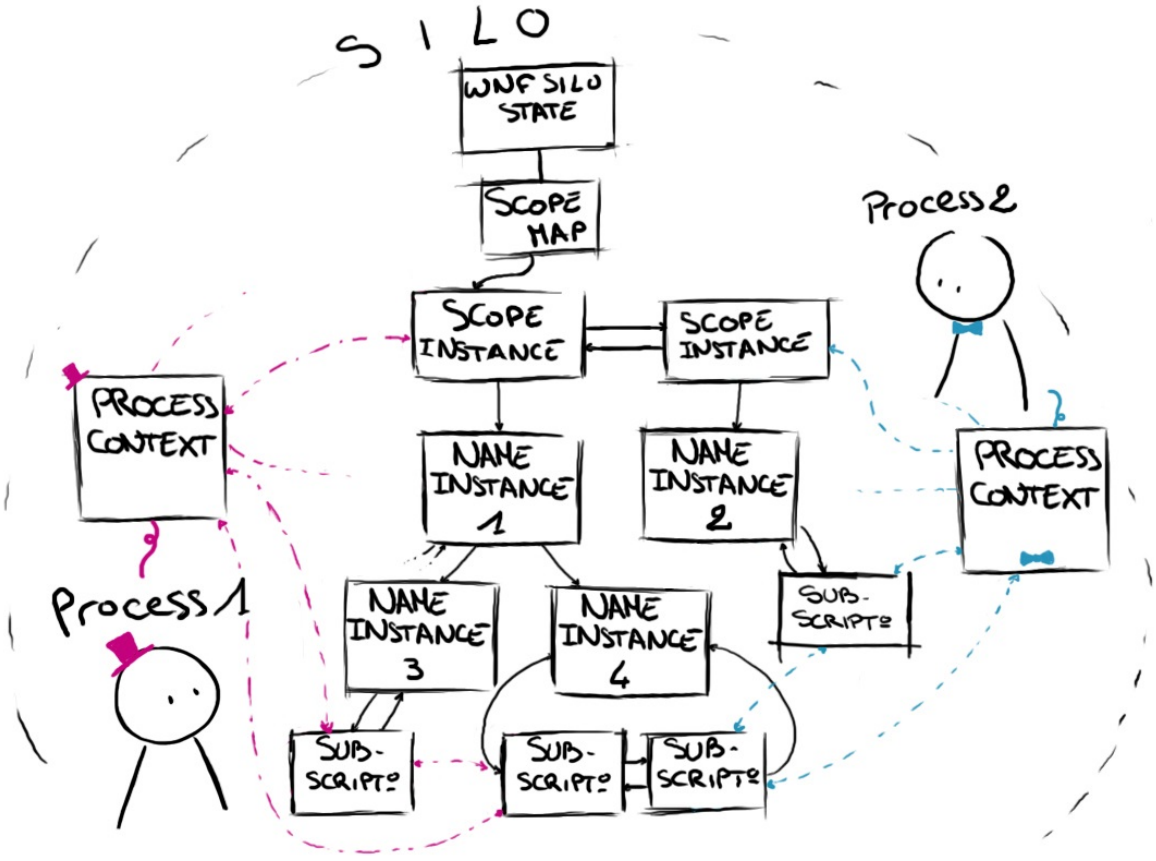
The Windows Notification Facility, or WNF, is a (not really well-known) kernel component used to dispatch notifications across the system. It can be used either in kernel-mode or in user-land with a set of exported (but obviously not documented) API functions and related data structures. An application may *subscribe* a specific type of event (identified by a **StateName**) in order to be notified every time a change to its state occurs (which can be associated with **StateData**). On the other hand, a *publisher* component is responsible for providing the data that will be sent with the notification and to trigger the event.

> It should be noted that WNF state names can be instanced (**Scope**) to a single process, a silo (Windows Container), or to the whole machine. For example, if the application is running inside a silo, it will only be notified for Silo-scoped events happening inside its own container.

In this blog post, I won't speak about the user-land mechanisms involved when using high level API: they are kind of out of scope for the exercise and the explanations would make the blog post a bit too heavy. Alex Ionescu and I gave an in-depth talk about WNF in both of its flavors at BlackHat USA 2018, which should be posting the videos and slides sometime in November 2018 (the release of which is pending some vulnerabilities to be addressed by MSRC).

Data Structures

There are a lot of structures involved in the WNF and here is a simplified view of their relationship in memory:



An *event*, or an instance of a WNF State Name, is represented in memory by a `WNF_NAME_INSTANCE` structure. These structures are sorted in binary trees and linked to the scope in which the event happens. The scopes determine what information a component is able to see or access. They also enable the instantiation of different data for the same State Name.

There are five possible types of scope that are defined as follows:

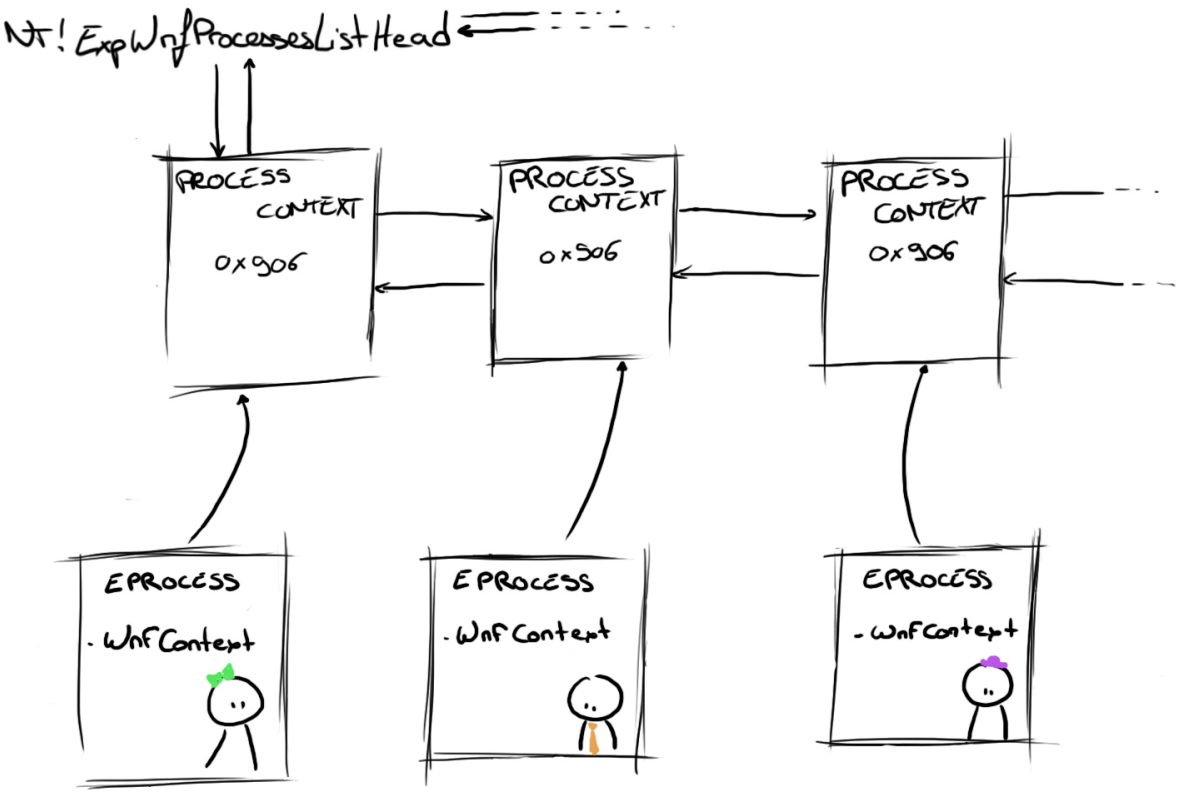
```
typedef enum _WNF_DATA_SCOPE
{
  WnfDataScopeSystem = 0x0,
  WnfDataScopeSession = 0x1,
  WnfDataScopeUser = 0x2,
  WnfDataScopeProcess = 0x3,
  WnfDataScopeMachine = 0x4,
} WNF_DATA_SCOPE;
```

The scopes, identified with `WNF_SCOPE_INSTANCE` structures, are stored by type in doubly linked lists and their heads are kept in a `WNF_SCOPE_MAP` that is silo specific.

When a component subscribes to a WNF State Name, a new `WNF_SUBSCRIPTION` structure is created and added to the linked list belonging to the associated `WNF_NAME_INSTANCE`. If the subscriber is using a low-level API (such as the one that is

described below), a callback is added in the `WNF_SUBSCRIPTION` and called when the component needs to be notified.

A `WNF_PROCESS_CONTEXT` object keeps track of all the different structures involved for a specific subscriber process. It also stores the `KEVENT` used to notify the process. This context is either accessible via the `EPROCESS` object or by crawling the doubly linked list pointed by `nt!ExpWnfProcessesListHead`. Below you'll find a representation of these connections.



In case you're wondering what the `0x906` refers to, this is related to the fact that all of the structures used by WNF have a tiny header (a common occurrence in Windows' File System-related data structures) that describes the structure type and size:

```
typedef struct _WNF_CONTEXT_HEADER
{
    CSHORT NodeTypeCode;
    CSHORT NodeByteSize;
} WNF_CONTEXT_HEADER, *PWNF_CONTEXT_HEADER;
```

This header is pretty handy when debugging as it's quite easy to spot the objects in memory. Here are some node type codes for the WNF structures:

```
#define WNF_SCOPE_MAP_CODE          ((CSHORT)0x901)
#define WNF_SCOPE_INSTANCE_CODE     ((CSHORT)0x902)
#define WNF_NAME_INSTANCE_CODE      ((CSHORT)0x903)
#define WNF_STATE_DATA_CODE         ((CSHORT)0x904)
#define WNF_SUBSCRIPTION_CODE       ((CSHORT)0x905)
#define WNF_PROCESS_CONTEXT_CODE    ((CSHORT)0x906)
```

Reverse time

Now that we have some background, let's start the exercises! The first part was to actually reverse the following functions in order to understand their purpose:

- `ExSubscribeWnfStateChange`
- `ExQueryWnfStateData`

ExSubscribeWnfStateChange

```
NTSTATUS
ExSubscribeWnfStateChange (
    _Out_ptr_ PWNF_SUBSCRIPTION* Subscription,
    _In_ PWNF_STATE_NAME StateName,
    _In_ ULONG DeliveryOption,
    _In_ WNF_CHANGE_STAMP CurrentChangeStamp,
    _In_ PWNF_CALLBACK Callback,
    _In_opt_ PVOID CallbackContext
    );
```

`ExSubscribeWnfStateChange` enables the registration of a new subscription in the WNF engine. It takes as parameters, among other things, a `StateName` specifying the type of event we are interested in and a `Callback` that will be called whenever the notification is triggered. It also returns a new `Subscription` pointer that can be used to query data associated with the notification.

Internally, this function only transfers execution flow to its private counterpart (`ExpWnfSubscribeWnfStateChange`) which handles all the processing.

As WNF State Names are stored in an opaque format, `ExpWnfSubscribeWnfStateChange` first decodes the "clear" version of the ID with `ExpCaptureWnfStateName`.

This clear WNF State Name can be decoded as follows:

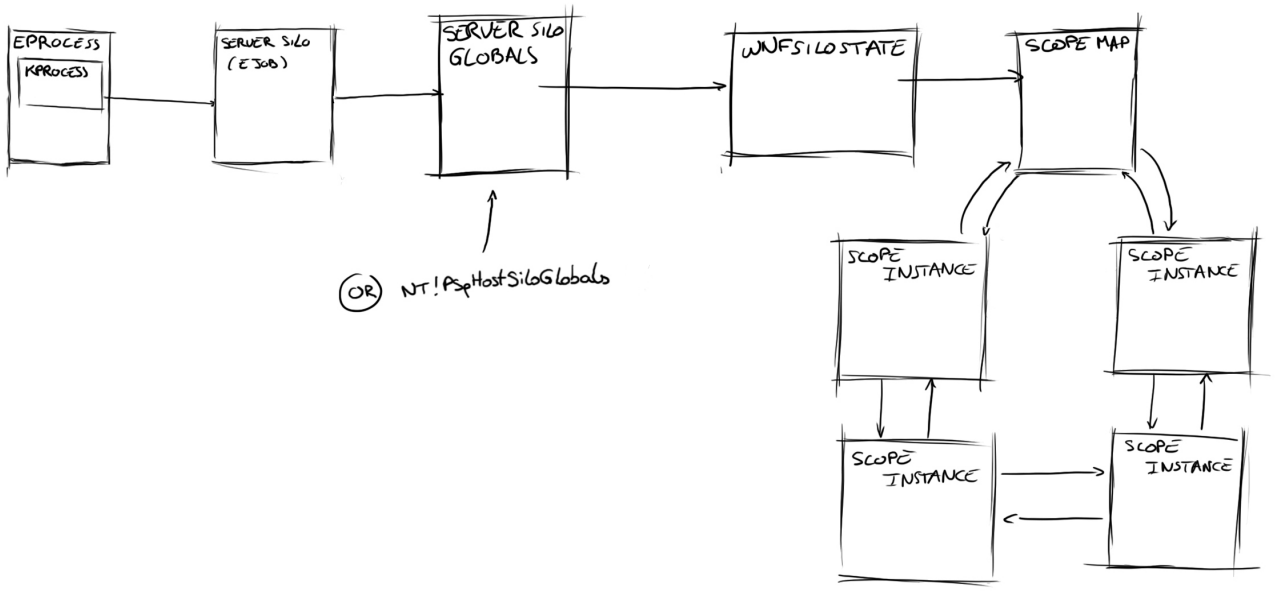
```
#define WNF_XOR_KEY 0x41C64E6DA3BC0074

ClearStateName = StateName ^ WNF_XOR_KEY;
Version = ClearStateName & 0xf;
LifeTime = (ClearStateName >> 4) & 0x3;
DataScope = (ClearStateName >> 6) & 0xf;
IsPermanent = (ClearStateName >> 0xa) & 0x1;
Unique = ClearStateName >> 0xb;
```

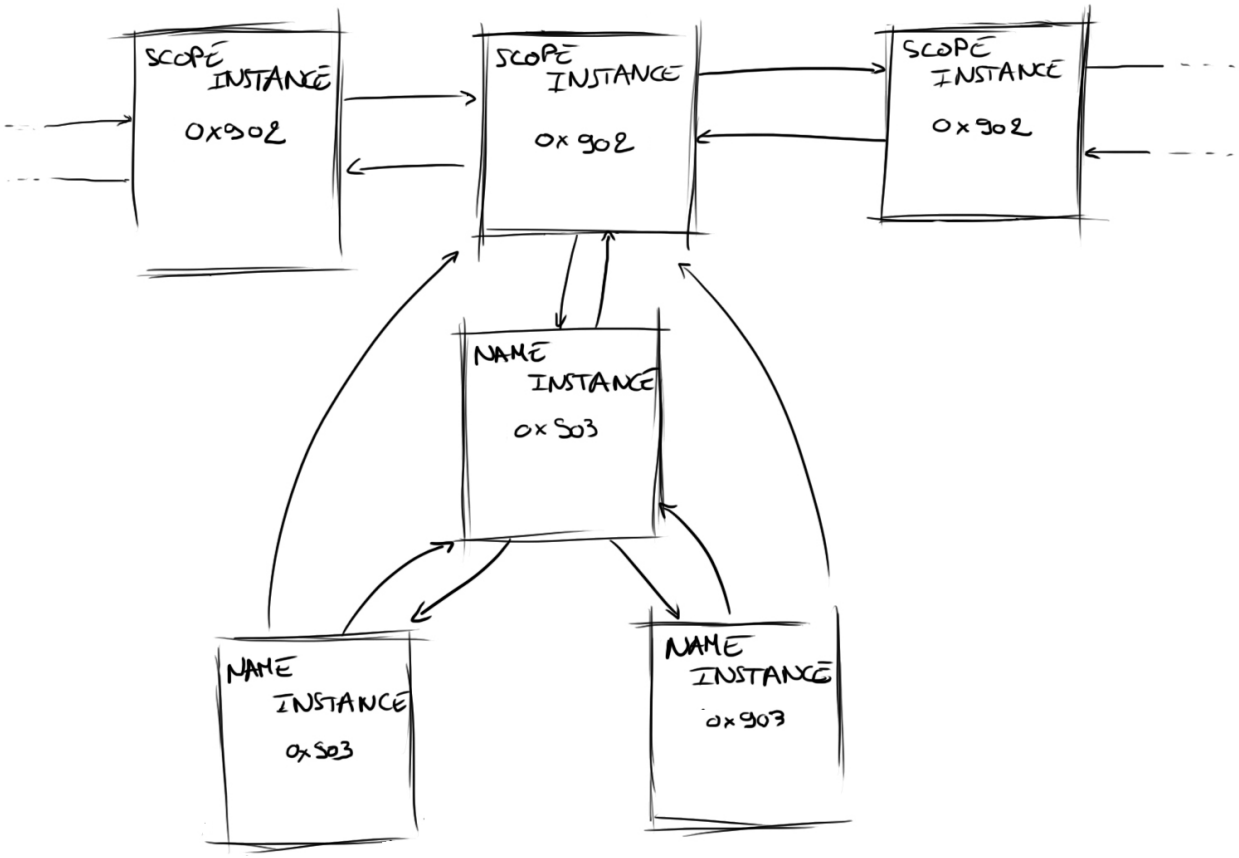
In a more formal way, this gives the following structure:

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

Then, `ExpWnfSubscribeWnfStateChange` calls `ExpWnfResolveScopeInstance`. This latter retrieves the *Server Silo Globals* (or `nt!PspHostSiloGlobals` in the case where no server silo is involved) and goes through several structures in order to find the `WNF_SCOPE_INSTANCE` to which the name instance belongs. If this Scope Instance doesn't exist, it is created and added to the corresponding `WNF_SCOPE_MAP` list. This is shown below:



From this Scope Instance structure, `ExpWnfSubscribeWnfStateChange` searches (with `ExpWnfLookupNameInstance`) for a `WNF_NAME_INSTANCE` matching the given WNF State Name.



If no match is found, it creates a new `WNF_NAME_INSTANCE` by using `ExpWnfCreateNameInstance`. This new instance is added to the binary tree rooted off the `WNF_SCOPE_INSTANCE`.

The next step for the function is to call `ExpWnfSubscribeNameInstance` to create a new subscription object. As it was explained previously, this object will hold all the information that the engine needs in order to trigger notifications.

Finally, `ExpWnfSubscribeWnfStateChange` calls `ExpWnfNotifySubscription` to insert the new subscription in a pending queue and trigger a notification.

ExQueryWnfStateData

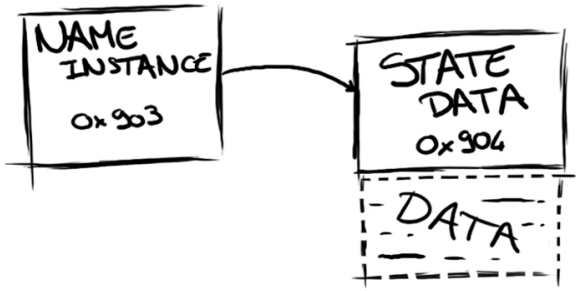
```
NTSTATUS
ExQueryWnfStateData (
    _In_ PWNF_SUBSCRIPTION Subscription,
    _Out_ PWNF_CHANGE_STAMP ChangeStamp,
    _Out_ PVOID OutputBuffer,
    _Out_ OPULONG OutputBufferSize
);
```

This function is quite simple as it only performs two actions. First it retrieves the `WNF_NAME_INSTANCE` from the `Subscription` with `ExpWnfAcquireSubscriptionNameInstance`. Then, it reads the data stored in it with

`ExpWnfReadStateData` and tries to copy it in `Buffer`. If this buffer is too small, it will only write the size needed in `OuputBufferSize` and return with `STATUS_BUFFER_TOO_SMALL`.

For the record, all WNF State Names store their data in memory under a `WNF_STATE_DATA` structure. This structure holds various metadata such as the data size and the number of times it has been updated (called the `ChangeStamp`). A pointer to the `WNF_STATE_DATA` is kept directly in the `WNF_NAME_INSTANCE`, as shown below.

> Alex would also like me to point out that WNF State Names can be marked as *persistent*, which means that the data (and change stamp) will be retained accross reboots (obviously by using a secondary data store). More details on this will be available in our presentation!



Let's code!

Basically, with the functions we reversed, we should be able to register a new subscription and be notified as any other legitimate application using WNF :)

However we are still missing one element: finding the required WNF State Name for microphone input.

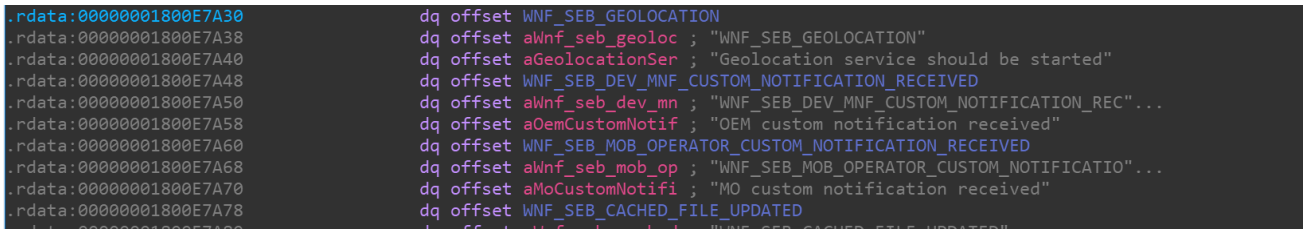
> I will only detail the parts of the driver that are relevant to interacting with WNF. If you are interested in driver development on Windows, you might want to take a look at the Windows Driver Kit documentation and their samples, or better yet, directly attend one of Bruce's training courses ;)

Looking for the right WNF State Name

As a hint for retrieving the WNF State Name, Bruce provided a link to a blog post and the name of a library (`contentDeliveryManager_Utilities.dll`).

In their blog entry, Redplait defined several State Names used by WNF. Unfortunately, the one we are looking for is not listed. However, that still gives us a good start, for we now know what WNF State Names look like.

One naive approach to find what we are looking for is to `grep` for one of the blog's WNF State Names in `contentDeliveryManager_Utilities.dll` and hope that other IDs will be around... Luckily, this works pretty well! By following the cross reference of the matched pattern in IDA, we can reach a full list of WNF State Names referenced in the DLL. Each entry in this list comes with its name and description which is really handy for our purposes! (For further information, this list is used by `GetWellKnownWnfStateByName`).



We now just have to look for the one specific to the microphone (remember the exercise? :p)

```
.rdata:00000001800E3680  dq offset WNF_AUDC_CAPTURE
.rdata:00000001800E3688  dq offset aWnf_audc_captu ; "WNF_AUDC_CAPTURE"
.rdata:00000001800E3690  dq offset aReportsTheNu_0 ; "Reports the number of, and
process ids "...
// “Reports the number of, and process ids of all applications currently capturing
audio.
//  Returns a WNF_CAPTURE_STREAM_EVENT_HEADER data structure”
```

> It should be noted that the same table is also available with the``perf_nt_c.dll`` library that is included with Windows Performance Analyzer.

Subscribing to the event

To subscribe a new event, we just have to call `ExSubscribeWnfStateChange` in our driver with the WNF State Name we found out from above. This function is exported but not defined in any header, therefore we have to manually declare it ourselves by pasting the definition from above. Note that `ntoskrnl.lib` contains the import library stub, so there is no need to retrieve its address by hand (thanks Alex for the protip ;)).

The only thing that needs to be done here is to call the function with the right parameters:

```
NTSTATUS
CallExSubscribeWnfStateChange (
    VOID
    )
{
    PWNF_SUBSCRIPTION wnfSubscription= NULL;
    WNF_STATE_NATE stateName;
    NTSTATUS status;

    stateName.Data = 0x2821B2CA3BC4075; // WNF_AUDC_CAPTURE

    status = ExSubscribeWnfStateChange(&wnfSubscription, &stateName, 0x1, NULL,
&notifCallback, NULL);
    if (NT_SUCCESS(status)) DbgPrint("Subscription address: %p\n",
Subscription_addr);

    return status;
}
```

Defining the callback

As we saw previously, `ExSubscribeWnfStateChange` takes among its parameters a callback that will be called every time the event is triggered. This callback will be used to get and process event data relative to the notification.

The callback prototype looks like that:

```
NTSTATUS
notifCallback (
    _In_ PWNF_SUBSCRIPTION Subscription,
    _In_ PWNF_STATE_NAME StateName,
    _In_ ULONG SubscribedEventSet,
    _In_ WNF_CHANGE_STAMP ChangeStamp,
    _In_opt_ PWNF_TYPE_ID TypeId,
    _In_opt_ PVOID CallbackContext
    );
```

To obtain the data in our callback, we have to call `ExQueryWnfStateDataName`. Once again this function is exported but not defined in any header so we have to define it ourselves:

```
NTSTATUS
ExQueryWnfStateData (
    _In_ PWNF_SUBSCRIPTION Subscription,
    _Out_ PWNF_CHANGE_STAMP CurrentChangeStamp,
    _Out_writes_bytes_to_opt_(*OutputBufferSize, *OutputBufferSize) PVOID
OutputBuffer,
    _Inout_ PULONG OutputBufferSize
    );
[...]
```

We need to call this API twice: once to get the size needed to allocate a buffer for the data and the second time to actually retrieve the data.

```
NTSTATUS
notifCallback(...)
{
    NTSTATUS status = STATUS_SUCCESS;
    ULONG bufferSize = 0x0;
    PVOID pStateData;
    WNF_CHANGE_STAMP changeStamp = 0;

    status = ExQueryWnfStateDataFunc(Subscription, &changeStamp, NULL, &bufferSize);
    if (status != STATUS_BUFFER_TOO_SMALL) goto Exit;

    pStateData = ExAllocatePoolWithTag(PagedPool, bufferSize, 'LULZ');
    if (pStateData == NULL) {
        status = STATUS_UNSUCCESSFUL;
        goto Exit;
    }

    status = ExQueryWnfStateDataFunc(Subscription, &changeStamp, pStateData,
&bufferSize);
    if (NT_SUCCESS(status)) DbgPrint("## Data processed: %S\n", pStateData);

    [...] // do stuff with the data

Exit:
    if (pStateData != nullptr) ExFreePoolWithTag(pStateData, 'LULZ');
    return status;
}
```

Cleaning the mess when unloading the driver

If you blindly try the code above, you'll get an ugly blue screen of death as I painfully learnt the first time I tried the exercise and unloaded my driver! :P We need to delete our subscription beforehand.

For this purpose, we can call `ExUnsubscribeWnfStateChange` in the driver unload routine (and make sure that `PWNF_SUBSCRIPTION wnfSubscription` is made into a global variable).

```
PVOID
ExUnsubscribeWnfStateChange (
    _In_ PWNF_SUBSCRIPTION Subscription
    );

VOID
DriverUnload (
    _In_ PDRIVER_OBJECT DriverObject
    )
{
   [...]
   ExUnsubscribeWnfStateChange(g_WnfSubscription);
}
```

Quite an epic fail

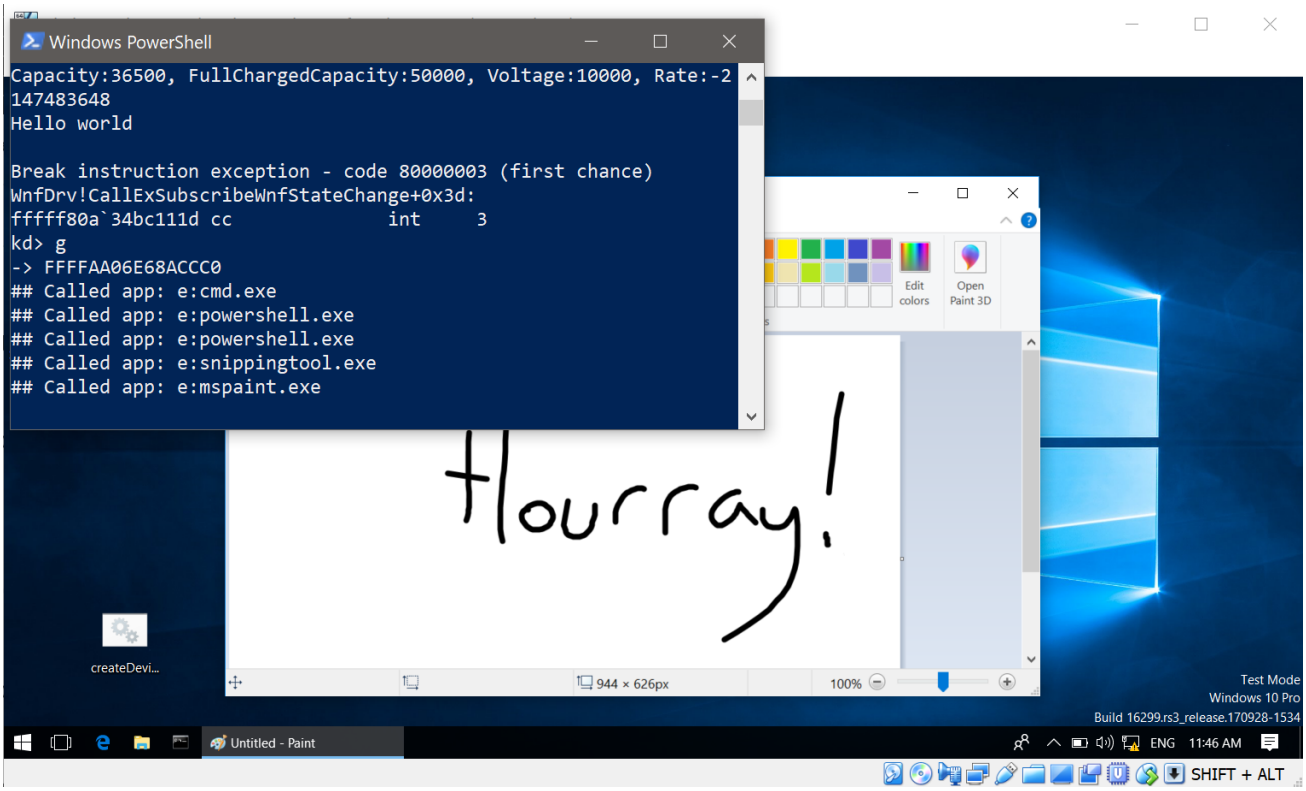
All we have to do now is to start the driver, enable Cortana, play a little bit with it and wait for the event to trigger.

...

Aaaand...! Nothing!

My exercise result totally failed as I forgot that I didn't have a sound card on my Virtual Machine (probably the reason why I couldn’t start any of the sound related apps..? >.>’) and most importantly, due to my host configuration, I couldn't make it work at all (don't ask).

Still, in order to ensure that my driver works correctly I had to choose another event (sigh) and went for `WNF_SHEL_DESKTOP_APPLICATION_STARTED`. This notification is signaled anytime a desktop application launches. In return, it simply outputs the application name that was started. With this WNF State Name, it was pretty straightforward to get some results:



Keeping up to date with WNF

I showed earlier a simple way of looking up WNF names by just searching for the name in IDA in one of the DLLs that contains the table. A more reliable way and extensible approach would be to retrieve the WNF State Names by parsing the DLL to find the table and dumping it. Although this wasn't the approach I used in the exercise, Alex had a need for easily keeping up to date with changes in the WNF State Names (additions/removals/modifications) due to his unhealthy obsession with `diff`-ing every build of the kernel, and I came up with a script to help him do so.

Since I forgot to mention it at BlackHat, I feel like doing some advertising now. :-^ This script can be used to diff two DLLs and quickly get the differences in the tables, as well as just dump the table data from a single DLL. The output is one that Alex and I used for our `wnftool` applications, and can easily be adapted for use in other C and Python programs.

```
$ 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 Output file (Default: output.txt)
    -py, --python Change the output language to python (by default it's c)
```

Example of output (I will publish the script once the November embargo is lifted):

```
typedef struct _WNF_NAME
{
    PCHAR Name;
    ULONG64 Value;
} WNF_NAME, *PWNF_NAME;

WNF_NAME g_WellKnownWnfNames[] =
{
    {"WNF_A2A_APPURIHANDLER_INSTALLED", 0x41877c2ca3bc0875}, // An app implementing
windows.AppUriHandler contract has been installed
    {"WNF_AAD_DEVICE_REGISTRATION_STATUS_CHANGE", 0x41820f2ca3bc0875}, // This event
is signalled when device changes status of registration in Azure Active Directory.
    {"WNF_AA_CURATED_TILE_COLLECTION_STATUS", 0x41c60f2ca3bc1075}, // Curate tile
collection for all allowed apps for current AssignedAccess account has been created
    {"WNF_AA_LOCKDOWN_CHANGED", 0x41c60f2ca3bc0875}, // Mobile lockdown
configuration has been changed
    [...]
}
```

Conclusion and Acknowledgment

Thanks to this exercise I had the opportunity to dig into a kernel component I didn't know at all and which is quite fun to play with. I learned a lot of things and really enjoyed trying to figure out how to use WNF. I definitely spent a marvelous week at Recon and I'm super happy to have been there.

So once again I would like to thank Bruce for the workshop and for this exercise. I'm so glad I could attend. That rocks! :)

I would also like to thank very much Alex for letting me tag along at BlackHat and for mentoring me in the process. I really appreciate that he tries to make me find the answers to my questions on my own, by giving me hints and making me question my own assumptions. Figuring out things you don't understand, on your own, is really rewarding! He is definitely a great pedagogue and I'm still amazed by the amount of knowledge I get from him.

And, while I was reluctant to finish this blog post, I'm sure I gained a lot by writing it so I'm also quite happy he pushed me to continue. :)

I'm really pleased to have met him and hope we will work together again at some point!

And at last, I'm really grateful to my colleagues for being awesome (;D) and for proofreading/fixing issues on this article.
