Exploiting a “Simple” Vulnerability – Part 1.5 – The Info Leak

windows-internals.com/exploiting-a-simple-vulnerability-part-1-5-the-info-leak

By Yarden Shafir

Introduction

This post is not actually directly related to the first one and does not use `CVE-2020-1034`. It just talks about a *second* vulnerability that I found while researching ETW internals, which discloses the approximate location of the `NonPaged` pool to (almost) any user. It was spurred by a tweet that challenged me to find an information leak. It turns out I found one that wasn’t actually patched after all!

The vulnerability itself is not especially interesting, but the process of finding and understanding it was fun so I wanted to write about that. Also, when I reported it Microsoft marked it as “Important” but would not pay anything for it and eventually marked it as “won’t fix” even though fixing this issue takes less time than writing an email, so the annoyance factor alone makes writing this post worth it. And this is a chance to rant about some more ETW internals stuff which didn’t really fit into any of the other posts, so you can read them or skip right to the PoC, your choice.

Update

This vulnerability was eventually acknowledged by Microsoft and received CVE-24107. It was fixed on 9/3/2021.

More ETW Internals!

Remember that the first thing you learn about ETW notifications are that they are asynchronous? Well, that was a lie. Sort of. *Most* ETW notifications really are asynchronous. However, in the previous blog post we used a vulnerability that relied on improper handling of the `ReplyRequested` field in the `ETWP_NOTIFICATION_HEADER` structure. The existence of this field implies that you can reply to an ETW event. But no one ever told you that you can *reply* to an ETW notification, how would that even work?

Normally, ETW works just the way you were told. That is the case for all Windows providers, and any other ETW provider I could find. But there is a “secret setting” that happens when someone notifies an ETW provider with `ReplyRequested = 1`. Then, as we saw in the previous blog post, the notification gets added to a reply queue and is waiting for a reply. Remember, there can only be `4` queues notifications waiting for a reply at any moment. When that happens, any process which registered for that provider has its registered callback

notified and has a chance to reply to the notification using `EtwReplyNotification`. When someone replies to the notification, the original notification gets removed from the queue and the reply notification gets added to the reply queue.

The only case I could see so far where a reply is sent to a notification is immediately after a GUID is enabled – `sechost!EnableTraceEx2` (which is the standard way of registering a provider and enabling a trace) has a call to `ntdll!EtwSendNotification` with `EnableNotificationPacket->DataBlockHeader.ReplyRequested` set to `1`. That creates an `EtwRegistration` object, so before returning to `Sechost`, `Ntdll` immediately replies to the notification with `NotificationHeader->NotificationType` set to `EtwNotificationTypeNoReply`, simply to get it removed from the notification queue.

Specifically, in this case, something a little more complicated happens. Even though `Ntdll` is enabling the GUID, it's not the "owner" of the registration instance and therefore doesn't have a registered callback (since this belongs to whoever registered the provider).
Yet `Ntdll` still needs to know when the kernel enables the provider, to queue the reply notification – it can't expect the caller to know that this needs to be done. So to do this, it uses a trick.

When `EtwRegisterProvider` is called, it calls `EtwpRegisterProvider`. The first time this function is called, it calls `EtwpRegisterTpNotificationOnce`:

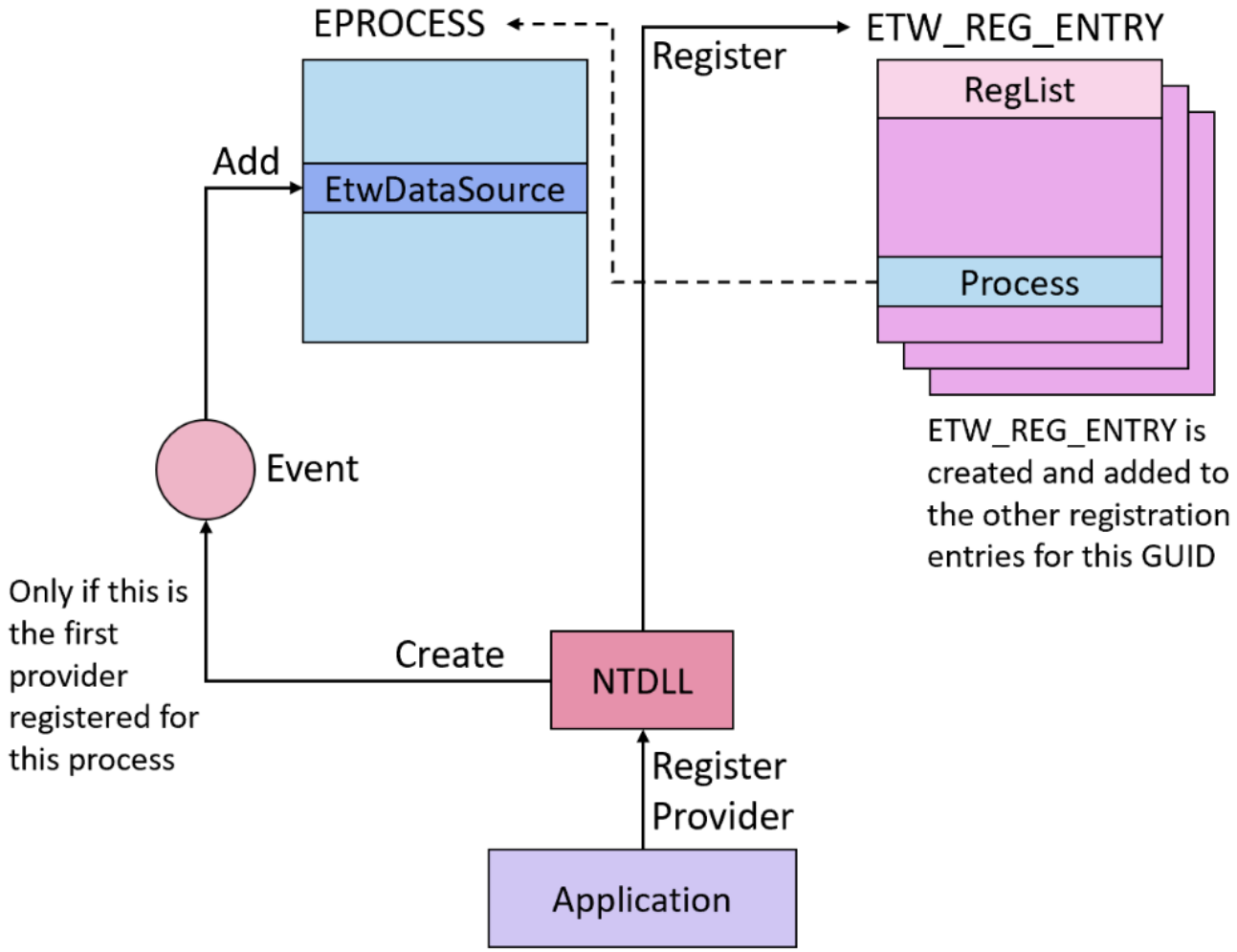
```
BOOLEAN __stdcall EtwpRegisterTpNotificationOnce()
{
  DWORD BytesReturned; // [rsp+30h] [rbp-20h] BYREF
  HANDLE Handle; // [rsp+38h] [rbp-18h]
  void *waitReturn; // [rsp+40h] [rbp-10h] BYREF
  int InputBuffer; // [rsp+78h] [rbp+28h] BYREF

  waitReturn = 0i64;
  Handle = NULL;
  if ( (int)ZwCreateEvent() >= STATUS_SUCCESS )
  {
    if ( (int)TpAllocWait(&waitReturn, EtwpNotificationThread, Handle, 0i64) >= STATUS_SUCCESS )
    {
      TpSetWaitEx(waitReturn, Handle, 0i64, 0i64);
      InputBuffer = (int)Handle;
      if ( (int)NtTraceControl(27u, &InputBuffer, 4u, NULL, 0, &BytesReturned) >= STATUS_SUCCESS )
      {
        return TRUE;
      }
    }
    if ( waitReturn )
    {
      TpReleaseWait(waitReturn);
    }
  }
  if ( Handle )
  {
    NtClose(Handle);
  }
  return FALSE;
}
```

Without getting into too many internal details about waits and the thread pool, this function essentially creates an event with the callback function `EtwpNotificationThread` and then calls `NtTraceControl` with an Operation value of `27` – an undocumented and unknown value. Looking at the kernel side of things, it's not too hard to give this value a name:

```
case 27:
    if ( inputBufferLen != 4 )
    {
        goto InvalidParameter;
    }
    eventHandle = *Src;
    if ( !eventHandle )
    {
        goto InvalidParameter;
    }
    status = EtwpAddNotificationEvent(eventHandle, wow64);
    goto Exit;
```

I'll call this operation `EtwAddNotificationEvent`.

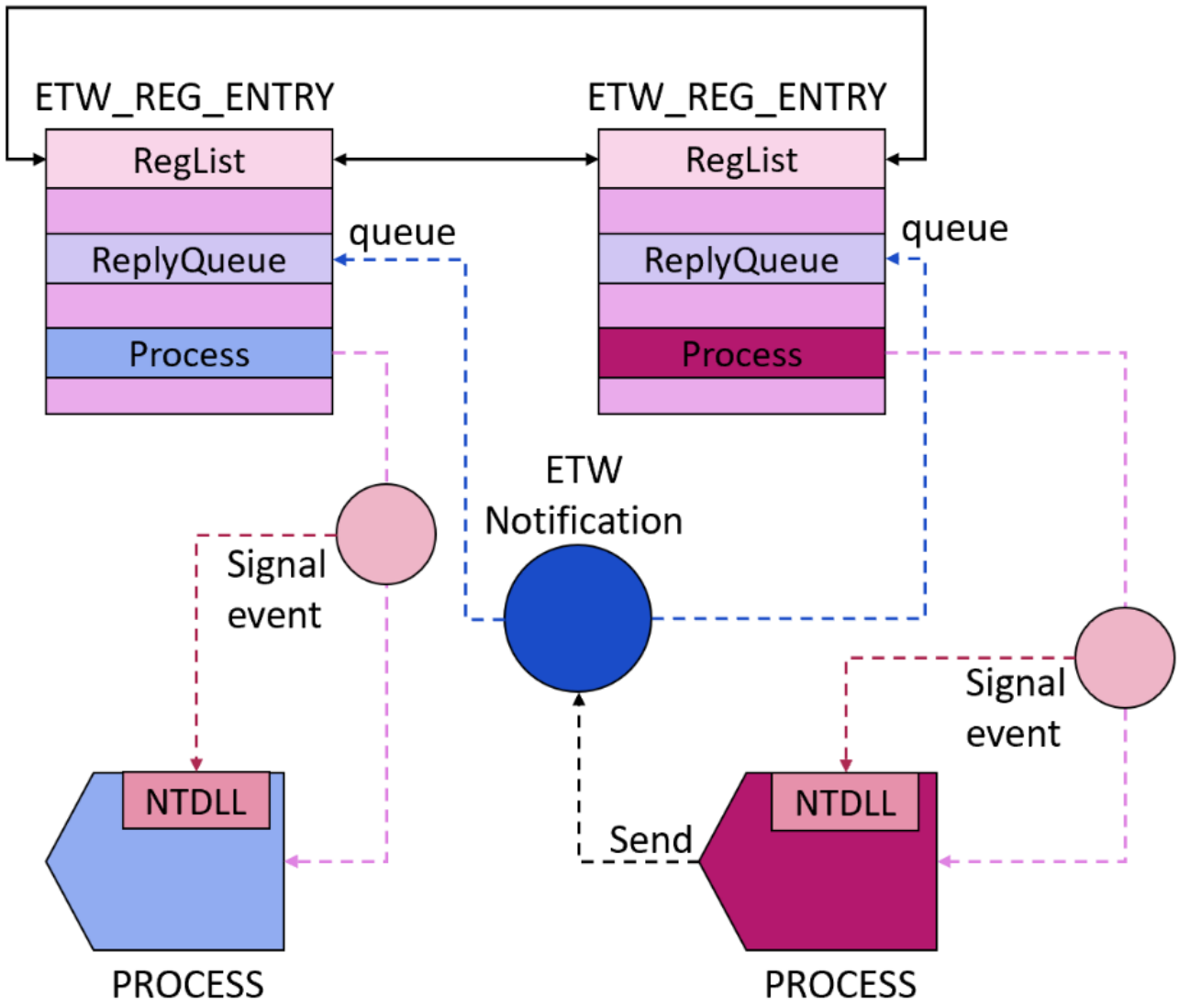
`EtwpAddNotificationEvent` is a pretty simple function: it receives an event handle, grabs the event object, and sets `EventDataSource->NotificationEvent` in the `EPROCESS` of the current process to the event (or `NotificationEventWow64`, if this is a `WoW64` process). Since this field is a pointer and not a list, it can only contain one event at a time. If this field is not set to `0`, the value won't be set and the caller will receive `STATUS_ALREADY_REGISTERED` as a response status.



Then, in `EtwpQueueNotification`, immediately after a notification is added to the notification queue for the process, this event is signaled:

```
dataSource->NotificationQueue.Blink = &newQueueEntry->ListEntry;
if ( foundEntry )
{
    if ( wow64 )
    {
        notificationEvent = dataSource->NotificationEventWow64;
    }
    else
    {
        notificationEvent = dataSource->NotificationEvent;
    }
    if ( notificationEvent )
    {
        KeSetEvent(notificationEvent, EVENT_INCREMENT, FALSE);
    }
}
```

The event being signaled makes the `EtwpNotificationCallback` get called, since it was registered to wait on this event, so it is, in a way, an ETW notification callback that is being notified whenever the process receives an ETW notification. However, this function is not a real ETW notification callback, so it doesn't receive the notification as any of its parameters and has to somehow get it by itself in order to reply to it. Luckily, it has a way to do that.



The first thing that `EtwpNotificationThread` does is make another call to `NtTraceEvent`, this time with operation number `16` – `EtwReceiveNotification`. This operation leads to a call to `EtwpReceiveNotification`, which chooses the first queued notification for the process (and matching the process' `WoW64` status) and returns it. This operation requires no input arguments – it simply returns the first queued notification. This gives `EtwpNotificationThread` all the information that it needs to reply to that last queued notification quietly, without disturbing the unaware caller that simply asked it to register a provider. After replying, the event is set to a waiting state again, to wait for the next notification to arrive.

Most of this pretty long explanation has nothing to do with this vulnerability, which really is pretty small and simple and can be explained in a much less complicated way. But I did say this post was mostly an excuse to dump some more obscure ETW knowledge in hope that one day someone other than me will read it and find it helpful, so you all knew what you were getting into.

And now that we have all this unnecessary background, we can look at the vulnerability itself.

The InfoLeak

The issue is actually in the last part we talked about – returning the last queued notification. If you remember from the last post, when a GUID is notified and the notification header has `ReplyRequested == 1`, this leads to the creation of a kernel object which will be placed in the `ReplyObject` field of the notification that is later put in the notification queue. And this is the same structure that can be retrieved using `NtTraceControl` with `EtwReceiveNotification` operation... Does that mean that we get a free kernel pointer by calling `NtTraceControl` with the right arguments?

Not exactly. To be precise, you get *half* of a kernel pointer. Microsoft didn't completely ignore the fact that retuning kernel pointers to user-mode callers is a bad idea, like they did in so many other cases. The `ReplyObject` field in `ETWP_NOTIFICATION_HEADER` is in a union with `ReplyHandle` and `RegIndex`. And after copying the data to the user-mode buffer, they set the value of `RegIndex`, which should overwrite the kernel pointer that is in the same union:

```
replyCount = _InterlockedIncrement((volatile signed __int32 *)&dataBlock->20);
memmove(Buffer, dataBlock, dataBlock->NotificationSize);
Buffer->ReplyCount = replyCount;
Buffer->RegIndex = queueEntry->RegIndex;
if ( dataBlock->ReplyRequested == 1 )
    Buffer->ReplyIndex = queueEntry->ReplyIndex;
```

The only thing that this code doesn't account for is the fact that `ReplyObject` and `RegIndex` don't have the same type: `ReplyObejct` is a pointer (`8` bytes on `x64`) while `RegIndex` is a `ULONG` (`4` bytes on `x64`). So setting `RegIndex` only removes the bottom half of the pointer, leaving the top half to be returned to the caller:

```
C:\Users\Administrator\Desktop>pool_address_leak.exe
reply object: 0xFFFF9D0800000008
```

Triggering this is extremely simple and includes exactly three steps:

1. Register a provider

2. Queue a notification where `ReplyObject` is a kernel object – do this by calling `NtTraceControl` with `operation == EtwSendDataBlock` and `ReplyRequested == TRUE` in the notification header.
3. Call `NtTraceControl` with `operation == EtwReceiveNotification` and get your half of a kernel pointer.

It's true that the top half of a kernel address is not all that much, but it can still give a caller a better guess of where the `NonPagedPool` (where those objects are allocated) is found. In fact, since the `NonPagedPool` is sized `16TB` (or `0x100000000000` bytes), this vulnerability tells us exactly where the `NonPaged` pool is, and we can validate that in the debugger:

```
!vm 21
...
System Region                Base Address    NumberOfBytes
SecureNonPagedPool    : ffff838000000000       8000000000
KernelShadowStacks    : ffff888000000000       8000000000
PagedPool             : ffff8a0000000000     100000000000
NonPagedPool          : ffff9d0000000000     100000000000
SystemCache           : ffffb00000000000     100000000000
SystemPtes            : ffffc40000000000     100000000000
UltraZero             : ffffd40000000000     100000000000
Session               : ffffe40000000000       8000000000
PfnDatabase           : ffffe78000000000      c8000000000
PageTables            : fffff40000000000       8000000000
SystemImages          : fffff80000000000       8000000000
Cfg                   : fffffaf0ea2331d0      28000000000
HyperSpace            : fffffd0000000000      10000000000
KernelStacks          : fffffe0000000000      10000000000
```

This can be triggered from almost any user, including `Low IL` and `AppContainer`, where most of the classic infoleaks don't work anymore, this might be of some use, even if a limited one.

I believe that when this code was introduced, it was completely safe – those areas of the code are pretty ancient and get very few changes. This code was probably introduced in the days before `x64`, when the size of a pointer and the size of a `ULONG` was the same, so setting `RegIndex` did overwrite the whole object address. When `x64` changed the size of a pointer, this code was left behind and was never updated to match this, so this bug appeared.

This makes you wonder what similar bugs might exist in other pieces of ancient code that even Microsoft forgot about?

Just Show Me the Code Already!

In case you want to see the three lines of code that trigger this bug, you can find them here.