Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Studying the logging capability of Windows Telemetry component using Reverse Engineering

Tesis de Licenciatura en Ciencias de la Computación

Pablo Agustín Artuso
LU: 282/11
artusopablo@gmail.com

Director: Rodolfo Baader <rbaader@dc.uba.ar>

Codirector: Aleksandar Milenkoski <amilenkoski@ernw.de>

Buenos Aires, 2019

ABSTRACT (ENGLISH VERSION)

Windows, one of the most popular OS, has a component called Telemetry. It collects information from the system with the goal of analyzing and fixing software & hardware problems, improving the user experience, among others. The kind of information that can be obtained by this component is partially configurable in four different levels: security, basic, enhanced and full, being "security" the level where less information is gathered and "full" the opposite case.

How Telemetry stores/process/administrates the information extracted? It employs a widely used framework called Event Tracer for Windows (ETW) [4]. Embedded not only in userland application but also in the kernel modules, the ETW framework has the goal of providing a common interface to log events and therefore help to debug and log system operations, by instrumenting it.

In this work, we are going to analyze a part of the Windows kernel to better understand how Telemetry works from an internal perspective. This work will make windows analysts, IT admins or even windows users, more aware about the functionality of the Telemetry component helping to deal with privacy issues, bug fixing, knowledge of collected data, etc. Our analysis implies performing reverse engineering [5],[6] on the Telemetry component, which involves chal- lenging processes such as kernel debugging, dealing with undocumented kernel internal structures, reversing of bigger frameworks (i.e: ETW), binary libraries which lack symbols, etc. As part of the analysis we will also develop an in depth comparison between the differences among each level of Telemetry, stressing the contrast in the amount of events written, verbosity of information, etc. Finally, we will study how the communication between the Windows instance and the Microsoft backend servers is carried out.

ABSTRACT (SPANISH VERSION)

ACKNOWLEDGMENTS

CONTENTS

1. MOTIVATION

The analysis presented in this work was performed against the Telemetry component of the Windows OS, with the following goals:

- Understand how the process of generating logs was carried out.
- How, where and what logs were stored.
- Which applications are involved in gathering Telemetry information

2. INTRODUCTION

The analysis presented, was carried out in a particular version of the Windows OS. Specifically, Windows 10 64 bits Enterprise, 1607. Windows delivers a new version, usually, each Some words about ERNW and the project with the FBI

There are several reasons why this version was chosen:

- It was one of the mainstream version of Windows at the moment of starting the project.
- It was a long support version (EOS: April 2019).
- It was used by the German Police Office

Although this version may sound a bit old as of today, all the analysis is also applicable to newer versions such as Windows 10 64 bits Enterprise 1909.

2.1 Basic concepts

This section will describe basic concepts needed to fully understand the carried out process to perform the analysis.

2.1.1 Reverse Engineering

Software engineering can be defined as the process of designing, building and testing computer software. The processor of a computer, works with 1's and 0's, therefore developing any kind of sofware will mean Key concepts about what RE means, from a general perspective, how it should tackled which tools are usually there. 64 bits, calling convention ..

Static and dinaymyc analysis!

2.1.2 Debugging

Explaniation of the concept of debugging, what is the different with doing static RE. Some words specifically for KERNEL debugging.

2.2 Tools

2.2.1 IDA pro

Introduction to IDA pro. Explaniation of what it is and how it works.talk about hxrays

2.2.2 WinDBG

Introduction to WINDBG. Explaniation of what it is and how it works.

2.2.3 YARA

2.2.4 XPERF?

2.3 Windows components

2.3.0.1 Event Tracing for Windows

Complete explainiation of how it works due to its importance for the rest of the analysis. Different components: session, providers, consumers ,etc . Talk about the guid of the providers

2.3.0.2 Telemetry

Full description of the different features / characteristic which are involved in this analysis. Explaniation of how the worflow of the data is followed. talk about the name of the session,the levels of configuration, when it can be configured.. etc.

3. PREVIOUS WORK

Some lines about previous works in this topic. Most of them focused on just analysis from traffic / documentation.

4. ANALYSIS

The main objective was always the analysis of Telemetry. The best and most accurate option to study this component would have been to analyze it source code. Unluckily this isn't possible as the Windows kernel is not open source. However, it was still possible to reverse engineer the Windows kernel to understand how Telemetry works. Several files (dynamic libraries, executables, drivers) had to be reversed and analyzed. Nonetheless, there was one file that had the main focus: **ntoskrnl.exe**. This binary held the actual implementation of the Windows Kernel.

Following sections will depict different challenges and achievements faced during the analysis of this and the other binaries.

4.1 Understanding how Telemetry makes use of ETW

When Windows OS boots, multiple sessions are created inside ETW. Among them there is one related to Telemetry: DiagTrack. As explained in 2.3.0.1, every session has "providers" -those entities that actually provide information to the session-. In order to understand how Telemetry made use of ETW, it was important to learn about DiagTrack's session providers. At this point, the answers for the following questions were unknown:

1. Who are they?
2. Where are they?
3. What information are they logging?

Windows allowed to query the list of providers registered to a particular session by executing the following powershell command:

```
Get-EtwTraceProvider | where {$_.SessionName -match
"<SESSION_NAME>"}
```

Fig. 4.1: Powershell command to list ETW providers registered against a particular session.

For each provider registered to the queried session, the following information is outputted:

- GUID
- TO_CHECK
- TO_CHECK

With this list, the question **Who are they?** seemed to be answered. Still, two questions remained.

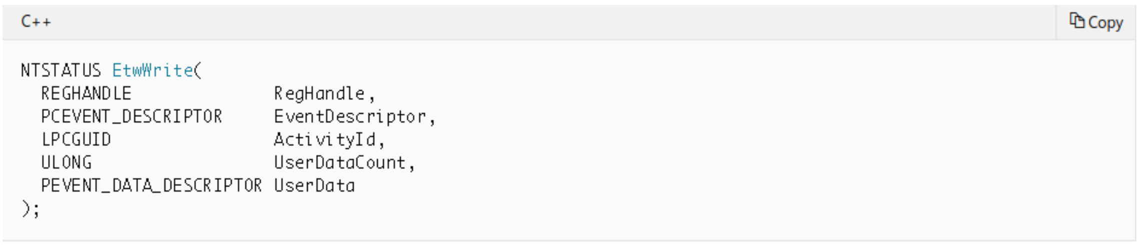
At that point, an interesting idea came up: To answer **Where are they running?** and **What are they logging?** it could be useful to hook into the exact moment when any of those providers are going to write to the DiagTrack's session. In other words, it could be useful to set a breakpoint in the function that performs the write to the DiagTrack's session. Once the breakpoint is hit, the following information could be extracted:

1. The piece of code that triggered the write (by inspecting the function's call stack). In other words, identify who was writing.
2. The actual content of the log being written.

That was when debugging (2.1.2) came into play. Analyzing the symbols exposed by the ntoskrnl binary, the function **EtwWrite** was found. This function seemed to be the one in charge of carrying out the process of writing inside sessions. Nonetheless, it wasn't worth to set a breakpoint at this function as every provider of the system (not necessarily related to DiagTrack) could use it. It was necessary to find a way of only detecting the writes performed by DiagTrack's providers.

The powershell command (4.24) returned information for each registered provider. Part of that information was the GUID. Due to the previous objective was to filter writes only from providers that were registered against the DiagTrack Session, it could be useful to set a conditional breakpoint inside function **EtwWrite** and try to check if the GUID provided was of interest.

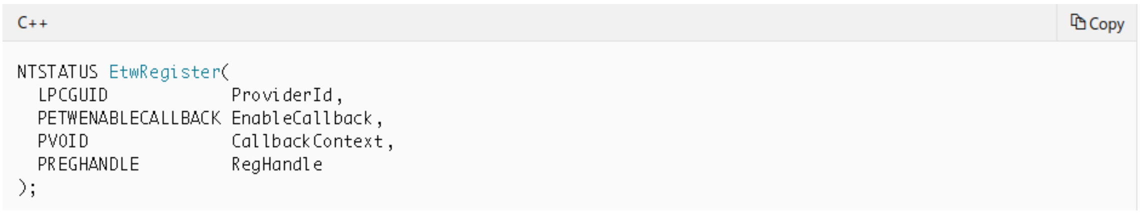
Unfortunately, this strategy had one minor issue. The function (**EtwWrite**) had five parameters and none of them would show directly the GUID:



```
NTSTATUS EtwWrite(
  REGHANDLE              RegHandle,
  PCEVENT_DESCRIPTOR     EventDescriptor,
  LPCGUID                ActivityId,
  ULONG                  UserDataCount,
  PEVENT_DATA_DESCRIPTOR UserData
);
```

Fig. 4.2: Documentation for EtwWrite function [1].

The first parameter was the registration handler. This object was returned once the provider executed the registration (**EtwRegister)** successfully. Taking a deeper look to **EtwRegister** it was possible to observe that it received the GUID as parameter:



```
NTSTATUS EtwRegister(
  LPCGUID            ProviderId,
  PETWENABLECALLBACK EnableCallback,
  PVOID              CallbackContext,
  PREGHANDLE         RegHandle
);
```

Fig. 4.3: Documentation for EtwRegister function [2].

This finding basically meant that having only one breakpoint in **EtwWrite** wasn't going to be enough as information from **EtwRegister** was also needed.In other words, to understand if the write was being done by a provider registered against the DiagTrack session it was necessary to:

1. Extract the whole list of providers registered against the DiagTrack session.
2. Intercept all the **EtwRegister** executions and check if the GUID being used was inside the list.

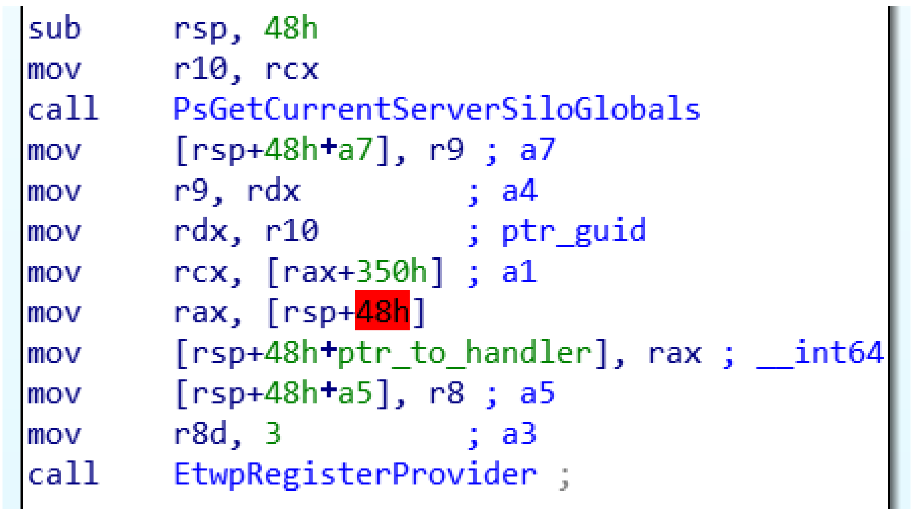
3. If it was, save the handler.
4. Intercept all the **EtwWrite** executions and check if the handler being used is one of the stored handlers.
5. If it was, the provider that is writing, is attached to the DiagTrack session.

Even though this strategy seemed to be theoretically promising, it was necessary to understand how to actually carry out each of these steps. Further sections will depict this process.

4.1.1 Reversing registration process

As mentioned in section 2.3.0.1, whenever a provider wants to register itself against a particular session it has to call the function **EtwRegister**. Because of this, the first step was to analyze the behavior of this function using **IDA**(2.2.1).

As can be seen in figure 4.4, the only interesting action being performed by **EtwRegister** was a call to another function named **EtwpRegisterProvider**.



```
sub     rsp, 48h
mov     r10, rcx
call    PsGetCurrentServerSiloGlobals
mov     [rsp+48h+a7], r9 ; a7
mov     r9, rdx          ; a4
mov     rdx, r10         ; ptr_guid
mov     rcx, [rax+350h] ; a1
mov     rax, [rsp+48h]
mov     [rsp+48h+ptr_to_handler], rax ; __int64
mov     [rsp+48h+a5], r8 ; a5
mov     r8d, 3           ; a3
call    EtwpRegisterProvider ;
```

Fig. 4.4: Dissasembly of ETWRegister function

A quick analysis of the latter function showed that it was the function holding the actual implementation of the registration process. However, due to the lack of documentation, it was necessary to understand in a more in-depth way what was actually happening inside it.

Following chapters will present a detailed description of reversing (partially, only the interesting parts for this research) **EtwpRegisterProvider**. To make it easier, it will be divided in different parts:

- 1. Understanding the layout of the function
- 2. Check if a GUID for this provider already exists.
- 3. If not, create a new one
- 4. Return the handler.

```
// 1. Understanding the layout of the function
signed __int64 __fastcall EtwpRegisterProvider(__int64 a1, _QWORD *a2, int a3,
    void (__fastcall *a4)(ULONG_PTR, __int64, __int128 *,__int64), __int64 a5,
    __int64 a6, __int64 *a7){

[..]

// 2. Check if a GUID for this provider already exists.
ptr_guid_entry = (char *)EtwpFindGuidEntryByGuid(etw_silo, ptr_guid, 0);

// 3. If not, create a new one
if ( ptr_guid_entry || (ptr_guid_entry = EtwpAddGuidEntry(ptr_etw_silo_cpy2,
    ptr_guid_cpy, 0)) != 0i64 )
{
  v15 = __readgsqword(0x188u);
  --*(_WORD *)(v15 + 484)

  [..]

  // 4. Return the handler.
  v35 = EtwpAddKmRegEntry((ULONG_PTR)ptr_guid_entry, v10, (__int64)v9, a5,
      (__int64)&ptr_handler);
  v20 = v35;

  [..]

}
return v20;
```

Fig. 4.5: EtwpRegisterProvider snippet .

4.1.1.1 1. Understanding the layout of the function

EtwpRegisterProvider received seven parameters:

```
signed __int64 __fastcall EtwpRegisterProvider(__int64 a1, _QWORD *a2,
       int a3, void (__fastcall *a4)(ULONG_PTR, __int64, __int128 *,
       __int64), __int64 a5, __int64 a6, __int64 *a7)
```

Usually when performing reverse engineering it is not necessary to understand every tiny detail but only the key points that are important to meet the proposed goals. This wasn't the exception.

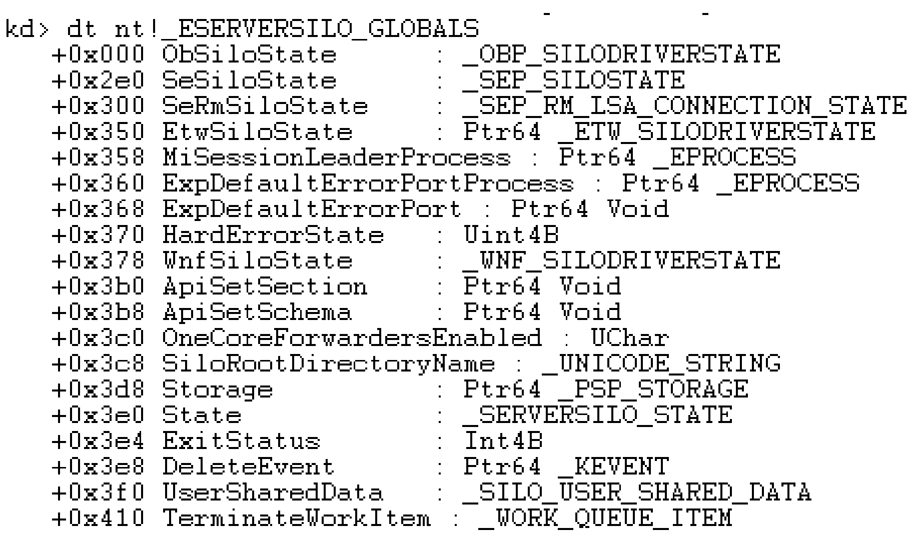
The main focus here was not to understand how the registration process fully worked but just to get an idea of it plus get to know the relation between GUID and registration handler.

After analyzing **EtwpRegisterProvider** it was possible to conclude that:

1. **a1**: Was the pointer to a structure.
2. **a2**: Was the pointer to the GUID structure.

3. **a7**: Was the address where the pointer to the registration handler would be placed (can be think as "function output").

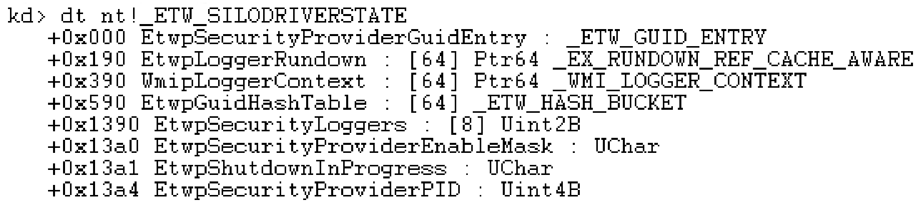
What is this **a1** structure? The figure 4.4 shows that before calling **EtwpRegister-Provider**, the function **PsGetCurrentServerSiloGlobals** is invoked. This latter one returns a pointer to a structure S of type **_ESERVERSILO_GLOBALS**.



```
kd> dt nt!_ESERVERSILO_GLOBALS
   +0x000 ObSiloState      : _OBP_SILODRIVERSTATE
   +0x2e0 SeSiloState      : _SEP_SILOSTATE
   +0x300 SeRmSiloState    : _SEP_RM_LSA_CONNECTION_STATE
   +0x350 EtwSiloState     : Ptr64 _ETW_SILODRIVERSTATE
   +0x358 MiSessionLeaderProcess : Ptr64 _EPROCESS
   +0x360 ExpDefaultErrorPortProcess : Ptr64 _EPROCESS
   +0x368 ExpDefaultErrorPort : Ptr64 Void
   +0x370 HardErrorState   : Uint4B
   +0x378 WnfSiloState     : _WNF_SILODRIVERSTATE
   +0x3b0 ApiSetSection    : Ptr64 Void
   +0x3b8 ApiSetSchema     : Ptr64 Void
   +0x3c0 OneCoreForwardersEnabled : UChar
   +0x3c8 SiloRootDirectoryName : _UNICODE_STRING
   +0x3d8 Storage          : Ptr64 _PSP_STORAGE
   +0x3e0 State            : _SERVERSILO_STATE
   +0x3e4 ExitStatus       : Int4B
   +0x3e8 DeleteEvent      : Ptr64 _KEVENT
   +0x3f0 UserSharedData   : _SILO_USER_SHARED_DATA
   +0x410 TerminateWorkItem : _WORK_QUEUE_ITEM
```

Fig. 4.6: _ESERVERSILO_GLOBALS structure layout (S).

However, the first parameter provided to **EtwpRegisterProvider** was not the pointer to S but the pointer to another structure S_2 of type **_ETW_SILODRIVERSTATE** which happens to be part of S, situated at offset 0x350.



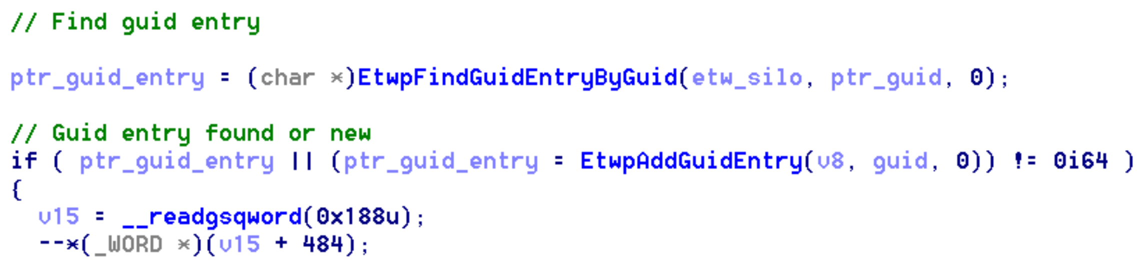
```
kd> dt nt!_ETW_SILODRIVERSTATE
   +0x000 EtwpSecurityProviderGuidEntry : _ETW_GUID_ENTRY
   +0x190 EtwpLoggerRundown : [64] Ptr64 _EX_RUNDOWN_REF_CACHE_AWARE
   +0x390 WmipLoggerContext : [64] Ptr64 _WMI_LOGGER_CONTEXT
   +0x590 EtwpGuidHashTable : [64] _ETW_HASH_BUCKET
   +0x1390 EtwpSecurityLoggers : [8] Uint2B
   +0x13a0 EtwpSecurityProviderEnableMask : UChar
   +0x13a1 EtwpShutdownInProgress : UChar
   +0x13a4 EtwpSecurityProviderPID : Uint4B
```

Fig. 4.7: _ETW_SILODRIVERSTATE structure layout (S_2).

With this information it was possible to conclude that **a1** will point to a global structure holding configurations, settings and information in general directly related with the **ETW** framework. **a2** and **a7** will hold pointers to a GUID and to a place were a registration handler will be stored afterwards.

With the information gathered from this analysis was enough to move forward.

Once inside the *EtwpRegisterProvider* function, after performing some sanity checks, it tries to get the guid entry related to the GUID provided. If it doesn't exist, it will create one.



```
// Find guid entry

ptr_guid_entry = (char *)EtwpFindGuidEntryByGuid(etw_silo, ptr_guid, 0);

// Guid entry found or new
if ( ptr_guid_entry || (ptr_guid_entry = EtwpAddGuidEntry(v8, guid, 0)) != 0i64 )
{
  v15 = __readgsqword(0x188u);
  --*(_WORD *)(v15 + 484);
```

Fig. 4.8: First lines of EtwpRegisterProvider

4.1.1.2 **2. Check if GUID for this provider already exists**

This part will be focused on understanding how the process of recovering the already existing ”GUID entry” works.

The action of recovering is performed by a particular function called **EtwpFindGuidEntryByGuid**:

```
ptr_guid_entry = (char *)EtwpFindGuidEntryByGuid(etw_silo, ptr_guid, 0);
```

As can be inferred from the previous line, two important parameters were provided: the **ETWSILODRIVERSTATE**(a1) structure S and the pointer to the GUID *ptr_guid*.

```
EtwpFindGuidEntryByGuid proc near

arg_8= qword ptr  10h
arg_10= qword ptr  18h

mov     [rsp+arg_8], rbx
mov     [rsp+arg_10], rbp
push    rsi
push    rdi
push    r12
push    r14
push    r15
sub     rsp, 20h
mov     eax, [rdx+8]
add     rcx, 590h
xor     eax, [rdx+0Ch]
xor     r12d, r12d
xor     eax, [rdx+4]
mov     rdi, rdx
xor     eax, [rdx]
mov     r14d, r12d
and     eax, 3Fh
movsxd  rsi, r8d
imul    rax, 38h
shl     rsi, 4
add     rcx, rax
mov     rax, gs:188h
add     rsi, rcx
dec     word ptr [rax+1E4h]
lea     rbp, [rcx+30h]
xor     r8d, r8d
```

Fig. 4.9: First basic block of EtwpFindGuidEntryByGuid.

Figure 4.9 depicts how the function gets the guid entry related to the provider (if it exists): rcx holds the pointer to S and rdx holds ptr_guid. Let's analyze this function deeper.

The first highlight is the *add* function which stores in rcx the pointer to the structure stored at the offset $0x590$ of S. Going back to the structure layout of S (figure 4.7), it can be appreciated that at the offset $0x590$, the structure $EtwpGuidHashTable$ of type $\_ETW\_HASH\_BUCKET[64]$ is present. Figure 4.11 depicts its layout.

Just before the *add* function, eax is filled with the content of the address $rdx + 8$. rdx held the ptr_guid, meaning that eax will have the third group of 4 bytes inside of the guid structure. Why the third? Because the offset was 8. Why 4 bytes? Because the register eax (32 bits) was used.

In the following lines, the value of eax is being constantly modified by xoring it successively with the different group of 4 bytes that compose the guid structure[3]. After performing these successive xor operations, a boolean-and is applied against eax (result of xoring) using a mask of $0x3f$. This mask will set all eax bits to 0 with the exception of the

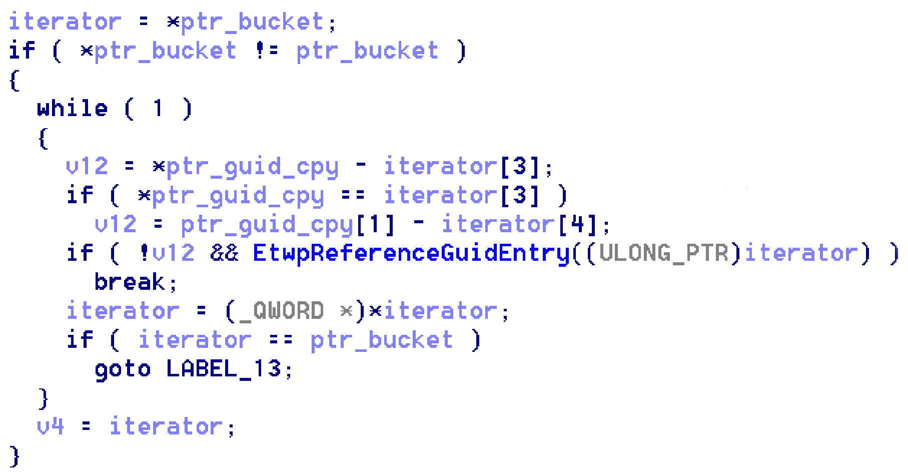
[3] Sometimes the structures weren't documented at all. Sometimes they were, but was not possible to find it until some kind of clue pointing to it was found. So far the layout of the structure pointed by ptr_guid was unknown, however from this function it was possible to conclude that the structure had a size of 16 bytes.

last 6 that will remain having its actual value. The reason to do this is because $2^6 = 64$. In other words, this mask is making the xoring result fit into the range of a valid bucket index. Afterwards, multiplies the result against $0x38$ (size of *_ETW_HASH_BUCKET* structure). Finally, the value of $rax(eax)$ is added to rcx which had the pointer to the *EtwpGuidHashTable* structure.

Writing the aforementioned function in a pseudo-code style (*ptr* is a short version of "pointer"):

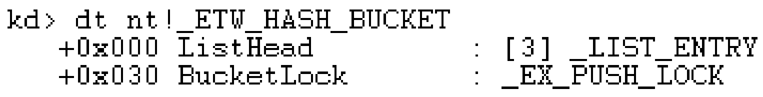
```
xor_guid_parts = ptr_guid[0] ^ ptr_guid[1] ^ ptr_guid[2] ^ ptr_guid[3]
ptr_hash_table = ptr_S + 0x590
ptr_bucket = ptr_hash_table + 0x38 * ((xor_guid_parts) & 0x3F)
```

Therefore, *ptr_bucket* is basically a pointer to a particular bucket inside the *EtwpGuidHashTable* calculated based on the GUID of the provider[4]. Once this value is obtained, a "look up" inside the structure is carried out in the following way:



```
iterator = *ptr_bucket;
if ( *ptr_bucket != ptr_bucket )
{
  while ( 1 )
  {
    v12 = *ptr_guid_cpy - iterator[3];
    if ( *ptr_guid_cpy == iterator[3] )
      v12 = ptr_guid_cpy[1] - iterator[4];
    if ( !v12 && EtwpReferenceGuidEntry((ULONG_PTR)iterator) )
      break;
    iterator = (_QWORD *)*iterator;
    if ( iterator == ptr_bucket )
      goto LABEL_13;
  }
  v4 = iterator;
}
```

Fig. 4.10: Part of EtwpFindGuidEntryByGuid function extracted using IDA Hex-Rays plugin.



```
kd> dt nt!_ETW_HASH_BUCKET
   +0x000 ListHead         : [3] _LIST_ENTRY
   +0x030 BucketLock       : _EX_PUSH_LOCK
```

Fig. 4.11: _ETW_HASH_BUCKET structure layout.

At first an iterator is built. This iterator will point initially to the Flink of the first list entry[5] of the bucket (figure 4.11). The right-after "if" will capture the special case were the list is empty. In that particular case, the whole cycle will be skipped and the **LABEL_13** (routine to exit, which isn't displayed in the figure) will be executed. It's worth to mention that this routine executes a return statement with the value of the variable $v4$ (which is initially defined as 0).

[4] There was also an additional value involved in the calculation of the bucket. However, in this particular context, the value wasn't taken into account as it was always 0.

[5] https://docs.microsoft.com/en-us/windows/desktop/api/ntdef/ns-ntdef-_list_entry

If the list is not empty, the first operation which is carried out is a subtraction between the first quadword of the GUID and a value of $iterator[3]$. Due to the variable $iterator$ is defined as a 8-bytes pointer, $iterator[3]$ will point to the offset $0x18$ of the structure stored inside the Flink. In the case that both values are equal, the second comparison (between the second quadword of the GUID and the $iteartor[4]$) is carried out.

At this point some things can be concluded:

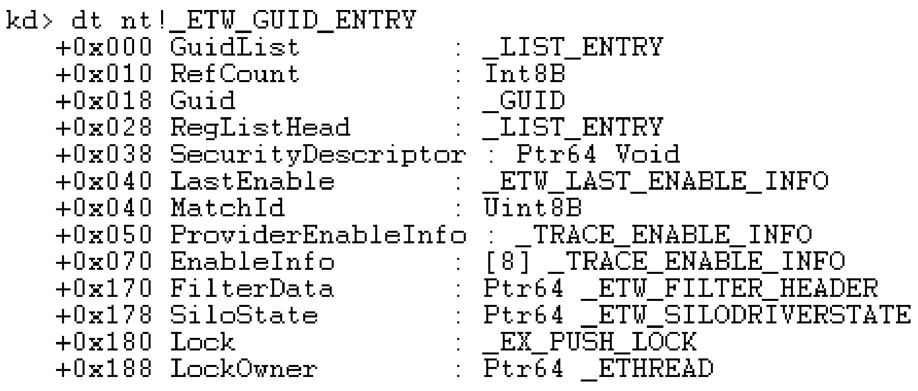
- The cycle is iterating a double-linked-list which holds a particular structure T.
- T has the GUID of the provider stored at offset $0x18$
- Again, seems that the GUID is 16 bytes long.
- From the function name it can concluded that T is a structure that represents the GUID entry.

Moving forward with the code analysis, if some of the comparisons failed, the iterator changes its values to the next one in the list. Before continuing, it ensures that the cycle is not finished by checking if the actual value of the iterator is the same one used as the starting point. If they are equal, the exit routine is executed meaning that the return value will be 0.

If both comparisons are equal (the GUID of the provider and the one stored in T are the same), a function called $EtwpReferenceGuidEntry$ with the current value of the iterator as parameter, is called. After this execution, the cycle is finished by the break statement. However, before executing the exit routine, the value of $v4$ is filled up with the value of the $iterator$, meaning that the return value will be pointer to the guid entry related to the GUID of the provider. The $EtwpReferenceGuidEntry$ function just made some security checks not relevant for this task.

Therefore, to summarize, it is possible to say that:
The function $EtwpFindGuidEntryByGuid$ looks for a particular structure (most probably called guid entry), which is stored inside a double-linked-list of a bucket inside the $EtwpGuidHashTable$ of the $\_ETW\_SILODRIVERSTATE$, based on doing some mathematical operations with the GUID of the provider.

After finishing with this analysis, the documentation of the guid entry structure was found:



```
kd> dt nt!_ETW_GUID_ENTRY
   +0x000 GuidList           : _LIST_ENTRY
   +0x010 RefCount           : Int8B
   +0x018 Guid               : _GUID
   +0x028 RegListHead        : _LIST_ENTRY
   +0x038 SecurityDescriptor : Ptr64 Void
   +0x040 LastEnable         : _ETW_LAST_ENABLE_INFO
   +0x040 MatchId            : Uint8B
   +0x050 ProviderEnableInfo : _TRACE_ENABLE_INFO
   +0x070 EnableInfo         : [8] _TRACE_ENABLE_INFO
   +0x170 FilterData         : Ptr64 _ETW_FILTER_HEADER
   +0x178 SiloState          : Ptr64 _ETW_SILODRIVERSTATE
   +0x180 Lock               : _EX_PUSH_LOCK
   +0x188 LockOwner          : Ptr64 _ETHREAD
```

Fig. 4.12: $\_ETW\_GUID\_ENTRY$ structure.

```
kd> dt nt!_GUID
   +0x000 Data1            : Uint4B
   +0x004 Data2            : Uint2B
   +0x006 Data3            : Uint2B
   +0x008 Data4            : [8] UChar
```

Fig. 4.13: _ETW_GUID structure.

Luckily, all the previous guesses made, were correct:

- The guid entry (now *_ETW_GUID_ENTRY*) had the GUID of at offset $0x18$ (figure 4.12)
- The GUID was a structure of 16 bytes long (figure 4.13)

4.1.1.3 **3. If not found, create a new one**

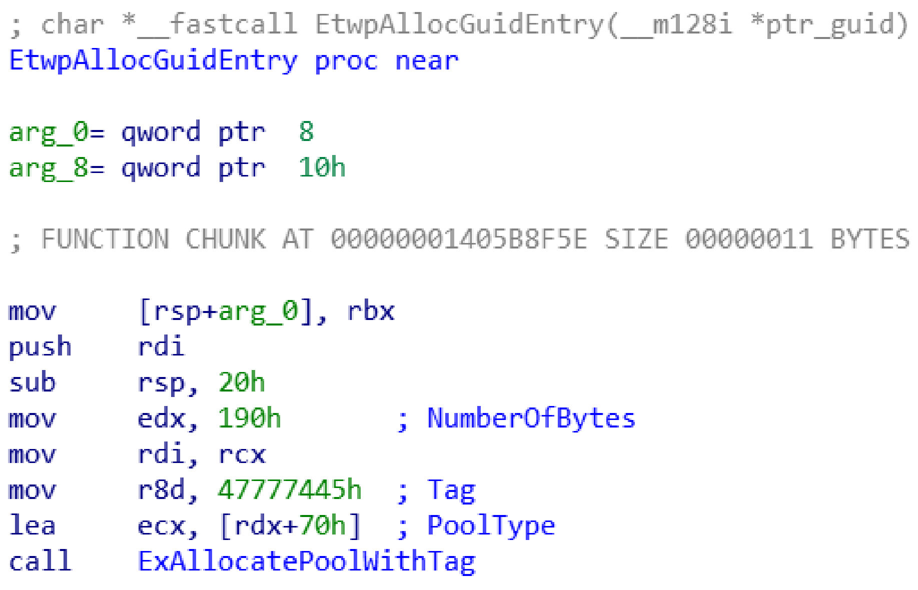
Previous section detailed how was the process to find an already existing guid entry based on the GUID of the provider. This one will explain the process of creating a new guid entry.

From figure 4.8 can be observed that the function in charge of this part is the function *EtwpAddGuidEntry*:

```
ptr_guid_entry = EtwpAddGuidEntry(ptr_etw_silo_cpy2, ptr_guid_cpy, 0)
```

As can be inferred from the previous line, two important parameters were provided: the pointer to the _ETW_SILODRIVERSTATE structure (for simplicity will be called *ptr_etw_silo* instead of *ptr_etw_silo_cpy*2) and the pointer to the GUID (for simplicity will be called *ptr_guid* instead of *ptr_guid_cpy*).

One of the first lines of *EtwpAddGuidEntry*, calls another function named *EtwpAllocGuidEntry*. As it can be quickly inferred from the name, it basically allocates certain amount of memory inside the heap to be used by the guid entry afterwards and returns the pointer to it. The allocation part happens in the first basic block of *EtwpAllocGuidEntry*:



```
; char *__fastcall EtwpAllocGuidEntry(__m128i *ptr_guid)
EtwpAllocGuidEntry proc near

arg_0= qword ptr  8
arg_8= qword ptr  10h

; FUNCTION CHUNK AT 00000001405B8F5E SIZE 00000011 BYTES

mov     [rsp+arg_0], rbx
push    rdi
sub     rsp, 20h
mov     edx, 190h       ; NumberOfBytes
mov     rdi, rcx
mov     r8d, 47777445h  ; Tag
lea     ecx, [rdx+70h]  ; PoolType
call    ExAllocatePoolWithTag
```

Fig. 4.14: EtwpAllocGuidEntry allocation.

As can be observed in figure 4.14 the Windows function *ExAllocatePoolWithTag*[6] is called with the following parameters:

- **PoolType**: 0x200 (**NonPagedPoolNx**). This value indicates that the system memory allocated will be nonpageable and not executable[7].
- **NumberOfBytes**: 0x190. This value is the size of the structure *_ETW_GUID_ENTRY* (figure 4.12).
- **Tag**: "0x47777445". According the documentation just a four character long to be used as the pool tag. Due to it is specified in reverse order: 0x45747747 → "EtwG".

Therefore, as it was thought, *EtwpAllocGuidEntry* allocs the necessary memory for holding the *_ETW_GUID_ENTRY* structure and returns a heap pointer to it.

The remaining code of *EtwpAddGuidEntry* is devoted to populate and adjust some parts of related structures. Some key points about it:

- A guid entry related to this GUID is looked up inside the guid entries double-linked list using the same technique as the one used in *EtwpFindGuidEntryByGuid*. If a structure is found, the pointer is freed.
- Only three parts of the *_ETW_GUID_ENTRY* structure are populated at this point:
  1. The pointer to the previous guid entry in the double-linked list (offset 0x0)
  2. The pointer to the following guid entry in the double-linked list (offset 0x8)
  3. The pointer to the SILO STATE (offset 0x178)

To summarize, once *EtwpAllocGuidEntry* is executed, the pointer to heap memory holding the *_ETW_GUID_ENTRY* structure is returned. The next step is insert this entry into *EtwpGuidHashTable*. To perform that action, first it looks for the correct place to insert it as depicted previously.

4.1.1.4 **4. Return the handler**

Going back to what the figure 4.3 states, the 4th parameter of *EtwRegister* it's something of type *PREGHANDLE*. Although it isn't very clear, this parameter is the output of the function (usually called "out" type of parameter). Furthermore, as it was mentioned previously the real registration logic is implemented by *EtwpRegisterProvider* therefore the output of *EtwRegister* is none other than the output of *EtwpRegisterProvider*.

Despite the fact the provider's GUID existed previously or not, at this point it exist a pointer to a *_ETW_GUID_ENTRY* structure holding its data and already inside the main structures of ETW. Once the code achieved this point, the next step is basically get the handler.

Just right after the pointer to the *_ETW_GUID_ENTRY* is found, the function *EtwpAddKmRegEntry* is called:

[6] Documentation: https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/wdm/nf-wdm-exallocatepoolwithtag

[7] Documentation: https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/wdm/ne-wdm-_pool_type

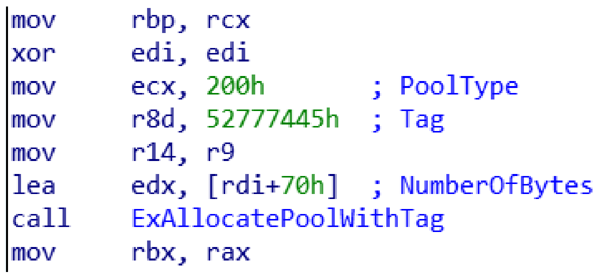
`__int64 __usercall EtwpAddKmRegEntry(ULONG_PTR a1, int a2, __int64 a3, __int64 a4, __int64 a5)`

where :

1. **a1**: Is the pointer to *_ETW_GUID_ENTRY*, called *ptr_guid*.
2. **a5**: Is the memory address provided by *EtwWrite* (and afterwards by *EtwpRegisterProvider*) where the handler should be placed.

The remaining parameters are not interesting for the sake of our research.

Once inside *EtwpAddKmRegEntry*, the first important lines were:



```
mov     rbp, rcx
xor     edi, edi
mov     ecx, 200h       ; PoolType
mov     r8d, 52777445h  ; Tag
mov     r14, r9
lea     edx, [rdi+70h]  ; NumberOfBytes
call    ExAllocatePoolWithTag
mov     rbx, rax
```

Fig. 4.15: EtwpAddKmRegEntry allocation.

As can be inferred from 4.14, this is also an allocation in the heap:

- **PoolType**: 0x200 (**NonPagedPoolNx**).
- **NumberOfBytes**: 0x70. As depicted in the image, *rdi* is first set to 0.
- **Tag**: "0x52777445". According the documentation just a four character long to be used as the pool tag. Due to it is specified in reverse order: 0x45747752 → "EtwR".

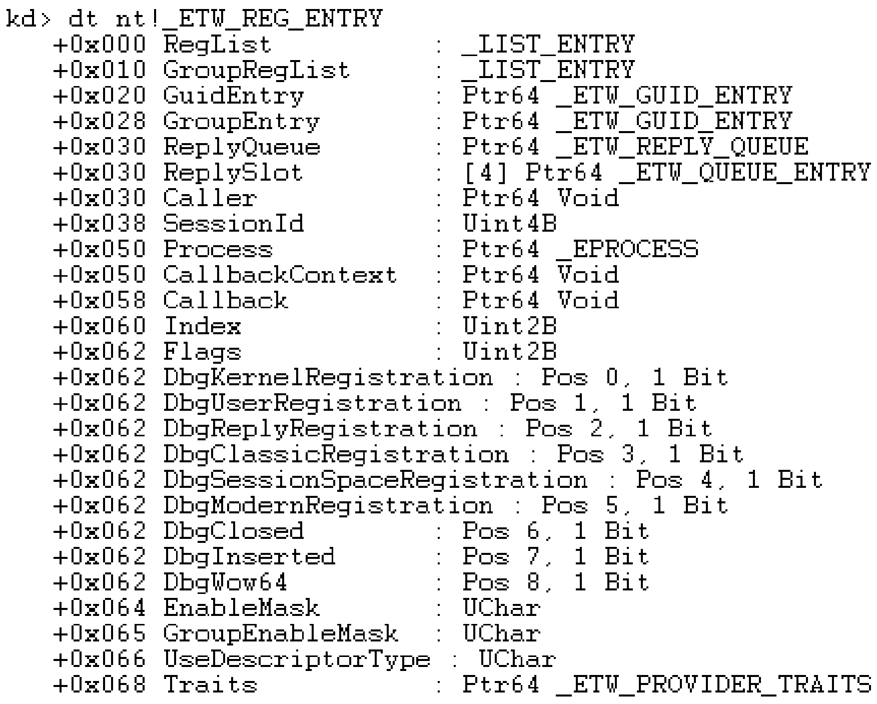
The analysis of the following lines of *EtwpAddKmRegEntry* showed how the aforementioned reserved space (structure) was being filled. While debugging this function the following line excelled from the rest:

```
mov     [rbx+20h], rbp
```

The reason to excelled was that, at that point, *rbp* held the pointer to the GUID entry. Meaning that this structure, potentially the registration handler structure, has the pointer to the GUID entry at offset 0x20. After finish filling the rest, the pointer to the structure is returned.

Going back to 4.5, it can be appreciated that the output of *EtwpAddKmRegEntry* is the output of *EtwpRegisterProvider* too. Which confirmed that this was the registration handler structure.

After finishing with this analysis, the documentation of the registration handler structure was found:



```
kd> dt nt!_ETW_REG_ENTRY
   +0x000 RegList          : _LIST_ENTRY
   +0x010 GroupRegList     : _LIST_ENTRY
   +0x020 GuidEntry        : Ptr64 _ETW_GUID_ENTRY
   +0x028 GroupEntry       : Ptr64 _ETW_GUID_ENTRY
   +0x030 ReplyQueue       : Ptr64 _ETW_REPLY_QUEUE
   +0x030 ReplySlot        : [4] Ptr64 _ETW_QUEUE_ENTRY
   +0x030 Caller           : Ptr64 Void
   +0x038 SessionId        : Uint4B
   +0x050 Process          : Ptr64 _EPROCESS
   +0x050 CallbackContext  : Ptr64 Void
   +0x058 Callback         : Ptr64 Void
   +0x060 Index            : Uint2B
   +0x062 Flags            : Uint2B
   +0x062 DbgKernelRegistration : Pos 0, 1 Bit
   +0x062 DbgUserRegistration : Pos 1, 1 Bit
   +0x062 DbgReplyRegistration : Pos 2, 1 Bit
   +0x062 DbgClassicRegistration : Pos 3, 1 Bit
   +0x062 DbgSessionSpaceRegistration : Pos 4, 1 Bit
   +0x062 DbgModernRegistration : Pos 5, 1 Bit
   +0x062 DbgClosed        : Pos 6, 1 Bit
   +0x062 DbgInserted      : Pos 7, 1 Bit
   +0x062 DbgWow64         : Pos 8, 1 Bit
   +0x064 EnableMask       : UChar
   +0x065 GroupEnableMask  : UChar
   +0x066 UseDescriptorType : UChar
   +0x068 Traits           : Ptr64 _ETW_PROVIDER_TRAITS
```

Fig. 4.16: _ETW_REG_ENTRY structure .

4.2 Hooking into providers' writes

In section 4.1 an idea on how to answer two important questions was presented: Hook in the exact moment when providers write to DiagTrack's session. However the idea would have been unfeasible without the analysis performed in section 4.1.1.

At this point it was possible to detect if a particular provider was writing by knowing just its GUID. However it was important to understand to which session this provider was writing.

For an initial analysis a mix between automatic and manual analysis was performed. A breakpoint in the function **EtwWrite** was set using the following Windbg(2.2.2) script:

```
bp nt!EtwWrite ".printf \"Handler: %N\\n\",@rcx"
```

This script just printed the address of provider's registration handle that is performing the write (@*rcx* holds the first parameter of the function according to Windows x64 calling convention). Once the handler's address was obtained it was possible to get the GUID as well. Comparing the GUID with the output of the powershell command **Get-EtwTraceProvider** (without filters) threw an interesting result: Most of the time a provider with E02A841C-75A3-4FA7-AFC8-AE09CF9B7F23 as GUID was the one writing. Unfortunately, this provider wasn't related to DiagTrack at all.

With the goal of filtering out writes carried out by this provider, the following script was used:

```
bp nt!EtwWrite ".if (@rcx != ffffda839f0c2c50){.printf \"Handler:
    %N\\n\",@rcx;gc;}.else{gc;} "
```

However, this wasn't enough. Providers which weren't related to DiagTrack continued flooding the output of the script. Clearly, this wasn't a good approach.

With the objective of pursuing interesting results, another ETW-related writing function called **EtwWriteTransfer** was used :

```
bp nt!EtwWriteTransfer ".if (@rcx != ffffda839f0c2c50){.printf \"Handler:
    %N\\n\",@rcx;gc;}.else{gc;}"
```

This time, a new provider attached to DiagTrack with GUID E9EAF418-0C07-464C-AD14-A7F353349A00 appeared. To get some extra information, the script was updated once more to get the Call Stack of the provider's process:

```
bp nt!EtwWriteTransfer ".if (@rcx == FFFFDA83A036F0D0){.printf \"Handler:
    %N\\n\",@rcx;kc;gc;}.else{gc;} "

 Call Site
00 nt!EtwWriteTransfer
01 nt!TlgWrite
02 nt!CmpInitHiveFromFile
03 nt!CmpCmdHiveOpen
04 nt!CmLoadAppKey
05 nt!CmLoadDifferencingKey
06 nt!NtLoadKeyEx
07 nt!KiSystemServiceCopyEnd
08 ntdll!NtLoadKeyEx
09 0x0
```

So far, the only way to detect when providers related to DiagTrack were writing consisted in two steps:

1. Hook in the *EtwWrite* function call.
2. Check if the provider is attached to DiagTrack based on its GUID and the output of the powershell command.

However, even if it was a DiagTrack's related provider the one writing, it was not enough to ensure that it is currently writing to DiagTrack's session (providers could be attached to several sessions). Furthermore, *EtwWriteTransfer* showed that *EtwWrite* wasn't the only function involved in the writing process. In other words, the following two questions were raised:

1. Is **EtwWrite** the only write function used? (clearly no!)
2. How can we sure that a provider is actually writing to DiagTrack's session

4.2.1 ETW functions to write

A quick analysis of symbols and cross references of **ntoskrnl.exe** binary showed that there were actually a "group of functions" that could be used to perform writes within the ETW framework:

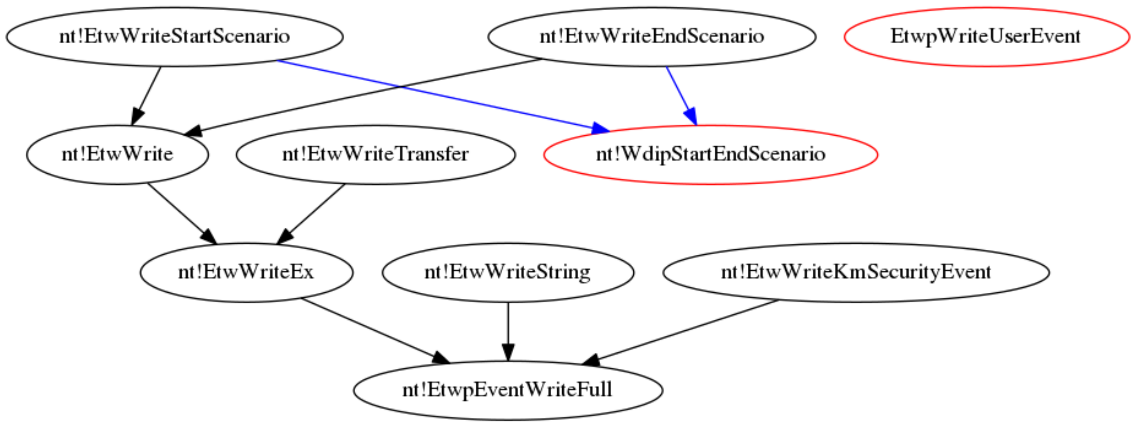


Fig. 4.17: Group of ETW Write functions.

As can be appreciated in figure 4.17 all functions end up calling either **nt!EtwpEventWriteFull**, **nt!EtwpWriteUserEvent** or **nt!WdispStartEndScenario**. After extensive analysis logging all actions and functions that were executed, it was possible to conclude that the function **nt!WdispStartEndScenario** was never called under the context of interest of this research. This meant, that the analysis of the writing phase could be carried out focusing only in the first two functions: **nt!EtwpEventWriteFull** and **nt!EtwpWriteUserEvent**.

4.2.2 Ensuring providers write to DiagTrack session

The previous idea of only analyzing writes from providers that were returned by the powershell command had, at least, three problems:

1. The output of the powershell command returned providers that were registered in a particular session at a given time t. If a provider registers, writes and unregisters itself from the session in a time frame tf where $t \notin tf$, that write won't be took into consideration.
2. Even if the provider is registered against DiagTrack's session, this doesn't ensure the provider is currently writing to it.
3. It relied too much on manual analysis.

With these problems in mind new ideas began to appear. In particular there was one that made the difference: What if it's possible to relate the handler that it's used at the moment of writing, with the session where the provider going to write. If that's possible, the first two problems would be solved. The third one could be solved as only one breakpoint at the writing function will be enough to make the full analysis.

Following sections will detail the process to find an answer to these problems.

4.2.2.1 **Inspecting ETW structures**

So far it happened a lot that ETW structures were the key to overcome different obstacles. We wanted to know if this was again the case.

The first structure analyzed was *_ETW_REG_ENTRY* (figure 4.16) as it would have been the most direct and easier way to relate session and provider. Unfortunately, none of its components were helpful.

The next structure analyzed was *_WMI_LOGGER_CONTEXT*. This structure seemed to be the actual representation of an ETW session. Due to its size, only necessary and representative offsets are depicted:

```
  +0x000 LoggerId       : Uint4B
  +0x004 BufferSize     : Uint4B
  +0x008 MaximumEventSize : Uint4B
  +0x00c LoggerMode     : Uint4B
  +0x010 AcceptNewEvents : Int4B
  [...]
  +0x070 ProviderBinaryList : _LIST_ENTRY
  +0x080 BatchedBufferList : Ptr64 _WMI_BUFFER_HEADER
  +0x080 CurrentBuffer  : _EX_FAST_REF
  +0x088 LoggerName     : _UNICODE_STRING
  +0x098 LogFileName    : _UNICODE_STRING
  +0x0a8 LogFilePattern : _UNICODE_STRING
  +0x0b8 NewLogFileName : _UNICODE_STRING
  [...]
  +0x428 LastBufferSwitchTime : _LARGE_INTEGER
  +0x430 BufferWriteDuration : _LARGE_INTEGER
  +0x438 BufferCompressDuration : _LARGE_INTEGER
```

Fig. 4.18: _WMI_LOGGER_CONTEXT structure.

Each ETW session will have an instance of this structure. At offset $0x70$ there is an attribute called **ProviderBinaryList**. After a quick analysis this attribute seemed to held all providers registered against the session in the format of a double linked list. In order to confirm that theory, the process of attaching new providers to an existing session was analyzed.

The function *nt!EtwpAddProviderToSession* seemed to be the one creating these links. During the analysis several problems were faced: lot of unknown new functions, hard to reverse, no documentation at all, among others. Besides all mentioned issues, there was a key reason that made the previous analysis to be called off: the structure *_TRACE_ENABLE_INFO*.

4.2.2.2 Provider GUID and Provider Group GUID

Usually an structure can have the answer, but if you cannot understand how that structure is being used you won't be able to see that answer. With that in mind, a further analysis over the ETW writes functions was carried out.

EtwWriteEx was one of the most used functions by providers to write inside events inside sessions. That was the reason why the analysis was focused on it.

The provider and the actual event that wanted to be logged were some of the parameters that *ETWWriteEx* received. An interesting piece of its source code can be depicted easily as a pseudocode, with the following listing:

```
1     // First part
2     v14 = *(_BYTE *)(ptr_handler + 0x64);
3     if(v14){
4       v15 = *(_QWORD **)(ptr_handler + 0x20);
5       if ( EtwpLevelKeywordEnabled((__int64)(v15 + 0x50), a2, a3) ){
6         EtwpEventWriteFull(...)
7       }
8     }
9     // Second part
10    v14 = *(_BYTE *)(ptr_handler + 0x65);
11    if (v14)}
12      v15 = *(_QWORD **)(ptr_handler + 0x28);
13      if ( EtwpLevelKeywordEnabled((__int64)(v15 + 0x50), a2, a3) ){
14        EtwpEventWriteFull(...)
15      }
16    }
```

Fig. 4.19: ETWWriteEx pseudocode

At first, both parts (line 1-7 and 8-14) may seem equal, but they have two key differences:

- The first one is filling **v14** using the offset $0x64$, while the second $0x65$
- The first one is filling **v15** using the offset $0x20$, while the second $0x28$

Going back to figure 4.16, it is possible to understand that the first part is using the attributes *GuidEntry* and $EnableMask$ while the second one is using *GroupGuidEntry* and $GroupEnableMask$. This was the key to understand that providers don't always use the same GUID entry, but they can also use a "Group GUID".

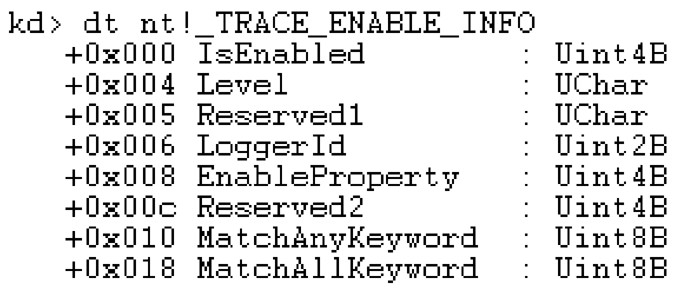
Furthermore, it was found that the function *EtwpLevelKeywordEnabled* received three parameters:

- $ProviderEnableInfo$ of type $\_TRACE\_ENABLE\_INFO$
- Event Level
- Event keyword

The main objective of *EtwpLevelKeywordEnabled* was to check if the event should be logged by the provider into the session according to the filtering rules defined when the provider was first registered against it. For more information, please refer to https://docs.microsoft.com/en-us/message-analyzer/system-etw-provider-event-keyword-level-settings.

In other words, listing 4.19 could be translated to "If $EnableMask$ is different from 0x0, check if the event should be logged using the $GuidEntry$. If it should, call $EtwpEventWriteFull$. If $GroupEnableMask$ is different from 0x0, check if the event should be logged using the $GroupEntry$. If it should, call $EtwpEventWriteFull$.

The first parameter received in *EtwpLevelKeywordEnabled* was of type $\_ENABLE\_TRACE\_INFO$. Analyzing its structure:



```
kd> dt nt!_TRACE_ENABLE_INFO
   +0x000 IsEnabled        : Uint4B
   +0x004 Level            : UChar
   +0x005 Reserved1        : UChar
   +0x006 LoggerId         : Uint2B
   +0x008 EnableProperty   : Uint4B
   +0x00c Reserved2        : Uint4B
   +0x010 MatchAnyKeyword  : Uint8B
   +0x018 MatchAllKeyword  : Uint8B
```

Fig. 4.20: Trace Enable Info structure.

This structure seemed to be promising as it had information that could relate to a session. Furthermore, this structure can be found inside *_ETW_GUID_ENTRY* at offset $0x50$ and $0x70$. However, the logic which filled or used this structure wasn't present inside *ETWWriteEx*. As shown in 4.19 after some checks, the function *EtwpEventWriteFull* was being called. In order to continue with this analysis, it was necessary to analyze this last function.

4.2.2.3 Identifying the destination session

The analysis of *EtwpEventWriteFull* was the hardest yet the most interest one. Although it received 17 parameters, had more that 1000 lines of code and helped a lot to understand several things, only few lines are going to be analyzed and illustrated here.

It turned out, as it was supposed, that the logic to understand to which session the execution was going to write was inside this function. The goal of this section is to explain with deep details how this process of choosing the session is carried out.

To begin with the analysis, let's start analyzing this simplified pseudocode which represents the key part of the process:

```
1    [...]
2    while ( 1 ){
3      bsf_found = !_BitScanForward((unsigned int *)&enable_info_bucket_index,
           enable_mask);
4      guid_ptr_shifted_by_enable_info_bucket_index = (__int64)&ptr_guid_entry[4
           * (unsigned int)enable_info_bucket_index]
5
6      if ( _bittest64(&ptr_local_addr, *(unsigned __int8
           *)(guid_ptr_shifted_by_enable_info_bucket_index + 0x76)) )
7        continue;
8
9      [...]
```

Fig. 4.21: Pseudocode snippet of *EtwpEventWriteFull* in charge of selecting the ETW session to write.

The **1st** line represents previous definitions and statements that are not important for the current analysis.
The **2nd** line represents the cycle that will be executed. In each iteration, it will pick one of the sessions that the provider is registered against and check if it should write this

event to that session. To perform such thing, it was found that the attribute $enableMask$ it's an 8-bit mask which represented the use of each of the 8 buckets from the $enableInfo$ attribute (*GuidEntry* structure). If the first bit (position 0) of $enableMask$ was set to 1, it meant that the information inside $enableInfo[0]$ should be considered. If its value was 0, the information of $enableInfo[0]$ should be dismissed. This was exactly what was happening in the **3rd** line. It performed a search[8] over the **enable_mask** variable which held the actual content of the $enableMask$ attribute from the $GuidRegEntry$, and wrote inside $enable\_info\_bucket\_index$ the index of the first bit set to 1.
The **4th** line shifted a temporary pointer to the $GuidEntry$ by $0x20$ * $enable\_info\_bucket\_index$. In this case, $0x20$ was the size of the structure $\_ENABLE\_TRACE\_INFO$ (figure 4.20), which was the type of each $enableInfo$ bucket. In other words, it was shifting the pointer depending on which bucket of $enableInfo$ should be considered in this iteration of the cycle.
The **6th** line was the one that gave sense to all the aforementioned steps. It added $0x76$ to the shifted pointer and checked[9] if the bit at the specified position was 1 or 0. This was basically checking if the $loggerId$ of the corresponding $enableInfo$ had some value. Let's do an example to better understand this:
If the result of the $enable\_info\_bucket\_index$ was equal to 0, it meant that the least significant bit of $enableMask$ had a 1. As explained before, this meant that the $enableInfo[0]$ should be considered. Therefore, the following step would be to shift the pointer to $GuidEntry$ by $0x20$ * 0, in other words, don't shift it. Finally, it checked if the position $0x76$ had something different from 0. $0x76 = 0x70 + 0x6$, $0x70$ was the offset of the first bucket of $enableInfo$ attribute from the $GuidEntry$, while $0x6$ was the offset of $loggerId$ inside the $\_ENABLE\_INFO\_TRACE$ structure. Now, if the $enable\_info\_bucket\_index$ was 1 instead of 0, the only difference from the previous example would have been that the pointer to the $guidEntry$ would be shifted $0x20 * 1 = 0x20$. This meant that the position to check with $_bittest64$ would have been $0x76 + 0x20$. Due to $0x20$ was the size of $\_ENABLE\_INFO\_TRACE$ structure it would have accessed the next bucket's $loggerId$ of $enableInfo$ ($enableInfo[1]$)
If the **6th** line condition was accomplished, meant that $loggerId$ was 0 and therefore not interesting. Hence, it would have jumped directly to the next iteration of the cycle. If the condition wasn't met, it would have continued with the rest of the statements.
The **9th** line represented all operations that were executed once a valid session of the provider was found (like checking if this event should be written to that session, the actual write to the session, etc).

The $loggerId$ finally became the way to relate a write execution with the destination session. Recalling the structure which represented an ETW session ($\_WMI\_LOGGER\_CONTEXT$, figure 4.18) the first of its attributes was the $loggerId$, an identification for the session. Although, until this point it was unknown the exact moment when the event was written, this logic was enough to understand how the session was chosen.

[8] https://docs.microsoft.com/en-us/cpp/intrinsics/bitscanforward-bitscanforward64?view=vs-2019
[9] https://docs.microsoft.com/en-us/cpp/intrinsics/bittest-bittest64?view=vs-2019

4.2.3 Detecting what is going to be written

Although the last analysis was key to understand how the destination session was chosen, it was necessary to find the exact moment when *EtwpAllocGuidEntry* was actually writing the event to that session. After all, that was the main goal.

As shown in listing 4.21, **9th** line involved a lot of further processing once a candidate session was found. Among all that processing, it was possible to find the following lines:

```
 1    [...]
 2    logger_id = *(unsigned __int16
          *)(guid_ptr_shifted_by_enable_info_bucket_index+0x76);
 3    [...]
 4    if ( (unsigned int)logger_id >= 0x40 ){
 5        ptr_wmi_trace_of_session_cpy = 1i64;
 6        ptr_wmi_trace_of_session = 1i64;
 7    }
 8    else {
 9        ptr_wmi_trace_of_session = *(_QWORD *)(ptr_silo_state + 8 * logger_id +
              0x390);
10        ptr_wmi_trace_of_session_cpy = ptr_wmi_trace_of_session;
11    }
12    [...]
13    ptr_to_buffer = EtwpReserveTraceBuffer(
14                        (unsigned int *)ptr_wmi_trace_of_session_cpy,
15                        event_size,
16                        (__int64)&ptr_to_buffer_offset,
17                        &return_value_of_function_to_get_cpu_clock,
18                        0
19                    );
20    if ( ptr_to_buffer ){
21      *(_OWORD *)(ptr_to_buffer + 0x18) = *(_OWORD *)(ptr_guid_entry_cpy + 3);
22      *(_OWORD *)(ptr_to_buffer + 0x28) = *(_OWORD *)ptr_event_descriptor_cpy;
23      [...]
24    }
25    [...]
```

Fig. 4.22: Pseudocode snippet of *EtwpEventWriteFull* in charge of writing event to the selected ETW session.

The **1st** line represented previous definitions and statements (like the whole listing 4.21) that were not important for the current analysis.
The **2nd** line assigned the *loggerId* value of the selected session to a variable. For the sake of clarity, this variable was called *logger_id*.
The **3rd** line represented definitions and statements that were not important for this analysis.
The **4th** line asked if *logger_id* $>= 0x40$. In case it did, it assigned weird values to a supposed pointer. The reason to do it, was because the maximum number of ETW sessions that could exists was 64 ($0x40$). In other words, it was ensuring that *logger_id* had a valid value.
In case *logger_id* was valid, lines **9** and **10** were executed. The former one, assigned the value of *ptr_silo_state* (variable defined before but whose content could be inferred from

its name) $+8 * logger\_id + 0x390$. What did all these values mean? At offset $0x390$ of the structure $\_ETW\_SILODRIVERSTATE$ (figure 4.7) there was an attribute called $WmipLoggerContext$. This attribute was an array of pointers ($Ptr64$) to structures of type $\_WMI\_LOGGER\_CONTEXT$ (figure 4.18). This array had 64 buckets (again, because of the maximum number of sessions possible) and its index in this array matcheed with their $loggerId$ value. Definitely, this was not coincidence. When adding a new ETW session, the free bucket of this $WmipLoggerContext$ is used and therefore its index within this array is used to fulfill the value of $loggerId$. Finally, 8 was the length of a $Ptr64$ structure.
As a conclusion, the value of $ptr\_wmi\_trace\_of\_session$ as the name suggests, will be the pointer to the corresponding $\_WMI\_LOGGER\_CONTEXT$ of the session associated with the $logger\_id$ found.
The **12th** line represented definitions and statements that were not important for this analysis.
The **13th** line represented a call to the most interesting function of this analysis:
$EtwpReserveTraceBuffer$. This function received parameters such as the pointer to the associated session's $\_WMI\_LOGGER\_CONTEXT$ and the size of the event to write. This function returned a pointer to the place (within the buffers of the ETW session) where the event data (and metadata) should be written afterwards (full analysis on the following section).
The **20th** line only checked if the pointer was different from null (in case the OS ran out of memory).
The **21th**, **22th** and **23th** lines represent the exact moment when all the data is being written inside the memory block returned by $EtwpReserveTraceBuffer$.
The **25th** represented statements that would have been executed in case the condition line **20** wasn't accomplished.

As a summary, it is possible to ensure that line **21** represented the first moment were the event data (and metadata) was being written to the session's buffers.

Despite the fact the previous analysis seemed to be correct, it was necessary to confirm that all data being logged to the ETW buffers, was the expected one. It was possible to force the system to log an event to the DiagTrack session (MORE IN TRIGGERS SECTION). Therefore, the following strategy was followed in order to confirm that all the findings were correct:

1. Get the logger id of the DiagTrack session.
2. Set a breakpoint few instructions before the actual writing, taking into account the logger id.
3. Once hit, print the event descriptor and the amount of events in the DiagTrack's buffer.
4. Set a new breakpoint few instruction after the writing.
5. Print the event descriptor again and compare it against previous value.
6. Print the event content to ensure everything is working as expected

A breakpoint was set few instructions before calling the $EtwpReserveTraceBuffer$ function, which meant that the event wouldn't be written yet. Once the debugger got that point, the event descriptor was printed in order to have something to compare with afterwards:

```
!wmitrace.strdump
bp nt!EtwpEventWriteFull+0x286 ".if(r12d == 0x22){.echo 'YEP! it
    entered';.ech ''}.else{gc};gc;"
g
dt nt!_EVENT_DESCRIPTOR @r13
```

Fig. 4.23: Steps 1, 2 and 3.

At that point, $0x22$ was the logger id of the DiagTrack session. After printing the event descriptor, the second part was executed:

```
bp nt!EtwpEventWriteFull+0x3c1
g
dt nt!_EVENT_DESCRIPTOR @r13
!wmitrace.eventlogdump 0x22
```

Fig. 4.24: Steps 4,5 and 6.

The new breakpoint (few instructions after writing) was hit. The event descriptor was exactly as before and the log content had the expected information. After this tests, it was possible to conclude that the analysis performed earlier was correct.

4.3 Automation

Although a big part of all the previous analysis was carried out using static analysis, dynamic analysis also played a central role on it. Furthermore, the final goal of the research was always to have a automatic framework capable of monitor every log of the DragTrack session in real time.

The following sections will show every Windbg script developed in order to overcome particular situations. Definitely there were several intermediate version of each script, but for the sake of simplicity only final version will be shown.

4.3.1 Automating DiagTrack logger id search

One of the very first things that needed to be automated was the search for the logger id of the DiagTrack session. This value is key for the whole process due to is the one which ensures always that the DiagTrack session is the one being processed and not any other session.

The first mission was to try to find in which moment this value was defined for each ETW session. By inspecting the symbols of ntoskrnl.exe the function $EtwpLookupLoggerIdByName$ appeared and because of its name was the first candidate. In order to confirm that this was a good candidate, we start reversing it and debugging it.

```

__int64 __fastcall EtwpLookupLoggerIdByName(__int64 ptr_silo_globals, const
    UNICODE_STRING *logger_name, unsigned int *addr_of_logger_id)
{
  unsigned int *logger_id_to_ret; // r14
  const UNICODE_STRING *logger_name_cpy; // r15
  __int64 ptr_silo_globals_cpy; // rbp
  unsigned int v6; // esi
  unsigned int logger_id_iterator; // ebx
  unsigned int *logger_context; // rax
  unsigned int *logger_context_cpy; // rdi

  logger_id_to_ret = addr_of_logger_id;
  logger_name_cpy = logger_name;
  v6 = 0xC0000296
  ptr_silo_globals_cpy = ptr_silo_globals;
  logger_id_iterator = 0;
  while ( 1 )                               // Iterating all loggers
  {
    logger_context = EtwpAcquireLoggerContextByLoggerId(ptr_silo_globals_cpy,
        logger_id_iterator, 0);
    if ( logger_context )
      break;
LABEL_3:
    if ( ++logger_id_iterator >= 0x40 )     // break conditions (there can't be
        morethat 64 sessions)
      return v6;
  }
  [...]

  EtwpReleaseLoggerContext(logger_context_cpy, 0);
  v6 = 0;
  *logger_id_to_ret = logger_id_iterator;
  return v6;
```

Fig. 4.25: Pseudocode snippet of *EtwpLookupLoggerIdByName*.

From it's code it was possible to understand that:

1. The first parameter (*rcx*) is the pointer to the **_ETW_SILODRIVERSTATE** structure (figure 4.7), the second one (*rdx*) a pointer to a unicode string (probably the name logger name) and the third one seems to be just a buffer where the logger id for this new will be placed.
2. There is an iterator which represents the logger id number, that will go from 0 up to 0x40 (64). This iterator won't be able to go further than 0x40 because is the maximum amount of sessions that can be held inside the ETW framework at the same time.
3. As a summary, this function seems to be the one that actually maps a logger name against a logger id and registers this relation inside the **_ETW_SILODRIVERSTATE** structure.

To help validating this information, we proceed to perform a dynamic analysis. Most probably, the call to this function was going to happen during (or right after) the operating system was booting. In order to confirm this idea, a breakpoint at function *EtwpLookupLoggerIdByName* was set and the debugee system was turned on:

```
bp nt!EtwpLookupLoggerIdByName
```

After a couple of seconds the breakpoint was hit. To confirm the aforementioned statements, two commands were issued:

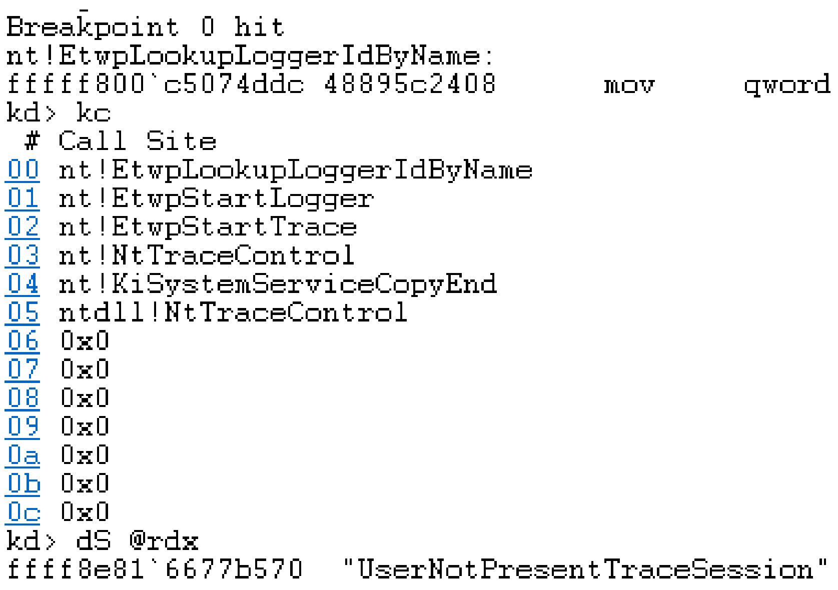

```
Breakpoint 0 hit
nt!EtwpLookupLoggerIdByName:
fffff800`c5074ddc 48895c2408      mov     qword
kd> kc
 # Call Site
00 nt!EtwpLookupLoggerIdByName
01 nt!EtwpStartLogger
02 nt!EtwpStartTrace
03 nt!NtTraceControl
04 nt!KiSystemServiceCopyEnd
05 ntdll!NtTraceControl
06 0x0
07 0x0
08 0x0
09 0x0
0a 0x0
0b 0x0
0c 0x0
kd> dS @rdx
ffff8e81`6677b570  "UserNotPresentTraceSession"
```

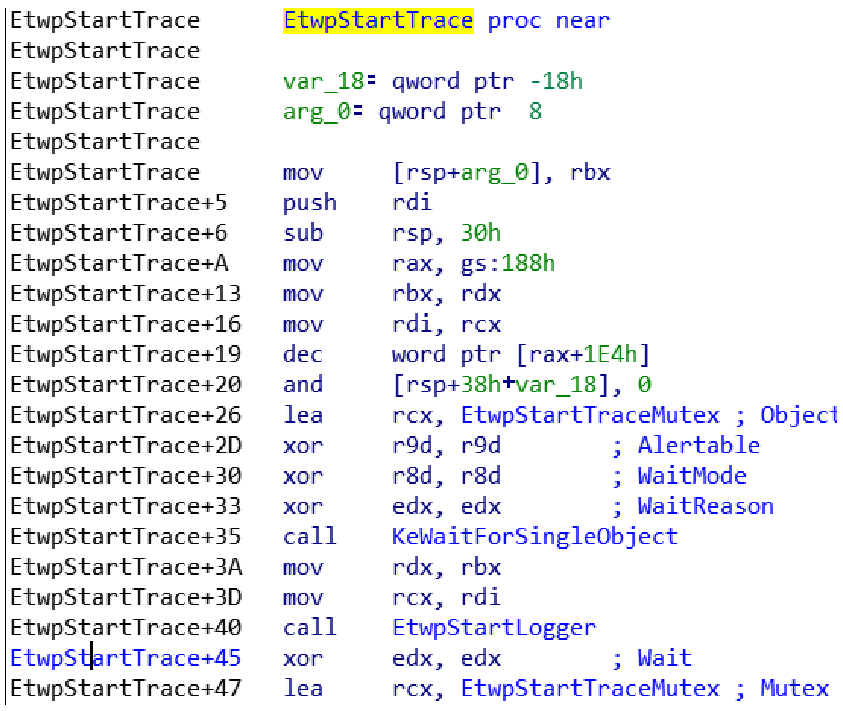
Fig. 4.26: Execution of two Windbg commands when breakpoint of *EtwpLookupLoggerIdByName* was hit.

The first one was **kc**. This prints out the call stack of the function. As can be appreciated, functions *nt!EtwpStartTrace* and *nt!EtwpStartLogger* were the callers. As the name suggests, these functions were creating and registering for the first time the loggers inside the ETW framework, confirming what was stated before.

The second one was **dS @rdx** which basically prints out a string which is unicode-encoded. In this particular case, the string "UserNotPresentTraceSession" was printed. This name is not other than the logger name of one of the several ETW sessions that are initialized by default in Windows.

After this analysis, it was possible to conclude that by setting a breakpoint inside *EtwpLookupLoggerIdByName* and comparing the logger name against the hardcoded one used for the DiagTrack session (**Diagtrack-Listener**) it was possible to detect when the logger was being created. However, the logger id won't be filled when the breakpoint is hit, but when this function ends.

The next step was to find a place (function) that will be executed once the function *EtwpLookupLoggerIdByName* finishes (so that the logger id for the Diagtrack session is already set). Analyzing the call stack shown in figure 4.27, it can be appreciated that *nt!EtwpStartTrace* would be a good candidate. Once the instruction that makes the call to *nt!EtwpStartLogger* finishes, it means that the logger id was already created and related to the corresponding session.



```
EtwpStartTrace       EtwpStartTrace proc near
EtwpStartTrace
EtwpStartTrace       var_18= qword ptr -18h
EtwpStartTrace       arg_0= qword ptr  8
EtwpStartTrace
EtwpStartTrace       mov     [rsp+arg_0], rbx
EtwpStartTrace+5     push    rdi
EtwpStartTrace+6     sub     rsp, 30h
EtwpStartTrace+A     mov     rax, gs:188h
EtwpStartTrace+13    mov     rbx, rdx
EtwpStartTrace+16    mov     rdi, rcx
EtwpStartTrace+19    dec     word ptr [rax+1E4h]
EtwpStartTrace+20    and     [rsp+38h+var_18], 0
EtwpStartTrace+26    lea     rcx, EtwpStartTraceMutex ; Object
EtwpStartTrace+2D    xor     r9d, r9d        ; Alertable
EtwpStartTrace+30    xor     r8d, r8d        ; WaitMode
EtwpStartTrace+33    xor     edx, edx        ; WaitReason
EtwpStartTrace+35    call    KeWaitForSingleObject
EtwpStartTrace+3A    mov     rdx, rbx
EtwpStartTrace+3D    mov     rcx, rdi
EtwpStartTrace+40    call    EtwpStartLogger
EtwpStartTrace+45    xor     edx, edx        ; Wait
EtwpStartTrace+47    lea     rcx, EtwpStartTraceMutex ; Mutex
```

Fig. 4.27: Dissassembly of *nt*!*EtwpStartTrace*

Inspecting the assembly code of *nt*!*EtwpStartTrace*, it was possible to confirm that *nt*!*EtwpStartTrace* + 45 would be a very good place where to set the second breakpoint as it the immediate instruction after the execution of the aforementioned functions.

In summary, in order to finally get the logger id of the Diagtrack session the following steps should be accomplished:

1. Set a breakpoint at *EtwpLookupLoggerIdByName*.
2. Until the logger name (which is held in *rdx*) is not "**Diagtrack-Listener**", resume execution.
3. Once it matches, create a one-time breakpoint at *nt*!*EtwpStartTrace* + 45 and jump to it.
4. Once this second breakpoint is hit, extract the information about the logger id from the _ESERVERSILO_GLOBALS structure.

The final version of the script to accomplish this task was splitted in two parts. First part illustrates steps 1, 2 and 3. The second part implements the 4th step.

```
$$ RCX --> Pointer to ETW_SILODRIVERSTATE
$$ RDX --> Pointer to the logger name (unicode) (+0x8)

$$ Save the pointer to the logger name
r $t0 = poi(@rdx+0x8);

$$ Compare the string using alias
as /mu ${/v:LOGG_NAME} $t0;
.block{r $t10 = 0x0;r $t10 = $spat(@"${LOGG_NAME}", "Diagtrack-Listener")}
ad LOGG_NAME

$$ If it was the Diagtrack-Listener
.if($t10 == 1){
    $$ Save the pointer to ETW_SILODRIVERSTATE
    r $t1 = rcx;

    $$ The array is in offset 0x1b0 of ETW_SILODRIVERSTATE
    r $t2 = $t1+0x1b0;

    $$ Lets put a breakpoint just after the logger was created
    bp /1 nt!EtwpStartTrace+0x45 "$$><\"path_to_second_part_script"
};
gc
```

Fig. 4.28: First part of script to get the logger id .

```
.for(r $t19 = 0;@$t19 < 0x40;r $t19 = @$t19+1){
  $$ First, check if it is empty
  .if ( poi(poi(@$t2)+@$t19*0x8) == 1){
      .continue
  }
  .else{
      $$ Save the pointer to the WMI_LOGGER_CONTEXT in t4
      r $t4 = poi(@$t2) + (@$t19*0x8);

      $$ Save the STRING UNICODE object in t5
      r $t5 = poi(@$t4)+0x98;

      $$ Save the UNICODE buffer
      r $t6 = poi(@$t5+0x8);

      $$ Create the alias of Unicode String
      as /mu ${/v:LOGG_NAME_NEW} $t6;
      .block{r $t10 = 0x0;r $t10 = $spat(@"${LOGG_NAME_NEW}",
          "Diagtrack-Listener")}
      ad LOGG_NAME_NEW

      .if($t10 == 1){
          .printf "Logger id of Diagtrack-Listener found!!: %N\n", @$t19
          $$ <PLACE FOR FUTURE CODE >
          .break
      };
  };
};
gc
```

Fig. 4.29: Second part of script to get the logger id .

4.4 Checking correctness of logged events

4.5 Automatization of event logging

4.6 Service isolation

4.7 Triggers

4.8 searching for new triggers

YARA

4.9 Difference among configuration levels of telemtry

4.10 Analysis of sent data over the channel to Microsfot backend services

5. RESULTS

6. CONCLUSIONS

GDPR?

7. APPENDIXES

7.1 Structures layout

- _ESERVERSILO_GLOBALS, figure 4.6
- _ETW_SILODRIVERSTATE, figure 4.7
- _ETW_HASH_BUCKET, figure 4.11
- _ETW_GUID_ENTRY, figure 4.12
- _GUID, figure 4.13
- _REG_GUID_ENTRY,
- _WMI_TRACE_INFO,
- _TRACE_ENABLE_INFO,

8. REFERENCES

BIBLIOGRAPHY

[1] Bolin Ding, Janardhan Kulkarni and Sergey Yekhanin. Collecting Telemetry Data Privately. In proceedings of Neural Information Processing Systems Conference (NIPS), 2017.

[2] Vasyl Pihur, Úlfar Erlingsson and Aleksandra Korolova. RAPPOR: Randomized Aggregatable Privacy-Preserving Ordinal Response. In CCS, pages 1054–1067, 2014

[3] Microsoft Corporation. Dynamic collection analysis and reporting of telemetry data. US 9,590,880 B2, 2017.

[4] Tarik Soulami. Inside Windows debugging. Chapter 12, 2012

[5] Hausi A. Miiller, Jens H. Jahnke, Kenny Wong ,Dennis B. Smith , Scott R. Tilley , Margaret-Anne Storey. Reverse Engineering: A Roadmap. In Proceedings of the Conference on The Future of Software Engineering, Pages 47-60, 2000.

[6] Bruce Dang, Alexandre Gazet, Sbastien Josse, Elias Bachaalany. Practical Reverse Engineering: x86, x64, ARM, Windows Kernel, Reversing Tools, and Obfuscation. 2014.

[7] Online Microsoft Documentation about ETW: https://docs.microsoft.com/en-us/windows/win32/etw/event-tracing-portal