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 following analysis was performed against the Telemetry component of the Windows OS, with looking forward to achieve 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

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 ..

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

Several files (dynamic libraries, executables, drivers) were analyzed and reversed in order to achieve the aforementioned goals. However, the main analysis was performed in the **ntoskrnl.exe** file, which is the one holding the actual implementation of the Windows Kernel.

It's important to stress that most of the general analysis was carried out using the Basic level of Telemetry.

Further sections will depict different challenges and achievements faced during the analysis.

4.1 Understanding how Telemetry makes use of ETW

Where, who and what were some of the questions that needed to be answered in order to be able to get information about the providers that were registered against the DiagTrack session. It's possible to obtain the whole 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 comand to list ETW providers registered against a particular session.

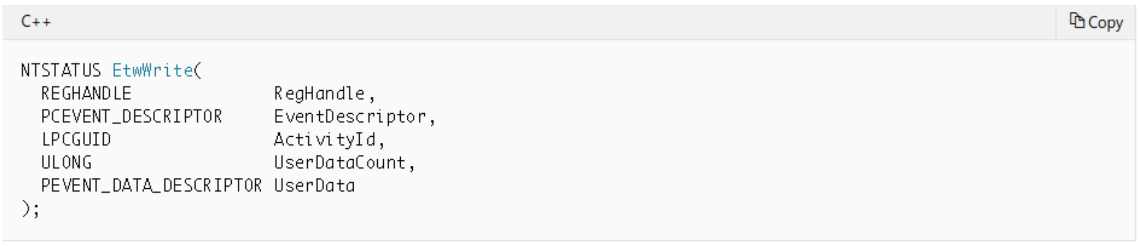
One way to answer all the aforementioned questions, was to intercept the moment when some provider was going to write a message. If a breakpoint was set at that exact moment, it would be possible to gather information such as:

1. The piece of code that triggered the write (by inspecting the call stack of the function).
2. The actual content of the log being written.

However, the function being used for executing writes (**EtwWrite**) was not just used by the providers attached to the DiagTrack session but also by all the providers using the ETW framework. It was necessary to find a way to filter them and only intercept the ones important for the analysis.

Using the powershell command shown in figure 4.1, it was possible to extract a GUID's list of the providers that were attached to the DiagTrack session. With this information, it would be possible to make the breakpoint to be triggered only when the provider's GUID that was trying to write was in the list.

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

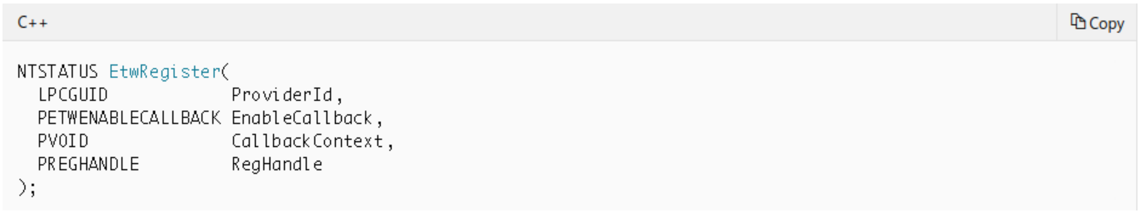


C++ Copy

```
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 is the registration handler. This object is returned once the provider executed the registration (**EtwRegister**) successfully. On the other hand, the **EtwRegister** receives the GUID as parameter:



C++ Copy

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

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

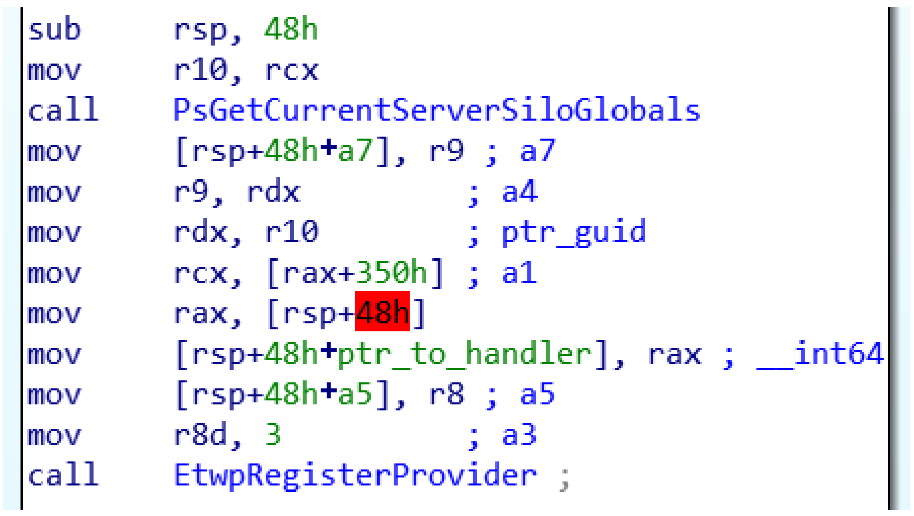
Therefore, in order to perform the cross-check it was not enough with information from the **EtwWrite** function, but also information from the **EtwRegister** was needed. To summarize, to understand if the write was being done by a provider registered against the DiagTrack session it was necessary to:

1. Extract the whole list of providers registered attached to 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 the strategy seemed to be theoretically promising, it was necessary to understand how to actually carry out these actions. Further sections will depict that 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 we can see in 4.4, the only 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: Dissassembly of ETWRegister function

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

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

4.1.1.1 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.

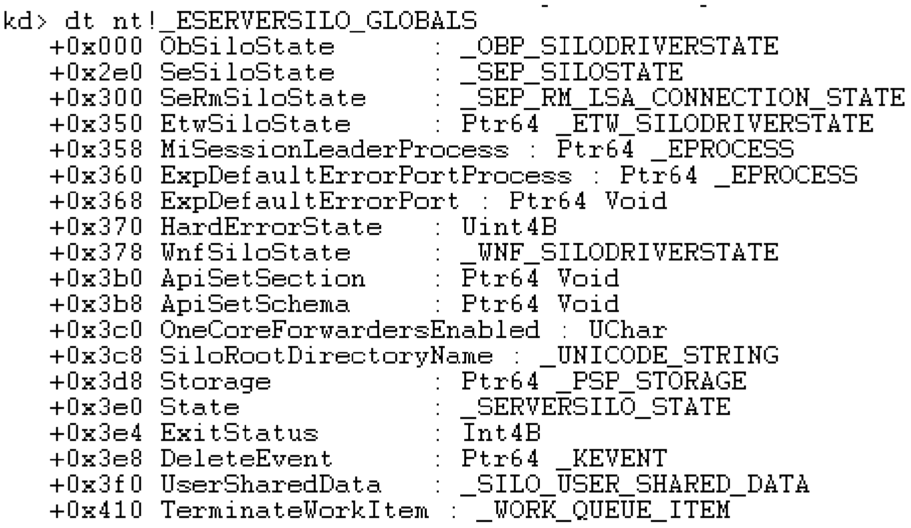
The main focus here was not to understand how the registration process fully worked but just to get an idea of it plus get to know the relation between GUID and registration handler.

After understanding a little bit more about this function, it was possible to conclude that:

1. **a1**: Is the pointer to a structure.

2. **a2**: Is the pointer to the GUID structure.
3. **a7**: Is the address where the pointer to the registration handler will be placed (can be think as "function output").

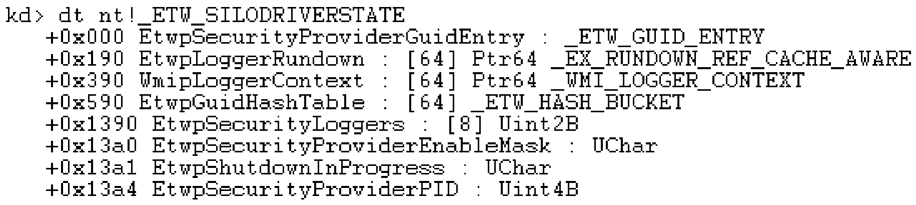
What is this **a1** structure? The figure 4.4 shows that before calling **EtwpRegister-Provider**, the function **PsGetCurrentServerSiloGlobals** is invoked. This latter one returns a pointer to a structure S of type **_ESERVERSILO_GLOBALS**.



```
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. 4.5: _ESERVERSILO_GLOBALS structure layout (S).

However, the first parameter provided to **EtwpRegisterProvider** is not the pointer to S but is the pointer to another structure S_2 of type **_ETW_SILODRIVERSTATE**. which is part S, more precisely, it is situated at the offset 0x350.



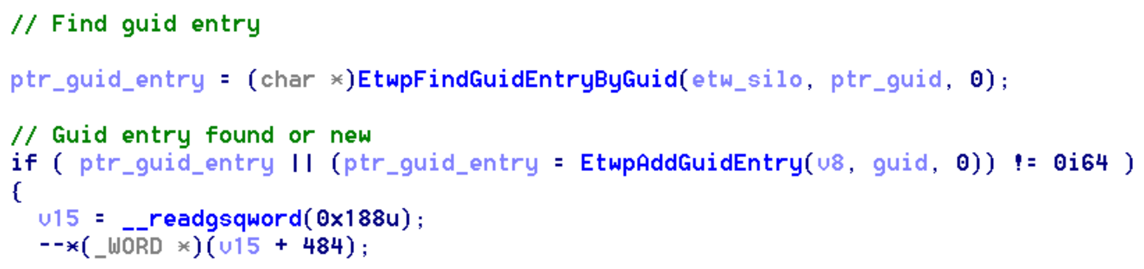
```
kd> dt nt!_ETW_SILODRIVERSTATE
   +0x000 EtwpSecurityProviderGuidEntry : _ETW_GUID_ENTRY
   +0x190 EtwpLoggerRundown : [64] Ptr64 _EX_RUNDOWN_REF_CACHE_AWARE
   +0x390 WmipLoggerContext : [64] Ptr64 _WMI_LOGGER_CONTEXT
   +0x590 EtwpGuidHashTable : [64] _ETW_HASH_BUCKET
   +0x1390 EtwpSecurityLoggers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

Fig. 4.6: _ETW_SILODRIVERSTATE structure layout (S_2).

With all this information, 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. In the other hand, **a2** and **a7** will hold pointers to a GUID and to a place were a registration handler will be placed afterwards, respectively. With the knowledge of the purpose of just these three parameters, was possible to move forward.

4.1.1.2 Check if GUID for this provider already exists

Once inside the *EtwpRegisterProvider* function, after performing some sanity checks, it tries to get the guid entry related with the GUID. 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);
```

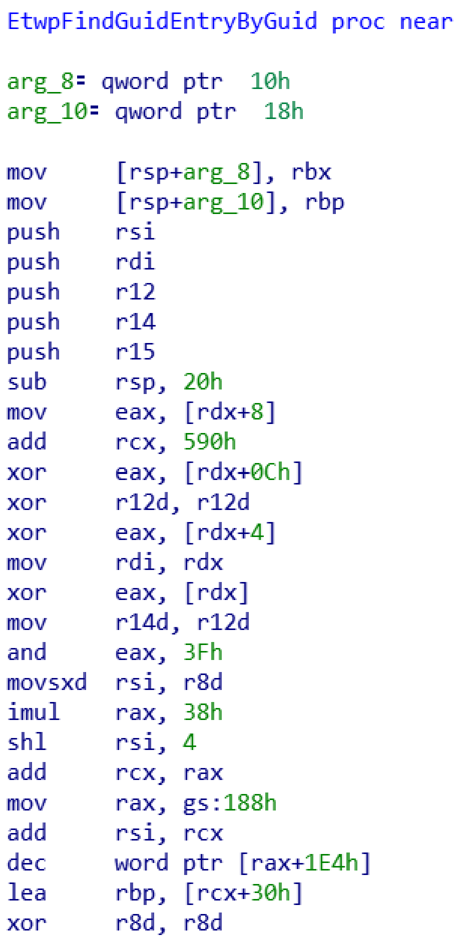
Fig. 4.7: First lines of EtwpRegisterProvider

In this part, the focus will be on understanding how the process of recovering the already existing "GUID entry" works.

The action of recovering is performed by a particular function called **EtwpFind-GuidEntryByGuid**:

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

As can be inferred from the previous line, two important parameters are 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. 4.8: First basic block of EtwpFindGuidEntryByGuid.

Figure 4.6 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*.

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 4.6), it can be appreciated that at the offset $0x590$, the structure *EtwpGuidHashTable* of type $\_ETW\_HASH\_BUCKET[64]$ is present. Figure 4.10 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, an and function is applied against *eax* (result of xoring) using a mask of $0x3f$. A mask like this means that only the last 6 bits of the result really cares, and this is totally reasonable due to $2^6 = 64$. 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.

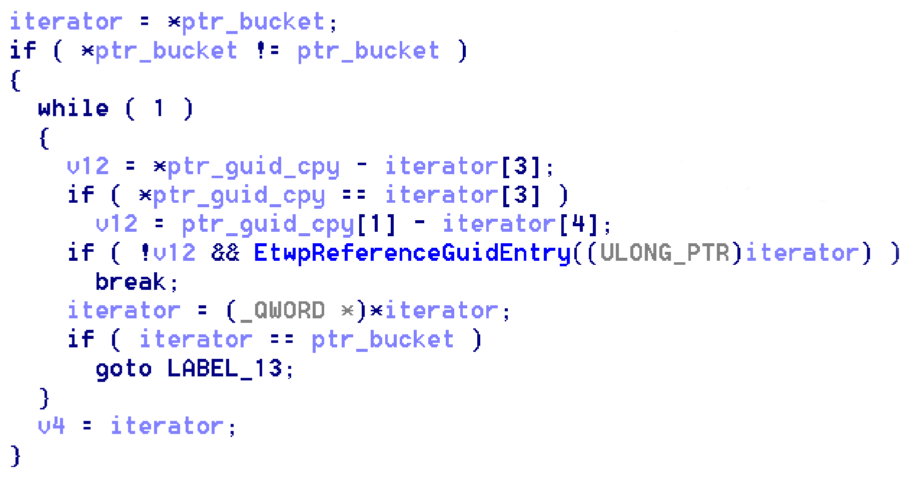
Writing the aforementioned function in a pseudo-code style (*ptr* is a short version of "pointer"):

```
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 determined, a "look up" inside the structure is carried out in the following way:

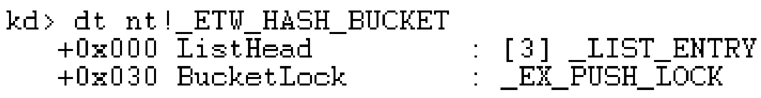
[3] Sometimes the structures weren't documented at all. Sometimes they were, but was not possible to find it until some kind of clue pointing to it was found. So far the layout of the structure pointed by *ptr_guid* was unknown, however from this function it was possible to conclude that the structure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

[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.



```
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. 4.9: 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. 4.10: _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 4.10). 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. The only important thing of this routine is that it executes a return statement with the value of the variable $v4$ (which is initially defined as 0).

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 structure that represents the GUID entry.

Continuing 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

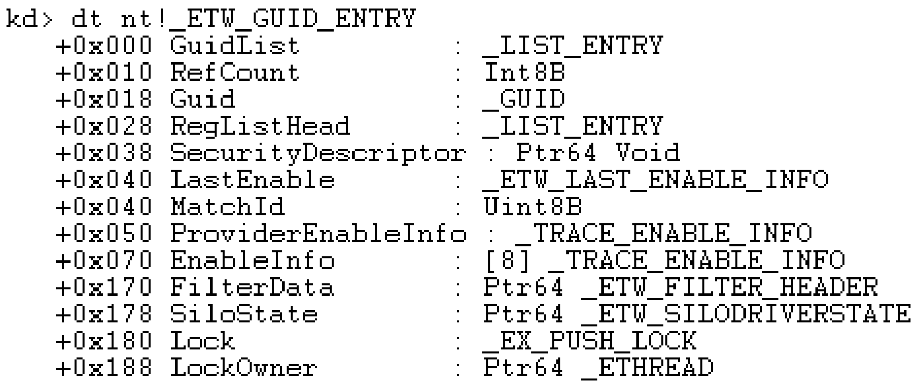
[5] https://docs.microsoft.com/en-us/windows/desktop/api/ntdef/ns-ntdef-_list_entry

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.

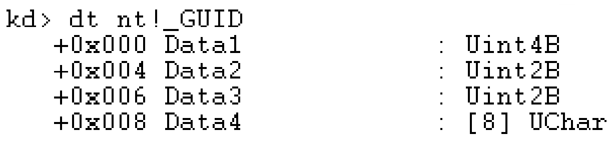
Therefore, as a summary, it is possible to say that:
The function *EtwpFindGuidEntryByGuid* looks for a particular structure (most probably called guid entry), which is stored inside a double-linked-list of a bucket inside the *EtwpGuidHashTable* of the *_ETW_SILODRIVERSTATE*, based on doing some mathematical 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. 4.11: _ETW_GUID_ENTRY structure.



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

Fig. 4.12: _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 4.11)
- The GUID was a structure of 16 bytes long (figure 4.12)

4.1.1.3 If not found, create a new one

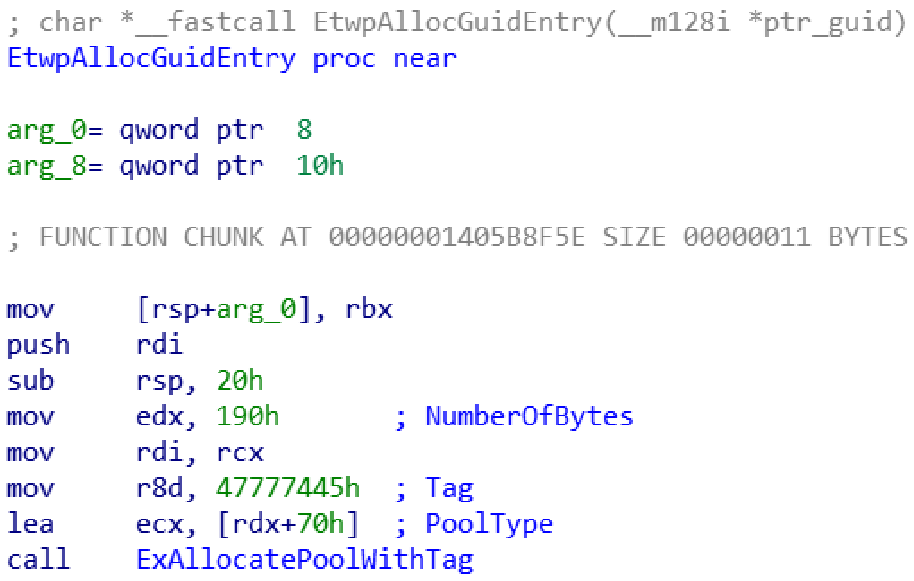
In the previous section, it was explained how was the process to find an already existing guid entry based on the GUID of the provider. In the current one, the process of creating a new one will be depicted.

From figure 4.7 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_cpy2*) 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. 4.13: *EtwpAllocGuidEntry* allocation.

As can be observed in figure 4.13 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* (**??**)layout.
- **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".

[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

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 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)

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 look for the correct place to insert it as depicted previously.

4.1.1.4 **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 will be the "output" of the function. 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*.

No matter if 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:

```
__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.

1. We couldn't ensure that the data was being written was actually going to the DiagTrack session
.

4.2 When and how providers are registered

4.3 How writes are carried out

4.4 Relation between ETW session and ETW providers

4.5 Identifying the buffers

4.6 Provider GUID vs Group Provider GUID

4.7 Checking correctness of logged events

4.8 Automatization of event logging

4.9 Service isolation

4.10 Triggers

4.11 searching for new triggers

YARA

4.12 Difference among configuration levels of telemtry

4.13 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 4.5
- _ETW_SILODRIVERSTATE, figure 4.6
- _ETW_HASH_BUCKET, figure 4.10
- _ETW_GUID_ENTRY, figure 4.11
- _GUID, figure 4.12

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.