ADVANCING THE STATE OF UEFI BOOT KITS

Persistence in the age of PatchGuard and Windows 10

a.k.a

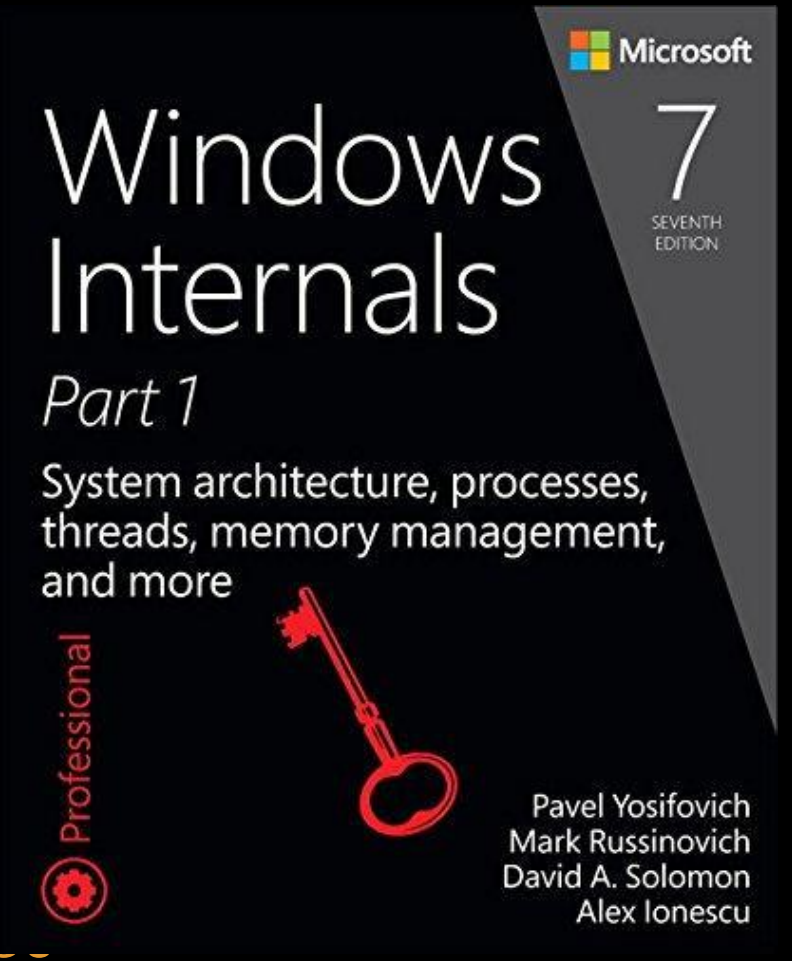
“lol Alex thinks LoJack is cool” -- @dwizzzleMSFT

Alex Ionescu [@aionescu]

OffensiveCon 2018, Berlin

BIO

- VP of EDR Strategy and Founding Architect at CrowdStrike
  - Previously worked at Apple on iOS Core Platform Team
- Co-author of *Windows Internals 5th-7th Editions*
- Reverse engineering NT since 2000
  - Lead kernel developer of ReactOS (now UEFI Boot Loac
- Instructor of worldwide Windows internals classes
- Author of various tools, utilities and articles
- Conference speaking:
  - SyScan 2012-2015, Infiltrate 2015, OffensiveCon 2018
  - NoSuchCon 2013-2014, Breakpoint 2012, EkoParty 2017
  - Recon 2010-2018, EuskalHack 2017
  - Blackhat 2008, 2013-2016, 18?
- For more info, see www.alex-ionescu.com or @aionesc



AGENDA

- Introduction / Bio / Motivation
- “File-Less” UEFI-based Kernel Hooks
- Loader Block Manipulation Galore
- Persistence and PatchGuard
- PatchGuard is Your Friend
- ACPI-Based Persistence
- Make ACPI Hooking and Vulnerabilities Great Again
- End-to-End Persistent C2 from OS<->Firmware
- VTL1, Hypervisors and ~~KSR~~

MOTIVATION

- Windows security technologies are getting really good and really serious
  - Especially when relying on Hyper-V (HyperGuard, Device Guard, KCFG, HVCI, Octagon)
- A lot of assumptions on correctness of firmware and pre-boot security
  - If SMM is broken -> Game Over
  - If SecureBoot is broken -> Game Over
  - New technologies like TXT-integration/System Guard/Trusted Boot might help here…
    - RS5+ Pray-to-your-OEM
- But given that, as of today at least, SMM & SecureBoot can be assumed broken…
  - Why aren't there any good UEFI-related persistence & boot kit techniques out

IT'S HARD…

- Firmware attacks are typically very 'targeted'
  - This makes them less useful for IC/Nation State real-life attacks vs. academic ideas
- The world isn't all on UEFI/Windows 8+
  - Granted – this is changing
- Not a lot of public research that's actually practical
  - And you know our APTs actors love their copy pasta
- There's actually good technologies such as PatchGuard that make it really hard to maintain persistence and not crash machines
  - Ongoing investment from Microsoft is making it harder still
- Still, a lot of holes that can affect 90% of today's machines
  - Let's make sure defenders know about them, and are watching
  - `Cause maybe everyone *is* bootkitted, but you just don't know.

GREETZ

- @matrosov – He gets it
  - Other people get it too – they just be' busy building T2 chips for Papa Legba
- @dwizzleMSFT – Always be trollin'
- @int0x6 – Useful review feedback
- @epakskape / @let's-get-ken-aka-skywing-on-twitter-already – Putting up with my BS
- @bluefrostsecurity – Holy sh*t! THIS! What a great con, eh?
  - Yep I'm Canadian
- @aall86 – First UEFI bootkit back in Windows 8 – pretty much set the bar that nobody went past…
- @ALotOfOtherPeople – you know who you are. <3

FILE-LESS UEFI-BASED HOOKS

Hooking like it's not 1999

FILE-LESS HOOKING

- Every single UEFI boot kit I've looked at modifies the bootmgfw.efi file on disk…
  - Hello, 1999 called and wants its b*lls*it MBR rootkits back
- This is so pathetic that even Microsoft's UEFI Signature Portal checks for file I/O from submitted .efi files
  - If the automated portal that WHQL-signs IOCTL_READ_WRITE_EVERYTHING is catching this… you should reconsider your rootkit-writing career
- So how can we actually mess with the kernel "in-memory"?
- ANSWER: UEFI Event Notifications!
  - See CreateEventEx in Boot Services UEFI Specification

UEFI EVENT NOTIFICATIONS

- There are two key events one can register for – one at the end of 'physical' memory mode, and one at the start of 'virtual' memory mode

- The first is EFI_EVENT_GROUP_EXIT_BOOT_SERVICES – called when ExitBootServices is called, right before actually exiting boot services
  - At this point, the boot loader has loaded all the interesting boot structures in physical memory, and has created virtual mappings (not yet active) for them

  - Additionally, the call stack is fully synchronous, meaning RSP eventually points back to *OslFwpKernelSetupPhase1*

  - This functions takes LOADER_PARAMETER_BLOCK as input, and often ends up in a nonvolatile register – lots of ways to recover its pointer
    - Or do many other kinds of fingerprinting techniques to find what you need…

UEFI EVENT NOTIFICATIONS

- The second GUID is EFI_EVENT_GROUP_VIRTUAL_ADDRESS_CHANGE
  - This occurs when virtual memory is enabled, and paging has been turned on
  - Your callback can how use UEFI services for converting any physical addresses it had stored into virtual addresses (with *ConvertPointer*)
  - Called inline by SetVirtualAddressMap which is called by *OslFwpKernelSetupPhase1* ~50 instructions after the earlier callback
- Seriously if you can't figure out what/where to hook in memory with these two calls…
  - You don't belong in the UEFI Boot Kit Business!
- So let's talk about LOADER_PARAMETER_BLOCK…

LOADER PARAMETER BLOCK MANIPULATION

PSA: Thank you whoever leaked the entire MinWin headers in TH1 and RS1

WHAT IS THE LOADER BLOCK?

- The boot loader does a metric ….ton of work to get the kernel loaded
  - Including loading the registry
  - And all the drivers
  - And the hypervisor
  - And the shim database
  - And the API set mappings
  - And the INF errata
  - And the ELAM hive
  - And the page tables
  - And the kernel imports
  - And gather boot entropy
  - And hash everything / TPM-all-the-things
  - And setup TCP/UDP for netboot if needed
  - And gather boot-time configuration parameters from the BCD and oh man I'm

WHAT'S IN THE LOADER BLOCK?

- So the boot loader needs to pass along all that data to the kernel
- This is done by sending a parameter to its entrypoint called the loader parameter block
- This structure leaked in NT4 sources, and Win2K source, and 2003 source… and eventually made it into the Windows 7 symbols (yay)
- Contains loads of interesting data that one can now hook from their UEFI boot-time component

WINDOWS 8 LOADER BLOCK

- Unfortunately, between NT4/2K/2003/7, most of the loader block stayed the same
  - That being said, they did add a 'header' in Windows 7, and it really helps for forward-compatible boot kit support
  - Even better since real UEFI support is only in Windows 7+
- But post Windows 7, they were smart enough to remove the symbol
  - And breaking changes were made to the structure, such as supporting ELAM
  - Symbol hasn't come back since ☹
- But that's OK, they leaked the entire structure in the Windows 10 SDKs for TH2
  - And they were told
  - And they leaked it again in RS1… and they were told…
  - And actually leaked in early RS2 Preview SDKs too

```
kd> dt nt!KeLoaderBlock
0x80bb6c18
   +0x000 OsMajorVersion   : 6
   +0x004 OsMinorVersion   : 2
   +0x008 Size             : 0xa0
   +0x00c Reserved         : 0
   +0x010 LoadOrderListHead : _LIST_ENTRY [ 0x80bddcb8 - 0x80c10e10 ]
   +0x018 MemoryDescriptorListHead : _LIST_ENTRY [ 0x80e2c000 - 0x80e2cd98 ]
   +0x020 BootDriverListHead : _LIST_ENTRY [ 0x80bb8a78 - 0x80be8a30 ]
   +0x028 EarlyLaunchListHead : _LIST_ENTRY [ 0x80bb8f30 - 0x80bb8f30 ]
   +0x030 CoreDriverListHead : _LIST_ENTRY [ 0x80bb9058 - 0x80bea380 ]
   +0x038 KernelStack      : 0x80f60000
   +0x03c Prcb             : 0
   +0x040 Process          : 0
   +0x044 Thread           : 0x81a720c0
   +0x048 KernelStackSize  : 0x3000
   +0x04c RegistryLength   : 0x800000
   +0x050 RegistryBase     : 0x80fa1000 Void
   +0x054 ConfigurationRoot : 0x80bb6e98 _CONFIGURATION_COMPONENT_DATA
   +0x058 ArcBootDeviceName : 0x80bb8768  "multi(0)disk(0)rdisk(0)partition(2)"
   +0x05c ArcHalDeviceName : 0x80bb8698  "multi(0)disk(0)rdisk(0)partition(1)"
   +0x060 NtBootPathName   : 0x80bb8738  "\Windows\"
   +0x064 NtHalPathName    : 0x80bb8608  "\"
   +0x068 LoadOptions      : 0x80ba9c48  " BOOTDEBUG  NOEXECUTE=OPTIN  DEBUG  E
   +0x06c NlsData          : 0x80c10518 _NLS_DATA_BLOCK
   +0x070 ArcDiskInformation : 0x80ba9be8 _ARC_DISK_INFORMATION
   +0x074 Extension        : 0x80bda5e8 _LOADER_PARAMETER_EXTENSION
   +0x078 u                : <unnamed-tag>
   +0x084 FirmwareInformation : _FIRMWARE_INFORMATION_LOADER_BLOCK
```

```
lkd> dt nt!_BLDR_DATA_TABLE_ENTRY
   +0x000 KldrEntry        : _KLDR_DATA_TABLE_ENTRY
   +0x05c CertificatePublisher : _UNICODE_STRING
   +0x064 CertificateIssuer : _UNICODE_STRING
   +0x06c ImageHash        : Ptr32 Void
   +0x070 CertificateThumbprint : Ptr32 Void
   +0x074 ImageHashAlgorithm : Uint4B
   +0x078 ThumbprintHashAlgorithm : Uint4B
   +0x07c ImageHashLength  : Uint4B
   +0x080 CertificateThumbprintLength : Uint4B
   +0x084 LoadInformation  : Uint4B
   +0x088 Flags            : Uint4B
```

LOADER BLOCK IDEAS

- Add dangerous load options even though SecureBoot is enabled?
  - Check ✅
- Unlink any driver from the boot driver list, including ELAM drivers?
  - Check ✅
- Edit the SHA hash that ELAM drivers will receive for any boot driver?
  - Check ✅
- Edit what the HAL will think are the real UEFI runtime services?
  - Check ✅
- Patch the registry in-memory? Mess with the kernel stack? Hide RAM from Mm?
  - Check ✅
- Seriously if you can't think of 10 other things as part of your UEFI Boot Kit…
  - Go home

LOADER BLOCK EXTENSION

- For compatibility reasons, the loader block doesn't actually have all the information the kernel uses
- The rest is in the "Extension" structure
  - Which itself has lots of sub-extensions (hypervisor extension, network boot extension, headless extension, etc…)
- Again, all in Windows 7 symbols as well as in 2015-2016 Windows 10 SDKs
- This provides even more interesting areas for an UEFI Boot Kit to mess around with

```
kd> dt 0x80bda5e8 _LOADER_PARAMETER_EXTENSION
nt!_LOADER_PARAMETER_EXTENSION
   +0x000 Size             : 0x870
   +0x004 Profile          : _PROFILE_PARAMETER_BLOCK
   +0x014 EmInfFileImage   : 0x82d0a000 Void
   +0x018 EmInfFileSize    : 0x15fff
   +0x01c TriageDumpBlock  : (null)
   +0x020 HeadlessLoaderBlock : (null)
   +0x024 SMBiosEPSHeader  : 0x80ba9b18 _SMBIOS_TABLE_HEADER
   +0x028 DrvDBImage       : 0x82ccf000 Void
   +0x02c DrvDBSize        : 0x3a4ee
   +0x030 NetworkLoaderBlock : (null)
   +0x034 HalpIRQLToTPR    : (null)
   +0x038 HalpVectorToIRQL : (null)
   +0x03c FirmwareDescriptorListHead : _LIST_ENTRY [ 0x80bda624 - 0x80bda624 ]
   +0x044 AcpiTable        : (null)
   +0x048 AcpiTableSize    : 0
   +0x04c LastBootSucceeded : 0y1
   +0x04c LastBootShutdown : 0y0
   +0x04c IoPortAccessSupported : 0y1
   +0x04c BootDebuggerActive : 0y0
   +0x04c Reserved         : 0y00000000000000000000000000000 (0)
   +0x050 LoaderPerformanceData : 0x80ba9b40 _LOADER_PERFORMANCE_DATA
   +0x054 BootApplicationPersistentData : _LIST_ENTRY [ 0x80bb4cf0 - 0x80bb8670 ]
   +0x05c WmdTestResult    : (null)
   +0x060 BootIdentifier   : _GUID {b3007561-0c55-11e2-938e-d8bd19662633}
   +0x070 ResumePages      : 0xc00
   +0x074 DumpHeader       : (null)
   +0x078 BgContext        : 0x80bdf668 Void
   +0x07c NumaLocalityInfo : (null)
   +0x080 NumaGroupAssignment : (null)
   +0x084 AttachedHives    : _LIST_ENTRY [ 0x80bec938 - 0x80bec938 ]
```

```
+0x098 BootEntropyResult : _BOOT_ENTROPY_LDR_RESULT
+0x7a8 ProcessorCounterFrequency : 0x938580c0
+0x7b0 HypervisorExtension : _LOADER_PARAMETER_HYPERVISOR_EXTENSION
+0x7e8 HardwareConfigurationId : _GUID {8ba0356a-59d7-444e-8989-f95a
+0x7f8 HalExtensionModuleList : _LIST_ENTRY [ 0x804dadf8 - 0x804dadf
+0x800 SystemTime       : _LARGE_INTEGER 0x01ce55cb`6b76e880
+0x808 TimeStampAtSystemTimeRead : 0x0000005e`fa28c493
+0x810 BootFlags        : 0
+0x818 InternalBootFlags : 2
+0x820 WfsFPData        : (null)
+0x824 WfsFPDataSize    : 0
+0x828 KdExtension      : _LOADER_PARAMETER_KD_EXTENSION
+0x858 AcpiBiosVersion  : _UNICODE_STRING "VBOX   - 1"
+0x860 SmbiosVersion    : _UNICODE_STRING "VirtualBox"
+0x868 EfiVersion       : _UNICODE_STRING ""
```

MESSING WITH DATA IN MEMORY IS BORING

Show me real persistence, Alex!

UEFI RUNTIME BINARIES/MEMORY

- UEFI actually allows you to allocate "Runtime Pool" as well as have "Runtime Code" (aka Runtime DXEs)
  - These allocations/code persist even after ExitBootServices() is called
  - And must correctly be relocated-fixed up when the Virtual Memory Map is set
- Legitimately, this is used for:
  - PXE Network Boot (UEFI network stack must survive as OS is booting…)
  - UEFI Runtime Variable Access / Capsules (Updates) / UEFI Reset / Monotonic Counter
- Windows stores points to UEFI Runtime functions that it expects to call in the LOADER_PARAMETER_BLOCK, and HAL then saves them internally
  - In RS4, there's now an undocumented HAL API to get them from a 3rd party driver ☺
- UEFI Runtime memory is 'hidden' from most crash dumps
  - In RS4+ they added a lot more visibility around this – but not perfect (kudos

UEFI RUNTIME PERSISTENCE

- OK – so now we have some 'runtime' code and data that has persisted Windows booting
  - We can patch what Windows thinks are the real runtime UEFI functions it normally calls
  - Now as soon as someone queries/sets a UEFI runtime variable (happens all the time) we get code execution
  - But this is very visible, and on modern UEFI systems with Hypervisor, harder to get right

- Is there some other way that we can get our UEFI runtime to 'execute' while Windows is running?
  - We would need to patch/hook something in ntoskrnl.exe's memory that is writable and not executable
    - Otherwise this isn't very effective in light of VTL1/Hypervisor EPT protections
  - We also have to worry about KCFG on RS2+ with Hypervisor Enabled

PERSISTENCE IS HARD…

- There's actually a lot of checks that PatchGuard in Windows 8+ does that I didn't really understand at first
  - Until I wrote my first UEFI Boot Kit…
    - Now you know. It's ok, you're running it for a few years already. Battery life still OK, right?
- One known check is:
  - Does RSP of executing thread point to 'real' RSP from kernel?
    - Detects ROP or 'floating threads' with arbitrary kernel stacks
- But it also checks:
  - Do you have any Ps/Ob callbacks that are real PE registered/signed drivers?
    - OK sure, we don't want floating rootkits out there…
- But also…
  - Do you have an NMI callback/MPX bounds callback that's a real PE signed driver?

THAT SEEMS… SPECIFIC

- Why would a ‘bad guy’ worry about MPX/NMI/LAPIC Timer callbacks?
  - These are all things you can ‘trigger’ or that are automatically triggered
  - Great way to have ‘floating code’ that isn’t ever ‘scheduled’ or shows up as a thread – instead it just random ‘executes’ from time to time as part of an existing thread
  - LAPIC Timer is actually a great one because it’s not subject to KCFG protection as its directly called by the CPU as an interrupt
- So how can we get UEFI Runtime Code to get ‘triggered’ by the OS kernel once booted?
  - In a way that’s non-obvious?
- Well… how does PatchGuard do it?
  - PatchGuard is obfuscated “floating code” that periodically runs
  - And detects *other* floating code – but it’s OK with itself

PATCHGUARD IS OUR FRIEND…

And other persistent-execution/hooking tricks

HOOKING GLOBAL DATA FOR EXECUTION

- We want to find some global variables in the kernel – that we can access from our UEFI Boot Services DXE to get our Runtime DXE/Code to run
  - And to find things that PatchGuard won't fire on (until +1 day after this talk)
- VR Approach: Look at any pre-initialized/non-overwritten/trusted pointers/global tables inside of ntoskrnl.exe
  - Found a good first attempt: AlpcpMessageLogListHead was pre-initialized
  - Adding the right 'entry' in here caused execution every single ALPC message
  - But… only if AlpcpMessageLogEnabled is set
  - Both of these variables are not exported
  - In RS3/4 this is now initialized at runtime (MSFT… please… stop spying on me)
- Found another candidate… EtwpWdfNotifyRoutines

HOOKING WDF

- If we over-write EtwpWdfNotifyRoutines we get access to:

```
lkd> dt WMI_WDF_NOTIFY_ROUTINES
   +0x000 Size              : Uint4B
   +0x008 DpcNotifyRoutine  : Ptr64     void
   +0x010 InterruptNotifyRoutine : Ptr64     void
   +0x018 WorkItemNotifyRoutine : Ptr64     void
```

- But how do we get access to EtwpWdfNotifyRoutines from our UEFI Boot-Time Component?
  - Easy, once you know where Ntoskrnl.exe is loaded…
  - GetProcAddress(WmiQueryTraceInformation) and… call it (see next slide)

- This call doesn't use any Kernel API/require side-effects that make it crash – it helpfully leaks the address of EtwpWdfNotifyRoutines for you ☺

```
746         //
747         // Retrieve a pointer to the data structure that cotains trace routines
748         // corresponding to WdfNotifyRoutinesClass from the SystemTraceProvider
749         // that we'll use for perf tracing of WDF operations. The trace
750         // routines inside the structuyre are present only when tracing is enabled
751         // by some trace client (e.g. tracelog or xperf)  for WDF specific perf
752         // groups. Note that no unregistration is necessary.
753         //
754         status = WmiQueryTraceInformation(WdfNotifyRoutinesClass,
755                                           &FxLibraryGlobals.PerfTraceRoutines,
756                                           sizeof(PWMI_WDF_NOTIFY_ROUTINES),
757                                           NULL,
758                                           NULL);
```

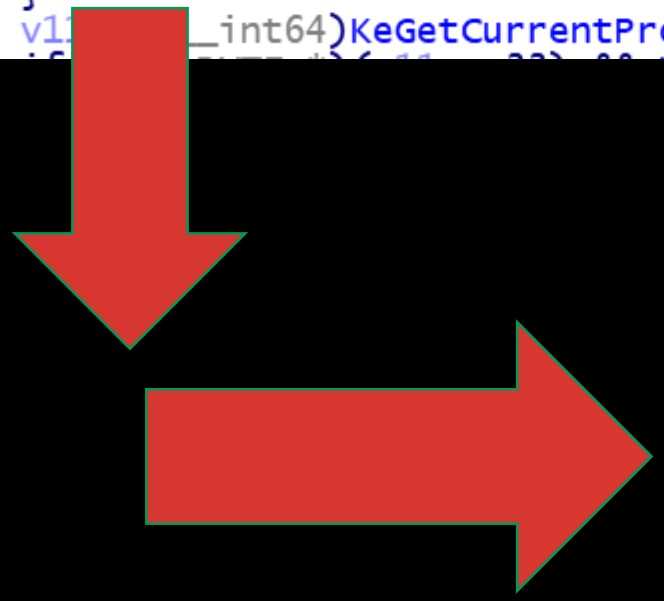
```
36  FORCEINLINE
37  VOID
38  FxPerfTraceDpc(
39      _In_ PVOID DriverCallback
40      )
41  {
42      PWMI_WDF_NOTIFY_ROUTINE perfTraceCallback = NULL;
43
44      //
45      // Trace driver's ISR using perf trace callback. If the perf trace callback
46      // is NULL, it means either perf tracing is not enabled, or this OS
47      // doesn't support perf tracing for WDF (note only win8+ supports WDF perf
48      // trace callbacks).
49      //
50      perfTraceCallback = FxLibraryGlobals.PerfTraceRoutines->DpcNotifyRoutine;
51      if (perfTraceCallback != NULL) {
52          (perfTraceCallback) (DriverCallback,            // event data
53                               sizeof(PVOID),             // sizeof event
54                               PERF_WDF_DPC,              // group mask
55                               PERFINFO_LOG_TYPE_WDF_DPC, // hook id
56                               WDF_DPC_EVENT_VERSION_2    // version
57                               );
58      }
59  }
60
61  FORCEINLINE
62  VOID
63  FxPerfTraceInterrupt(
64      _In_ PVOID DriverCallback
65      )
66  {
67      PWMI_WDF_NOTIFY_ROUTINE perfTraceCallback = NULL;
68
69      perfTraceCallback = FxLibraryGlobals.PerfTraceRoutines->InterruptNotifyRoutine
70      if (perfTraceCallback != NULL) {
71          (perfTraceCallback) (DriverCallback,            // event data
72                               sizeof(PVOID),             // sizeof event
73                               PERF_WDF_INTERRUPT,        // group mask
74                               PERFINFO_LOG_TYPE_WDF_INTERRUPT, // hook id
75                               WDF_INTERRUPT_EVENT_VERSION_2    // version
76                               );
77      }
78  }
```

WHAT DO WE GET?

- As soon as any WDF driver has an interrupt or DPC hit, our code executes first
  - At ISR/DPC IRQL, which has helpful benefits
- We really only want a ‘one-shot’ execution so we can then establish persistence in a more stealthier way now that the OS is running
  - Without triggering PatchGuard
  - Keeping this global variable hooked is easily detectable
- Question: How does PatchGuard get ‘periodic’ execution?
  - Answer: Lots of sneaky ways
- One of these ways… is as-of-yet undocumented:
  - Let’s go take a look at the HAL…

EVERY CLOCK INTERRUPT…

```
v10 = HalpClockWorkUnion;
if ( HalpClockWorkUnion && (_WORD)HalpClockWorkUnion )
{
  LOWORD(HalpClockWorkUnion) = 0;
  HalpMcaQueueDpc(v10, SHIBYTE(v10));
}
v1  _int64)KeGetCurrentPrcb();
```



```
1  void __fastcall HalpMcaQueueDpc(char LolWmiCallbackYaRight, char LolWhatIsAnMcaDpc)
2  {
3    _KPRCB *prcb; // rax
4    struct _KDPC *thisIsReservedTrustMe; // rcx
5
6    if ( LolWmiCallbackYaRight && McaWmiCallback )
7      McaWmiCallback(0x59364117i64, 1i64);
8    if ( LolWhatIsAnMcaDpc )
9    {
10     prcb = (_KPRCB *)KeQueryPrcbAddress(0i64);
11     thisIsReservedTrustMe = (struct _KDPC *)prcb->HalReserved[7];
12     if ( thisIsReservedTrustMe )
13     {
14       prcb->HalReserved[7] = 0i64;
15       KeInsertQueueDpc(
16         thisIsReservedTrustMe,
17         (PVOID)MEMORY[0xFFFFF78000000014],
18         (PVOID)(MEMORY[0xFFFFF78000000014] >> 32));
19     }
20   }
21 }
```

GRATUITOUS DPC EVERY 2 MINUTES

- Ultimately this code path is hit whenever 1200000000 100ns units have elapsed (every 2 minutes)
- Therefore, all we have to do is place an appropriate KDPC somewhere in UEFI memory with a callback to our UEFI routine… and we get code execution
  - Sure, KCFG still matters – but I'll leave that one as an 'exercise to the reader'
- A custom 'free floating' KDPC structure might be detected though (doubt it)
- What if instead of creating a custom KDPC, we pick an existing one…?
- Won't go into details here, but there are certain 'pre-existing' DPCs you can make the PRCB point to…
  - Those DPCs queue work items… based on 'pre-existing' EX_WORK_ITEM structures
  - Those work items then have various pointers to the actual 'work item callback'
- You can make the blue team have literally no idea of what is really

ACPI-BASED PERSISTENCE

I told you there was going to be LoJack

WINDOWS BINARY PLATFORM TABLE

- LoJack and other similar software legitimately has a need to persist even if you wipe Windows and reinstall or re-format your entire drive
- The way this type of software achieves this is…
  - … not something I can talk about without dropping F bombs at these vendors
- So Microsoft built a 'cleaner' way of doing this
  - While also blocking stuff like autochk.exe overwrites with "TCB Validation" as part of the Code Signing Infrastructure in CI.DLL
    - *SepIsMinTcb* stuff better left for another talk…
  - This was largely unknown to the world until Lenovo went and messed it up for everyone
  - But now it's mostly forgotten again
- tl;dr You can create an ACPI Table of type 'WBPT' and point it to a physical address and length…

WBPT INTERNALS

- Kernel will read this table during initialization and save it in memory, mapping the RAM address into virtual address space…
- SMSS uses an undocumented, one-shot system call to request this binary
- It gets written to disk as SYSTEMROOT\SYSTEM32\WPPBIN.EXE
- SMSS executes it, after doing a fairly weak (imo) signature check – any SHA1 Authenticode image with /INTEGRITYCHECK is A-OK.
- Execution is written in the Event Log (yay MSFT <3)
- Then an ACPI _PBS (Platform Binary Status) method is executed
  - Parameters are passed to it, including the exit code of the binary
  - This allows ACPI code to be aware of this stage of the boot having been reached
  - Obviously interesting for something that's 'persisting'…
- !acpitabl in WinDBG shows you if you have this table – see my SySCan 2015

WBPT

- So a cheap/free way to get Windows persistence is to just add/create the right ACPI Table without even bothering with any UEFI runtime component…
- Good thing Tiano/EDK2 makes this easy:

```
EFI_ACPI_TABLE_PROTOCOL_GUID
typedef
EFI_STATUS
(EFIAPI *EFI_ACPI_TABLE_INSTALL_ACPI_TABLE) (
  IN  EFI_ACPI_TABLE_PROTOCOL *This,
  IN  VOID                    *AcpiTableBuffer,
  IN  UINTN                   AcpiTableBufferSize,
  OUT UINTN                   *TableKey,
  );
```

- Of course this is very forensically weak – file is dropped on disk, etc (but native!)

ACTUAL ACPI-BASED PERSISTENCE

Shout Out to **LOÏC DUFLOT**

A BIT ABOUT ACPI

- ACPI contains a table called “Differentiated System Descriptor Table” (DSDT)
  - Can also have N-”Secondary System Descriptor Tables” (SSDT)
- These tables have specialized ACPI Markup Language (AML) routines written in ACPI Script Language (ASL)
- They are used whenever you
  - Touch the brightness controls in Windows
  - Check your battery status
  - Close/open your lid
  - And potentially a lot more…
- The OS parses this AML code and executes it in Ring 0 kind of like a VM…

WHAT CAN AML DO?

- AML generally speaking can access any IO port and RAM address by marking it as part of an "Operation Region" (OpRegion)
  - In ACPI 5.0, you can even do GPIO ☺
  - AML can perform arbitrary read/write to any OpRegion as part of its execution
- Windows actually has some good defenses against rogue AML code
  - But not perfect (see SyScan 2012 talk) – for example other tables can point to OS RAM
  - Windows won't allow OpRegions that point to OS RAM (w00t)
  - Windows won't allow OpRegions that target 'dangerous' IO ports (DMA/PIC/BIOS/etc)
  - VTL1/VSM also means that SLAT/NPT/EPT protects against VTL0 access of HyperVisor RAM
- Linux/Mac (and Intel Reference APICA – used in ReactOS, for example) don't have these protections

ACPI ATTACKS AGAINST UEFI

- OpRegions are actually 64-bit: But length is 32-bit. Issues with < 4GB boundary check
  - Windows ACPI parser does not check for overflow when doing "Is OS RAM?" check
  - OpRegion at 0x1000 (not OS RAM) + 0xFFFFFFFF (seen as "0" when incorrectly adding) gives full access to 0-4GB RAM region
    - Modern Windows is no longer guaranteed to be there – but UEFI always is (See Recon Brussels 2017 talk)
  - Silently fixed in Windows 10 (not backported)
- "Is OS RAM" check explicitly verifies only OS RAM pages
  - UEFI Runtime pages are "not OS"
  - ACPI tables (including the DSDT/SSDT) are "not OS"
  - Any "unknown (hidden? ☺) RAM" is "not OS"
- Can use ACPI methods to interact/patch/hook with UEFI Runtime Code… or

ACPI METHOD EXECUTION

- ACPI methods execute automatically based on certain user-level actions as shown earlier
- But you can actually call ACPI methods from user mode in Windows 8+
- There is now an \GLOBAL?? \ACPI_ROOT_OBJECT symbolic link
  - Accessible through CreateFi L"\\??\\ACPI_ROOT_OBJEC
  - Can send IOCTL_ACPI_EVAL_METHOD_EX which works with arbitrary namespace methods
- Useful: !amli dns and !nstree in WinDBG

```
->Dacl    : ->Ace[2]: ->AceType: ACCESS_ALLOWED_ACE_TYPE
->Dacl    : ->Ace[2]: ->AceFlags: 0x0
->Dacl    : ->Ace[2]: ->AceSize: 0x14
->Dacl    : ->Ace[2]: ->Mask : 0x001201bf
->Dacl    : ->Ace[2]: ->SID: S-1-1-0 (Well Known Group: localhost\Everyone)
->Sacl    :
->Sacl    : ->AclRevision: 0x2
->Sacl    : ->Sbz1       : 0x0
->Sacl    : ->AclSize    : 0x1c
->Sacl    : ->AceCount   : 0x1
->Sacl    : ->Sbz2       : 0x0
->Sacl    : ->Ace[0]: ->AceType: SYSTEM_MANDATORY_LABEL_ACE_TYPE
->Sacl    : ->Ace[0]: ->AceFlags: 0x0
->Sacl    : ->Ace[0]: ->AceSize: 0x14
->Sacl    : ->Ace[0]: ->Mask : 0x00000001
->Sacl    : ->Ace[0]: ->SID: S-1-16-4096 (Label: Mandatory Label\Low Mandato
```

ACPI <-> KERNEL COMMS

- As part of ACPI specifications, Linux and Windows (not sure about Mac) allow registering an "OpRegion Handler"
  - acpi_install_address_space_handler on Linux, **RegisterOpRegionHandler** on Windows
- This allows a Ring 0 "driver" (or any component) to receive the buffer and offset (and data) or any ACPI read/write on an OpRegion
  - Obviously can be used as a nice side channel to 'talk' to a Ring 0 persistent component from within ACPI
- See:
  - https://docs.microsoft.com/en-us/windows-hardware/drivers/acpi/implementing-an-operation-region-handler
  - https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/oprghdlr/nf-oprghdlr-registeropregionhandler

ACPI <-> KERNEL NOTIFICATIONS

- Even more undocumented is the ability for a Ring 0 driver to register for ACPI Interrupts
  - This is achieved through RegisterForDeviceNotifications as part of an internal ACPI Bus Interface
  - Must send IRP_MN_QUERY_INTERFACE for ACPI_INTERFACE_STANDARD2
  - See https://github.com/Microsoft/Windows-driver-samples/blob/master/usb/UcmCxUcsi/Acpi.cpp for a nice example ☺
- Now ACPI AML code can 'interrupt' a Ring 0 driver… how?
  - Using the 'Notify' ASL with a code > 0x80
  - Code is received by your Notification Callback, operating as an ISR
  - Your callback can of course now 'eval' any ACPI method or variable
- Thus, we can actually build an End-to-End (E2E) Channel…

END-TO-END FIRMWARE<->USER MODE

Nation States Do Like Their E2E Packages. Maybe Even Offer 3 Years Of Support?

UEFI COMPONENT

- "Somehow" you are executing in UEFI…
  - Use your "Boot Services" DXE to register the required events
  - Install/modify any ACPI Tables to load custom SSDT/patch DSDT/load WPBT/WDAT
- The event callbacks execute…
  - Allocate some UEFI Runtime Pool (to avoid requiring a runtime DXE) and make it RWX
  - Patch, if you want, some LOADER_PARAMETER_BLOCK structures…
    - Patch what Windows thinks the UEFI Runtime Callbacks are… or patch the callbacks…
  - Or don't patch anything yet… because your ACPI code will?
  - Bonus if you allocate RWX memory is that your UEFI firmware will appear 'insecure'
    - So UEFI Runtime calls will run in VTL1 – yay (more on this if we have time)

ACPI COMPONENT

- You have an OpRegion that points to 'interesting' RAM that is not OS RAM
  - Problem is each system is going to have UEFI RAM at different addresses (not due to ASLR, lol, just that firmware sizes/ordering is different)
- So you can…
  - As part of your UEFI InstallAcpiTable… patch/dynamically build the OpRegion making it point to the RAM where you're loaded or…
- Store your address somewhere in a UEFI Variable
  - Have some user-mode component (dropped through WBPT?) read the variable
  - Then execute an ACPI method passing the variable
  - Have the ACPI method overwrite the OpRegion with the value sent to it
- Then have the user-mode code execute a different ACPI Method
  - Which now has the UEFI-runtime-memory OpRegion accessible, and can patch it

ACPI COMPONENT

- Without any patching of the 'legitimate' UEFI runtime callbacks in LOADER_PARAMETER_BLOCK, you can drop a Ring 0 driver which calls the right virtual addresses that correspond to your UEFI Runtime Memory
  - These addresses can be stored in UEFI variables, or retrieved by calling ACPI methods which return them based on ACPI Objects where you stored the right address from UEFI as part of the Virtual Address Map Callback and calling ConvertPointer()
- But could you call UEFI Runtime C code… from AML?
  - Sure! Get UEFI Runtime Code persistence (as shown earlier), and read from some physical address what your 'next command' should be and 'next command context'
  - Use an OpRegion that maps 'Next Command' and 'Next Command Context'
    - Have an ACPI Method that writes arbitrary user input into that OpRegion
  - Each 'execution' of the UEFI Runtime C code acts on that 'command' and the 'context' that tells it what to do

USER MODE COMPONENT

- This can be something you drop through WBPT or through some other persistence method
  - Its job would be to read various interesting UEFI variables you've deposited, and then execute the ACPI methods passing in those interesting UEFI variables
  - Or you could just use _PBS itself which sends the return value of WppBin.exe, which you could code to return an 'interesting' variable that your _PBS routine will use as input to do something 'interesting'
- Even better, though, you can hook existing functions in the DSDT, or provide your own DSM ("Device Specific Method") that Windows calls if present, for various devices/batteries/controllers/etc
  - Now you get code execution 'for free', but the input buffer/etc probably isn't meaningful to you
  - Unless you find some ACPI routine Windows calls with input you can somehow control from a "living off the land" binary/registry key/virtual address

WMI!

- It's worth noting that Microsoft managed to create a standard where ACPI methods can be called from WMI
- Many motherboards vendors actually use this
- And if you read the WMIACPI spec, you can see how it works – and how to make your ACPI method WMI-callable
- Let that sink in, you can call ACPI methods from WMI
  - And you know you can call WMI from PowerShell
  - And JScript
  - And VBA Macros…

HYPERVISORS, VTLS AND KSR

Overflow if we made it this far

VTL1

- At a few conferences already, I've brought up VTL1 and UEFI relationships, but didn't really understand them fully (See Recon Brussels 2017)

- Here is the final understanding:
  - UEFI Runtime Code should have its code pages non-executable (NX)
  - If that's actually the case, UEFI Runtime Code runs in VTL 0 like normal

- However… if your UEFI firmware does not leverage NX, the boot loader detects this
  - And if any pages are RWX, it enables a 'protection' mechanism to enforce W^X
  - This is because, if Device Guard/HVCI is enabled, but UEFI pages are RWX, you could find them in virtual memory, overwrite them, and then get Ring 0 code execution even on a DG/HVCI system

VTL1…

- But how can we ‘protect’ against this?
  - Solution: make the pages RO/X from VTL0, and keep them RWX in VTL 1 for compatibility (using EPTE). VTL0/R0 cannot modify UEFI Firmware anymore.
  - Run firmware in VTL1 to support it having RWX permissions, while at the same time protecting it from any malicious VTL0 users
- This does make non-DEP/NX compliant firmware ‘secure’ in the world of HVCI
- But it also makes pre-boot FW attacks a lot more powerful
  - On non-compliant firmware (or firmware were your pre-boot attack makes it seem non-compliant)…
    - … you now have your UEFI Runtime operating in VTL1
  - Forget SMM attacks, this is way easier
    - TianoCore, where is your bug bounty program???
      - Kudos, Intel. Seriously. Finally.

KSR

- ~~KSR is a new facility for Kernel Sof~~
- Oh, sorry, I've run out of time.
- Nevermind.
- There is no KSR.
- Don't look into this. See you next year.

THANK YOU!

Q & A