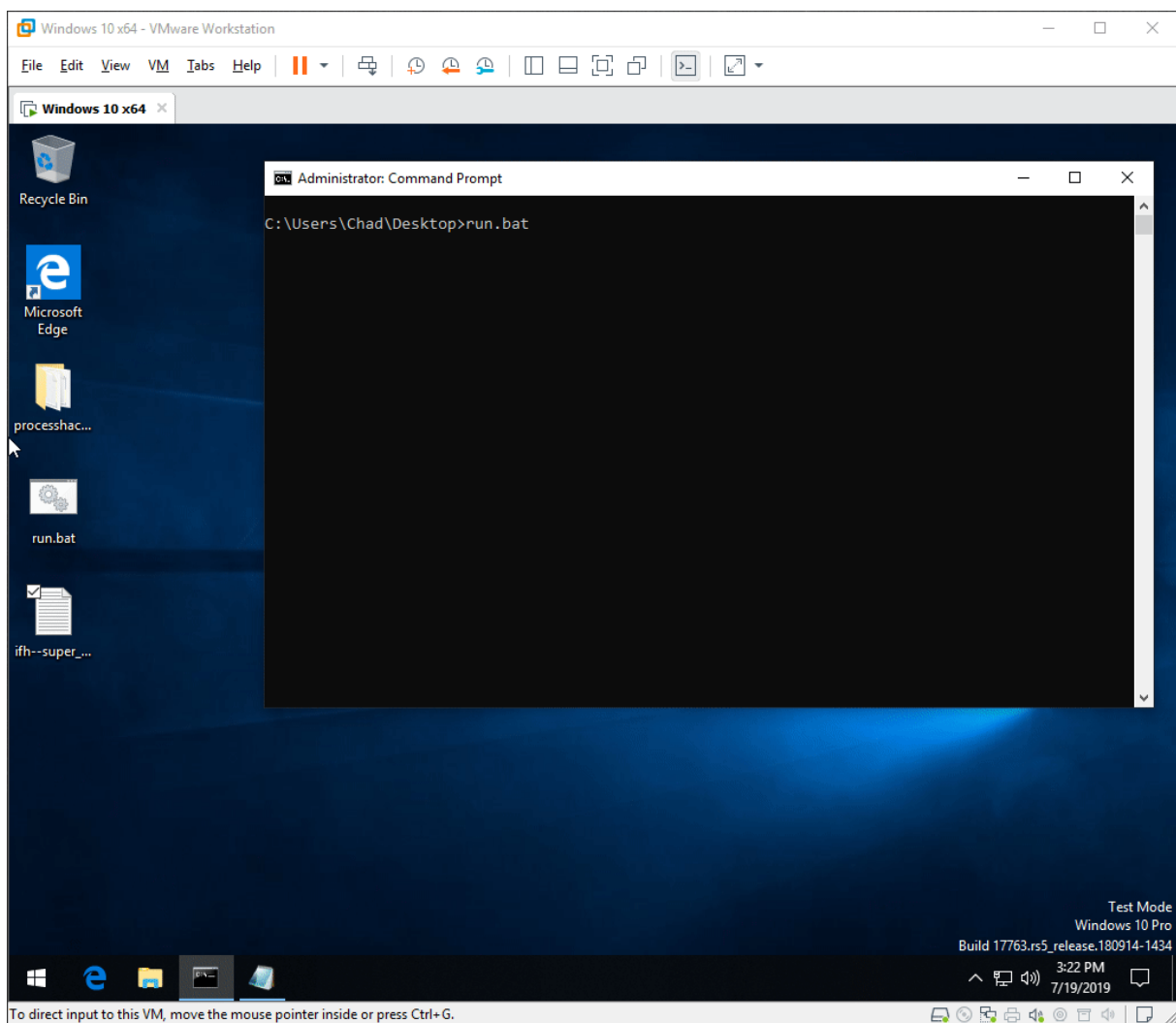

Hook system calls, context switches, page faults, DPCs and more. InfinityHook works along side Patchguard and VBS/Hypervguard to subtly hook various kernel events. InfinityHook is incredibly portable and stealthy, it works in all versions of Windows 7 to the latest versions of Windows 10.

InfinityHook stands to be one of the best tools in the rootkit arsenal over the last decade.

InfinityHook



Usage

The sample in this repository is a kernel driver that will hook system calls for you. It is extremely easy to use and requires you to call a single API. Please read below for usage instructions. We leave it upon the

reader to decipher the implementation details and create hooks for other events like context switches, page faults, and DPCs. The comments embedded in the source files can help you toward this task.

To use InfinityHook, simply reference the **libinfinityhook** library in your kernel driver and include `infinityhook.h`:

Call `IfhInitialize`. You will need to pass a function pointer to a user-defined routine:

```
1 NTSTATUS IfhInitialize(_In_ INFINITYHOOKCALLBACK InfinityHookCallback)
```

Your callback should be of this type:

```
1 typedef void (__fastcall* INFINITYHOOKCALLBACK)(_In_ unsigned int
    SystemCallIndex, _Inout_ void** SystemCallFunction);
```

Your `InfinityHookCallback` is invoked before the system executes the actual system call. The first argument passed to your callback handler is the system call index, and the second is a function pointer to the system call that is about to be invoked. You may choose to overwrite this function pointer, and the system will branch to the routine of your choosing instead. This allows you to receive all of its arguments. Ideally, you would save off the original routine pointed to by `SystemCallFunction`, so your hook can invoke the original at some point allowing you to monitor/filter the data.

How does InfinityHook actually work?

To understand InfinityHook, a little background in ETW (Event Tracing for Windows) is helpful. ETW is a construct within the Windows kernel for logging and consuming a rather enormous amount of possible events. The three main components of this are controllers, providers, and consumers. A controller typically creates and defines a trace session. A trace session consists of a name, an identifier GUID, flags about how the kernel should serialize and prepare the data for consumers, and information about what providers are enabled for that session. A controller can also manage and modify existing built-in trace sessions. The main interface for a controller to do all of the aforementioned work is through the `NtTraceControl` API.

A provider gives event data to logger sessions. It's typically through the `NtTraceEvent` API or the kernel equivalent, `EtwWrite`. Based on how the session is setup by the controller, a consumer, which is previously aware of the event data, either consumes the data in real time, a file, or perhaps occasionally from a circular buffer.

To understand more on ETW internals, please read: <https://docs.microsoft.com/en-us/windows/win32/etw/about-event-tracing>

When a session is created, it has the opportunity to collect events from `SystemTraceProvider`, instead of collecting events from registered providers. A list of these events fired by `SystemTraceProvider` can be found here: <https://docs.microsoft.com/en-us/windows/win32/etw/event-trace-properties>

It should be of note that this is not the complete list. There are plenty of undocumented ones;).

You'll probably notice that the list of items in `EnableFlags` is the same that `InfinityHook` allows you to hook. This is because each active logger session is put into an array of `WMI_LOGGER_CONTEXT` structures. They look like this:

```
1 0: kd> dt nt!_WMI_LOGGER_CONTEXT
2      +0x000 LoggerId           : Uint4B
3      +0x004 BufferSize         : Uint4B
4      +0x008 MaximumEventSize   : Uint4B
5      +0x00c LoggerMode         : Uint4B
6      +0x010 AcceptNewEvents     : Int4B
7      +0x014 EventMarker        : [2] Uint4B
8      +0x01c ErrorMarker        : Uint4B
9      +0x020 SizeMask           : Uint4B
10     +0x028 GetCpuClock         : Ptr64      int64
11     +0x030 LoggerThread        : Ptr64      _ETHREAD
12     +0x038 LoggerStatus        : Int4B
13     +0x03c FailureReason       : Uint4B
14     +0x040 BufferQueue         : _ETW_BUFFER_QUEUE
15     +0x050 OverflowQueue       : _ETW_BUFFER_QUEUE
16     +0x060 GlobalList          : _LIST_ENTRY
17     +0x070 DebugIdTrackingList : _LIST_ENTRY
18     +0x080 DecodeControlList   : Ptr64      _ETW_DECODE_CONTROL_ENTRY
19     +0x088 DecodeControlCount  : Uint4B
20     +0x090 BatchedBufferList   : Ptr64      _WMI_BUFFER_HEADER
21     +0x090 CurrentBuffer       : _EX_FAST_REF
22     +0x098 LoggerName          : _UNICODE_STRING
23     +0x0a8 LogFileName         : _UNICODE_STRING
24     +0x0b8 LogFilePattern      : _UNICODE_STRING
25     +0x0c8 NewLogFileName      : _UNICODE_STRING
26     +0x0d8 ClockType           : Uint4B
27     +0x0dc LastFlushedBuffer   : Uint4B
28     +0x0e0 FlushTimer          : Uint4B
29     +0x0e4 FlushThreshold      : Uint4B
30     +0x0e8 ByteOffset          : _LARGE_INTEGER
31     +0x0f0 MinimumBuffers      : Uint4B
32     +0x0f4 BuffersAvailable    : Int4B
33     +0x0f8 NumberOfBuffers     : Int4B
34     +0x0fc MaximumBuffers      : Uint4B
35     +0x100 EventsLost          : Uint4B
36     +0x104 PeakBuffersCount    : Int4B
37     +0x108 BuffersWritten      : Uint4B
38     +0x10c LogBuffersLost      : Uint4B
39     +0x110 RealTimeBuffersDelivered : Uint4B
40     +0x114 RealTimeBuffersLost : Uint4B
```

```

41      +0x118 SequencePtr      : Ptr64 Int4B
42      +0x120 LocalSequence    : Uint4B
43      +0x124 InstanceGuid     : _GUID
44      +0x134 MaximumFileSize  : Uint4B
45      +0x138 FileCounter      : Int4B
46      +0x13c PoolType         : _POOL_TYPE
47      +0x140 ReferenceTime    : _ETW_REF_CLOCK
48      +0x150 CollectionOn     : Int4B
49      +0x154 ProviderInfoSize : Uint4B
50      +0x158 Consumers        : _LIST_ENTRY
51      +0x168 NumConsumers     : Uint4B
52      +0x170 TransitionConsumer : Ptr64 _ETW_REALTIME_CONSUMER
53      +0x178 RealtimeLogfileHandle : Ptr64 Void
54      +0x180 RealtimeLogfileName : _UNICODE_STRING
55      +0x190 RealtimeWriteOffset : _LARGE_INTEGER
56      +0x198 RealtimeReadOffset : _LARGE_INTEGER
57      +0x1a0 RealtimeLogfileSize : _LARGE_INTEGER
58      +0x1a8 RealtimeLogfileUsage : Uint8B
59      +0x1b0 RealtimeMaximumFileSize : Uint8B
60      +0x1b8 RealtimeBuffersSaved : Uint4B
61      +0x1c0 RealtimeReferenceTime : _ETW_REF_CLOCK
62      +0x1d0 NewRTEventsLost : _ETW_RT_EVENT_LOSS
63      +0x1d8 LoggerEvent      : _KEVENT
64      +0x1f0 FlushEvent       : _KEVENT
65      +0x208 FlushTimeOutTimer : _KTIMER
66      +0x248 LoggerDpc        : _KDPC
67      +0x288 LoggerMutex      : _KMUTANT
68      +0x2c0 LoggerLock       : _EX_PUSH_LOCK
69      +0x2c8 BufferListSpinLock : Uint8B
70      +0x2c8 BufferListPushLock : _EX_PUSH_LOCK
71      +0x2d0 ClientSecurityContext : _SECURITY_CLIENT_CONTEXT
72      +0x318 TokenAccessInformation : Ptr64 _TOKEN_ACCESS_INFORMATION
73      +0x320 SecurityDescriptor : _EX_FAST_REF
74      +0x328 StartTime        : _LARGE_INTEGER
75      +0x330 LogFileHandle     : Ptr64 Void
76      +0x338 BufferSequenceNumber : Int8B
77      +0x340 Flags             : Uint4B
78      +0x340 Persistent        : Pos 0, 1 Bit
79      +0x340 AutoLogger        : Pos 1, 1 Bit
80      +0x340 FsReady           : Pos 2, 1 Bit
81      +0x340 RealTime          : Pos 3, 1 Bit
82      +0x340 Wow               : Pos 4, 1 Bit
83      +0x340 KernelTrace       : Pos 5, 1 Bit
84      +0x340 NoMoreEnable      : Pos 6, 1 Bit
85      +0x340 StackTracing      : Pos 7, 1 Bit
86      +0x340 ErrorLogged       : Pos 8, 1 Bit
87      +0x340 RealtimeLoggerContextFreed : Pos 9, 1 Bit
88      +0x340 PebsTracing       : Pos 10, 1 Bit
89      +0x340 PmcCounters       : Pos 11, 1 Bit
90      +0x340 PageAlignBuffers  : Pos 12, 1 Bit
91      +0x340 StackLookasideListAllocated : Pos 13, 1 Bit

```

```

92     +0x340 SecurityTrace      : Pos 14, 1 Bit
93     +0x340 LastBranchTracing : Pos 15, 1 Bit
94     +0x340 SystemLoggerIndex : Pos 16, 8 Bits
95     +0x340 StackCaching      : Pos 24, 1 Bit
96     +0x340 ProviderTracking  : Pos 25, 1 Bit
97     +0x340 ProcessorTrace    : Pos 26, 1 Bit
98     +0x340 QpcDeltaTracking   : Pos 27, 1 Bit
99     +0x340 SpareFlags2       : Pos 28, 4 Bits
100    +0x344 RequestFlag        : Uint4B
101    +0x344 DbgRequestNewFile   : Pos 0, 1 Bit
102    +0x344 DbgRequestUpdateFile : Pos 1, 1 Bit
103    +0x344 DbgRequestFlush     : Pos 2, 1 Bit
104    +0x344 DbgRequestDisableRealtime : Pos 3, 1 Bit
105    +0x344 DbgRequestDisconnectConsumer : Pos 4, 1 Bit
106    +0x344 DbgRequestConnectConsumer : Pos 5, 1 Bit
107    +0x344 DbgRequestNotifyConsumer : Pos 6, 1 Bit
108    +0x344 DbgRequestUpdateHeader : Pos 7, 1 Bit
109    +0x344 DbgRequestDeferredFlush : Pos 8, 1 Bit
110    +0x344 DbgRequestDeferredFlushTimer : Pos 9, 1 Bit
111    +0x344 DbgRequestFlushTimer : Pos 10, 1 Bit
112    +0x344 DbgRequestUpdateDebugger : Pos 11, 1 Bit
113    +0x344 DbgSpareRequestFlags : Pos 12, 20 Bits
114    +0x350 StackTraceBlock     : _ETW_STACK_TRACE_BLOCK
115    +0x3d0 HookIdMap           : _RTL_BITMAP
116    +0x3e0 StackCache          : Ptr64 _ETW_STACK_CACHE
117    +0x3e8 PmcData             : Ptr64 _ETW_PMC_SUPPORT
118    +0x3f0 LbrData             : Ptr64 _ETW_LBR_SUPPORT
119    +0x3f8 IptData             : Ptr64 _ETW_IPT_SUPPORT
120    +0x400 BinaryTrackingList  : _LIST_ENTRY
121    +0x410 ScratchArray        : Ptr64 Ptr64 _WMI_BUFFER_HEADER
122    +0x418 DisallowedGuids     : _DISALLOWED_GUIDS
123    +0x428 RelativeTimerDueTime : Int8B
124    +0x430 PeriodicCaptureStateGuids : _PERIODIC_CAPTURE_STATE_GUIDS
125    +0x440 PeriodicCaptureStateTimer : Ptr64 _EX_TIMER
126    +0x448 PeriodicCaptureStateTimerState : _ETW_PERIODIC_TIMER_STATE
127    +0x450 SoftRestartContext  : Ptr64 _ETW_SOFT_RESTART_CONTEXT
128    +0x458 SiloState           : Ptr64 _ETW_SILODRIVERSTATE
129    +0x460 CompressionWorkItem : _WORK_QUEUE_ITEM
130    +0x480 CompressionWorkItemState : Int4B
131    +0x488 CompressionLock     : _EX_PUSH_LOCK
132    +0x490 CompressionTarget   : Ptr64 _WMI_BUFFER_HEADER
133    +0x498 CompressionWorkspace : Ptr64 Void
134    +0x4a0 CompressionOn       : Int4B
135    +0x4a4 CompressionRatioGuess : Uint4B
136    +0x4a8 PartialBufferCompressionLevel : Uint4B
137    +0x4ac CompressionResumptionMode : ETW_COMPRESSION_RESUMPTION_MODE
138    +0x4b0 PlaceholderList     : _SINGLE_LIST_ENTRY
139    +0x4b8 CompressionDpc      : _KDPC
140    +0x4f8 LastBufferSwitchTime : _LARGE_INTEGER
141    +0x500 BufferWriteDuration  : _LARGE_INTEGER
142    +0x508 BufferCompressDuration : _LARGE_INTEGER

```

```
143 +0x510 ReferenceQpcDelta : Int8B
144 +0x518 CallbackContext  : Ptr64 _ETW_EVENT_CALLBACK_CONTEXT
```

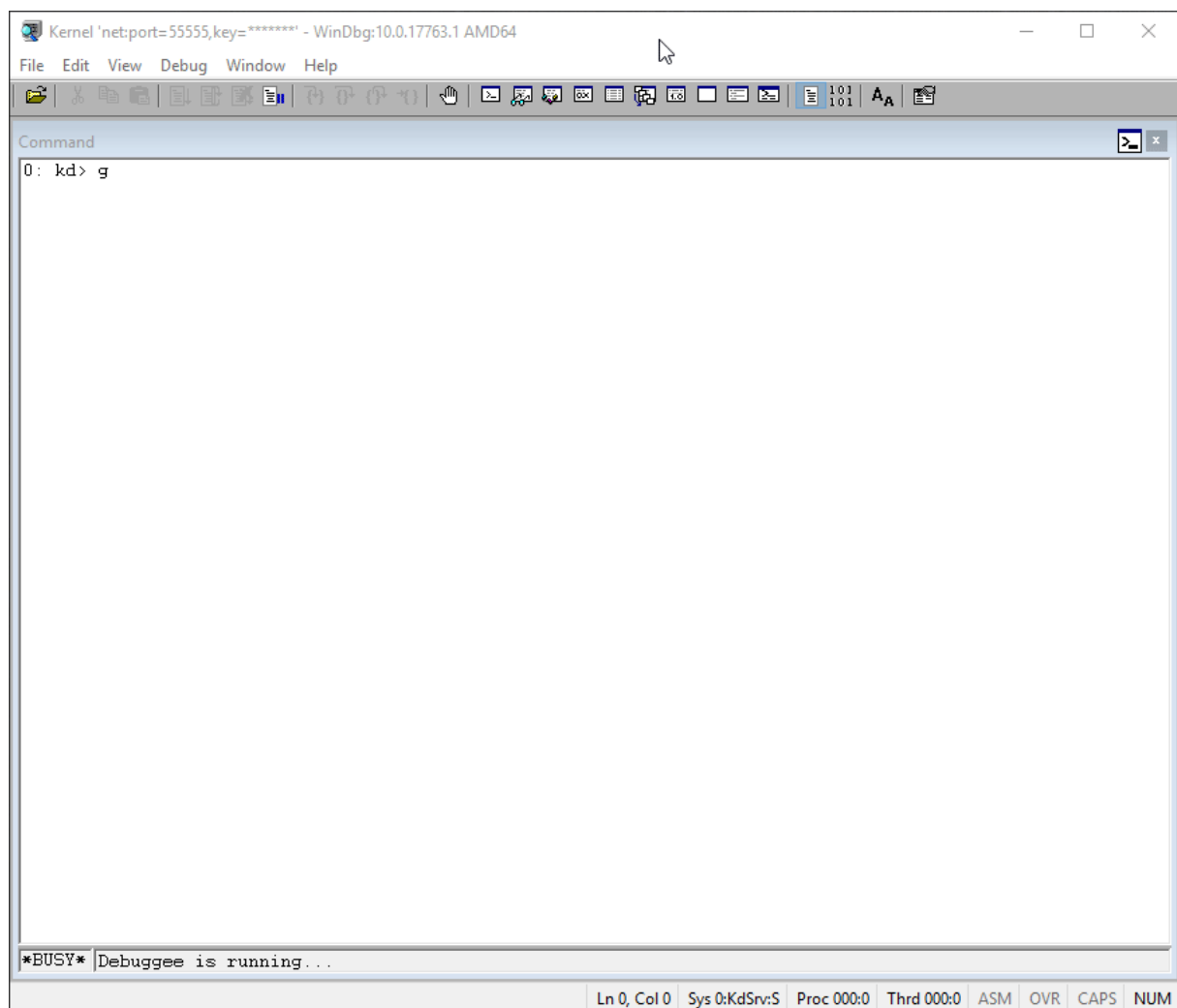
Although not exported, this array is easily resolvable because a pointer to it exists right after `EtwDebuggerData`, which interestingly enough can be signature scanned for Windows 7, 8, 8.1, and all the existing versions of Windows 10, using just a 5 byte signature: `0x2c`, `0x08`, `0x04`, `0x38`, `0x0c`.

At `+0x28` in the `_WMI_LOGGER_CONTEXT` structure, you can see a member called `GetCpuClock`. This is a function pointer that can be one of three values based on how the session was configured: `EtwGetCycleCount`, `EtwGetSystemTime`, or `PpmQueryTime`. We simply overwrite this function pointer with a custom routine, but this is only half the battle.

First, we choose to hijack the circular kernel context logger session because it's always running by default. If not, we turn it on, and we configure it to log syscalls only, in a circular memory buffer.

After this, we walk up the stack to locate magic values, in order to filter out the fact that this is not a syscall exit being logged. We grab `SystemCallNumber` saved into the current `_KTHREAD` from logic in `KiSystemCall64`. The real magic here occurs because prior to `KiSystemCall64` invoking `PerfInfoLogSyscallEntry`, it saves the resolved system call target pointer on the stack. We locate this pointer for you and, if you so choose, you are able to overwrite it in your handler.

```
loc_1401776AB: ; CODE XREF: KiSystemCall64+2CA↑j
    sub     rsp, 50h
    mov     [rsp+138h+var_118], rcx
    mov     [rsp+138h+var_110], rdx
    mov     [rsp+138h+var_108], r8
    mov     [rsp+138h+var_100], r9
    mov     [rsp+138h+var_F8], r10 ←
    mov     rcx, r10
    call    PerfInfoLogSyscallEntry
    mov     rcx, [rsp+138h+var_118]
    mov     rdx, [rsp+138h+var_110]
    mov     r8, [rsp+138h+var_108]
    mov     r9, [rsp+138h+var_100]
    ← mov     r10, [rsp+138h+var_F8]
    add     rsp, 50h
    ← call    r10
    mov     [rbp-50h], rax
    mov     rcx, rax
    call    PerfInfoLogSyscallExit
    mov     rax, [rbp-50h]
    jmp     loc_140177313
```



The sample code provided is for system calls only, and as mentioned above, it's up to the reader to implement it for other events. This sample was also only quickly whipped up and tested for 1903 and 1803. The stack walk function may need to be tweaked for earlier Windows 10 builds and 7.