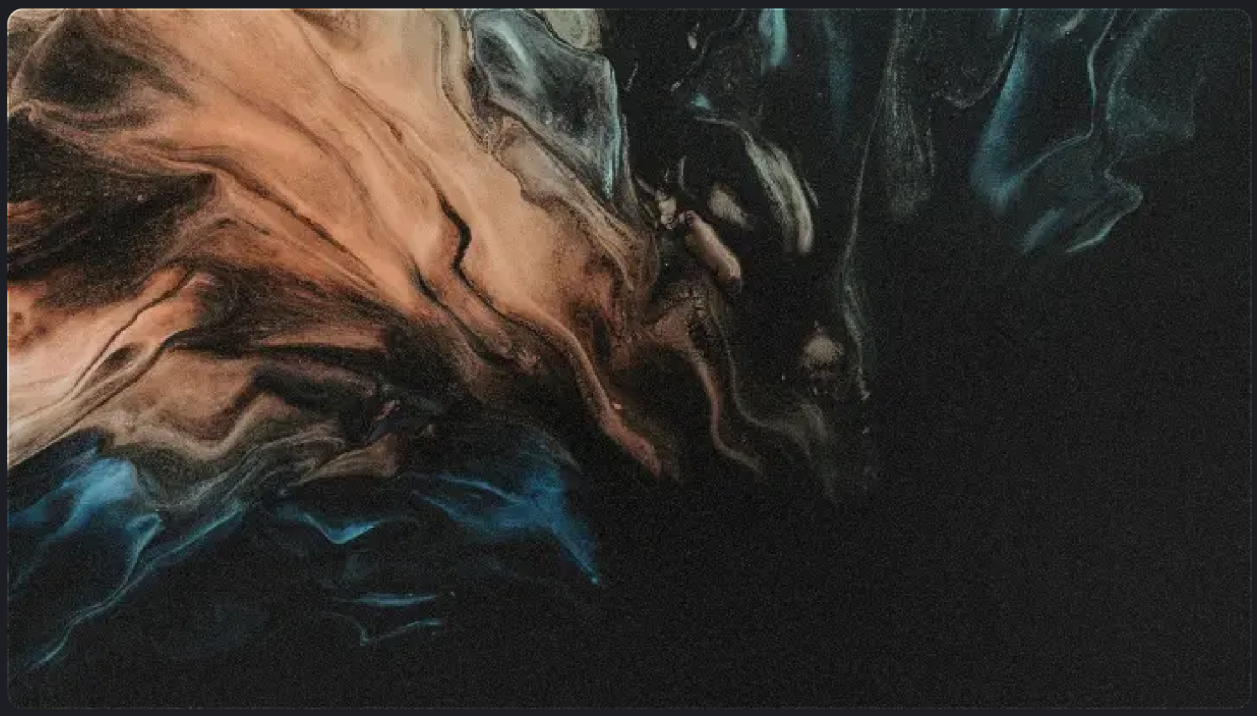
7 DECEMBER 2022

JOHN UHLMANN

Get- InjectedThreadEx – Detecting Thread Creation Trampolines

In this blog, we will demonstrate how to detect each of four classes of process trampolining and release an updated PowerShell detection script – Get-InjectedThreadEx

19 min read Security research

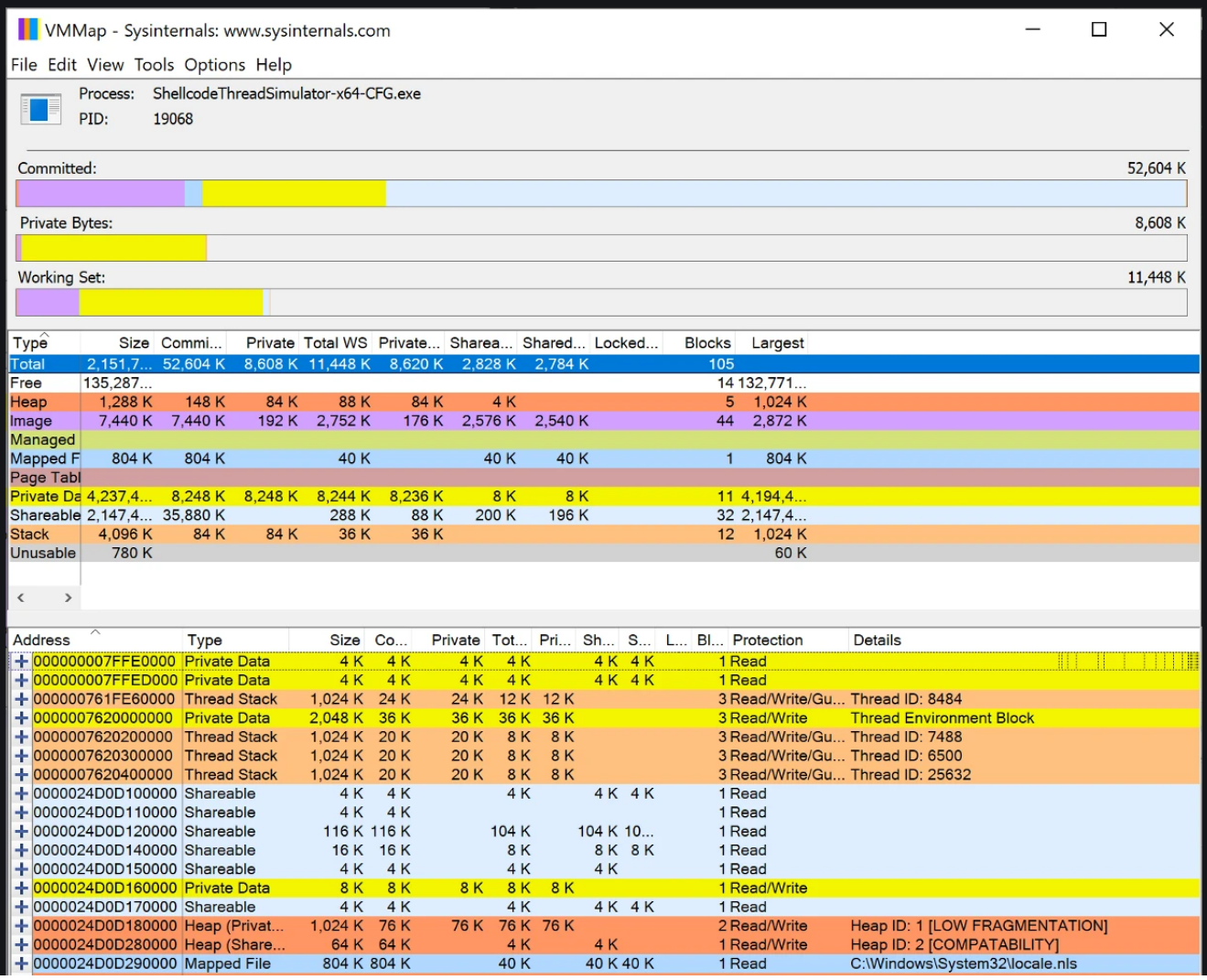


The prevalence of **memory resident malware** remains extremely high. Defenders have imposed significant costs on file-based techniques, and malware must typically utilize **in-memory techniques** to avoid detection. In Elastic's recently-published **Global Threat Report**,

defense evasion is the most diverse tactic we observed and represents an area of rapid, continuous innovation.

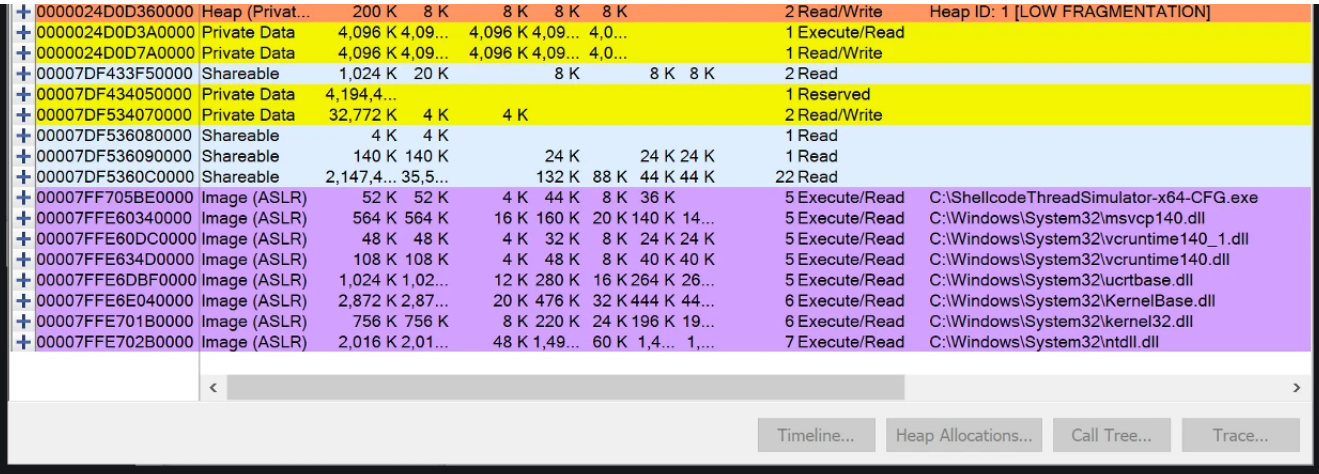
It is convenient, and sometimes necessary, for memory-resident malware to create its own threads within its surrogate process. Many such threads can be detected with relatively low noise by identifying those which have a start address not backed by a Portable Executable (PE) image file on disk. This detection technique was originally conceived by Elastic's **Gabriel Landau** and Nicholas Fritts for the Elastic Endgame product. Shortly thereafter, it was released as a PowerShell script for the benefit of the community in the form of **Get-InjectedThread** with the help of **Jared Atkinson** and Elastic's **Joe Desimone** at the **2017 SANS Threat Hunting and IR Summit**.

At a high level, this approach detects threads created with a user start address in unbacked executable memory. Unbacked executable memory itself is quite normal in many processes such as those that do just-in-time (JIT) compilation of bytecode or scripts like .NET or javascript. However, that JIT'd code rarely manages its own threads – usually that is handled by the runtime or engine.



Type	Size	Commi...	Private	Total WS	Private...	Sharea...	Shared...	Locked...	Blocks	Largest
Total	2,151,7...	52,604 K	8,608 K	11,448 K	8,620 K	2,828 K	2,784 K		105	
Free	135,287...								14	132,771...
Heap	1,288 K	148 K	84 K	88 K	84 K	4 K			5	1,024 K
Image	7,440 K	7,440 K	192 K	2,752 K	176 K	2,576 K	2,540 K		44	2,872 K
Managed										
Mapped F	804 K	804 K		40 K		40 K	40 K		1	804 K
Page Tabl										
Private Da	4,237,4...	8,248 K	8,248 K	8,244 K	8,236 K	8 K	8 K		11	4,194,4...
Shareable	2,147,4...	35,880 K		288 K	88 K	200 K	196 K		32	2,147,4...
Stack	4,096 K	84 K	84 K	36 K	36 K				12	1,024 K
Unusable	780 K									60 K

Address	Type	Size	Co...	Private	Tot...	Pri...	Sh...	S...	L...	Bl...	Protection	Details
000000007FFE0000	Private Data	4 K	4 K	4 K	4 K		4 K	4 K		1	Read	
000000007FFED000	Private Data	4 K	4 K	4 K	4 K		4 K	4 K		1	Read	
000000761FE60000	Thread Stack	1,024 K	24 K	24 K	12 K	12 K				3	Read/Write/Gu...	Thread ID: 8484
0000007620000000	Private Data	2,048 K	36 K	36 K	36 K	36 K				3	Read/Write	Thread Environment Block
0000007620200000	Thread Stack	1,024 K	20 K	20 K	8 K	8 K				3	Read/Write/Gu...	Thread ID: 7488
0000007620300000	Thread Stack	1,024 K	20 K	20 K	8 K	8 K				3	Read/Write/Gu...	Thread ID: 6500
0000007620400000	Thread Stack	1,024 K	20 K	20 K	8 K	8 K				3	Read/Write/Gu...	Thread ID: 25632
0000024D0D100000	Shareable	4 K	4 K		4 K		4 K	4 K		1	Read	
0000024D0D110000	Shareable	4 K	4 K							1	Read	
0000024D0D120000	Shareable	116 K	116 K		104 K		104 K	10...		1	Read	
0000024D0D140000	Shareable	16 K	16 K		8 K		8 K	8 K		1	Read	
0000024D0D150000	Shareable	4 K	4 K		4 K		4 K			1	Read	
0000024D0D160000	Private Data	8 K	8 K	8 K	8 K	8 K				1	Read/Write	
0000024D0D170000	Shareable	4 K	4 K		4 K		4 K	4 K		1	Read	
0000024D0D180000	Heap (Privat...	1,024 K	76 K	76 K	76 K	76 K				2	Read/Write	Heap ID: 1 [LOW FRAGMENTATION]
0000024D0D280000	Heap (Share...	64 K	64 K		4 K		4 K			1	Read/Write	Heap ID: 2 [COMPATABILITY]
0000024D0D290000	Mapped File	804 K	804 K		40 K		40 K	40 K		1	Read	C:\Windows\System32\locale.nls



Virtual Memory layout of a simple process using Sysinternal's VMMap. Purple regions are image-backed and it is normal for threads to start there.

However, an adversary often has sufficient control to create a thread with an image-backed start address which will subsequently transfer execution to their unbacked memory. When this transfer is done immediately, it is known as a "trampoline" as you are quickly catapulted somewhere else.

There are four broad classes of trampolines – you can build your own from scratch, you can use an illusionary trampoline, you can repurpose something else as a trampoline, or you can simply find an existing trampoline.

In other words - hooks, hijacks, gadgets and functions.

Each of these will bypass our original unbacked executable memory heuristic.

I highly recommend these two excellent blogs as background:

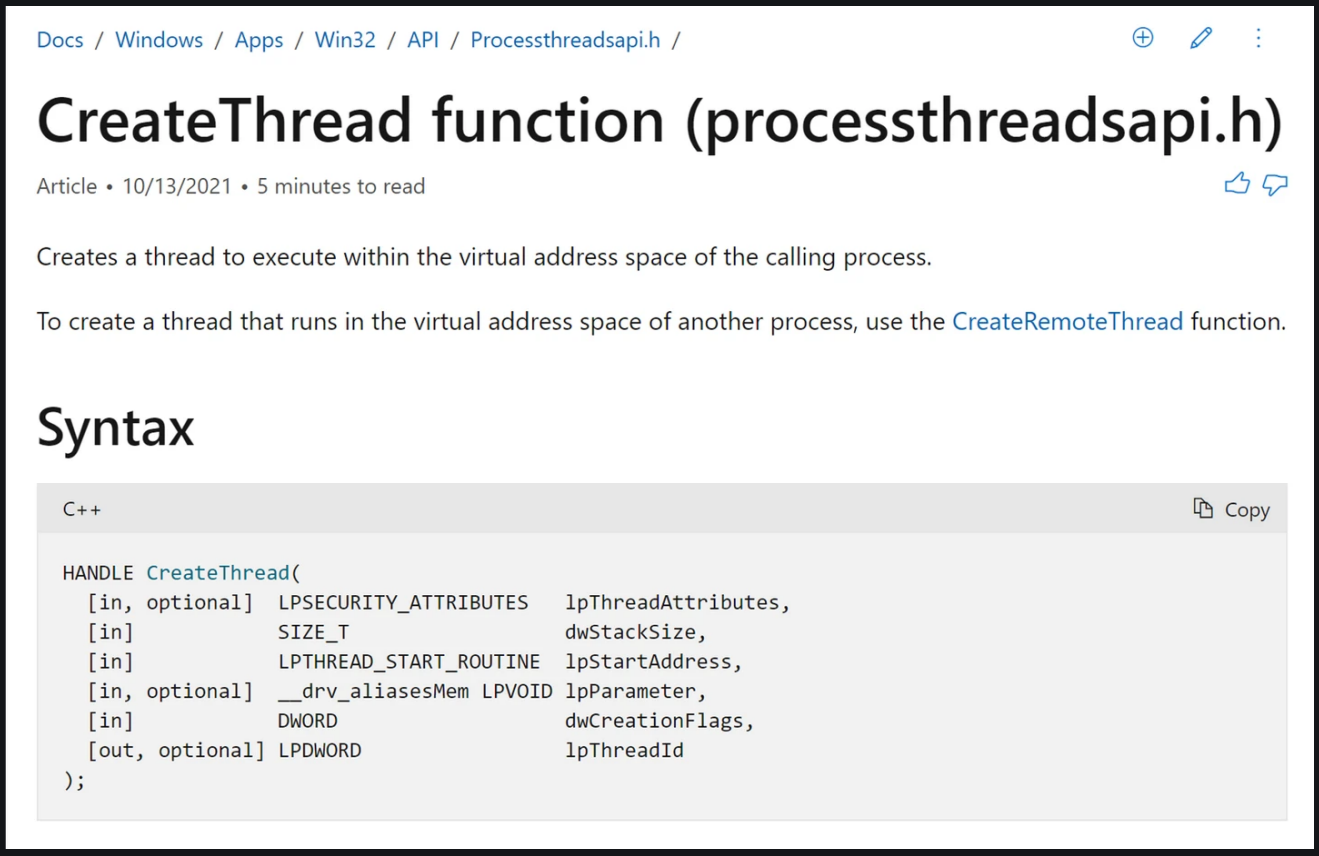
- **Understanding and Evading Get-InjectedThread** by Adam Chester.
- **Avoiding Get-InjectedThread for Internal Thread Creation** by Christopher Paschen.

In this blog, we will demonstrate how to detect each of these classes of bypass and release an updated PowerShell detection script – **Get-InjectedThreadEx**.

CreateThread() overview

As a quick recap, the Win32 **CreateThread()** API lets you specify a pointer to a desired StartAddress which will be used as the entrypoint of a function that takes exactly one user-

provided parameter.



Docs / Windows / Apps / Win32 / API / Processthreadsapi.h /

CreateThread function (processthreadsapi.h)

Article • 10/13/2021 • 5 minutes to read

Creates a thread to execute within the virtual address space of the calling process.

To create a thread that runs in the virtual address space of another process, use the CreateRemoteThread function.

Syntax

C++ Copy

```
HANDLE CreateThread(
  [in, optional]  LPSECURITY_ATTRIBUTES   lpThreadAttributes,
  [in]            SIZE_T                  dwStackSize,
  [in]            LPTHREAD_START_ROUTINE  lpStartAddress,
  [in, optional]  __drv_aliasesMem LPVOID lpParameter,
  [in]            DWORD                   dwCreationFlags,
  [out, optional] LPDWORD                 lpThreadId
);
```

Microsoft documentation for the CreateThread API

So, CreateThread() is effectively a simple shellcode runner.

```
void RunShellcode(_In_ LPVOID pShellcode)
{
    CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)pShellcode, NULL, 0, NULL);
}
```

CreateThread == RunShellcode

And its sibling, **CreateRemoteThread()** is effectively remote process injection.

The value of the lpStartAddress parameter is stored by the kernel in the Win32StartAddress field within the **ETHREAD** structure for that thread.

```
lkd> dt nt!_ETHREAD ffff95034114f080
   +0x000 Tcb              : _KTHREAD
   +0x430 CreateTime       : _LARGE_INTEGER 0x01d8bb67`78dad9f5
   +0x438 ExitTime         : _LARGE_INTEGER 0xffff9503`4114f4b8
   +0x438 KeyedWaitChain   : _LIST_ENTRY [ 0xffff9503`4114f4b8 - 0xffff9503`4114f4b8 ]
   +0x448 PostBlockList    : _LIST_ENTRY [ 0x00000000`00000000 - 0x00007ffd`eb402630 ]
   +0x448 ForwardLinkShadow : (null)
   +0x450 StartAddress     : 0x00007ffd`eb402630 Void ntdll!RtlUserThreadStart
   +0x458 TerminationPort  : (null)
```

```
+0x458 ReaperLink       : (null)
+0x458 KeyedWaitValue   : (null)
+0x460 ActiveTimerListLock : 0
+0x468 ActiveTimerListHead : _LIST_ENTRY [ 0xffff9503`4114f4e8 - 0xffff9503`4114f4e8 ]
+0x478 Cid              : _CLIENT_ID
+0x488 KeyedWaitSemaphore : _KSEMAPHORE
+0x488 AlpcWaitSemaphore : _KSEMAPHORE
+0x4a8 ClientSecurity   : _PS_CLIENT_SECURITY_CONTEXT
+0x4b0 IrpList          : _LIST_ENTRY [ 0xffff9503`4114f530 - 0xffff9503`4114f530 ]
+0x4c0 TopLevelIrp      : 0
+0x4c8 DeviceToVerify   : (null)
+0x4d0 Win32StartAddress : 0x00000001`80001000 Void  unbacked executable memory
+0x4d8 ChargeOnlySession : (null)
+0x4e0 LegacyPowerObject : (null)
+0x4e8 ThreadListEntry  : _LIST_ENTRY [ 0xffff9503`40c56568 - 0xffff9503`4108e568 ]
+0x4f8 RundownProtect   : _EX_RUNDOWN_REF
+0x500 ThreadLock       : _EX_PUSH_LOCK
+0x508 ReadClusterSize  : 7
+0x50c MmLockOrdering   : 0n0
+0x510 CrossThreadFlags : 0x5442
```

Suspicious ETHREAD entry viewed with a kernel debugger

This value can be queried from user mode using the documented **NtQueryInformationThread()** syscall with the ThreadQuerySetWin32StartAddress information class. A subsequent call to **VirtualQueryEx()** can be used to make a second syscall requesting the **basic memory information** for that virtual address from the kernel. This includes an enumeration indicating whether the memory is a mapped PE image, a mapped file, or simply private memory.

Original detection logic

While the original script was a point-in-time retrospective detection implementation, the same information is available inline during **create thread notify** kernel callbacks. All effective Endpoint Detection and Response (EDR) products should be providing telemetry of suspicious thread creations.

And all effective Endpoint Protection Platform (EPP) products should be denying suspicious thread creations by default – with a mechanism to add allowlist entries for legitimate software exhibiting this behavior.

In the wild, you'll see "legitimate" instances of this behavior such as from other security

products, anti-cheat software, older copy-protection software and some Unix products that have been shimmed to work on Windows. Though, in each instance, this security **code smell** may be indicative of software that you might not want in an enterprise environment. The use of these methods may be a leading indicator that other **security best practices** have not been followed. Even with this finite set of exceptions to handle, this detection and/or prevention approach remains highly relevant and successful today.

1 - Bring your own trampoline

The simplest trampoline is a small hook. The adversary only needs to write the necessary jump instruction into existing image-backed memory. This is the approach that Filip Olszak used to bypass Get-InjectedThread with **DripLoader**.

These bytes can even be restored to their original values immediately after thread creation. This helps to avoid retrospective detections such as our script – but recall that your endpoint security product should be doing *inline* detection and will be able to scrutinize the hooked thread entrypoint at execution time, and deny execution if necessary.

Basic hook trampoline

The above proof-of-concept hooks ntdll!DbgUiRemoteBreakin, which is a legitimate remote thread start address, though it should rarely be seen in production environments. In practice, the hook can be placed on any function bytes unlikely to be called in normal operation– or even slack space between functions, or at the end of the PE section.

Also note the use of WriteProcessMemory() instead of a simple memcpy(). MEM_IMAGE pages are typically read only, and the former handles toggling the page protections to writable and back for us.

We can detect hooked start addresses fairly easily because we can detect persistent inline

hooks fairly easily. In order to save memory, allocations for shared libraries use the same backing physical memory pages and are marked COPY_ON_WRITE in each process's address space. So, as soon as the hook is inserted, the whole page can no longer be shared. Instead, a copy is created in the working set of the process.

Using the **QueryWorkingSetEx()** API, we can query the kernel to determine whether the page containing the start address is sharable or is in a private working set.

Now we know that something on the page was modified – but we don't know if our address was hooked. And, for our updated PowerShell script, this is all that we do. Recall that the bytes can be unhooked after the thread has started– so any further checks on already running threads could result in a false negative.

However, this could also be a false positive if there is a "legitimate" hook or other modification.

In particular, many, many security products still hook ntdll.dll. This was an entirely legitimate technical approach back in 2007 when Vista was released: it allowed existing x86 features based on kernel syscall hooks to be quickly ported to the nascent x64 architecture using user mode syscall hooks instead. The validity of such approaches has been more questionable since Windows 10 was released in 2015. Around this time, x64 was cemented as the primary Windows architecture and we could firmly relegate the less secure x86 Windows to legacy status. The value proposition for user mode hooking was further reduced in 2017 when Windows 10 Creators Update **added additional kernel mode instrumentation** to provide more robust detection approaches for malicious usage of certain abused syscalls.

For reference, our original Elastic Endgame product has features implemented using user mode hooks whereas our newer Elastic Endpoint has not yet determined a need to use a user mode hook at all in order to attain equal or better protection compared to Endgame. This means that Elastic Endgame must defend these hooks from tampering whereas Elastic Endpoint is currently invulnerable to the various so-called "universal EDR bypasses" that perform ntdll.dll unhooking.

Older security products aside, there are also many products that extend the functionality of other products via hooks– or perhaps unpack their code at runtime, etc. So, if that 4KB page is private, then security products need to additionally compare the start address bytes to an original pristine copy and alert if they differ.

And, to deploy at scale, they also need to maintain an allowlist for those rare legitimate uses.

2 - Shifting the trampoline mat

Technically the security product will only be able to see the bytes at the time of the thread notification callback which is slightly before the thread executes. Malware could create a **suspended** thread, let the thread callback execute, and only then hook the start bytes before finally resuming the thread. Don't worry though - effective security products can detect that inline too. But that's a topic for another day.

This brings us to the second trampoline approach though: hijacking the execution flow before the entrypoint is ever called. Why obviously hook the thread entrypoint of our suspended thread when, with a little sleight of hand, we can usurp execution by modifying its instruction pointer directly (or an equivalent context manipulation) with **SetThreadContext()**, or by queuing an **"early bird" Asynchronous Procedure Call** (APC)?

The problem with creating the illusion of a legitimate entrypoint like this is that it doesn't hold up to any kind of rigorous inspection.

In a normal thread, the user mode start address is typically the third function call in the thread's stack – after ntdll!RtlUserThreadStart and kernel32!BaseThreadInitThunk. So when the thread has been hijacked, this is going to be obvious in the call stack.

For instruction pointer manipulation, the first frame will belong to the injected code.

For "early bird" APC injection, the base of the call stack will be ntdll!LdrInitializeThunk, ntdll!NtTestAlert, ntdll!KiUserApcDispatcher and then the injected code.

The updated script detects various anomalous call stack bases.

False positives are possible where legitimate software finds it necessary to modify Windows process or thread initialisation. For example, this was observed with the **MSYS2** Linux environment. There is also an edge case where a function might have been generated with a **Tail Call Optimisation** (TCO), which eliminates unnecessary stack frames for performance. However, these cases can all be easily handled with a small exception list.

3 - If it walks like a trampoline, and it talks like a trampoline...

The third trampoline approach is to find a suitable gadget within image-backed memory so

that no code modification is necessary. This is one of the approaches that Adam Chester employed in his blog.

Our earlier hook was 12 bytes and finding an exact 12-byte gadget is unlikely in practice.

However, on x64 Windows, functions use a four-register fast-call calling convention by default. So when the OS calls our gadget we will have control over the RCX register which will contain the parameter we passed into CreateThread().

The simplest x64 gadget is the two-byte JMP RCX instruction "ff e1" – which is fairly trivial to find.

JMP RCX gadget in ntdll.dll

Gadgets don't even need to be instructions per se – they could be within operands or other data in the code section. For example, the above "ff e1" gadget in ntdll.dll was part of the relative address of a GUID.

We can detect this too- because it doesn't work generically yet.

In all modern Windows software, thread start addresses are protected by Control Flow Guard (CFG) which has a bitmap of valid indirect call targets computed at compile time. In order to use this gadget, malware must either first disable CFG or call the **SetProcessValidCallTargets()** function to ask the kernel to dynamically set the bit corresponding to this gadget in the CFG bitmap.

Just to be clear: this is not a CFG bypass. It is a CFG feature to support legitimate software doing weird things. Remember that CFG is an exploit protection– and being able to call SetProcessCallTargets() in order to call CreateThread() is a chicken and egg problem for exploit developers.

Like before, to save memory, the CFG bitmap pages for DLLs are also shared between

processes. This time we can detect whether the start address's CFG bitmap entry is on a sharable page or in a private working set- and alert if it is private.

Control Flow Guard is described in detail **elsewhere**, but a high level CFG overview here is helpful to understanding our approach to detection. Each two bits in the CFG bitmap corresponds to 16 addresses. Two bits gives us four states. Specifically, in a pretty neat optimization by Microsoft, two states correspond only to the 16-byte aligned address (allowed, and export suppressed) and two states correspond to all 16 addresses (allowed and denied).

Modern CPUs fetch instructions in 16-byte lines so modern compilers typically align the vast majority of function entrypoints to 16-bytes. The vast majority of CFG entries only set a single address as a valid indirect call target, and very few entries will specify a whole block of 16 addresses as valid call targets. This means that the CFG bitmap can be an eighth of the size without any appreciable increase in the risk of valid gadgets due to an overly permissive bitmap.

However, if each two bits corresponds to 16 addresses, then a private 4K page of CFG bits corresponds to 256KB of code. That's quite the false positive potential!

Therefore, we just have to hope that legitimate code never does this… nevermind. You should never hope that legitimate code won't do obscure things. To date, we've identified three contemporary scenarios:

- The legacy Edge browser would **harden its javascript host process** by un-setting CFG bits for certain abusable functions
- user32.dll appears to be too kind to legacy software – and will un-suppress export addresses if they are registered as call back functions
- Some security products will drop a page of hook trampolines too close to legitimate modules and private executable memory always has private bitmap entries (Actually they'll often drop this at a module's preferred load address – which prevents the OS from sharing memory for that module)

So we need to rule out false positives by comparing against an expected CFG bitmap value. We could read this from the PE file on disk, but the x64 bitmap is already mapped into our process as part of the shared CFG bitmap.

The PowerShell script implementation we've released alerts on both cases: a modified CFG

page and a start address with a non-original CFG value.

A very small number of CFG-compatible gadgets **might exist** at a given point in time, but only in very specific DLLs that will likely appear anomalous in the surrogate process.

4 - It's literally already a trampoline

The third bypass category is to find an existing function that does exactly what we want, and there are many of these. For example, the one highlighted by Christopher Paschen is Microsoft's C Runtime (CRT). This implementation of the C standard library works as an API layer that sits above Win32– and it includes thread creation APIs.

These APIs perform some extra CRT bookkeeping on thread creation/destruction by passing an internal CRT thread entrypoint to CreateThread() and by passing the user entrypoint to subsequently call as part of the structure pointed to by the CreateThread() parameter.

So, in this case, the Win32StartAddress observed will be the non-exported msvcrt!_startthread(ex). The shellcode address will be at a specific offset from the thread parameter during thread creation (Microsoft CRT source is available), and the shellcode will be the next frame on the call stack after the CRT.

Note: without additional tricks this can only be used to create in-process threads and there is no CreateRemoteThread() equivalent. Those tricks exist, however, and you should not expect this module as a start address in remote threads.

Unfortunately, there is no operating system bookkeeping that will tell you if a thread was created remotely after the fact. Consequently, we can't scan for this with our script– but the inline callbacks used by security products can make this distinction.

Currently, the script simply traverses the stack bottom-up and infers the first handful of frames by looking at candidate return addresses. This code could definitely be improved via disassembly or using unwind information, which are less rewarding to implement in PowerShell. The current approach is reliable enough for demonstration purposes:

Get-InjectedThead - 1 hit

Get-InjectedThreadEx - 5 hits

The updated script detects the original suspicious thread in addition to the four classes of bypass described in this research.

Hunting suspicious thread creations

In addition to detections for the four known major classes of thread start address trampolines, the updated script also includes some additional heuristics. Some of these have medium false positive rates and are hidden behind an -Aggressive flag. However, they may still be useful in hunting scenarios.

The first looks at the starting bytes of the thread's user entrypoint. **Function prologs** have structure- except when they don't. There is no decompiler in PowerShell as far as we know – so we approximated with a byte pattern regular expression instead. Identifying code that doesn't follow convention is useful but could easily exist in a compiler that we haven't tested against.

Interestingly, we had to account for the "MZ" magic bytes that correspond to a **DOS Executable** being a purportedly valid thread entrypoint. The Windows loader **ignores** the value of the AddressOfEntry field in the PE header for Common Language Runtime (CLR) executables such as .NET.

Instead, execution always starts in MsCorEE!_CorExeMain() in the CLR Runtime which determines the actual process entrypoint from the CLR metadata. This makes sense as CLR assembly might only contain bytecode which needs to be JIT'd by the runtime before being called. However, the value of this field is still passed to CreateThread() and it is often zero-

which results in the unexpected MZ entrypoint bytes.

tail byte regex

The second heuristic examines the bytes immediately preceding the user entrypoint. This is usually a return, a jump, or a filler byte. Common filler bytes are zero, nop, and int 3. However, this is only a convention.

In particular, older compilers would regularly place data side by side with code- presumably to achieve performance through data locality. For example, we previously analysed the x64 binaries on Microsoft's symbol server and noticed that this mixing of code and data was normal in Visual Studio 2012, was mostly remediated in VS2013, and appears to have been finally fixed in VS2015 Update 2.

16-byte pseudo alignment

The third heuristic is yet another compiler convention. As mentioned earlier, compilers like to output functions that maximize the instruction cache performance which typically use 16-byte fetches. But compilers appear to also like to save space– so they typically only ensure that the first basic block fits within the smallest number of 16-byte lines as opposed to strict 16-byte alignment. In other words, if a basic block is 20 bytes then it'll always need at least two fetches, but we want to ensure that it doesn't need three.

unexpected Win32 modules

Many common Win32 modules have no valid thread entrypoints at all– so check for these.

This list is definitely non-exhaustive.

Kernel32.dll is a special case. LoadLibrary is not technically a valid thread entrypoint– but CreateRemoteThread(kernel32!LoadLibraryA, "signed.dll") is actually how most security products would prefer software to do code injection into running processes when necessary. That is, the injected code is signed and loaded into read-only image-backed memory. To the best of our knowledge, we believe that this approach was first proposed by Jeffrey Richter in an article in the May 1994 edition of the Microsoft System Journal and later included in his

Advanced Windows book. So treat LoadLibrary as suspicious- but not necessarily malicious.

unexpected ntdll entrypoint

ntdll.dll is loaded everywhere so is often the first choice for a gadget or hook. There are only four valid ntdll entrypoints that we know of and the script explicitly checks for these.

Two of these functions aren't exported, and rather than using P/Invoke to download the public symbols and find the offset in the PDB, the script dynamically queries the start addresses of its own threads for their start addresses to find these. PowerShell already uses worker threads, and the script starts a private ETW logger session to force a thread with the final address.

unsigned DLL start address

Side-loaded DLLs remain a highly popular technique- and are still predominantly unsigned.

SYSTEM impersonation

This one isn't a thread start heuristic- but it was too simple not to include. Legitimate threads might impersonate SYSTEM briefly, but (lazy) malware authors (or operators) tend to escalate privileges initially and hold them indefinitely.

Wrapping up

As flagged last time, nothing in security is a silver bullet. You should not expect 100% detection from suspicious thread creations alone.

For example, an adversary could modify their tools to simply not create any new threads, restricting their execution to hijacked threads only. The distinction is perhaps subtle, but Get-InjectedThreadEx only attempts to detect anomalous thread creation addresses – not the broader case of legitimate threads that were subsequently hijacked. This is why, in addition to imposing costs at thread creation, **Elastic Security** employs other defensive layers including **memory signatures**, **behavioral detections** and **defense evasion detections**.

While it is somewhat easy to hijack a single thread after creation (ensuring that all your malware's threads, including any third-party payloads, uses the right version of the right detection bypass for the installed security products), this is a maintenance cost for the adversary and mistakes will be made.

Let's keep raising the bar. We'd love to hear about thread creation bypasses- and scalable detection approaches. We're stronger together.