



Detecting Thread Creation Trampolines

Get-InjectedThreadEx.ps1cpp

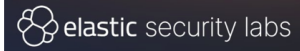
John Uhlmann

BSides Canberra 2023

whoami

Security Research Engineer at Elastic

- Elastic Defend (“EDR”) developer
- Elastic Security Labs Blogger
- <https://www.elastic.co/blog/author/john-uhlmann>



[illegible]

Since its debut at the 2017 SANS Threat Hunting Summit, [Get-InjectedThread.ps1](#) has been a blue team staple for identifying suspicious threads via their start addresses.

However, that JIT'd code rarely manages its own threads – usually that is handled by the runtime or engine. Looking at the sysinternal's VirtualMemoryMap picture, Thread Start Addresses should be in 'purple' image memory – not yellow, blue, orange etc.

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
);
```



As a quick recap, the `CreateThread()` API lets you provide 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 / Procssthreadsapi.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  
void RunShellcode(_In_ LPVOID pShellcode)  
{  
    CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)pShellcode, NULL, 0, NULL);  
}
```

elastic

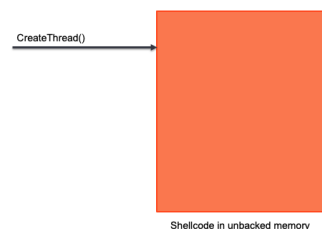
In other words, it's a simple shellcode runner.

And its sibling function `CreateRemoteThread()` is effectively remote process injection.

shellcode_thread detection heuristic

- Detect thread creation where pShellcode is not MEM_IMAGE

```
CreateThread(NULL, 0, pShellcode, pParameter, 0, &threadId)
```




The original detection was based off inspection of the StartAddress parameter being passed to a CreateThread() function – and then determining whether it was backed by a PE file on disk, or not.

While this particular script was an after-the-fact point-in-time scan implementation based on asking the kernel for the value of the Win32StartAddress stored in the relevant ETHREAD structure, the same information is available inline during create thread notify kernel callbacks.

All good EDR products should be providing telemetry of suspicious thread creations.

And all good EPP products should be denying suspicious thread creations by default – with a mechanism to add allowlist entries for legitimate software exhibiting this behaviour.

You'll see such behaviour from other security products (!!), older copy-protection software, anti-cheat software and some Unix software which has been shimmed to run on Windows – like Java. Even with this finite set of exceptions to handle, this detection and/or prevention approach remains highly relevant and successful.



XPEN
Adam Chester
Hacker and Infosec
Researcher
About Me

Twitter LinkedIn GitHub RSS

Understanding and Evading Get-InjectedThread

Posted on 2018-04-09 Tagged in [redteam](#), [windows](#)

One of the many areas of this field that I really enjoy is the "cat and mouse" game played between RedTeam and BlueTeam, each forcing the other to up their game. Often we see some awesome tools being released to help defenders detect malware or shellcode execution, and knowing just how these defensive capabilities function is important when performing a successful pentest or RedTeam engagement.

Recently I came across the awesome post "Defenders Think in Graphs Tool", which can be found over on the SpectreOps blog [here](#). This post is the start of a series looking at "data acquisition, data quality, and data analysis through a case study focused on detecting Process Injection". If you haven't read it, I highly recommend that you do.

One of the tools discussed in the post is "Get-InjectedThread", a Powershell script capable of enumerating running processes and displaying information on any that it believes have been victim to process injection. The tool can be found over on GitHub [here](#).

One thing I thought of when I saw this tool, was just how I would go about bypassing detection if I encountered it during an engagement. Also, with an interest in this area of Windows security, I really wanted a good starting point to build on when this detection technique evolves, either through the next iteration of **Get-InjectedThread**, or through other tools integrating the same method. This post will go through a few different techniques to help us understand just how we could bypass this kind of analysis.



SERVICES EVENTS

BLOG

[Home](#) > [Resources](#) > [Blog](#) > [Avoiding Get-InjectedThread for Internal Thread Creation](#)

AVOIDING GET-INJECTEDTHREAD FOR INTERNAL THREAD CREATION

March 12, 2020
By [Christopher Paschen](#) in [Application Security Assessment](#), [Security Testing & Analysis](#)

Often, a malicious author wants to be able to load non-disk backed code into memory. This could include code that was decrypted and unpacked (a second stage providing more functionality) or plugins to existing running code. After this non-disk backed code is loaded via some mechanism, it can be called normally, or a thread can be started in it. A fairly common detection for malware (and the gist behind the Powershell script Get-InjectedThread) is detecting threads running in memory that are not backed by files on disk.

What if I told you that we could create a thread with a built-in Microsoft Win32 API function, and by using this other function, we would not be detected by Get-InjectedThread? You might scoff because surely it could not be that simple.

However, a number of bypasses have been demonstrated and published.

Most notably these two excellent blogs –

Understanding and Evading Get-InjectedThread by Adam Chester.

Avoiding Get-InjectedThread for Internal Thread Creation by Christopher Paschen.

In today's talk we'll go through each major bypass class and the forensic traces they leave in detail.

shellcode_thread detection heuristic

- Detect thread creation where pShellcode is not MEM_IMAGE

CreateThread(NULL, 0, pShellcode, pParameter, 0, &threadId)

```
MEMORY_BASIC_INFORMATION mbi = { 0 };
if (VirtualQueryEx(hProcess, lpStartAddress, &mbi, sizeof(mbi)) &&
    mbi.Type != MEM_IMAGE)
{
    printf("shellcode thread detected @ %p\n", lpStartAddress);
}
```



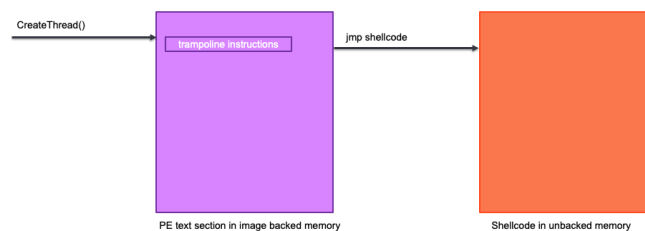
The original Get-InjectedThread detection was implemented by using the VirtualQueryEx API to ask the kernel for information about the start address's memory region. That information includes a flag indicating whether the memory is a PE image, a mapped file or simply private memory. If it's not image and it's a thread start address – it's suspicious.

Simple. Quick. And deterministic FPs for certain software.

shellcode_thread bypass

- Bypass with a MEM_IMAGE trampoline

CreateThread(NULL, 0, **pTrampoline**, pParameter, 0, &threadId)



So how to bypass? We need the `Win32StartAddress` that is provided to the the kernel to be an image-backed location – and we need that address to point to instructions that pass execution to the ultimate shellcode address.

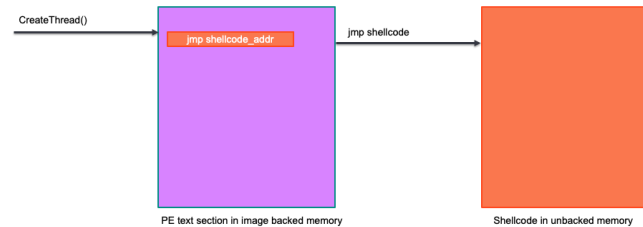
So, one way this detection can be bypassed by passing in a thread start address which is an image-backed location, but which contains instructions which will transfer execution to the unbacked memory. This is known as a trampoline – as you are quickly catapulted somewhere else.

There are four broad classes of trampolines – you can build your own trampoline from scratch, you can use an illusionary trampoline, you can repurpose something else a a trampoline, or you can find an existing trampoline. Aka hooking, spoofing, gadgets and wrapper functions.

Hook

- Malware overwrites instruction bytes with trampoline

`CreateThread(NULL, 0, pTrampoline, pParameter, 0, &threadId)`



#1 – bring your own trampoline

The simplest trampoline is a small hook. You just write the needed jump instruction into existing image-backed memory.

These bytes can even be restored to their original values straight after thread creation. This helps with avoiding retrospective detection – but recall that your endpoint security product should be doing **inline** detection and will be able to see hooked thread entryptoint at execution time.

Hook

- Malware overwrites instruction bytes with trampoline

CreateThread(NULL, 0, **pTrampoline**, pParameter, 0, &threadId)

```
char hook[] =  
{  
    0x48, 0xb8, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88, // movabs rax, <placeholder>  
    0xff, 0xe0 // jmp rax  
};  
*(ULONG_PTR*)(hook + 2) = (ULONG_PTR)pShellcode;  
  
auto pHookedFunc = GetProcAddress(GetModuleHandle(L"ntdll.dll"), "DbgUiRemoteBreakin");  
WriteProcessMemory(GetCurrentProcess(), pHookedFunc, hook, sizeof(hook), NULL);  
  
CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)pHookedFunc, NULL, 0, NULL);
```



Here's an example.

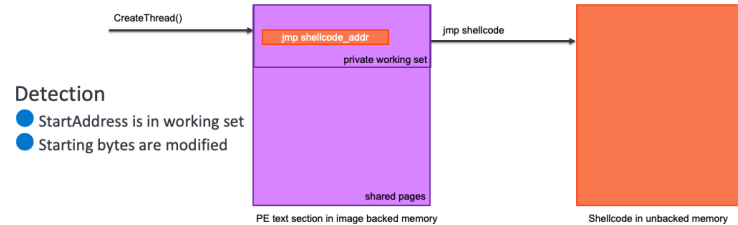
ntdll!DbgUiRemoteBreakin is my favourite function to trampoline via.
It's a legitimate, but rarely used in production, remote thread start address.

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

Hook

- Malware overwrites instruction bytes with trampoline

`CreateThread(NULL, 0, pTrampoline, pParameter, 0, &threadId)`



Now, we can detect hooked start addresses fairly easily.

To save memory, Windows ensures that the virtual memory for shared libraries uses the same backing physical memory pages and that virtual memory is tagged as `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 private working set of the process.

Hook

- Malware overwrites instruction bytes with trampoline

CreateThread(NULL, 0, pTrampoline, pParameter, 0, &threadId)

```
// Has our MEM_IMAGE Win32StartAddress been (naively) hooked?  
// https://blog.redbluepurple.io/offensive-research/bypassing-injection-detection//creating-the-thread  
// Note - checking against bytes on disk after the fact won't help with false positives  
// as the hook can easily be removed after thread start.  
// Detection gap - the hook could easily be deeper, potentially even in a subsequent call. :-(  
// Microsoft-Windows-Threat-Intelligence ETW events should detect this more robustly.  
PSAPI_WORKING_SET_EX_INFORMATION pwsei{};  
pwsei.VirtualAddress = thread.Win32StartAddress;  
if (K32QueryWorkingSetEx(hProcess, &pwsei, sizeof(pwsei)) && !pwsei.VirtualAttributes.Shared)  
    detections.push_back("private_image");
```



We can query the kernel memory manager and ask whether the page containing the start address is shared or is in a private working set.

So now we know that something on the page was modified (which is uncommon) – but not yet if our address was hooked.

This could be a false positive if there is a “legitimate” hook or other modification (or a hook on the executable image which is typically not shared) on the same page.

In particular, most security products like to hook ntdll and there are also a plethora of software products out there that extend functionality of other products via hooks, as well as some older copy protection libraries that just unpack their code at runtime etc.

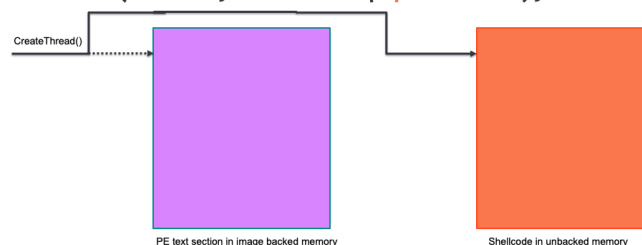
So, if that 4KB page is private, then an inline detection would additionally compare the start address bytes to an original pristine copy and only alert if they differ.

And, to deploy at scale, it would also need to maintain an allow list for those rare legitimate uses.

Spoof

- Malware modifies the entry point just prior to execution

```
CreateThread(NULL, 0, pTrampoline, 0, CREATE_SUSPENDED, &threadId)  
SetThreadContext(hThread, CONTEXT.Rip=pShellcode);
```



#2 – shifting the mat

I mentioned earlier that security products can do inline prevention at the time of the thread notification callback. That's true – but it's also not the whole story.

The thread notify routine is called before the thread starts and it turns out that you can modify the thread's state after the callback but before the thread is executed.

In particular, Microsoft does not provide security vendors with a mechanism to determine whether a thread was created suspended or not. And, on x64 Windows, Microsoft does not allow security vendors to do our own kernel hooking – so we can't easily check the parameters either.

So, malware authors quickly realised that they could create a `SUSPENDED` thread which triggers the thread callback, then after passing that initial inspection, they can modify the thread's register state with `SetThreadContext` (or alternatively queue an `EarlyBird` APC) and then resume the thread.

Again, this is not the whole story - effective security products can still detect this inline too. But that's a topic for another day.

What about retrospective detection via our scanning tool?

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.

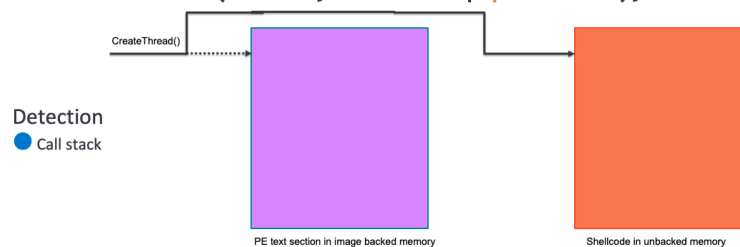
For context manipulation, the first (or third) frame will belong to the injected code.

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

Spoof

- Malware modifies the entry point just prior to execution

```
CreateThread(NULL, 0, pTrampoline, 0, CREATE_SUSPENDED, &threadId)  
SetThreadContext(hThread, CONTEXT.Rip=pShellcode);
```



I eschewed the traditional easy walk down the stack using the `StackWalk64()` API and decided to attempt to manually climb up the stack and look for candidate return addresses.

These can be further validated by disassembling the preceding bytes to a candidate return address looking for a preceding call instruction, and by using unwind information present in PE files to determine if the candidate stack frame was the correct size.

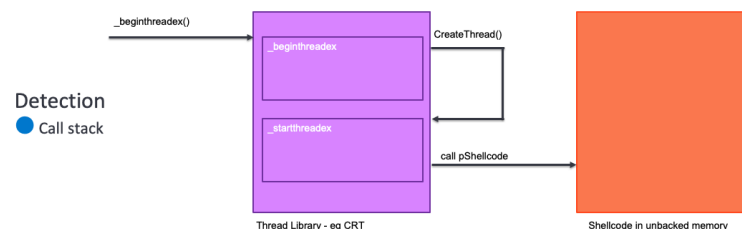
Alas, I sometimes encountered the dreaded `SET_FPREG` unwind opcode. The presence of a frame pointer means the function can legally use `_alloca()` for a dynamic stack allocation. And, since I wasn't walking down the stack in the correct fashion, I couldn't emulate execution to determine the frame pointer's exact value at time of unwind. So, I didn't have the stack frame's size – just a lower bound for it.

This improved call stack climbing has accurately identified the initial stack frames in all of the (limited :-p) testing that I have done so far.

Wrapper Functions

- Malware calls a thread library that wraps Win32 threads

```
msvcrt!_beginthreadex(NULL, 0, pShellcode, NULL, 0, &threadId)
```



The third bypass category is to find a function that does exactly what we want. There are multiple of these. For example, Microsoft's C Runtime is 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 a CRT thread entrypoint to `CreateThread` and passing the user entrypoint to for the CRT thread start function to then 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, after the fact, the shellcode will be the next frame on the callstack after the CRT. Note – without additional tricks this can only be used to create in-process threads and there is no `CreateRemoteThread()` equivalent in the CRT. Those additional tricks exist though, so you should also 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. So, we can't scan for this – but the inline callbacks used by security products can make this distinction and be more aggressive in blocking anomalous remote thread creations.

On the retrospective detection front, we just need to collect an an extra stack frame during our reconstruction of the initial stack frames – and look for a private return address in the fourth frame from the bottom. Theoretically, this could false positive on JIT or (legitimate) packed code, but I've yet to encounter a sample of this.

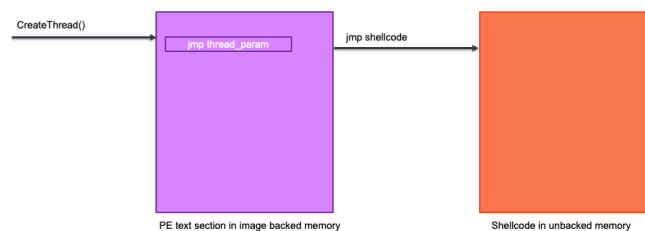
Red Team – Yes, you can bypass my stack climbing scan by writing some fake call stack in the slack space left by stack randomization. But -

- a) endpoint security products are likely doing to be stack walking properly rather than my stack climbing approach.
- b) security products can easily store the size of that randomization during thread creation and check for shenanigans later
- c) I can just climb the stack a little higher...

Gadget

- Malware repurposes existing instruction bytes as trampoline

`CreateThread(NULL, 0, pTrampoline, pShellcode, 0, &threadId)`



#4 – we repurpose something to do something else that the original author did not intend. Basically, we attempt to find a suitable gadget within image backed memory so that no code modification is necessary.

Our earlier 64-bit 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()`.

So, the simplest x64 gadget is the two-byte “`jmp rcx`” instruction ‘`ffe1`’ – which it turns out is fairly trivial to find.

Gadget

- Malware repurposes existing instruction bytes as trampoline

`CreateThread(NULL, 0, pTrampoline, pShellcode, 0, &threadId)`



Gadgets don't even need to have originally been instructions per se – they could be within operands or other data within the code section.

For example, this “ff e1” gadget in `ntdll.dll` was part of the relative address of a GUID.

We could potentially come up with a list of possible RCX-RIP pivot gadgets and detect those...

But we can actually detect unknown gadgets too. Because it doesn't actually work...yet. ;-)

In all modern Windows software, thread start address are protected by Control Flow Guard (CFG) which has a bitmap of valid indirect call targets computed at compile time.

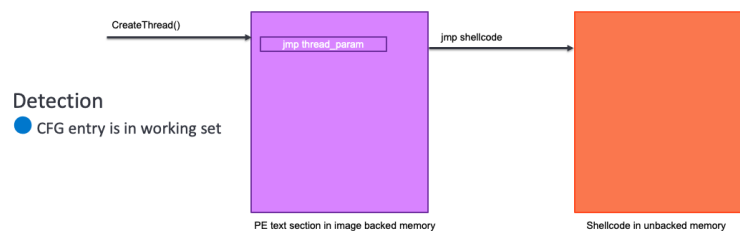
So, to use this gadget, the malware must typically first call the `SetProcessValidCallTargets()` function to ask the kernel to dynamically set the bit corresponding to this gadget in the CFG bitmap.

(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 `SetProcessValidCallTargets` in order to call `CreateThread` is a chicken and egg problem for exploit writers.)

Gadget

- Malware repurposes existing instruction bytes as trampoline

```
CreateThread(NULL, 0, pTrampoline, pShellcode, 0, &threadId)
```



Detection

- CFG entry is in working set



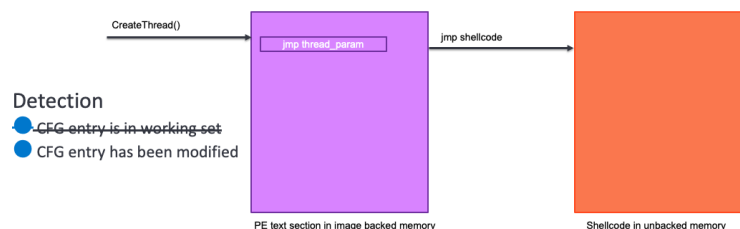
Like before, to save memory, the CFG bitmap pages for DLLs are also shared between processes. So, this time we can detect whether the start address's CFG bitmap entry is on a shared page or in a private working set. And alert if it is private.

Each two CFG bits corresponds to 16 addresses. Two bits is four states. Specifically, in a pretty awesome optimisation 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 like to align the vast majority of function entry points to 16-bytes. 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.

Gadget

- Malware repurposes existing instruction bytes as trampoline

`CreateThread(NULL, 0, pTrampoline, pShellcode, 0, &threadId)`



Detection

- CFG entry is in working set
- CFG entry has been modified



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

So far, I've identified three false positive scenarios -

- The legacy Edge browser would harden its javascript host process by un-setting CFG bits for certain abusable functions.
- user32.dll is 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 different module's preferred system load address – and so force that DLL to be loaded non-shared and waste a few MB of your host's physical memory as well).

So, we need to rule out FPs 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 the shared CFG bitmap!

Gadget

- Malware repurposes existing instruction bytes as trampoline

CreateThread(NULL, 0, pTrampoline, pShellcode, 0, &threadId)

```
////////////////////////////////////  
// Check for suspicious CFG BitMap states in our local pristine copy of the x64 bitmap  
// Notes - executable CFG bitmaps are not shared - only library (dll) ones.  
//         - only 16-bytes aligned addresses, as this is a SetProcessValidCallTargets() requirement.  
ULONG cfgBits;  
if (InSystemImageRange(thread.Win32StartAddress) && 0 == ((ULONG_PTR)thread.Win32StartAddress & 0xF) &&  
    GetCfgBitsForAddress(thread.Win32StartAddress, &cfgBits) && 0 == cfgBits)  
{  
    detections.push_back("cfg_invalid");  
}
```

PE text section in image backed memory

Shellcode in unbacked memory



This only work for DLLs because of their shared nature.

Microsoft's doesn't tell us where the CFG BitMap is – but we can easily find it as various exported functions need to know where it is.

Note – to detect Wow64 variants of this, you would need to run a Wow64 version of the tool.

So, to detect these gadget trampolines, we just need to to calculate the offset into the CFG BitMap for the Win32StartAddress and then read those 2 bits of memory from our own process.

And once we have that it is straight forward to check whether the address is a valid indirect call target.

Get-InjectedThreadEx

```
case ZYDIS_MNEMONIC_PUSH:
    // push nonvolatile
    bPrologStarted = true;
    break;
case ZYDIS_MNEMONIC_MOV:
    // mov [RSP+n], nonvolatile
    if (IsSaveRegisterOperation())
        bPrologStarted = true;
    // mov frame-pointer, RSP
    else if (IsFramePointerOperation())
        bPrologStarted = true;
    // mov RAX, fixed-allocation-size
    // mov RAX, indirect-call-target
    else if (IsRegDestination(ZYDIS_REGISTER_RAX)) {
        SaveRegister(ZYDIS_REGISTER_RAX);
        bRaxSet = true;
        bThreadParameterInRax |= IsRegSource(ZYDIS_REGISTER_RCX);
    }
    else if (IsRegDestination(ZYDIS_REGISTER_EAX))
        bRaxSet = true;
    // sometimes stub functions reorder parameters
    // mov FastcallParamReg, RCX
    else
        bValidInstruction = IsRegSource(ZYDIS_REGISTER_RCX) &&
        (IsRegDestination(ZYDIS_REGISTER_RDX) || IsRegDestination(ZYDIS_REGISTER_RAX));
    break;
case ZYDIS_MNEMONIC_CALL:
    // call __chkstk() is the only call allowed in a prolog
    // it uses a special calling convention
    bValidInstruction = bRaxSet & bRaxSetInstruction;
    break;
case ZYDIS_MNEMONIC_LEA:
    // lea frame-pointer, [RSP-n]
    if (IsFramePointerOperation())
        break;
    // Some "functions" are just stubs around other functions with
    // one (or more) fixed parameters.
    // lea RCX, [n] - set first parameter.
    else
        bValidInstruction = IsRegDestination(ZYDIS_REGISTER_RCX);
    break;
case ZYDIS_MNEMONIC_SUB: // prolog delimiter
    bPrologFinished = IsStackOperation();
    break;
case ZYDIS_MNEMONIC_TEST:
    // test RCX, RCX - is the first parameter NULL?
    // Checking for a non-NULL parameter and bailing early is
    // a common optimisation.
    bTestRcx = IsRegSource(ZYDIS_REGISTER_RCX) && IsRegDestination(ZYDIS_REGISTER_RCX);
    break;
case ZYDIS_MNEMONIC_JZ:
    // test RCX, RCX
    // jz early-exit - don't follow
    bValidInstruction = bTestRcx;
    break;
case ZYDIS_MNEMONIC_JNZ:
    // test RCX, RCX
    // jnz true-entry-point - follow
    bValidInstruction = bTestRcx;
    if (!FollowJump())
        bPrologFinished = bTestRcx;
    break;
case ZYDIS_MNEMONIC_JMP:
    // Some functions start with a short jmp to provide hotpatch space.
    // jmp n - follow
    bPrologFinished = FollowJump();
    break;
default:
    bValidInstruction = false;
```



Another approach to checking Win32StartAddress legitimacy would be comparison against a list of known gadget instructions.

Sure, you could attempt that, but instead of detecting known-bad can we instead create a model for good entry points?

Theoretically x64 prologs should be quite constrained because of the need to describe the prolog via UNWIND_INFO.

In theory, that's great. In practice, there are compiler writers.

It's usually just some pushes and a stack pointer adjustment and maybe a frame pointer.

But there could be some hot patch space, jump tables, optimizations for wrapper functions or early returns...

But that's still a small enough set of cases to handle.

Identifying code that doesn't follow known convention is useful – but it could easily be a rare compiler that I haven't tested against.

Get-InjectedThreadEx

```
// False positives can occur if data was included in a code section. This was common in older compilers...
// ...and also in new compilers that support XFG. In this case, the 8-byte XFG hash is immediately before.
// https://blog.quarkslab.com/how-the-msvc-compiler-generates-xfg-function-prototype-hashes.html
const auto& tailbytesEnd = tailBytes.data() + tailBytes.size();
auto bIsValidTail = tailBytes.size() >= sizeof(UINT64) &&
    IsValidXfgHash(*(UINT64*)(tailbytesEnd - sizeof(UINT64)));

// The byte preceding a function prolog is typically a return, or filler byte.
bIsValidTail |= tailBytes.empty() || '\x00' == tailBytes.back(); // NUL filled.
for (auto i = 1; !bIsValidTail && i <= tailBytes.size(); i++) {
    if (!ZYAN_SUCCESS(ZydisDecoderDecodeInstruction(&decoder, NULL, tailbytesEnd - i, i, &instruction)) ||
        continue;
    switch (instruction.mnemonic) {
        // valid basic block end instructions
        case ZYDIS_MNEMONIC_CALL:
        case ZYDIS_MNEMONIC_JMP:
        case ZYDIS_MNEMONIC_RET:
        // valid alignment filler instructions
        case ZYDIS_MNEMONIC_NOP:
        case ZYDIS_MNEMONIC_INT3:
            bIsValidTail = true;;
    }
```



Similarly, the bytes before an entry point are usually a filler byte (00, 90 [nop] or cc [int3/breakpoint]), a return or a jump. But again – this is only convention.

And older compilers would regularly place raw data side by side with code in the executable .text section. Based on an analysis of x64 binaries on the Microsoft's symbol server I did in 2018 - this mixing of code and data was normal in Visual Studio 2012, mostly remediated in VS2013 and appears to have been finally fixed in VS2015 Update 2.

So, you might still see false positives with this heuristic – but only for non-Microsoft software. Or so I thought. Until I hit a false positive on a very recently compiled Microsoft binary. Looking closer though, that data had some very interesting cross references... I'd completely forgotten that eXtended Flow Guard litters ~55 bit hashes before any XFG protected indirect call targets. The folks at Quarks Lab reversed the algorithm and found two ~9-bit masks. One for always on. And one for always off. And I could check those.

It would be nice if Microsoft required thread entry points in images to be named exports...

Get-InjectedThreadEx

```
// There are no valid thread entry points (that I know of) in many Win32 modules.
const std::array<std::string, 11> modulesWithoutThreadEntrypoints = {
    "kernel32", "kernelbase", "user32", "advapi32",
    "psapi", "dbghelp", "imagehlp", "powrprof",
    "verifier", "setupapi", "rpcrt4" }; // ...and many more
const auto startModule = std::filesystem::path(mappedPath).stem().string();
for (const auto& module : modulesWithoutThreadEntrypoints)
    if (startModule == module) {
        (void)GetNearestSymbolWithPdb(hProcess, thread.Win32StartAddress, symbol);
        detections.push_back("unexpected(" + startModule + ")");
    }

// kernel32!LoadLibrary
// And, even if there are, LoadLibrary is always a suspicious start address.
static auto hKernel32 = GetModuleHandle(L"kernel32.dll");
static auto pLoadLibraryW = GetProcAddress(hKernel32, "LoadLibraryW");
static auto pLoadLibraryA = GetProcAddress(hKernel32, "LoadLibraryA");
if (pLoadLibraryA == thread.Win32StartAddress || pLoadLibraryW == thread.Win32StartAddress)
    detections.push_back("unexpected(" + symbol + ")");
```



So, modules that are already loaded are typically where you want to find your gadgets. But many Win32 modules have no valid thread entry points at all – so this is an easy check to make gadget and wrapper function use harder. Though, my list is definitely non-exhaustive.

Kernel32/base is a special case.

LoadLibrary is not technically a valid thread entrypoint – but CreateRemoteThread(kernel32!LoadLibrary, “signed.dll”) is actually how most security products would prefer software to do code injection if it really must.

Get-InjectedThreadEx

```
// ntdll.dll but not a known entrypoint.  
// These are the only valid thread entry points in ntdll that I know of.  
static const std::array<PVOID, 4> ntdllThreadEntryPoints = {  
    GetSymbolAddress("ntdll!TppWorkerThread"),  
    GetSymbolAddress("ntdll!EtwpLogger"),  
    GetSymbolAddress("ntdll!DbgUiRemoteBreakin"),  
    GetSymbolAddress("ntdll!RtlpQueryProcessDebugInformationRemote")  
};  
  
if (mappedPath.ends_with(L"\\System32\\ntdll.dll")) {  
    auto bValidNtdllEntry = false;  
    for (const auto& address : ntdllThreadEntryPoints)  
        bValidNtdllEntry |= address == thread.Win32StartAddress;  
  
    if (!bValidNtdllEntry) {  
        detections.push_back("unexpected(" + symbol + ")");  
    }  
}
```

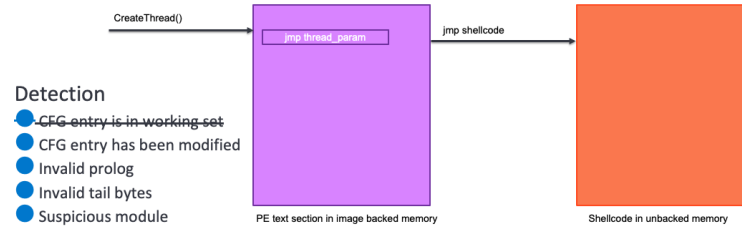


ntdll – it's loaded everywhere so is often the first choice for a gadget or hook.
There are only four valid ntdll entry points that I know of – so explicitly check for these.

Gadget

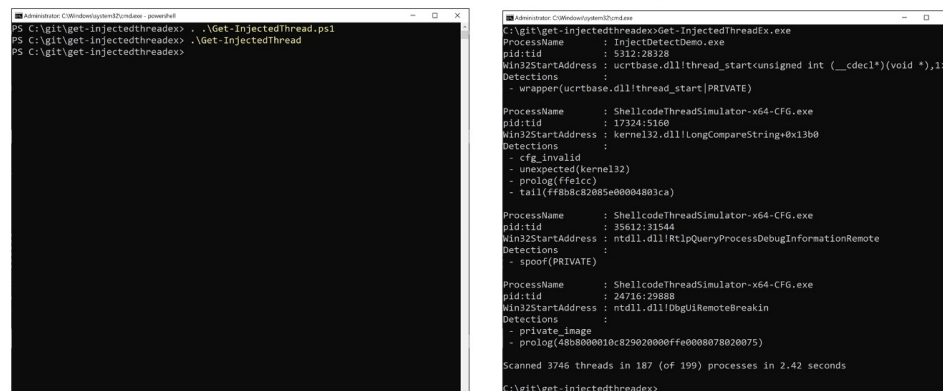
- Malware repurposes existing instruction bytes as trampoline

`CreateThread(NULL, 0, pTrampoline, pShellcode, 0, &threadId)`



So, in fact, there are quite a few ways to detect suspicious start address gadgets.

Get-InjectedThread vs Get-InjectedThreadEx



```
PS C:\git\get-injectedthread> .\Get-InjectedThread.ps1
PS C:\git\get-injectedthread> .\Get-InjectedThread
PS C:\git\get-injectedthread>
```

```
C:\git\get-injectedthreadex> Get-InjectedThreadEx.exe
ProcessName : InjectDetectDemo.exe
pid:tid : 5312:28328
Win32StartAddress : ucrtbase.dll!thread_startunsigned int (__cdecl*)(void *),1>
Detections :
- wrapper(ucrtbase.dll!thread_start|PRIVATE)

ProcessName : ShellcodeThreadSimulator-x64-CFG.exe
pid:tid : 17324:5160
Win32StartAddress : kernel32.dll!LongCompareString@x13b0
Detections :
- cfg_invalid
- unexpected(kernel32)
- prolog(ffelcc)
- tail(ffb8c9285e0004803ca)

ProcessName : ShellcodeThreadSimulator-x64-CFG.exe
pid:tid : 35612:31544
Win32StartAddress : ntdll.dll!RtlpQueryProcessDebugInformationRemote
Detections :
- spoof(PRIVATE)

ProcessName : ShellcodeThreadSimulator-x64-CFG.exe
pid:tid : 24716:29888
Win32StartAddress : ntdll.dll!DbgUiRemoteBreakin
Detections :
- private_image
- prolog(48b800010c82902000ffe0008078020075)

Scanned 3746 threads in 187 (of 199) processes in 2.42 seconds
C:\git\get-injectedthreadex>
```

0 hits 😞

4 hits 😊



The end result – we can now detection the original technique plus all four classes of bypass.

~~100% Detection~~ Layered Defences

- Thread creation callbacks are a defensible boundary – so defend it.
 - Especially for thread injection into remote processes.
 - But don't rely on it.
- It's easy to hijack a single thread after creation...
 - ... but harder to ensure that all your tools are thread-less.



Don't expect 100% detection from suspicious thread creations alone.

You'll want defence-in-depth with memory scanning of unbacked memory and detection when unbacked memory is calling out to suspicious APIs etc.

That said I'd love to hear about thread **creation** bypasses.

It's somewhat easy to hijack a single thread after creation, but ensuring that all your malware's threads, including any third-party payloads (statically linked libraries, reflective DLLs, BOFs etc) use the right detection bypass for the installed security product(s) is a maintenance cost for the adversary. And mistakes will be made.

References

- <https://www.elastic.co/security-labs/hunting-memory>
 - <https://www.slideshare.net/JoeDesimone4/taking-hunting-to-the-next-level-hunting-in-memory>
 - <https://gist.github.com/jaredcatkinson/23905d34537ce4b5b1818c3e6405c1d2>
- <https://blog.xpnsec.com/understanding-and-evading-get-injectedthread/>
- <https://www.trustedsec.com/blog/avoiding-get-injectedthread-for-internal-thread-creation/>

Links

- @jdu2600
- <https://www.elastic.co/security-labs/get-injectedthreadex-detection-thread-creation-trampolines>
- <https://github.com/jdu2600/Get-InjectedThreadEx>

