

# Maelstrom: EDR Kernel Callbacks, Hooks, and Call Stacks

---

[pre.empt.dev/posts/maelstrom-edr-kernel-callbacks-hooks-and-callstacks](https://pre.empt.dev/posts/maelstrom-edr-kernel-callbacks-hooks-and-callstacks)

## Introduction

---

To recap the series so far, we've gone from looking at the high level purposes and intentions of a Command and Control Server (C2) in general to designing and implementing our first iteration of our implant and server. If you've been following along, you might think you've written a C2...

This is a common mindset. In our experience, getting to this point *does not require much sophistication*. All of our previous work could easily be achieved (and has been achieved!) using C#, Python, Go, in an evening's worth of frenetic caffeine-fuelled typing. Leading features of C2s can often be linked to pretty old solved concepts and patterns from software engineering, such as thread management, handling idle processes, and ensuring correction execution and program flow.

But as we found when writing our various C2s, and as numerous other offensive developers have found when writing their own implants and servers, once you have the code working and you can get a pingback, you stop running your implant on your development computer and try it on a second computer. This is where the questions start creeping in. Questions like "Why can't I access remote files?", "Why can I make outbound requests over *this* protocol, but not *this*?", "Why does this command just *fail* with no explanation", and for the cynical self-doubter with enough imposter syndrome "Why isn't Defender stopping me from doing *this*?".

This, personally, is the post we were looking forward to writing. It's going to be a discussion, with a few examples, of increasing common behaviours within environments with active endpoint protection. In 2022, implants face far more scrutiny - the implant and C2 operator must to be prepared to face or evade this scrutiny, and the defender must be aware of how it works so that it can be used to the best of its ability.

Whilst writing this, we also want to clear up the 'it avoids <insert company here> EDR' tweets. Just because the implant is able to execute, doesn't mean that Endpoint Protection is blind to it - it *can* mean that, but we want to demonstrate some techniques these solutions use to identify malicious behaviour and raise the suspicion of an implant.

In a nutshell, proof of execution is *not* proof of evasion.

## Objectives

---

This post will cover:

- Setting up [The Hunting ELK](#)
- Reviewing three ways EDRs can detect or block malicious execution:
  - Kernel Callbacks
  - Hooking
  - Thread Call Stacks

By the end of this post, we will have covered how modern EDRs can protect against malicious implants, and how these protections can be bypassed. We will move from having an implant which technically works to an awareness of how to write an implant which actually starts to work, and can achieve the goals of an operator.

As we've said many times, we are *not* creating an operational C2. The output from this series is poorly written and riddled with flaws - it only does enough to act as a broken proof-of-concept of the specific items we discuss in this series to avoid this code from being used by bad actors. For this same reason, we are trying to avoid discussing Red Team operational tactics in this series. However, as we go on, it will become obvious why blending in with the compromised users typical behaviour will work. This is something that [xpn](#) has discussed on Twitter:

Find Confluence, read Confluence.. become the employee!

— Adam Chester (@\_xpn\_) [January 22, 2022](#)

If your implant has been flagged by EDR, querying `NetSessionEnum` on every AD-joined computer to find active sessions is probably not typical user behaviour. You likely will not know your implant has been flagged until it stops responding. From here, it's a race until your implant is uploaded to [VirusTotal](#) and you have to go back to the drawing board.

We will be referring to the following programs a lot during this blog:

[The Hunting ELK](#) (HELK): HELK is an [Elastic](#) stack best summarised by themselves:

The Hunting ELK or simply the HELK is one of the first open source hunt platforms with advanced analytics capabilities such as SQL declarative language, graphing, structured streaming, and even machine learning via Jupyter notebooks and Apache Spark over an ELK stack.

This project was developed primarily for research, but due to its flexible design and core components, it can be deployed in larger environments with the right configurations and scalable infrastructure.

[PreEmpt](#): A pseudo-EDR which has the capability to digest EtwTi, memory scanners, hooks, and so on. Although, this is not public but code will be shared when necessary.

These two tools will allow us to generate proof-of-concept data when required.

## **Important Concepts**

---

Similar to [Maelstrom: Writing a C2 Implant](#), we want to have a section dedicated to clearing up some topics we feel need some background before moving on.

## **What do we mean by Endpoint Detection and Response**

---

Endpoint Detection and Response (EDR) software goes by a number of different acronyms, and there may well be distinctions between different companies programs and their functionality. For the sake of simplicity, we are call all programs that are limited to scanning files on disk statically "anti-virus", and all programs that go further and scan device memory, look at the behaviour of programs while they are running, and responding to threats as they happen "EDR"s. These may be called various names, including XDR, MDR, or just plain AV.

Throughout this series, as we have done so far, we will be sticking with "EDR".

A good overview of this is [CrowdStrike's post "What is Endpoint Detection and Response \(EDR\)":](#)

Endpoint Detection and Response (EDR), also referred to as endpoint detection and threat response (EDTR), is an endpoint security solution that continuously monitors end-user devices to detect and respond to cyber threats like ransomware and malware.

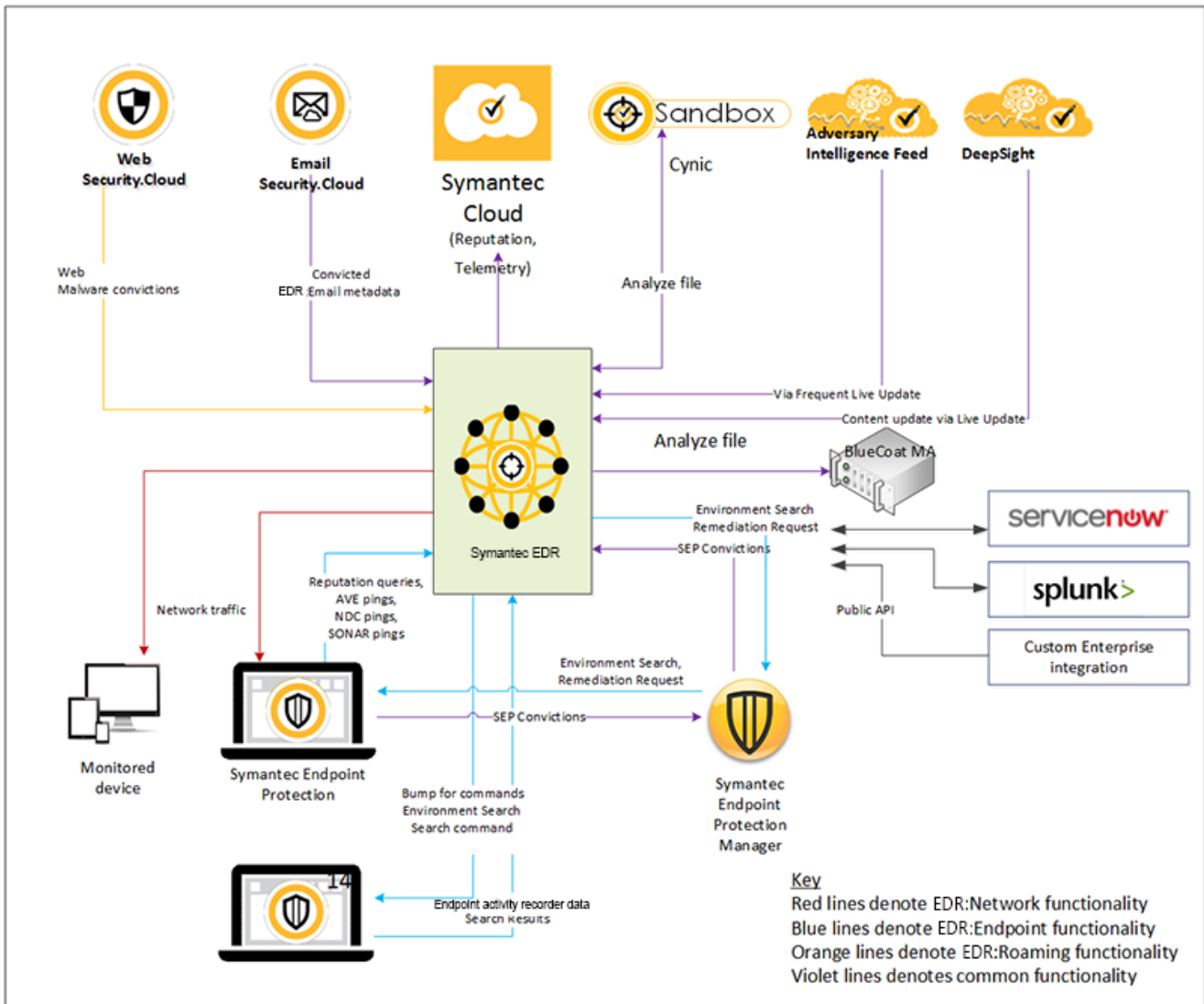
Coined by Gartner's Anton Chuvakin, EDR is defined as a solution that “records and stores endpoint-system-level behaviors, uses various data analytics techniques to detect suspicious system behavior, provides contextual information, blocks malicious activity, and provides remediation suggestions to restore affected systems.”

Because it's relevant to this post, the next section will look at EDR architecture and comparing EDR behaviours across the various vendors. Without going hugely off-topic, we won't look at a number of also relevant areas, such as how Anti-Virus works, how disk-based protection may work to also stymie your implant execution (if you're still running on disk), and how AV and EDR actually goes about scanning files and their behaviours while they are doing so. Turns out, that's like, a whole field of study.

## **Common EDR Architecture**

---

When discussing endpoint protection, it may help to be somewhat familiar with their architecture. The [Symantec EDR Architecture](#) looks something like this:



A similar approach can be seen for Defender for Endpoint. Essentially, a device with the product installed will have an agent which can consist of several drivers and processes, which gather telemetry from various aspects of the machine. Through this post and the next, we will go over a few of those.

As an aside, in a Windows environment, Microsoft inherently have an edge here. While this is currently aimed at "Large Enterprise" customers (or at least, we assume, given their price point for Azure!), Microsoft's Defender and new Defender MDE can both access Microsoft's knowledge of ... their own operating system, but also influence the development of new operating system functionality. Long-term, it wouldn't be a surprise to see Microsoft Defender MDE impact the EDR market in a similar way that Microsoft Defender impacted the anti-virus market.

The general gist of all EDR is that telemetry from the agent is sent to the cloud where it's run through various sandboxes and other test devices, and its behaviour can be further analysed by machine and human operators.

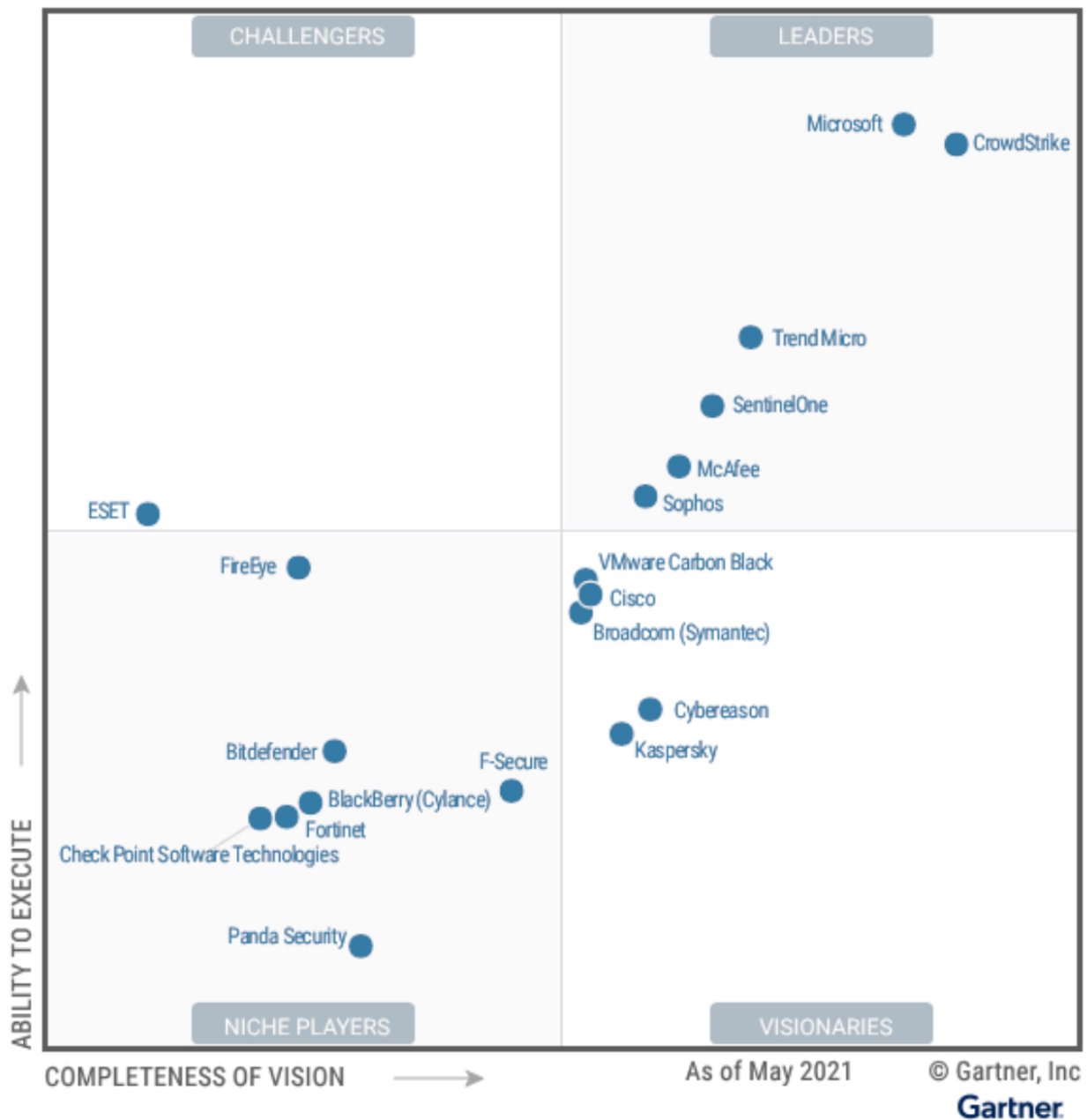
For the excessively curious reader, the following links go in to more detail about specific vendor approaches to EDR architecture:

### **Briefly Reviewing and Comparing EDR Behaviour at a High Level**

---

Without going hugely off-topic, just as how not every red team assessment is a red team, not every EDR is an EDR.

The following "[Gartner Magic Quadrant](#)", from [Gartner's May 2021 Report](#) roughly maps out the EDR landscape. It's worth noting that [CrowdStrike's hire of Alex Ionescu](#) (a maintainer for the Kernel in [ReactOS](#)) demonstrates that the current best-in-class EDR's heavily leverage knowledge of internal Windows functionality to maximise their performance:



Source: Gartner (May 2021)

With so much of EDR functionality relying on implementing the methods we will discuss here such as custom-written direct behaviours like kernel callbacks and hooking, being able to quickly implement new Microsoft Windows features and develop your own custom ways of reliably interacting with and interrupting malicious processes seems to be the distinguishing feature of a modern EDR from its peers.

Another metric that EDR Vendors tend to use, especially because the reports are made so public, is the Mitre Engenuity. The Attack Evaluations is described as thus:

The MITRE Engenuity ATT&CK® Evaluations (Evals) program brings together product and service providers with MITRE experts to collaborate in evaluating security solutions. The Evals process applies a systematic methodology using a threat-informed purple teaming approach to capture critical context around a solution's ability to detect or protect against known adversary behavior as defined by the ATT&CK knowledge base. Results from each evaluation are thoroughly documented and openly published.

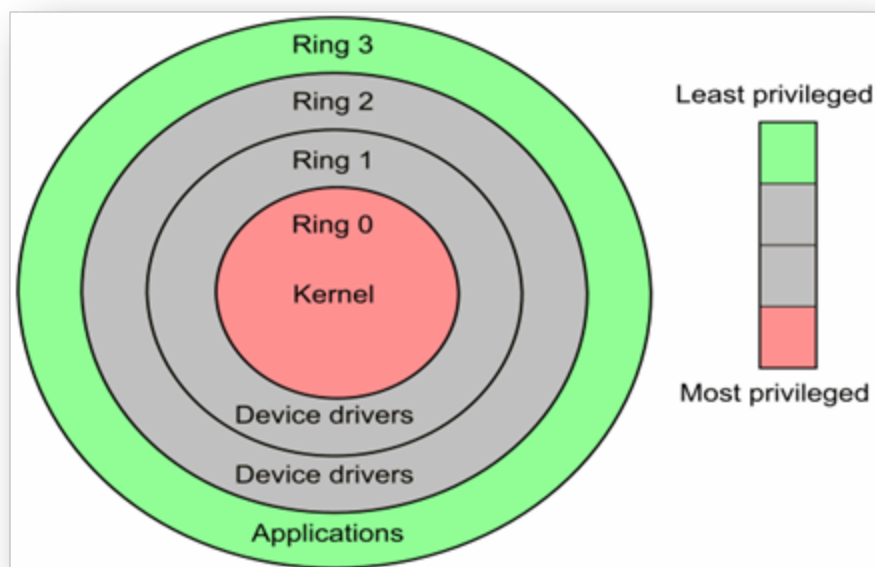
For example, with SentinelOne, their results can be seen in: SentinelOne Overview. The overview goes through APT scenarios and marks whether or not the technique was detected and can be used as a tracker for its "effectiveness". However, some have expressed feelings online that this is not a thorough way to determine the effectiveness of the product.

When looking at EDRs from a purchasing perspective, there are a few methods of determining effectiveness and we wanted to briefly highlight them here. The main thing to consider is that some vendors do not necessarily provide more functionality than an anti-virus. As with any product, ensure that you purchase the right solution for your businesses needs.

## User-land and Kernel-land

---

When discussing the kernel and user-land model, the following architectural image familiar to any Computer Science graduate will be used:



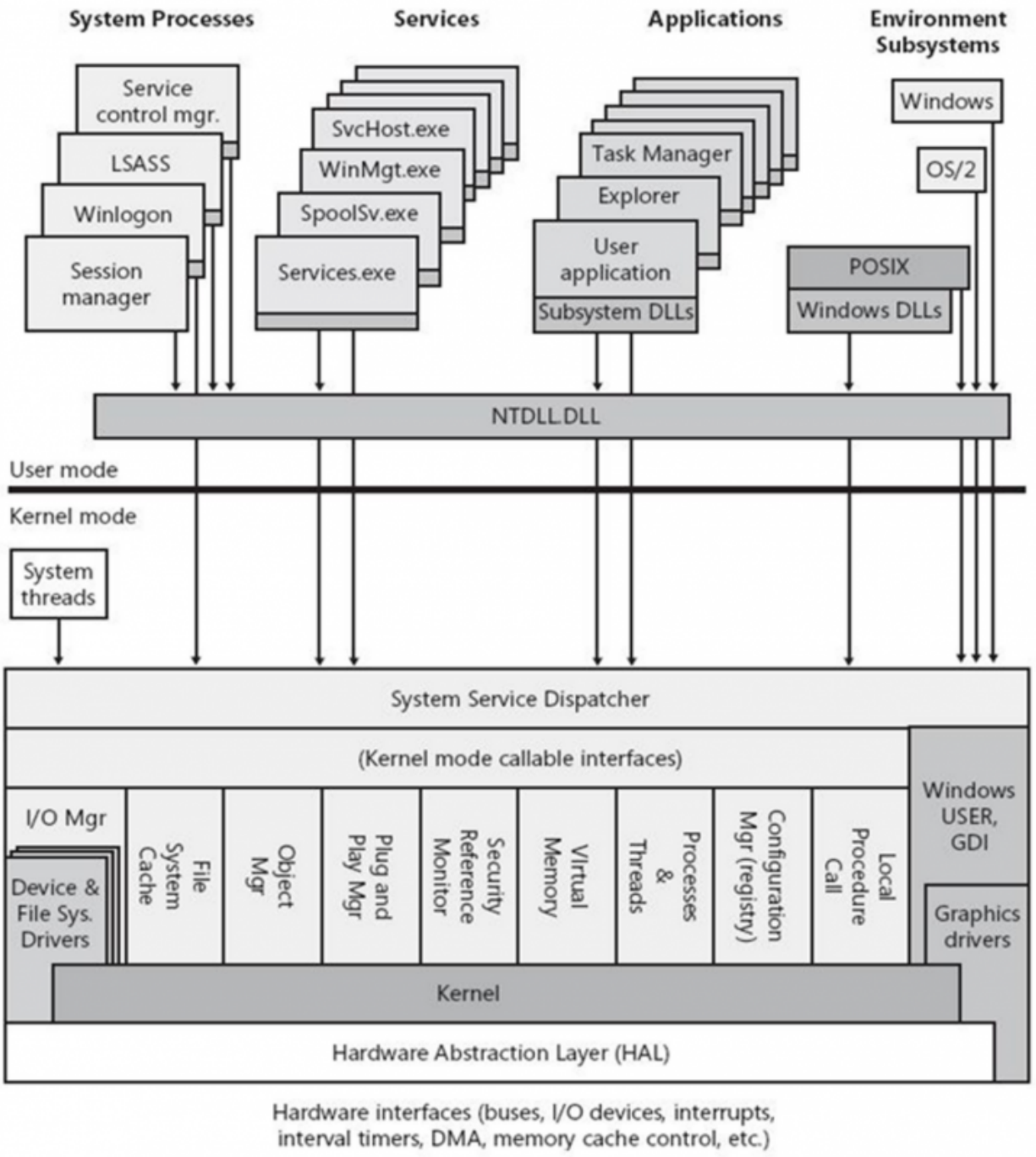
A big majority of user activity will occur at ring 3, User Mode, surprisingly the Kernel operates within Kernel Mode.

For more information on this, see [Windows Programming/User Mode vs Kernel Mode](#). A worthwhile note is that cross-over between user mode and kernel mode can and does happen. The following definitions from the previous link summarise the differences between these layers:

- *Ring 0* (also known as ***kernel mode***) has full access to every resource. It is the mode in which the Windows kernel runs.
- Rings 1 and 2 can be customized with levels of access but are generally unused unless there are virtual machines running.
- *Ring 3* (also known as ***user mode***) has restricted access to resources.

Again, to save this post from being longer than it already is, see the [Overview of Windows Components](#) documentation for more detail on the following diagram. However, its simply showing the Windows architecture from processes, services, etc, crossing over to the Kernel. We will cover more on this shortly.





Reprinted, by permission, from *Inside Microsoft Windows 2000, 3rd Edition* (ISBN 0-7356-1021-5). © 2000 by David A Solomon and Mark E. Russinovich. All rights reserved.

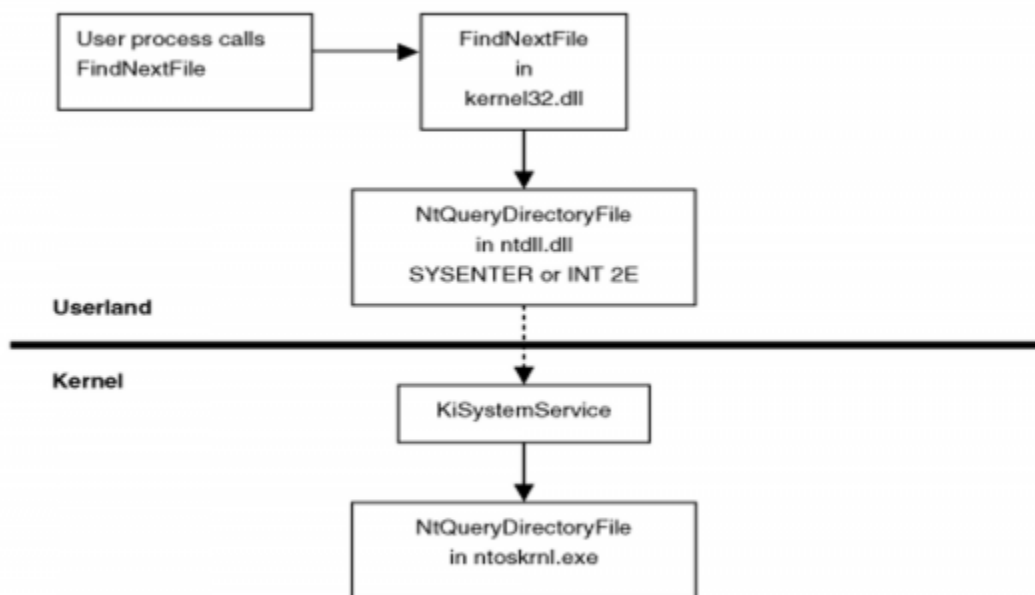
Applications that use the WinAPI will traverse through to the Native API (NTAPI) which operates within Kernel Mode.

As an example, API Monitor can be used to look at the calls being executed:

60	9:14:54.978 PM	1	green-light.exe	CreateThread ( NULL, 0, 0x000001e0c7f30000, NULL, 0, NULL )	0x000000000000...	0.0003487
61	9:14:54.978 PM	1	KERNELBASE.dll	RtlQueryInformationActivationContext ( 1, NULL, NULL, 1, 0x000000822...	STATUS_SUCCESS	0.0000010
62	9:14:54.978 PM	1	KERNELBASE.dll	NtCreateThreadEx ( 0x000000822b91f238, THREAD_ALL_ACCESS, NULL, G.		0.0003426
63	9:14:54.978 PM	1	green-light.exe	WaitForSingleObject ( 0x00000000000000c8, INFINITE )		
64	9:14:54.978 PM	1	KERNELBASE.dll	NtWaitForSingleObject ( 0x00000000000000c8, FALSE, NULL )		
65	9:14:54.978 PM	1	kernel32.dll	FindNextFile ( 0x0000000000000000, NULL, TRUE, ATTACH, NULL )	TRUE	0.0003413

The above shows `CreateThread` being called and then, subsequently, `NtCreateThreadEx` being called shortly after.

When a function within KERNEL32.DLL is called, for example `CreateThread`, it will make a subsequent call to the NTAPI equivalent in NTDLL.DLL. For example, `CreateThread` calls `NtCreateThreadEx`. This function will then fill `RAX` register with the System Service Number (SSN). Finally, NTDLL.dll will then issue a `SYSENTER` instruction. This will then cause the processor to switch to kernel mode, and jumps to a predefined function, called the System Service Dispatcher. The following image is from Rootkits: Subverting the Windows Kernel, in the section on Userland Hooks:



## Drivers

A driver is a software component of Windows which allows the operating system and device to communicate with each other. Here is an example from What is a driver?:

For example, suppose an application needs to read some data from a device. The application calls a function implemented by the operating system, and the operating system calls a function implemented by the driver. The driver, which was written by the same company that designed and manufactured the device, knows how to communicate with the device hardware to get the data. After the driver gets the data from the device, it returns the data to the operating system, which returns it to the application.

In the case of Endpoint Protection, there are a few reasons why drivers are useful:

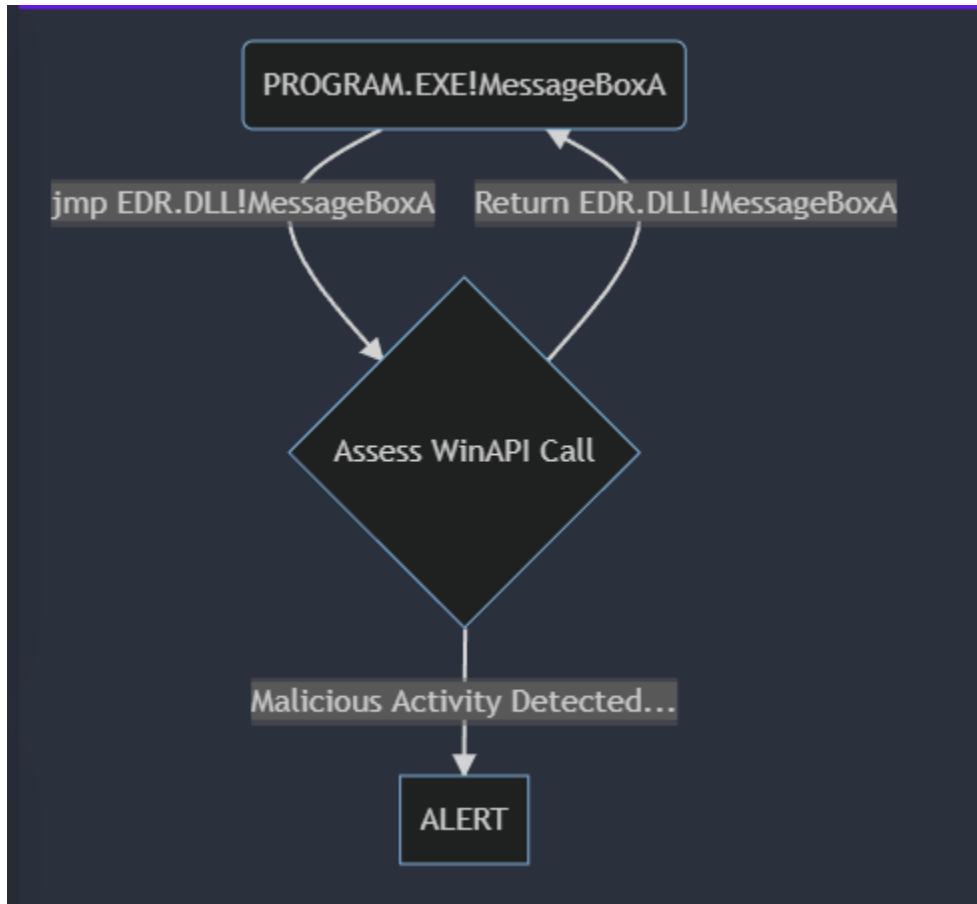
- The use of Callback Objects which allows for a function to be called if an action occurs. For example, later on we will see the usage of PsSetLoadImageNotifyRoutine which is the call-back object for DLLs being loaded.
- Access to privileged information from Event Tracing for Windows Threat Intelligence which is only accessible from the Kernel with an ELAM Driver.

## **Hooks**

---

**DISCLAIMER:** Before moving on, we highly recommend watching REcon 2015 - Hooking Nirvana (Alex Ionescu). Please come back to this post after.

Another common feature of EDR's are the Userland Hooking DLLs. Typically, these are loaded into a process on creation, and are used to proxy WinAPI Calls through themselves to assess the usage, then redirect onto whichever DLL is being used. As an example, if `VirtualAlloc` was being used, the flow would look something like this:



A hook allows for function instrumentation by intercepting WinAPI calls, by placing a `jmp` instruction in place of the function address. This `jmp` will redirect the flow of a call. We will take a look at this in action in the following section. By hooking a function call, it gives the author the ability to:

- Assess arguments
- Allowing Execution
- Blocking Execution

This isn't an exhaustive list, but should serve to demonstrate the functionality which we will be coming across most when running our implants.

Examples of this in use are:

- Understanding and Bypassing AMSI: Bypass AMSI by hooking the `AmsiScanBuffer` call
- RDPThief: Intercept and read credentials from RDP
- Windows API Hooking: Redirect `MessageBoxA`
- Import Address Table (IAT) Hooking: Redirect `MessageBoxA`
- Intercepting Logon Credentials by Hooking `msv1_0!SpAcceptCredentials`: Intercept and read credentials from `msv1_0!SpAcceptCredentials`

- Protecting the Heap: Encryption & Hooks: Hook `RtlAllocateHeap`, `RtlReAllocateHeap` and `RtlFreeHeap` to monitor heap allocations
- LdrLoadDll Hook: Prevent DLLs being loaded

## Hunting ELK

---

To access our kernel callbacks without having to write all of that intimidating logic from scratch, we will be using [the] Hunting ELK (HELK):

The Hunting ELK or simply the HELK is one of the first open source hunt platforms with advanced analytics capabilities such as SQL declarative language, graphing, structured streaming, and even machine learning via Jupyter notebooks and Apache Spark over an ELK stack. This project was developed primarily for research, but due to its flexible design and core components, it can be deployed in larger environments with the right configurations and scalable infrastructure.

We also use the following script is used from Exploring DLL Loads, Links, and Execution to search through the `Sysmon` logs:

```
param (
    [string]$Loader = "",
    [string]$dll = ""
)

$eventId = 7
$logName = "Microsoft-Windows-Sysmon/Operational"

$Yesterday = (Get-Date).AddHours(-1)
$events = Get-WinEvent -FilterHashtable @{logname=$logName; id=$eventId ;StartTime = $Yesterday;}

foreach($event in $events)
{
    $msg = $event.Message.ToString()
    $image = ($msg|Select-String -Pattern 'Image:.*').Matches.Value.Replace("Image: ", "")
    $imageLoaded = ($msg|Select-String -Pattern 'ImageLoaded:.*').Matches.Value.Replace("ImageLoaded: ", "")
    if($image.ToLower().contains($Loader.ToLower()) -And $imageLoaded.ToLower().Contains($dll.ToLower()))
    {
        Write-Host Image Loaded $imageLoaded
    }
}
```

## Kernel Callbacks

---

Kernel Callbacks, according to Microsoft:

The kernel's callback mechanism provides a general way for drivers to request and provide notification when certain conditions are satisfied.

Essentially, they allow drivers to receive and handle notifications for specific events. From [veil-ivy/block\\_create\\_process.cpp](#), here is an implementation of using the

`PsSetLoadImageNotifyRoutine` Callback to BLOCK process creation:

```

#include <ntddk.h>
#define BLOCK_PROCESS "notepad.exe"
static OB_CALLBACK_REGISTRATION obcallback_registration;
static OB_OPERATION_REGISTRATION obooperation_callback;
#define PROCESS_CREATE_THREAD (0x0002)
#define PROCESS_CREATE_PROCESS (0x0080)
#define PROCESS_TERMINATE (0x0001)
#define PROCESS_VM_WRITE (0x0020)
#define PROCESS_VM_READ (0x0010)
#define PROCESS_VM_OPERATION (0x0008)
#define PROCESS_SUSPEND_RESUME (0x0800)
static PVOID registry = NULL;
static UNICODE_STRING altitude = RTL_CONSTANT_STRING(L"300000");
//1: kd > dt nt!_EPROCESS ImageFileName
//+ 0x5a8 ImageFileName : [15] UChar
static const unsigned int imagefilename_offset = 0x5a8;
auto drv_unload(PDRIVER_OBJECT DriverObject) {
    UNREFERENCED_PARAMETER(DriverObject);
    ObUnRegisterCallbacks(registry);
}
OB_PREOP_CALLBACK_STATUS
PreOperationCallback(
    _In_ PVOID RegistrationContext,
    _Inout_ POB_PRE_OPERATION_INFORMATION PreInfo
) {
    UNREFERENCED_PARAMETER(RegistrationContext);

    if (strcmp(BLOCK_PROCESS, (char*)PreInfo->Object + imagefilename_offset) == 0) {
        if ((PreInfo->Operation == OB_OPERATION_HANDLE_CREATE))
        {
            if ((PreInfo->Parameters->CreateHandleInformation.OriginalDesiredAccess &
PROCESS_TERMINATE) == PROCESS_TERMINATE)
            {
                PreInfo->Parameters->CreateHandleInformation.DesiredAccess &=
~PROCESS_TERMINATE;
            }

            if ((PreInfo->Parameters->CreateHandleInformation.OriginalDesiredAccess &
PROCESS_VM_READ) == PROCESS_VM_READ)
            {
                PreInfo->Parameters->CreateHandleInformation.DesiredAccess &=
~PROCESS_VM_READ;
            }

            if ((PreInfo->Parameters->CreateHandleInformation.OriginalDesiredAccess &
PROCESS_VM_OPERATION) == PROCESS_VM_OPERATION)
            {
                PreInfo->Parameters->CreateHandleInformation.DesiredAccess &=
~PROCESS_VM_OPERATION;
            }
        }
    }
}

```

```

        if ((PreInfo->Parameters->CreateHandleInformation.OriginalDesiredAccess &
PROCESS_VM_WRITE) == PROCESS_VM_WRITE)
        {
            PreInfo->Parameters->CreateHandleInformation.DesiredAccess &=
~PROCESS_VM_WRITE;
        }
    }
}

return OB_PREOP_SUCCESS;
}
VOID
PostOperationCallback(
    _In_ PVOID RegistrationContext,
    _In_ POB_POST_OPERATION_INFORMATION PostInfo
)
{
    UNREFERENCED_PARAMETER(RegistrationContext);
    UNREFERENCED_PARAMETER(PostInfo);
}

extern "C" auto DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING
RegistryPath) -> NTSTATUS {
    UNREFERENCED_PARAMETER(RegistryPath);
    DriverObject->DriverUnload = drv_unload;
    auto status = STATUS_SUCCESS;
    static OB_CALLBACK_REGISTRATION ob_callback_register;
    static OB_OPERATION_REGISTRATION oboperation_registration;
    oboperation_registration.Operations = OB_OPERATION_HANDLE_CREATE;
    oboperation_registration.ObjectType = PsProcessType;
    oboperation_registration.PreOperation = PreOperationCallback;
    oboperation_registration.PostOperation = PostOperationCallback;
    ob_callback_register.Altitude = altitude;
    ob_callback_register.Version = OB_FLT_REGISTRATION_VERSION;
    ob_callback_register.OperationRegistrationCount = 1;
    ob_callback_register.OperationRegistration = &oboperation_registration;
    status = ObRegisterCallbacks(&ob_callback_register, &registry);
    if (!NT_SUCCESS(status)) {
        DbgPrint("failed to register callback: %x \r\n",status);
    }
    return status;
}
}

```

In this instance, `ObRegisterCallbacks` is being used to block the creation of `notepad`. An Endpoint Protection solution may not use it in this way, but its very likely this type of callback will be used as telemetry to determine if malicious activity is occurring.

In this section, we are going to discuss `PsSetLoadImageNotifyRoutine`. This callback is responsible for exactly what it says: Sending a notification when an image is loaded into a process. For an example implementation, see [Subscribing to Process Creation, Thread](#)



## Creation and Image Load Notifications from a Kernel Driver.

### **Triggering the callback**

---

To understand how `PsSetLoadImageNotifyRoutine` works, we need to determine what its trigger is.

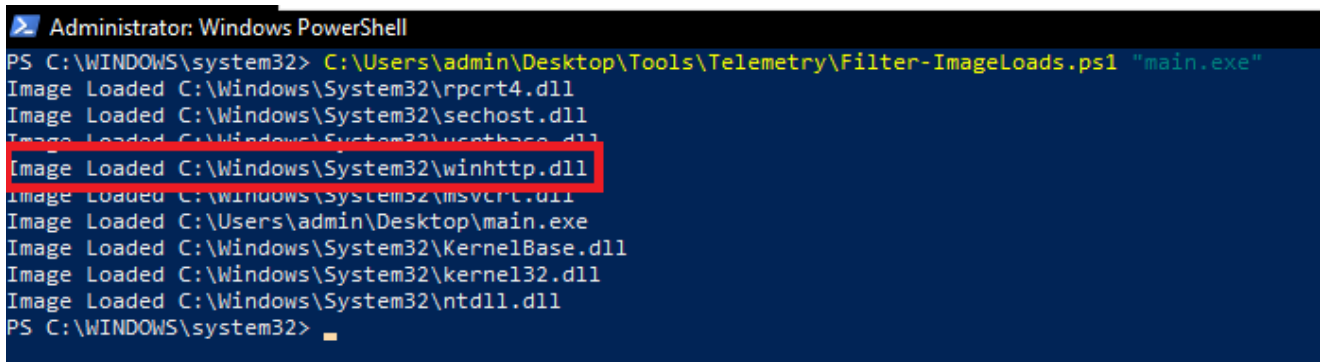
Assuming the following code:

```
#include <windows.h>
#include <stdio.h>

int main()
{
    HMODULE hModule = LoadLibraryA("winhttp.dll");
    printf("WinHTTP: 0x%p\n", hModule);
    return 0;
}
```

When `LoadLibraryA` is called, the function registers a callback to notify the driver that this has happened. In order to see this log in HELK, we use [the script we mentioned earlier on](#).

If we filter for `main.exe`, which is the above code, we can see the `winhttp.dll` loaded:



```
Administrator: Windows PowerShell
PS C:\WINDOWS\system32> C:\Users\admin\Desktop\Tools\Telemetry\FILTER-ImageLoads.ps1 "main.exe"
Image Loaded C:\Windows\System32\rpcrt4.dll
Image Loaded C:\Windows\System32\sechost.dll
Image Loaded C:\Windows\System32\userbase.dll
Image Loaded C:\Windows\System32\winhttp.dll
Image Loaded C:\Windows\System32\msvcrt.dll
Image Loaded C:\Users\admin\Desktop\main.exe
Image Loaded C:\Windows\System32\KernelBase.dll
Image Loaded C:\Windows\System32\kernel32.dll
Image Loaded C:\Windows\System32\ntdll.dll
PS C:\WINDOWS\system32>
```

In Elastic, we can also use the following KQL:

```
process_name : "main.exe" and event_id: 7 and ImageLoaded: winhttp.dll
```

`event_original_message` holds the whole log:

Image loaded:  
RuleName: -  
UtcTime: 2022-04-29 18:50:10.780  
ProcessGuid: {3ebcda8b-3362-626c-a200-000000004f00}  
ProcessId: 6716  
Image: C:\Users\admin\Desktop\main.exe  
ImageLoaded: C:\Windows\System32\winhttp.dll  
FileVersion: 10.0.19041.1620 (WinBuild.160101.0800)  
Description: Windows HTTP Services  
Product: Microsoft® Windows® Operating System  
Company: Microsoft Corporation  
OriginalFileName: winhttp.dll  
Hashes:  
SHA1=4F2A9BB575D38DBDC8DBB25A82BDF1AC0C41E78C, MD5=FB2B6347C25118C3AE19E9903C85B451, SHA  
  
Signed: true  
Signature: Microsoft Windows  
SignatureStatus: Valid  
User: PUNCTURE\admin

To see what this is doing, we can float through the [ReactOS](#) source code:

This is good to get some familiarity with how this would work. However, in [Bypassing Image Load Kernel Callbacks](#), by [batsec](#), identifies that the trigger is in [NtCreateSection](#) call which is then called in the [LdrpCreateDllSection](#). So, we don't need to spend too much time debugging to find this.

## **Spooing Loads**

---

In the article from batsec, they show that the aforementioned events can be spammed with the following code:

```

#include <stdio.h>
#include <windows.h>
#include <winternl.h>

#define DLL_TO_FAKE_LOAD L"\\??\\C:\\windows\\system32\\calc.exe"

BOOL FakeImageLoad()
{
    HANDLE hFile;
    SIZE_T stSize = 0;
    NTSTATUS ntStatus = 0;
    UNICODE_STRING objectName;
    HANDLE SectionHandle = NULL;
    PVOID BaseAddress = NULL;
    IO_STATUS_BLOCK IoStatusBlock;
    OBJECT_ATTRIBUTES objectAttributes = { 0 };

    RtlInitUnicodeString(
        &objectName,
        DLL_TO_FAKE_LOAD
    );

    InitializeObjectAttributes(
        &objectAttributes,
        &objectName,
        OBJ_CASE_INSENSITIVE,
        NULL,
        NULL
    );

    ntStatus = NtOpenFile(
        &hFile,
        0x100021,
        &objectAttributes,
        &IoStatusBlock,
        5,
        0x60
    );

    ntStatus = NtCreateSection(
        &SectionHandle,
        0xd,
        NULL,
        NULL,
        0x10,
        SEC_IMAGE,
        hFile
    );

    ntStatus = NtMapViewOfSection(
        SectionHandle,
        (HANDLE)0xFFFFFFFFFFFFFFFF,

```

```

        &BaseAddress,
        NULL,
        NULL,
        NULL,
        &stSize,
        0x1,
        0x800000,
        0x80
    );

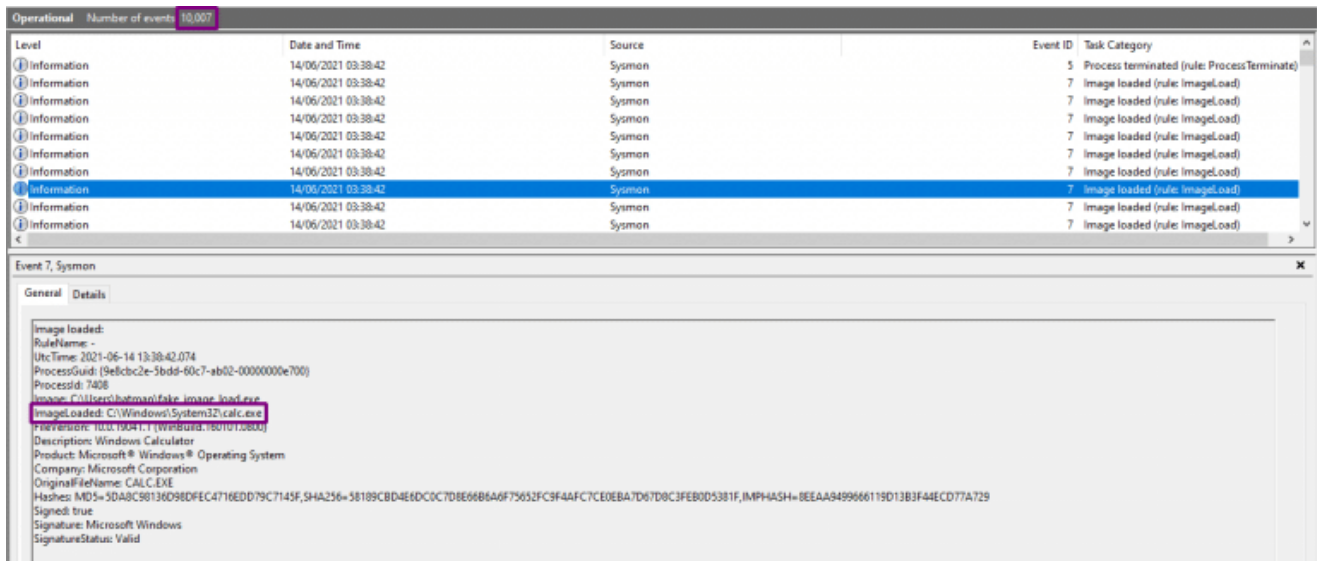
    NtClose(SectionHandle);
}

int main()
{
    for (INT i = 0; i < 10000; i++)
    {
        FakeImageLoad();
    }

    return 0;
}

```

The following screenshot is also from that blog post:



batsec identified that by making the call to `NtCreateSection`, the event can be spammed whilst not actually loading a DLL. Similarly, the spooof can be somewhat weaponised/manipulated to do other things by updating the LDR\_DATA\_TABLE\_ENTRY struct:

```

typedef struct _LDR_DATA_TABLE_ENTRY {
    LIST_ENTRY InLoadOrderLinks;
    LIST_ENTRY InMemoryOrderModuleList;
    LIST_ENTRY InInitializationOrderModuleList;
    PVOID DllBase;
    PVOID EntryPoint;
    ULONG SizeOfImage;
    UNICODE_STRING FullDllName;
    UNICODE_STRING BaseDllName;
    ULONG Flags;
    USHORT LoadCount;
    USHORT TlsIndex;
    union {
        LIST_ENTRY HashLinks;
        struct
        {
            PVOID SectionPointer;
            ULONG CheckSum;
        };
    };
    union {
        ULONG TimeDateStamp;
        PVOID LoadedImports;
    };
    PVOID EntryPointActivationContext;
    PVOID PatchInformation;
} LDR_DATA_TABLE_ENTRY, *PLDR_DATA_TABLE_ENTRY;

```

In this example, we will use `CertEnroll.dll` for no reason at all:

```

UNICODE_STRING uFullPath;
UNICODE_STRING uFileName;

WCHAR* dllPath = L"C:\\Windows\\System32\\CertEnroll.dll";
WCHAR* dllName = L"CertEnroll.dll";

RtlInitUnicodeString(&uFullPath, dllPath);
RtlInitUnicodeString(&uFileName, dllName);

```

Now we just need to step through the struct and fill out the required information.

Load Time:

```
status = NtQuerySystemTime(&pLdrEntry2->LoadTime);
```

Load Reason (LDR\_DLL\_LOAD\_REASON):

```
pLdrEntry2->LoadReason = LoadReasonDynamicLoad;
```

Because the Loader needs a module base address, we'll just load shellcode for `CALC.EXE` here (we'll discuss this part more afterwards):

```

SIZE_T bufSz = sizeof(buf);
LPVOID pAddress = VirtualAllocEx(hProcess, 0, bufSz, MEM_COMMIT | MEM_RESERVE,
PAGE_READWRITE);
memcpy(pAddress, buf, sizeof(buf));

```

### Hashed Base Name (RtlHashUnicodeString):

```
pLdrEntry2->BaseNameHashValue = UnicodeToHash(uFileName, FALSE);
```

Fill out the rest of the struct:

```

pLdrEntry2->ImageDll = TRUE;
pLdrEntry2->LoadNotificationsSent = TRUE;
pLdrEntry2->EntryProcessed = TRUE;
pLdrEntry2->InLegacyLists = TRUE;
pLdrEntry2->InIndexes = TRUE;
pLdrEntry2->ProcessAttachCalled = TRUE;
pLdrEntry2->InExceptionTable = FALSE;
pLdrEntry2->OriginalBase = (ULONG_PTR)pAddress;
pLdrEntry2->DllBase = pAddress;
pLdrEntry2->SizeOfImage = 6969;
pLdrEntry2->TimeDateStamp = 0;
pLdrEntry2->BaseDllName = uFileName;
pLdrEntry2->FullDllName = uFullPath;
pLdrEntry2->ObsoleteLoadCount = 1;
pLdrEntry2->Flags = LDRP_IMAGE_DLL | LDRP_ENTRY_INSERTED | LDRP_ENTRY_PROCESSED |
LDRP_PROCESS_ATTACH_CALLED;

```

### Complete the **DdagNode** struct:

The screenshot shows the Windows Task Manager interface. The 'Processes' tab is active, and 'spooof-load.exe' is selected. Below it, the 'spooof-load.exe (4364) Properties' window is open, showing the 'Modules' tab. The loaded modules are listed in a table:

Name	Base address	Size	Description
apphelp.dll	0x7ffa9803...	576 kB	Application Compatibility Cle...
CertEnroll.dll	0x150000	6,81 kB	Microsoft® Active Directory...
kernel32.dll	0x7ffa9c35...	760 kB	Windows NT BASE API Clie...
KernelBase.dll	0x7ffa9ae2...	2,78 MB	Windows NT BASE API Clie...
locale.nls	0x70000	804 kB	
msvrt.dll	0x7ffa9b29...	632 kB	Windows NT CRT DLL
ntdll.dll	0x7ffa9d09...	1,96 MB	NT Layer DLL
<b>spooof-load.exe</b>	<b>0x400000</b>	<b>68 kB</b>	

Overlaid on the bottom right of the Properties window is a black terminal window with the following output:

```

C:\Users\admin\Desktop\spooof-load.exe
[*] Path: C:\Windows\System32\CertEnroll.dll
[*] File: CertEnroll.dll
[*] Setting DLL Load Time
[*] Setting DLL Load Reason to LoadReasonDynamicLoad
[+] Base Address: 0x000000000150000
[*] Written 277 bytes to 0x000000000150000!
[*] Configuring DdagNode
Inspect me! (press any key to finish inject)

```

```

pLdrEntry2->DdagNode = (PLDR_DDAG_NODE)HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY,
sizeof(LDR_DDAG_NODE));
if (!pLdrEntry2->DdagNode)
{
    return -1;
}

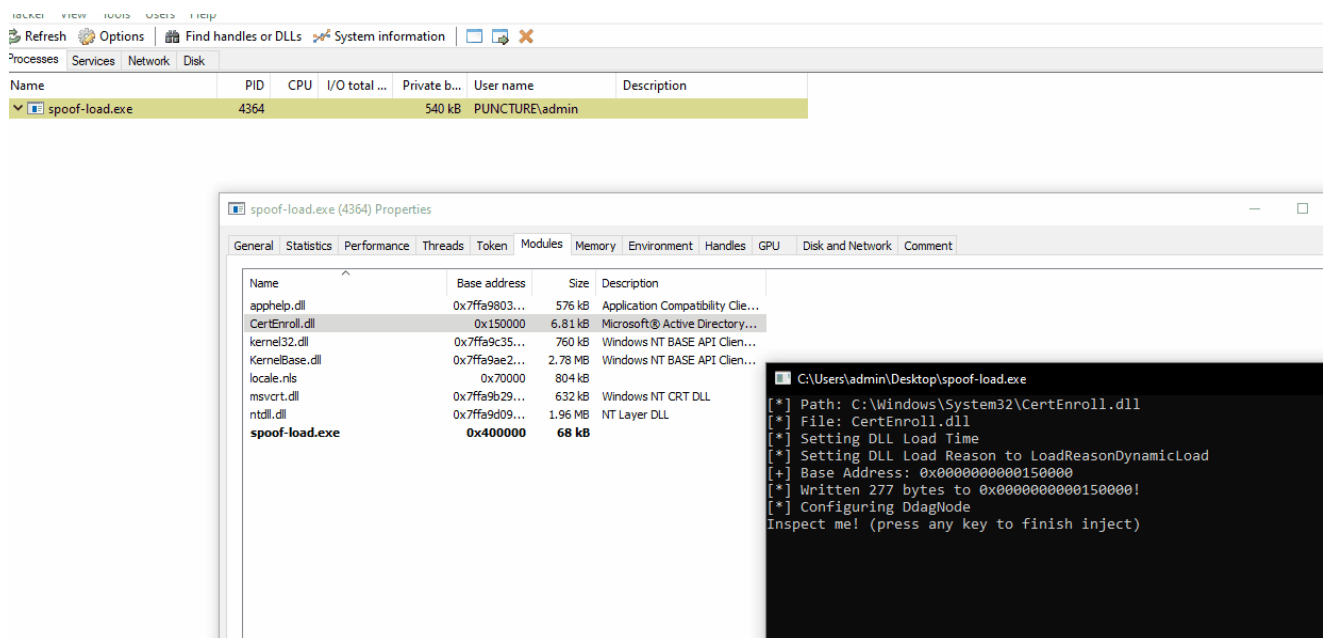
```

```

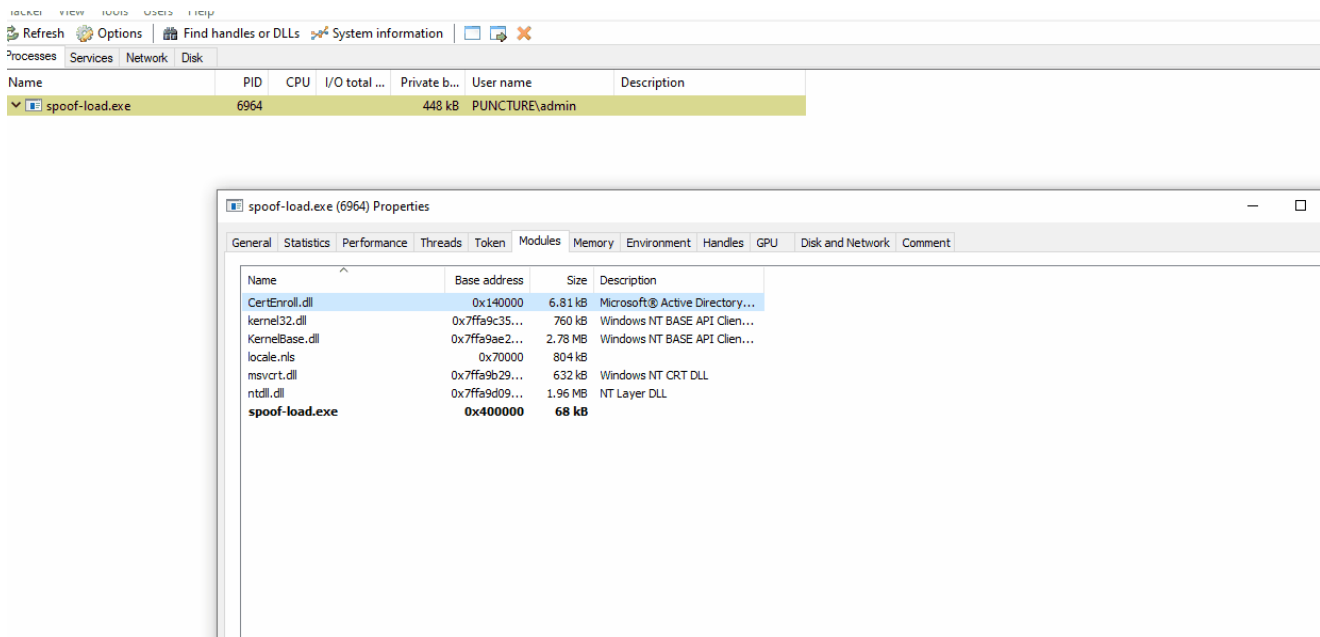
pLdrEntry2->NodeModuleLink.Flink = &pLdrEntry2->DdagNode->Modules;
pLdrEntry2->NodeModuleLink.Blink = &pLdrEntry2->DdagNode->Modules;
pLdrEntry2->DdagNode->Modules.Flink = &pLdrEntry2->NodeModuleLink;
pLdrEntry2->DdagNode->Modules.Blink = &pLdrEntry2->NodeModuleLink;
pLdrEntry2->DdagNode->State = LdrModulesReadyToRun;
pLdrEntry2->DdagNode->LoadCount = 1;

```

Here is it in action:



In the above, `CertEnroll.dll` can be seen loaded in the `spoof-load.exe` process. Remember, this is not loaded. The only thing that happened here is that a string for that DLL was passed in. We then told the loader than the base address of the DLL is that of the shellcode:



Looking at this technique, there are two obvious use cases:

- Tie the implant base address (C2IMPLANT.REFLECTIVE.DLL) to a legitimate DLL (ADVAP32.DLL) causing it to appear less suspicious
- Remove an IOC Library (WinHTTP.DLL) by loading ADVAPI32.DLL but pointing it to a WinHTTP.DLL base address.

## Bypassing the Callback

We aren't going to reinvent the wheel here, its explained wonderfully in [Bypassing Image Load Kernel Callbacks](#). Essentially, to cause the callback to not trigger, a full loader needs to be rewritten. The conclusion to that research was [DarkLoadLibrary](#):

In essence, `DarkLoadLibrary` is an implementation of `LoadLibrary` that will not trigger image load events. It also has a ton of extra features that will make life easier during malware development.

A proof-of-concept usage of this library was taken from [DLL Shenanigans](#).

Let's inspect it:

```

Administrator: C:\Users\admin\Desktop\Tools\PowerShell\x64\powershell.x64.exe
PS C:\WINDOWS\system32> C:\Users\admin\Desktop\Tools\dark-loader.exe 1 C:\Windows\System32\winhttp.dll
|> Loading: C:\Windows\System32\winhttp.dll
:: Setting: LOAD_LOCAL_FILE
:: Letting DarkLoadLibrary read from disk!
:: Module: 0000000176080000
:: Freed DarkModule->ErrorMsg
:: Freed DarkModule
PS C:\WINDOWS\system32> C:\Users\admin\Desktop\Tools\Telemetry\Filter-ImageLoads.ps1 "dark-loader.exe" "kernel32.dll"
Image Loaded C:\Windows\System32\kernel32.dll
Image Loaded C:\Windows\System32\kernel32.dll
Image Loaded C:\Windows\System32\kernel32.dll
PS C:\WINDOWS\system32> C:\Users\admin\Desktop\Tools\Telemetry\Filter-ImageLoads.ps1 "dark-loader.exe" "winhttp.dll"
PS C:\WINDOWS\system32>

```



Then the above 3 commands are ran:

- `dark-loader` uses the `LOAD_LOCAL_FILE` flag to load a disk from disk, as `LoadLibraryA` does.
- The Image Load logs are searched for Kernel32 to make sure logs were found.
- `winhttp.dll` was searched, and nothing returned

To avoid the call to `NtCreateSection` which was identified to be registering the callback, the section mapping is done with `NtAllocateVirtualMemory` or `VirtualAlloc`, as seen in [MapSections\(\)](#).

## **Kernel Callback Conclusion**

---

Obviously, `PsSetLoadImageNotifyRoutine` is not the only callback, and there are quite a few other callbacks readily available. [Kernel Callback Functions](#) has a (non-comprehensive!) list:

- `CmRegisterCallbackEx()`
- `ExAllocateTimer()`
- `ExInitializeWorkItem()`
- `ExRegisterCallback()`
- `FsRtlRegisterFileSystemFilterCallbacks()`
- `IoInitializeThreadedDpcRequest()`
- `IoQueueWorkItem()`
- `IoRegisterBootDriverCallback()`
- `IoRegisterContainerNotification()`
- `IoRegisterFsRegistrationChangeEx()`
- `IoRegisterFsRegistrationChangeMountAware()`
- `IoRegisterPlugPlayNotification()`
- `IoSetCompletionRoutineEx()`
- `IoWMISetNotificationCallback()`
- `KeExpandKernelStackAndCalloutEx()`
- `KeInitializeApc()`
- `KeInitializeDpc()`
- `KeRegisterBugCheckCallback()`
- `KeRegisterBugCheckReasonCallback()`
- `KeRegisterNmiCallback()`
- `KeRegisterProcessorChangeCallback()`
- `KeRegisterProcessorChangeCallback()`
- `ObRegisterCallbacks()`
- `PoRegisterDeviceNotify()`
- `PoRegisterPowerSettingCallback()`
- `PsCreateSystemThread()`

- `PsSetCreateProcessNotifyRoutineEx()`
- `PsSetCreateThreadNotifyRoutine()`
- `PsSetLoadImageNotifyRoutine()`
- `SeRegisterLogonSessionTerminatedRoutine()`
- `TmEnableCallbacks()`

One that would be powerful would be `PsSetCreateProcessNotifyRoutineEx()` as the notification for process creation would be crippling for system telemetry. At the time of writing, we are not aware of any research in this space. Although to be totally honest, we haven't looked.

## **Hooking and Process Instrumentation**

---

In this section, we are going to look at some popular, but elementary, hooking techniques.

### **Hooking Example**

---

Lets look at two examples before looking into some libraries - Manual Hooks in x86 and NtSetProcessInformation Callbacks.

#### **Manual Hooks (x86)**

---

Using [Windows API Hooking](#) as a x86 example (easier to demonstrate), we can adapt the code to look something like this:

```

#include <windows.h>
#include <stdio.h>

#define BYTES_REQUIRED 6

int __stdcall HookedMessageBoxA(HWND hWnd, LPCSTR lpText, LPCSTR lpCaption, UINT
uType)
{
    printf("\n[ HOOKED MESSAGEBOXA ]\n");
    printf("-> Arguments:\n");
    printf(" 1. lpText: %s\n", lpText);
    printf(" 2. lpCaption: %s\n", lpCaption);
    printf(" 3. uType: %ld\n", uType);
    return 1;
}

void PrintHexA(char* data, int sz)
{
    printf(" -> ");
    for (int i = 0; i < sz; i++)
    {
        printf("\\x%02hhX", data[i]);
    }

    printf("\n");
}

int main()
{
    SIZE_T lpNumberOfBytesRead = 0;
    HMODULE hModule = nullptr;
    FARPROC pMessageBoxAFunc = nullptr;
    char pMessageBoxABytes[BYTES_REQUIRED] = {};

    void* pHookedMessageBoxFunc = &HookedMessageBoxA;

    hModule = LoadLibraryA("user32.dll");
    if (!hModule)
    {
        return -1;
    }

    pMessageBoxAFunc = GetProcAddress(hModule, "MessageBoxA");

    printf("-> Original MessageBoxA: 0x%p\n", pMessageBoxAFunc);

    if (ReadProcessMemory(GetCurrentProcess(), pMessageBoxAFunc, pMessageBoxABytes,
BYTES_REQUIRED, &lpNumberOfBytesRead) == FALSE)
    {
        printf("[!] ReadProcessMemory: %ld\n", GetLastError());
        return -1;
    }
}

```

```

}

printf("-> MessageBoxA Hex:\n");

PrintHexA(pMessageBoxABytes, BYTES_REQUIRED);

printf("-> Hooked MessageBoxA: 0x%p\n", pHookedMessageBoxFunc);

char patch[BYTES_REQUIRED] = { 0 };
memcpy_s(patch, 1, "\x68", 1);
memcpy_s(patch + 1, 4, &pHookedMessageBoxFunc, 4);
memcpy_s(patch + 5, 1, "\xC3", 1);

printf("-> Patch Hex:\n");
PrintHexA(patch, BYTES_REQUIRED);

if (WriteProcessMemory(GetCurrentProcess(), (LPVOID)pMessageBoxAFunc, patch,
sizeof(patch), &lpNumberOfBytesRead) == FALSE)
{
    printf("[!] WriteProcessMemory: %ld\n", GetLastError());
    return -1;
}

MessageBoxA(NULL, "AAAAA", "BBBBB", MB_OK);

return 0;
}

```

Lets walk through this...

First off, `MessageBoxA` is in `User32.dll` so we load that:

```

hModule = LoadLibraryA("user32.dll");
if (!hModule)
{
    return -1;
}

```

Next, we need the address of `USER32!MessageBoxA` :

```
pMessageBoxAFunc = GetProcAddress(hModule, "MessageBoxA");
```

With that address, the bytes can now be read:

```

if (ReadProcessMemory(GetCurrentProcess(), pMessageBoxAFunc, pMessageBoxABytes,
BYTES_REQUIRED, &lpNumberOfBytesRead) == FALSE)
{
    printf("[!] ReadProcessMemory: %ld\n", GetLastError());
    return -1;
}

```

This will read the first 6 bytes of the function call which will later be updated to hold a `push` to the new function, resulting in a `jmp` .

The bytes:

```
\x8B\xff\x55\x8B\xEC\x83
```

Now, the patch needs to be built. This is done like so:

```
char patch[BYTES_REQUIRED] = { 0 };
memcpy_s(patch, 1, "\x68", 1);
memcpy_s(patch + 1, 4, &pHookedMessageBoxFunc, 4);
memcpy_s(patch + 5, 1, "\xc3", 1);
```

The hex produced from this:

```
\x68\x12\x12\xBD\x00\xc3
```

Using [defuse.ca](#) to disassemble this, the above can be translated into Assembly:

```
0: 68 12 12 bd 00      push  0xbd1212
5: c3                  ret
```

Note that `0x00BD1212` being pushed is the address of the function we want to jump to INSTEAD of the `USER32!MessageBoxA` call:

```
void* pHookedMessageBoxFunc = &HookedMessageBoxA;
```

At this point, the patch is prepared. It's going to replace the first 6 bytes with a `push` to the new address.

The next thing is to actually write this new address in:

```
if (WriteProcessMemory(GetCurrentProcess(), (LPVOID)pMessageBoxAFunc, patch,
sizeof(patch), &lpNumberOfBytesRead) == FALSE)
{
    printf("[!] WriteProcessMemory: %ld\n", GetLastError());
    return -1;
}
```

Then, in the disassembly:

```
00BB1212 jmp HookedMessageBoxA (0BB1A80h)
```

A `jmp` is added to jump to the new function. Allowing this to run calls the hooked function and the arguments are printed:

```

int __stdcall HookedMessageBoxA(HWND hwnd, LPCSTR lpText, LPCSTR lpCaption, UINT
uType)
{
    printf("\n[ HOOKED MESSAGEBOXA ]\n");
    printf("-> Arguments:\n");
    printf(" 1. lpText: %s\n", lpText);
    printf(" 2. lpCaption: %s\n", lpCaption);
    printf(" 3. uType: %ld\n", uType);
    return 1;
}

```

Running it:

```

Microsoft Visual Studio Debug Console
-> Original MessageBoxA: 0x75780A50
-> MessageBoxA Hex:
-> \x8B\xff\x55\x8B\xEC\x83
-> Hooked MessageBoxA: 0x00BB1212
-> Patch Hex:
-> \x68\x12\x12\xBB\x00\xC3

[ HOOKED MESSAGEBOXA ]
-> Arguments:
 1. lpText: Not Hooked
 2. lpCaption: Not Hooked
 3. uType: 0

```

## NtSetProcessInformation Callbacks

---

Setting up the callback is straight forward:

```

PROCESS_INSTRUMENTATION_CALLBACK_INFORMATION InstrumentationCallbackInfo;
InstrumentationCallbackInfo.Version = 0;
InstrumentationCallbackInfo.Reserved = 0;
InstrumentationCallbackInfo.Callback = CALLBACK_FUNCTION_GOES_HERE;
HANDLE hProcess = (HANDLE)-1;

HMODULE hNtdll = GetModuleHandleA("ntdll");
if (hNtdll == nullptr)
{
    return FALSE;
}

_NtSetInformationProcess pNtSetInformationProcess =
reinterpret_cast<_NtSetInformationProcess>(GetProcAddress(hNtdll,
"NtSetInformationProcess"));

if (pNtSetInformationProcess == nullptr)
{
    return FALSE;
}

NTSTATUS Status = pNtSetInformationProcess(hProcess,
(PROCESS_INFORMATION_CLASS)ProcessInstrumentationCallback,
&InstrumentationCallbackInfo, sizeof(InstrumentationCallbackInfo));
if (NT_SUCCESS(Status))
{
    return TRUE;
}
else
{
    return FALSE;
}

```

Where the callback function is included as follows:

```
InstrumentationCallbackInfo.Callback = CALLBACK_FUNCTION_GOES_HERE;
```

`CALLBACK_FUNCTION_GOES_HERE` is a function to use as the callback and then `ProcessInstrumentationCallback` is:

```
#define ProcessInstrumentationCallback 0x28
```

An additional point is that by setting the callback to `NULL`, any callbacks sent will be removed. This was documented by [modexp](#) in [Bypassing User-Mode Hooks and Direct Invocation of System Calls for Red Teams](#).

This talk was then built on by [Secrary](#) and again in Secrary's blog [Hooking via InstrumentationCallback](#). The original code from Alex Ionescu can be found in the [HookingNirvana](#) repo.

Borrowing the hooks from Secrary gives access to the function and return value, giving us the following Assembly:

```
.code

PUBLIC asmCallback
EXTERN Hook:PROC

asmCallback PROC
    push rax ; return value
    push rcx
    push RBX
    push RBP
    push RDI
    push RSI
    push RSP
    push R12
    push R13
    push R14
    push R15

    ; without this it crashes :)
    sub rsp, 1000h
    mov rdx, rax
    mov rcx, r10
    call Hook
    add rsp, 1000h

    pop R15
    pop R14
    pop R13
    pop R12
    pop RSP
    pop RSI
    pop RDI
    pop RBP
    pop RBX
    pop rcx
    pop rax

    jmp R10
asmCallback ENDP

end
```

**Hook:** With the assembly written, we also need to write the function called by the assembly, allowing us to take in all of the provided registers and return their function names:



```

DWORD64 counter = 0;
bool flag = false;

EXTERN_C VOID Hook(DWORD64 R10, DWORD64 RAX/* ... */) {

    // This flag is there for prevent recursion
    if (!flag)
    {
        flag = true;

        counter++;

        CHAR buffer[sizeof(SYMBOL_INFO) + MAX_SYM_NAME] = { 0 };
        PSYMBOL_INFO pSymbol = (PSYMBOL_INFO)buffer;
        pSymbol->SizeOfStruct = sizeof(SYMBOL_INFO);
        pSymbol->MaxNameLen = MAX_SYM_NAME;
        DWORD64 Displacement;

        // MSDN: Retrieves symbol information for the specified address.
        BOOLEAN result = SymFromAddr(GetCurrentProcess(), R10, &Displacement,
pSymbol);

        if (result) {
            printf("%s => 0x%llx\n", pSymbol->Name, RAX);
        }

        flag = false;
    }
}

```

Then, in `main`, `SymInitialize` is called, then the instrumentation is set:

```

int main()
{
    SymSetOptions(SYMOPT_UNDNAME);
    SymInitialize(GetCurrentProcess(), NULL, TRUE);

    SetInstrumentationCallback();

    return 0;
}

```

Running this completed example, we can now see all of the function names and return codes:

The hook could be updated to get access to the arguments for a full analysis, but we didn't feel the need to look into that for this initial proof-of-concept.

One final mention for this technique is that it can be used to enumerate the the System Service Number (SSN) for a given function call. This was documented by [Paranoid Ninja in EtwTi-Syscall-Hook](#) and [Release vo.8 - Warfare Tactics](#), where the hook is significantly smaller (at the cost of doing far less):

```

VOID HuntSyscall(ULONG_PTR
ReturnAddress, ULONG_PTR
retSyscallPtr) {
    PVOID ImageBase = ((EtwPPEB)
(((_EtwPTEB)(NtCurrentTeb()-
>ProcessEnvironmentBlock))))-
>ImageBaseAddress;
    PIMAGE_NT_HEADERS NtHeaders =
RtlImageNtHeader(ImageBase);
    if (ReturnAddress >=
(ULONG_PTR)ImageBase &&
ReturnAddress <
(ULONG_PTR)ImageBase + NtHeaders-
>OptionalHeader.SizeOfImage) {
        printf("[+] Syscall
detected: Return address: 0x%X
Syscall value: 0x%X\n",
ReturnAddress, retSyscallPtr);
    }
}

```

And its companion assembly:

```

section .text

extern HuntSyscall
global hookedCallback

hookedCallback:
    push rcx
    push rdx
    mov rdx, [r10-0x10]
    call HuntSyscall
    pop rdx
    pop rcx
    ret

```

```

Microsoft Visual Studio Debug Console
ZwQueryInformationThread => 0x0
ZwOpenSection => 0xc0000034
NtQueryAttributesFile => 0x0
ZwOpenFile => 0x0
ZwCreateSection => 0x0
ZwMapViewOfSection => 0x0
NtQueryPerformanceCounter => 0x0
NtProtectVirtualMemory => 0x0
NtProtectVirtualMemory => 0x0
NtProtectVirtualMemory => 0x0
NtProtectVirtualMemory => 0x0
ZwOpenSection => 0x0
ZwMapViewOfSection => 0x0
NtQueryPerformanceCounter => 0x0
NtProtectVirtualMemory => 0x0
NtProtectVirtualMemory => 0x0
NtProtectVirtualMemory => 0x0
NtReleaseWorkerFactoryWorker => 0x0
NtClose => 0x0
NtProtectVirtualMemory => 0x0
NtClose => 0x0
NtClose => 0x0
ZwApphelpCacheControl => 0x0
ZwApphelpCacheControl => 0x0
ZwSetInformationVirtualMemory => 0xc00000bb
NtAllocateVirtualMemory => 0x0
NtFreeVirtualMemory => 0x0
NtAllocateVirtualMemory => 0x0
NtQueryVolumeInformationFile => 0x0
NtQueryVolumeInformationFile => 0x0
NtQueryVolumeInformationFile => 0x0
NtAllocateVirtualMemory => 0x0
NtAllocateVirtualMemory => 0x0
NtOpenKey => 0x0
NtQueryValueKey => 0xc0000034
NtClose => 0x0
NtAllocateVirtualMemory => 0x0
NtAllocateVirtualMemory => 0x0
NtAllocateVirtualMemory => 0x0
NtAllocateVirtualMemory => 0x0
ZwSetInformationVirtualMemory => 0xc00000bb
ZwSetEvent => 0x0
ZwQuerySecurityAttributesToken => 0xc0000225
ZwQuerySecurityAttributesToken => 0xc0000225
NtTerminateProcess => 0x0
NtClose => 0x0
NtClose => 0x0
NtClose => 0x0

```

## Bypassing Userland Hooks

Back in 2019 [Cneelis](#) published [Red Team Tactics: Combining Direct System Calls and sRDI to bypass AV/EDR](#) which had a subsequent release of [SysWhispers](#):

SysWhispers provides red teamers the ability to generate header/ASM pairs for any system call in the core kernel image (ntoskrnl.exe). The headers will also include the necessary type definitions.

Then modexp provided an update which corrected a shortcoming with version 1 and gave us SysWhispers2:

The specific implementation in SysWhispers2 is a variation of @modexpblog's code. One difference is that the function name hashes are randomized on each generation. @ElephantSe4l, who had published this technique earlier, has another implementation based in C++17 which is also worth checking out.

The main change is the introduction of base.c which is a result of Bypassing User-Mode Hooks and Direct Invocation of System Calls for Red Teams.

And again, KlezVirus produced SysWhispers3:

The usage is pretty similar to SysWhispers2, with the following exceptions:

- It also supports x86/WoW64
- It supports syscalls instruction replacement with an EGG (to be dynamically replaced)
- It supports direct jumps to syscalls in x86/x64 mode (in WOW64 it's almost standard)
- It supports direct jumps to random syscalls (borrowing @ElephantSeal's idea)

A better explanation of these features are better outlined in the blog post SysWhispers is dead, long live SysWhispers!

This is just one suite of SysCall techniques, there's a whole other technique based on Heavens Gate.

See Gatekeeping Syscalls for a breakdown on these different techniques.

EVEN THEN! There are more:

- FreshyCalls
- EtwTi-Syscall-Hook
- FireWalker

RECAP!

With the ability to transition into Kernel-Mode, we have the ability to go unseen by the User-land hooks. So, lets build something.

For our example, we are going to use MinHook:

The Minimalistic x86/x64 API Hooking Library for Windows

## The DLL

---

So, this is going to be a DLL which gets loaded into a process and then hooks functionality and makes some decision based on its behaviour. Here is `DllMain` :

```
BOOL APIENTRY DllMain(HINSTANCE hInst, DWORD reason, LPVOID reserved)
{
    switch (reason)
    {
    case DLL_PROCESS_ATTACH:
    {
        HANDLE hThread = CreateThread(nullptr, 0, SetupHooks, nullptr, 0, nullptr);
        if (hThread != nullptr) {
            CloseHandle(hThread);
        }
        break;
    }
    case DLL_PROCESS_DETACH:
        break;
    }
    return TRUE;
}
```

When a `DLL_PROCESS_ATTACH` is the load reason, then we create a new thread and point it at our "main" function. This is where we initialise minhook, and set up some hooks:

```

DWORD WINAPI SetupHooks(LPVOID param)
{
    MH_STATUS status;

    if (MH_Initialize() != MH_OK) {
        return -1;
    }

    status = MH_CreateHookApi(
        L"ntdll",
        "NtAllocateVirtualMemory",
        NtAllocateVirtualMemory_Hook,
        reinterpret_cast<LPVOID*>(&pNtAllocateVirtualMemory_Original)
    );

    status = MH_CreateHookApi(
        L"ntdll",
        "NtProtectVirtualMemory",
        NtProtectVirtualMemory_Hook,
        reinterpret_cast<LPVOID*>(&pNtProtectVirtualMemory_Original)
    );

    status = MH_CreateHookApi(
        L"ntdll",
        "NtWriteVirtualMemory",
        NtWriteVirtualMemory_Hook,
        reinterpret_cast<LPVOID*>(&pNtWriteVirtualMemory_Original)
    );

    status = MH_EnableHook(MH_ALL_HOOKS);

    return status;
}

```

`MH_Initialize()` is a mandatory call, so we start with that. Next, we create 3 hooks:

- `NtAllocateVirtualMemory`
- `NtProtectVirtualMemory`
- `NtWriteVirtualMemory`

Hooks are created with the `MH_CreateHookApi()` call:

```

MH_STATUS WINAPI MH_CreateHookApi(LPCWSTR pszModule, LPCSTR pszProcName, LPVOID
pDetour, LPVOID *ppOriginal);

```

To create a hook, 4 things are needed:

- Module Name
- Function Name
- Function to "replace" the desired function

- Somewhere to store the original function address

Below is an example:

```
MH_STATUS status = MH_CreateHookApi(
    L"ntdll",
    "NtAllocateVirtualMemory",
    NtAllocateVirtualMemory_Hook,
    reinterpret_cast<LPVOID*>(&pNtAllocateVirtualMemory_Original)
);
```

`NtAllocateVirtualMemory_Hook()` is the function used to replace the original function:

```
NTSTATUS NTAPI NtAllocateVirtualMemory_Hook(IN HANDLE ProcessHandle, IN OUT PVOID*
BaseAddress, IN ULONG_PTR ZeroBits, IN OUT PSIZE_T RegionSize, IN ULONG
AllocationType, IN ULONG Protect)
{
    if (Protect == PAGE_EXECUTE_READWRITE)
    {
        printf("[INTERCEPTOR]: RWX Allocation Detected in %ld (0x%p)\n",
GetProcessId(ProcessHandle), ProcessHandle);
        if (BLOCKING)
        {
            return 5;
        }
        else
        {
            return pNtAllocateVirtualMemory_Original(ProcessHandle, BaseAddress,
ZeroBits, RegionSize, AllocationType, Protect);
        }
    }
    else
    {
        return pNtAllocateVirtualMemory_Original(ProcessHandle, BaseAddress,
ZeroBits, RegionSize, AllocationType, Protect);
    }
}
```

The function is declared exactly the same as `typedef` for the function:

```
typedef NTSTATUS(NTAPI* _NtAllocateVirtualMemory)(IN HANDLE ProcessHandle, IN OUT
PVOID* BaseAddress, IN ULONG_PTR ZeroBits, IN OUT PSIZE_T RegionSize, IN ULONG
AllocationType, IN ULONG Protect);
```

This is so that there are no issues with typing between hooks.

In the `NtAllocateVirtualMemory_Hook` function, the only thing we are checking here is if the protection type is `PAGE_EXECUTE_READWRITE`, `RWX`, because this is commonly a sign of malicious activity (COMMONLY). If it matches, we just print that we found something.

Then, we have a concept of blocking. This simply means that if `BLOCKING` is true, then it returns. If its false, then we return the pointer to the original function, allowing the function to execute as the user expects.

In `NtProtectVirtualMemory`, we just check for changes to `PAGE_EXECUTE_READ` as this is the common protection type to avoid RWX allocations:

```
NTSTATUS NTAPI NtProtectVirtualMemory_Hook(IN HANDLE ProcessHandle, IN OUT PVOID*
BaseAddress, IN OUT PULONG NumberOfBytesToProtect, IN ULONG NewAccessProtection, OUT
PULONG OldAccessProtection) {

    if (NewAccessProtection == PAGE_EXECUTE_READ) {
        printf("[INTERCEPTOR]: Detected move to RX in %ld (0x%p)\n",
GetProcessId(ProcessHandle), ProcessHandle);
        if (BLOCKING)
        {
            return 5;
        }
        else
        {
            return pNtProtectVirtualMemory_Original(ProcessHandle, BaseAddress,
NumberOfBytesToProtect, NewAccessProtection, OldAccessProtection);
        }
    }
    else
    {
        return pNtProtectVirtualMemory_Original(ProcessHandle, BaseAddress,
NumberOfBytesToProtect, NewAccessProtection, OldAccessProtection);
    }
}
```

In `NtWriteVirtualMemory`, no additional checks are made:

```
NTSTATUS NTAPI NtWriteVirtualMemory_Hook(IN HANDLE ProcessHandle, IN PVOID
BaseAddress, IN PVOID Buffer, IN SIZE_T NumberOfBytesToWrite, OUT PSIZE_T
NumberOfBytesWritten OPTIONAL)
{
    printf("[INTERCEPTOR]: Detected write of %I64u in %ld (0x%p)\n",
NumberOfBytesToWrite, GetProcessId(ProcessHandle), ProcessHandle);
    if (BLOCKING)
    {
        return 5;
    }
    else
    {
        return pNtWriteVirtualMemory_Original(ProcessHandle, BaseAddress, Buffer,
NumberOfBytesToWrite, NumberOfBytesWritten);
    }
}
```

## **The Loader**

---

In this instance, we have a PE which just calls `LoadLibraryA` on the DLL, and then runs a fake injection:

```
#include <Windows.h>
#include <stdio.h>

int main()
{
    HMODULE hModule = LoadLibraryA("Interceptor.dll");

    if (hModule == nullptr)
    {
        printf("[LOADER] [LOADER] Failed to load: %ld\n", GetLastError());
        return -1;
    }
    printf("[LOADER] Interceptor.dll: 0x%p\n", hModule);

    Sleep(3000);

    CHAR buf[8] = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 };

    LPVOID pAddress = VirtualAlloc(nullptr, 8, MEM_COMMIT | MEM_RESERVE,
PAGE_EXECUTE_READWRITE);
    if (pAddress == nullptr)
    {
        printf("[LOADER] VirtualAlloc: %ld\n", GetLastError());
        return -1;
    }
    printf("[LOADER] Base: 0x%p\n", pAddress);

    if (WriteProcessMemory((HANDLE)-1, pAddress, buf, sizeof buf, nullptr) == FALSE)
    {
        printf("[LOADER] WriteProcessMemory: %ld\n", GetLastError());
        return -1;
    }
    printf("[LOADER] Wrote!\n");

    if (VirtualProtect(pAddress, sizeof buf, PAGE_EXECUTE_READ, nullptr) == FALSE)
    {
        printf("[LOADER] VirtualProtect: %ld\n", GetLastError());
        return -1;
    }
    printf("[LOADER] Protected!\n");

    return 0;
}
```

## **Detecting Functionality**

---

Running this shows the calls being detected (in a non-blocking mode):



```

Microsoft Visual Studio Debug Console

[LOADER] Interceptor.dll: 0x00007FFFD91C0000
[INTERCEPTOR]: Detected move to RX in 10552 (0xFFFFFFFFFFFFFFFF)
[INTERCEPTOR]: Detected move to RX in 10552 (0xFFFFFFFFFFFFFFFF)
[INTERCEPTOR]: RWX Allocation Detected in 10552 (0xFFFFFFFFFFFFFFFF)
[LOADER] Base: 0x0000020AE4010000
[INTERCEPTOR]: Detected write of 8 in 10552 (0xFFFFFFFFFFFFFFFF)
[LOADER] Wrote!
[INTERCEPTOR]: Detected move to RX in 10552 (0xFFFFFFFFFFFFFFFF)
[LOADER] VirtualProtect: 998

```

In the screenshot, we can see:

- Moves to RX
- RWX Allocations
- Writes of 8 bytes

This is everything we planned on detecting. So, how would a bypass work here? Well, because of a lot of community development, its quite easy in practice. But before that, we need to discuss User-land and Kernel-land.

### Bypassing the User-land hooks

For this example, we are going to use [Tartarus Gate](#). All we have to do is make this one call in `wmain` :

```
LoadLibraryA("Interceptor.dll");
```

And then change the payload in `Payload` :

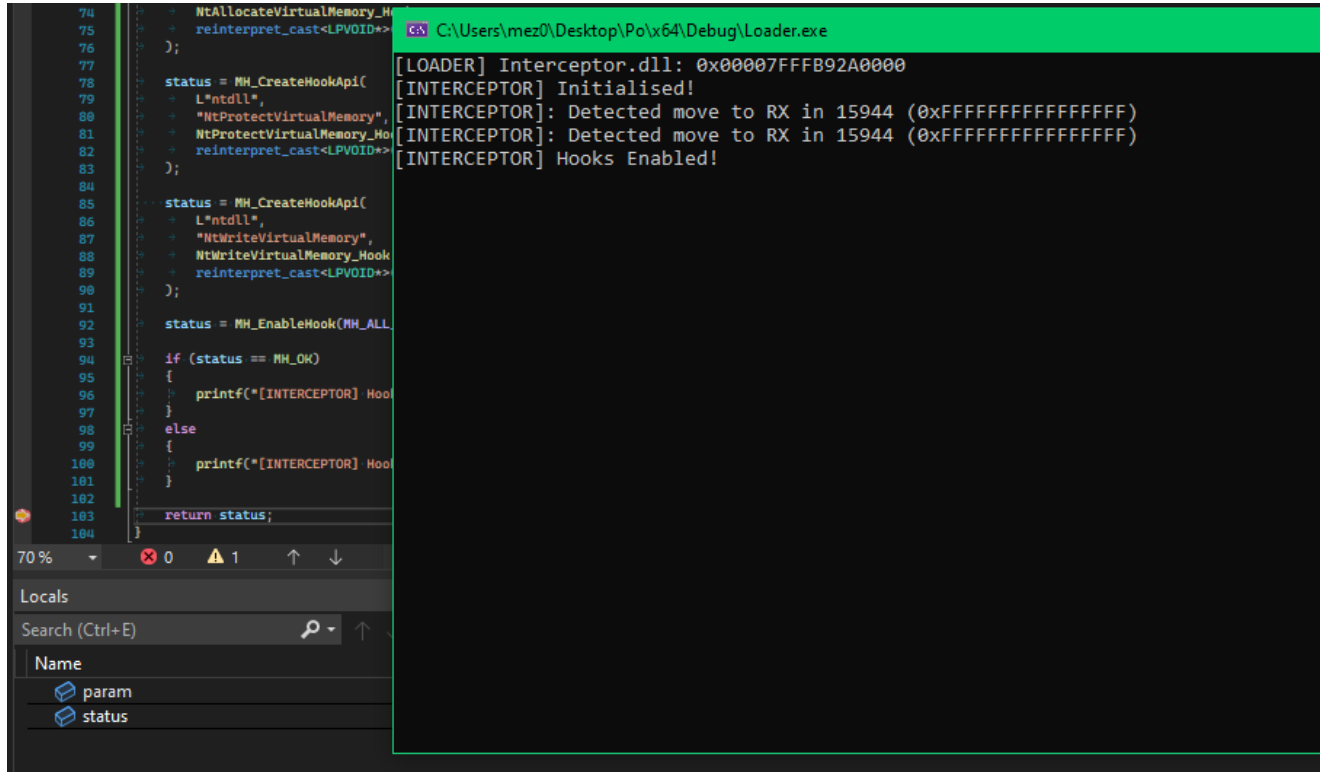
```
unsigned char payload[] = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00 };
```

Checking the loaded modules:

Name	Base address	Size	Description	Load reason
<b>HellsGate.exe</b>	<b>0x7ff6f7c00000</b>	<b>160 kB</b>		<b>Dynamic</b>
apphelp.dll	0x7ff841620000	576 kB	Application Compatibility Clie...	Dynamic
Interceptor.dll	0x7fff901e0000	504 kB		Dynamic
kernel32.dll	0x7ff845310000	756 kB	Windows NT BASE API Clie...	Dynamic
KernelBase.dll	0x7ff843f80000	2.8 MB	Windows NT BASE API Clie...	Static dependency
locale.nls	0x1e63b8d0000	804 kB		
ntdll.dll	0x7ff8466b0000	1.96 MB	NT Layer DLL	Static dependency
ucrtbased.dll	0x7fff90260000	2.12 MB	Microsoft® C Runtime Library	Static dependency
vcruntime140d.dll	0x7fff95970000	172 kB	Microsoft® C Runtime Library	Static dependency

The DLL is loaded..

Running it, and setting a breakpoint on the thread creation because the payload is junk:



The screenshot shows a WinDbg session with a C++ loader. The code on the left includes hooks for `NtAllocateVirtualMemory`, `NtProtectVirtualMemory`, and `NtWriteVirtualMemory` using `MinHook`. The debugger console on the right shows the following output:

```
C:\Users\mez0\Desktop\Po\x64\Debug\Loader.exe
[LOADER] Interceptor.dll: 0x00007FFF92A0000
[INTERCEPTOR] Initialised!
[INTERCEPTOR]: Detected move to RX in 15944 (0xFFFFFFFFFFFFFFFF)
[INTERCEPTOR]: Detected move to RX in 15944 (0xFFFFFFFFFFFFFFFF)
[INTERCEPTOR] Hooks Enabled!
```

The above shows minhook being initialised, and then the hooks being enabled. Between this, there is a move to RX. However, it happens before the hooks are set up. So this is likely either minhook, or CRT doing something. We did not take the time to check this out.

## Hooking and Process Instrumentation Conclusion

As we've stated, this isn't a comprehensive review of every potential technique. Another which might specifically be worth exploring is Vectored Exception Handling (VEH). Two examples of this are [ethicalchaos's post on In-Process Patchless AMSI Bypass](#) and [Countercept's CallStackSpoofers](#).

As with Kernel callbacks, this is a live field of study and there's far more to be explored than we have time or space in this post.

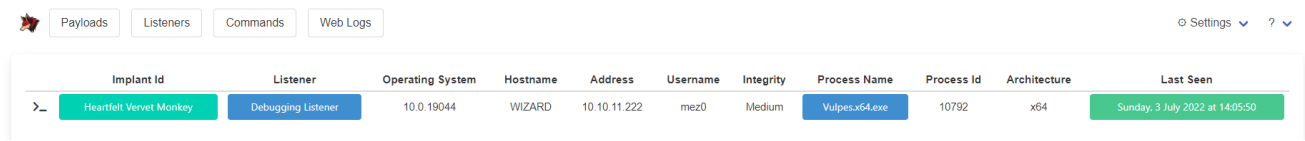
## Thread Call Stacks

Another component of a process which gets interrogated is via the Threads Call Stack. As detailed in Viewing the Call Stack in WinDbg, the Call Stack is defined as:

The call stack is the chain of function calls that have led to the current location of the program counter. The top function on the call stack is the current function, the next function is the function that called the current function, and so on. The call stack that is displayed is based on the current program counter, unless you change the register context. For more information about how to change the register context, see [Changing Contexts](#).

As the call stack can help determine the intention of a thread, it often undergoes scrutiny to determine its validity. In this section, we want to demonstrate how the call stack can be used to determine malicious behaviour (in a rudimentary example), and then discuss the offensive strategy for handling this.

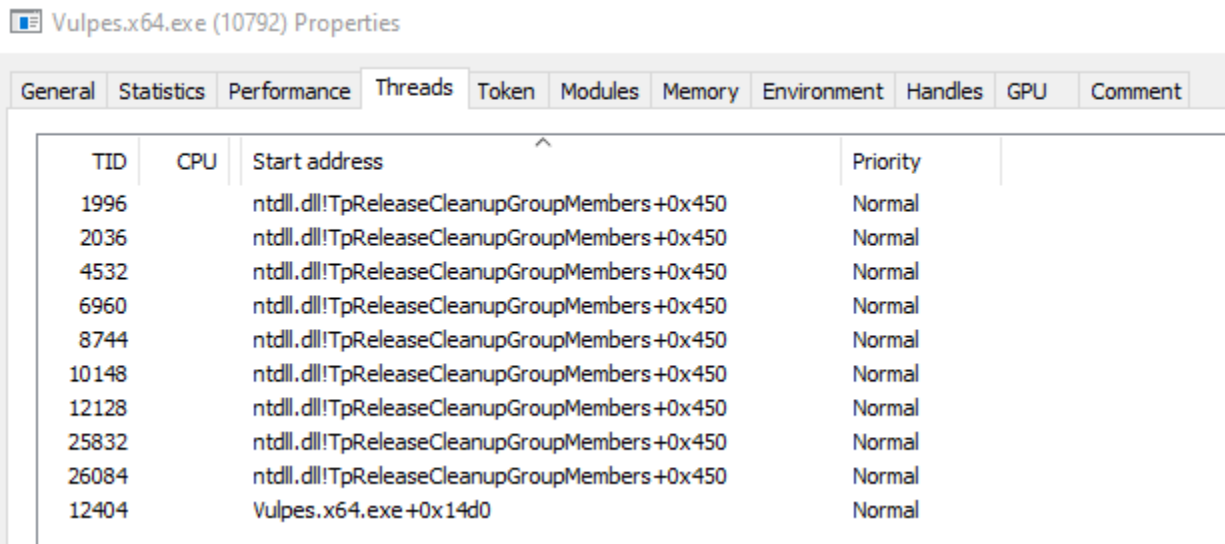
Here, we have a an implant in Vulpes:



The screenshot shows the Vulpes interface with a table of implants. The selected implant is 'Heartfelt Velvet Monkey' with a 'Debugging Listener'.

Implant Id	Listener	Operating System	Hostname	Address	Username	Integrity	Process Name	Process Id	Architecture	Last Seen
Heartfelt Velvet Monkey	Debugging Listener	10.0.19044	WIZARD	10.10.11.222	mez0	Medium	Vulpes.x64.exe	10792	x64	Sunday, 3 July 2022 at 14:05:50

If we look at the processes (10792) threads, we can see a bunch of threads starting at the elusive TpReleaseCleanupGroupMembers:



The screenshot shows the 'Threads' tab for 'Vulpes.x64.exe (10792) Properties'. The table lists several threads starting at the same address in ntdll.dll.

TID	CPU	Start address	Priority
1996		ntdll.dll!TpReleaseCleanupGroupMembers+0x450	Normal
2036		ntdll.dll!TpReleaseCleanupGroupMembers+0x450	Normal
4532		ntdll.dll!TpReleaseCleanupGroupMembers+0x450	Normal
6960		ntdll.dll!TpReleaseCleanupGroupMembers+0x450	Normal
8744		ntdll.dll!TpReleaseCleanupGroupMembers+0x450	Normal
10148		ntdll.dll!TpReleaseCleanupGroupMembers+0x450	Normal
12128		ntdll.dll!TpReleaseCleanupGroupMembers+0x450	Normal
25832		ntdll.dll!TpReleaseCleanupGroupMembers+0x450	Normal
26084		ntdll.dll!TpReleaseCleanupGroupMembers+0x450	Normal
12404		Vulpes.x64.exe+0x14d0	Normal

This is quite common amongst processes, here is an example of `chrome.exe` :

TID	CPU	Start address	Priority
29108		ntdll.dll!TpReleaseCleanupGroupMembers+0x450	Normal
21508		ntdll.dll!TpReleaseCleanupGroupMembers+0x450	Normal
21340		ntdll.dll!TpReleaseCleanupGroupMembers+0x450	Normal
17300		ntdll.dll!TpReleaseCleanupGroupMembers+0x450	Normal
16620		ntdll.dll!TpReleaseCleanupGroupMembers+0x450	Normal
25036		directmanipulation.dll+0x14350	Normal
10720		chrome.exe!Ordinal0+0x4ba70	Normal
17780		chrome.exe!GetHandleVerifier+0x79ef0	Normal
28676		chrome.dll!IsSandboxedProcess+0x7e9d80	Normal
28612		chrome.dll!IsSandboxedProcess+0x7e9d80	-4
27592	0.02	chrome.dll!IsSandboxedProcess+0x7e9d80	Normal
26676		chrome.dll!IsSandboxedProcess+0x7e9d80	Normal
24112		chrome.dll!IsSandboxedProcess+0x7e9d80	Normal
23948		chrome.dll!IsSandboxedProcess+0x7e9d80	Normal
21364		chrome.dll!IsSandboxedProcess+0x7e9d80	Normal
20964		chrome.dll!IsSandboxedProcess+0x7e9d80	Normal
19856		chrome.dll!IsSandboxedProcess+0x7e9d80	Normal
18852		chrome.dll!IsSandboxedProcess+0x7e9d80	-4
16836		chrome.dll!IsSandboxedProcess+0x7e9d80	Normal
16760		chrome.dll!IsSandboxedProcess+0x7e9d80	Normal
14956		chrome.dll!IsSandboxedProcess+0x7e9d80	Normal
9508		chrome.dll!IsSandboxedProcess+0x7e9d80	Normal
9320		chrome.dll!IsSandboxedProcess+0x7e9d80	Normal
9104		chrome.dll!IsSandboxedProcess+0x7e9d80	Normal
8868		chrome.dll!IsSandboxedProcess+0x7e9d80	Normal
7276		chrome.dll!IsSandboxedProcess+0x7e9d80	-4
6460		chrome.dll!IsSandboxedProcess+0x7e9d80	Normal

Start module:

Started: N/A

State: N/A      Priority: N/A

Kernel time: N/A      Base priority: N/A

User time: N/A      I/O priority: N/A

Context switches: N/A      Page priority: N/A

Cycles: N/A      Ideal processor: N/A

And then `RuntimeBroker.exe` :

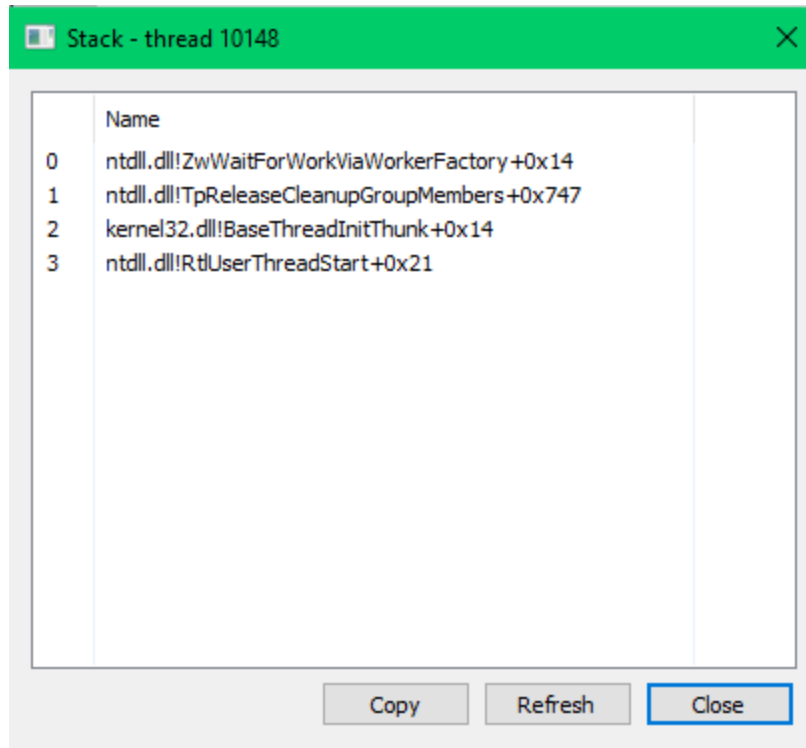
TID	CPU	Start address	Priority
19176		LockAppBroker.dll+0xdfc0	Normal
1808		ntdll.dll!TpReleaseCleanupGroupMembers+0x450	Normal
20596		RuntimeBroker.exe+0x6740	Normal
356		TetheringStation.dll!TetheringStationFreeMemory+0xf720	Normal
6656		wlanapi.dll!WlanQueryVirtualInterfaceType+0xca30	Normal
18848		wlanapi.dll!WlanQueryVirtualInterfaceType+0xca30	Normal
19052		wlanapi.dll!WlanQueryVirtualInterfaceType+0xca30	Normal

This is a good side-note for attackers. If the implant in question is reliant on masquerading as something else, then this needs to be considered. For example, if the implant is operating out of browsers such as chrome, then the HTTP should be handled the same way, and then entry-point and call stack of the thread should be mimicked.

Back to the Vulpes implant, the call stack is primarily `TpReleaseCleanupGroupMembers` which is fine. However, if we go through some of the threads, here is the thread responsible for WinHTTP:

	Name
0	ntdll.dll!NtDelayExecution+0x14
1	KernelBase.dll!SleepEx+0x9e
2	winhttp.dll!WinHttpFreeProxyResultEx+0xb155
3	winhttp.dll!WinHttpFreeProxyResultEx+0xb034
4	ntdll.dll!TpSimpleTryPost+0x2dc
5	ntdll.dll!TpReleaseCleanupGroupMembers+0x8a6
6	kernel32.dll!BaseThreadInitThunk+0x14
7	ntdll.dll!RtlUserThreadStart+0x21

And here is a generic thread started by the process:



There are a few others, but let's focus on the second example because this is a call stack for a thread that will be found in a lot of processes. Let's look at how to programmatically read the thread stack and how a spoofed thread base address can look suspicious.

Here is the entry point:

```
int main()
{
    DWORD dwProcessId = 10792;
    DWORD dwSussThread = 1996;
    DWORD dwNormalThread = 26084;

    HANDLE hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, dwProcessId);

    SymInitialize(hProcess, NULL, TRUE);

    StackWalkThread(hProcess, dwSussThread);

    SymCleanup(hProcess);
}
```

In the above, we have two thread IDs.

- **1996** : The spoofed thread
- **26084** : A somewhat normal stack

With that, we need to write a function to enumerate the call stack of the thread. We can do that with the following code:

```

void StackWalkThread(HANDLE hProcess, DWORD dwThreadId)
{
    STACKFRAME64 frame = { 0 };
    CONTEXT context    = { 0 };
    int idx           = 0;

    HANDLE hThread = OpenThread(MAXIMUM_ALLOWED, FALSE, dwThreadId);

    if (!hThread) return;

    context.ContextFlags = CONTEXT_FULL;

    if (GetThreadContext(hThread, &context) == FALSE) return;

    frame.AddrPC.Offset = context.Rip;
    frame.AddrPC.Mode = AddrModeFlat;
    frame.AddrStack.Offset = context.Rsp;
    frame.AddrStack.Mode = AddrModeFlat;
    frame.AddrFrame.Offset = context.Rbp;
    frame.AddrFrame.Mode = AddrModeFlat;

    printf("# Thread: %ld\n\n", dwThreadId);

    while (StackWalk64(IMAGE_FILE_MACHINE_AMD64, hProcess, hThread, &frame, &context,
    NULL, SymFunctionTableAccess64, SymGetModuleBase64, NULL))
    {
        DWORD64 moduleBase = SymGetModuleBase64(hProcess, frame.AddrPC.Offset);
        DWORD64 offset = 0;
        char symbolBuff[sizeof(SYMBOL_INFO) + MAX_SYM_NAME * sizeof(TCHAR)] = { 0 };
        PSYMBOL_INFO symbol = (PSYMBOL_INFO)symbolBuff;
        symbol->SizeOfStruct = sizeof(SYMBOL_INFO);
        symbol->MaxNameLen = MAX_SYM_NAME;

        if (SymFromAddr(hProcess, frame.AddrPC.Offset, &offset, symbol))
        {
            printf(
                "\\_ Frame %d\n"
                "  |_ Name: %s\n"
                "  |_ Address: 0x%p\n\n",
                idx,
                symbol->Name,
                symbol->Address
            );

            idx++;
        }
    }
}

```

From the [DbgHelp](#) library, we are using:

- StackWalk64: Obtains a stack trace.
- SyGetModuleBase64: Retrieves the base address of the module that contains the specified address.
- SymFromAddr: Retrieves symbol information for the specified address.

Pointing the code to the normal thread stack:

This matches what we saw earlier on. Changing this to point to the *bad* thread:

This now shows a different thread stack we haven't seen so far. Looking at frame 3, its CreateTimeQueueTimer which was described as the sleep obfuscation technique in:

As a disclaimer, this technique has full kudos to Peter Winter-Smith.

So, programmatically, its easy to find out the callstack of a thread. Let's expand this into something completely rudimentary that we can start to work with.

First, we'll define a hard-coded list of expected functions that we saw earlier on that we can use as an *integrity* check:

```
std::vector < std::string > expected = {
    "ZwWaitForWorkViaWorkerFactory",
    "TpReleaseCleanupGroupMembers",
    "BaseThreadInitThunk",
    "RtlUserThreadStart"
};
```

And then an empty one, to track everything we find:

```
std::vector<std::string> found;
```

Now, instead of just printing, lets add all the symbol names into a vector:

```
Microsoft Visual Studio Debug Console
# Thread: 26084
\ Frame 0
|_ Name: ZwWaitForWorkViaWorkerFactory
|_ Address: 0x00007FFD47FD0780
\ Frame 1
|_ Name: TpReleaseCleanupGroupMembers
|_ Address: 0x00007FFD47F82680
\ Frame 2
|_ Name: BaseThreadInitThunk
|_ Address: 0x00007FFD45FA7020
\ Frame 3
|_ Name: RtlUserThreadStart
|_ Address: 0x00007FFD47F82630
```

```
Microsoft Visual Studio Debug Console
# Thread: 1996
\ Frame 0
|_ Name: NtWaitForSingleObject
|_ Address: 0x00007FFD47FCCDB0
\ Frame 1
|_ Name: WaitForSingleObjectEx
|_ Address: 0x00007FFD45941A40
\ Frame 2
|_ Name: NtDuplicateToken
|_ Address: 0x00007FFD47FCD570
\ Frame 3
|_ Name: CreateTimerQueueTimer
|_ Address: 0x00007FFD45FAE2A0
```



```

while (StackWalk64(IMAGE_FILE_MACHINE_AMD64, hProcess, hThread, &frame, &context,
NULL, SymFunctionTableAccess64, SymGetModuleBase64, NULL))
{
    DWORD64 moduleBase = SymGetModuleBase64(hProcess, frame.AddrPC.Offset);
    DWORD64 offset = 0;
    char symbolBuff[sizeof(SYMBOL_INFO) + MAX_SYM_NAME * sizeof(TCHAR)] = { 0 };
    PSYMBOL_INFO symbol = (PSYMBOL_INFO)symbolBuff;
    symbol->SizeOfStruct = sizeof(SYMBOL_INFO);
    symbol->MaxNameLen = MAX_SYM_NAME;

    if (SymFromAddr(hProcess, frame.AddrPC.Offset, &offset, symbol))
    {
        found.push_back(symbol->Name);
        printf(
            "\\_ Frame %d\\n"
            "  |_ Name: %s\\n"
            "  |_ Address: 0x%p\\n\\n",
            idx,
            symbol->Name,
            symbol->Address
        );

        idx++;
    }
}

```

Once the code has ran, and found all the symbols, lets see if the vectors match:

```

if (std::equal(expected.begin(), expected.end(), found.begin()))
{
    printf("[ CLEAN ]\\n");
}
else
{
    printf("[ DIRTY ]\\n");
}

```

Pointing this at the *good* thread:

We get the **CLEAN** message. And then the *dirty* thread:

Obviously, this code isn't production ready and the nuances of writing this kind of logic properly is extremely challenging. However, it is something that some EDR vendors are starting to pick up. Given the increase into research to confuse and blind endpoint protection, this is a good technique to have in the arsenal for both the blue and red teams.

Speaking of red teams, research into correcting this thread-mishap has already been ongoing.

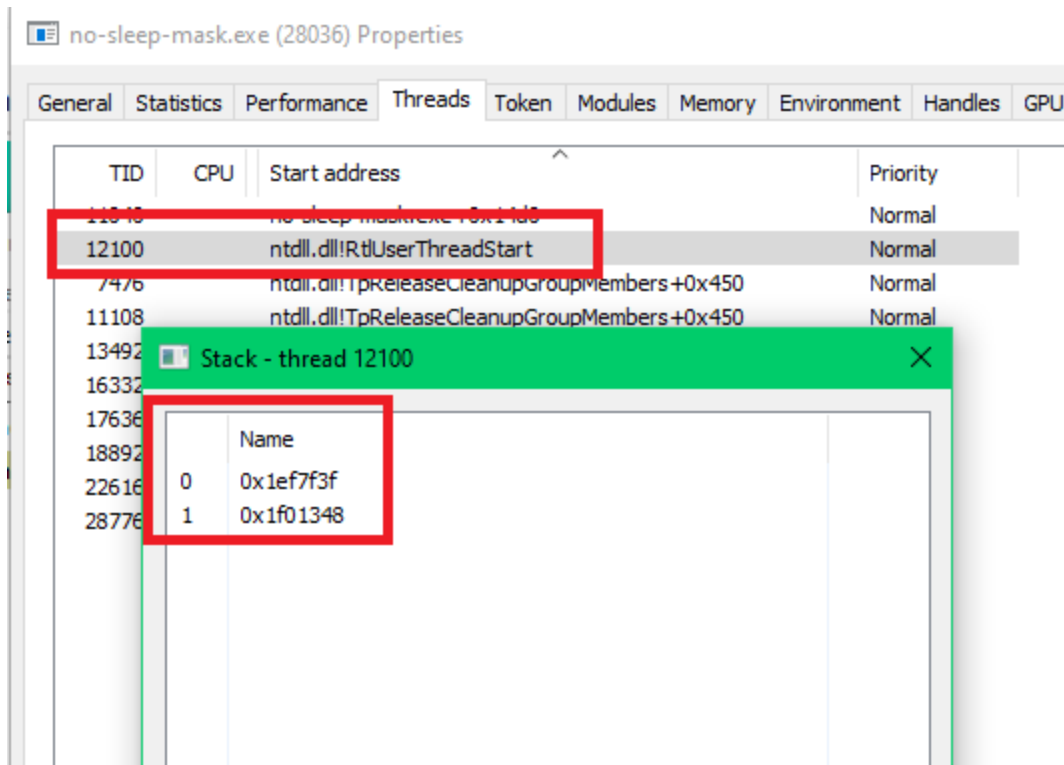
This technique was first popularized by [Peter Winter-Smith](#), who is a common reoccurrence in this space, and then reinterpreted by [mgeeky](#) in [ThreadStackSpoof](#). However, this proof-of-concept sets the return address to 0, removing references to memory addresses for shellcode injection.

This is an example implementation for *Thread Stack Spoofing* technique aiming to evade Malware Analysts, AVs and EDRs looking for references to shellcode's frames in an examined thread's call stack. The idea is to hide references to the shellcode on thread's call stack thus masquerading allocations containing malware's code.

If we remove the sleep masking from Vulpes, here is how the call stack looks:

```
Microsoft Visual Studio Debug Console
# Thread: 26084
\__ Frame 0
|_ Name: ZwWaitForWorkViaWorkerFactory
|_ Address: 0x00007FFD47FD0780
\__ Frame 1
|_ Name: TpReleaseCleanupGroupMembers
|_ Address: 0x00007FFD47F82680
\__ Frame 2
|_ Name: BaseThreadInitThunk
|_ Address: 0x00007FFD45FA7020
\__ Frame 3
|_ Name: RtlUserThreadStart
|_ Address: 0x00007FFD47F82630
[ CLEAN ]
```

```
Microsoft Visual Studio Debug Console
# Thread: 1996
\__ Frame 0
|_ Name: NtWaitForSingleObject
|_ Address: 0x00007FFD47FCCDB0
\__ Frame 1
|_ Name: WaitForSingleObjectEx
|_ Address: 0x00007FFD45941A40
\__ Frame 2
|_ Name: NtDuplicateToken
|_ Address: 0x00007FFD47FCD570
\__ Frame 3
|_ Name: CreateTimerQueueTimer
|_ Address: 0x00007FFD45FAE2A0
[ DIRTY ]
```



The technique would aim to mask these addresses by storing the return address into a variable, setting the return address to 0, and then restoring the return address.

For a quick code example from the above repository:

```
void WINAPI MySleep(DWORD _dwMilliseconds)
{
    [...]
    auto overwrite = (PULONG_PTR)_AddressOfReturnAddress();
    const auto origReturnAddress = *overwrite;
    *overwrite = 0;

    [...]
    *overwrite = origReturnAddress;
}
```

In a more recent project, [CallStackSpoof](#) by [William Burgess](#) was produced to take this a step further and fully mask the stack.

See: [Spoofing Call Stacks to confused EDRs](#)

By using predefined vectors of stacks, the project is able to mimic:

- WMI
- RPC
- SVCHost

An example of a [predefined vector for WMI](#):

```

std::vector<StackFrame> wmiCallStack =
{
    StackFrame(L"C:\\Windows\\SYSTEM32\\kernelbase.dll", 0x2c13e, 0, FALSE),

    StackFrame(L"C:\\Windows\\Microsoft.NET\\Framework64\\v4.0.30319\\CorperfmonExt.dll",
0xc669, 0, TRUE),

    StackFrame(L"C:\\Windows\\Microsoft.NET\\Framework64\\v4.0.30319\\CorperfmonExt.dll",
0xc71b, 0, FALSE),

    StackFrame(L"C:\\Windows\\Microsoft.NET\\Framework64\\v4.0.30319\\CorperfmonExt.dll",
0x2fde, 0, FALSE),

    StackFrame(L"C:\\Windows\\Microsoft.NET\\Framework64\\v4.0.30319\\CorperfmonExt.dll",
0x2b9e, 0, FALSE),

    StackFrame(L"C:\\Windows\\Microsoft.NET\\Framework64\\v4.0.30319\\CorperfmonExt.dll",
0x2659, 0, FALSE),

    StackFrame(L"C:\\Windows\\Microsoft.NET\\Framework64\\v4.0.30319\\CorperfmonExt.dll",
0x11b6, 0, FALSE),

    StackFrame(L"C:\\Windows\\Microsoft.NET\\Framework64\\v4.0.30319\\CorperfmonExt.dll",
0xc144, 0, FALSE),
        StackFrame(L"C:\\Windows\\SYSTEM32\\kernel32.dll", 0x17034, 0, FALSE),
        StackFrame(L"C:\\Windows\\SYSTEM32\\ntdll.dll", 0x52651, 0, FALSE),
};

```

By implementing this type of technique, it will make it extremely difficult to implement the callstack integrity checking we showed earlier (granted our demo was hard-coded values, but the point still stands).

## **Conclusion**

---

This was a fairly long post in which we tried to provide some clarity into the mechanisms EDRs can use to not only identify malicious activity, but prevent it. Along the way we've discussed common pitfalls and some enhancements that can be made to protect against the bypasses.

Whilst doing this, we've tried to shed more light onto the 'X bypasses EDR' narrative in which, yes, the beacon might have comeback but there is likely logs of the activity.

The next episode will look at ETW and AMSI!