Introduction to EDRs

Introduction

Endpoint Detection and Response (EDR) is a security solution that detects and responds to threats like ransomware and malware. It works by continuously monitoring endpoints for suspicious activity by collecting data on events such as system logs, network traffic, interprocess communications (IPCs), RPC calls, authentication attempts, and user activity.

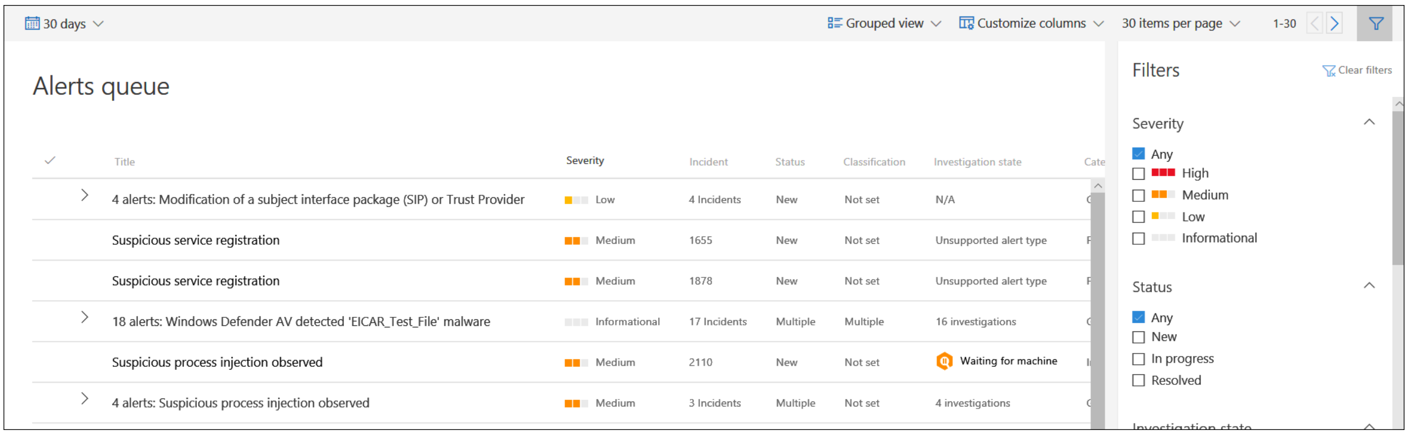
EDRs will collect data when installed on endpoints and then analyze and correlate them to identify potential threats. When a threat is detected, EDR solutions can automatically respond by containing and isolating the affected endpoint from the network or by taking other predefined actions such as deleting malicious files or terminating suspicious processes.

Additionally, EDRs will run programs in sandboxes when executed and then continue to monitor them while they are running in search of malicious behavior.

EDRs should be used as a part of a larger cyber security strategy and used alongside other solutions such as firewalls, intrusion detection systems (IDS), intrusion prevention systems (IPS), and security information and event management (SIEM) solutions. Blue teamers also use EDR logs to perform threat hunting and search for IoCs that could have potentially been missed by the solution.

How EDRs Work

An EDR agent typically consists of two parts: a user-mode application and a kernel-mode driver. These parts gather information using the variety of methods mentioned earlier. The collected data is then analyzed and matched against signatures and malicious behavior. Upon detecting malicious or suspicious behavior, the EDR will log the finding in the security dashboard. EDR settings are highly customizable and depending on its settings, it may either take an action on its own or simply provide an alert. Below is an image from one of Microsoft's articles showing the security dashboard for Microsoft Defender For Endpoint with a few alerts.



Title	Severity	Incident	Status	Classification	Investigation state
4 alerts: Modification of a subject interface package (SIP) or Trust Provider	Low	4 Incidents	New	Not set	N/A
Suspicious service registration	Medium	1655	New	Not set	Unsupported alert type
Suspicious service registration	Medium	1878	New	Not set	Unsupported alert type
18 alerts: Windows Defender AV detected 'EICAR_Test_File' malware	Informational	17 Incidents	Multiple	Multiple	16 investigations
Suspicious process injection observed	Medium	2110	New	Not set	Waiting for machine
4 alerts: Suspicious process injection observed	Medium	3 Incidents	Multiple	Not set	4 investigations

Signature Detection

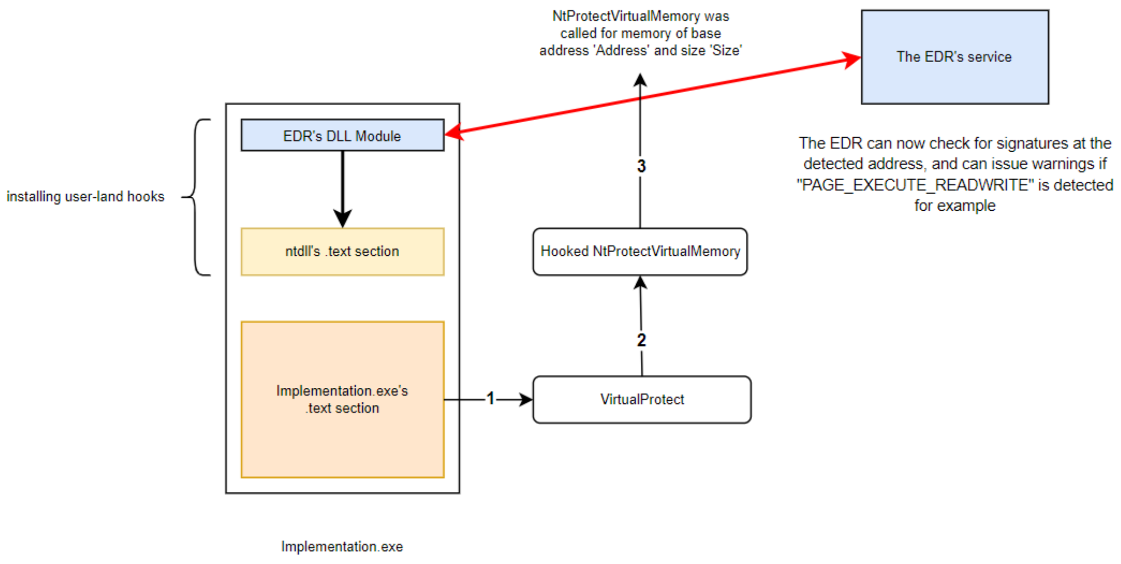
Recall that antiviruses are generally limited to basic signature detection and can be easily bypassed. Although an EDR is far more complex and contains more functionality, it does incorporate AV features to detect known malware. Furthermore, defenders can expand their EDR detection capabilities by creating custom rules.

Detection Based on Behavior

Behavior and runtime detection are one of the main features of an EDR. It can monitor running processes using several methods which are mentioned below.

Userland Hooking

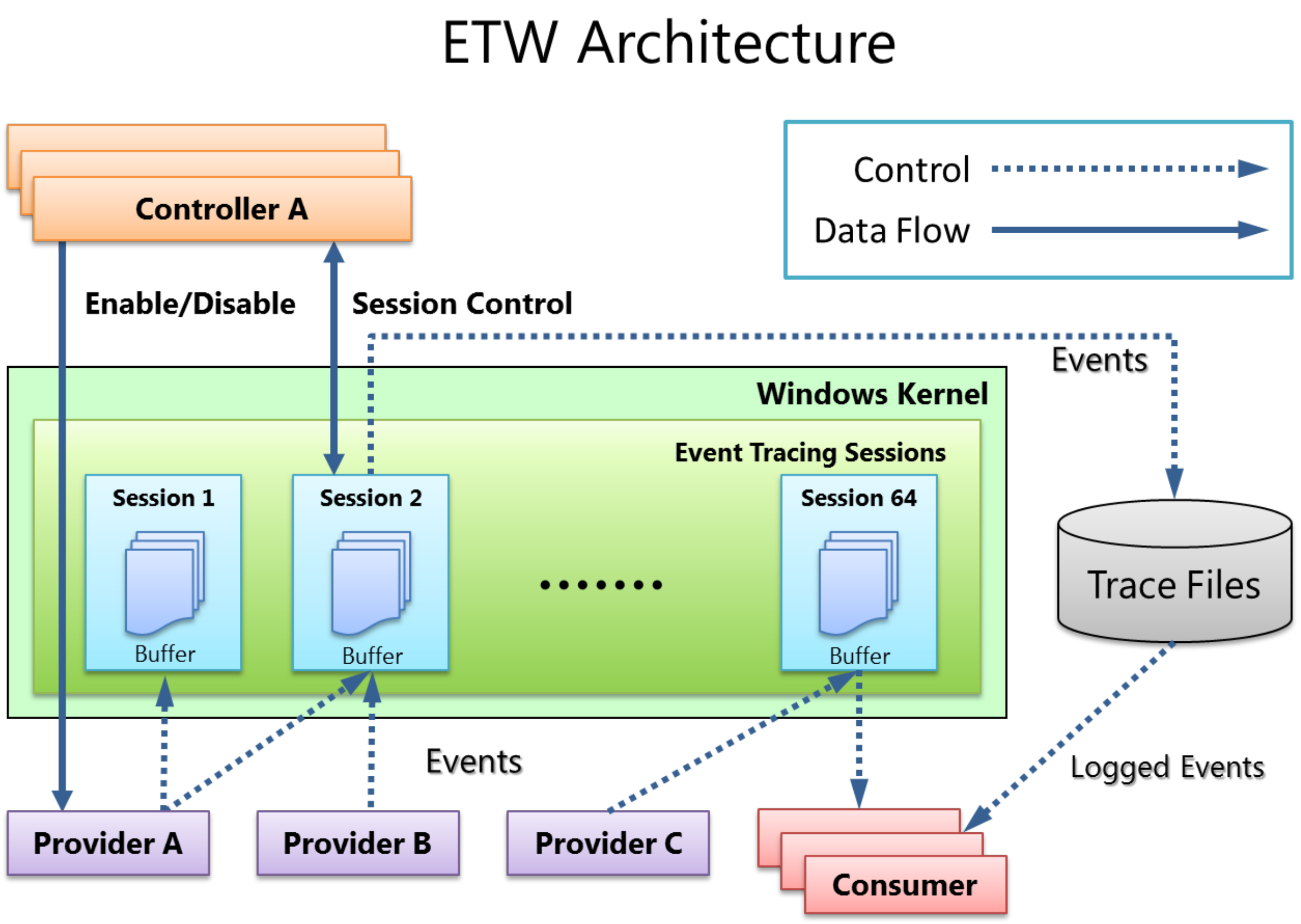
EDRs utilize userland hooking to detect malicious arguments passed to functions as well as see payloads after their decryption. Userland hooking was previously explained in the *Syscalls - Userland Hooking* module. The image below further illustrates userland hooking in action.


NtProtectVirtualMemory was called for memory of base address 'Address' and size 'Size'
The EDR's service
EDR's DLL Module
3
The EDR can now check for signatures at the detected address, and can issue warnings if "PAGE_EXECUTE_READWRITE" is detected for example
installing user-land hooks
ntdll's .text section
Hooked NtProtectVirtualMemory
2
Implementation.exe's .text section
1
VirtualProtect
Implementation.exe

Event Tracing for Windows (ETW)

ETW or Event Tracing for Windows is a kernel mode mechanism built into the Windows operating system that tracks and records events that are triggered by drivers and user-mode applications on the current system.

The following image is from Microsoft's Instrumenting Your Code with ETW article, which shows the ETW architecture.



ETW can log events like process creation and termination, device driver loading and unloading, file and registry access, and user input events. It can also capture network events by logging established connections and authentication requests.

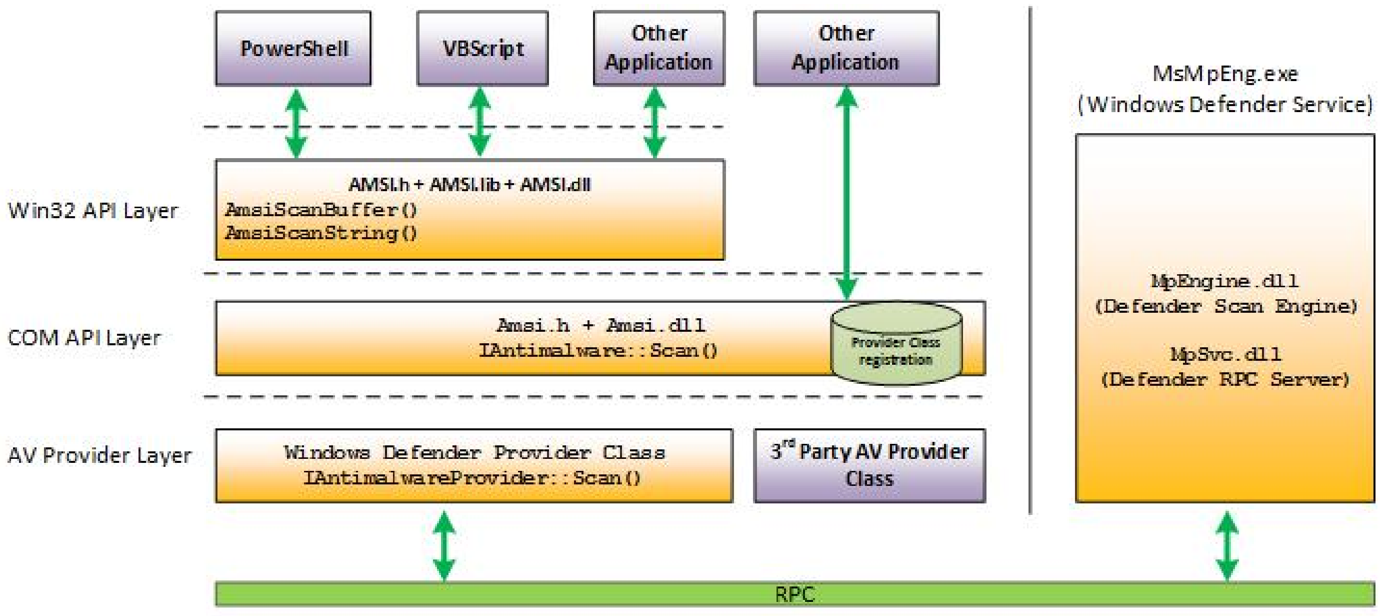
EDRs can utilize this built-in mechanism to further enhance their ability in collecting information about a specific endpoint. On the other hand, several tools also utilize ETW such as Sysmon and Procmon.

Bypassing ETW will be discussed in future modules.

Antimalware Scan Interface (AMSI)

AMSI or Antimalware Scan Interface is another security mechanism built into the Windows OS starting from Windows 10. It allows third-party software to integrate with it and scan and detect malicious applications.

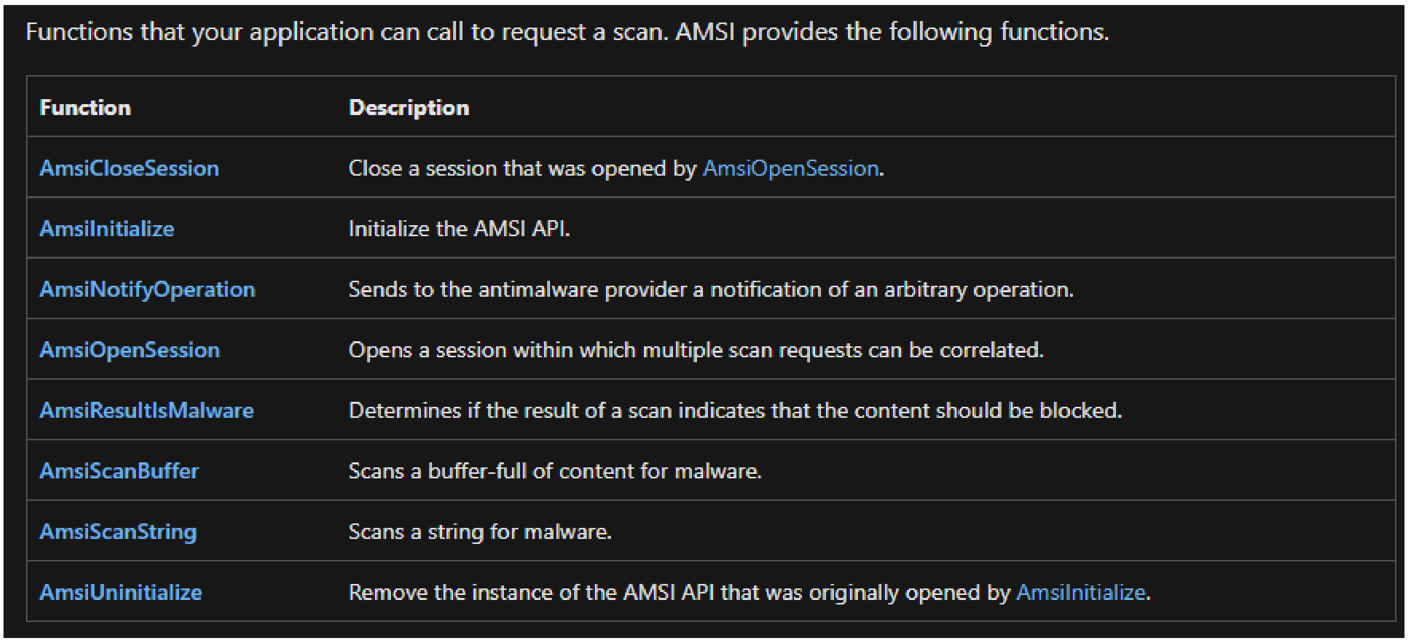
The following image is from Microsoft's How the Antimalware Scan Interface (AMSI) helps you defend against malware article in which AMSI's architecture is visualized.


PowerShell
VBScript
Other Application
Other Application
MsMpEng.exe (Windows Defender Service)
Win32 API Layer
AMSI.h + AMSI.lib + AMSI.dll
AmsiScanBuffer()
AmsiScanString()
COM API Layer
Amsi.h + Amsi.dll
IAntimalware::Scan()
Provider Class registration
AV Provider Layer
Windows Defender Provider Class
IAntimalwareProvider::Scan()
3rd Party AV Provider Class
MpEngine.dll (Defender Scan Engine)
MpSvc.dll (Defender RPC Server)
RPC

Through the use of AMSI, security software is capable of examining scripts, code, and .NET assemblies being executed and injected dynamically, such as those written in JavaScript, VBScript, PowerShell, or other scripting languages. Additionally, AMSI can scan .NET assemblies, which are programs built with Microsoft's .NET framework and programmed in C# and VB.NET.

AMSI is utilized through a group of APIs that are categorized by Microsoft as follows:

- Antimalware Scan Interface Enumerations - Enumerations used by AMSI programming elements.
- Antimalware Scan Interface Functions - Functions that an application can call to request a scan. The image below shows the available AMSI scanning functions.


Functions that your application can call to request a scan. AMSI provides the following functions.

Function	Description
AmsiCloseSession	Close a session that was opened by AmsiOpenSession.
AmsiInitialize	Initialize the AMSI API.
AmsiNotifyOperation	Sends to the antimalware provider a notification of an arbitrary operation.
AmsiOpenSession	Opens a session within which multiple scan requests can be correlated.
AmsiResultIsMalware	Determines if the result of a scan indicates that the content should be blocked.
AmsiScanBuffer	Scans a buffer-full of content for malware.
AmsiScanString	Scans a string for malware.
AmsiUninitialize	Remove the instance of the AMSI API that was originally opened by AmsiInitialize.

- Antimalware Scan Interface Interfaces - COM interfaces that make up the AMSI API.

The core implementation of the AMSI API is provided by `amsi.dll` which is the main DLL that AMSI uses to carry out its operations (reference the above-mentioned functions). The operating system's security subsystem and third-party security products that integrate with AMSI are two other sets of DLLs that are used by AMSI.

Memory-Based Detection

Memory-based detections refer to the IoCs and signatures that are generated after executing your payload and are often created by it. These IoCs can be heap allocations, trampolines when hooking APIs, thread stacks, and RWX memory sections.

Bypassing such measures take place post-execution as the payload is running where adjustments can be made to the payload's layout in memory. Memory-based detection is an advanced concept and is one of the most effective ways to detect malicious code execution.

Bypassing memory-based detections will be covered in future modules.

Kernel CallBacks and Minifilter Drivers

Kernel callbacks are a mechanism used in the Windows OS to allow kernel-mode code to register functions to be called by the OS at specific times or when an event occurs. Some example events are file creation, registry key modification, and a DLL being loaded.

When the event takes place, the OS will call the registered callback function and notify the kernel-mode code that it occurred. This "kernel-mode code" can be a device driver that is created by security products, which in this case is an EDR.

It is worth noting that poorly written or misconfigured callbacks can cause system instability, performance issues, or even security vulnerabilities therefore this isn't a method used by all EDR vendors.

Some example callbacks are listed below.

- PspCreateProcessNotifyRoutine - Registers a driver-supplied callback to be called whenever a process is created or deleted.
- PspLoadImageNotifyRoutine - Registers a driver-supplied callback to be called whenever an image (DLL or EXE) is loaded (or mapped) into memory.
- CmRegisterCallbackEx - Registers a driver-supplied callback to be called whenever a thread operates on the registry.

To intercept, examine, and potentially block I/O events, Microsoft advises security vendors to use minifilter drivers. Minifilter drivers are used in the Windows OS to intercept and modify I/O requests between applications and the file system. These drivers operate at a layer between the file system and the device driver that handles the physical I/O requests. EDRs can utilize minifilter drivers to register a callback for each I/O operation which will notify the driver of specific actions, such as process creation, registry modification, etc.

Additionally, kernel callbacks can be registered by the EDR's Minifilter component in order to get unmodified data directly from the kernel, instead of having data coming from user-land resources, since these can be tampered with and modified.

An example of how EDRs could use minifilter drivers and kernel callbacks is by calling `PspCreateProcessNotifyRoutine` to trigger the EDR to load its user-mode DLL into the created processes, in which it can perform system call hooking, and then using the minifilter driver functionality to monitor I/O file system requests by this newly created process.

Network IoCs

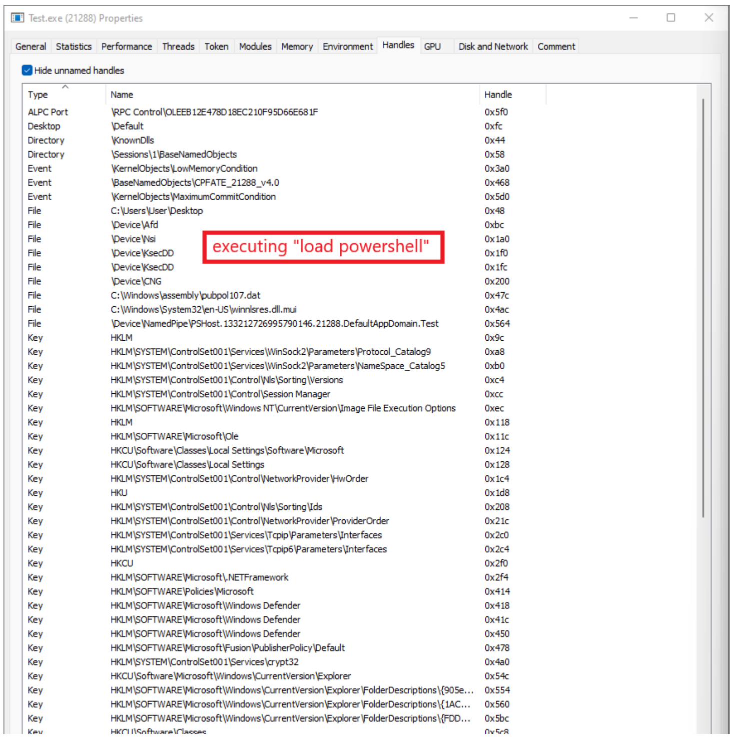
Processes that establish network connections possess a higher degree of suspicion due to the possibility of the connection being to an attacker-controlled C&C server. Network connections will be monitored by EDRs and an alert will be triggered when a process that would not normally use a network connection begins doing so. For example, if process injection was done on `notepad.exe` and it began reaching out to the internet this is considered highly suspicious. Furthermore, aspects of the network connection are analyzed such as the target IP address, domain name, port number and network traffic.

Bypassing EDRs

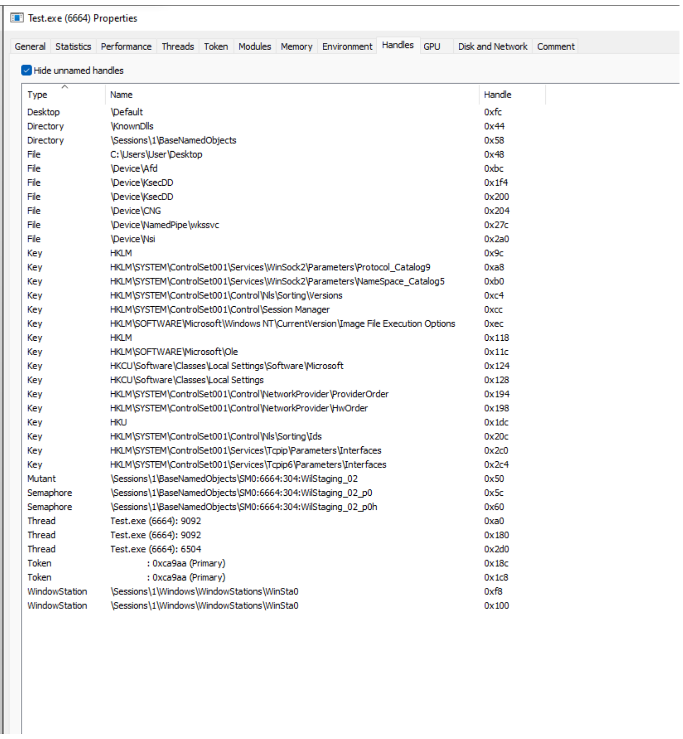
Bypassing EDRs can be difficult to pull off at first and requires a group of methods and techniques instead of relying on a single approach. The reason multiple methods are required is that EDRs use more than one technique to monitor the process. For example, unhooking doesn't block ETWs events but will solve the userland hooking problem. Sometimes multiple implementations will be required to solve the same problem (this will be demonstrated in the NTDLL unhooking modules).

It is important to bear in mind that some EDR bypass techniques allow the loader to evade detection but not the C&C payload in use. This can be the case due to several reasons:

- The C&C network anomalies are well-known and signatured by the EDR.
- The loader uses direct/indirect syscalls and successfully evaded detection, but the C&C payload doesn't and still uses hooked functions.
- The C&C payload executed a noisy command, either intentionally or unintentionally. Such commands will catch the attention of an EDR, and thus your implementation will be detected (e.g. spawn cmd.exe and execute the `whoami` command).
- The C&C uses recognizable named IPCs handles or open specific ones (recall that IPCs are Pipes - Events - Metaphors - Semaphores). For example, executing the "load powershell" command using Meterpreter results in the following.


Test.exe (21288) Properties
General | Statistics | Performance | Threads | Token | Modules | Memory | Environment | Handles | GPU | Disk and Network | Comment
Hide unnamed handles
executing "load powershell"

Type	Name	Handle
ALPC Port	\RPC Control\OLEEB12E478D18EC210F95D66E681F	0x5f0
Desktop	\Default	0xfc
Directory	\KnownDlls	0x44
Directory	\Sessions\1\BaseNamedObjects	0x58
Event	\KernelObjects\LowMemoryCondition	0x3a0
Event	\BaseNamedObjects\CPFATE_21288_v4.0	0x468
Event	\KernelObjects\MaximumCommitCondition	0x5d0
File	C:\Users\User\Desktop	0x48
File	\Device\Afd	0xbc
File	\Device\Nsi	0x1a0
File	\Device\KsecDD	0x1f0
File	\Device\KsecDD	0x1fc
File	\Device\CNG	0x200
File	C:\Windows\assembly\pubpol107.dat	0x47c
File	C:\Windows\System32\en-US\winnlsres.dll.mui	0x4ac
File	\Device\NamedPipe\PSHost.133212726995790146.21288.DefaultAppDomain.Test	0x564
Key	HKLM	0x9c
Key	HKLM\SYSTEM\ControlSet001\Services\WinSock2\Parameters\Protocol_Catalog9	0xa8
Key	HKLM\SYSTEM\ControlSet001\Services\WinSock2\Parameters\NameSpace_Catalog5	0xb0
Key	HKLM\SYSTEM\ControlSet001\Control\Nls\Sorting\Versions	0xc4
Key	HKLM\SYSTEM\ControlSet001\Control\Session Manager	0xcc
Key	HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options	0xec
Key	HKLM	0x118
Key	HKLM\SOFTWARE\Microsoft\Ole	0x11c
Key	HKCU\Software\Classes\Local Settings\Software\Microsoft	0x124
Key	HKCU\Software\Classes\Local Settings	0x128
Key	HKLM\SYSTEM\ControlSet001\Control\NetworkProvider\HwOrder	0x1c4
Key	HKU	0x1d8
Key	HKLM\SYSTEM\ControlSet001\Control\Nls\Sorting\Ids	0x208
Key	HKLM\SYSTEM\ControlSet001\Control\NetworkProvider\ProviderOrder	0x21c
Key	HKLM\SYSTEM\ControlSet001\Services\Tcpip\Parameters\Interfaces	0x2c0
Key	HKLM\SYSTEM\ControlSet001\Services\Tcpip6\Parameters\Interfaces	0x2c4
Key	HKCU	0x2f0
Key	HKLM\SOFTWARE\Microsoft\.NETFramework	0x2f4
Key	HKLM\SOFTWARE\Policies\Microsoft	0x414
Key	HKLM\SOFTWARE\Microsoft\Windows Defender	0x418
Key	HKLM\SOFTWARE\Microsoft\Windows Defender	0x41c
Key	HKLM\SOFTWARE\Microsoft\Windows Defender	0x450
Key	HKLM\SOFTWARE\Microsoft\Fusion\PublisherPolicy\Default	0x478
Key	HKLM\SYSTEM\ControlSet001\Services\crypt32	0x4a0
Key	HKCU\Software\Microsoft\Windows\CurrentVersion\Explorer	0x54c
Key	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\FolderDescriptions\{905e…	0x554
Key	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\FolderDescriptions\{1AC…	0x560
Key	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\FolderDescriptions\{FDD…	0x5bc
Key	HKCU\Software\Classes	0x5c8


Test.exe (6664) Properties
General | Statistics | Performance | Threads | Token | Modules | Memory | Environment | Handles | GPU | Disk and Network | Comment
Hide unnamed handles

Type	Name	Handle
Desktop	\Default	0xfc
Directory	\KnownDlls	0x44
Directory	\Sessions\1\BaseNamedObjects	0x58
File	C:\Users\User\Desktop	0x48
File	\Device\Afd	0xbc
File	\Device\KsecDD	0x1f4
File	\Device\KsecDD	0x200
File	\Device\CNG	0x204
File	\Device\NamedPipe\wkssvc	0x27c
File	\Device\Nsi	0x2a0
Key	HKLM	0x9c
Key	HKLM\SYSTEM\ControlSet001\Services\WinSock2\Parameters\Protocol_Catalog9	0xa8
Key	HKLM\SYSTEM\ControlSet001\Services\WinSock2\Parameters\NameSpace_Catalog5	0xb0
Key	HKLM\SYSTEM\ControlSet001\Control\Nls\Sorting\Versions	0xc4
Key	HKLM\SYSTEM\ControlSet001\Control\Session Manager	0xcc
Key	HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options	0xec
Key	HKLM	0x118
Key	HKLM\SOFTWARE\Microsoft\Ole	0x11c
Key	HKCU\Software\Classes\Local Settings\Software\Microsoft	0x124
Key	HKCU\Software\Classes\Local Settings	0x128
Key	HKLM\SYSTEM\ControlSet001\Control\NetworkProvider\ProviderOrder	0x194
Key	HKLM\SYSTEM\ControlSet001\Control\NetworkProvider\HwOrder	0x198
Key	HKU	0x1dc
Key	HKLM\SYSTEM\ControlSet001\Control\Nls\Sorting\Ids	0x20c
Key	HKLM\SYSTEM\ControlSet001\Services\Tcpip\Parameters\Interfaces	0x2c0
Key	HKLM\SYSTEM\ControlSet001\Services\Tcpip6\Parameters\Interfaces	0x2c4
Mutant	\Sessions\1\BaseNamedObjects\SM0:6664:304:WilStaging_02	0x50
Semaphore	\Sessions\1\BaseNamedObjects\SM0:6664:304:WilStaging_02_p0	0x5c
Semaphore	\Sessions\1\BaseNamedObjects\SM0:6664:304:WilStaging_02_p0h	0x60
Thread	Test.exe (6664): 9092	0xa0
Thread	Test.exe (6664): 9092	0x180
Thread	Test.exe (6664): 6504	0x2d0
Token	: 0xca9aa (Primary)	0x18c
Token	: 0xca9aa (Primary)	0x1c8
WindowStation	\Sessions\1\Windows\WindowStations\WinSta0	0xf8
WindowStation	\Sessions\1\Windows\WindowStations\WinSta0	0x100

For such reasons, and more, there would be a lot of cases where your implementation would succeed in returning a connection to your C2 server, but would get detected when running some specific commands. So choosing your C2 is an important decision for runtime evasion. It is always advised to use a highly flexible and malleable C2 framework rather than a limited one.

In the following modules, multiple strategies will be presented to address EDR detection mechanisms. One may select the method that best fits their needs and combine it with other previously shown techniques to create successful implementations that can bypass EDR solutions.