Gatekeeping SysCalls

 mez0.cc/posts/gatekeeping-syscalls

Introduction

Over the years, SysCalls has became significantly more important. And over those years, multiple techniques have spawned with the naming scheme of:

```
something biblical + Gate
```

As of writing this, there are four techniques that I am aware of. So, my this post is a glorified note containing an *explanation* of each. No original code, just notes on the techniques.

Heavens Gate

The long and short of Heavens Gate is that it allows WOW64 Processes to execute 64-bit SysCalls. Originally, I was going to summarise this component in a similar fashion to the others in this post. However, Closing "Heaven's Gate" by Alex Ionescu is too good to not read. The following is a quote from that blog:

> Heaven's Gate, then, refers to subverting the fact that a 64-bit NTDLL exists (and a 64-bit heap, PEB and TEB), and manually jumping into the long-mode code segment without having to issue a system call and being subjected to the code flow that WoW64 will attempt to enforce. In other words, it gives one the ability to create "naked" 64-bit code, which will be able to run covertly, including issuing system calls, without the majority of products able to intercept and/or introspect its execution

Hells Gate

Hells Gate was developed by smelly__vx and am0nsec, along with code and a whitepaper. Hells Gate is an adaption of Heavens Gate, which was originally designed to execute 32-bit code from a 64-bit process.

The main problem they identified with tools such as SysWhispers(1) was that it relied heavily on hard-coded SysCalls which is based on Windows X86-64 System Call Table (XP/2003/Vista/2008/7/2012/8/10), as seen below:

```
.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_Check_6_X_XXXX:             ; Check minor version for
Windows Vista/7/8.
    cmp dword ptr [rax+11ch], 0
    je  NtAllocateVirtualMemory_Check_6_0_XXXX
    cmp dword ptr [rax+11ch], 1
    je  NtAllocateVirtualMemory_Check_6_1_XXXX
    cmp dword ptr [rax+11ch], 2
    je  NtAllocateVirtualMemory_SystemCall_6_2_XXXX
    cmp dword ptr [rax+11ch], 3
    je  NtAllocateVirtualMemory_SystemCall_6_3_XXXX
    jmp NtAllocateVirtualMemory_SystemCall_Unknown
NtAllocateVirtualMemory_Check_6_0_XXXX:             ; Check build number for
Windows Vista.
    cmp word ptr [rax+120h], 6000
    je  NtAllocateVirtualMemory_SystemCall_6_0_6000
    cmp word ptr [rax+120h], 6001
    je  NtAllocateVirtualMemory_SystemCall_6_0_6001
    cmp word ptr [rax+120h], 6002
    je  NtAllocateVirtualMemory_SystemCall_6_0_6002
    jmp NtAllocateVirtualMemory_SystemCall_Unknown
NtAllocateVirtualMemory_Check_6_1_XXXX:             ; Check build number for
Windows 7.
    cmp word ptr [rax+120h], 7600
    je  NtAllocateVirtualMemory_SystemCall_6_1_7600
    cmp word ptr [rax+120h], 7601
    je  NtAllocateVirtualMemory_SystemCall_6_1_7601
    jmp NtAllocateVirtualMemory_SystemCall_Unknown
NtAllocateVirtualMemory_Check_10_0_XXXX:            ; Check build number for
Windows 10.
    cmp word ptr [rax+120h], 10240
    je  NtAllocateVirtualMemory_SystemCall_10_0_10240
    cmp word ptr [rax+120h], 10586
    je  NtAllocateVirtualMemory_SystemCall_10_0_10586
    cmp word ptr [rax+120h], 14393
    je  NtAllocateVirtualMemory_SystemCall_10_0_14393
    cmp word ptr [rax+120h], 15063
    je  NtAllocateVirtualMemory_SystemCall_10_0_15063
    cmp word ptr [rax+120h], 16299
    je  NtAllocateVirtualMemory_SystemCall_10_0_16299
    cmp word ptr [rax+120h], 17134
```

```
    je  NtAllocateVirtualMemory_SystemCall_10_0_17134
    cmp word ptr [rax+120h], 17763
    je  NtAllocateVirtualMemory_SystemCall_10_0_17763
    cmp word ptr [rax+120h], 18362
    je  NtAllocateVirtualMemory_SystemCall_10_0_18362
    cmp word ptr [rax+120h], 18363
    je  NtAllocateVirtualMemory_SystemCall_10_0_18363
    cmp word ptr [rax+120h], 19041
    je  NtAllocateVirtualMemory_SystemCall_10_0_19041
    cmp word ptr [rax+120h], 19042
    je  NtAllocateVirtualMemory_SystemCall_10_0_19042
    cmp word ptr [rax+120h], 19043
    je  NtAllocateVirtualMemory_SystemCall_10_0_19043
    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
NtAllocateVirtualMemory_SystemCall_10_0_19042:        ; Windows 10.0.19042 (20H2)
    mov eax, 0018h
    jmp NtAllocateVirtualMemory_Epilogue
NtAllocateVirtualMemory_SystemCall_10_0_19043:        ; Windows 10.0.19043 (21H1)
    mov eax, 0018h
    jmp NtAllocateVirtualMemory_Epilogue
NtAllocateVirtualMemory_SystemCall_Unknown:           ; Unknown/unsupported version.
    ret
NtAllocateVirtualMemory_Epilogue:
    mov r10, rcx
    syscall
    ret
NtAllocateVirtualMemory ENDP
end
```

The sheer size of this shows how inefficient this is. Sometime later, SysWhispers2 was released which reduced the code down to:

```
.code

EXTERN SW2_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, 04FDF5971h        ; Load function hash into ECX.
    call SW2_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
```

Due to the way this works, its starting to see signatures; more on this later.

So, how is Hells Gate actually used? Instead of having the SysCalls for each function in a `.asm` file, and then `EXTERN` functions, it uses a `struct` called `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;
```

Each NTAPI Call is a struct called `VX_TABLE_ENTRY` within the table, which is defined as such:

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

The project also requires two functions, and one variable from the hellsgate.asm:

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

.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
```

The whitepaper explains why/how this works:

> 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.

Using their example, this can be seen below:

```
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>
```

They then explain:

> as shown, functions move into the R10 register from the RCX register and then move the system call into EAX.

This matches up to the assembly shipped with the project:

```
HellsGate PROC
    mov wSystemCall, 000h
    mov wSystemCall, ecx
    ret
HellsGate ENDP

HellDescent PROC
    mov r10, rcx
    mov eax, wSystemCall

    syscall
    ret
HellDescent ENDP
```

This can all be utilised, as seen in line 166 of main.c:

```
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
    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);

    return TRUE;
}
```

For each NTAPI Call, the following events must occur:

```
HellsGate(pVxTable->NtAllocateVirtualMemory.wSystemCall);
status = HellDescent((HANDLE)-1, &lpAddress, 0, &sDataSize, MEM_COMMIT,
PAGE_READWRITE);
```

1. The SysCall is obtained in GetVXTableEntry, this populates the aforementioned `struct`
2. `HellsGate()` is then called which moves the SysCall into `ecx`:

```
HellsGate PROC
    mov wSystemCall, 000h
    mov wSystemCall, ecx
    ret
HellsGate ENDP
```

3. `HellsDescent()` is then called which moves the `r10` register to the `rcx`, and then move the system call into `eax`

```
HellDescent PROC
    mov r10, rcx
    mov eax, wSystemCall

    syscall
    ret
HellDescent ENDP
```

4. The NTAPI is executed.

Halos Gate

Hells Gate is a great project. However, Sektor7 identified a problem with it in Halo's Gate - twin sister of Hell's Gate. The problem:

> One limitation of **Hell's Gate** is that it needs access to a clean *ntdll* module. Otherwise, it cannot populate needed syscall numbers and eventually fails to deliver Native API calls.

If a hook is placed on a function, and Hells Gate is used, then the hooked call will be used. An explanation from Sektor 7:

> A hook on **ZwMapViewOfSection** is clearly visible (`jmp <offset>` instruction, instead of `mov r10, rcx`). But "neighbors" of *ZwMapViewOfSection*, **ZwSetInformationFile** and **NtAccessCheckAndAuditAlarm** are clean and their syscall numbers are `0x27` and `0x29`, respectively.
> ...
>
> It's like a ripple on a lake - you start from the center and move outwards up until you find a clean syscall.

A few people have implemented this:

The closest sample relating to the code shown from Sektor7 can be found in trickster0/TartarusGate/blob/master/HellsGate/main.c#L151:

```
if (djb2(pczFunctionName) == pVxTableEntry->dwHash) {
pVxTableEntry->pAddress = pFunctionAddress;

// First opcodes should be :
//    MOV R10, RCX
//    MOV RAX, <syscall>
if (*((PBYTE)pFunctionAddress) == 0x4c
    && *((PBYTE)pFunctionAddress + 1) == 0x8b
    && *((PBYTE)pFunctionAddress + 2) == 0xd1
    && *((PBYTE)pFunctionAddress + 3) == 0xb8
    && *((PBYTE)pFunctionAddress + 6) == 0x00
    && *((PBYTE)pFunctionAddress + 7) == 0x00) {

    BYTE high = *((PBYTE)pFunctionAddress + 5);
    BYTE low = *((PBYTE)pFunctionAddress + 4);
    pVxTableEntry->wSystemCall = (high << 8) | low;

    return TRUE;
}
 //if hooked check the neighborhood to find clean syscall
if (*((PBYTE)pFunctionAddress) == 0xe9) {
    for (WORD idx = 1; idx <= 500; idx++) {
        // check neighboring syscall down
        if (*((PBYTE)pFunctionAddress + idx * DOWN) == 0x4c
            && *((PBYTE)pFunctionAddress + 1 + idx * DOWN) == 0x8b
            && *((PBYTE)pFunctionAddress + 2 + idx * DOWN) == 0xd1
            && *((PBYTE)pFunctionAddress + 3 + idx * DOWN) == 0xb8
            && *((PBYTE)pFunctionAddress + 6 + idx * DOWN) == 0x00
            && *((PBYTE)pFunctionAddress + 7 + idx * DOWN) == 0x00) {
            BYTE high = *((PBYTE)pFunctionAddress + 5 + idx * DOWN);
            BYTE low = *((PBYTE)pFunctionAddress + 4 + idx * DOWN);
            pVxTableEntry->wSystemCall = (high << 8) | low - idx;

            return TRUE;
        }
        // check neighboring syscall up
        if (*((PBYTE)pFunctionAddress + idx * UP) == 0x4c
            && *((PBYTE)pFunctionAddress + 1 + idx * UP) == 0x8b
            && *((PBYTE)pFunctionAddress + 2 + idx * UP) == 0xd1
            && *((PBYTE)pFunctionAddress + 3 + idx * UP) == 0xb8
            && *((PBYTE)pFunctionAddress + 6 + idx * UP) == 0x00
            && *((PBYTE)pFunctionAddress + 7 + idx * UP) == 0x00) {
            BYTE high = *((PBYTE)pFunctionAddress + 5 + idx * UP);
            BYTE low = *((PBYTE)pFunctionAddress + 4 + idx * UP);
            pVxTableEntry->wSystemCall = (high << 8) | low + idx;

            return TRUE;
        }

    }
    return FALSE;
}
```

So, the difference between Hells and Halos Gate is simply ensuring that the SysCalls are not hooked.

Tartarus Gate

The latest implementation is Tartarus Gate by trickster0. The difference here:

> Hell's Gate evolved to Halo's Gate to bypass EDRs by unhooking some of them and now it turned to Tartarus' Gate to handle even more WINAPI hooking methods.
>
> I have added some more ASM commands just for "obfuscation" for the syscalls.

Hells Gate uses the following assembly functions:

```
.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 has now became:

```
.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
```

As seen above, `HellDescent()` has some additional `nop` and light *obfuscation*. Furthermore, as shown in Halos Gate, the implementation of Halos Gate has been added: trickster0/TartarusGate/blob/master/HellsGate/main.c#L151.

For a reference, Cracked5pider/KayLdr has implemented this in the form of a Reflective DLL Loader. The core of the SysCall identification is done in https://github.com/Cracked5pider/KaynLdr/blob/main/KaynLdr/src/Syscall.c#L9.

Conclusion

As far as I can tell, these are the only `gates` I am aware of: Heaven, Hell, Halo, and Tartarus. And in that order, they progressively become more developed. Leaving off with Tartarus, currently, being the most functional.

For more detail, I'd just suggest going to the referenced blogs/projects to get a better understanding of the technique in question, this is just a note to self!