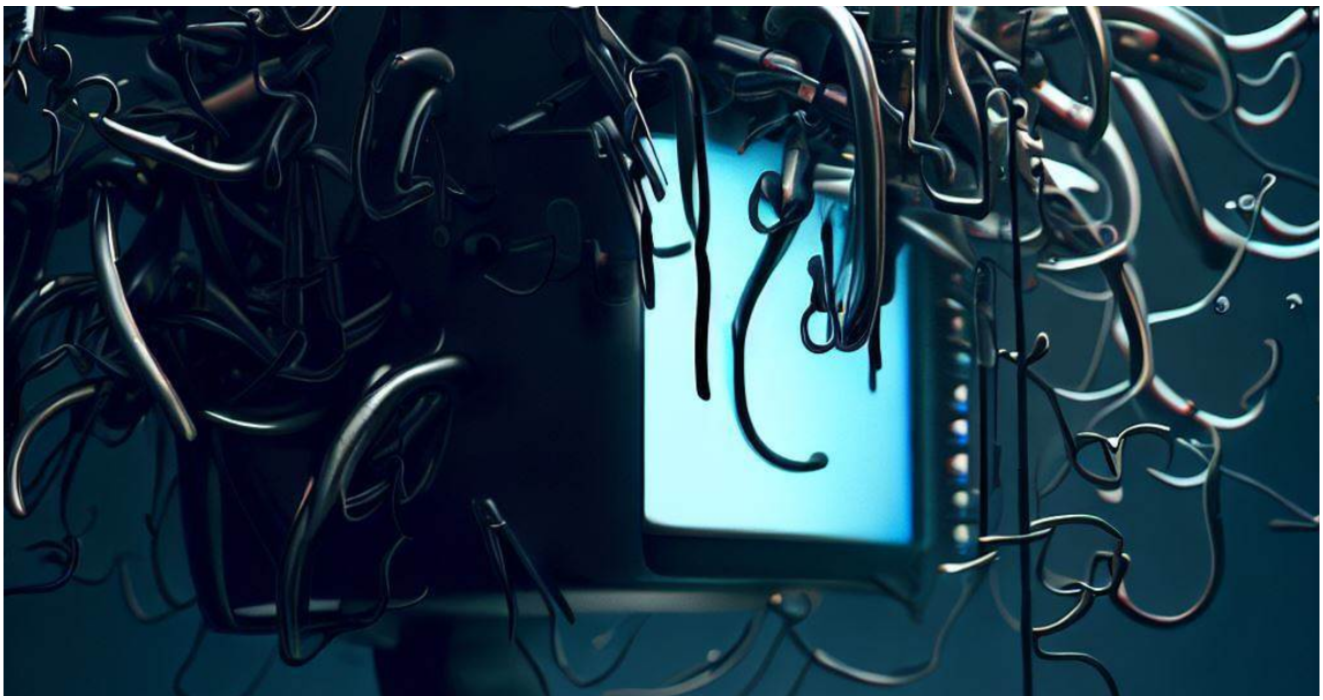
Understanding Syscalls: Direct, Indirect, and Cobalt Strike Implementation

Mohamed Adel August 18, 2023

Contents

Mohamed Adel included in Reverse Engineering Research
2023-08-18 3243 words 16 minutes



> In case images fail to load, it might be due to jsDelivr CDN ban in Egypt. To resolve this, consider using a VPN. :)

Syscalls? Why?

To Bypass user-mood hooks. why?
- For Hiding a code inside a legitimate process (Process Injection)
- Avoiding EDR alerts!

User-mood Hooks

Hooking user-mode functions by placing a jump to another code section. EDRs use hooks to check the function parameters. For example, if you are trying to change the memory protections of some data to add executable protections. This is a very suspicious activity so

EDRs will be alert to that. Most Hooks are on the lowest level of the user-mode interface in **ntdll.dll** which are the system calls.

Direct syscalls

Windows has a defined schema of how `syscalls` are used. Most of the documented windows APIs are just a wrapper of a lower-level Functions in `ntdll.dll` which are compiled to a `syscall` with the right SSN (System Service Number). To look at how `Nt*` version of the higher-level API is implemented.

```
0:018> uf NtOpenProcess
ntdll!NtOpenProcess:
00007ffa`4874d4c0 4c8bd1          mov     r10,rcx
00007ffa`4874d4c3 b826000000      mov     eax,26h
00007ffa`4874d4c8 f604250803fe7f01 test    byte ptr [SharedUserData+0x308
(00000000`7ffe0308)],1
00007ffa`4874d4d0 7503            jne     ntdll!NtOpenProcess+0x15
(00007ffa`4874d4d5)  Branch

ntdll!NtOpenProcess+0x12:
00007ffa`4874d4d2 0f05            syscall
00007ffa`4874d4d4 c3              ret

ntdll!NtOpenProcess+0x15:
00007ffa`4874d4d5 cd2e            int     2Eh
00007ffa`4874d4d7 c3              ret
```

At address `00007ffa~4874d4d2` there `syscall` instruction. This instruction transfers the execution to the system-handler at the kernel. The handler is specified using pre-defined SSN number loaded into `EAX` Register (In this case `EAX = 0x26` at address `00007ffa~4874d4c3`). So, to make a `syscall` The SSN associated. The code stub of the `syscalls` is simple.

```
mov r10, rcx
mov eax,
<syscall_number>
syscall
ret
```

Now, the missing thing is the `syscall_number`. These numbers are changing based on the Build version of windows. There are some techniques to get these numbers.

1. **SysWhispers**

SysWhispers That generate the table of these numbers in the form of a header file and assembly file that can be embedded in the code. The generated code contains `syscall` number for multiple versions, The right windows build version is detected at runtime using PEB structure.

```
...
+0x118 OSMajorVersion   :
Uint4B
+0x11c OSMinorVersion   :
Uint4B
+0x120 OSBuildNumber    :
Uint2B
...
```

The assembly code generated (Full document at example-output)

```
...
NtOpenProcess PROC
        mov rax, gs:[60h]                       ; Load PEB into RAX.
NtOpenProcess_Check_X_X_XXXX:               ; Check major version.
        cmp dword ptr [rax+118h], 5
        je  NtOpenProcess_SystemCall_5_X_XXXX
        cmp dword ptr [rax+118h], 6
        je  NtOpenProcess_Check_6_X_XXXX
        cmp dword ptr [rax+118h], 10
```

```
        je  NtOpenProcess_Check_10_0_XXXX
        jmp NtOpenProcess_SystemCall_Unknown
NtOpenProcess_Check_6_X_XXXX:               ; Check minor version for Windows
Vista/7/8.
        cmp dword ptr [rax+11ch], 0
        je  NtOpenProcess_Check_6_0_XXXX
        cmp dword ptr [rax+11ch], 1
        je  NtOpenProcess_Check_6_1_XXXX
        cmp dword ptr [rax+11ch], 2
        je  NtOpenProcess_SystemCall_6_2_XXXX
        cmp dword ptr [rax+11ch], 2
        je  NtOpenProcess_SystemCall_6_3_XXXX
        jmp NtOpenProcess_SystemCall_Unknown
NtOpenProcess_Check_6_0_XXXX:               ; Check build number for Windows
Vista.
        cmp dword ptr [rax+120h], 6000
        je  NtOpenProcess_SystemCall_6_0_6000
        cmp dword ptr [rax+120h], 6001
        je  NtOpenProcess_SystemCall_6_0_6001
        cmp dword ptr [rax+120h], 6002
        je  NtOpenProcess_SystemCall_6_0_6002
        jmp NtOpenProcess_SystemCall_Unknown
NtOpenProcess_Check_6_1_XXXX:               ; Check build number for Windows 7.
        cmp dword ptr [rax+120h], 7600
        je  NtOpenProcess_SystemCall_6_1_7600
        cmp dword ptr [rax+120h], 7601
        je  NtOpenProcess_SystemCall_6_1_7601
        jmp NtOpenProcess_SystemCall_Unknown
NtOpenProcess_Check_10_0_XXXX:              ; Check build number for Windows 10.
        cmp dword ptr [rax+120h], 10240
        je  NtOpenProcess_SystemCall_10_0_10240
        cmp dword ptr [rax+120h], 10586
        je  NtOpenProcess_SystemCall_10_0_10586
        ...
```

1. **SSN code stub** This technique doesn't Look for SSN number, instead it gets the code stub of the required API. This can be done by opening the PE file and parsing the Export table of `ntdll`

2. **Extract SSN** It Extract the SSN from `ntdll` by parsing the Export table. The difference between it and the previous one is that it only extracts the `syscall` number. Both methods load `ntdll.dll` from the disk first using win32 API `OpenFile` which might be hooked. hell's gate for more.
3. **Syscalls' number sequence** This method take advantage of the SSNs are in a sequence for example if a syscall number is 0x26 the following will be 0x27 and so on. This relies also on the fact that not all the system calls are hooked! So, to get the SSN of a function, you need to find the nearest unhooked syscall. this was presented by halos gate. But This is not valid in newer versions of Windows as the SSNs sequence is no longer valid.
4. **Parallel loading** This is an interesting technique explained in this blog. It uses windows feature introduced in windows 10 to load DLLs through multiple threads instead of one in older versions of windows. It was found that the syscall stub of native Functions `NtOpenFile()`, `NtCreateSection()`, `ZwQueryAttributeFile()`, `ZwOpenSection()` and `ZwMapViewOfFile()` -There is a lot of things happens between the two actions, detailed explanation in the previously mentioned blog -are copied into `LdrpThunkSignature` array. This is done to check the integrity of the functions' code. These APIs' syscall numbers can be used to load a new version of ntdll.dll from the disk and avoid any user-mood hooks.
5. **Sorting by system call address** This technique uses the relation between the address of the system call stub and the SSN. It is known as FreshyCalls . In simple words, it walks the Export Address Table of `ntdll` and saves the Name -or a hash of the name- and Address of each entry in a table. Then, it sorts the entries by the addresses in ascending order. It was found that the first function `NtAccessCheck` (by address) has an SSN = 0

```
0:007> uf NtAccessCheck
ntdll!NtAccessCheck:
00007ffa`4874d000 4c8bd1          mov     r10,rcx
00007ffa`4874d003 b800000000      mov     eax,0
00007ffa`4874d008 f604250803fe7f01 test    byte ptr [SharedUserData+0x308
(00000000`7ffe0308)],1
00007ffa`4874d010 7503            jne     ntdll!NtAccessCheck+0x15
(00007ffa`4874d015)  Branch

ntdll!NtAccessCheck+0x12:
00007ffa`4874d012 0f05            syscall
00007ffa`4874d014 c3              ret

ntdll!NtAccessCheck+0x15:
00007ffa`4874d015 cd2e            int     2Eh
00007ffa`4874d017 c3              ret
```

and if we unassembled the next function by adding one to the last address (as ret opcode is one byte) we will get that the next function's SSN is 1!

```
0:007> uf 00007ffa`4874d017+1
ntdll!NtAccessCheck+0x18:
00007ffa`4874d018 0f1f840000000000 nop     dword ptr [rax+rax]
00007ffa`4874d020 4c8bd1          mov     r10,rcx
00007ffa`4874d023 b801000000      mov     eax,1
00007ffa`4874d028 f604250803fe7f01 test    byte ptr [SharedUserData+0x308
(00000000`7ffe0308)],1
00007ffa`4874d030 7503            jne     ntdll!NtWorkerFactoryWorkerReady+0x15
(00007ffa`4874d035)  Branch

ntdll!NtWorkerFactoryWorkerReady+0x12:
00007ffa`4874d032 0f05            syscall
00007ffa`4874d034 c3              ret

ntdll!NtWorkerFactoryWorkerReady+0x15:
00007ffa`4874d035 cd2e            int     2Eh
00007ffa`4874d037 c3              ret
```

So, by sorting the functions by the addresses, we have the SSN. for the code, look at MDSec (8. Sorting by System Call Address) blog or see FreshlyCalls implementation. The execution of the system call is not direct by calling `syscall` instruction. Instead. It uses the method explained below. Briefly, it uses the `syscall` instructions from `ntdll`.

Indirect syscalls

All the methods described are workarounds to get the system call number without getting caught. `syscall` instruction reveals that some suspicious activity is going on. This is done using `KPROCESS!InstrumentationCallback` in windows.

```
0:030> dt _kprocess
ntdll!_KPROCESS
   +0x000 Header           :
_DISPATCHER_HEADER
   ...
   +0x3d8 InstrumentationCallback : Ptr64
Void
   ...
   +0x3f8 EndPadding       : [8] Uint8B
```

Any time the windows is done with a syscall and returns to user-mode, it checks this member it is not `NULL`, the execution will be transferred to that pointer. To check if the syscall is legit, the return address after finishing the syscall is checked to see if it is not from a valid place. If the address is in the address space of the process running, it's not a legitimate place to make a syscall. This check was done by ScyllaHide to detect manual syscalls, the source code can be found here.

```
 if (InterlockedOr(TlsGetInstrumentationCallbackDisabled(), 0x1) == 0x1)
        return ReturnVal; // Do not recurse

    const PVOID ImageBase = NtCurrentPeb()->ImageBaseAddress;
    const PIMAGE_NT_HEADERS NtHeaders = RtlImageNtHeader(ImageBase);
    if (NtHeaders != nullptr && ReturnAddress >= (ULONG_PTR)ImageBase &&
        ReturnAddress < (ULONG_PTR)ImageBase + NtHeaders-
>OptionalHeader.SizeOfImage)
    {
        // Syscall return address within the exe file
        ReturnVal = (ULONG_PTR)(ULONG)STATUS_PORT_NOT_SET;

        // Uninstall ourselves after we have completed the sequence { NtQIP,
NtQIP }. More NtSITs will follow but we can't do anything about them
        NumManualSyscalls++;
        if (NumManualSyscalls >= 2)
        {
            InstallInstrumentationCallbackHook(NtCurrentProcess, TRUE);
        }
    }

    InterlockedAnd(TlsGetInstrumentationCallbackDisabled(), 0);

    return ReturnVal;
}
```

It checks the return address of the successful system call. If it resides on the address space of the binary we are running, it is an indication of manual system call.

The Solution The solution to this hooking method is done by Bouncy Gate and Recycled Gate method. The idea is quite simple, it is an adjusted version of Hell's Gate. Instead of directly executing `syscall` instruction and getting caught by static signatures and system call callbacks described above, the author replaces the `syscall` instruction with a trampoline jump (`JMP`) to a `syscall` instruction address from `ntdll.dll`. now there is no direct `syscall` instruction and the system call originated from a legitimate place `ntdll`. This is also implemented in SysWhispers3. To get the address of the syscall instruction in `ntdll` we can parse the export table and search for syscall, ret opcodes `0F 05 0C` or the constant pattern of syscalls in `ntdll` can be used to get the syscall address. If the function is not hooked, the syscall instruction is on offset `0x12` from the function's address, we can verify that by comparing the opcodes.

Indirect syscalls in Cobalt Strike

The sample from Dodo's blog Where he already analyzed how indirect syscalls implemented in Cobalt Strike. for easy access, here is UnpacMe Results 020b20098f808301cad6025fe7e2f93fa9f3d0cc5d3d0190f27cf0cd374bcf04. The sample is packed. The unpacking process is easy. Just put a breakpoint on `VirtualProtect` and get the base address (First Argument). Function `sub_18001B6B0` contains the important part, system call SSN retrieving and execution methods. You can get to this function by following the `call` instruction to `rax` which contains a `qword` memory area or a call to the `qword` directly. These locations are populated with addresses of the required APIs in this function. We can see multiple calls to `sub_18001A73C` with arguments: `qword_*`, a hash (such as `0B12B7A69h`), variable passed to the function `sub_18001A7F4` and another allocated memory which is also passed to `sub_18001A7F4`.

```
.text:000000018001B6E5                 and     [rbp+1E70h+arg_0], 0
.text:000000018001B6ED                 mov     edi, 1F4h
.text:000000018001B6F2                 lea     r8, [rbp+1E70h+arg_0]
.text:000000018001B6F9                 lea     rcx, [rsp+1F70h+var_1F40] ; void *
.text:000000018001B6FE                 mov     edx, edi
.text:000000018001B700                 call    sub_18001A7F4
.text:000000018001B705                 mov     rbx, [rbp+1E70h+arg_0]
.text:000000018001B70C                 lea     rax, qword_18004FFA0
.text:000000018001B713                 lea     rcx, [rsp+1F70h+var_1F40]
.text:000000018001B718                 mov     r8, rbx
.text:000000018001B71B                 mov     r9d, 0B12B7A69h
.text:000000018001B721                 mov     edx, edi
.text:000000018001B723                 mov     [rsp+1F70h+var_1F50], rax
.text:000000018001B728                 call    sub_18001A73C
.text:000000018001B72D                 lea     rax, qword_18004FFB8
.text:000000018001B734                 lea     rcx, [rsp+1F70h+var_1F40]
.text:000000018001B739                 mov     r9d, 0C508CF8Bh
.text:000000018001B73F                 mov     r8, rbx
.text:000000018001B742                 mov     edx, edi
.text:000000018001B744                 mov     [rsp+1F70h+var_1F50], rax
.text:000000018001B749                 call    sub_18001A73C
.text:000000018001B74E                 lea     rax, qword_18004FFD0
.text:000000018001B755                 lea     rcx, [rsp+1F70h+var_1F40]
.text:000000018001B75A                 mov     r9d, 35AF2123h
.text:000000018001B760                 mov     r8, rbx
.text:000000018001B763                 mov     edx, edi
.text:000000018001B765                 mov     [rsp+1F70h+var_1F50], rax
.text:000000018001B76A                 call    sub_18001A73C
.text:000000018001B76F                 lea     rax, qword_18004FFE8
.text:000000018001B776                 lea     rcx, [rsp+1F70h+var_1F40]
.text:000000018001B77B                 mov     r9d, 0B59BB90Bh
.text:000000018001B781                 mov     r8, rbx
.text:000000018001B784                 mov     edx, edi
.text:000000018001B786                 mov     [rsp+1F70h+var_1F50], rax
.text:000000018001B78B                 call    sub_18001A73C
.text:000000018001B790                 lea     rax, qword_180050000
.text:000000018001B797                 lea     rcx, [rsp+1F70h+var_1F40]
.text:000000018001B79C                 mov     r9d, 765EF075h
.text:000000018001B7A2                 mov     r8, rbx
.text:000000018001B7A5                 mov     edx, edi
.text:000000018001B7A7                 mov     [rsp+1F70h+var_1F50], rax
.text:000000018001B7AC                 call    sub_18001A73C
```

Function `sub_18001A73C` is to resolve the function address (`syscall` stub address) by the hash. And function `sub_18001A7F4` used to populate the list with the system call SSN and system call stub. So, `sub_18001A7F4` is our target. In the following picture is the beginning of the function.

```
.text:000000018001A80D                 sub     rsp, 30h
.text:000000018001A811                 mov     rax, gs:30h
.text:000000018001A81A                 mov     r13, rcx
.text:000000018001A81D                 mov     r8d, 20202020h
.text:000000018001A823                 mov     r9, [rax+_TEB.ProcessEnvironmentBlock]
.text:000000018001A827                 mov     r10, [r9+_PEB.Ldr]
.text:000000018001A82B                 add     r10, _PEB_LDR_DATA.InLoadOrderModuleList
.text:000000018001A82F                 mov     r9, [r10]
.text:000000018001A832
.text:000000018001A832 loc_18001A832:                          ; CODE XREF: sub_18001A7F4+5B↓j
.text:000000018001A832                                         ; sub_18001A7F4+65↓j ...
.text:000000018001A832                 cmp     r9, r10
.text:000000018001A835                 jz      loc_18001A9E4
```

The function starts with getting a pointer to the first entry in `InLoadOrderModuleList` structure by going through reading the Process Environment Block (PEB). here in the picture, r10 is holding the current entry of the structure and r9 is like a variable to get each entry, this is the breaking condition of the loop as the `_LIST_ENTRY` structure wrap around itself (doubly linked list).

The next step is to get the Export directory of `ntdll.dll` but first, get `ntdll` address in memory.

```
.text:000000018001A83B                 mov     rdi, [r9+LDR_DATA_TABLE_ENTRY.DllBase]
.text:000000018001A83F                 mov     r9, [r9]
.text:000000018001A842                 movsxd  rax, [rdi+IMAGE_DOS_HEADER.e_lfanew]
.text:000000018001A846                 mov     ecx, [rax+rdi+IMAGE_NT_HEADERS.OptionalHeader.DataDirectory.VirtualAddress]
.text:000000018001A84D                 test    ecx, ecx
.text:000000018001A84F                 jz      short loc_18001A832
.text:000000018001A851                 lea     rsi, [rdi+rcx]  ; data_dir_RVA + BaseAddress = address of IMAGE_DATA_DIRECTORY
.text:000000018001A851                                         ; First member is IMAGE_EXPORT_DIRECTORY
.text:000000018001A855                 cmp     [rsi+IMAGE_EXPORT_DIRECTORY.NumberOfNames], 0
.text:000000018001A859                 jz      short loc_18001A832
.text:000000018001A85B                 mov     ecx, [rsi+IMAGE_EXPORT_DIRECTORY.Name]
.text:000000018001A85E                 mov     eax, [rcx+rdi]  ; DLL name
.text:000000018001A861                 or      eax, r8d
.text:000000018001A864                 cmp     eax, 'ldtn'
.text:000000018001A869                 jnz     short loc_18001A832
.text:000000018001A86B                 mov     eax, [rcx+rdi+4]
.text:000000018001A86F                 or      eax, r8d
.text:000000018001A872                 cmp     eax, 'ld.l'
.text:000000018001A877                 jnz     short loc_18001A832
.text:000000018001A879                 movzx   eax, word ptr [rcx+rdi+8]
.text:000000018001A87E                 or      ax, 20h ; ' '
.text:000000018001A882                 cmp     ax, 'l'
.text:000000018001A886                 jnz     short loc_18001A832
.text:000000018001A888                 mov     ebx, [rsi+IMAGE_EXPORT_DIRECTORY.AddressOfFunctions]
.text:000000018001A88B                 mov     ebp, [rsi+IMAGE_EXPORT_DIRECTORY.AddressOfNames]
.text:000000018001A88E                 mov     eax, [rsi+IMAGE_EXPORT_DIRECTORY.AddressOfNameOrdinals]
.text:000000018001A891                 mov     r8d, edx
.text:000000018001A894                 add     rax, rdi
.text:000000018001A897                 add     rbx, rdi
.text:000000018001A89A                 add     rbp, rdi
.text:000000018001A89D                 mov     rcx, r13        ; void *
.text:000000018001A8A0                 shl     r8, 4           ; Size
.text:000000018001A8A4                 xor     edx, edx        ; Val
.text:000000018001A8A6                 mov     [rsp+68h+AddressOfFunctions], rbx
.text:000000018001A8AB                 mov     [rsp+68h+AddressOfNameOrdinals], rax
.text:000000018001A8B3                 mov     [rsp+68h+AddressOfNames], rbp
.text:000000018001A8B8                 xor     r15d, r15d
.text:000000018001A8BB                 call    memset
```

It is looking for the right module in the `InLoadOrderModuleList` by going through each entry, the `flink` is a pointer to `LDR_DATA_TABLE_ENTRY` where we can get a pointer to the module. By parsing the module (going through PE file headers) to get the name of the DLL which resides in the Export directory (First member) which is the first member of `IMAGE_DATA_DIRECTORY` structure. It is then tested to see if it is the target module (`ntdll`). If the module is `ntdll`, it saves a pointer to `AddressOfFunctions`, `AddressOfNames` and `AddressOfNameOrdinals`. A memory region of size 0x1f40 is then zeroed as it will hold the structures of the system call information needed. The next part is checking the function prefix `Ki` and `Zw`. It looks for only one function prefixed by `Ki` with the hash `8DCD4499h`, but I couldn't find function with this hash (using debugger). Then, a call to a hashing function is made. The hashing function is simple.

```
.text:000000018001A7C8 sub_18001A7C8   proc near               ; CODE XREF: sub_18001A7F4+F5↓p
.text:000000018001A7C8                                         ; sub_18001A7F4+138↓p
.text:000000018001A7C8                 xor     r9d, r9d
.text:000000018001A7CB                 mov     r8, rcx
.text:000000018001A7CE                 mov     eax, 52964EE9h
.text:000000018001A7D3                 cmp     [rcx], r9b
.text:000000018001A7D6                 jz      short locret_18001A7F2
.text:000000018001A7D8
.text:000000018001A7D8 loc_18001A7D8:                          ; CODE XREF: sub_18001A7C8+28↓j
.text:000000018001A7D8                 movzx   ecx, word ptr [rcx]
.text:000000018001A7DB                 mov     edx, eax
.text:000000018001A7DD                 inc     r9d
.text:000000018001A7E0                 ror     edx, 8
.text:000000018001A7E3                 add     edx, ecx
.text:000000018001A7E5                 mov     ecx, r9d
.text:000000018001A7E8                 add     rcx, r8
.text:000000018001A7EB                 xor     eax, edx
.text:000000018001A7ED                 cmp     byte ptr [rcx], 0
.text:000000018001A7F0                 jnz     short loc_18001A7D8
.text:000000018001A7F2
.text:000000018001A7F2 locret_18001A7F2:                       ; CODE XREF: sub_18001A7C8+E↑j
.text:000000018001A7F2                 rep retn
.text:000000018001A7F2 sub_18001A7C8   endp
.text:000000018001A7F2
```

It uses `0x52964EE9` as an initial key value to start the process then:

- Get 2-bytes of the Function name (little endian).
- Rotate the key by 8 (2 characters).
- Add the key and the 2-bytes of the name.
- Increment the counter by 1 (Resulting that all the chars in between the start and end taken two times in the calculation for example `ZwOpenProcess` will take `Wz` in the first iteration and `Ow` in the second and so on).
- The result of the addition is XORed with the key to produce the new key. The hash value returned is the last result of the XOR operation.

The resulting value is stored in the following form, in the pre-allocated space.

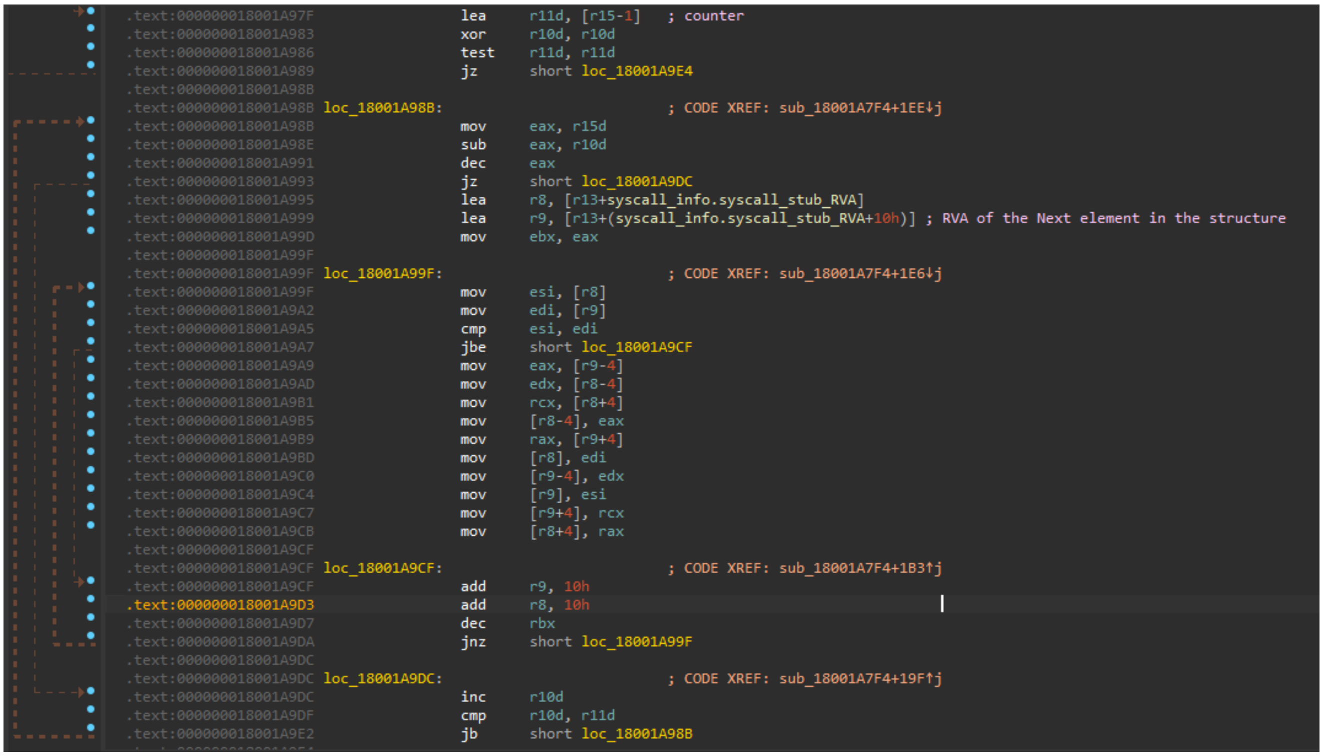
```
Address           Hex                                              ASCII
000000000013D6C0  62 7F CD 64 40 19 05 00 40 19 62 77 00 00 00 00  b.Íd@...@.bw....
000000000013D6D0  A1 0F 9C 9C 50 19 05 00 50 19 62 77 00 00 00 00  ¡...P...P.bw....
000000000013D6E0  CE 26 A2 DC A0 15 05 00 A0 15 62 77 00 00 00 00  Î&¢Ü ... .bw....
000000000013D6F0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000000013D700  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000000013D710  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000000013D720  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

- The first `DWORD` is the hash.
- The second `DWORD` is the Relative Virtual Address (RVA) of the system call0.
- The third `QWORD` is the Virtual Address (VA) of the system call stub (RVA + ntdll Base Address).

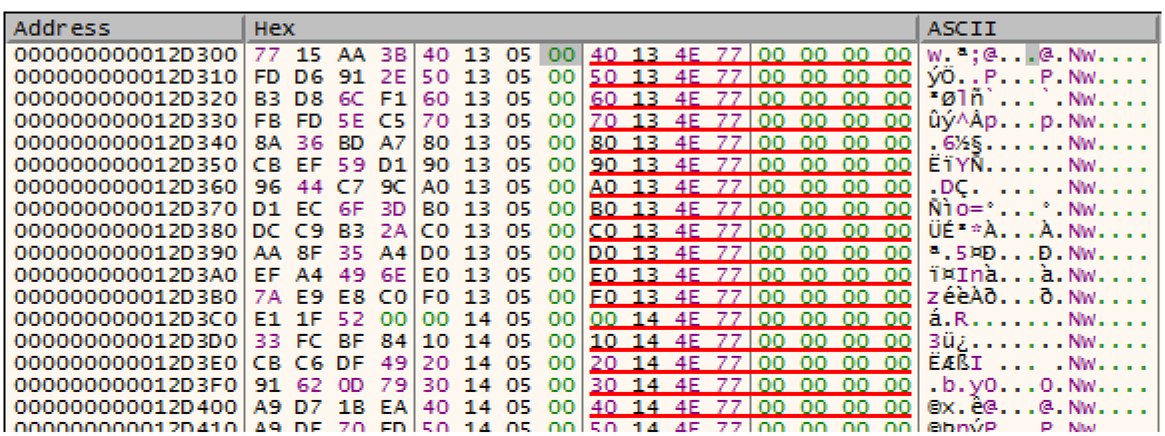
So, it can be written as:

```
struct syscall_info {
DWORD API_hash;
DWORD
syscall_stub_RVA;
QWORD
syscall_stub_address;
};
```

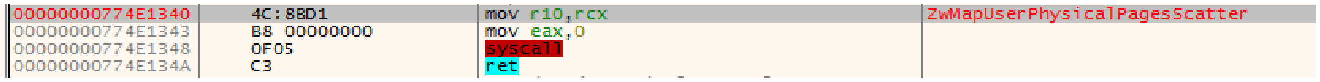
After populating the structure with the addresses. The structure elements are being sorted by the RVA of the system call stub (second entry in the structure).



After the sorting algorithm is done, the memory structure look like the following:



The first address is the address to the Lowest address `ZwMapUserPhysicalPagesScatter` (Could be different at newer versions of windows) at address `00000000774E1340` If we see the system call SSN of it:



system call number is zero. This is how it gets the SSN for any function, by iterating the structure to get the right hash, the counter will be used to get the SSN (SSN = counter). So far, this is remarkably like MDSec (8. Sorting by System Call Address) implementation of the technique known as `FreshlyCalls`. We could rewrite the technique using MDSec implementation as follows:

```
#define RVA2VA(Type, DllBase, Rva) (Type)((ULONG_PTR) DllBase + Rva)

static
void
GetSyscallList(PSYSCALL_LIST List) {
    PPEB_LDR_DATA           Ldr;
    PLDR_DATA_TABLE_ENTRY   LdrEntry;
    PIMAGE_DOS_HEADER       DosHeader;
    PIMAGE_NT_HEADERS       NtHeaders;
    DWORD                   i, j, NumberOfNames, VirtualAddress, Entries=0;
    PIMAGE_DATA_DIRECTORY   DataDirectory;
    PIMAGE_EXPORT_DIRECTORY ExportDirectory;
    PDWORD                  Functions;
    PDWORD                  Names;
    PWORD                   Ordinals;
    PCHAR                   DllName, FunctionName;
    PVOID                   DllBase;
    PSYSCALL_ENTRY          Table;
    SYSCALL_ENTRY           Entry;

    //
    // Get the DllBase address of NTDLL.dll
    // NTDLL is not guaranteed to be the second in the list.
    // so it's safer to loop through the full list and find it.
    Ldr = (PPEB_LDR_DATA)NtCurrentTeb()->ProcessEnvironmentBlock->Ldr;

    // For each DLL loaded
    for (LdrEntry=(PLDR_DATA_TABLE_ENTRY)Ldr->Reserved2[1];
         LdrEntry->DllBase != NULL;
         LdrEntry=(PLDR_DATA_TABLE_ENTRY)LdrEntry->Reserved1[0])
    {
      DllBase = LdrEntry->DllBase;
      DosHeader = (PIMAGE_DOS_HEADER)DllBase;
      NtHeaders = RVA2VA(PIMAGE_NT_HEADERS, DllBase, DosHeader->e_lfanew);
      DataDirectory = (PIMAGE_DATA_DIRECTORY)NtHeaders-
>OptionalHeader.DataDirectory;
      VirtualAddress =
DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress;
      if(VirtualAddress == 0) continue;

      ExportDirectory = (PIMAGE_EXPORT_DIRECTORY) RVA2VA(ULONG_PTR, DllBase,
VirtualAddress);
      DllName = RVA2VA(PCHAR,DllBase, ExportDirectory->Name);
```

```
      if((*(ULONG*)DllName | 0x20202020) != 'ldtn') continue;
      if((*(ULONG*)(DllName + 4) | 0x20202020) == 'ld.l') break;
    }

    NumberOfNames = ExportDirectory->NumberOfNames;

    Functions = RVA2VA(PDWORD,DllBase, ExportDirectory->AddressOfFunctions);
    Names     = RVA2VA(PDWORD,DllBase, ExportDirectory->AddressOfNames);
    Ordinals  = RVA2VA(PWORD, DllBase, ExportDirectory->AddressOfNameOrdinals);

    Table     = List->Table;

    do {
      FunctionName = RVA2VA(PCHAR, DllBase, Names[NumberOfNames-1]);
      if(*(USHORT*)FunctionName == 'iK' && HashSyscall(FunctionName) ==
0x8DCD4499)  {
        Table[Entries].API_Hash = HashSyscall(&FunctionName);
        Table[Entries].syscall_stub_RVA = Functions[Ordinals[NumberOfNames-1]];
        Table[Entries].syscall_stub_address = RVA2VA(void,
DllBase,Functions[Ordinals[NumberOfNames-1]]);

        Entries++;
        if(Entries == MAX_SYSCALLS) break;

      }
      if(*(USHORT*)FunctionName == 'wZ') {
        Table[Entries].API_Hash = HashSyscall(&FunctionName);
        Table[Entries].syscall_stub_RVA = Functions[Ordinals[NumberOfNames-1]];
        Table[Entries].syscall_stub_address = RVA2VA(void,
DllBase,Functions[Ordinals[NumberOfNames-1]]);

        Entries++;
        if(Entries == MAX_SYSCALLS) break;
      }
    } while (--NumberOfNames);

    //
    // Save total number of system calls found.
    //
    List->Entries = Entries;

    //
    // Sort the list by address in ascending order.
    //
    for(i=0; i<Entries - 1; i++) {
      for(j=0; j<Entries - i - 1; j++) {
        if(Table[j].syscall_stub_RVA > Table[j+1].syscall_stub_RVA) {
          //
          // Swap entries.
          //
          Entry.Hash = Table[j].Hash;
          Entry.Address = Table[j].Address;

          Table[j].API_Hash = Table[j+1].API_Hash;
          Table[j].syscall_stub_RVA = Table[j+1].syscall_stub_RVA;
          Table[j].syscall_stub_address = Table[j+1].syscall_stub_address;

          Table[j+1].API_Hash = Entry.API_Hash;
          Table[j+1].syscall_stub_RVA = Entry.syscall_stub_RVA;
          Table[j+1].syscall_stub_address = Entry.syscall_stub_address;
```

```
            }
        }
      }
  }
```

The next thing is to use the structure to get the SSN. and `syscall` instruction to call. This is done by function `sub_18001A73C`.

```
.text:000000018001A73C                 test    edx, edx
.text:000000018001A73E                 jz      short locret_18001A78B
.text:000000018001A740                 mov     [rsp+counter], rbx
.text:000000018001A745                 push    rdi
.text:000000018001A746                 sub     rsp, 20h
.text:000000018001A74A                 xor     edi, edi
.text:000000018001A74C                 mov     rax, rcx
.text:000000018001A74F
.text:000000018001A74F loc_18001A74F:                          ; CODE XREF: get_api+20↓j
.text:000000018001A74F                 cmp     [rax], r9d
.text:000000018001A752                 jz      short loc_18001A760
.text:000000018001A754                 inc     edi
.text:000000018001A756                 add     rax, 10h
.text:000000018001A75A                 cmp     edi, edx
.text:000000018001A75C                 jb      short loc_18001A74F
.text:000000018001A75E                 jmp     short loc_18001A781
.text:000000018001A760 ; ---------------------------------------------------------------
.text:000000018001A760
.text:000000018001A760 loc_18001A760:                          ; CODE XREF: get_api+16↑j
.text:000000018001A760                 mov     eax, edi
.text:000000018001A762                 add     rax, rax
.text:000000018001A765                 mov     rbx, [rcx+rax*8+8] ; API address
.text:000000018001A76A                 mov     rcx, rbx
.text:000000018001A76D                 call    get_syscall_ret_address
```

The function takes the following parameters:

- The array of structures that has the system call info (called `syscall_info` above)
- constant value 0x1F4 the maximum length of the structure members (structure size = 0x1F4 * 0x10).
- Pre-Allocated memory
- The function hash.
- Global variable to get the system call SSN and stub. The function is simple, it searches the populated structure to find the given hash. If it's found, the counter value is taken and to get the Address of the system call stub. To get the address, the base address of the structure is added to the offset multiplied by 0x10 (struct size) and add 8 to get the last QWORD.

```
API_Address = *(STRUCT_BASE_ADDR + COUNTER * 0x10
+ 8)
```

The address the passed to `get_syscall_ret_address` to get the `syscall ret` addresses to use it to execute the system call to bypass the callback mentioned before (call stack tracing is be used to detect this trick).

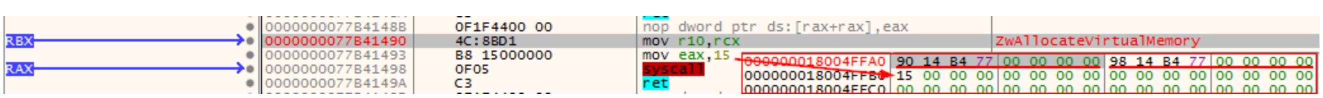
```
.text:000000018001A78C
.text:000000018001A78C                 mov     [rsp+arg_8], 50Fh ; syscall
.text:000000018001A793                 movzx   r8d, [rsp+arg_8]
.text:000000018001A799                 mov     r9b, 0C3h       ; ret
.text:000000018001A79C                 xor     eax, eax
.text:000000018001A79E
.text:000000018001A79E loc_18001A79E:                          ; CODE XREF: get_syscall_ret_address+28↓j
.text:000000018001A79E                 movsxd  rdx, eax
.text:000000018001A7A1                 cmp     r8w, [rdx+rcx]
.text:000000018001A7A6                 jnz     short loc_18001A7AF
.text:000000018001A7A8                 cmp     r9b, [rdx+rcx+2]
.text:000000018001A7AD                 jz      short loc_18001A7B9
.text:000000018001A7AF
.text:000000018001A7AF loc_18001A7AF:                          ; CODE XREF: get_syscall_ret_address+1A↑j
.text:000000018001A7AF                 inc     eax
.text:000000018001A7B1                 cmp     eax, 20h ; ' '
.text:000000018001A7B4                 jl      short loc_18001A79E
.text:000000018001A7B6                 xor     eax, eax
.text:000000018001A7B8                 retn
.text:000000018001A7B9 ; ---------------------------------------------------------------------------
.text:000000018001A7B9
.text:000000018001A7B9 loc_18001A7B9:                          ; CODE XREF: get_syscall_ret_address+21↑j
.text:000000018001A7B9                 cdqe
.text:000000018001A7BB                 add     rax, rcx
.text:000000018001A7BE                 retn
.text:000000018001A7BE get_syscall_ret_address endp
```

The global variable is used to store:

- QWORD to store System call address (function address at `ntdll`)
- QWORD to store `syscall` , `ret` instruction sequence address.
- DWORD to store system call number SSN. We can rewrite it as follows:

```
struct
syscall_required_addresses {
QWORD syscall_stub_address;
QWORD
syscall_intruction_address;
DWORD syscall_number;
};
```

(Creative names I know :))



There are some choices to call the required function. This is done based on the value at a global variable (0x18004BC6C):

- 1 : Direct call using the first member of the structure (Address of the function in `ntdll`)
- 2 : Indirect system call using trampoline jump using the system call number and the `syscall` address stored before.

```
.text:000000018001D0BF ; __int64 sub_18001D0BF()
.text:000000018001D0BF sub_18001D0BF   proc near               ; CODE XREF: sub_18001B11C+5C↑p
.text:000000018001D0BF                 mov     r11, cs:syscall_inst_addr
.text:000000018001D0C6                 mov     eax, cs:SSN
.text:000000018001D0CC                 mov     r10, rcx
.text:000000018001D0CF                 jmp     r11
.text:000000018001D0CF sub_18001D0BF   endp
.text:000000018001D0CF
```

anything else: Direct call to Win32 API.

Detecting syscalls

System calls can be used to bypass user mood hooks but there are other methods to detect Direct and Indirect syscalls. To detect Direct system calls, Windows provides a large set of callback functions, one of them is `KPROCESS!InstrumentationCallback` . This callback is triggered whenever the system returns from the kernel mode to user mode. This could be used to check the return address of the `syscall` which reveals the location of `syscall` instruction execution. This location should be `ntdll` but in case of the direct system calls, it will be from the `.text` section of the PE file. This was used by ScyllaHide. Indirect system calls solved this problem by getting the address of `syscall` instruction in `ntdll` and jump to it. To detect indirect syscalls the call stack tracing method can be used to check from where the system call originated -before jumping to `ntdll`-. This also can be bypassed by creating a new thread to get a new call stack using callback functions like `TpAllocWork` and `RtlQueueWorkItem`. If you want to know more about this, you can read Hiding In PlainSight 1&2

Note: This was personal notes I wrote when I was learning about syscalls, if there's anything not accurate, please let me know

References

https://www.mdsec.co.uk/2020/12/bypassing-user-mode-hooks-and-direct-invocation-of-system-calls-for-red-teams/

https://www.youtube.com/watch?v=elA_eiqWefw&t=3176s

https://offensivedefence.co.uk/posts/dinvoke-syscalls/

https://www.felixcloutier.com/x86/syscall.html

https://www.mdsec.co.uk/2022/04/resolving-system-service-numbers-using-the-exception-directory/

https://github.com/j00ru/windows-syscalls/

https://cocomelonc.github.io/malware/2023/06/07/syscalls-1.html https://www.crummie5.club/freshycalls/

https://github.com/x64dbg/ScyllaHide/blob/master/HookLibrary/HookedFunctions.cpp

https://eversinc33.com/posts/avoiding-direct-syscall-instructions/

https://redops.at/en/blog/direct-syscalls-a-journey-from-high-to-low

https://github.com/dodo-sec/Malware-Analysis/blob/main/Cobalt Strike/Indirect Syscalls.md

https://github.com/crummie5/FreshyCalls/blob/112bdf0db63a5f7104ba5243af6a672bc098a1ad/syscall.cpp#L65

https://0xdarkvortex.dev/hiding-in-plainsight/

https://0xdarkvortex.dev/proxying-dll-loads-for-hiding-etwti-stack-tracing/

Pikabot deep analysis