

Forum:[Hack Forums](#)

Thread: [Syscalls explained pt.1](#)

Jatamano - August 19, 2020, 9:07 am

Kernel Basics

The kernel in an operating system abstracts the interaction of a program with hardware.

What this means is that the kernel is developed for the platform it is running on, but programs are always made to interact with the kernel, hence a programmer can write 1 set of code for a specific instruction set and have it run on any hardware regardless of how input and output is handled.

To simplify this even further:

Imagine you have a setup that utilizes a certain output device like a graphics card.

As a programmer we do not actually want to program for that specific graphics adapter, hence the kernel might expose something like "drawsquare()" and when "drawsquare()" is called it will handle the process of making whatever is responsible for the graphics draw a square.

Now, there are absolutely vendor specific implementations on how the kernel should interact with a certain component: DRIVERS.

These were implemented due to the complexity of having to implement device interaction for each and every new device released.

We can ask the creators of said device to implement their own behavior.

This offloads a lot of work from the creators of the kernel AND also allows for modularity.

(a kernel cannot implement an abstraction for each and every device ever released.)

Most modern used CPU architectures have protection rings.

Protection rings allow for limitations on what a program running in a certain protection ring can do. For instance, in windows a program running in ring 0 is allowed full access to the physical hardware.

Unfortunately, writing about how these work on a more complex scale would be too extensive for this tutorial.

In much older windows versions (pre Windows NT, 2000, XP, etc) any program executed ran alongside the kernel in ring 0.

This posed issues, so in order to separate this, applications used by the user of the computer were set into ring 3 in the "usermode" space.

This meant that they no longer have the privilege of invoking direct kernel functions or interacting with the underlying hardware (cpu and etc).

Code from a higher ring when calling code in a lower ring would be calling a 'privileged instruction' = NOT allowed.

So how does a program call the kernel to interact with the underlying hardware?

The windows kernel allows for the calling of kernel ("ring 0") functions from usermode("ring 3") through the use of... *drum roll*... syscalls!

Syscalls essentially transfer control of the execution to the kernel; which handles the rest.

Function arguments before hand are setup as usual with 1 slight modification (see below).

(google: 'MSDN windows x64 calling convention' ;)

We also pass what syscall we are after into EAX, you can easily find a list of syscall ID's.
(google: 'windows x64 syscall table')

In windows x64 a syscall looks like this:
MOV EAX, THE SYSCALL ID (b8, ??, ??, ??, ??)

mov r10,rcx - (0x4c, 0x8b, 0xd1) convention says we pass first arg in r10 since syscalls clobber rcx.

syscall - (0x0F, 0x05) Call the syscall with ID EAX.

Put in practice

Well, its not exactly easy for all programmers to keep track of syscall id's.. that would be tedious.
Syscall ID's may change between windows versions/builds.

HENCE 'NTDLL.DLL'!

In all NT programs, NTDLL.dll is loaded!

What better than to have a DLL file export NATIVE WINDOWS API (NT) calls that will be used by programmers for kernel interaction!

Syscalls in raw asm do not depend on a function address, they are position independent.

Functions / programs that have to do with the kernel will call these ntFunctions.

If you call a function like ntAllocateVirtualMemory() it will go to a part of ntdll.dll where the proper syscall IDs are set and the syscall is made.

If you open the ntdll.dll file in a disassembler you will see the aforementioned system calls with their syscall id's in place.

WHY IS THIS IMPORTANT?!

Well lets say that maybe you are a sp00ky anti-cheat running in usermode and you want to see if someone is modifying memory in your application,
you will perhaps modify these exported functions (jmp instr to bla) in the ntdll.dll loaded into the memory of a game hack program!

This means that the code in ntdll will no longer be a syscall, but rather a middle man function that then calls the syscall after checking for sp00ky behavior.

SOOO essentially in this case if you would want to avoid this, you call the syscall by yourself and that sp00ky anti cheat cannot see you anymore!

Epilogue

I tried to explain it simply, understanding this concept opens up an entire world of fun!

If you have any questions: I'd be extremely happy to help, leave a comment!
Same goes for any corrections: in no way do I claim to have perfect or even good understanding, leave a comment!

Thank you for reading,
Jatamano

EDIT: Perhaps this should have been posted in coding general? :o

Lemme fix that, example code for a x64 program on win10x64, where we use syscall 0x18 for NtAllocVirtualMemory ("\\xB8\\x18"):

EDIT2: When running an WinX32 application in a WinX64 environment, you require a different approach.

See google: 'FastSysCall wow64'... for now.

Code #include <windows.h>

```
typedef NTSTATUS(NTAPI* pNtAllocVirtualMemory) (HANDLE ProcessHandle, PVOID*
BaseAddress, ULONG_PTR ZeroBits, PSIZE_T RegionSize, ULONG AllocationType, ULONG
Protect);
```

```
int main(){
    pNtAllocVirtualMemory NtAllocVirtualMemory =
(pNtAllocVirtualMemory)VirtualAllocEx(GetCurrentProcess(), NULL, 64,
    MEM_COMMIT, PAGE_EXECUTE_READWRITE);
```

```
    char shellcodeSc[] = "\\xB8\\x18\\x00\\x00\\x00\\x4C\\x8B\\xD1\\x0F\\x05\\xC3";
    memcpy(NtAllocVirtualMemory, shellcodeSc, sizeof shellcodeSc);
```

```
    PVOID baseAddr = NULL;
    SIZE_T allocSz = 4096;
    NTSTATUS ret = NtAllocVirtualMemory(GetCurrentProcess(), &baseAddr, 0, &allocSz,
    MEM_COMMIT, PAGE_EXECUTE_READWRITE);
```

```
    if(ret == 0) //OK!
        printf("We allocated memory using a syscall!\\n");
```

```
    return 0;
}
```

HFDone - August 19, 2020, 11:01 am

you forgot to clear that the syscall ID (number) is change between different windows versionS
64 System Call Table <https://j00ru.vexillium.org/syscalls/nt/64>

Jatamano - August 19, 2020, 11:25 am

(Aug 19, 2020 11:01 AM)HFDone Wrote: [\[CITE\]](#) you forgot to clear that the syscall ID (number) is
change between different windows versionS

64 System Call Table --

"Syscall ID's also change between kernel version and so forth.."

changed to:

"Syscall ID's may change between windows versions/builds."

That should make it more clear, thank you for reading!

HFDone - August 19, 2020, 12:01 pm

what about 32bit applications running under a WOW64 environment

Jatamano - August 19, 2020, 12:14 pm

(Aug 19, 2020 12:01 PM)HFDone Wrote: [\[CITE\]](#) what about 32bit applications running under a

WOW64 environment

FastSysCall would double the size of the thread!

It is not something I will cover now, perhaps part 2! ?

HFDone - *August 19, 2020, 12:43 pm*

<http://blog.rewolf.pl/blog/?p=102>