What Is a Syscall?

Definition

A system call (syscall) is a mechanism that allows a user-mode application to request services from the kernel-mode of the operating system. Since user-mode processes cannot directly access privileged memory or hardware, they invoke syscalls to:

- Allocate memory
- Open files
- Create processes or threads
- Access the Windows Registry

In Windows, syscall functions typically start with `Nt` or `Zw`, such as:

- `NtAllocateVirtualMemory`
- `NtReadVirtualMemory`
- `NtCreateThreadEx`
- `ZwCreateFile`

What Is a Syscall Stub?

Definition

A syscall stub is a small function—usually inside `ntdll.dll` that prepares registers and executes the `syscall` instruction. It acts as a wrapper for the transition to kernel-mode.

Purpose

- Loads the syscall number (SSN) into `EAX`
- Moves the first argument from `RCX` to `R10`
- Executes `syscall` to jump into kernel-mode

This stub is hookable by EDRs (Endpoint Detection and Response), which monitor or patch these functions to detect malicious use.

What Is SSN (System Service Number)?

Definition

An SSN is a unique integer assigned to each syscall. It tells the kernel which function to invoke from the System Service Dispatch Table (SSDT).

Example:

- On one version of Windows, `NtAllocateVirtualMemory` may have SSN `0x18`
- On another, it could be `0x1A`

Thus, SSNs are version-dependent a critical detail when implementing direct syscalls.

Direct Syscalls

What Are They?

Direct syscalls involve bypassing the standard API layers and invoking the `syscall` instruction directly from user-defined code. Instead of calling a function in `ntdll.dll`, the application includes its own syscall stub with the appropriate SSN.

Why Use Direct Syscalls?

Security solutions like Endpoint Detection and Response (EDR) systems often employ user-mode API hooking to monitor and intercept API calls. By implementing direct syscalls, attackers can evade these hooks, as their code doesn't rely on the hooked APIs.

Why This Bypasses EDR?

🔍 Attacker resolves syscall number for the function they want (e.g., `NtAllocateVirtualMemory`)

🧱 They build their own syscall stub (small assembly code that loads syscall number into `eax`/`rcx` and executes `syscall`)

🚀 The syscall executes → kernel takes over → EDR sees nothing at user level

Code:

```
mov r10, rcx            ; First parameter
mov eax, <syscall_number> ; syscall number for NtCreateThreadEx or any other
syscall                 ; Jump into kernel mode
Ret
```

```
mov r10, rcx        ; move the first parameter to r10 (required by Windows calling convention)
mov eax, 0x18       ; syscall number for NtAllocateVirtualMemory
syscall             ; trigger kernel execution
ret
```

🔍 Line-by-Line Explanation

🔸 `mov r10, rcx`

💬 "Copy the first argument (`rcx`) into `r10`."

- **Why?** Because when using the `syscall` instruction on **x64 Windows**, the first parameter must be in **`r10`**, not `rcx` — it's part of the **Windows syscall calling convention**.
- So: you **mirror `rcx` into `r10`** before calling the kernel.

📌 Without this line, your syscall could **crash** or pass the wrong value.

🔸 `mov eax, 0x18`

💬 "Set the syscall number in `eax`."

- Every Windows syscall has a unique number (like `NtAllocateVirtualMemory = 0x18` on some builds).
- The kernel checks `eax` to know **which function** you're trying to call.
- This is like **telling the receptionist at the kernel, “I want to talk to memory allocation services.”**

📌 These syscall numbers can change between Windows versions — that's why tools like **SysWhispers** exist.

🔸 `syscall`

💬 "Make the actual syscall."

- This is a **CPU instruction** (like `call` or `jmp`) that does one special thing:
  - Switches from **ring 3 (user mode)** to **ring 0 (kernel mode)**
  - Jumps into the kernel’s syscall handler
- Based on the value in `eax`, the kernel executes the right internal function.

📌 This is the heart of **direct syscall** — no userland API, no logging by EDR.

🔸 `ret`

💬 "Return back to the caller."

- After the syscall completes and you're back in user mode, `ret` ensures you **go back to the address that originally called this stub.**

Registr	What it's used for
`rcx`	First argument to a function (user-mode)
`r10`	Special: must hold `rcx` before a syscall (syscall convention)
`eax`	Holds the syscall number (like an ID)
`syscal`	CPU instruction to switch to kernel mode
`ret`	Go back to the code that called this stub

📌 Real-World Usage

Tools & Techniques that use direct syscalls:

- 🛠️ SysWhispers: Generates C-compatible syscall stubs
- 🛠️ Hell's Gate: Resolves syscall number dynamically
- 🧠 Malware like Cobalt Strike beacons (custom builds)

Indirect Syscalls

What Are They?

Indirect syscalls aim to retain the benefits of direct syscalls while mimicking legitimate behavior more closely. Instead of executing the `syscall` instruction directly, the application jumps to the `syscall` instruction within `ntdll.dll`.

Why Use Indirect Syscalls?

By executing the `syscall` instruction within `ntdll.dll`, indirect syscalls:

- Avoid user-mode API hooks by not calling the API directly.
- Maintain a call stack that appears more legitimate, reducing detection risk.

YourApp.exe

↳ kernel32.dll (VirtualAlloc)

↳ ntdll.dll (NtAllocateVirtualMemory)

↳ syscall → kernel

- But in **indirect syscall**, you skip the `NtAllocateVirtualMemory` *function call* and **jump straight to the syscall stub**, like this:

txt

CopyEdit

```
YourApp.exe

  ↳ ??? (no normal caller)

      ↳ ntdll.dll+0x1234 (syscall)
```