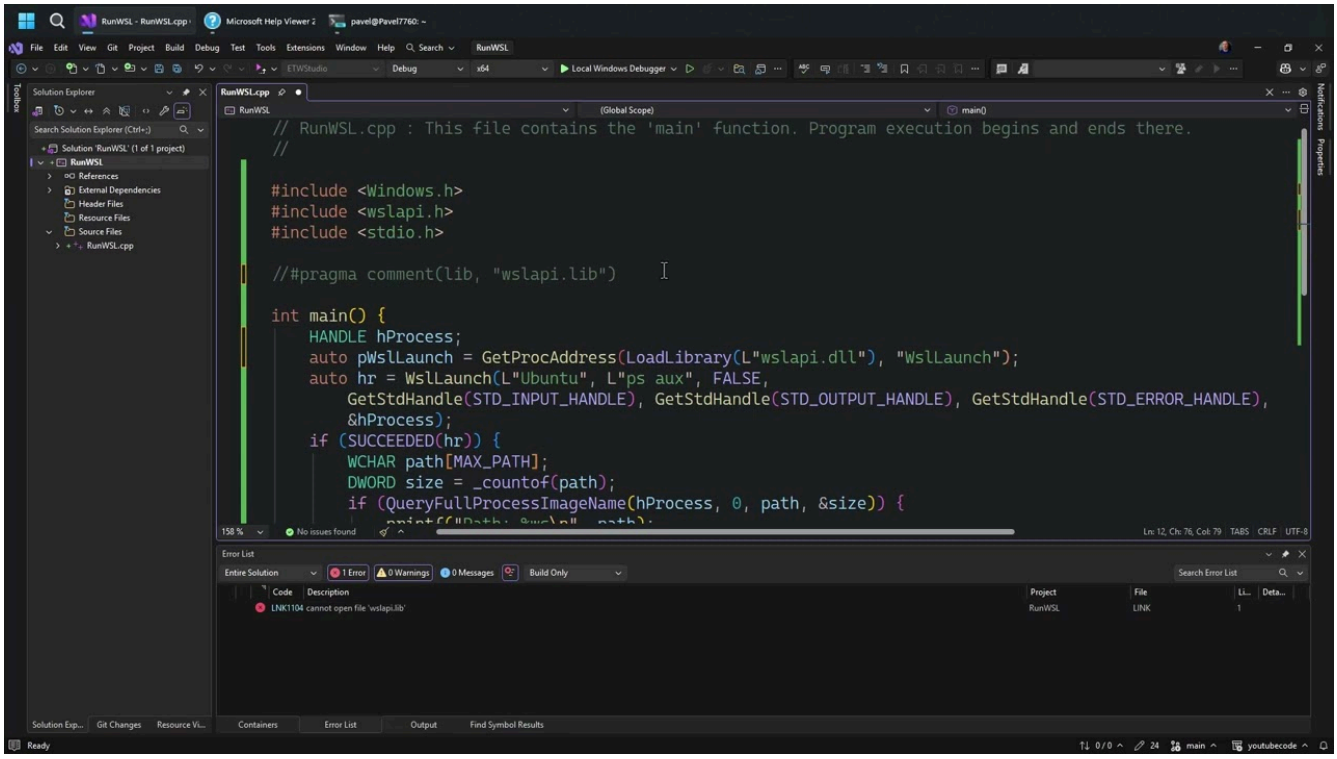
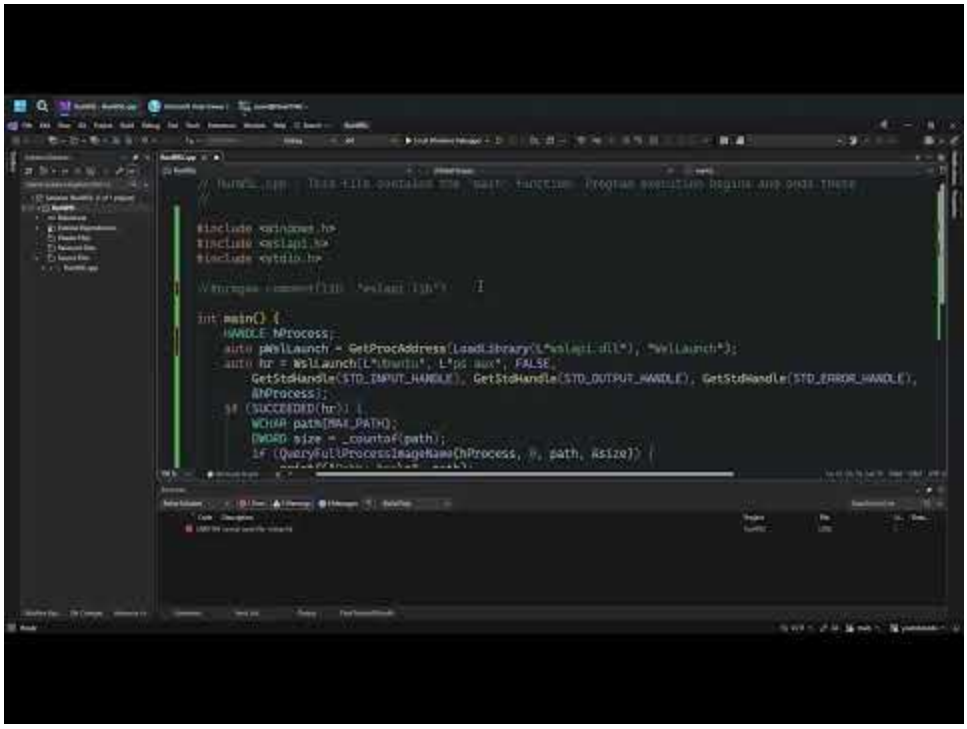
Launch WSL Applications from Windows with WslLaunch

trainsec.net/library/windows-kernel/launch-wsl-applications-from-windows-with-wsllaunch

Pavel Yosifovich April 21, 2026





Watch Video At: https://youtu.be/GTONQpqCFvw

Introduction

The Windows Subsystem for Linux runs Linux applications natively on Windows, no recompile required. You take an ELF binary and it just runs. Most people interact with it the obvious way: open a WSL shell and launch things from inside Linux.

But there is a less obvious direction. You can launch a Linux process from a Windows process. Directly. With a proper Win32 API call, and even standard handles can be specified. There are at least two ways to do this, and in this post I will walk through one of them: `WslLaunch`.

A quick note on WSL versions, because it matters for some of what we are about to see. WSL 1 emulates the Linux kernel inside the Windows kernel and idenities Linux processes as pico processes. It is still supported, but it did not scale well. WSL 2, the current version, is a full Linux system including its own kernel, running in what Microsoft calls a lightweight utility VM. Microsoft maintains its own fork of the Linux kernel for WSL 2, and it is open source. The technique in this post works on both versions without any code change.

The API

The API we want is declared in `wslapi.h`. The function of interest is `WslLaunch`, and its signature looks like this:

```
HRESULT WslLaunch(
    PCWSTR  distributionName,
    PCWSTR  command,
    BOOL    useCurrentWorkingDirectory,
    HANDLE  stdIn,
    HANDLE  stdOut,
    HANDLE  stdErr,
    PHANDLE process
);
```

Six inputs and one output. Let us go through them in order.

`distributionName` is the Linux distro to run the command in. On my machine I have Ubuntu (WSL 2) and Ubuntu-20.04 (WSL 1). You can list them with **`wsl -l -v`** from a command prompt.

`command` is the Linux command line. Anything you would type at a shell prompt works here: `ps aux`, `ls -l`, or a full path to some Linux executable with arguments.

`useCurrentWorkingDirectory` is a BOOL. If TRUE, the Linux process inherits the current directory of the Windows process. If FALSE, it uses the default Linux home directory for the distro. For something like **`ps aux`** it does not matter, but for a command that reads files it usually does.

Then come **`stdIn`**, **`stdOut`**, **`stdErr`**. These are Windows HANDLEs. Whatever the Linux process reads from standard input will come from `stdIn`, and whatever it writes will go to `stdOut` and `stdErr`. This is the convenient part: the handles are just Windows handles, so you can pass console handles, pipes, file handles, anything that makes sense.

The last parameter is an output HANDLE, and this is where things get interesting. More on that in a moment.

The Missing Import Library

Time to actually call the function. We include `<wslapi.h>`, write the call, and try to build. The compiler is happy. The linker is not:

If you look at the official documentation, Microsoft tells you to link against **`WslApi.lib`**. Unfortunately, that file does not exist. I searched my entire system for it, and it is not there. The DLL that implements the function, **`wslapi.dll`**, does exist in `System32`. But the import library that would let the linker resolve the symbol statically is missing. Whether that is a documentation bug or an oversight in the SDK, I do not know. Either way, we need a workaround.

Two options. We can build our own import library from the DLL, which is not very difficult but takes some time. Or we can skip the import library altogether and bind to the function dynamically with `LoadLibrary` and `GetProcAddress`. The second option is easier and perfectly reasonable here.

```
HMODULE hLib = LoadLibrary(L"wslapi.dll");
auto pWslLaunch = reinterpret_cast<decltype(&WslLaunch)>(
    GetProcAddress(hLib, "WslLaunch"));
```

The `decltype(&WslLaunch)` trick gives us the correct function pointer type without having to write it out by hand. The header file already declares the signature, so we let the compiler derive the type for us.

`wslapi.dll` lives in `System32`, so `LoadLibrary` finds it without a full path. From here on we call through `pWslLaunch` instead of `WslLaunch`.

Wiring the Standard Handles

For this example I want the Linux process to read from and write to the same console my Windows application is using. The simplest way to get those handles is `GetStdHandle`:

For a console application these come from the keyboard and the console output buffer. The Linux process will read its stdin from the keyboard and write its stdout and stderr to the same console we are running in. **`ps aux`** does not read stdin, so passing the console handle for `hIn` is harmless here.

You could just as easily open a file and pass its handle for `hOut`. The Linux process writes, the bytes land in the file, and nothing cares that one side is Linux and the other is Windows. That is the useful part of this API.

The Call

Now we put it all together:

Run it, and the console fills with the output of `ps aux`, exactly as if you had typed it inside a WSL shell. The bridging is transparent: UTF-8 bytes from the Linux side come out of the Windows console as if they had been printed by a Windows process.

What is That HANDLE, Exactly?

The last parameter is the one that needs some explanation. `WslLaunch` returns a HANDLE in the output parameter, and the natural question is: a handle to what?

In WSL 2, there is no Windows process representing the Linux process. The Linux process runs inside a utility VM with its own kernel. It has a PID, but that PID is a Linux PID, not a Windows PID, and there is no Windows kernel object to get a handle to.

So what is the HANDLE pointing at? It is pointing at an intermediary Windows process: **`WSL.exe`** itself. **`WSL.exe`** is the thing that talks to the utility VM and sets up the bridging. The handle we get back keeps that process alive long enough for us to inspect it.

In WSL 1 the situation is different because of pico processes, but the API returns the same kind of handle for consistency.

You can verify this with Process Explorer. Run the application, break in the debugger while the handle is still live, find its value, and locate it in Process Explorer's handle view. You will see a process handle with a name. On my machine it points to `WSL.exe`, and by the time you

look the process has already exited (`ps aux` is fast). So what you actually have is a handle to a zombie process.

$2,111

$1,478 or $150 X 10 payments

Windows Master Developer

Takes you from a “generic” C programmer to a master Windows programmer in user mode and kernel mode.

Become Windows Master Developer

Zombie Processes and QueryFullProcessImageName

Once we have the handle, the obvious thing to try is to ask the system what executable it belongs to. The first API most people reach for is `QueryFullProcessImageName`:

This does not work. It returns an error, and `GetLastError` reports something unhelpful like `ERROR_GEN_FAILURE`.

The reason is worth understanding. `QueryFullProcessImageName` reads the image path from the Process Environment Block (PEB). The PEB lives in the process's user address space. When the process exits, the address space is torn down. The kernel object (the `EPROCESS`) stays around as long as someone holds a handle to it, so the handle itself is valid. But there is no address space behind it anymore. There is no PEB to read from. So the function fails.

This is a general property of zombie processes on Windows. Anything that reads from user-mode memory of the target process will fail once the process has exited, even if the handle is still valid. This has practical implications for any tool that walks process handles: you cannot rely on user-mode queries for processes that might already be dead.

GetProcessImageFileName: Going Through the Kernel

The fix is to use an API that does not rely on the PEB. `GetProcessImageFileName` from `psapi.h` goes through the kernel instead:

This works, even on a zombie process, because the kernel keeps the image file information attached to the EPROCESS, not in user-mode memory. As long as the kernel object exists, the answer is available.

There is one cosmetic oddity. The path comes back in NT device form, something like:

Instead of drive letters. If you need a DOS-form path you can translate it yourself, but for logging and identification purposes the NT form is fine. It confirms that the handle points to `WSL.exe`, which is what we expected.

Key Takeaways

- `WslLaunch` lets you launch a Linux process from a Windows process, with control over stdin, stdout, and stderr.

- The documented `WslApi.lib` import library does not exist. Bind to `WslLaunch` dynamically via `LoadLibrary("wslapi.dll")` and `GetProcAddress`. Use `decltype(WslLaunch)*` to get the function pointer type.
- The returned HANDLE is not a handle to the Linux process. It is a handle to `WSL.exe`, the intermediary process.
- `QueryFullProcessImageName` fails on zombie processes because it reads the PEB, which is gone once the process exits. Use `GetProcessImageFileName` from `psapi.h` instead, which queries the kernel.
- The technique works identically on WSL 1 and WSL 2, no code changes needed.

Keep Learning

If you want to go deeper on the Win32 APIs behind all of this, take a look at the Windows System Programming 1 and Windows System Programming 2 courses at TrainSec. They cover process and handle management, standard I/O, and the kernel objects that this post depends on.

For the kernel side of the picture, the Windows Internals series covers how processes, handles, and the PEB actually work under the hood. Start with Windows Internals: Day 1.