Curso de Bash Scripting

Notas previas: Hace unos meses comencé a trabajar en este curso de scripting en bash gracias a un creador de contenido bastante especial para mí. Antes de comenzarlo tenía la idea de subirlo en Udemy, sin embargo quienes me siguen conocen que no llevo especialmente mucho tiempo en la creación de contenido ni tengo los recursos económicos para costearme todos los recursos necesarios para crear un curso de gran calidad (micrófono, cámara, etcétera) por lo que finalmente termine el curso y quede algo decepcionado con que a veces la edición no fuera buena, con cambios en el audio dada la mala calidad del micrófono de unos audífonos rotos cuyo micrófono integrado utilice y termine decidiendo no subir el curso, puesto que no quisiera intentar vender algo que no posee una calidad excelente. Habiendo tomado esa decisión no estaba seguro de si subir el curso de manera gratuita o por otra parte no hacerlo, puesto que estos fallos de calidad hablan un poco mal de mi imagen pública y tal. Aun a sabiendas de esto, es mejor hablar que no hablar nada, he dejado de subir mucho contenido al no sentirme del todo satisfecho con todo lo que creo, pero creo que es mejor que a pesar de esa insatisfacción me mantenga subiendo contenido, de otra forma jamás podre mejorar y terminar creando videos que me hagan sentir satisfecho conmigo mismo. Además si todo lo anterior fuera el punto más importante probablemente las circunstancias serian distintas y ustedes no estuvieran leyendo esto. Pero en realidad existe un factor con más influencia y es que el objetivo fundamental es enseñar a las personas que como yo quieren aprender de este mundo, estas que se sienten cautivadas por las tecnologías informáticas, y ustedes merecen toda la información del mundo; con mayor o menor calidad el curso posee información que de seguro le será valiosa a más de uno de ustedes, pues para mi habría sido valiosa en su momento, y dado que aprendí bash scripting en un curso de pago, no se me ocurre una mejor manera de compensar la falta de recursos que con enseñanzas que les puedan ser útiles y prácticas. En fin, no tengo derecho a privarlos de los conocimientos que les pueda transmitir, por ello publico este PDF, el cual deberá ser distribuido de manera gratuita. Lamento de ante mano cualquier error que pueda existir en el mismo, sea ortográfico o cualquier otro, conste que aún no estaba listo para ser publicado y que no preste gran importancia a corregir puntualmente todo al respecto. Y sin más royos los dejo con este curso no sin antes informarles que existe también un formato en videos para quien quiera verlo, podrá encontrarlo en mi canal de YouTube también de manera gratuita claro: https://youtube.com/c/black0utx.

Un breve preámbulo

En este capítulo simplemente voy a comentarles varios temas que deberían conocer antes de ver el curso. Primeramente me gustaría recordar que uno de los requisitos previos es tener mínimos conocimientos sobre Linux. Lo cual incluye por lo menos saber instalarlo y algún que otro conocimiento básico sobre cómo funciona y en qué consiste, puesto que este curso a pesar de que es introductorio y que comenzaremos desde 0 en un terminal, a fin de cuentas es un curso de bash scripting. No de Linux, por lo cual asumiré que los conceptos necesarios sobre sistemas operativos, arquitectura de las distribuciones de GNU/Linux y demás ya lo tenéis. En caso de no ser así, igualmente pueden ver el curso, a fin de cuentas comenzaremos desde 0 y seguramente podrás aprender de igual modo. Pero es sumamente recomendable tener conocimientos previos, bajos.

Pasar el curso es relativamente simple. Estaré explicando todos los puntos necesarios para aprender y para posteriormente comenzar en scripting en bash. Pero los exhorto a hacer dos cosas que únicamente dependen de ustedes. La primera, a pesar de les daré todo bastante masticado desde el comienzo. Es recomendable que no se quede únicamente con lo que yo le explique, yo puedo cometer algún error en cuanto algún concepto. O quizás cierta explicación no le quede claro y necesite ver otras explicaciones para combinándolas todas poder sacar una conclusión de en qué consiste cierto aspecto hipotético. Por lo que en resumidas cuentas, investiguen y vayan más allá. Lo segundo que queda por vosotros es que de poco va a servir que vean el curso tal cual. La mejor manera de aprender, de tomar confianza con el terminal es practicar, ósea ver lo que yo haga y replicarlo, apuntar todo lo que no entiendan para posteriormente buscar más sobre el tema. También pueden buscarlo en el libro (este) en donde estarán cubiertas casi todas las palabras que diga en el curso y por ende por escrito quedara todo archivado para poder repasar posteriormente.

Otra nota importante es que yo utilizo Kali Linux como distribución. Es una distribución orientada a profesionales de la ciberseguridad. Usted no tiene por qué utilizarla. Fácilmente puede utilizar cualquier otra distribución, preferiblemente alguna basada en Debian, podría ser el mismo Debian o Ubuntu en caso de que sean principiantes. En caso de que ya tengan conocimientos intermedios no es necesario recomendarles ninguna distribución.

Sin más, vamos a comenzar con un curso de Bash Scripting del cual tengo la absoluta certeza de que si completas. Saldrás de aquí conociendo Bash, o conociéndolo mucho mejor que como lo conozcas ahora. Sin más, nos vemos en el próximo capítulo.

Contáctame:

youtube: https://youtube.com/c/black0utx

twitter: https://twitter/Black0utq

Comunidad de Discord: https://discord.gg/JSAW5nTMDB

Canal de Telegram: https://t.me/B14ck0u7

Grupo de Telegram: https://t.me/black0utx

Sin más, no dudes en preguntar cualquier cosa que necesites, estaré atento.

PD: Te sugiero escribirme por twitter, como que no soy muy activo por otros lares :/

Primer encuentro con el terminal

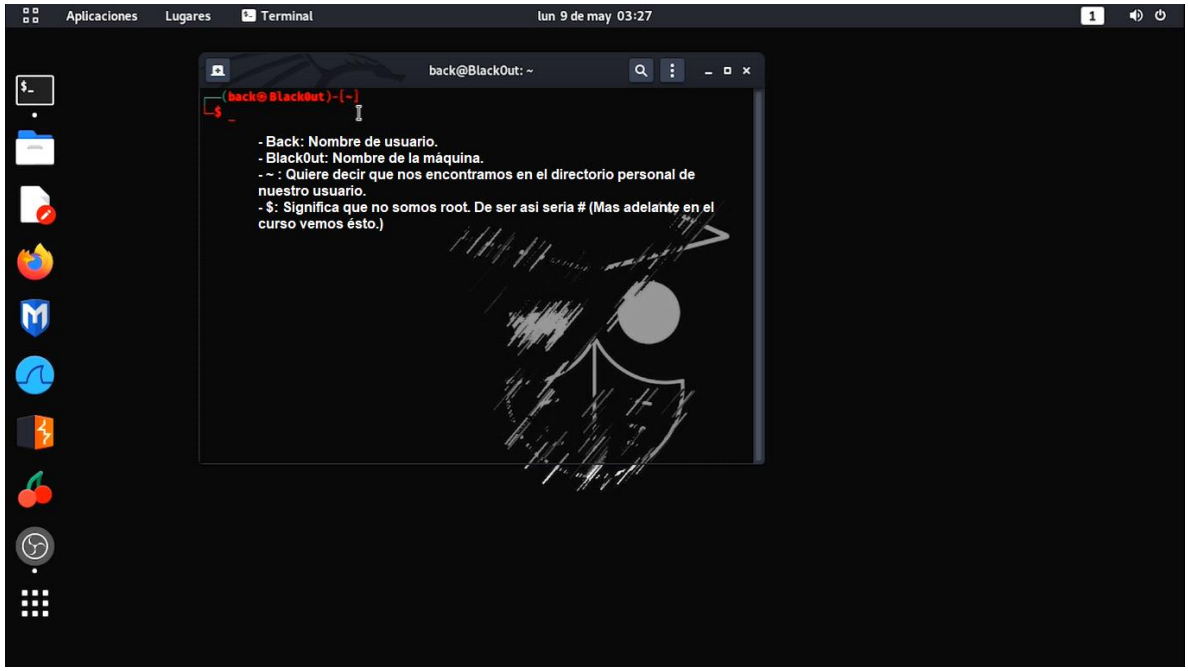
Vamos a comenzar utilizando un terminal, puesto que si utilizáis Linux, sea por las razones que lo prefieran al fin y al cavo vais a necesitar trabajar una

¿Dónde se encuentra la terminal?

Primeramente y quiero que esto quede claro, lo que estamos viendo es un emulador de terminal, realmente un terminal sería un sistema operativo que no tenga ningún tipo de interface gráfica.

Naturalmente en tu caso seguramente tengas una interface gráfica, donde puedas moverte más naturalmente, con lo cual en realidad este es un emulador, no una terminal en si

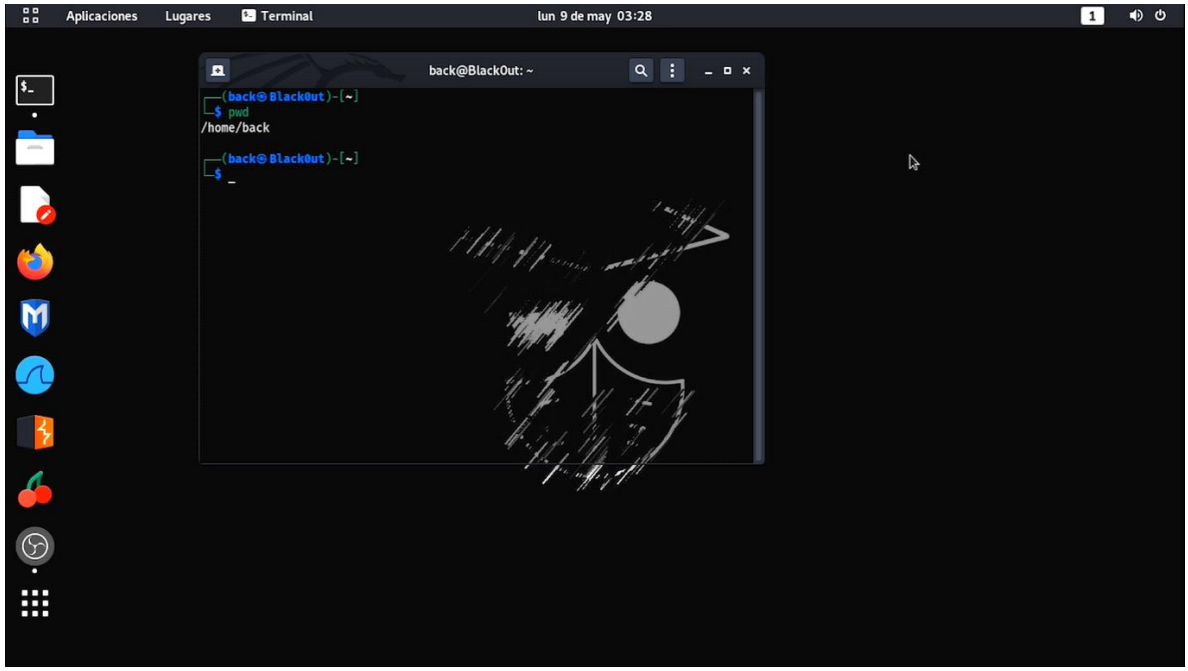
Aquí vemos el prompt, el cual nos proporciona un poco de información, por ejemplo el usuario, el hostname seguido del directorio en que nos encontramos



También tenemos este **~** que en realidad se llama “comodín” y se usa para referirse al directorio personal de nuestro usuario.

Si queremos saber el directorio en el cual nos encontramos bastaría con escribir el que sería nuestro primer comando si nunca has utilizado una terminal, el cual sería:

pwd



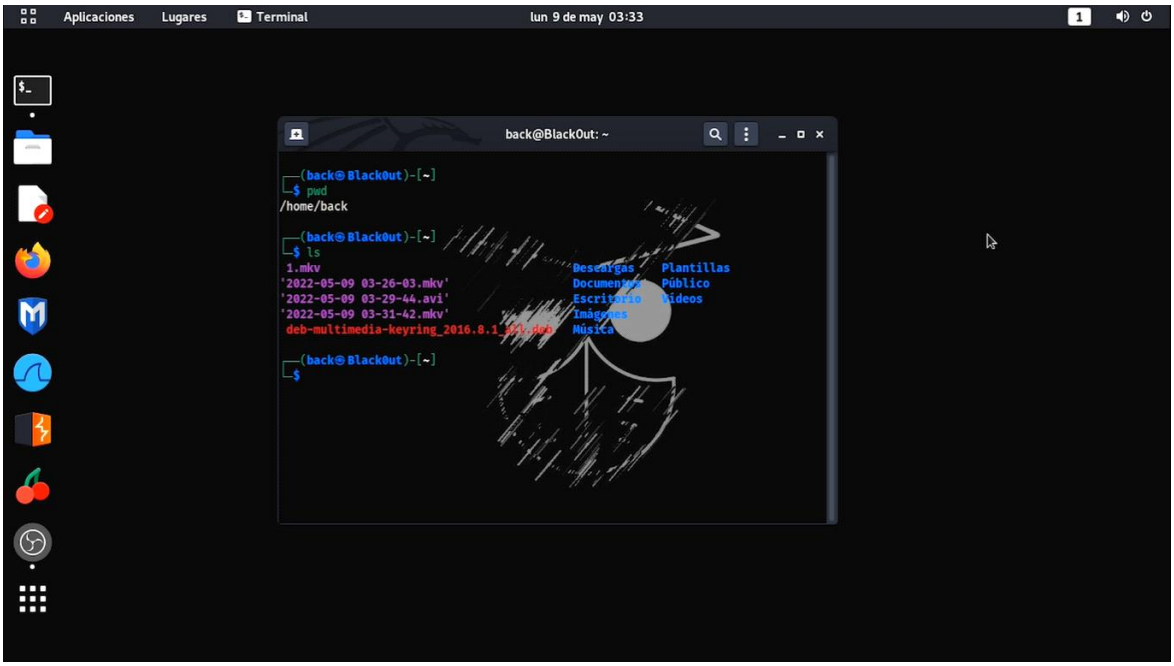
Y lo que significa es: ***Print Working Directory***

Ficheros y rutas

Primeramente desde el terminal lo que vamos a hacer será ver cómo movernos de un directorio a otro, por ejemplo en el momento en el que abres una carpeta se podría decir que te estás moviendo a ella. Me gustaría recalcar que el concepto de carpeta y directorio es relativamente idéntico en este sentido, del mismo modo que vamos a ver el proceso idéntico de moverse entre carpetas pero en este caso desde el emulador de terminal.

Por lo que sabemos si escribimos el comando ***pwd.***

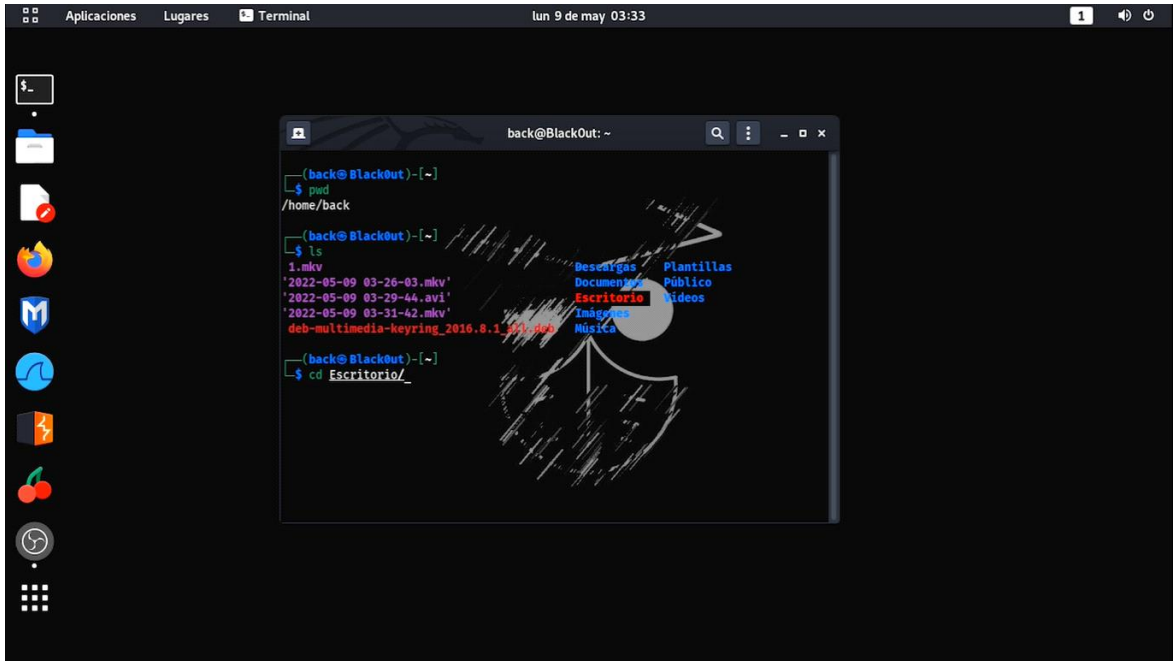
Podemos ver que nos encontramos en el directorio personal de nuestro usuario, pero para saber a dónde podríamos movernos primero deberíamos ver que es lo que se encuentra dentro del mismo. Para ello podríamos hacerlo con el comando ***ls***.



Fíjate que básicamente lo que hace es mostrarme, no solo los directorios, sino también los ficheros que se encuentran en la ruta actual en la que me encuentro, que repito es la de nuestro usuario personal, además pueden verlo desde arriba, en el prompt

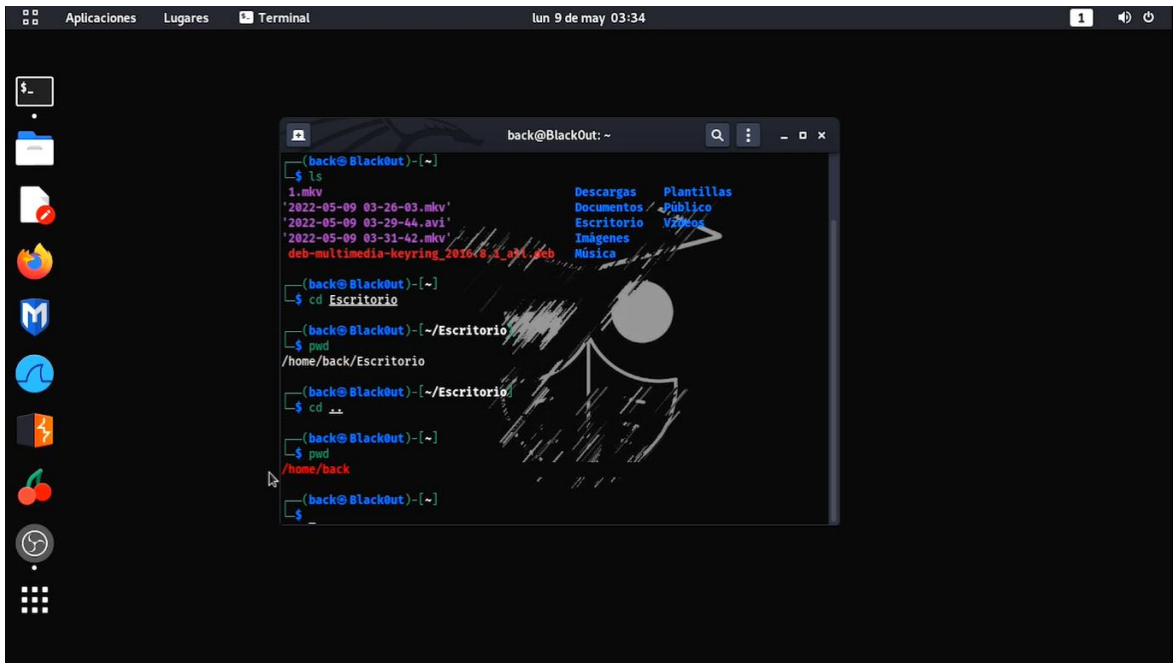
Pues si se fijan tenemos varias cosas, si fuéramos desde una interface gráfica podríamos darnos cuenta de que en efecto, el contenido que estamos viendo desde el terminal es el mismo.

Entonces, podríamos movernos a alguna de esos directorios, dígase el escritorio por poner un ejemplo, si hiciéramos uso del comando ***cd***, el cual significa ***change directory.***



Meramente con esto nos habríamos movido hacia el escritorio. ¿Pero y si ahora queremos ir hacia atrás?

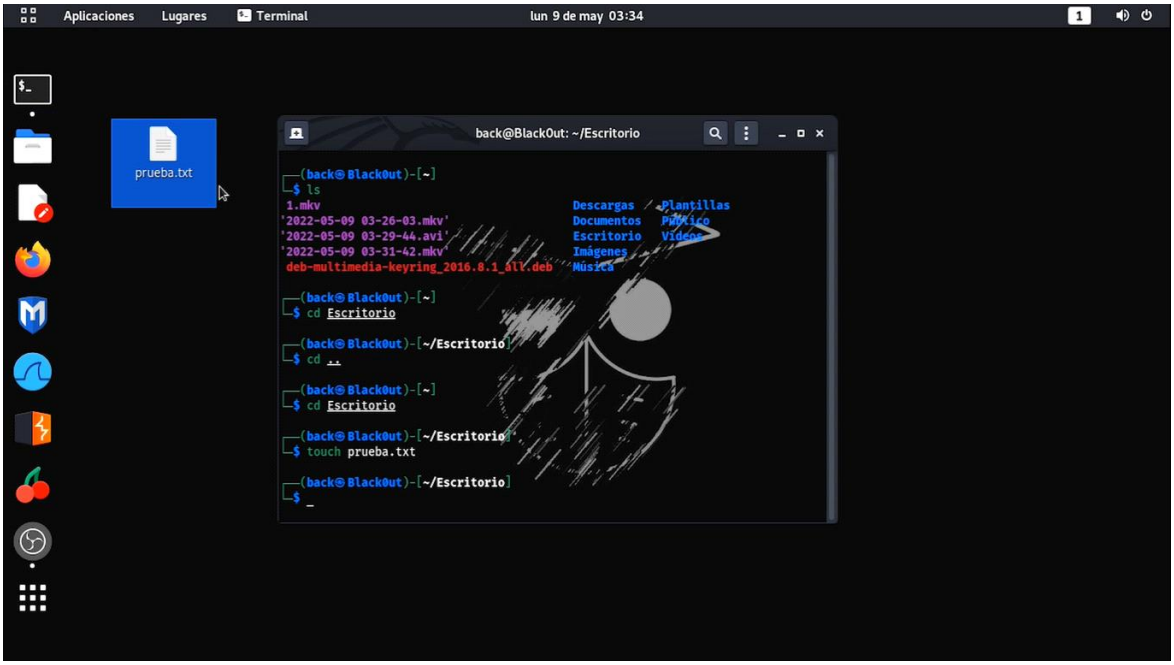
No hay problema, podemos hacerlo con ***cd*** seguido de **..**



y si hacemos un ***pwd*** podemos ver que efectivamente estamos nuevamente en el directorio personal de usuario. Pero bueno, vamos a movernos al escritorio para ver

ejemplos igualmente de cómo podríamos crear ficheros desde el terminal.
Para crear un fichero, no confundir fichero con directorio. Simplemente podemos hacerlo con el comando ***touch*** seguido del nombre que queramos asignar a dicho fichero.

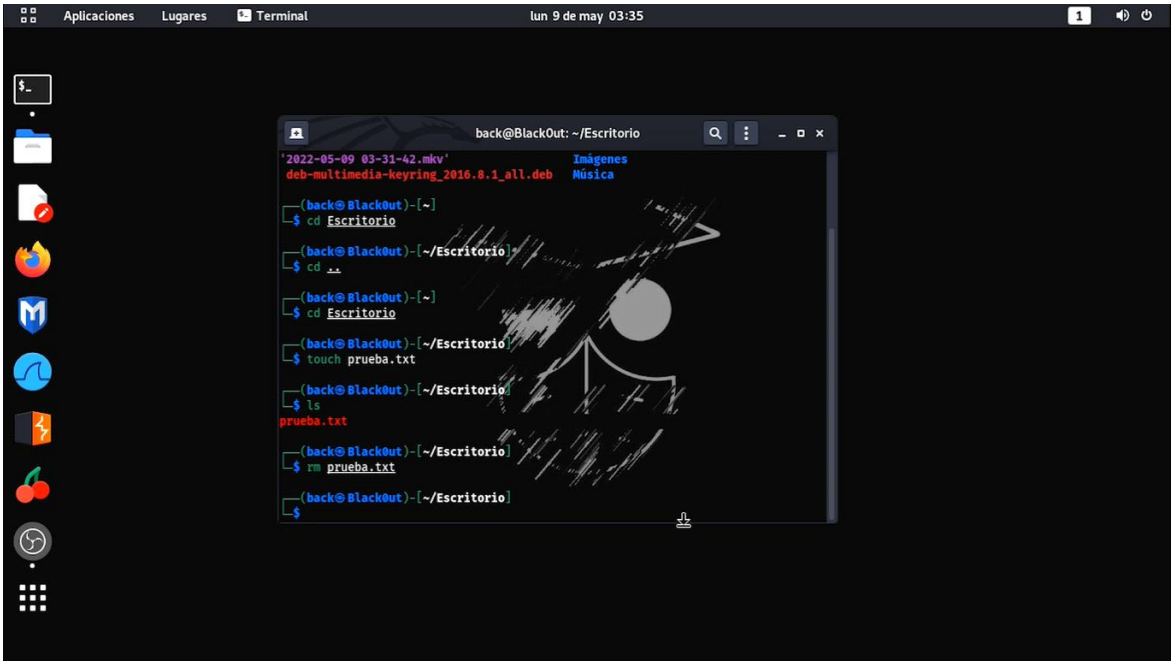
por ejemplo:



Y por cierto, si hacemos un ls podemos ver que efectivamente ahí se encuentra el fichero.

¿Y bueno, como borramos ese fichero?

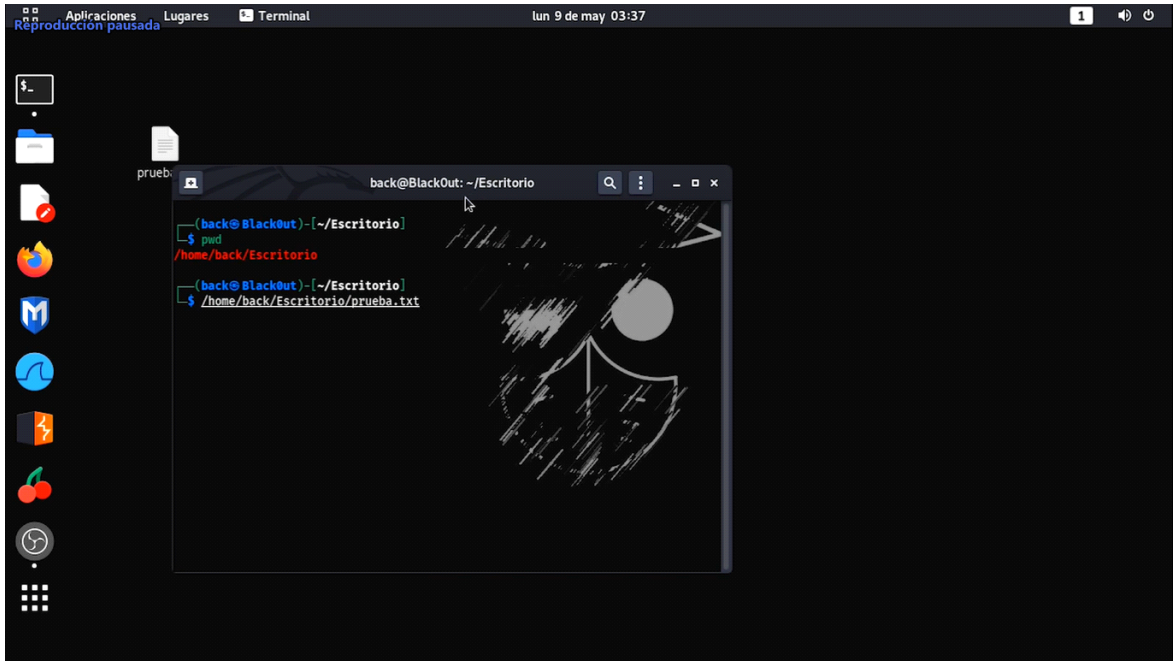
Pues podemos hacerlo con el comando ***rm***, que viene de ***remove***. seguido del fichero, en este caso de prueba.txt



Digamos que ahora quisiéramos volver a crearlo, para aprovechar así la ocasión. Para esto no tendríamos que volver a escribir el comando completo, sino que tenemos como que una especie de historial de los comandos que hemos ido utilizando por la cual podemos navegar con la flecha de arriba y abajo en su teclado.

Y aprovechando para hablar de rutas, hemos estado creando ficheros hasta ahora de una manera bastante simple. En realidad, lo estamos haciendo desde la ruta relativa ¿qué quiero decir?

Actualmente bajo el directorio actual en el que nos encontramos la ruta del fichero es meramente prueba.txt, pero la ruta absoluta no es esa, sino que si hacemos un ***pwd*** para ver la ruta:



Esta sería la ruta absoluta en la cual se encuentra el mismo.

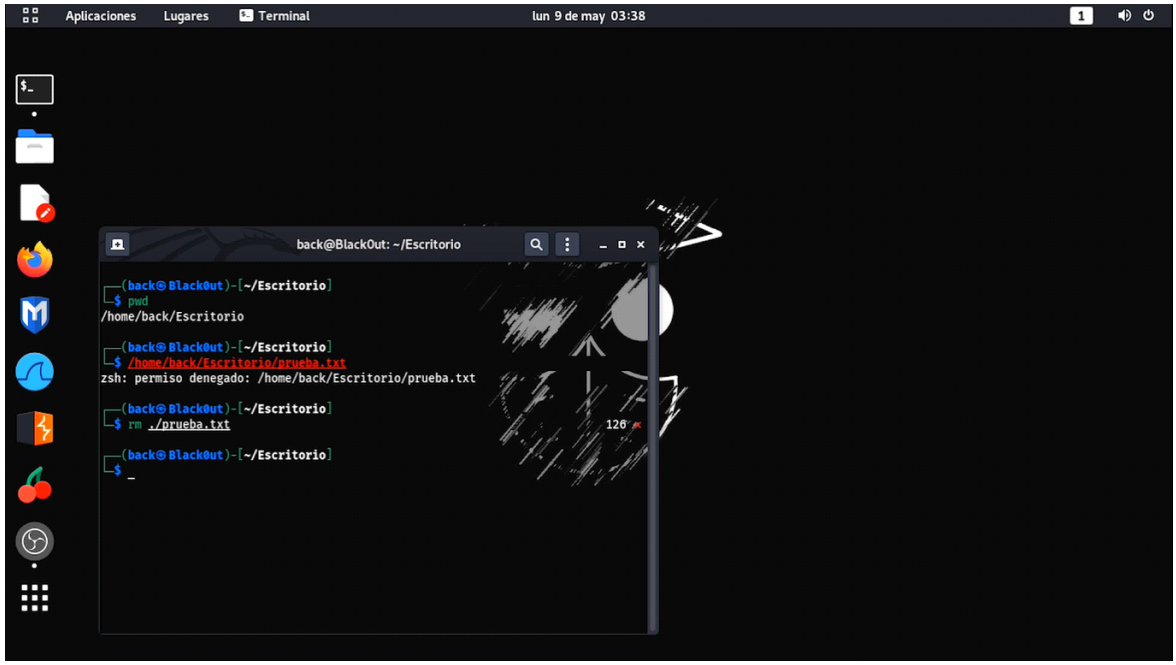
Ósea, si ahora quisiéramos por ejemplo volver a eliminarlo podríamos hacerlo de tres maneras.

rm prueba.txt

rm /home/back/Escritorio/prueba.txt

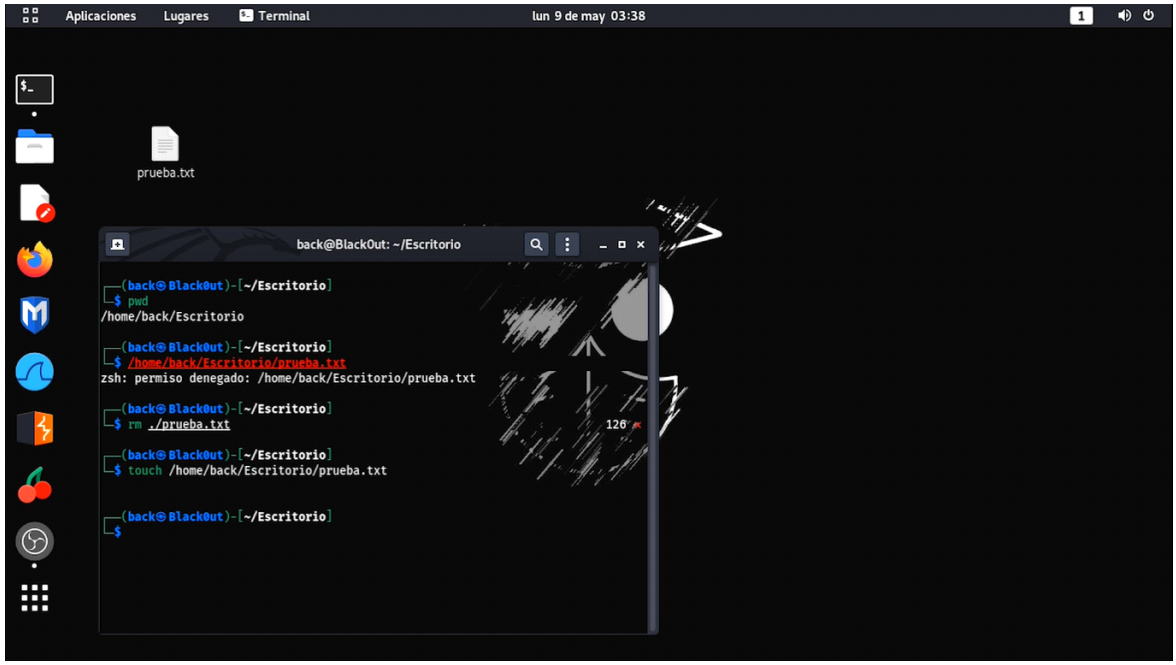
También de la siguiente forma

rm ./prueba.txt



En esta, ese punto, no me lo he inventado porque si, sino que el punto simboliza el directorio actual en el cual nos encontramos. Quizás a primera vista no encuentres la utilidad de usarlo. Pero ten claro que estamos comenzando construir bases. Más adelante en el curso vamos a replicar todo lo que estamos viendo ahora de una manera más avanzada y podrás notar que realmente cada cosa tiene una importancia significativa.

Entonces, solo para que conste, no es un aspecto atípico de ***rm*** lo que estamos viendo, también podríamos crear el mismo fichero utilizando estas rutas

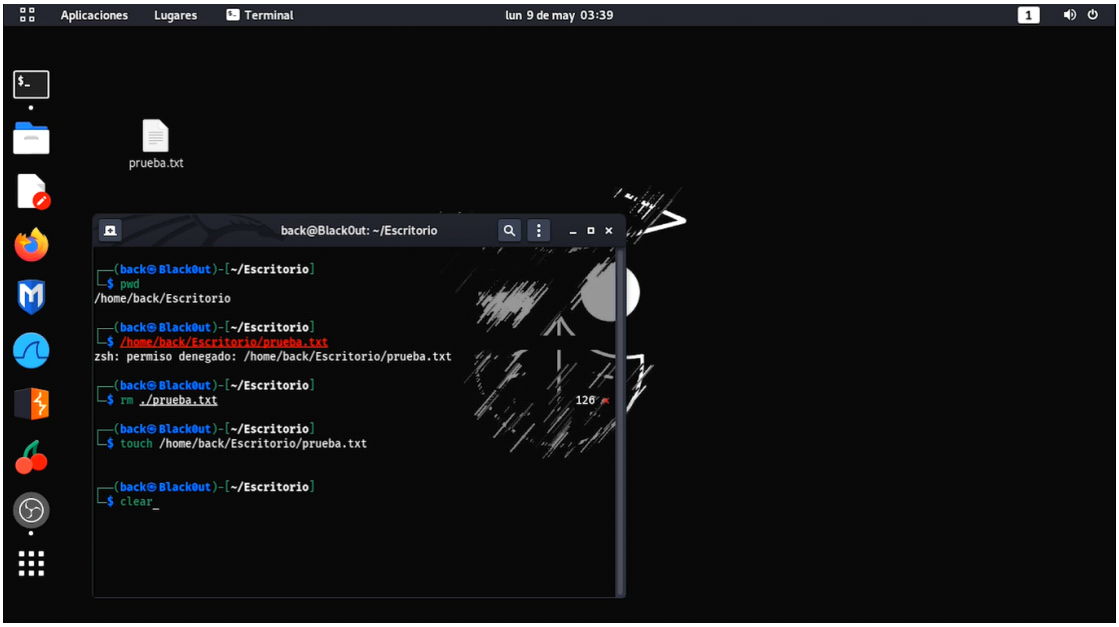


Ósea esto es aplicable a cualquier comando, podéis utilizar siempre la ruta relativa o absoluta.

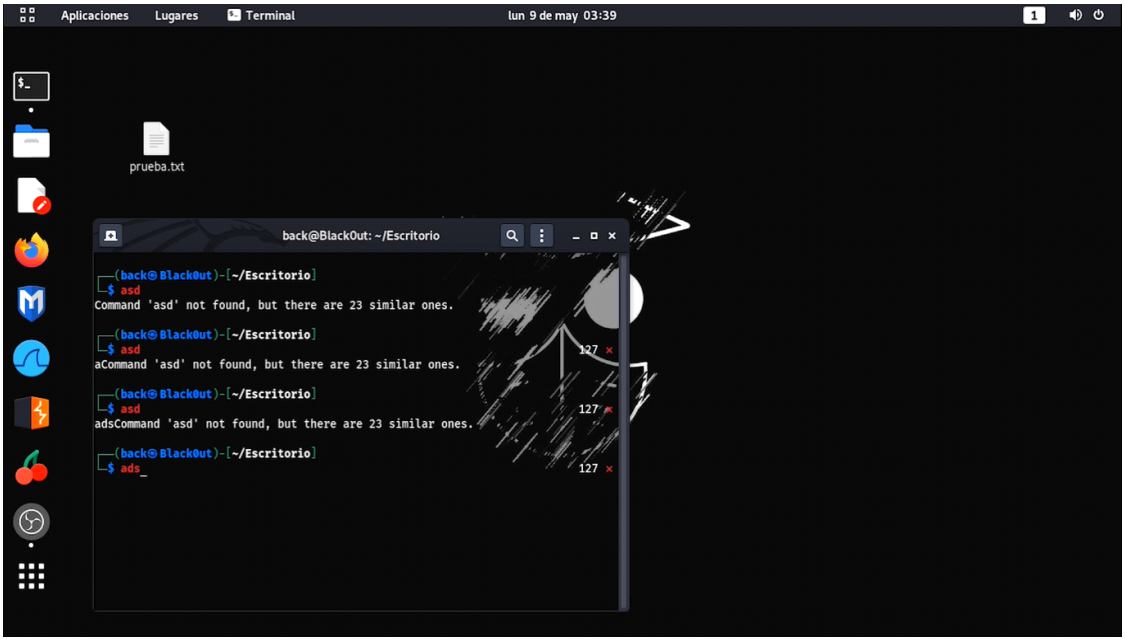
Y aprovechando que estamos. Si os fijáis en el terminal se encuentra ya demasiado código innecesario, si quisiéramos hacerlo tenemos dos variantes.

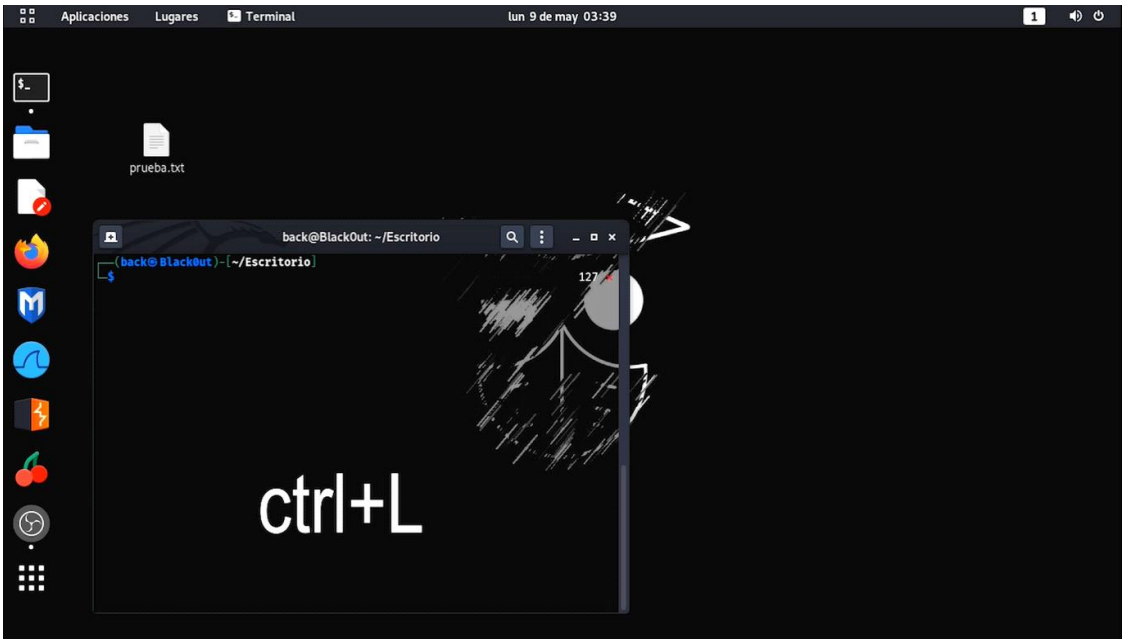
La primera seria con el comando clear que se encargaría de limpiar el terminal

clear



la segunda seria con la combinación de teclas ***ctrl l***





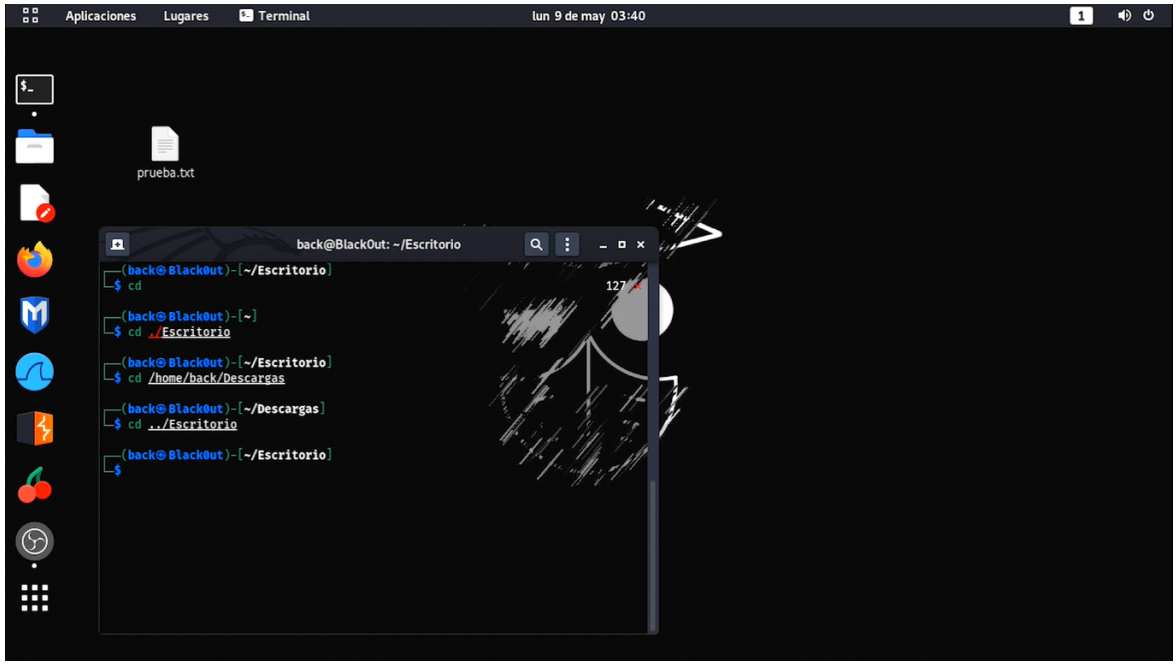
Mas detalles significativos con respecto al movimiento entre directorios, si escribimos ***cd***, sin ningún tipo de parámetro o argumento, nos llevara automáticamente a nuestro directorio personal de usuario.

Si de aquí quisiéramos ir al escritorio podríamos aplicar el mismo concepto previamente visto, podríamos hacerlo con ***./Escritorio*** o meramente omitiendo él **.** y la **/** ya que Linux entiende que queremos movernos de nuestro directorio actual al escritorio.

Si desde el escritorio quisiéramos movernos directamente a otro que se encuentre en el directorio personal de usuario, por ejemplo a Descargas, podemos hacerlo también desde la ruta absoluta que sería la siguiente: ***cd /home/back/Descargas***

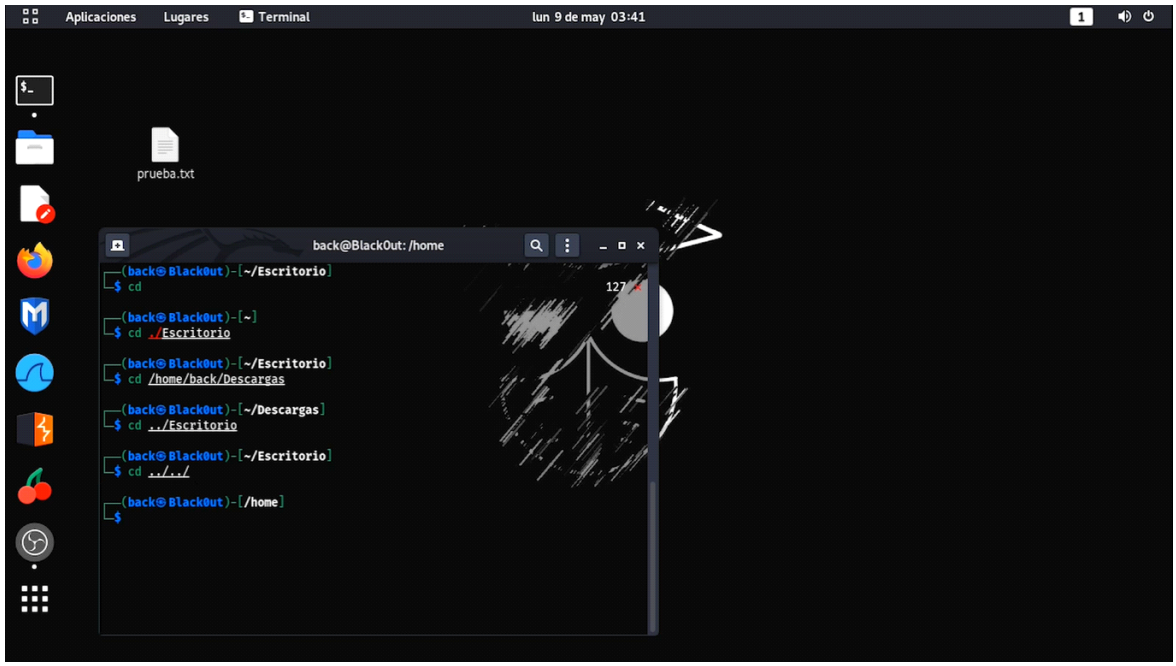
Donde back seria el nombre del directorio personal de vuestro usuario por supuesto.

Y además en el caso de que quisiéramos volver por ejemplo al ***/Escritorio*** también podríamos hacerlo indicando que queremos movernos un directorio atrás, que seria a nuestro directorio personal de usuario y posteriormente a ***Escritorio***, ósea quedaría conformado como ***cd ../Escritorio***



Con los dos puntos recuerden que nos movemos al directorio padre, ósea, del cual cuelga este directorio actual seria el directorio padre, como comúnmente se le llama en Linux

También deberían saber que este tipo de movimientos pueden hacerlos cuantas veces quieran, en el sentido de que podríamos movernos, no solo al directorio padre, sino al padre del padre, meramente con cd **../../**

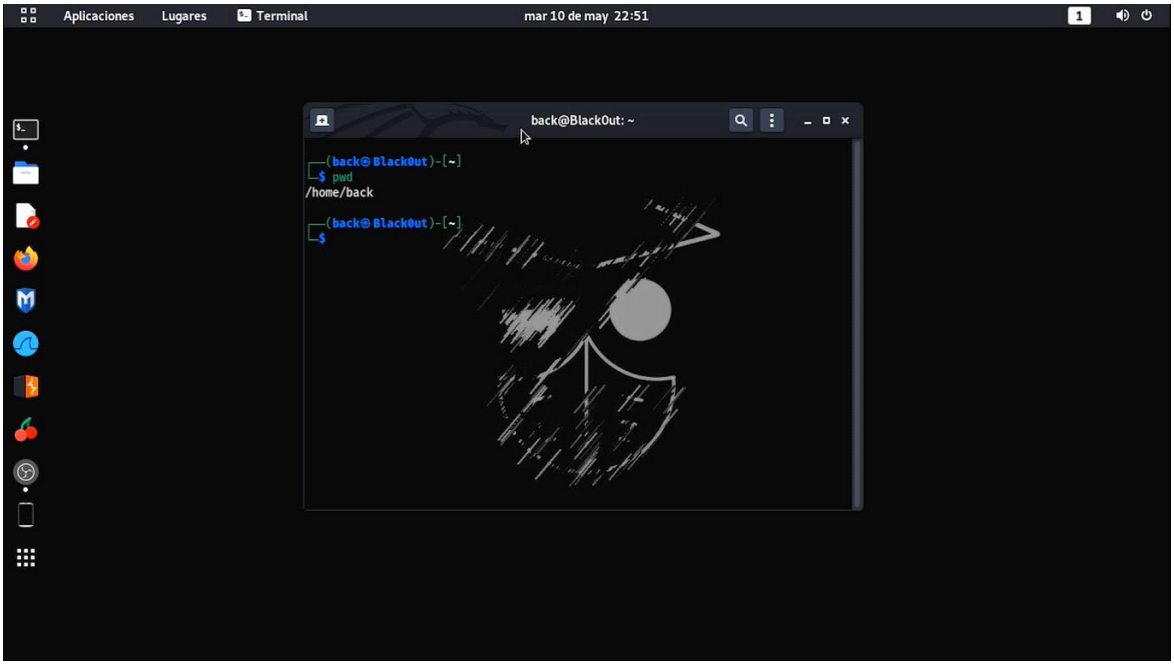


Y en un principio tampoco habrá ningún problema.

Crear y borrar directorios desde diversas rutas

En el capítulo pasado vimos entre otras cosas como podríamos crear ficheros, en este para perfeccionar los conocimientos sobre las rutas y para aprender un nuevo concepto veremos ejemplos similares pero con directorios

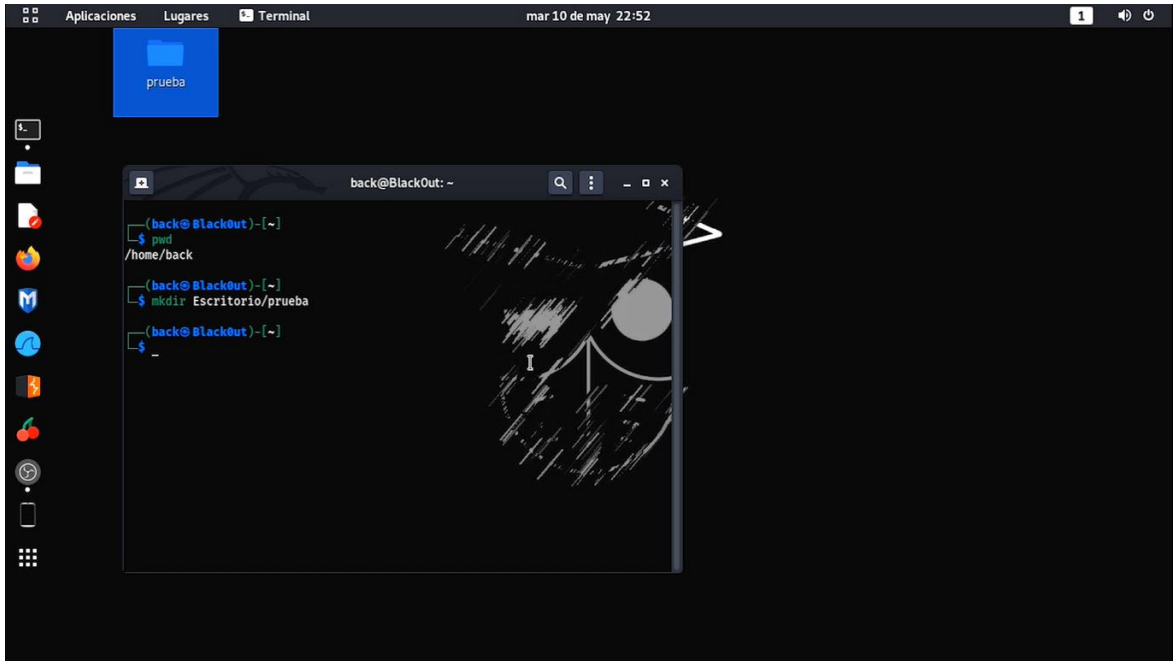
Primeramente fijaos que me encuentro en mi directorio personal de usuario. Recordad que esto lo sabemos porque el mismo prompt nos lo dice.



¿Qué pasa si desde aquí quisiéramos crear un directorio en el escritorio?
Quiero decir, no movernos al escritorio sino hacerlo desde el directorio actual en el que nos encontramos.

Pues para ello tendríamos que utilizar el comando ***mkdir***, esto significa ***make directory***, dándole como parámetro el nombre que queremos asignar a este nuevo directorio. Y en este caso en particular la ruta del escritorio, que es a fin de cuentas donde queremos crear esta carpeta. Esto lo haríamos de la siguiente forma:

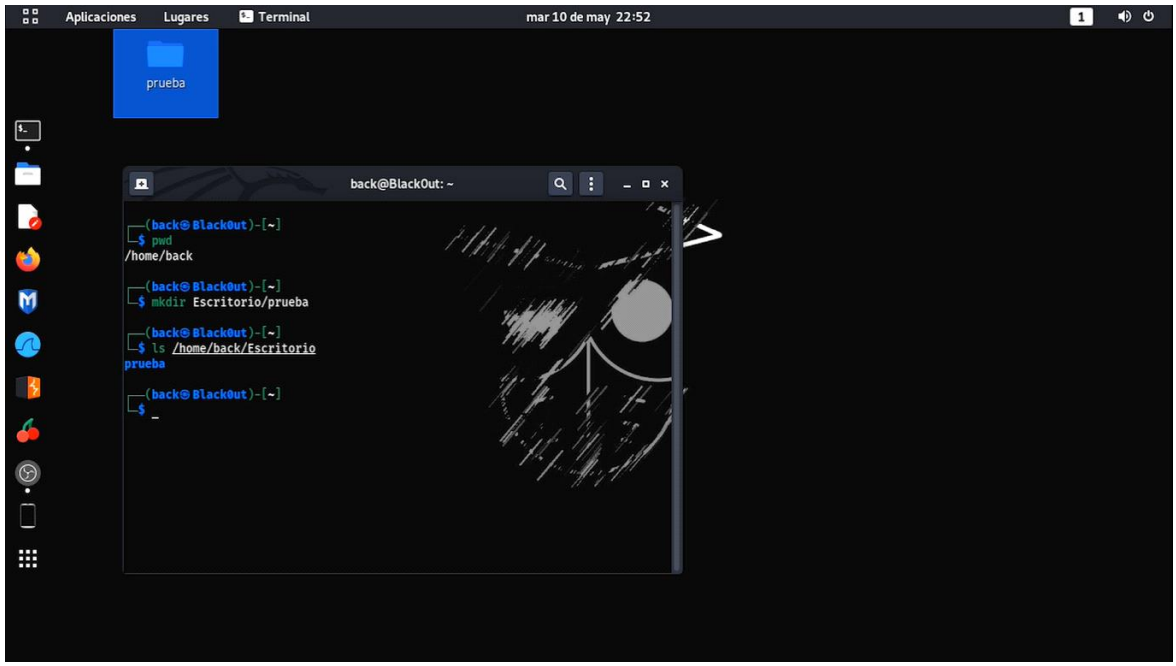
mkdir Escritorio/prueba



Y fijaos que nos crea el directorio en el escritorio. En este caso podemos verlo, pero si estuviera en un directorio diferente ¿cómo podríamos verificar que en efecto nuestro directorio se ha creado?

Simple, podríamos utilizar ***ls*** para listarlo, en este caso en concreto lo hare desde la raíz, ósea utilizando la ruta absoluta

ls /home/back/Escritorio/



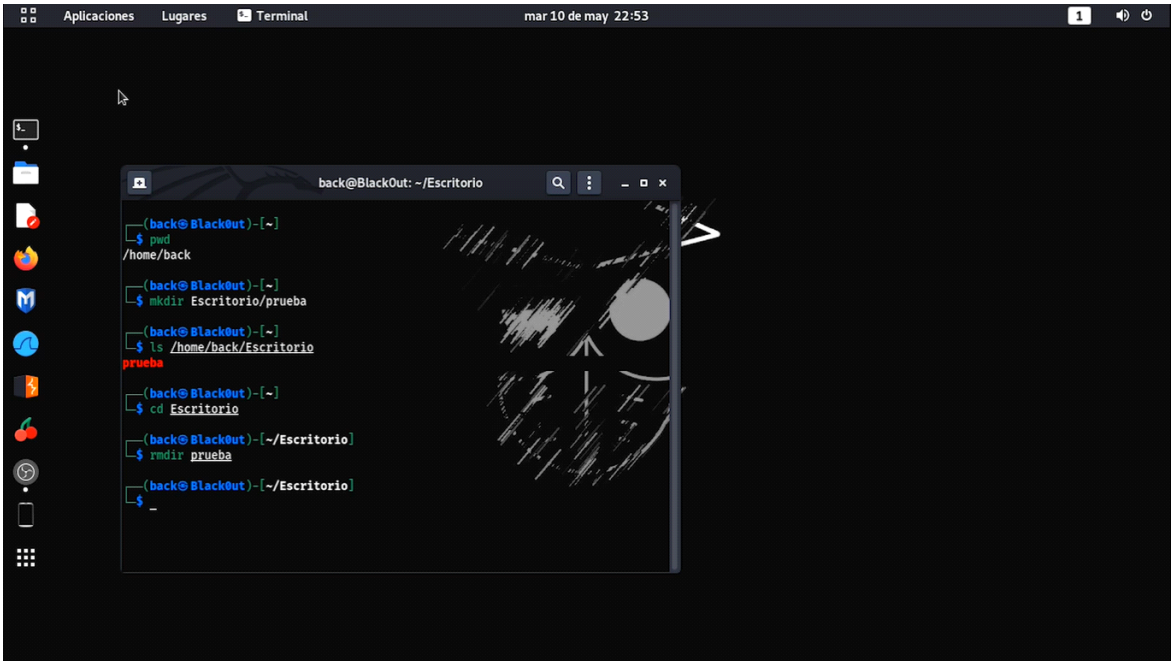
Pero bueno, sabemos cómo crear y eliminar ficheros, al igual que movernos entre

directorios, solo falta saber cómo borrar estos directorios. Puesto que si eres nuevo probablemente pudieras pensar que podrías hacerlo con el comando ***rm***, de hecho en teoría si. Pero en realidad, existe un comando particular para eliminar directorios que se encuentran vacios, que es el comando ***rmdir*** que viene de ***remove directory***.

Podríamos movernos a esa ruta con ***cd /Escritorio***

y directamente eliminarlo

rmdir ./prueba

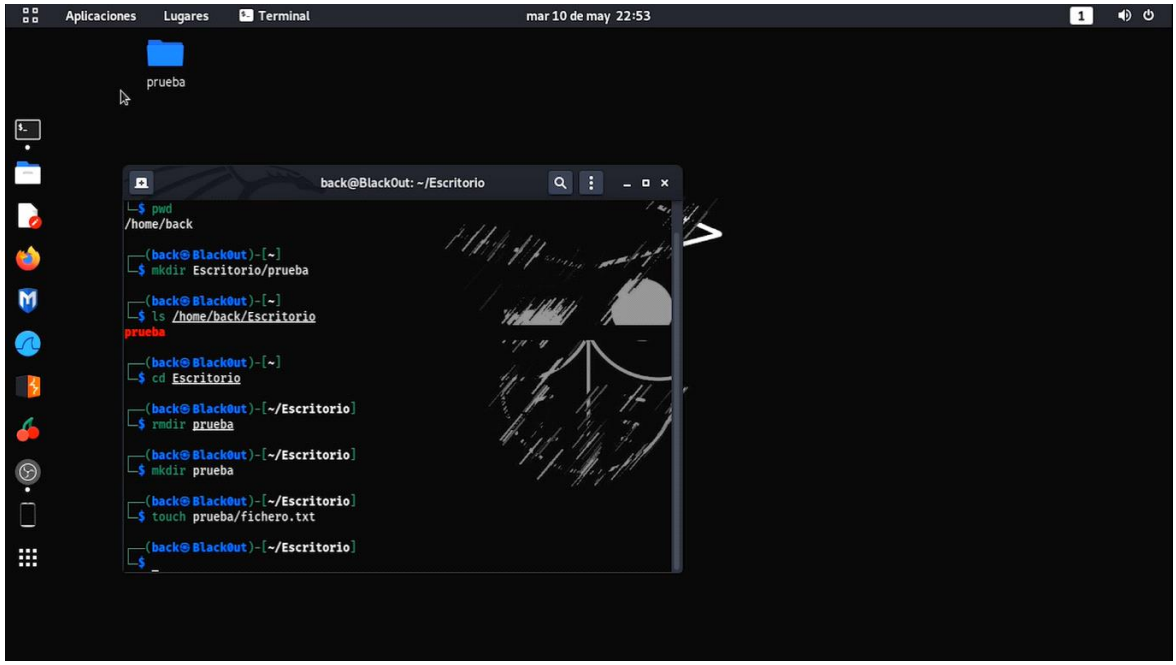


Pero recordando las palabras que os dije antes, únicamente podemos borrar directorios que se encuentren vacios.

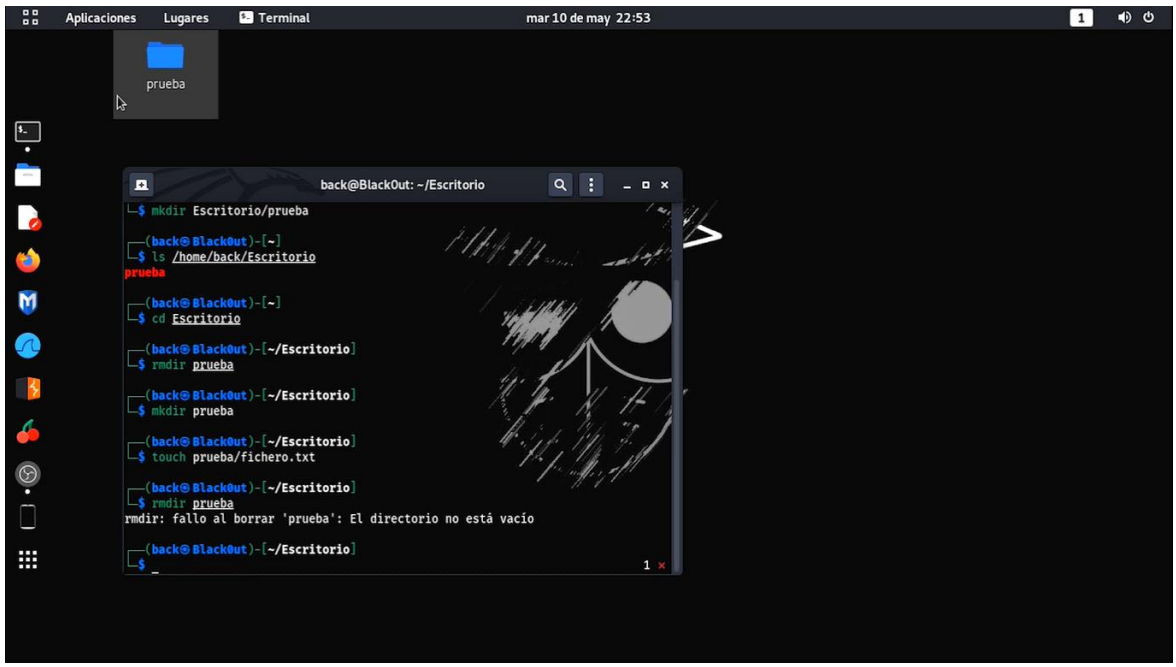
Si creásemos otro directorio y le añadimos un fichero

mkdir prueba

touch prueba/fichero.txt



Y posteriormente intentamos eliminarlo con ***rmdir prueba***

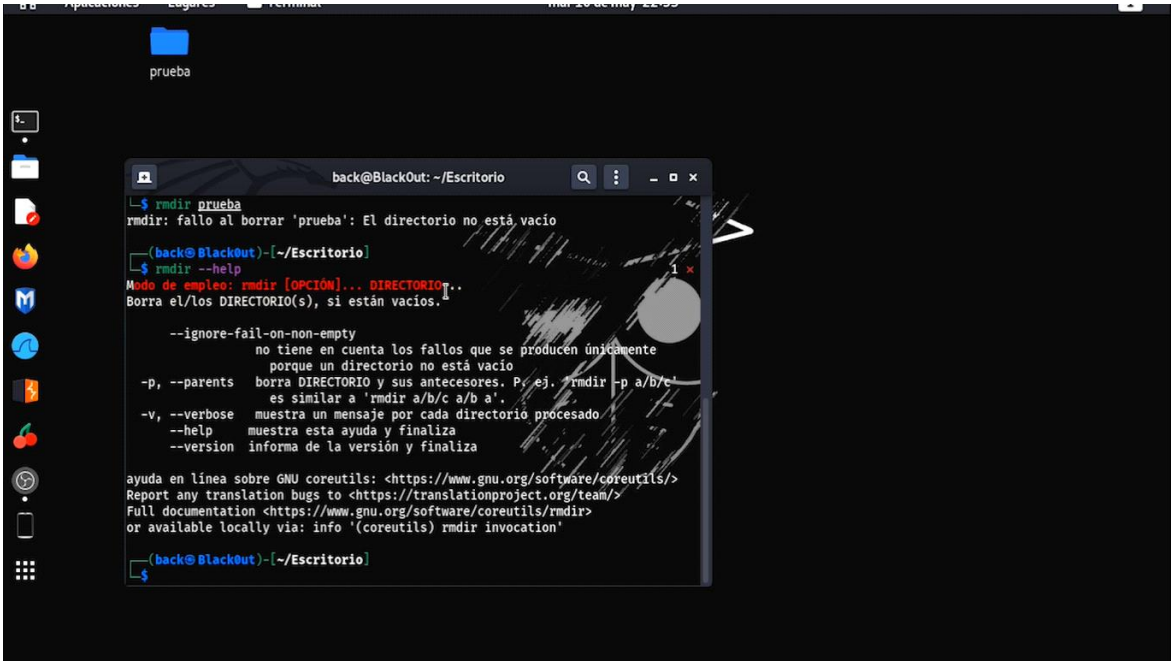


Podremos comprobar que no podemos

Y quiero aprovechar esto para darles la lección más simple pero a la vez importante en todo este curso. Que sería no limitarse a ver lo que te muestran o lo que tienes en frente, sino a ir más allá.

Cada herramienta en Linux tiene algo que se llama ayuda. A la cual podemos acceder con el nombre de cualquier comando seguido de ***--help o -h***

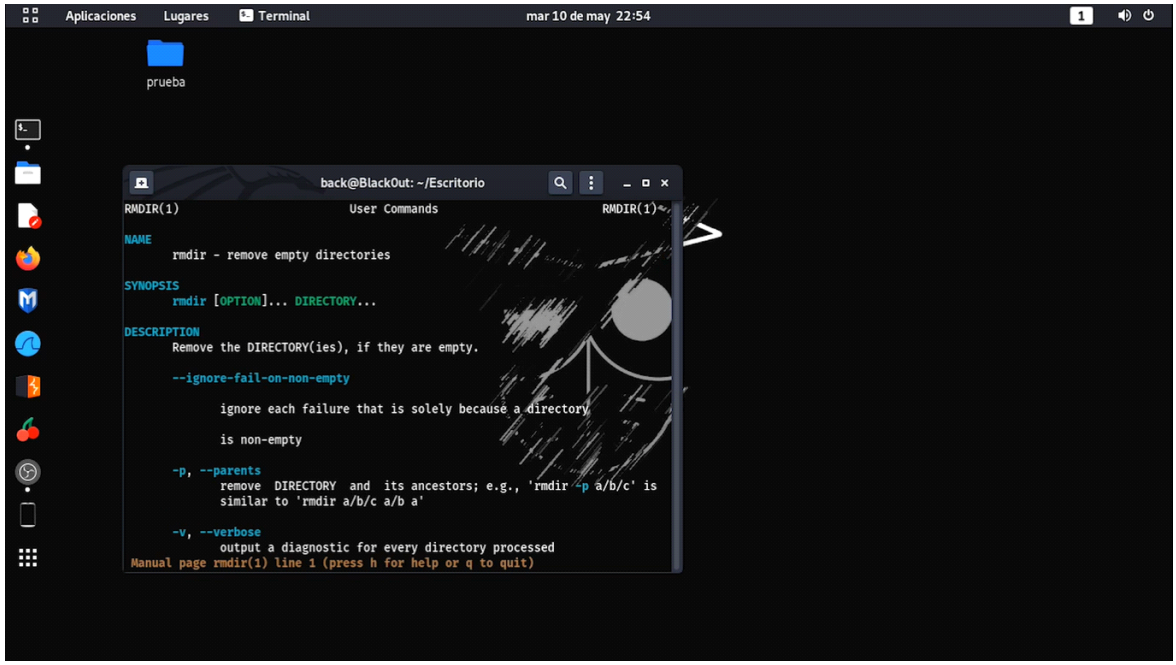
por ejemplo si vamos a ***rmdir --help***



Podemos ver que claramente nos explica el modo de uso y los parámetros de la misma. Además la gran mayoría de los comandos populares y/o importantes en Linux tienen algo que es el manual, y se puede acceder al manual de cada comando con el comando ***man***

En este caso

man mkdir

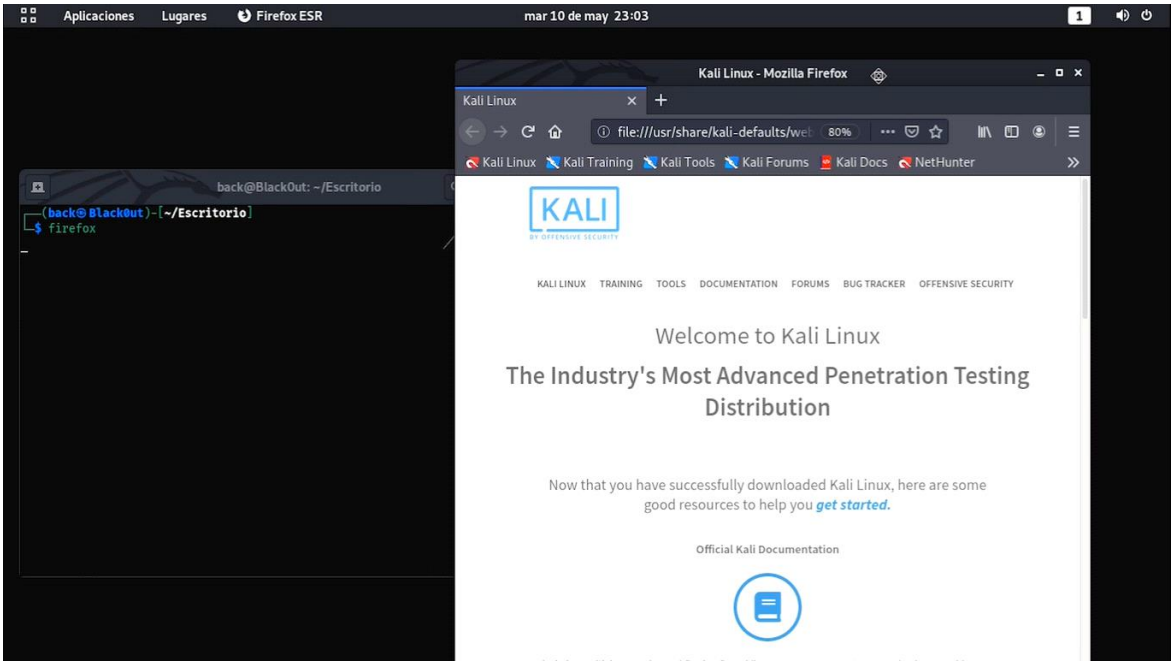

Aplicaciones
Lugares
Terminal
mar 10 de may 22:54
1
prueba
back@BlackOut: ~/Escritorio
RMDIR(1) User Commands RMDIR(1)
NAME
rmdir - remove empty directories
SYNOPSIS
rmdir [OPTION]... DIRECTORY...
DESCRIPTION
Remove the DIRECTORY(ies), if they are empty.
--ignore-fail-on-non-empty
ignore each failure that is solely because a directory
is non-empty
-p, --parents
remove DIRECTORY and its ancestors; e.g., 'rmdir -p a/b/c' is
similar to 'rmdir a/b/c a/b a'
-v, --verbose
output a diagnostic for every directory processed
Manual page rmdir(1) line 1 (press h for help or q to quit)

Que son realmente los comandos

Si quiero en este capítulo que quede claro el concepto de ***comando***, esto porque veo que está bastante mal explicado o ni siquiera explicado, por lo que mucha gente no entiende verdaderamente este concepto. Y si vamos a estar escribiendo comandos deberíamos tener idea de que significan.

La gente a veces piensa que los comandos son como que algo a muy bajo nivel he interpretado por el ordenador de cierta forma rara o algo por el estilo. Pero no es así, de hecho es mucho más simple. ***Los comandos son programas***. Ósea es un programa que está escrito en algún lenguaje de programación, por ejemplo bash y básicamente cualquiera puede escribir sus propios comandos, que de hecho en dependencia de que comando quieras crear sea fácil o difícil. Ósea ***ls, cp, mv, touch.*** Todos son programas y de hecho nosotros mismo más adelante en el curso veremos cómo podremos escribir nuestros propios comandos.

Y para que vean mejor ejemplarizado esto, desde el mismo terminal del mismo modo que hemos utilizado comandos por el terminal que no utilizan interface gráfica, también podríamos utilizar comandos que si que la tengan, por ejemplo, como no "***firefox***"



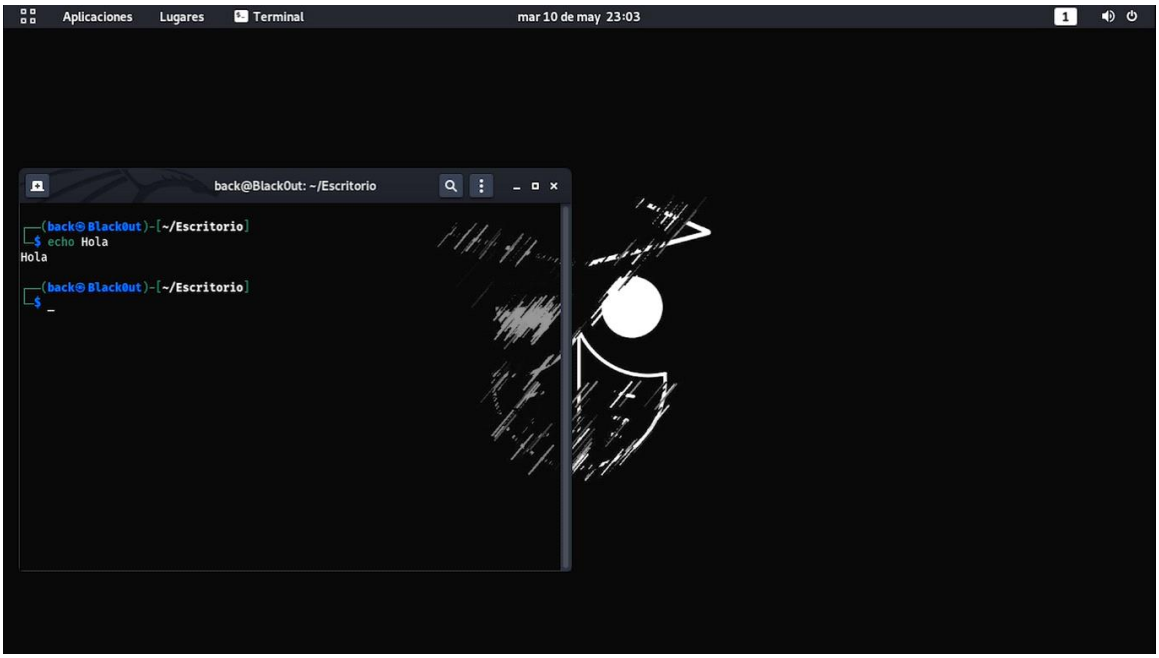
Como vez, también podemos utilizarlo desde el terminal.

Y podemos utilizar comandos en este emulador de terminal porque funciona como un intérprete de comandos, en este caso un intérprete de bash

Comandos básicos

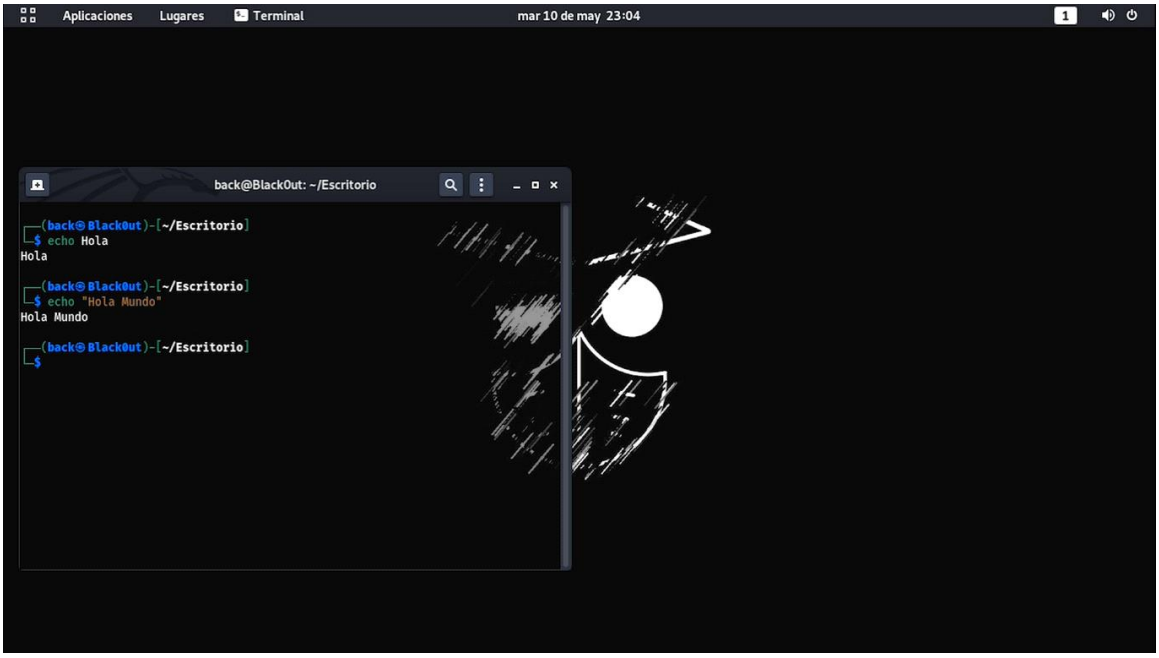
Vamos a ver en este capítulo más comandos básicos que podríamos utilizar desde el terminal

El primero de ellos sería el comando ***echo***, con el cual podemos imprimir texto desde el terminal escribiendo el comando seguido del texto, por ejemplo: ***echo Hola***



Y fijaos que lo que hace es mostrarnos por debajo ese texto, ósea nos imprime por la pantalla lo que le estamos pasando

Si quisiéramos hacerlo con una oración o algo por el estilo, recordad que el capitulo pasado os dije que en *Linux* deberíamos de englobarlo entre *comillas, como podria ser el siguiente caso:* ***echo 'Hola Mundo'***



Podéis utilizar comillas simples o dobles, es cuestión de la ocasión o en este caso mera preferencia.

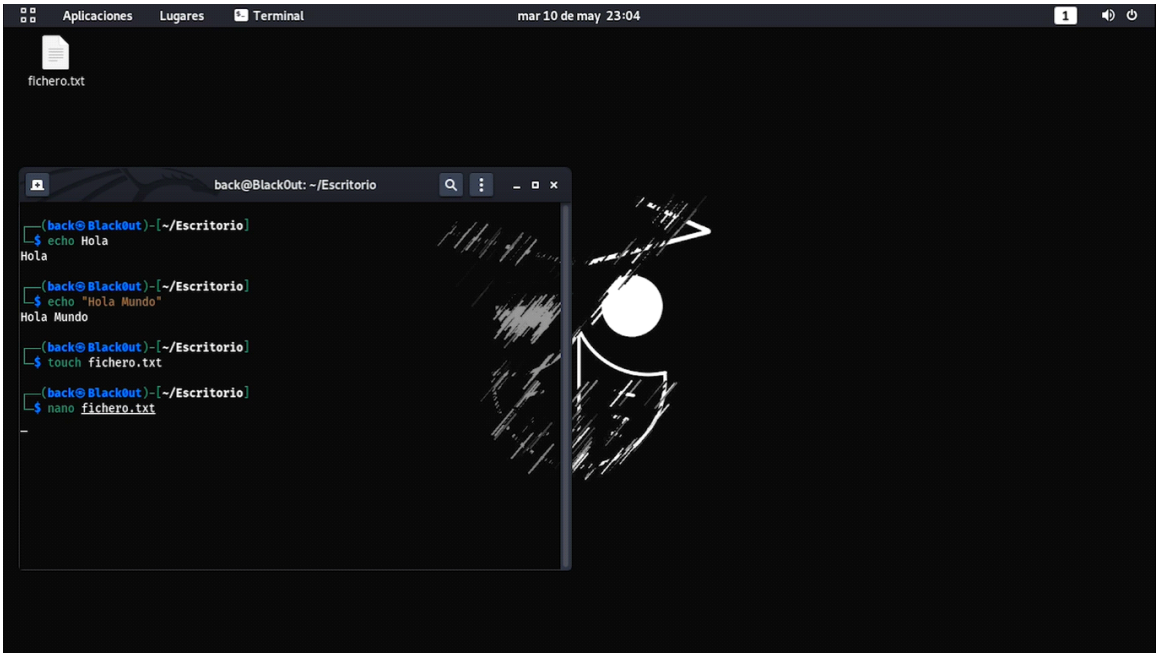
Que por cierto quizás piensen que este comando es un tanto inútil, porque si, después de todo tienes lo mque pasarle ismo que el te imprima. Pero recuerda que estamos fomentando bases, este comando en este ejemplo podrá resultar inútil. Pero en el futuro veremos que tiene inmensidad de usos prácticos.

Otro de los comandos que deberíamos aprender a utilizar y que viene pre instalado por defecto en la mayoría de los sistemas *Linux* es el comando ***nano***

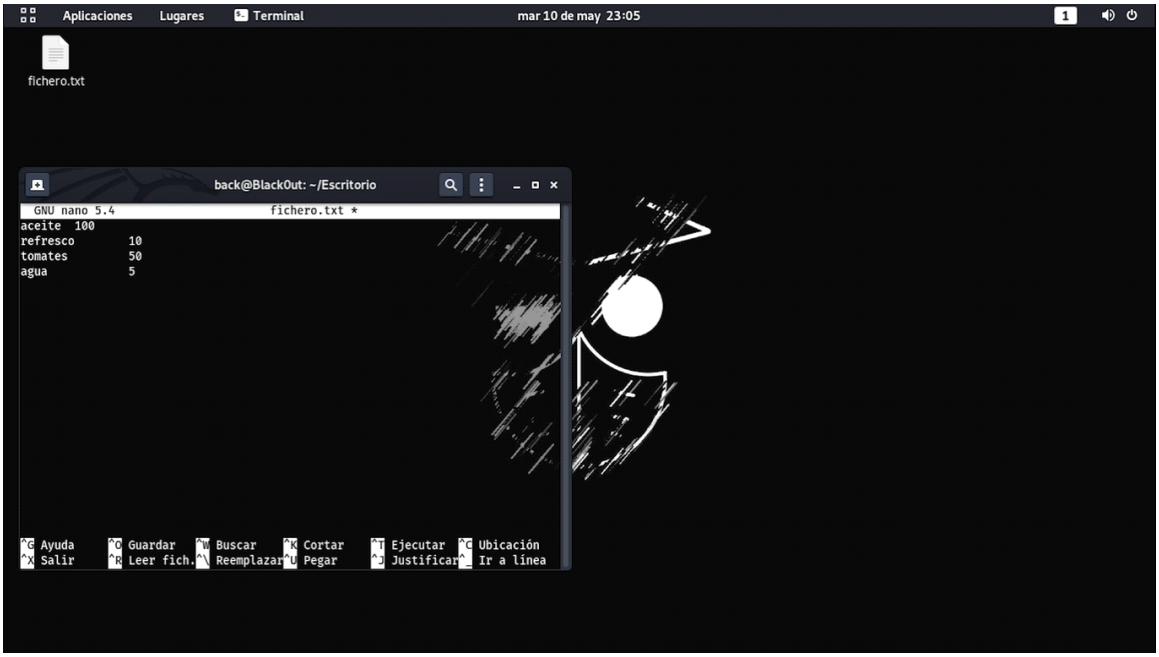
Este es un editor de texto que podemos utilizar desde el mismo terminal, y su uso es bastante simple.

Por ejemplo vamos a crear un fichero, el cual podríamos editar con nano escribiendo el comando seguido del nombre del fichero, es decir:

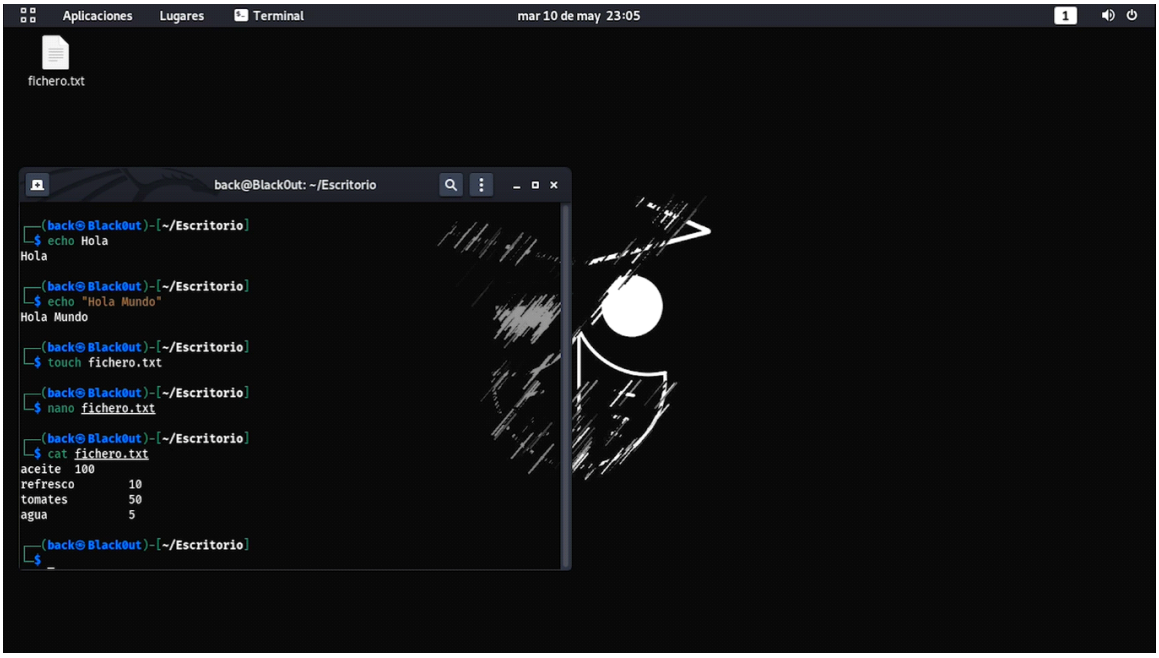
nano fichero.txt



Lo cual nos desplegaria el editor nano, en el cual podriamos editar el contenido del archivo, en este caso por ejemplo vamos a escribir una lista sencilla para mostraros el próximo comando.



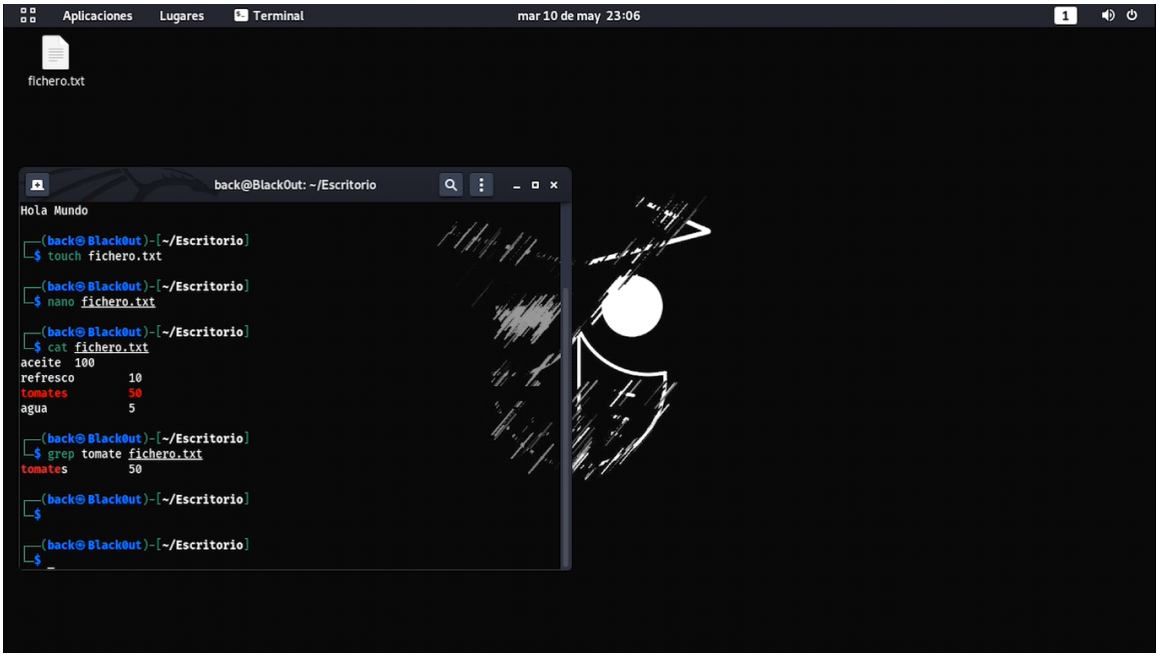
Y vamos a guardar esto que hemos escrito utilizando la combinación de teclas ***ctrl x*** nos preguntara si estamos seguros de que queremos guardar, escribimos ***s de si o y de yes*** en dependencia del idioma de tu sistema y habríamos guardado el contenido.



Ahora tenemos una lista de lo que podría ser el menú de un mercado, imagina que vamos al mismo a buscar tomates por ejemplo. Vamos a automatizar la labor que debería de cumplir nuestro cerebro para encontrar los tomates en el menú. Ósea, vamos a ver el comando ***grep***, *que nos permitirá filtrar por diversos patrones en un fichero o una cadena de texto.*

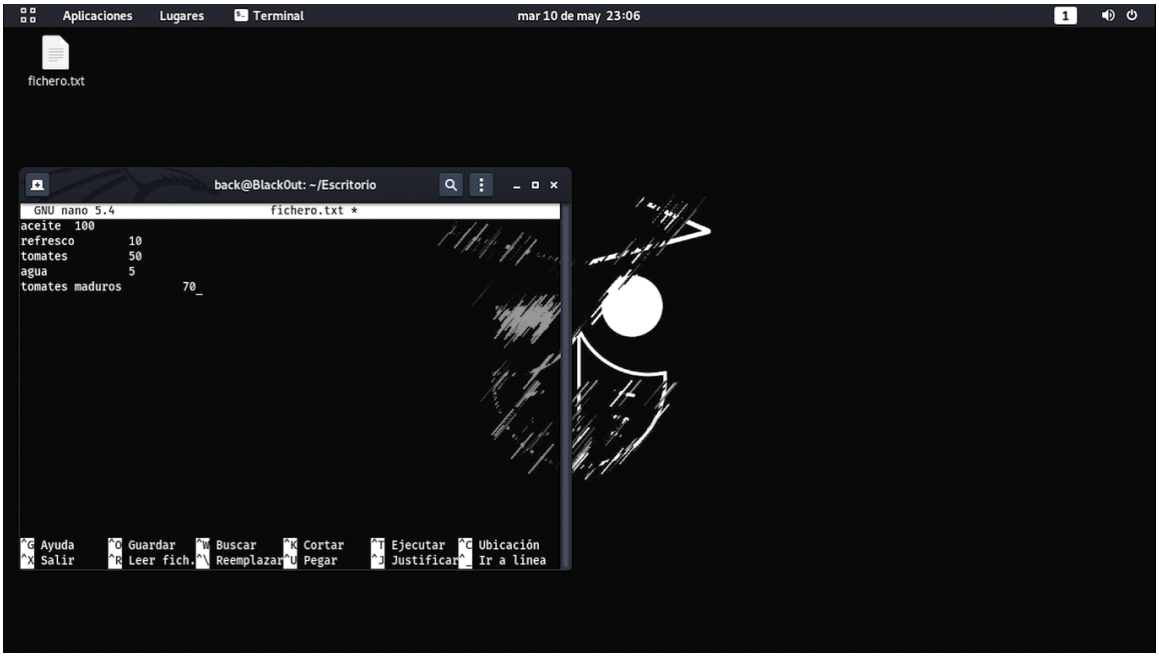
Para esto utilizaremos el comando ***grep*** seguido de lo que queremos buscar en este caso, y como es *tomates* pues *tomates*, esto seguido del fichero en el cual queremos filtrar por dicho contenido, es decir:

grep tomate fichero.txt

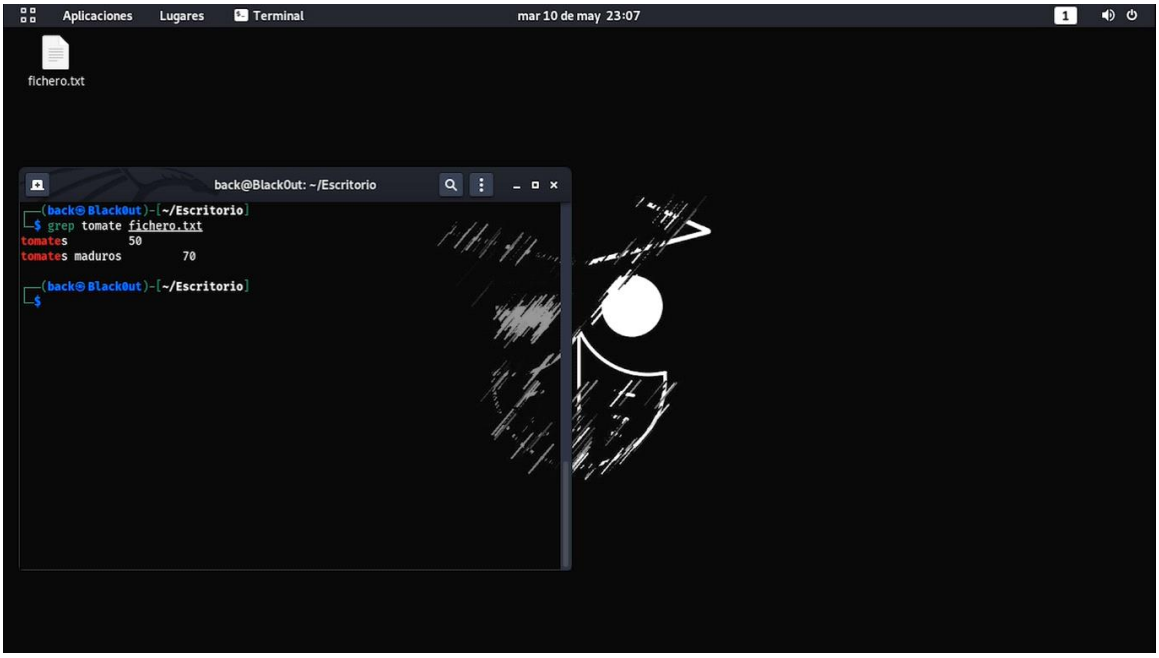


Y fijaos que nos salen ahora únicamente los tomates.

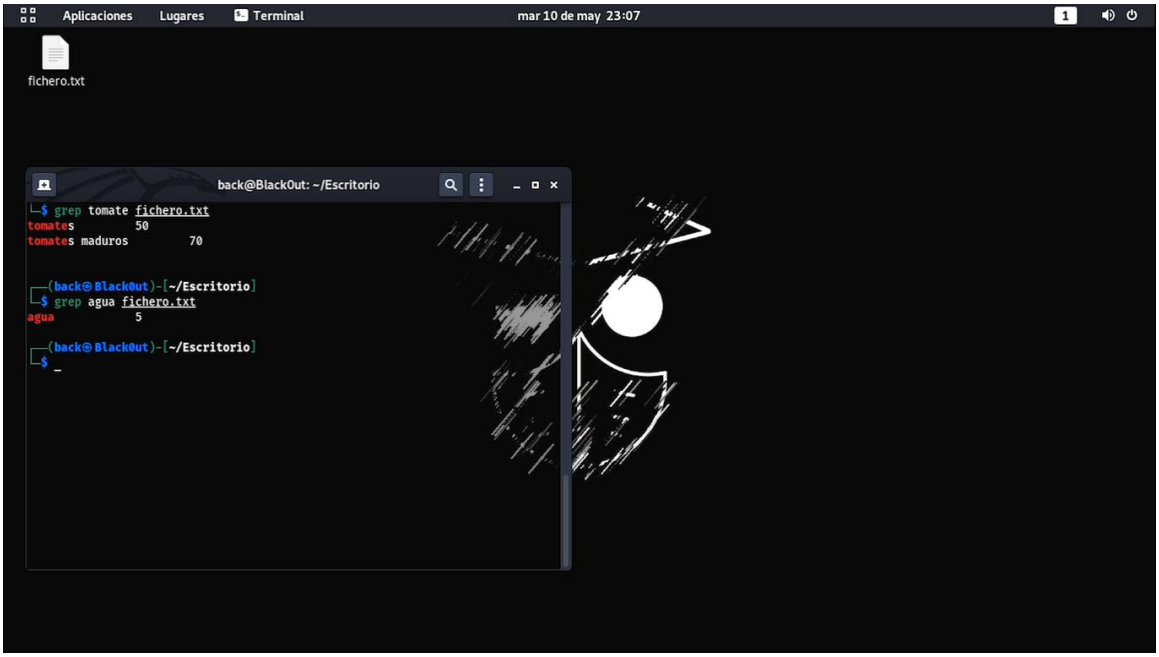
vamos a volver a editar el fichero y agregar, no se... Tomates maduros.



Si volvemos a buscar en el fichero por tomates ahora nos saldrian ambos y nos resaltara el lugar en el que se encuentra lo que le indicamos que queríamos filtrar, que en este caso es *tomate.*



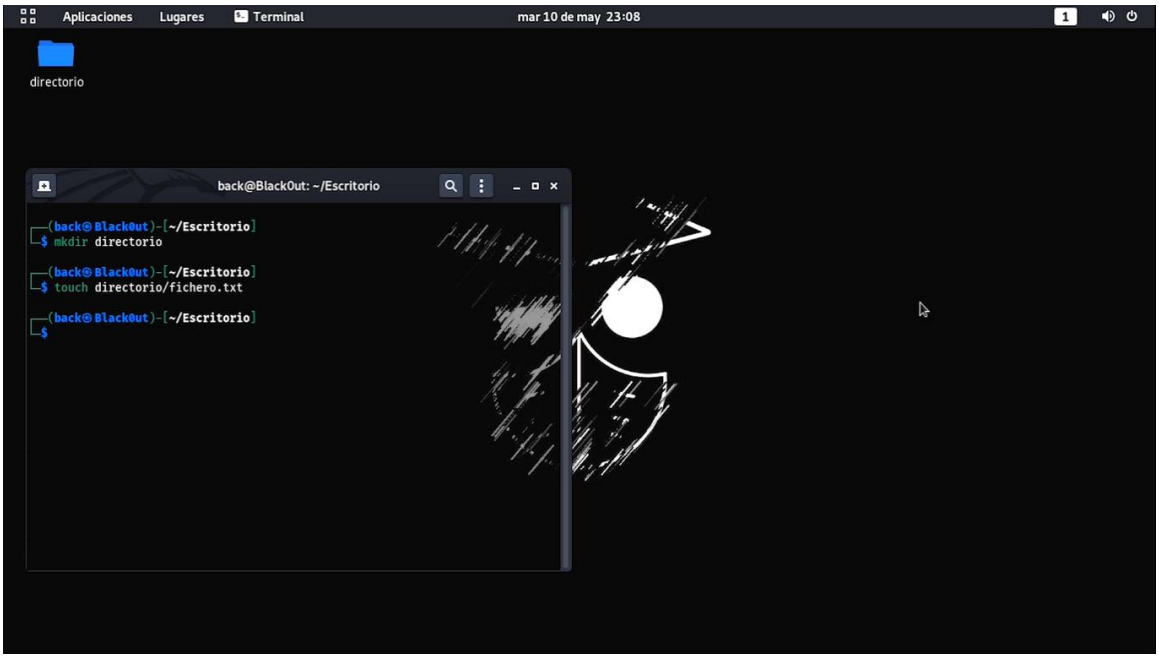
Claro podríamos utilizarlo para filtrar otras cosas, por ejemplo agua y nos filtraria por la palabra agua sin ningun problema



Opciones de los comandos

En este capítulo veremos cómo utilizar parámetros en los comandos, para ello se me ocurre un ejemplo excelente a utilizar para expandir lo que hemos visto hasta ahora.

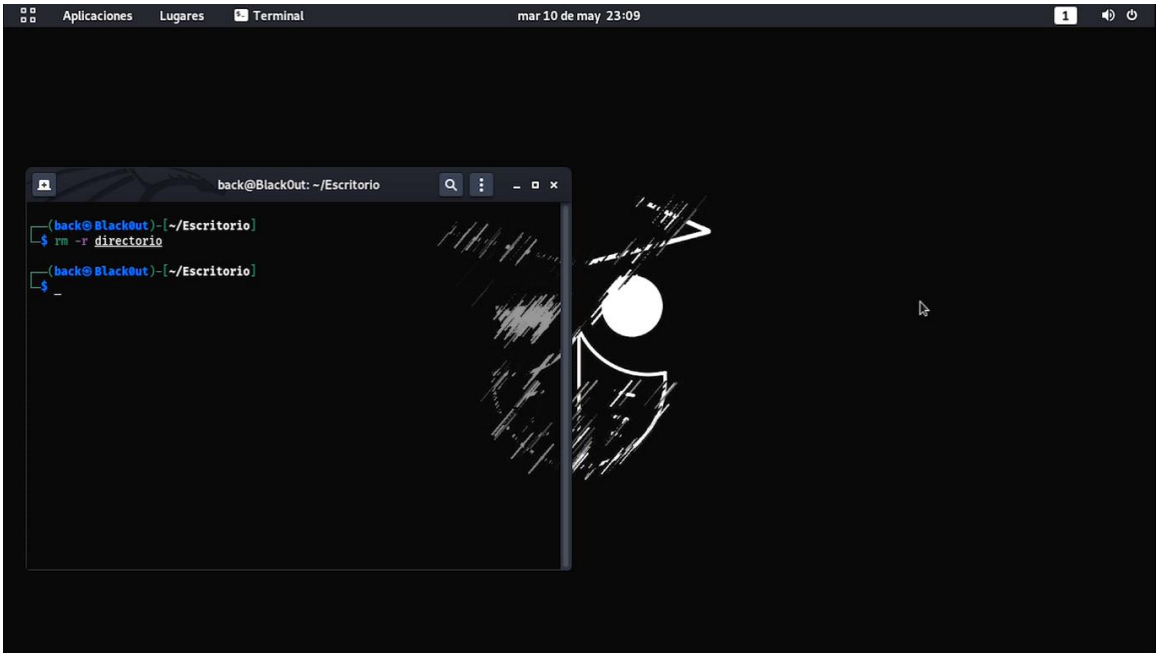
Por ejemplo vamos a crear un directorio y poner dentro del mismo un fichero:



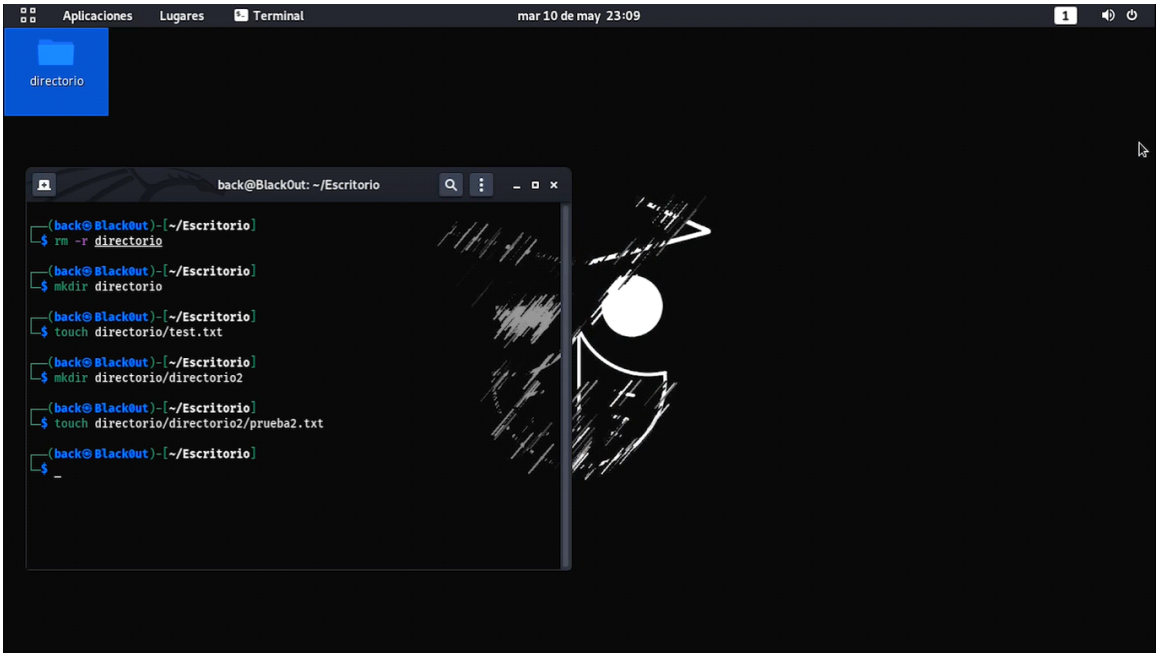
Entonces, sabemos cómo crear y borrar directorios pero no hemos borrado hasta ahora un directorio que tenga dentro un fichero.

Si intentamos borrarlo con el comando ***rmdir*** nos dará un fallo, ya que ***rmdir*** se utiliza para eliminar directorios que se encuentren vacios. Entonces vamos a utilizar por primera vez la opción de un comando, en este caso vamos a utilizar ***rm*** pero con la opción ***recursiva.***

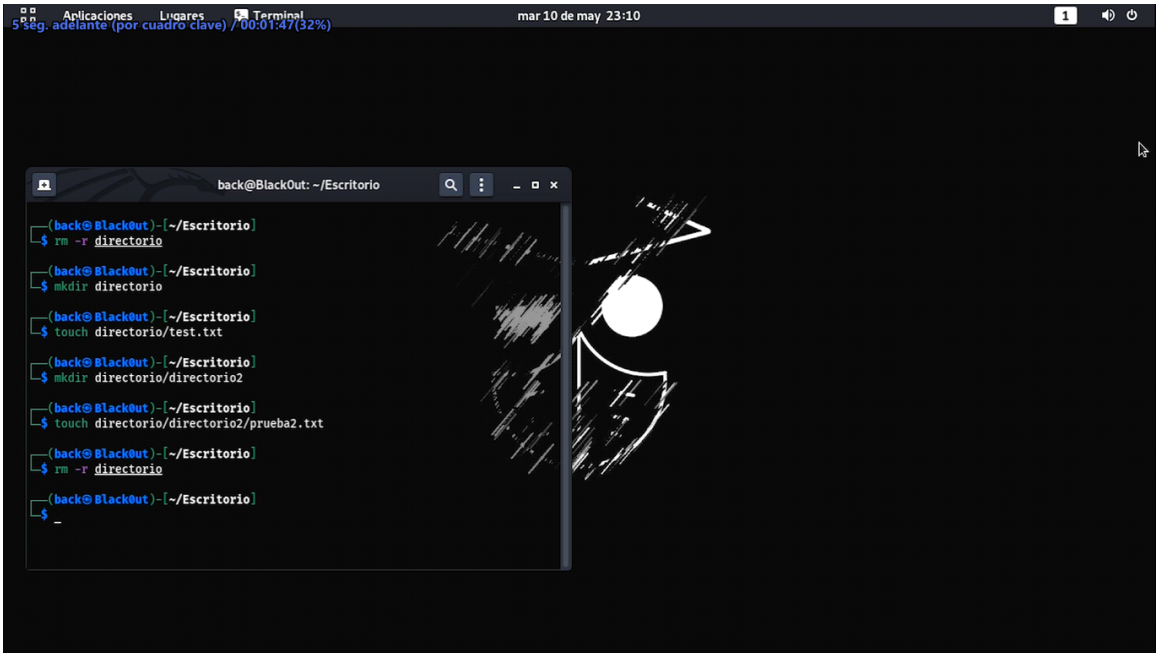
rm -r directorio



Esto lo que hará será eliminar el directorio junto con todo su contenido, por ejemplo creemos otro directorio y mas ficheros dentro de ese mismo directorio



Si utilizamos el comando ***rm -r*** sobre el directorio eliminara el mismo junto con todo su contenido. Cosas a tomar en cuenta, eliminar ese contenido no lo enviara a la papelera de reciclaje o algo por el estilo, si lo eliminas, adiós. Por lo que deberías utilizar el comando ***rm*** con precaución.



Y si os fijáis, a desaparecido totalmente junto con todo su contenido.
Podéis ver más información de este parámetro en el menú de ayuda de la misma herramienta ***rm con rm --help*** o en el manual de la misma, con ***man rm***

Sobre todo tomen una nota mental de que hay que tener cuidado con el comando ***rm -r***, puesto que si eliminan alguna cosa que no quieren o algo fundamental en su sistema podría ser que se encuentren ante un grave problema. Ya que ***Linux es un sistema que proporciona bastantes libertades y si le pides borrar todo el sistema no es que te saltara una alerta de que no puedas hacerlo, Linux meramente ejecutara tu orden.***

Otra cosa importante, existen varias maneras de especificar las opciones de cada herramienta. Tantas como haya definido su creador, por lo cual el ***-r*** nos sirve como una abreviación, pero también podríamos utilizar la opción ***recursiva*** para indicar lo mismo y funcionaria de igual manera, por ejemplo volvamos a crear un directorio y poner varias cosas dentro

mkdir dir

mkdir dir/dir2

touch dir/f.txt

touch dir/dir2/f2.txt

Podríamos eliminarlo también, como muestra la misma ayuda de rm con ***--recursive***

Y aprovechando la ocasión, os explico que no tienen porque precisamente utilizar una única opción, sino que podéis pasar varias, por ejemplo esta la opción ***-i*** o ***--interactive***

Esto lo que hará será preguntarnos si queremos borrar cada cosa que se encuentre dentro de ese directorio

Podemos ir dándole a si y a no especificándole lo que queremos eliminar.

Pero no es la única forma de utilizar las opciones, en ***Linux*** existe otra manera de abreviar y es que si volvemos a crear el directorio y ponemos dentro un fichero:

lo que podríamos hacer con ***rm -r -i directorio*** también podríamos resumirlo a ***rm -ri directorio.***

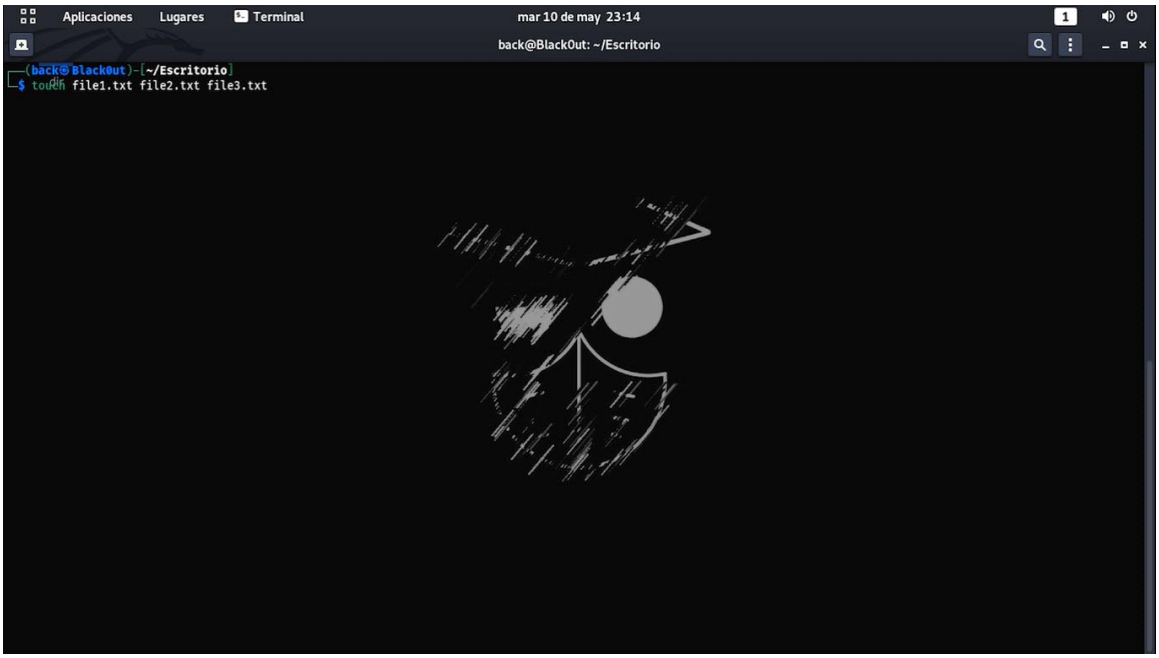
Esto funcionaria de la misma manera, nos ejecutaría las dos opciones, en este caso ***recursive*** e ***interactive***

Otro ejemplo interesante es que por ejemplo digamos que queremos crear varios ficheros dentro del directorio

Podríamos crear dentro de dir varios ficheros como hemos visto hasta ahora meramente con ***touch file1.txt***, ***touch file2.txt***, etcetera. Pero vamos a eliminarlos para ver una manera mas efectiva y resumida de hacerlo.

Para resumir este trabajo podríamos hacer lo siguiente:

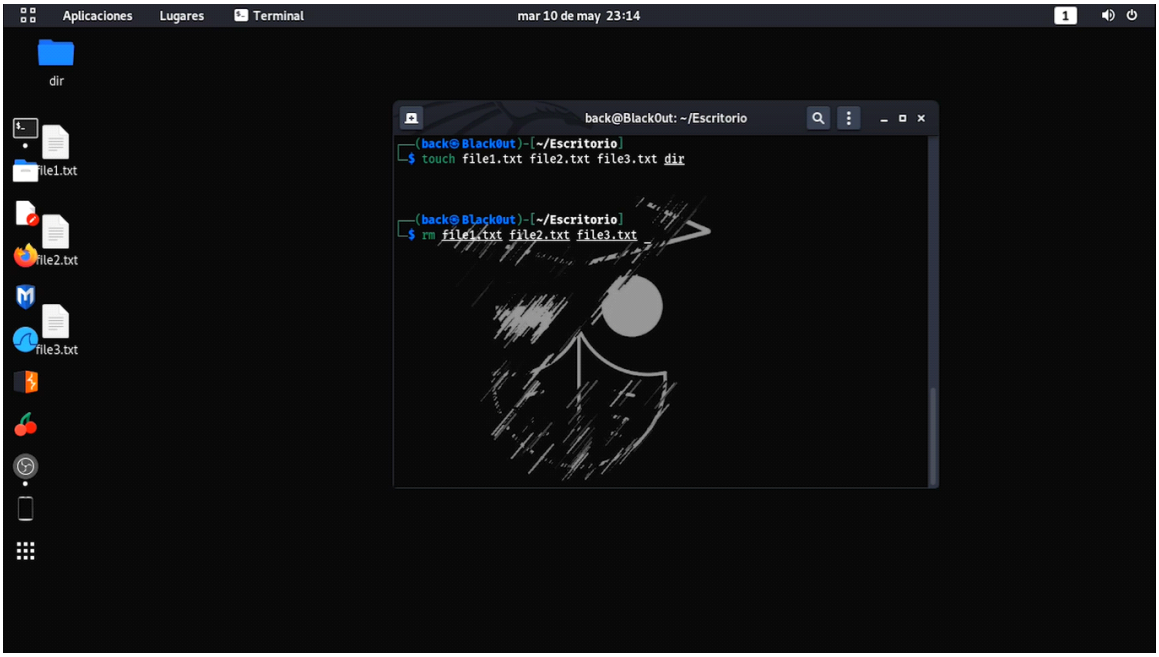
touch file1.txt file2.txt file3.txt



Podríamos crear tantos ficheros como quisiéramos, nuevamente recuerdo que no es únicamente que el comando ***touch*** sirva para esto, también podríamos crear directorios de la misma manera, borrar, o básicamente hacer cualquier cosa por el estilo en ***Linux***.

Lo que quiero decir es que esta no es una manera específica del comando ***touch*** para crear ficheros, sino un esquema de comportamiento propio de las distribuciones de ***GNU/Linux*** o mas en concreto, de ***bash***.

Por ejemplo podríamos borrar estos ficheros del mismo modo:


Aplicaciones
Lugares
Terminal
mar 10 de may 23:14
dir
file1.txt
file2.txt
file3.txt
back@BlackOut: ~/Escritorio
(back㉿BlackOut)-[~/Escritorio]
$ touch file1.txt file2.txt file3.txt dir
(back㉿BlackOut)-[~/Escritorio]
$ rm file1.txt file2.txt file3.txt

Ejercicio 1

Realice las siguientes tareas vistas previamente para que pueda practicar lo antes visto (Es muy importante que puedas resolver cada ejercicio que se encuentre en el curso, pues la madre de la enseñanza en este caso no es la lectura sino la práctica, puedes leer pero te aseguro que leer por leer será meramente leer sin aprender, si realmente quieres dominar todo lo que en el curso se expone debes replicar absolutamente todo, o como mínimo pasar los ejercicios antes de seguir leyendo, si te crees incapaz te recomiendo repasar los puntos en los que puedas haber tenido problemas:

1- Muévase al escritorio de su directorio personal de usuario

2- Cree un directorio con el nombre ejercicio en el directorio actual donde se encuentra (Escritorio)

3- Sin moverse del directorio actual, cree dentro del directorio ejercicio dos directorios con los nombres dir1 y dir2.

4- Ahora puede moverse al directorio ejercicio, le recomiendo hacerlo desde la absoluta del mismo. En caso de que no tenga claro de a que me refiero, puede volver a ver el capítulo de ficheros y rutas.

5- Utilizando únicamente un comando cree dentro de dir1 los ficheros 1.txt, 2.txt y 3.txt

6- Nuevamente sin moverse del directorio actual, borre los ficheros 2.txt y 3.txt

7- Muévase al directorio Padre (Escritorio)

8- Desde el escritorio borre dir2, que sería el directorio que se encuentra vacio dentro de directorio ejercicio.

9- Utilizando un único comando elimine el directorio ejercicio junto con todo su contenido

NOTA:

Recuerda que si tienes cualquier duda, problema o sugerencia puedes comentármela, mi bandeja de mensajes en Twitter se encuentra abierta para que lo necesites, es la red social que más frecuento, con lo cual ahí tiendo a responder de manera más rápida:

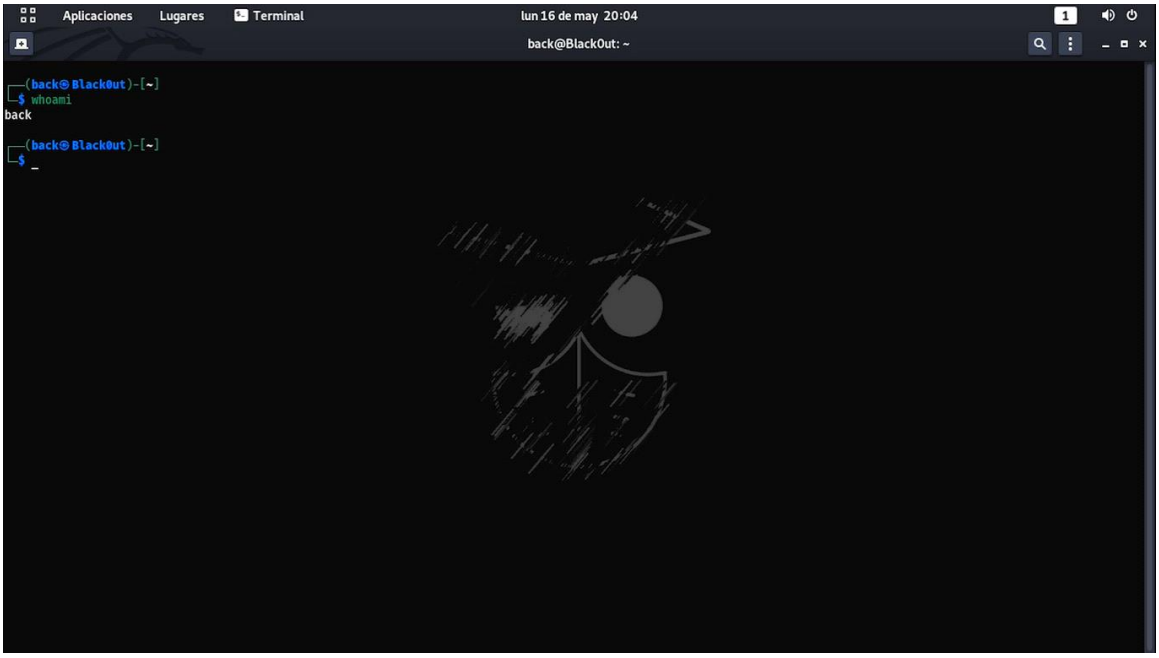
https://twitter.com/black0utq

Súper Usuario root

Haciendo una pequeña pausa para explicar este concepto antes de continuar. Naturalmente debes estar logueado con vuestro usuario personal, el cual debe ser el que estés utilizando en este momento. De todas maneras puedes ver esto desde el prompt de tu maquina como dije antes.

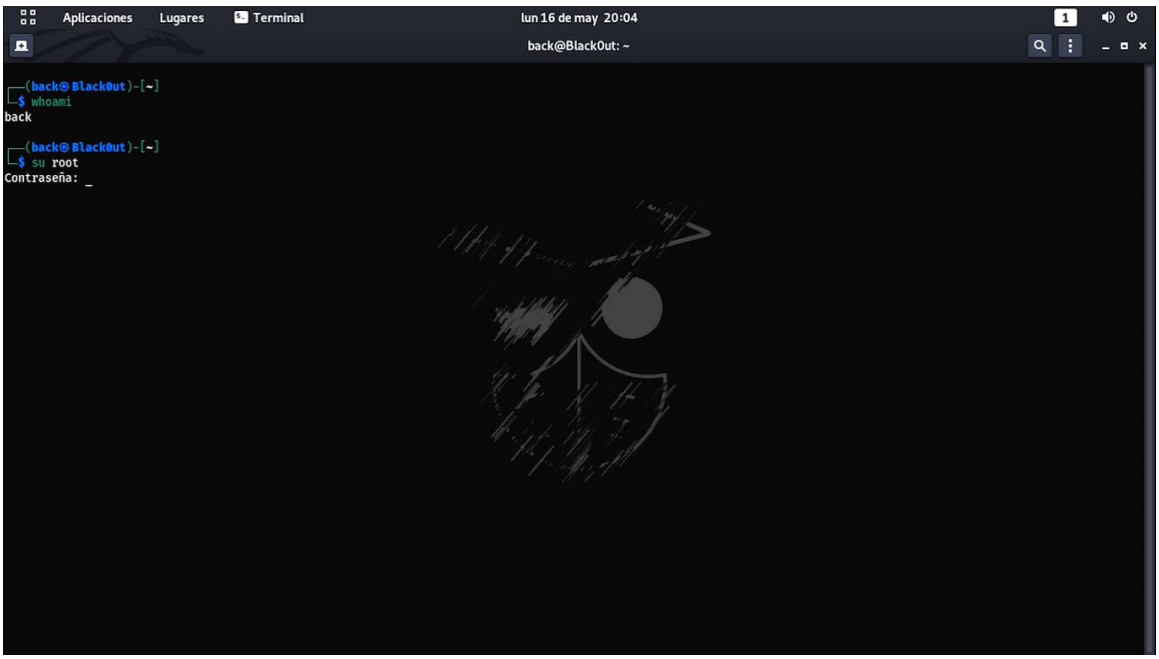
El simbolo $ nos dice de manera resumida que somos un usuario cualquiera, es decir, que no somos root.

Pero, también tienes un comando para saber quién eres, el mismo es ***whoami***



Ahora, podemos cambiar entre usuarios desde el terminal, esto con el comando ***su***

En mi sistema actualmente existe un único usuario, el cual es mi usuario personal. Y fuera de esto existe el usuario ***root***, que existe en todos los sistemas. Anteriormente dije, pero ahora me gustaría profundizar más, ***root*** es el ***súper usuario de Linux***. Que sería lo mismo que decir el administrador de Windows, aunque realmente sus privilegios serian aun mayores. ¿Que pasa que si intentamos movernos al usuario ***root*** con ***su root***?



nos pide la contraseña de ese usuario. La cual no le hemos asignado, por lo tanto, no van a conseguir acceder al mismo pongan la contraseña que pongan hasta no asignársela.

Pero de momento, para ejecutar operaciones con privilegios administrativos podemos hacerlo de manera temporal como ***root***, esto sería con otro comando en ***Linux*** que se llama ***sudo***. Que viene de ***Súper user do.***

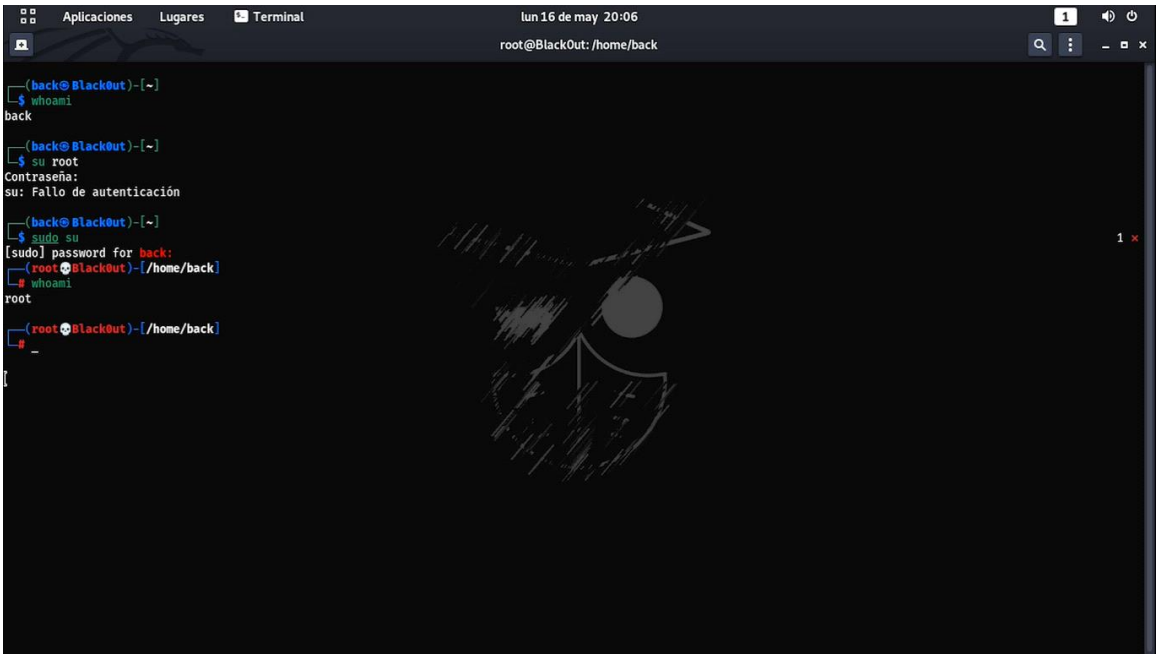
Este comando nos permitirá ejecutar cualquier cosa como si fuéramos ***root***, pasándole nuestra contraseña de usuario personal.

Pues bien si ejecutáramos ese comando y ahora quisiéramos hacer un ***su root***, sí que nos va a dejar, es decir: ***sudo su root***

Puesto que nos pedirá nuestra contraseña de súper usuario. Y una vez siéndolo podríamos hacer lo que queramos, incluido movernos al usuario ***root*** sin saber su contraseña ni tener la necesidad de proporcionarla.

Sin embargo, si no ponemos absolutamente nada va a asumir que queremos ser ***root***, es decir, podriamos abreviar con ***sudo su***

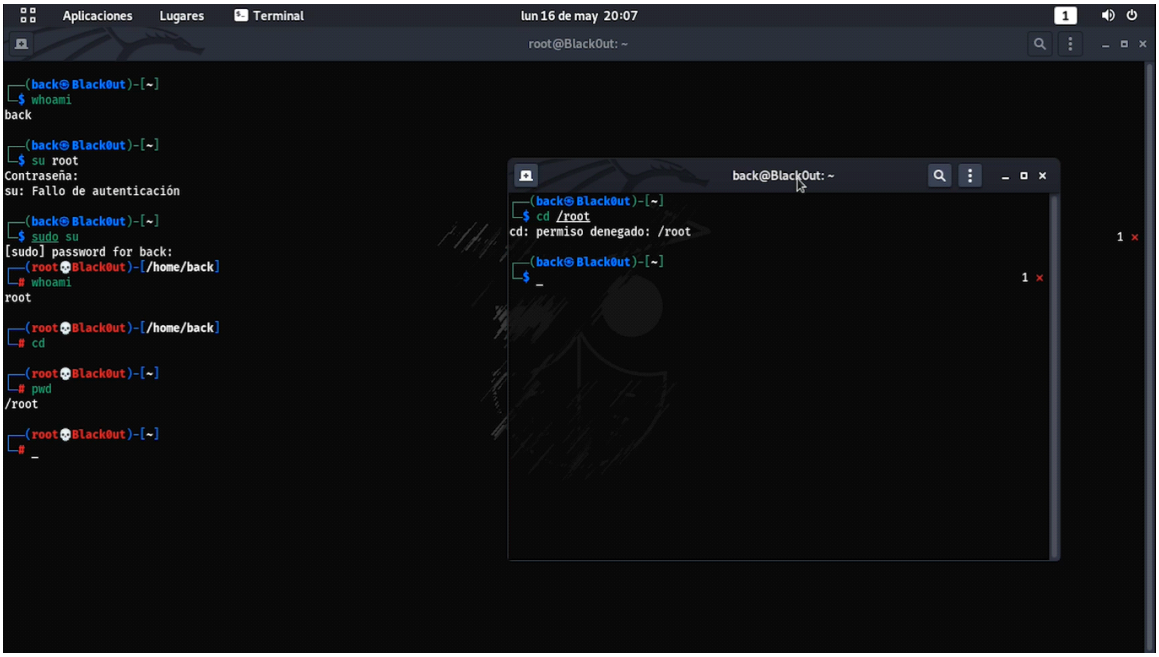
Posteriormente pondríamos nuestra contraseña, y si verificamos que usuario somos con el comando: ***whoami.***



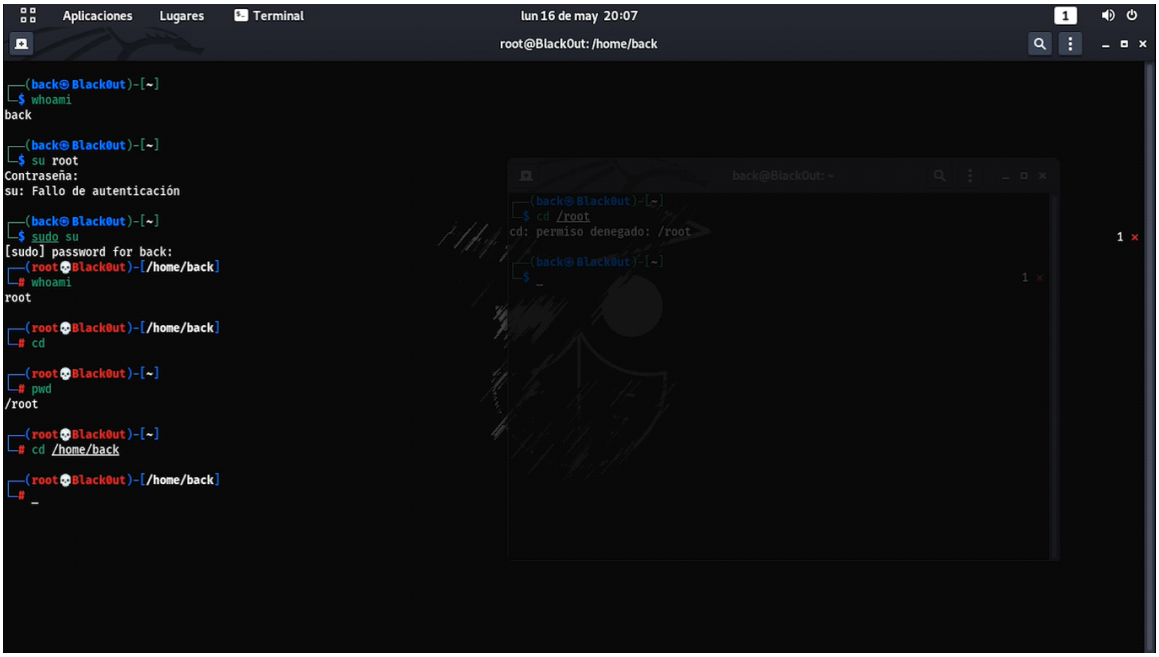
Efectivamente, somos root

Incluso fijaos que si ejecutamos el comando ***cd*** nos moverá a nuestro directorio personal de usuario, pero que en este caso sera */root* que es el directorio personal de ***root***.

Si abrimos otro terminal e intentamos movernos al directorio /*root*, verán que siendo el usuario ***back (Tu usuario personal)*** y sin proporcionar los privilegios no hay modo de que podamos acceder al directorio /*root*



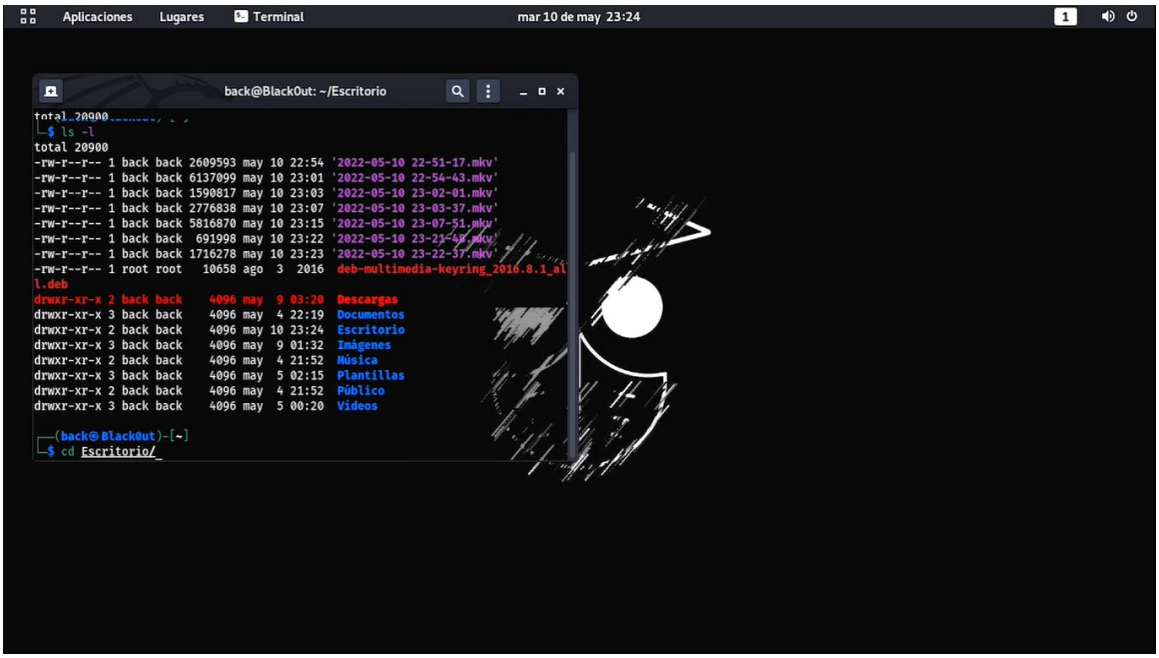
Pero si desde root queremos ir a nuestro directorio personal de usuario no nos pedirá explicaciones de ningún tipo. Tenemos total libertad sobre nuestro sistema y podamos hacer lo que queramos. ¿Se ve excelente, no?



Pues no, es sumamente desaconsejado utilizar el usuario ***root***. Puesto que esas libertades te pueden hacer cometer algún error fatal y en el peor de los casos irreparable. Más aun si apenas estáis empezando.

Lectura e interpretación de permisos

En este caso desde nuestro directorio personal, vamos a listar el contenido que se encuentra, pasándole una nueva opción a ***ls***, que será ***-l***



Y si os fijáis podemos ver mucha más información. Vamos a ver qué significa todo esto de una manera bastante simple.

Vamos a copiar cualquiera de estas cosas y en este caso no es tan importante practicar como interpretar lo que estaré haciendo. Por ejemplo yo me voy a enfocar unicamente en el directorio "Descargas".

drwxr-xr-x back back 4096 may 9 03:20 Descargas

La primera ***d*** es de directorio, puesto que básicamente esto es un directorio.

Y posteriormente tennos esta serie de caracteres repetidamente, aunque a veces hay un guion.

drwxr-xr-x

r viene de read, ósea de leer

w viene de write, ósea de escribir y

x viene de execute, ósea ejecutar

Si se fijan en este caso es un directorio, pero tiene privilegios de ejecución. Cuando un directorio tiene estos privilegios básicamente significa que podemos atravesarlo, ósea acceder a ese directorio.

Ósea si es un fichero lo vamos a poder ejecutar y si es un directorio lo vamos a poder atravesar.

Pero entonces ¿Por qué se repite?

Pues bueno en realidad estos componentes serian realmente

rwx rwx rwx

El primer privilegio de izquierda a derecha corresponde a los permisos que tenga asignado sobre este archivo el usuario propietario del mismo, que en este caso es ***back***, podemos ver que ***back*** es el propietario de este directorio puesto que es el nombre correspondiente que muestra seguido de los permisos

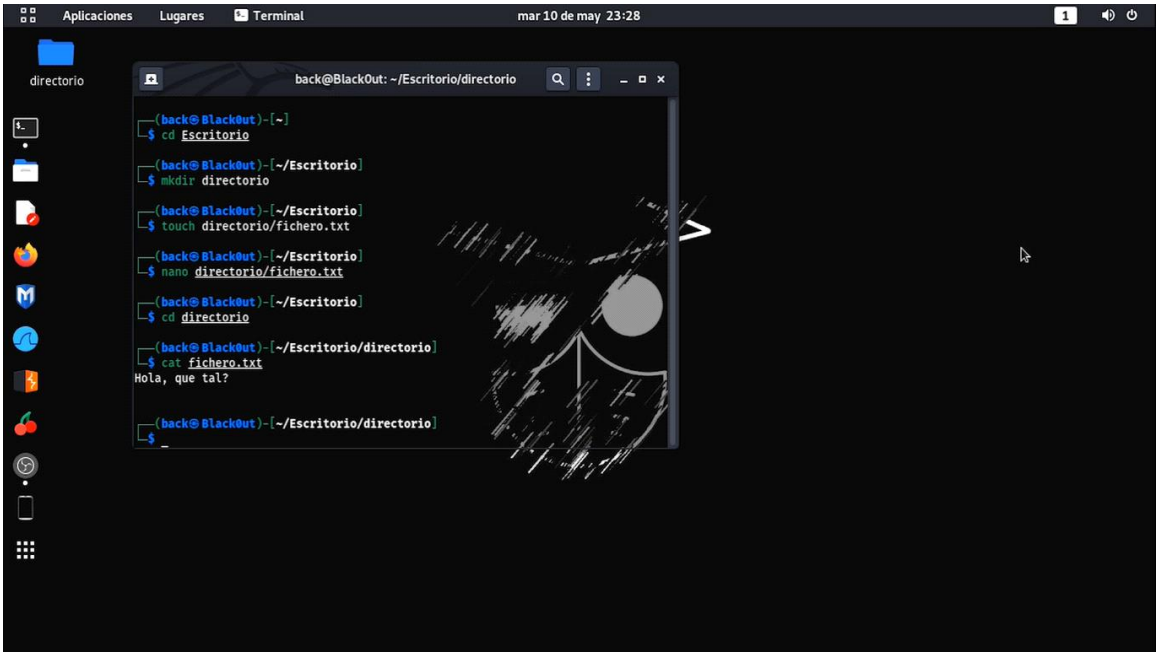
El segundo corresponde al grupo al que pertenezca ese usuario. Del mismo modo que en Windows existen grupos para los usuarios, en Linux también. Por defecto un usuario creado en Linux tiene un nombre que se llama como el propio usuario.

Y la última repetición seria los privilegios de otros. Es decir los de cualquier usuario que no sea el propietario de este archivo.

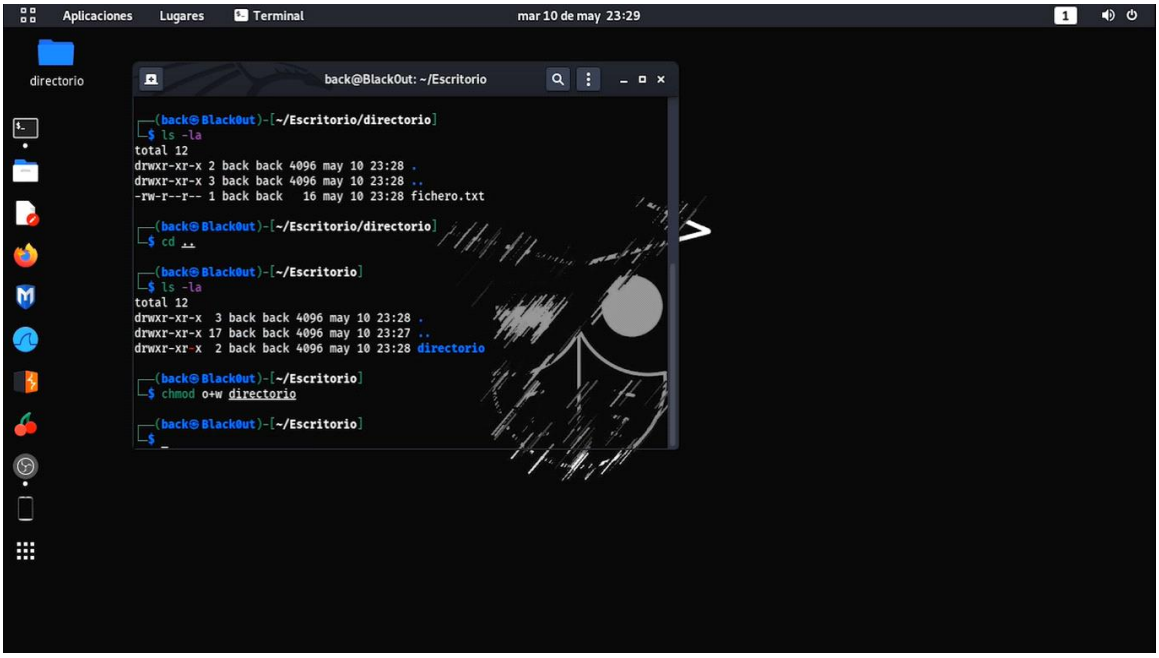
Asignación de permisos

Vamos a pasar a ver ejemplos de asignación de permisos, para lo cual primero vamos a movernos al escritorio y crear un directorio y archivo desde nuestro usuario para que vean más o menos como va esto

Y vamos a poner algo, por ejemplo "Hola que tal" en el fichero, puedes poner cualquier cosa.

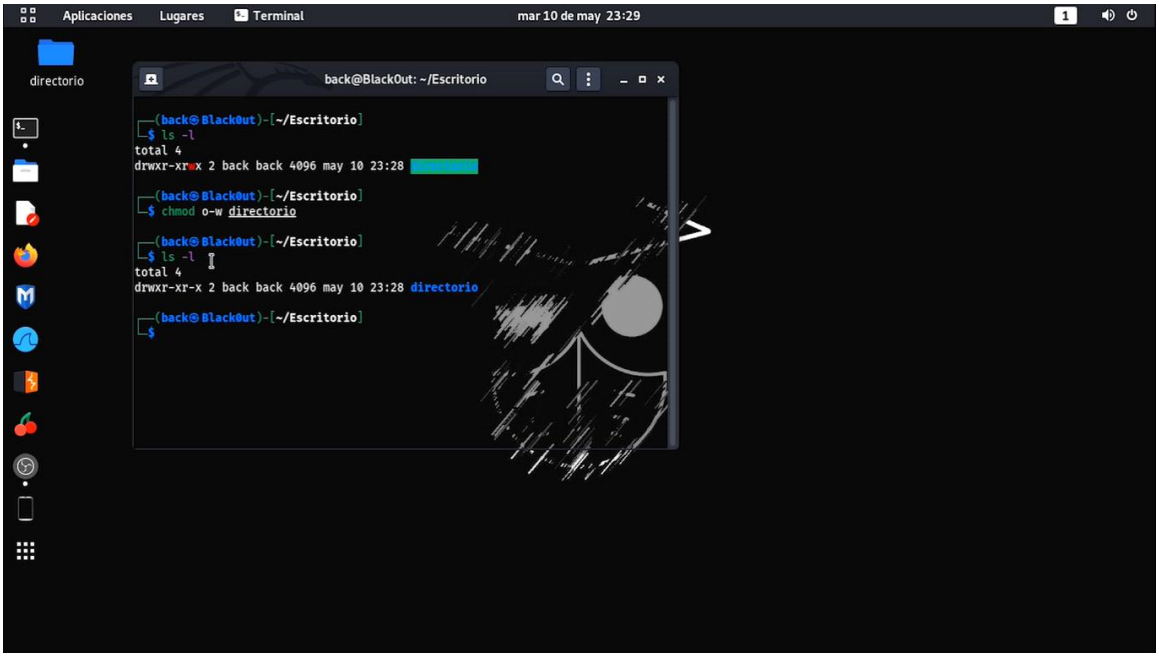


Ahora por ejemplo si listamos los permisos podremos ver que otros tienen permiso para traspasar y leer en este directorio. Pero por otra parte no para escribir. Si quisiéramos asignarle a otros privilegios de escritura sobre este fichero simplemente podríamos decirle con el comando ***chmod*** seguro del indicativo de otros, que es ***o*** + el privilegio que queramos asignarle que en este caso sería escritura, es decir lo siguiente: ***chmod o+w directorio***



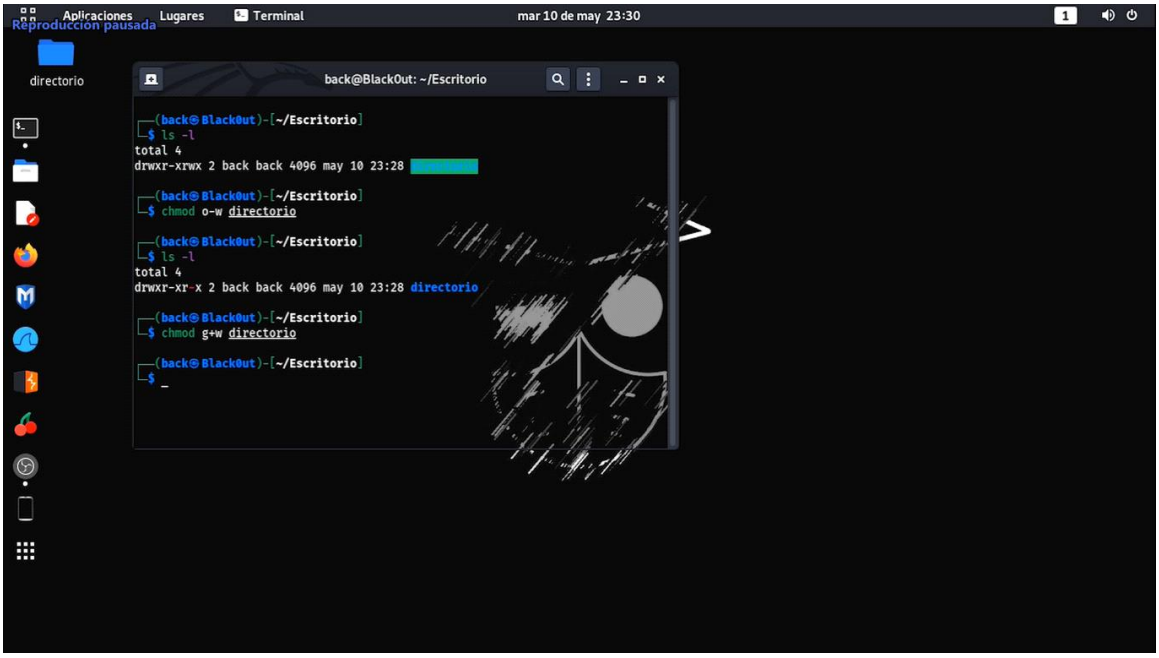
Si ahora hacemos un ***ls*** podemos fijarnos en que otros tienen una ***w***, ósea que ahora pueden escribir dentro de este documento

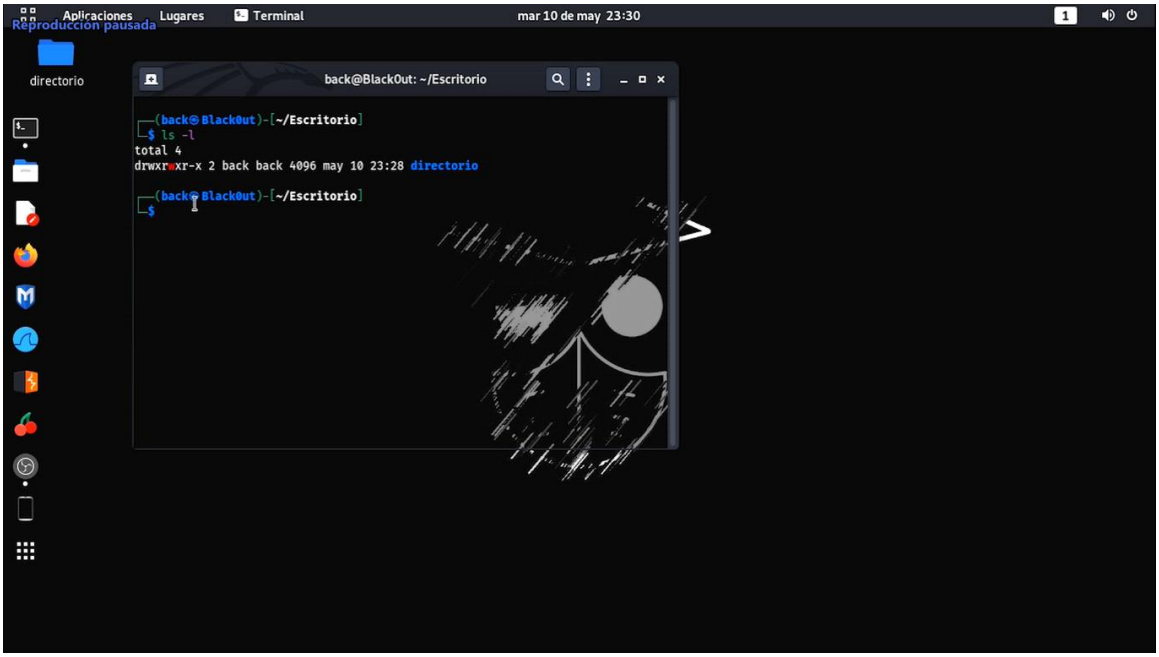
De la misma manera podríamos quitarla, con: ***chmod o-w directorio***



Igualmente aquí lo estamos haciendo con otros, pero podríamos hacerlo igualmente con los grupos, ósea:

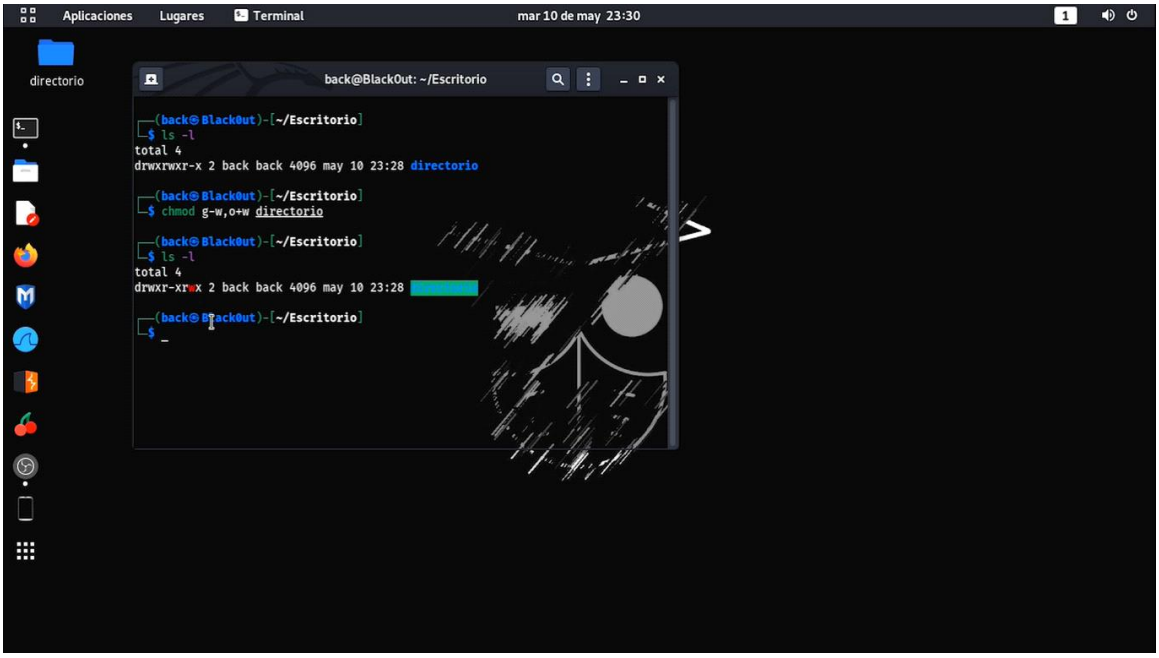
chmod g+w directorio





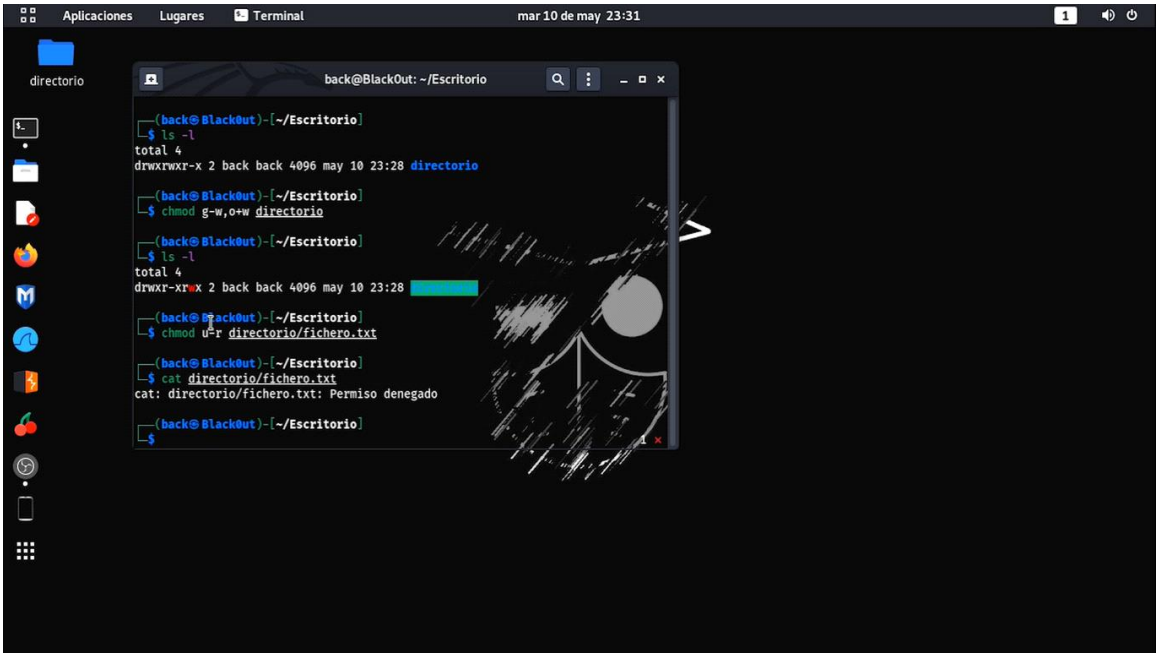
De igual modo podemos pasarle varios parámetros diferentes a la hora de asignar o retirar permisos, por ejemplo podemos decirle ahora que queremos quitarle al grupo los permisos de escritura, pero al mismo tiempo queremos que otros ahora sí que tengan dicho privilegio

chmod g-w,o+w directorio



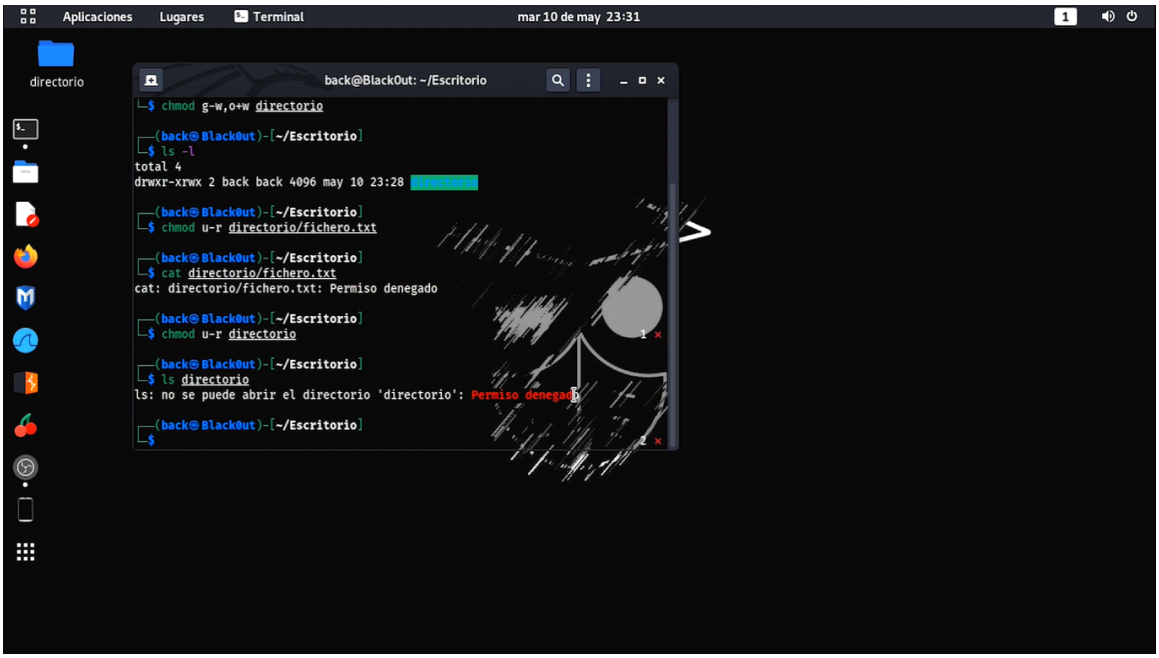
Para probar que esto que estamos haciendo es real, óseo que no es un invento mío que así funcionan los permisos vamos a retirar a nuestro usuario por ejemplo los permisos de lectura sobre el fichero que habíamos creado previamente, esto con ***chmod u-r directorio/fichero.txt***

y si ahora intentamos leer el archivo que se encuentra dentro verán que no tenemos permisos.

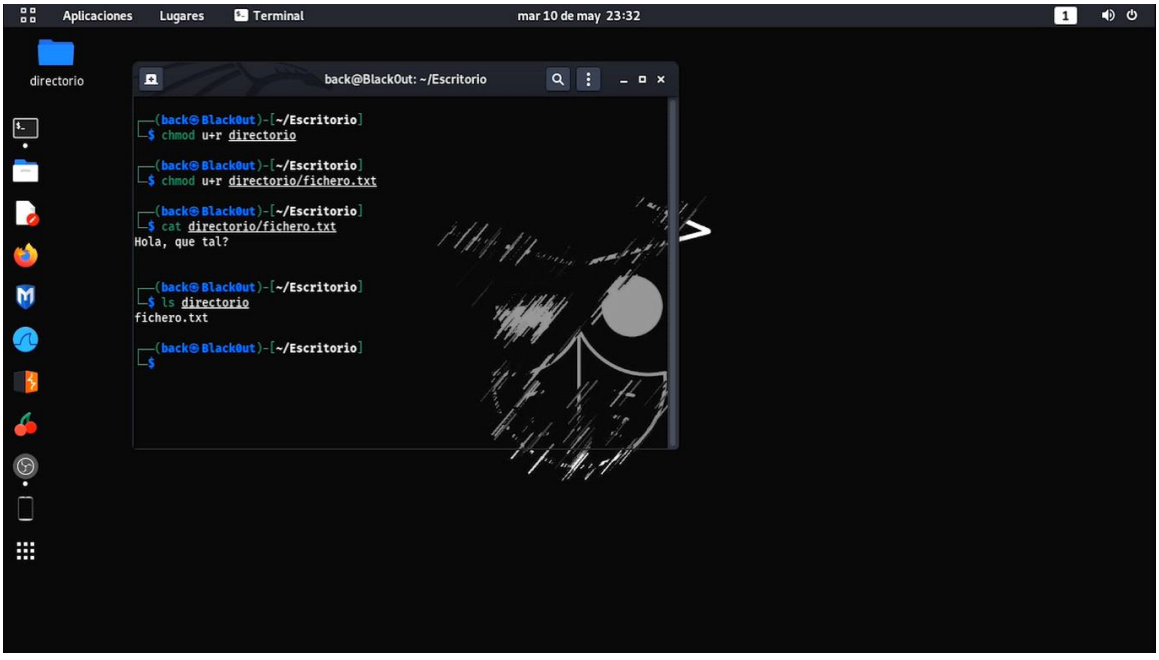


Si por otra parte quitamos los permisos sobre el directorio, es decir: ***chmod u-r directorio.***

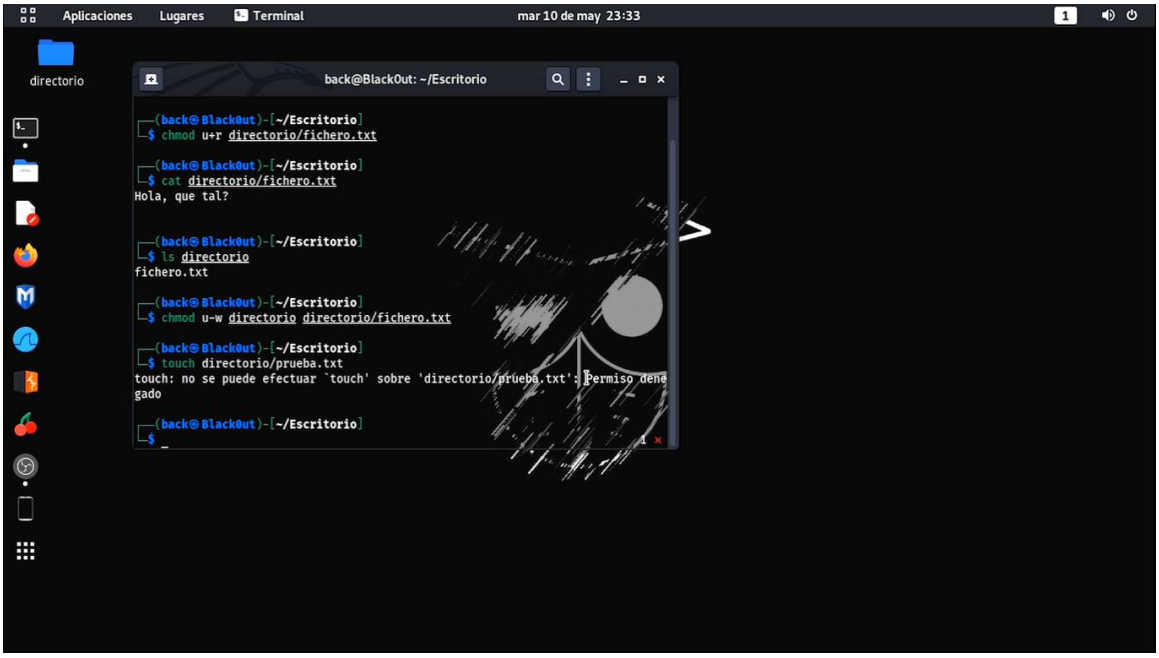
Ahora no podremos ver el contenido que se encuentra dentro del directorio

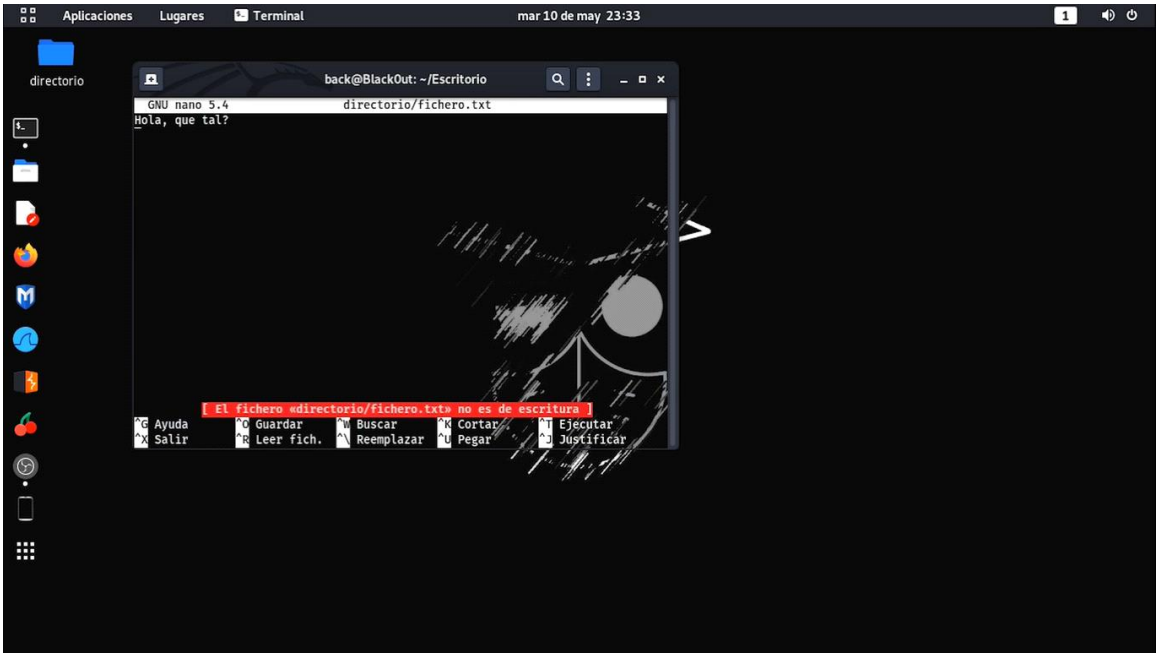


Si volvemos a asignar los permisos si podríamos hacer ambas cosas



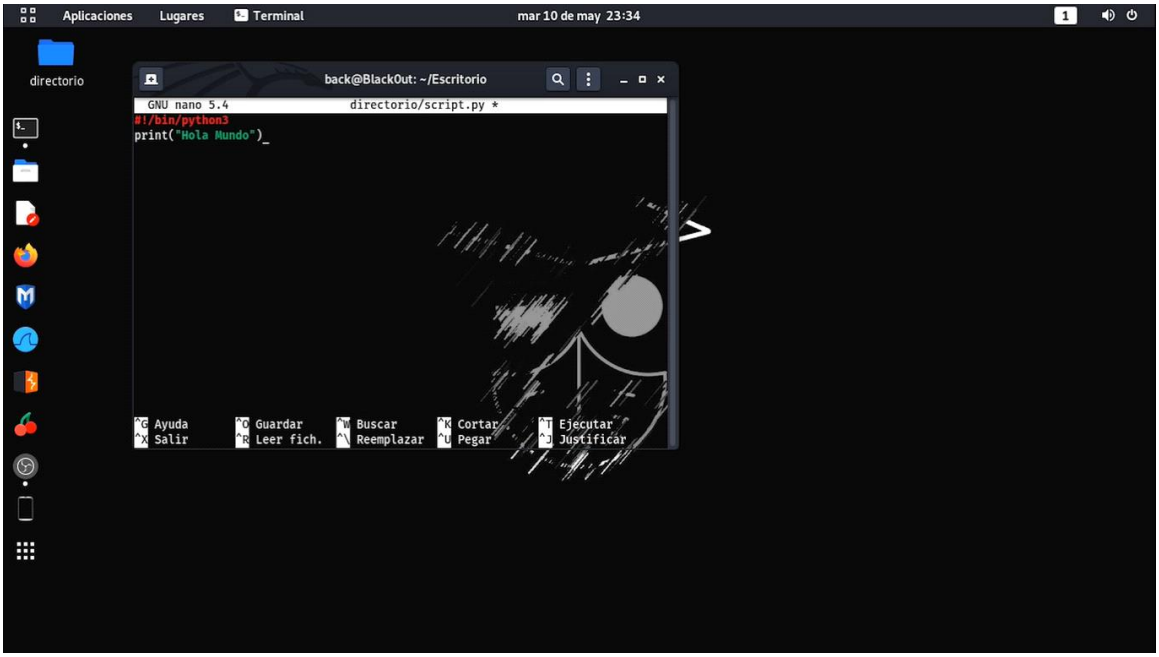
Si probamos ahora a quitarnos los de escritura en el directorio y fichero por otra parte no podríamos ni escribir en el fichero por una parte, ni tampoco podríamos crear contenido dentro del directorio





Volvamos a asignar estos permisos con ***chmod u+w directorio directorio/fichero.txt***

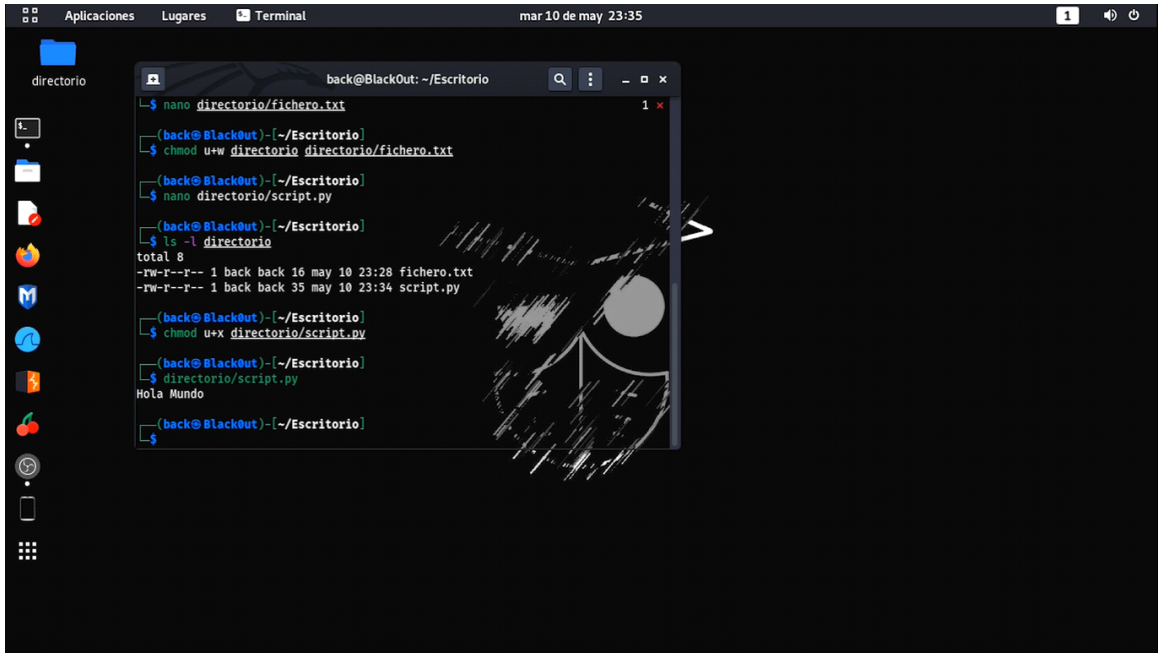
Y por ultimo vamos a verificar si nos quitasemos permisos de ejecución. Que pasa que para este ejercicio necesitaremos un fichero que pueda ejecutarse para comprender el caso, así que creare un simple "Hola mundo" en ***python***, pero esto no es importante para vosotros, quedaros con el concepto sobre la asignación e interpretación de permisos.



Una nota sobre este script, que no pasa nada porque luego vamos a ver sobre esto. Lo que defino encima del script es la ruta en la que se encuentra el intérprete de ***python***. Veremos más adelante esto, pero quédense con que es importante, porque así nuestro intérprete de ***bash*** puede entender que quiere que ejecutemos un fichero desde ***python***.

Asignémosle privilegios de ejecución a este simple trozo de código, con el comando: ***chmod u+x script.py***

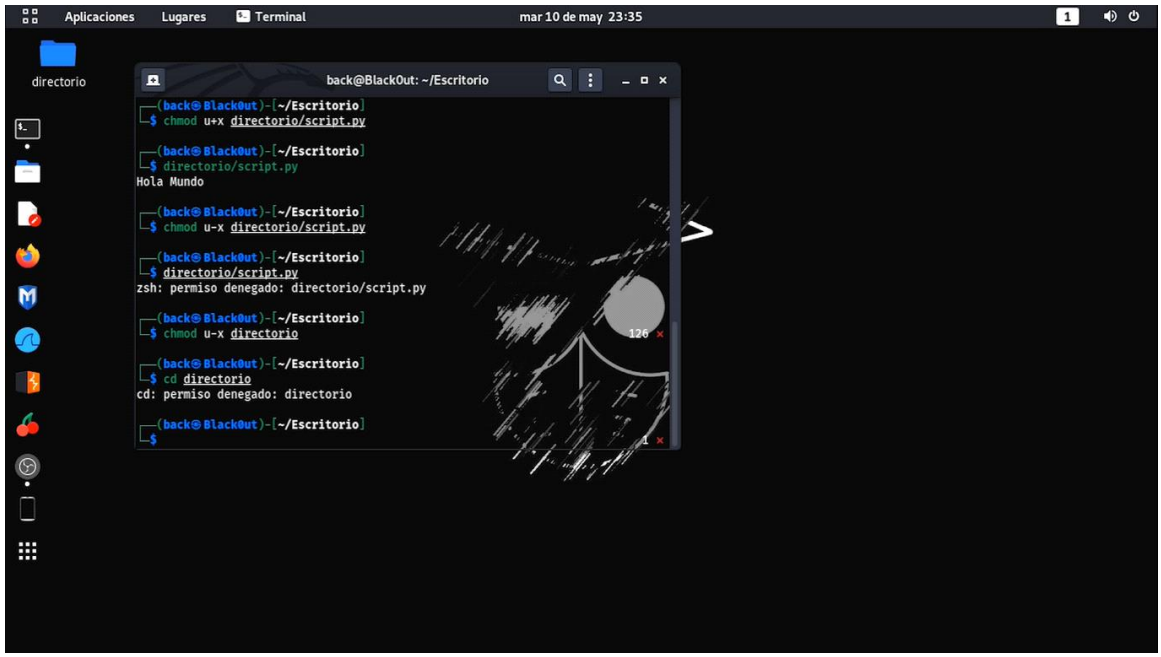
Y si lo ejecutamos podemos ver que nos devuelve en efecto un "Hola Mundo".



Si quitasemos este permiso no podríamos ejecutarlo, puesto que no tenemos permisos de ejecución.

Tambien probemos a quitar los permisos de ejecucion sobre el directorio.

Y ahora sencillamente si intentásemos traspasar el directorio, daria un error, igualmente que si intentamos ejecutar el script en ***python***.



Interpretación numérica de permisos

Vamos a movernos al escritorio y a crear un fichero donde veremos qué es esto de la interpretación numérica de permisos.

Vamos a tomar los permisos de cualquier directorio o fichero, por ejemplo yo tome los del Escritorio y pegarlos dentro del fichero para lo que veremos.

Bueno, para la interpretación de estos permisos lo que tendremos que hacer será realizar una conversión a binario, y luego de binario a decimal. Esto es sumamente sencillo, por ejemplo por cada vez que tengamos un permiso asignado pondremos un 1, y por los que no lo estén un 0.

Quedaría conformado de la siguiente forma:

rwx r-x r-x

111 101 101

La posición de estos números deberán interpretarse de derecha a izquierda, comenzando desde el numero 0. Ósea en este caso tenemos tres números, pues comenzaríamos a contar de derecha a izquierda y seria la posición 0, 1 y 2.

Lo que vamos a hacer será sumar 2 elevado a la posición, pero únicamente contando la posición si esa posición tiene un 1. Ósea primeramente calcularíamos:

111

2^0 + 2^1 + 2^2 = 1 + 2 + 4 = 7

Para el siguiente hacemos básicamente lo mismo, como la primera posición, ósea la 0 contando de derecha a izquierda es 1 elevamos 2^0, como la segunda posición es un 0 no hacemos nada. Y puesto que la tercera posición, que sería en realidad 2 si que tiene un 1. Pues elevamos 2^2. Calculamos y tendríamos el resultado

101

2^0 + 2^2 = 1 + 4 = 5

En el caso de los permisos de otros, puesto que es el mismo que el del grupo, simplemente va a dar 5. No tiene sentido que lo calculemos.

Por lo que finalmente quedaría conformado que estos permisos también se podrían traducir a una interpretación numérica como ***755***.

rwxr-xr-x = 755

Quizás te preguntes por que debes saber esto. Pues bueno, en dependencia de a lo que te dediques pero seguramente en cualquiera de los casos si en su trabajo se necesita trabajar con Linux, en algunas ocasiones os encontrareis con que deben proporcionar ciertos permisos a algún que otro fichero, como por ejemplo las claves de ***ssh*** por nombrar un

ejemplo, pero sin entrar en detalle sobre ello.

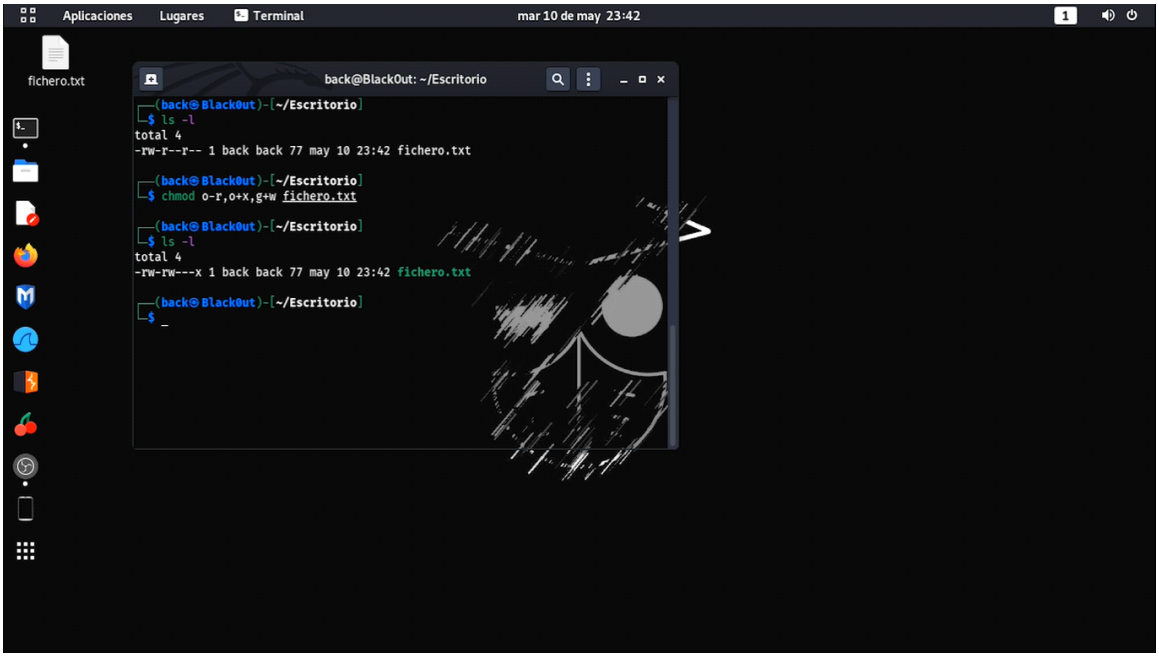
El caso es que me niego a permitir que salgan de este curso y se encuentren con algo por el estilo y luego puedan mirar atrás y decir, vaya, esto no lo vi en el curso.

Prefiero que aunque incluso lo vean y pasen de aprender a hacerlo cuando en el futuro se encuentren con este escenario miren atrás y recuerden que se los dije.

Además de esto, también se puede utilizar de manera práctica. Por ejemplo podemos asignar permisos al mismo fichero en el cual nos encontramos pasándole directamente este número que hemos calculado.

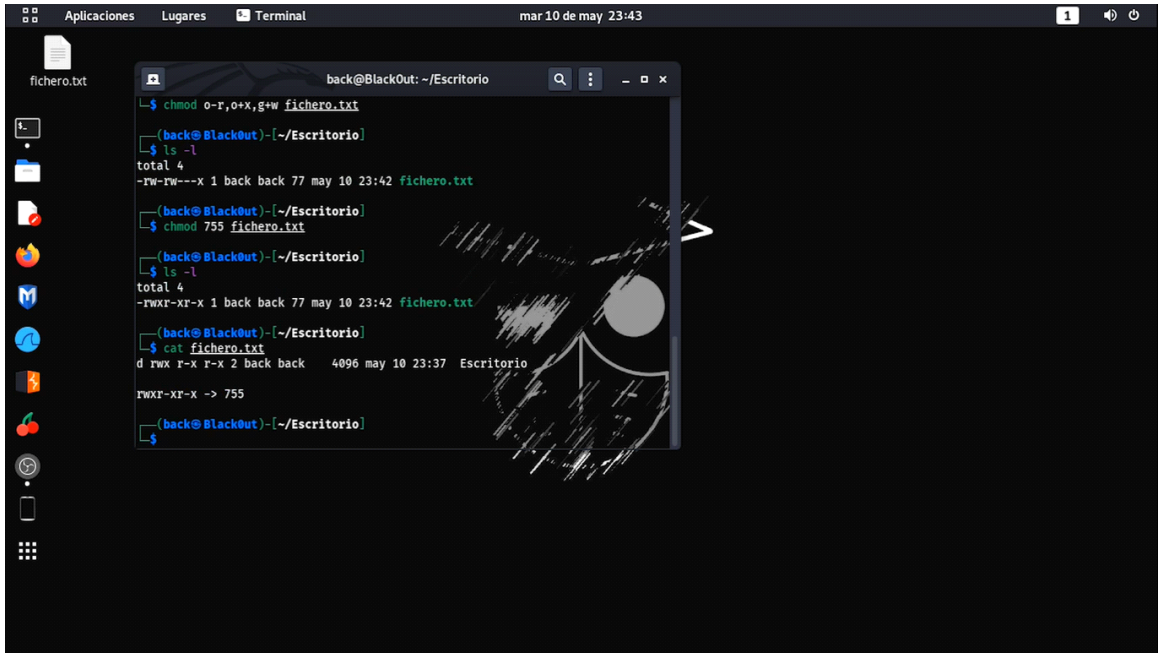
Antes vamos a modificar los permisos del fichero como hemos visto hasta ahora

chmod o-r,o-w,o-x,g+2 fichero.txt



Por poner un ejemplo, fíjense también en que cuando tenemos que asignar varios permisos al mismo tiempo. Si, se puede hacer y es más fácil que calcularlo. Pero créanme que aunque cuesta asimilarlo en un principio una vez entiendes la base y lo prácticas puedes hacerlo de memoria.

Entonces ahora que tenemos permisos extraños asignados al fichero vamos a verlos y posteriormente a aplicar esta nueva técnica, es decir, pasemosle los permisos que calculamos anteriormente, es decir: ***chmod 755 fichero.txt***



Y como pueden apreciar ahora los permisos en efecto son los que hemos calculado.

Ejercicio 2 Lectura e interpretación de permisos

Pruebe a responder correctamente las siguientes preguntas:

1- Hipotéticamente digamos que existen dos ficheros donde al leer sus permisos se encuentra con lo siguiente:

Fichero 1: ----------

fichero2: -rwxrwxrwx

a) Conviértalos decimal y responda cual sería el valor numérico correspondiente a cada uno de estos ficheros.

2- Segun el siguiente permiso en decimal, ¿Que usuario no tiene permisos de escritura, lectura y ejecución? ¿El propietario, el grupo u otros?

707

Nota:
Recuerda que puedes escribirme si tienes cualquier duda, problema o sugerencia. Naturalmente soy bastante activo en http://twitter.com/black0utq con lo cual no estaria mal que te comunicaras conmigo desde alli.

Respondiendo el ejercicio anterior (no te chives)

En el primero de los casos donde un fichero no tenga ninguno de los permisos simplemente en decimal seria:

000, no hay necesidad de calcular nada en absoluto

En el fichero 2 por el contrario donde tiene todos los permisos, sabiendo que cuando tenemos permisos de lectura, escritura y ejecución podemos llevarlo a binario como 111 y posteriormente, como debemos calcular 2 elevado a la posición de derecha a izquierda comenzando desde 0, el cálculo quedaría conformado como 2^0 + 2^1 + 2^2, 2^0=1, 2^1=2 y 2^2=4, si sumamos todo esto, ósea 1+2+4 sería igual a 7. Y como en este caso tanto el propietario como el grupo y otros tienen asignados todos los permisos, podemos anticiparnos a que el resultado del grupo y del propietario es idéntico, por lo cual la respuesta seria ***777***.

rwx

2^0 + 2^1 + 2^2 = 1 + 2 + 4 = 7

En el ejercicio dos no era necesario siquiera calcular, se puede intuir incluso que bajo el permiso ***707*** el único que no tiene permisos en absoluto es ***el grupo***.

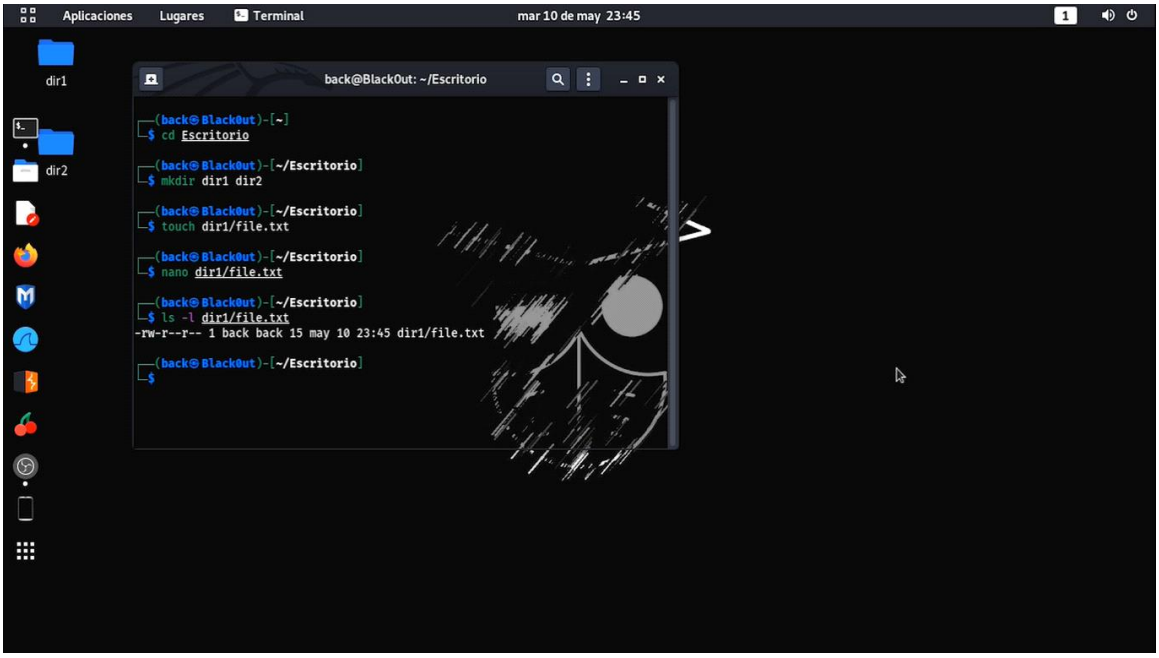
Enlaces en ficheros

Con los capítulos anteriores espero haber fomentado suficientemente bien la interpretación de permisos en Linux, por lo que para aquellos que os hayáis sentido agobiados, no se preocupen porque vamos a volver a tomar aire repasando algo diferente que son los enlaces simbólicos.

Vamos a comenzar creando dos directorios, y dentro de uno vamos a crear un fichero.

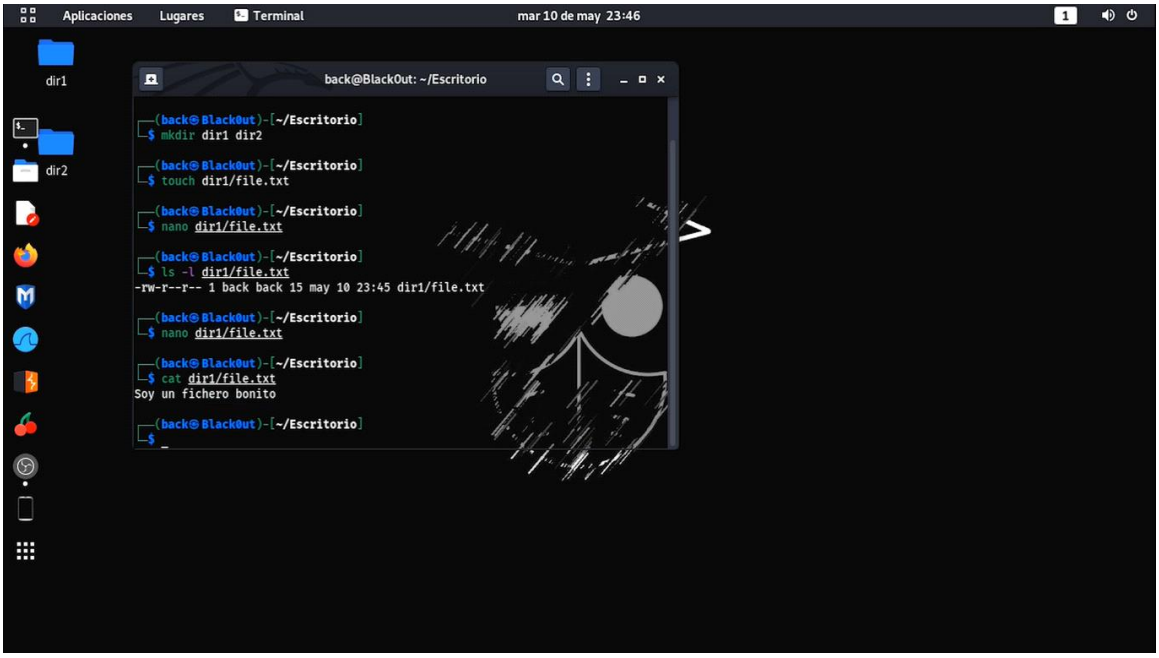
Dentro del fichero vamos a poner cualquier cosa, por ejemplo "Soy un fichero"

Si hacemos un ls -l para ver ese fichero:



Fijaos que al lado de los permisos tenemos un ***1***. Este es el numero de enlaces duros, un enlace duro básicamente es un puntero a un fichero, esto lo que significa es que si modificamos ese fichero pues nada, se modifica. Si, se que suena un poco extraño, pero vamos a ver ahora como va esto.

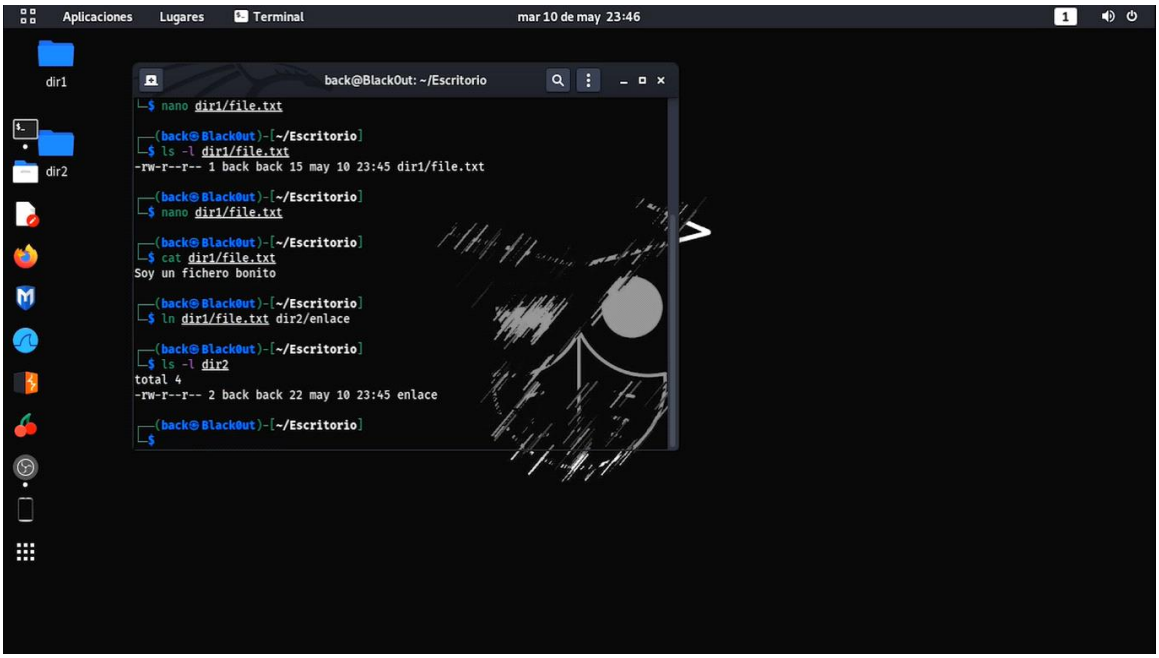
Si editamos este fichero, por ejemplo con nano ponemos... "Soy un fichero... bonito" y vemos el fichero, nada se modifica.



Entonces, lo que vamos a hacer será crear otro enlace a este mismo fichero, que actualmente tiene uno. para ello vamos a crear en el dir2 dicho enlace con el comando ***ln***, seguido de la ruta a la cual queremos hacer ese enlace y seguido del nombre que vamos a darle a este enlace duro, es decir algo asi:

ln dir1/file.txt ../dir1/file.txt

Y si ahora listamos para ver los detalles de este fichero fijaos que pasa:

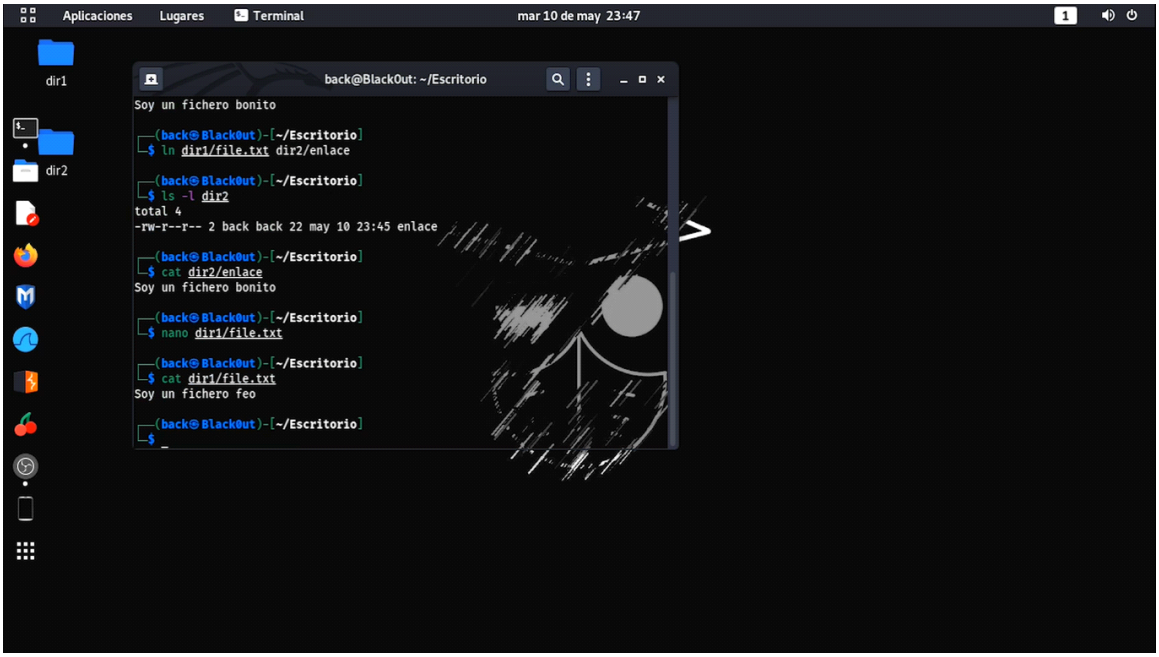


En efecto, tenemos dos enlaces a ese fichero. Esto significa básicamente que el primer fichero y este que hemos creado son exactamente los mismos, ósea estamos apuntando al mismo contenido. Podemos hacer un cat al enlace y verán que efectivamente, el contenido es el mismo.

¿Que pasa si intentamos modificar el enlace duro?

Pues bueno en este caso voy a editarlo con nano y pondremos, "Soy un fichero... feo"

Si ahora intentamos ver el contenido del fichero original podremos verificar que el contenido ha cambiado



¿Por que?

Pues porque los cambios que se aplican a enlaces duros, se aplican a todos los ficheros que conformen dicha red de enlaces, ósea los cambios se aplican sobre todo el sistema. Y los ficheros serán siempre idénticos.

Pero ahora veremos la contra parte que son enlaces blando, para ello primeramente eliminaremos el enlace duro:

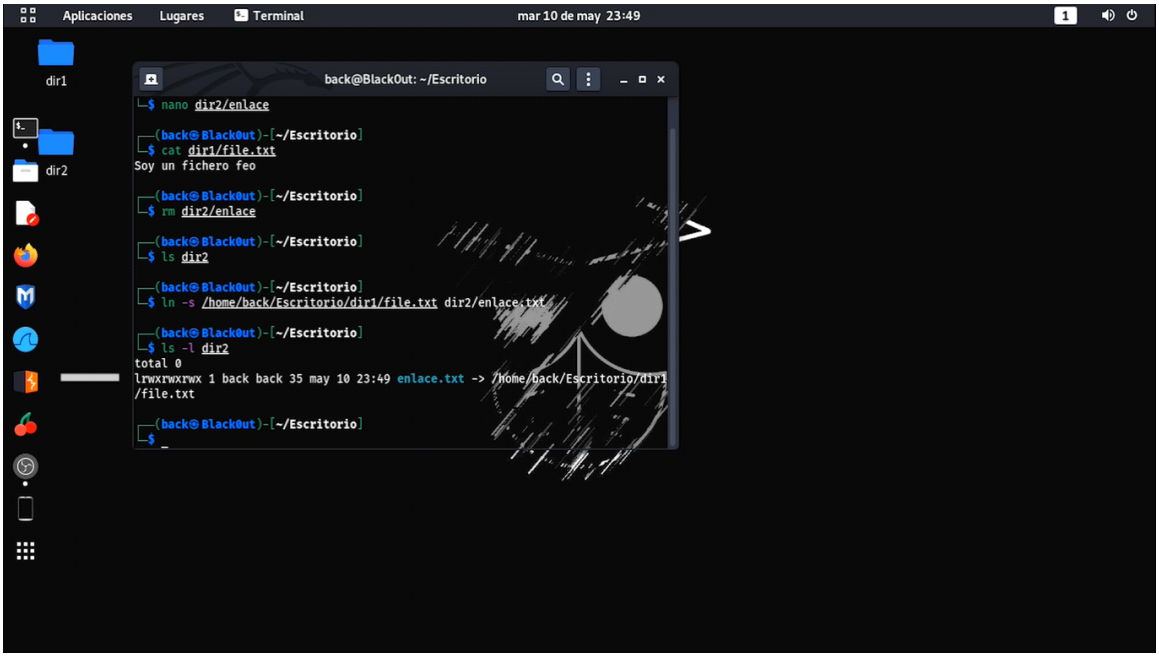
rm dir2/enlace.txt

Entonces para crear un enlace blando, o mejor conocido como enlace simbólico podemos hacerlo con el comando ***ln***, que significa ***link***. El mismo viene acompañado de

-s que viene de *simbólico*, esto seguido del objetivo que en este caso se encuentra en ***/home/back/Escritorio/dir1/file.txt*** y el nombre del enlace que le voy a llamar ***enlace.txt***, es decir:

ln -s /home/back/Escritoire/dir1/file.txt enlace.txt

Si listamos el contenido de cualquiera de estos directorios, por ejemplo dir2



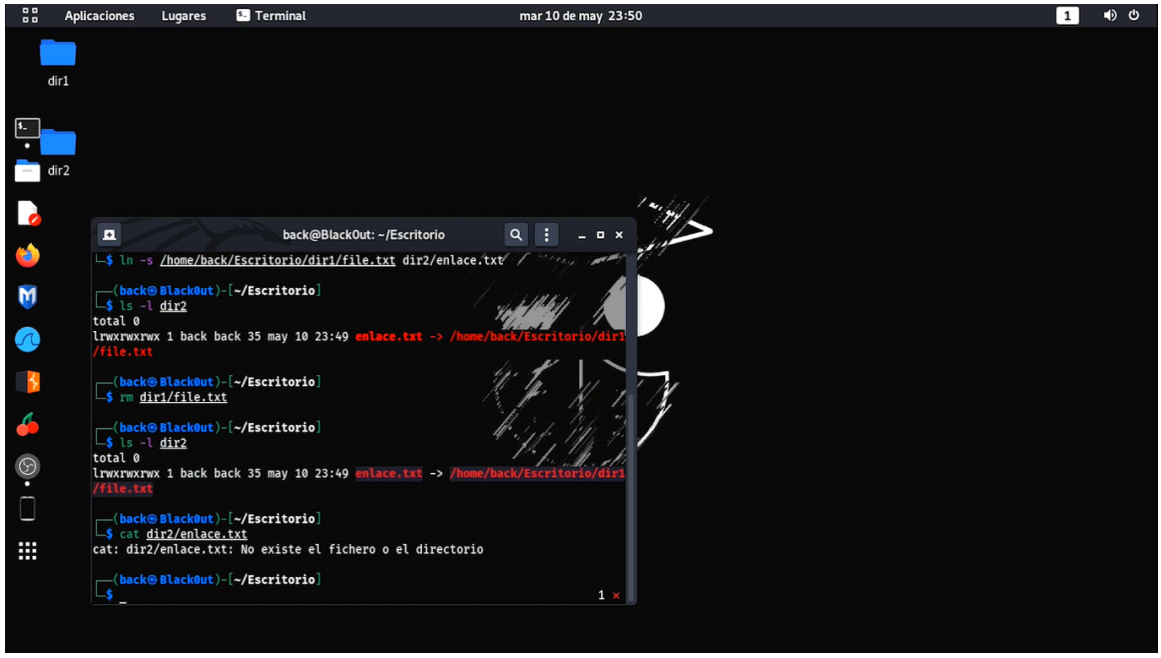
Entre otras cosas fijaos que a la izquierda de los permisos sale una ***l***, esta ***l*** es únicamente para los enlaces simbólicos

veremos que en este caso tiene un solo puntero, y al lado nos dice que el enlace apunta al fichero. ¿Cual es la diferencia?

Pues que los enlaces duros apuntan al contenido del fichero en sí. En este caso, los enlaces simbólicos no, sino que apuntan al fichero en sí mismo.

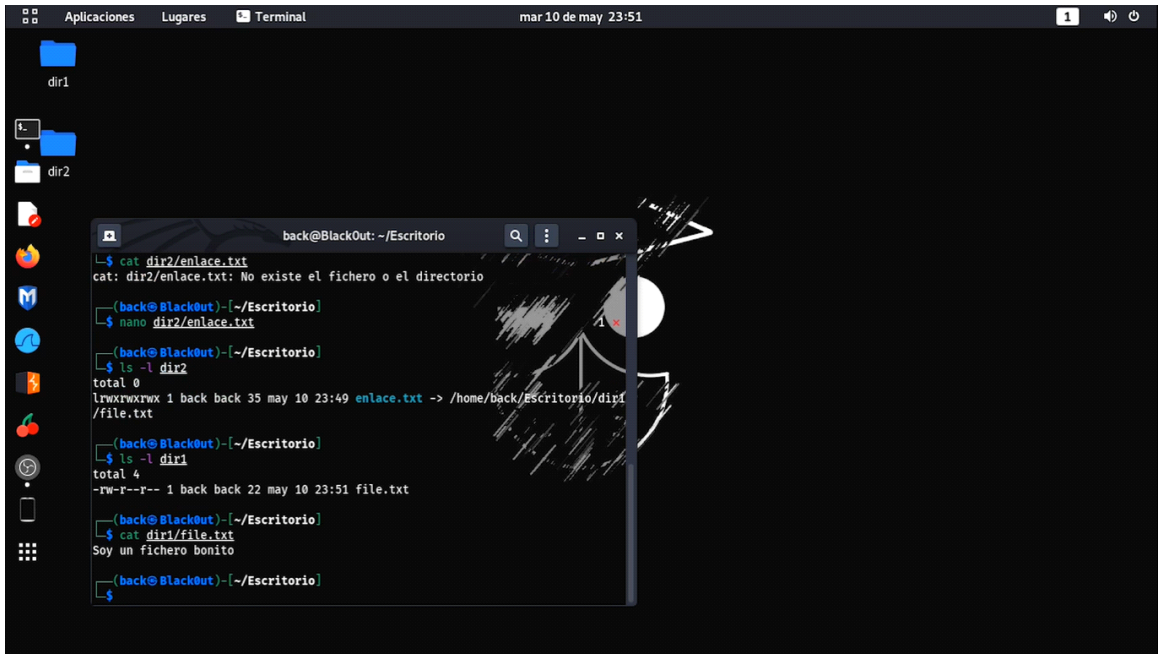
Claro está que si modificamos el enlace el cambio también se verá reflejado en el otro fichero, pero no tenemos dos enlaces duros, sino uno.

Pero por ejemplo si ahora borramos el fichero original.



Ahora el enlace simbólico apunta a un fichero que no existe, esto es un *enlace colgante*. Por lo cual si intentásemos ver el contenido del enlace nos dirá que no existe.

Fijaos en algo curioso, y es que si ahora intentamos modificar el enlace con nano y escribimos cualquier cosa y lo guardamos. Vuelve a crearse el fichero original, aunque bueno, no es el original en sí.



Esto es relativamente equivalente a los accesos directos en Windows, por ejemplo si tienes un acceso directo a una aplicación, esto más o menos es lo mismo.

Así que la diferencia sería esa, un enlace duro apunta al contenido, y un enlace simbólico apunta al fichero.

Enlaces en directorios

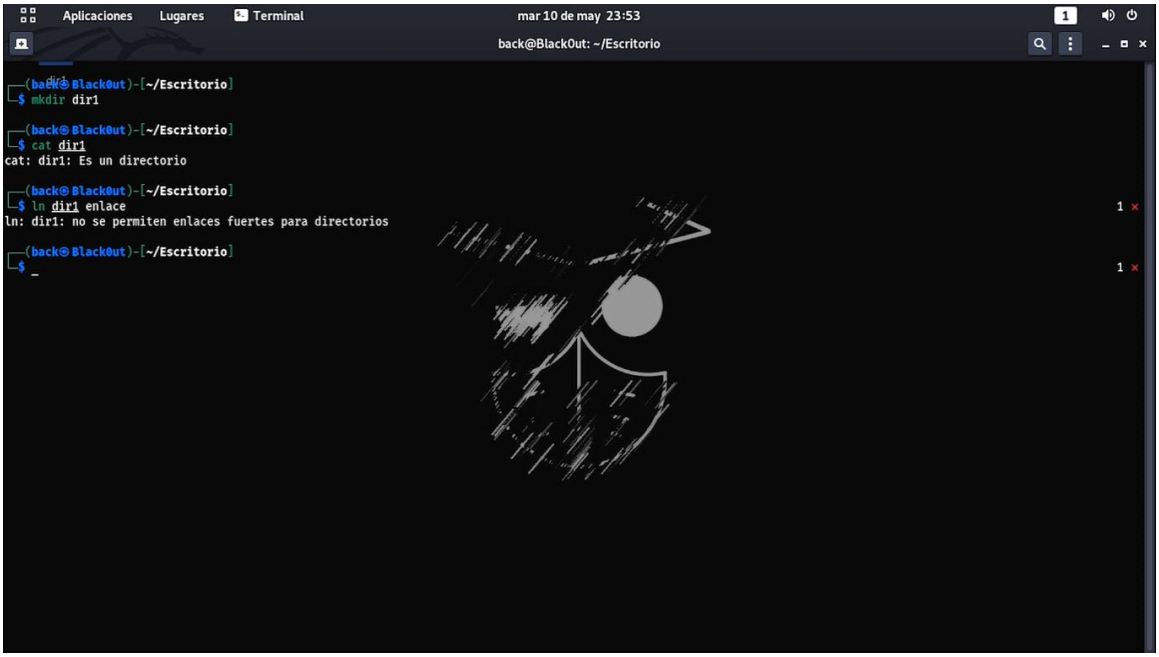
Ya hemos visto el tema enlaces en ficheros, nos faltaría ver cómo funcionan estos mismos enlaces, pero orientado a los directorios.

Primeramente vamos a crear un directorio que vamos a utilizar como ejemplo:

mkdir dir1

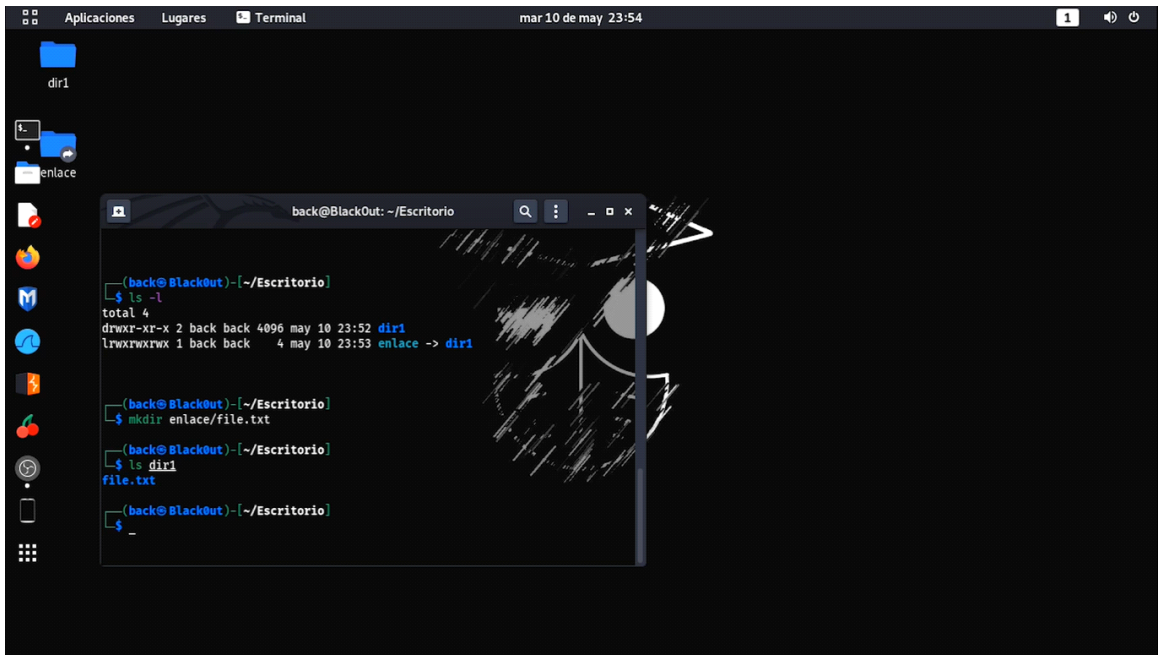
Y bueno, anteriormente habíamos visto como crear enlaces duros y blandos. Pero si interiorizaron el concepto sabrán que no se puede hacer un cat a un directorio, y que un enlace duro lo que hace es apuntar al contenido de este fichero, con lo cual es imposible crear un enlace duro a este directorio, de hecho, podríamos intentarlo.

ln dir1 enlace

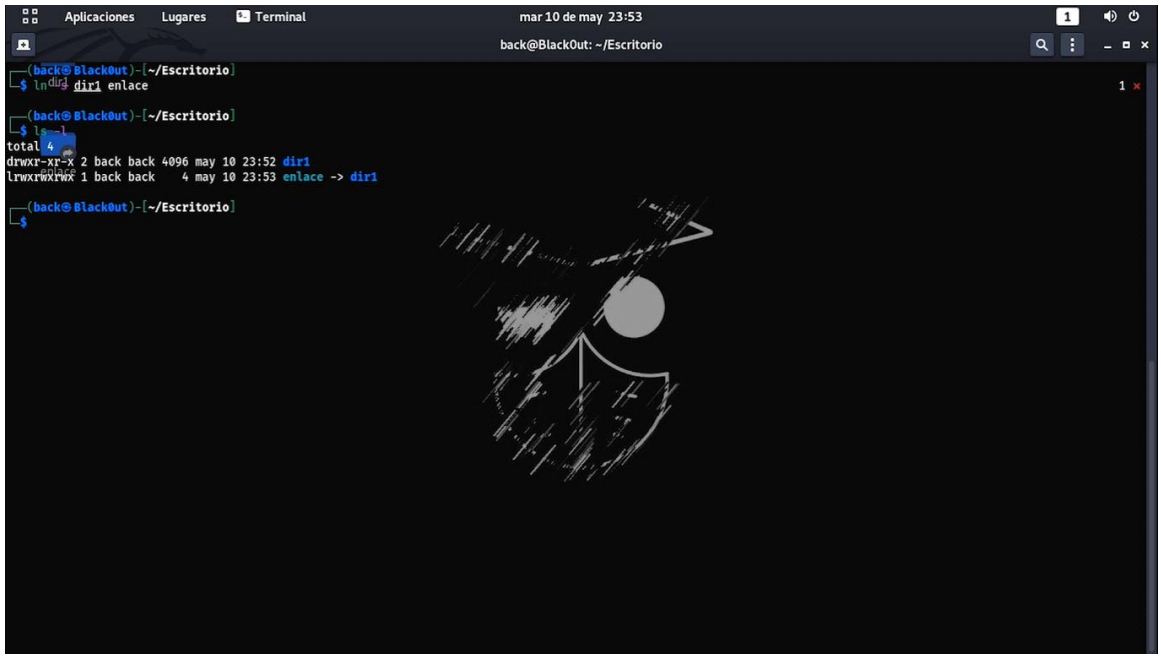


Pero podréis notar en el mensaje de error que en realidad esto no se puede. Pero del mismo modo en que en Windows pueden crear accesos directos a cualquier carpeta en el

sistema, y como a fin de cuentas los enlaces blandos son equitativos a los accesos directos de Windows, si que podríamos crear un enlace blando a ese directorio.



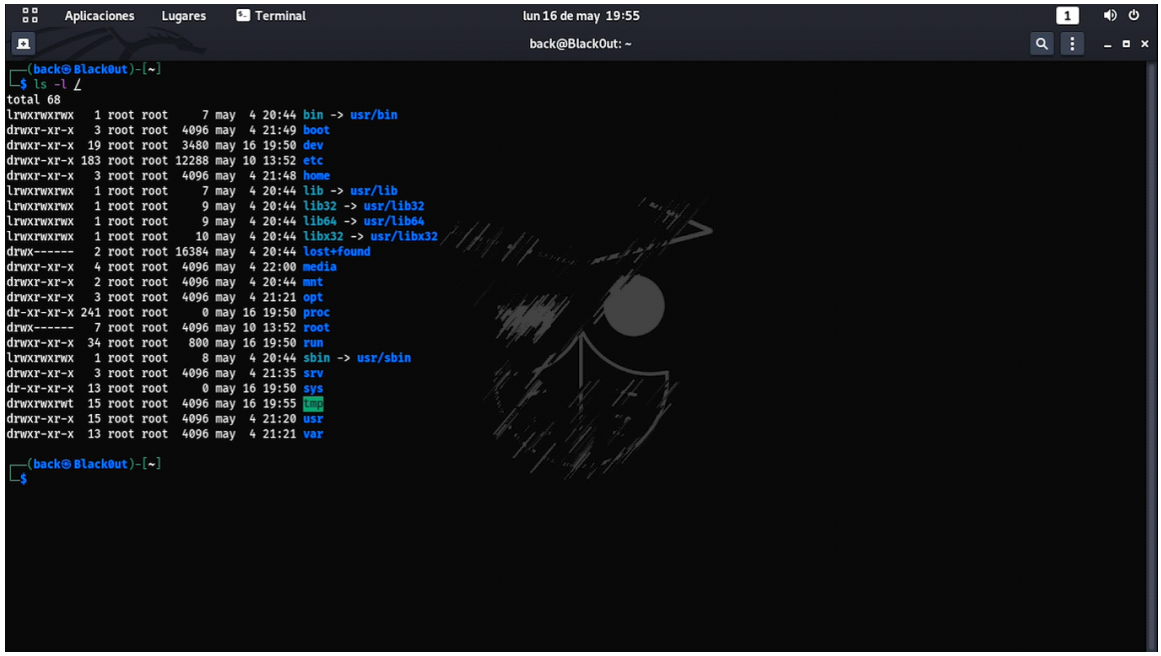
Una vez creado podríamos crear un directorio o fichero dentro de este enlace y se vería reflejado en el directorio original, como han de intuir.



Jerarquía de directorios

Antes de seguir vamos a ver cuál es la jerarquía de directorios en Linux, no a gran profundidad puesto que esto no es un curso de administración de sistemas ni nada por el estilo, pero sí que es importante conocerlos medianamente

Por lo cual, vamos a listar el directorio del cual cuelga el sistema y posteriormente voy a dar un breve preámbulo de para qué es cada directorio que cuelga del mismo:



Primeramente tenemos un enlace simbólico que es ***/bin***, que si os fijáis nos redirige a ***/usr/bin***. Este enlace existe porque dicho directorio es verdaderamente importante y muy utilizado, así que el crear dicho enlace resume el hecho de tener que dirigirnos ***a /usr/bin***.

Aquí podremos encontrar una gran cantidad de comandos utilizados por el sistema, por ejemplo todos los que hemos estado utilizando se encuentran aquí, ósea ls, mkdir, rmdir, etcétera.

El segundo directorio que tenemos seria ***/boot***. En este meramente se encuentran archivos necesarios para que el sistema arranque.

/cdrom es un directorio que se utiliza, como su nombre indica básicamente para los cd. Aunque se podrían considerar prácticamente extintos aun algunas personas utilizan algunos, por lo cual dicho directorio permanece ahí por si acaso en algunos sistemas,

aunque en el mio ya no sale.

/dev es donde están los dispositivos

Aquí podemos encontrar por ejemplo los discos duros y otros tipos de dispositivos que tengamos en el PC.

/etc Aquí se almacenan los archivos de configuración, de diversidad de cosas.

Básicamente casi cualquier archivo de configuración del sistema y de algunas herramientas se encontraran aquí, exceptuando herramientas que traigan su propio fichero de configuración y por alguna razón lo pongan en otra parte, que en teoría, de poder, pueden.

en el directorio ***/home*** tenemos el directorio donde se almacenan los directorios personales de cada usuario del sistema.

Por ejemplo aquí podrás encontrar tu directorio personal de usuario. Y si hubieran más usuarios en el sistema seguramente, ósea por defecto encontrarías sus directorios personales aquí

/lib, /lib32, /lib64, libx32 básicamente son directorios que se utilizan para las librerías del sistema, ósea de comandos del sistema. Que fijaos que en concreto son enlaces simbólicos cuyos directorios verdaderamente se encuentran dentro del directorio ***/usr***

Para el que no se sienta familiarizado con ningún lenguaje de programación, quiero decirle que son muy escasos por no decir nulos en los que un programador no utiliza una librería para crear su programa. Pero, ¿Que es una librería? Pues en resumidas cuentas una librería es un trozo de código que sirve para ejecutar instrucciones diversas y específicas. Por ejemplo a lo mejor quieres crear un programa que ejecute cierta función muy compleja y a bajo nivel, ósea utilizando instrucciones verdaderamente complicadas. Puedes hacerlo, claro está, pero en la mayoría de los lenguajes de programación existen estas libreras, que en gran parte de las ocasiones puede que te faciliten lo que quieras hacer, ósea podrían ser miles de líneas de comando. Pero bueno, este capítulo no va sobre programación.

/lost+found El mejor ejemplo para explicarte la funcionalidad de este fichero es que te pongas en el escenario que se me ocurre, que estés trabajando tranquilamente en tu curso de bash scripting, creando tu temario en un documento y de repente se vaya la luz. Como no se ha guardado el fichero con el contenido, este fichero queda corrupto. Entonces es enviado a ***/lost+found***, que es donde se almacenan este tipo de archivos para posteriormente intentar recuperar algo de los datos que se perdieron.

/media Aquí básicamente se guarda cualquier dispositivo de almacenamiento que introduzcas en tu ordenador, un dispositivo USB, un disco duro externo, etc

/mnt Es un directorio que se utiliza para algo similar, que es montar temporalmente sistemas de ficheros.

/opt Es donde se instalan paquetes, distintos... Por ejemplo que no sean de código abierto

/proc Aquí podremos encontrar información del sistema

/root ya sabéis que es el directorio personal del súper usuario root.

/run almacena archivos temporales de aplicaciones que se encuentren corriendo.

/sbin es similar a /bin

Pero en este caso los comandos no son accesibles para todos los usuarios, sino que deberían ser meramente utilizados por el usuario root por su severidad.

/srv Es utilizado para almacenar servicios de tu sistema, por ejemplo si tienes una página web varios datos podrían almacenarse aquí.

/sys Aquí básicamente hay cosas del sistema, tipo firmware, kernel, etcétera y te sugiero no toquetear nada de esto si no tienes ni idea de lo que estás haciendo.

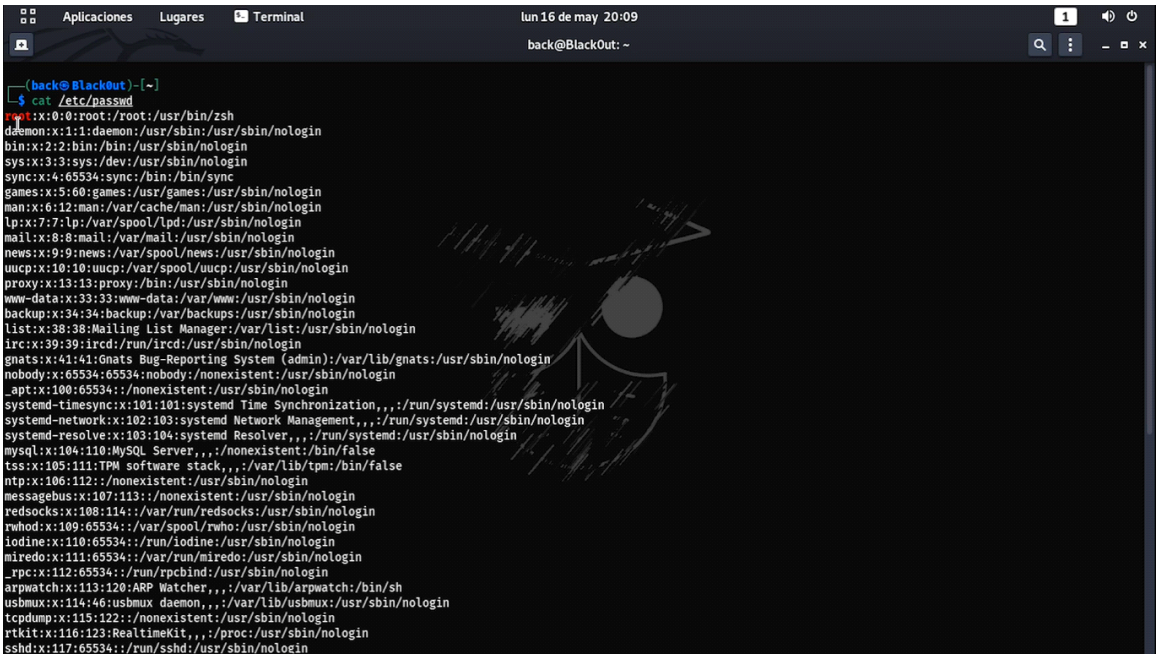
/tmp lo que hace es almacenar información de manera temporal, por ejemplo nosotros mismos podemos guardar cualquier cosa en dicho directorio, pero bueno, todo lo que tengamos aquí es temporal. Ósea que no se almacenara para siempre.

/usr Aquí se encuentran comandos utilizados por los usuarios del sistema

/var Aquí básicamente podremos encontrar archivos variables. Por ejemplo logs de ciertos aplicativos o información que pueda variar.

Usuarios y grupos en Linux

En este capítulo vamos a ver la información acerca de los usuarios y grupos en Linux. La información de estos se guarda en un fichero que es el ***/etc/passwd***

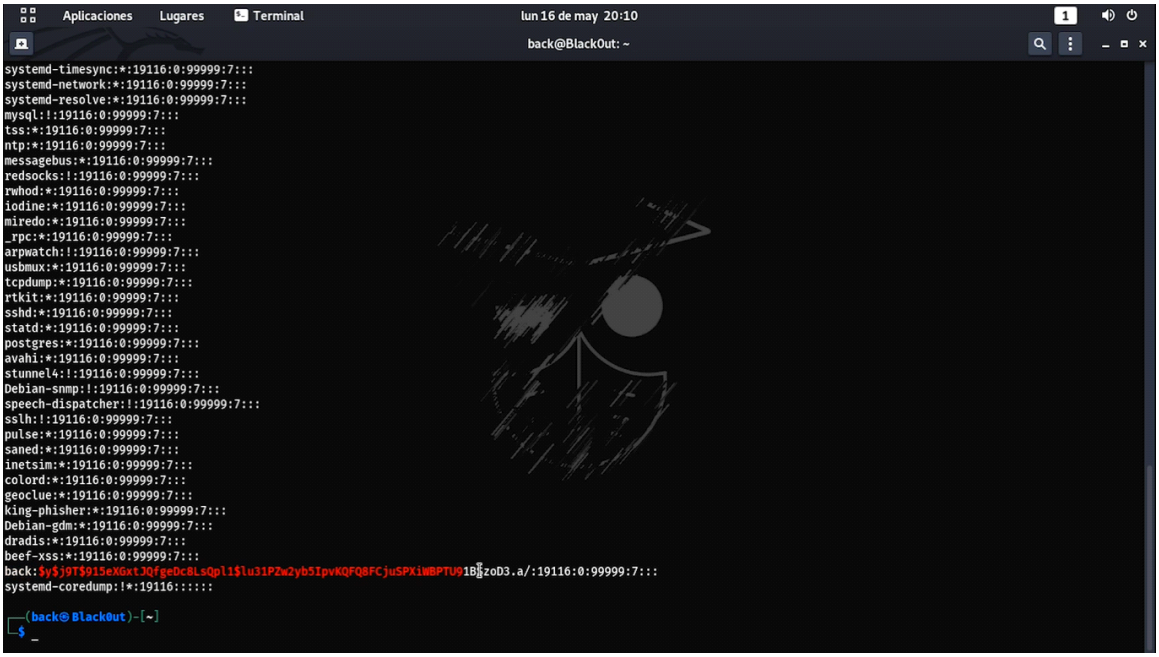


Vamos a ver cómo podríamos desglosar todo lo que vemos aquí. Primeramente a la izquierda tenemos el nombre de cada usuario, por ejemplo fijaos que el primero de arriba es ***root***. Los dos puntos sirven como delimitador de cada sección, la ***x*** que está al lado hace referencia a la contraseña del usuario root. Y se guarda como x porque la misma se encuentra encriptada.

Para ver la contraseña tendríamos que ver el fichero ***/etc/shadow***, y necesitamos permisos de súper usuario para hacerlo, por lo cual utilizaremos el comando:

sudo cat /etc/shadow

Por ejemplo podemos ver la contraseña de nuestro usuario personal, encriptada.



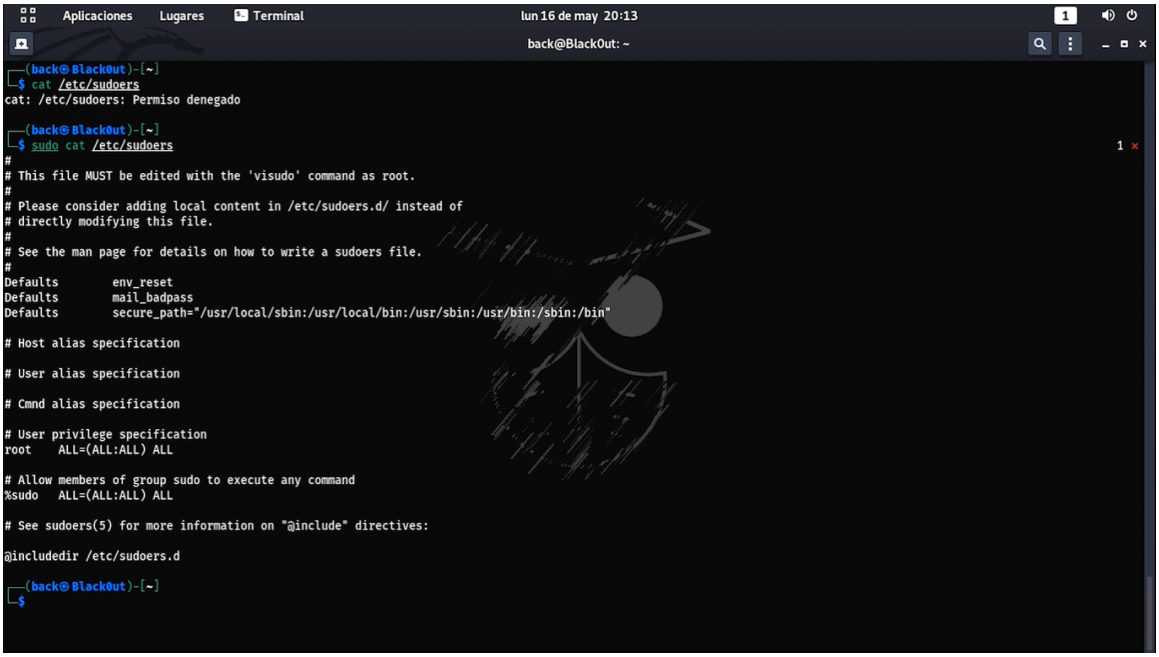
Pero volviendo al /etc/passwd

Seguido tenemos el ***id de cada usuario***, el otro es el ***id del grupo***, posteriormente el *nombre del grupo*, que en el caso de *root* es ***root***. Seguido el ***/home de root*** y luego el ***Shell*** que estas utilizando, que en este caso es ***/bin/zsh*** que es mi interprete de comandos, en el tuyo debe de ser ***bash***. Veremos más sobre estos más adelante.

Pero fijaos que posteriormente existen una serie de usuarios que quizás te sorprenda si solo pensabas que había dos. Pues veras, estos usuarios son los daemons o demonios, que estos usuarios no son para personas, independientemente de que sea posible utilizarlos en algunos casos no es necesario ni tendrías por que.

En realidad esos usuarios se encuentran ahí para correr y administrar ciertos servicios específicos que puedan ser necesarios, por listar un ejemplo, el usuario ***www-data*** como quizás alguno de ustedes ya sepa naturalmente se dedica a moverse detrás de los servidores webs, por decirle de algún modo.

Aprovechando la ocasión vamos a ver un grupo en concreto que es el ***sudoers***. Esto para dar explicación a porque si nosotros no somos el usuario ***root*** podemos ejecutar comandos como ***root*** e incluso convertirnos en root.



Este fichero básicamente sirve para saber cuáles son los usuarios que pueden ejecutar el comando ***sudo***, o en algunos casos específicos también pueden haber usuarios que no pueden convertirse en root pero que pueden utilizar algún que otro comando como si lo fueran.

En este caso nuestro usuario tiene todos los permisos y ya pertenece al grupo sudo, con lo cual ya tiene permisos de administración

Además sobre los grupos podríamos filtrar para saber en qué grupos nos encontramos, esto con ***grep*** seguido del nombre de tu usuario y el ***/etc/group.***

Tuberías y redireccionamiento

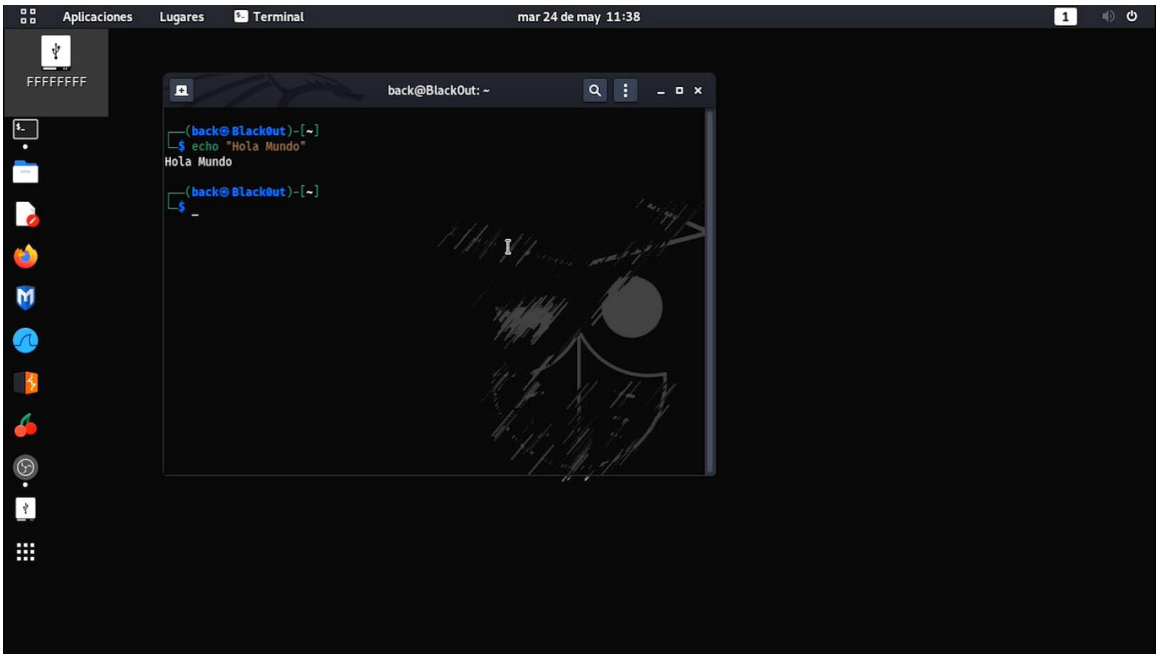
Cada programa que se ejecuta desde la línea de comando tiene tres flujos de datos conectados que sirven como canales de comunicación con el entorno externo. Estas corrientes se definen de la siguiente manera:

STDIN, STDOUT y STDERR

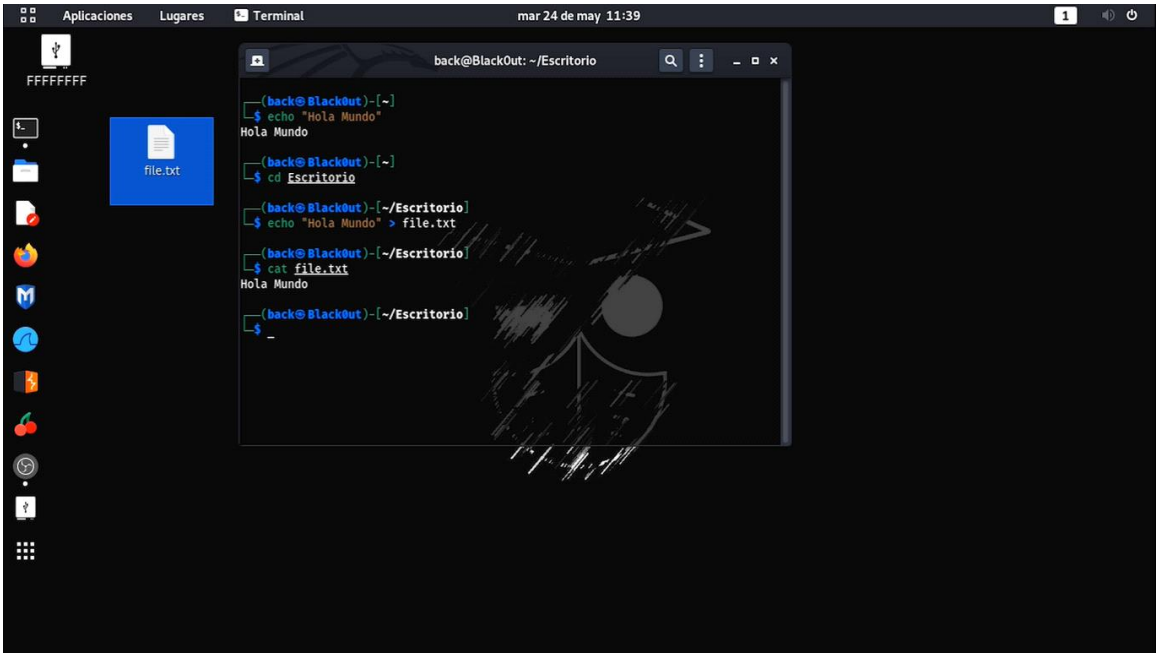
El ***STDI,*** que viene de *standar input* es de donde entran los datos a la ejecución de un programa, por ejemplo por defecto el ***STDIN*** es el teclado, porque claramente por defecto los datos siempre entran por el teclado al nosotros escribirlos. Aunque no tiene por que ser así necesariamente.

Bien, el ***STDOUT***, que viene de *standar output* hace referencia a la salida de los datos, el stdout por defecto se muestra por pantalla, un claro ejemplo de esto podríamos verlo si escribiésemos

echo "Hola Mundo"

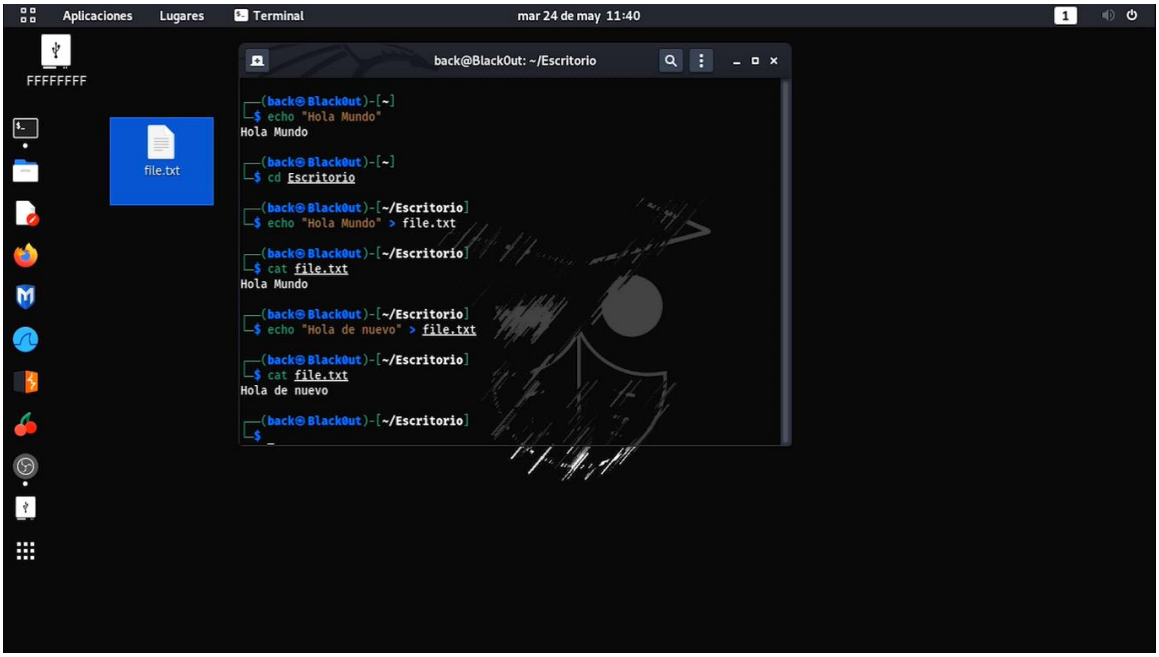


La respuesta que nos da seria la salida de este programa, es decir, el ***STDOUT*** y como ves lo muestra por pantalla. Pero nuevamente te digo, no tiene por que ser así. Podríamos redirigirlo a donde quisiéramos. En este caso vamos a redirigirlo a algún fichero, da igual si lo rediriges a un fichero que existe o no, por ejemplo vamos a volver a ejecutar el programa pero esta vez re direccionar el output a file.txt (aunque no exista)

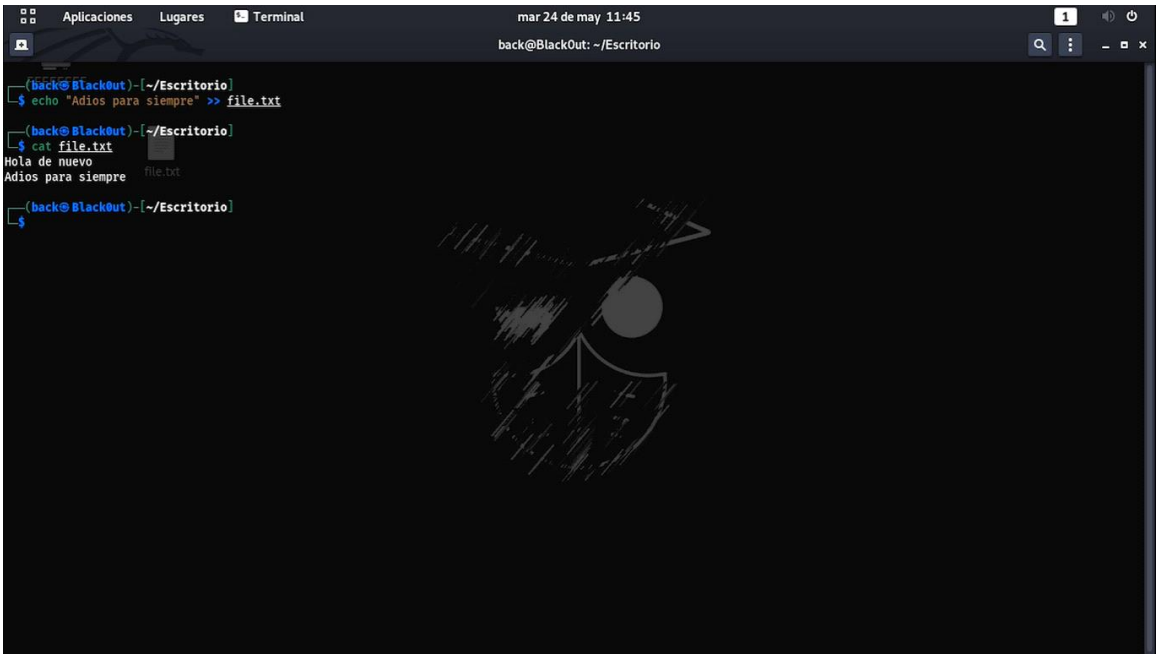


Como ven, no nos muestra nada por pantalla, esto ocurre porque el standar output ha sido redirigido a file.txt, por lo cual si verificamos dicho archivo que antes no existía y ahora se ha creado con que dentro del mismo se encuentra el output del comando anterior.

Este tipo de redirecciones es verdaderamente útil, pero para poder sacarles el máximo provecho debemos de ver otro concepto. Y es que si volvemos a ejecutar algo y enviarlo al fichero, por ejemplo: ***echo "Hola de nuevo" > file.txt***



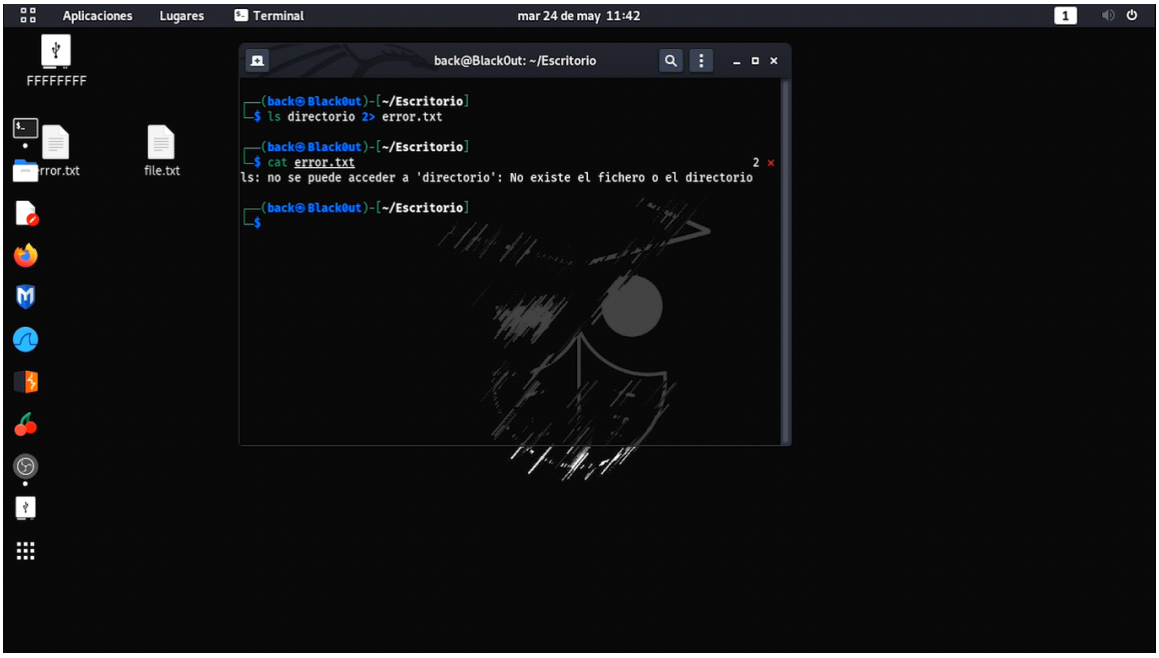
Al ver el archivo podremos notar que al re direccionar el output nuevamente al mismo, a remplazado el contenido original. Quizás en varias ocasiones no queramos que esto ocurra, sino que nos resulte más conveniente ir almacenando todo lo que se envie a ese archivo sin remplazar el contenido original. Para esto utilizaríamos **>>**, es decir:



Podremos notar que ahora conserva el contenido original, y ha guardado el output del programa debajo del contenido existente.

Por ultimo tenemos el ***STDERR*** que son los Mensajes de error, se muestra de manera predeterminada por la pantalla. Aunque para comprender mejor como redirigir los errores debemos de comprender que son las descripciones de salida.

Las descripciones para el ***STDIN STDOUT y STDERR*** se definen como ***0, 1 y 2*** respectivamente. Y estos números son verdaderamente relevantes, ya que se pueden utilizar para manipular los flujos de datos correspondientes desde la línea de comandos mientras se ejecutan o unen diferentes comandos. Para comprender mejor como funcionan vamos a intentar redirigir el ***STDERR***, ósea el s*tandar error* a algún fichero. Básicamente el *standar error* esta presente al mismo tiempo que lo esté algún error en algo que estemos ejecutando, para poner un ejemplo, si hacemos un ls a un directorio inexistente esto nos devuelve un error, porque ese directorio no existe. Y como el indicador para referirse al ***STDERR*** es ***2***, simplemente tendríamos que volver a ejecutar el comando y redirigir el ***STDERR*** a algún fichero, que yo le nombrare como *error.txt*



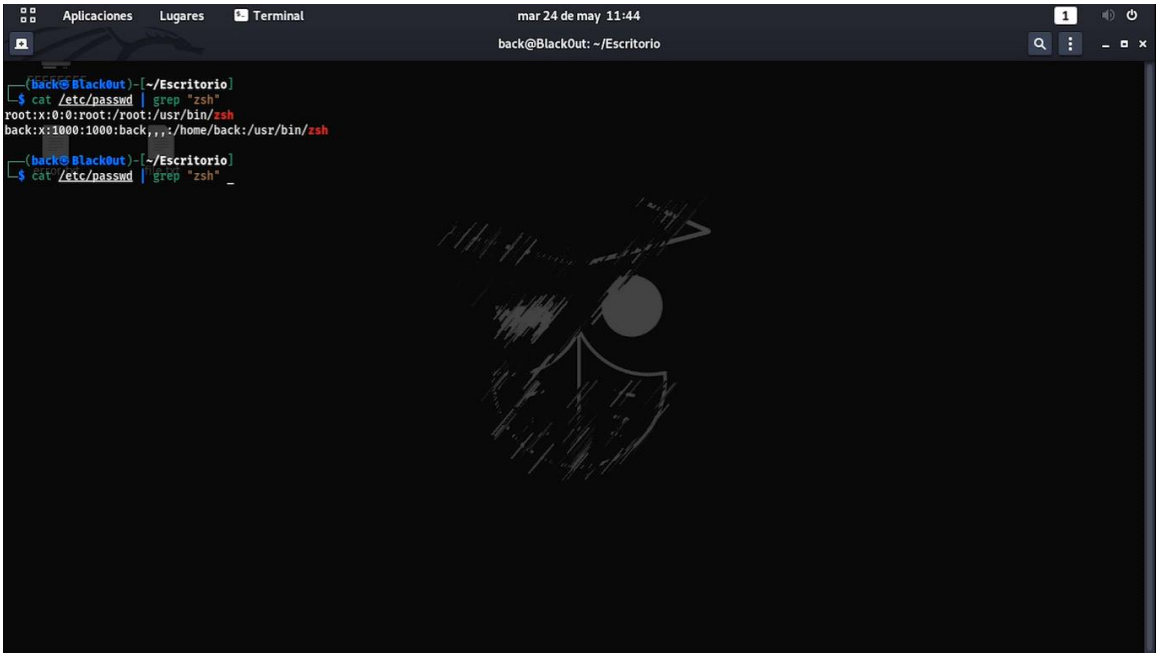
Si os fijáis ahora no se muestran los errores por pantalla, pero sí que se encuentra en el archivo al que hemos redirigido los errores.

También vamos a ver qué cosa son las tuberías. Para ello utilizaremos una barra vertical.

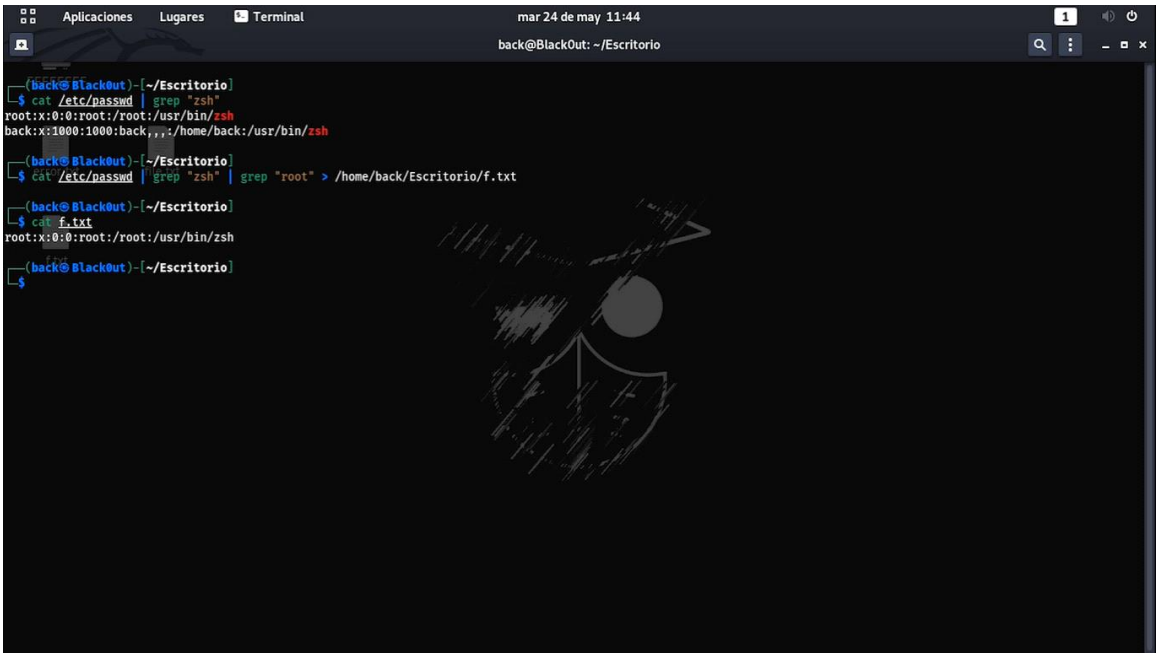
Pues bien una tubería nos sirve para redirigir la salida de un comando a la entrada de otro. Este concepto puede parecer trivial, pero unir diferentes comandos será en el futuro una manera poderosa cuando queramos manipular todo tipo de datos.

Por ejemplo para este caso vamos a ver el fichero /etc/passwd

Como vimos en el capítulo anterior, aquí tenemos todos los usuarios del sistema, pero recordad que en realidad hay usuarios que nos son bastante poco relevantes, por ende vamos a filtrar por los que tengan como *shell* una "***zhs***" en mi caso, que es la *shell* que utilizan por defecto los usuarios de ***Kali Linux***, aunque en tu caso podria ser "***bash***", para ver únicamente los usuarios que nos interesan, que realmente serian ***root***, y nuestro usuario personal.



Es importante que sepas que podemos utilizar tuberías y redirecciones cuantas veces necesitemos a la hora de escribir un comando, por ejemplo podríamos volver a ejecutar el comando anterior y posteriormente a grepear por los dos únicos usuarios, que sería lo que nos mostraría por pantalla, podemos volver a utilizar una tubería para volver a filtrar por root y re direccionar ese output final a un archivo en el escritorio, por poner un ejemplo.



Aunque conste que este ejemplo es bastante inútil, puesto que en realidad bastaría por utilizar el comando ***grep "root" /etc/passwd*** y re direccionarlo, pero bueno, era para que

entendieran el concepto.

Ejercicio 3

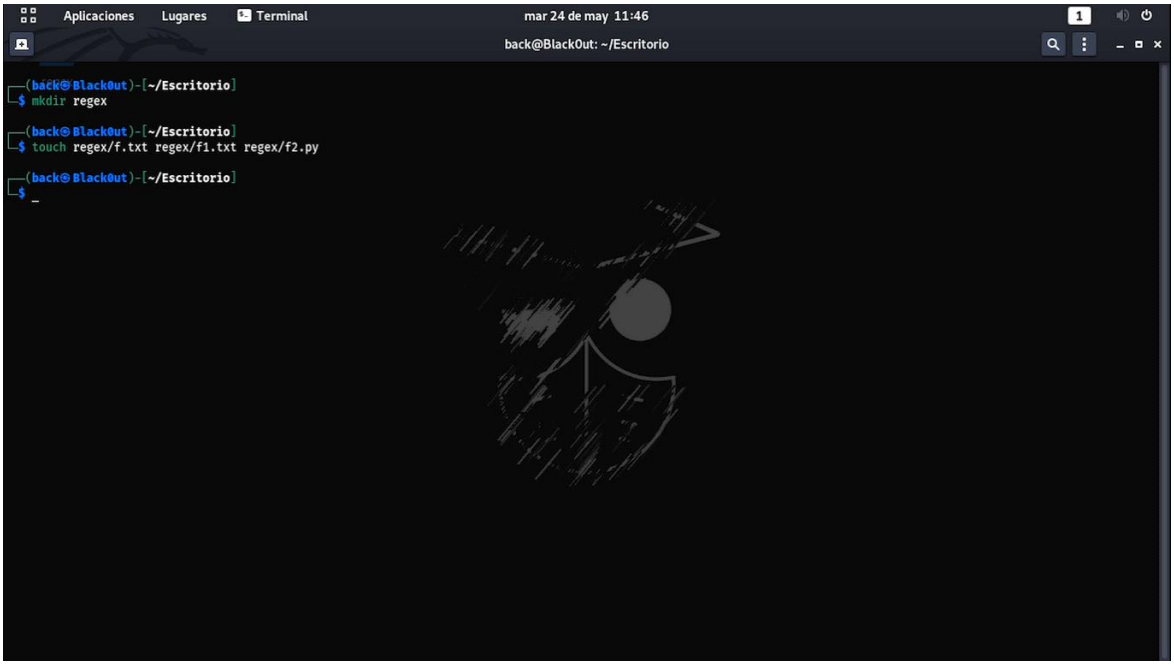
En el capítulo anterior vimos sobre redirecciones y tuberías.

1- Filtre en el */etc/passwd* por su usuario y redireccione el output donde se muestre únicamente su usuario personal a un fichero con un nombre cualquiera en el escritorio.

Expresiones regulares básicas (regex)

Las expresiones regulares describen patrones que engloban un conjunto de secuencias de caracteres similares. En este capítulo voy explicar cuáles son las expresiones regulares básicas y utilizarlas.

Pues bueno, para comenzar a ver sobre las expresiones regulares primeramente vamos a crear un directorio en el mismo escritorio que será, "*regex*" y dentro del mismo vamos a poner 2 o 3 ficheros



Fijaos que la extensión de f2, no es .txt, sino .py que es la extensión de P*ython*.

Para poneros en situación, vamos a suponer que de los ficheros que se encuentran dentro del archivo "*regex*", únicamente necesitamos el que termina en la extensión .py, ósea que necesitamos eliminar los demás.

Con lo que sabemos hasta ahora podríamos hacer un ***rm regex/f1.txt regex/f2.txt***

En este caso estaría bien, pero si se tratasen de más archivos, probablemente sería tozudo ir uno por uno eliminándoles. Con lo cual, sería recomendable automatizar el proceso para seleccionar a través de patrones en cadenas de texto para identificarles y que sorpresa, de esto se encargan las expresiones regulares. En este caso podríamos utilizar la expresión: ***

Básicamente esta expresión regular representa cualquier número de caracteres (incluido 0) en el carácter que le precede. ¿Qué no entiendes ni pitoche? No te preocupes, vamos a verlo de manera práctica.

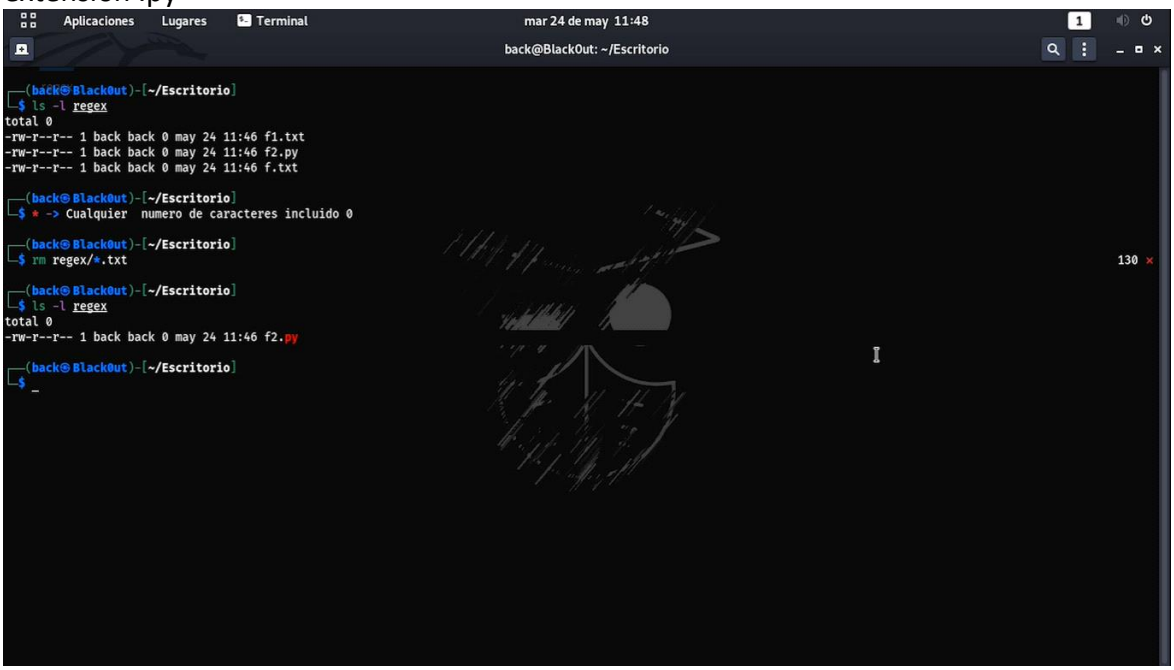
Recordando, queríamos conseguir eliminar los dos ficheros con extensión .txt pero no el que termina en .py

Pues si pusiéramos por ejemplo ***rm regex/**** básicamente eliminaría todo lo que se encuentra dentro del fichero, puesto que como dicha expresión regular simplemente simboliza cualquier cosa independientemente de lo que sea y la cantidad de caracteres que tenga, entonces tendríamos que delimitar mejor que queremos eliminar únicamente los archivos con la extensión .txt

Lo que podríamos hacer seria decirle: ***rm regex/*.txt***

De esta forma vamos a eliminar cualquier cosa que se encuentre dentro de regex, pero que termine en la extensión .txt, ósea que si no termina en .txt no cumple las características del fichero a eliminar que le estamos especificando. Por lo cual podrán apreciar que se han eliminado los ficheros con la extensión .txt, pero no el que tiene la

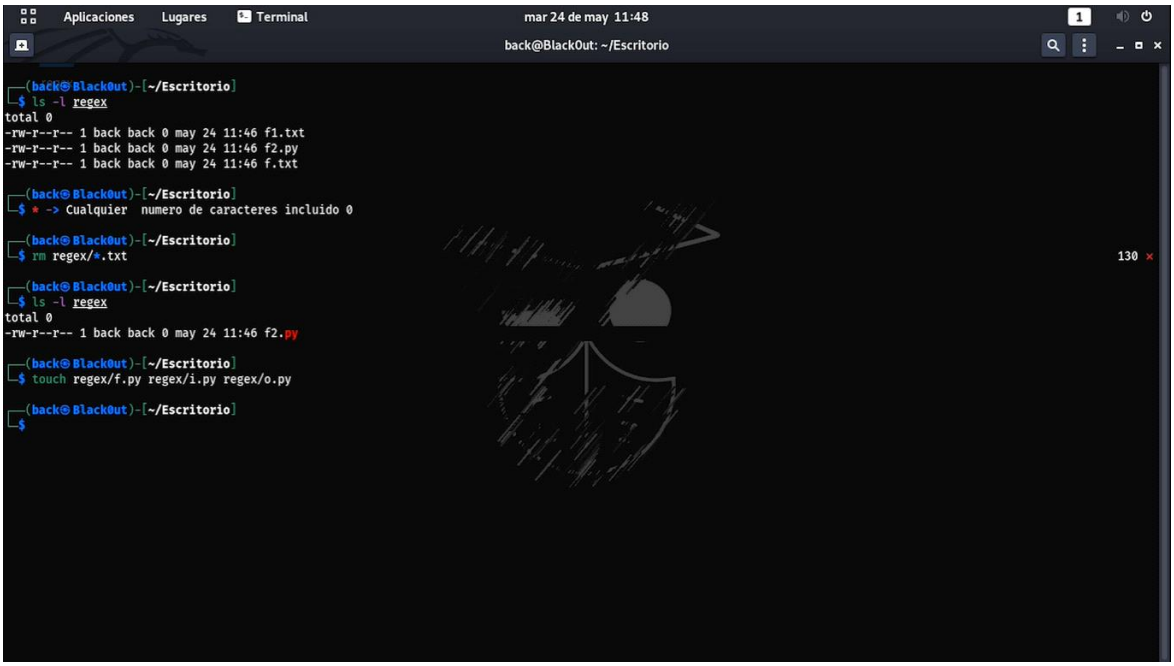
extensión .py



Nuevamente repito y me disculpan la gran mayoría que estoy seguro de que lo entienden.

Las expresiones regulares no son aplicables únicamente al comando ***rm***, sino que podrían utilizarse para copiar, crear ficheros, directorios, para filtrar, para lo que sea. Son características propias de las distribuciones de *GNU/Linux*

Para ver la segunda expresión regular vamos a crear otros archivos con la extensión .py

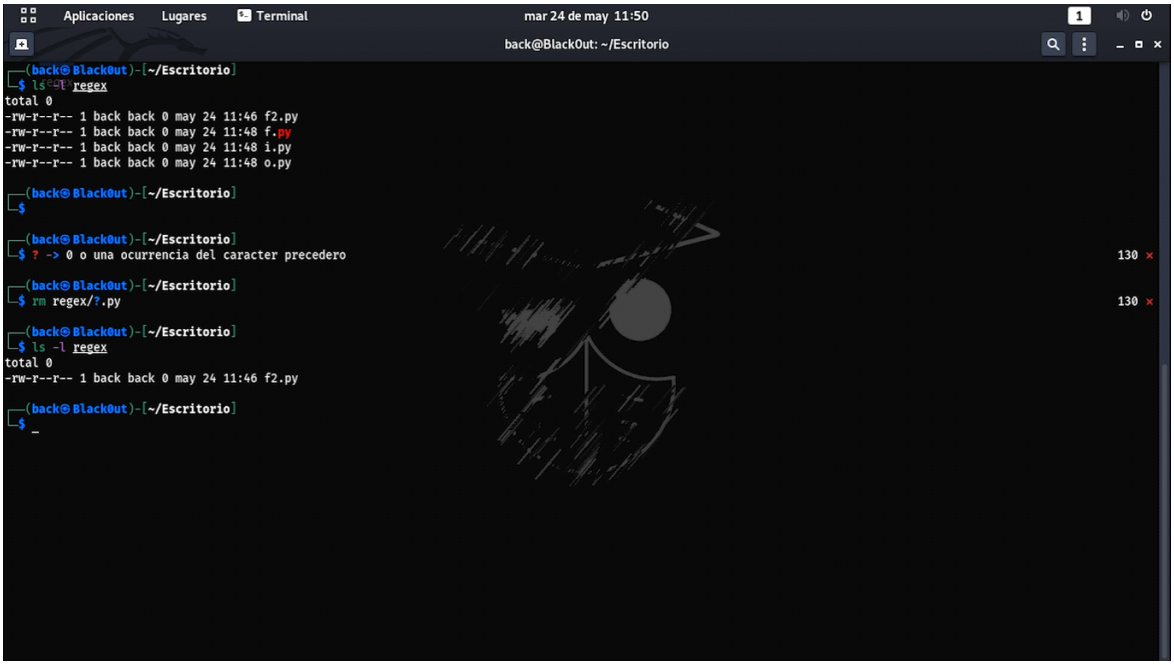


Podemos apreciar que en este caso todos los archivos están compuestos de un carácter, seguido de .py a excepción de f2.py que tiene dos caracteres, seguido de .py

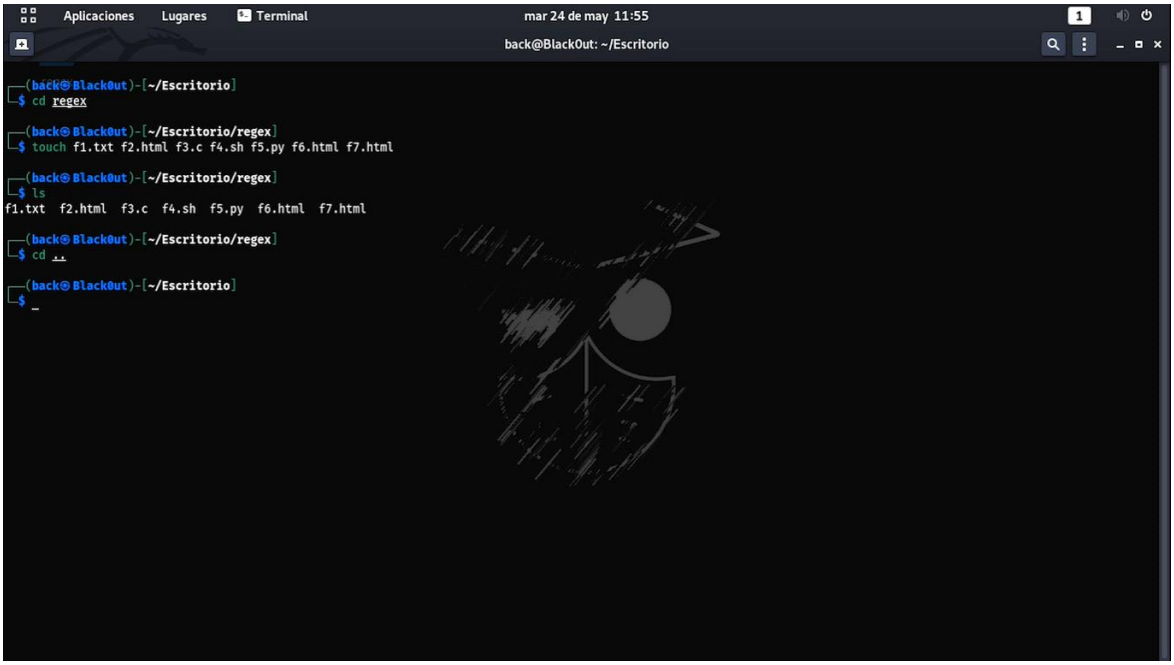
Pues bueno yo el único que quiero conservar es precisamente ese, ósea f2.py, pero la expresión regular vista anteriormente se podría utilizar en este caso, pero como que no es la mejor opción, por lo cual vamos a conocer la expresión regular: **?**
Esta expresión regular representa cero o una ocurrencia del carácter perecedero.

Así que para completar el ejercicio anterior en el cual queríamos únicamente conservar el archivo f2.py podríamos indicarle que queremos eliminar cualquier cosa que comience con un único carácter seguido de .py aprovechándonos de que los demás ficheros únicamente tienen un carácter antes de la extensión.

Si hubiésemos querido conservar todos los demás ficheros menos f2.py también podíamos haberlo hecho utilizando esta expresión regular, simplemente podríamos haberle dicho que borre cualquier cosa que comience con f, seguido de cualquier carácter y la extensión .py, aunque existen otros ejemplos



Para ver la siguiente expresión regular vamos a crear varios ficheros nuevamente.

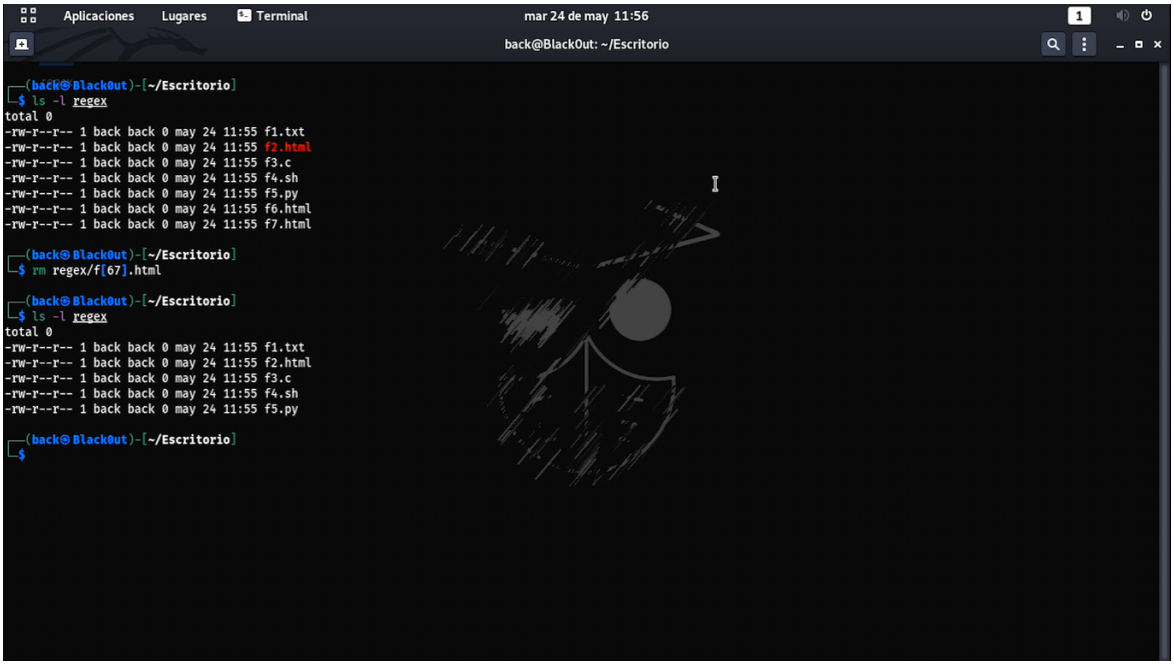


Bien, fijaos en que tenemos 3 ficheros con la extensión .html. Para este ejercicio vamos a intentar eliminar f6.html y f7.html, pero no queremos eliminar f2.html. Con lo cual las expresiones regulares anteriormente vistas no servirían de mucho. Puesto que casi cualquier manera en la que las utilicen terminarían borrando los 3 archivos, a lo mejor si te detienes un momento a pensar encuentres alguna manera de borrarlos y conservar f2.html, pero la verdad es que yo no voy a hacerlo, además, no es la mejor forma de hacerlo.

Lo que podríamos utilizar seria ***[ ]*** esta expresión regular encierra un conjunto de caracteres llamados "*Clase de caracteres*", si ponemos un acento circunflejo que es este de acá: **^** como primer carácter, la clase de caracteres representara todos los caracteres menos los que aparezcan a continuación entre corchetes, también se puede utilizar un guion ***(-)*** para indicar un rango de caracteres de la ***A*** a la ***Z o*** de 0 a 9, es decir, se pueden especificar rangos alfanumericos***.***

Estos tres son los únicos meta caracteres de una clase de caracteres. Todos los demás pierden su significado cuando se encierran en una clase de caracteres.

Por lo cual, como queremos conservar los demas, pero borrar 6.html y 7.html, lo que podríamos hacer seria utilizar el comando: ***rm f[67].html***



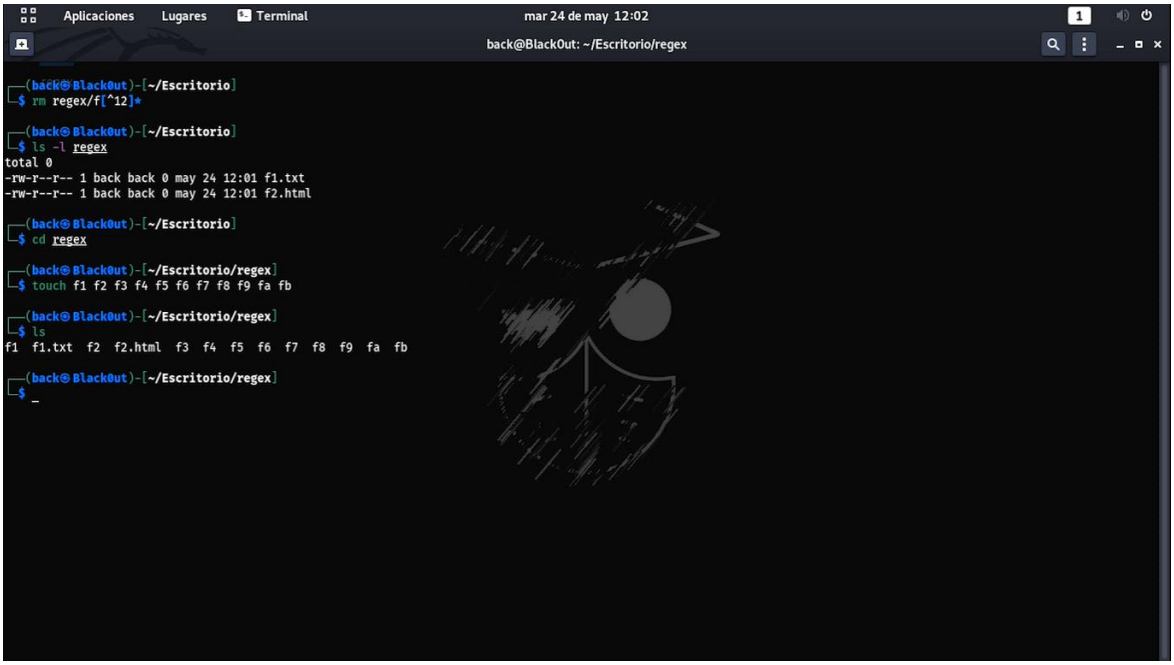
Y borraría todo lo que comience por *f*, seguido de *6 o 7* y luego de *.html*, de esta forma eliminaríamos ambos ficheros dejando que permanezca intacto el *2.html*

También podríamos utilizar esta expresión para indicar lo opuesto, por ejemplo digamos que ahora queremos eliminar todos los ficheros, menos f1.txt. y f2.html

Pues lo que podríamos hacer seria indicarle lo siguiente: ***rm f[^12]****

Y en este caso como f1.txt y f2.html comienzan con f y le siguen con el carácter 1 o 2 seguido de cualquier cosa, mientras los demás ficheros no cumplen dichas características, eliminara todo menos f1.txt y f2.html.

Para ver la siguiente expresión regular vamos a crear diferentes ficheros, por ejemplo



Digamos que queremos borrar todos los ficheros, menos fa y fb.

Bueno, pues podemos hacerlo con rm ***regex/f[123456789]***

Pero si se tratase de 100 ficheros sería un poco más molesto. Con lo cual vamos a utilizar la expresión regular "-" que se utiliza para especificar rangos, con lo cual quedaría mejor conformado de la siguiente manera: ***rm regex/[1-9]***

Del mismo modo podríamos utilizar rangos entre letras, por ejemplo

Igualmente si quisiéramos borrar las mayúsculas y minúsculas podríamos hacerlo con esta misma expresión regular de la siguiente forma: ***rm regex/f[a-zA-Z]***

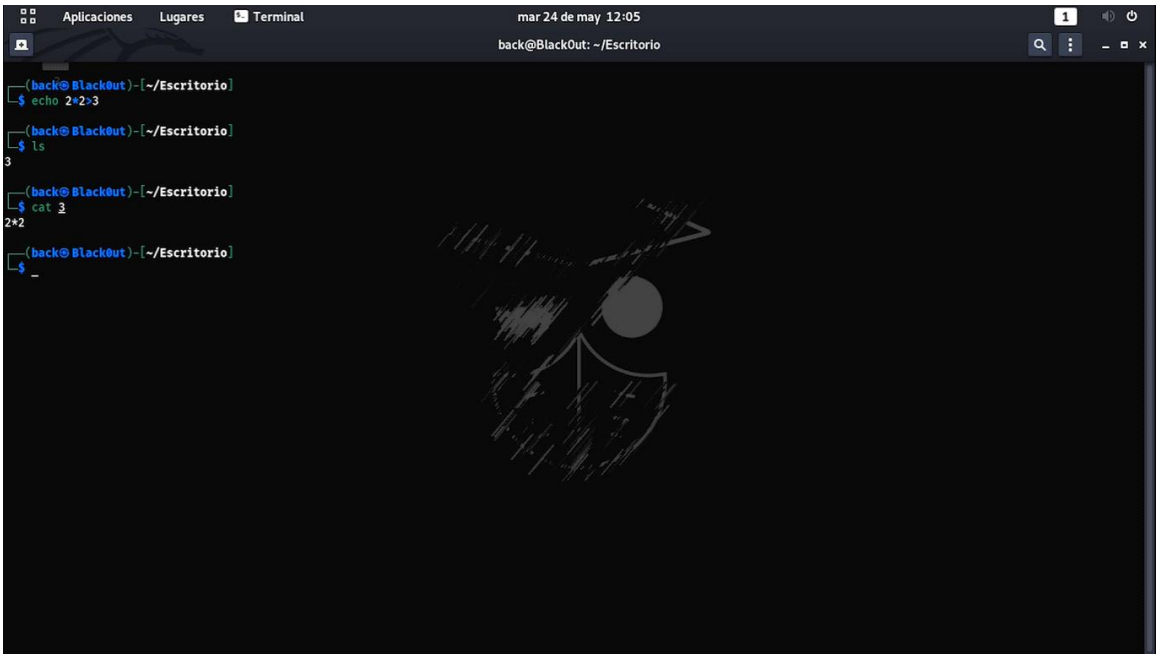
En el caso anterior borraríamos cualquier cosa que comience con f seguido de “a” mayúscula o minúscula y que termine en “z” mayúscula o minúscula.

Expresiones regulares

Expresión	Información
*	Representa cualquier número de ocurrencias (incluyendo cero)
[ ]	Esta expresión encierra un conjunto de caracteres comúnmente llamados "clase de caracteres". Poniendo un acento circunflejo en ella como primer carácter, la clase de caracteres hará representación de todos los caracteres menos los que aparezcan a continuación entre corchetes. Se puede utilizar un guion (-) para indicar un rango de caracteres, como se puede ver en el capítulo del curso sobre expresiones regulares. Para incluir el corchete cerrado ( ]) en esa clase de caracteres, se debe poner como primer carácter: Estos son los únicos meta caracteres de una clase de caracteres. Todos los demás meta caracteres pierden su significado cuando se encierran en una clase de caracteres.
^	Tiene diferentes métodos de uso, por ejemplo combinado con [ ] vimos en el curso que si lo poníamos al comienzo podíamos utilizarlo para hacer exactamente lo opuesto, pero además en sed por ejemplo, este solo es un carácter cuando aparece al final de una expresión regular representando el final de la última línea. En awk por su parte es un meta carácter que representa el final de una línea.
\	Convierte el meta carácter que le sigue en carácter o el carácter que sigue en meta carácter.
$	En awk es un meta carácter que representa un final de línea.
?	Representa cero o un carácter
|	Representa expresiones regulares alternativas
{m,n}	Representa un rango de ocurrencias entre el carácter que le precede "m" y exactamente m ocurrencias. {m,n} por su parte representa entre m y n ocurrencias.
(,)	Permite agrupar expresiones regulares.
.	Representa cualquier carácter excepto el cambio de línea: \n
+	Representa una o más ocurrencias del carácter que le precede.

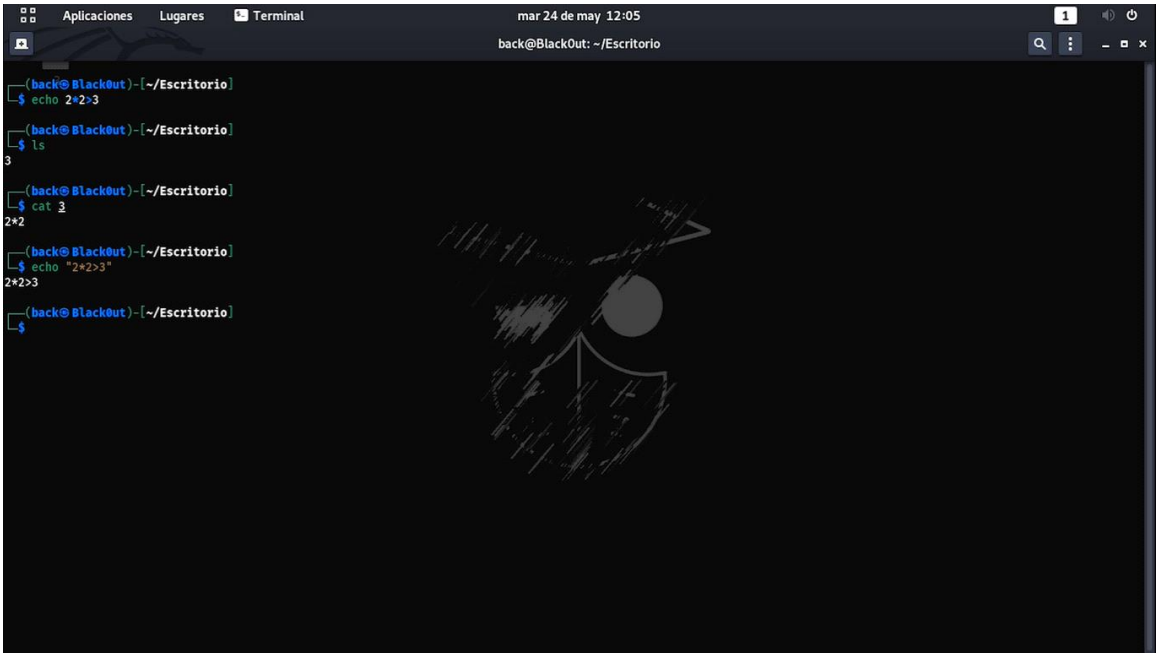
Entrecomillado

Los caracteres especiales que hemos estado utilizando en el capítulo pasado sobre expresiones regulares son bastante importantes. Pero a veces queremos usar alguno de esos caracteres. Ósea, literalmente, es decir sin su significado especial, sino que queremos utilizarlo por ejemplo en una cadena de texto. En este caso sería preciso entrecomillar dicho carácter. Veamos un ejemplo



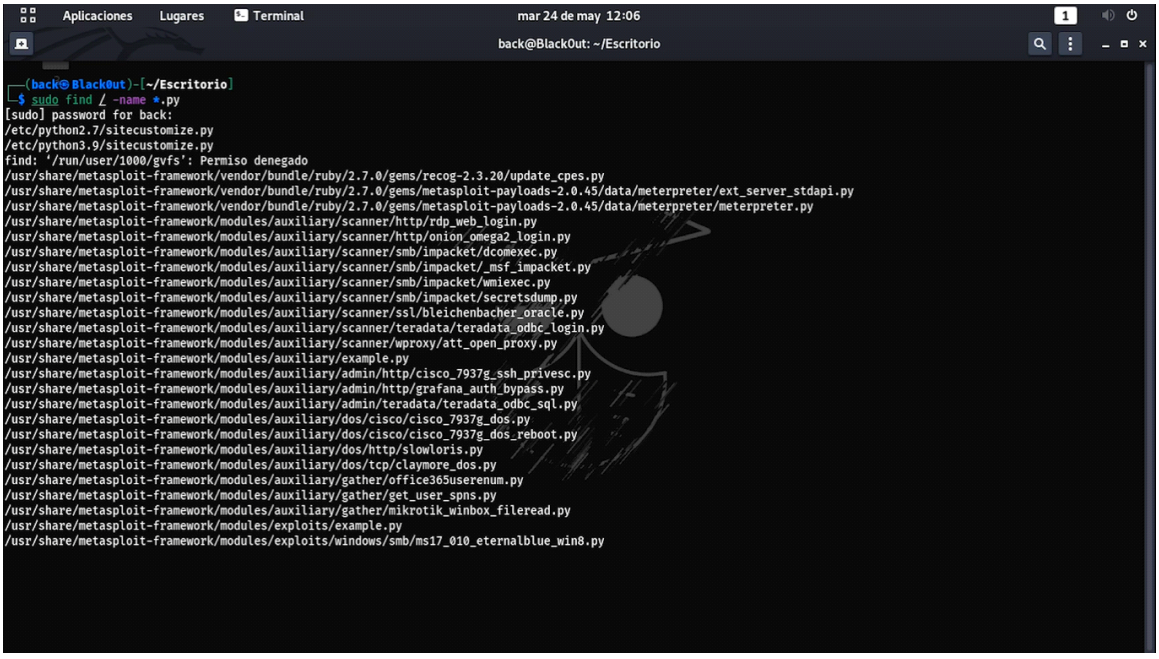
Como vera, en el comando anterior, dos por dos es cuatro, por ende es mayor que tres, sin embargo no da ninguna salida, sino que a creado el archivo 3 con el texto 2*2. Esto lo podemos entender porque hemos visto en capítulos pasados que el carácter de mayor que (**>**) es utilizado para redirigir el output de un programa, lo cual a hecho en este caso, ósea ha impreso 2*2, literal, imagino que con respecto al asterisco “*” haya intentado ejecutar el carácter especial, pero al no encontrar ningún fichero que cumpliera con el patrón no se a expandido y se a pasado el parámetro a echo tal cual, posteriormente redirigiendo el output al archivo 3.

Sin embargo, si entrecomillamos lo mismo que hemos hecho anteriormente con comillas simples. Obtenemos el resultado deseado:



Para ver un ejemplo más práctico podríamos utilizar el comando ***find*** para buscar por ejemplo todos los ficheros que terminen en la extensión *.py* de nuestra maquina con:

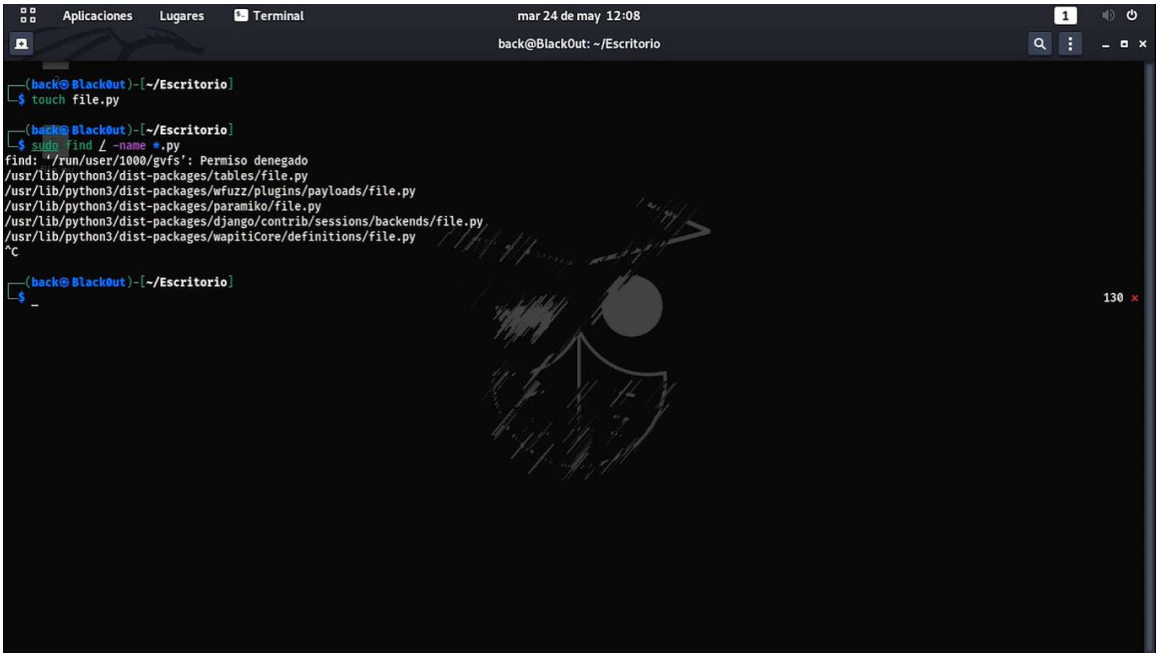
find / -name *.py



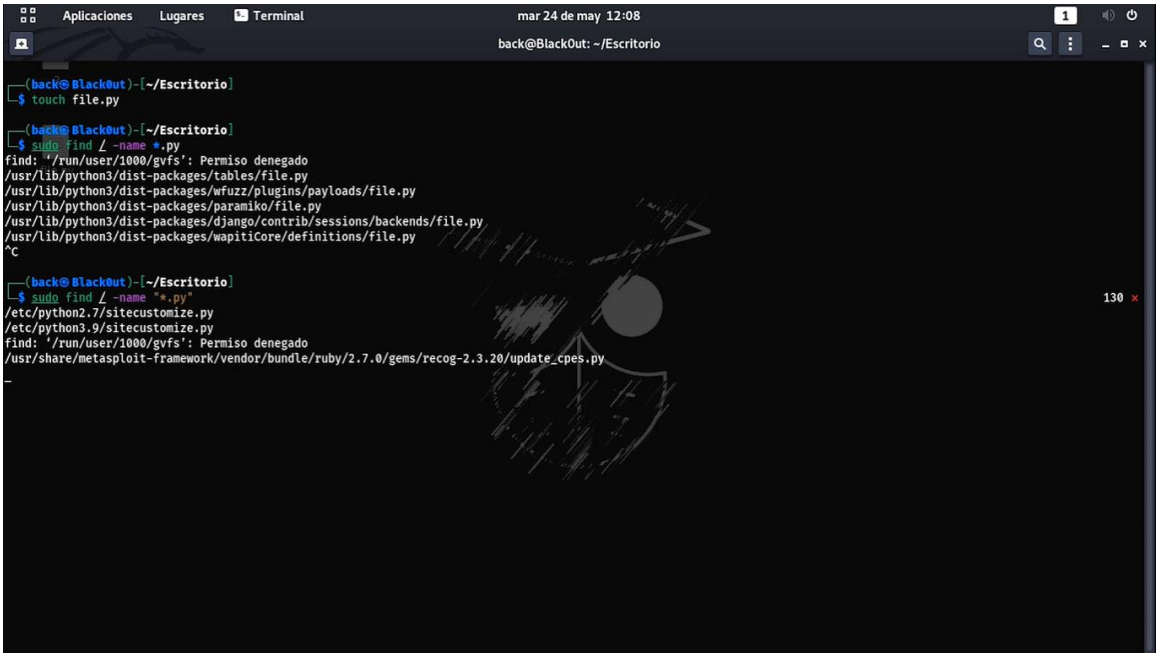
Este comando funciona perfectamente y busca todos los comandos de todo el sistema

que terminen en la extensión *.py*

Pero si somos de estas personas que como que siempre tiene mala suerte puede que dé la coincidencia que en el directorio actual en el que te encuentras exista un archivo con la extensión *.py*. Y como el carácter ***** hace referencia a cualquier cosa, el ***** notara rápidamente que en la ruta actual existe un fichero que cumple con esas características y lo que hará será tomar el nombre de dicho fichero y pasárselo a ***find***, por lo que ***find*** terminaría buscando en todo el sistema el nombre de el fichero que se encuentra en el directorio actual en lugar de buscar por todos los ficheros que terminen en la extensión *.py*, vamos a ver esto en la practica:

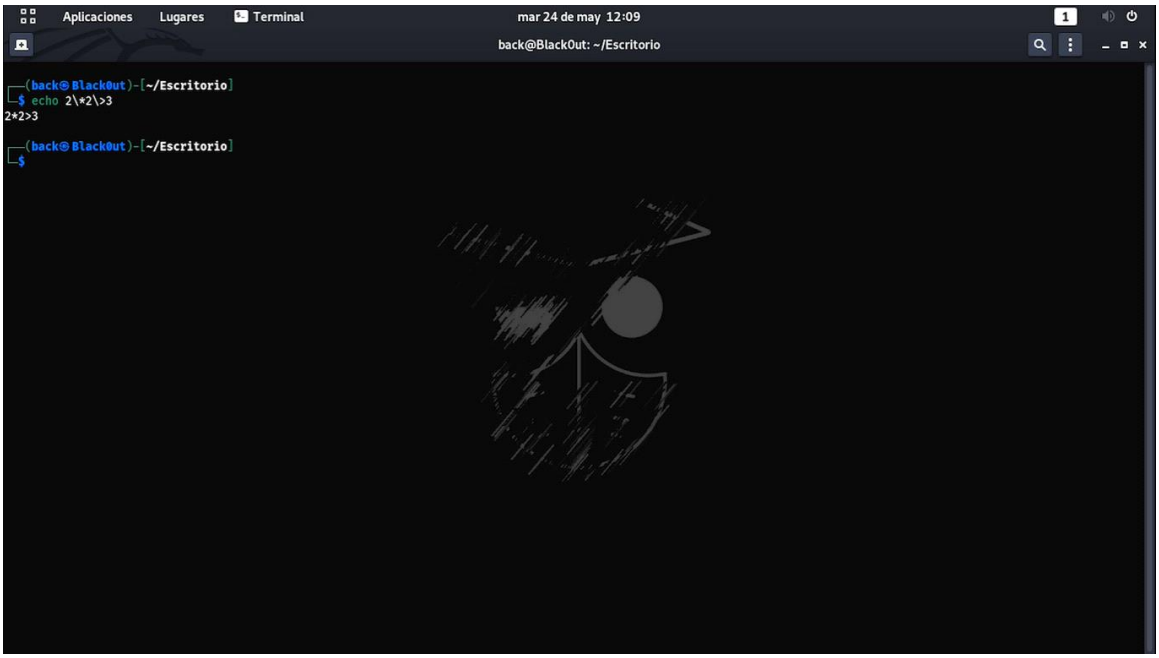


Para solucionar esto simplemente tendríamos que poner entre comillas el parámetro que le estamos pasando a la opción name de ***find***, osea: ***find / -name '*.py'***


Aplicaciones
Lugares
Terminal
mar 24 de may 12:08
back@BlackOut: ~/Escritorio
┌──(back㉿BlackOut)-[~/Escritorio]
└─$ touch file.py
┌──(back㉿BlackOut)-[~/Escritorio]
└─$ sudo find / -name *.py
find: ‘/run/user/1000/gvfs’: Permiso denegado
/usr/lib/python3/dist-packages/tables/file.py
/usr/lib/python3/dist-packages/wfuzz/plugins/payloads/file.py
/usr/lib/python3/dist-packages/paramiko/file.py
/usr/lib/python3/dist-packages/django/contrib/sessions/backends/file.py
/usr/lib/python3/dist-packages/wapitiCore/definitions/file.py
^C
┌──(back㉿BlackOut)-[~/Escritorio]
└─$ sudo find / -name "*.py"
130
/etc/python2.7/sitecustomize.py
/etc/python3.9/sitecustomize.py
find: ‘/run/user/1000/gvfs’: Permiso denegado
/usr/share/metasploit-framework/vendor/bundle/ruby/2.7.0/gems/recog-2.3.20/update_cpes.py

Caracteres de escape

El carácter de escape es **** y podemos utilizar este ejemplo para lo mismo que en el caso anterior, es decir, podríamos haber hecho: ***echo 2*2\>3***

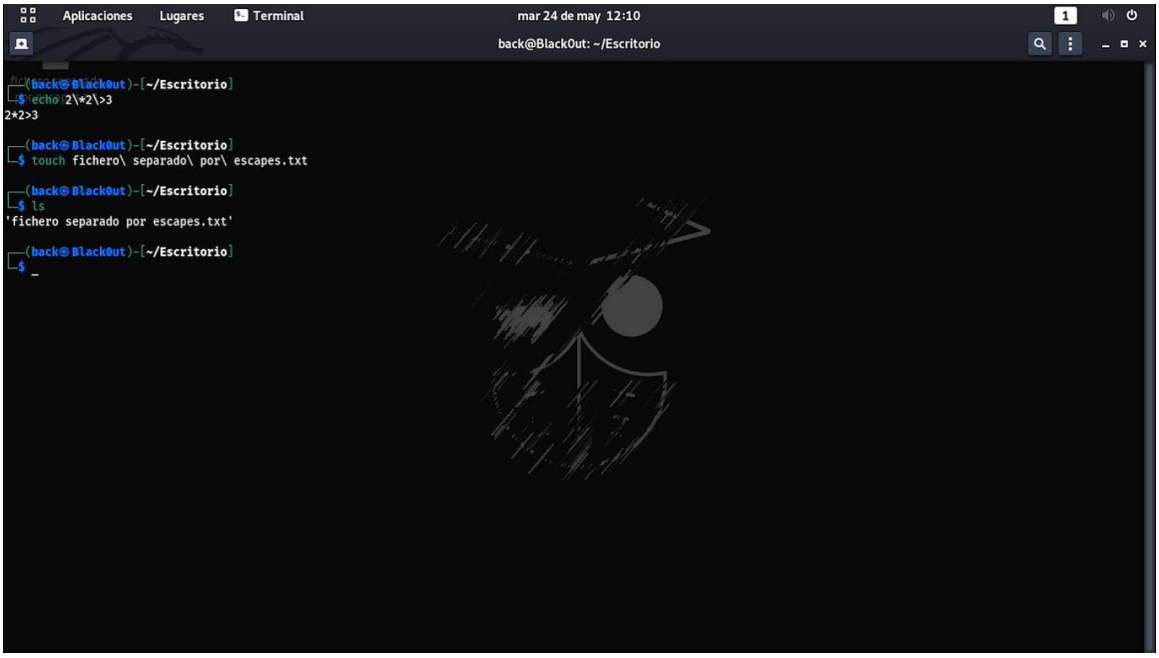


Ósea hemos puesto el carácter de escape a los caracteres especiales cosa que bash no se ponga a interpretarlos.

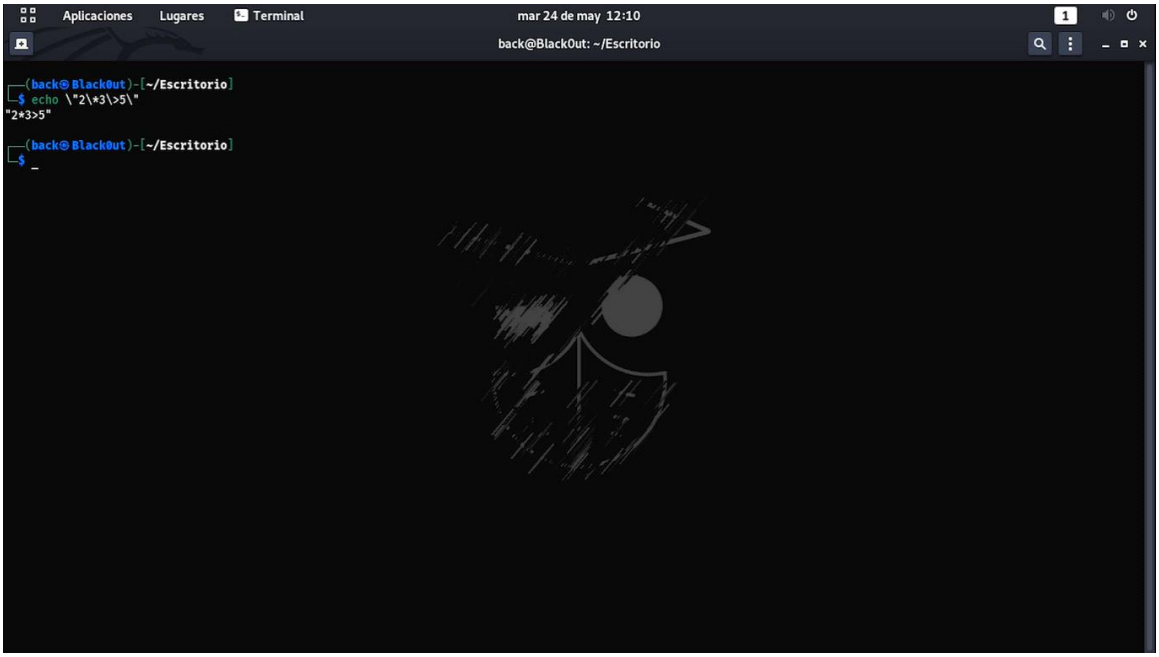
Además, puede que lo sepan, se hayan dado cuenta o no. Pero también podemos utilizar ese carácter para asignar nombres a directorios, ficheros, etcétera ¿Por que?

Pues porque bash interpreta los espacios como un separador de argumentos de la propia línea de comandos. Para que no considere esto podemos poner como carácter perecedero al espacio un caracter de escape: "\"

Un breve ejemplo:



También podemos escapar los entrecomillados dobles o simples para evitar que bash los interprete, ósea podemos escapar lo que sea, por ejemplo:



Y como podemos ver, hemos imprimido hasta las comillas.

Imprimir varias líneas de texto

Otro problema es cómo escribir un comando que ocupe varias líneas. Pues ***bash*** nos permite utilizar el carácter de escape para ignorar los cambios de línea ¿No se los dije?

¡Podemos escaparlo todo!

Para poner un ejemplo:

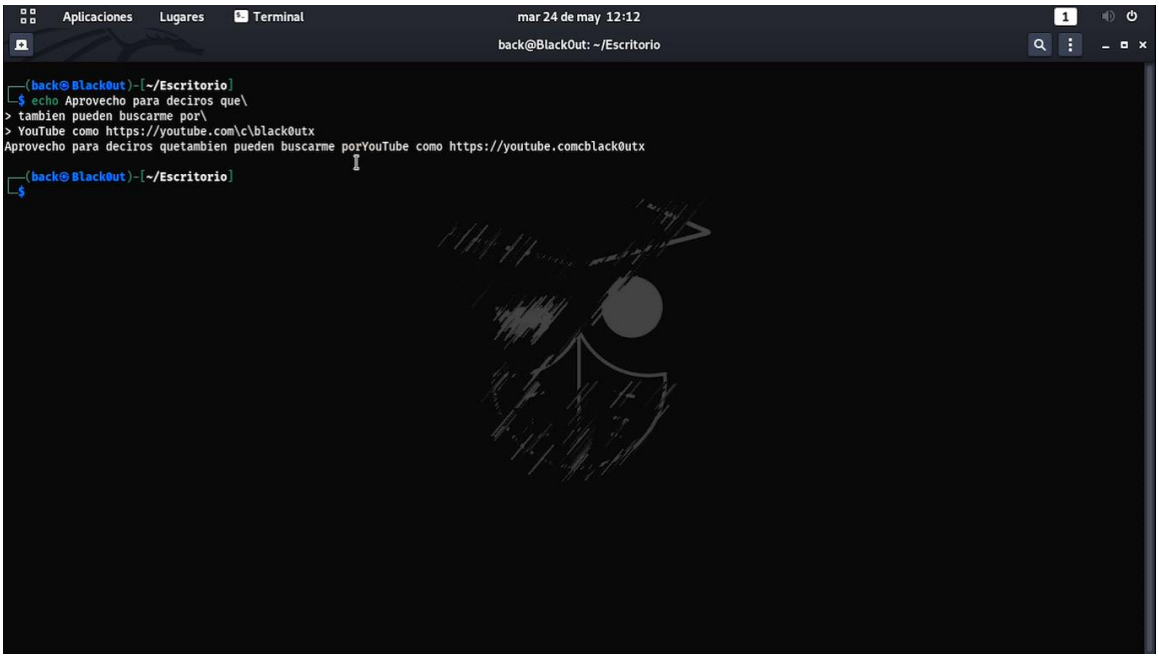
"""

echo Aprovecho para deciros que \

también podeis buscarme por \

youtube como https://youtube.com/c/black0utx

"""



Como veis, a medida que vamos presionando intro, bash nos devuelve el segundo prompt y nos permite seguir escribiendo.

En realidad hay una manera que me gusta más y me parece más simple, aunque explicaba esto de los escapes para que se entienda podemos simplemente poner comillas al comenzar a escribir y simplemente no cerrarlas, si damos intro simplemente podremos

seguir escribiendo en la línea siguiente hasta que no cerremos las comillas, por ejemplo:

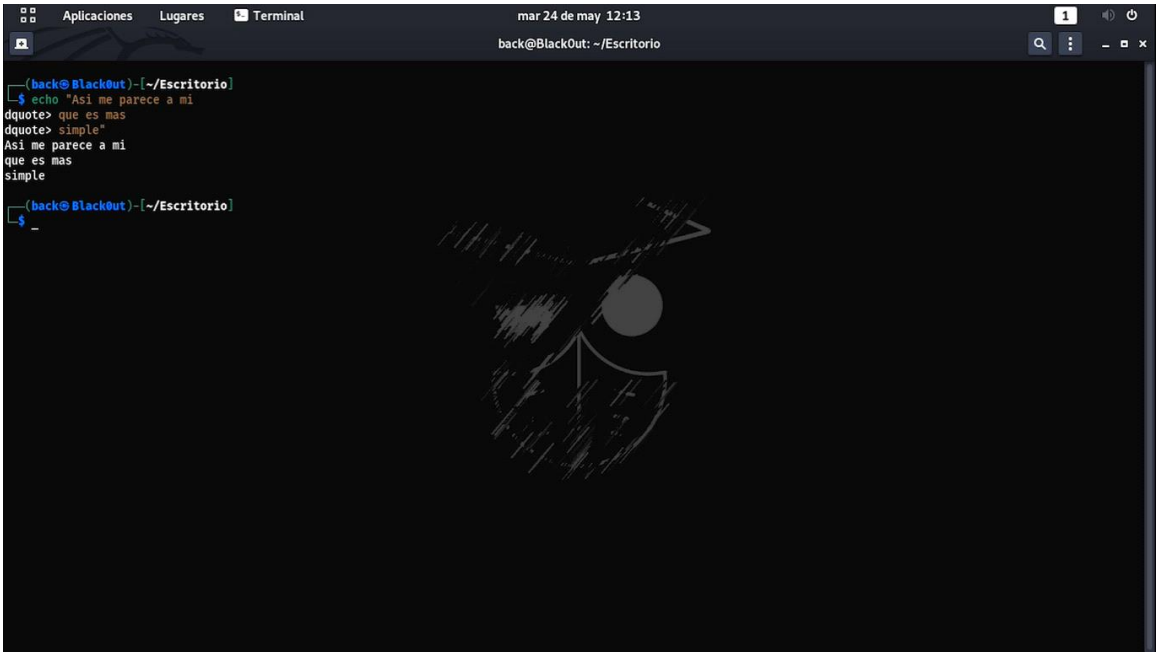
"""

Echo "Así me parece a mi

Que es más

Simple"

"""



Ustedes háganlo como quieran, la única diferencia es que en el primer caso los cambios de línea son eliminados y en el segundo no lo son.

Búsqueda y manipulación de textos

En este capítulo vamos a ganar eficiencia en el manejo de archivos y texto al introducir algunos comandos para realizar estas funciones, como lo será profundizar sobre ***grep***, ***sed, cut y awk.***

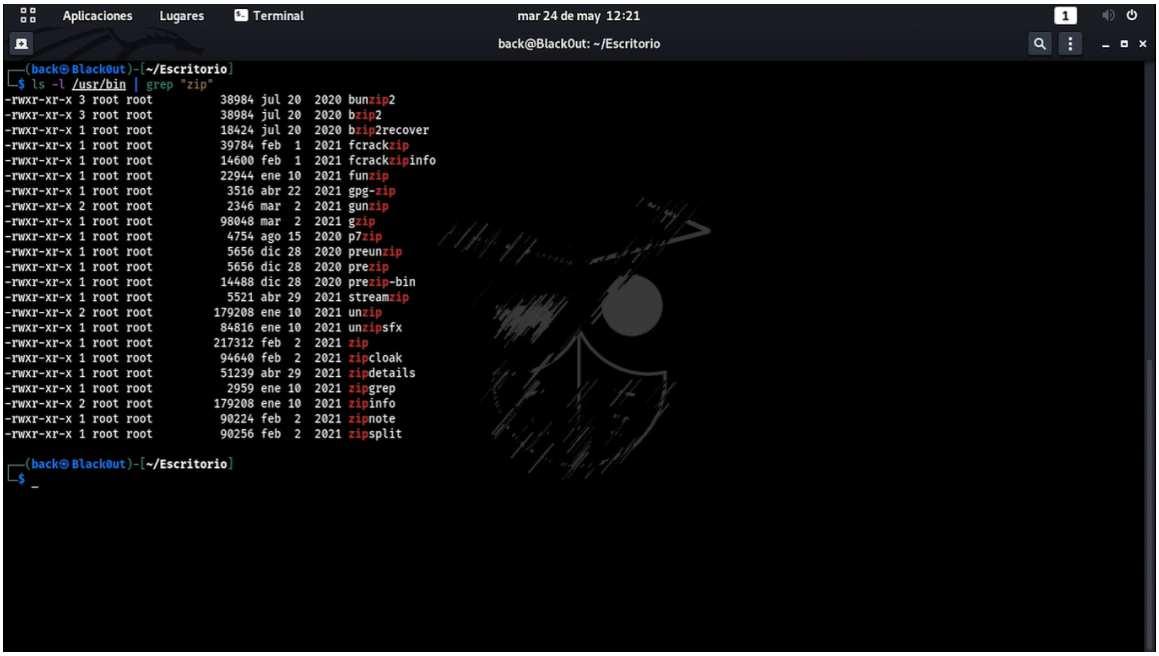
El uso avanzado de algunas de estas herramientas requiere una buena comprensión de *expresiones regulares*. Por lo cual, si no te quedo clara la clase pasada donde veíamos sobre expresiones regulares te sugiero volver a verla o investigar más por tu cuenta, por ejemplo desde los siguientes enlaces:

http://www.rexegg.com/
http://www.regular-expressions.info/

grep

Anteriormente habíamos visto ***grep***, pero esta herramienta esta tan bien y es tan necesaria y potente que no viene mal volver a incluirla aquí. Como hemos visto se encarga de buscar archivos de textos para bajo las expresiones regulares dadas generar cualquier línea que contenga una coincidencia con la salida estándar, que como vimos naturalmente es el terminal.

Pero puesto que la hemos tocado, vamos a profundizar más, por ejemplo sobre las opciones de esta herramienta. Por ejemplo vamos a buscar en el /usr/bin, que recuerden que es donde se almacenan binarios, ósea comandos del sistema y vamos a filtrar por zip, por ejemplo, pero haciendo uso de dos opciones “-r” para búsquedas recursivas y –I para ignorar las mayúsculas y minúsculas.



De hecho habiendo visto el uso práctico de grep, estoy segurísimo de que podrás intuir cuán importante es. Les recomiendo investigar más acerca de ella.

cut

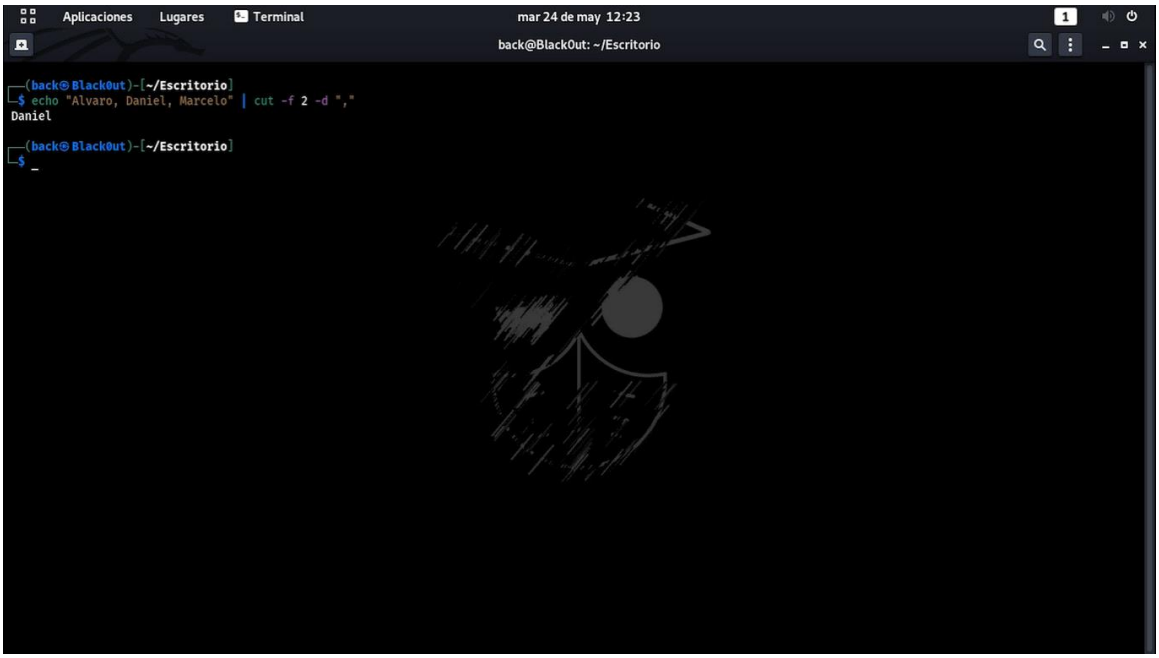
Otra herramienta importante en cuanto a manipulación de texto refiere es la herramienta ***cut***, que viene de cortar. El comando ***cut*** tiene un uso bastante sencillo, pero al mismo tiempo útil. Se utiliza para extraer una sección de texto de una línea y enviarla al output. Vamos a ver un ejemplo práctico, por ejemplo en capítulos anteriores habíamos intentado filtrar por nuestro usuario personal en el /etc/passwd.

En esta ocasión haremos lo mismo pero intentaremos filtrar únicamente por el campo de nuestro usuario.

Con ***cut*** lo que podríamos hacer seria utilizar el parámetro –f para especificar el número del campo que queremos cortar y con –d especificar a raíz de que quiero que delimite dichos campos.

Viendo el /etc/passwd podríamos utilizar “:” como delimitador, y filtrando por el usuario root, que se encuentra en el primer campo, el comando final quedaría conformado de la siguiente forma:

Otro ejemplo sencillo sería el siguiente:

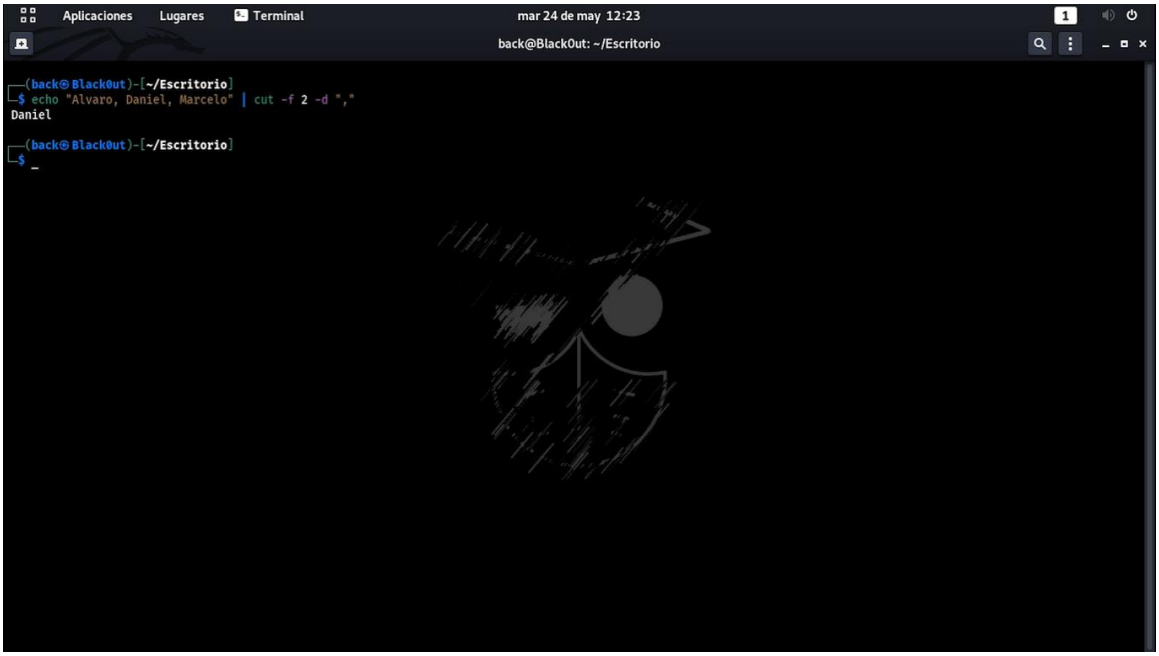


En el que imprimimos tres nombres y posteriormente con cut filtramos por el segundo campo aprovechándonos y utilizando como delimitador para estos campos las comas.

sed

Por otro lado también tenemos sed, un potente editor de transmisiones. Aunque este es más complejo que cut, por ahora nosotros vamos a verlo de una manera bastante

superficial.
A un nivel muy alto, sed realiza la edición de texto en un flujo de texto, ya sea un conjunto de archivos específicos o una salida estándar, es decir al igual que con cut.

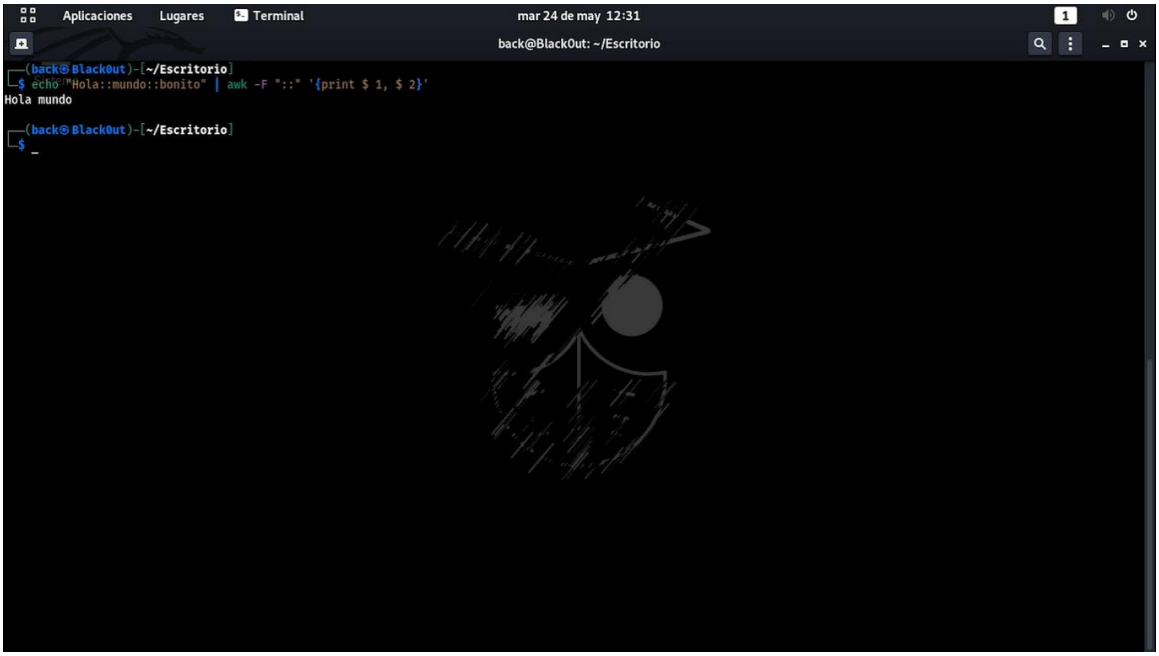


Y si se fijan lo que hemos hecho es remplazar “Ni”, por “Aun”.

awk

AWK es un lenguaje de programación en si mismo, diseñado para el procesamiento de texto y normalmente se utiliza como herramienta de extracción de datos y generación de informes. Bajo ese uso de herramienta, supongo que es de suponer que es en extremo potente, aunque su uso puede resultar complejo si no se profundiza sobre la misma y se practica bastante. En este capítulo solo vamos a verla superficialmente sin profundizar verdaderamente en sus múltiples modos de uso.

Para que vean un ejemplo:



En este caso no queremos ver la palabra bonito, así que de la siguiente forma podríamos indicarle que queremos imprimir solo la primera palabra y la segunda.

Ejercicio 4 Manipulación de textos

Con los conocimientos adquiridos en el capítulo pasado sobre búsqueda y manipulación de textos intente realizar los siguientes ejercicios:

1. Utilizando un solo comando intente ver el archivo /etc/passwd y solo filtrar por el usuario *root*, ósea el output por pantalla debe de ser solamente la palabra "*root*"

2. Nuevamente trabajando sobre el /etc/passwd y en un solo comando realice un procedimiento similar, pero esta vez el output por pantalla a de ser la Shell de root, es decir "/bin/bash, /bin/sh, /bin/zsh" o la Shell que tenga asignada el usuario root.

3. Opcional: Repase volviendo a ver el video o leyendo la página correspondiente, buscando en internet, viendo el manual de ayuda o como usted prefiera los parámetros vistos e incluso los no vistos de herramientas como grep, sed, cut y awk.

Concepto de gestión de procesos

El kernel de Linux, que no explicare a fondo que es. Puesto que esto no es un curso de administración en sistemas Linux. Pero sí que os resumiré que es una de las capas del sistema operativo, en si se podría decir que es el corazón del mismo.

Este se encarga de gestionar multitareas mediante el uso de procesos. El Kernel mantiene información sobre cada proceso para ayudar a mantener las cosas organizadas, y a cada proceso se le asigna un numero llamado **ID** de proceso, o en resumidas cuentas (**PID**)

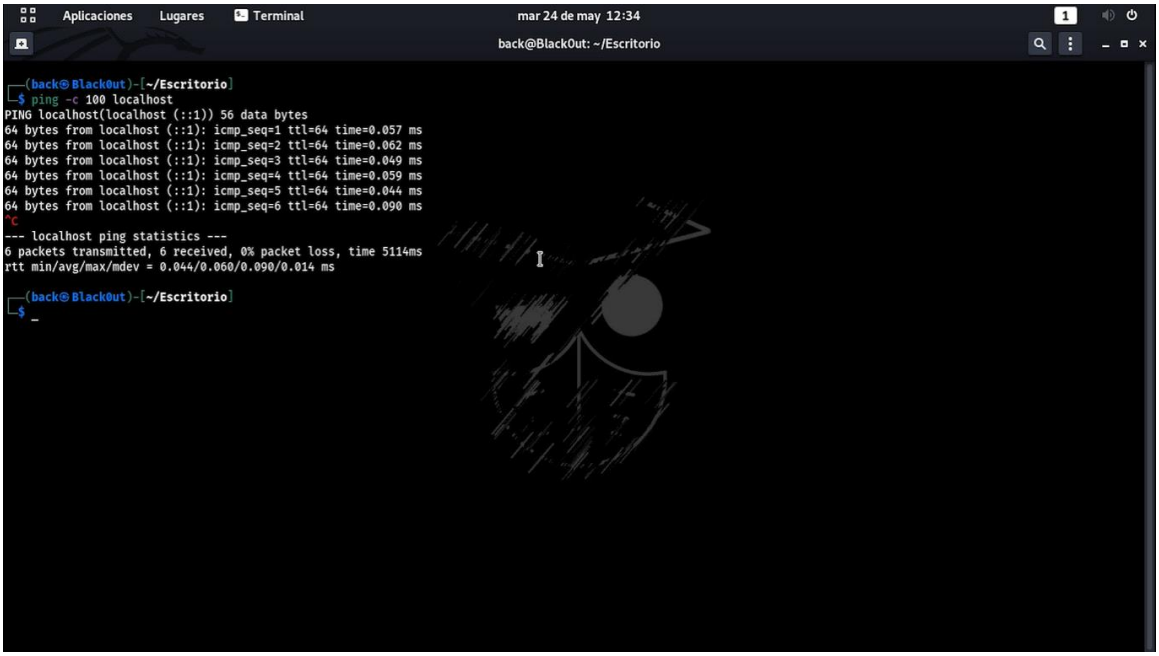
El Shell de Linux también introduce el concepto de “Trabajos” para facilitar el flujo de trabajo del usuario durante una sesión de terminal. El Shell básicamente se encarga de canalizar 1 o más procesos y considerarlos como un solo trabajo.

El control de estos trabajos se refiere a la capacidad de suspender selectivamente la ejecución de trabajos y continuar su ejecución en un momento posterior. Esto se puede lograr con la ayuda de comandos específicos que pronto veremos

Los procesos de fondo

Todos los comandos que hemos utilizado anteriormente en el curso se han ejecutado en un primer plano. Lo que se puede resumir en que el terminal se encuentra ocupado y no se pueden ejecutar otros comandos hasta que finalice la ejecución del actual. Dado que la mayoría de los ejemplos han sido breves y sencillos, esto no ha causado ningún problema. Sin embargo, vamos a ejecutar comandos más largos posteriormente y en la práctica es posible que necesiten hacerlo, pero que estos se ejecuten en segundo plano, puesto que necesiten seguir utilizando el terminal en primera instancia mientras transcurre lo que haga el otro comando por detrás.

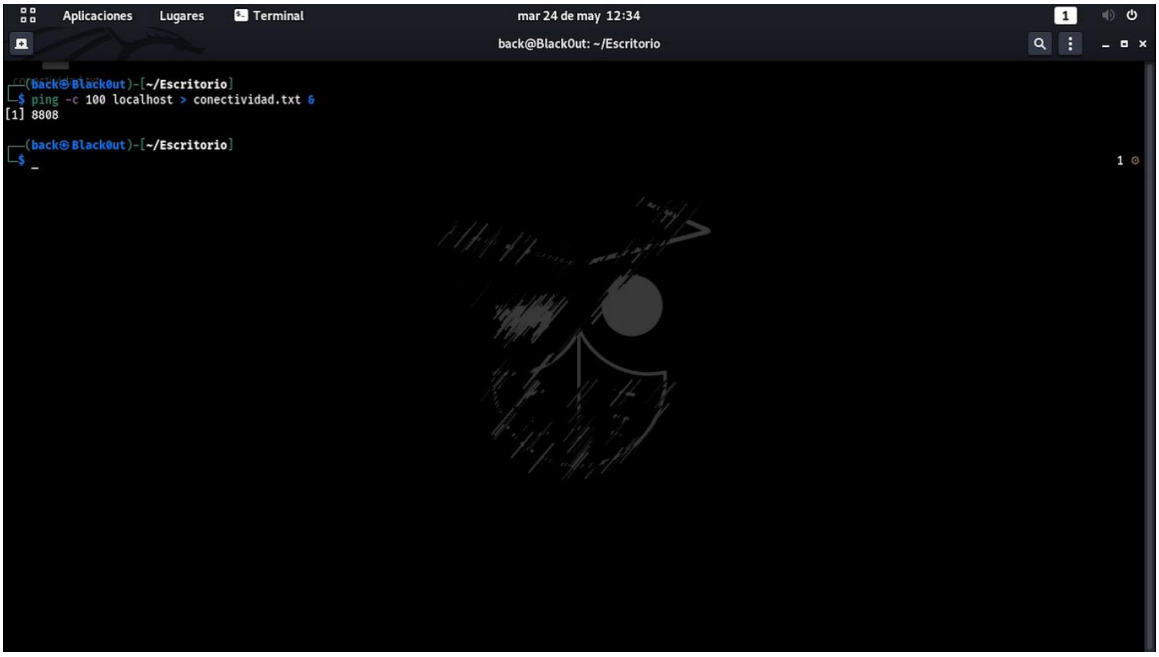
Por ejemplo vamos a enviar un ping, que no son más que meras trazas para verificar si un dispositivo se encuentra activo, en este caso lo enviaremos a localhost, que se resuelve como nuestra propia maquina, para ver que hace. En este caso enviaremos 100 trazas ICMP para verificar la conectividad.



Como podréis apreciar, comienza a enviar las trazas ICMP a nosotros mismos, claramente tenemos Buena conectividad con nosotros mismos, pero no vamos por esa parte del ejemplo. Sino que tenemos que enfocarnos en que tendremos que esperar a que se envíen las 100 trazas ICMP para volver a recuperar el terminal.

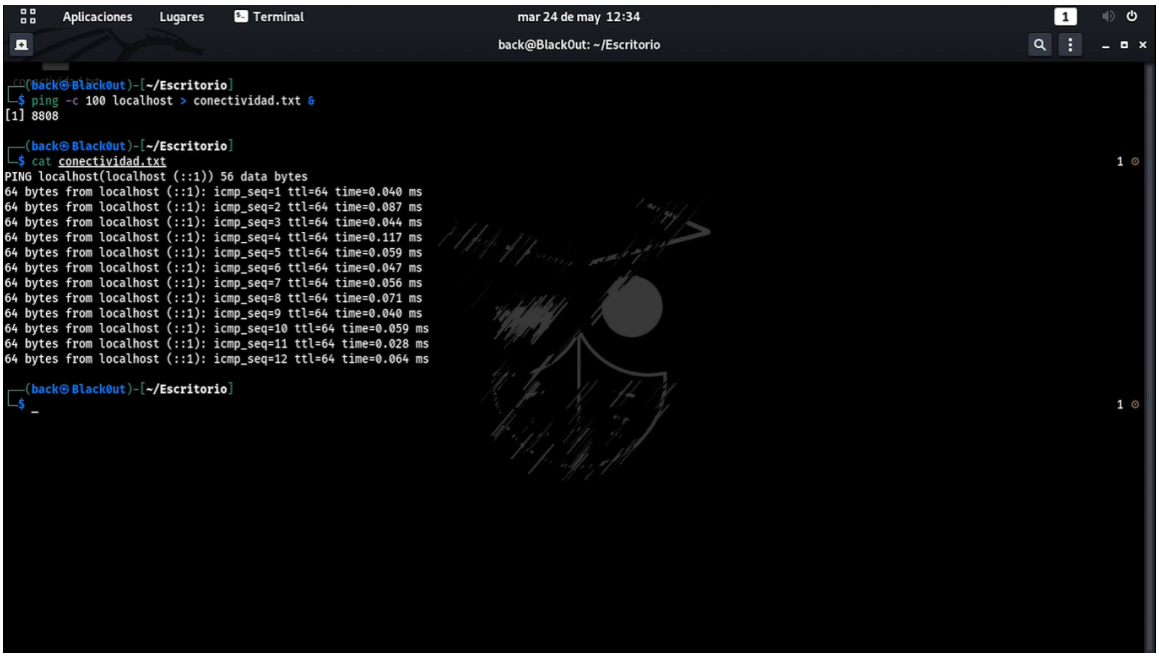
Por lo cual por otra parte podríamos repetir lo mismo pero dejando ese proceso corriendo en segundo plano cosa que podamos continuar utilizando el terminal.

La manera más rápida de crear un proceso que se ejecute en segundo plano es añadir un ampersand al final del comando para enviarlo al segundo plano inmediatamente comience a ejecutarse.

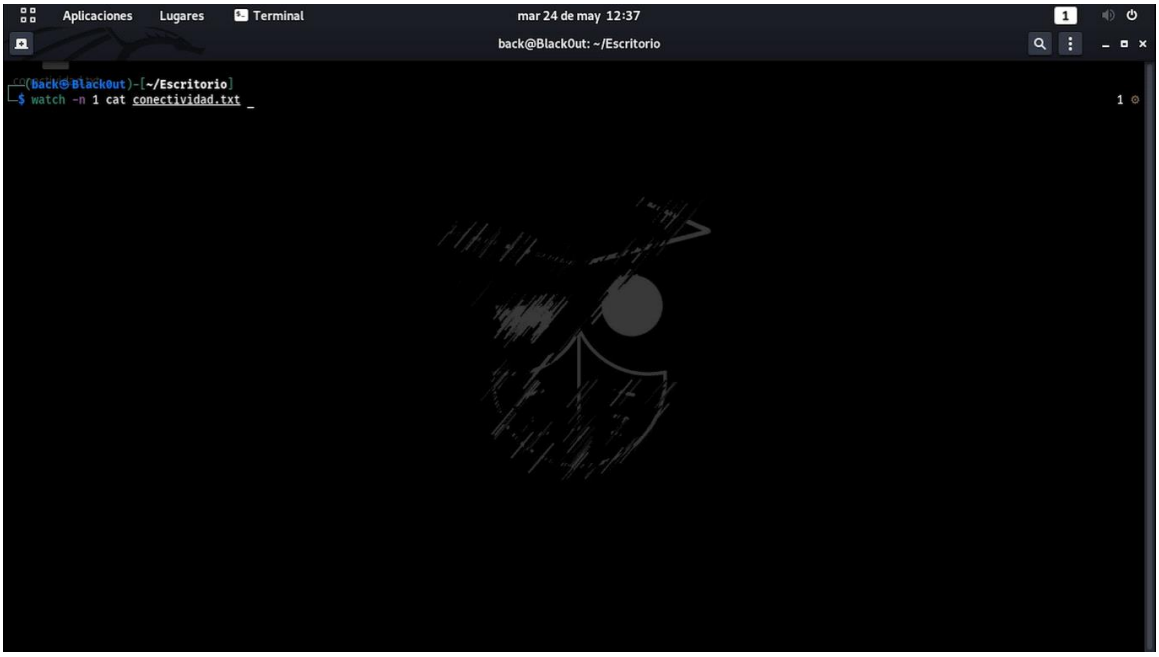


La ejecución se ejecuta automáticamente en segundo plano, dejando el Shell libre para hacer lo que queramos.

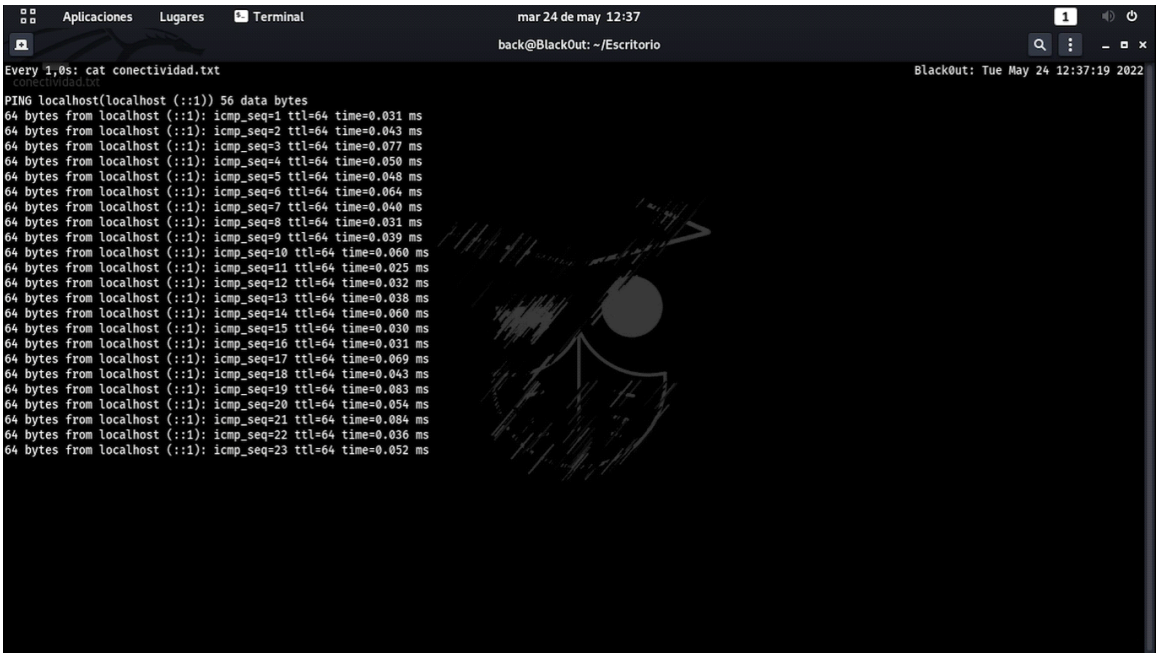
Podemos verificar que aunque no podemos verlo, en el fichero de conectividad se está almacenando todo lo que estamos enviando



Y si utilizamos un nuevo comando, que es el comando ***watch***, que viene de reloj, podemos indicar que queremos ver el contenido de conectividad cada cierto periodo de tiempo, por ejemplo cada 1 segundo de la siguiente forma:

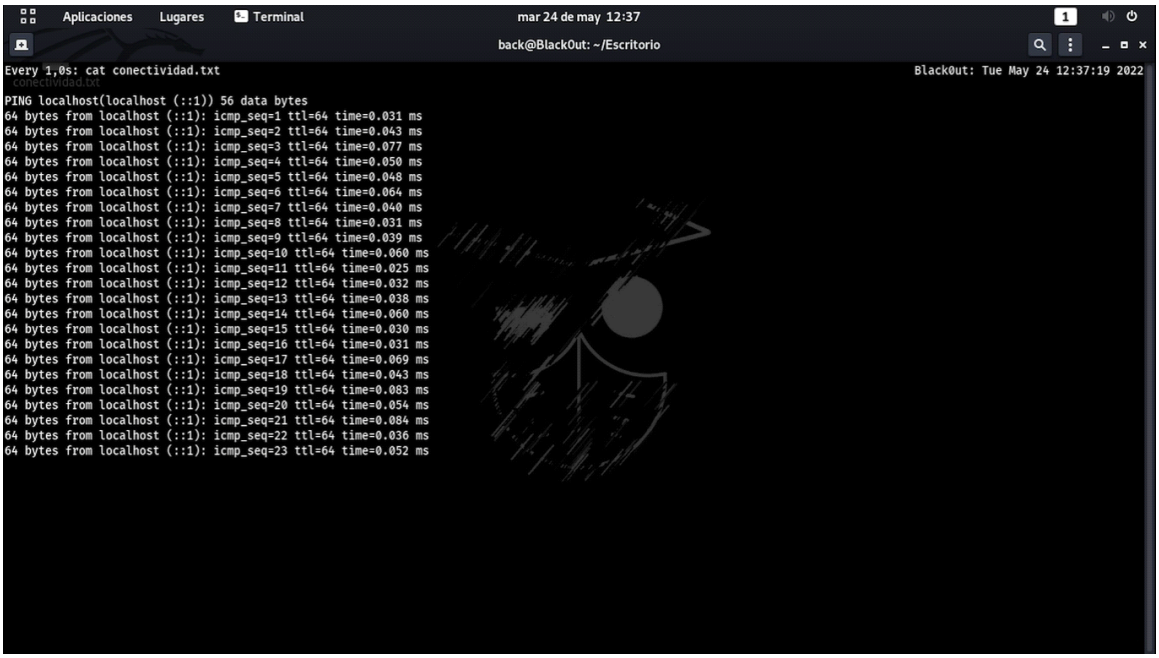


Y podrán notar que cada vez se va incrementando, ósea que por detrás sigue corriendo el comando ping.



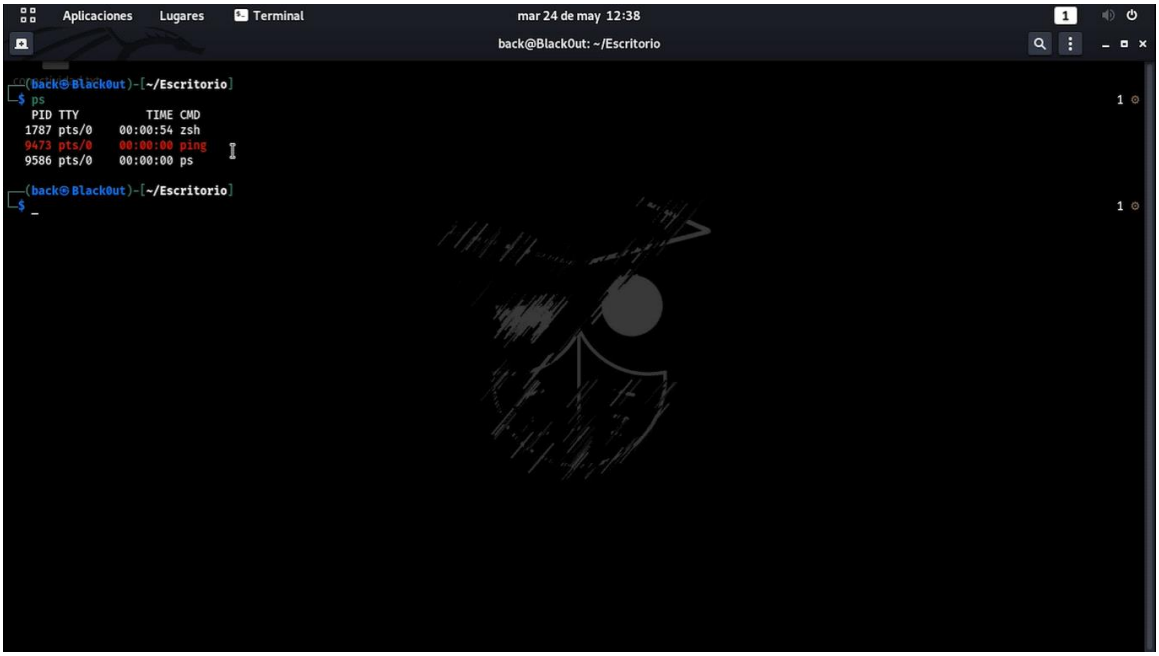
Esto también podríamos verlo con el comando ***ps***, el cual nos mostraría rápidamente los procesos que se están ejecutando en el sistema. Al menos algunos, si quisiéramos ver más

podríamos hacerlo con la combinatoria de parámetros: ps –faux



También podemos matar un proceso con el comando kill. Por ejemplo volvamos a ejecutar ping en segundo plano con ***ping –c 100 localhost > conectividad.txt &***

Verifiquemos en los procesos cual es el ***PID*** de este proceso y matémoslo con el comando *kill*, seguido del numero de proceso, es decir:



Ejercicio Gestión de procesos

1. Inicie desde su terminal el navegador de su preferencia, por ejemplo “Firefox” en segundo plano.
2. Busque el número de proceso del comando Firefox o el navegador que haya utilizado.
3. Termine el proceso del navegador desde el terminal utilizando su PID.

Variables de entorno

Cuando abrimos el emulador de terminal se inicia un nuevo proceso en bash, que este a su vez tiene sus propias variables de entorno. Estas variables son una forma de almacenar globalmente configuraciones heredadas por cualquier aplicación que se está ejecutando durante esa sesión de terminal.

Podemos ver el contenido que almacena una variable de entorno simplemente imprimiéndola por pantalla con el comando echo seguido del carácter de $ que se utiliza en bash para hacer llamado a una variable que ya exista.
Por ejemplo vamos a hacer uso de una variable de entorno que es ***$USER***, la cual hace referencia al usuario con el cual nos encontramos.

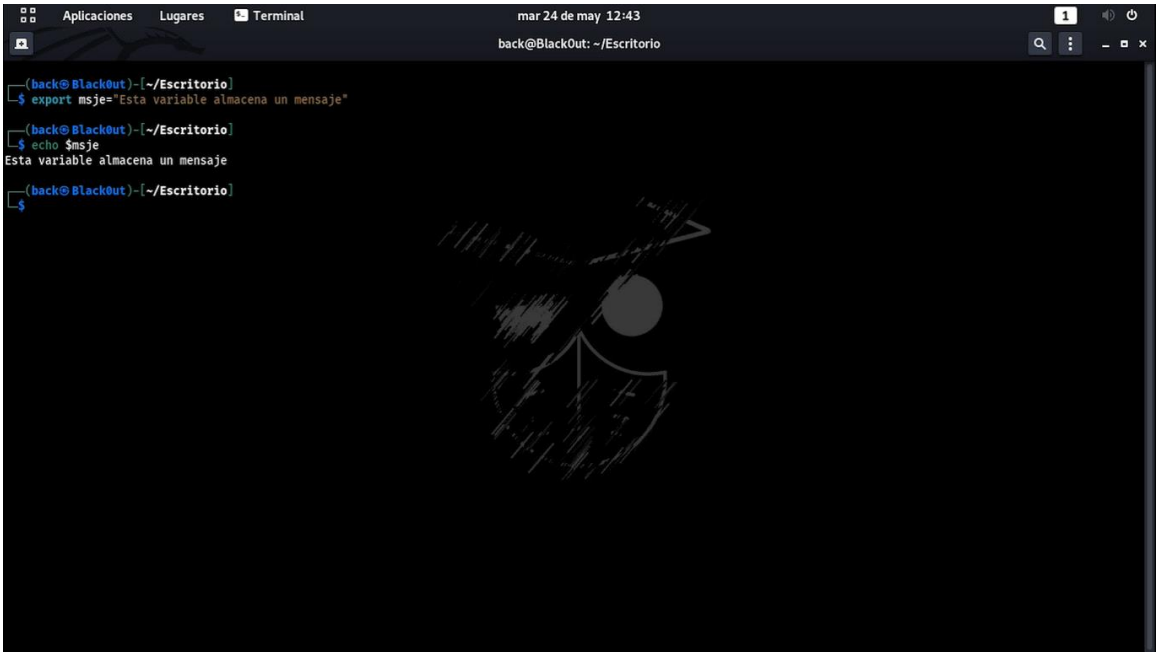
Y como ven nos muestra el usuario, en este caso el valor almacenado en dicha variable de entorno, que viene previamente pre establecida.

Hay otras como: ***$PWD*** y **$HOME**

Existen muchas más variables de entorno y posteriormente en el curso vamos a tocar algunas de ellas, por el momento quédense con que existen.

Pero no simplemente tenemos que limitarnos a conocer o utilizar variables que existen, también podemos definir el valor de una variable, de la que queramos.

Por ejemplo podríamos crear una variable que almacene un mensaje con:



Y presten principal atención en que a la hora de declarar dicha variable no le e pasado el símbolo de dólar. El mismo no se utiliza para asignar valores a variables, sino que solo para hacer uso de las mismas.

Por otra parte con el comando export estamos consiguiendo que dicha variable sea accesible para cualquier subproceso que podamos generar desde la instancia de bash actual. Podríamos asignar una variable sin el comando export, pero solo estaría disponible en el Shell actual.

El Shell que estamos usando

Cualquier distribución de Linux que estés utilizando trae pre instalado por defecto el Shell de bash y ahora vamos a aprender un poco de cosillas sobre esto.

Habíamos hablado anteriormente de las variables de entorno, pero de que no eran las únicas, en este caso vamos a hacer un ***echo*** de la variable ***$SHELL*** para ver que Shell estamos utilizando.

Naturalmente tendrás por defecto bash como tipo de shell que estas utilizando, aunque también podría ser una zsh, una sh, entre otras. Para los siguientes capítulos del curso y para evitar que tengas algún problema te sugiero que utilices bash como tipo de Shell.

Podemos ver donde se encuentra básicamente la shell de bash con :

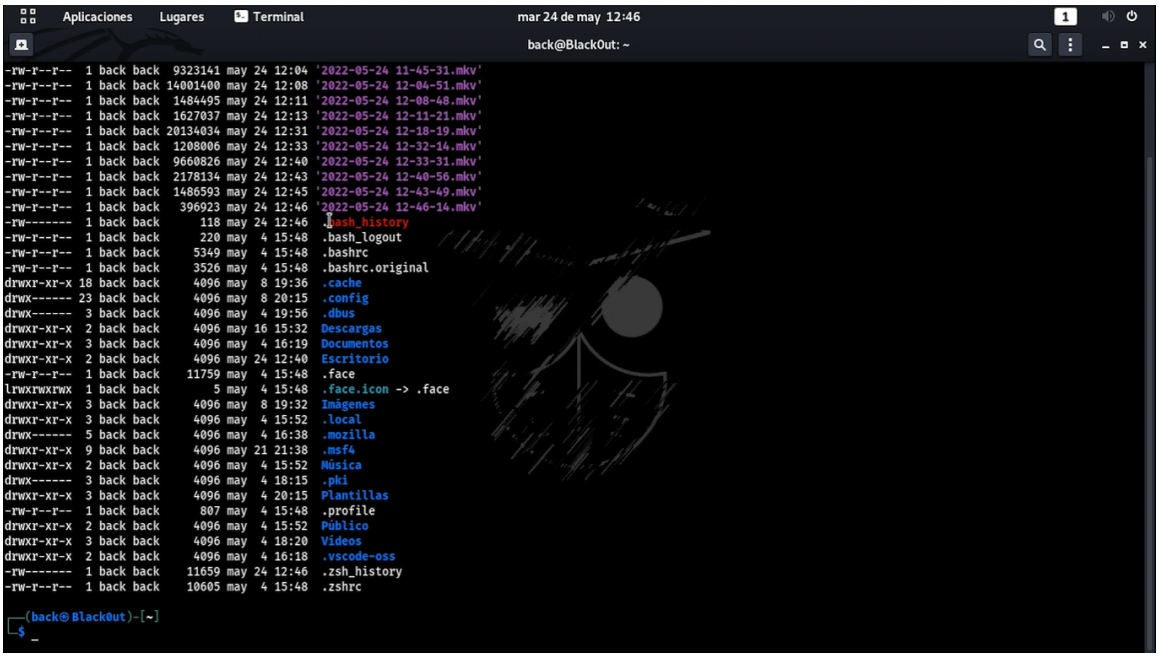
Además podemos ver todos los shells de los que dispone tu sistema si echamos un breve vistazo al archivo /etc/shells

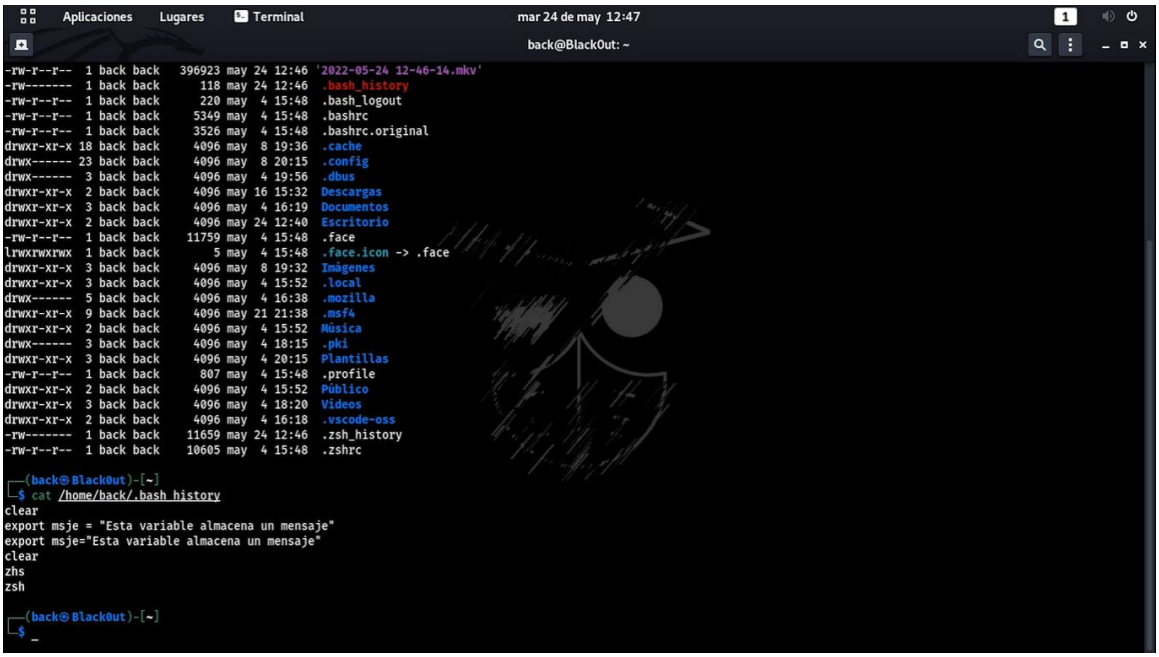
Que por cierto, ya que no lo dije antes, sino tienes bash como tipo de Shell y quisieras utilizarlo bastaría con ejecutar:


Aplicaciones
Lugares
Terminal
mar 24 de may 12:45
back@BlackOut: ~/Escritorio
┌──(back㉿BlackOut)-[~/Escritorio]
└─$ echo $SHELL
/usr/bin/zsh
┌──(back㉿BlackOut)-[~/Escritorio]
└─$ which bash
/usr/bin/bash
┌──(back㉿BlackOut)-[~/Escritorio]
└─$ cat /etc/shells
/etc/shells: valid login shells
/bin/sh
/bin/bash
/usr/bin/bash
/bin/rbash
/usr/bin/rbash
/bin/dash
/usr/bin/dash
/bin/zsh
/usr/bin/zsh
/usr/bin/tmux
/usr/bin/screen
/usr/bin/pwsh
/opt/microsoft/powershell/7/pwsh
┌──(back㉿BlackOut)-[~/Escritorio]
└─$ chsh -s /bin/bash
Contraseña:

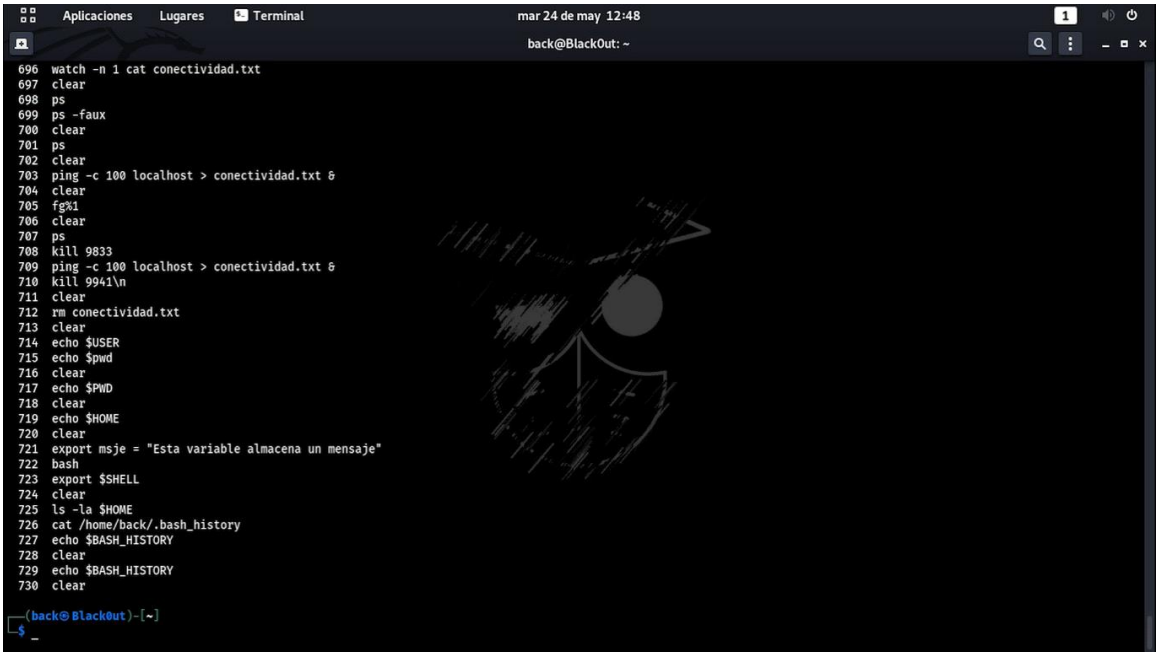
El historial de comandos

En bash existe un historial de comandos que por defecto se guarda en el archivo ***.bash_history*** en nuestro directorio personal de usuario. Este punto que tiene delante básicamente simboliza que el fichero se encuentra oculto, no sé si lo he dicho antes pero con ***ls*** para listar por archivos que se encuentren ocultos tendríamos que utilizar el parámetro "-**a**"





Ademas también podríamos hacer uso del comando ***history*** para optener un listado del historial:



Y sobre el historial de comandos, existen conjuntos de comandos internos de bash que pueden sernos utiles en caso de necesitar ejecutar comandos anteriores, por ejemplo:

Con **!!** podríamos ejecutar el ultimo comando que se haya ejecutado (ver video del curso)

Y posteriormente podríamos utilizar ***$!*** Para referirnos al ouput del comando ejecutado anteriormente, que en este caso es pues todo el contenido de el /etc/passwd, es decir que podríamos hacer algo como

Preparándonos para saltar

En este momento deberías de ser capaz de manejarte bastante bien en el modo interactivo del terminal, al menos con más soltura que antes de ver el curso. Pero les tengo una buena noticia a los frikis del conocimiento y la tecnología, y es que todo lo que hemos hecho hasta ahora a sido fomentar bases muy básicas. Si ya dominas todo lo anteriormente visto estás listo para dar este saldo al mundo del bash scripting, que en realidad comienza ahora.

Entonces desde este punto asumiré que tienes conceptos básicos de Linux. Y aunque no será del todo necesario conocer sobre otros lenguajes de programación para continuar, he de decirte que sería bastante recomendable. Puesto que habrá muchos conceptisismos típicos de la programación de los cuales se habla en todo curso que en realidad, no es que no hablare. Simplemente no profundizare mucho.

Entonces, un consejo antes de continuar. Mantén claro en tu cabeza que bash es un lenguaje críptico, ¿qué quiero decir con esto? Simple, que será recomendable que te fijes bien en la sintaxis de lo que programes, puesto que un pequeño y leve error, tipo un ; que se te olvide poner. Simplemente echarían a perder todo el código. Sobre esto te recomiendo simplemente practicar con los ejercicios que iré dejando. La mejor manera de aprender es practicar.

Scripts

Es importante conocer y saber qué cosa es un script, al fin y al cabo, pues bueno. Un script no va a ser más que un fichero que contendrá dentro, como contenido comandos de bash.

He de decir que para crear scripts debe de estar utilizando un editor de texto preferiblemente cómodo para usted. Por ejemplo, en mi caso para facilitarles la labor a todos estaré utilizando nano en gran medida. Que no es de lo mejor que haya, sino que simplifica la usabilidad de parte de los más vagos de ustedes por si no quieren descargar otro editor de textos o que se yo.

Entonces vamos a crear nuestro primer script, que básicamente lo único que hará será ejecutar un “Hola Mundo”.

Creamos un fichero con el nombre que queramos asignarle al script.

Fijaos que la extensión que le e asignado es .sh, la cual es la correspondiente a bash y tendrán que utilizarla en todos los scripts.

Primeramente fijaos en lo primero que vamos a escribir. Básicamente estamos utilizando el carácter de número, que en este caso en realidad es un comentario. Y que se utiliza para que bash entienda que no queremos que nos ejecute nada de lo que hay seguido del comentario. Y posteriormente: ! y la ruta donde se encuentra bash.

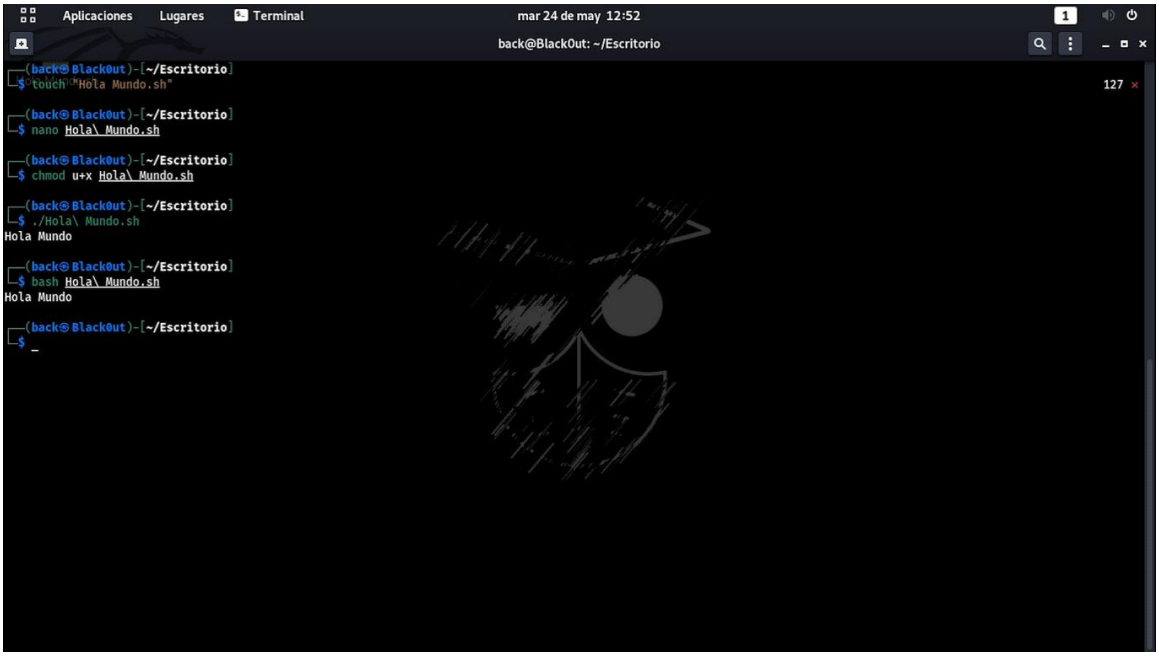
Esto lo hacemos para que al ejecutar el por ahora no, pero pronto ejecutable. El sistema entienda que se trata de un script de bash.

Guardamos el archivo y en este caso como es un ejecutable tendremos que proporcionarle permisos de ejecución, es decir chmod u+x “Hola Mundo.sh”

Posteriormente podríamos ejecutarlo con***: “./Hola Mundo.sh”***

Y nos imprimiría el hola mundo que tenía este, nuestro primer script.

Pero también podemos hacerlo directamente con bash de la siguiente forma:


Aplicaciones
Lugares
Terminal
mar 24 de may 12:52
back@BlackOut: ~/Escritorio
(back㉿BlackOut)-[~/Escritorio]
$ touch "Hola Mundo.sh"
127
(back㉿BlackOut)-[~/Escritorio]
$ nano Hola\ Mundo.sh
(back㉿BlackOut)-[~/Escritorio]
$ chmod u+x Hola\ Mundo.sh
(back㉿BlackOut)-[~/Escritorio]
$ ./Hola\ Mundo.sh
Hola Mundo
(back㉿BlackOut)-[~/Escritorio]
$ bash Hola\ Mundo.sh
Hola Mundo
(back㉿BlackOut)-[~/Escritorio]
$ _

Funciones

En bash, las funciones a diferencia de los scripts se ejecutan dentro de la memoria del mismo proceso de bash, esto se resumiría meramente en que son más eficientes que ejecutar scripts aparte. Aunque tienen un inconveniente poco interesante. Y es que están siempre cargadas en la memoria del proceso de bash. Actualmente debido a la cantidad de memoria que tienen los ordenadores, el tener funciones cargadas en la memoria de bash tiene un coste insignificante. Así que no te preocupes, simplemente lo digo para que lo conozcas.

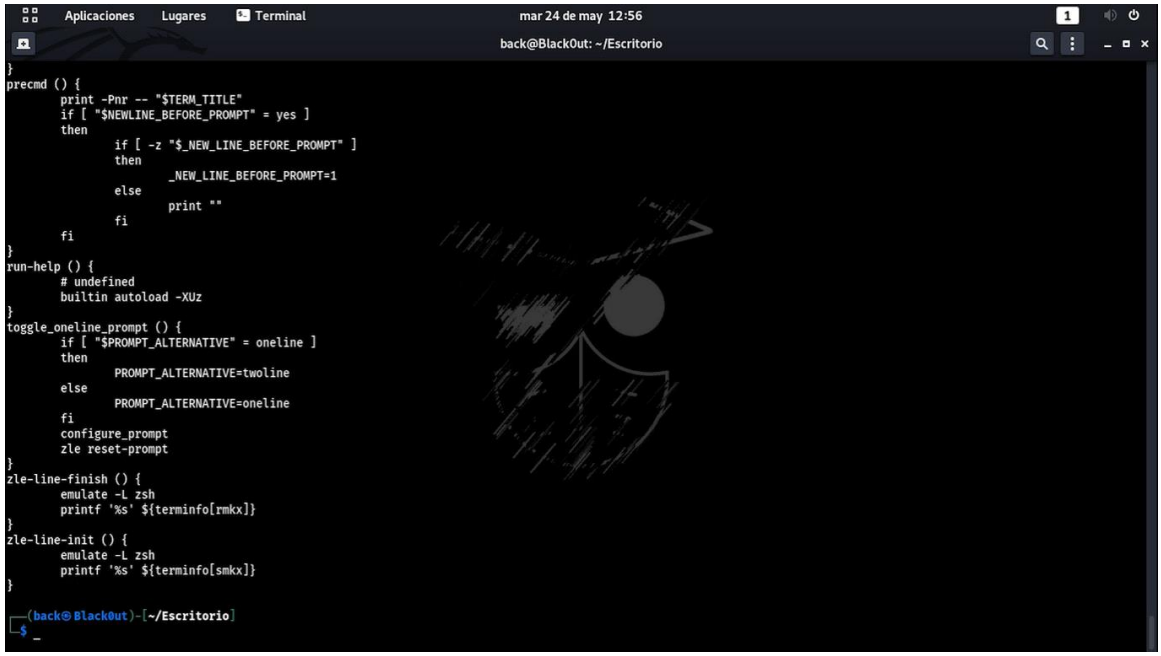
Pero dejando todo el royo, vamos a definir nuestra primera función, para ello existen dos formas:

En realidad ambos casos son aplicables y da lo mismo si lo haces de una manera o de otra. Ademas también podríamos borrar una función previamente declarada, por ejemplo si quisiéramos borrar la funcion1 bastaria con :

Donde como vez –f funciona como una opción para indicar el nombre de la función.

Al definir funciones se almacenan como si fueran variables de entorno. Para ejecutar la función bastaría con escribir el nombre de la misma seguido de argumentos, estos argumentos serian los parámetros de dicha función.

Tambien podemos ver las funciones que tenemos definidas en una sesión usando el comando: ***declare –f***



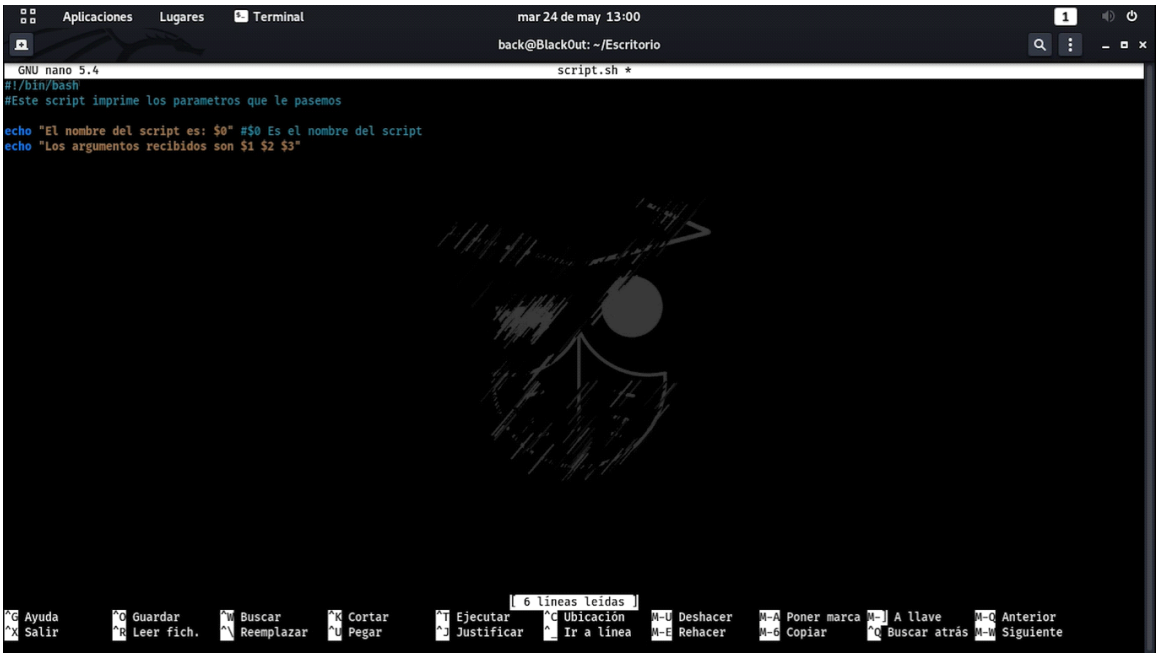
El Shell imprimirá las funciones y su definición, ordenadas alfabéticamente.

Parámetros posicionales

Los parámetros posicionales son los que se encargan de recibir los argumentos que se le pasen a un script y los parámetros de cada función.

Los nombres de estos parámetros posicionales son 1, 2, 3, etcétera. Con lo que para acceder a ellos vamos a utilizar normalmente el símbolo de $ tal cual fueran variables, tipo $1, $2, $3, etc. Ademas tenemos el parámetro posicional 0 que almacena el nombre del script donde se ejecuta.

Por ejemplo vamos a crear el siguiente script:



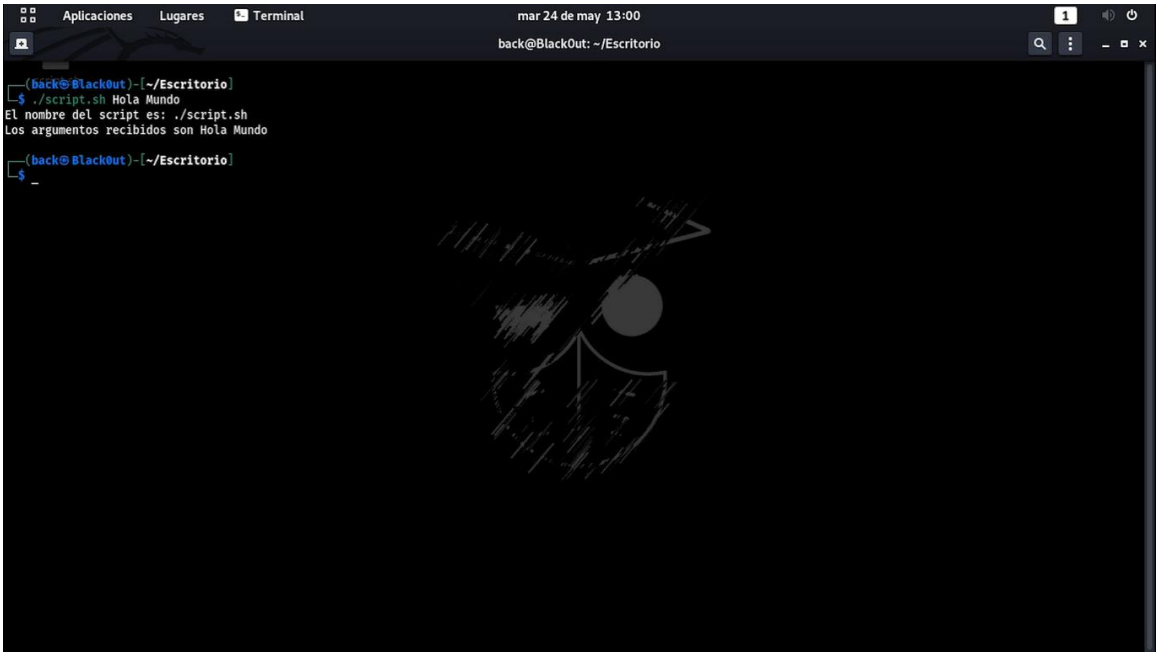
Ahora, fijaos en la función de comentario, los comentarios como he dicho anteriormente, pero ahora vamos a profundizar un poquito... Los comentarios son utilizados para indicarle al script que todo lo que escribamos en esa misma línea seguido del comentario es un comentario y que por ende queremos que lo omita.

¿Qué funcionalidad tiene esto?

Ninguna, precisamente por eso es funcional, los comentarios sirven para escribir notas en el código, para orientarnos y no perdernos mientras estamos programando, o incluso para que otros programadores que puedan ver nuestro código tengan idea de que es lo que estamos haciendo.

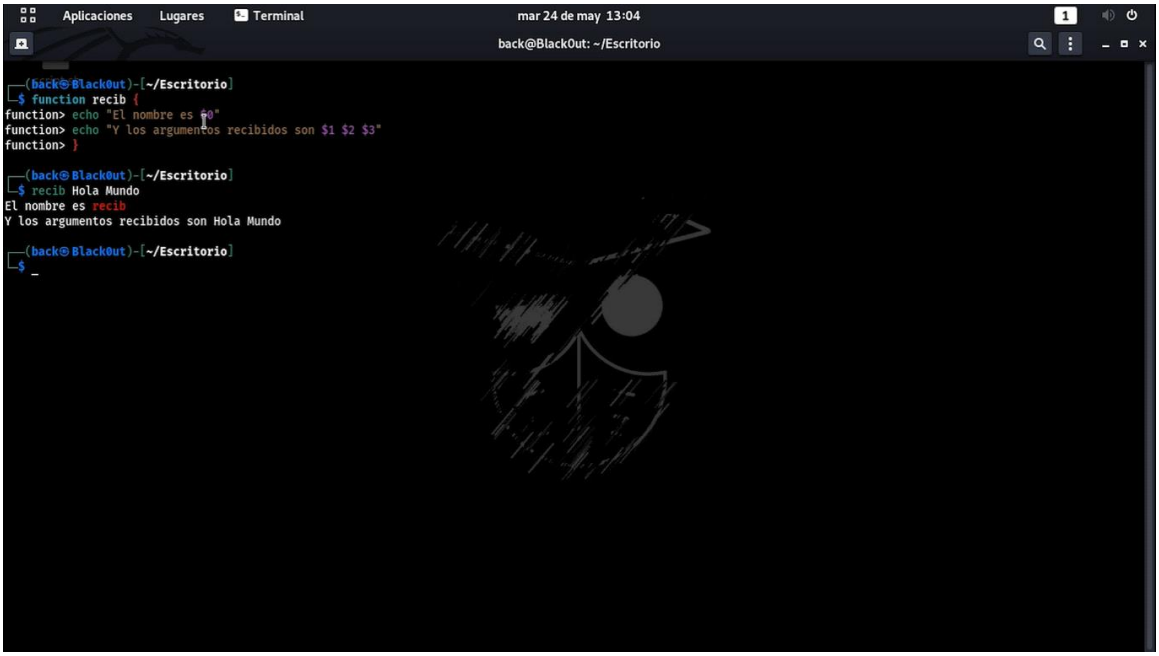
Lo guardamos y le asignamos permisos de ejecución

Posteriormente vamos a ejecutarlo y tal cual le hemos asignado entonces vamos a pasarle parámetros para ver cómo nos los imprime



Primeramente nos imprime el primer ***echo*** donde nos dice el nombre del script, que les dije que se encuentra almacenado en la variable $0 por defecto y posteriormente nos muestra los dos parámetros que le hemos pasado, es decir “Hola Mundo”. Aunque en realidad le hemos dicho que nos imprima los tres primeros parámetros. En realidad no lo a hecho puesto que solo le hemos pasado dos. En este caso el $3 se convierte en argumentos nulos, que por ende dan lugar a una cadena vacía que no es imprimible.

Entonces vamos a combinar esto en el modo interactivo con lo que habíamos visto anteriormente sobre las funciones



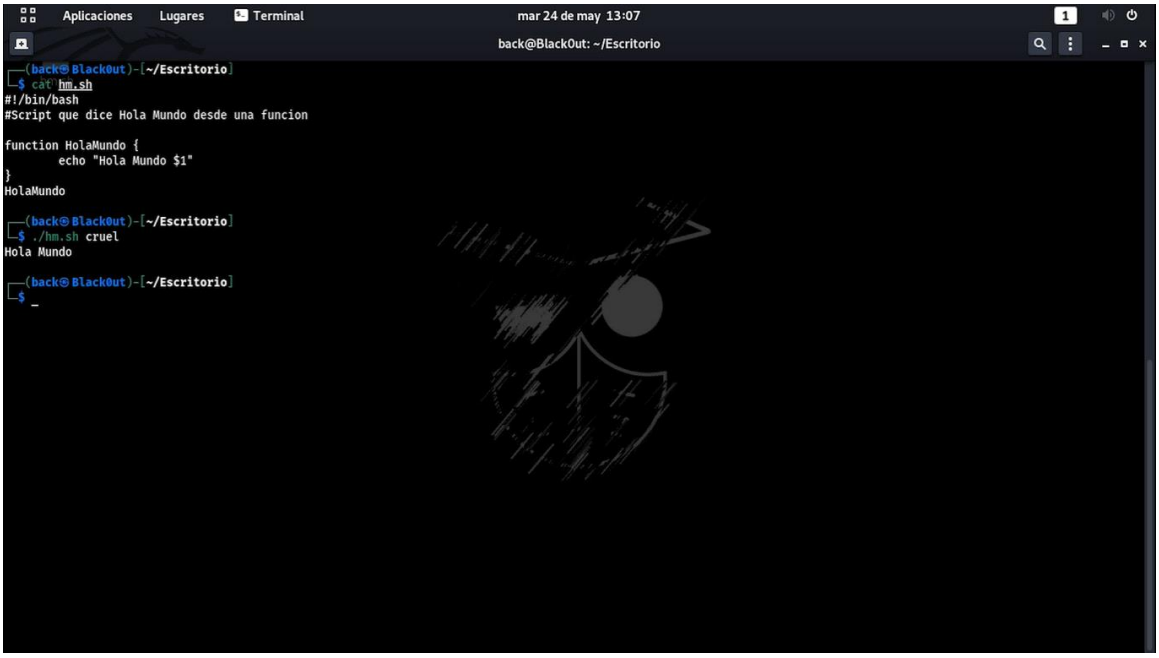
Como ven funciona de igual manera. Además podemos apreciar que como habíamos visto antes podemos hacer uso del modo multilinea de bash, el cual no cierra la función hasta que nosotros no cerremos la llave que hemos abierto para declarar la misma.

Comentar además que no se puede modificar el valor que tienen las variables posicionales, solo se puede leer, si intentamos asignarle un favor a una variable posicional se producirá un error.

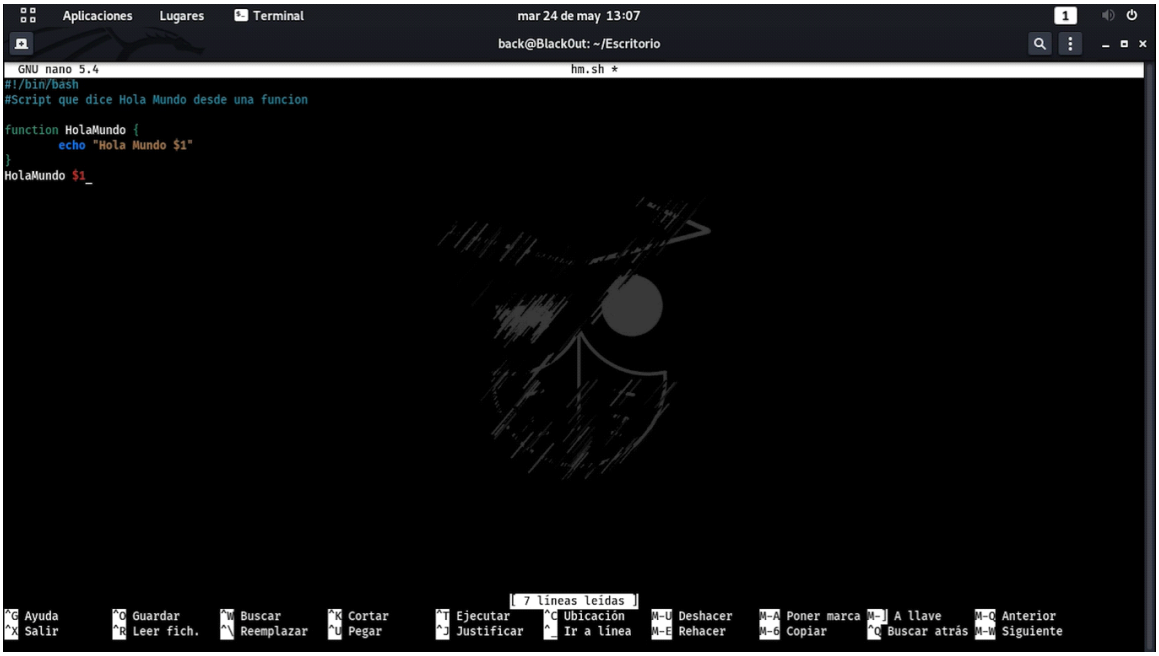
Las variables locales y globales

Como hemos visto los parámetros posicionales en realidad son variables locales del script o función, en realidad no se puede acceder o modificar estas variables desde otra función o desde fuera del script.

Por ejemplo crearemos esta vez una función dentro de un script para que entiendan a que me refiero y crearemos una función que diga “Hola Mundo”, seguido de un parámetro adicional que nosotros le pasemos.



Vemos que el argumento que le hemos pasado al programa no ha sido cogido por la función, esto pasa porque $1 es un argumento local del script y si queremos que lo reciba la función tendremos que pasárselo directamente desde el código



De esta forma al ejecutar la función si que le estaremos pasando el argumento tomado del script.

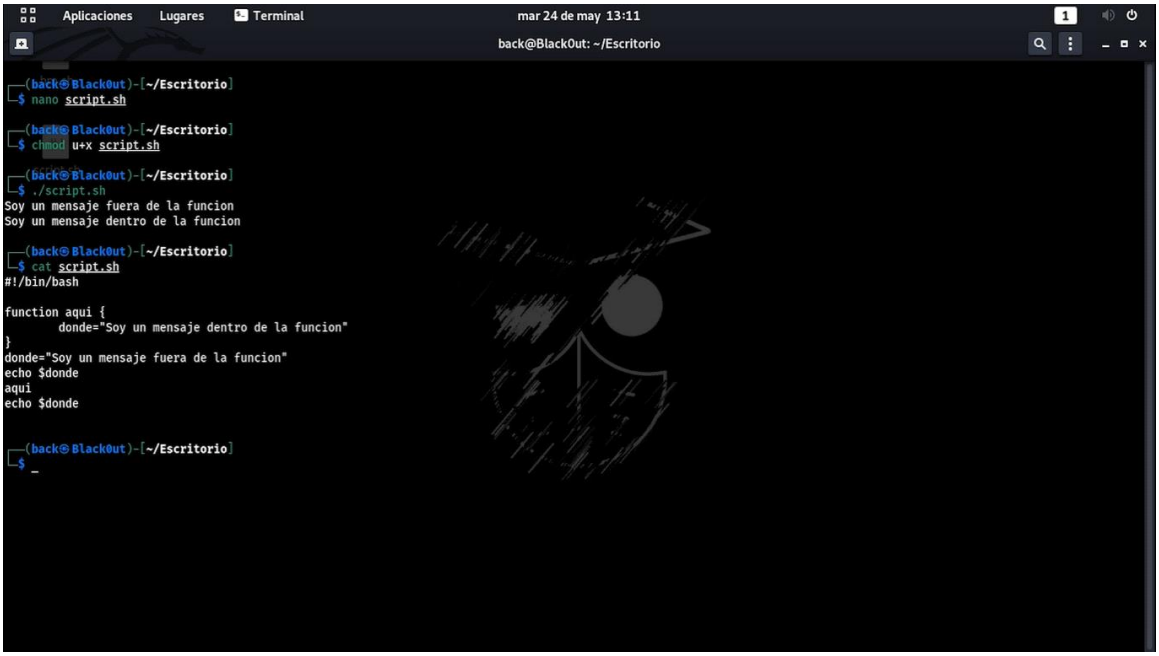
Pero que conste que esto solo va a ocurrir en el caso de los parámetros posicionales, osea que el resto de variables que definamos en un script o función son globales, es decir que una vez definas el script serán accesibles y variables.

Por ejemplo vamos a crear un script donde podamos comprender mejor esto de las variables globales.

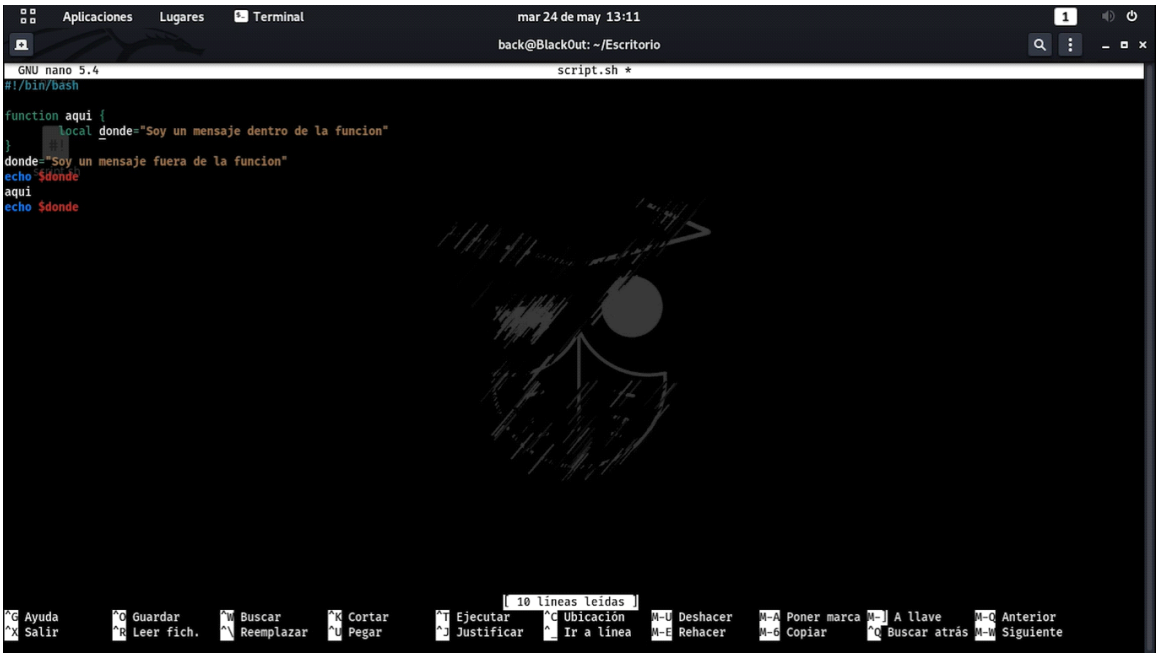
En este caso vamos a editar un script con un nombre cualquiera, definimos que queremos que se ejecute con bash. Vamos a declarar una función que se llamara aquí.

Dentro de la misma crearemos una variable que almacenara un valor característico para que podamos identificar que está dentro de la función, posteriormente la cerramos.

Y seguido de esto vamos a volver a declarar debajo de la función la variable donde con otro valor, que deje claro que es un mensaje en el código del script, ósea fuera de la función. En este punto estamos declarando ese mensaje dos veces, dentro y fuera de la función. Claramente se ejecutaría el que se encuentra fuera de la función por el momento, podemos verlo si posteriormente hacemos un echo de la variable $donde y acto seguido vamos a hacer llamado a la función para que vean como al volver a ejecutar el script el valor de donde a variado.



Básicamente esto es lo que ocurre con las variables globales. Si quisiéramos que una variable fuera local debemos de ponerle el modificador (local), que solo se puede utilizar dentro de una función, no en el código tal cual del script. Por ejemplo podemos ver el siguiente caso:



Si lo ejecutamos encontraríamos que ambas variables ahora están fuera de la función.

Otras variables curiosas y necesarias

En este caso vamos a ver otras variables que nos serán verdaderamente útiles, comenzaremos por:

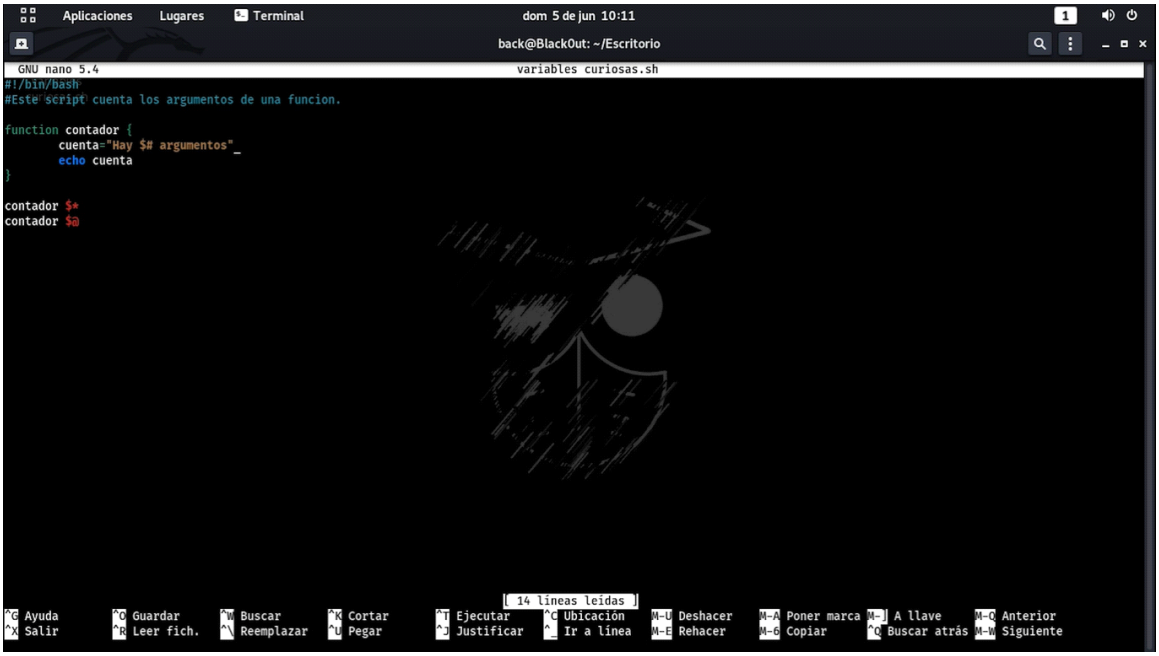
$# Esta variable almacena el número de argumentos o parámetros recibidos por el script o la función. El valor es del tipo por defecto, ósea cadena de caracteres.

Y por otra parte las variables **$ @** y **$*** que simplemente almacenan los argumentos recibidos por el script o función. Cuando no se entrecomillan se compartan exactamente igual, pero al encerrarlos entre comillas la variable “$ @” creara un token por cada argumento recibido y “$*” creara un único toquen correspondiente a los argumentos recibidos.

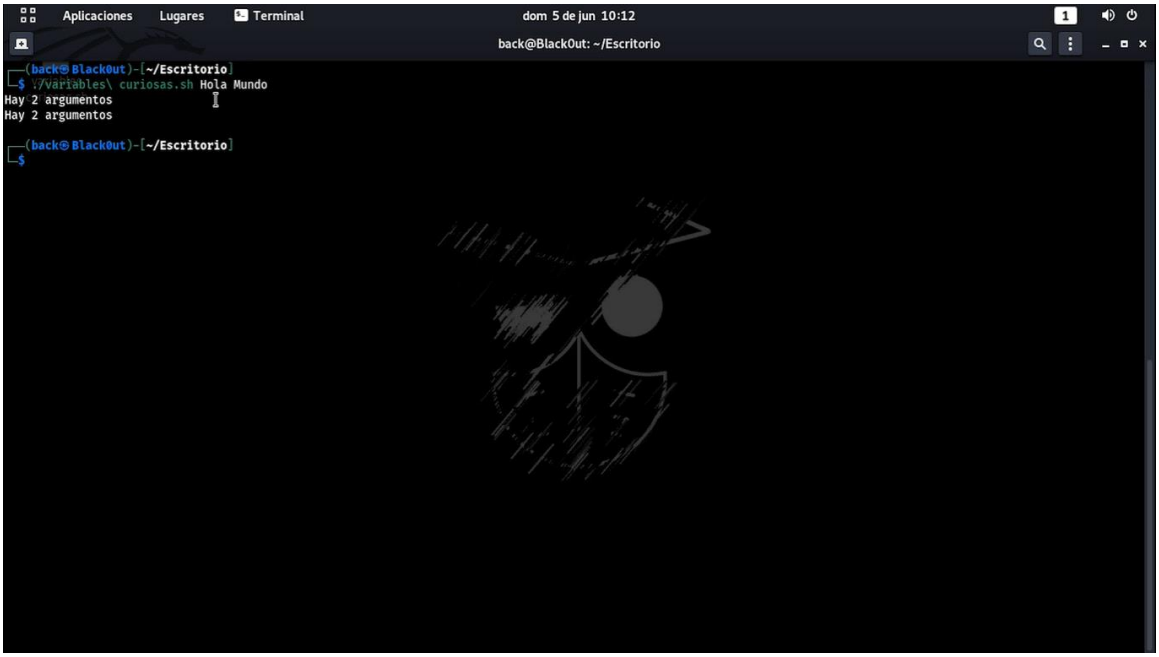
Podríamos utilizarlos de una manera bastante básica de la forma siguiente:

Nano “variables curiosas.sh”

Podríamos hacerlo de otras maneras, por ejemplo haciendo uso de alguna función, por hacer algo distinto.



Posteriormente lo ejecutamos pasándole algún argumento claramente.



Y funcionaria de una manera bastante similar.

Expandir variables usando llaves

Para expandir una variable tenemos una simplificación, la cual sería meramente ${variable}

Esta forma de ampliación se usa siempre que la variable vaya seguida por una letra, digito o guion bajo. En caso contrario podemos utilizar la forma más simplificada, ósea simplemente $variable. Por ejemplo si queremos escribir algún texto almacenado en una variable y luego otro almacenado en otro separado por un guion podríamos hacer algo como lo siguiente:

Saludo=Hola

Saludando=Mundo

Echo “$saludo_$saludando”

Y si os fijáis produce una salida que no es la que buscamos, bash a intentado buscar la variable nombre_, si ósea con el guion bajo incluido que como no existe pues lo a tomado como que la variable nombre_ tiene un valor nulo. Podríamos solucionar algo como esto utilizando las llaves, como en el ejemplo anterior.

Sentencias if, elif y else

Los que vengan de otros lenguajes de programación ya se sentirán identificados con esto, para los que no, no se preocupen, aquí va la parte más simple de esto.

Estas sentencias condicionales en bash han de tener un formato similar al de la imagen siguiente.

Bash nos obliga a que las sentencias estén organizadas con estos cambios de línea, aunque algunas personas prefieren poner los then en la misma línea de los if, para lo cual debemos de usar el ; que en bash se utiliza como separador de comandos.

Código de salida

A raíz de todos los sistemas al fin y al cavo basados en UNIX los comandos terminan devolviendo siempre un código numérico de finalización, o código de salida de un programa, el exit status. Que indicara si el comando tiene existo o no.

Esto lo hemos visto anteriormente, claramente nos referimos al stderr y el stdout

Normalmente el código de terminación 0 significara que un comando termino correctamente y un código entre 1 y 255 quiere decir que puede haber ocurrido algún error. En casos específicos esto no tendría por que ser asi necesariamente, siempre sería recomendable consultar la documentación de cada comando para poder interpretar mejor cuales y porque son tales y cuales códigos de salida.

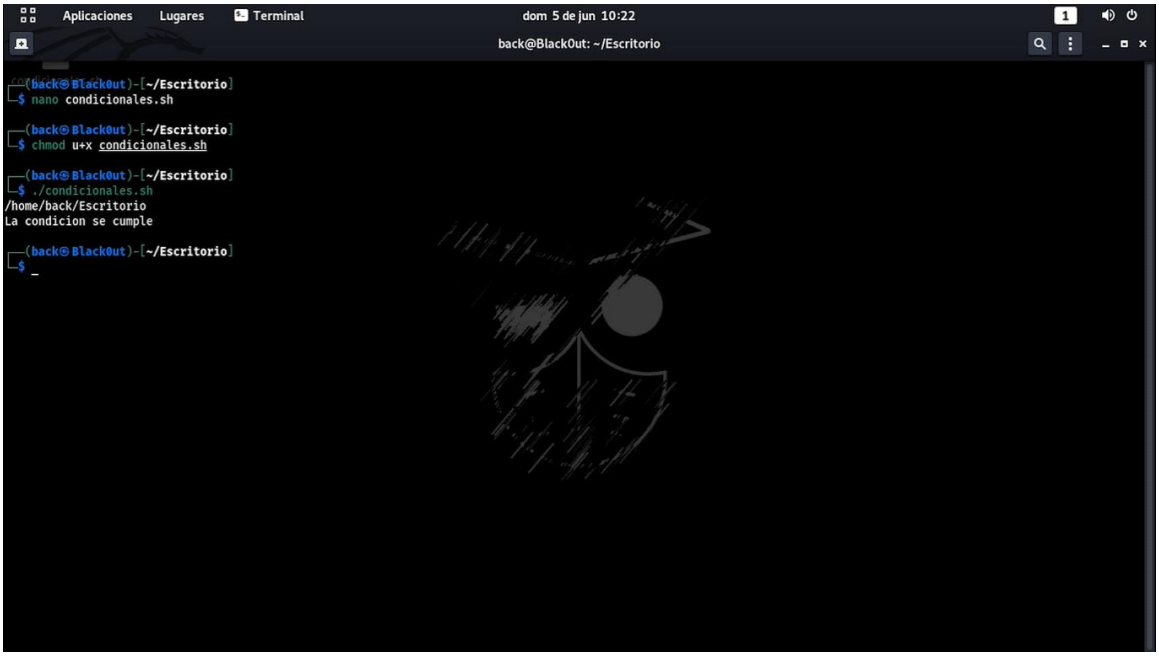
¿Pero de que viene mi salto de sentencias sin siquiera haber mostrado como declarar una a esto de los códigos de salida?

Pues la sentencia if comprueba a fin de cuentas el código de salida de un comando en la condición. Si el código de salida es 0 la condición se evalua como cierta, con lo cual una manera acertada de escribir el condicional if seria

Como podrías intuir, leer este código es bastante sencillo. Simplemente con if le estamos diciendo que si se cumple la condición “pwd” que lógicamente es un uso correcto del comando, con lo cual tendrá un código de salida exitosa. Entonces queremos que entre dentro de las sentencias definidas dentro de la condicional if. Y else que lo definimos por

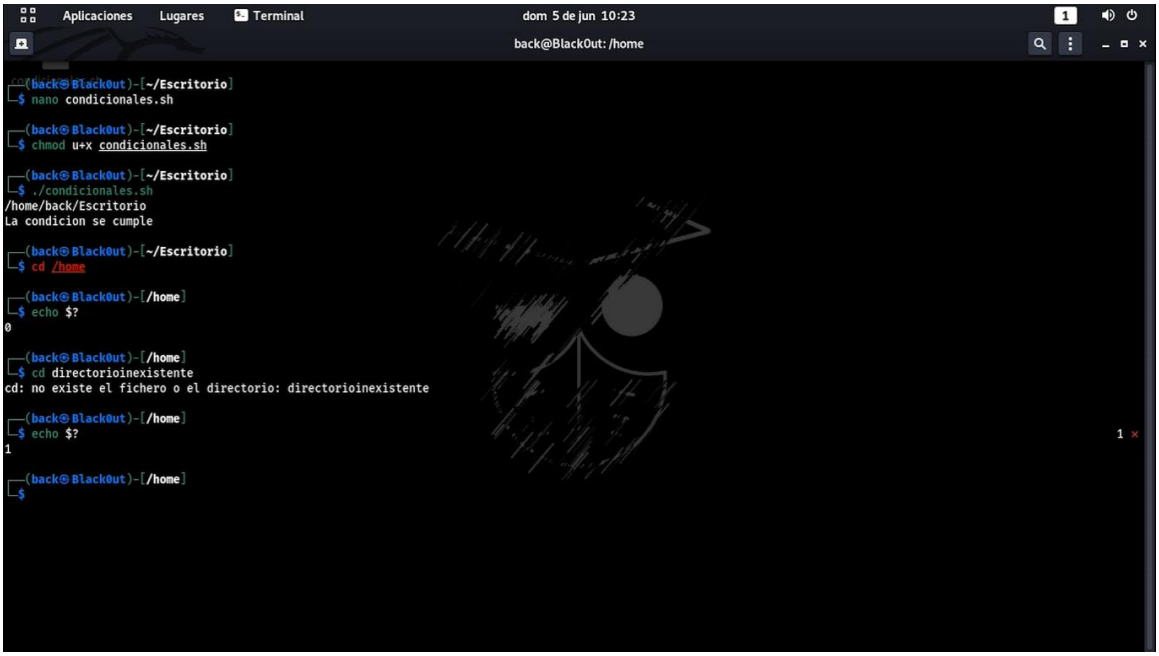
debajo sirve para decir que si no, ósea si no se cumple ninguna condición antes expuestas, que en este caso hay una sola, nos diga que no se cumple la condición.

Simplemente guardamos como siempre, asignamos permisos de ejecución y ejecutamos.



Además, anteriormente habíamos visto sobre el código de terminación de cada programa, al menos espero que se estén dando cuenta poco a poco que todo lo que en un principio explicaba y resultaba inútil en ese entonces comienza a cobrar sentido. En este caso vamos a profundizar todavía un poquito más sobre esto. Y es que quizás quieras ver el código de terminación de un programa por ejemplo.

Vamos a ver el código de terminación de un programa exitoso por ejemplo y de uno fallido.



Evidentemente en el primer caso, si intentamos movernos al /home, el código de terminación es 0. Puesto que a fin de cuentas la salida del programa es exitosa.

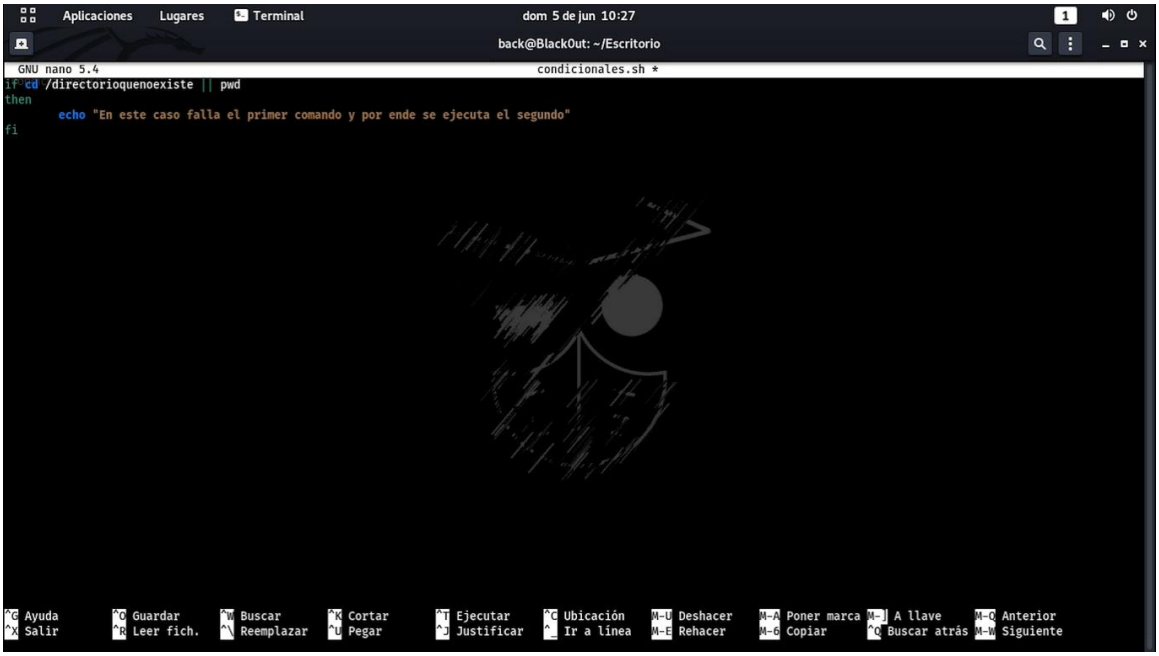
En caso contrario, por ejemplo si intentásemos movernos a un directorio inexistente, claramente el código es diferente de 0, que sería el exitoso. En este caso en concreto es 1, que a fin de cuentas es una salida no exitosa.

Operadores lógicos y códigos de terminación

También se pueden combinar varios códigos de terminación mediante operadores lógicos, como and, que sería el operador “&&” y cuya función es bastante simple, es decir si se cumple la primera condición ejecuta la que le siga, sino, no.

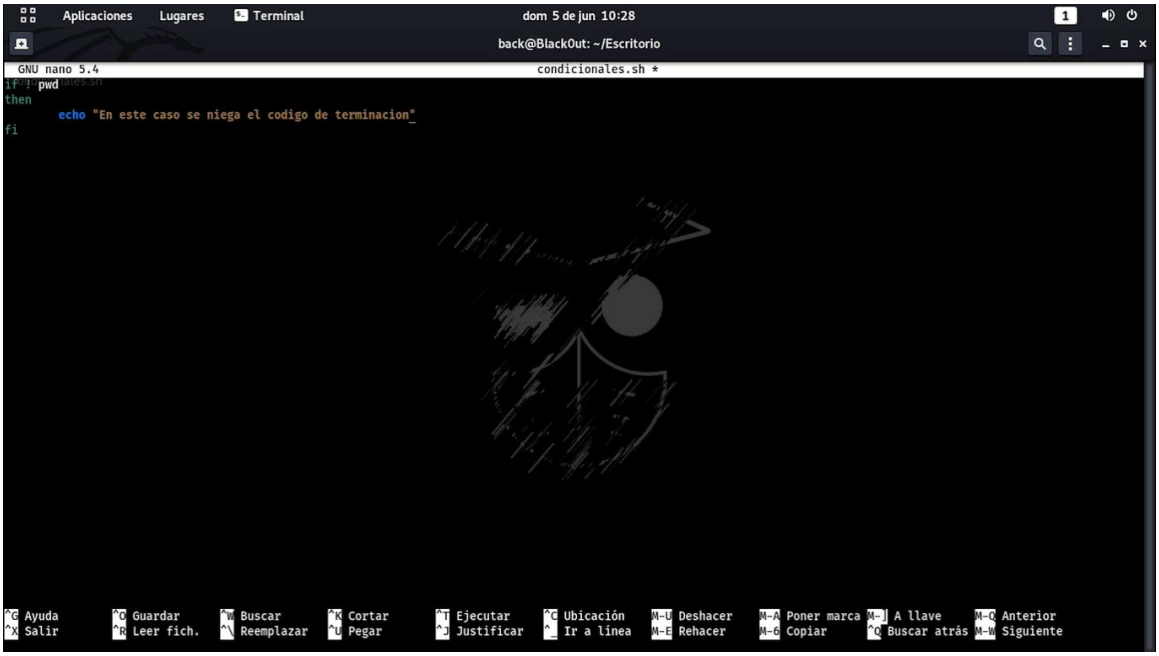
Claramente ejecuta ambos comandos, osa como que los concatena, pero que conste que este operador solo funciona si la primera condición se cumple, es decir si se cumple la primera condición, va a por la segunda, si no, no. Lamento repetirme pero intento que quede claro.

Tambien tenemos otro operador lógico que le haga la contra por llamarle de alguna forma a este que hemos visto, osea que ejecute el primer comando y solo si este falla ejecute el segundo:



En caso de que el primer comando fuera exitoso no se ejecutaría el segundo, ya que tiene éxito el primero, con lo cual se cumple un operando.

Y por ultimo tenemos el operador ! que niega el código de terminación.



Es decir en este último caso aunque pwd es un uso correcto y la salida es exitosa, "!" lo niega.

Operadores de comparación

La verdad es que hay bien poca teórica para explicar por el momento sobre los operadores de comparación, al menos sin entrar en la práctica, al fin y al cavo eso son, operadores que se utilizan para comparar.

Los operadores de comparación son:

Operador	**Verdadero si...**
var = var1	Son iguales
var != var1	Son distintas
var < var1	Var Es menor lexicográficamente
var > var1	Var Es mayor lexicográficamente
-n var	Var no es nula, tiene una longitud mayor a 0
-z var	Var es nula, su longitud es 0

Para ver esto en la práctica vamos a desarrollar varios scripts.

Primeramente crearemos un script con lo visto hasta ahora

En el mismo crearemos una función que se llamara ejemplo y definiremos su contenido como un mero, “si la variable $variable que aun no hemos definido” no es nula o tiene una longitud mayor que 0, que lo indicamos con el –n y además poniendo la variable entre comillas que verán luego y es que si la variable esta vacia se expandirá por –n produciendo un error. Con las comillas se expandirá por –n “” que es lo que queremos, entonces vamos a imprimir que la variable no es nula. Y si no vamos a decir que, la variable es nula, o su valor es 0.

Y en caso de que definiéramos la variable previamente con un valor diferente al de 0

En este caso como declaramos variable anteriormente, no es nula al momento de comparar en la funcion, con lo cual, entrara en el if.

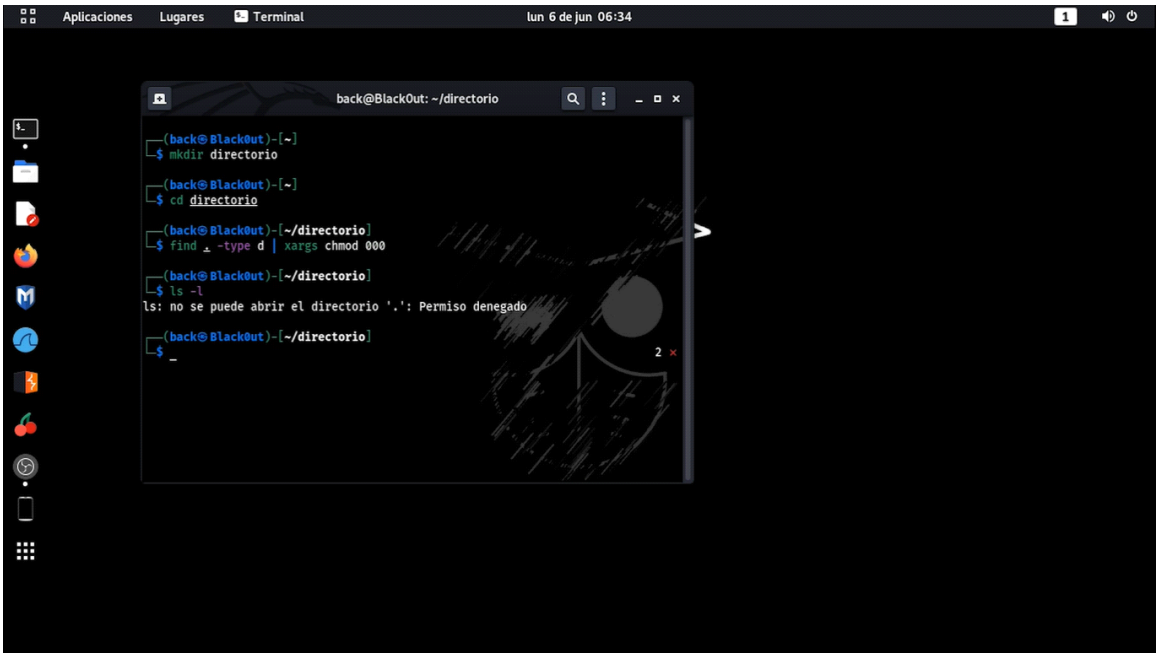
Bucle for

El bucle for es un poco diferente a los bucles for tradicionales de otros lenguajes de programación, si no que parece más al bucle for each de otros lenguajes, puesto que aquí no se repite un número fijo de veces, sino que se procesan las palabras de una frase una a una.

Esta lista del blucle for puede contener comodines, por ejemplo podemos crear un buble for que muestre la información detallada de todos los ficheros del directorio actual.

Comando xargs

El comando xargs es bastante útil si queremos ejecutar una operación sobre una lista de cadenas. El comando recibe en su entrada estándar una lista que puede estar separada por espacio, tabulador o saltos de líneas. En fin podemos utilizarlo para ejecutar la utilidad pasada como argumento sobre cada elemento de una lista. Para los no duchos de la teórica vamos a intentar cambiar los permisos a todos los subdirectorios del directorio actual para que entiendan.



En este caso el comando find escribe en su salida estándar los subdirectorios. Después el comando xargs lee estos directorios y ejecuta sobre cada uno de ellos el comando chmod 000.

Los bucles while y until

Los bucles while en concreto tienen la siguiente sintaxis

While condicion

Do

    comando

Done

Until condicion

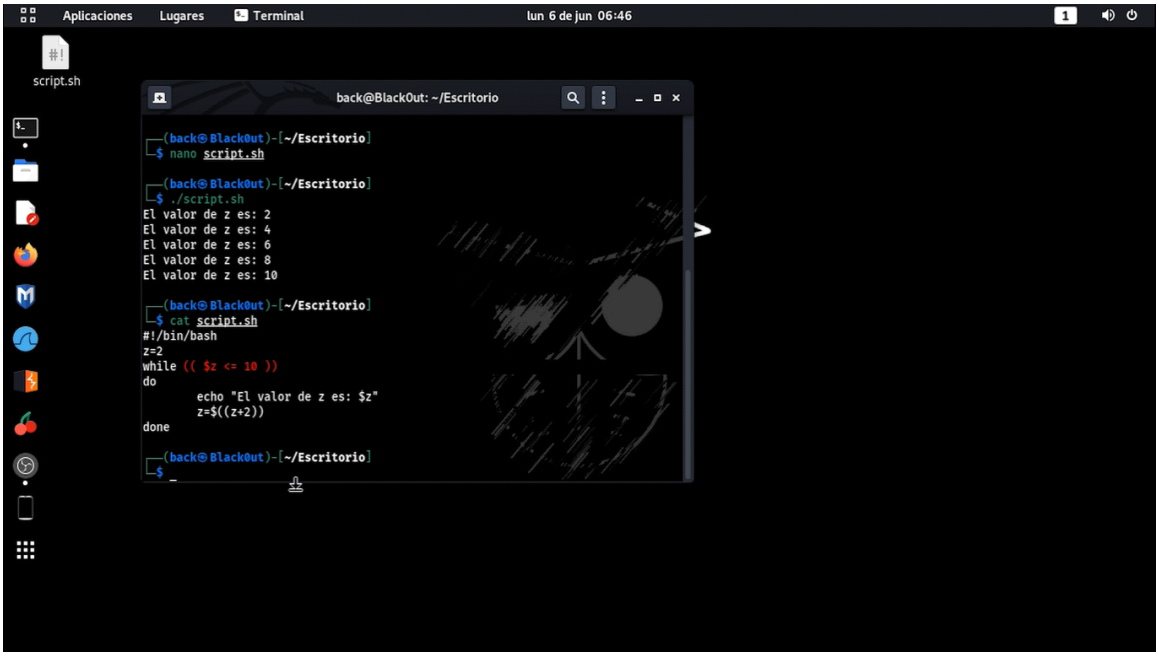
Do

    comando

Done

Vamos a crear la variable z cullo valor será “2”., posteriormente con un bucle while vamos a decir que mientras z sea menor o igual que 10 pues nos imprima el numero, osea, el valor de z. y además dentro de ese mismo bucle while vamos a incrementar el valor de z en dos unidades.

Tambien que sepan que voy a estar incorporando expresiones aritmeticas, no te agobies si no comprendes, no pasa nada, lo explicare mas adelante en el curso, no te enfrasques en la sintaxis y quedate con la logica de todo, que es lo mas importante.



Pero no tenemos que cambiar nuestros corchetes a los que estamos acostumbrados por los paréntesis.

Podríamos volver a modificar el script y

Sentencia case

La sentencia “case” en bash se utiliza para realizar una comparación de patrones con la cadena a examinar.

La sintaxis seria:

Case cadena in

Patron1)

Sentencias ;;

Patron2)

Sentencias ;;

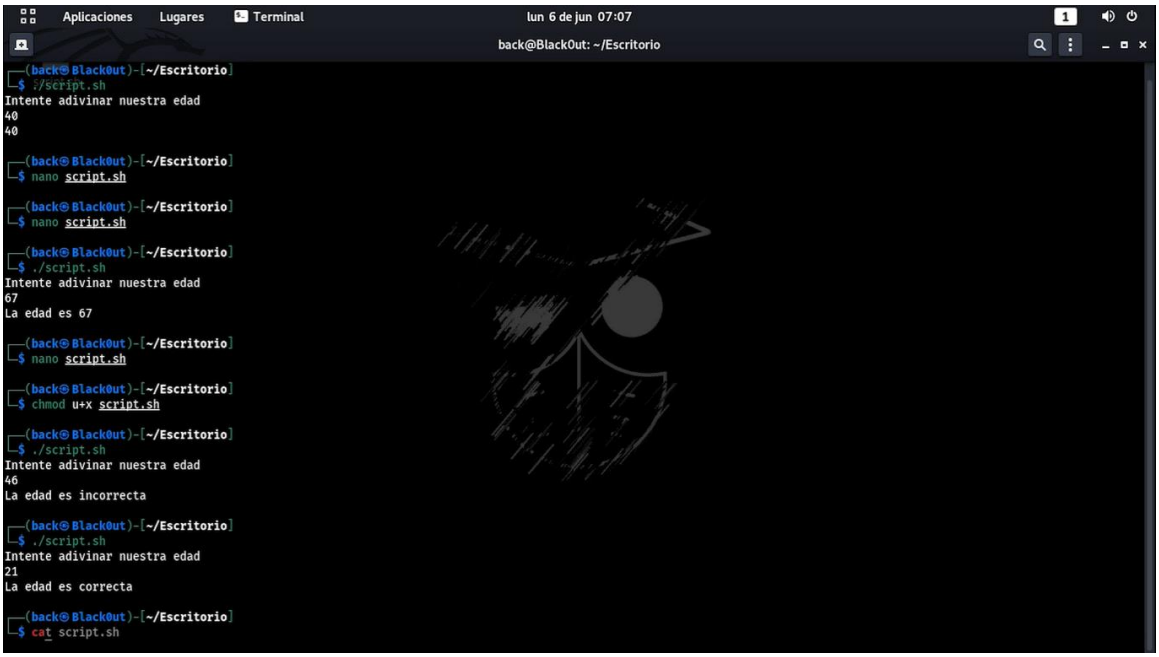
....

Esac

Cada patrón puede estar formado por varios patrones separados por el carácter **“|”.** Si la cadena cumple alguno de los patrones, se ejecutan sus correspondientes sentencias (las cuales se separan por ;) hasta ;;.

Por ejemplo vamos a crear un script que no hara mas que adivinar una edad, por ejemplo

En este script hay simplemente dos posibles casos, el primero es cuando el valor es 21, entonces imprime correcto y el segundo caso que contiene un * que se utiliza para indicar todas las cadenas que no cumplen con las características de los casos anteriores, pues como saben el asterisco simboliza cualquier cosa, incluso 21, pero como 21 se encuentra antes según el flujo de ejecución del programa. Pues de ser 21, nunca llegaría a ejecutar el *, en fin. Y por cierto si ejecutamos el script verán que lo que hace el read simplemente es esperar a que el usuario pase por pantalla los datos y los asigna posteriormente a la variable. La verdad es que no recuerdo si he dicho eso en el curso o no, por si no pues ahí lo tienen.



SENTENCIA SELECT

La sentencia select nos permite generar fácilmente un menú simple.

Su sintaxis es

Select variable [in lista]

Do

    Sentencias que usan $variable

Done

Como ven es la misma sintaxis que el bucle for, excepto por la palabra clave select en vez de for.

La sentencia genera un menú con los elementos de lista, done asigna un número a cada elemento y pide al usuario que introduzca un número. El valor elegido se almacena en $variable y el número elegido en la variable REPLY. Una vez elegida una opción por parte del usuario se ejecuta el cuerpo de la sentencia correspondiente a la opción elegida y el proceso se repite de manera infinita.

Aunque el blucle select es infinito (lo cual nos permite volver a pedir una opción cuantas veces haga falta) el bucle se puede abandonar usando la sentencia break que creo hasta ahora no hemos visto. La misma se usa pues para eso, en la mayoría de los lenguajes de programación, abandonar un bucle. Y se puede usar en este caso como en los bucles for, while y until.

Para ver un ejemplo y aquí se me va a ir un poquito la pinza vamos a crear una calculadora de números enteros.


Aplicaciones
Lugares
Terminal
lun 6 de jun 07:35
back@BlackOut: ~/Escritorio
GNU nano 5.4
script.sh *
#!/bin/bash
calcular () {
read -p "Escribe el primer numero: " n1
read -p "Escribe el segundo numero: " n2
let calculo="$n1 $1 $n2"
echo $calculo
}
PS3="Selecciona una operacion: "
select opt in sumar restar multiplicar dividir ;
do
case $opt in
sumar) calcular "+";;
restar) calcular "-";;
multiplicar) calcular "*";;
dividir) calcular "/";;
*) echo "$REPLY es una opcion invalida"
;;
esac
done
[23 líneas leídas]
^G Ayuda ^O Guardar ^W Buscar ^K Cortar ^T Ejecutar ^C Ubicación M-U Deshacer M-A Poner marca M-] A llave M-Q Anterior
^X Salir ^R Leer fich. ^\ Reemplazar ^U Pegar ^J Justificar ^_ Ir a línea M-E Rehacer M-6 Copiar ^Q Buscar atrás M-W Siguiente


Aplicaciones
Lugares
Terminal
lun 6 de jun 07:36
back@BlackOut: ~/Escritorio
(back㉿BlackOut)-[~/Escritorio]
$ ./script.sh
1) sumar
2) restar
3) multiplicar
4) dividir
Selecciona una operacion: 1
Escribe el primer numero: 2
Escribe el segundo numero: 2
4
Selecciona una operacion:

Sobre los parámetros y argumentos

Es muy típico que los comandos en distribuciones de GNU Linux tengan formatos como:

comando --parámetros argumentos

Es decir los parámetros suelen estar antes de los argumentos y tienen un guion delante.

Los parámetros, al igual que los argumentos se reciben en las variables posicionales, con lo que si por ejemplo ejecutamos hacer -o esto.txt aquello.txt en $1 recibimos -o, en $2 recibimos esto.txt y en $3 recibimos aquello.txt. Luego en principio para tratar las opciones no necesitaríamos nada más. El problema recae en que los parámetros a veces son opcionales, es decir, pueden darse, pero no necesariamente, con lo cual el script que procesa la opción anterior debería tener la forma:

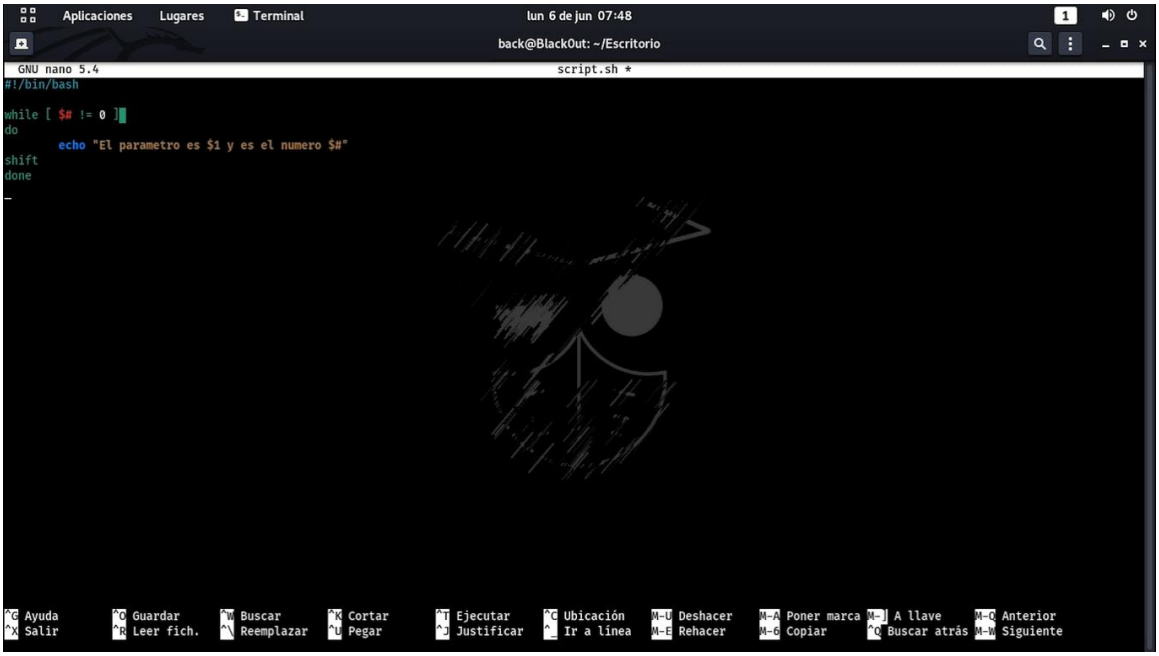
En consecuencia cuando el número de opciones crece, la programación de scripts se vuelve engorrosa.

Sentencia shift

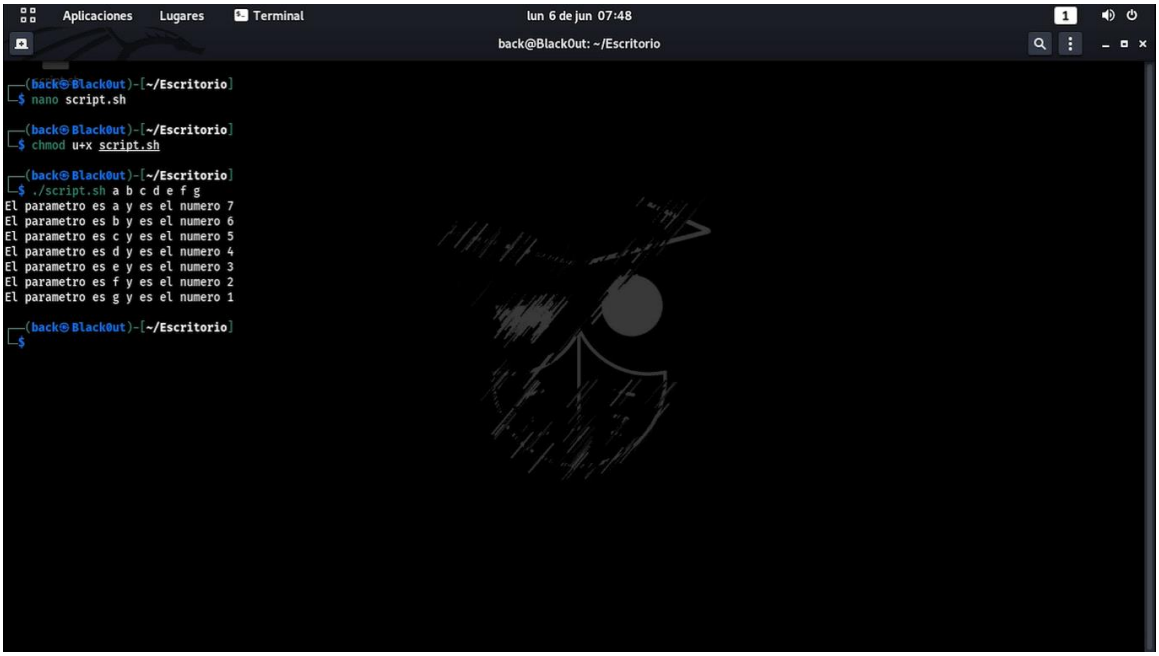
Afortunadamente la sentencia shift nos permite solucionar este engorro elegantemente.

Esta sentencia se suele utilizar para mover parámetros, ósea desplazarlos a la izquierda. Normalmente se utiliza para recorrer cada parámetro a su vez sin conocer el número de parámetros entrantes y luego realizar el procesamiento correspondiente (común en los scripts de inicio de varios programas en Linux)

Por ejemplo vamos a crear un script que lea los parámetros de entrada uno por uno e imprima el número de parámetros



Cada vez que ejecuta un turno (sin parámetros) se destruye un parámetro y los siguientes parámetros se mueven hacia adelante.



El número de parámetros que mueve el comando shift a la vez se especifica mediante los parámetros que lleva. Por ejemplo después de que el programa de Shell haya procesado los primeros nueve parámetros de la línea de comandos puede usar el comando shift 9 para mover $10 a $1.

TIPOS DE VARIABLE

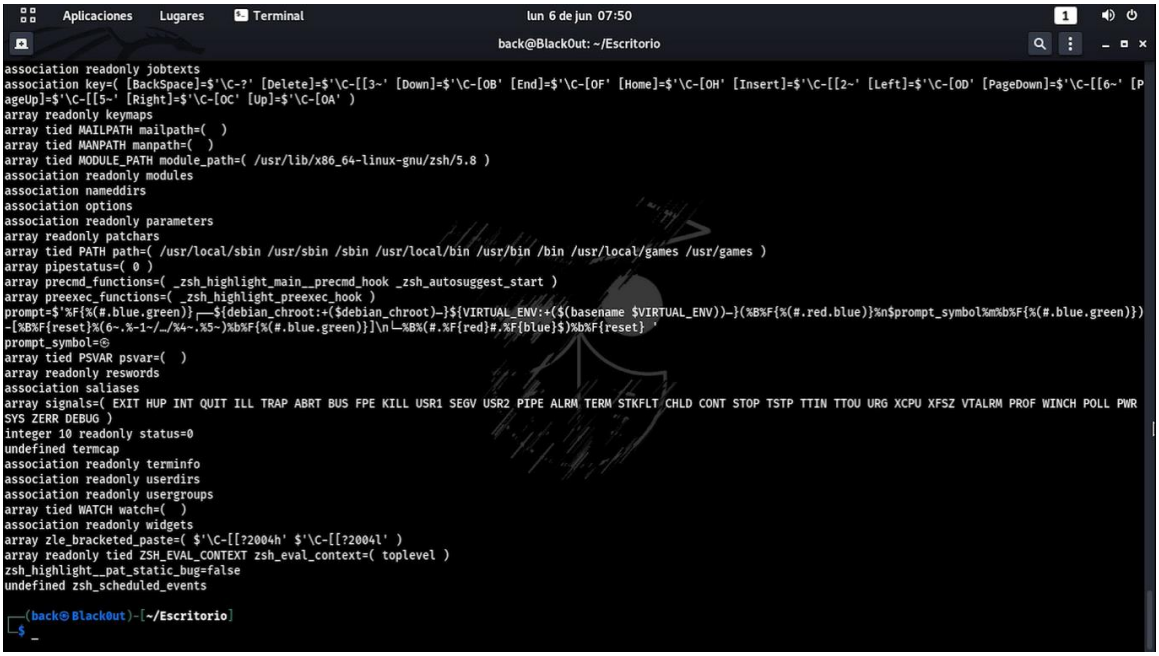
Hasta ahora todas las variables que hemos visto que yo recuerde eran strings, ósea cadenas de caracteres.

Para fijar los atributos de las variables podemos utilizar el comando interno *declare*.

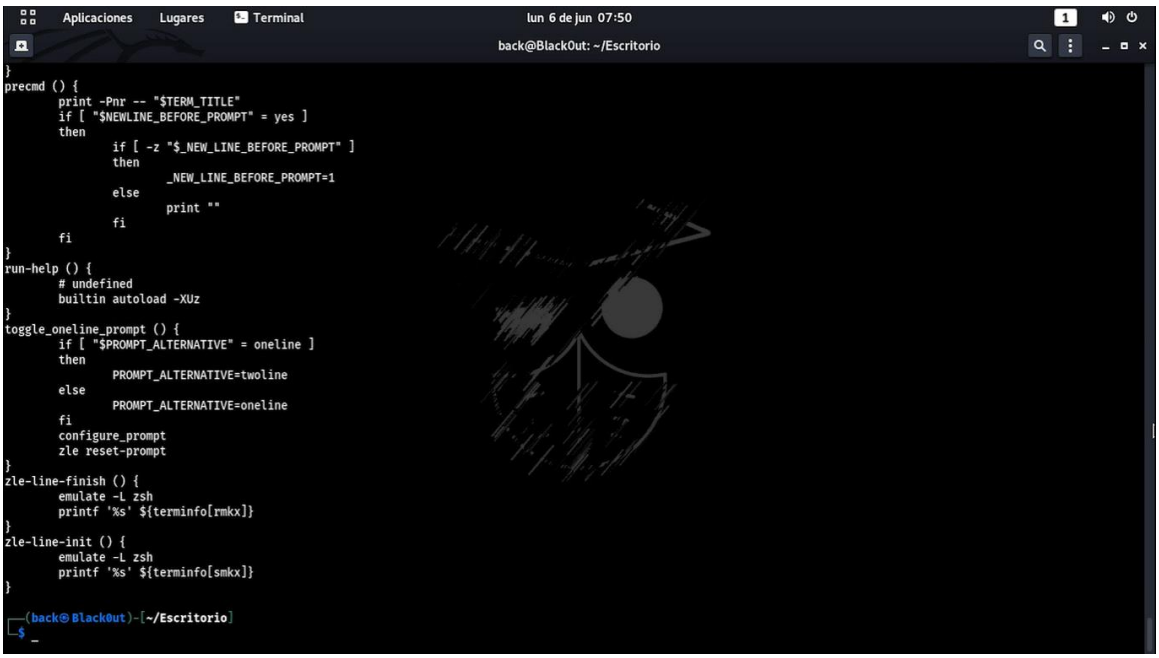
En la siguiente tabla tienen los parámetros que puede recibir este comando. Una peculiaridad de este es que para activar un atributo se precede la opción por un guion, con lo que para desactivar un atributo decidieron preceder la opción por +, en fin veremos más luego.

Opción	**Información**
-a	La variable es de tipo array
-f	Mostrar el nombre e implementación de las funciones.
-F	Mostrar solo el nombre de las funciones.
-i	La variable es de tipo entero
-x	Exporta la variable (equivalente a export)
-r	La variable es de solo lectura.

Si utilizaramos *declare* sin pasarle argumentos nos mostrara todas las variables de entorno:



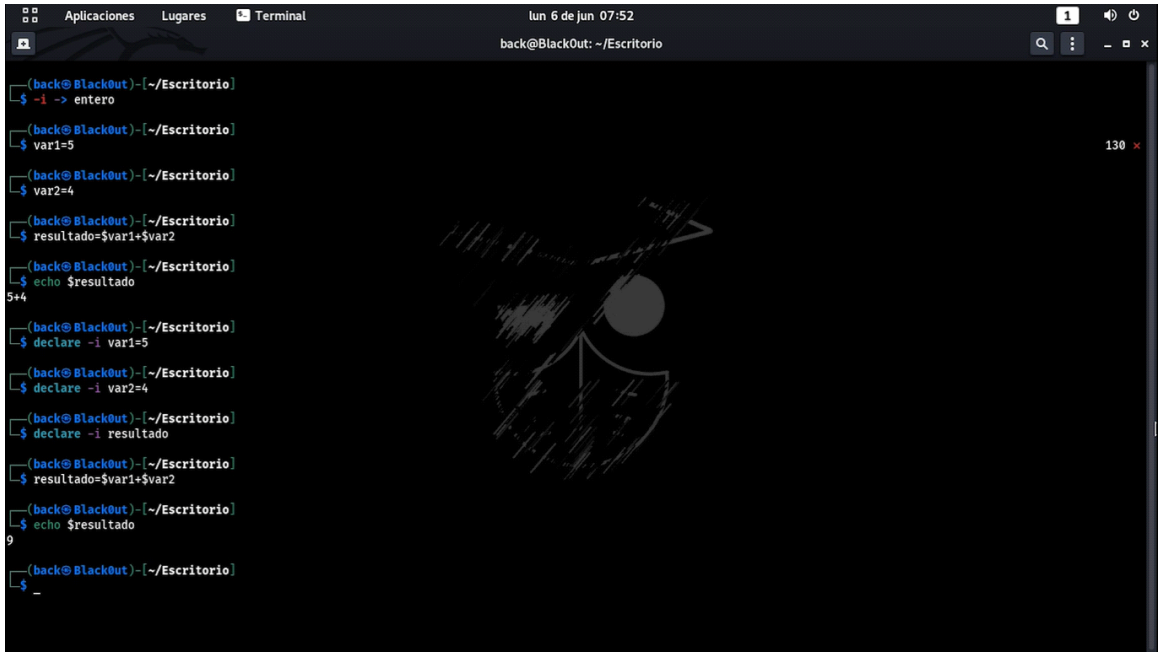
Si usamos **–f** nos muestra solo los nombres de funciones y su implementación y con –F nos muestra solo los nombres.



Las variables que se declaran con declare dentro de una función son variables locales a la función de la misma forma que si hubiésemos usado el modificador local.

La opción –i nos permite declarar una variable de tipo entero, lo cual nos permite que podamos realizar operaciones aritméticas con ella, por ejemplo si usamos variables de entorno normales para realizar operaciones aritméticas simplemente va a juntar ambos numeros tomandolos como cadenas de texto.

Sin embargo si le especificamos que son de tipo enteros>



Si quisiéramos saber el tipo de una variable podemos hacerlo con –p, por ejemplo:

Y nos muestra –i que es como la habíamos declarado

Las expresiones aritméticas

Las expresiones aritméticas se evalúan dentro de las comillas blandas. Fuera de esto no hay mucho concepto del cual hablar.

Podríamos utilizar un comando basándonos en expresiones aritméticas para calcular por ejemplo el número de días que faltan para el 31 de diciembre, esto restando a 365 días el número de días transcurridos, de la siguiente forma:

Echo “$((365 - $(date +%j))) días para el 31 de diciembre

Para profundizar más tenemos el comando interno let permite asignar el resultado de una expresión aritmética a una variable. En fin, tiene la sintaxis siguiente

Let var=expresión

Donde expresión es cualquier expresión aritmética.

A diferencia del comando declare –i. let no crea una variable de tipo entero, sino una de tipo cadena de caracteres, normal, por ejemplo

Let a=4*3

Declare –p a

Vemos que a es del tipo normal. Mientras que si usamos declare –i nos la crea de tipo entero

Declare –i a =3*4

Declare –p a

Esto mismo ocurre con las variables normales, las declaradas con let no pueden tener espacios entre la variable y el signo =, ni entre el signo = y el valor. Aunque si pueden tener espacios si encerramos la expresión regular entre comillas:

Let x=” (9*5) / 7 “

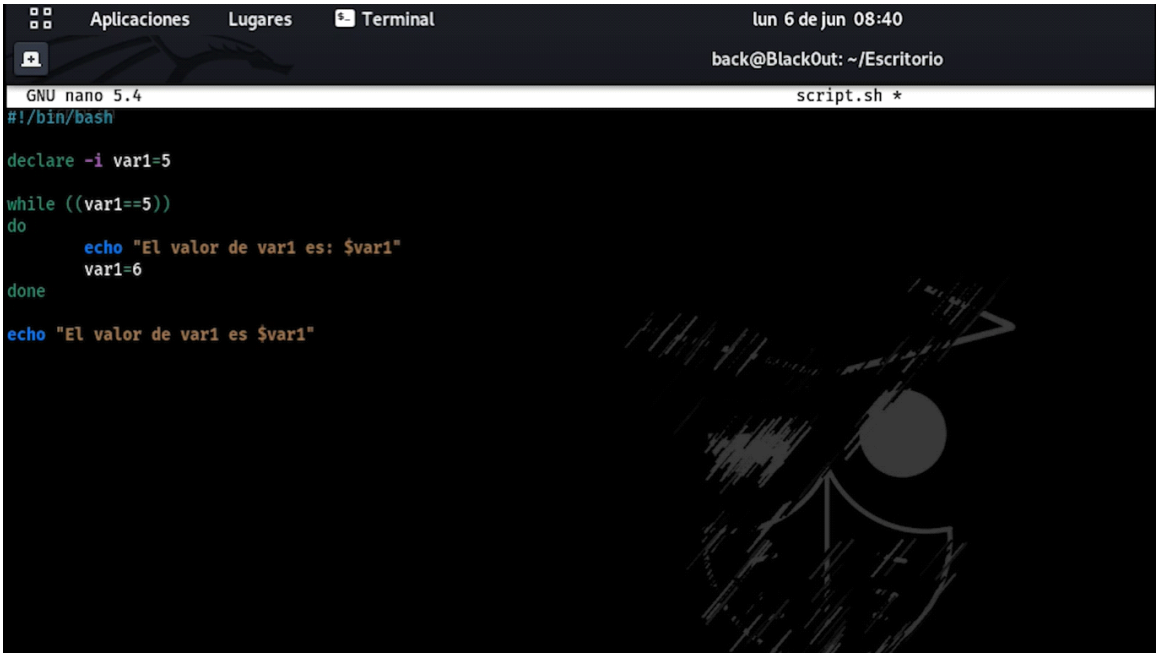
Echo $x

En fin, la aritmética pierde los redondeos absolutamente en este caso

Sentencias aritméticas de control de flujo

Las expresiones aritméticas pueden ser utilizadas en las distintas sentencias de control de flujo, en cuyo caso la expresión va entre dobles parentisis, pero sin el $ delante, por ejemplo el if aritmético tendría la forma:

O el while aritmético:



En el caso pasado declaramos var1 y le asignamos el valor 5 (es una variable entera "-i", posteriormente con el bucle while mientras dicha variable sea igual a 5 queremos que se ejecute toda la sintaxis dentro del while, que no es mas que un echo y un nuevo valor en el que var1 ahora sera 6, por lo que saldra del bucle al intentar entrar en la segunda iteracion al no cumplirse la condicion, por lo que imprimira el echo fuera del bucle "El valor de var1 es $var1".

Explicación de un final abrupto: ¿Por qué final, por qué abrupto?

Pues llegados a este punto si has seguido al pie de la letra todas las instrucciones del curso, lo más probable es que ya tengas ciertas nociones básicas pero bien fundamentadas, sobre todo sobre la práctica en bash. El curso aun necesitaba más contenido que quedara pendiente y el libro no concluirá jamás. Pero dado los pocos libros prácticos que existen me parece que este cubre una cantidad de conocimientos bastante sólida hasta este punto, aunque como siempre a mí me ha dejado insatisfecho.

No obstante espero que en realidad hayan aprendido de este libro todo lo que he intentado enseñarles y que salgan de aquí con una visión de Linux mucho mas empática que antes de conocerlo.

Si ha sido posible que aunque inconcluso, este libro haya salido a la luz de manera gratuita, ha sido gracias al interés que han mostrado en recibir la información, pues no es que gane nada con esto, simplemente vi que al hablar de scripting en bash la mayoría de personas saltaban para pedirlo, luego buscando me fije que el material gratuito era escaso o casi nulo, por lo que espero haberles podido ayudar. Sin más, reiterar a quienes me siguen que os quiero y que espero que con su apoyo pueda llegar a crecer y hacer llegar un mejor material en algún futuro.

Y sígueme si no lo haces, si me sigues deja de seguirme, ¿Por qué?, porque si :P

youtube: https://youtube.com/c/black0utx

twitter: https://twitter/Black0utq

Comunidad de Discord: https://discord.gg/JSAW5nTMDB

Canal de Telegram: https://t.me/B14ck0u7

Grupo de Telegram: https://t.me/black0utx