# CURSO HACKING ÉTICO CON PYTHON DESDE CERO

## VIRTUALBOX

Descargar el programa y el **VirtualBox extensión pack**=>**All supported platform**.

Para instalar el **extensión pack**, sólo hay que hacer doble clic sobre la extensión que hemos descargado.

## KALI LINUX

Se puede descargar desde **Kali.org/downloads** y dentro de la ´pagina, hay que buscar el específico para **VirtualBox**. Una vez dentro, hay que seleccionar **Kali Linux VirtualBox Images** y luego se elige la versión adecuada.

El archivo que se descarga tiene extensión **.ova**; haciendo doble clic sobre él, se instala en nuestro **VirtualBox**.

Ahora hay que configurar Kali:

1. Ir a **Settings.**
2. Elegimos **Network**.
3. En el desplegable **Attached to** hay que seleccionar **Bridge Adapter**.
4. En el desplegable **Name**, hay que elegir la tarjeta de red.

Al arrancar Kali, siempre se va a pedir un usuario y una contraseña; en ambos casos es **Kali**.

### Actualización de Kali

Una vez instalado el sistema operativo, hay que actualizarlo; para ello, hay que tener acceso como administrador usando el comando **sudo** delante de otro comando. En el caso de las actualizaciones sería:

*sudo apt-get upgrade*

y después:

*sudo apt-get update*

Si en el proceso del **upgrade** da errores, se hace el **update** y luego se hace el **upgrade**. Por útimo, es conveniente hacer otro **update**.

## INSTALACIÓN TERMINATOR

La instalación de esta aplicación se realizaría escribiendo:

*sudo apt-get install terminator*

Este programa es un terminal que permite correr distintos terminales dentro de él. Es muy útil para hacer ataques, ya que será necesario abrir varios terminales.

Para configurarlo:

1. Ejecutar **terminator**.
2. Clic derecho sobre la ventana.
3. En el menú se selecciona **Preferences**.
4. Es recomendable cambiar a **Infinite Scrollback** en **Profiles/Scrolling**.

## COMANDO PWD (PRINT WORKIN DIRECTORY)

Muestra el directorio en el que nos encontramos.

## COMPROBAR VERSION DE PYTHON

Sólo hay que:

1. Abrir un terminal.
2. Escribir **Python**.
3. Para salir, se escribe **exit()**.

## INSTALACIÓN DEL IDE VISUAL STUDIO CODE

1. Abrir un navegador.
2. En Google, buscar: *visual studio code for Linux*.
3. Descargarlo en formato **Debian** (**.deb**)
4. Seleccionar **Save File.**
5. Ir a la carpeta **Downloads** desde un terminal.
6. Instalar el archivo **.deb**:

*sudo dpkg -i <nombre del archivo>*

1. Ahora hay que ejecutarlo para instalarle una extensión:
   1. Ir a la rueda dentada de abajo a la izquierda.
   2. Seleccionar **Extensions**.
   3. Elegir **Python** e instalarla.

Dentro de **Visual Studio Code**, abajo, hay la opción **Terminal** que lo que hace es presentarnos un terminal. Desde éste es donde se van a ejecutar nuestros comandos y programas.

Se puede tener tantos terminales como se necesiten pinchando en el **+** que hay en la parte superior derecha del terminal.

## INSTALAR PIP

Para sabir si **pip** está instalado:

1. Abrir **Visual Studio Code**.
2. En su terminal, escribimos: *pip*.

Si no está instalado, dará un mensaje de error y la instrucción necesaria para instalarlo.

Este programa permite instalar paquetes de **Python** de la siguiente forma:

*pip install <nombre del paquete>*

## INSTALAR PAQUETE PYLINT

En un terminal de **Visual Studio Code**, hay que escribir:

*pip install pylint*

## INSTALACIÓN DE NET TOOLS

1. Abrir un terminal fuera de **Visual Studio Code**.
2. Escribir:

*sudo apt-get install net-tools*

Una vez instalado, se puede usar el comando **ifconfig**. Este comando permite ver la configuración de nuestra tarjeta de red.

## CÓMO CAMBIAR LA MAC A NUESTRA TARJETA DE RED (mac\_changer.py)

Lo primero que se podría hacer es ver la configuración de nuestra tarjeta de red:

*# Indicamos dónde se encuentra nuestro intérprete de Python (se puede omitir)*

*!/usr/bin/env Python*

*# Importamos* ***subprocess****, que permite correr comandos desde el código del programa*

*import subprocess*

*# Ejecutamos ifconfig*

*subprocess.call(“ifconfig”,shell=True)*

Para ejecutar un programa, solo hay que ir a la carpeta en donde se encuentra y escribir:

*sudo python3 <nombre del programa incluyendo la extensión>*

Esto se puede hacer desde un terminal abierto en **Visual Studio Code**.

Para cambiar la **MAC** de un dispositivo, habría que escribir el siguiente código:

En la primera llamada a **ifconfig**, se para (**down**) el dispositivo (**eth0**) al que queremos cambiar la **MAC**. En este caso, se la queremos cambiar a la tarjeta de red principal.

*# Indicamos dónde se encuentra nuestro intérprete de Python (se puede omitir)*

*!/usr/bin/env Python*

*# Importamos* ***subprocess****, que permite correr comandos desde el código del programa*

*import subprocess*

*# Ejecutamos ifconfig*

*subprocess.call(“ifconfig eth0 down”,shell=True)*

*subprocess.call(“ifconfig eth0 hw ether 00:11:22:33:44:55”,shell=True)*

*subprocess.call(“ifconfig eth0 up”,shell=True)*

En la siguiente línea, se cambia la **MAC** indicando el dispositivo afectado (**eth0**), diciéndolo que se va a efectuar un cambio en el hardware (**hw**) y que el cambio se va a realizar sobre la **MAC** de la tarjeta de red (**ether**).

En la última línea, se activa (**up**) de nuevo el dispositivo (**eth0**) al que se le ha cambiado la **MAC**.

Con **eth0** se identifican las redes con cable, las redes wifi se identifican con **wlan0**.

El código se puede refinar un poco más:

Si queremos que a las variables **interface** y **new\_mac** les de valores el usuario en tiempo de ejecución:

En algunas versiones de **python**, hay que sustituir **raw\_input** por **input**.

Este programa no es seguro, porque si por ejemplo, en el **input** de la **interface** escribimos: *eth0;ls;pwd*, le estamos diciendo al programa que ejecute los comandos **ls** y **pwd** además de asignar **eth0** a la variable **interface**. Esto es así, porque con cada **;** se le está indicando que se ejecute el comando que le sigue debido a que hemos puesto en cada *subprocess.call* el parámetro *shell=True.*

*# Indicamos dónde se encuentra nuestro intérprete de Python (se puede omitir)*

*!/usr/bin/env Python*

*# Importamos* ***subprocess****, que permite correr comandos desde el código del programa*

*import subprocess*

*interface = “eth0”*

*new\_mac = “00:11:22:33:44:55”*

*print(“Cambiando dirección MAC para “ + interface + “ a “ + new\_mac)*

*# Ejecutamos ifconfig*

*subprocess.call(“ifconfig “ + interface + “ down”,shell=True)*

*subprocess.call(“ifconfig “ + interface + “ hw ether “ +new\_mac,shell=True)*

*subprocess.call(“ifconfig “+ interface + “ up”,shell=True)*

Para solucionarlo, vamos a usar listas como parámetros de *subprocess.call*:

*# Indicamos dónde se encuentra nuestro intérprete de Python (se puede omitir)*

*!/usr/bin/env Python*

*# Importamos* ***subprocess****, que permite correr comandos desde el código del programa*

*import subprocess*

*interface = raw\_input(“Interface: “)*

*new\_mac = raw\_input(“Nueva MAC: “)*

*print(“Cambiando dirección MAC para “ + interface + “ a “ + new\_mac)*

*# Ejecutamos ifconfig*

*subprocess.call(“ifconfig “ + interface + “ down”,shell=True)*

*subprocess.call(“ifconfig “ + interface + “ hw ether “ +new\_mac,shell=True)*

*subprocess.call(“ifconfig “+ interface + “ up”,shell=True)*

Si en vez de usar los **inputs** queremos introducir estos valores en la línea de comando como parámetros de nuestro programa, hay que:

1. Importar el paquete **optparse**. Este paquete permite crear los argumentos.
2. Crear una variable para el argumento:

*argumentos = optparse.OptionParser()*

1. La variable **argumentos** es la que va a permitir vincular un comando, por ejemplo **-i**, a una interfaz en nuestro caso:

*# Indicamos dónde se encuentra nuestro intérprete de Python (se puede omitir)*

*!/usr/bin/env Python*

*# Importamos* ***subprocess****, que permite correr comandos desde el código*

*del programa*

*import subprocess*

*interface = raw\_input(“Interface: “)*

*new\_mac = raw\_input(“Nueva MAC: “)*

*print(“Cambiando dirección MAC para “ + interface + “ a “ + new\_mac)*

*# Ejecutamos ifconfig*

*subprocess.call([“ifconfig “, interface, “ down”])*

*subprocess.call([“ifconfig “, interface, “hw”, ”ether “, new\_mac])*

*subprocess.call([“ifconfig “, interface, “ up”])*

*argumentos.add\_option(“-i”, “—interface”, dest=”interface”, help=”Dispositivo al que cambiar su dirección MAC”*

En esta línea se añaden dos formas de indicar que a continuación viene un argumento (**-i** y **–interface**), con **dest** indicamos dónde se va a almacenar el valor que vamos a dar como argumento, y por último, con **help** añadimos una pequeña ayuda de lo que hace el argumento.

1. Ahora hay que hacer el **parse**:

*(options, arguments) = argumentos.parse\_args()*

En **options** se guardarán los valores y en **arguments** los argumentos. Por ejemplo: *(eth0, -i*)

1. Ahora a las variables se les da valor de la siguiente forma:

*interface = options.interface*

*new\_mac = options.mac*

En definitiva, el código quedaría así:

*# Indicamos dónde se encuentra nuestro intérprete de Python (se puede omitir)*

*!/usr/bin/env Python*

*# Importamos* ***subprocess****, que permite correr comandos desde el código*

*del programa*

*import subprocess*

*import optparse*

*argumentos = optparse.OptionParser()*

*argumentos.add\_option(“-i”, “—interface”, dest = “interfaces”, help=*

*“Dispositivo al que se le va a cambiar la dirección MAC”)*

*argumentos.add\_option(“-m”, “—mac”, dest = “mac”, help = “Nueva dirección MAC”)*

*(options, arguments)=argumentos.parse\_args()*

*interface = options.interfaces*

*new\_mac = options.mac*

*print(“Cambiando dirección MAC para “ + interface + “ a “ + new\_mac)*

*# Ejecutamos ifconfig*

*subprocess.call([“ifconfig “, interface, “ down”])*

*subprocess.call([“ifconfig “, interface, “hw”, ”ether “, new\_mac])*

*subprocess.call([“ifconfig “, interface, “ up”])*

Este código aún se podría mejorar más si usamos una función para hacer el cambio de **MAC**:

*# Indicamos dónde se encuentra nuestro intérprete de Python (se puede omitir)*

*!/usr/bin/env Python*

*# Importamos* ***subprocess****, que permite correr comandos desde el código del programa*

*import subprocess*

*import optparse*

*# Definimos la función*

*def change\_mac(interfaz, dir\_mac):*

*print(“Cambiando dirección MAC para “ + interfaz + “ a “ + dir\_mac)*

*# Ejecutamos ifconfig*

*subprocess.call([“ifconfig “, interfaz, “ down”])*

*subprocess.call([“ifconfig “, interfaz, “hw”, ”ether “, dir\_mac])*

*subprocess.call([“ifconfig “, interfaz, “ up”])*

*argumentos = optparse.OptionParser()*

*argumentos.add\_option(“-i”, “—interface”, dest = “interfaces”, help=*

*“Dispositivo al que se le va a cambiar la dirección MAC”)*

*argumentos.add\_option(“-m”, “—mac”, dest = “mac”, help = “Nueva dirección MAC”)*

*(options, arguments)=argumentos.parse\_args()*

*change\_mac(options.interfaces, options.mac)*

La obtención de los argumentos también se puede poner en una función, por lo que el código quedaría así:

Ahora se podría controlar que los argumentos que digite el usuario en la línea de comando sean correctos; para ello, la función **get\_arguments()** debería quedar así:

*# Indicamos dónde se encuentra nuestro intérprete de Python (se puede omitir)*

*!/usr/bin/env Python*

*# Importamos* ***subprocess****, que permite correr comandos desde el código del programa*

*import subprocess*

*import optparse*

*# Definimos las funciones*

*def get\_arguments():*

*argumentos = optparse.OptionParser()*

*argumentos.add\_option(“-i”, “—interface”, dest = “interfaces”, help=“Dispositivo al que se le va a cambiar la dirección MAC”)*

*argumentos.add\_option(“-m”, “—mac”, dest = “mac”, help = “Nueva dirección MAC”)*

*return argumentos.parse\_args()*

*def change\_mac(interfaz, dir\_mac):*

*print(“Cambiando dirección MAC para “ + interfaz + “ a “ +*

*dir\_mac)*

*# Ejecutamos ifconfig*

*subprocess.call([“ifconfig “, interfaz, “ down”])*

*subprocess.call([“ifconfig “, interfaz, “hw”, ”ether “, dir\_mac])*

*subprocess.call([“ifconfig “, interfaz, “ up”])*

*(options, arguments)=get\_arguments()*

*change\_mac(options.interfaces, options.mac)*

*# Indicamos dónde se encuentra nuestro intérprete de Python (se puede omitir)*

*!/usr/bin/env Python*

*# Importamos* ***subprocess****, que permite correr comandos desde el código del programa*

*import subprocess*

*import optparse*

*# Definimos las funciones*

*def get\_arguments():*

*argumentos = optparse.OptionParser()*

*argumentos.add\_option(“-i”, “—interface”, dest = “interfaces”, help=“Dispositivo al que se le va a cambiar la dirección MAC”)*

*argumentos.add\_option(“-m”, “—mac”, dest = “mac”, help = “Nueva dirección MAC”)*

*(options, arguments) = argumentos.parse\_args()*

*if not options.interfaces:*

*argumentos.error(“Por favor indica un dispositivo. Usa*

*--help para más información”)*

*elif not options\_mac:*

*argumentos.error(“Por favor indica una nueva dirección MAC.*

*Usa --help para más información”)*

*return options*

*def change\_mac(interfaz, dir\_mac):*

*print(“Cambiando dirección MAC para “ + interfaz + “ a “ +*

*dir\_mac)*

*# Ejecutamos ifconfig*

*subprocess.call([“ifconfig “, interfaz, “ down”])*

*subprocess.call([“ifconfig “, interfaz, “hw”, ”ether “, dir\_mac])*

*subprocess.call([“ifconfig “, interfaz, “ up”])*

*options = get\_arguments()*

*change\_mac(options.interfaces, options.mac)*

## IDENTIFICAR USUARIOS CONECTADOS A UNA RED

### Hacer un escaner de red (net\_scanner.py)

El objetivo de los escáneres de red es obtener las **MAC**s e **IP**s de los usuarios de una red.

Lo primero que hay que saber es la **puerta de enlace** de nuestra red:

*sudo route -n*

Lo siguiente, es iniciar el programa importando el paquete **scapy.all**. Después hay que crear una función a la que se le pasa como argumento una **IP**. A esta función se la llamadando como argumento la **IP** de la **puerta de enlace** seguido de **/24**, que indica que queremos escanear todas las **IP**s de ese rango.

El código quedaría así:

*import scapy.all as scapy*

*#from scapy.all import \**

*def escanea(ip):*

*scapy.arping(ip)*

*escanea("192.168.16.50/24")*

### Crear con scapy un REQUEST ARP(request\_arp.py)

En el protocolo **ARP**, para poder saber todos los usuarios conectados a una red, primero hay que preguntar quién tiene una **IP** determinada; entonces, el ordenador que esa **IP** responderá.

Debido a ello, habrá que dar los siguientes pasos:

1. Crear un **paquete ARP**:

*import scapy.all as scapy*

*#from scapy.all import \**

*def escanear(ip):*

*arp\_request = scapy.ARP(pdst = ip)*

La variable **ip** contendrá la dirección **IP** por la que estamos preguntando.

1. Se imprime el resultado del **arp\_request:**

*import scapy.all as scapy*

*#from scapy.all import \**

*def escanear(ip):*

*arp\_request = scapy.ARP(pdst = ip)*

*print(arp\_request.summary())*

1. Llamamos a la función dando como argumento la **IP** de la víctima:

*import scapy.all as scapy*

*#from scapy.all import \**

*def escanear(ip):*

*arp\_request = scapy.ARP(pdst = ip)*

*print(arp\_request.summary())*

*escanear(“192.168.16.5/24”)*

1. Ahora queremos ver las **MAC** de cada **IP** del rango. Para ello:
   1. Debemos obtener la **MAC** de destino; para hacerlo, en la función *escanear* se añaden dos nuevas líneas:

*import scapy.all as scapy*

*#from scapy.all import \**

*def escanear(ip):*

*arp\_request = scapy.ARP(pdst = ip)*

*print(arp\_request.summary())*

*broadcast = scapy.Ether()*

*scapy.ls(broadcast)*

*escanear(“192.168.16.5/24”)*

Ahora ya se tiene la **MAC** de destino, que es la que viene indicada en el campo **dst: DestMACField**.

* 1. Copiamos la **MAC** de destino.
  2. A continuación, se modifica la línea en donde se da valor a la variable **broadcast**, dándole como argumento la **MAC** de destino:

*import scapy.all as scapy*

*#from scapy.all import \**

*def escanear(ip):*

*arp\_request = scapy.ARP(pdst = ip)*

*print(arp\_request.summary())*

*broadcast = scapy.Ether(dst=”<MAC de destino>”)*

*scapy.ls(broadcast)*

*escanear(“192.168.16.5/24”)*

* 1. Mostramos el contenido de las variables **arp\_request** y **broadcast**:

*import scapy.all as scapy*

*#from scapy.all import \**

*def escanear(ip):*

*arp\_request = scapy.ARP(pdst = ip)*

*print(arp\_request.summary())*

*broadcast = scapy.Ether(dst=”<MAC de destino>”)*

*scapy.ls(broadcast)*

*arp\_request.show()*

*broadcast.show()*

*escanear(“192.168.16.5/24”)*

* 1. Ahora se puede concatenar el **arp\_request** y el **broadcast** de la siguiente forma:

*import scapy.all as scapy*

*#from scapy.all import \**

*def escanear(ip):*

*arp\_request = scapy.ARP(pdst = ip)*

*print(arp\_request.summary())*

*broadcast = scapy.Ether(dst=”<MAC de destino>”)*

*scapy.ls(broadcast)*

*arp\_request.show()*

*broadcast.show()*

*arp\_request\_broadcast = broadcast/arp\_request*

*escanear(“192.168.16.5/24”)*

* 1. Seguidamente, podemos crear dos variables: **respuestas** y **norepuestas**, por ejemplo. A estas variables se les da valor de la siguiente forma:

*import scapy.all as scapy*

*#from scapy.all import \**

*def escanear(ip):*

*arp\_request = scapy.ARP(pdst = ip)*

*print(arp\_request.summary())*

*broadcast = scapy.Ether(dst=”<MAC de destino>”)*

*scapy.ls(broadcast)*

*arp\_request.show()*

*broadcast.show()*

*respuesta, norespuesta = scapy.srp(arp\_request\_broadcast, timeout=1)*

*escanear(“192.168.16.5/24”)*

Esto hace que se recorra toda la red, preguntando a cada nodo quién tiene esa **IP**.

Así, ya se pueden ver los paquetes respondidos y as **IP**s y sus direcciones **MAC** correspondientes.

Como nos interesan sólo las respuestas, la línea que hemos añadido, podríamos dejarla así:

*import scapy.all as scapy*

*#from scapy.all import \**

*def escanear(ip):*

*arp\_request = scapy.ARP(pdst = ip)*

*print(arp\_request.summary())*

*broadcast = scapy.Ether(dst=”<MAC de destino>”)*

*scapy.ls(broadcast)*

*arp\_request.show()*

*broadcast.show()*

*respuesta = scapy.srp(arp\_request\_broadcast, timeout=1)[0]*

*escanear(“192.168.16.5/24”)*

* 1. Queremos ver más información sobre estos paquetes, por lo que, después de dar valor a la variable **respuesta**, hay que hacer un **for**:

*import scapy.all as scapy*

*#from scapy.all import \**

*def escanear(ip):*

*arp\_request = scapy.ARP(pdst = ip)*

*print(arp\_request.summary())*

*broadcast = scapy.Ether(dst=”<MAC de destino>”)*

*scapy.ls(broadcast)*

*arp\_request.show()*

*broadcast.show()*

*respuesta = scapy.srp(arp\_request\_broadcast, timeout=1)[0]*

*for element in respuesta:*

*print[element[1].show()]*

*escanear(“192.168.16.5/24”)*

Al ejecutar la aplicación, en la sección **ARP**, veremos varios campos interesantes:

* **psrc**: Indica la **IP** del cliente que envió el paquete.
* **hwsrc**: Es el **MAC** del cliente.
* **pdst**: **IP** del ordenador destino.
* **hwdst**: La **MAC** del ordenador destino.

De toda esta información sólo interesan los campos **hwscr** y **pscr**; por ello, se puede sustituir la instrucción:

*print[element[1].show()]*

Por otras dos:

*import scapy.all as scapy*

*#from scapy.all import \**

*def escanear(ip):*

*arp\_request = scapy.ARP(pdst = ip)*

*print(arp\_request.summary())*

*broadcast = scapy.Ether(dst=”<MAC de destino>”)*

*scapy.ls(broadcast)*

*arp\_request.show()*

*broadcast.show()*

*respuesta = scapy.srp(arp\_request\_broadcast, timeout=1)[0]*

*for element in respuesta:*

*print(element[1].psrc)*

*print(element[1].hwsrc)*

*escanear(“192.168.16.5/24”)*

De esta forma, se muestran todas las **IP**s conectadas a la red de la puerta de enlace que se ha especificado en:

*escanear(“192.168.16.5/24”)*

# WEB CTF: APRENDE HACKING JUGANDO – SOTHIS

## CIFRADO XOR

Es un tipo de encriptación basado en el uso de puertas lógicas para codificar y verificar la autenticidad de los datos. Es una operación como la suma o la multiplicación, pero que funciona con **bits** (1 y 0). Esta operación se usa en casi todos los algoritmos criptográficos.

Normalmente, una puerta lógica **XOR** tiene dos entradas binarias, aunque pueden ser más. Independientemente del número de entradas, siempre habrá una sola salida binaria.

Para saber la salida que se obtendrá con dos entradas, se crea la **tabla de la verdad**:

|  |  |  |
| --- | --- | --- |
| **ENTRADA A** | **ENTRADA B** | **SALIDA** |
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 1 | 0 |

Como se puede observar en la tabla, las salidas con valor 0 (nivel bajo) se producen cuando las entradas A y B son iguales; y son 1 (nivel alto) cuando éstas son distintas.

Esta técnica es muy vulnerable, por lo que se suele combinar con más herramientas y funciones.

Ejemplos de cifrados **XOR**:

Tabla . Cifrado Cadena-Clave (8Bits = 1 byte = 1 caracter)

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Representación | Valor ASCII | Valor decimal | | | | | | |
| Cadena | FISI | 70, 73, 83, 73 | 01000110 | 01001001 | | 01010011 | | 01001001 |
| Clave | U | 85 | 01010101 | | 01010101 | | 01010101 | 01010101 | |
| Resultado | !!, ∟, ♠, ∟ | 19,28,6,28 | 00010011 | | 00011100 | | 00000110 | 00011100 | |

Tabla . Cifrado Cadena-Clave (n caracteres)

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Representación | Valor ASCII | Valor decimal | | | | | | |
| Cadena | FISI | 70, 73, 83, 73 | 01000110 | 01001001 | | 01010011 | | 01001001 |
| Clave | 1234 (string) | 49, 50, 51, 52 | 00110001 | | 00110010 | | 00110011 | 00110100 | |
| Resultado | w{`} | 119,123,96,125 | 01110111 | | 01111011 | | 01100000 | 01111101 | |

Si tenemos la clave y el resultado de la tabla 2, podemos deducir cuál es la cadena original:

Tabla . Descifrado de XOR

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Representación | Valor ASCII | Valor decimal | | | | | | |
| Resultado | w{`} | 119,123,96,125 | 01110111 | | 01111011 | | 01100000 | 01111101 | |
| Clave | 1234 (string) | 49, 50, 51, 52 | 00110001 | | 00110010 | | 00110011 | 00110100 | |
| Cadena | FISI | 70, 73, 83, 73 | 01000110 | 01001001 | | 01010011 | | 01001001 |

## CYBERCHEF

Herramienta basada en web que nos [permitirá analizar y decodificar datos fácilmente](https://www.redeszone.net/tutoriales/seguridad/ver-hash-archivos-hashcheck/), sin necesidad de utilizar múltiples herramientas ni utilizar lenguajes de programación para realizar diferentes acciones.

Gracias a que esta herramienta está basada en web, podremos ejecutarla directamente de manera online o descargarnos el archivo .htm y abrirlo en el navegador con nuestro equipo. El motivo por el que esta herramienta existe, es que en el mundo de la informática hay una gran cantidad de formatos de datos, algoritmos de cifrado y descifrado de datos, compresores, algoritmos de hash y un largo etcétera. CyberChef nos permitirá pasar de unos datos a otros fácilmente, sin necesidad de saber internamente cómo es un determinado formato de datos.

Esta herramienta tiene una interfaz web muy fácil de entender y manejar, ideal para que cualquier pueda utilizarla. Con esta herramienta podremos realizar el típico copiar y pegar para introducir o sacar los datos, asimismo, también permite cargar un archivo arrastrándolo a la casilla de «Input» y extraer los datos pulsando sobre «Save to file».

CyberChef permite realizar más de 100 operaciones entre datos, por ejemplo es capaz de codificar una cadena en formato Base64, y también es capaz de decodificar una cadena Base64 en texto plano. Es capaz de mostrarnos las direcciones IP de un determinado rango, como hace cualquier calculadora de IP, también es capaz de trabajar con direcciones IPv6 y mostrarnos si una determinada IP es válida para túneles Teredo.

Otras funciones que incorpora es realizar la función hash de los datos introducidos, soporta los típicos algoritmos como MD5, SHA1, SHA224, SHA256, SHA512 e incluso el nuevo SHA3. También es capaz de cifrar datos con el algoritmo AES y descifrarlos, siempre y cuando introduzcamos la contraseña correcta. Otros algoritmos de cifrado simétrico que soporta son Blowfish, DES, 3DES e incluso implementa PBKDF2.

### Ejemplo de uso de funciones hashing: SHA512

Si por ejemplo queremos realizar la función hash de una cadena de texto, simplemente tenemos que arrastrar la «receta» Hashing/SHA512 a la parte central de la interfaz gráfica. A continuación, introducimos un texto en el «Input» y automáticamente hará el hash de esa información.

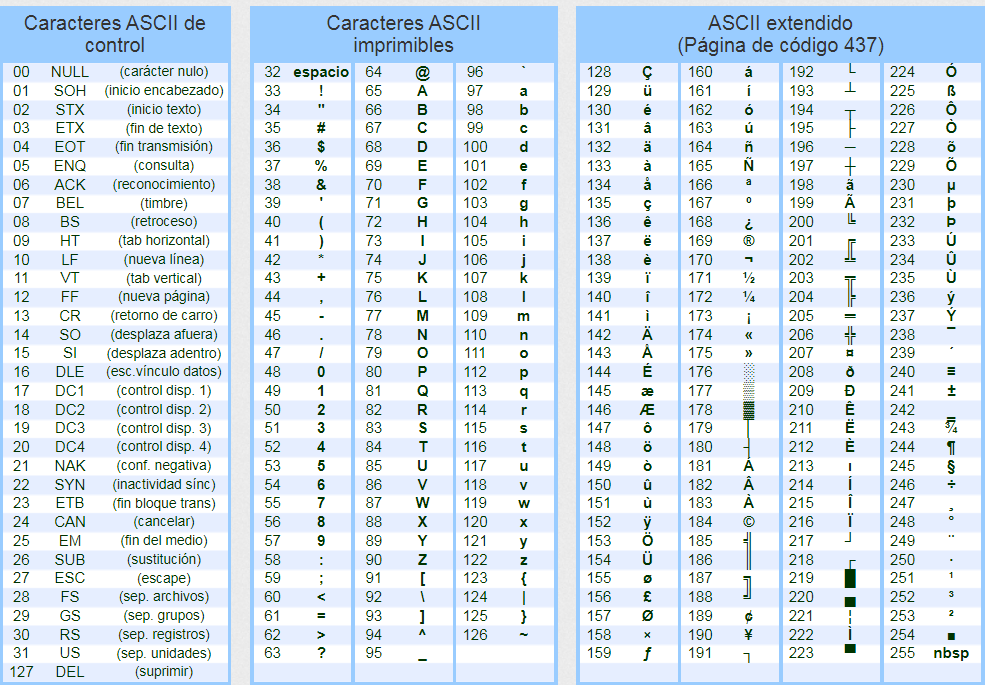
### Ejemplo de uso de cifrado de datos con AES

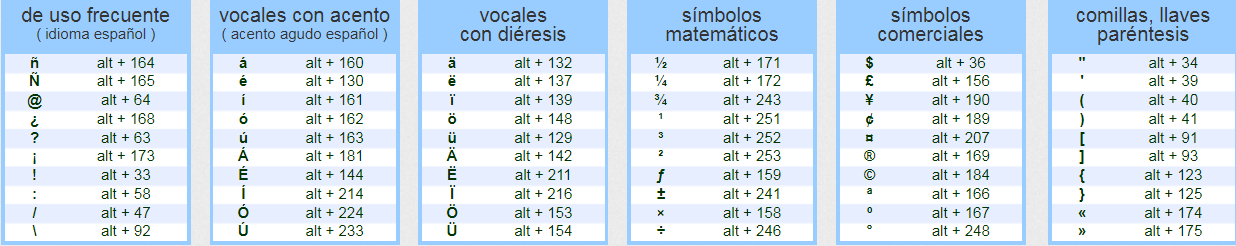
Si por ejemplo queremos realizar la función de cifrado AES de una cadena de texto, simplemente tenemos que arrastrar la «receta» Encryption/Decryption/ AES Encrypt a la parte central de la interfaz gráfica. A continuación, deberemos introducir la contraseña o Passphrase, el IV (opcional) y la Salt (opcional). También podremos seleccionar el modo de cifrado por bloques, por defecto es CBC, el padding y en qué formato queremos que nos de la salida.

### Ejemplo de uso de subnetting IPv4

Si seleccionamos Networking/ Parse IP Range podremos introducir una dirección de red y nos dirá el rango de direcciones IP que tenemos disponibles.

## CÓDIGOS ASCII





## COMANDOS ÚTILES

### base64

Codifica o decodifica una cadena de texto o un archivo.

Para codificar una cadena de caracteres mostrando el resultado en pantalla:

*echo “<cadena de caracteres>” | base64*

Si lo que queremos es guardar el resultado en un archivo:

*echo “<cadena de caracteres>” | base64 <nombre del archivo>*

Para decodificar, se usa la opción **-d**.

Para decodificar una cadena codificada y mostrar el resultado por pantalla:

*echo “<cadena codificada>” | base64 -d*

Si el resultado se quiere guardar en un fichero:

*echo “<cadena codificada>” | base64 -d <nombre del archivo>*

Si lo que se quiere es codificar un archivo y mostrar el resultado en pantalla:

*base64 <nombre del archivo>*

Si el resultado se quiere guardar en un archivo:

*base64 <nombre del archivo> <nombre del archivo destino>*

Para decodificar un archivo y mostrar el resultado en pantalla:

*base64 -d <nombre del archivo>*

Para guardar el resultado en un archivo:

*base64 -d <nombre del archivo> <nombre del archivo destino>*

### cat

Entre otros usos, se utiliza para ver el contenido de los archivos; en este caso, su sintaxis sería:

*cat <nombre del archivo>*

### cron

Es un demonio que se ejecuta al inicio del sistema y nos ayuda a controla las tareas que se van ejecutando. Permite programar la ejecución de aplicaciones.

Si se quiere trabajar con **cron**, se puede hacer a través de **crontab** que es un archivo de texto donde se listan todas las tareas que deben ejecutarse y cuándo.

En **crontab** se pueden especificar los siguientes intervalos de tiempo:

|  |  |
| --- | --- |
| **Campos** | **Valores permitidos** |
| minute | 0-59 |
| hour | 0-23 |
| day of month | 1-31 |
| month | 1-12 |
| day of week | 0-7 |

Cada usuario tiene su propio archivo **crontab**.

Para agregar una tarea a **crontab** se escribe lo siguiente:

*crontab -e*

El archivo tiene la siguiente estructura:

*\* \* \* \* \* comando a ejecutar*

Cada uno de los asteriscos significan:

* Minuto
* Hora
* Día del mes
* Mes
* Día de la semana

Si sólo se ponen asteriscos significa que el comando se debe ejecutar continuamente con un intervalo de un minuto.

En Linux existen algunas cadenas de texto reservadas para ejecutar procesos durante determinados periodos en el propio Crontab, esto nos será de mucha ayuda:

* **@reboot**: Ejecuta una vez y nada más iniciarse el equipo.
* **@yearly**: ejecuta sólo una vez al año: 0 0 1 1 \*
* **@monthly**: ejecuta una vez al mes y el primer día: 0 0 1 \* \*
* **@weekly**: todas las semanas, el primer minuto de la primera hora de la semana: 0 0 \* \* 0.
* **@daily**: todos los días a las 12 de la noche: 0 0 \* \* \*
* **@midnight**: Tiene el mismo efecto que el anterior.
* **@hourly**: todas las horas durante su primer minuto: 0 \* \* \* \*

### cut

Este comando borra secciones, campos o caracteres de la salida de un comando o de cada una de las líneas de un fichero.

Antes de nada hay que saber que las vocales acentuadas, se consideran dos caracteres.

Si, por ejemplo, queremos mostrar el cuarto carácter de cada línea de un fichero, se especificaría así:

*cut -c 4 <nombre del archivo>*

### diff

Compara archivos y directorios. Su sintaxis es:

*diff <fichero1> <fichero2>*

El resultado que se muestra es:

* Número de línea del primer fichero en el que hay cambios; en el caso de haber varias líneas, sus números se separan por comas.
* Una letra que representa:
  + **a**: Para añadir.
  + **c**: Para cambiar.
  + **d**: Para borrar.
* Número de línea del segundo archivo.
* Símbolo **<** seguido por el texto original del primer fichero.
* Símbolo **>** seguido por el texto modificado del segundo fichero.

Por ejemplo:

*4c4*

*< texto original*

*> Texto modificado*

### du

Indica la cantidad estimada de espacio en disco usado por ciertos archivos o directorios. La sintaxis es:

*du [opciones] <nombre del archivo o directorio>*

### file

Se usa para saber el tipo de un fichero. La sintaxis es:

*file [opciones] <nombre del archivo o directorio>*

Si queremos saber qué tipo de fichero es un archivo cuyo nombre es o empieza por un guión:

*file -- -[<nombre del fichero>]*

Los guiones dobles se usan en muchos comandos para poder trabajar con archivos cuyo nombre es o empiezan por un guión.

Para ver si dentro de una carpeta hay algún archivo cuyo contenido sea legible por un usuario:

*file \* --mime-type*

Si queremos buscar archivos cuyo nombre sea un guión o empiece por este carácter:

*file -- -\* --mime-type*

Para ver qué tipos de archivos hay en una carpeta y sus subcarpetas desde fuera de la primera:

*file \*/\**

Para ver los tipos de archivo que hay en una carpeta y en sus subcarpetas incluyendo los archivos cuyo nombre empiecen por un guión o sea sólo un guión:

*file -- \* (Esta instrucción se debe digitar dentro de la carpeta)*

### find

Busca archivos y directorios. La sintaxis es:

*find <ruta> [opciones] [acciones]*

Para buscar sólo archivos ejecutables:

*find -executable*

Para buscar archivos dentro de una carpeta y sus subcarpetas:

*find . [opciones]*

Para buscar archivos ocultos:

*find -type f -name “.\*” -ls*

La opción **-type f**, se usa para indicar que se buscan archivos; si se quisiera buscar sólo directorios la opción sería: **-type d**.

Si además, la búsqueda anterior se quiere restringir a un tamaña de archivo:

*find -type f -name “.\*” -size [+/-]<tamaños><unidad de medida>*

Para ver qué archivos de una carpeta y sus subcarpetas tienen un tamaño específico, por ejemplo, más de 1K:

*find . -size [+/-]<tamaño><unidad de medida>*

Las unidades de medida son:

|  |  |
| --- | --- |
| CARÁCTER | UNIDAD DE MEDIDA |
| C | Byte |
| K | Kilobyte |
| M | Megabyte |
| G | Gigabyte |

Los símbolos **+** y  **-** se usan para indicar que los archivos que se deben listar sea mayores o menores que el tamaño indicado.

La sintaxis para la búsqueda de archivos de un determinado tamaño es la misma para buscar archivos ocultos y no ocultos.

### git

Es un sistema de control de versiones diseñada para manejar proyectos. Este nos permite clonar proyectos creados para poder usarlos o modificarlos.

### grep

Se usa para buscar un archivo que contenga el texto especificado. También se puede usar para buscar un texto en la información mostrada tras la ejecución de un comando. Un ejemplo de este segundo caso podría ser:

*find -ls | grep <texto>*

En este caso, se realiza una búsqueda de todos los ficheros y se muestran sólo aquellos que en la salida de la búsqueda contenga **<texto>**.

### hostname

Se usa para saber el nombre de nuestro ordenador.

### jobs

Muestra los procesos que se están ejecutando en segundo plano.

### less

Muestra página a página la salida por pantalla de cualquier comando.

### ls

Se usa para listar los archivos y directorios que hay en una carpeta. En el listado se pude ver que hay nombres que tiene distintos colores, los más comunes son:

* **Verdes**: Archivos ejecutables.
* **Azul**: Carpetas.
* **Celeste**: Enlaces simbólicos.
* **Rojo**: Archivos comprimidos.
* **Magenta**: Archivos de imágenes.
* **Negro o Blanco**: Archivos normales.

Si queremos que en ese listado aparezcan también los archivos y carpetas ocultos, hay que usar este comando seguido de las opciones **-la**:

*ls -la*

Si lo que se pretende es localizar un archivo por su tamaño:

*ls -s <tamaño>*

### nc

Mediante este comando podemos conectarnos a los puertos **TCP/UDP** de un **host**. Por defecto, se usa **TCP**.

La sintaxis es:

*nc <opciones> <host> <puerto>*

También se puede usar este comando para:

* Escanear puertos.
* Conectar con un servidor.
* Crear un servidor.
* Quedar a la escucha por un puerto dado en una máquina local.

Algunas opciones son:

* **l**: Abre un puerto y lo mantiene a la escucha
* **n**: Obliga a introducir una **IP**, no nombres **DNS**.
* **p**: Permite especificar el puerto al que conectarse.
* **v**: Se usa para mostrar información acerca de la conexión.

### nmap

se usa para escanear redes para auditorías de seguridad y pruebas de penetración. Se usa para:

* Detectar problemas de conectividad en la red.
* Escaneo de puertos.
* Detectar las direcciones **MAC**.
* Saber el sistema operativo de una máquina.
* Conocer la versión de un servicio.
* Etc.

Un ejemplo de uso sería:

*nmap <URL>*

Si se usa como usuario **no root**, que no tiene privilegios de paquetes sin procesar, es ejecuta un escaneo de conexión **TCP**. Las opciones **-sT** están activadas de forma predeterminada en el modo sin privilegios.

Las opciones más usadas son:

* **TCP SYN (-sS)**: Acelera el escaneo y funciona con todas las pilas **TCP** compatibles.
* **-v ó -vv**: Se usan para obtener de salida más información:

*nmap -v <URL>*

*nmap -vv <URL>*

* **-sU**: Hace un escaneo **UDP**:

*sudo nmap -sU <URL>*

Todas las opciones deben ir precedidas de uno o dos guiones, si no los tienen, son direcciones de servidores objetivos.

Para pasar al comando dos o más direcciones, se pone una coma entre cada una de ellas para separarlas:

*nmap <URL1>, <URL2>*

Se puede especificar la notación **CDIR** para especificar un rango de red:

*nmap <URL>****/24***

Para especificar un rango de octetos se usa un guión:

*nmap 192.168.****10-12****.1*

En este ejemplo, se escanearían las **IPs** 192.168.10.1, 192.168.11.1 y 192.168.12.1.

Otra forma de hacer esto mismo es usando comas para separar cada **IP**

*nmap 192.192.168.****10,11,12****.1*

Para asegurarnos de haber especificado el host correcto antes de escanear, se usa la opción **-sL** que sólo enumera los destinos sin escanear.

El comando **nmap** sólo escanea los mil puertos más populares, que no tienen porqué se los mil primeros. Para buscar en todos los puertos, se usa la opción **-p**:

*nmap -p <URL>*

El estado de los puertos puede ser:

* **open**: El programa que se está ejecutando en el puerto responde a la solicitud.
* **close**: Ningún programa se ejecuta en el puerto, pero el **host** responde a las solicitudes.
* **filtered**: El **host** no responde a la solicitud.

Si se quiere usar para escanear un puerto determinado:

*nmap -p <número de puerto> <URL>*

Para especificar más de un puerto, éstos se separan por una coma:

*nmap -p <número de puerto1>, <número de puerto2>, …, <número de puertoN> <URL>*

Para seleccionar un rango de puertos:

*nmap -p* ***<número de puerto1>-<número de puertoN>*** *<URL>*

### nslookup

Da información de la red para nuestra máquina.

### sort

Ordena numérica o alfabéticamente el contenido de un fichero de texto:

*sort <nombre del archivo>*

Si además, queremos eliminar líneas duplicadas, se usa la opción **-u**:

*sort -u <nombre del archivo>*

### ssh

Establece una comunicación remota segura.

La sintaxis básica es:

*ssh <usuario>@<URL del host> -p <puerto>*

Para conectarnos a un servidor mediante **SSH** y una clave privada, hay que hacer lo siguiente:

*ssh <usuario>@<URL del host> -i <ruta donde se encuentra el archivo de la clave privada>/<archivo con la clave privada>*

En la misma línea de ejecución de este comando se pueden ejecutar otras instrucciones de la siguiente forma:

*ssh <usuario>@<URL> -p <Puerto> “<otra instrucción”*

Por ejemplo:

*ssh* [*bandit18@banditlabs.othewire.org*](mailto:bandit18@banditlabs.othewire.org) *-p 2220 “ls -al”*

ó

*ssh* [*bandit18@banditlabs.othewire.org*](mailto:bandit18@banditlabs.othewire.org) *-p 2220 “cat readme”*

### string

Muestra cadenas de texto contenidos en ficheros binarios. La sintaxis es:

*strings <nombre del archivo>*

### tr

Este comando reemplaza caracteres en los archivos de texto; lee la entrada y envía los resultados. El archivo original no se modifica, para guardar el resultado se haría:

*tr ‘<carácter o caracteres a reemplazar>’ ‘<carácter o caracteres por los que se reemplazan> < <fichero origen> > <fichero destino>*

Hay que tener en cuenta que cada carácter del primer argumento será cambiado por el carácter que le corresponda en el segundo argumento; por ejemplo:

*tr ‘abc’ ‘tuv’ < datos*

Cambia las letras ‘a’ por una ‘t’, las ‘b’ por una ‘u’ y las ‘c’ por una ‘v’ que estén contenidas en el archivo **datos**.

### uniq

Este comando muestra u omite líneas repetidas. A menudo se usa junto a **sort**.

Sin opciones elimina los duplicados, mientras que con la opción **-d**, veremos la lista de duplicados.

Este comando solo funciona si las líneas contenidas en el archivo están ordenadas.

Usando la opción **-u** sólo muestra líneas únicas.

### xxd

Crea un volcado hexadecimal de un archivo. También puede convertir un volcado hexadecimal a su forma binaria inicial. Su sintaxis es:

*xxd [opciones] [archivo]*

Las opciones son:

* **-p**: Muestra el archivo en hexadecimal plano.
* **-r**: Transforma de hexadecimal a binario.

## FUNCIONES ÚTILES DE PHP

* **base64\_decode**: Decodifica un valor codificado en **Base 64**.
* **base64\_encode**: Codifica un **string** en **Base 64**.
* **bin2hex**: Convierte un **string** a **hexadecimal**.
* **hex2bin**: Convierte a **string** o **binario** un valor **hexadecimal**.
* **passthru**: Permite ejecutar comandos de un sistema operativo tal y como hacen también las funciones **exec** y **system**.
* **preg\_match**: Comprueba si hay alguna coincidencia en una cadena de caracteres según una expresión regular.
* **strrev**: Le da la vuelta a un **string**. Por ejemplo, el resultado de la instrucción *strrev(“Hello World”)* sería: *dlroW olleH*.

## ARCHIVOS BASH

Son archivos que contienen un **script** con el que se ejecutan diversos comandos.

Este archivo siempre debe empezar por la cadena *#! /bin/bash*; esto indica dónde se encuentra el ejecutable del **bash.** Y luego en cada línea se van ejecutando comandos.

## ARCHIVOS CON ESPACIOS EN SU NOMBRE

Para crearlos hay que escribir:

*touch <nombre del archivo>*

Para poder ver su contenido hay que poner una **\** antes de cada espacio en blanco. Por ejemplo:

*cat espacios\ en \mi\ archivo*

## ARCHIVOS CON NOMBRE CON GUION

Los archivos cuyo nombre es **–** y aquellos que empiezan por un guión (por ejemplo: **-prueba**), no puede verse su contenido usando el comando **cat**, para verlo se tendría que usar la siguiente sintaxis:

*cat < <nombre del archivo>*

Para crear un archivo de este tipo, hay que hacer lo siguiente:

*echo <contenido> > -[nombre del fichero]*

Donde el guión será el primer carácter del nombre del archivo.

## ARCHIVOS CON SETUID ACTIVADO

Son archivos que se identifican porque al hacer un **ls**, se ve que en los permisos se añade una **s**:

*-rw****s****r-xr-x 1 root root 27920 ago 15 22:45 /home/<ejecutable>*

Normalmente, esta característica se active en archivos ejecutables y se usa para que usuarios normales puedan ejecutarlos y obtener los privilegios del usuario que posee ese archivo. Si tiene privilegios de administrador, la aplicación puede hacer tareas que los usuarios normales no pueden.

Por ejemplo, en el fichero **/usr/bin/passwd**, se usa el **setuid** para que cualquiera pueda cambiar su contraseña de forma controlada. También se consigue que el usuario pueda escribir en el fichero de contraseñas **/etc/shadow** pero sin tener que dar permisos de escritura al fichero para evitar agujeros de seguridad.

Esta característica también se puede aplicar a directorios. En este caso, lo que hace es que todos los archivos y subdirectorios que haya dentro, van a perteneces al grupo del dueño del directorio y no al grupo del usuario que crea el archivo o subdirectorio.

Para añadir el **setuid** a nivel de usuario:

c*hmod u+s <ruta y nombre del fichero>*

Para añadir el **setuid** a nivel de grupo:

c*hmod g+s <ruta y nombre del fichero>*

Para eliminarlo a nivel de usuario:

c*hmod u-s <ruta y nombre del fichero>*

Para eliminarlo a nivel de grupo:

c*hmod g-s <ruta y nombre del fichero>*

## DIRECTORIOS PADRES Y ACTUAL (O DE TRABAJO)

Para referirnos al directorio padre se usa ‘**..**’; por ejemplo:

*cd ..* ó *../home*

Para referirnos al directorio en donde nos encontramos o directorio de trabajo, se usa ‘**.**’; por ejemplo:

*./<nombre de archivo>*

## LOCALHOST

**IPv4** reserva todo el bloque de direcciones *127.0.0.0/8* para fines de **loopback**. Esto significa que cualquier paquete enviado a una de estas direcciones, se devuelve. La dirección *127.0.0.1* se usa para el tráfico de **loopback IPv4**; el resto de direcciones no son compatibles en todos los sistemas operativos. Sin embargo, se pueden usar para configurar múltiples aplicaciones de servidor en el **host**, todas escuchando en el mismo puerto. **IPv6** asigna una sola dirección para **loopback**: *::1*.

La resolución del nombre **localhost** a una o más direcciones **IP**, normalmente se configura mediante las siguientes líneas en el archivo **host** del sistema operativo:

*127.0.0.1 servidor local*

*::1 servidor local*

## OPERADORES

* **<<<**: Envía el texto que se indique a continuación del operador a la entrada del comando que se va a ejecutar en la misma línea de comando.

Por ejemplo:

*nc -p 3030 <<< [texto]*

* **&**: Hace que la ejecución de un comando se haga en segundo plano.

Por ejemplo:

*nc -p 3030 &*

## PIPES (TUBERÍAS)

Para coger el resultado de un comando y usarlo para ejecutar otro, se usan las **pipes** (**|**):

*sort <fichero> | tr -s ‘ ‘ | grep “hola”*

En este ejemplo, lo que se hace es:

1. Se ordena el contenido de **<fichero>**.
2. Se eliminan los espacios que sobran.
3. Busca dentro del **<fichero>** la palabra **hola**.

## PROTOCOLO SSH

Es un protocolo criptográfico de red para operar servicios de red de forma segura en una red que no lo es.

Se basa en una arquitectura cliente-servidor.

Tiene tres capas:

* **Capa de transporte**: Proporciona autenticación, confidencialidad e integridad del servidor.
* **Protocolo de autenticación del usuario**: Valida al usuario ante el servidor.
* **Protocolo de conexión**: Multiplexa el túnel encriptado en múltiples canales lógicos de comunicación.

Usa criptografía de clave pública para autenticar la computadora remota y, si es necesario, al usuario también.

Para usar este protocolo, se utiliza el comando **ssh**.

En el sistema de autenticación con clave pública, la entidad de autenticación tiene una clave pública y otra privada. La privada se guarda en el ordenador desde el que se inicia la sesión, y la pública se almacena en **.ssh/authorized\_keys** dentro de los ordenadores en los que se quiere iniciar la sesión.

La mayoría de los programas **SSH** almacenan la clave privada protegida por una contraseña.

La sintaxis para generar una clave pública generalmente es:  
 *[ssh-rsa/ssh-dss] <cadena larga inventada> <nombre del usuario>@<host>*

### Inicio de sesión SSH basado en claves

Hay dos opciones: usar claves **RSA** o **DSA**. La primera es más segura, por ello es la opción recomendada.

La clave pública la puede ver cualquiera y la privada sólo la puede ver el propietarios.

### Generación de claves RSA

Se debe hacer en el cliente.

Para crearlas, hay que escribir en la línea de comandos:

*mkdir ῀/.ssh*

*chmod 700 ῀/.ssh*

*ssh -keygen -t rsa*

El proceso pedirá una ubicación para guardar las claves y una contraseña para proteger la clave privada mientras esté almacenada en el disco duro.

Una vez acabado el proceso, la clave pública se encontrará como **.ssh/id\_rsa\_pub** en la carpeta de inicio.

Si nos roban la clave, podemos entrar en cualquier ordenador en el que tengamos cuenta, eliminar la clave del archivo **.ssh/authorized\_keys** y agregar una nueva clave.

### Nivel de cifrado de la clave

El valor predeterminado es de **2048 bits**, si se quiere aumentar a **4096 bits** hay que escribir:

*ssh -keygen -t rsa -b 4096*

### Transferir la clave del cliente al host

La clave que se transfiere es la pública si se puede iniciar sesión en un ordenador a través de **SSH** usando una contraseña, se pude hacer la transferencia usando:

*ssh-copy-id <nombre del usuario>@<host>*

Donde **host** es el nombre del ordenador al que se le está transfiriendo la clave. Si se produce algún error, se puede escribir:

*ssh-copy-id <nombre del usuario>@<host> -p <número de puerto>*

Otra opción es copiar el archivo de la clave pública en el servidor y concatenarlo manualmente al archivo de **claves autorizadas**, haciendo antes copia de seguridad del archivo:

*cp authorized\_keys authorized\_keys\_backup*

*cat id\_rsa.pub >> authorized\_keys*

Ahora Podemos asegurarnos que funcionón;

*ssh <nombre de usuario>@<host>*

Pedirá la contraseña y si el servidor permite inicio de sesión basado en clave, se debe iniciar la sesión.

El protocolo **SSH** soluciona dos problemas básicos de la comunicación segura con un **host** **remoto**: comprueba que el **host remoto** es quien dice ser (previniendo los ataques **Hombre en el medio** y encripta las comunicaciones entre el ordenador local y el **host remoto**.

Normalmente, las conexiones se hacen por el puerto **TCP 22**.

Para permitir que un sistema reciba conexiones remotas, se debe instalar el paquete **OpenSSH-server**.

Si no se tiene un servidor remoto al que conectarnos, para probarlo se usa el nombre **localhost** como nombre del **host remoto**.

La instrucción básica para conectar a un **host remoto** sería:

*ssh <nombre del host remoto>*

Por ejemplo:

*ssh remote-sys*

La primera vez que se hace la conexión, se muestra un aviso en el que se nos indica que la autenticidad del **host remoto** no puede ser establecida; esto se debe a que el programa cliente nunca ha visto al **host remoto** antes. Hay que introducir **yes** cuando lo pida.

## PUERTOS DE COMUNICACIÓN

Los servidores ponen sus servicios a disposición de internet usando puertos numerados. Los clientes se conectan a un servicio a una dirección **IP** y un puerto específico. Algunos puertos comunes son:

* **7**: Echo
* **13**: Daytime
* **17**: Qotd (Quote of the day)
* **21**: FTP
* **22**: SSH
* **23**: Telnet
* **25**: SMTP
* **37**: Time
* **43**: Nickname o apodo (Who is)
* **53**: Servidor de nombres (DNS o nameserver)
* **67** y **68**: DHCP
* **70**: Gopher
* **79**: Finger
* **80**: WWW
* **110**: Pop3
* **143**: IMAP
* **161**: SNMP (Protocolo simple de administración de red)
* **194**: IRC (Chat)
* **443**: HTTPS

Si el servidor acepta conexiones externas a algún puerto sin proteger el acceso por **firewall**, cualquierapuede conectarse al servicio de ese puerto.

Los protocolos de transporte más comunes que usan números de puertos son el **protocolo de control de transmisión** (**TCP**) y el **protocolo de datagramas de usuario** (**UDP**).

Un número de puerto siempre está asociado con una dirección **IP** de un **host** y el tipo de protocolo de transporte usado para la comunicación. Los números de puertos inferiores a 1024 identifican a los servicios más usados y se llaman **número de puertos conocidos**; los que tienen número mayores que 1024, están disponibles para uso general de las aplicaciones y se conocen como **puertos efímeros**.

En **TCP** el puerto 0 está reservado y no se puede usar, mientras que en **UDP** el valor 0 significa que no hay puerto.

## STDIN, STDOUT Y STDERR

Cada proceso en Linux suele disponer de tres archivos abiertos al comienzo de su ejecución: la entrada, la salida y la salida de errores.

Un programa siempre tomará los datos de entrada por el canal 0, enviará los resultados por el canal 1 y mostrará los errores por el 2. Para referirnos a cada uno de ellos se usa **STDIN**, **STDOUT** y **STDERR**.

El **[shell](https://es.wikipedia.org/wiki/Int%C3%A9rprete_de_comandos" \o "Intérprete de comandos)** del sistema es el encargado, generalmente, de abrir esta entrada y conectarla con el [terminal](https://es.wikipedia.org/wiki/Terminal_de_computadora) al que está asignado el programa, puesto que es el programa más usual para comunicarse con un operador.

La entrada estándar también se suele conectar a la salida estándar de un proceso, de manera que el programa lea directamente los resultados que otro produce. En [Linux](https://es.wikipedia.org/wiki/Unix) se utiliza el carácter **|** para expresar dicho tipo de conexión:

*programa1 | programa2 | programa3*

Lo que viene a significar que se ejecuta el **programa1** y su salida estándar (descriptor de archivo 1) se conecta con la entrada estándar del **programa2** (descriptor 0), que a su vez hace lo mismo con **programa3**.

La **salida estándar** (**STDOUT**) determina el destino del resultado de un programa o proceso, por defecto es la pantalla y se muestra el resultado en la consola de la terminal; sin embargo, puede redirigirse hacia un archivo por ejemplo.

Ejemplo:

*ls -l > miarchivo.txt*

## SUBREDES

Las redes se pueden dividir en subredes. Por ello, una dirección **IP** consta de dos partes:

1. El prefijo de red en los bits de orden superior.
2. **Campo de descanso**, **identificador de host** o **identificador de interfaz** (**IPV6**), que son losbits restantes. Se usa para la numeración del **host** dentro de la red.

La **máscara de subred** o la notación **CIDR** determina cómo se divide la dirección **IP** en partes de red y de **host**. La **máscara de subred** sólo se usa en **IPv4**. Sin embargo, tanto **IPv4** como **IPv6** usan el concepto y la notación **CIDR**. En esta notación, la dirección **IP** va seguida de una barra inclinada y el número decimal de **bits** usados por la parte de la red, también llamado **prefijo de enrutamiento**.

Por ejemplo, una dirección **IPv4** y su **máscara de subred** pueden ser: *192.168.0.1* y *255.255.255.0*. La notación **CIDR** para la misma **IP** y la **máscara de subred** es *192.168.0.1/24* porque los primeros 24 **bits** (los tres primeros **octetos**: 192.168.0) de la **IP** indican la red y la subred.

## TLS

Es un protocolo criptográfico que proporciona seguridad en las comunicaciones a través de una red. Se usa para los correos electrónicos, mensajería instantánea, **voz IP** y protección **HTTPS**.

Proporciona criptografía, incluida la privacidad (confidencialidad), integridad y autenticidad mediante el uso de certificados entre dos o más aplicaciones que se comunican. Se ejecuta en la capa de aplicación y tiene dos capas: el registro **TLS** y el protocolo de enlace **TLS**.