Fuente: Bing Chat AI

Laboratorio 3. Aplicaciones de la Criptografía (I)

DEPARTMENTO DE INFORMÁTICA. UNIVERSIDAD DE OVIEDO

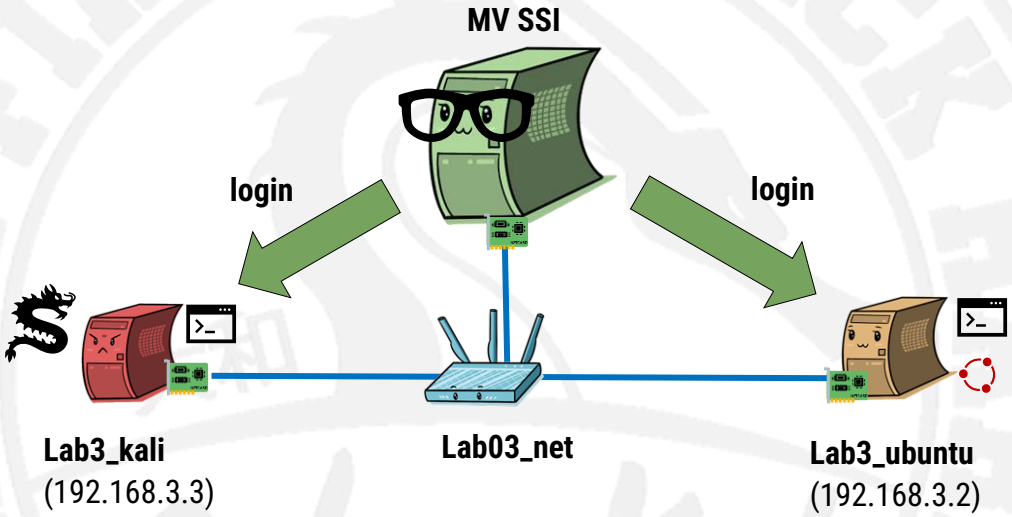
Seguridad de Sistemas Informáticos | 2023 – 2024 (V4.0 *"S-81 Isaac Peral"*)

Contenido

Infraestructura de este laboratorio

En primer lugar, asegúrate de tener la estructura de carpetas `ssi_labs` que te proporcionamos dentro de tu máquina virtual. Este laboratorio (`lab_03`) consiste en una infraestructura compuesta por dos contenedores comunicados a través de una LAN privada: un **SO *Kali*** y un **SO *Ubuntu***. Ambos tienen instalado el *software* para este laboratorio, aunque **la parte *de John the Ripper* solo se puede completar desde el contenedor *Kali*** debido a la herramienta `openssl2john`. No todas las actividades requieren el uso de todos ellos. Consulta el siguiente diagrama para entender el diseño de la infraestructura.

🔍 ***NOTA: La máquina virtual tiene disponible una copia offline de la herramienta CyberChef accesible vía web en la MV si no tienes conectividad a Internet en un momento dado***



Bloque 1: Técnicas de no cifrado. Ocultación de la información

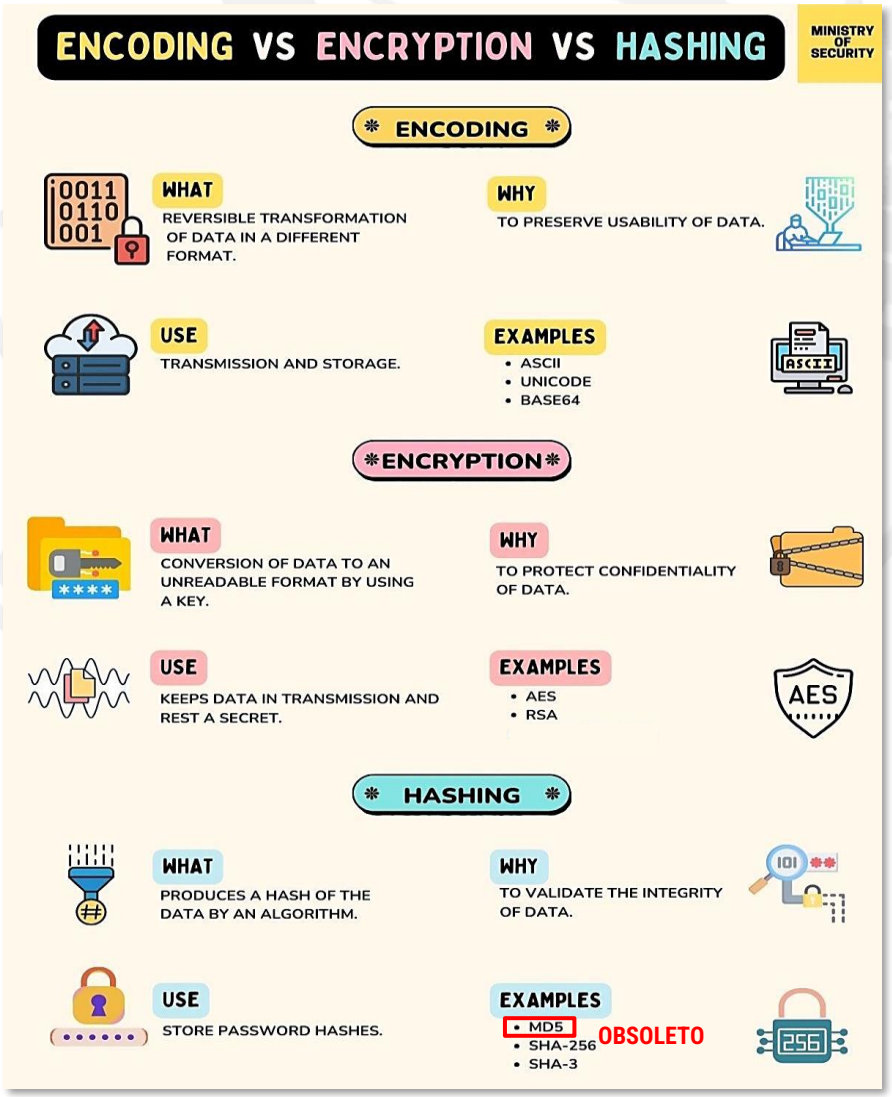
Ejercicio L3B1_ENCODE: Probar los mecanismos de codificación con *CyberChef*

Descripción de la actividad / Aplicación práctica

Consiste en que **entiendas para que sirve la codificación** probando a codificar y descodificar tu mismo información con la herramienta *CyberChef*: https://gchq.github.io/CyberChef/.

Resultados Esperados

Entiendes el manejo básico de *CyberChef* probando a codificar y descodificar información. El objetivo de esta actividad es que **distingas correctamente la diferencia entre codificar, cifrar y crear un hash** (cifrar y crear un hash lo veremos luego), modelando lo que aparece en esta imagen.



Además, puedes contestar a estas preguntas:

- *¿Entiendes que la codificación no es en absoluto una mecanismo de seguridad sino para ofuscar información al no usar ningún medio que impida descifrarla a cualquiera?*
- *¿Comprendes que el principal uso de la codificación es permitir transmitir información cuando el medio de transmisión no admite ciertos caracteres de la información original?*
- *¿Has entendido cómo hace el email para que puedas transferir datos binarios cuando el protocolo no los admite? Ahora que sabes esto, ¿Por qué crees que muchos proveedores de correo no admiten adjuntos mayores de 20Mb?*

Otra información necesaria para su realización

CyberChef (https://gchq.github.io/CyberChef/) es una página web de referencia para realizar todo tipo de transformaciones a datos, desde codificación hasta algoritmos completos de cifrado/descifrado, todo vía su web (*online* u *offline*) y con poco esfuerzo.

🔍 ***Cada transformación es una "receta" (Recipe), y se pueden seleccionar múltiples desde el panel izquierdo (arrastrar y soltar) para transformar ("Hornear" o Bake) la entrada que quieras. Familiarízate con el uso de Cyberchef, ya que es una de las herramientas preferidas usadas en CTF cuando encuentras datos codificados o cifrados y no tienes otras herramientas disponibles para procesar la información***

Esta página web tiene MUCHAS recetas, por lo que normalmente encuentras lo que buscas aquí, incluso si no está relacionado con el cifrado: por ejemplo, **también permite ver metadatos de archivos** (**Laboratorio 4**). Para practicar con algunos ejemplos, puedes hacer lo siguiente.

- **Codificar y descodificar cualquier texto** que quieras usando el formato ***Base64***
- Que pruebes **la codificación en acción en la vida real a través del mail** de esta manera:
  - Busca una imagen lo más pequeña que puedas (pocos Kb) y envíatela a ti mismo vía email.
  - Los protocolos de email **no admiten el envío de datos binarios**, por lo que para poder enviarte la imagen necesita **codificarla a formato texto** gracias a las especificaciones MIME (https://es.wikipedia.org/wiki/Multipurpose_Internet_Mail_Extensions)
  - MIME **codificará la imagen en el formato *Base64*** anterior y el cliente de correo gestiona automáticamente el proceso de codificación / descodificación y visualización para que tú no tengas que preocuparte por eso. Pero puedes investigar lo que hace así:
    - Abre el mensaje y busca la opción que permita **descargártelo completo** en formato `.eml`:
      - En *Gmail*, vete a botón de los tres puntos verticales y "*Descargar Mensaje*"
      - En *Outlook*, vete a botón de los tres puntos horizontales y "*Guardar Como*"
    - Abre el fichero `.eml` descargado con el *Bloc de notas* o editor de texto similar
      - Busca la cabecera `Content-Transfer-Encoding: base64`. Justo debajo (en *Outlook*) o dos cabeceras más abajo (en *Gmail*) verás **un gran bloque de texto** que es la imagen que has enviado codificada en *Base64*
      - Copia todo el bloque de texto hasta la siguiente línea en blanco y pégalo en *CyberChef* con la receta que permita deshacer la codificación *Base64*. *¿Salen datos binarios o en texto?*
      - Descarga la salida de *CyberChef* y ponle la misma extensión que tenía la imagen original. *¿Lo que has descargado es una copia de la imagen que enviaste?*

Ejercicio L3B1_STEGO: Uso de esteganografía con *steghide*

Descripción de la actividad / Aplicación práctica

Puedes **ocultar secretos en imágenes** sin alterar lo que la imagen muestra

Resultados Esperados

Esta actividad finaliza cuando **se logra ocultar un mensaje mediante esteganografía** en una imagen y extraerlo posteriormente. Puedes investigar qué ocurre cuando la imagen con información oculta se sube a una red social.

🔍 ***Se sabe que cuando uno sube una imagen a una red social, email, aplicación de mensajería, foro o similar mucho software manipula dicha imagen para recodificarla, añadirle información, comprimirla o hacer transformaciones que, aunque normalmente no tiene efectos visuales, cambiar completamente su contenido. Esto puede hacer que, si has ocultado un mensaje con esteganografía en ella, al recodificar la imagen ese mensaje se pierda. La pregunta es: ¿Puedes postear una imagen en Twitter (o cualquier otra red social) para comprobar si el contenido esteganografiado se pierde?***

En caso de que una red social no elimine el contenido esteganografiado: *¿Podrías enviar mensajes ocultos a alguien a plena vista?*

Otra información necesaria para su realización

Para hacer este ejercicio podemos elegir cualquier imagen de Internet e incrustarle un mensaje cualquiera con `steghide`, utilizando como contraseña una que puedas recordar. Las imágenes deben estar en un formato reconocido por la aplicación; la mayoría de `.jpg` deberían servir. Un ejemplo de imagen válida es este: https://images.idgesg.net/images/article/2018/06/intel-8086-100760661-large.jpg

Dispones de `steghide` instalado en el contenedor *Kali* de la **infraestructura del lab 3**. Usa las opciones apropiadas de `steghide` para ocultar el mensaje dentro de la imagen (consulta el *cheatsheet* siguiente) y envía la imagen con el mensaje oculto al contenedor *Ubuntu*, que deberá extraer el mensaje.

Para esto, recuerda que el directorio `/shared` de cada contenedor se puede usar para compartir archivos entre la máquina virtual y cada contenedor, lo que hace muy fácil compartir ficheros entre ellos. Usa las opciones adecuadas para extraer el mensaje cifrado.

steghide 0.5.1 Cheatsheet (ingenieriainformatica.uniovi.es)
Classic steganography tool

http://steghide.sourceforge.net/

GENERAL USAGE

`./steghide <first argument> <options>`

NOTES

The image you use must be of a compatible format (typically .jpg is)

OPTIONS

Possible First arguments (one is mandatory)	Embedding Options
embed, --embed: embed data	**-cf <filename>**: embed into the file <filename>
encinfo, --encinfo: display a list of supported encryption algorithms	**-cf, --coverfile**: select cover-file
extract, --extract: extract data	**-e <a>[<m>]|<m>[<a>]**: specify an encryption algorithm and/or mode
help, --help: display this usage information	**-e none**: do not encrypt data before embedding
info <filename>: display information about <filename>	**-e, --encryption**: select encryption parameters
info, --info: display information about a cover- or stego-file	**-ef <filename>**: embed the file <filename>
license, --license: display steghide's license	**-ef, --embedfile**: select file to be embedded
version, --version: display version information	**-f, --force**: overwrite existing files
Extracting Options	**-K, --nochecksum**: do not embed crc32 checksum of embedded data
-f, --force: overwrite existing files	**-N, --dontembedname**: do not embed the name of the original file
-p <passphrase>: use <passphrase> to extract data	**-p <passphrase>**: use <passphrase> to embed data
-p, --passphrase: specify passphrase	**-p, --passphrase**: specify passphrase
-q, --quiet: suppress information messages	**-q, --quiet**: suppress information messages
-sf <filename>: extract data from <filename>	**-sf <filename>**: write result to <filename> instead of cover-file
-sf, --stegofile: select stego file	**-sf, --stegofile**: select stego file
-v, --verbose: display detailed information	**-v, --verbose**: display detailed information
-xf <filename>: write the extracted data to <filename>	**-z <l>**: using level <l> (1 best speed...9 best compression)
-xf, --extractfile: select file name for extracted data	**-z, --compress**: compress data before embedding (default)
EXAMPLES	**-Z, --dontcompress**: do not compress data before embedding
To embed emb.txt in cvr.jpg: steghide embed -cf cvr.jpg -ef emb.txt *To extract embedded data from stg.jpg: steghide extract -sf stg.jpg*	**Options for the Info Command**
	-p, --passphrase: specify passphrase
	-p <passphrase>: use <passphrase> to get info about embedded data

Ejercicio L3B1_OBFUSCATE: Código ofuscado para que sea difícil de entender

Descripción de la actividad / Aplicación práctica

Necesitas **ofuscar código *JavaScript*** para hacer más difícil que alguien averigüe como has implementado cierta funcionalidad

Resultados Esperados

Esta actividad finaliza cuando hayas utilizado **todas las opciones de ofuscación mencionadas** y has comprobado que la salida de su ejecución sigue siendo la misma que la original aunque el código en sí tenga un aspecto muy diferente.

Otra información necesaria para su realización

La ofuscación del código **NO es un método de cifrado**, y transforma el código de manera que, **aunque aún pueda ejecutarse y su resultado sea el mismo que el original**, pero es más difícil de entender a menos que se desofusque. En otras palabras, ambos mecanismos tienen la misma forma de operar, aunque el resultado de la ofuscación es legible por cualquier persona

🔍 *¡Legible no significa comprensible!* 😊

Para probar rápidamente cómo funciona la ofuscación podemos usar *JavaScript* como ejemplo. Ten en cuenta que **la mayoría de los lenguajes tienen sus propias herramientas de ofuscación** adaptadas a su sintaxis, aunque los principios son los mismos. Para comprobar cómo funciona la ofuscación, sigue este procedimiento:

- Entra a esta página y mira el ejemplo de *JavaScript* que muestra. Puedes guardar la salida en un archivo para verificarla más tarde: https://js.do/
- Copia el *JavaScript* del código de ejemplo (**SOLO el bloque de código *JavaScript* (no las etiquetas `script`), ¡y no copies el HTML!**) y "empaquétalo" (sin opciones) con este empaquetador online: http://dean.edwards.name/packer/
- Copia el código empaquetado en el bloque `<script>` de la página del primer enlace y comprueba que el resultado de la ejecución es el mismo.
- Empaqueta el código de nuevo, **pero activando las opciones *Base62 encode* y *shrink variable***
- Copia de nuevo este código muy ofuscado y comprueba que funciona de nuevo en la página web del primer enlace.
- Vuelve sobre este *JavaScript* muy ofuscado y procésalo con los "embellecedores" de esta web http://jsnice.org/ o de esta https://beautifier.io/. Mira lo que hacen estas herramientas.
- Finalmente, procesa el código muy ofuscado en esta página web: https://obfuscator.io/, copia los resultados y verifica que el código aún funciona de nuevo.
- *¿Qué pasa si intentas "embellecer" este último código "terriblemente ofuscado"? ¿Puedes obtener de nuevo el código original?*
- *¿Crees que el código ofuscado es más pequeño que el original y por tanto ocupa menos? Si es así, ¿Para que crees que puede servir esto?*

Bloque 2: Criptografía Simétrica

Ejercicio L3B2_SYMCRYPT: Cifrar un archivo utilizando un algoritmo de clave simétrica fuerte

Descripción de la actividad / Aplicación práctica

Necesitas cifrar un fichero para que nadie sin su clave de cifrado pueda leer sus contenidos usando `openssl`, un *software* de cifrado que implementa el mismo protocolo que gestiona todas las conexiones cifradas a una web que ocurren al navegar.

Resultados Esperados

Esta actividad finaliza cuando puedas **cifrar un archivo mediante un algoritmo de clave simétrica fuerte** de forma que puedas enviar el archivo a cualquier parte y sólo aquellas personas que conozcan la contraseña de cifrado puedan leer su contenido. Puedes contestar a estas preguntas:

- *¿Eres consciente que AES-128 es un algoritmo de cifrado fuerte, que aún no se ha roto, y que, por tanto, si la clave fuese buena (algo que veremos más adelante 😉) nadie podría leer lo que estás enviando sin la clave (es decir, que este modo de cifrado es verdaderamente seguro)?*
- *¿Tienes claro que aunque tu mensaje se pueda romper con un ordenador cuántico, si cambias el algoritmo por un AES-256 ya no sería posible?*

Otra información necesaria para su realización

Empieza a trabajar en el contenedor *Kali* de la **infraestructura del Lab 3.** Crea un nuevo archivo de texto con un mensaje de texto sin cifrar que se enviará al receptor (por ejemplo, tu UO y nombre). Para cifrar el mensaje **usa** `openssl`, un programa que **ya está instalado en el contenedor *Kali***. Esta imagen que contiene las instrucciones que necesitas para ello (https://cheatography.com/albertx/cheat-sheets/openssl/). **Usar como clave** "`password`" (es una mala práctica, pero aquí lo hacemos a propósito para luego 😊). Selecciona el algoritmo de cifrado **AES-128-ECB**.

SYMMETRIC ENCRYPTION
List all supported symmetric encryption ciphers **openssl enc -list**
Encrypt a file using an ASCII encoded password provided and AES-128-ECB algorithm **openssl enc -aes-128-ecb -in cleartext.file -out ciphertext.file -pass pass:thisisthepassword**
Decrypt a file using AES-256-CBC and a keyfile **openssl enc -d -aes-256-cbc -in ciphertext.file -out cleartext.file -pass file:./key.file**
Encrypt a file using a specific encryption key (K) provided as hex digits **openssl enc -aes-128-ecb -in cleartext.file -out ciphertext.file -K 1881807b2d1b3d22f14e9ec52563d981 -nosalt**
Encrypt a file using ARIA 256 in CBC block cipher mode using a specified encryption key (K:256 bits) and initialization vector (iv:128 bits) **openssl enc -aria-256-cbc -in cleartext.file -out ciphertext.file -K f92d2e986b7a2a01683b4c40d0cbcf6feaa669ef2bb5ec3a25ce85d9548291c1 -iv 470bc29762496046882b61ecee68e07c -nosalt**
Encrypt a file using Camellia 192 algorithm in COUNTER block cipher mode with key and iv provided **openssl enc -camellia-192-ctr -in cleartext.file -out ciphertext.file -K 6c7a1b3487d28d3bf444186d7c529b48d67dd6206c7a1b34 -iv 470bc29762496046882b61ecee68e07c**

¿La salida cifrada es binaria o texto? Si es binaria, *¿Como podrías convertirla a texto de forma que puedas enviarla en el cuerpo de un email o pegarla en cualquier lugar que solo admita texto?*. Hazlo usando esta imagen como guía:

ENCODING / DECODING
Encoding a file using Base64 **openssl base64 -in file.data**
Encoding some text using Base64 **echo -n "some text" | openssl base64**
Base64 decode a file with output to another file **openssl base64 -d -in encoded.data -out decoded.data**

🛠 Ejercicio L3B2_SYMDECRYPT: Descifrar un archivo que utiliza cifrado simétrico

Descripción de la actividad / Aplicación práctica

Necesitas **obtener los contenidos de un fichero cifrado previamente** si sabes la clave correspondiente

Resultados Esperados

Esta actividad finaliza cuando se consigue **descifrar un archivo previamente cifrado**, tanto en forma binaria como convertido a texto en la actividad anterior. Puedes contestar a estas preguntas:

- *¿Puedes descifrar el fichero cifrado convertido a Base64 directamente o necesitas convertirlo a formato binario primero?*

- *¿Crees que puedes usar el cifrado simétrico para enviar información confidencial por medios que no lo son, como el email?*
- *¿Entiendes por qué es importante separar la forma de enviar la clave de la de enviar el contenido cifrado?*

Otra información necesaria para su realización

En el ejercicio anterior, a partir del *cheatsheet* proporcionado, pudiste cifrar un archivo con la clave indicada. Para descifrarlo, `openssl` usa la misma orden que diste pero añadiendo `-d` tras el parámetro `enc`.

Ejercicio L3B2_CIPHMODES: Probar los modos de cifrado ECB y CBC del algoritmo simétrico robusto de cifrado AES

Descripción de la actividad / Aplicación práctica

Puedes practicar con los **distintos modos de cifrado del algoritmo AES** y entender cómo esto impacta a la salida en función del tipo de datos que se use como entrada.

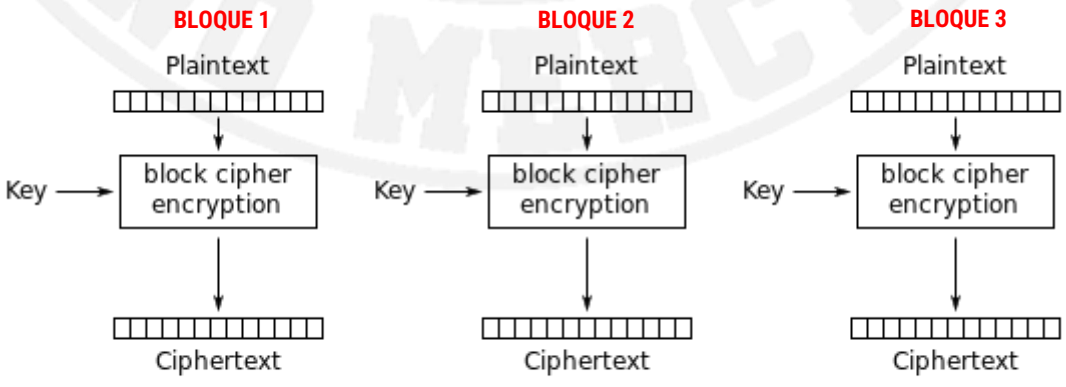
Resultados Esperados

Entiendes que **determinados modos de cifrado son inadecuados para algunos tipos de información**, como imágenes, cuando los bloques se cifran de manera individual y, por ello, los bloques cifrados con el mismo valor (color) siempre tienen la misma salida. Eres capaz de contestar las preguntas que se hacen durante el desarrollo del ejercicio.

Otra información necesaria para su realización

Practicando con el modo ECB

El modo ECB del algoritmo de cifrado AES es un modo de cifrado peligroso con algunos tipos de archivo, como las imágenes. La razón es que **cifra bloques de 128 bits uno a uno independientemente**, sin que haya algo que haga depender cómo cifra un bloque de lo que ya haya cifrado anteriormente, como ocurre en otros modos. En otras palabras, los bloques de la entrada se cifran así:



Electronic Codebook (ECB) mode encryption

> 🔍 ***Lo que, por desgracia, quiere decir que los bloques que tengan el mismo valor devolverán el mismo resultado al cifrarse***

Si la entrada es un bloque que representa un color, la salida será un bloque que representará un color distinto si el que ve el archivo cifrado averigua o intuye que se ha cifrado una imagen *¿Quieres saber qué pasaría entonces con imágenes que tengan grandes cantidades del mismo color?* Vamos a probarlo siguiendo estas instrucciones:

- Descárgate una **imagen PNG** que tenga fondo blanco y una serie de colores diferentes distintos para potenciar el efecto del que hablamos. En esta dirección tienes unas cuantas: https://commons.wikimedia.org/wiki/Category:PNG_files. Esta puede ser una: https://commons.wikimedia.org/wiki/File:Eslogan_Oficial.png
- **Convierte la imagen descargada** a un fichero de tipo `.ppm` (*Portable Pixmap Format*) para poder separar fácilmente cabecera y cuerpo de la imagen, y que el cuerpo sea un mapa de píxeles. Esto nos permitirá cifrar solo el cuerpo de la misma, y poder ver luego una imagen con solo el cuerpo cifrado. Para convertir la imagen podemos instalar la siguiente librería: `sudo apt install graphicsmagick-imagemagick-compat`. Una vez instalada, esta conversión puede hacerse con `convert <nombre de la imagen>.png <nombre de la imagen>.ppm`
- **Parte ahora la imagen convertida en dos**, cabecera y cuerpo:

```
head -n 3 <nombre de la imagen>.ppm > cabecera.txt
tail -n +4 <nombre de la imagen>.ppm > cuerpo.bin
```

- **Cifra nada más que el fichero** `cuerpo.bin` de la misma forma que en el ejercicio anterior
- Cuando tengas el cuerpo cifrado, **añádele la cabecera que extrajiste antes** para que el fichero vuelva a ser interpretado como una imagen `.ppm`. Puede hacerse así:

```
cat cabecera.txt <fichero con el cuerpo cifrado> > <nombre de la unión>.ppm
```

- **NOTA**: Una buena herramienta para abrir/editar imágenes es GIMP. Si su host (*Windows* en los laboratorios de la escuela) no abre la imagen una vez que la pongas en la carpeta compartida de la máquina virtual, puede instalarla en nuestra MV con `sudo apt install gimp`

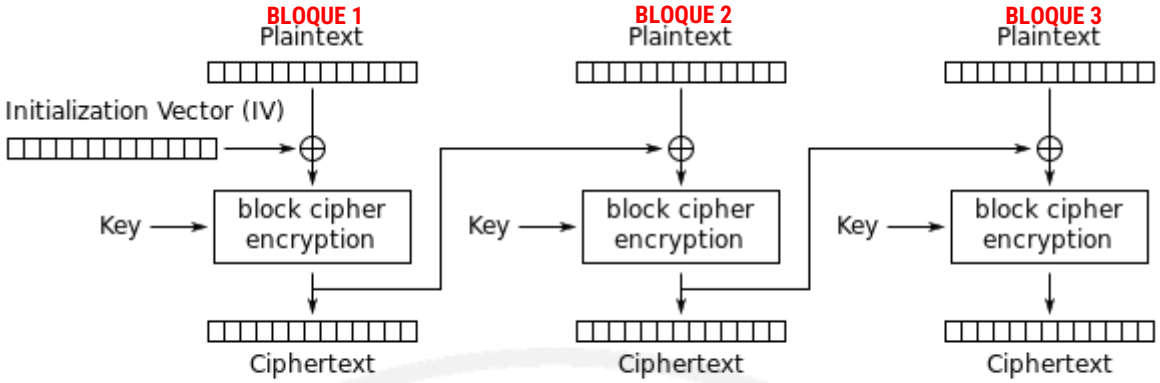
¿Qué pasa ahora cuando abres esta imagen? ¿Están los datos cifrados? Aun así, *¿Puedes ver los patrones de colores? ¿Crees que se puede distinguir la imagen? ¿Por qué crees que esto es peligroso y puede terminar en un filtrado de información?*

Practicando con el modo CBC

En la teoría hemos visto otros modos de cifrado de AES, como el modo CBC. En ese modo, un bloque cifrado **se usa como entrada** a la hora de cifrar el siguiente. Para el primero de todos, se usa un ***Initialization Vector*** (IV),

> 🔍 ***Un IV es un nº aleatorio único que suele tener el mismo tamaño que el bloque a cifrar***

El esquema sería algo así:



Cipher Block Chaining (CBC) mode encryption

Este ***Initialization Vector*** (IV) se puede generar con el parámetro `rand` de *OpenSSL*. Como estamos usando una AES de 128 bits y cada carácter son 8 bits, necesitaremos un IV de 16 caracteres (16 x 8 = 128). Ej.: `openssl rand -hex 16`

Cifra ahora el cuerpo de la imagen como antes, pero con el algoritmo **AES-128-CBC** y añadiendo el vector de inicialización que has generado como parámetro de la opción `-iv` que ahora debes añadir a la llamada al comando de cifrado. Hecho esto, vuelve a unir la cabecera y abre el fichero resultante. *¿Se distinguen ahora los patrones de colores o no? ¿Qué modo es mejor para cifrar imágenes entonces?*

Ejercicio L3B2_DISKCIPH: Cifrado de discos

Descripción de la actividad / Aplicación práctica

Necesitas **crear un directorio cuyos contenidos sea cifran automáticamente cuando se copian a él**, para que nadie que no sea tu usuario pueda acceder a ellos.

Resultados Esperados

Esta actividad finaliza **cuando puedas cifrar una carpeta** y comprobar que el contenido está realmente cifrado cuando la desmontas. Puedes responder a estas preguntas:

- *¿Entiendes por qué a este tipo de cifrado se le suele denominar "cifrado de datos en reposo"?*
- *¿Crees que sería buena idea hacer lo que hemos visto en este ejercicio en un lápiz USB que tenga datos confidenciales?*

Otra información necesaria para su realización

🔍 ***El cifrado de sistemas de archivos, directorios y discos es una de las aplicaciones más comunes de los algoritmos de clave simétrica. Cuando Ubuntu se instala ofrece cifrar nuestro directorio de inicio como opción (esto también cifra la partición `/swap`). Incluso si elegimos no hacerlo en ese momento, es posible crear discos cifrados posteriormente: https://www.howtogeek.com/116032/how-to-encrypt-your-home-folder-after-installing-ubuntu/***

En este laboratorio no vamos a hacer un cifrado de disco completo, sino una versión más rápida de la misma aplicación de la criptografía: cifrar solo una carpeta (y su contenido). Para ello, sigue este

procedimiento **en la máquina virtual** (**¡esto no funciona dentro de los contenedores de la infraestructura!** ☹️):

- Instala el *software* de cifrado de disco: `sudo apt install ecryptfs-utils`
- Crea una carpeta para cifrar en tu directorio *home*. Si usas una existente, asegúrate de que esté vacía (si no, haz una copia de seguridad de su contenido... por si acaso 😊)
- Cifra la carpeta montándola bajo el sistema de ficheros `ecryptfs` `mount -t ecryptfs <tu ruta de carpeta> <tu ruta de carpeta>` (sí, el punto de montaje y la carpeta son los mismos. Por ejemplo, `mount -t ecryptfs /home/ssiuser/secure /home/ssiuser/secure`). Necesitarás permisos de `root`
- Elije un algoritmo de cifrado simétrico apropiado de acuerdo con la teoría.
- Elije `32` bytes para la clave.
- Elije `No` en la opción *plaintext passthrough*.
- Elije `No` a cifrar nombres de archivo (aunque este es un detalle menor)
- Responde `Yes` a las dos preguntas siguientes (aparecen si este es el primer uso de cifrado de disco en el sistema operativo, que debería)
- Comprueba con el comando `mount` que el punto de montaje está montado como cifrado.
- Copia o crea un archivo de texto en la carpeta cifrada. *¿Puedes leerlo?*
- Desmonta el punto de montaje cifrado con `sudo umount /home/secure`. *¿Puedes leer su contenido ahora?* (por favor, ¡sal de la carpeta antes de desmontarla! 😊)
- Móntalo de nuevo con los mismos parámetros. *¿Puedes volver a leer el contenido?*

Bloque 3. Introducción al cracking de contraseñas sin conexión

Ejercicio L3B3_SSLCRACK: *John the Ripper* para romper OpenSSL

Descripción de la actividad / Aplicación práctica

Necesitas **crackear la contraseña de un fichero cifrado** con `openssl` que usa una password común

Resultados Esperados

Esta actividad finaliza **cuando se descifra la hash de la contraseña del mensaje cifrado** en un ejercicio anterior con `openssl` (por eso insistimos en usar "`password`" como contraseña 😊). Puedes contestar a las siguientes preguntas:

- *¿Entiendes que crackear una hash es solo un proceso repetitivo hasta dar con la entrada que genera la hash correcta?*
- *¿Has entendido que un crackeador de hashes actúa sobre hashes y no otro tipo de contenido cifrado?*
- *¿Entiendes que, por suerte, hay muchas herramientas capaces de extraerte o calcularte la hash de la contraseña a partir de ficheros cifrados que se hayan generado con una gran cantidad de algoritmos?*
- *¿Entiendes la diferencia entre el proceso de hashear un candidato a una contraseña y generar muchos candidatos a esa contraseña, junto con los distintos métodos de generación de posibles contraseñas que existen?*

Otra información necesaria para su realización

🔍 ***Probablemente hayas visto en películas el "arte" de crackear contraseñas de formas variopintas: son todas mentira 😊. La principal razón es que un negocio serio no guarda las contraseñas tal cual las introduces ("en texto plano" sino que, como vimos en la teoría, se guardan en forma de hash (KDF).***

Como vimos en la teoría, **una hash no es reversible** (y aún menos "a cachos"), por lo que todo lo que hayas podido ver en la tele acerca de ir revelando una contraseña carácter a carácter **también es mentira**.

🔍 ***Además, ¿Tú te imaginas intentar entrar en una cuenta de Gmail de la que no te acuerdas de la clave y que el servicio te diga "mira, tienes las dos letras del final mal nada más"?***

¡Es hora de romper mitos y estereotipos y saber cómo se hace en realidad!.

Entiende bien qué es "crackear una contraseña" y que es lo que crackeas

John the Ripper es una de las herramientas para crackear hashes de contraseñas más populares (junto con `hashcat`, `hydra`, etc.) pero es eso: **una herramienta que crackea hashes**. *¿Qué significa?* Que es capaz de generar **un enorme nº de combinaciones de caracteres por segundo**, hashearlos según el algoritmo y otros

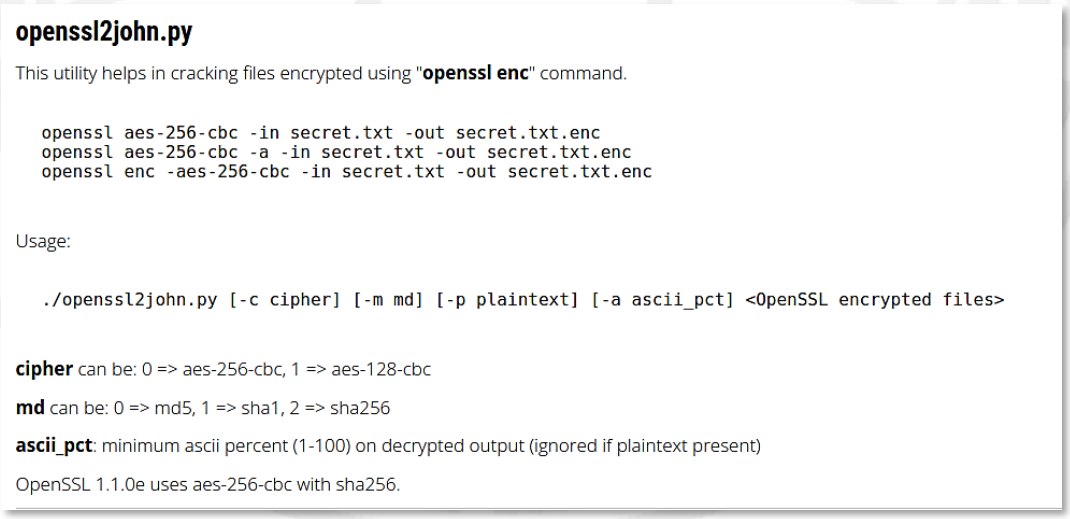
parámetros que especifiquemos (los usados en el fichero de hashes sobre el que vamos a trabajar) y comparar el resultado del hasheo las hashes que tiene el fichero. Si coincide, como `john` sabe la combinación de caracteres original que usó, habrá averiguado la contraseña correspondiente a la hash. Eso es lo que se llama "crackear una contraseña", ni más ni menos 😊.

🔍 ***Piensa que tiene todo el sentido del mundo, puesto que si una hash no es reversible lo que se hace es repetir el proceso de hasheo con distintas entradas hasta que des con la "buena". Si te estás planteando que esto se parece mucho a cuando un niño pequeño te fríe a preguntas hasta que le dices que si a algo, no andas muy desencaminado*** 😊**.**

La consecuencia de entender cómo se hace esto en la realidad lleva a otra conclusión: `john` **no puede trabajar con ficheros cifrados directamente**. Es decir, no puede abrir un fichero cifrado y probar directamente *passwords* contra él (¡de nuevo como pasa en las películas!). Por suerte, esto se puede resolver fácilmente. Hay **herramientas para extraer la hash de la password de un fichero cifrado** si sabes el programa y el algoritmo que se usó para ello.

🔍 ***En nuestro caso lo sabemos directamente, pero en la "vida real" puedes probar con varios que te parezcan más próximos, buscar o sonsacar información a alguien vía ingeniería social o usar un detector de algoritmo de cifrado (https://www.dcode.fr/cipher-identifier), pero esto es ya material para una asignatura exclusiva de criptografía.***

En la instalación de `john` que tienes en el contenedor Kali, puedes ver cuantas herramientas de este estilo hay tecleando: `locate *2john*`. La lista completa también está aquí: https://miloserdov.org/?p=5008&PageSpeed=noscript. Aquí puedes ver cómo la que necesitamos en nuestro caso es `openssl2john`:



openssl2john.py

This utility helps in cracking files encrypted using "**openssl enc**" command.

```
openssl aes-256-cbc -in secret.txt -out secret.txt.enc
openssl aes-256-cbc -a -in secret.txt -out secret.txt.enc
openssl enc -aes-256-cbc -in secret.txt -out secret.txt.enc
```

Usage:

```
./openssl2john.py [-c cipher] [-m md] [-p plaintext] [-a ascii_pct] <OpenSSL encrypted files>
```

cipher can be: 0 => aes-256-cbc, 1 => aes-128-cbc

md can be: 0 => md5, 1 => sha1, 2 => sha256

ascii_pct: minimum ascii percent (1-100) on decrypted output (ignored if plaintext present)

OpenSSL 1.1.0e uses aes-256-cbc with sha256.

Ya sabes cómo hashear tus posibles contraseñas pero...¿Como generar un nº enorme de ellas para probarlas?

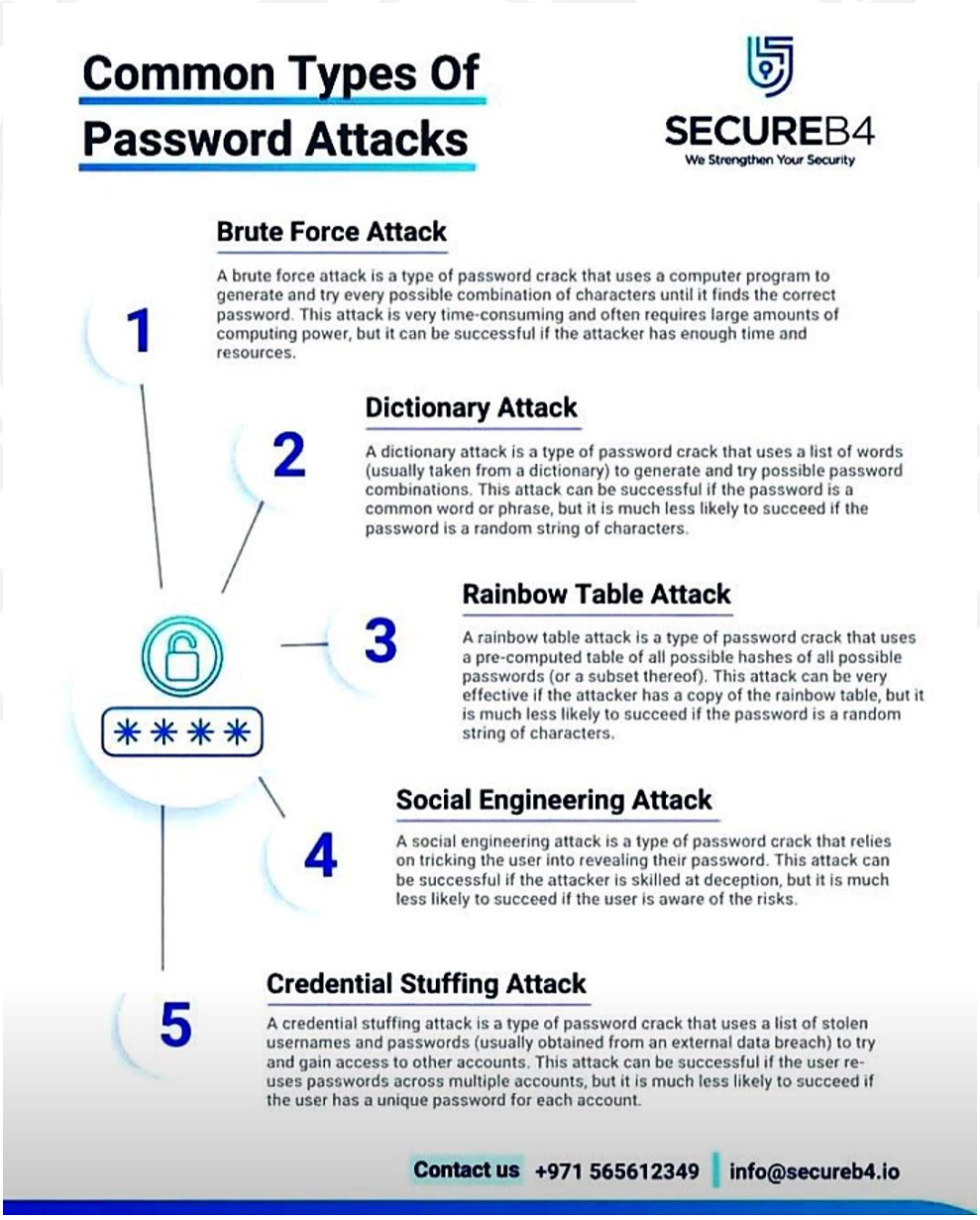
Ahora ya sabes cómo convertir tus contraseñas candidatas **a un formato que se pueda comparar con las hashes que tienes** y que quieres descifrar. Queda por resolver un último problema *¿Cómo genero un nº enorme de posibles contraseñas para que* `john` *vaya probándolas?* Hay tres técnicas básicas:

- **Fuerza bruta**: que intenta adivinar la contraseña **iterando secuencialmente** a través de cada posible combinación de letras, números y caracteres especiales de un vocabulario de caracteres que

establecemos de antemano (¡o cualquier combinación posible si no ponemos uno!). Este es un proceso tremendamente lento, pero efectivo... si tienes el tiempo suficiente para esperar a que termine 😊. Si la contraseña es muy fuerte, puede acabar más o menos en la fecha en la que el universo se espera que colapse sobre si mismo :P

- **Diccionario** (también conocido como ***listas de palabras***)*:* Este ataque utiliza un archivo que contiene listas de contraseñas comunes (generalmente extraídas de distintos robos de ficheros de contraseñas reales) para adivinar una contraseña determinada. Puede ser útil en CTFs, pero podría ser difícil tener éxito en el mundo real si los usuarios siguen una política de contraseñas fuerte.
- ***Rainbow tables***: son una serie de **hashes precalculados**. La idea es que estas tablas incluyan todos los hashes para un algoritmo dado. Entonces, en lugar de descifrar el hash / contraseña / etc., se realiza una búsqueda del hash en la tabla. Esto requiere una potencia de procesamiento considerable para lograrlo, pero podría ser más efectivo. No las veremos en esta asignatura, pero si te interesa puedes consultar más datos de ellas aquí: https://en.wikipedia.org/wiki/Rainbow_table .

En realidad puedes añadir dos técnicas más, pero se aplican más sobre personas que sobre ficheros, como ves en la siguiente imagen 😊. Estas dos están más relacionadas con el **Trabajo en Grupo** de la asignatura.



En este ejercicio vamos a utilizar **un ataque basado en diccionario** para romper la contraseña de nuestro mensaje cifrado anteriormente con `openssl` siguiendo estos pasos:

- Usar `john` como herramienta **de descifrado de contraseñas**, proporcionada en la **infraestructura del Lab 3** (contenedor *Kali*), pero instalable con `sudo apt install john`
- Necesitamos un **diccionario,** y usaremos uno de contraseñas comprometidas muy popular llamado `rockyou.txt` que está disponible en el directorio `/wordlist` del contenedor *Kali*. Si necesitas hacer algo como esto en el futuro, puedes usar diccionarios más grandes o especializados como los que están aquí: https://ns2.elhacker.net/wordlists/
- **Crea un fichero a descifrar** de la misma forma que lo hiciste en el ejercicio **L3B2_SYMCRIPT** (con la contraseña "`password`"), pero **usando el formato AES-128-CBC**. Como dijimos, debes procesarlo previamente con la utilidad `openssl2john` instalada en el contenedor *Kali* de la **infraestructura del Lab 3** (el paquete `john` predeterminado de *Ubuntu* no lo incluye). Almacena la salida en un archivo y recuerda bien el formato de la hash que te extrae que le has puesto en el parámetro `-m`.

🔍 ***NOTA: Es posible realizar este ejercicio utilizando los algoritmos de cifrado AES-128-CBC y AES-256-CBC. En ambos casos, el formato hash para la salida de `openssl2john` debe ser SHA256 (`-m 2`). Sin embargo, ten en cuenta que el parámetro `-c` de `openssl2john` cambia dependiendo del algoritmo de cifrado que hayas utilizado. Por favor, NO USES AES-128-ECB o AES-256-ECB, ya que `openssl2john` no soporta estos modos de operación de AES.***

Usa este *cheatsheet* de `john` con la lista de palabras proporcionada contra el archivo anterior para descifrar la contraseña del mensaje cifrado con `openssl`. **Asegúrate de pasarle bien el formato de la hash** del fichero, que es el que has puesto en `openssl2john`: https://pentestmonkey.net/cheat-sheet/john-the-ripper-hash-formats

John the Ripper 1.9.0-jumbo Cheatsheet (ingenieriainformatica.uniovi.es)
Offline password cracking tool

https://github.com/openwall/john

GENERAL USAGE

john [OPTIONS] [PASSWORD-FILES]

NOTES

John cracking modes:	https://www.openwall.com/john/doc/MODES.shtml
John FAQ:	https://www.openwall.com/john/doc/FAQ.shtml
John wordlist rules:	https://www.openwall.com/john/doc/RULES.shtml

OPTIONS

Cracking modes

--external=MODE: external MODE or word filter
--incremental[=MODE]: "incremental" mode [using section MODE]
--markov[=OPTIONS]: "Markov" mode (see doc/MARKOV)
--single[=SECTION[,..]]: "single crack" mode, using default or named rules
--single=:rule[,..]: same, using "immediate" rule(s)
--wordlist[=FILE]: --stdin wordlist mode, read words from FILE or stdin

EXAMPLES

Please consult a full example set on:
https://www.openwall.com/john/doc/EXAMPLES.shtml

```
ssiuser@ubuntussi:~$ john -wordlist:myDict passwd.db
Created directory: /home/ssiuser/.john
Loaded 1 password hash (crypt, generic crypt(3) [?/64])
Press 'q' or Ctrl-C to abort, almost any other key for status
test123...       (ssiuser)
1g 0:00:00:00 100% 2.173g/s 417.3p/s 417.3c/s 417.3C/s test096.....test191...
Use the "--show" option to display all of the cracked passwords reliably
Session completed
ssiuser@ubuntussi:~$
```

Other options

--costs=[-]C[:M][,...]: load salts with[out] cost value Cn [to Mn]. For tunable cost parameters, see doc/OPTIONS
--dupe-suppression: suppress all dupes (duplicates) in wordlist (and force preload)
--encoding=NAME: input encoding (eg. UTF-8, ISO-8859-1). See also doc/ENCODINGS and --list=hidden-options.
--fork=N: fork N processes
--format=NAME: force hash of type NAME. the supported formats can be seen with --list=formats and --list=subformats
--groups=[-]GID[,..]: load users [not] of this (these) group(s) only
--list=WHAT: list capabilities, see --list=help or doc/OPTIONS
--loopback[=FILE]: like --wordlist, but extract words from a .pot file
--make-charset=FILE: make a charset file. It will be overwritten
--mask[=MASK]: mask mode using MASK (or default from john.conf)
--node=MIN[-MAX]/TOTAL: this node's number range out of TOTAL count
--pipe: like --stdin, but bulk reads, and allows rules
--pot=NAME: pot FILE to use
--prince[=FILE]: PRINCE mode, read words from FILE
--restore[=NAME]: restore an interrupted session [called NAME]
--rules[=SECTION[,..]]: enable word mangling rules (for wordlist or PRINCE modes), using default or named rules
--rules=:rule[;..]]: same, using "immediate" rule(s)
--rules-stack=:rule[;..]: same, using "immediate" rule(s)
--rules-stack=SECTION[,..]: stacked rules, applied after regular rules or to modes that otherwise don't support rules
--salts=[-]COUNT[:MAX]: load salts with[out] COUNT [to MAX] hashes
--save-memory=LEVEL: enable memory saving, at LEVEL 1..3
--session=NAME: give a new session the NAME
--shells=[-]SHELL[,..]: load users with[out] this (these) shell(s) only
--show[=left]: show cracked passwords [if =left, then uncracked]
--status[=NAME]: print status of a session [called NAME]
--stdout[=LENGTH]: just output candidate passwords [cut at LENGTH]
--subsets[=CHARSET]: "subsets" mode (see doc/SUBSETS)
--test[=TIME]: run tests and benchmarks for TIME seconds each
--users=[-]LOGIN|UID[,..]: [do not] load this (these) user(s) only

🛠 Ejercicio L3B3_PASS: *John the Ripper* + contraseñas de usuario + generador de listas de palabras *crunch*

Descripción de la actividad / Aplicación práctica

Necesitas **crackear la hash de la clave de un usuario** con un diccionario de palabras personalizado

Resultados Esperados

Esta actividad se considera finalizada cuando se descifre la contraseña del usuario `ssiuser` en los archivos `passwd` y `shadow` de ejemplo proporcionados (directorio `/wordlist` de los contenedores de **infraestructura del Lab 3**)

Otra información necesaria para su realización

Vamos a usar `john` otra vez, pero para un fin diferente. `rockyou.txt` es un buen diccionario de contraseñas filtradas, pero a veces usarlo es contraproducente porque de alguna manera sabemos de antemano que las contraseñas de usuario siguen un patrón, una regla, o tienen cierta forma. En ese caso, generar un diccionario de palabras personalizado, adaptado a esas suposiciones, **es el enfoque más inteligente**, y este es precisamente el objetivo de este ejercicio. Sin embargo, en lugar de descifrar un archivo, te vamos a

enseñar cómo los archivos filtrados de usuarios y contraseñas de *Linux* se rompen habitualmente. Aparte de `john`, para hacer esto necesitas:

- **Una herramienta para combinar** el `passwd` **filtrado** que tiene los nombres de usuario y el archivo con **las hashes de sus claves** (`shadow`) como `unshadow` (ya instalado en la **infraestructura del Lab 3).** Esta herramienta lee los archivos `passwd` y `shadow` que se le pasan como primer y segundo parámetro y los une con un formato combinado usable por `john`, que se puede guardar en un tercer archivo.
- **Un generador de listas de palabras**, como `crunch` (ya instalado en la **infraestructura del Lab 3,** pero instalarlo es muy fácil con `sudo apt install crunch`)
- **Antes de continuar**, primero lee el *cheatsheet* y sigue las siguientes las instrucciones.

`crunch` te permite **generar una lista de palabras** siguiendo los patrones de palabras que especifiques si conoces o sospechas parte de la contraseña de un usuario, la forma de la clave (números + letras, solo minúsculas...), o cualquier dato que pueda ahorrarte probar contraseñas imposibles. Entrar en la sintaxis de generación de patrones de `crunch` está más allá del alcance de este laboratorio, pero para facilitar el proceso puedes usar el *cheatsheet* y estos consejos:

- El símbolo `%` significa "cualquier número".
- El símbolo `@` significa "cualquier letra minúscula".
- Si tecleas una cadena de caracteres, se tratará como una constante (las claves generadas tendrán esa cadena de caracteres en la posición que indiques sí o sí).

La contraseña a adivinar tiene una forma muy fácil de generar. Comienza con "`test`" y termina con "`...` ". También sabemos que la parte media es tres **números o tres letras minúsculas** (siempre 3, nunca mezclando números y letras). Las longitudes de los patrones para `crunch` son siempre número de caracteres + 1 (los string C siempre agregan '\0' al final, por lo que la longitud debe ser también un carácter más que la longitud real que necesitamos usar).

crunch 3.6 Cheatsheet (ingenieriainformatica.uniovi.es)

Tool that generates wordlists from a character set

https://tools.kali.org/password-attacks/crunch

```
ssluser@ubuntussi:~$ crunch 10 10 -t test%%%... -o myDict
Crunch will now generate the following amount of data: 11000 bytes
0 MB
0 GB
0 TB
0 PB
Crunch will now generate the following number of lines: 1000

crunch: 100% completed generating output
```

GENERAL USAGE

```
crunch <min-len> <max-len> [<charset string>] [options]
```

NOTES

This is a reference tool to generate wordlists to bruteforce passwords and other similar requirements. Crunch can create a wordlist based on criteria you specify. The output from crunch can be sent to the screen, file, or to another program

OPTIONS

Required parameters

charset string: You may specify character sets for crunch to use on the command line or if you leave it blank crunch will use the default character sets. The order MUST BE lower case characters, upper case characters, numbers, and then symbols. If you don't follow this order you will not get the results you want. You MUST specify either values for the character type or a plus sign. NOTE: If you want to include the space character in your character set you must escape it using the \ character or enclose your character set in quotes i.e. "abc ". See the examples 3, 11, 12, and 13 for examples.

max-len: The maximum length string you want crunch to end at. This option is required even for parameters that won't use the value.

min-len: The minimum length string you want crunch to start at. This option is required even for parameters that won't use the value.

Options

-b number[type]: Specifies the size of the output file, only works if -o START is used, i.e.: 60MB The output files will be in the format of starting letter-ending letter for example: ./crunch 4 5 -b 20mib -o START will generate 4 files: aaaa-gvfed.txt, gvfee-ombqy.txt, ombqz-wcydt.txt, wcydu-zzzzz.txt valid values for type are kb, mb, gb, kib, mib, and gib. The first three types are based on 1000 while the last three types are based on 1024. NOTE There is no space between the number and type. For example 500mb is correct 500 mb is NOT correct.

-c number: Specifies the number of lines to write to output file, only works if -o START is used, i.e.: 60. The output files will be in the format of starting letter-ending letter for example: ./crunch 1 1 -f /pentest/password/crunch/charset.lst mixalpha-numeric-all-space -o START -c 60 will result in 2 files: a-7.txt and 8-\ .txt The reason for the slash in the second filename is the ending character is space and ls has to escape it to print it. Yes you will need to put in the \ when specifying the filename because the last character is a space.

-d numbersymbol: Limits the number of duplicate characters. -d 2@ limits the lower case alphabet to output like aab and aac. aaa would not be generated as that is 3 consecutive letters of a. The format is number then symbol where number is the maximum number of consecutive characters and symbol is the symbol of the the character set you want to limit i.e. @,%^ See examples 17-19.

-e string: Specifies when crunch should stop early

-f /path/to/charset.lst charset-name: Specifies a character set from the charset.lst

-i: Inverts the output so instead of aaa,aab,aac,aad, etc you get aaa,baa,caa,daa,aba,bba, etc

-l: When you use the -t option this option tells crunch which symbols should be treated as literals. This will allow you to use the placeholders as letters in the pattern. The -l option should be the same length as the -t option. See example 15.

-m: Merged with -p. Please use -p instead.

-o wordlist.txt: Specifies the file to write the output to, eg: wordlist.txt

-p charset OR -p word1 word2 …: Tells crunch to generate words that don't have repeating characters. By default crunch will generate a wordlist size of #of_chars_in_charset ^ max_length. This option will instead generate #of_chars_in_charset!. The ! stands for factorial. For example say the charset is abc and max length is 4.. Crunch will by default generate 3^4 = 81 words. This option will instead generate 3! = 3x2x1 = 6 words (abc, acb, bac, bca, cab, cba). THIS MUST BE THE LAST OPTION! This option CANNOT be used with -s and it ignores min and max length however you must still specify two numbers.

-q filename.txt: Tells crunch to read filename.txt and permute what is read. This is like the -p option except it gets the input from filename.txt.

-r: Tells crunch to resume generate words from where it left off. -r only works if you use -o. You must use the same command as the original command used to generate the words. The only exception to this is the -s option. If your original command used the -s option you MUST remove it before you resume the session. Just add -r to the end of the original command.

-s startblock: Specifies a starting string, eg: 03god22fs

-t @,%^: Specifies a pattern, eg: @@god@@@@ where the only the @'s, ,'s, %'s, and ^'s will change; @ will insert lower case characters; , will insert upper case characters; % will insert numbers; ^ will insert symbols

-u: The -u option disables the printpercentage thread. This should be the last option.

-z gzip, bzip2, lzma, and 7z: Compresses the output from the -o option. Valid parameters are gzip, bzip2, lzma, and 7z. gzip is the fastest but the compression is minimal. bzip2 is a little slower than gzip but has better compression. 7z is slowest but has the best compression.

EXAMPLES

Example 1: crunch 1 8 - crunch will display a wordlist that starts at a and ends at zzzzzzzz

Example 2: crunch 1 6 abcdefg - crunch will display a wordlist using the character set abcdefg that starts at a and ends at gggggg

Example 3: crunch 1 6 abcdefg - there is a space at the end of the character string. In order for crunch to use the space you will need to escape it using the \ character. In this example you could also put quotes around the letters and not need the \, i.e. "abcdefg ". Crunch will display a wordlist using the character set abcdefg that starts at a and ends at (6 spaces)

Example 4: crunch 1 8 -f charset.lst mixalpha-numeric-all-space -o wordlist.txt - crunch will use the mixalpha-numeric-all-space character set from charset.lst and will write the wordlist to a file named wordlist.txt. The file will start with a and end with " "

Example 5: crunch 8 8 -f charset.lst mixalpha-numeric-all-space -o wordlist.txt -t @@dog@@@ -s cbdogaaa - crunch should generate a 8 character wordlist using the mixalpha-number-all-space character set from charset.lst and will write the wordlist to a file named wordlist.txt. The file will start at cbdogaaa and end at " dog "

Example 6: crunch 2 3 -f charset.lst ualpha -s BB - crunch with start generating a wordlist at BB and end with ZZZ. This is useful if you have to stop generating a wordlist in the middle. Just do a tail wordlist.txt and set the -s parameter to the next word in the sequence. Be sure to rename the original wordlist BEFORE you begin as crunch will overwrite the existing wordlist.

Example 7: crunch 4 5 -p abc - The numbers aren't processed but are needed. crunch will generate abc, acb, bac, bca, cab, cba.

Example 8: crunch 4 5 -p dog cat bird - The numbers aren't processed but are needed. crunch will generate birdcatdog, birddogcat, catbirddog, catdogbird, dogbirdcat, dogcatbird.

Example 9: crunch 1 5 -o START -c 6000 -z bzip2 - crunch will generate bzip2 compressed files with each file containing 6000 words. The filenames of the compressed files will be first_word-last_word.txt.bz2

Example 10: crunch 4 5 -b 20mib -o START - will generate 4 files: aaaa-gvfed.txt, gvfee-ombqy.txt, ombqz-wcydt.txt, wcydu-zzzzz.txt the first three files are 20MBs (real power of 2 MegaBytes) and the last file is 11MB.

Example 11: crunch 3 3 abc + 123 !@# -t @%^ - will generate a 3 character long word with a character as the first character, and number as the second character, and a symbol for the third character. The order in which you specify the characters you want is important. You must specify the order as lower case character, upper case character, number, and symbol. If you aren't going to use a particular character set you use a plus sign as a placeholder. As you can see I am not using the upper case character set so I am using the plus sign placeholder. The above will start at a1! and end at c3#

Example 12: crunch 3 3 abc + 123 !@# -t ^%@ - will generate 3 character words starting with !1a and ending with #3c

Example 13: crunch 4 4 + + 123 + -t %%@^ - the plus sign (+) is a place holder so you can specify a character set for the character type. crunch will use the default character set for the character type when crunch encounters a + (plus sign) on the command line. You must either specify values for each character type or use the plus sign. I.E. if you have two characters types you MUST either specify values for each type or use a plus sign. So in this example the character sets will be:

```
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ123
!@#$%^&*()-_+=~`[]{}|\:;"'<>,.?/
```

there is a space at the end of the above string the output will start at 11a! and end at "33z ". The quotes show the space at the end of the string.

Example 14: crunch 5 5 -t ddd@@ -o j -p dog cat bird - any character other than one of the following: @,%^ is the placeholder for the words to permute. The @,%^ symbols have the same function as -t. If you want to use @,%^ in your output you can use the -l option to specify which character you want crunch to treat as a literal. So the results are

```
birdcatdogaa
birdcatdogab
birdcatdogac
<skipped>
dogcatbirdzy
dogcatbirdzz
```

Example 15: crunch 7 7 -t p@ss,%^ -l a@aaaaa - crunch will now treat the @ symbol as a literal character and not replace the character with a uppercase letter. this will generate

```
p@ssA0!
p@ssA0@
p@ssA0#
p@ssA0$
<skipped>
p@ssZ9
```

Example 16: crunch 5 5 -s @4#S2 -t @%^,2 -e @8 Q2 -l @dddd -b 10KB -o START - crunch will generate 5 character strings starting with @4#S2 and ending at @8 Q2. The output will be broken into 10KB sized files named for the files starting and ending strings.

Example 17: crunch 5 5 -d 2@ -t @@@%% - crunch will generate 5 character strings staring with aab00 and ending at zzy99. Notice that aaa and zzz are not present.

Example 18: crunch 10 10 -t @@@^%%%%^^ -d 2@ -d 3% -b 20mb -o START - crunch will generate 10 character strings starting with aab!0001!! and ending at zzy9998. The output will be written to 20mb files.

Example 19: crunch 8 8 -d 2@ - crunch will generate 8 characters that limit the same number of lower case characters to 2. Crunch will start at aabaabaa and end at zzyzzyzz.

Example 20: crunch 4 4 -f unicode_test.lst japanese -t @@%% -l @xdd - crunch will load some Japanese characters from the unicode_test character set file. The output will start at @日00 and end at @語99.

Ejercicio L3B3_BRUTE: "Fuerza bruta pura" con *John the Ripper*

Descripción de la actividad / Aplicación práctica

Necesitas crackear las claves de los usuarios **usando fuerza bruta pura**, porque no tienes ninguna pista acerca de cómo podrían ser las claves de esos usuarios

Resultados Esperados

Esta actividad finaliza cuando se consigue **crackear la hash de la contraseña** del usuario `Android`.

Otra información necesaria para su realización

Si no tenemos ni idea de la forma de la contraseña, y ninguna contraseña común nos ha funcionado (por ejemplo del diccionario del ejercicio anterior), podemos probar a usar fuerza bruta pura con `john` contra el archivo anterior, con todos los valores por defecto, y ver qué pasa

Quizás tengamos suerte... o tal vez debamos esperar días / semanas / años / ...

Ten en cuenta que `john` aplica permutaciones predeterminadas a las palabras que genera y tiene múltiples modelos de crack. Para probar esto, sigue estas reglas:

- Elimina el archivo `john.pot` que está en el directorio `.john` oculto de la carpeta en la que estás trabajando. Si no lo haces, y `john` ya ha descifrado una contraseña en una ejecución anterior, `john` te informará de que la contraseña ya ha sido descifrada y no calculará nada, ¡aunque queden algunas contraseñas sin descifrar! 😟 .
- Ejecuta `john` en modo **incremental,** especificando un parámetro adicional `--max-length=<numero>` para limitar la cantidad de permutaciones de caracteres que `john` puede usar y, por tanto, hacer que el proceso finalice en un tiempo razonable. Usa el *cheatsheet* anterior para especificar las opciones adecuadas para descifrar solo la contraseña del usuario `Android`, ya que sabemos que este usuario **usa una contraseña muy corta** (la longitud de la contraseña es igual o inferior a 4 caracteres).

Finalmente, te damos esta documentación como referencia si tienes curiosidad, pero realmente no es necesaria para terminar esta actividad.

- Modos de craqueo de John: https://www.openwall.com/john/doc/MODES.shtml
- Preguntas frecuentes de John: https://www.openwall.com/john/doc/FAQ.shtml
- Ejemplos de John: https://www.openwall.com/john/doc/EXAMPLES.shtml
- Reglas de la lista de palabras de John: https://www.openwall.com/john/doc/RULES.shtml
- Más información: https://manualdehacker.com/john-the-ripper-cracking-passwords/

✍️ Insignias y Autoevaluación

NOTA: Tienes una versión de esta tabla de insignias en formato editable disponible en el *Campus Virtual*. Puedes usar este archivo para crear un documento en el formato que desees y tomar notas extendidas de tus actividades para crear un log de lo que has hecho. Recuerda que el material que elabores se puede llevar a los exámenes de laboratorio.

Nivel de Insignia	Desbloqueado cuando	¿Desbloqueado?
	Puedas codificar cualquier información en formato Base64 o similar gracias a *CyberChef*	
	Seas capaz de cifrar cualquier archivo utilizando un algoritmo de cifrado simétrico fuerte. Responde a la siguiente pregunta: *¿El contenido del archivo cifrado es binario o texto sin formato?*	
	Seas capaz de convertir un fichero codificado en *Base64* a uno binario equivalente	
	Puedas responder a las siguientes preguntas: *¿Puedes enviar un archivo codificado en Base64 por correo electrónico? ¿O a través de WhatsApp, etc. (excluidos los límites de longitud)?*	
	Seas capaz de descifrar un archivo cifrado con un algoritmo simétrico	
	Puedas utilizar `crunch` para generar diccionarios de palabras simples de longitud limitada	
	Puedas responder a la siguiente pregunta: *¿Entiendes ahora por qué la ingeniería social es tan popular a la hora de obtener contraseñas de alguien teniendo en cuenta lo que cuesta un proceso de crackeo típico?*)	
	Comprendas cómo cifrar carpetas y su contenido en el sistema de archivos. Responde a estas preguntas: *¿Qué tipo de ataque dificulta? ¿El cifrado es transparente (se puede usar sin introducir claves al trabajar con ficheros cifrados)?*	
	Entiendas el proceso de ocultar información en un archivo de imagen. Puedes responder a esta pregunta: *¿el archivo de texto introducido en la imagen usando esteganografía hace que sea visualmente diferente de la original?*	
	Sabes cómo utilizar los “embellecedores” de *JavaScript* y sus posibles limitaciones.	
	Puedas usar *John the Ripper* para romper la hash de una contraseña de *Linux* por fuerza bruta	
	Puedas usar *John the Ripper* para romper la hash de una contraseña de un fichero cifrado con openssl con un ataque de diccionario	
	Puedas usar *John the Ripper* para romper una contraseña de *Linux* con un ataque de diccionario	

	Puedas responder a la siguiente pregunta a la vista de lo que has aprendido en este laboratorio: si una de mis claves aparece en "*Have I been pwnd?*" *¿significa eso que está ya comprometida y cualquier atacante conoce mi contraseña directamente?*	
	Puedas responder a las siguientes preguntas: *¿Fue rápido el proceso de crackeo por fuerza bruta que hiciste? ¿Qué crees que podría pasar si la contraseña no estuviera entre las primeras en el diccionario? ¿Entiendes lo que se necesita para mejorar la velocidad de estos procesos?*	
	Conoces las ventajas/desventajas de usar un diccionario grande basado en fugas de información frente a uno personalizado.	
	Puedes ofuscar y desofuscar código *JavaScript*, y probar que funciona correctamente incluso en su modo ofuscado. También puedes responder a esta pregunta: *¿cuál crees que es la principal utilidad de la ofuscación de código con respecto a la seguridad?*	
	Entiendes claramente por qué CADA servicio insiste en que uses contraseñas lo suficientemente largas y complejas para sus cuentas	
	Entiendes que, aunque la información de salida esté cifrada, no todos los modos de cifrado son válidos para determinados tipos de información	
	Comprendes el uso de *CyberChef*, sus modos de operación y cadenas de recetas.	
	Comprendes las ventajas y desventajas de los ataques de fuerza bruta. Puedes responder a esta pregunta: *¿qué método es más propenso a fallar si la contraseña es común, o una palabra típica? ¿Y si no?*	
	Puedes responder a esta pregunta: *¿Puedes imaginar un procedimiento para enviar a alguien un mensaje secreto oculto a simple vista y protegido con contraseña utilizando cosas "normales" que utilizas diariamente (correo electrónico, redes sociales ...)?*	
	**Aquí se holdea**: Tu conocimiento del cifrado aplicado te permite utilizar el cifrado fuerte de clave simétrica para enviar mensajes privados a cualquier persona, aplicar técnicas para ocultar tus datos y tu código, y romper contraseñas de orígenes comunes.	