



Source: Stable Diffusion AI

LABORATORIO 3. APLICACIONES DE LA CRİPTOGRAFÍA (I)

DEPARTAMENTO DE INFORMÁTICA. UNIVERSIDAD DE OVIEDO

Seguridad de Sistemas Informáticos | 2022 – 2023 (v3.1 “S-81 Isaac Peral”)



CONTENIDO

Infraestructura de este laboratorio.....	3
Bloque 1: Criptografía Simétrica	5
Cifrar un archivo utilizando un algoritmo de clave simétrica fuerte.....	6
Cifrar archivos en formato "ASCII-armored"	7
Descifrar un archivo que utiliza cifrado simétrico	7
Intercambiar una clave mediante el algoritmo <i>Diffie-Hellman</i>	7
Cifrado de discos	9
Bloque 2. Codificación, ofuscación y esteganografía	10
<i>Cyberchef</i> : codificación de datos	11
Uso de esteganografía con <i>steghide</i>	11
Código ofuscado para que sea difícil de entender	12
Bloque 3. Introducción al cracking de contraseñas sin conexión	14
<i>John the ripper</i> para romper PGP	15
<i>John the ripper</i> + contraseñas de usuario + <i>crunch</i>	16
"Fuerza bruta pura" con <i>John the Ripper</i>	19
Insignias y Autoevaluación	20



AVISO

Este documento forma parte de la asignatura “Seguridad de Sistemas Informáticos”, impartida en la *Escuela de Ingeniería Informática* de la *Universidad de Oviedo*. Es fruto del trabajo continuado de elaboración, soporte, mejora, actualización y revisión del siguiente equipo de profesores desde el año 2019

- Enrique Juan de Andrés Galiana
- Fernando Cano Espinosa
- Miguel Riesco Albizu
- José Manuel Redondo López
- Luís Vinuesa Martínez

Te pedimos por favor que **NO lo compartas públicamente en Internet**. No obstante, entendemos que puedas considerar este material interesante para otras personas. Por ese motivo, hemos creado una versión de este adaptada para que pueda cursarse de forma online, disponible gratuitamente para todo el mundo y que puedes encontrar en esta dirección: <https://ocw.uniovi.es/course/view.php?id=109>

A diferencia de esta versión, **la versión libre puedes promocionarla todo lo que quieras**, que para eso está 😊

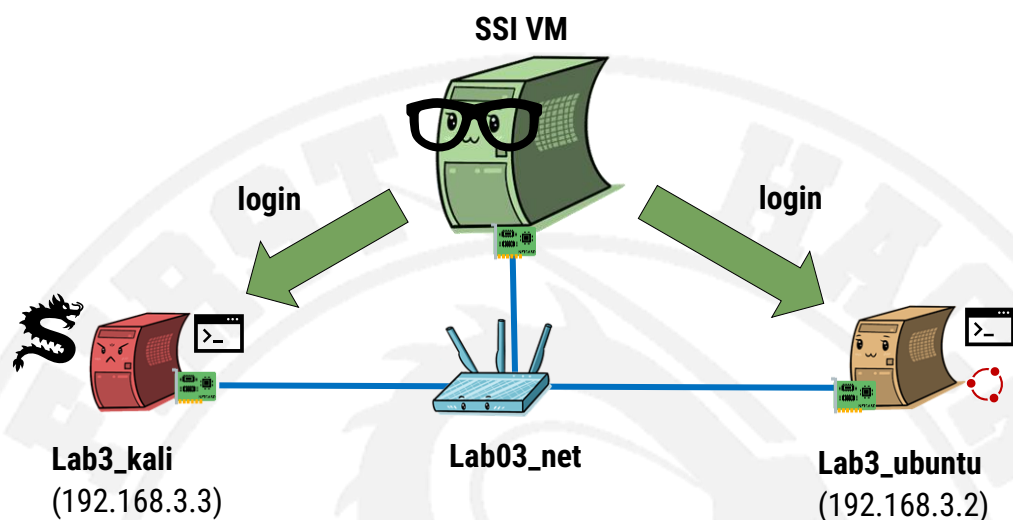
GRACIAS POR TU COLABORACIÓN



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 **gpg2john**. No todas las actividades requieren el uso de todos ellos. Consulta el siguiente diagrama para entender el diseño de la infraestructura:





BLOQUE 1:

CRIPTOGRAFÍA


SIMÉTRICA



Cifrar un archivo utilizando un algoritmo de clave simétrica fuerte

Aplicación práctica: Necesitas cifrar un fichero para que nadie sin su clave pueda leer sus contenidos

- 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)
- A continuación, comprueba los algoritmos de cifrado/hash admitidos por GPG en su distribución de Linux con `gpg --version`.
- Elige un algoritmo de cifrado fuerte y un tamaño de clave adecuado, de acuerdo con los conceptos dados en teoría. Usa el siguiente *cheatsheet* para elegir las opciones correctas para cifrar simétricamente el archivo con el algoritmo de cifrado elegido.

gpg (GnuPG) 2.2.20 Cheatsheet (ingenieriainformatica.uniovi.es) Multipurpose GPL cipher/hash/pubkey software https://gnupg.org/		 <pre>redondo@server1804:~\$ gpg -o original.txt -d cryptandsign.enc gpg: encrypted with 3072-bit RSA key, ID CA493341D9E0E95, created 2019-11-18 "redondo <redondo@mail.es>" gpg: Signature made lun 18 nov 2019 10:12:34 UTC gpg: using RSA key 230C013753283601EDE49B6B4811A20BF1861B2A gpg: Good signature from "operario <operario@mail.es>" [full] operario@server1804:~\$ gpg --list-keys gpg: checking the trustdb gpg: marginals needed: 3 completes needed: 1 trust model: pgp gpg: depth: 0 valid: 1 signed: 0 trust: 0-, 0q, 0n, 0m, 0f, 1u gpg: next trustdb check due at 2021-11-17 /home/operario/.gnupg/pubring.kbx pub rsa3072 2019-11-18 [SC] [expires: 2021-11-17] 230C013753283601EDE49B6B4811A20BF1861B2A uid [ultimate] operario <operario@mail.es> sub rsa3072 2019-11-18 [E] [expires: 2021-11-17]</pre>	
GENERAL USAGE gpg [options] [files]			
NOTES Sign, check, encrypt or decrypt Default operation depends on the input data Also supports the following compression algorithms: No compression, ZIP, ZLIB, BZIP2			
OPTIONS			
Symmetric encryption Supported cipher algorithms: IDEA, 3DES, CAST5, BLOWFISH, AES, AES192, AES256, TWOFISH, CAMELLIA128, CAMELLIA192, CAMELLIA256. Use --cipher-algo option to choose one -c, --symmetric: encryption only with symmetric cipher		Public/private key handling (II) --full-generate-key: full featured key pair generation --generate-key: generate a new key pair --generate-revocation: generate a revocation certificate --import: import/merge keys --list-signatures: list keys and signatures --lsign-key: sign a key locally --print-md: print message digests --quick-add-uid: quickly add a new user-id --quick-generate-key: quickly generate a new key pair --quick-ls-sign-key: quickly sign a key locally --quick-revoke-uid: quickly revoke a user-id --quick-set-expire: quickly set a new expiration date --quick-sign-key: quickly sign a key --receive-keys: import keys from a keyserver --refresh-keys: update all keys from a keyserver --search-keys: search for keys on a keyserver --send-keys: export keys to a keyserver --server: run in server mode --sign-key: sign a key --tofu-policy VALUE: set the TOFU policy for a key --update-trustdb: update the trust database -k, --list-keys: list keys -K, --list-secret-keys: list secret keys	
Signatures Supported hash algorithms: SHA1, RIPEMD160, SHA256, SHA384, SHA512, SHA224 --clear-sign: make a clear text signature --verify: verify a signature -b, --detach-sign: make a detached signature -s, --sign: make a signature			
Asymmetric encryption Supported public key algorithms: RSA, ELG, DSA, ECDH, ECDSA, EDDSA -d, --decrypt: decrypt data (default) -e, --encrypt: encrypt data			
Public/private key handling (I) --card-status: print the card status --change-passphrase: change a passphrase --change-pin: change a card's PIN --check-signatures: list and check key signatures --delete-keys: remove keys from the public keyring --delete-secret-keys: remove keys from the secret keyring --edit-card: change data on a card --edit-key: sign or edit a key --export: export keys --fingerprint: list keys and fingerprints			
EXAMPLES Sign and encrypt for user Bob: gpg -se -r Bob [file] Make a clear text signature: gpg --clear-sign [file] Make a detached signature: gpg --detach-sign [file] Show keys: gpg --list-keys [names] Show fingerprints: gpg --fingerprint [names] Cipher a file using a strong symmetric key algorithm: gpg --symmetric --cipher-algo AES256 -c message.txt Cipher files with a user-friendly Data Format: gpg --armor --symmetric --cipher-algo AES256 -c message.txt Decipher a file that used symmetric encryption: gpg --decrypt --output=dmessage.txt message.txt.gpg Clear signature for a file: gpg --clearsign file.txt Asymmetric encryption for a particular user (redondo): gpg -o file_for_redondo.gpg -e -r redondo file.txt Decryption of asymmetrically encrypted text: gpg -o file.txt -d file.txt.gpg		Miscellaneous options --openpgp: use strict OpenPGP behavior --textmode: use canonical text mode -a, --armor: create ascii armored output -i, --interactive: prompt before overwriting -n, --dry-run: do not make any changes -o, --output FILE: write output to FILE -r, --recipient USER-ID: encrypt for USER-ID -u, --local-user USER-ID: use USER-ID to sign or decrypt -v, --verbose: verbose -z N: set compress level to N (0 disables)	

- **Cuando se pida una clave de cifrado, dar como clave "password"** (es una mala práctica, pero aquí lo hacemos a propósito para luego 😊). Debería aparecer el archivo `message.txt.gpg` cuando finalices todo el proceso.

Resultados esperados: Esta actividad finaliza cuando puedas cifrar un archivo mediante un algoritmo de clave simétrica fuerte.

Cifrar archivos en formato "ASCII-armored"

Aplicación práctica: Necesitas cifrar un archivo para que nadie sin su clave pueda saber sus contenidos, pero en formato de texto para que puedan enviar el contenido cifrado fácilmente.

Repite el proceso anterior pero añadiendo la opción `--armor`. Mira el resultado y piensa qué opciones te da este formato respecto al anterior. Por ejemplo: ¿Puedes mandar lo que se genera por correo? ¿Podrías escribirlo en alguna red social?

Resultados esperados: Esta actividad finaliza cuando puedas cifrar un archivo utilizando un algoritmo de clave simétrica fuerte en formato "armored".

Descifrar un archivo que utiliza cifrado simétrico

Aplicación práctica: Necesitas saber los contenidos de un fichero cifrado previamente si sabes la clave correspondiente

Usando el *cheatsheet* anterior, busca la opción adecuada para descifrar ambos archivos generados previamente (el *armored* y el no *armored*) en archivos de salida distintos. Comprueba que el resultado de ambas operaciones es el mismo. **NOTA:** Si acabas de cifrar un archivo en tu VM y descifras el archivo inmediatamente después en la misma VM, a lo mejor no se te pide una clave para descifrarlo. Esto es porque la clave se cachea unos minutos, ni porque hayas hecho algo mal.

Envía ambos archivos cifrados usando `scp` al contenedor de **Ubuntu de la infraestructura** y descifra los en él (`scp <fichero> <usuario>@<IPDeLaMáquinaUbuntu>:.`). Si la transferencia falla, asegúrate de que el servicio `ssh` esté activo en el contenedor destino (`service --status-all` debe mostrar un `[+]` junto al servicio `ssh`) y haz `service ssh start` si no. **NOTA:** En el caso de la versión *armored*, puedes enviarle por correo a un/a compañero/a el contenido cifrado (y la clave) y que lo descifre.

Resultados esperados: Esta actividad finaliza cuando se consigue descifrar un archivo previamente cifrado. ¿Entiendes por qué es importante separar la forma de enviar la clave de la de enviar el contenido cifrado?

Intercambiar una clave mediante el algoritmo Diffie-Hellman

Aplicación práctica: Sabes cómo funciona a nivel práctico el algoritmo de intercambio de claves Diffie-Hellman usando mundialmente para muchas aplicaciones hoy en día



(NOTA: Este ejercicio supone que se trabaja con un compañero. Sin embargo, es posible hacer esto de forma individual. Simplemente envía el archivo desde el contenedor Kali al Ubuntu en la infraestructura y sigue las instrucciones).

El algoritmo de intercambio de claves *Diffie-Hellman* necesita un par de números para intercambiar claves. Usaremos los siguientes números, que se supone que son conocidos por ambas partes que establecen la comunicación.

- **p:** El número primo 997
- **a:** La raíz p primitiva, 7

Ahora **cada estudiante/usuario debe pensar en un número $X < p$ que sólo él/ella conozca** (es decir, su **clave secreta/privada de cifrado/descifrado**) para generar una **clave pública Y** distinta para cada uno. Calculamos Y usando la siguiente fórmula $Y = \text{pow}(a, X) \bmod p$. Podemos usar la *Calculadora de Windows* para esto. Una vez hecho esto, almacenamos el resultado en un archivo. Por ejemplo, si tenemos Usuario₁ y Usuario₂:

- Usuario₁ elige el número privado **X = 35**
- Usuario₂ elige el número privado **X = 55**

Entonces podemos calcular cada valor Y (clave pública)

- $Y_{\text{usuario1}} = 7^{35} \bmod 997 = \mathbf{389}$ (**pkPubUser1**)
- $Y_{\text{usuario2}} = 7^{55} \bmod 997 = \mathbf{195}$ (**pkPubUser2**)

Una vez calculado Y, podemos seguir este procedimiento:

- Escribe el valor Y (clave pública) en un archivo (para cada usuario)

```
echo -e "La dato público es <coloca aquí la Y calculada>" | tee -a pkPubU0xxxxx.txt
```

- Envía la clave pública al otro usuario. Una opción es enviarlo vía **Secure Copy (scp)** a la otra máquina de la infraestructura, utilizando el mismo procedimiento que vimos antes, o por email (como es información pública, en un caso real da igual que se conozca si se usan n°s lo suficientemente grandes).
- El otro usuario hará la misma operación con nosotros, por lo que ambos pueden calcular la clave privada utilizando la clave pública que se ha enviado. Para calcular la clave privada correspondiente, utilizamos esta fórmula (de nuevo con la *Calculadora de Windows*) $\text{pkPrv} = \text{pow}(\text{pkPubDeLOtro}, X) \bmod p$. Ejemplo:

- Para Usuario₁: $\text{pkPrvUser1} = \text{pow}(\text{pkPubUser2}, X) \bmod p \rightarrow (195^{35}) \bmod 997 \rightarrow 146$
- Para Usuario₂: $\text{pkPrvUser2} = \text{pow}(\text{pkPubUser1}, X) \bmod p \rightarrow (389^{55}) \bmod 997 \rightarrow 146$

(NOTA: Obviamente en una situación real **se deben elegir números primos mucho más grandes para generar claves verdaderamente seguras**. No todos los primos son igualmente válidos, ya que hay números primos que pueden debilitar el algoritmo. Hoy en día se recomienda un mínimo de tamaño de clave de 2048 bits para los intercambios de claves *Diffie-Hellman*, pero el procedimiento de intercambio es exactamente el que acabamos de explicar. *Diffie-Hellman* lo usa principalmente otro software, como *OpenSSH*, en lugar de usarlo directamente un usuario final como en este ejercicio. Si tienes curiosidad, puede encontrar más información aquí:



- **Un servicio público para generar parámetros Diffie-Hellman verdaderamente fuertes** (más de 2048 bits de tamaño, descartando números primos inadecuados): <https://2ton.com.au/dhtool/>
- **Cómo utilizar estos parámetros para fortalecer las conexiones OpenSSH:** <https://www.linode.com/docs/security/advanced-ssh-server-security/>

Resultados esperados: Esta actividad finaliza cuando se puedan calcular correctamente las claves *pkPrv* para ambos usuarios y las claves obtenidas sean iguales para ambos usuarios.

Cifrado de discos

Aplicación práctica: Necesitas crear un directorio cuyos contenidos sea cifran automáticamente, para que nadie que no sea tu usuario pueda acceder a ellos

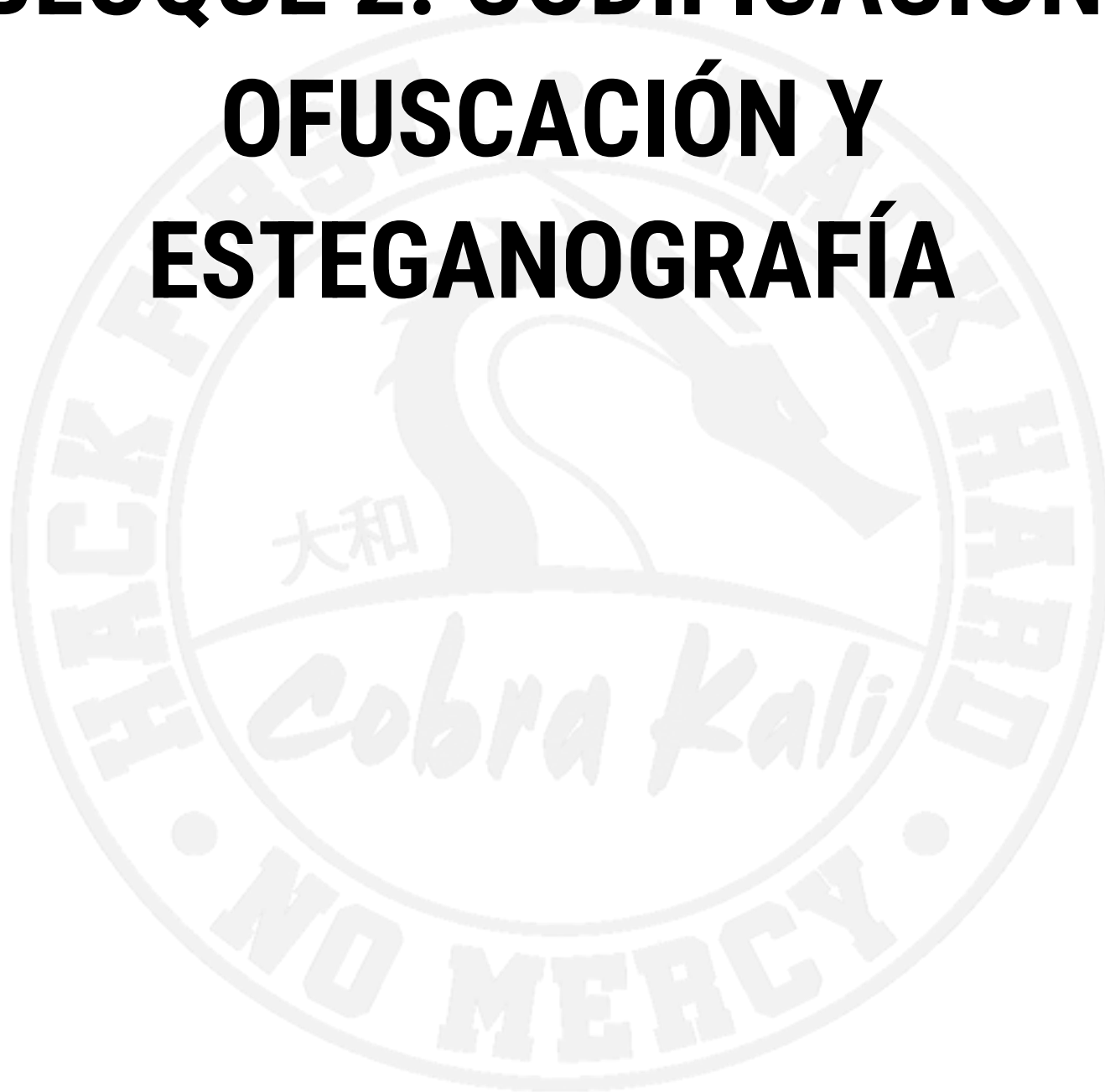
El cifrado de sistemas de archivos, directorios y discos es una de las aplicaciones más comunes de los algoritmos de clave simétrica. Cuando *Ubuntu* se instala ofrece cifrar nuestro directorio de inicio como opción (esto también cifra la partición `/swap`). Incluso si elegimos no hacerlo en ese momento,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

Sin embargo, 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`
- Elige un algoritmo de cifrado simétrico apropiado de acuerdo con la teoría.
- Elige `32` bytes para la clave.
- Elige `No` en la opción *plaintext passthrough*.
- Elige `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?

Resultados esperados: Esta actividad finaliza cuando puedas cifrar una carpeta y comprobar que el contenido está realmente cifrado cuando la desmontas.

BLOQUE 2. CODIFICACIÓN, OFUSCACIÓN Y ESTEGANOGRAFÍA



Cyberchef: codificación de datos

Aplicación práctica: Necesitas saber cómo transformar datos y un nº grande de posibles formas de hacerlo usando una herramienta de referencia en el mercado

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 online 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**. Para practicar con algunos ejemplos, puedes hacer lo siguiente.

- Cifrado de sustitución ROT13 sobre un texto que elijas.
- Cifrados de sustitución ROT13+ROT47 sobre un texto que elijas (verás que puedes hacer una cadena de cifrado, es decir, cifrar datos con varios algoritmos en secuencia)
- Descifrar la salida del punto anterior realizando la transformación necesaria a la cadena.
- Blowfish encriptado con una clave e IV lo suficientemente largas.
- Descifrado Blowfish de la operación anterior (¡asegúrate de guardar los parámetros de cifrado que usaste antes! 😊)
- Codificación Base64 (¡la más común hoy en día!)
- ...

Resultados esperados: Esta actividad finaliza cuando te encuentres lo suficientemente familiarizado con el uso de Cyberchef.

Uso de esteganografía con steghide

Aplicación práctica: Puedes ocultar secretos en imágenes sin alterar lo que la imagen muestra

Para hacer este ejercicio podemos elegir cualquier imagen de Internet e incrustarle el mensaje que ciframos anteriormente con **steghide**, utilizando como contraseña de **steghide** la que se ha intercambiado en el paso anterior usando **Diffie-Hellman (pkPriv)**. 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 cifrado 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. **NOTA:** Sirve también que envíes la imagen por correo a un compañero/a a ver

si puede sacar el mensaje secreto de ella (pásale la clave). ¿Funcionaría si posteas la imagen en una red social o la red social altera la imagen de alguna forma?

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	
-f, --force: overwrite existing files	-K, --nochecksum: do not embed crc32 checksum of embedded data
-p <passphrase>: use <passphrase> to extract data	-N, --dontembedname: do not embed the name of the original file
-p, --passphrase: specify passphrase	-p <passphrase>: use <passphrase> to embed data
-q, --quiet: suppress information messages	-p, --passphrase: specify passphrase
-sf <filename>: extract data from <filename>	-q, --quiet: suppress information messages
-sf, --stegofile: select stego file	-sf <filename>: write result to <filename> instead of cover-file
-v, --verbose: display detailed information	-sf, --stegofile: select stego file
-xf <filename>: write the extracted data to <filename>	-v, --verbose: display detailed information
-xf, --extractfile: select file name for extracted data	-z <l>: using level <l> (1 best speed..9 best compression)
	-Z, --compress: compress data before embedding (default)
	-Z, --dontcompress: do not compress data before embedding
EXAMPLES	Options for the Info Command
To embed emb.txt in cvr.jpg: steghide embed -cf cvr.jpg -ef emb.txt	-p, --passphrase: specify passphrase
To extract embedded data from stg.jpg: steghide extract -sf stg.jpg	-p <passphrase>: use <passphrase> to get info about embedded data

Resultados esperados: Esta actividad finaliza cuando se logra ocultar un mensaje mediante esteganografía en una imagen y extraerlo posteriormente.

Código ofuscado para que sea difícil de entender

Aplicación práctica: Necesitas ofuscar tu código JavaScript para hacer más difícil que alguien averigüe como has implementado cierta funcionalidad

La ofuscación del código está relacionada con el cifrado y la esteganografía porque, aunque **NO es cifrado**, 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 “empaquéalo” (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?


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



BLOQUE 3. INTRODUCCIÓN AL CRACKING DE CONTRASEÑAS SIN CONEXIÓN

John the ripper para romper PGP

Aplicación práctica: Necesitas crackear un mensaje cifrado con PGP que usa una password común

John the Ripper 1.9.0-jumbo Cheatsheet (ingenieriainformatica.uniovi.es) 		Other options
Offline password cracking tool		
https://github.com/openwall/john		--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)
GENERAL USAGE		--encoding=NAME: input encoding (eg. UTF-8, ISO-8859-1). See also doc/ENCODINGS and --list-hidden-options.
john [OPTIONS] [PASSWORD-FILES]		--fork=N: fork N processes
		--format=NAME: force hash of type NAME. the supported formats can be seen with --list=formats and --list=subformats
NOTES		--groups=[-]GID[,...]: load users [not] of this (these) group(s) only
John cracking modes: https://www.openwall.com/john/doc/MODES.shtml		--list=WHAT: list capabilities, see --list=help or doc/OPTIONS
John FAQ: https://www.openwall.com/john/doc/FAQ.shtml		--loopback[=FILE]: like --wordlist, but extract words from a .pot file
John wordlist rules: https://www.openwall.com/john/doc/RULES.shtml		--make-charset=FILE: make a charset file. It will be overwritten
		--mask[=MASK]: mask mode using MASK (or default from john.conf)
OPTIONS		--node=MIN[-MAX]/TOTAL: this node's number range out of TOTAL count
Cracking modes		--pipe: like --stdin, but bulk reads, and allows rules
--external=MODE: external MODE or word filter		--pot=NAME: pot FILE to use
--incremental[=MODE]: "incremental" mode [using section MODE]		--prince[=FILE]: PRINCE mode, read words from FILE
--markov[=OPTIONS]: "Markov" mode (see doc/MARKOV)		--restore[=NAME]: restore an interrupted session [called NAME]
--single[=SECTION[,...]]: "single crack" mode, using default or named rules		--rules=[SECTION[,...]]: enable word mangling rules (for wordlist or PRINCE modes), using default or named rules
--single=:rule[,...]: same, using "immediate" rule(s)		--rules=:rule[,...]: same, using "immediate" rule(s)
--wordlist[=FILE]: --stdin wordlist mode, read words from FILE or stdin		--rules-stack=:rule[,...]: same, using "immediate" rule(s)
EXAMPLES		--rules-stack=SECTION[,...]: stacked rules, applied after regular rules or to modes that otherwise don't support rules
Please consult a full example set on: https://www.openwall.com/john/doc/EXAMPLES.shtml		--salts=[-]COUNT[:MAX]: load salts with[out] COUNT [to MAX] hashes
ssiuser@ubuntussi:~\$ john -wordlist:myDict passwd.db		--save-memory=LEVEL: enable memory saving, at LEVEL 1..3
Created directory: /home/ssiuser/.john		--session=NAME: give a new session the NAME
Loaded 1 password hash (crypt, generic crypt(3) [?/64])		--shells=[-]SHELL[,...]: load users with[out] this (these) shell(s) only
Press 'q' or Ctrl-C to abort, almost any other key for status		--show[=left]: show cracked passwords [if =left, then uncracked]
test123... (ssiuser)		--status[=NAME]: print status of a session [called NAME]
1g 0:00:00:00 100% 2.173g/s 417.3p/s 417.3c/s 417.3C/s test096.....test191...		--stdout[=LENGTH]: just output candidate passwords [cut at LENGTH]
Use the "--show" option to display all of the cracked passwords reliably		--subsets[=CHARSET]: "subsets" mode (see doc/SUBSETS)
Session completed		--test[=TIME]: run tests and benchmarks for TIME seconds each
ssiuser@ubuntussi:~\$		--users=[-]LOGIN UID[,...]: [do not] load this (these) user(s) only

Las contraseñas se almacenan en formato *hash*, como vimos en el tema de teoría de criptografía. Por lo tanto, para romperlas solo tenemos tres estrategias posibles:

- **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 😊.
- **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 CTF, pero podría ser difícil obtener un 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.

Vamos a utilizar un ataque basado en diccionario para romper la contraseña de nuestro mensaje PGP codificado anteriormente, siguiendo estos pasos:

- Utilizar una herramienta **de descifrado de contraseñas** popular como *John the Ripper* (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/>
- **El mensaje para descifrar**. Los archivos cifrados GPG no son directamente procesables con `john`, por lo que debes procesarlos previamente con la utilidad `gpg2john` 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.

Usa el *cheatsheet* de `john` con la lista de palabras proporcionada contra el archivo anterior para descifrar la contraseña del mensaje GPG.

Resultados esperados: Esta actividad finaliza cuando se descifra la contraseña del mensaje PGP anterior.

John the ripper + contraseñas de usuario + crunch

Aplicación práctica: Necesitas crackear la clave de un usuario con un diccionario de palabras personalizado

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 y el archivo de `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).

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](#))

**crunch 3.6 Cheatsheet** (ingenieriainformatica.uniovi.es)

Tool that generates wordlists from a character set

<https://tools.kali.org/password-attacks/crunch>**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 20mb -o START will generate 4 files: aaaa-gvfd.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 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, ,',%, 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.

```
ssuser@ubuntu:~$ 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
```

EXAMPLES

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

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-numeric-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 will 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 20mb -o START - will generate 4 files: aaaa-gvfd.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 a!l and end at c#3

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

there is a space at the end of the above string the output will start at !1a! 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
dogcatbirddz
```

Example 15: crunch 7 7 -t p@ss,%^ -l aaaaaa - 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@ss29
```

Example 16: crunch 5 5 -s @4#S2 -t @%^,2 -e @8 Q2 -l @ddd -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 starting 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 zzy 9998. 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 zzyzzzz.

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.

"Fuerza bruta pura" con John the Ripper

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

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/>

Resultados esperados: Esta actividad finaliza cuando se consigue descifrar la contraseña del usuario **Android**.












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?
	Ser capaz de cifrar cualquier archivo utilizando un algoritmo de cifrado simétrico fuerte. Responde a las siguientes preguntas: <i>¿El contenido del archivo cifrado es binario o texto sin formato?</i>	
	Ser capaz de cifrar cualquier archivo utilizando un algoritmo de cifrado simétrico fuerte en formato <i>armored</i> . Responde a las siguientes preguntas: <i>¿El contenido del archivo cifrado es binario o texto sin formato? ¿Genera un archivo de salida diferente?</i>	
	Responde las siguientes preguntas: <i>¿Puedes enviar un archivo armored por correo electrónico? ¿O a través de WhatsApp, etc. (excluidos los límites de longitud)? ¿Cómo deberías transmitir la clave al receptor?</i>	
	Ser capaz de descifrar un archivo cifrado con un algoritmo simétrico, sin importar el formato utilizado (<i>armored</i> o no) en un archivo de salida que elijas. Responde a la siguiente pregunta: <i>¿Cambia el proceso de descifrado en función del formato de archivo cifrado?</i>	
	Sabes cómo utilizar scp . Puedes responder a esta pregunta: <i>¿qué servicio utiliza scp para copiar archivos de forma segura?</i>	
	Utilizar crunch para generar diccionarios de palabras simples de longitud limitada	
	Responde a la siguiente pregunta: <i>¿Cómo crees que puedes adivinar la "forma" de una contraseña de usuario (es decir, hacer una suposición de parte de su contenido)? (PISTA: ¿Cómo puedes saber cosas sobre personas que no conoces personalmente?)</i>	
	Comprender cómo cifrar carpetas y su contenido en el sistema de archivos. Responde a esta pregunta: <i>¿Qué tipo de ataque dificulta? ¿El cifrado es transparente (se puede usar sin introducir claves al trabajar con ficheros cifrados)?</i>	
	Entender el proceso de ocultar información en un archivo de imagen. Puedes responder a esta pregunta: <i>¿el archivo de texto introducido en la imagen usando esteganografía hace que sea visualmente diferente de la original?</i>	
	Sabes cómo utilizar los “embellecedores” de JavaScript y sus posibles limitaciones.	
	Usas <i>John the Ripper</i> para romper una contraseña de Linux por fuerza bruta	
	Usas <i>John the Ripper</i> para romper una contraseña PGP con un ataque de diccionario	
	Usas <i>John the Ripper</i> para romper una contraseña de Linux con un ataque de diccionario	
	Responde a la siguiente pregunta: si una de mis claves aparece en “ <i>Have I been pwned?</i> ” ¿significa eso que está ya comprometida y cualquier atacante conoce mi contraseña directamente?	



	Responde a las siguientes preguntas: <i>¿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?</i>	
	Conoces las ventajas/desventajas de usar un diccionario grande basado en fugas de información frente a uno personalizado.	
	Puedes ofuscar y desofuscar código <i>JavaScript</i> , y probar que funciona correctamente incluso en su modo ofuscado. También puedes responder a esta pregunta: <i>¿cuál crees que es la principal utilidad de la ofuscación de código con respecto a la seguridad?</i>	
	Entiendes claramente por qué CADA servicio insiste en que uses contraseñas lo suficientemente largas y complejas para sus cuentas	
	Entiendes cómo funciona el algoritmo de intercambio de claves <i>Diffie-Hellman</i> .	
	Comprendes el uso de <i>Cyberchef</i> , sus modos de operación y cadenas de recetas.	
	Comprendes las ventajas y desventajas de los ataques de fuerza bruta. Puedes responder a esta pregunta: <i>¿qué método es más propenso a fallar si la contraseña es común, o una palabra típica? ¿Y si no?</i>	
	Puedes responder a esta pregunta: <i>¿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 ...)?</i>	
	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.	